

S4D437

Transactional Apps with the ABAP RESTful Programming Model

PARTICIPANT HANDBOOK INSTRUCTOR-LED TRAINING

Course Version: 22
Course Duration: 3 Day(s)
Material Number: 50158033

SAP Copyrights, Trademarks and Disclaimers

© 2022 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. Please see <https://www.sap.com/corporate/en/legal/copyright.html> for additional trademark information and notices.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors.

National product specifications may vary.

These materials may have been machine translated and may contain grammatical errors or inaccuracies.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP SE or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP SE or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platform directions and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, which speak only as of their dates, and they should not be relied upon in making purchasing decisions.

Typographic Conventions

American English is the standard used in this handbook.

The following typographic conventions are also used.

This information is displayed in the instructor's presentation



Demonstration



Procedure



Warning or Caution



Hint



Related or Additional Information



Facilitated Discussion



User interface control

Example text

Window title

Example text

Contents

vii Course Overview

1 Unit 1: The ABAP RESTful Programming Model (RAP)

2	Lesson: Understanding the Concept and Architecture of RAP
17	Exercise 1: Define a CDS-based Data Model
23	Lesson: Defining an OData UI Service
31	Exercise 2: Define and Preview an OData UI Service

37 Unit 2: RAP Business Objects (RAP BOs)

39	Lesson: Defining RAP Business Objects and their Behavior
49	Exercise 3: Define an RAP Business Object and its Behavior
56	Lesson: Using Entity Manipulation Language (EML) to Access RAP Business Objects
65	Exercise 4: Read and Update an RAP Business Object
73	Exercise 5: Optional: Use Long Version of EML Statements and Entity Aliases
82	Lesson: Understanding Concurrency Control in RAP
87	Exercise 6: Establish Optimistic Concurrency Control
93	Lesson: Defining Actions and Messages
107	Exercise 7: Define and Implement an Action
123	Lesson: Implementing Authority Checks
133	Exercise 8: Implement Authority Checks

143 Unit 3: Update and Create in Managed Transactional Apps

144	Lesson: Enabling Input Fields and Value Help
153	Exercise 9: Provide Input Fields and Value Help
159	Lesson: Implementing Input Checks with Validations
165	Exercise 10: Provide Input Checks Through Validations
177	Lesson: Providing Values with Determinations
187	Exercise 11: Enable Managed Numbering and Implement Determinations
198	Lesson: Implementing Dynamic Feature Control
205	Exercise 12: Implement Dynamic Action and Field Control

215 Unit 4: Draft-Enabled Transactional Apps

216	Lesson: Understanding the Draft Concept
231	Exercise 13: Enable Draft Handling for a RAP Business Object
240	Lesson: Developing Draft-Enabled Applications
251	Exercise 14: Enable Draft Handling in SAP Fiori Elements App and Adjust Implementations

267 Unit 5: Transactional Apps with Composite Business Object

- 268 Lesson: Defining Composite RAP Business Objects
- 279 Exercise 15: Define a Composite RAP Business Object
- 290 Lesson: Defining Compositions in OData UI Services
- 297 Exercise 16: Define a Composite OData UI Service with RAP
- 307 Lesson: Implementing the Behavior for Composite RAP BOs
- 311 Exercise 17: Implement the Behavior of a Composite RAP Business Object

329 Unit 6: Transactional Apps with Unmanaged Business Object

- 330 Lesson: Understanding Data Access in Unmanaged Implementations
- 335 Exercise 18: Define an Unmanaged Business Object
- 343 Lesson: Implementing Unmanaged Business Objects
- 347 Exercise 19: Implement an Unmanaged Business Object

Course Overview

TARGET AUDIENCE

This course is intended for the following audiences:

- Development Consultant
- Developer

UNIT 1

The ABAP RESTful Programming Model (RAP)

Lesson 1

Understanding the Concept and Architecture of RAP	2
Exercise 1: Define a CDS-based Data Model	17

Lesson 2

Defining an OData UI Service	23
Exercise 2: Define and Preview an OData UI Service	31

UNIT OBJECTIVES

- Understand the concept of RAP
- Use ABAP development tools
- Explain the RAP architecture and business use case
- Define a CDS projection view
- Enrich a projection view with UI metadata
- Create and preview an OData UI service

Understanding the Concept and Architecture of RAP

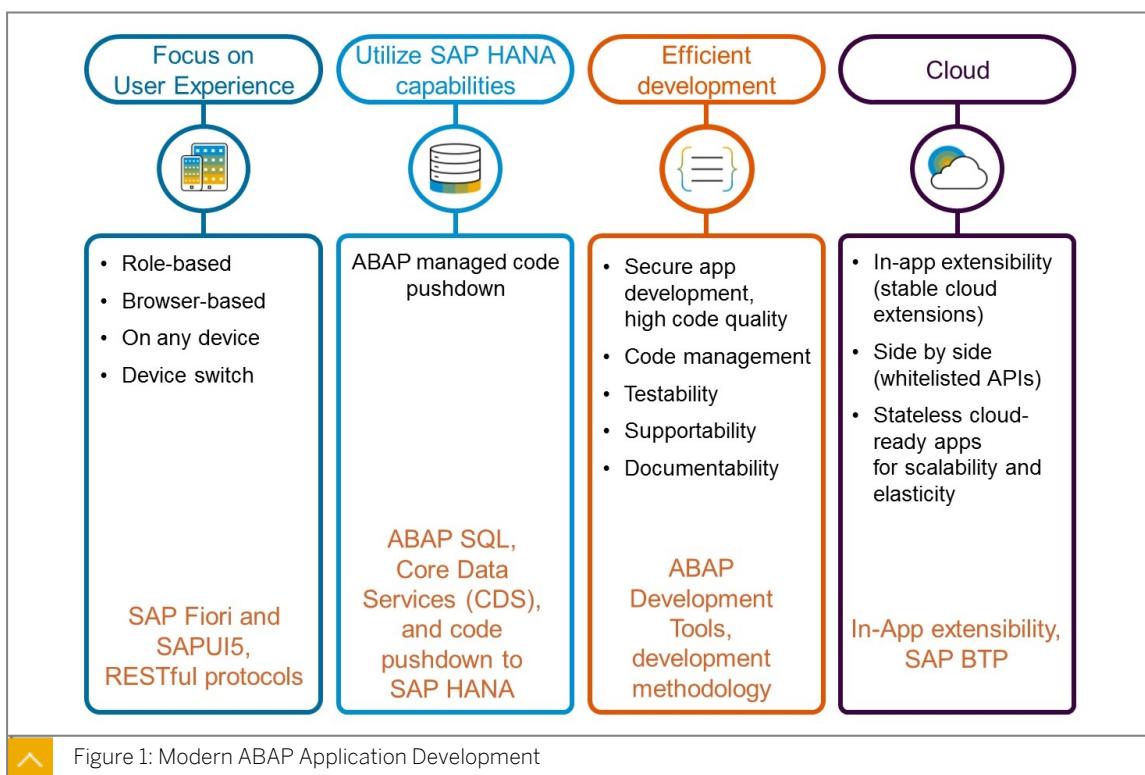


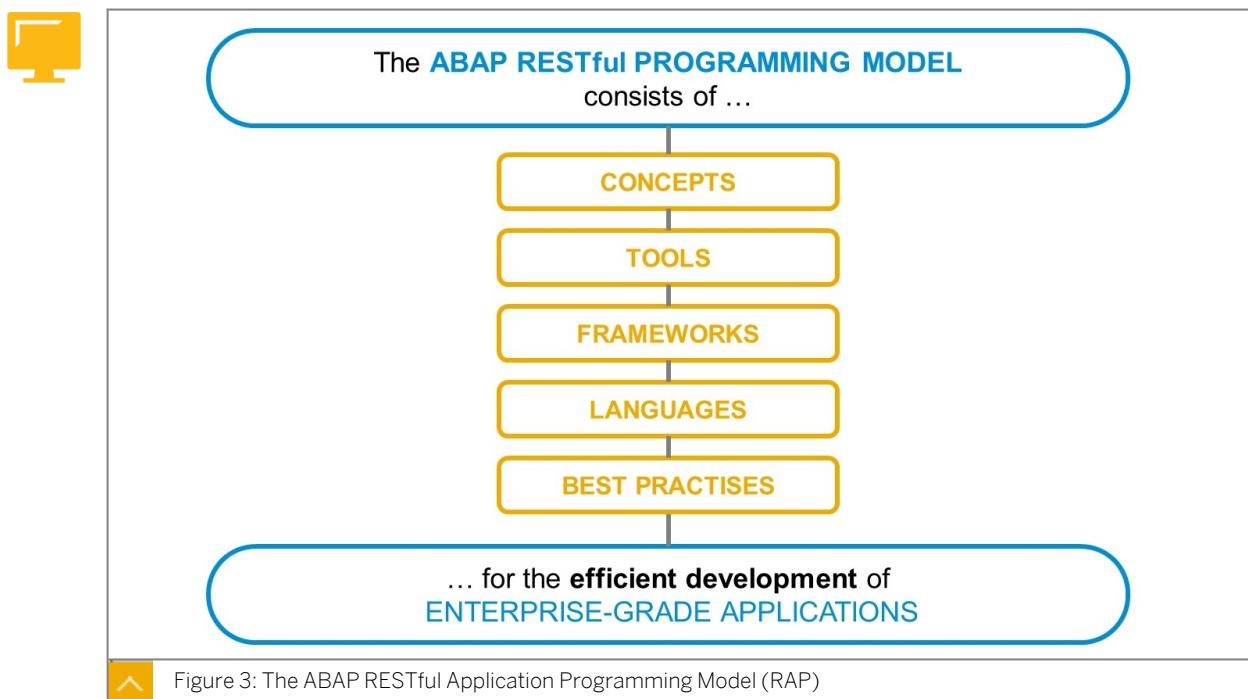
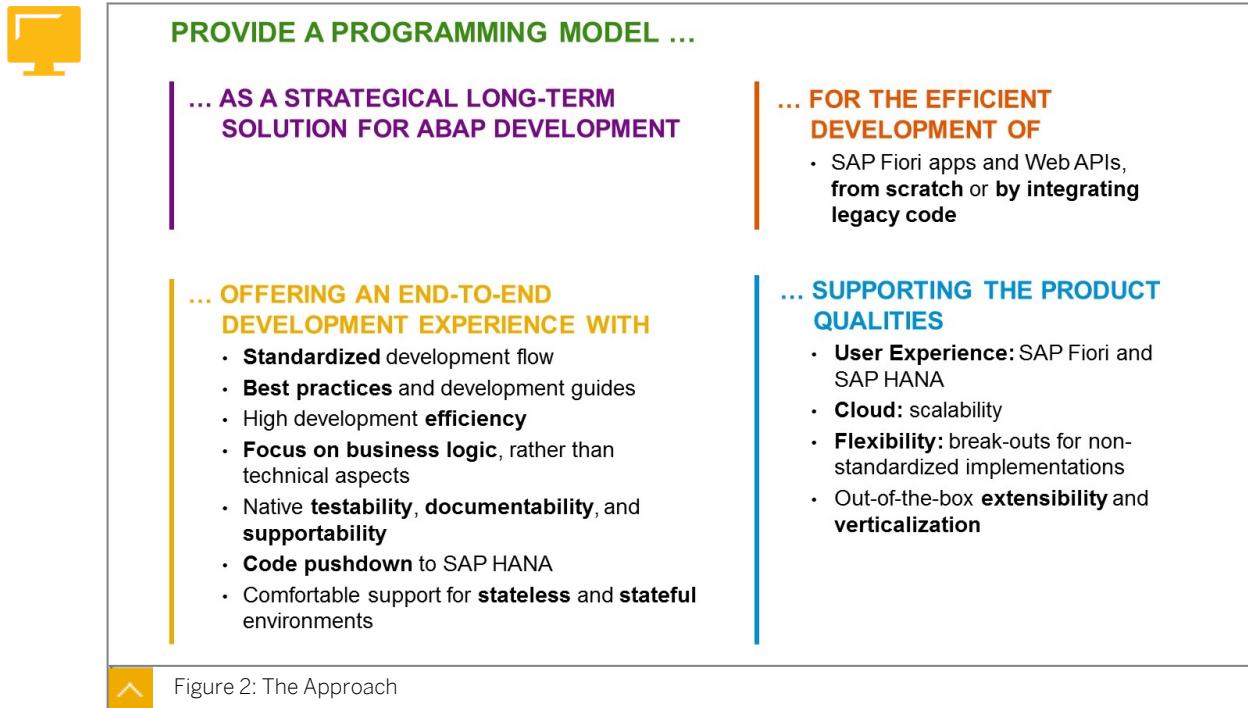
LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand the concept of RAP
- Use ABAP development tools
- Explain the RAP architecture and business use case

Overview of the ABAP RESTful Programming Model (RAP)





The ABAP RESTful Application Programming Model, also known as the ABAP RESTful Programming Model, ABAP RAP, or RAP for short, is a programming model for ABAP that is RESTful. This means that it meets the requirements of a REST architecture. In ABAP RAP, AS ABAP plays the role of a stateless Web server.

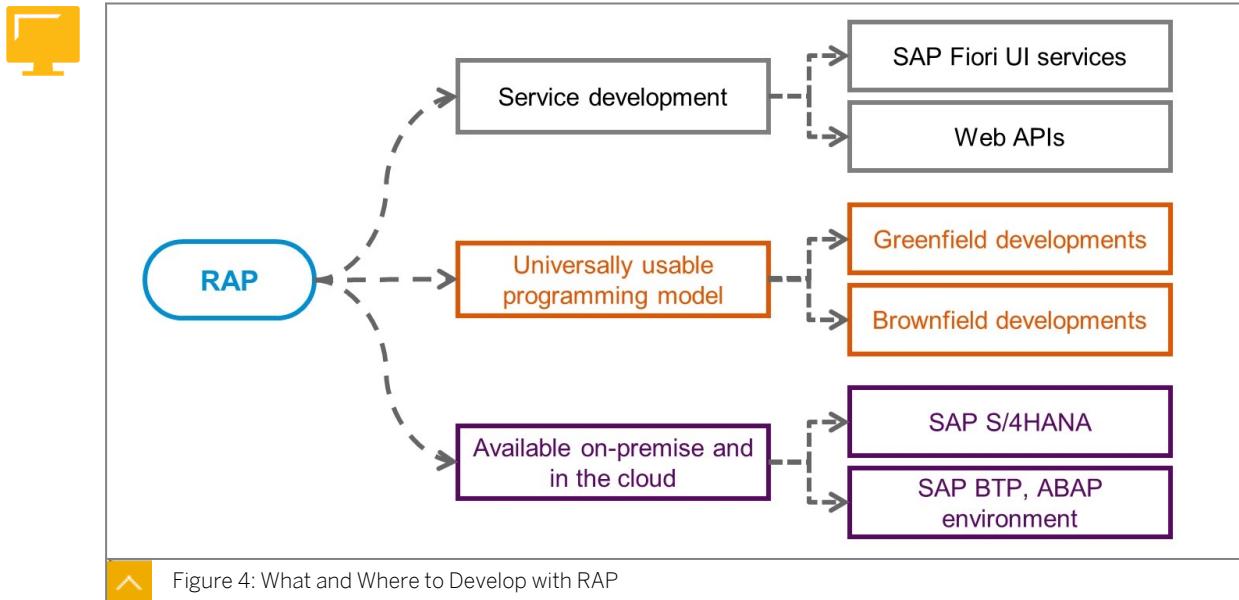


Figure 4: What and Where to Develop with RAP

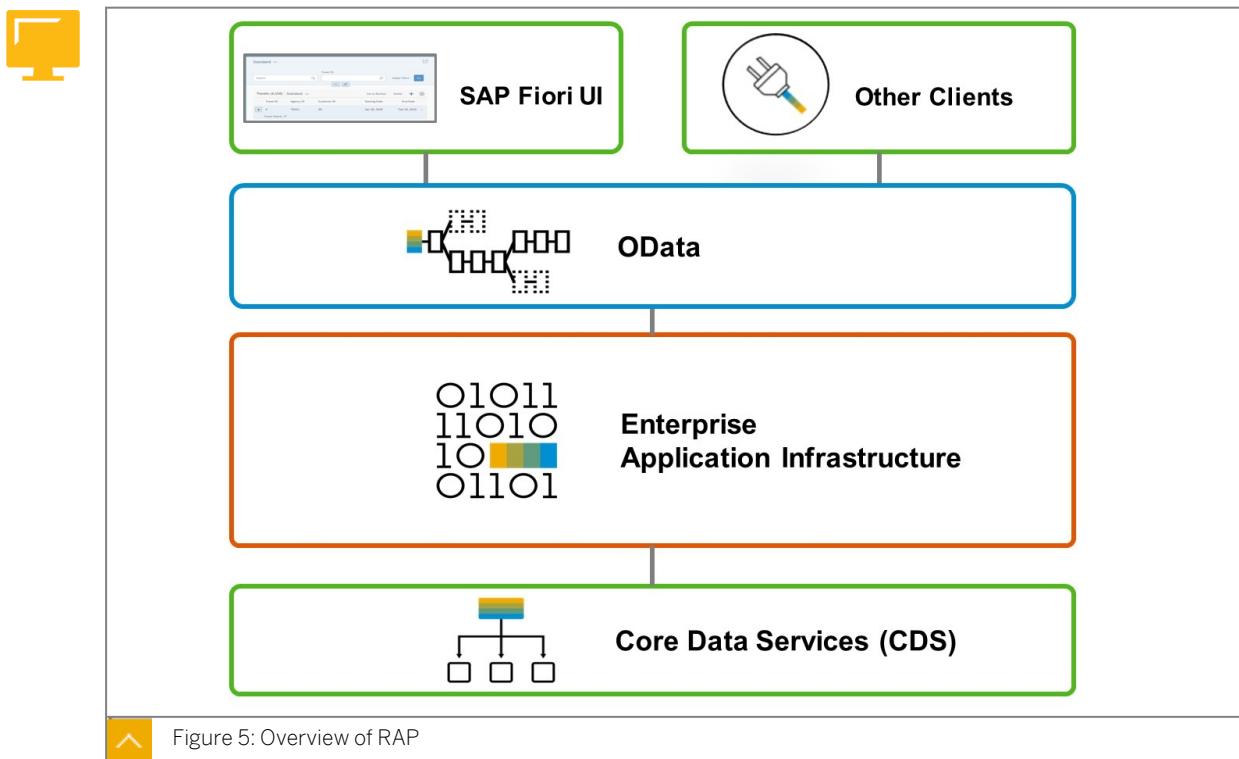


Figure 5: Overview of RAP

RAP consists of the following building blocks:

ABAP Core Data Services

CDS is our ubiquitous modeling language to declare domain data models.

Enterprise Application Infrastructure

The enterprise Application Infrastructure offers:

- Powerful service runtime frameworks
- First-class support for SAP Fiori and SAP HANA

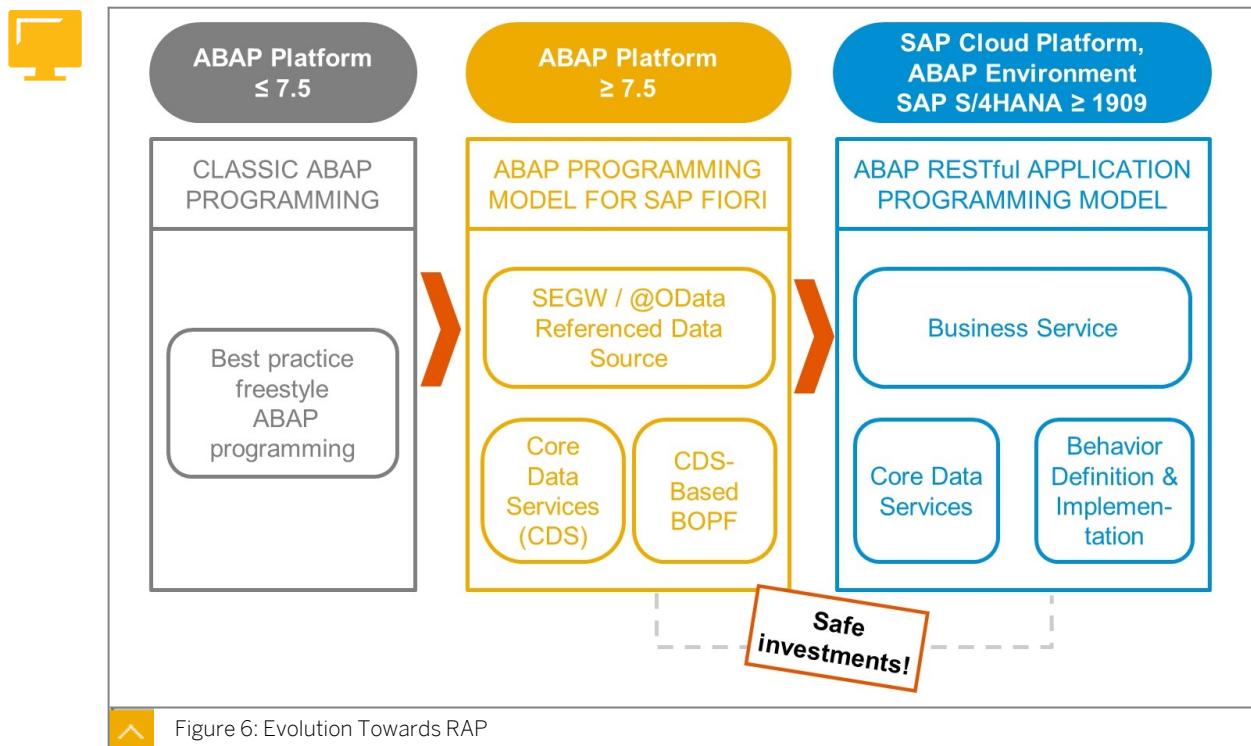
- Out-of-the-box implementations
- Draft support for SAP Fiori UIs
- Built-in extensibility capabilities

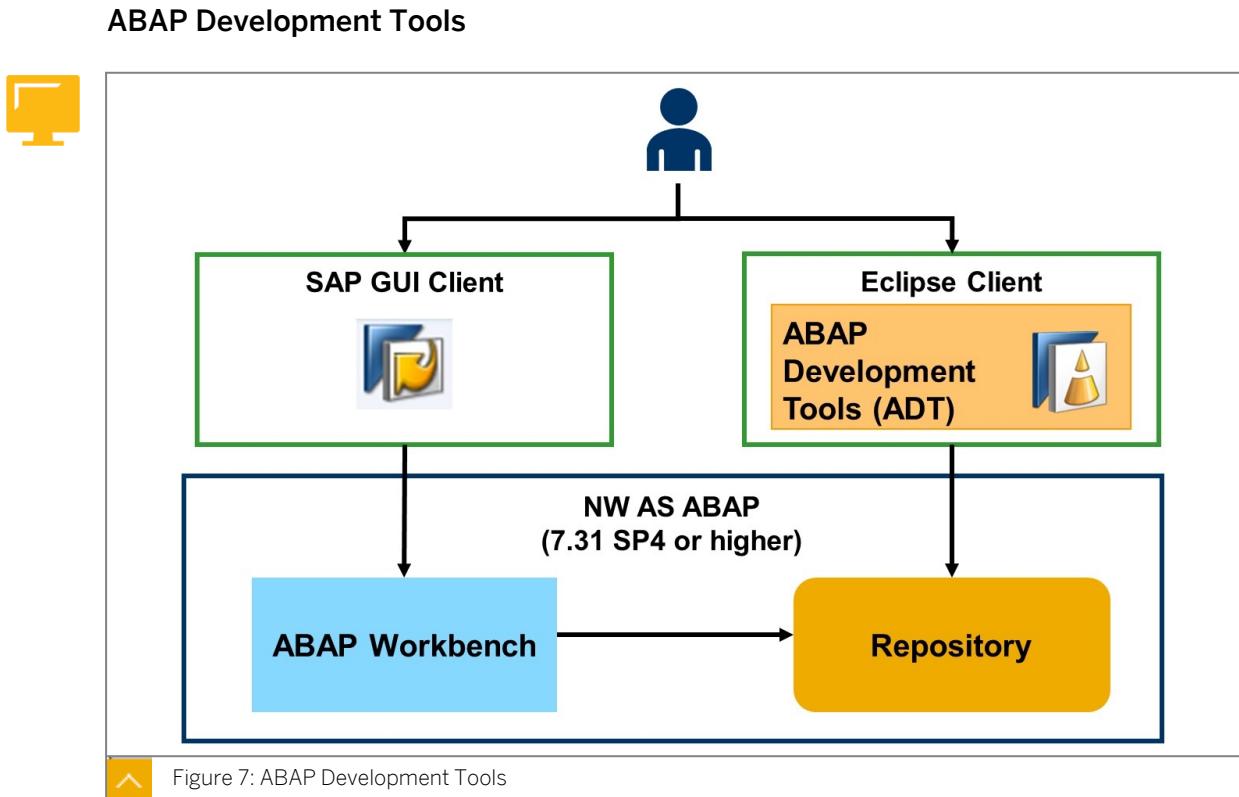
OData

OData is a standardized protocol for defining and consuming.

Service Consumption

RAP supports UI development, either based on SAP Fiori elements or as freestyle SAPUI5 development. It also supports service consumption via Web APIs.





ABAP Development Tools (ADT) is an alternative to the ABAP Workbench. ADT provides the following features:

- A completely new ABAP development experience on top of the Eclipse platform
- An open platform for developing new ABAP-related tools
- A set of open, language-independent, and platform-independent APIs that developers can use to build new custom tools for the ABAP environment

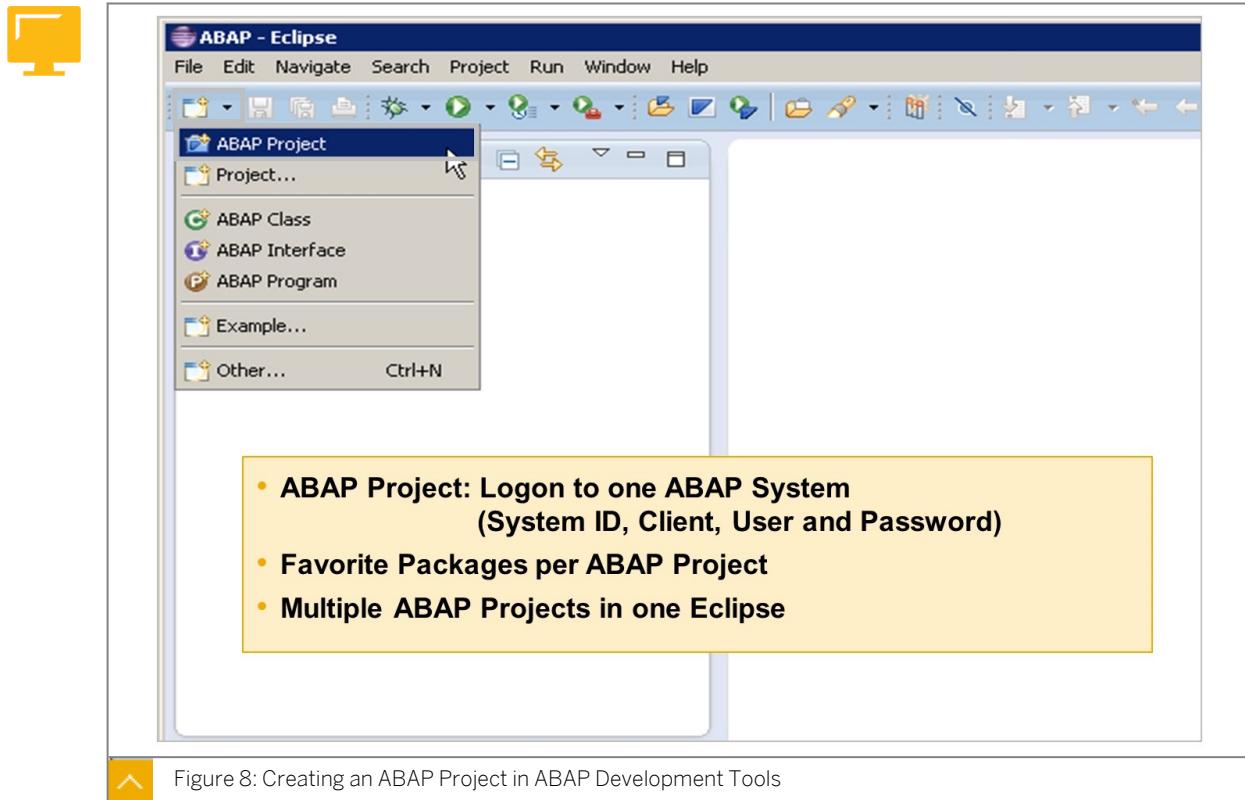


Figure 8: Creating an ABAP Project in ABAP Development Tools

An ABAP project serves as a container for the development objects that are stored in a particular ABAP back-end system and contains the logon parameters for the system logon: system, client, user, and language. You must be logged on to the system to edit an object. Within the ABAP project, you can access any development object in the repository. In addition, to make it easier to manage objects, you can set up favorite packages for each project.

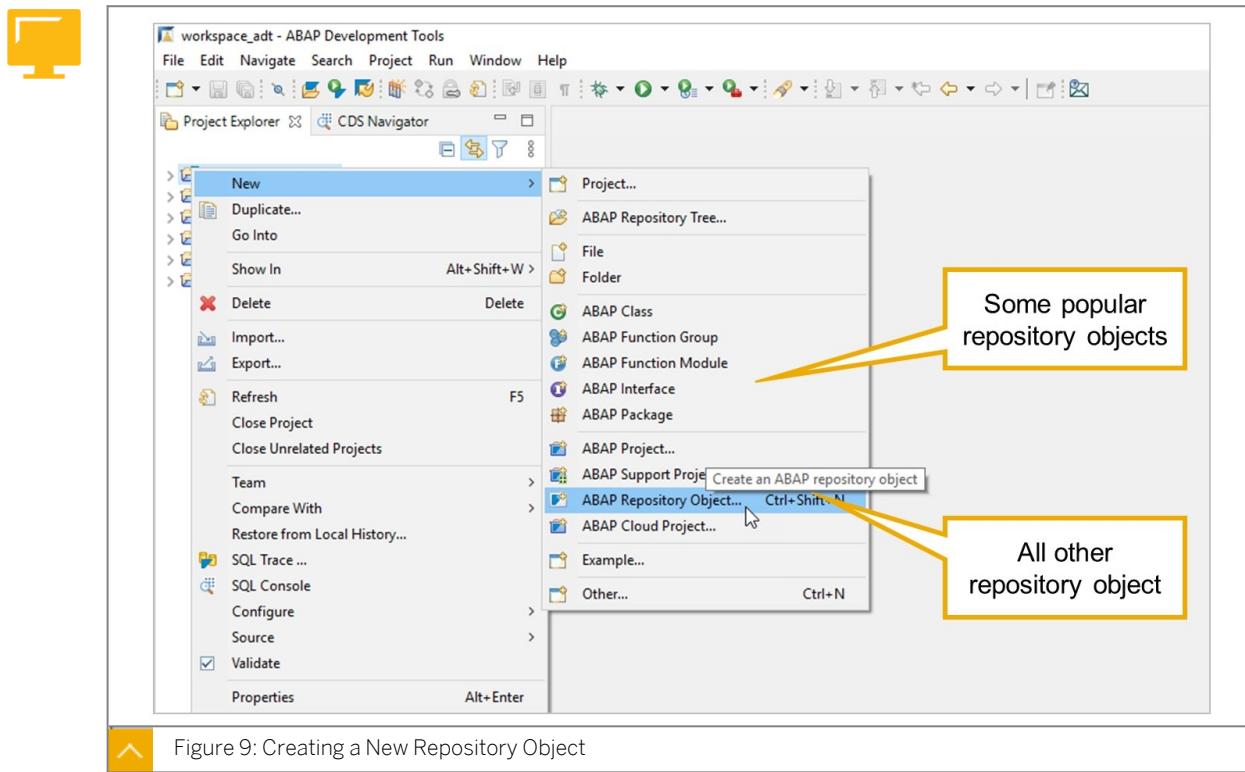


Figure 9: Creating a New Repository Object

To create a new repository object in ADT, right-click the project in the project explorer and choose *New* → *ABAP Repository Object* In the following dialog box, you can search for the type of repository object you want to create.



Note:

For some popular repository objects, there are direct menu entries under the *New* menu.

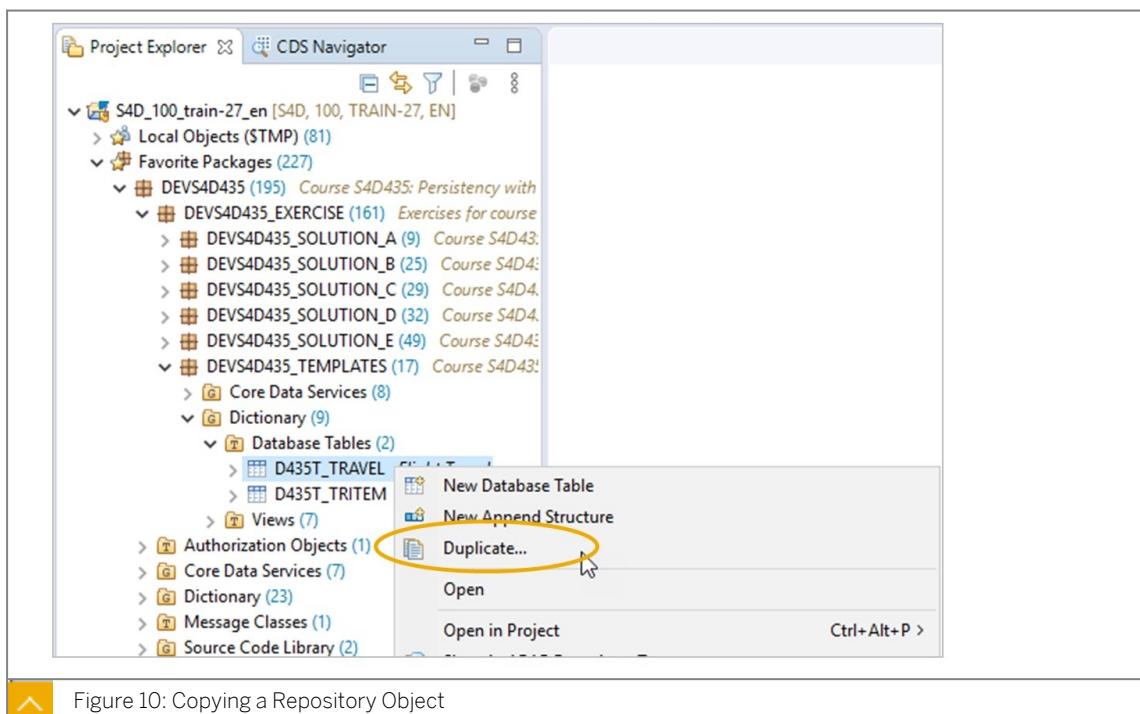


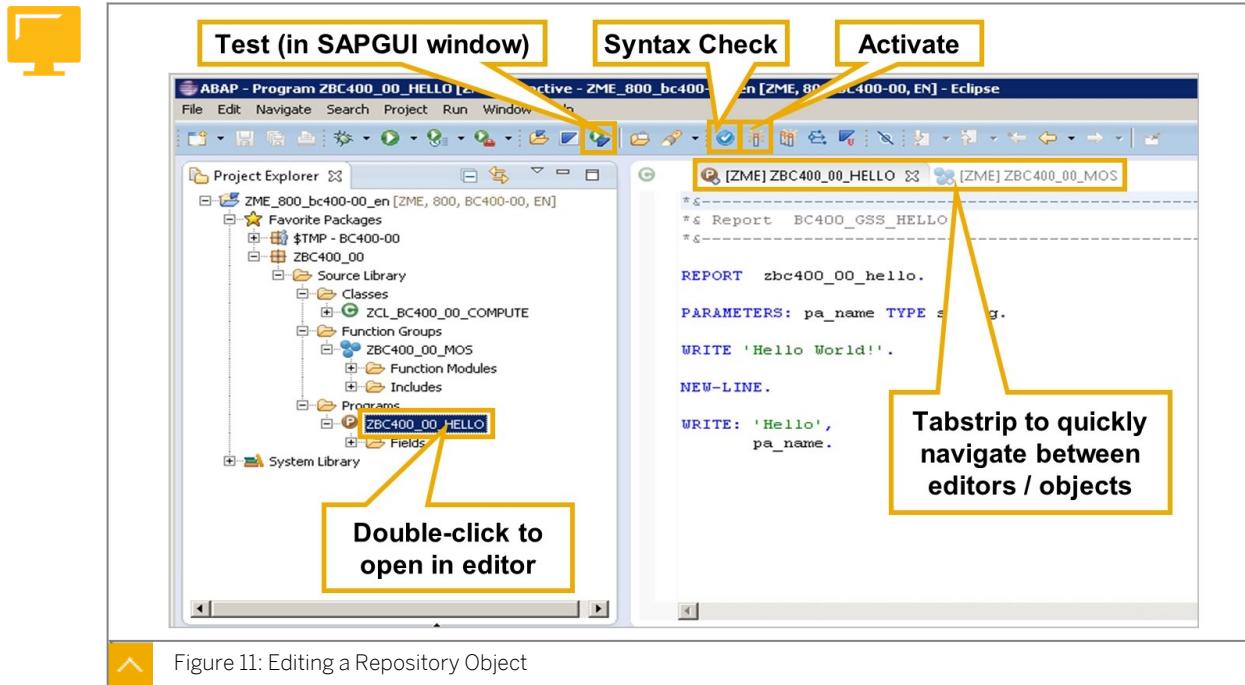
Figure 10: Copying a Repository Object

To create a copy of an existing repository object, right-click the source object in the Project Explorer and choose *Duplicate...* .



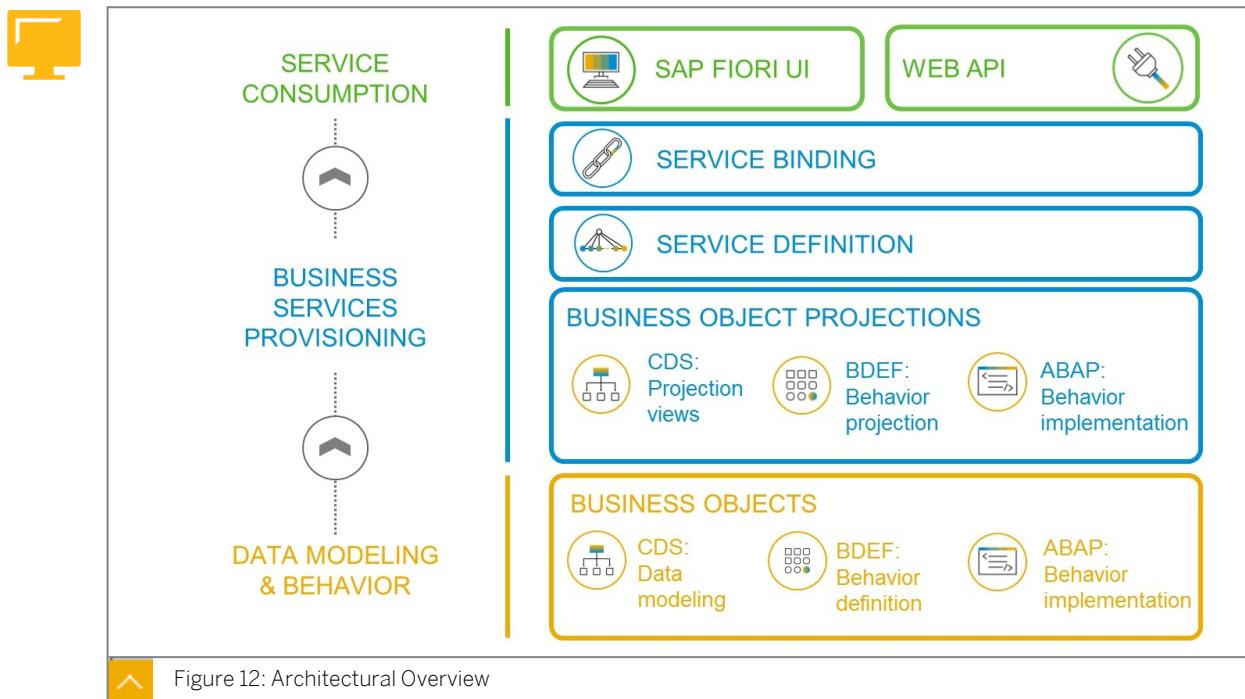
Note:

You must specify the new package at the beginning of the dialog. In the classical ABAP workbench, you specify the package at the end.



To open a specific repository object in its respective editor, double-click it. The editor is shown on the right side of the ABAP perspective.

RAP Architecture



Applications that are developed with the ABAP RESTful Application Programming Model consist of the following building-blocks:

Business Objects

Business Objects represent the data model and define the data related logic, called behavior, independent of specific consumption. RAP business objects are defined

through CDS data modeling views, CDS behavior definitions, and Behavior implementations in ABAP classes.

Business Object Projections

The Business Object Projection is an approach to project and 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. In RAP, a BO Projection consists of CDS projection views, CDS Behavior projections, and, if needed, additional or consumption specific implementations.

Service Definition

A service definition defines the scope of a business service, in particular, the business object projection to be exposed via this service.

Service Binding

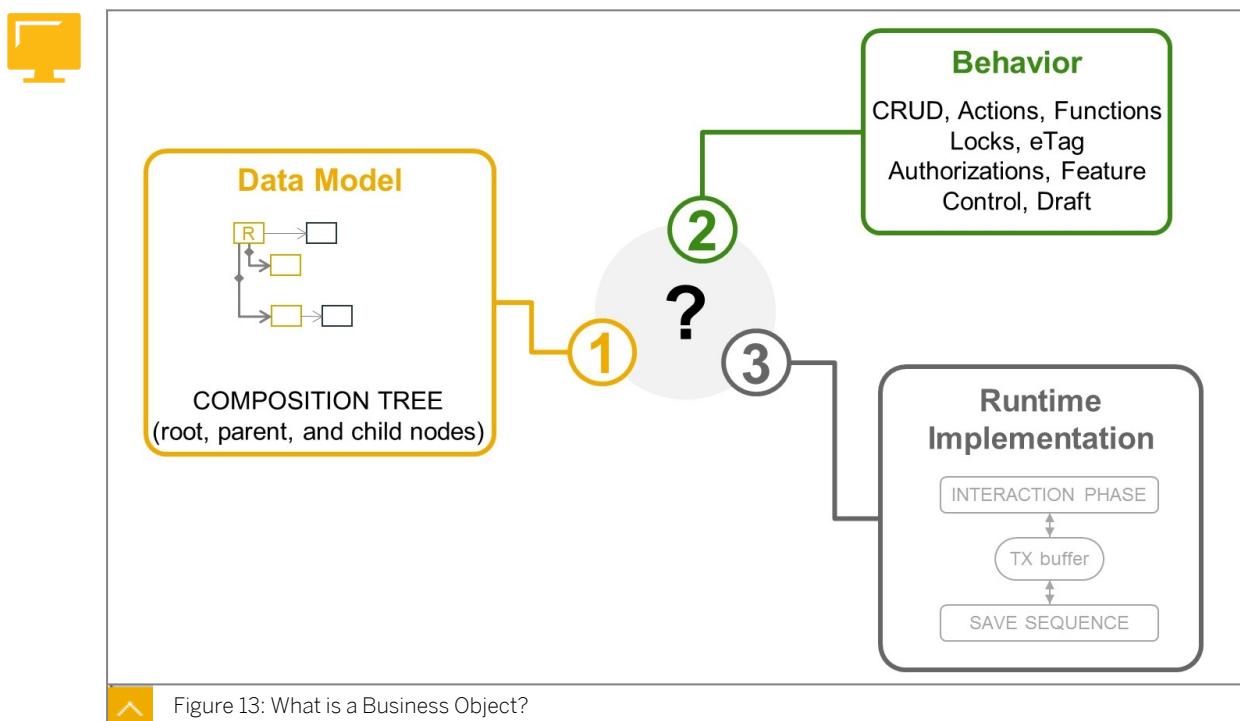
A service binding defines the communication protocol such as OData V2 or OData V4 and the kind of service to be offered for a consumer, such as UI services or a Web service.

SAP Fiori UI

SAP Fiori UI provides a designated UI for commonly used application patterns based on OData Services.

Web API

A Web API provides a public interface to access the OData service by any OData client.



A business object (BO) represents an entity of the data model. A RAP BO can either consist of a single node (Simple BO) or of a hierarchy of nodes (Composite BO). An example for a Composite BO is a document that consists of a header (root node) and items (child node).

The behavior of a BO defines operations that can be executed on the data, for example the standard operations Create, Update, Delete (CRUD), but also specific actions and functions.

It also provides feature control (for example, the definition which data is mandatory and which is read-only), concurrency control (for example, the handling of locks), and authorization control.

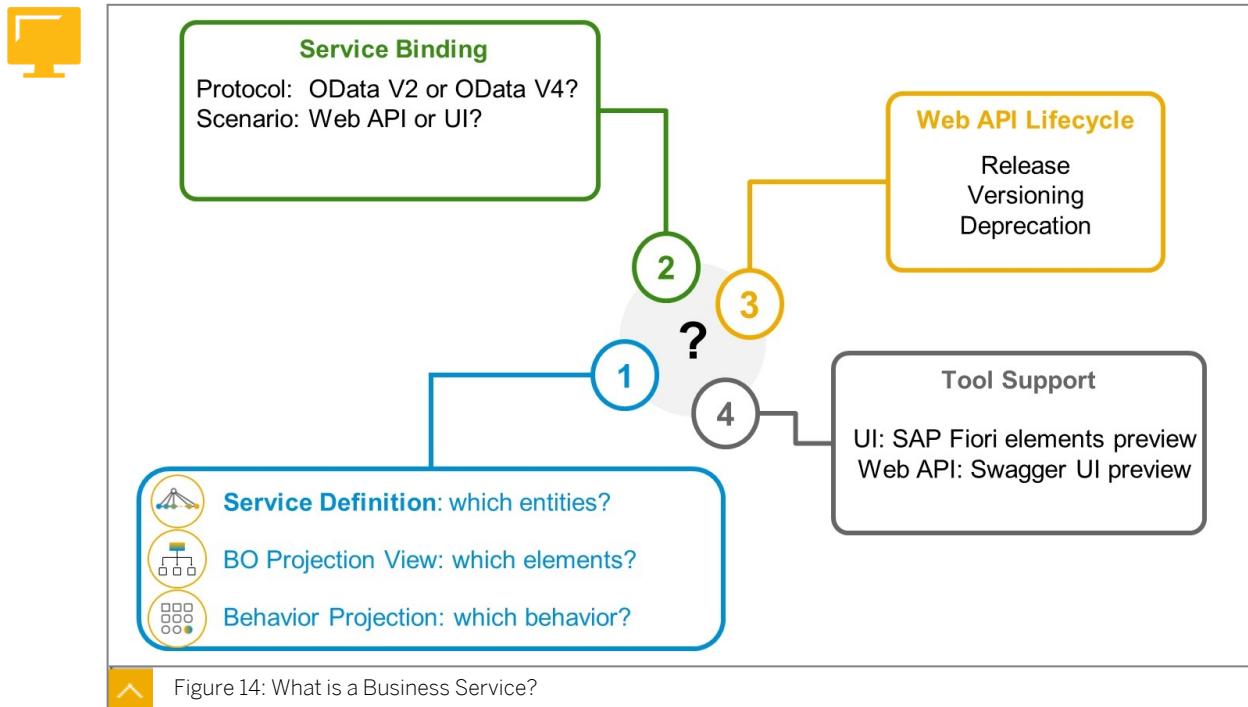


Figure 14: What is a Business Service?

In the context of the ABAP RESTful application programming model, a business service is a RESTful service that 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.

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 or OData V4, 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 need for reimplementation.

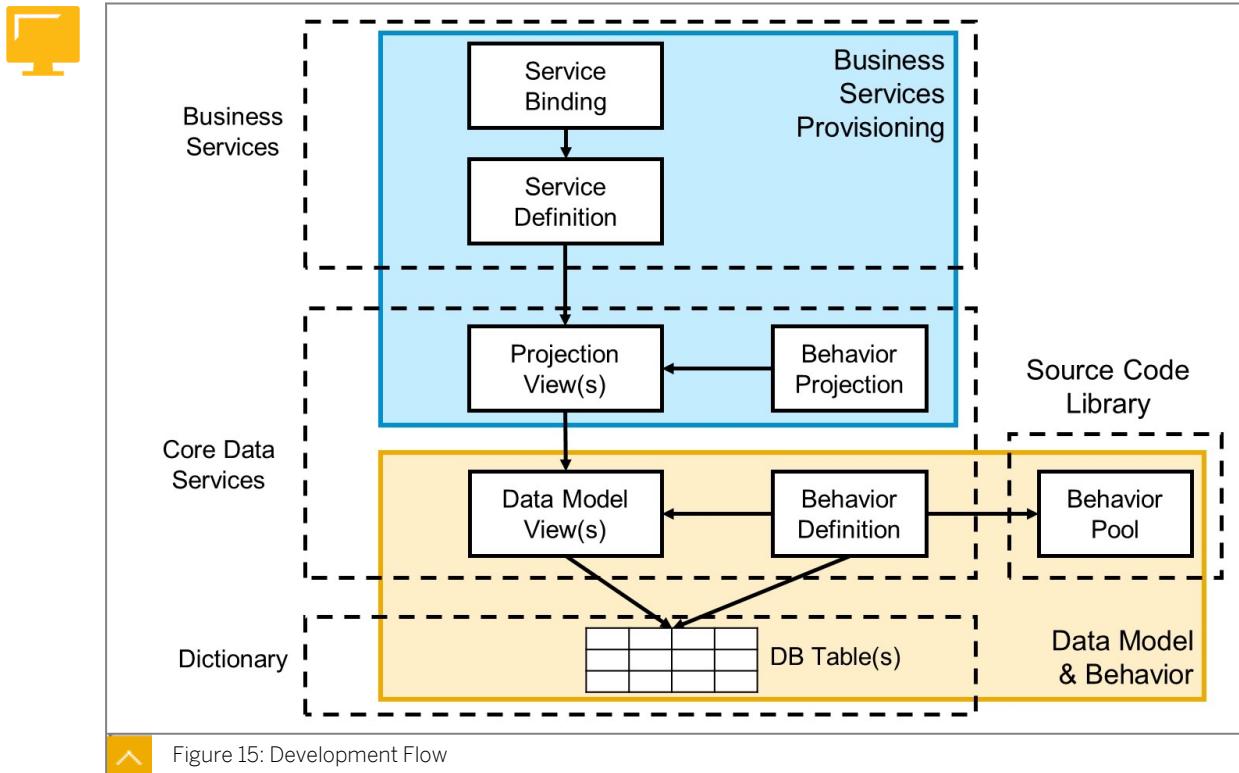


Figure 15: Development Flow

Developing a RAP application consists of the following main steps:

1. Provide the Database Tables.

Developing a RAP application starts with providing the database tables. Depending on the development scenario, these can be existing tables, legacy tables, or tables created specifically for this application.

2. Define the Data Model.

The data model of the Business Object is defined with CDS Views. Depending on whether it is a simple or a composite BO, one or more CDS Views are required. In case of a composite BO, this is also the place where you define the entity hierarchy.

3. Define and Implement the Behavior (Transactional apps only).

The behavior of a RAP BO is defined in a repository object called CDS Behavior Definition. Usually, the behavior of a RAP BO also requires some additional logic which is implemented in a certain type of global ABAP class, called a Behavior Pool. For a non-transactional application, for example, a list report, the behavior definition or implementation can be omitted.

4. Project the RAP Business Object and provide service specific Metadata.

The projection of the RAP BO consists of a data model projection, and, if a behavior has been defined, a behavior projection. To define a projection, you create one or more CDS projection views, a type of CDS View, and a Behavior Projection, a type of behavior definition. For UI services, the projection view(s) should be enriched with UI specific metadata. To support future extensibility of the application, we recommend placing the service specific annotations in metadata extensions.

5. Define the Service.

In RAP, a service is defined by creating a Service Definition. The service definition references the projection views and specifies which of them should be exposed, that is, which of them are visible for the service consumer.

6. Bind the Service and Test the Service.

To specify how the service should be consumed (UI or Web API) and via which protocol (OData V2 or OData V4), a service binding is needed. For UI services, a Preview is available

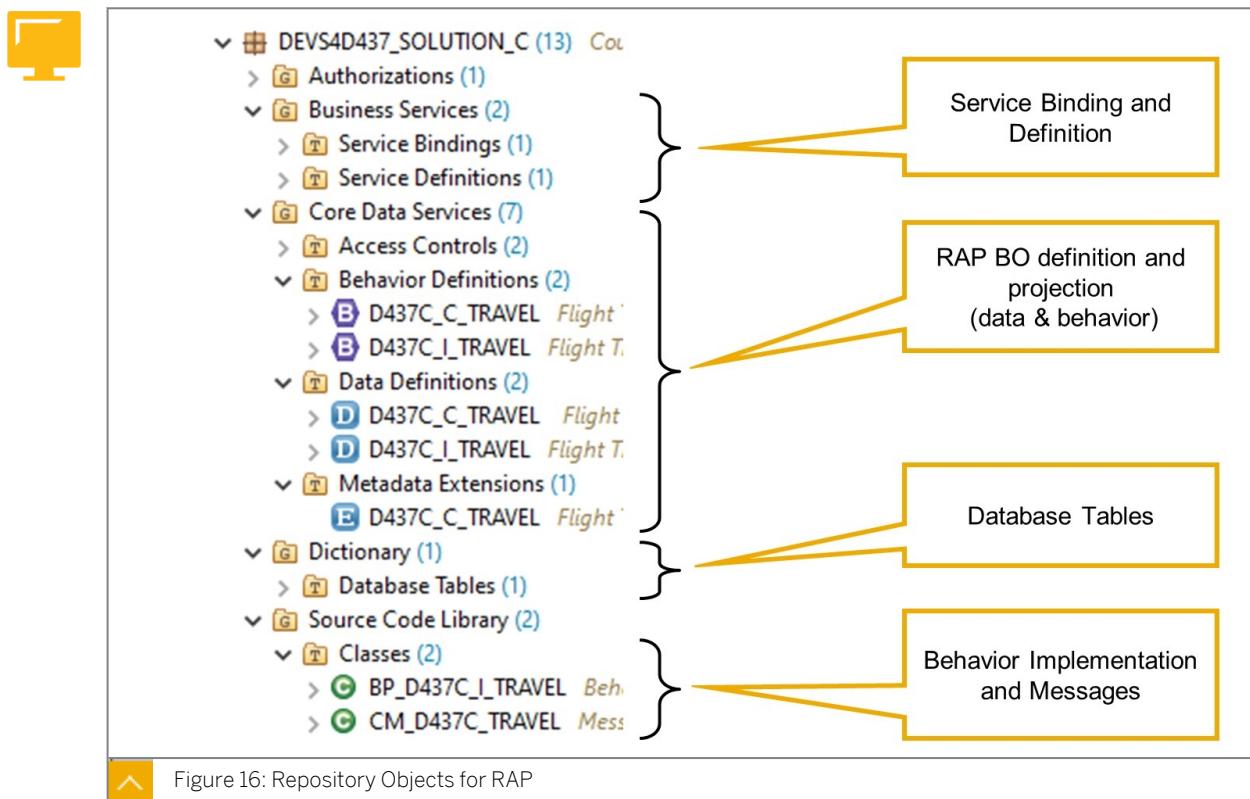


Figure 16: Repository Objects for RAP

When expanding the content of an ABAP Development Package in the Project Explorer of ADT, the repository objects are found in different categories of repository objects.

The database tables, along with the include structures, data elements, and domains needed for their definition, are found in category **Dictionary**.

The repository objects defining the data model and projection views, are found in the **Core Data Services** → **Data Definitions** node. According to the recommended naming pattern, the data model views should have a letter "I" (for Interface) and the projection views a letter "C" (for Consumption).

The repository objects for the definition and projection of the RAP BO behavior are located in **Core Data Services** → **Behavior Definitions**. Again, the letters "I" and "C" should help to distinguish between repository objects for definition and for projection.

Service Definitions and Service Bindings have their dedicated sub-nodes under the **Business Services** category.

The Behavior Implementation is done in behavior pools, which are global ABAP classes that fulfill certain requirements. Like all global ABAP classes, they can be found in **Source Code Library** → **Classes**. To make behavior pools distinguishable from other ABAP classes, the naming pattern requests their names to start with "BP_" or "<namespace>BP_" instead of the usual "CL_" or "<namespace>CL_" for ordinary ABAP classes.

Similarly, a certain group of global classes used to represent messages at runtime should have names starting with "CM_" or "<namespace>CM_".



Note:

Where and how CDS Access Controls and CDS Metadata come into play in a RAP project will be discussed later in this course.

The Business Scenario

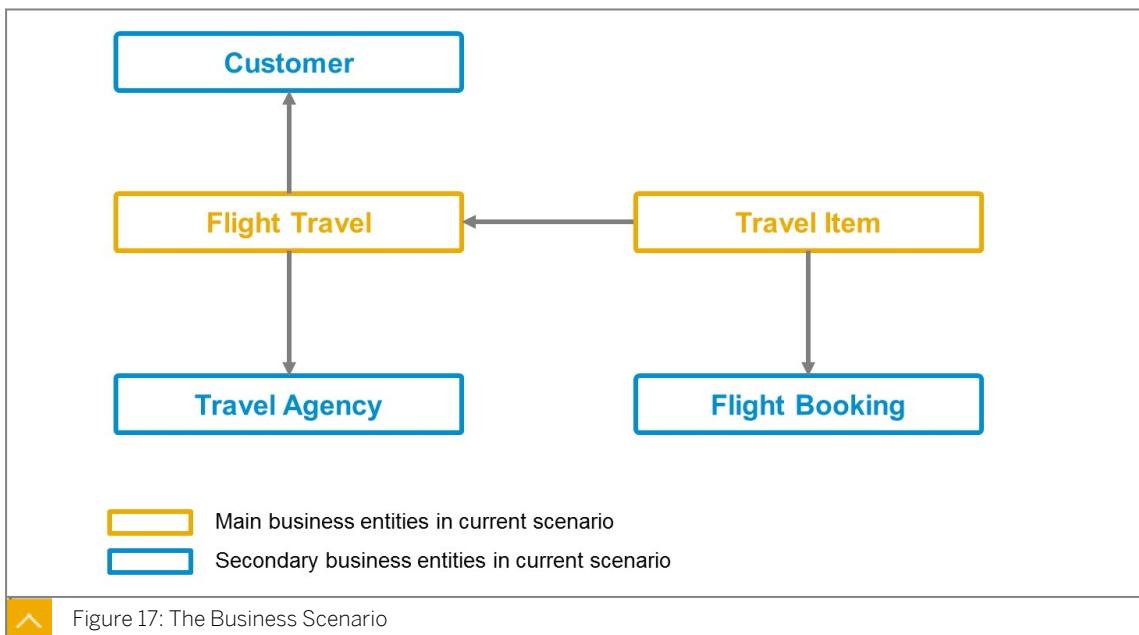


Figure 17: The Business Scenario

The Business Scenario in this course is based on the flight data model, which has been used in ABAP documentation and training for many years.

For this course, we introduce new entities to the model, Flight Travel that a customer books at a Travel Agency. The Flight Travel consists of several Travel Items, for example bookings on several flights (a flight and a return flight or a flight and a connecting flight).

To make things easier, we will start with Flight Travel as a simple BO that is not composite. Later in the course, we will introduce the Travel Agency.



- **Step 1 – Read-Only App** (Package DEVS4D437_SOLUTION_A)
 - OData UI Service for non-transactional app
 - Data Model, Projection and Business service
- **Step 2 – Transactional App with Action** (Package DEVS4D437_SOLUTION_B)
 - Behavior (definition, implementation, projection)
 - concurrency control and authorization control
- **Step 3 – Enable Direct Editing** (Package DEVS4D437_SOLUTION_C)
 - Update and create operations (managed)
 - Value helps, Input checks, dynamic default values, dynamic feature control
- **Step 4 – Draft-Enabled App** (Package DEVS4D437_SOLUTION_D)
 - Draft-Enable the Business Object
- **Step 5 – Composition** (Package DEVS4D437_SOLUTION_E)
 - Add child entity *Travel Item*
- **Step 6 – Unmanaged Scenario** (Package DEVS4D437_SOLUTION_F)
 - Transactional app to update *Travel Agency*

Figure 18: Outline of the Course

We will use the ABAP RESTful programming model (RAP) to build an OData UI service for Flight Travels and preview the OData UI Service in a generated app that is based on the List Report Floorplan of SAP Fiori elements.

First, we develop an OData Service for a non-transactional, read-only app that displays a list and a detail page for flight travels.

Then, we add behavior to the business object, namely an action that the user can execute by pressing a button on the UI. We will add concurrency control and authorization control to ensure data consistency and make sure that users only cancel travels for which they have the required authorization.

Next, we enable direct edition by the user. We discuss value helps, implement input checks, provide dynamic default values, and we discuss how to enable or disable editing dynamically.

Finally, we develop a RAP Business Object and a transactional app through which the user can update the master data of a Travel Agency. In this case, we want to reuse legacy code, namely existing function modules, to update an existing database table. Therefore, we use the unmanaged implementation type for the business object.

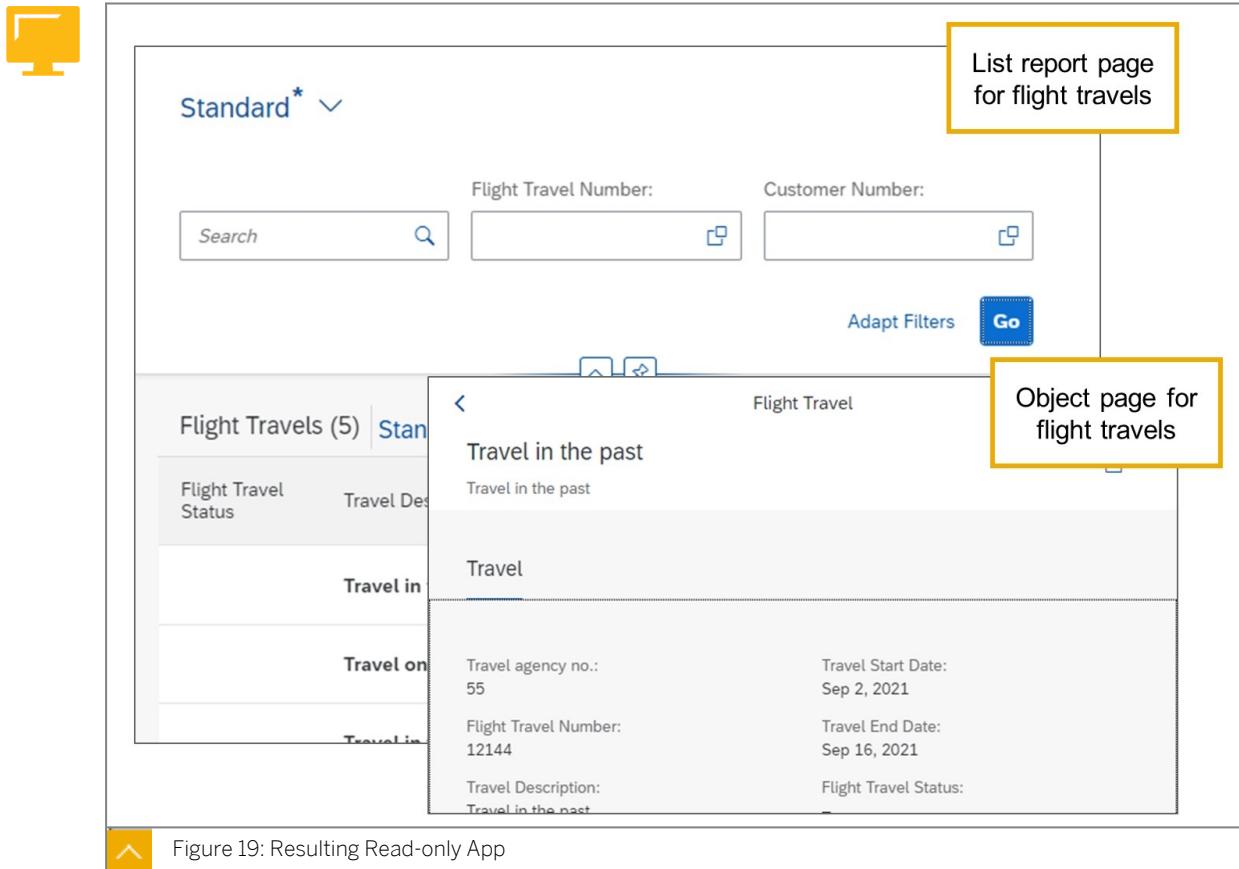


Figure 19: Resulting Read-only App

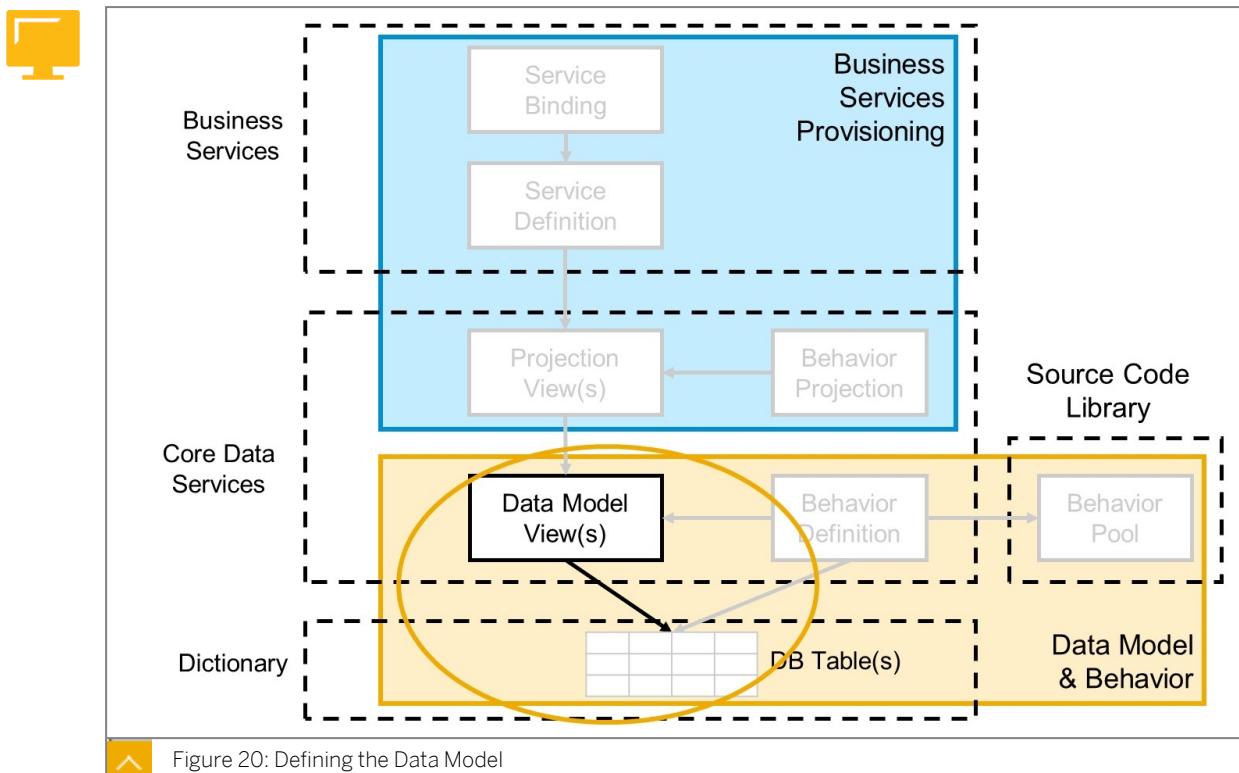


Figure 20: Defining the Data Model

To define the data model, we create a database table and a CDS View entity.

Unit 1

Exercise 1

Define a CDS-based Data Model

Business Scenario

In this exercise, you create copies of the repository objects that define a CDS-based data model.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 1: Solution

Repository Object Type	Repository Object ID
Transparent Table	D437A_TRAVEL
Data Definition	D437A_I_TRAVEL

Task 1: Open ABAP Development Tools in Eclipse (ADT) and Create a Package

Start the Eclipse IDE, open the ABAP Perspective, and create an ABAP Project to connect to your ABAP development system. Create an ABAP Package (suggested name: Z##_TRAVEL) and add the package to the favorite packages of your ABAP project.

1. Start the Eclipse IDE.
2. Change to the ABAP perspective.
3. Create an ABAP Project.
4. Create an ABAP Package (suggested name: Z##_TRAVEL). Create it with the following attributes:

Table 2: ABAP Package

Field Name	Value
Software Component	HOME
Application Component	CA
Transport Layer	ZS4D

5. Add your new package to the favorite packages of your project.

Task 2: Copy and Fill a Database Table for Travel Data

Create a copy of the database table D437T_TRAVEL (suggested name: Z##_TRAVEL). Use executable program D437_TRAVEL_FILL to fill the table with data.

1. In the Project Explorer, navigate to the *D437T_TRAVEL* database table in the *DEVS4D437_TEMPLATES* package.
2. Create a copy of the database table.
3. Activate the new database table.
4. Run the executable program *D437_TRAVEL_FILL* to fill your table with data.

Task 3: Copy a Root View Entity

Create a copy of data source *D437T_I_TRAVEL* (suggested name: *Z##_I_TRAVEL*). Adjust the view definition to read from your database table *Z##_TRAVEL*.

1. Create a copy of data source *D437T_I_TRAVEL* (suggested name: *Z##_I_TRAVEL*), assign it to your own package and use the same transport request as before.
2. Edit the data source. Adjust the view name to use a mixture of upper case and lower case (*Z##_I_TRAVEL*) and make the view read from your own database table.
3. Activate the data source.
4. Test the CDS view by opening it in the data preview tool.

Unit 1

Solution 1

Define a CDS-based Data Model

Business Scenario

In this exercise, you create copies of the repository objects that define a CDS-based data model.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 1: Solution

Repository Object Type	Repository Object ID
Transparent Table	D437A_TRAVEL
Data Definition	D437A_I_TRAVEL

Task 1: Open ABAP Development Tools in Eclipse (ADT) and Create a Package

Start the Eclipse IDE, open the ABAP Perspective, and create an ABAP Project to connect to your ABAP development system. Create an ABAP Package (suggested name: Z##_TRAVEL) and add the package to the favorite packages of your ABAP project.

1. Start the Eclipse IDE.
 - a) Open the Windows Start Menu and choose the latest Eclipse IDE version.
2. Change to the ABAP perspective.
 - a) Close the *Welcome* tab.
 - b) From the Eclipse menu, choose *Window → Perspective → Open Perspective → Other ...*.
 - c) Choose *ABAP* and *Open*.
3. Create an ABAP Project.
 - a) From the Eclipse menu, choose *File → New → ABAP Project*.
 - a) From the list, choose *S4D SAP GUI non-SNC* and choose *Next >*.
 - a) Choose *Next >*.
 - b) Enter client, user name, password, and language (logon details will be provided by your instructor).
4. Create an ABAP Package (suggested name: Z##_TRAVEL). Create it with the following attributes:

Table 2: ABAP Package

Field Name	Value
Software Component	HOME
Application Component	CA
Transport Layer	ZS4D

- a) Right-click the project and choose New → ABAP Package.
 - a) Enter the suggested name and a description and choose Next >.
 - b) Enter the data as shown in the table and choose Next >.
 - c) From the list, select the transport request in which your user is involved and choose Finish.
5. Add your new package to the favorite packages of your project.
- a) Right-click the Favorite Packages node and choose Add Package
 - b) Enter the name of your new package and choose OK.

Task 2: Copy and Fill a Database Table for Travel Data

Create a copy of the database table *D437T_TRAVEL* (suggested name: *Z##_TRAVEL*). Use executable program *D437_TRAVEL_FILL* to fill the table with data.

1. In the Project Explorer, navigate to the *D437T_TRAVEL* database table in the *DEVS4D437_TEMPLATES* package.
 - a) Perform an ABAP Object Search or add the package to your Favorite Packages.
2. Create a copy of the database table.
 - a) Right-click the *D437T_TRAVEL* database table and choose Duplicate
 - b) Enter the name of your own package and the suggested name for the new database table, and choose Next >.
 - c) Select the same transport request as before and choose Finish.
3. Activate the new database table.
 - a) In the Eclipse toolbar, choose Activate or press Ctrl + F3.
4. Run the executable program *D437_TRAVEL_FILL* to fill your table with data.
 - a) In the Eclipse menu, choose Run → Run ABAP Development Object ... or press Alt + F8.
 - b) Enter **D437_TRAVEL_FILL** and choose OK.
 - c) Confirm that the suggested entry matches the name of the table you just created and choose Execute or press F8.
 - d) From the toolbar of the ALV Grid control, choose Generate Data.

Task 3: Copy a Root View Entity

Create a copy of data source *D437T_I_TRAVEL* (suggested name: *Z##_I_TRAVEL*). Adjust the view definition to read from your database table *Z##_TRAVEL*.

1. Create a copy of data source *D437T_I_TRAVEL* (suggested name: *Z##_I_TRAVEL*), assign it to your own package and use the same transport request as before.
 - a) Perform this step as you did in the previous task.
2. Edit the data source. Adjust the view name to use a mixture of upper case and lower case (*Z##_I_TRAVEL*) and make the view read from your own database table.
 - a) Edit the `FROM` clause of the `DEFINE VIEW` statement and replace the template database table with your own database table.
3. Activate the data source.
 - a) In the Eclipse toolbar, choose *Activate* or press *Ctrl + F3*.
4. Test the CDS view by opening it in the data preview tool.
 - a) Right-click anywhere on the source code of the data definition and choose *Open with → Data Preview* or press *F8*.



LESSON SUMMARY

You should now be able to:

- Understand the concept of RAP
- Use ABAP development tools
- Explain the RAP architecture and business use case

Defining an OData UI Service



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define a CDS projection view
- Enrich a projection view with UI metadata
- Create and preview an OData UI service

Data Model Projection

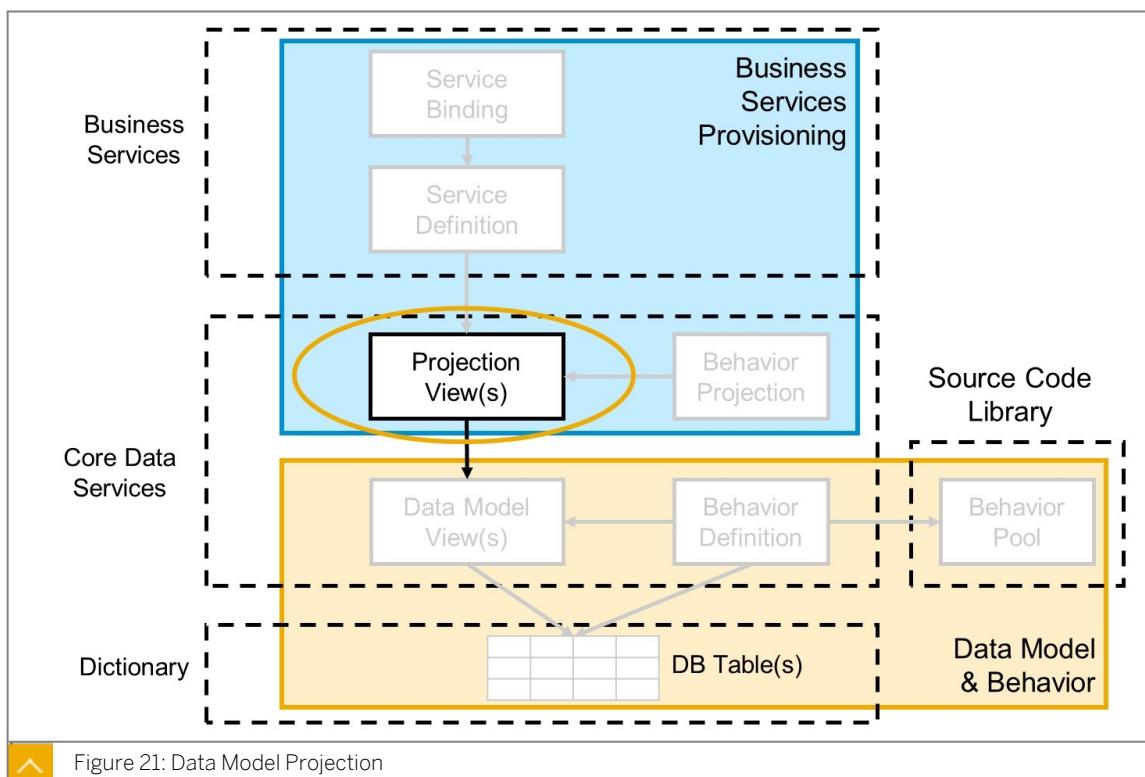


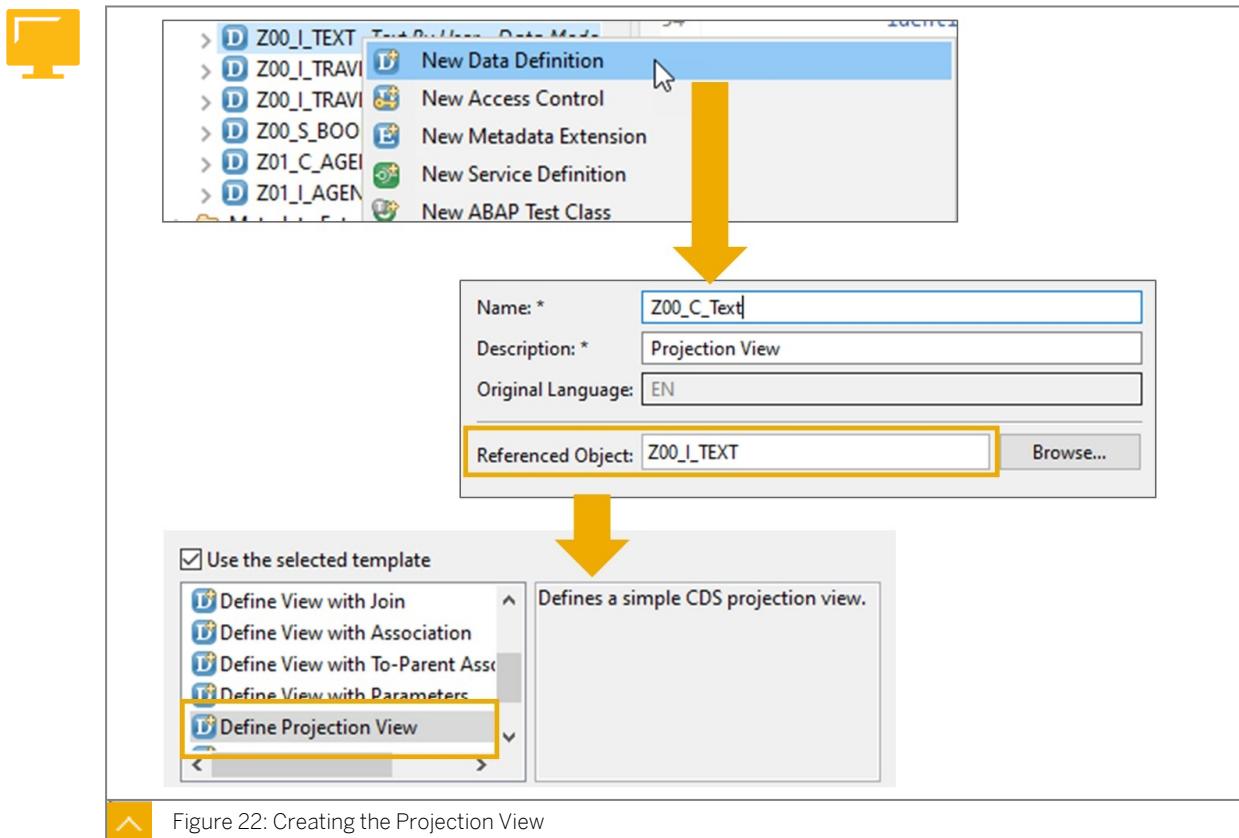
Figure 21: Data Model Projection

The data model projection consists of one CDS projection view for each data definition view of the data model.

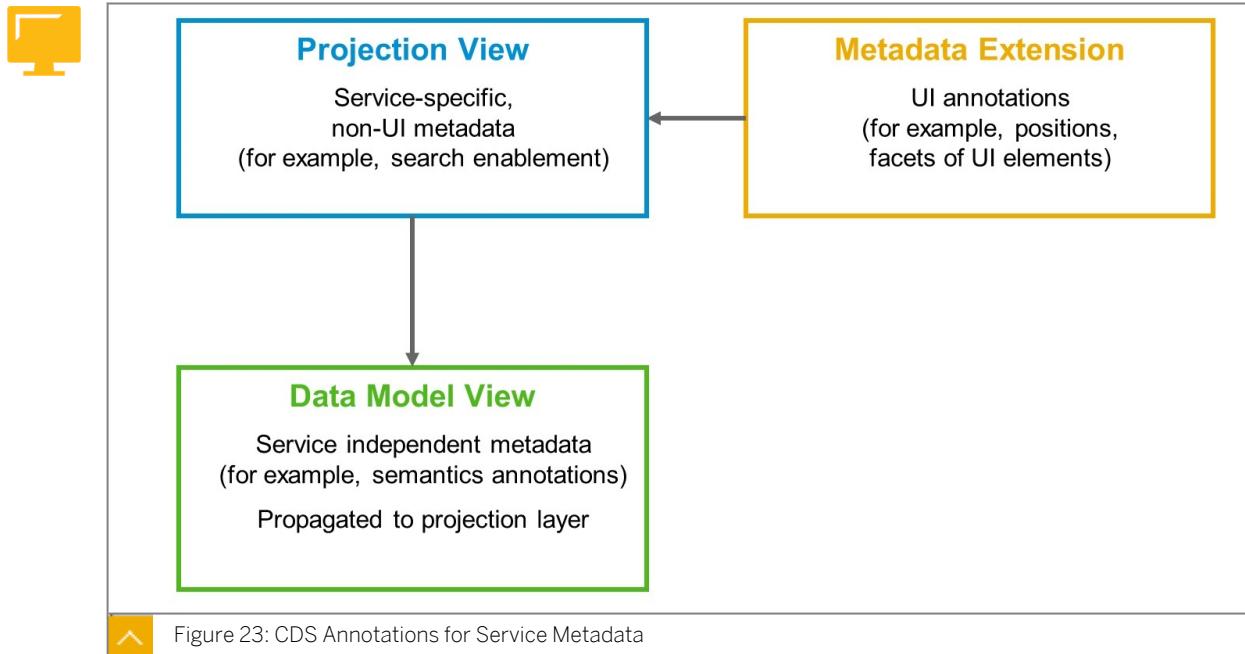
Projection views provide means within the specific service to define service-specific projections including denormalization of the underlying data model. Fine-tuning, which does not belong to the general data model layer, is defined in projection views, for example, UI annotations, value helps, calculations,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

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 done automatically. That means that the elements can be renamed to match the business service context of the

respective projection. You cannot 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, you can add virtual elements to projection views. These elements must be calculated by ABAP logic.



To create the data definition for a CDS Projection view, we recommend using the context menu on the name of the data definition view. By doing so, the name of the data definition view is automatically set as *Referenced Object* and template define *Projection View* is pre-selected by default.



From a design time point of view, the projection layer is the first service-specific layer. All service specific metadata must be defined in the CDS projection views via CDS annotations.

Element annotations that are not service-specific should be placed in the data model views from where they are propagated into the projection layer.

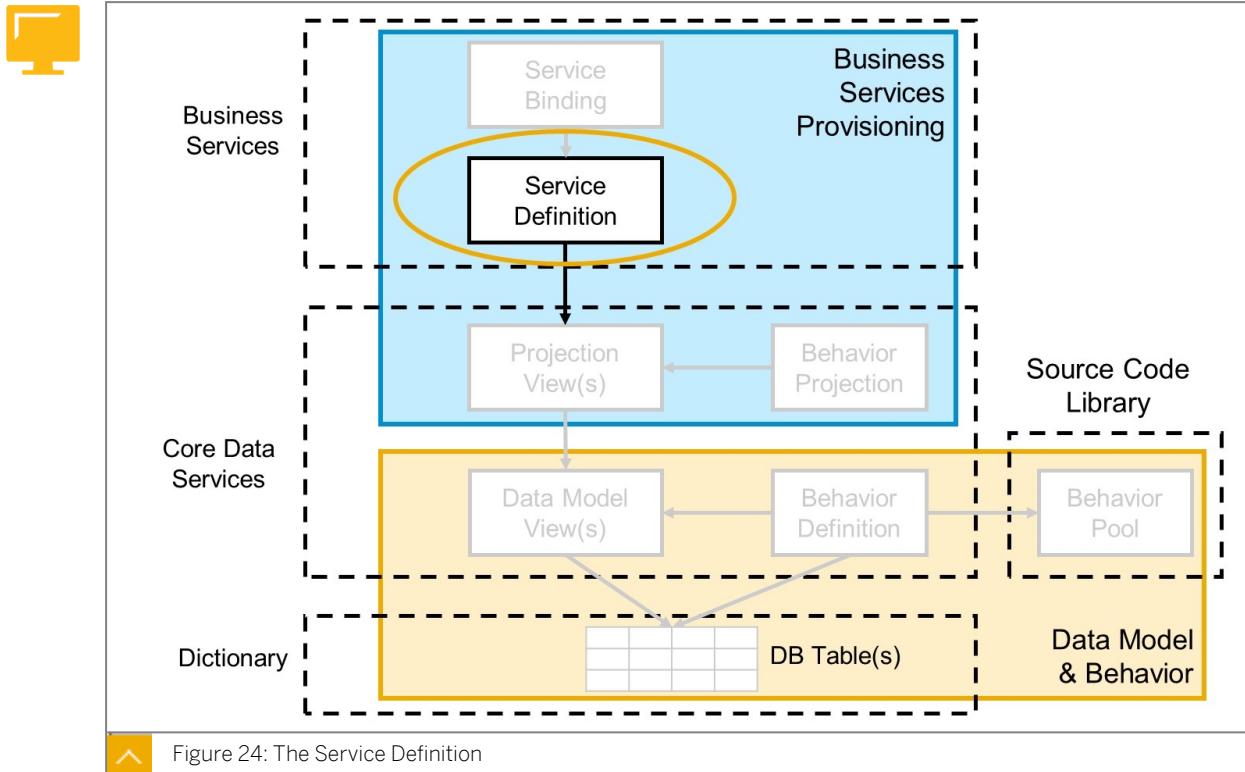
An example is annotation `@semantics.amount.currencycode`, which is used to establish the connection between an amount field and its currency code field.

The following service 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 Help

It is common practice to outsource the UI-annotations of a projection view in a metadata extension. This increases readability and facilitates later adjustments of the UI through additional metadata extensions.

Service Definition



A business service definition (or service definition) describes which CDS entities of a data model are to be exposed so that a specific business service can be enabled. It is an ABAP Repository object that describes the consumer-specific but protocol-agnostic perspective on a data model. This means that a service definition itself is independent of the version or type of the protocol that is used for the business service.

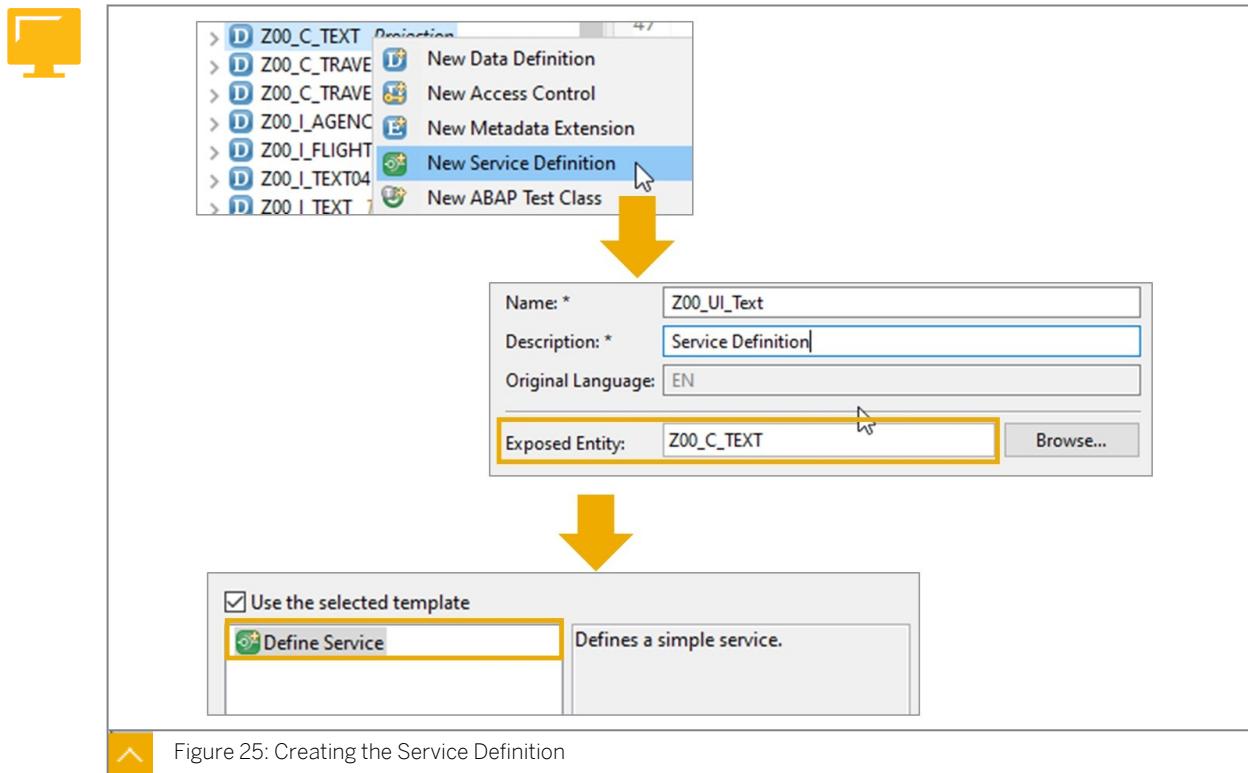


Figure 25: Creating the Service Definition

To create the service definition, we recommend using the context menu on the name of the CDS projection view. By doing so, the name of the projection view is automatically set as *Exposed Entity*.



Note:

At present, only one template is available for service definitions.

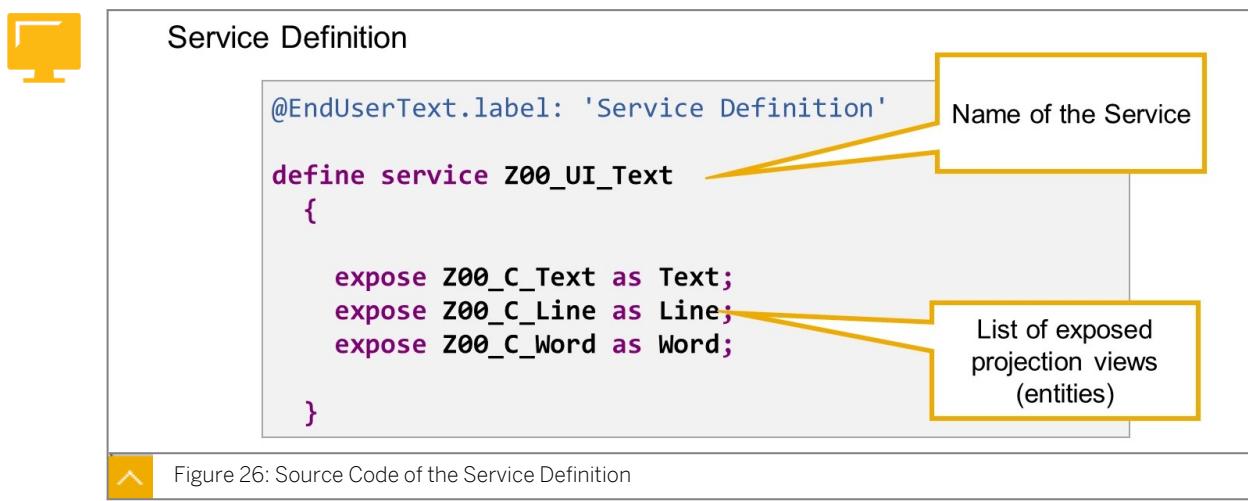


Figure 26: Source Code of the Service Definition

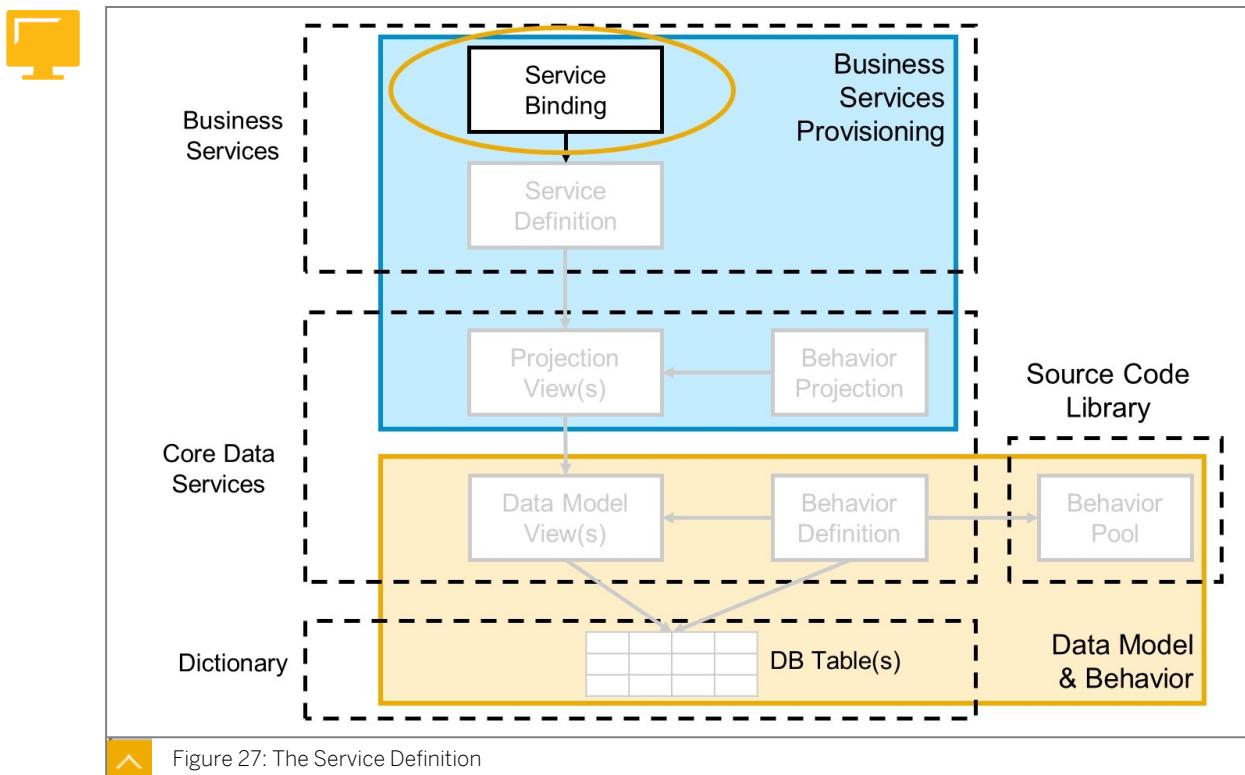
The source code of the actual service definition is preceded by the optional CDS annotation `@EndUserText.label` that is available for all objects that can contain CDS annotations.

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

Because a service definition, 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. However, if no reuse of the same service definition is planned for UI and API services, the prefix may follow the rules of the service binding, that, `UI_` if the service is exposed as a UI service and `API_` if the service is exposed as Web API.

A pair of curly brackets surrounds a list of `EXPOSE` statements with the names of the exposed projection views. The alias names after the keyword `AS` are optional. They define alternative names to be used by the consumer of the service.

Service Binding



The business service binding (or service binding) is an ABAP Repository object used to bind a service definition to a client-server communication protocol, such as OData.

A service binding relies directly on a service definition derived from the underlying CDS-based data model. Based on an individual service definition, several 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.

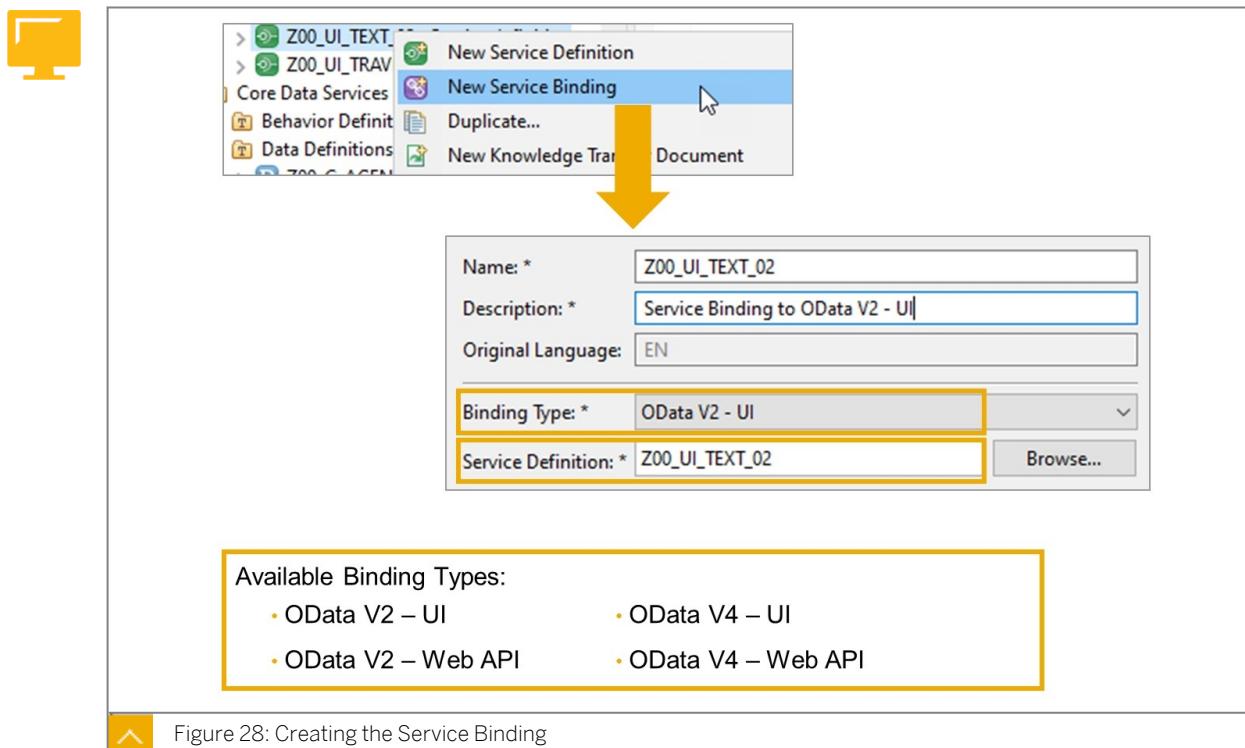


Figure 28: Creating the Service Binding

To create a business service binding, we recommend that you use the context menu on the name of the service definition. By doing so, the name of the service definition is automatically set as *Service Definition* in the *Create* dialog.

To choose a binding type, both the name of the service binding and a description are required.

There are currently four different values for Binding Type available, with different rules for the service binding name:

OData V2 - UI

Defines an UI service based on version 2 of the OData Protocol. The naming convention is: Prefix: UI_, Suffix: _02.

OData V2 - Web API

Defines a Web API service based on version 2 of the OData Protocol. The naming convention is: Prefix: API_, Suffix: _02.

OData V4 - UI

Defines an UI service based on version 4 of the OData Protocol. The naming convention is: Prefix: UI_, Suffix: _04.

OData V4 - Web API

Defines a Web API service based on version 4 of the OData Protocol. The naming convention is: Prefix: API_, Suffix: _04.

Note:

We recommend using to use OData V4 wherever possible for transactional services.

We will start with an OData V2 UI service, because SAP Fiori element UIs only fully support V4 in draft scenarios, which we will cover later.

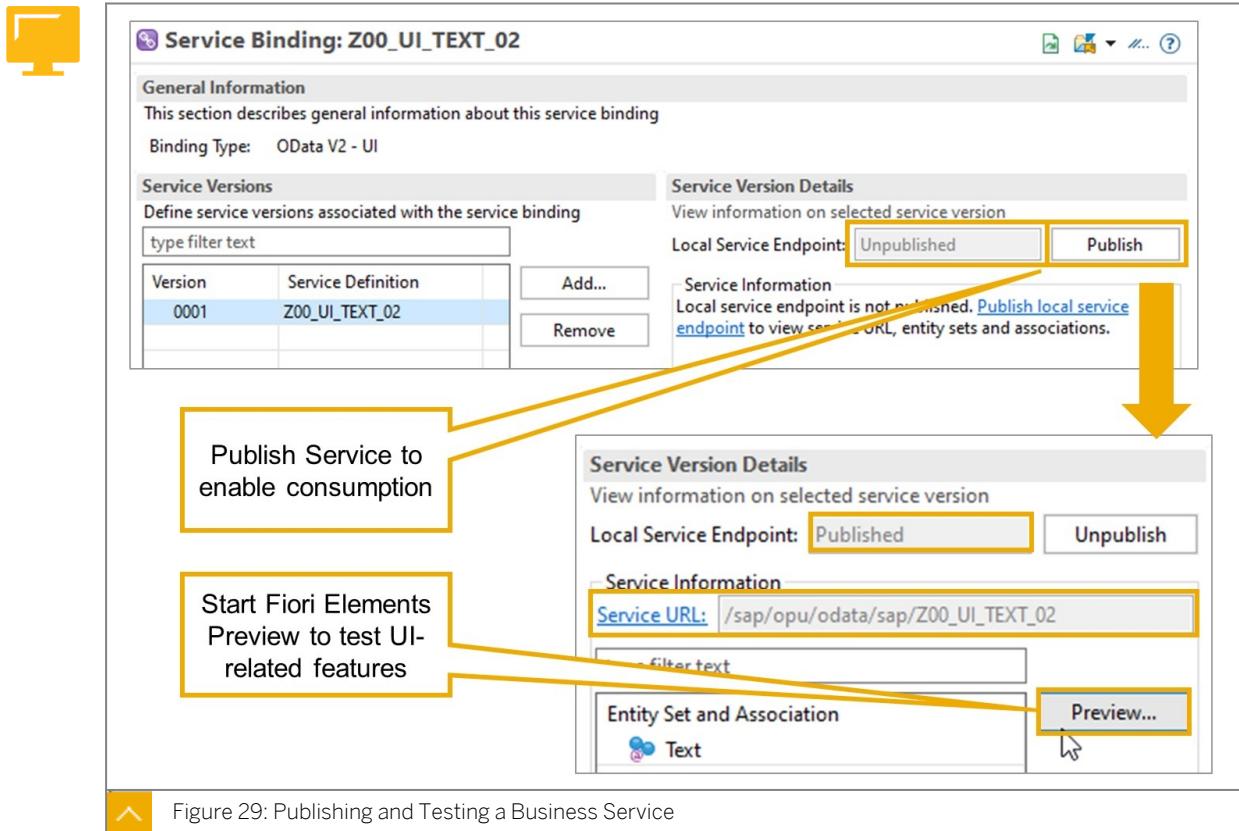


Figure 29: Publishing and Testing a Business Service

After creating the Service Binding, the local service endpoint of an OData service must be published using the *Publish* button in the service binding editor. This triggers several task lists to enable the service for consumption. By publishing the service binding, the service is only enabled for the current system. It is not consumable from other systems.

**Note:**

The service binding needs to be active to be published. To activate the service binding use the activation button in the tool bar.

After publishing, 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>**

You can start a *Fiori Elements Preview* directly from the service binding. With this, you can test UI-related features directly from your ABAP system.

Unit 1

Exercise 2

Define and Preview an OData UI Service

Business Scenario

In this exercise, you create a projection of your CDS-based data model and enrich it with search metadata and UI metadata. Based on this projection, you define and bind an OData UI Service and preview the result.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 3: Solution

Repository Object Type	Repository Object ID
Data Definition (Projection)	D437A_C_TRAVEL
Metadata Extension	D437A_C_TRAVEL
Service Definition	D437A_UI_TRAVEL
Service Binding	D437A_UI_TRAVEL_02

Task 1: Create a CDS Projection View

Create a CDS projection view (suggested name: `Z##_C_Travel`, that reads all fields from your data model view `Z##_I_Travel`.

1. Create a new data definition (suggested name: `Z##_C_Travel`) for a CDS view that is based on your data model view `Z##_I_Travel`. Assign it to your own package and use the same transport request as before. Use the *Define Projection View* template to create the data definition.
2. Adjust the definition of the projection view. Add keyword `root`, which is needed because the projection view reads from a data model view that is a root view.
3. Activate and test the CDS projection view.

Task 2: Enrich the CDS Projection View with Meta Data

Enrich your projection view with search meta data and UI meta data. For the UI meta data, create a copy of metadata extension `D437T_C_TRAVEL` (suggested name: `Z##_C_TRAVEL`) and replace the name of the annotated view with your projection view.

1. Edit the data definition of your `Z##_C_Travel` projection view and enable the search functionality with `AgencyID` and `CustomerID` as the default search elements.
2. For your projection view, allow extensions with metadata extensions.

3. Activate the data source.
4. Create a copy of the *D437t_C_Travel* metadata extension (suggested name: *Z##_C_TRAVEL*). Assign it to your own package and use the same transport request as before.
5. Edit the metadata extension. In the `ANNOTATE VIEW` statement, replace `D437t_C_Travel` with the name of your projection view *Z##_C_Travel*.
6. Activate the metadata extension.

Task 3: Create an OData UI Service

Create a service definition (suggested name: *Z##_UI_TRAVEL*) and a service binding (suggested name: *Z##_UI_TRAVEL_02*), that exposes your *Z##_C_Travel* projection view as an OData V2 UI Service.

1. Create a service definition (suggested name: *Z##_UI_TRAVEL*) for your *Z##_C_Travel* projection view. Assign it to your own package and use the same transport request as before.
2. Activate the service definition.
3. Create a service binding (suggested name: *Z##_UI_TRAVEL_02*) that binds your service *Z##_UI_TRAVEL* as an OData V2 UI Service.

Do you have to activate repository object service binding?

Task 4: Publish and Test the OData UI Service

Publish the local service endpoint and preview the service as an SAP Fiori elements app.

1. In your service binding, publish the local service endpoint.
2. Preview the service as an SAP Fiori elements app.

Unit 1

Solution 2

Define and Preview an OData UI Service

Business Scenario

In this exercise, you create a projection of your CDS-based data model and enrich it with search metadata and UI metadata. Based on this projection, you define and bind an OData UI Service and preview the result.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 3: Solution

Repository Object Type	Repository Object ID
Data Definition (Projection)	D437A_C_TRAVEL
Metadata Extension	D437A_C_TRAVEL
Service Definition	D437A_UI_TRAVEL
Service Binding	D437A_UI_TRAVEL_02

Task 1: Create a CDS Projection View

Create a CDS projection view (suggested name: Z##_C_Travel, that reads all fields from your data model view Z##_I_Travel.

1. Create a new data definition (suggested name: Z##_C_Travel) for a CDS view that is based on your data model view Z##_I_Travel. Assign it to your own package and use the same transport request as before. Use the *Define Projection View* template to create the data definition.
 - a) In the *Project Explorer* view on the left, open the context menu for the data definition of your data model view.
 - b) From the context menu, choose *New Data Definition*.
 - c) Enter the name of the new data definition and a description and choose *Next >*.
 - d) Select the same transport request as before and choose *Next >*.
 - e) From the list of templates, choose *Define Projection View* and choose *Finish*.
2. Adjust the definition of the projection view. Add keyword `root`, which is needed because the projection view reads from a data model view that is a root view.
 - a) Add keyword `root` between keywords `define` and `view`.
3. Activate and test the CDS projection view.

- a) Perform this step as before.

Task 2: Enrich the CDS Projection View with Meta Data

Enrich your projection view with search meta data and UI meta data. For the UI meta data, create a copy of metadata extension *D437T_C_TRAVEL* (suggested name: *Z##_C_TRAVEL*) and replace the name of the annotated view with your projection view.

1. Edit the data definition of your *Z##_C_Travel* projection view and enable the search functionality with *AgencyID* and *CustomerID* as the default search elements.
 - a) In the data definition of your projection view, insert the `@Search.searchable: true` view annotation before the `define root view entity` statement.
 - b) Insert the element annotation `@Search.defaultSearchElement: true` in front of the *AgencyID* and *CustomerID* view elements.
2. For your projection view, allow extensions with metadata extensions.
 - a) In the data definition of your projection view, add the `@Metadata.allowExtensions: true` view annotation.
3. Activate the data source.
 - a) Perform this step as before.
4. Create a copy of the *D437t_C_Travel* metadata extension (suggested name: *Z##_C_TRAVEL*). Assign it to your own package and use the same transport request as before.
 - a) Perform this step as you did in the previous task.
5. Edit the metadata extension. In the `ANNOTATE VIEW` statement, replace *D437t_C_Travel* with the name of your projection view *Z##_C_Travel*.
 - a) Perform this step as you did in the previous task.
6. Activate the metadata extension.
 - a) Perform this step as before.

Task 3: Create an OData UI Service

Create a service definition (suggested name: *Z##_UI_TRAVEL*) and a service binding (suggested name: *Z##_UI_TRAVEL_02*), that exposes your *Z##_C_Travel* projection view as an OData V2 UI Service.

1. Create a service definition (suggested name: *Z##_UI_TRAVEL*) for your *Z##_C_Travel* projection view. Assign it to your own package and use the same transport request as before.
 - a) In the *Project Explorer* view on the left, open the context menu for the data definition of your projection view.
 - b) From the context menu, choose *New Service Definition*.
 - c) Enter the name of the service definition and a description and choose *Next >*.
 - d) Select the same transport request as before and choose *Next >*.
 - e) From the list of templates, choose *Define Service* and choose *Finish*.
2. Activate the service definition.
 - a) Perform this step as before.

3. Create a service binding (suggested name: Z##_UI_TRAVEL_02) that binds your service Z##_UI_TRAVEL as an OData V2 UI Service.
 - a) In the *Project Explorer* view on the left, open the context menu for your service definition.
 - b) From the context menu, choose *New Service Binding*.
 - c) Enter the name of the service binding and a description.
 - d) Set the *Binding Type* to *OData V2 - UI* and choose *Next >*.
 - e) Select the same transport request as before and choose *Finish*.

Do you have to activate repository object service binding?

No, there are no inactive versions of service bindings.

Task 4: Publish and Test the OData UI Service

Publish the local service endpoint and preview the service as an SAP Fiori elements app.

1. In your service binding, publish the local service endpoint.
 - a) Open your service binding.
 - b) On the right, under *Service Versions Detail*, choose *Publish*.
2. Preview the service as an SAP Fiori elements app.
 - a) Under *Entity Set and Association*, choose the name of your projection view.
 - b) Choose *Preview....*

The *Preview for Fiori Elements App* opens in a new browser window or browser tab.



Note:

You might need to enter your user and password for the ABAP application server, when opening the preview for the first time.



LESSON SUMMARY

You should now be able to:

- Define a CDS projection view
- Enrich a projection view with UI metadata
- Create and preview an OData UI service

UNIT 2

RAP Business Objects (RAP BOs)

Lesson 1

Defining RAP Business Objects and their Behavior	39
Exercise 3: Define an RAP Business Object and its Behavior	49

Lesson 2

Using Entity Manipulation Language (EML) to Access RAP Business Objects	56
Exercise 4: Read and Update an RAP Business Object	65
Exercise 5: Optional: Use Long Version of EML Statements and Entity Aliases	73

Lesson 3

Understanding Concurrency Control in RAP	82
Exercise 6: Establish Optimistic Concurrency Control	87

Lesson 4

Defining Actions and Messages	93
Exercise 7: Define and Implement an Action	107

Lesson 5

Implementing Authority Checks	123
Exercise 8: Implement Authority Checks	133

UNIT OBJECTIVES

- Create a CDS behavior definition
- Create a CDS behavior projection
- Describe the purpose and syntax of EML
- Describe the derived data types for RAP Business Objects
- Use the Entity Manipulation Language (EML)
- Describe pessimistic concurrency control (locking)
- Enable optimistic concurrency control

- Define and implement an action
- Expose actions to OData services
- Provide a button in SAP Fiori elements
- Define exception classes for RAP
- Access application data in behavior implementations
- Restrict read access with access controls
- Implement explicit authority checks

Defining RAP Business Objects and their Behavior

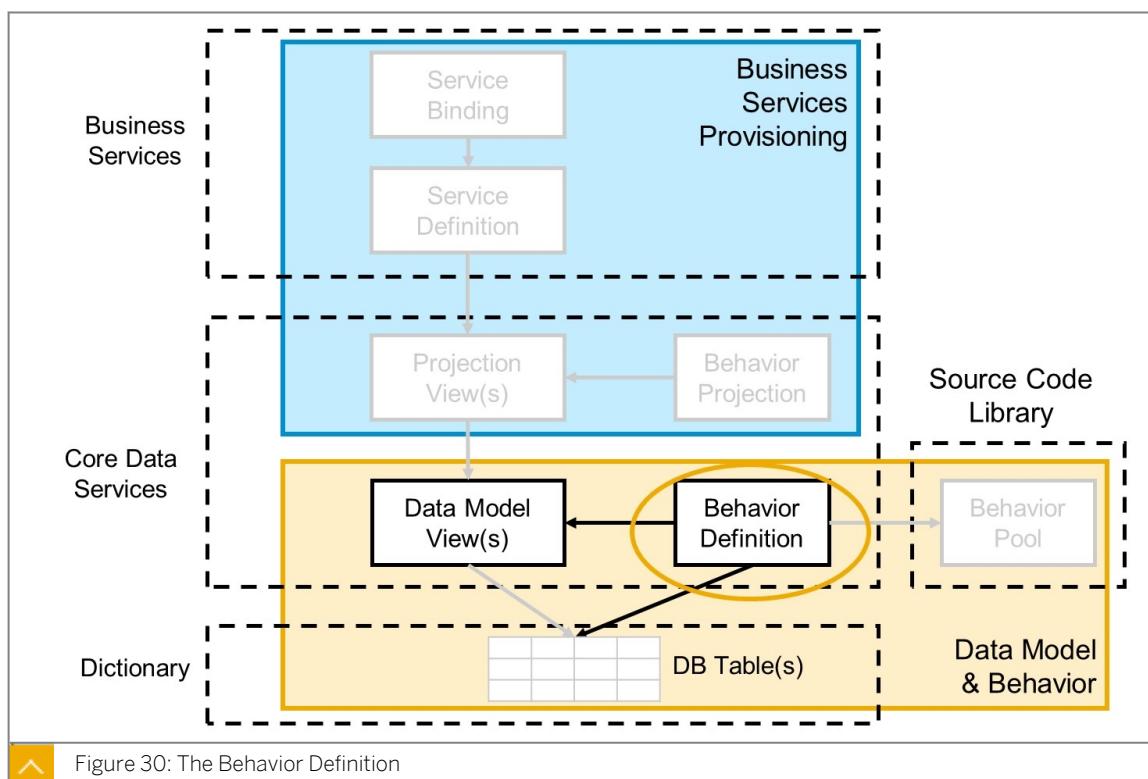


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create a CDS behavior definition
- Create a CDS behavior projection

Behavior Definition

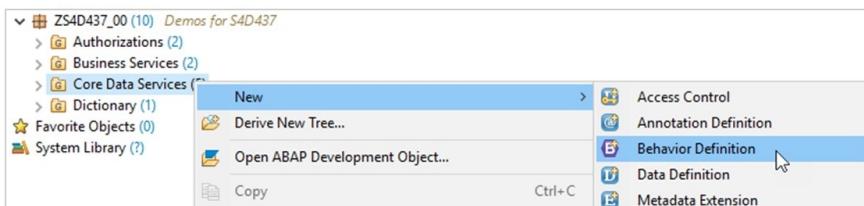


To specify the business object's behavior, the behavior definition of the corresponding development object is used. A business object behavior definition (behavior definition for short) is an ABAP Repository object that describes the behavior of a business object in the context of the ABAP RESTful application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

A behavior definition always refers to a CDS data model. It 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 (which also handles all included child entities that are included in the composition tree).



Alternative A: Start from Package



Alternative B: Start from CDS View (Data Definition)

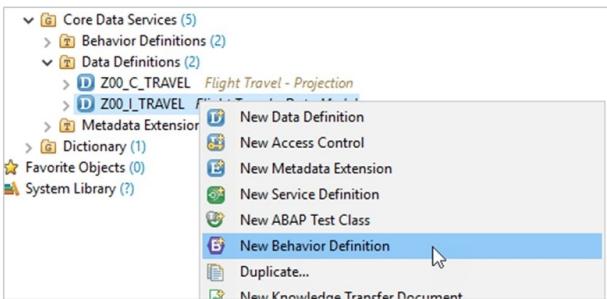
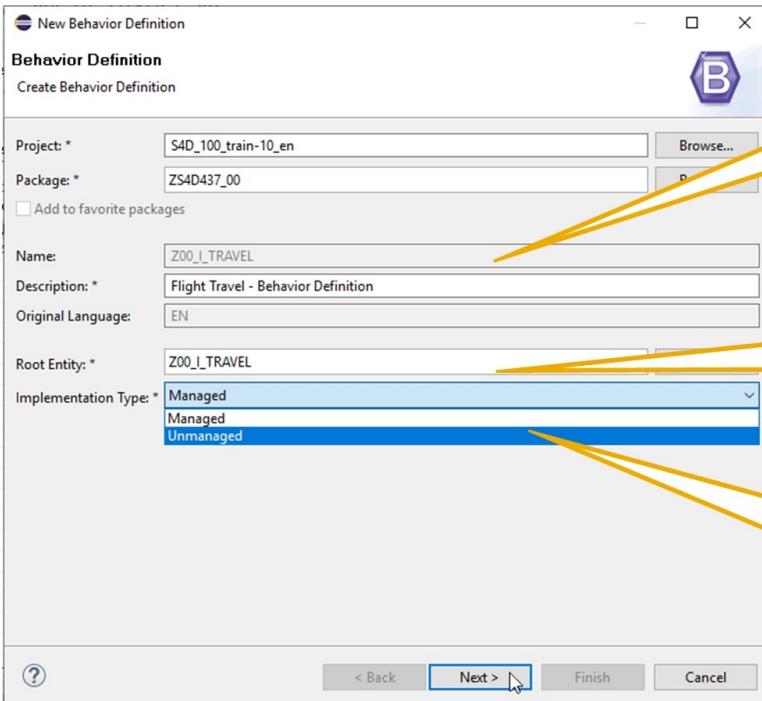


Figure 31: Create a New Behavior Definition (1)

You can create behavior definitions like any other repository object, that is, using the context menu in the project explorer, starting from the package or from its *Core Data Services* subnode.

However, it is easier to open the context menu on the data definition of the Root CDS View of the RAP Business Object. In this way, some of the properties of the behavior definition are preset by the development tools.



Always identical to name of CDS View

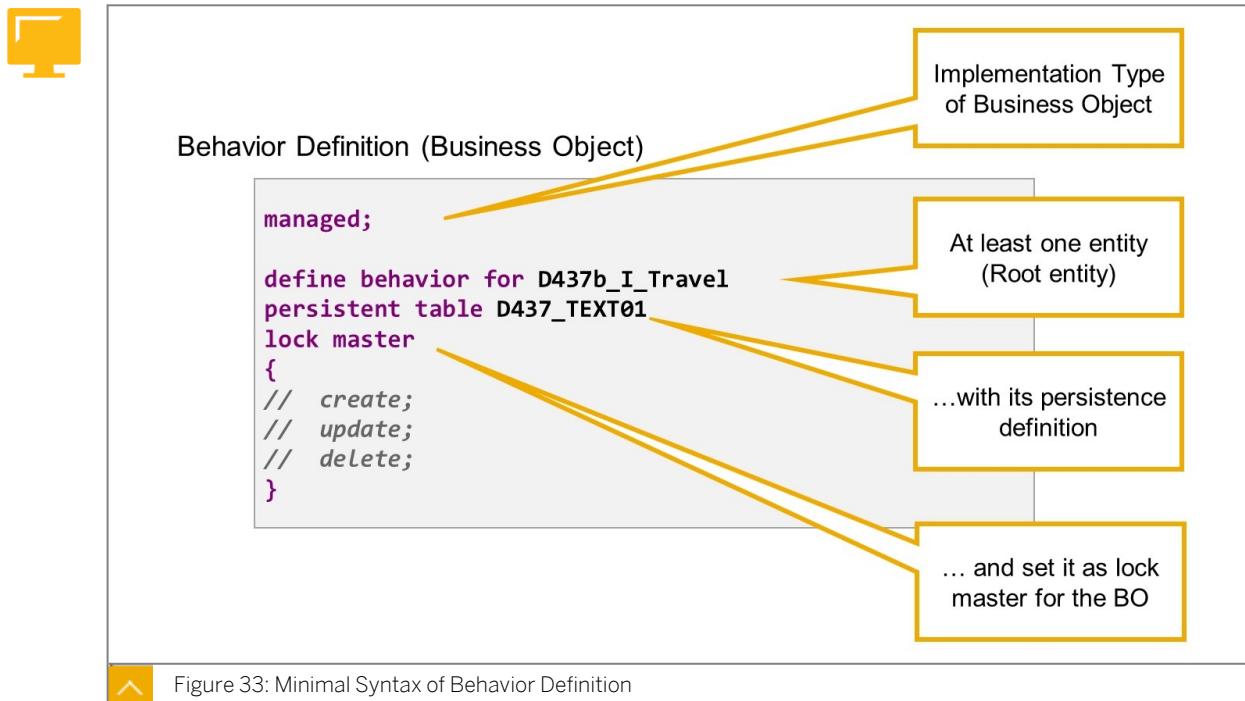
Name of CDS Root View

Managed
or
Unmanaged

Figure 32: Create a New Behavior Definition (2)

When creating a behavior definition, you cannot specify a name for the new repository object directly. Instead, the name of the behavior definition is derived from the name of the CDS View that is used to define the root entity of the business object. For this reason, you must specify the CDS root view at this early stage.

You must also specify the implementation type of the business object. The possible values depend on the nature of the related CDS view. For data definition views, you can choose between *Managed* and *Unmanaged*. In behavior definitions for projection views, only the *Projection* implementation type is supported.



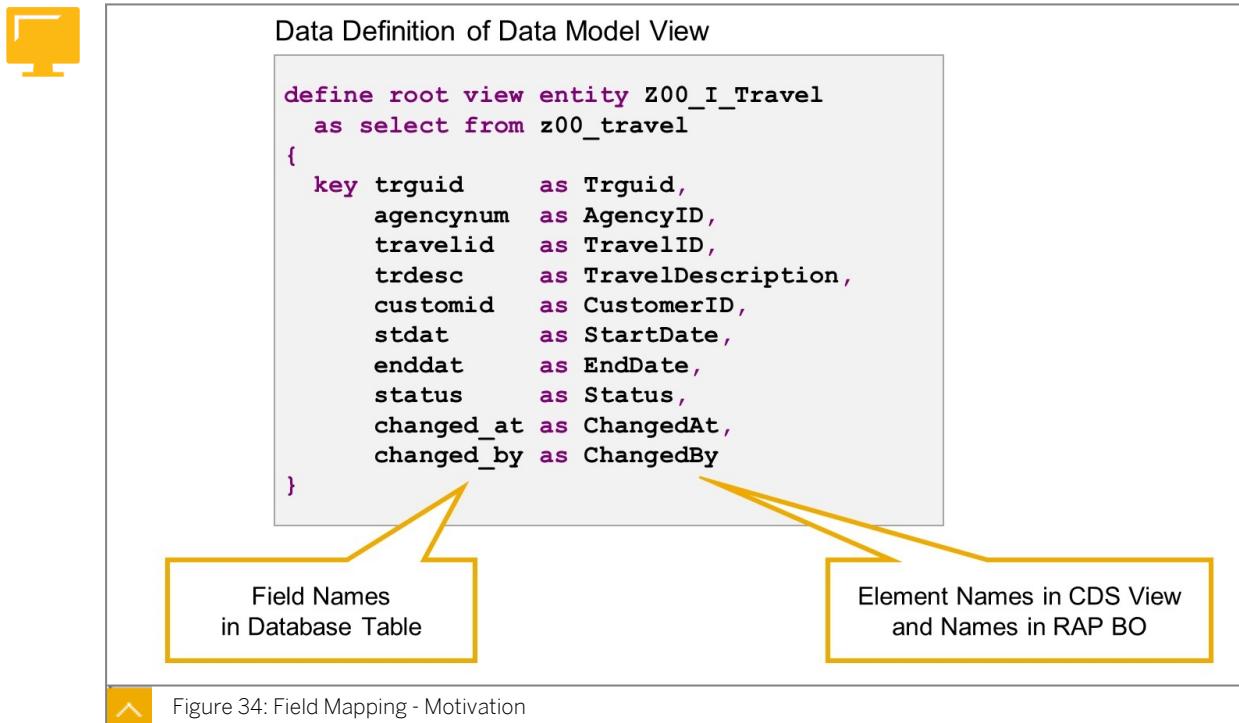
The minimal syntax of a behavior definition includes the implementation type of the business object and the behavior definition of at least one entity, namely the BO root entity.

If the implementation type is managed, an additional persistent table is required for each entity of the BO to allow write access to the related database table.

For the managed implementation type, it is also mandatory to set a lock type for each entity. The root entity has to be set as lock master.

The standard operations (create, update, delete) are part of the template for behavior definitions. They are not mandatory and can be commented or removed if not required.

Define the Field Mapping



When defining data models, developers often choose to introduce more readable element names in CDS, especially when, for example, the table is legacy and has cryptic short field names, maybe even based on German terminology (examples are BUKRS, KUNNR, and so on).

In the figure, Field Mapping - Motivation, the table field names `stdat` and `enddat` are replaced by the more readable alias names `StartDate` and `EndDate`.

Because the field names in the RAP business object are derived from the element names in the related CDS view, the field names in RAP no longer match the field names in the database table.

There is no way the RAP framework can determine the table field name in which to store a certain attribute.

To allow the persisting of such fields, the developer has to provide the information about which RAP BO field belongs to which table field.

This mapping between field names in the RAP BO and database table fields is defined in the behavior definition, using an additional mapping, followed by the name of the database table.



```

managed;

define behavior for ZOO_I_Travel
persistent table z00_travel
lock master
{
  create;
  update;
  delete;

mapping for z00_travel
{
  Trguid      = trguid;
  AgencyID    = agencynum;
  TravelID    = travelid;
  TravelDescription = trdesc;
  CustomerID  = customid;
  StartDate   = stdat;
  EndDate     = enddat;
  Status       = status;
  ChangedAt   = changed_at;
  ChangedBy   = changed_by;
}

```

Behavior for RAP BO root entity ZOO_I_Travel

Mapping for related DB table Z00_TRAVEL

Complete list of field names

Figure 35: Field Mapping - Complete List

In the most complete form, the mapping contains a full list of all field names, with the CDS element names on the left and the DB table field names on the right.



Note:

The assignment is also required for fields like Trguid and Status, where the names are the same, apart from differences in upper/lower case.



```

managed;

define behavior for ZOO_I_Travel
persistent table z00_travel
lock master
{
  create;
  update;
  delete;

  mapping for ZOO_travel corresponding
  {
    // Trguid          = trguid;
    AgencyID        = agencynum;
    // TravelID       = travelid;
    TravelDescription = trdesc;
    CustomerID      = customid;
    StartDate        = stdat;
    EndDate          = enddat;
    // Status          = status;
    ChangedAt        = changed_at;
    ChangedBy        = changed_by;
  }
}

```

Implicit mapping of fields with same name (apart from upper/lower case)

Explicit mapping only for fields with different names



Figure 36: Field Mapping - Addition Corresponding

By adding the corresponding keyword, the framework implicitly maps all fields for which the CDS element name and the database table name only differ in upper or lower case. Then, only the fields for which this is not the case need explicit mapping.



Note:

For the case where all field names are identical (apart from upper or lower case), the following short form exists: `mapping for <...> corresponding;`

Behavior Projection

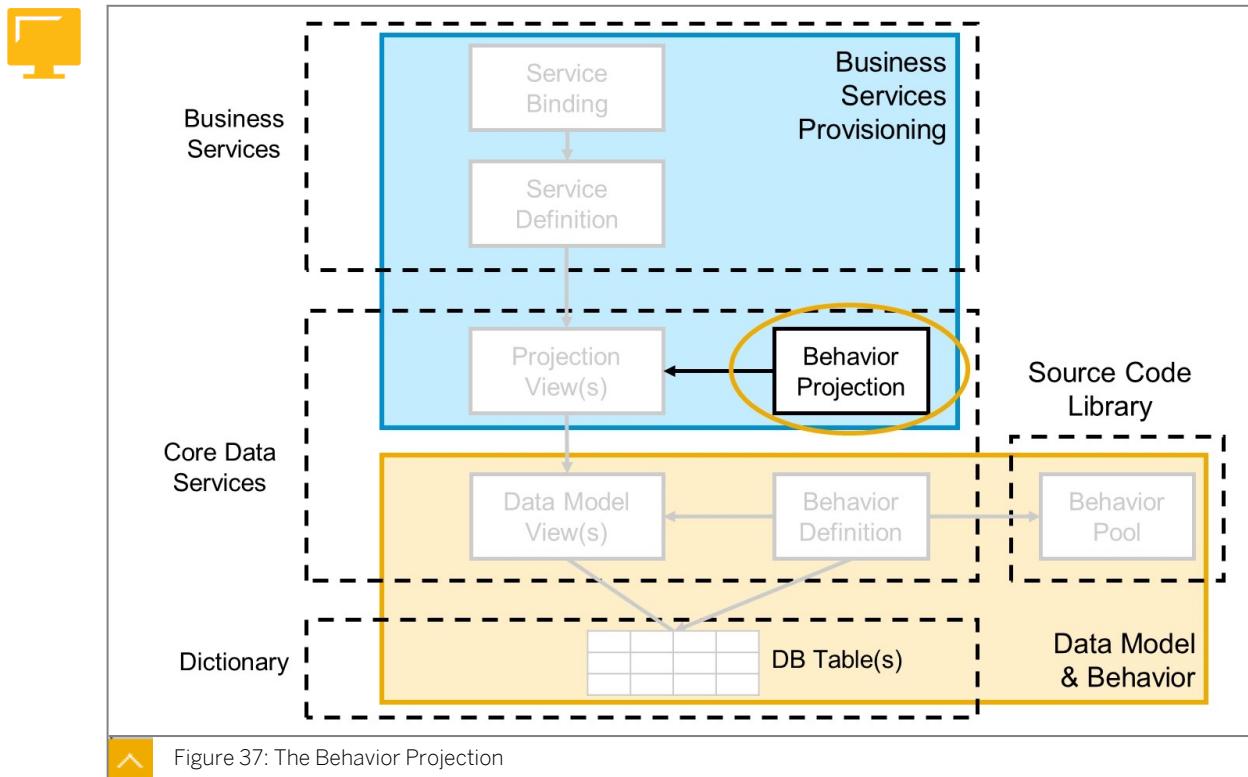


Figure 37: The Behavior Projection

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.

The projection behavior definition delegates to the underlying layer. The implementation of the individual characteristics is only done in the general BO.

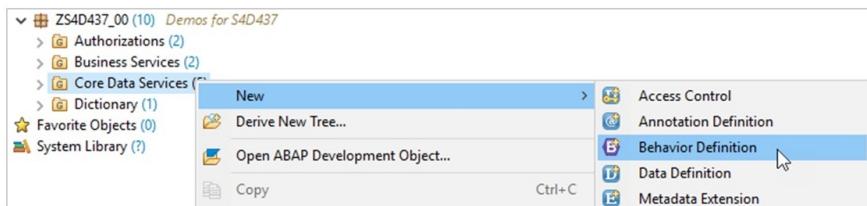


Note:

Although the behavior projection is built on top of the behavior definition of the business object, it does not refer to it directly. Instead, the behavior projection references the projection views, which are built on top of the data model views that are referenced by the behavior definition.



Alternative A: Start from Package



Alternative B: Start from CDS View (Data Definition)

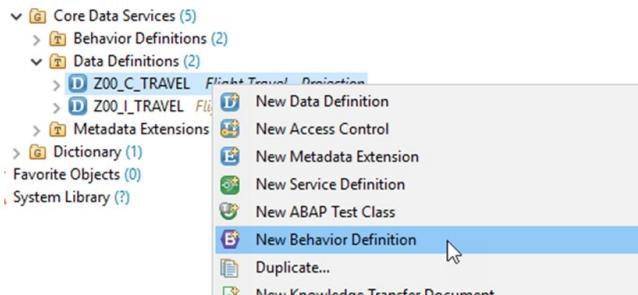


Figure 38: Create a New Behavior Projection (1)

The procedure to create a behavior projection is identical to the creation of a behavior definition. The only difference is that you create the repository object based on the projection view of a root entity rather than the data model view.

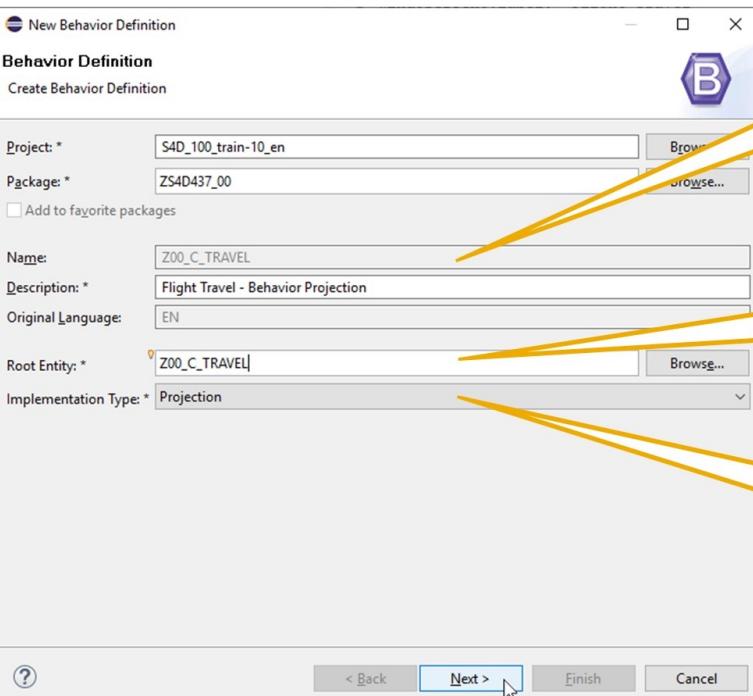


Figure 39: Create a New Behavior Definition (2)

When creating a behavior definition based on a CDS view that is of type projection view, the implementation is automatically set to *Projection* and no other value is supported.



Behavior Definition (Projection)

```
projection;  
  
define behavior for ZOO_C_Travel  
{  
    // use create;  
    // use update;  
    // use delete;  
}
```

Implementation Type
*Projection*Projection view
of root entity

Figure 40: Minimal Syntax of Behavior Projection

The minimal syntax of a behavior projection includes key word `projection` and the behavior definition for at least one projection view, that is the projection view of the BO root entity.

If the behavior definition of an entity contains standard operations (create, update, delete), the template for behavior projections adds the standard operations to the projection by default. They are not mandatory and can be commented out or removed if not required.

Unit 2

Exercise 3

Define an RAP Business Object and its Behavior

Business Scenario

In this exercise, you define a RAP Business Object (RAP BO) by extending your CDS-based data model with a CDS behavior definition. You then extend your data model projection with a CDS behavior projection, to make the behavior of the RAP BO visible to your OData UI Service.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 4: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Projection)	D437B_C_TRAVEL

Task 1: Create a CDS Behavior Definition

Create a CDS behavior definition (suggested name: Z##_I_TRAVEL) that defines the behavior of a RAP Business Object (RAP BO) with your data model view (Z##_I_Travel) as root entity. Define the root entity of your RAP Business Object as lock master and link the behavior definition to your database table.



Note:

Use the *New Behavior Definition* wizard from the context menu of the project explorer to create the behavior definition based on a template.

1. Create a CDS behavior definition for your Z##_I_Travel data model view. Assign the new repository object to your own package and use the same transport request as before.
2. Perform a syntax check for the new behavior definition.

Are there any syntax errors?

3. Edit the source code of the behavior definition. Uncomment the source code line `lock master`, which is mandatory for the behavior of root views with implementation type *Managed*.
4. Use the addition `persistent table` to link the behavior definition to your database table `Z##_TRAVEL`.
5. To enable persistence for all elements of your data model, define the mapping between table field names and CDS view element names.
6. Activate the CDS behavior definition.

Task 2: Create a CDS Behavior Projection

Create a CDS behavior definition (suggested name: `Z##_C_TRAVEL`) that defines the behavior of your projection view `Z##_C_Travel` and makes the behavior of the RAP Business Object visible for your OData UI Service.



Note:

Use the *New Behavior Definition* wizard in the context menu of the project explorer to create the behavior definition based on a template.

1. Create a CDS behavior definition for your projection view `Z##_C_Travel`. Assign it to your own package and use the same transport request as before. Make use of the context menu in the project explorer to create the behavior definition based on a template.
2. Activate the CDS behavior definition.
3. Refresh the preview of the service as SAP Fiori Elements app.

What is different on the *List Report Page*?

What is new on the *Object Page*?



Hint:

If you want, you can play around with the new functions a bit, but remember that the behavior definition is still incomplete and that the Business Object does not yet ensure data consistency. For example, the BO does not yet provide a unique value for key field `TRGUID` when you create a new travel. If you lose your data, go back to executable program `D437_TRAVEL_FILL`, where you can delete and regenerate the sample data.

Define an RAP Business Object and its Behavior

Business Scenario

In this exercise, you define a RAP Business Object (RAP BO) by extending your CDS-based data model with a CDS behavior definition. You then extend your data model projection with a CDS behavior projection, to make the behavior of the RAP BO visible to your OData UI Service.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 4: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Projection)	D437B_C_TRAVEL

Task 1: Create a CDS Behavior Definition

Create a CDS behavior definition (suggested name: Z##_I_TRAVEL) that defines the behavior of a RAP Business Object (RAP BO) with your data model view (Z##_I_Travel) as root entity. Define the root entity of your RAP Business Object as lock master and link the behavior definition to your database table.



Note:

Use the *New Behavior Definition* wizard from the context menu of the project explorer to create the behavior definition based on a template.

1. Create a CDS behavior definition for your *Z##_I_Travel* data model view. Assign the new repository object to your own package and use the same transport request as before.
 - a) In the *Project Explorer* view on the left, open the context menu for the data definition of your data model view.
 - b) From the context menu, choose *New Behavior Definition*.
 - c) Enter a description, confirm the suggested *Implementation Type (managed)* and choose *Next >*. (Note that you cannot change the suggested name of the new behavior definition).
 - d) Select the same transport request as before and choose *Finish*.

2. Perform a syntax check for the new behavior definition.
 - a) Press Ctrl + F2 and analyze the *Problems* tab below the source code editor.

Are there any syntax errors?

Yes, there are two syntax errors. One is about a missing lock master/lock dependent flag, the other about the missing persistence definition.

 3. Edit the source code of the behavior definition. Uncomment the source code line `lock master`, which is mandatory for the behavior of root views with implementation type *Managed*.
 - a) Remove the characters `//` in front of `lock master`.
 - b) See the source code extract from the model solution. - 4. Use the addition `persistent table` to link the behavior definition to your database table `Z##_TRAVEL`.
 - a) Remove characters `//` in front of `persistent table <????>` and replace `<????>` with the name of your database table.
 - b) See the source code extract from the model solution. - 5. To enable persistence for all elements of your data model, define the mapping between table field names and CDS view element names.
 - a) Within the curly brackets of the behavior definition, add keyword `mapping for` followed by the name of your database table, addition `corresponding`, and a pair of curly brackets.
 - b) Within the curly brackets, list all view elements and table fields that differ in more than just uppercase or lowercase.
 - c) See the source code extract from the model solution.
 - 6. Activate the CDS behavior definition.
 - a) Press Ctrl + F3.
 - b) Compare your source code to the following source code extract from the model solution:

```
managed; //implementation in class bp_d437b_i_travel unique;

define behavior for D437b_I_Travel //alias <alias_name>
persistent table d437b_travel
lock master

//authorization master ( instance )

//etag master <field_name>
{
    create;
    update;
    delete;

mapping for ZOO_TRAVEL corresponding
{
    AgencyID          = agencynum;
```

```

TravelDescription = trdesc;
CustomerID       = customid;
StartDate        = stdat;
EndDate          = enddat;
ChangedAt        = changed_at;
ChangedBy         = changed_by;
}
}

```

Task 2: Create a CDS Behavior Projection

Create a CDS behavior definition (suggested name: `Z##_C_TRAVEL`) that defines the behavior of your projection view `Z##_C_Travel` and makes the behavior of the RAP Business Object visible for your OData UI Service.



Note:

Use the *New Behavior Definition* wizard in the context menu of the project explorer to create the behavior definition based on a template.

1. Create a CDS behavior definition for your projection view `Z##_C_Travel`. Assign it to your own package and use the same transport request as before. Make use of the context menu in the project explorer to create the behavior definition based on a template.
 - a) In the *Project Explorer* view on the left, open the context menu for the data definition of your projection view.
 - b) From the context menu, choose *New Behavior Definition*.
 - c) Enter a description, confirm the *Implementation Type (Projection)* and choose *Next >*. (Note that you cannot change the suggested name of the new behavior definition).
 - d) Select the same transport request as before and choose *Finish*.
2. Activate the CDS behavior definition.
 - a) Press **Ctrl + F3**.
3. Refresh the preview of the service as *SAP Fiori Elements* app.

What is different on the *List Report Page*?

There are *New* and *Delete* buttons on the header toolbar of the list.

What is new on the *Object Page*?

There are *Edit* and *Delete* buttons in the right upper corner.



Hint:

If you want, you can play around with the new functions a bit, but remember that the behavior definition is still incomplete and that the Business Object does not yet ensure data consistency. For example, the BO does not yet provide a unique value for key field TRGUID when you create a new travel. If you lose your data, go back to executable program *D437_TRAVEL_FILL*, where you can delete and regenerate the sample data.

- a) If the browser window with the preview of the SAP Fiori elements app is still open, reload the page, otherwise open your service binding, choose the name of your projection view and choose *Preview*....
- b) Compare your source code to the following source code extract from the model solution:

```
projection;  
  
define behavior for D437b_C_Travel //alias <alias_name>  
//use etag  
{  
    use create;  
    use update;  
    use delete;  
}
```



LESSON SUMMARY

You should now be able to:

- Create a CDS behavior definition
- Create a CDS behavior projection

Using Entity Manipulation Language (EML) to Access RAP Business Objects



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe the purpose and syntax of EML
- Describe the derived data types for RAP Business Objects
- Use the Entity Manipulation Language (EML)

The EML Principle



- **Is a new set of ABAP statements**
 - Available in all ABAP programs
 - Special variants for use in behavior implementations
- **Provides Access to RAP BOs**
 - Read or modify BOs
 - Trigger persistent storage
 - Or reset changes
- **Uses special ABAP Types (Derived Data Types)**
 - Table types and structure types
 - Based on RAP BO definition



Figure 41: Entity Manipulation Language (EML)

Entity Manipulation Language (EML) is a part of the ABAP language that is used to control the business object's behavior in the context of ABAP.

The ABAP EML is a subset of ABAP for accessing RAP business objects (RAP BOs). EML statements allow the data content of a RAP BO (transactional buffer) to be read or modified and the persistent storage of modified data to be triggered.

ABAP EML can be used in all ABAP programs to consume RAP BOs. In particular, they can be used in the implementation of a RAP BO in a behavior implementation (ABAP behavior pool) itself. For the latter, there are some special EML variants.

The execution of an EML statement triggers processes in the RAP runtime framework that call the implementation of the RAP BOs. For unmanaged RAP BOs or unmanaged parts of managed RAP BOs, the implementation is part of an ABAP behavior pool. Otherwise, it is part of the RAP provider framework.

The operands of EML statements are mainly special data objects for passing data to and receiving results or messages from RAP BOs. These data objects are structures and internal

tables whose types are tailor-made for this purpose and derived from the RAP BO definition, namely the involved CDS views and behavior definitions.

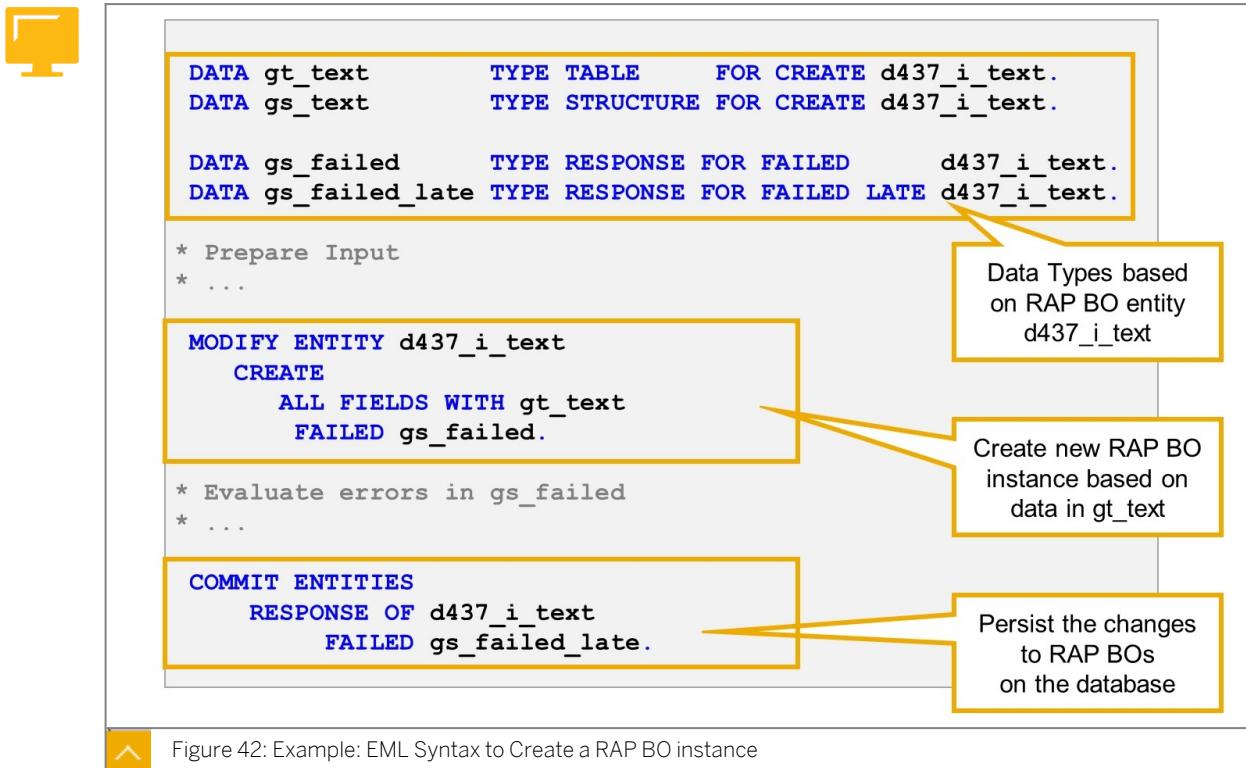


Figure 42: Example: EML Syntax to Create a RAP BO instance

The example for EML shows the creation of one or several new instances of RAP Business Object `d437_i_text`. This RAP BO consists of only one entity (root entity). The name of the root entity is also `d437_i_text`.

The actual creation of the new RAP BO instances takes place in EML statement `MODIFY ENTITY`, where `d437_i_text` specifies the (root) entity of the BO. The input for the create statement is provided by placing ABAP data object `gt_text` after addition `WITH`. (To simplify the example, we omitted the coding to fill internal table `gt_text`).

ABAP data object `gt_text` is a good example of a derived data type. The `DATA` declaration uses the new syntax variant `TYPE TABLE FOR`, followed by a keyword to specify the purpose of the data object and the name of the RAP BO entity. In our case, the keyword is `CREATE` and the name of the entity is `d437_i_text`. As a result, data object `gt_text` is an internal table which is tailor-made for a create access to RAP BO entity `d437_i_text`.

The declaration of data object `gs_text` uses `TYPE STRUCTURE FOR`. It is not meant to be used in an EML statement, but rather as a work area for internal table `gt_text`.

Data objects `gs_failed` and `gs_failed_late` are also declared with derived data types. They belong to a group of derived types called response types. Response types are always structures. They depend on the RAP BO entity but not on a specific operation (Create, Update, Delete, and so on).

After the EML statement `MODIFY`, the changes are not sent to the database directly. The persistence of the data changes is triggered by EML statement `COMMIT ENTITIES`. There is also a statement `ROLLBACK ENTITIES`, which can be used to undo changes that are persisted if there are errors.

**Note:**

Statements COMMIT ENTITIES and ROLLBACK ENTITIES are only needed outside RAP BOs. When using EML inside a RAP BO implementation, the RAP runtime framework takes care of this.

EML Commands



Operation	Syntax	Related Derived Data Types
Read	READ ENTITY <entity> <...> .	FOR READ IMPORT <entity> FOR READ RESULT <entity> FOR READ LINK <entity>
Create	MODIFY ENTITY <entity> CREATE <...> .	FOR CREATE <entity>
Update	MODIFY ENTITY <entity> UPDATE <...> .	FOR UPDATE <entity>
Delete	MODIFY ENTITY <entity> DELETE <...> .	FOR DELETE <entity>
Execute Action	MODIFY ENTITY <entity> EXECUTE <...> .	FOR ACTION IMPORT <ent>~<act> FOR ACTION RESULT <ent>~<act> FOR ACTION REQUEST <ent>~<act>

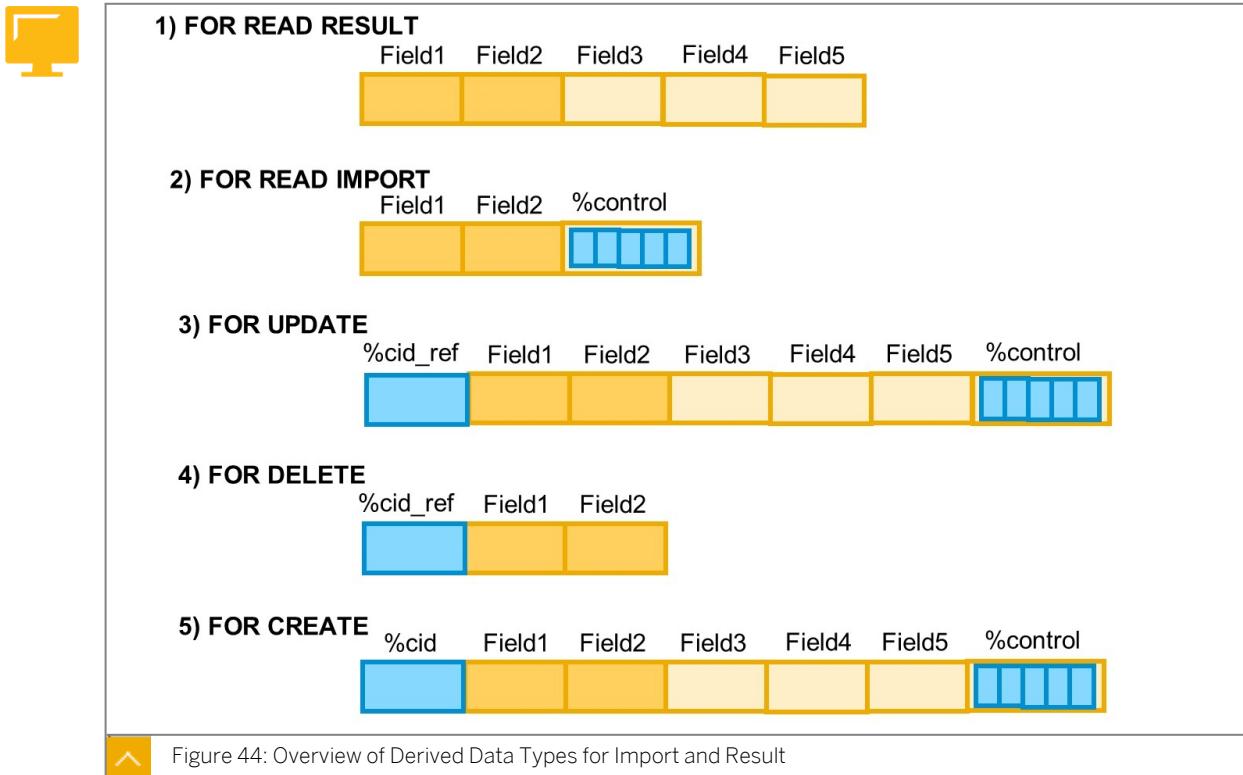
Figure 43: EML Commands - Overview

The figure, EML Commands - Overview, provides an overview of the most important EML commands, which are the basic operations Read, Create, Update, Delete, Execution Actions. Depending on the operation, the statement expects one or more internal tables as operands for input and output. The data types of these operands are derived data types that depend on the RAP BO entity and the individual operation.

Note the following:

- Only the read operation has its own statement, `READ ENTITY`. The other operations are variants of the `MODIFY ENTITY` statement.
- The variants of `MODIFY ENTITY` are distinguished by a keyword after the name of the entity. `READ ENTITY` does not have such a keyword.
- Operations `Read` and `Execute` have several input operands and an output operand (the result). Therefore these operations have more than one related derived types, distinguished by keywords `IMPORT`, `RESULT`, `LINK`, `REQUEST`.
- Operations `Read`, `Create`, `Update`, and `Delete`, only have one operand for input. No `IMPORT` keyword is needed.
- The derived types for actions identify the action via the name of the entity and the name of the action, separated by the tilde (~) character.

Derived Data Types



The structure types of derived data types depend on the RAP BO entity and the operations. They contain components of RAP BO entities, that is, persisted key and data field values that retain their original line type. However, derived types contain additional components that do not derive their type from the entity. They have special, tailor-made line types that provide additional information required in the context of transactional processing. The names of those additional components begin with % to avoid naming conflicts with components of the CDS entities.

Let us consider a RAP BO entity that consists of five fields, named field1, field2, and so on, of which the first two fields are key fields.

If we look at the derived data types for operation Read, we can see that the result contains all five fields, whereas the derived data type FOR READ IMPORT contains only the key fields. The same is true for the derived type for operation Delete.

On the other hand, the types for Read, Import, Update, and Create contain a generated substructure of generic name %control. This substructure has as many components as there are fields in the RAP BO entity. The names of the components are identical to the fields in the entity but their data type is ABAP_BEHV_FLAG. % is used in certain cases to specify which fields are requested or provided.



Note:

Data element ABAP_BEHV_FLAG serves as a Boolean type in RAP. Allowed values are available in structured constant MK of interface IF_ABAP_BEHV.



Note:
The technical type of data element ABAP_BEHV_FLAG is RAW(1) and not CHAR(1), as you might expect. Constants ABAP_TRUE, ABAP_FALSE or literals 'X' and '' are not compatible.

Components %cid and %cid_ref are needed for situations where the values for the key fields of newly created instances are not provided by the consumer, but calculated by the BO logic (internal numbering). In such a situation, the coding calling the Create operation, has to provide unique string values for %cid to identify the new instances. The framework returns a table with the mapping of %cid values and the calculated key values.

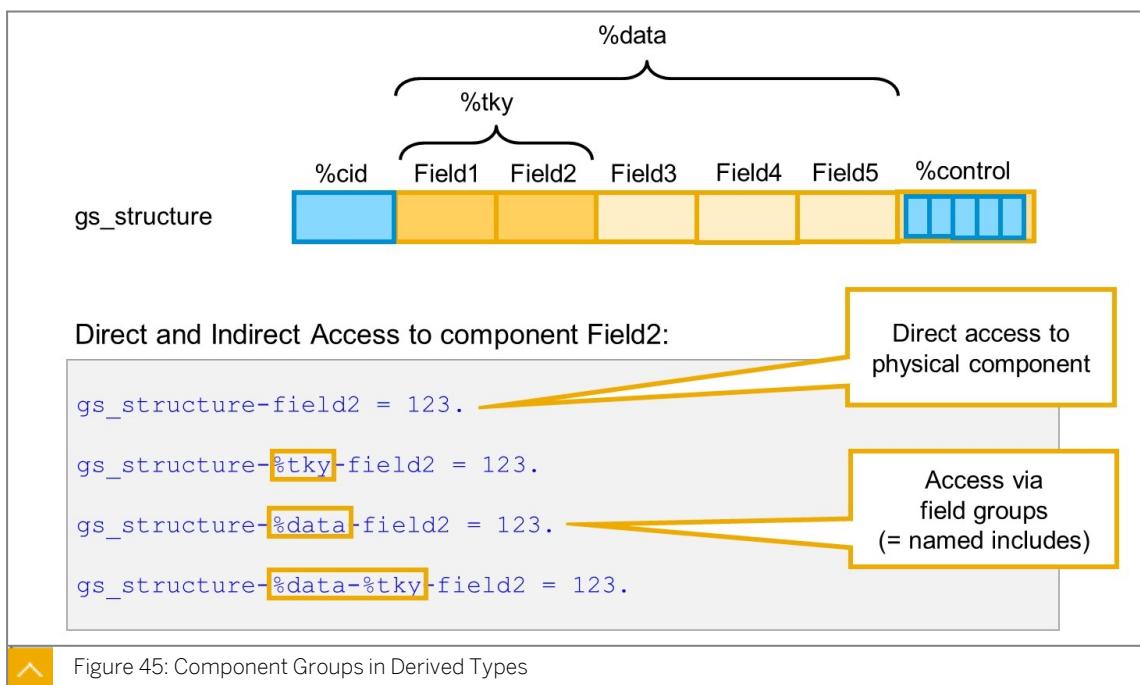


Figure 45: Component Groups in Derived Types

In addition to the physical components, derived types also contain component groups. They begin with % too and serve the purpose of summarizing groups of table columns under a single name. For example, %data summarizes all elements of the related entity (CDS view).

Technically, the component groups are named includes and the components can be addressed by the name of the include.

In the example in the figure, Component Groups in Derived Types, field2 is addressed directly as part of the named include %tky, and as part of the named include %data. Because %tky is part of %data, the field can even be addressed as component %tky of named include %data.



Note:
Named include %key is obsolete and should not be used anymore. It has been replaced with %tky. Although in non-draft scenarios, %key and %tky are identical, they differ in draft scenarios, where %tky contains an additional field %is_draft, by which the framework distinguishes draft and active version of an entity.

Response Operands



▪ FAILED

- Derived Type: `RESPONSE FOR FAILED <business object>`
- Returns entities, for which the operation failed

▪ REPORTED

- Derived Type: `RESPONSE FOR REPORTED <business object>`
- Returns messages, either related to entities or general

▪ MAPPED

- Derived Type: `RESPONSE FOR MAPPED <business object>`
- Returns lists of temporary keys and mapped final keys
- Relevant for CREATE with internal numbering



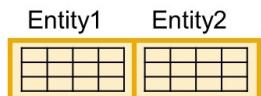
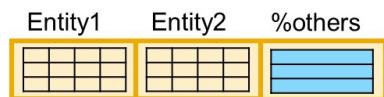
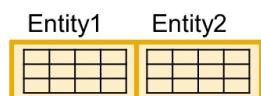
Figure 46: Response Operands

In addition to the input and result operands, EML statements use a set of response operands to provide feedback on the outcome of an operation. While the types of the operands for input and result depend on the entity and the individual operation, the type of the response operands only depends on the RAP business object, identified by the name of its root entity. Currently, there are three response operands, but not all are available for all EML statements.

Responses are imported by adding keywords FAILED, REPORTED, or MAPPED to the EML statement. These keywords have to be followed by a deep structure of the related derived data type for the RAP BO root entity. The import of responses is optional.

EML offers the following response operands:

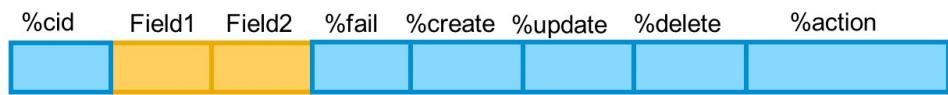
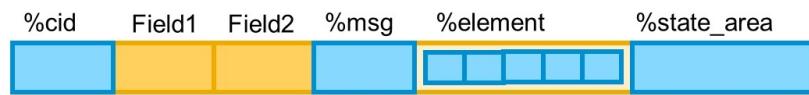
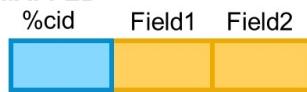
- FAILED is used for logging instances for which an operation has failed. The related derived type is `RESPONSE FOR FAILED`.
- REPORTED is used for returning messages. These messages are either related to a specific instance or static, that is, independent from a specific data set.
- MAPPED is used to map the calculated key values of created instances to the provided temporary IDs (component `%cid`). It is only relevant for CREATE operations.

**1) FOR RESPONSE FAILED****2) FOR RESPONSE REPORTED****3) FOR RESPONSE MAPPED**

↗ Figure 47: Overview of Derived Data Types for Response

The three response types are deep structures with a table-like component for each entity of the RAP BO.

If we consider a RAP BO with a root entity named Entity1 and a child entity Entity2, then the response types have two components named Entity1 and Entity2. Only the derived type for reported has an additional component named %others, which is also an internal table but with an elementary line type.

**1) FOR FAILED****2) FOR REPORTED****3) FOR MAPPED**

↗ Figure 48: Components of Response Operands

The line types of the table-like components are also derived types, namely the derived types STRUCTURE FOR FAILED <entity name>, STRUCTURE FOR REPORTED <entity name>, and STRUCTURE FOR MAPPED <entity name>.

**Note:**

Be aware of the difference between `RESPONSE FOR FAILED <root entity>` and `STRUCTURE FOR FAILED <entity>.` `RESPONSE FOR FAILED <root entity>` is based on the entire RAP BO, represented by its root entity and defines a deep structure for the entire RAP BO. `STRUCTURE FOR FAILED <entity>` is based on a single entity, root or child.

These types contain the key fields of the related entity plus some generic fields for the details. As usual, the additional fields start with % to avoid naming conflicts.

We will discuss some of these additional fields later in this course.

Short Form and Long Form



```
READ ENTITY d437_i_text
  ALL FIELDS WITH gt_read_import
  RESULT gt_read_result
  FAILED gs_failed.
```

Long form of EML statement READ ENTITY

```
READ ENTITIES OF d437_i_text
  ENTITY d437_i_text
  ALL FIELDS WITH gt_read_import
  RESULT gt_read_result
  FAILED gs_failed.
```

```
MODIFY ENTITY d437_i_text
  UPDATE
    FIELDS ( text ) WITH gt_text
    FAILED gs_failed
    REPORTED gs_reported.
```

Name of the RAP BO
(= Name of root entity)

```
MODIFY ENTITIES OF d437_i_text
  ENTITY d437_i_text
  UPDATE
    FIELDS ( text ) WITH gt_text
    RESULT gt_read_result
    FAILED gs_failed
    REPORTED gs_reported.
```



Figure 49: Short Form versus Long Form

Statements `READ ENTITY` and `MODIFY ENTITY` are short forms of their longer versions `READ ENTITIES OF` and `MODIFY ENTITIES OF`.

In the long version, the keyword `OF` is followed by the name of a RAP BO, which is identical to the name of its root entity. The affected entity is specified after keyword `ENTITY`. The rest of the statement is the same in long form and in short form.

The long form allows you to bundle several operations in one statement, either different operations on the same entity (for example, delete some instances and update some other), or operations on different entities of the same RAP BO (for example, create a root entity instance and related instances of a child entity in one call).

The short form is most suitable for RAP BOs, which consist of only one entity (the root entity).

Alias Names for Entities

Behavior Definition

```
managed;

define behavior for D437_I_TEXT alias text
persistent table d437_text
lock master
{
...
}
```

Alias text for entity name

EML Statement (long form without alias)

```
MODIFY ENTITIES OF d437_i_text
ENTITY d437_i_text
UPDATE
FIELDS ( text ) WITH gt_text
RESULT gt_read_result
FAILED gs_failed
REPORTED gs_reported.
```

EML Statement (long form, use of alias)

```
MODIFY ENTITIES OF d437_i_text
ENTITY text
UPDATE
FIELDS ( text ) WITH gt_text
RESULT gt_read_result
FAILED gs_failed
REPORTED gs_reported.
```

Figure 50: Aliases for RAP BO Entities

In behavior definition, the name of an entity is derived from the name of the related CDS view. In addition, you can provide an alias name for the entity. In the example, alias text is assigned to entity D437_I_TEXT.

In some positions, the technical name of an entity can be replaced with the alias name. This can help increasing readability and re-usability of code. In the example, the alias text is used in the long form of an EML statement to identify the entity.



Note:

The alias can only replace the entity name after keyword ENTITY. It cannot replace the name of the RAP BO after keyword OF. For the same reason, aliases are not available when using the short form of EML statements.



Note:

The ABAP compiler issues a warning, if an EML statement uses the technical name of an entity for which an alias exists.

Unit 2 Exercise 4

Read and Update an RAP Business Object

Business Scenario

In this exercise, you use the Entity Manipulation Language (EML) to read and update a RAP Business Object.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 5: Solution

Repository Object Type	Repository Object ID
Executable Program	D437B_EML_S1
Executable Program	D437B_EML_S1_OPT

Task 1: Copy Template Program

Create a copy of executable program *D437T_EML_T1* (suggested name: *Z##_EML_1*). Replace the default value of input parameter *PA_GUID* with a value from your database table *Z##_TRAVEL*.

1. Create a copy of executable program *D437T_EML_T1* (suggested name: *Z##_EML_1*), assign it to your own package and use the same transport request as before.
2. Edit the executable program. To simplify testing, replace the initial default value for parameter *PA_GUID* with a value that exists in the *TRGUID* column of your *Z##_TRAVEL* database table.
3. Activate the executable program.

Task 2: Read Data from a RAP Business Object

In your new program, use EML statement `READ ENTITY` to retrieve all fields of one root entity instance of your RAP BO (*Z##_I_TRAVEL*). Use the value of parameter *PA_GUID* as input to identify the travel that you want to read. Issue an error message in case the read access fails. Compare the current status of the travel with the content of parameter *PA_STATUS* and issue an error message if the travel already has the requested status.

1. Implement a `READ ENTITY` statement for your RAP BO (*Z##_I_TRAVEL*), read all fields of one root entity instance, import the `RESULT` and response `FAILED`.
2. Define an internal table (suggested name: *gt_read_import*), type it with the required derived type to use it as input for a read access to your RAP BO, and fill it with one row that contains the value of parameter *PA_GUID* in the key field.



Note:

Alternatively, you can use a `VALUE` expression as input for the `READ ENTITY` statement

3. Define an internal table (suggested name: `gt_read_result`), and type it with the required derived type to store the result of a read access to your RAP BO.



Note:

Alternatively, you can use an inline declaration after addition `RESULT` of the `READ ENTITY` statement.

4. Define a structure (suggested name: `gs_failed`), and type it with the required derived type to use store the response `FAILED` of a read access to your RAP BO.



Note:

Alternatively, you can use an inline declaration after addition `FAILED` of the `READ ENTITY` statement.

5. Evaluate the response. If `gs_failed` is not initial, issue error message 103 of message class `DEVS4D437`.
6. If `gs_failed` is initial, read the status of the travel from internal table `gt_read_result` and compare it to the value of parameter `pa_stat`. If they are the same, issue error message 110 of message class `DEVS4D437`.

Task 3: Update Data of a RAP Business Object Entity

If the read access was successful, use EML statement `MODIFY ENTITY` to update the status of the travel. Rollback your changes and issue an error message in case the update fails. Commit your changes if the update was successful.

1. Implement a `MODIFY ENTITY` statement for your RAP BO (`Z##_I_TRAVEL`) to `UPDATE` the `Status` field of the root entity instance and import response `FAILED`.
2. Define an internal table (suggested name: `gt_update_import`), type it with the required derived type to use it as input for an update access to your RAP BO, and fill it with one row that contains the value of parameter `pa_guid` in the key field and the value of parameter `pa_stat`.



Note:

Alternatively, you can use a `VALUE` expression as input for the `MODIFY ENTITY` statement.

3. Use the existing structure `gs_failed` for the response `FAILED` of the modify access to your RAP BO.
4. Evaluate the response. If `gs_failed` is not initial, execute statement `ROLLBACK ENTITY` and issue error message 102 of message class `DEVS4D437`.

5. If `gs_failed` is initial, execute statement `COMMIT ENTITY` and execute the `WRITE` statement that was already part of the template program.
6. Activate and test your program. Display the content of your DB table `Z##_TRAVEL` to confirm that the changed status is stored on the database.

Unit 2 Solution 4

Read and Update an RAP Business Object

Business Scenario

In this exercise, you use the Entity Manipulation Language (EML) to read and update a RAP Business Object.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 5: Solution

Repository Object Type	Repository Object ID
Executable Program	<i>D437B_EML_S1</i>
Executable Program	<i>D437B_EML_S1_OPT</i>

Task 1: Copy Template Program

Create a copy of executable program *D437T_EML_T1* (suggested name: *Z##_EML_1*). Replace the default value of input parameter *PA_GUID* with a value from your database table *Z##_TRAVEL*.

1. Create a copy of executable program *D437T_EML_T1* (suggested name: *Z##_EML_1*), assign it to your own package and use the same transport request as before.
 - a) Perform this step as you learned to do in previous classes.
2. Edit the executable program. To simplify testing, replace the initial default value for parameter *PA_GUID* with a value that exists in the TRGUID column of your *Z##_TRAVEL* database table.
 - a) To find an existing GUID, use the data preview for your table or any of your CDS views. Alternatively you can execute program *D437_TRAVEL_FILL*.
3. Activate the executable program.
 - a) Press Ctrl + F3.

Task 2: Read Data from a RAP Business Object

In your new program, use EML statement `READ ENTITY` to retrieve all fields of one root entity instance of your RAP BO (*Z##_I_TRAVEL*). Use the value of parameter *PA_GUID* as input to identify the travel that you want to read. Issue an error message in case the read access fails. Compare the current status of the travel with the content of parameter *PA_STATUS* and issue an error message if the travel already has the requested status.

1. Implement a `READ ENTITY` statement for your RAP BO (`Z##_I_TRAVEL`), read all fields of one root entity instance, import the `RESULT` and response `FAILED`.
 - a) See source code extract from the model solution.
2. Define an internal table (suggested name: `gt_read_import`), type it with the required derived type to use it as input for a read access to your RAP BO, and fill it with one row that contains the value of parameter `PA_GUID` in the key field.

**Note:**

Alternatively, you can use a `VALUE` expression as input for the `READ ENTITY` statement

- a) See source code extract from the model solution.
3. Define an internal table (suggested name: `gt_read_result`), and type it with the required derived type to store the result of a read access to your RAP BO.

**Note:**

Alternatively, you can use an inline declaration after addition `RESULT` of the `READ ENTITY` statement.

- a) See the source code extract from the model solution.
4. Define a structure (suggested name: `gs_failed`), and type it with the required derived type to use store the response `FAILED` of a read access to your RAP BO.

**Note:**

Alternatively, you can use an inline declaration after addition `FAILED` of the `READ ENTITY` statement.

- a) See the source code extract from the model solution.
5. Evaluate the response. If `gs_failed` is not initial, issue error message 103 of message class `DEVS4D437`.
 - a) See source code extract from the model solution.
6. If `gs_failed` is initial, read the `status` of the travel from internal table `gt_read_result` and compare it to the value of `pa_stat`. If they are the same, issue error message 110 of message class `DEVS4D437`.
 - a) See the source code extract from the model solution.

Task 3: Update Data of a RAP Business Object Entity

If the read access was successful, use EML statement `MODIFY ENTITY` to update the status of the travel. Rollback your changes and issue an error message in case the update fails. Commit your changes if the update was successful.

1. Implement a `MODIFY ENTITY` statement for your RAP BO (`Z##_I_TRAVEL`) to `UPDATE` the `Status` field of the root entity instance and import response `FAILED`.
 - a) See the source code extract from the model solution.

2. Define an internal table (suggested name: `gt_update_import`), type it with the required derived type to use it as input for an update access to your RAP BO, and fill it with one row that contains the value of parameter `pa_guid` in the key field and the value of parameter `pa_stat`.

**Note:**

Alternatively, you can use a `VALUE` expression as input for the `MODIFY ENTITY` statement.

- a) See the source code extract from the model solution.
3. Use the existing structure `gs_failed` for the response `FAILED` of the modify access to your RAP BO.
- a) See the source code extract from the model solution.
4. Evaluate the response. If `gs_failed` is not initial, execute statement `ROLLBACK ENTITY` and issue error message 102 of message class `DEVS4D437`.
- a) See the source code extract from the model solution.
5. If `gs_failed` is initial, execute statement `COMMIT ENTITY` and execute the `WRITE` statement that was already part of the template program.
- a) See source code extract from the model solution.
6. Activate and test your program. Display the content of your DB table `Z##_TRAVEL` to confirm that the changed status is stored on the database.
- a) Press `Ctrl + F3`.
 - b) Press `F8`.
 - c) Compare your code to the following extract from the model solution:

Executable program **D437B_EML_S1**:

```

REPORT d437b_eml_s1 MESSAGE-ID devs4d437.

PARAMETERS pa_guid TYPE s_trguid
        DEFAULT '00000000000000000000000000000000'.
PARAMETERS pa_stat TYPE s_status VALUE CHECK.

* Data declarations for read access

DATA gt_read_import      TYPE TABLE      FOR READ IMPORT d437b_i_travel.
DATA gs_read_import      TYPE STRUCTURE FOR READ IMPORT d437b_i_travel.

DATA gt_read_result      TYPE TABLE      FOR READ RESULT d437b_i_travel.
DATA gs_read_result      TYPE STRUCTURE FOR READ RESULT d437b_i_travel.

* Data declarations for update access

DATA gt_update_import    TYPE TABLE      FOR UPDATE      d437b_i_travel.
DATA gs_update_import    TYPE STRUCTURE FOR UPDATE      d437b_i_travel.

* Data declarations for response

DATA gs_failed           TYPE RESPONSE   FOR FAILED      d437b_i_travel.
DATA gs_reported          TYPE RESPONSE   FOR REPORTED   d437b_i_travel.

```

```

START-OF-SELECTION.

* Read the RAP BO entity to check for current status
*-----*
CLEAR gt_read_import.

gs_read_import-%tky-trguid = pa_guid. "Recommended
"gs_read_import-TRguid = pa_guid.      " Alternative

APPEND gs_read_import TO gt_read_import.

READ ENTITY d437b_i_travel
ALL FIELDS WITH gt_read_import
RESULT   gt_read_result
FAILED   gs_failed.

IF gs_failed IS NOT INITIAL.

MESSAGE e103 WITH pa_guid.

ELSE.

READ TABLE gt_read_result INTO gs_read_result INDEX 1.

IF gs_read_result-status = pa_stat.
  MESSAGE e110 WITH pa_guid pa_stat.
ENDIF.

ENDIF.

* Update RAP BO with new status
*-----*

CLEAR gt_update_import.

gs_update_import-%tky-trguid = pa_guid.
gs_update_import-status = pa_stat.

APPEND gs_update_import TO gt_update_import.

MODIFY ENTITY d437b_i_travel
  UPDATE FIELDS(`status`) WITH gt_update_import
  FAILED gs_failed.

IF gs_failed IS NOT INITIAL.
  ROLLBACK ENTITIES.
  MESSAGE e102 WITH pa_guid.
ELSE.

  COMMIT ENTITIES.

  WRITE: / 'Status of instance', pa_guid,
         'successfully set to', pa_stat.
ENDIF.

```

Executable Program D437B_EML_S1_OPT:

(Alternative solution using modern, expression-based syntax to provide input and retrieve results.)

```
REPORT d437b_eml_s1_opt MESSAGE-ID devs4d437.
```

```

PARAMETERS pa_guid TYPE s_trguid
          DEFAULT '00000000000000000000000000000000'.
PARAMETERS pa_stat TYPE s_status VALUE CHECK.

* Data declarations for read access

* Data declarations for update access

* Data declarations for response

START-OF-SELECTION.

* Read the RAP BO entity to check for current status
*-----*

READ ENTITY d437b_i_travel
ALL FIELDS WITH VALUE #( ( %tky-trguid = pa_guid ) )
RESULT   data(gt_read_result)
FAILED   DATA(gs_failed).

IF gs_failed IS NOT INITIAL.

  MESSAGE e103 WITH pa_guid.

ELSE.

  IF gt_read_result[ 1 ]-status = pa_stat.
    MESSAGE e110 WITH pa_guid pa_stat.
  ENDIF.

ENDIF.

* Update RAP BO with new status
*-----*

MODIFY ENTITY d437b_i_travel
  UPDATE FIELDS(`status`)
    WITH VALUE #( ( %tky-trguid = pa_guid
                  status      = pa_stat ) )
  FAILED gs_failed.

IF gs_failed IS NOT INITIAL.
  MESSAGE e102 WITH pa_guid.
ELSE.

  COMMIT ENTITIES.

  WRITE: / 'Status of instance', pa_guid,
         'successfully set to', pa_stat.
ENDIF.

```

Unit 2

Exercise 5

Optional: Use Long Version of EML Statements and Entity Aliases

Business Scenario

In this exercise, you use the long version of EML statements to read and update a RAP Business Object. To support readability of the statements you provide an alias for the root entity of your RAP BO.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Repository Object Type	Repository Object ID
Executable Program	D437B_EML_S2
Executable Program	D437B_EML_S2_OPT
CDS Behavior Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Projection)	D437B_C_TRAVEL

Task 1: Copy Template Program

Create a copy of your executable program `Z##_EML_1` (suggested name: `Z##_EML_2`). Replace the default value of input parameter `PA_GUID` with a value from your database table `Z##_TRAVEL`.



Note:

Alternatively, you can copy executable program `D437B_EML_S1`.

1. Create a copy of executable program `Z##_EML_1` or executable program `D437B_EML_S1` (suggested name: `Z##_EML_1`), assign it to your own package and use the same transport request as before.
2. Edit the executable program. To simplify testing, replace the initial default value for parameter `PA_GUID` with a value that exists in the `TRGUID` column of your `Z##_TRAVEL` database table.



Note:

If you copied your own executable program, you can skip this step.

3. Activate the executable program.

Task 2: Define Alias for Root Entity

Edit the behavior definition and the behavior projection of your RAP Business Object (`Z##_I_TRAVEL`) and define an alias for the root entity `Z##_I_TRAVEL` (suggested alias name: `Travel`).

1. Open the source code of your RAP BO (`Z##_I_TRAVEL`).
2. In the `define behavior for` statement, after the name of the data model view, add `alias` followed by the alias name.
3. Activate the behavior definition.
4. Edit the behavior projection of your RAP BO (`Z##_I_TRAVEL`) and provide the same alias for the root entity.

Task 3: Use Long Syntax of EML

In your new program (`Z##_EML_2`), replace EML statements `READ ENTITY` and `MODIFY ENTITY` with the respective long versions (`READ ENTITIES` and `MODIFY ENTITIES`). Use the alias you defined in the previous task to address the root entity of your RAP BO.

1. In your new program, find EML statement `READ ENTITY` and replace it with EML statement `READ ENTITIES`.
2. After addition `ENTITY` use the alias of the root entity (`Travel`) rather than its technical name (`Z##_I_Travel`).
3. Do the same with the `MODIFY ENTITY` statement.
4. Evaluate the response. If `gs_failed` is not initial, issue error message 103 of message class `DEVS4D437`.

Unit 2 Solution 5

Optional: Use Long Version of EML Statements and Entity Aliases

Business Scenario

In this exercise, you use the long version of EML statements to read and update a RAP Business Object. To support readability of the statements you provide an alias for the root entity of your RAP BO.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Repository Object Type	Repository Object ID
Executable Program	D437B_EML_S2
Executable Program	D437B_EML_S2_OPT
CDS Behavior Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Projection)	D437B_C_TRAVEL

Task 1: Copy Template Program

Create a copy of your executable program Z##_EML_1 (suggested name: Z##_EML_2). Replace the default value of input parameter PA_GUID with a value from your database table Z##_TRAVEL.



Note:

Alternatively, you can copy executable program D437B_EML_S1.

1. Create a copy of executable program Z##_EML_1 or executable program D437B_EML_S1 (suggested name: Z##_EML_1), assign it to your own package and use the same transport request as before.
 - a) Perform this step as you learned to do in previous classes.
2. Edit the executable program. To simplify testing, replace the initial default value for parameter PA_GUID with a value that exists in the TRGUID column of your Z##_TRAVEL database table.



Note:
If you copied your own executable program, you can skip this step.

- a) To find an existing GUID, use the data preview for your table or any of your CDS views.
Alternatively, you can execute program *D437_TRAVEL_FILL*.
3. Activate the executable program.
 - a) Press Ctrl + F3.

Task 2: Define Alias for Root Entity

Edit the behavior definition and the behavior projection of your RAP Business Object (*Z##_I_TRAVEL*) and define an alias for the root entity *Z##_I_TRAVEL* (suggested alias name: *Travel*).

1. Open the source code of your RAP BO (*Z##_I_TRAVEL*).
 - a) Perform this step as before.
2. In the `define behavior for` statement, after the name of the data model view, add `alias` followed by the alias name.
 - a) See the source code extract from the model solution.
3. Activate the behavior definition.
 - a) Press Ctrl + F3.
4. Edit the behavior projection of your RAP BO (*Z##_I_TRAVEL*) and provide the same alias for the root entity.
 - a) See the source code extract from the model solution.
 - b) Compare your source code to the following source code extract from the model solution:

Behavior Definition for **D437B_EML_S1**:

```
managed; //implementation in class bp_d437b_i_travel unique;

define behavior for D437b_I_Travel alias Travel
persistent table d437b_travel
lock master

//authorization master ( instance )

//etag master <field_name>
{
  ...
}
```

Behavior Definition **D437B_C_TRAVEL**

```
projection;

define behavior for D437b_C_Travel alias Travel
//use etag
{
  use create;
```

```

use update;
use delete;
}

```

Task 3: Use Long Syntax of EML

In your new program (Z##_EML_2), replace EML statements `READ ENTITY` and `MODIFY ENTITY` with the respective long versions (`READ ENTITIES` and `MODIFY ENTITIES`). Use the alias you defined in the previous task to address the root entity of your RAP BO.

1. In your new program, find EML statement `READ ENTITY` and replace it with EML statement `READ ENTITIES`.
 - a) See source code extract from the model solution.
2. After addition `ENTITY` use the alias of the root entity (*Travel*) rather than its technical name (`Z##_I_Travel`).
 - a) See source code extract from the model solution.
3. Do the same with the `MODIFY ENTITY` statement.
 - a) See source code extract from the model solution.
4. Evaluate the response. If `gs_failed` is not initial, issue error message 103 of message class DEVS4D437.
 - a) See the source code extract from the model solution.
 - b) Compare your code to the following extract from the model solution:

Executable program D437B_EML_S2:

```

REPORT d437b_eml_s2 MESSAGE-ID devs4d437.

PARAMETERS pa_guid TYPE s_trguid
          DEFAULT '00000000000000000000000000000000'.
PARAMETERS pa_stat TYPE s_status VALUE CHECK.

* Data declarations for read access

DATA gt_read_import      TYPE TABLE      FOR READ IMPORT d437b_i_travel.
DATA gs_read_import       TYPE STRUCTURE FOR READ IMPORT d437b_i_travel.

DATA gt_read_result      TYPE TABLE      FOR READ RESULT d437b_i_travel.
DATA gs_read_result       TYPE STRUCTURE FOR READ RESULT d437b_i_travel.

* Data declarations for update access

DATA gt_update_import    TYPE TABLE      FOR UPDATE      d437b_i_travel.
DATA gs_update_import     TYPE STRUCTURE FOR UPDATE      d437b_i_travel.

* Data declarations for response

DATA gs_failed            TYPE RESPONSE   FOR FAILED      d437b_i_travel.
DATA gs_reported           TYPE RESPONSE   FOR REPORTED    d437b_i_travel.

START-OF-SELECTION.

* Read the RAP BO entity to check for current status
*-----*
CLEAR gt_read_import.

```

```

gs_read_import-%tky-trgguid = pa_guid. "Recommended
"gs_read_import-trgguid      = pa_guid. "Alternative

APPEND gs_read_import TO gt_read_import.

*  READ ENTITY d437b_i_travel
*  ALL FIELDS WITH gt_read_import
*  RESULT    gt_read_result
*  FAILED    gs_failed.

READ ENTITIES OF d437b_i_travel
  ENTITY Travel
  ALL FIELDS WITH gt_read_import
  RESULT    gt_read_result
  FAILED    gs_failed.

IF gs_failed IS NOT INITIAL.

  MESSAGE e103 WITH pa_guid.

ELSE.

  READ TABLE gt_read_result INTO gs_read_result INDEX 1.

  IF gs_read_result-status = pa_stat.
    MESSAGE e110 WITH pa_guid pa_stat.
  ENDIF.

ENDIF.

* Update RAP BO with new status
*-----*
CLEAR gt_update_import.

gs_update_import-%tky-trgguid = pa_guid.
gs_update_import-status = pa_stat.

APPEND gs_update_import TO gt_update_import.

*  MODIFY ENTITY d437b_i_travel
*      UPDATE FIELDS (status) WITH gt_update_import
*      FAILED gs_failed.

MODIFY ENTITIES of D437b_I_Travel
  ENTITY Travel
  UPDATE FIELDS (status) WITH gt_update_import
  FAILED gs_failed.

IF gs_failed IS NOT INITIAL.
  ROLLBACK ENTITIES.
  MESSAGE e102 WITH pa_guid.
ELSE.

  COMMIT ENTITIES.

  WRITE: / 'Status of instance', pa_guid,
         'successfully set to', pa_stat.
ENDIF.

```

Executable Program D437B_EML_S2_OPT

(Alternative solution using modern, expression-based syntax to provide input and retrieve results.)

```

REPORT d437b_eml_s2_opt MESSAGE-ID devs4d437.

PARAMETERS pa_guid TYPE s_trguid
          DEFAULT '00000000000000000000000000000000'.
PARAMETERS pa_stat TYPE s_status VALUE CHECK.

* Data declarations for read access
* Data declarations for update access
* Data declarations for response

START-OF-SELECTION.

* Read the RAP BO entity to check for current status
*-----*
*  READ ENTITY d437b_i_travel
*    ALL FIELDS WITH VALUE #( ( %tky-trguid = pa_guid ) )
*    RESULT   DATA(gt_read_result)
*    FAILED   DATA(gs_failed).

READ ENTITIES OF d437b_i_travel
  ENTITY travel
    ALL FIELDS WITH VALUE #( ( %tky-trguid = pa_guid ) )
    RESULT   DATA(gt_read_result)
    FAILED   DATA(gs_failed).

IF gs_failed IS NOT INITIAL.

  MESSAGE e103 WITH pa_guid.

ELSE.

  IF gt_read_result[ 1 ]-status = pa_stat.
    MESSAGE e110 WITH pa_guid pa_stat.
  ENDIF.

ENDIF.

* Update RAP BO with new status
*-----*
*  MODIFY ENTITY d437b_i_travel
*    UPDATE FIELDS ( status )
*    WITH VALUE #( ( %tky-trguid = pa_guid
*                  status      = pa_stat ) )
*    FAILED gs_failed.

MODIFY ENTITIES OF d437b_i_travel
  ENTITY travel
    UPDATE FIELDS ( status )
    WITH VALUE #( ( %tky-trguid = pa_guid
                  status      = pa_stat ) )
    FAILED gs_failed.

IF gs_failed IS NOT INITIAL.

```

```
MESSAGE e102 WITH pa_guid.  
ELSE.  
    COMMIT ENTITIES.  
  
    WRITE: / 'Status of instance', pa_guid,  
          'successfully set to', pa_stat.  
ENDIF.
```



LESSON SUMMARY

You should now be able to:

- Describe the purpose and syntax of EML
- Describe the derived data types for RAP Business Objects
- Use the Entity Manipulation Language (EML)

Unit 2

Lesson 3

Understanding Concurrency Control in RAP



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe pessimistic concurrency control (locking)
- Enable optimistic concurrency control

Concurrency Control Concepts



▪ Pessimistic Concurrency Control

- Prevent simultaneous modifications
- Based on exclusive locks (classic enqueue technique)
- Tied to the ABAP session
- Handled by RAP BO framework (in managed scenarios)

▪ Optimistic Concurrency Control

- Allow concurrent control but avoid inconsistencies
- Based on ETag field (a field that is updated in every write access)
- Only relevant if RAP BO is consumed via Odata
- Independent of ABAP session



Figure 51: Concurrency Control in RAP

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. If more than one user has transactional database access, you must make sure that every user only executes changes based on the current state of the data so the data stays consistent. In addition, you must make sure 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.

Pessimistic Concurrency Control

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 by using a global lock table.

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.

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. In unmanaged scenarios, the application developer has to implement the method for lock and implement the locking mechanism. This will be covered later in this course.

Optimistic Concurrency Control

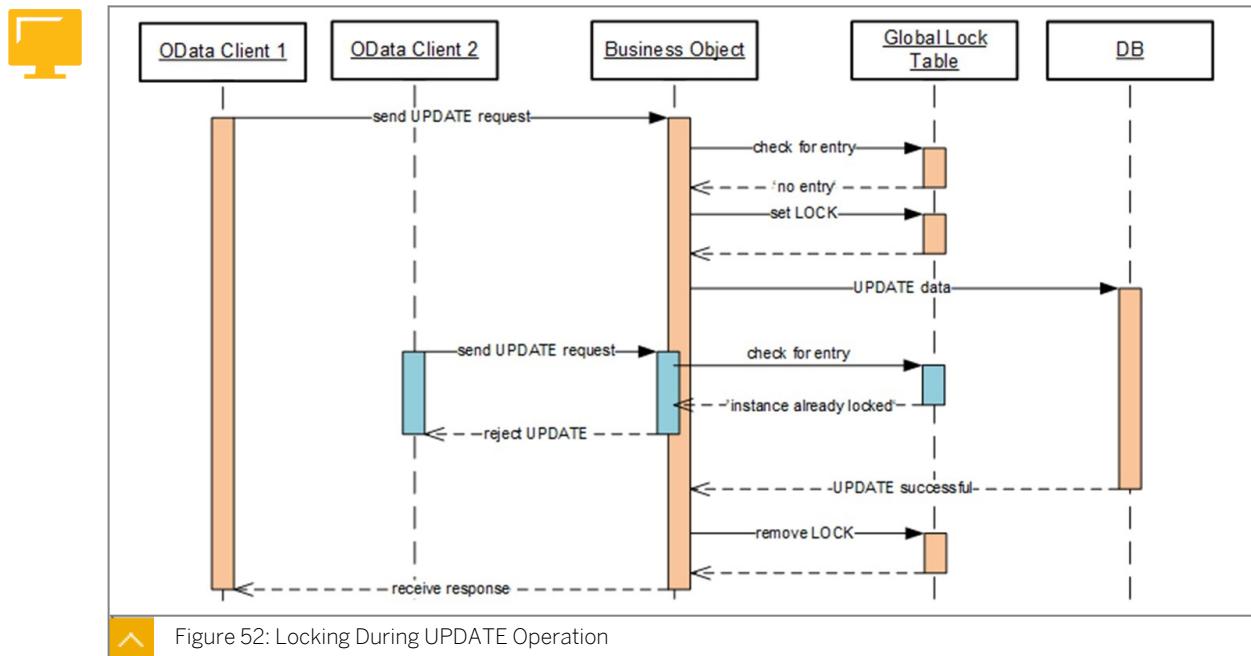
Optimistic concurrency control enables transactional access to data by multiple users at the same time, while avoiding inconsistencies and unintentional changes of already modified data.

The approach of optimistically controlling data relies on the concept that every change on a data set is logged by a specified field, called the 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.

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

Optimistic concurrency control is only relevant when consuming business objects via OData. That is why the ETag is also referred to as OData ETag.

Pessimistic Concurrency Control



Technically, pessimistic concurrency control 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 example illustrates how the lock is set on the global lock table during an UPDATE operation. 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.



Hint:

You can use transaction SM12 to analyze current enqueue locks.

Optimistic Concurrency Control

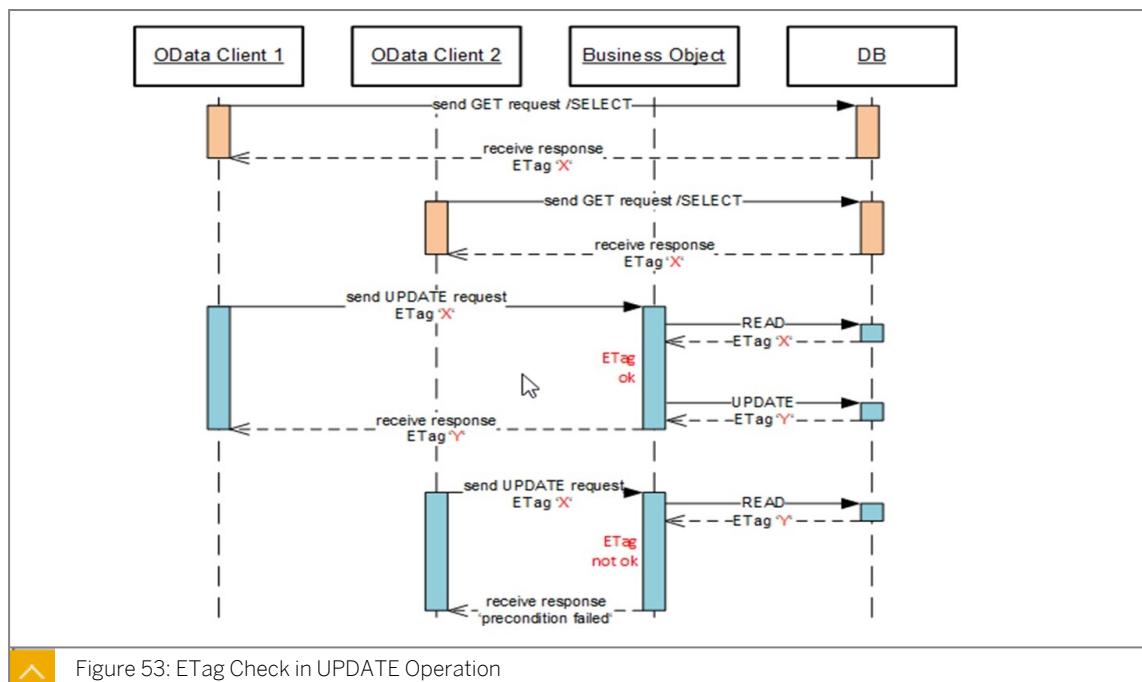


Figure 53: ETag Check in UPDATE Operation

When optimistic concurrency control is enabled for a RAP business object, the OData client reads the current *ETag* value with every read request and sends this value back with every modifying operation.

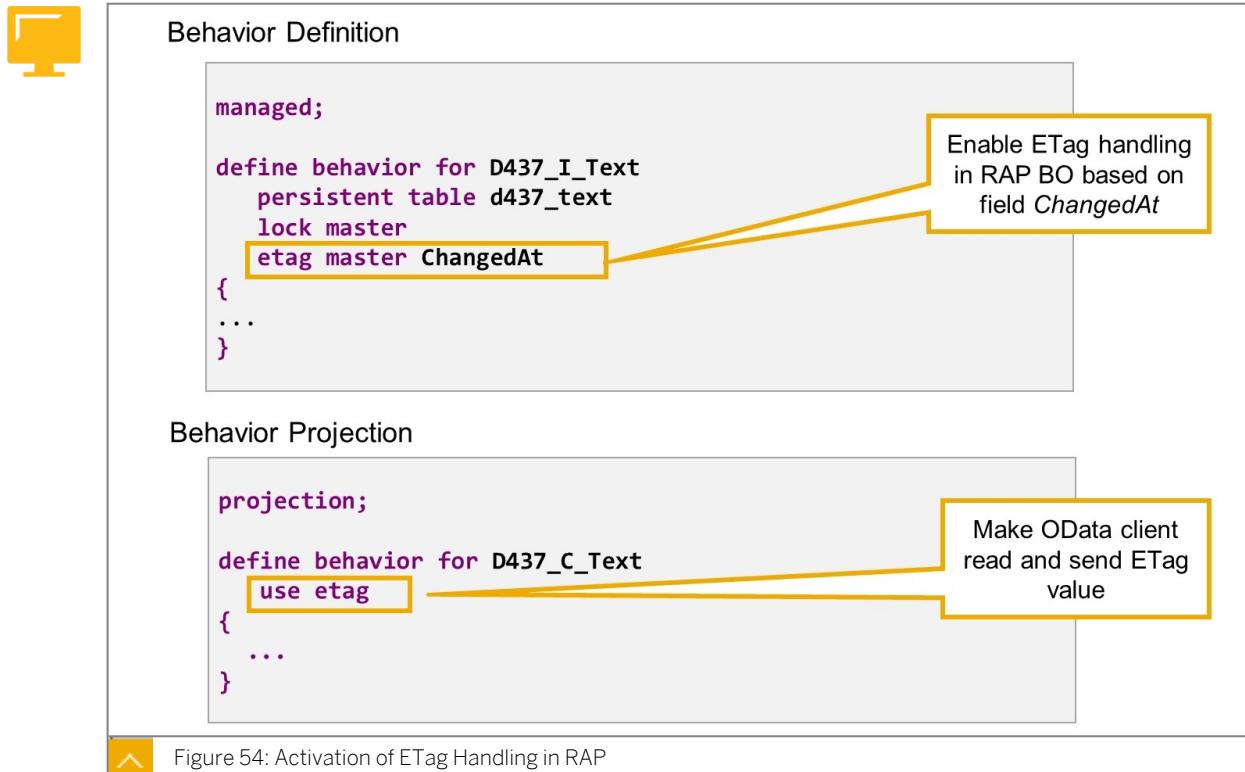
On each ETag relevant operation, the value the client sends with the request is compared to the current value of the *ETag* field on the database. If these values match, the change request is accepted and the data can be modified. At this point, the business object logic changes the value of the *ETag* field.

In the example in the figure, ETag Check in UPDATE Operation, both OData Clients read the data with value 'X' in the *ETag* field. When Client 1 sends an update request with *ETag* value 'X', this request is accepted because *ETag* field value 'X' matches the value on the database. During the update of the data, the *ETag* field value is changed, to 'Y' in our example.

The *ETag* 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.

In the example the update request of Client 2 is rejected because it is sent with *ETag* field value 'X'. By comparing this value to the current value on the database, the business object logic sees that a concurrent modify access took place and that Client 2 is operating on an outdated version of the data.

ETag Definition and Implementation



In RAP Business Objects, ETag handling is activated by adding keywords `etag master` or `etag dependent` to the behavior definition of the related entity. Root entities are often ETag masters that log the changes of every business object entity that is part of the BO. The keyword `master` is followed by the name of a field that is part of the business object entity. You must make sure that the value of this field is changed during every modify operation on this entity.

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. Once the ETag is exposed to the service, OData clients will include the Tag value in any relevant request.

Data Definition (Data Model View)

```

define root view entity D437_I_TEXT
  as select from d437_text
{
  ...
  @Semantics.systemDateTime.lastChangedAt: true
  changed_at as ChangedAt,
  @Semantics.user.lastChangedBy: true
  changed_by as ChangedBy,
  @Semantics.systemDateTime.createdAt: true
  created_at as CreatedAt,
  @Semantics.user.createdBy: true
  created_by as CreatedBy
}

```

Timestamp of last change: ideal Etag field

These fields are maintained by RAP runtime

Figure 55: Recommended ETag Field in Managed Scenario

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.

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.systemDateTimeChangedAt: true

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

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.

Unit 2

Exercise 6

Establish Optimistic Concurrency Control

Business Scenario

In this exercise, you will establish optimistic concurrency control in your RAP Business Object. To do this, you set one field as an *ETag* (*Entity Tag*) field and make sure the RAP runtime changes the field value during every write access to the Business Object.



Note:

In this exercise, replace `##` with the number that your instructor assigned to you.

Table 6: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Projection)	D437B_C_TRAVEL

Task 1: Enable Automatic Update of Administrative Fields

Make sure that the RAP runtime automatically updates administrative fields (`ChangedAt`, `ChangedBy`) when writing changes to the database.

1. In the definition of your data model view (`Z##_I_Travel`), add annotation `@Semantics.user.lastChangedBy: true` to view field `ChangedBy`.
2. Add annotation `@Semantics.systemDateTime.lastChangedAt: true` to the view field `ChangedAt`.
3. Activate the data definition.
4. Test your executable program (`Z##_EML_1`) and change the status of one of your flight travels.
5. Use the *Data Preview* tool in Eclipse to confirm that fields `ChangedAt` and `ChangedBy` contain up to date values.

Task 2: Define ETag Field

Enable optimistic concurrency control for your RAP Business Object by defining a suitable field as the ETag field and expose the ETag definition to the OData Service definition.

1. In your behavior definition for the data model view (`Z##_I_Travel`), add addition `etag master`, followed by field name `ChangedAt`.
2. Activate the behavior definition.

3. In your behavior projection, that is, in the behavior definition for the projection view (`Z##_C_Travel`), add addition use `etag`.
4. Activate the behavior projection.

Establish Optimistic Concurrency Control

Business Scenario

In this exercise, you will establish optimistic concurrency control in your RAP Business Object. To do this, you set one field as an *ETag* (*Entity Tag*) field and make sure the RAP runtime changes the field value during every write access to the Business Object.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 6: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Model)	D437B_I_TRAVEL
CDS Behavior Definition (Projection)	D437B_C_TRAVEL

Task 1: Enable Automatic Update of Administrative Fields

Make sure that the RAP runtime automatically updates administrative fields (`ChangedAt`, `ChangedBy`) when writing changes to the database.

1. In the definition of your data model view (`Z##_I_Travel`), add annotation `@Semantics.user.lastChangedBy: true` to view field `ChangedBy`.
 - a) See the source code extract from the model solution.
2. Add annotation `@Semantics.systemDateTime.lastChangedAt: true` to the view field `ChangedAt`.
 - a) See the source code extract from the model solution.
3. Activate the data definition.
 - a) Choose `Activate` or press `Ctrl + F3`.
4. Test your executable program (`Z##_EML_1`) and change the status of one of your flight travels.
 - a) Perform this step as in previous exercises.
5. Use the *Data Preview* tool in Eclipse to confirm that fields `ChangedAt` and `ChangedBy` contain up to date values.
 - a) Right-click anywhere in the definition of your data model view and choose `Open With → Data Preview`.

- b) Compare your code to the following extract from the model solution:

 Data Definition **D437B_I_TRAVEL**:

```
define root view entity D437b_I_Travel
  as select from d437b_travel

{
  ...

  @Semantics.systemDateTime.lastChangedAt: true
  changed_at as ChangedAt,
  @Semantics.user.lastChangedBy: true
  changed_by as ChangedBy

  ...
}
```

Task 2: Define ETag Field

Enable optimistic concurrency control for your RAP Business Object by defining a suitable field as the ETag field and expose the ETag definition to the OData Service definition.

1. In your behavior definition for the data model view (*Z##_I_Travel*), add addition `etag master`, followed by field name `ChangedAt`.
 - a) See source code extract from the model solution.
2. Activate the behavior definition.
 - a) Perform this step as in previous exercises.
3. In your behavior projection, that is, in the behavior definition for the projection view (*Z##_C_Travel*), add addition `use etag`.
 - a) See the source code extract from the model solution.
4. Activate the behavior projection.
 - a) Perform this step as in previous exercises.
 - a) Compare your code to the following extract from the model solution:

 Behavior Definition **D437B_I_TRAVEL**

```
...
define behavior for D437b_I_Travel alias Travel
persistent table d437b_travel
lock master
authorization master ( instance )
etag master ChangedAt
{
  ...
}
```

 Behavior Definition **D437B_C_TRAVEL**

```
...
```

```
define behavior for D437b_C_Travel alias Travel
use etag
{
    ...
}
```



LESSON SUMMARY

You should now be able to:

- Describe pessimistic concurrency control (locking)
- Enable optimistic concurrency control

Defining Actions and Messages



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define and implement an action
- Expose actions to OData services
- Provide a button in SAP Fiori elements
- Define exception classes for RAP
- Access application data in behavior implementations

Action Definition



- **Actions in RAP**
 - Part of business object behavior
 - Non-standard modifying operation
 - Defined in behavior definition
 - Implemented in local handler class
- **Special Action Types (can be combined)**
 - Internal action
 - Static action
 - Factory action
- **Parameters (optional)**
 - Input parameters
 - One result parameter



Figure 56: Actions in RAP

In RAP, an action is a non-standard modifying operation that is part of the business logic.

You define an action for an entity in the behavior definition using the `action` keyword. The logic of the action is implemented in a dedicated method of the handler class.

By default, actions have an instance reference and are executable by OData requests as well as by EML from any ABAP coding (public instance action).

The following special types of actions exist:

Internal Action

To only provide an action for the same BO, the *Internal* option can be set before the action name. An internal action can only be accessed from the business logic inside the business object implementation.

Static Action

The *Static* option allows you to define a static action that is not bound to any instance but relates to the complete entity.

Factory Action

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.



Note:

Instance factory actions are not supported for the managed implementation type.

When defining an RAP action, defining parameters is optional. Parameters can be input or output parameters.

Input Parameters

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

Output Parameters

The output parameter for an action is defined with the keyword `result`. The `result` parameter for actions is optional. However, if a result parameter is declared in the action definition, it must be filled in the implementation. If the result parameter is defined with addition selective, the action consumer can decide whether the result shall be returned completely or only parts of it. This can help improve performance. A result cardinality determines the multiplicity of the output.



Behavior Definition

```

managed;

define behavior for D437_I_Text
  ...
{
    static action issue_message;

    action condense_text result [0..1] $self;
}

```

Figure 57: Example: Action Definition

Instance Action – operates on specific entity instance

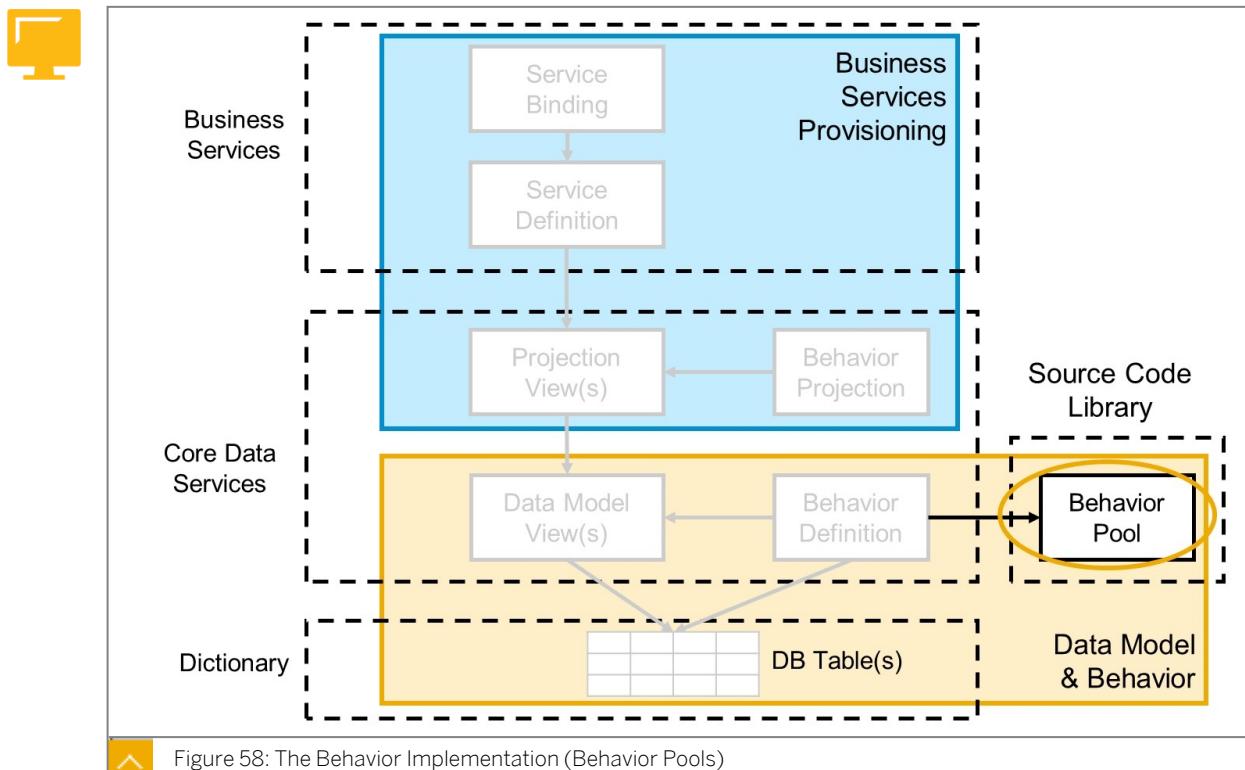
Result parameter up to one data sets

Result type is D437_I_Text (same as entity)

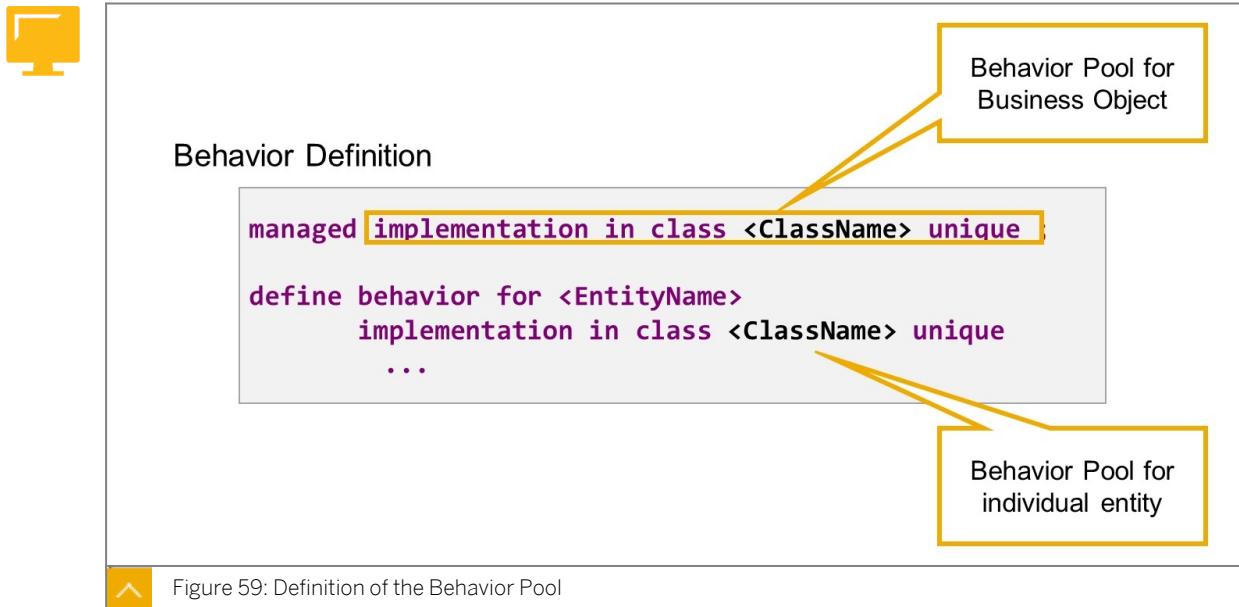
Static Action – independent from entity instances

The behavior definition in the example defines two public actions: a static action and an instance action. The static action has no parameters at all. The instance action has a result parameter with cardinality [0..1] and the symbolic type \$self. The type \$self indicates that the result has the same type as the entity. The action can return up to one data sets of type D437_I_Text.

Action Implementation



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. 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.



In an unmanaged implementation scenario, a behavior pool is always required.

In a managed implementation scenario, a behavior pool is only required if the behavior definition contains components that can't be handled by the managed RAP BO provider, such as actions or validations.

You specify a behavior pool for the RAP business object by adding implementation in class <ClassName> unique to the behavior definition header (statement managed or unmanaged). The mandatory addition unique defines that each operation can be implemented exactly once.



Hint:

The recommended name for the behavior pool starts with BP_ (or <namespace>BP_), followed by the name of the RAP Business Object. This name is part of the comment, generated by the template for behavior definitions.



Note:

By adding the syntax above to the behavior definition header, you define one behavior pool for the entire RAP BO. In more complex scenarios, with BOs that consist of many entities, you can define behavior pools for individual entities by adding the syntax to the define behavior for statement.



Warning: "Class ... does not exists"

Quick fix to create the class

```

1 managed implementation in class zbp_00_i_text unique;
2
3 define behavior
4 persistent table d437_tex^~^
5 lock master
{
  static action
    action condense_text [0..1] $self;
  validation_text
}

```

Create behavior implementation class zbp_00_i_text

Invoking Quickfix:

- Click Warning Icon or
- Right-click class name and choose Quick Fix or
- Click class name and press Ctrl + 1

Figure 60: Creating the Behavior Pool

If the ABAP class to which the behavior definition refers does not yet exist, the editor displays a warning. You can create the ABAP class using a quick fix.

This quick fix creates the class pool and generates the required local class or classes and the required methods for all parts of the behavior definition that need implementation.



Note:

To invoke the quick fix, the behavior definition has to be saved and activated.



No warning, behavior pool already exists

Quick fix to add implementation method for new action

```

1 managed implementation in class zbp_00_i_text unique;
2
3 define behavior for Z00_I_TEX   alias text
4 persistent table d437_tex^~^
5 lock master
{
  static action issue_message;
  action condense_text result [0..1] $self;
  validation_text
}

```

Add missing method for action condense_text in local handler class lhc_text

Invoking Quickfix:

- Right-click action name and choose *Quick Fix*
- or
- Click action name and press Ctrl + 1

Figure 61: Creating the Action Handler Method

If the behavior definition contains the action definition, the quick fix will automatically create the local handler class and the action implementation method.

If the behavior pool already exists when you add an action (or anything else that needs implementation), you have to add the missing implementation method to the behavior pool yourself.

There is a quick fix for updating the behavior pool. To invoke this quick fix, place the cursor on the name of the action and choose Ctrl + 1.



Note:

The editor displays no warning about the missing implementation. So you cannot invoke the quick fix by clicking the warning icon. There is warning when you open the behavior pool and perform a syntax check, but no quick fix is available.



Source code of (empty) global class

Local handler class for entity text

Implement the action here

Keys of affected entity instances

Navigate to the local class(es)

```
[S4M] ZBP_00_I_TEXT
1 CLASS zbp_00_i_text DEFINITION PUBLIC ABSTRACT FINAL FOR BEHAVIOR OF
2 ENDCLASS.
3
4 CLASS zbp_00_i_text IMPLEMENTATION.
5 ENDCLASS.
```

```
[S4M] ZBP_00_I_TEXT
10 CLASS lhc_text DEFINITION INHERITING FROM cl_abap_behavior_handler.
2 PRIVATE SECTION.
3
4 METHODS issue_message FOR MODIFY
5   IMPORTING keys FOR ACTION text~issue_message.
6
7 ENDCLASS.
8
9 CLASS lhc_text IMPLEMENTATION.
10
11 METHOD issue_message.
12   ENDMETHOD.
13
14 ENDCLASS.
```

Figure 62: Navigating to the Action Handler Method

The implementation of an action is contained in a local handler class as part of the behavior pool. To navigate to the definition and implementation of the local class choose the *Local Types* tab.

The generated action handler method is located in the local handler class for the related RAP BO entity. This local class inherits from the base handler class CL_ABAP_BEHAVIOR_HANDLER. The name of the generated local handler class is lhc_, followed by the alias name of the entity. If there is no alias, the technical name of the entity is used instead.

Actions in OData Services

 Behavior Definition (Projection)

```

projection;

define behavior for D437_C_Text
{
    ...
    use action issue_message external 'IssueMessage';

    use action condense_text as condense;
}

```

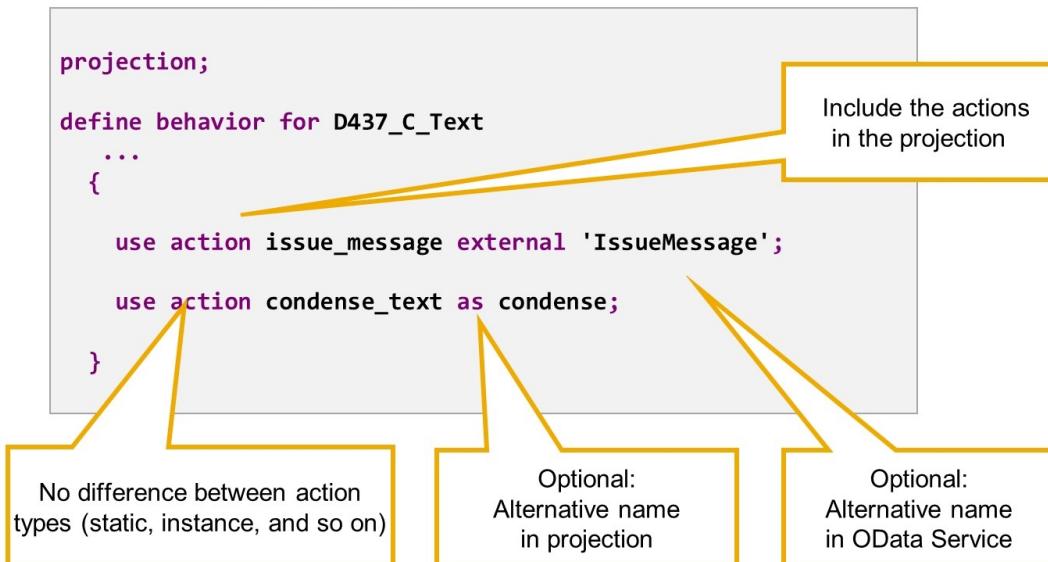


Figure 63: Example: Action Projection

The syntax element `use action` is used to add actions from the underlying base behavior definition to the projection. Only such actions can be reused that are defined in the underlying behavior definition.

Some additions existing to adjust or extend the action definition. Among those is addition `as` to define an alias for the action in the projection layer. Addition `external` defines an alias for external usage, for example in the OData Service. This external name can be much longer than the alias name in ABAP and needs to be used when defining the corresponding UI annotations.



Note:

You can use additions `as` and `external` for the same action projection.

Actions in SAP Fiori Elements



Metadata Extension of Projection View

```

...
annotate view D437_C_Text with
{
  ...
@UI: {
  lineItem: [ { position: 20, importance: #HIGH },
    { type: #FOR_ACTION,
      dataAction: 'IssueMessage',
      label: 'Issue a Message' }
  ],
  identification: [ { position: 20, importance: #HIGH },
    { type: #FOR_ACTION,
      dataAction: 'condense',
      label: 'Condense the Text' }
  ]
}
Text;
...
}

```

Add Button to List Report Page

Add Button to Object Page

Figure 64: Action Buttons in UI Metadata

On the UI, actions are triggered by choosing an action button. This action button must be configured in the backend in the metadata of the related CDS view. Depending on where you want to use an action button on the UI (list report, or object page), use the annotation `@UI.lineItem` or `@UI.identification` to display an action button.



Note:

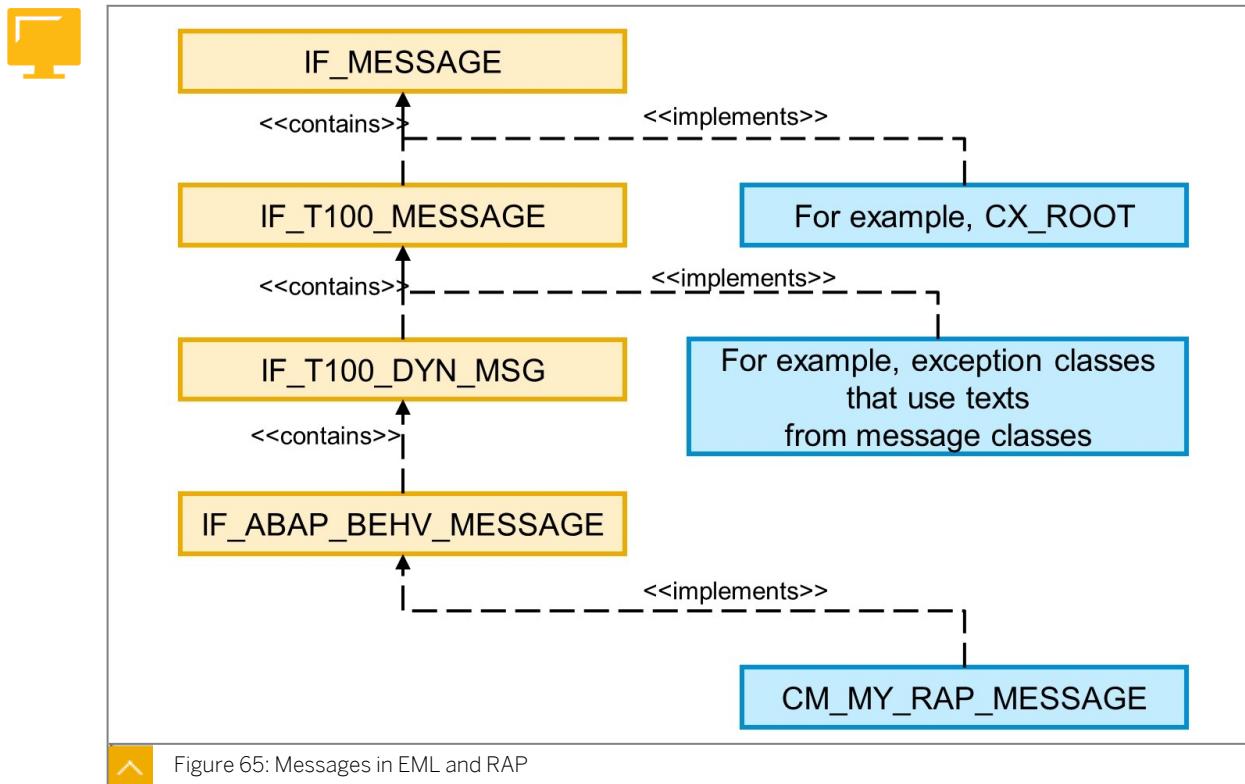
The UI-annotations for actions must be used as element annotation. However, it does not have an impact on which element the action is annotated. The action button always appears at the same position on the UI.



Hint:

If the behavior projection defines an external alias for action, you have to specify this alias after the `dataAction` keyword. If the projection defines an ABAP alias (keyword `as`) but no external alias, you have to use this ABAP alias.

Messages in RAP



Messages offer an important way to guide and validate consumer and user actions, and help to avoid and resolve problems. Messages are important to communicate problems to a consumer or user. Well-designed messages help to recognize, diagnose, and resolve issues.

The message concept of EML and RAP is based on the proven concept of class-based messages. At runtime, a message is represented by an instance of an ABAP class that implements global interface `IF_MESSAGE`.



Note:

In ABAP, all class-based exceptions are message objects, too, because exception classes inherit from `CX_ROOT`, which implements interface `IF_MESSAGE`.

For messages in EML and RAP, it is not sufficient to implement interface `IF_MESSAGE`. The relevant classes have to implement the specialized interface `IF_ABAP_BEHV_MESSAGE`.



Note:

To distinguish the classes for messages from exception classes and usual ABAP classes, their name should start with `CM_` instead of `CX_` or `CL_`.

ABAP Class for Class-based Messages

```

CLASS zcm_00_message DEFINITION PUBLIC
  FINAL CREATE PUBLIC .

  PUBLIC SECTION.

    INTERFACES if_abap_behv_message .
    *  INTERFACES if_t100_dyn_msg .
    *  INTERFACES if_t100_message .

  METHODS constructor
    IMPORTING
      textid  LIKE if_t100_message=>t100key OPTIONAL
      severity LIKE if_abap_behv_message~m_severity OPTIONAL
      ...
    CONSTANTS:
      BEGIN OF textid_example,
        msgid TYPE symsgid VALUE 'DEVS4D437',
        msgno TYPE symsgno VALUE '100',
        attr1 TYPE scx_attrname VALUE '',
        attr2 TYPE scx_attrname VALUE '',
        attr3 TYPE scx_attrname VALUE '',
        attr4 TYPE scx_attrname VALUE ''
      END OF textid_example.
  ENDCCLASS.

```

Annotations:

- Class implements required interface**: Points to the line `INTERFACES if_abap_behv_message .`
- Implicitly implemented, often listed for clarity**: Points to the lines `* INTERFACES if_t100_dyn_msg .` and `* INTERFACES if_t100_message .`
- Constructor parameters for text ID and severity**: Points to the line `textid LIKE if_t100_message=>t100key OPTIONAL`.
- Constant with a possible value for TextID**: Points to the line `attr1 TYPE scx_attrname VALUE ''`.

Figure 66: ABAP Class for Class-based Messages

Classes which implement interface `IF_ABAP_BEHV_MESSAGE` have the following important instance attributes:

`IF_ABAP_BEHV_MESSAGE~M_SEVERITY`

Based on elementary type `IF_ABAP_BEHV_MESSAGE~T_SEVERITY`. Used to specify the message type (error, warning, information, success).

Possible values in `IF_ABAP_BEHV_MESSAGE~SEVERITY`.

`IF_T100_MESSAGE=>t100key`

Based on structure type `IF_T100_MESSAGE=>SCX_T100KEY`. Used to identify the message class (ID), the message number, a type and, if applicable, values for the placeholders &1, &2, &3, and &4 in the message text.

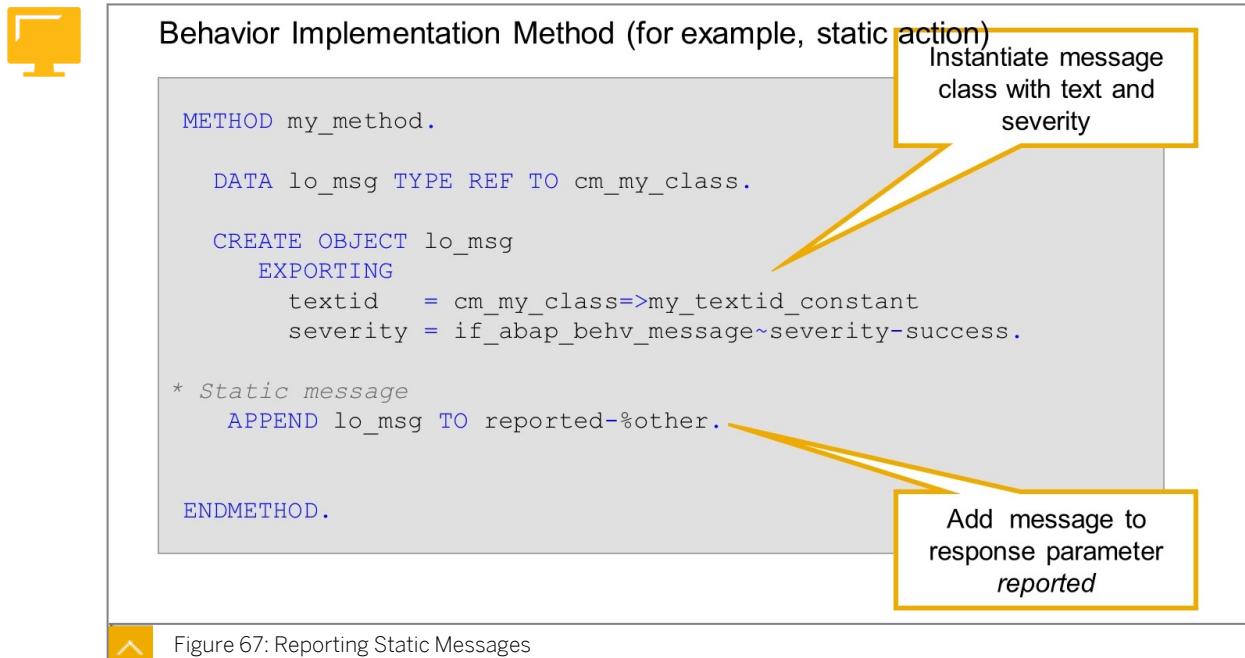
During instantiation of the class, both attributes need to be filled in the constructor logic, either hard coded or with suitable import parameters. It is a recommended practice to define one or more structured constants in the public section, to define possible value combinations for `TEXTID`. These constants can then be used to supply a constructor parameter `textid` when instantiating the message class.



Hint:

The SAP GUI based class builder (SE24) offers a dedicated *Texts* tab to easily define such constants. In ABAP Development Tools, you can use source code template `textIDExceptionClass` for this purpose.

Reporting Static Messages



The screenshot shows a portion of SAP ABAP code within a graphical interface. A yellow callout box points to the line `DATA lo_msg TYPE REF TO cm_my_class.` with the text "Instantiate message class with text and severity". Another yellow callout box points to the line `APPEND lo_msg TO reported-%other.` with the text "Add message to response parameter reported".

```

METHOD my_method.

DATA lo_msg TYPE REF TO cm_my_class.

CREATE OBJECT lo_msg
EXPORTING
textid   = cm_my_class->my_textid_constant
severity = if_abap_behv_message~severity-success.

* Static message
APPEND lo_msg TO reported-%other.

ENDMETHOD.

```

Figure 67: Reporting Static Messages

Certain behavior implementation methods, like, for example, action implementation methods and validation implementation methods, have an implicit response parameter `reported`. This parameter indicates that you can issue messages as part of the implementation. To do so, you have to create an instance of a message class, that is, a class that implements interface `IF_ABAP_BEHV_MESSAGE`, and store a reference to this instance in parameter `reported`.

Static messages, which are not related to an entity instance, are stored in table-like component `%other`.

Reporting Instance Messages

 Behavior Implementation Method (for example, instance action)

```

METHOD my_method.

  DATA lo_msg TYPE REF TO cm_my_class.

  CREATE OBJECT lo_msg
    EXPORTING
      textid   = cm_my_class->my_textid_constant
      severity = if_abap_behv_message~severity-success.

  * Instance message

  DATA ls_wa LIKE LINE OF reported-text.
  ls_wa-%tky = ...
  ls_wa-%msg = lo_msg.

  APPEND ls_wa TO reported-text.

ENDMETHOD.
```

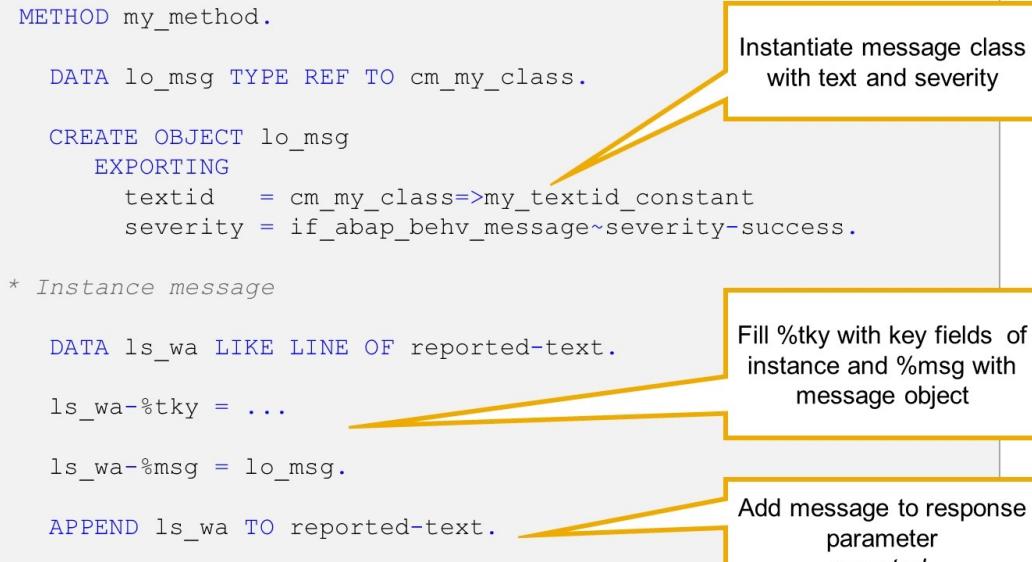


Figure 68: Reporting Instance Messages

Instance messages, that is, messages related to a instance of a RAP BO entity, are stored in one of the other components of parameter `reported`.

If, for example, the message relates to an instance of entity `Z00_C_Text`, the message is stored in internal table `reported-Z00_C_Text`.



Note:
If the behavior definition contains an alias name for the entity, `reported` uses this alias name.

First, the key fields of the related entity instance are filled, for example via named include `%tky`. Then component `%msg` is filled with a reference to the created message object.

EML in RAP BO Implementations

 Behavior Implementation Method (for example, Action Implementation)

```

METHOD my_method.

    READ ENTITY IN LOCAL MODE my_entity
        ALL FIELDS WITH ...
        RESULT ...

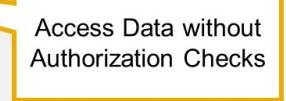
    ...

    MODIFY ENTITY IN LOCAL MODE my_entity
        UPDATE FIELDS ( ... )
        WITH ...
        FAILED ...

    ...

ENDMETHOD.

```

 Access Data without Authorization Checks

 Update Fields that are Read-only for consumers

 Figure 69: EML Inside Behavior Implementation

To access a RAP BOs data from inside its behavior implementation, EML statements are used. There is no major difference in the syntax.

But there is an addition, `IN LOCAL MODE`, that can currently only be used in the RAP BO implementations for the particular RAP BO itself. That means that `IN LOCAL MODE` can only be used for this RAP BO's implementation classes in the behavior pool or other classes that are called from those implementation classes (for example, legacy code or other reused logic contained elsewhere).

The addition is used to exclude feature controls and authorization checks. It can be added to `READ ENTITY/MODIFY ENTITY` and to the long forms `READ ENTITIES/MODIFY ENTITIES`.

An example use case could be an action implementation that wants to update a field, that is set to `readOnly` for consumers of the BO.



Note:

The editor issues a warning if it detects an EML statement where `IN LOCAL MODE` could be used, but is missing.

Unit 2

Exercise 7

Define and Implement an Action

Business Scenario

In this exercise, you define an action in the behavior definition of your RAP Business Object and implement it to set the status of Travel to *Canceled*. You then extend the UI of your SAP Fiori elements app with a button to trigger the action.



Note:

In this exercise, replace `##` with the number that your instructor assigned to you.

Table 7: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	<code>D437B_I_TRAVEL</code>
ABAP Class	<code>CM_D437B_TRAVEL</code>
ABAP Class	<code>BP_D437B_I_TRAVEL</code>
Behavior Definition (Projection)	<code>D437B_C_TRAVEL</code>
Metadata Extension	<code>D437B_C_TRAVEL</code>

Task 1: Define an Exception Class for Class-Based Messages

Define an ABAP class inheriting from super class `CX_STATIC_CHECK` (suggested name: `ZCM_##_TRAVEL`) and implementing interface `IF_ABAP_BEHV_MESSAGE`. In the class, define two public structured constants for `TEXTID` that refer to messages 120 and 130 of message class `DEVS4D437` (suggested name for the constants: `cancel_success` and `already_cancelled`). Make sure that the constructor offers an optional parameter to set attribute `if_abap_behv_message~m_severity`.

1. Create a new ABAP class inheriting from class `CX_STATIC_CHECK`.
2. In the public section of the class, define a structured constant (suggested name: `cancel_success`).



Note:

Use source code template `TextIDExceptionClass`.

3. Edit the values for the components of the structured constant. Make the constant refer to message 120 of message class `DEVS4D437`.

4. In the public section of the class, define a second structured constant (suggested name: `already_cancelled`) and make it refer to message 130 of message class `DEVS4D437`.
5. Edit the definition of the constructor of your exception class. Add an optional parameter (suggested name: `severity`) with the same type that was used for attribute `if_abap_behv_message~m_severity`.
6. Edit the implementation of the constructor of your exception class. Add a statement to use the new import parameter to set the value of attribute `if_abap_behv_message~m_severity`.
7. Activate the exception class.

Task 2: Define an Instance Action

For the root node of your RAP Business Object, define a new action (suggested name: `set_to_cancelled`). Use a quick fix to generate the behavior pool (global class) and navigate to the source code of the action implementation.

1. Open the behavior definition for your data model view (`Z##_I_TRAVEL`).
2. Add statement `action`, followed by the name of the action (suggested name: `set_to_cancelled`).
3. Perform a syntax check.

Are there any syntax errors?

4. Uncomment the `implementation in .. addition to statement managed;` and keep the suggested name for the behavior implementation class (`ZBP##_I_TRAVEL`).
5. Activate the behavior definition.
6. Use the quick fix function of the editor to generate the behavior implementation class.



Note:

To use this quick fix, it is mandatory that the behavior definition is saved and active.

7. Navigate to the implementation of generated handler method `set_to_cancelled`.

Task 3: Retrieve Data of Selected Flight Travels

In the handler method for the action, use EML to retrieve the data of the selected flight travel(s). Loop at the data and issue an error message for all travels that are already canceled (use an instance of your exception class `ZCM##_TRAVEL` with a suitable text ID).

1. In the implementation of method `set_to_cancelled`, implement a `READ ENTITY` statement or a `READ ENTITIES` statement for your RAP BO (`Z##_I_TRAVEL`). Read all fields of all travels that the user selected before triggering the action.

**Note:**

Import parameter `keys` contains the keys of the selected travels.

2. Define an internal table (suggested name: `gt_read_in`), type it with the required derived type to use it as input for a read access to your RAP BO, and fill it with the keys from import parameter `keys`.

**Note:**

Alternatively, you can use a `VALUE` or `CORRESPONDING` expression as input for the `READ ENTITY` or `READ ENTITIES` statement.

3. Define an internal table (suggested name: `gt_travel`), and type it with the required derived type to store the result of a read access to your RAP BO.

**Note:**

Alternatively, you can use an inline declaration after addition `RESULT` of the `READ ENTITY` or `READ ENTITIES` statement.

4. Implement a loop over the result, either with a suitable work area or with a suitable field symbol.
5. If the component `status` already contains value `C` create an instance of your exception class `ZCM_##_TRAVEL` with text ID `already_cancelled` and severity `if_abap_behv_message~severity-error`.
6. Issue the message by adding a new row to component `z##_i_travel` of export parameter `reported`. Do not forget to specify the key of the affected flight travel.

Task 4: Update Status of Selected Flight Travels

For travels that are not yet canceled, set the status to `C` and update the node instance data. Issue a suitable success message from your exception class `ZCM_##_TRAVEL`.

1. If the value of `Status` is not yet `C`, implement a `MODIFY ENTITY` statement or a `MODIFY ENTITIES` statement for your entity (`Z##_I_TRAVEL`) to set the value of `status` to `C`.
2. Define an internal table (suggested name: `gt_update`), type it with the required derived type to use it as input for an update access to your RAP BO, and fill it with one row that contains the key of the current flight travel and value `C` in the `Status` column .

**Note:**

Alternatively, you can use a `VALUE` expression as input for the `MODIFY ENTITY` statement.

3. Define a structure (suggested name: `gs_failed`), and type it with the required derived type to store the response `FAILED` of a read access to your RAP BO.

**Note:**

Alternatively, you can use an inline declaration after addition FAILED of the MODIFY ENTITY or MODIFY ENTITIES statement.

4. Evaluate the response. If `gs_failed` is initial, create an instance of your exception class `ZCM_##_TRAVEL` with text ID `cancel_success` and severity
`if_abap_behv_message~severity-success`, and report the message as before.
5. Activate the behavior implementation class.

Task 5: Test the Action in the SAP Fiori Elements App

Add the action to the behavior projection to make it available in the OData UI Service and comment out the basic operations (Create, Update, and Delete). In the metadata extension for your projection view (`Z##_C_TRAVEL`) add the necessary annotation to define a clickable button on the list reporting page and link it to your action. Test the action by clicking the button in your SAP Fiori elements app.

1. Edit the behavior definition for the projection view (behavior projection) and add the statement `use action set_to_cancelled;`
2. Comment out the three statements `use create;`, `use update;`, and `use delete;`.
3. Activate the behavior definition.
4. Open the metadata extension for your consumption view and scroll down to view element `Status`.
5. In the annotation array for annotation `@UI.lineItem`, add an array element with sub annotations `type`, `dataAction`, and `label`.

**Note:**

An annotation array is a comma-separated list of array elements in square brackets. At the moment, the annotation array of annotation `@UI.lineItem` contains just one array element `{ position: 10, importance: #HIGH }`. To add another array element, add a comma and a pair of curly brackets.

6. In the annotation array element, add three subannotations with the following values:

Subannotation	Value
<code>type:</code>	<code>#FOR_ACTION</code>
<code>dataAction:</code>	<code>'set_to_cancelled'</code>
<code>label:</code>	<code>'Cancel Travel '</code>

**Note:**

If your action has a different name, you have to adjust the value for annotation `dataAction` accordingly.

7. Activate the metadata extension.
8. Refresh the preview of the service as SAP Fiori elements app and test the button.

Unit 2 Solution 7

Define and Implement an Action

Business Scenario

In this exercise, you define an action in the behavior definition of your RAP Business Object and implement it to set the status of Travel to *Canceled*. You then extend the UI of your SAP Fiori elements app with a button to trigger the action.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 7: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437B_I_TRAVEL
ABAP Class	CM_D437B_TRAVEL
ABAP Class	BP_D437B_I_TRAVEL
Behavior Definition (Projection)	D437B_C_TRAVEL
Metadata Extension	D437B_C_TRAVEL

Task 1: Define an Exception Class for Class-Based Messages

Define an ABAP class inheriting from super class `CX_STATIC_CHECK` (suggested name: `ZCM##_TRAVEL`) and implementing interface `IF_ABAP_BEHV_MESSAGE`. In the class, define two public structured constants for `TEXTID` that refer to messages 120 and 130 of message class `DEVS4D437` (suggested name for the constants: `cancel_success` and `already_cancelled`). Make sure that the constructor offers an optional parameter to set attribute `if_abap_behv_message~m_severity`.

1. Create a new ABAP class inheriting from class `CX_STATIC_CHECK`.
 - a) In the *Project Explorer* window, right-click your own package and choose *New → ABAP Class*.
 - b) Enter the name of the new class, a description and super class `CX_STATIC_CHECK`.
 - c) To add the interface, choose *Add....*
 - d) On the *Add ABAP Interface* window, enter the name of the interface and choose *OK* to close the dialog window.
 - e) Choose *Next >*, and *Finish* to confirm the preselected transport request.

2. In the public section of the class, define a structured constant (suggested name: `cancel_success`).



Note:

Use source code template `TextIDExceptionClass`.

- a) Navigate to the definition part of the class and place the cursor right after statement `PUBLIC SECTION.`
- b) Enter `text` and press **Ctrl + Space**.
- c) On the suggestion list, place the cursor on `TextIDExceptionClass` and press **Enter**.
3. Edit the values for the components of the structured constant. Make the constant refer to message 120 of message class `DEVS4D437`.
 - a) See the source code extract from the model solution.
4. In the public section of the class, define a second structured constant (suggested name: `already_cancelled`) and make it refer to message 130 of message class `DEVS4D437`.
 - a) Perform this step as before.
 - b) See the source code extract from the model solution.
5. Edit the definition of the constructor of your exception class. Add an optional parameter (suggested name: `severity`) with the same type that was used for attribute `if_abap_behv_message~m_severity`.
 - a) See the source code extract from the model solution.
6. Edit the implementation of the constructor of your exception class. Add a statement to use the new import parameter to set the value of attribute `if_abap_behv_message~m_severity`.
 - a) See the source code extract from the model solution.
7. Activate the exception class.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution:

ABAP class `CM_D4357B_TRAVEL`:

```

CLASS cm_d437b_travel DEFINITION
  PUBLIC
  INHERITING FROM cx_static_check
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.

    INTERFACES if_abap_behv_message.
    INTERFACES if_t100_dyn_msg .
    INTERFACES if_t100_message .

  CONSTANTS:
    BEGIN OF cancel_success,
      msgid TYPE symmsgid VALUE 'DEVS4D437',

```

```

msgno TYPE symsgno VALUE '120',
attr1 TYPE scx_attrname VALUE '',
attr2 TYPE scx_attrname VALUE '',
attr3 TYPE scx_attrname VALUE '',
attr4 TYPE scx_attrname VALUE '',
END OF cancel_success.

CONSTANTS:
BEGIN OF already_cancelled,
  msgid TYPE symsgid VALUE 'DEVS4D437',
  msgno TYPE symsgno VALUE '130',
  attr1 TYPE scx_attrname VALUE '',
  attr2 TYPE scx_attrname VALUE '',
  attr3 TYPE scx_attrname VALUE '',
  attr4 TYPE scx_attrname VALUE '',
END OF already_cancelled.

METHODS constructor
  IMPORTING
    !textid  LIKE if_t100_message=>t100key OPTIONAL
    !previous LIKE previous OPTIONAL
    !severity TYPE if_abap_behv_message~t_severity OPTIONAL
  .
  .
PROTECTED SECTION.
PRIVATE SECTION.

ENDCLASS.

CLASS cm_d437b_travel IMPLEMENTATION.

METHOD constructor ##ADT_SUPPRESS_GENERATION.
  CALL METHOD super->constructor
  EXPORTING
    previous = previous.

  if_abap_behv_message~m_severity = severity.

  CLEAR me->textid.
  IF textid IS INITIAL.
    if_t100_message~t100key = if_t100_message=>default_textid.
  ELSE.
    if_t100_message~t100key = textid.
  ENDIF.
ENDMETHOD.
ENDCLASS.
```

Task 2: Define an Instance Action

For the root node of your RAP Business Object, define a new action (suggested name: `set_to_cancelled`). Use a quick fix to generate the behavior pool (global class) and navigate to the source code of the action implementation.

1. Open the behavior definition for your data model view (`Z##_I_TRAVEL`).
 - a) Perform this step as you did in previous exercises.
2. Add statement `action`, followed by the name of the action (suggested name: `set_to_cancelled`).
 - a) See the source code extract from the model solution.
3. Perform a syntax check.
 - a) Press **Ctrl + F2** and analyze the *Problems* tab below the source code editor.

Are there any syntax errors?

Yes, there is a syntax error "implementation in class ... unique" must be specified for the implementation category "managed".

4. Uncomment the `implementation in ..` addition to statement `managed`; and keep the suggested name for the behavior implementation class (`ZBP_##_I_TRAVEL`).
 - a) See the source code extract from the model solution.
5. Activate the behavior definition.
 - a) Press `Ctrl + F3`.
6. Use the quick fix function of the editor to generate the behavior implementation class.



Note:

To use this quick fix, it is mandatory that the behavior definition is saved and active.

- a) Either right-click the name of the class and choose *Quick Fix* or choose the warning icon next to statement `managed`.
- b) Double-click *Create behavior implementation class ZBP_##_I_TRAVEL*.
- c) Enter a description for implementation class and choose *Next >*.
- d) Select the same transport request as before and choose *Finish*.
7. Navigate to the implementation of generated handler method `set_to_cancelled`.
 - a) In the behavior pool (global class `ZBP_##_I_TRAVEL`), navigate to the *Local Types* tab.
 - b) The `set_to_cancelled` method is part of local behavior handler class `Ihc_z##_i_travel`.
 - c) Compare your code to the following extract from the model solution:

Behavior Definition **D437B_I_TRAVEL**:

```
managed implementation in class bp_d437b_i_travel unique;

define behavior for D437b_I_Travel alias Travel
persistent table d437b_travel
lock master
etag master ChangedAt
{
  create;
  update;
  delete;

  action set_to_cancelled;

  mapping for D437b_Travel corresponding
  {
    AgencyID = agencynum;
    TravelDescription = trdesc;
```

```

CustomerID = customid;
StartDate = stdat;
EndDate = enddat;
ChangedAt = changed_at;
ChangedBy = changed_by;

}

```

Task 3: Retrieve Data of Selected Flight Travels

In the handler method for the action, use EML to retrieve the data of the selected flight travel(s). Loop at the data and issue an error message for all travels that are already canceled (use an instance of your exception class `ZCM_##_TRAVEL` with a suitable text ID).

1. In the implementation of method `set_to_cancelled`, implement a `READ ENTITY` statement or a `READ ENTITIES` statement for your RAP BO (`Z##_I_TRAVEL`). Read all fields of all travels that the user selected before triggering the action.



Note:
Import parameter `keys` contains the keys of the selected travels.

- a) See the source code extract from the model solution.
2. Define an internal table (suggested name: `gt_read_in`), type it with the required derived type to use it as input for a read access to your RAP BO, and fill it with the keys from import parameter `keys`.



Note:
Alternatively, you can use a `VALUE` or `CORRESPONDING` expression as input for the `READ ENTITY` or `READ ENTITIES` statement.

- a) See the source code extract from the model solution.
3. Define an internal table (suggested name: `gt_travel`), and type it with the required derived type to store the result of a read access to your RAP BO.



Note:
Alternatively, you can use an inline declaration after addition `RESULT` of the `READ ENTITY` or `READ ENTITIES` statement.

- a) See the source code extract from the model solution.
4. Implement a loop over the result, either with a suitable work area or with a suitable field symbol.
 - a) See source code extract from the model solution.
 5. If the component `status` already contains value C create an instance of your exception class `ZCM_##_TRAVEL` with text ID `already_cancelled` and severity `if_abap_behv_message~severity-error`.
 - a) See the source code extract from the model solution.

6. Issue the message by adding a new row to component `z##_i_travel` of export parameter `reported`. Do not forget to specify the key of the affected flight travel.
 - a) See the source code extract from the model solution.

Task 4: Update Status of Selected Flight Travels

For travels that are not yet canceled, set the status to C and update the node instance data. Issue a suitable success message from your exception class `ZCM_##_TRAVEL`.

1. If the value of `Status` is not yet C, implement a `MODIFY ENTITY` statement or a `MODIFY ENTITIES` statement for your entity (`Z##_I_TRAVEL`) to set the value of `status` to C.
 - a) See the source code extract from the model solution.
2. Define an internal table (suggested name: `gt_update`), type it with the required derived type to use it as input for an update access to your RAP BO, and fill it with one row that contains the key of the current flight travel and value C in the `Status` column .



Note:

Alternatively, you can use a `VALUE` expression as input for the `MODIFY ENTITY` statement.

- a) See the source code extract from the model solution.
3. Define a structure (suggested name: `gs_failed`), and type it with the required derived type to store the response `FAILED` of a read access to your RAP BO.



Note:

Alternatively, you can use an inline declaration after addition `FAILED` of the `MODIFY ENTITY` or `MODIFY ENTITIES` statement.

- a) See the source code extract from the model solution.
4. Evaluate the response. If `gs_failed` is initial, create an instance of your exception class `ZCM_##_TRAVEL` with text ID `cancel_success` and severity `if_abap_behv_message~severity-success`, and report the message as before.
 - a) See the source code extract from the model solution.
5. Activate the behavior implementation class.
 - a) Press Ctrl + F3.
 - b) Compare your code to the following extract from the model solution:

ABAP Class `BP_D437B_TRAVEL`:

```
METHOD set_to_cancelled.

  DATA lo_msg TYPE REF TO cm_devs4d437.

  DATA ls_reported_travel LIKE LINE OF reported-travel.

  DATA lt_read_in TYPE TABLE FOR READ IMPORT d437b_i_travel.
  DATA ls_read_in LIKE LINE OF lt_read_in.
```

```

DATA lt_travel TYPE TABLE FOR READ RESULT d437b_i_travel.

DATA lt_update TYPE TABLE FOR UPDATE d437b_i_travel.
DATA ls_update LIKE LINE OF lt_update.

DATA ls_failed TYPE RESPONSE FOR FAILED d437b_i_travel.

FIELD-SYMBOLS <ls_key> LIKE LINE OF keys.
FIELD-SYMBOLS <ls_travel> LIKE LINE OF lt_travel.

DATA ls_key LIKE LINE OF keys.

LOOP AT keys ASSIGNING <ls_key>.
  MOVE-CORRESPONDING <ls_key> TO ls_read_in.
  APPEND ls_read_in TO lt_read_in.
ENDLOOP.

READ ENTITY IN LOCAL MODE d437b_i_travel
  ALL FIELDS WITH lt_read_in
  RESULT lt_travel.

*  READ ENTITIES OF d437b_i_travel IN LOCAL MODE
*    ENTITY travel
*      ALL FIELDS WITH lt_read_in
*        RESULT lt_travel.

LOOP AT lt_travel ASSIGNING <ls_travel>.

  IF <ls_travel>-status = 'C'. "already cancelled
    CREATE OBJECT lo_msg
      EXPORTING
        textid    = cm_d437b_travel=>already_cancelled
        severity = if_abap_behv_message=>severity-error.

    ls_reported_travel-%tky = <ls_travel>-%tky.
    ls_reported_travel-%msg = lo_msg.

    APPEND ls_reported_travel TO reported-travel.

  ELSE.
    CLEAR lt_update.
    ls_update-%tky = <ls_travel>-%tky.
    ls_update-status = 'C'.
    APPEND ls_update TO lt_update.

* Update Status of travel not yet cancelled

    MODIFY ENTITY IN LOCAL MODE d437b_i_travel
      UPDATE FIELDS ( status ) WITH lt_update
      FAILED ls_failed.

*    MODIFY ENTITIES OF d437b_i_travel IN LOCAL MODE
*      ENTITY travel
*        UPDATE FIELDS ( status ) WITH lt_update
*        FAILED ls_failed.

    IF ls_failed IS INITIAL.

* Success message
    CREATE OBJECT lo_msg
      EXPORTING
        textid    = cm_d437b_travel=>cancel_success

```

```

        severity = if_abap_behv_message=>severity-success.

        ls_reported_travel-%tky = <ls_travel>-%tky.
        ls_reported_travel-%msg = lo_msg.
        APPEND ls_reported_travel TO reported-travel.

    ENDIF.
ENDIF.

ENDLOOP.

ENDMETHOD.

```

Alternative coding with modern, expression-based syntax:

```

METHOD set_to_cancelled.

    READ ENTITY IN LOCAL MODE d437b_i_travel
        ALL FIELDS WITH CORRESPONDING #( keys )
        RESULT DATA(lt_travel).

    *  READ ENTITIES OF d437b_i_travel IN LOCAL MODE
    *  ENTITY travel
    *      ALL FIELDS WITH CORRESPONDING #( keys )
    *      RESULT DATA(lt_travel).

    LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

        IF <ls_travel>-status = 'C'. "already cancelled

            APPEND VALUE #( %tky = <ls_travel>-%tky
                            %msg = NEW_cm_d437b_travel(
                                textid =
cm_d437b_travel=>already_cancelled
                                severity =
if_abap_behv_message=>severity-error
                            )
                            )
            TO reported-travel.

        ELSE.

    * Update Status of travel

            MODIFY ENTITY IN LOCAL MODE d437b_i_travel
                UPDATE FIELDS ( status )
                    WITH VALUE #(
                        ( %tky = <ls_travel>-%tky
                            status = 'C'
                        )
                    )
            FAILED DATA(ls_failed).

    *  MODIFY ENTITIES OF d437b_i_travel IN LOCAL MODE
    *  ENTITY travel
    *      UPDATE FIELDS ( status )
    *          WITH VALUE #(
    *              ( %tky = <ls_travel>-%tky
    *                  status = 'C'
    *              )
    *          )
    *      FAILED DATA(ls_failed).

```

```

        IF ls_failed IS INITIAL.

            APPEND VALUE #( %tky = <ls_travel>-%tky
                            %msg = NEW cm_d437b_travel(
                                textid   =
cm_d437b_travel=>cancel_success
                                severity =
if_abap_behv_message=>severity-success
                            )
                )
            TO reported-travel.

        ENDIF.
ENDIF.

ENDLOOP.

ENDMETHOD.

```

Task 5: Test the Action in the SAP Fiori Elements App

Add the action to the behavior projection to make it available in the OData UI Service and comment out the basic operations (Create, Update, and Delete). In the metadata extension for your projection view (`Z##_C_TRAVEL`) add the necessary annotation to define a clickable button on the list reporting page and link it to your action. Test the action by clicking the button in your SAP Fiori elements app.

1. Edit the behavior definition for the projection view (behavior projection) and add the statement `use action set_to_cancelled;`
 - a) See the source code extract from the model solution.
2. Comment out the three statements `use create;`, `use update;`, and `use delete;`
 - a) See the source code extract from the model solution.
3. Activate the behavior definition.
 - a) Press Ctrl + F3.
4. Open the metadata extension for your consumption view and scroll down to view element `Status`.
 - a) Perform this step as in previous exercises.
5. In the annotation array for annotation `@UI.lineItem`, add an array element with sub annotations `type`, `dataAction`, and `label`.



Note:

An annotation array is a comma-separated list of array elements in square brackets. At the moment, the annotation array of annotation@UI.lineItem contains just one array element (`{ position: 10, importance: #HIGH }`). To add another array element, add a comma and a pair of curly brackets.

- a) See the source code extract from the model solution.
6. In the annotation array element, add three subannotations with the following values:

Subannotation	Value
<i>type:</i>	#FOR_ACTION
<i>dataAction:</i>	'set_to_cancelled'
<i>label:</i>	'Cancel Travel'

**Note:**

If your action has a different name, you have to adjust the value for annotation `dataAction` accordingly.

- a) See the source code extract from the model solution.
7. Activate the metadata extension.
- a) Press Ctrl + F3.
8. Refresh the preview of the service as SAP Fiori elements app and test the button.
- a) If the browser window with the SAP Fiori elements app is still open, reload the page, otherwise open your service binding, choose the name of your projection view and choose *Preview*....
 - b) Execute the query by choosing *Go*.
 - c) Choose one travel and choose *Cancel Travel* on the header toolbar of the table.
 - d) Verify that your action implementation is executed, you should see a success or error message. When in doubt, set a breakpoint in your method and debug your coding.
 - e) Compare your code to the following extract from the model solution:

Behavior Definition **D437B_C_TRAVEL**:

```
projection;

define behavior for D437b_C_Travel
use etag
{
//  use create;
//  use update;
//  use delete;

  use action set_to_cancelled;
}
```

Metadata Extension **D437B_C_TRAVEL**:

```
@UI: {
    lineItem:      [ { position: 10, importance: #HIGH },
                     { type:          #FOR_ACTION,
                       dataAction: 'set_to_cancelled',
                       label:        'Cancel Travel' } ],
    identification: [ { position: 30, importance: #HIGH } ]
}

Status;
```



LESSON SUMMARY

You should now be able to:

- Define and implement an action
- Expose actions to OData services
- Provide a button in SAP Fiori elements
- Define exception classes for RAP
- Access application data in behavior implementations

Implementing Authority Checks



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Restrict read access with access controls
- Implement explicit authority checks

Authorization Overview



Authorisation Checks for Read Operations

- Access Controls in ABAP CDS
- Filter result returned by CDS entity based on authorization

Authorization Checks for Modify Operations

- **Authority master/dependent** in behavior definition
- Implementation in RAP handler method(s)
- Restrict execution of actions
- Restrict create/update/delete

Authorization Check for OData Services Consumption

- Restrict access to OData services
- No further steps required for service developer

Figure 70: Authorization Control in RAP

Business applications require an authorization concept for their data and for the operations on their data. Display and CRUD operations, as well as specific business-related activities, are therefore, allowed for authorized users only.

In a transactional development scenario in RAP, you can add authorization checks to various components of an application. In this case, different mechanisms are used to implement the authorization concept.

Authorization Checks for Read Operations

To protect data from unauthorized read access, the ABAP CDS provides its own authorization concept based on a data control language (DCL). To restrict read access to RAP business objects, it is sufficient to model DCL for the CDS entities used in RAP business objects. 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 model and 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.

Authorization Checks for Modify Operations

In RAP business objects, modifying operations, such as standard operations (create, update, delete) and actions can be checked against unauthorized access during runtime. To retrieve user authorizations for incoming requests, authority checks are included in the behavior definition and implementation for your business object. In case of negative authorization results, the modification request is rejected.

Note:

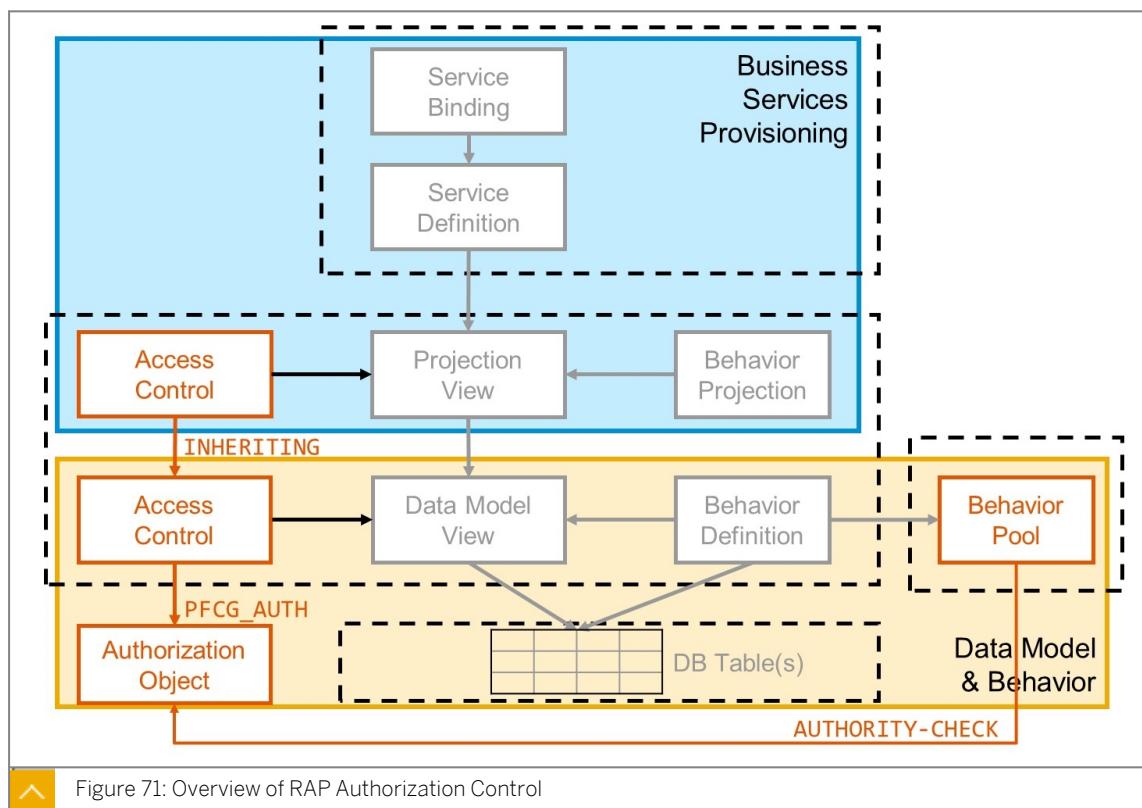
In UI scenarios, authority checks for modify operations is particularly important, because the rejection of a modification request is visualized to the user (Consumer hints). For example, an action button will be disabled for line items for which the user lacks execution authority.

Authorizations for OData Services Consumption

SAP Gateway provides predefined roles as templates for developers, administrators, end users of the content scenarios, and support colleagues. SAP customers will configure the roles based on these templates and assign users to the roles.

Note:

Important: For you as service developer, there are no further steps required for the service to be consumed externally within the customer's landscape. In particular, you don't need to provide any authorization default values of the authorization objects and specific role templates required for execution of your service. SAP Gateway already provides predefined roles as templates for accessing SAP Fiori apps.



The figure, Overview of RAP Authorization Control, shows the main design time components in a transactional development scenario, including the artifacts required for enabling authorization checks at all levels of the application.

Normalized views serve as the data source for modeling the data associated with the business object layer. To check the authorizations for read access, corresponding CDS roles are defined in CDS access controls, using the data control language (DCL). A CDS role specifies access rules. Each access rule defines access to the CDS view that the role is assigned to. Different access controls are created for access control at business object data model level (Data Model View in this figure) and at the consumption level (Projection View in this figure). While the access rules on data model level are usually based on ABAP authorization objects, access controls on consumption level often inherit the rules from the underlying access controls.

The behavior definition on data model level, contains the authorization definition. It specifies for which entities of the RAP BO individual authority checks are applied and which of the checks are performed for individual instances. The handler classes in the behavior pool then provide appropriate code exits for implementing the authorization checks, for example with ABAP statement AUTHORITY-CHECK.

For developers at SAP, no further steps (concerning authorizations) are required for the resulting OData service to be consumed in the customer's landscape. SAP gateway already provides predefined roles as templates for accessing the OData services and SAP Fiori apps.

CDS Access Controls

Access Control with PFCG Authorization

```
define role Z00_I_FLIGHT
{ grant select on Z00_I_Flight
  where (CarrierID) = aspect pfcg_auth(S_CARRID,
                                      CARRID,
                                      ACTVT = '03');
```

Authorization object, Authorization field(s), and Activity "Display"

Access Control with Inheritance

```
define role Z00_C_FLIGHT
{ grant select on Z00_C_Flight
  where inheriting conditions from entity Z00_I_Flight;}
```

Projection view inherits authorizations from data model view

Figure 72: Creating Access Controls

Access controls enable you to filter access to data in the database. If no access control is created and deployed for the CDS entity, a user who can access the CDS entity can view all the data returned.

If you use the PFCG_AUTH aspect in the access control, user-dependent authorizations are used when accessing the CDS view. To implement this, you need an authorization object in the ABAP repository on which to base your authorization check. If you want to see the data, your user must be assigned a role that includes this authorization object with the matching values in the relevant fields.

When CDS views are built on top of each other, each CDS view needs its own access control. For example, an access control defined for an data model view does not also apply to the projection view built on top of this data model view. But it is not necessary to repeat the same conditions repeatedly. By using addition `INHERITING CONDITIONS FROM ENTITY`, one access control can inherit the conditions from another, typically an underlying CDS entity. In this way, a projection view can inherit its conditions from the underlying data model view.



Note:

When inheriting conditions from one entity, it is possible to combine them with the conditions from another entity or to add further restrictions. Simply use the keyword `AND` to link the conditions.



```

@EndUserText.label: '${dcl_source_description}'
@MappingRole: true
define role ${dcl_source_name} {
    ...
}

```

Figure 73: Templates for Creating Access Controls

When creating an Access Control, ADT offers a number of templates for the source code.

The *Define Role with PFCG Aspect* template is a blue print for an Access Control that defines conditions based on authorization objects.

The *Define Role with Inherited Conditions* template uses the addition `INHERITING CONDITIONS FROM ENTITY` instead.

Authority Check in Behavior Implementation

The screenshot shows a code editor window with the title "Behavior Definition". The code is written in a syntax similar to ABAP or Java:

```

managed implementation in class zbp_00_i_text unique;

define behavior for Z00_I_Text alias Text
  persistent table d437_text
  lock master

  authorization master ( instance )
  {
    ...
  }

```

A yellow callout box highlights the line "authorization master (instance)". A yellow arrow points from this box to a yellow-bordered callout box containing the text: "Authorization check based on instances of this (root) entity".

Figure 74: Activate Authorization Control in RAP BO

Authorization control in RAP protects your business object against unauthorized access to data. Authorization control is defined on entity level by adding `authorization master (instance)` or `authorization dependent to the define behavior statement.`



Note:

Currently, only root entities can be authorization masters.

In the brackets after `authorization master`, the following variants are available:

`global`

- Limits access to data or the permission to perform certain operations for a complete RAP BO, independent of individual instances, for example, depending on user roles.
- Must be implemented in the RAP handler method FOR GLOBAL AUTHORIZATION.

`instance`

- Authorization check that is dependent on the state of an entity instance.
- Must be implemented in the RAP handler method FOR INSTANCE AUTHORIZATION. For compatibility reasons FOR AUTHORITY is also supported.

`global, instance`

- Combination of global and instance authorization control: Instance-based operations are checked in the global and in the instance authority check.
- Both RAP handler methods, FOR GLOBAL AUTHORIZATION and FOR INSTANCE AUTHORIZATION, must be implemented.
- The checks are executed at different points in time during runtime.



Note:
In ABAP Release 7.55, only value instance is supported.



Behavior Definition

```

1 managed implementation in class zbp_00_i_text unique;
2
3 define behavior for Z00_I_TEXT alias text
4 persistent table d437_text04
5 lock master
6
7 authorization master( instance )
8
9 {
10     static action is

```

Add missing method for authorization in local handler class lhc_text

Quick fix to create handler method for authorization

Invoking Quickfix:

- Right-click keyword master and choose Quick Fix
- or
- Click keyword master and press Ctrl + 1



Figure 75: Creating the Authorization Handler Method

If the behavior definition contains the authorization addition when you create the behavior pool, the quick fix will automatically create the local handler class and the method or methods for authorization implementation.

If you add the authorization definition when the behavior pool already exists, you have to add the missing implementation method.

There is a quick fix for updating the behavior pool. To invoke this quick fix, place the cursor on the keyword `master` and press `Ctrl + 1`.



Note:
The quick fix only works if you place the cursor on master. It is not offered if the cursor stands on authority or instance.



```

CLASS lhc_text DEFINITION
  INHERITING FROM cl_abap_behavior_handler.

  PRIVATE SECTION.

    METHODS get_authorizations FOR AUTHORIZATION
      IMPORTING keys
      REQUEST requested_authorizations
      FOR text
      RESULT result.

ENDCLASS.

CLASS lhc_text IMPLEMENTATION.

  METHOD get_authorizations.

ENDMETHOD.

ENDCLASS.

```

Generated handler method with specific parameters

Implement checks for individual instances here



Figure 76: Implementing the Authorization Handler Method

Authorization handler methods are defined with addition FOR INSTANCE AUTHORIZATION or with addition FOR GLOBAL AUTHORIZATION. The methods that are required depend on the behavior definition.



Note:

For compatibility reasons, FOR AUTHORITY is also supported and has the same meaning as FOR INSTANCE AUTHORITY. The quick fix version that is generated depends on the system release.

Like all handler methods, authorization handler methods require specific parameters that are supplied or evaluated by the RAP runtime framework. The types of these parameters are derived from the CDS data definition and the CDS behavior definition.



Note:

You can rename the methods and parameters in the definition part of the handler class. But in this course, we stick to the names provided by the quick fix to avoid confusion.



KEYS

- Keys of the affected entity instances
- Type: TABLE FOR AUTHORIZATION KEY

REQUESTED_AUTHORIZATIONS

- Used to perform only requested checks (performance optimization)
- Structure of flags for operations and actions
- Values from constant IF_ABAP_BEHV~MK
- Type: STRUCTURE FOR AUTHORIZATION REQUEST

RESULT

- Used to return instance based authorization
- Internal table with instance keys and flags for operations and actions
- Values in constant IF_ABAP_BEHV~AUTH
- Type: TABLE FOR AUTHORIZATION RESULT



Figure 77: Parameters of the Authority Handler Method

The authority handler method has the following parameters:

KEYS

- Keys of the affected entity instances
- Typed with derived data type TABLE FOR AUTHORIZATION KEY

Requested_authorizations

- Can be used to only perform requested checks (performance optimization)
- Contains flags for requested basic operations (create, update, delete) and actions
- Possible values are the components of constant IF_ABAP_BEHV~MK
- Components depend on behavior definition
- Typed with derived data type STRUCTURE FOR AUTHORIZATION REQUEST

Result

- Contains a table of instance keys and flags for basic operations and actions
- Possible values are the components of constant IF_ABAP_BEHV~AUTH
- Type depends on behavior definition
- Typed with derived data type TABLE FOR AUTHORIZATION RESULT



```

METHOD get_authorizations.

...
READ ENTITY IN LOCAL MODE ...
LOOP AT lt_data INTO ls_data.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD ls_data-carrid
    ID 'ACTVT' FIELD '02'.
*
  IF sy-subrc <> 0.
    ls_result-%tky
      = ls_data-%tky.
    ls_result-%update
      = if_abap_behv=>auth-unauthorized.
    ls_result-%action-my_action
      = if_abap_behv=>auth-unauthorized.
    APPEND ls_result TO result.
  ENDIF.
  ENDLOOP.
ENDMETHOD.

```

Annotations for the code:

- Read data for affected entity instances
- Authorization checks per instance
- No authorization!
- Fill *result* with instance key and flags
- Disallow basic operation *update* and action *my_action*

Figure 78: Example: Instance Authorization Check

The code example shows the implementation of an instance base authority check. The check itself is done with the ABAP statement AUTHORITY-CHECK.

First, the method uses importing parameter keys, to read the data of all entity instances for which the authority check is to be performed.

It then performs the authorization check for each data set (entity instance) in turn. If the user does not have the requested authorization, the logic adds a row to parameter result that contains the key of the entity instance and flags for the disallowed operations.



Note:

The structured constant `IF_ABAP_BEHV=>AUTH` contains components `AUTHORIZED` and `UNAUTHORIZED`. If an authorization check is successful, you can explicitly set the flags to `AUTHORIZED`. This is not necessary if you properly initialize the structure, because the value of `AUTHORIZED` equals the built-in initial value of the flags.

Unit 2 Exercise 8

Implement Authority Checks

Business Scenario

In this exercise, you implement authority checks for your application: read access in CDS access controls, write access by implementing a method of the behavior handler class.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 8: Solution

Repository Object Type	Repository Object ID
CDS Access Control (Model)	D437B_I_TRAVEL
CDS Access Control (Projection)	D437B_C_TRAVEL
CDS Behavior Definition (Model)	D437B_I_TRAVEL
ABAP class	BP_D435B_I_TRAVEL

Task 1: Analyze Authorization Object S_AGENCY

Open the Definition of Authorization Object S_AGENCY and analyze it.

1. Open the definition of authorization object S_AGENCY.
2. Analyze the definition of the authorization object.

Which are the authorization fields of this authorization object?

Which are the permitted activities?

Task 2: Access Controls for CDS Views

Create an access control for your data model view (Z##_I_Travel) and grant access to all data sets where the master record of the current user contains display rights for the related travel agency.

Create an access control for your projection view (`Z##_C_Travel`) that inherits the conditions from the data model view.

Finally, in your SAP Fiori elements app, verify that display access is restricted .

1. Create an access control for your data model view (suggested name: `Z##_I_TRAVEL`). When prompted for a template choose *Define Role with PFCG Aspect*.
2. In the brackets after the keyword `WHERE`, specify the view element that contains the travel agency number.
3. Supply the function `pfcg_auth()` with the name of the authority object and its authorization fields. Specify the authorization field `ACTVT` as a filter field and assign the value for display authorization as filter value.
4. Activate the new access control.
5. Create an access control for your projection view (suggested name: `Z##_C_TRAVEL`). When prompted for a template, choose *Define Role with Inherited Conditions*. Let the access control inherit the conditions from your data model view.
6. After the keyword `ENTITY`, specify your data model view.
7. Activate the new access control.
8. Test your SAP Fiori elements app to verify that not all entries of your database table are now displayed.

Task 3: Instance-based Authorization Control

In the behavior definition for your data model view, enable instance-based authorization control. Define and implement the method for authorization in the local behavior handler class for your root entity. Make sure the user needs change authorization for the travel agency to execute action `SET_TO_CANCELLED`.



Note:

Instead of using statement `AUTHORITY-CHECK` directly, use the method `authority_check()` of the helper class `CL_S4D437_MODEL`. This method simulates different authorizations for different users.

1. Edit the behavior definition for your data model view (`Z##_I_TRAVEL`) and remove the comment sign before addition authorization master (instance).
2. Use a quick fix to generate the method `get_authorizations` in the local behavior handler class.



Note:

You have to place the cursor precisely on the keyword `master`, otherwise the editor does not offer this quick fix.

3. Implement the method. For all affected flight travels, read the content of field `AgencyID`.

**Note:**

You will find the keys of the affected flight travels in the import parameter `keys` of the method.

4. For each affected flight travel, check whether the current user has change authorization for the respective travel agency.

**Note:**

Instead of implementing the statement `AUTHORITY-CHECK`, call the method `authority_check()` of helper class `CL_S4D437_MODEL`. This method simulates different authorization profiles for different users.

5. If the user does not have sufficient authorizations, add an entry to parameter `result` with the current flight travel. Make sure to set the values for `%update` and `%action-set_to_cancelled` appropriately to disallow basic operation `update` and action `%action-set_to_cancelled` for this travel instance.

**Note:**

You find suitable constants in interface `if_abap_behv`.

6. Optional: Only perform the check if the framework request authorization for basic operation `update` or authorization for action `%action-set_to_cancelled`.

**Note:**

Analyze import parameter `requested_authorizations` and compare its components to components of constant `if_abap_behv=>mk` (`mk` stands for marked).

7. Activate your coding and test the explicit authority check in your SAP Fiori elements app.

Implement Authority Checks

Business Scenario

In this exercise, you implement authority checks for your application: read access in CDS access controls, write access by implementing a method of the behavior handler class.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 8: Solution

Repository Object Type	Repository Object ID
CDS Access Control (Model)	D437B_I_TRAVEL
CDS Access Control (Projection)	D437B_C_TRAVEL
CDS Behavior Definition (Model)	D437B_I_TRAVEL
ABAP class	BP_D435B_I_TRAVEL

Task 1: Analyze Authorization Object S_AGENCY

Open the Definition of Authorization Object S_AGENCY and analyze it.

1. Open the definition of authorization object S_AGENCY.
 - a) In ABAP Development Tools, choose *Open ABAP Development Object* or press Ctrl + Shift + A.
 - b) Enter **s_AGENCY**.
 - c) On the list of matching items, choose **S_AGENCY (Authorization Object)** and choose **OK**.
2. Analyze the definition of the authorization object.

Which are the authorization fields of this authorization object?

AGENCYNUM and ACTVT

Which are the permitted activities?

01 (Add and Create), 02(Change), 03(Display)

Task 2: Access Controls for CDS Views

Create an access control for your data model view (*Z##_I_Travel*) and grant access to all data sets where the master record of the current user contains display rights for the related travel agency.

Create an access control for your projection view (*Z##_C_Travel*) that inherits the conditions from the data model view.

Finally, in your SAP Fiori elements app, verify that display access is restricted .

1. Create an access control for your data model view (suggested name: *Z##_I_TRAVEL*). When prompted for a template choose *Define Role with PFCG Aspect*.
 - a) In the *Project Explorer* view on the left, open the context menu for the data definition of your data model view.
 - b) From the context menu, choose *New Access Control*.
 - c) Enter a name and a description and choose *Next >*.
 - d) Confirm the preselected transport request and choose *Next >*.
 - e) In the list of available templates, select *Define Role with PFCG Aspect* and choose *Finish*.
2. In the brackets after the keyword `WHERE`, specify the view element that contains the travel agency number.
 - a) See the source code extract from the model solution.
3. Supply the function `pfcg_auth()` with the name of the authority object and its authorization fields. Specify the authorization field `ACTVT` as a filter field and assign the value for display authorization as filter value.
 - a) See the source code extract from the model solution.
4. Activate the new access control.
 - a) Choose *Activate* or press `Ctrl + F3`.
5. Create an access control for your projection view (suggested name: *Z##_C_TRAVEL*). When prompted for a template, choose *Define Role with Inherited Conditions*. Let the access control inherit the conditions from your data model view.
 - a) Perform this step as before.
6. After the keyword `ENTITY`, specify your data model view.
 - a) See the source code extract from the model solution.
7. Activate the new access control.
 - a) Perform this step as before.
8. Test your SAP Fiori elements app to verify that not all entries of your database table are now displayed.
 - a) Perform this step as before.

- b) Compare your code to the following source code extract from the model solution:

Access control **D437B_I_TRAVEL**:

```
@EndUserText.label: 'Access Control for Flight Travel'
@MappingRole: true
define role D437B_I_TRAVEL {
    grant select on D437B_I_TRAVEL
        where (AgencyID) =
            aspect pfcg_auth(S_AGENCY, AGENCYNUM, ACTVT =
'03');
}
```

Access control **D437B_C_TRAVEL**

```
@EndUserText.label: 'Access Control for Flight Travel Projection'
@MappingRole: true
define role D437B_C_TRAVEL {
    grant select on D437B_C_TRAVEL
        where inheriting conditions from entity
D437b_I_Travel;
}
```

Task 3: Instance-based Authorization Control

In the behavior definition for your data model view, enable instance-based authorization control. Define and implement the method for authorization in the local behavior handler class for your root entity. Make sure the user needs change authorization for the travel agency to execute action `SET_TO_CANCELLED`.



Note:

Instead of using statement `AUTHORITY-CHECK` directly, use the method `authority_check()` of the helper class `CL_S4D437_MODEL`. This method simulates different authorizations for different users.

1. Edit the behavior definition for your data model view (`Z##_I_TRAVEL`) and remove the comment sign before addition authorization master (instance).
 - a) See the source code extract from the model solution.
2. Use a quick fix to generate the method `get_authorizations` in the local behavior handler class.



Note:

You have to place the cursor precisely on the keyword `master`, otherwise the editor does not offer this quick fix.

- a) Place the cursor on the keyword `master`.
- b) Open the context menu and choose *Quick Fix* or press `Ctrl + 1`.
- c) Choose *Add missing method for authorization in local handler class*
3. Implement the method. For all affected flight travels, read the content of field `AgencyID`.

**Note:**

You will find the keys of the affected flight travels in the import parameter keys of the method.

- a) See the source code extract from the model solution.
- 4. For each affected flight travel, check whether the current user has change authorization for the respective travel agency.

**Note:**

Instead of implementing the statement AUTHORITY-CHECK, call the method authority_check() of helper class CL_S4D437_MODEL. This method simulates different authorization profiles for different users.

- a) See the source code extract from the model solution.
- 5. If the user does not have sufficient authorizations, add an entry to parameter result with the current flight travel. Make sure to set the values for %update and %action-set_to_cancelled appropriately to disallow basic operation update and action %action-set_to_cancelled for this travel instance.

**Note:**

You find suitable constants in interface if_abap_behv.

- a) See the source code extract from the model solution.
- 6. Optional: Only perform the check if the framework request authorization for basic operation *update* or authorization for action %action-set_to_cancelled.

**Note:**

Analyze import parameter requested_authorizations and compare its components to components of constant if_abap_behv=>mk (mk stands for marked).

- a) See the source code extract from the model solution.
- 7. Activate your coding and test the explicit authority check in your SAP Fiori elements app.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition D437B_I_TRAVEL:

```
...
define behavior for D437b_I_Travel alias Travel
persistent table d437b_travel
lock master
```

```

etag master ChangedAt
authorization master ( instance )
{
...
}

```

Class BP_D437B_TRAVEL:

```

METHOD get_authorizations.

*      IF      requested_authorizations-%update          =
if_abap_behv=>mk-on
*      OR      requested_authorizations-%action-set_to_cancelled  =
if_abap_behv=>mk-on.

      READ ENTITY IN LOCAL MODE d437b_i_travel
          FIELDS ( AgencyID ) WITH CORRESPONDING #( keys )
          RESULT DATA(lt_travel).

*      READ ENTITIES OF d437b_i_travel IN LOCAL MODE
*          ENTITY travel
*              FIELDS ( agencyid ) WITH CORRESPONDING #( keys )
*                  RESULT DATA(lt_travel).

      LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).
*          AUTHORITY-CHECK OBJECT 'S_AGENCY'
*              ID 'AGENCYNUM' FIELD <ls_travel>-agencyid
*              ID 'ACTVT'      FIELD '02'.
*
*          IF sy-subrc <> 0.

            "Use simulation of different roles for different users
            DATA(lv_subrc) = cl_s4d437_model=>authority_check(
                EXPORTING
                    iv_agencynum = <ls_travel>-agencyid
                    iv_actvt     = '02'
            ).

            IF lv_subrc <> 0.

                APPEND VALUE #( %tky = <ls_travel>-%tky
                    %update = if_abap_behv=>auth-unauthorized
                    %action-set_to_cancelled =
if_abap_behv=>auth-unauthorized
                )
                TO result.

            ENDIF.

            ENDLOOP.
*
        ENDIF.

    ENDMETHOD.

```



LESSON SUMMARY

You should now be able to:

- Restrict read access with access controls
- Implement explicit authority checks

UNIT 3

Update and Create in Managed Transactional Apps

Lesson 1

Enabling Input Fields and Value Help	144
Exercise 9: Provide Input Fields and Value Help	153

Lesson 2

Implementing Input Checks with Validations	159
Exercise 10: Provide Input Checks Through Validations	165

Lesson 3

Providing Values with Determinations	177
Exercise 11: Enable Managed Numbering and Implement Determinations	187

Lesson 4

Implementing Dynamic Feature Control	198
Exercise 12: Implement Dynamic Action and Field Control	205

UNIT OBJECTIVES

- Enable input fields
- Set input fields to read-only and mandatory
- Define value help for input fields
- Explain validations
- Define and implement input checks
- Link messages to input fields
- Describe the numbering concepts in RAP
- Define and implement determinations
- Explain dynamic action, operation, and field control in RAP
- Implement dynamic feature control

Enabling Input Fields and Value Help



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Enable input fields
- Set input fields to read-only and mandatory
- Define value help for input fields

Basic Operation UPDATE



Behavior Definition

```
managed implementation in class zbp_00_i_text unique;

define behavior for Z00_I_Text alias Text
  persistent table d437_text
  lock master
  authorization master ( instance )

{
  ...
  update;
  ...
}
```

Enable standard Operation *Update* for this RAP BO entity

Figure 79: Standard Operations in Behavior Definition

In RAP, `update` is one of the standard operations. Standard operations are also known as CRUD operations, which is an acronym for create, read, update, delete.

While the read operation is always implicitly enabled for each entity listed in a CDS behavior, the modifying operations have to be listed to be available.

To enable standard operation update for an entity, add the statement `update;` to the entity behavior body (curly brackets after statement `define behavior for ...`). To disable the operation, remove the statement or turn it into a comment.

In a managed scenario, the standard operations don't require an ABAP behavior pool, because they are completely handled by the RAP provisioning framework. In an unmanaged scenario, the standard operations must be implemented in the ABAP behavior pool.

Some interesting additions to statement `update` are as follows:

Internal

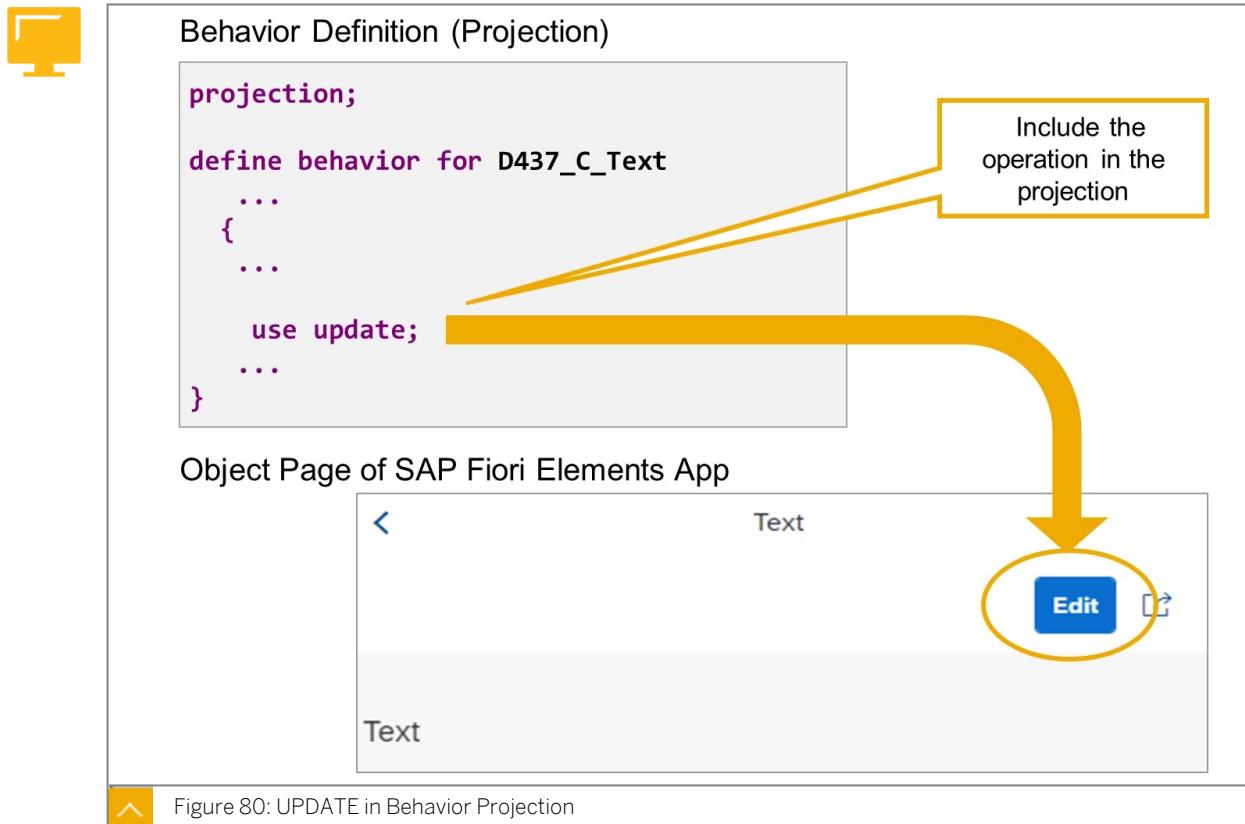
Prefix which disables the operation for external consumers of the BO.

Features: instance or features: global

Enable dynamic feature control. The decision, where and when the operation is enabled, is made dynamically in a behavior implementation.

precheck

A method is called before the update request to prevent unwanted changes from reaching the application buffer.



To make standard operation update available in OData and SAP Fiori, it has to be reused in the behavior definition for the projection view (behavior projection).

To include standard operation update in the OData service, add statement `use update;` to the entity behavior body (curly brackets after statement `define behavior for ...`). To disable the operation, remove the statement or turn it into a comment.



Note:

Behavior projections can have their own implementations, which can be used to augment the implementation of a standard operation or provide additional prechecks. This is a special case that we will not cover in this class.

As soon as standard operation Update is available in the OData service, the generated SAP Fiori elements app displays an *Edit* button on the *Object* page for the related RAP BO entity. By choosing this button, the user enters an edit mode for the data displayed on the page.

**Hint:**

After editing the behavior projection, you might have to perform a hard refresh of the service preview (Ctrl + F5) before you see the new button. Sometimes, you have to clear your browsing data too (Ctrl + Shift + Del).

Static Field Control



Behavior Definition

```
...
define behavior for Z00_I_Text
...
{
    ...
    ...
    field ( readonly ) Field1, Field2, Field3;
    field ( readonly : update ) Field4, Field5;
    ...
}
```

RAP BO does not allow external access to these fields

Editing supported during Create, but not during Update

Object Page of SAP Fiori Elements App

Flight Travel Number:	Travel End Date: *
12119	Sep 24, 2021
Readonly	Editable

Figure 81: Read-only Input Fields

The statement field is used in behavior definitions, to specify the characteristics of fields. The statement is always located between the curly brackets after the statement define behavior (the entity behavior body). Commas can be used to classify multiple fields in the same way.

For the read-only characteristic of fields, the following variants exists:

Field (readonly)

- Static field attribute.
- The RAP BO does not allow consumers to change the values of the specified field or fields.
- This is independent from the standard operation the consumers wants to perform (update, create).

Field (readonly : update)

- Dynamic field attribute.

- Defines a field as read-only during update operations. That means that an external consumer can set the fields in question during create, but cannot change them later.



Note:

If an RAP BO consumer tries to modify a read-only field using EML, a runtime error occurs. The RAP BO behavior logic, for example, an action implementation, can bypass the restriction by using EML statements with the addition `IN LOCAL MODE`.

To include the field characteristics into the OData service, it is not necessary to add the field statements in the behavior projection. The related information is directly included into the OData Service. The SAP Fiori elements UI uses this information for UI hints, that is, depending on the operation, the fields will be displayed as editable or read-only.



Behavior Definition

```
...
define behavior for Z00_I_Text
...
{
  ...
  ...
  field ( mandatory ) Field1, Field2, Field3;
  field ( mandatory : create ) Field4, Field5;
  ...
}
```

RAP BO always requires a value for this field

Value only required during Create but not during Update

Object Page of SAP Fiori Elements App

Travel Description:	Flight Travel Status:*
<input type="text" value="Need a Break"/>	<input type="text" value="A"/>
Not mandatory	Mandatory

Figure 82: Mandatory Input Fields

For the mandatory characteristic of fields, the following variants exists:

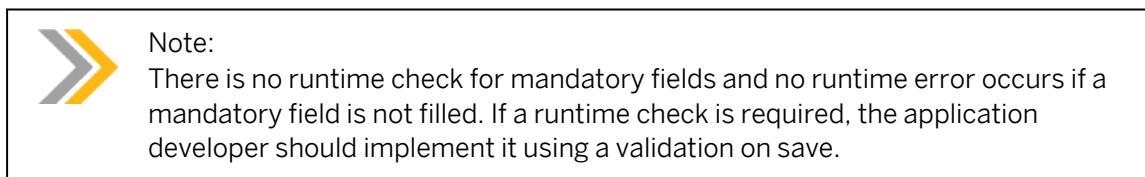
Field (mandatory)

- Static field attribute.
- The RAP BO always requires a value for the specified field(s) before persisting them on the database.
- This is independent from the standard operation the consumers wants to perform (update, create).

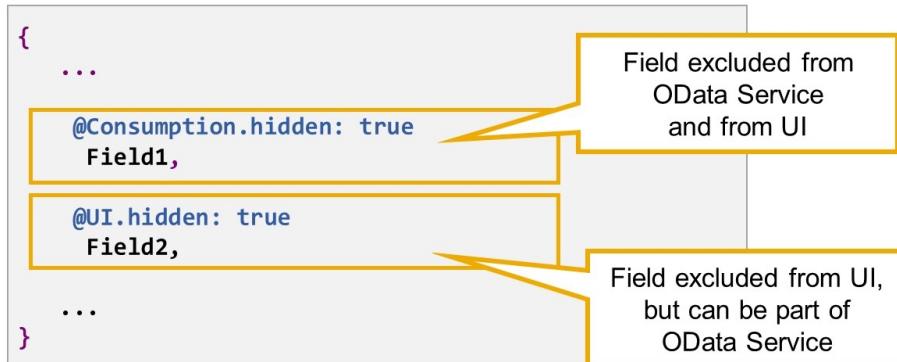
Field (mandatory : create)

- Dynamic field attribute.
- Defines a field as mandatory during create operations. This means that an external consumer must set the fields in question only during create.

In an OData scenario, the fields are marked as mandatory on the user interface.



Data Definition or Metadata Extension



Hiding fields is done on consumption or UI level.

This is not part of the object model (behavior)



Figure 83: Hidden Fields

Hiding fields is a question of consumption and the user interface (UI). It is not part of the object model, therefore, it is not possible to hide fields by editing the CDS behavior definition.

Fields can be hidden by one of the following annotations:

`@Consumption.hidden: true`

The field is not exposed for any consumption. This means that it is part of the OData Service and, therefore, not available on the UI.

`@UI.hidden: true`

The field is not available on the UI. This means that it is not displayed and the user cannot make it visible using personalization. If, however, there is no additional annotation `@Consumption.hidden: true`, the field can still be part of the OData Service.

**Note:**

If you want to hide a field that is needed in the OData service, you cannot use `@Consumption.hidden: true`. You have to use `@UI.hidden: true` instead. A good example is the exclusion of technical key fields and ETag fields from the UI.

Value Help for Input Fields

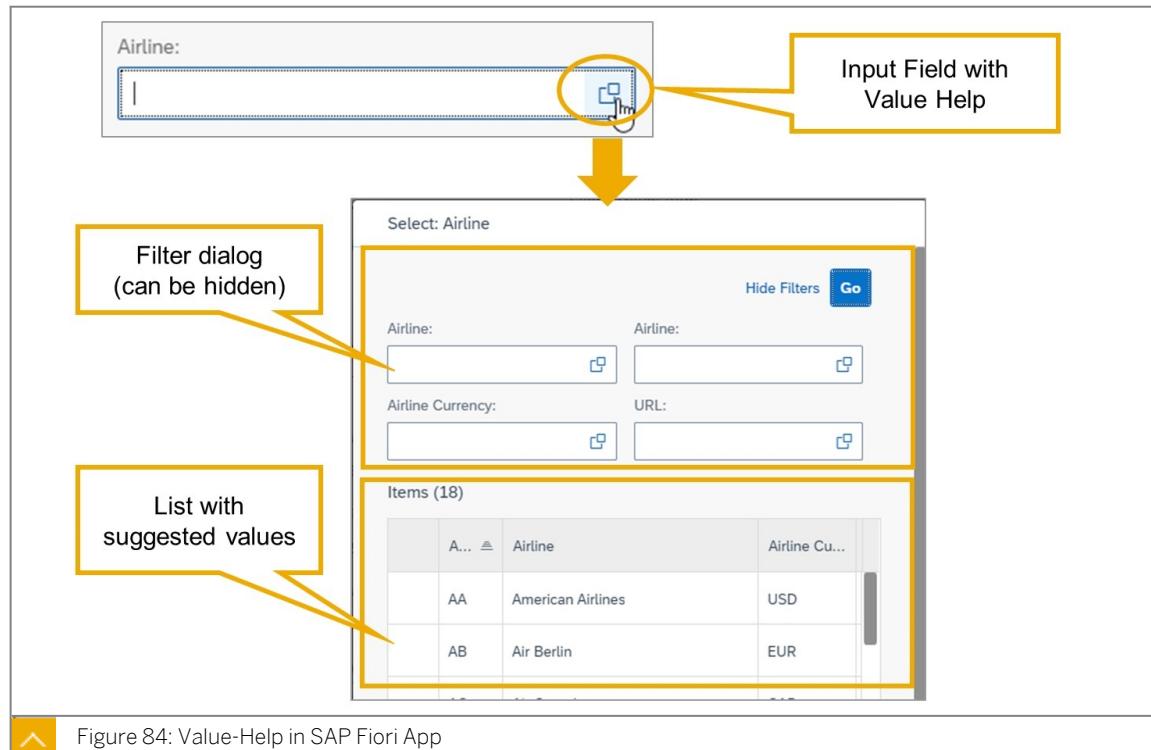
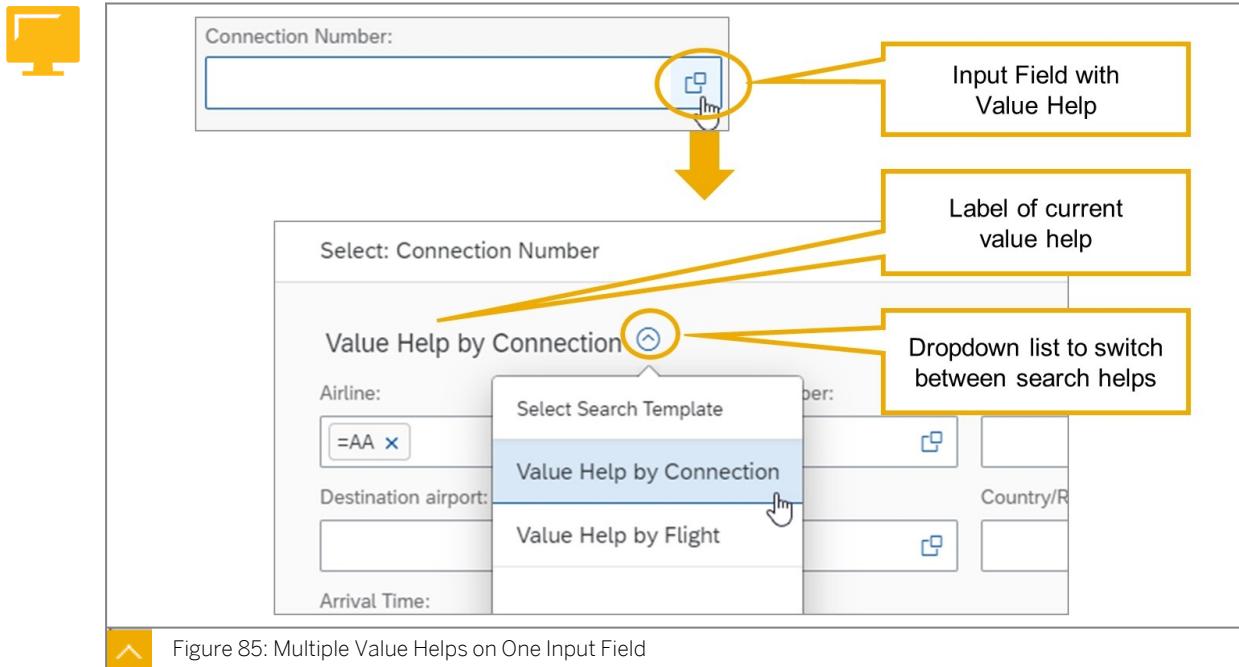


Figure 84: Value-Help in SAP Fiori App

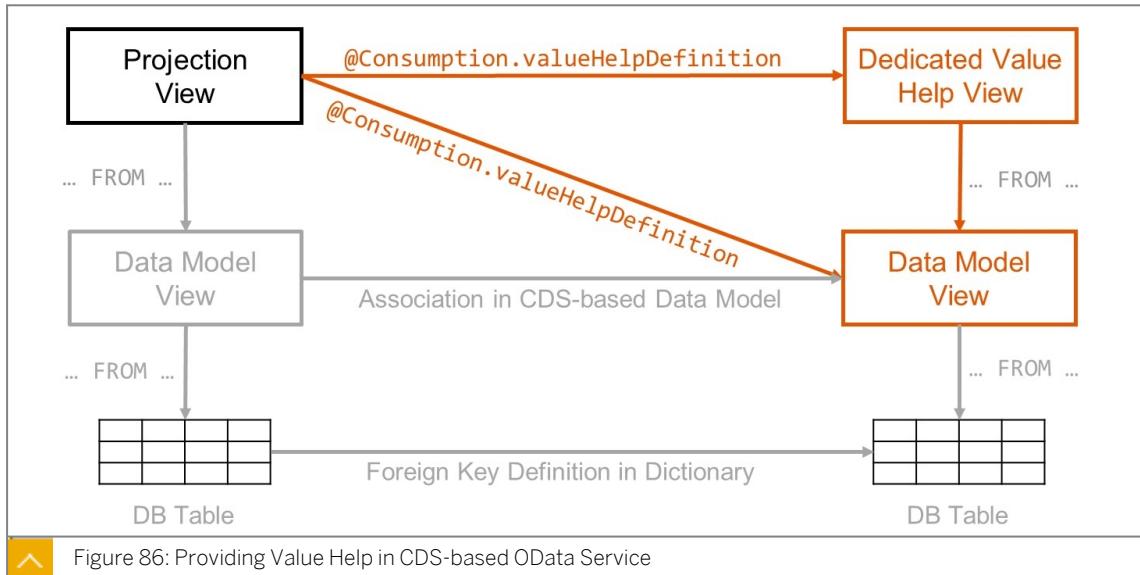
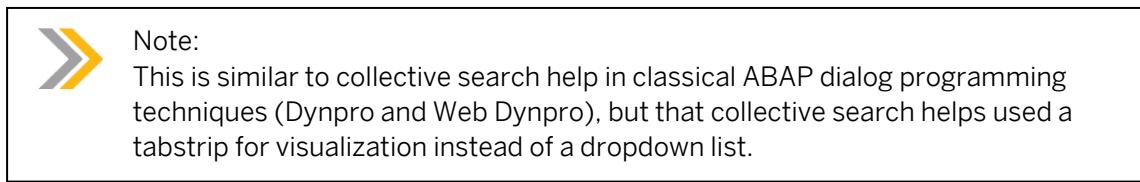
The implementation of a value help in CDS enables the end user to choose values from a predefined list for input fields on a user interface.

In SAP Fiori elements apps, value helps are invoked by choosing the value help button next to the input field or by pressing F4.

By default, the value help dialog consists of a filter dialog, which the user can hide and unhide, and a list with suggested values.



It is possible to provide more than one value help on one input field. The end user can select which value help to use from a dropdown list.



To provide a value help for a given field, you first need a CDS view that contains the values for the value help. This view is referred to as the value help provider. Then, you annotate your field with `@Consumption.valueHelpDefinition` and provide the name of the value help provider and an element for the mapping in the annotation.

**Note:**

It is also possible to provide value help based on associations, but some restrictions apply in this case. We won't discuss this option in this course.

You can use any CDS entity as value help provider that contains the desired values of the element for the input field. However, developers often define dedicated value help views, because the layout and functionality of the value help dialog is derived from the metadata of the value help provider view.

**Note:**

If you are looking for a CDS entity that you can use as value help provider, it might be helpful to look for a foreign key relation in the underlying table or an association in the CDS-based data model. Note that the existence of foreign keys or associations is not a prerequisite for providing a value help.

**Data Definition (Projection View)**

```
define root view entity Z00_C_Booking
  as projection on Z00_I_Booking
{
  ...
  @Consumption.valueHelpDefinition:
    [ { entity: { name: 'D437_I_Carrier',
      element: 'CarrierID' }
    }
  ]
  CarrierID,
  ...
}
```

Figure 87: Simple Value Help

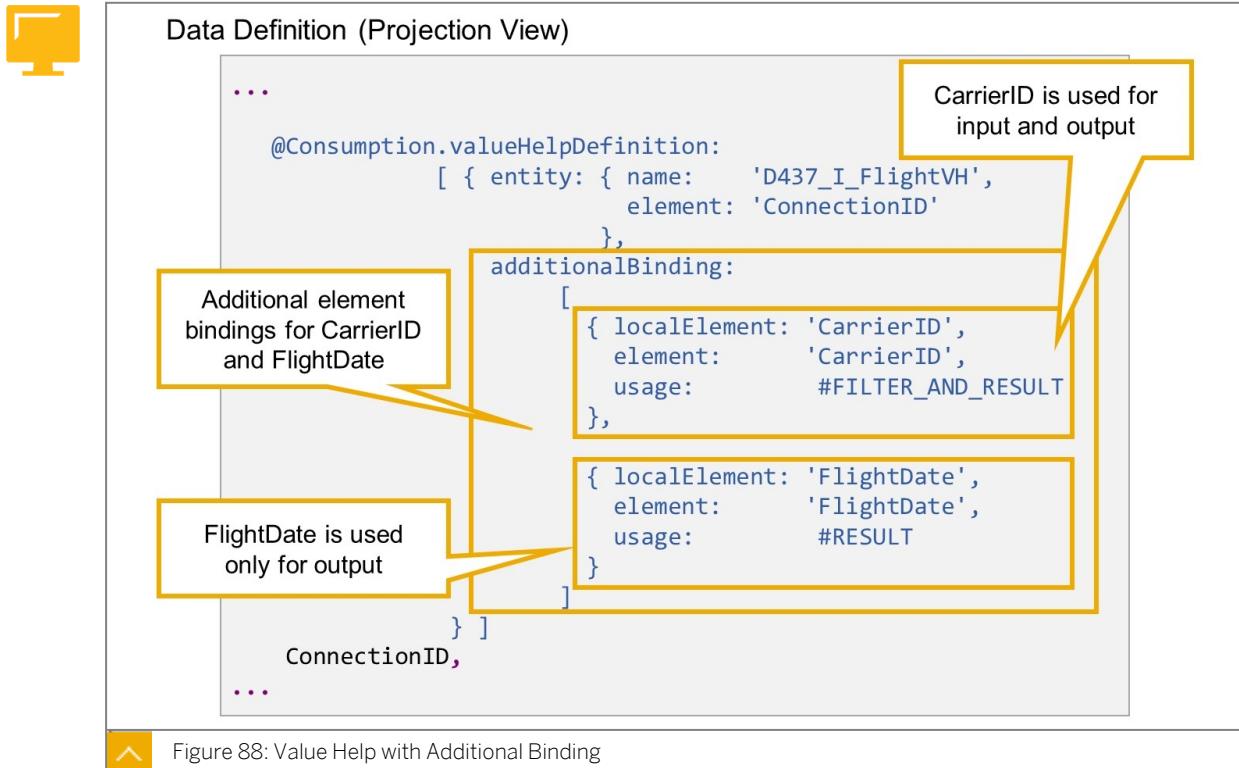
To provide a simple value help for a field, edit the data definition of the source view and annotate the field with the following annotation:

```
@Consumption.valueHelpDefinition: [ { entity: { name: 'entityRef',
  element:
  'elementRef' } } ]
```

Here, `entityRef` is the name of the CDS entity that is used as value help provider and `elementRef` the element of the value help provider that is used as output parameter of the value help.

When you expose the source view in an OData service, the value help provider view is automatically exposed with it. You do not have to list value help provider views in the service definition.

On an SAP Fiori UI, 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.



You use additional binding to define more than one binding conditions between elements of the source view and elements of the value help provider view.

The additional bindings are defined with the subannotation `additionalBindings` of annotation `@Consumption.valueHelpDefinition`. The subannotation is followed by a pair of square brackets (`[..]`) with a comma-separated list of element bindings.

Each element binding connects exactly one element of the source view (`localElement`) and one element of the value help provider view (`element`).

In addition, it specifies a usage with one of the following values:

#FILTER

The value of the referenced element in `localElement` is used as input for the value help. The value help proposes only entries that match this filter value.

#RESULT

The value of the referenced element in `element` is used as additional output. When an entry is selected in the value help, this value is transferred to the input field that is based on the referenced element in `localElement`.

#FILTER_AND_RESULT

The binding is used for input and output.

Unit 3 Exercise 9

Provide Input Fields and Value Help

Business Scenario

In this exercise, you enable direct editing of data in your SAP Fiori elements app. You extend the behavior definition to define input fields as read-only or as mandatory and use CDS annotations to provide value help.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 9: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437C_I_TRAVEL
CDS Behavior Definition (Projection)	D437C_C_TRAVEL
CDS Data Definition (Projection)	D437C_C_TRAVEL

Task 1: Enable Direct Editing of Data

Enable direct editing on the object page of your transactional SAP Fiori elements app. To achieve this, make sure that the behavior definition and behavior projection contain the necessary statements.

1. Open the behavior definition for your data model view (*Z##_I_Travel*) and make sure it contains statement `update;`. If it is missing, add it. If it is commented out, uncomment it.
2. Activate the behavior definition.
3. Open the behavior projection, that is, the behavior definition for the projection view (*Z##_C_Travel*) and make sure it contains statement `use update;`. If it is missing, add it. If it is commented out, uncomment it.
4. Activate the behavior definition.
5. Test your SAP Fiori elements app. For an arbitrary travel, navigate to the *Object* page and make sure that it shows an *Edit* button.

Task 2: Set Input Fields to Read-Only or Mandatory

In the behavior definition for your CDS view entity (*Z##_I_Travel*), add the necessary statements to disallow editing of the semantic key fields and the status field. Add the necessary statements to define all other fields as mandatory, except for the travel description and the hidden fields.

1. Edit the behavior definition of your data model view. Add a `field (...)` statement followed by view elements `AgencyID`, `TravelID`, and `Status`. Inside the brackets, add the relevant key word to disallow editing of the fields.
2. Add a `field (...)` statement, followed by view elements `CustomerID`, `StartDate`, and `EndDate`. Inside the brackets, add the relevant key word to define the fields as mandatory.
3. Repeat the test of your SAP Fiori app to see the effect.

Is there an automatic check on the mandatory fields?

Task 3: Use CDS Annotations to Provide Value-Help

Add the necessary annotation for view element `CustomerID` to define a value help based on CDS view `D437_I_Customer`.

1. Edit the definition of your projection view (`Z##_C_Travel`). Before view element `CustomerID`, add the necessary sub-annotation of annotation `@Consumption` to define the value help for this field.
2. Within the annotation, define one value help based on the CDS view `D437_I_Customer` and its key field.
3. Activate the definition of your projection view.
4. Test your SAP Fiori app. Verify that there is now a value help for the customer field.

Is there an automatic check for existing customer IDs?

Provide Input Fields and Value Help

Business Scenario

In this exercise, you enable direct editing of data in your SAP Fiori elements app. You extend the behavior definition to define input fields as read-only or as mandatory and use CDS annotations to provide value help.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 9: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437C_I_TRAVEL
CDS Behavior Definition (Projection)	D437C_C_TRAVEL
CDS Data Definition (Projection)	D437C_C_TRAVEL

Task 1: Enable Direct Editing of Data

Enable direct editing on the object page of your transactional SAP Fiori elements app. To achieve this, make sure that the behavior definition and behavior projection contain the necessary statements.

1. Open the behavior definition for your data model view (*Z##_I_Travel*) and make sure it contains statement `update;`. If it is missing, add it. If it is commented out, uncomment it.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition.
 - a) Choose *Activate* or press *Ctrl + F3*.
3. Open the behavior projection, that is, the behavior definition for the projection view (*Z##_C_Travel*) and make sure it contains statement `use update;`. If it is missing, add it. If it is commented out, uncomment it.
 - a) See the source code extract from the model solution.
4. Activate the behavior definition.
 - a) Choose *Activate* or press *Ctrl + F3*.
5. Test your SAP Fiori elements app. For an arbitrary travel, navigate to the *Object* page and make sure that it shows an *Edit* button.
 - a) Perform this step as in previous exercises.

Task 2: Set Input Fields to Read-Only or Mandatory

In the behavior definition for your CDS view entity (*Z##_I_Travel*), add the necessary statements to disallow editing of the semantic key fields and the status field. Add the necessary statements to define all other fields as mandatory, except for the travel description and the hidden fields.

1. Edit the behavior definition of your data model view. Add a `field (...)` statement followed by view elements `AgencyID`, `TravelID`, and `Status`. Inside the brackets, add the relevant key word to disallow editing of the fields.
 - a) See the source code extract from the model solution.
2. Add a `field (...)` statement, followed by view elements `CustomerID`, `StartDate`, and `EndDate`. Inside the brackets, add the relevant key word to define the fields as mandatory.
 - a) See the source code extract from the model solution.
3. Repeat the test of your SAP Fiori app to see the effect.
 - a) Perform this step as before.

Is there an automatic check on the mandatory fields?

No, the app displays a red asterisk next to the labels of the mandatory fields, but there is no check yet.

Task 3: Use CDS Annotations to Provide Value-Help

Add the necessary annotation for view element `CustomerID` to define a value help based on CDS view *D437_I_Customer*.

1. Edit the definition of your projection view (*Z##_C_Travel*). Before view element `CustomerID`, add the necessary sub-annotation of annotation `@Consumption` to define the value help for this field.
 - a) See the source code extract from the model solution.
2. Within the annotation, define one value help based on the CDS view *D437_I_Customer* and its key field.
 - a) See the source code extract from the model solution.
3. Activate the definition of your projection view.
 - a) Perform this step as before.
4. Test your SAP Fiori app. Verify that there is now a value help for the customer field.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition **D435C_I_TRAVEL**:

```
...
define behavior for D437c_I_Travel alias Travel
persistent table d437c_travel
lock master
etag master ChangedAt
authorization master ( instance )
```

```
{
...
    update;
...
    field ( readonly ) AgencyID, TravelID, Status;
    field ( mandatory ) CustomerID, StartDate, EndDate;
...
}
```

Behavior Definition D435C_C_TRAVEL:

```
...
define behavior for D437c_C_Travel alias Travel
use etag
{
...
    use update;
...
}
```

Data Definition D435C_C_TRAVEL:

```
...
define root view entity D437c_C_Travel
    as projection on D437c_I_Travel
{
...
    @Consumption.valueHelpDefinition: [
        entity: {
            name:      'D437_I_Customer',
            element:   'Customer'
        }
    ]
}

@Search.defaultSearchElement: true
CustomerID,
...
}
```

Is there an automatic check for existing customer IDs?

No, there is no check yet. The user can enter any customer ID and save the travel.



LESSON SUMMARY

You should now be able to:

- Enable input fields
- Set input fields to read-only and mandatory
- Define value help for input fields

Implementing Input Checks with Validations



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain validations
- Define and implement input checks
- Link messages to input fields

Validation Definition



▪ Validations in RAP

- Part of business object behavior
- Implemented in local handler class
- Executed based on trigger conditions

▪ Trigger Conditions

- Modify operations (create, update, delete)
- Modified fields

▪ Reaction

- Reject inconsistent data
- Return error messages

▪ Restriction

- Not available for unmanaged, non-draft scenarios

Figure 89: Validations in RAP

A validation is an optional part of the business object behavior that checks the consistency of business object instances based on trigger conditions. Validations, like actions, are defined in the behavior definition of the RAP BO and implemented in the behavior pool through a dedicated method of the local handler class.

A validation is implicitly invoked by the business objects framework if the trigger condition of the validation is fulfilled. Trigger conditions can be modify operations (create, update, delete) and modified fields. The trigger condition is evaluated at the trigger time, a predefined point during the BO runtime.

An invoked validation can reject inconsistent instance data from being saved by passing the keys of failed instances to the corresponding table in the FAILED structure. Additionally, a validation can return messages to the consumer by passing them to the corresponding table in the REPORTED structure.

**Note:**

Validations are available for managed scenarios and for unmanaged scenarios with draft. They are not available for unmanaged, non-draft scenarios.

**Behavior Definition**

```
managed;

define behavior for D437_I_Text
  ...
  {

    validation OwnerEqualsUser on save { create;
      update;
    }

    validation textNotEmpty on save { create;
      update;
      field Text;
    }
  }
}
```

update; only works together with create;

For validations only on save is supported

One field or list of fields (comma-separated)



Figure 90: Example: Validation Definition

Validations are defined in the entity behavior definition with the following statement:

`validation <validation_name> on save { <trigger_conditions> }.`

**Note:**

For validations, only the trigger time on save can be stated.

It is mandatory to provide at least one trigger condition within the curly brackets.

The following trigger conditions are supported:

Create;

Validation is executed when an instance is created.

Update

Validation is executed when an instance is updated.

Delete;

Validation is executed when an instance is deleted.

Field <field1>, <field2>, ...;

Validation is executed when the value of one of the specified fields is changed by a create or update operation.

Multiple trigger conditions can be combined.



Note:

The trigger condition `update;` works only in combination with the trigger condition `create;`

The behavior definition in the example defines two validations. The first is triggered by any create or update operation. The other is triggered by changes to the field `text`, either during a create operation or during an update operation.



Note:

The execution order of validations is not fixed. If there is more than one validation triggered by the same condition, you cannot know which validation is executed first.

Validation Implementation



Quick fix to add implementation method for new validation

```

1 managed implementation in class zbp_00_i_text unique;
2
3④ define behavior for Z00_I_TEXT alias text
4 persistent table d437_text
5 lock master
6
7 authorization master( instance )
8 {
9
10④ validation textNotEmpty on save { create;
11
12
13
  
```

Add missing method for validation textnotempty in local handler class lhc_text

Invoking Quick Fix:

- Right-click validation name and choose *Quick Fix* or
- Click validation name and press **CTRL + 1**



Figure 91: Creating the Validation Handler Method

If the behavior definition already contains the validation definition, the quick fix for creating the behavior pool will automatically create the validation implementation method in the local handler class.

If the behavior pool already exists when you add the validation definition, you can use a quick fix to add the missing method to the local handler class. To invoke the quick fix, place the cursor on the name of the validation and press **Ctrl + 1**.



```

CLASS lhc_text DEFINITION
  INHERITING FROM cl_abap_behavior_handler.

PRIVATE SECTION.

  METHODS textnotempty FOR VALIDATE ON SAVE
    IMPORTING keys FOR text~textnotempty.

ENDCLASS.

CLASS lhc_text IMPLEMENTATION.

  METHOD textnotempty.
    ...
    IF ....
      APPEND ... TO failed-text.
      APPEND ... TO reported-text.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

```

Keys of affected entity instances

Reject changes to an instance

Return validation message

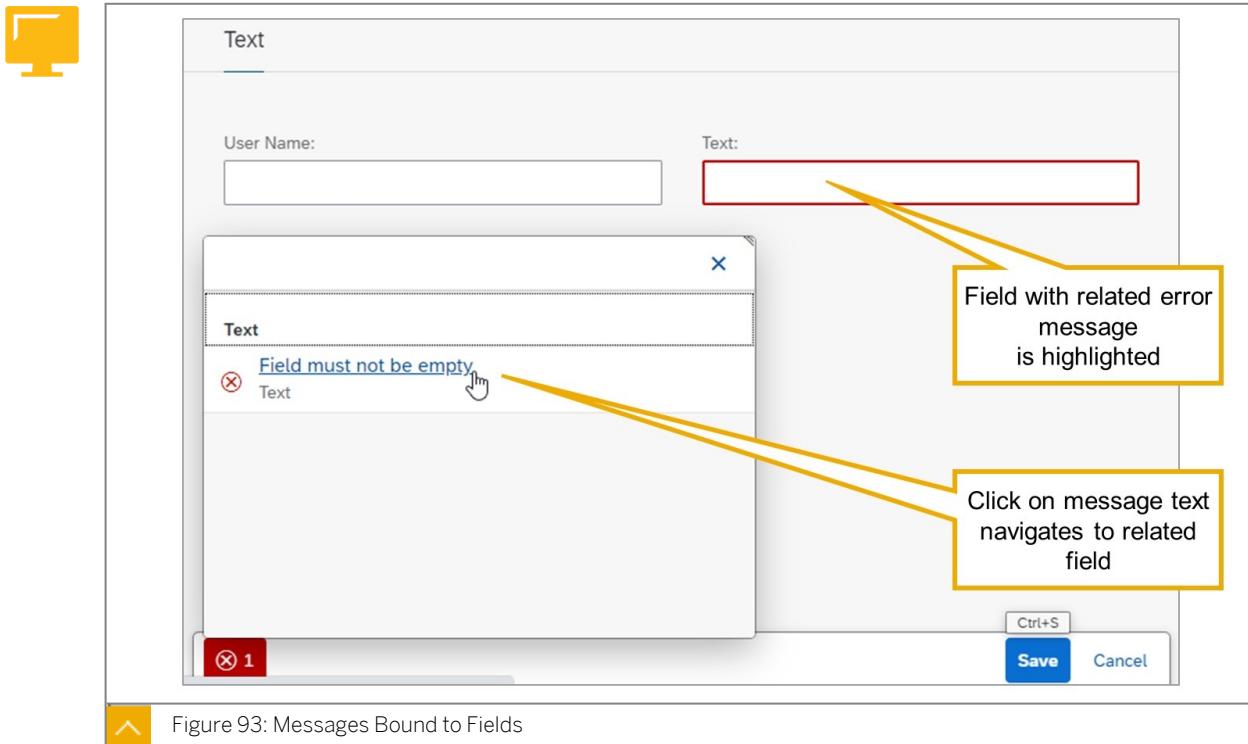
Figure 92: Implementing the Validation Handler Method

The implementation of a validation is contained in a local handler class as part of the behavior pool. This local class inherits from the base handler class `CL_ABAP_BEHAVIOR_HANDLER`.

The signature of a validation method is typed using the keyword `FOR VALIDATE ON SAVE` followed by the importing parameter. The type of the importing parameter is an internal table containing the keys of the instances the validation will be executed on.

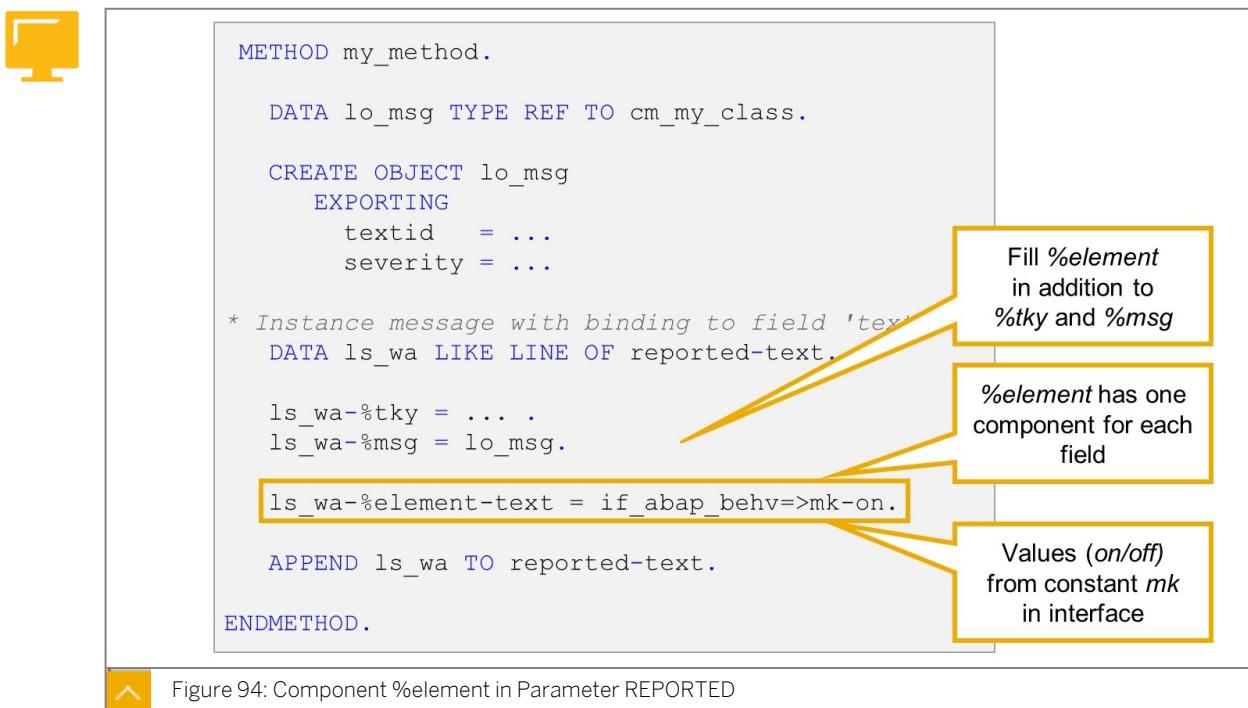
Although not visible in the method definition, all validation handler methods have response parameters `failed` and `reported`. These parameters are deep structures and their types are derived from the definition of the related RAP BO. By adding the key values of an entity instance to the corresponding table in structure `failed`, you reject the instance data from being saved. Additionally, you can return a message to the consumer by passing them to the corresponding table in the `REPORTED` structure.

Validation Messages



In RAP, messages are either related to a RAP BO entity instance or they are returned in the %OTHER component of the REPORTED structure.

For messages related to a RAP BO entities, it is possible to further bind them to one or more fields of the entity. This is particularly helpful for error messages from validations. Binding the messages to fields improves the user experience, because it enables navigation and clear allocation of errors when there are multiple error messages.



To report a message that is related to field `FIELD` of RAP BO entity `ENTITY`, proceed as follows:

1. Create a message object with message text and message severity.
2. Add a new entry to the table-like component `ENTITY` of deep structure `REPORTED`.
3. In the new entry, fill field group `%t_ky` with the entity instance key.
4. Fill component `%msg` with a reference to the message object.
5. If sub-component `FIELD` of component `%element` with `IF_ABAP_BEHV=>MK-ON`.

If you want to bind the message to more than one field, repeat the last step.

Unit 3 Exercise 10

Provide Input Checks Through Validations

Business Scenario

In this exercise, you define and implement validations to provide input checks for input fields.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 10: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437C_I_TRAVEL
ABAP Class	BP_D437C_I_TRAVEL

Task 1: Validation of Customer ID

In the behavior definition for your RAP Business Object, define a validation (suggested name: *ValidateCustomer*) that is performed if the user changed the value in input field `CustomerID`. Implement the validation to ensure that the field is not empty and that the entered customer number belongs to an existing customer. Otherwise, issue suitable error messages and add the affected travel to the list of failed instances.

1. In your behavior definition for the CDS view entity (`Z##_I_Travel`), define a new validation (suggested name: *ValidateCustomer*).
2. In the curly brackets after the addition on `on save`, add field `CustomerID` as the trigger for this validation.
3. Activate the behavior definition.
4. Add the validation method to the local handler class.



Hint:

The editor offers a quick fix for this if you place the cursor on the validation name.

5. In the implementation of the method `validatecustomer`, derive the values of field `CustomerID` for the affected flight travel(s).
6. Implement a loop over the retrieved data. If the `CustomerID` field contains the initial value, create an instance of exception class `CM_DEVS4D437` with a suitable text ID and severity `error`.

7. If the customer ID is not initial, implement an existence check for the customer number in the field `CustomerID`.

**Note:**

We recommend that you read from the CDS view that is the target of the value help and not directly from the database table `SCUSTOM`.

8. If no customer exists with this ID, create an instance of exception class `CM_DEVS4D437` with a suitable text ID and with severity `error`.
9. Before the end of the loop, check whether a message object was created. If that is the case, add the current flight travel to the failed travels in the response parameter `failed`.
10. In addition, add the message and the key of the current flight travel to response parameter `reported`. Make sure that the message is linked to input field `CustomerID`.

**Hint:**

Set the component `CustomerID` of component `%element` to a component of the constant structure `if_abap_behv=>mk`.

11. Activate the behavior pool and test the validation in your SAP Fiori elements app.

Task 2: Validation of Dates

In the behavior definition for your RAP Business Object, define a validation (suggested name: `ValidateStartDate`) that is triggered if the value of the input field `StartDate` is changed. Implement the validation to ensure that the field is not empty and that it lies on or after the current system date. In case of errors, issue suitable error messages and add the affected travel to the list of failed instances. Define and implement a similar validation for the field `EndDate` (suggested name: `ValidateEndDate`). Finally, define and implement a validation that is triggered by a change of either start date or end date and that ensures that the end date lies after the start date (suggested name: `ValidateDateSequence`).

1. In your behavior definition for the data model view (`Z##_I_Travel`), define a new validation (suggested name: `ValidateStartDate`) that is triggered by changes in the `StartDate` field.
2. Activate the behavior definition.
3. Use the quick fix to add the validation method to the local handler class.
4. In the implementation of the method `validatestartdate`, derive the value of field `StartDate` for the affected flight travel(s).
5. Implement a loop over the retrieved data. If the field `StartDate` contains the initial value, add the current flight travel to the list of failed travels and report it with a suitable error message from exception class `CM_DEVS4D437`.
6. If the start date is not initial and lies in the past, add the current flight travel to the list of failed travels and report it with a suitable error message from exception class `CM_DEVS4D437`.
7. Define another validation (suggested name: `ValidateEndDate`) that is triggered by changes in field `EndDate`.

8. Use the quick fix to add the validation method to the local handler class and implement it in the same way as the previous validation.



Hint:

You can reduce typing effort by copying the implementation from the previous method and replacing `StartDate` with `EndDate`.

9. Define another validation (suggested name: `ValidateSequence`) that is triggered by changes in either of the fields `StartDate` and `EndDate`.
10. Implement this last validation as follows: If neither of the two dates is initial, check that the end date lies after the start date. If it lies on or before the start date, add the current flight travel to the list of failed travels and report it with a suitable error message from exception class `CM_DEVS4D437`. Make sure that the error message is linked to both input fields.
11. Activate the behavior pool and test the validations in your SAP Fiori elements app.

Provide Input Checks Through Validations

Business Scenario

In this exercise, you define and implement validations to provide input checks for input fields.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 10: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437C_I_TRAVEL
ABAP Class	BP_D437C_I_TRAVEL

Task 1: Validation of Customer ID

In the behavior definition for your RAP Business Object, define a validation (suggested name: *ValidateCustomer*) that is performed if the user changed the value in input field *CustomerID*. Implement the validation to ensure that the field is not empty and that the entered customer number belongs to an existing customer. Otherwise, issue suitable error messages and add the affected travel to the list of failed instances.

1. In your behavior definition for the CDS view entity (*Z##_I_Travel*), define a new validation (suggested name: *ValidateCustomer*).
 - a) See the source code extract from the model solution.
2. In the curly brackets after the addition on `on save`, add field *CustomerID* as the trigger for this validation.
 - a) See the source code extract from the model solution.
3. Activate the behavior definition.
 - a) Perform this step as in previous exercises.
4. Add the validation method to the local handler class.



Hint:

The editor offers a quick fix for this if you place the cursor on the validation name.

- a) Place the cursor on the name of the validation.

- b) Open the context menu and choose *Quick Fix* or press **Ctrl + 1**.
 - c) Double-click the proposed quick fix.
5. In the implementation of the method `validatecustomer`, derive the values of field `CustomerID` for the affected flight travel(s).
- a) Use the EML statement `READ ENTITY` or `READ ENTITIES`.
 - b) See the source code extract from the model solution.
6. Implement a loop over the retrieved data. If the `CustomerID` field contains the initial value, create an instance of exception class `CM_DEVS4D437` with a suitable text ID and severity `error`.
- a) See the source code extract from the model solution.
7. If the customer ID is not initial, implement an existence check for the customer number in the field `CustomerID`.



Note:

We recommend that you read from the CDS view that is the target of the value help and not directly from the database table `SCUSTOM`.

- a) See the source code extract from the model solution.
8. If no customer exists with this ID, create an instance of exception class `CM_DEVS4D437` with a suitable text ID and with severity `error`.
- a) See the source code extract from the model solution.
9. Before the end of the loop, check whether a message object was created. If that is the case, add the current flight travel to the failed travels in the response parameter `failed`.
- a) See the source code extract from the model solution.
10. In addition, add the message and the key of the current flight travel to response parameter `reported`. Make sure that the message is linked to input field `CustomerID`.



Hint:

Set the component `CustomerID` of component `%element` to a component of the constant structure `if_abap_behv=>mk`.

- a) See the source code extract from the model solution.
11. Activate the behavior pool and test the validation in your SAP Fiori elements app.
- a) Perform this step as in previous exercises.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition `D437C_I_TRAVEL`:

```
...
define behavior for D437C_I_Travel alias Travel
persistent table d437c_travel
```

```

lock master
etag master ChangedAt
authorization master ( instance )
{
  ...
  validation validateCustomer on save { field CustomerID; }
  ...
}

```

Class BP_D437C_TRAVEL

```

METHOD validatecustomer.

* for message object
  DATA lo_msg TYPE REF TO cm_devs4d437.

* work areas for response parameters
  DATA ls_reported_travel LIKE LINE OF reported-travel.
  DATA ls_failed_travel   LIKE LINE OF failed-travel.

* read required data
***** READ ENTITY IN LOCAL MODE d437c_i_travel
      FIELDS ( customerid ) WITH CORRESPONDING #( keys )
      RESULT DATA(lt_travel).

  LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

* validate data and create message object in case of error
***** IF <ls_travel>-customerid IS INITIAL.
*   "error because of initial input field
  CREATE OBJECT lo_msg
    EXPORTING
      textid    = cm_devs4d437=>field_empty
      severity  = cm_devs4d437=>severity-error.

*   " expression-based alternative
*   lo_msg = new #( textid    = cm_devs4d437=>field_empty
*                  severity  = cm_devs4d437=>severity-error ).
ELSE.
  "existence check for customer
  SELECT SINGLE @abap_true
    FROM d437_i_customer
    INTO @DATA(lv_exists)
    WHERE customer = <ls_travel>-customerid.

  IF lv_exists <> abap_true.
    " error because of non-existent customer
    CREATE OBJECT lo_msg
      EXPORTING
        textid    = cm_devs4d437=>customer_not_exist
        customerid = <ls_travel>-customerid
        severity  = cm_devs4d437=>severity-error.

*   " expression-based alternative
*   lo_msg = new #

```

```

*
*          textid      = cm_devs4d437=>customer_not_exist
*          customerid = <ls_travel>-customerid
*          severity    = cm_devs4d437=>severity-error
*          .
*        ENDIF.
*
*      ENDIF.

* report message and mark flight travel as failed
***** IF lo_msg IS BOUND.

      CLEAR ls_failed_travel.
      MOVE-CORRESPONDING <ls_travel> TO ls_failed_travel.
      APPEND ls_failed_travel TO failed-travel.

      CLEAR ls_reported_travel.
      MOVE-CORRESPONDING <ls_travel> TO ls_reported_travel.
      ls_reported_travel-%element-customerid = if_abap_behv=>mk-on.
      ls_reported_travel-%msg = lo_msg.
      APPEND ls_reported_travel TO reported-travel.

      " expression-based alternative without helper variables
      APPEND CRRESPONDING #( <ls_travel> )
          TO failed-travel.
*
*      APPEND VALUE #(
*          %tky      = <ls_travel>-%ky
*          %element = VALUE #( custmerid = if_abap_behv=>mk-on )
*          %msg     = NEW cm_devs4d437
*                      textid      = cm_devs4d437=>customr_not_exist
*                      customerid = <ls_travel>-cusomerid
*                      sevrtiy   = cm_devs4d437=>sverity-error
*                      )
*          )
*      TO reported-travel.
*
*      CLEAR lo_msg.
*    ENDIF.

    ENDLOOP.

  ENDMETHOD.

```

Task 2: Validation of Dates

In the behavior definition for your RAP Business Object, define a validation (suggested name: *ValidateStartDate*) that is triggered if the value of the input field *StartDate* is changed. Implement the validation to ensure that the field is not empty and that it lies on or after the current system date. In case of errors, issue suitable error messages and add the affected travel to the list of failed instances. Define and implement a similar validation for the field *EndDate* (suggested name: *ValidateEndDate*). Finally, define and implement a validation that is triggered by a change of either start date or end date and that ensures that the end date lies after the start date (suggested name: *ValidateDateSequence*).

1. In your behavior definition for the data model view (*Z##_L_Travel*), define a new validation (suggested name: *ValidateStartDate*) that is triggered by changes in the *StartDate* field.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition.
 - a) Perform this step as in previous exercises.

3. Use the quick fix to add the validation method to the local handler class.
 - a) Perform this step as you did before.
4. In the implementation of the method `validatestartdate`, derive the value of field `StartDate` for the affected flight travel(s).
 - a) Use the EML statement `READ ENTITY`.
 - b) See the source code extract from the model solution.
5. Implement a loop over the retrieved data. If the field `StartDate` contains the initial value, add the current flight travel to the list of failed travels and report it with a suitable error message from exception class `CM_DEVS4D437`.
 - a) See the source code extract from the model solution.
6. If the start date is not initial and lies in the past, add the current flight travel to the list of failed travels and report it with a suitable error message from exception class `CM_DEVS4D437`.
 - a) See the source code extract from the model solution.
7. Define another validation (suggested name: `ValidateEndDate`) that is triggered by changes in field `EndDate`.
 - a) See the source code extract from the model solution.
8. Use the quick fix to add the validation method to the local handler class and implement it in the same way as the previous validation.

**Hint:**

You can reduce typing effort by copying the implementation from the previous method and replacing `StartDate` with `EndDate`.

- a) See the source code extract from the model solution.
9. Define another validation (suggested name: `ValidateSequence`) that is triggered by changes in either of the fields `StartDate` and `EndDate`.
 - a) See the source code extract from the model solution.
10. Implement this last validation as follows: If neither of the two dates is initial, check that the end date lies after the start date. If it lies on or before the start date, add the current flight travel to the list of failed travels and report it with a suitable error message from exception class `CM_DEVS4D437`. Make sure that the error message is linked to both input fields.
11. Activate the behavior pool and test the validations in your SAP Fiori elements app.
 - a) Perform this step as in previous exercises.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition `D437C_I_TRAVEL`:

```
...
define behavior for D437c_I_Travel alias Travel
persistent table d437c_travel
lock master
```

```

etag master ChangedAt
authorization master ( instance )
{
  ...
  validation validateStartDate on save { field StartDate; }
  validation validateEndDate   on save { field EndDate; }
  validation validateSequence  on save { field StartDate, EndDate; }
  ...
}

```

Class BP_D437C_I_TRAVEL:

```

METHOD validatestartdate.

READ ENTITY IN LOCAL MODE d437c_i_travel
FIELDS ( startdate ) WITH CORRESPONDING #( keys )
RESULT DATA(lt_travel).

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

  "Start Date
  -----
  IF <ls_travel>-startdate IS INITIAL.

    APPEND CORRESPONDING #( <ls_travel> )
      TO failed-travel.
    APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element  = VALUE #( startdate = if_abap_behv=>mk-on )
      %msg      = NEW cm_devs4d437(
        textid   = cm_devs4d437=>field_empty
        severity = cm_devs4d437=>severity-error
      )
    )
    TO reported-travel.

  ELSEIF <ls_travel>-startdate < sy-datum.
    " or use cl_abap_context_info=>get_system_date( )

    APPEND CORRESPONDING #( <ls_travel> )
      TO failed-travel.
    APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element  = VALUE #( startdate = if_abap_behv=>mk-on )
      %msg      = NEW cm_devs4d437(
        textid   = cm_devs4d437=>start_date_past
        severity = cm_devs4d437=>severity-error
      )
    )
    TO reported-travel.

  ENDIF.

ENDLOOP.
ENDMETHOD.

METHOD validateenddate.


```

```

READ ENTITY IN LOCAL MODE d437c_i_travel
FIELDS ( enddate ) WITH CORRESPONDING #( keys )
RESULT DATA(lt_travel).

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

  "End Date
  -----
  IF <ls_travel>-enddate IS INITIAL.

    APPEND CORRESPONDING #( <ls_travel> )
      TO failed-travel.
    APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element = VALUE #( enddate = if_abap_behv=>mk-on )
      %msg      = NEW cm_devs4d437(
        textid   = cm_devs4d437=>field_empty
        severity = cm_devs4d437=>severity-error
      )
    )
    TO reported-travel.

  ELSEIF <ls_travel>-enddate < sy-datum.
    " or use cl_abap_context_info=>get_system_date( )

    APPEND CORRESPONDING #( <ls_travel> )
      TO failed-travel.
    APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element = VALUE #( enddate = if_abap_behv=>mk-on )
      %msg      = NEW cm_devs4d437(
        textid   = cm_devs4d437=>end_date_past
        severity = cm_devs4d437=>severity-error
      )
    )
    TO reported-travel.

  ENDIF.

ENDLOOP.
ENDMETHOD.
```

```

METHOD validatesequence.

READ ENTITY IN LOCAL MODE d437c_i_travel
FIELDS ( startdate enddate ) WITH CORRESPONDING #( keys )
RESULT DATA(lt_travel).

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

  "Sequence of Dates
  -----
  IF <ls_travel>-startdate IS INITIAL
  OR <ls_travel>-enddate IS INITIAL.
    " ignore empty fields, already covered above
  ELSEIF <ls_travel>-enddate < <ls_travel>-startdate.

    APPEND CORRESPONDING #( <ls_travel> )
      TO failed-travel.
    APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element = VALUE #( startdate = if_abap_behv=>mk-on
```

```
        enddate = if_abap_behv=>mk-on )
%msg      = NEW cm_devs4d437(
                textid = cm_devs4d437=>dates_wrong_sequence
                severity = cm_devs4d437=>severity-error
            )
TO reported-travel.

ENDIF.

ENDLOOP.
ENDMETHOD.
```



LESSON SUMMARY

You should now be able to:

- Explain validations
- Define and implement input checks
- Link messages to input fields

Providing Values with Determinations



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe the numbering concepts in RAP
- Define and implement determinations

Standard Operation CREATE



Behavior Definition

```
managed implementation in class zbp_00_i_text unique;

define behavior for Z00_I_Text alias Text
  persistent table d437_text
  lock master
  authorization master ( instance )

{
  ...
  create;
  ...
}
```

Enable standard Operation Create for this RAP BO entity

Figure 95: Standard Operation CREATE

To enable standard operation `create` for an entity, add the statement `create;` to the entity behavior body (curly brackets after statement `define behavior for ...`). To disable the operation, remove the statement or turn it into a comment.



Note:

In a managed implementation scenario, `create` can only be declared for root entities. Child entities are implicitly create-enabled for internal usage. That means, an external consumer can only create a new instance of a child entity via its parent (create-by-association operation). In an unmanaged implementation scenario, direct creates on child entities are possible but not recommended.

Interesting additions to the `create` statement are as follows:

- Internal

Prefix which disables the operation for external consumers of the BO.

- features: global

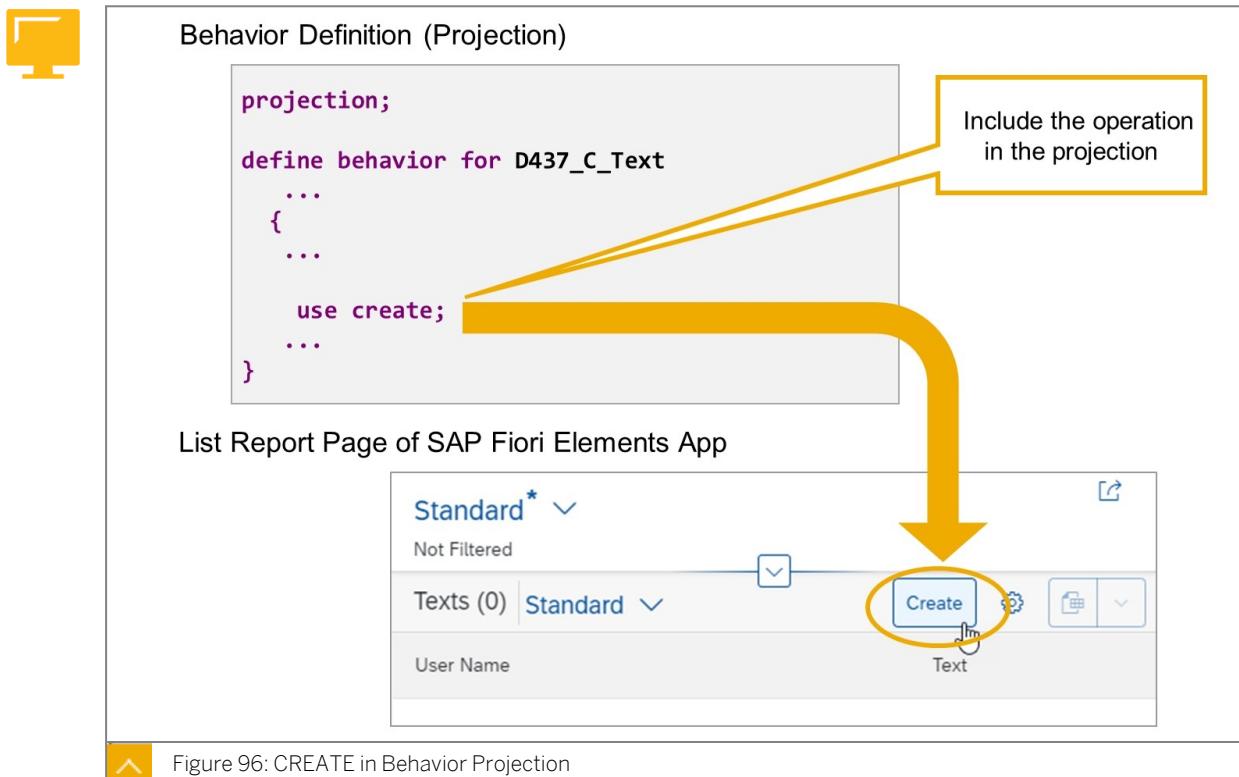
Enable dynamic feature control. The decision, when the operation is enabled, is made dynamically in a behavior implementation.

- precheck

A method is called before the create request to prevent unwanted changes from reaching the application buffer.

- authorization: none

Excludes the create operation from the global authorization checks.



To make standard operation `create` available in OData and SAP Fiori, it has to be reused in the behavior definition for the projection view (behavior projection).

To include standard operation `create` in the OData service, add the statement `use create;` to the entity behavior body (curly brackets after statement `define behavior for ...`). To disable the operation, remove the statement or turn it into a comment.

Note:

Behavior projections can have their own implementations, which can be used to augment the implementation of a standard operation or provide additional prechecks. But this is a special case that we will not cover in this class.

As soon as standard operation `create` is available in the OData service, the generated SAP Fiori elements app displays a *Create* button on the *List Report* page for the related RAP BO entity. By choosing this button, the user navigates to an *Object* page for a new entity instance.

**Hint:**

After editing the behavior projection, you might have to perform a hard refresh of the service preview (Ctrl + F5) before you see the new button. Sometimes, you have to clear your browsing data (Ctrl + Shift + Del) too.

Numbering



▪ Primary Key (in general)

- One or more fields
- Uniquely identifies entity instance
- Read-only in *UPDATE* operation
- Receive values during *CREATE* operation

▪ Primary Key (in RAP BOs)

- Defined in underlying CDS view (keyword **key**)
- Listed after **field (read-only)** or **field (read-only:update)**
- Various techniques for setting key values

Numbering: Providing values for primary key during *Create* operation.



Figure 97: Primary Keys in RAP

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 identifies each instance of a business object entity.

The logic of the business object has to ensure that all entity instances are created with a unique set of primary key values and that the primary key values of an existing instance cannot be changed during modify operation update.

The editing is easily prevented by listing the key fields after statement **field (read-only)** in the behavior definition body. To only prevent editing during the operation **update** while still allowing the consumer to provide values during operation **create**, the keyword **field (read-only: update)** is used instead.

The process of providing a unique key during the creation of a new instance is called numbering.

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:

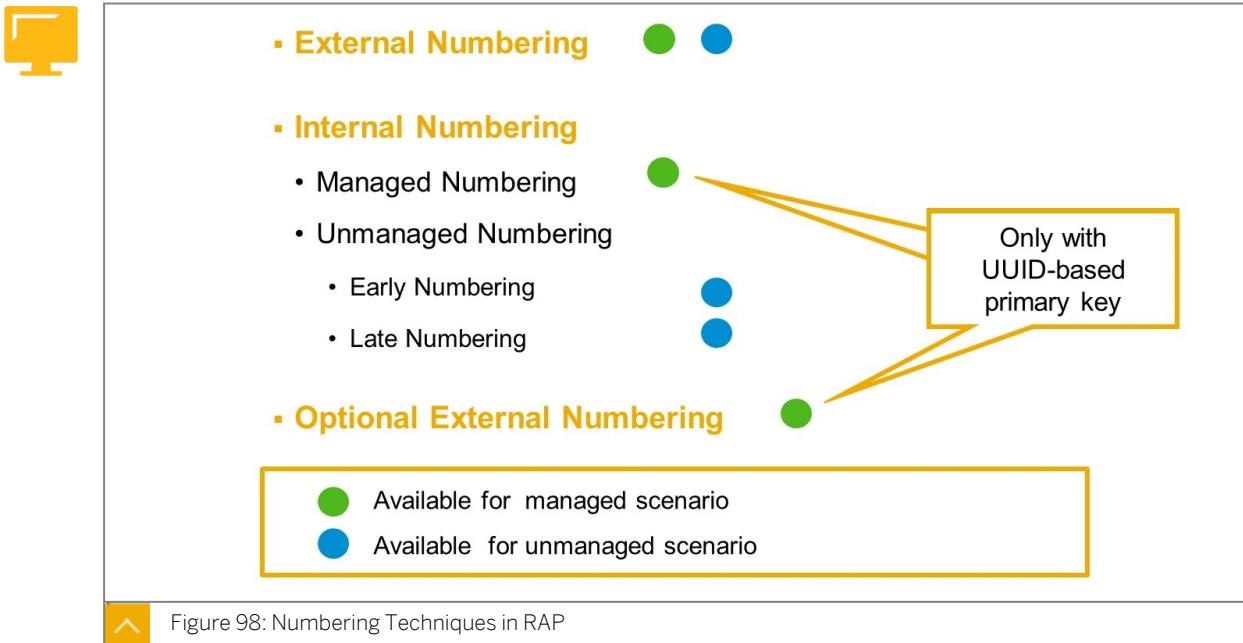


Figure 98: Numbering Techniques in RAP

There are various options to handle the numbering for primary key fields depending on when and by whom the primary key values are set. You can assign a numbering type for each primary key field separately. Note the following distinctions:

External vs Internal

In External Numbering, 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, the behavior implementation has to ensure that the primary key value given by the consumer is uniquely identifiable. This is opposed to Internal Numbering where the key values are provided by the RAP BO logic. Optional External Numbering is a combination of external and internal numbering: The RAP BO logic only provides key values in case the consumer hands over initial values.

Managed vs Unmanaged

Internal numbering can either be managed or unmanaged. In Managed Numbering, the unique key is drawn automatically during the CREATE request by the RAP managed runtime. This is opposed to Unmanaged Numbering, where the key values are provided in a dedicated handler method, implemented by the application developer.

Early vs Late

Unmanaged internal numbering can be either early or late. In Early Numbering, the final key value is available in the transactional buffer instantly after the MODIFY request for CREATE. This is opposed to Late Numbering, where the final number is only assigned just before the instance is saved on the database. 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.

The following restrictions apply:

- Managed Numbering is only possible for key fields with ABAP type raw(16) (UUID) of BOs with implementation type managed.
- Optional External Numbering is only possible in combination with managed numbering

- Unmanaged Numbering is currently only possible in unmanaged BOs.

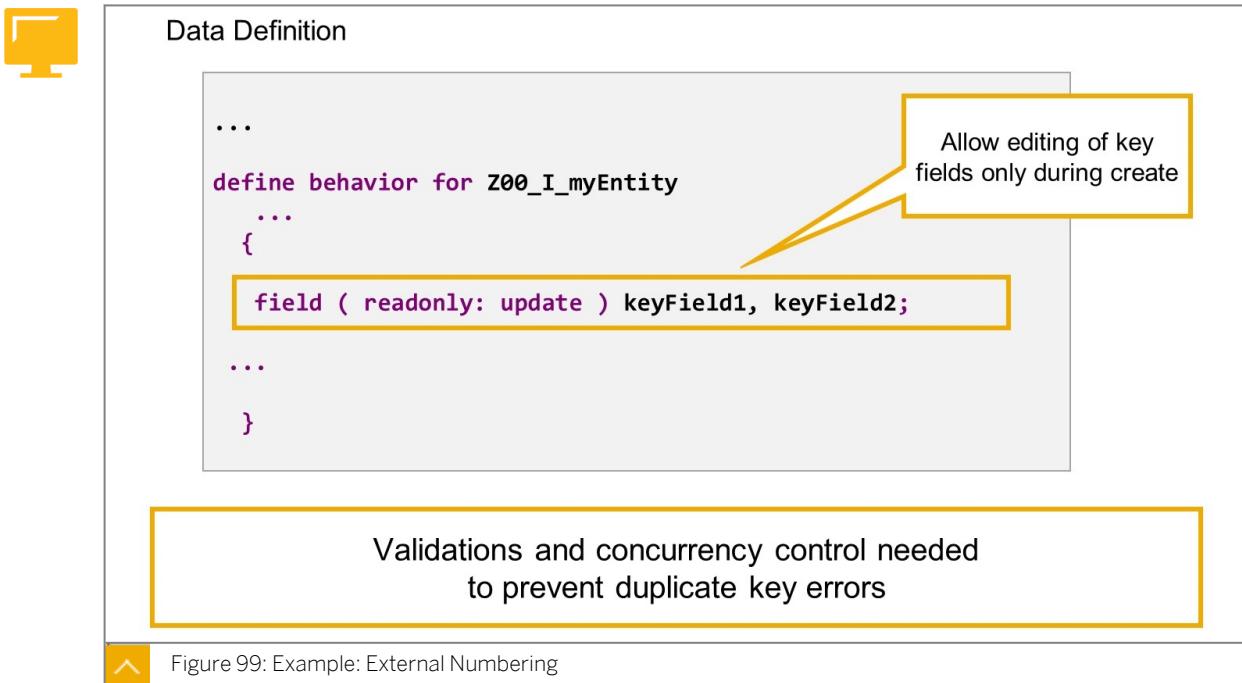


Figure 99: Example: External Numbering

In External Numbering, the consumer hands over the primary key values for the CREATE operation, just like any other values for non-key fields.

To ensure that the consumer can edit the primary key during create operations, but not during update operations, the primary key fields should be listed in the entity behavior body after the keyword `readonly: update`.

In addition, validations and pessimistic concurrency control should be used to avoid duplicate key errors during the save phase.

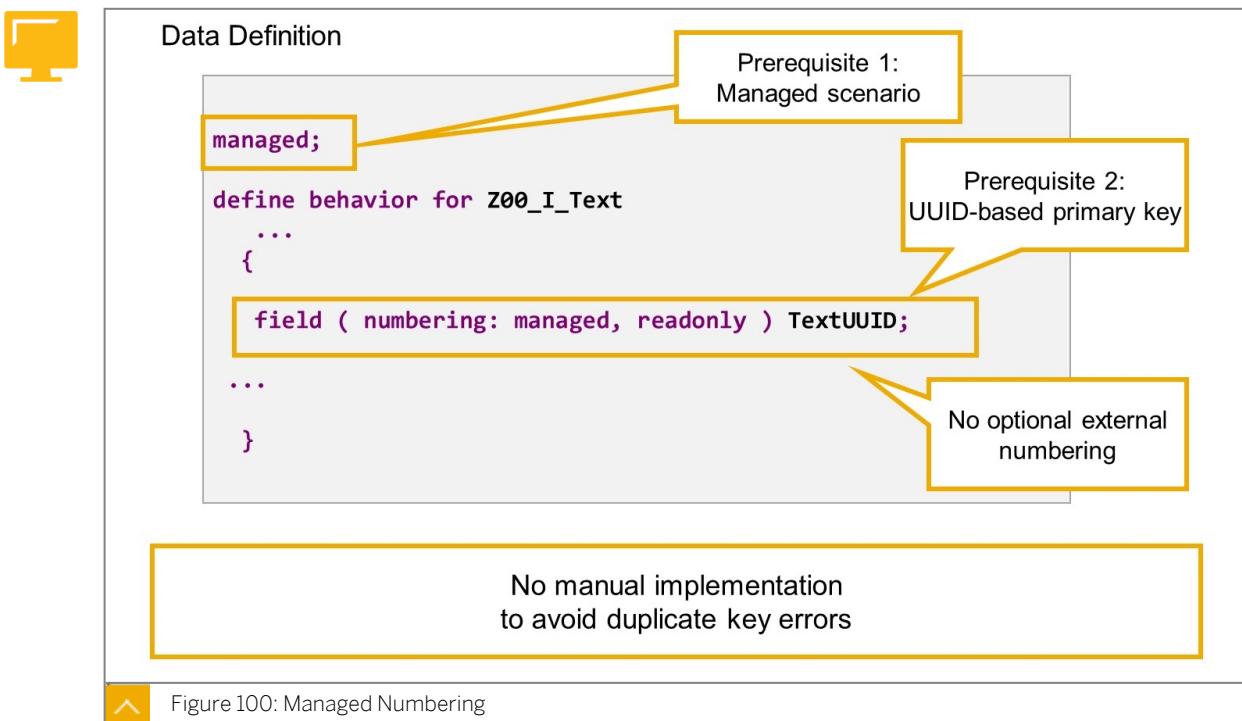


Figure 100: Managed Numbering

To enable managed internal numbering, you have to list the key field after the keyword field (numbering: managed) in the entity behavior body. The field in question is automatically assigned values on creation of a new entity instance. No implementation in the ABAP behavior pool is required.

The following restrictions apply:

- Only for primary key fields with ABAP type raw(16) (UUID).
- Only in managed implementation scenario



Note:

It is recommended, but not necessary, to define the key field as `readonly: update` or `readonly` to make sure the key of an existing instance cannot be changed in update operations. If the field is defined as `readonly: update`, the key value can also be given by the consumer (Optional External Numbering).

Determination Definition



Determinations in RAP

- Part of business object behavior
 - Implemented in local handler class
 - Executed based on trigger conditions and trigger time
- **Trigger Conditions**
 - Modify operations or modified fields
 - **Trigger Time**
 - `on modify` or `on save`
 - **Reaction**
 - Compute data
 - Modify entity instance
 - Return messages
 - **Restriction**
 - Not available for unmanaged, non-draft scenarios



Figure 101: Determinations in RAP

A determination is an optional part of the business object behavior that modifies instances of business objects based on trigger conditions. Determinations, like actions and validations, are defined in the behavior definition of the RAP BO and implemented in the behavior pool through a dedicated method of the local handler class.

A determination is implicitly invoked by the business objects framework if the trigger condition of the determination is fulfilled. Trigger conditions can be modify operations (create, update, delete) and modified fields. The trigger condition is evaluated at the trigger time, a predefined point during the BO runtime. Two types of determinations are distinguished, depending on the stage of the program flow the determination is executed: `on modify` determinations and `on save` determinations.

An invoked determination can compute data, modify entity instances according to the computation result and return messages to the consumer by passing them to the corresponding table in the REPORTED structure.



Note:

Determinations are available for managed scenarios and for unmanaged scenarios with draft. They are not available for unmanaged, non-draft scenarios.



Behavior Definition

```
managed;

define behavior for D437_I_Text
  ...
  {
    determination set_owner on modify { create; }
  }
```

For determinations
on save and on modify
is supported

Only triggered for
new instances



Figure 102: Example: Determination Definition

Determinations are defined in the entity behavior definition with the following statement:

`determination <determination_name> <trigger time>`

`{ <trigger_conditions> }.`

For determinations, the following trigger times are available:

on modify

The determination is executed immediately after data changes take place in the transactional buffer so that the result is available during the transaction.

on save

The determination is executed during the save sequence at the end of a transaction, when changes from the transactional buffer are persistent on the database.



Note:

For determinations, two trigger times are available. Validations are only available with trigger time `on save`.

It is mandatory to provide at least one trigger condition within the curly brackets.

The following trigger conditions are supported:

Create;

Determination is executed when an instance is created.

Update;

Determination is executed when an instance is updated.

Delete;

Determination is executed when an instance is deleted.

Field <field1>, <field2>, ...;

Determination is executed when the value of one of the specified fields is changed by a create or update operation.

Multiple trigger conditions can be combined.



Note:

For determinations defined as `on save`, trigger condition `update`; works only in combination with the trigger condition `create`;

The behavior definition in the example defines one determination. It is executed during the modify phase, and it is triggered by the `create`; operation alone. It is not triggered during `update`; operations on existing entity instances.



Note:

The execution order of determinations is not fixed. If there is more than one determination triggered by the same condition, you cannot know which determination is executed first.

Determination Implementation



```

1 managed implementation in class zbp_00_i_text unique;
2
3@define behavior for Z00_I_TEXT alias text
4 persistent table d437_text04
5 lock master
6
7 authorization master( instance )
8 {
9
10
11   determination set_owner on modify { create; }
12
13@ validation textNot
14
15

```

Quick fix to add implementation method for new determination

Add missing method for determination set_owner in local handler class lhc_te

Invoking Quickfix:

- Right-click determination name and choose *Quick Fix* or
- Click determination name and press *Ctrl + 1*



Figure 103: Creating the Determination Handler Method

If the behavior definition already contains the determination definition, the quick fix for creating the behavior pool will automatically create the determination implementation method in the local handler class.

If the behavior pool already exists when you add the determination definition, you can use a quick fix to add the missing method to the local handler class. To invoke the quick fix, place the cursor on the name of the determination and press Ctrl + 1.



Note:

Depending on the trigger time specified in the behavior definition, the method definition is generated with the addition FOR DETERMINE ON MODIFY or with the addition FOR DETERMINE ON SAVE. If you change the trigger time later, there will be a syntax error in the behavior pool, but no error or warning in the behavior definition. To fix the syntax error, you have to navigate to the method definition and adjust it manually.



```

CLASS lhc_text DEFINITION
  INHERITING FROM cl_abap_behavior_handler.

PRIVATE SECTION.

  METHODS set_owner FOR DETERMINE ON MODIFY
    IMPORTING keys FOR text~set_owner.

ENDCLASS.

CLASS lhc_text IMPLEMENTATION.

  METHOD set_owner.
    ...
    IF ...
      MODIFY ENTITY IN LOCAL MODE ...
        UPDATE FIELDS ( ... )
        WITH ...
      APPEND ... TO reported-text.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

```

Depends on trigger time in definition

Keys of affected entity instances

Use EML to make changes to node instances

Optional: Return messages



Figure 104: Implementing the Determination Handler Method

The implementation of a determination is contained in a local handler class as part of the behavior pool. This local class inherits from the base handler class CL_ABAP_BEHAVIOR_HANDLER.

The signature of a determination method is typed using the keyword FOR DETERMINE followed by the chosen determination time and the import parameter. The type of the importing parameter is an internal table containing the keys of the instances the determination will be executed on.

Although not visible in the method definition, all determination handler methods have a response parameter reported which allows you to report messages in the determination implementation.

The actual changes to the node instances are performed using the EML statement `MODIFY ENTITY` or `MODIFY ENTITIES`, based on the keys in importing parameter keys.

Unit 3 Exercise 11

Enable Managed Numbering and Implement Determinations

Business Scenario

In this exercise, you enable the creation of flight travels. You activate managed internal numbering to make the RAP framework provide a unique value for the technical key field of each new flight travel. To provide consistent values for the semantic key, you define and implement a determination.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 11: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D435C_I_TRAVEL
CDS Behavior Definition (Projection)	D435C_C_TRAVEL
ABAP Class	BP_D437C_I_TRAVEL

Task 1: Enable the Creation of Data

Enable the creation of flight travels in your transactional SAP Fiori elements app. To achieve this, make sure that the behavior definition and behavior projection contain the necessary statements.

1. Open the behavior definition for the data model view (*Z##_I_Travel*) and make sure that it contains the statement `create;`. If it is missing, add it. If it is commented out, uncomment it.
2. Activate the behavior definition.
3. Open the behavior projection, that is, the behavior definition for the projection view (*Z##_C_Travel*) and make sure it contains the statement `use create;`. If it is commented out, uncomment it. If it is missing, add it.
4. Activate the behavior definition.
5. Test your SAP Fiori elements app. Make sure that there is a *Create* button on the *Report List* page.
6. Create a new flight travel.

Why are the checks on customer ID and travel dates not executed?

What do you have to do to make sure the validations are triggered during the `create` operation as well?

7. Adjust the definitions of the validations so they are executed during the `create` operation.
8. Use your SAP Fiori elements app again and create another new flight travel. This time enter valid values in the mandatory fields.
9. Use the *Data Preview* tool in Eclipse to analyze the new data set.

Are there any fields with initial values? Which fields?

Why are fields `ChangedAt` and `ChangedBy` not initial?

Task 2: Enable Managed Internal Numbering

Make sure the RAP run time automatically provides a unique value for key field `Trguid` when the user creates a new flight travel.

1. Edit the behavior definition of your data model view. Add a `field` statement followed by the view element `BP_D437C_I_TRAVEL`. Inside the brackets, add the required additions to make the field read-only and to enable managed internal numbering.
2. Activate the behavior definition.
3. Test your SAP Fiori elements app and create a new flight travel.
4. Use the *Data Preview* tool in Eclipse again to confirm that the new dataset contains a non-unique value in the `Trguid` field.

Task 3: Determine Values for the Semantic Key Fields

Create and implement a determination for fields `AgencyID` and `TravelID`. Make sure the determination is only triggered during creation of a new travel. Use suitable methods of class `CL_S4D437_MODEL` to derive the values.

1. In your behavior definition for the data model view (`Z##_I_Travel`), define a new determination (suggested name: `determineSemanticKey`).

2. In the curly brackets after addition on `modify`, add `create`; as the trigger for this determination.
3. Activate the behavior definition.
4. Add the determination method to the local handler class.



Hint:

The editor offers a quick fix for this if you place the cursor on the determination name.

5. In the implementation of method `determineSemanticKey`, implement an update of fields `AgencyID` and `TravelID` for all affected travels.
6. Supply the `MODIFY ENTITY` with an internal table that contains the same keys as import parameter `keys`.
7. Call method `get_agency_by_user()` of class `CL_S4D437_MODEL` to get the new value for field `AgencyID` based on the current user.



Hint:

You do not have to call this method for each affected flight travel.

8. For each affected flight travel, call method `get_next_travelid_for_agency()` of class `CL_S4D437_MODEL` to retrieve a new flight travel number.



Note:

This method call simulates reading a new number from a number range object.

9. Finally, copy the response of statement `MODIFY ENTITY` into the response parameter `reported` of the determination method.
10. Activate the behavior pool and test the determination in your SAP Fiori elements app.

Enable Managed Numbering and Implement Determinations

Business Scenario

In this exercise, you enable the creation of flight travels. You activate managed internal numbering to make the RAP framework provide a unique value for the technical key field of each new flight travel. To provide consistent values for the semantic key, you define and implement a determination.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 11: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D435C_I_TRAVEL
CDS Behavior Definition (Projection)	D435C_C_TRAVEL
ABAP Class	BP_D437C_I_TRAVEL

Task 1: Enable the Creation of Data

Enable the creation of flight travels in your transactional SAP Fiori elements app. To achieve this, make sure that the behavior definition and behavior projection contain the necessary statements.

1. Open the behavior definition for the data model view (`Z##_I_Travel`) and make sure that it contains the statement `create;`. If it is missing, add it. If it is commented out, uncomment it.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition.
 - a) Choose *Activate* or press *Ctrl + F3*.
3. Open the behavior projection, that is, the behavior definition for the projection view (`Z##_C_Travel`) and make sure it contains the statement `use create;`. If it is commented out, uncomment it. If it is missing, add it.
 - a) See the source code extract from the model solution.
4. Activate the behavior definition.
 - a) Choose *Activate* or press *Ctrl + F3*.

5. Test your SAP Fiori elements app. Make sure that there is a *Create* button on the *Report List* page.
 - a) Perform this step as in previous exercises.
6. Create a new flight travel.
 - a) Open the preview of your Service Binding.
 - b) Choose *Create*.
 - c) Leave all fields empty and choose *Save*.

Why are the checks on customer ID and travel dates not executed?

At present, the validations are only triggered if the values in the related fields change.

What do you have to do to make sure the validations are triggered during the `create` operation as well?

You have to change the trigger conditions for the validations.

7. Adjust the definitions of the validations so they are executed during the `create` operation.
 - a) Add `create;` inside the curly brackets.
 - b) See the source code extract from the model solution.
8. Use your SAP Fiori elements app again and create another new flight travel. This time enter valid values in the mandatory fields.
 - a) Perform this step as in previous exercises.
9. Use the *Data Preview* tool in Eclipse to analyze the new data set.

Are there any fields with initial values? Which fields?

The technical key field (`Trguid`), the semantic key fields (`AgencyID` and `TravelID`, and the status field `Status` have initial values.

Why are fields `ChangedAt` and `ChangedBy` not initial?

The fields are populated by the RAP run time because in the data model view they are annotated with the corresponding sub-annotations of the annotation `Semantics`.

- a) Right-click anywhere in the definition of any of your CDS views and choose *Open With → Data Preview*.
- b) Compare your code to the following extract from the model solution.

Behavior Definition D437C_I_TRAVEL:

```
...
define behavior for D437c_I_Travel alias Travel
persistent table d437c_travel
lock master
etag master ChangedAt
authorization master ( instance )
{
    ...
    ...
    create;
    update;
    ...
    validation validateCustomer on save { create;
        field CustomerID;
    }
    validation validateStartDate on save { create;
        field StartDate;
    }
    validation validateEndDate on save { create;
        field EndDate;
    }
    validation validateSequence on save { create;
        field StartDate, EndDate;
    }
}
```

Behavior Definition D437C_C_TRAVEL:

```
...
define behavior for D437c_C_Travel alias Travel
use etag
{
    use create;
    use update;
    ...
}
```

Task 2: Enable Managed Internal Numbering

Make sure the RAP run time automatically provides a unique value for key field Trguid when the user creates a new flight travel.

1. Edit the behavior definition of your data model view. Add a `field` statement followed by the view element `BP_D437C_I_TRAVEL`. Inside the brackets, add the required additions to make the field read-only and to enable managed internal numbering.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition.
 - a) Choose `Activate` or press `Ctrl + F3`.
3. Test your SAP Fiori elements app and create a new flight travel.
 - a) Perform this step as before.

4. Use the *Data Preview* tool in Eclipse again to confirm that the new dataset contains a non-unique value in the `Trguid` field.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition **D437C_I_TRAVEL**:

```

...
define behavior for D437c_I_Travel alias Travel
persistent table d437c_travel
lock master
etag master ChangedAt
authorization master ( instance )
{
  ...
  field (readonly, numbering: managed ) Trguid;
  ...
}

```

Task 3: Determine Values for the Semantic Key Fields

Create and implement a determination for fields `AgencyID` and `TravelID`. Make sure the determination is only triggered during creation of a new travel. Use suitable methods of class `CL_S4D437_MODEL` to derive the values.

1. In your behavior definition for the data model view (`Z##_I_Travel`), define a new determination (suggested name: `determineSemanticKey`).
 - a) See source code extract from the model solution.
2. In the curly brackets after addition on `modify`, add `create;` as the trigger for this determination.
 - a) See the source code extract from the model solution.
3. Activate the behavior definition.
 - a) Perform this step as in previous exercises.
4. Add the determination method to the local handler class.



Hint:

The editor offers a quick fix for this if you place the cursor on the determination name.

- a) Place the cursor on the name of the determination.
- b) Open the context menu and choose *Quick Fix* or press `Ctrl + 1`.
- c) Double-click the proposed quick fix.
5. In the implementation of method `determineSemanticKey`, implement an update of fields `AgencyID` and `TravelID` for all affected travels.

- a) Use the EML statement `MODIFY ENTITY` with the addition `UPDATE FIELDS (agencyid travelid)`.
 - b) See the source code extract from the model solution.
6. Supply the `MODIFY ENTITY` with an internal table that contains the same keys as import parameter `keys`.
- a) Define an internal table with a suitable type to place after addition `WITH`.



Note:

Alternatively, you can use a `VALUE` expression to provide the input for the `MODIFY ENTITY` statement.

- b) Fill the internal table with the keys from import parameter `keys`.
 - c) See the source code extract from the model solution.
7. Call method `get_agency_by_user()` of class `CL_S4D437_MODEL` to get the new value for field `AgencyID` based on the current user.



Hint:

You do not have to call this method for each affected flight travel.

8. For each affected flight travel, call method `get_next_travelid_for_agency()` of class `CL_S4D437_MODEL` to retrieve a new flight travel number.



Note:

This method call simulates reading a new number from a number range object.

9. Finally, copy the response of statement `MODIFY ENTITY` into the response parameter `reported` of the determination method.
- a) See the source code extract from the model solution.

10. Activate the behavior pool and test the determination in your SAP Fiori elements app.

- a) Perform this step as before.

- b) Compare your code to the following extract from the model solution.

Behavior Definition `D437C_I_TRAVEL`:

```
...
define behavior for D437C_I_Travel alias Travel
persistent table d437c_travel
lock master
etag master ChangedAt
authorization master ( instance )
{
```

```

...
determination determineSemanticKey on modify { create; }

...
}

```

Class BP_D437C_TRAVEL:

```

METHOD determinesemantickey.

DATA lt_travel_upd TYPE TABLE FOR UPDATE d437c_i_travel.

* get AgencyID for all new travels
*****+
DATA(lv_agencyid) =
    cl_s4d437_model=>get_agency_by_user(
*        iv_user = SY-UNAME
*        iv_user = cl_abap_context_info=>get_user_technical_name( )
*        ).

* prepare input for MODIFY ENTITY
*****+
lt_travel_upd = CORRESPONDING #( keys ).

LOOP AT lt_travel_upd ASSIGNING FIELD-SYMBOL(<ls_travel_upd>).

    <ls_travel_upd>-agencyid = lv_agencyid.
    <ls_travel_upd>-travelid =
        cl_s4d437_model=>get_next_travelid_for_agency(
            iv_agencynum = lv_agencyid
        ).
ENDLOOP.

* Update entities
*****+
MODIFY ENTITY IN LOCAL MODE d437c_i_travel
    UPDATE FIELDS ( agencyid travelid )
    WITH lt_travel_upd
    REPORTED DATA(ls_reported).

MOVE-CORRESPONDING ls_reported-travel
    TO      reported-travel.

ENDMETHOD.

```

Alternative coding with even more expression-based syntax:

```

METHOD determinesemantickey.

get AgencyID for all new travels
DATA(lv_agencyid) =
    cl_s4d437_model=>get_agency_by_user( ).

MODIFY ENTITY IN LOCAL MODE d437c_i_travel
    UPDATE FIELDS ( agencyid travelid )
    WITH VALUE #( FOR key IN keys
        (
            %tky = key-%tky
            agencyid = lv_agencyid

```

```
    travelid =
        cl_s4d437_model=>get_next_travelid_for_agency(
            iv_agencynum = lv_agencyid
        )
    )
)
REPORTED DATA(ls_reported).

reported = CORRESPONDING #( DEEP ls_reported ).

ENDMETHOD.
```



LESSON SUMMARY

You should now be able to:

- Describe the numbering concepts in RAP
- Define and implement determinations

Implementing Dynamic Feature Control



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain dynamic action, operation, and field control in RAP
- Implement dynamic feature control

Action, Operation, and Field Control



▪ Action Control

- Enable/disable execution of actions
- dependent on instance

▪ Operation Control

- Enable/disable standard operations Create, Update, Delete
- dependent on instance

▪ Field Control

- Set fields as read-only or mandatory
- Static or dynamic



Figure 105: Dynamic Feature Control in RAP

With feature control, you can add information to the service on how data has to be displayed for consumption in an SAP Fiori UI. Feature control can relate to actions (action control), standard operations (operation control), and fields (field control).

Action Control is the dynamic enabling and disabling of actions. The decision can either depend on the data of the affected instance (instance feature control) or not (global feature control). For example, it should not be possible to cancel an already delivered order.

Operation Control means the enabling and disabling of standard operations create, update, and delete on node instances. In the case of global feature control, the respective operation is dynamically enabled or disabled for all instances of the entity, independent of its data. In the case of instance feature control, the availability of an operation depends on the data. For example, it should not be possible to edit an order or to add new items to it once the order is already being processed.

Field Control means the classification of node attributes as read-only or mandatory. In the case of static field control, the property is the same for all instances of the node, independent of its data or the change operation. In the case of dynamic field control, the property depends on the data or the operation. For example, a certain field could be mandatory when creating a new instance but should be read-only afterwards.



Note:

Global feature control is not yet supported in ABAP release 7.55.

Feature Control Definition



Behavior Definition

```

managed implementation in class ... ;

define behavior for ...
...
{
    create;
    update ( features: instance );
    delete ( features: instance );

    ...
    action ( features : instance ) my_action;
    ...
    field ( features : instance ) field1, field2;
}

```

Instance feature
control not supported
for create operation



Figure 106: Addition FEATURES : INSTANCE

Dynamic feature control on instance level is enabled by adding the option (features: instance) to the respective statement in the behavior definition body. The option is available for statements update, delete, action, and field.



Note:

Instance feature control is not supported for the create operation, as there is no instance information available yet. We will see later that instance feature control is supported for the creation of child entities (Create by association).

Instance Feature Handler Method



```

1 managed implementation in class zbp_00_i_text unique;
2
3 define behavior for Z00_I_TEXT alias text
4 persistent table d437_text
5 lock master
6
7 authorization master( instance )
8 {
9
10
11 update (features: instance);
12
13 create;
14 delete;

```

Quick fix to add implementation method for feature control

Add missing method for feature_control in local handler class lhc_text

Invoking Quickfix:

- Right-click keyword *feature* and choose *Quick Fix* or
- Click keyword *feature* and press *Ctrl + 1*



Figure 107: Creating the Feature Handler Method

If the behavior definition already contains (*features : instance*) options, the quick fix for creating the behavior pool will automatically create the feature control implementation method in the local handler class.

If the behavior pool already exists when you add the first (*features : instance*) option, you can use a quick fix to add the missing method to the local handler class. To invoke the quick fix, place the cursor on keyword *feature* and press *Ctrl + 1*.



```

CLASS lhc_text DEFINITION
  INHERITING FROM cl_abap_behavior_handler.

  PRIVATE SECTION.

    METHODS get_features FOR FEATURES
      IMPORTING keys
      REQUEST requested_features
      FOR text
      RESULT result.
  ENDCLASS.

  CLASS lhc_text IMPLEMENTATION.

    METHOD get_features.
      ...
    ENDMETHOD.
  ENDCLASS.

```

Keys of affected entity instances

Which operations, actions, fields are requested

What is enabled, mandatory, read-only, and so on.



Figure 108: Implementing the Feature Handler Method

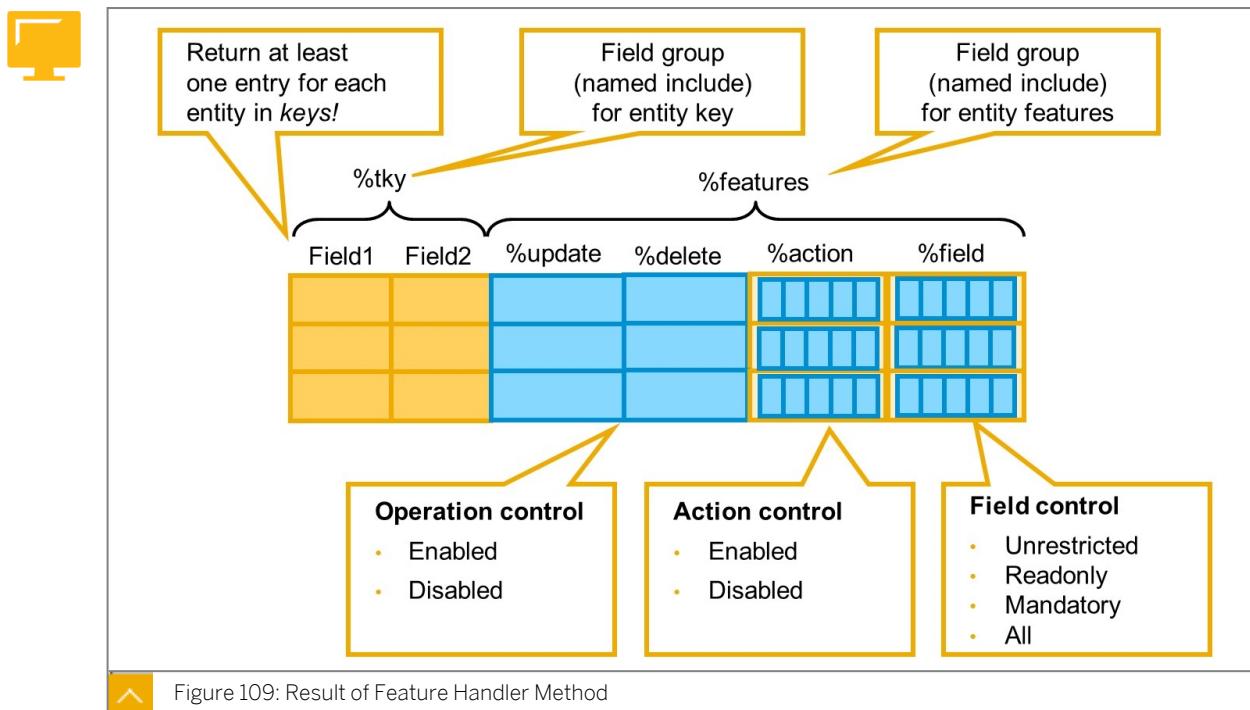
The logic of dynamic feature control is implemented in a local handler class as part of the behavior pool. This local class inherits from the base handler class `CL_ABAP_BEHAVIOR_HANDLER`.

The signature of the feature control method is typed using the keyword `FOR FEATURES` followed by the import parameters. The type of importing parameter keys is an internal table containing the keys of the instances the feature control will be executed on. Importing parameter `requested_features` is a structure of Boolean-like components that reflect which elements (actions, standard operations, fields) of the entity are requested for dynamic feature control by the consumer. You can improve the performance of the handler method by evaluating this parameter and only executing the logic for the requested elements.

Exporting the parameter result is used to return the feature control values. The table-like parameter includes, besides the key fields, all actions, standard operations, and fields of the entity, for which the feature control was defined in the behavior definition.

Although not visible in the method definition, the feature handler method also has a response parameters failed and reported for indicating failures and returning messages

Response Parameter RESULT



The result parameter of the instance feature handler method is an internal table. The first columns are the key fields of the CDS entity, accessible directly or via named include `%tky`.

The columns `%update` and `%delete` only exist if feature control has been defined for the related standard operation. The type of these columns is `ABP_BEHV_FLAG`, with possible values `if_abap_behv=>fc-o-enabled` and `if_abap_behv=>fc-o-disabled`.

The column `%action` only exists if feature control has been defined for at least one instance action. The components of `%action` are named after the actions for which feature control has been defined. The type of these components is `ABP_BEHV_FLAG` with possible values `if_abap_behv=>fc-o-enabled` and `if_abap_behv=>fc-o-disabled`.

The column `%field` only exists if feature control has been defined for at least one field. The components of `%field` are named after the fields, for which feature control has been defined.

The type of these components is ABP_BEHV_FEATURE with possible values if_abap_behv=>fc-f-unrestricted, if_abap_behv=>fc-f-read_only, if_abap_behv=>fc-f-mandatory, and if_abap_behv=>fc-f-all.



Caution:

It is mandatory that the feature handler method returns at least one entry for each entity instance listed in import parameter keys. If this is not the case, the RAP runtime framework terminates with an exception.

Feature Control Implementation



```
METHOD get_features.
  READ ENTITY IN LOCAL MODE ...
  LOOP AT lt_data INTO ls_data.
    ls_result-%tky = ls_data-%tky.
    IF .... .
      ls_result-%update = if_abap_behv=>fc-o-disabled.
    ENDIF.
    APPEND ls_result TO result.
  ENDOLOOP.
ENDMETHOD.
```

Figure 110: Example: Dynamic Operation Control

A typical implementation of dynamic operation control starts with retrieving the data of all affected entity instances. For each instance in turn, an entry with the same key is added to the result parameter.



Note:

It is recommended to use field group %tky to copy the values of the key.

Based on the data, a decision is made about whether to allow the operation for this instance or not.

To disable the operation, the related component of the result parameter is filled with the value of the constant if_abap_behv=>fc-o-disabled before adding the new entry.



Note:

If you keep the initial value for the component, the operation stays enabled.



```

METHOD get_features.

READ ENTITY IN LOCAL MODE ... └─> Retrieve data for
                                         affected entity instances

LOOP AT lt_data INTO ls_data.

ls_result-%tky = ls_data-%tky. └─> Check if action
                                         should be allowed

IF ... . └─> ls_result-%action-my_action = if_abap_behv=>fc-o-disabled.

ENDIF. └─> Disable action
                                         my_action

APPEND ls_result TO result. └─> Fill result parameter

ENDLOOP.

ENDMETHOD.

```

Figure 111: Example: Dynamic Action Control

A typical implementation of dynamic action control follows the same pattern as dynamic operation control. The difference is that a related component of substructure %action is filled with the value of constant `if_abap_behv=>fc-o-disabled`.

In the example, action `my_action` is disabled if the condition is true for an entity instance.



```

METHOD get_features.

READ ENTITY IN LOCAL MODE ... └─> Retrieve data for
                                         affected entity instances

LOOP AT lt_data INTO ls_data.

ls_result-%tky = ls_data-%tky. └─> Check if field should
                                         be read-only

IF ... . └─> ls_result-%field-my_field = if_abap_behv=>fc-f-readonly.

ENDIF. └─> Set field my_field to
                                         read-only

APPEND ls_result TO result. └─> Fill result parameter

ENDLOOP.

ENDMETHOD.

```

Figure 112: Example: Dynamic Field Control

Finally, a dynamic field control implementation uses the component of substructure %field that has the same name as the affected field. In the example, field `my_field` set to read-only if the condition is true for an entity instance. Other values for the field behavior are `unrestricted`, `mandatory`, `all`.



Note:

If you keep the initial value for the component, the field remains unrestricted.

Unit 3 Exercise 12

Implement Dynamic Action and Field Control

Business Scenario

In this exercise, you disable the action `SET_TO_CANCELLED` for flight travels that already have status `cancelled` or which have already ended. You also disallow direct editing of such flight travels.

Additionally, you allow or disallow the editing of specific fields if the start date lies in the past but the end date still lies in the future.



Note:

In this exercise, replace `##` with the number that your instructor assigned to you.

Table 12: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	<code>D437C_I_TRAVEL</code>
ABAP Class	<code>BP_D437C_I_TRAVEL</code>

Task 1: Dynamic Action Control

In the behavior definition for your RAP Business Object, enable dynamic feature control for action `SET_TO_CANCELLED`. Add the `get_features` method to the local handler class and implement it to disable action `SET_TO_CANCELLED` for all flight travels that already have the status `cancelled` or which have already ended.

1. In your behavior definition for the data model view (`Z##_I_Travel`), locate the definition of action `SET_TO_CANCELLED` and add the relevant option to enable dynamic feature control based on entity instances.
2. Activate the behavior definition.
3. Add the method for feature control to the local handler class.



Hint:

The editor offers a quick fix for this if you place the cursor on the keyword `features` or keyword instance.

4. In the implementation of method `get_features`, derive the values of fields `status`, `StartDate`, and `EndDate` for the affected flight travel(s).

5. Implement a loop over the retrieved data. For each flight travel in turn, append a new line to result parameter `result` that contains the key of this flight travel.
6. If the `Status` field contains the value 'c', disable the action by filling the related component of the structure `%features-%action` with the required value.



Hint:

You find constants for feature control in attribute `fc-o` of interface `if_abap_behv`.

7. Disable the action if the value of the field `EndDate` is not empty and lies before or on the current system date.



Hint:

Instead of using the system field `sy-datum`, you can derive the current system date by calling the method `get_system_date()` of ABAP class `CL_ABAP_CONTEXT_INFO`.

8. Activate the behavior pool.
9. Restart your SAP Fiori elements app and verify that the `Set to Cancelled` button is disabled for flight travels that already have status `cancelled`, or which lie in the past.

Task 2: Dynamic Entity Control

Enable dynamic control for operation `Update` and disable editing for all flight travels for which you already disabled action `SET_TO_CANCELLED`.

1. In your behavior definition for the data model view (`Z##_I_Travel`), locate the statement `update;` and add the relevant option to enable dynamic feature control based on entity instances.
2. Activate the behavior definition.
3. Navigate to the method for feature control and disable editing for the same flight travels for which you disabled action `SET_TO_CANCELLED`.



Hint:

Set the component `%update` of the structure `%features` to the required value.

4. Activate the implementation class.
5. Restart your SAP Fiori app and verify that you cannot edit flight travels that have status `cancelled`, or which lie in the past.

Task 3: Dynamic Field Control

Enable dynamic field control for the input fields `CustomerID` and `StartDate`. Set the fields to read-only if the travel has already started, that is, if the start date lies in the past.

1. In your behavior definition for the data model view (`Z##_I_Travel`), locate the statement `field` for the fields `CustomerID`, `StartDate`, and `EndDate`. Split the statement into two statements, one for the field `EndDate`, one for fields `CustomerID` and `StartDate`.
2. In the statement for fields `CustomerID` and `StartDate`, add the relevant option to enable dynamic field control based on entity instances.
3. Activate the behavior definition.
4. Navigate to the method for feature control and disable editing of fields `CustomerID` and `StartDate` for which the `StartDate` is not empty and lies in the past or is today. Set the fields to `mandatory`, as before, if the start date lies in the future.



Hint:

Set the respective components of `%features-%field` to one of the components of the constant structure `fc-f` of interface `if_abap_behv`.

5. Activate the implementation class.
6. Restart your SAP Fiori app and verify that, for flight trips that already started, you can only edit the description and the end date but neither customer nor start date.

Unit 3

Solution 12

Implement Dynamic Action and Field Control

Business Scenario

In this exercise, you disable the action `SET_TO_CANCELLED` for flight travels that already have status `cancelled` or which have already ended. You also disallow direct editing of such flight travels.

Additionally, you allow or disallow the editing of specific fields if the start date lies in the past but the end date still lies in the future.



Note:

In this exercise, replace `##` with the number that your instructor assigned to you.

Table 12: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	<code>D437C_I_TRAVEL</code>
ABAP Class	<code>BP_D437C_I_TRAVEL</code>

Task 1: Dynamic Action Control

In the behavior definition for your RAP Business Object, enable dynamic feature control for action `SET_TO_CANCELLED`. Add the `get_features` method to the local handler class and implement it to disable action `SET_TO_CANCELLED` for all flight travels that already have the status `cancelled` or which have already ended.

1. In your behavior definition for the data model view (`Z##_I_Travel`), locate the definition of action `SET_TO_CANCELLED` and add the relevant option to enable dynamic feature control based on entity instances.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition.
 - a) Perform this step as in previous exercises.
3. Add the method for feature control to the local handler class.



Hint:

The editor offers a quick fix for this if you place the cursor on the keyword `features` or keyword `instance`.

- a) Place the cursor on keyword `features` or keyword `instance`.

- b) Open the context menu and choose *Quick Fix* or press **Ctrl + 1**.
 - c) Double-click the proposed quick fix.
4. In the implementation of method `get_features`, derive the values of fields `Status`, `StartDate`, and `EndDate` for the affected flight travel(s).
- a) Use the EML statement `READ ENTITY`.
 - b) See the source code extract from the model solution.
5. Implement a loop over the retrieved data. For each flight travel in turn, append a new line to result parameter `result` that contains the key of this flight travel.
- a) See the source code extract from the model solution.
6. If the `Status` field contains the value '`C`', disable the action by filling the related component of the structure `%features-%action` with the required value.



Hint:

You find constants for feature control in attribute `fc-o` of interface `if_abap_behv`.

- a) See the source code extract from the model solution.
7. Disable the action if the value of the field `EndDate` is not empty and lies before or on the current system date.



Hint:

Instead of using the system field `sy-datum`, you can derive the current system date by calling the method `get_system_date()` of ABAP class `CL_ABAP_CONTEXT_INFO`.

- a) See the source code extract from the model solution.
8. Activate the behavior pool.
- a) Perform this step as in previous exercises.
9. Restart your SAP Fiori elements app and verify that the `Set to Cancelled` button is disabled for flight travels that already have status `cancelled`, or which lie in the past.
- a) Perform this step as before.

Task 2: Dynamic Entity Control

Enable dynamic control for operation `Update` and disable editing for all flight travels for which you already disabled action `SET_TO_CANCELLED`.

1. In your behavior definition for the data model view (`Z##_I_Travel`), locate the statement `update;` and add the relevant option to enable dynamic feature control based on entity instances.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition.
 - a) Perform this step as in previous exercises.

3. Navigate to the method for feature control and disable editing for the same flight travels for which you disabled action SET_TO_CANCELLED.



Hint:

Set the component %update of the structure %features to the required value.

- a) See the source code extract from the model solution.
4. Activate the implementation class.
 - a) Perform this step as before.
5. Restart your SAP Fiori app and verify that you cannot edit flight travels that have status cancelled, or which lie in the past.
 - a) Perform this step as before.

Task 3: Dynamic Field Control

Enable dynamic field control for the input fields CustomerID and StartDate. Set the fields to read-only if the travel has already started, that is, if the start date lies in the past.

1. In your behavior definition for the data model view (`Z##_L_Travel`), locate the statement `field` for the fields `CustomerID`, `StartDate`, and `EndDate`. Split the statement into two statements, one for the field `EndDate`, one for fields `CustomerID` and `StartDate`.
 - a) See the source code extract from the model solution.
2. In the statement for fields `CustomerID` and `StartDate`, add the relevant option to enable dynamic field control based on entity instances.
 - a) See the source code extract from the model solution.
3. Activate the behavior definition.
 - a) Perform this step as in previous exercises.
4. Navigate to the method for feature control and disable editing of fields `CustomerID` and `StartDate` for which the `StartDate` is not empty and lies in the past or is today. Set the fields to `mandatory`, as before, if the start date lies in the future.



Hint:

Set the respective components of `%features->%field` to one of the components of the constant structure `fc-f` of interface `if_abap_behv`.

- a) See the source code extract from the model solution.
5. Activate the implementation class.
 - a) Perform this step as before.
6. Restart your SAP Fiori app and verify that, for flight trips that already started, you can only edit the description and the end date but neither customer nor start date.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition D437C_I_TRAVEL:

```

...
define behavior for D437c_I_Travel
persistent table d437c_travel
lock master
etag master ChangedAt
authorization master ( instance )
{
    ...
    ...
// field ( mandatory ) CustomerID, StartDate, EndDate;
    field ( mandatory ) EndDate;
    field ( features : instance ) CustomerID, StartDate;
    ...
update ( features : instance );
    ...
action ( features : instance ) set_to_cancelled;
    ...
}

```

Class BP_D437C_TRAVEL:

```

METHOD get_features.

* work area for parameter result
    DATA ls_result LIKE LINE OF result.

* helper objects to shorten the code
    CONSTANTS c_enabled    TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-o-enabled.
    CONSTANTS c_disabled   TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-o-disabled.
    CONSTANTS c_read_only  TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-f-read_only.
    CONSTANTS c_mandatory  TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-f-mandatory.

    DATA lv_today TYPE cl_abap_context_info=>ty_system_date.

*****
* Get system date
    lv_today = cl_abap_context_info=>get_system_date( ).

* Read data of all affected
    READ ENTITY IN LOCAL MODE d437c_i_travel
        FIELDS ( status startdate enddate )
        WITH CORRESPONDING #( keys )
        RESULT DATA(lt_travel).

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).
    ls_result-%tky = <ls_travel>-%tky.

```

```

* Dynamic action control
  IF <ls_travel>-status = 'C'. "already cancelled
    ls_result-%features-%action-set_to_cancelled = c_disabled.
  ELSEIF <ls_travel>-enddate IS NOT INITIAL
    AND <ls_travel>-enddate <= lv_today.
    ls_result-%features-%action-set_to_cancelled = c_disabled.
  ELSE.
    ls_result-%features-%action-set_to_cancelled = c_enabled.
  ENDIF.

* dynamic operation control (update)
  IF <ls_travel>-status = 'C'. "already cancelled
    ls_result-%features-%update = c_disabled.
  ELSEIF <ls_travel>-enddate IS NOT INITIAL
    AND <ls_travel>-enddate <= lv_today.
    ls_result-%features-%update = c_disabled.
  ELSE.
    ls_result-%features-%update = c_enabled.
  ENDIF.

* dynamic field control (Customer, StartDate)
  IF <ls_travel>-startdate IS NOT INITIAL
    AND <ls_travel>-startdate <= lv_today.
    ls_result-%features-%field-startdate = c_read_only.
    ls_result-%features-%field-customerid = c_read_only.
  ELSE.
    ls_result-%features-%field-startdate = c_mandatory.
    ls_result-%features-%field-customerid = c_mandatory.
  ENDIF.

  APPEND ls_result TO result.
ENDLOOP.
ENDMETHOD.
```

Alternative coding with even more expression-based syntax:

```

METHOD get_features.

DATA(lv_today) = cl_abap_context_info=>get_system_date( ).

READ ENTITY IN LOCAL MODE d437c_i_travel
  ALL FIELDS WITH CORRESPONDING #( keys )
  RESULT DATA(lt_travel).

result =
  VALUE #( FOR <travel> IN lt_travel
    (
      "key
      %tky = <travel>-%tky
      "action control
      %features-%action-set_to_cancelled
      = COND #( WHEN <travel>-status = 'C'
        THEN if_abap_behv=>fc-o-disabled
        WHEN <travel>-enddate IS NOT INITIAL
          AND <travel>-enddate <= lv_today
            THEN if_abap_behv=>fc-o-disabled
            ELSE   if_abap_behv=>fc-o-enabled
      )
    )
  )
```

```
"operation control
%features-%update
= COND #( WHEN <travel>-status = 'C'
    THEN if_abap_behv=>fc-o-disabled
WHEN <travel>-enddate IS NOT INITIAL
    AND <travel>-enddate <= lv_today
    THEN if_abap_behv=>fc-o-disabled
    ELSE if_abap_behv=>fc-o-enabled
)
"field control
%features-%field-startdate
= COND #( WHEN <travel>-startdate IS NOT INITIAL
    AND <travel>-startdate <= lv_today
    THEN if_abap_behv=>fc-f-read_only
    ELSE if_abap_behv=>fc-f-mandatory
)
%features-%field-customerid
= COND #( WHEN <travel>-startdate IS NOT INITIAL
    AND <travel>-startdate <= lv_today
    THEN if_abap_behv=>fc-f-read_only
    ELSE if_abap_behv=>fc-f-mandatory
)
)
).

ENDMETHOD.
```



LESSON SUMMARY

You should now be able to:

- Explain dynamic action, operation, and field control in RAP
- Implement dynamic feature control

UNIT 4

Draft-Enabled Transactional Apps

Lesson 1

Understanding the Draft Concept	216
Exercise 13: Enable Draft Handling for a RAP Business Object	231

Lesson 2

Developing Draft-Enabled Applications	240
Exercise 14: Enable Draft Handling in SAP Fiori Elements App and Adjust Implementations	251

UNIT OBJECTIVES

- Explain the need for draft in stateless applications
- Enable draft handling in the Business Object
- Enable draft handling in a SAP Fiori elements app
- Explain the difference between transition messages and state messages
- Describe the draft-specifics in behavior implementations

Understanding the Draft Concept

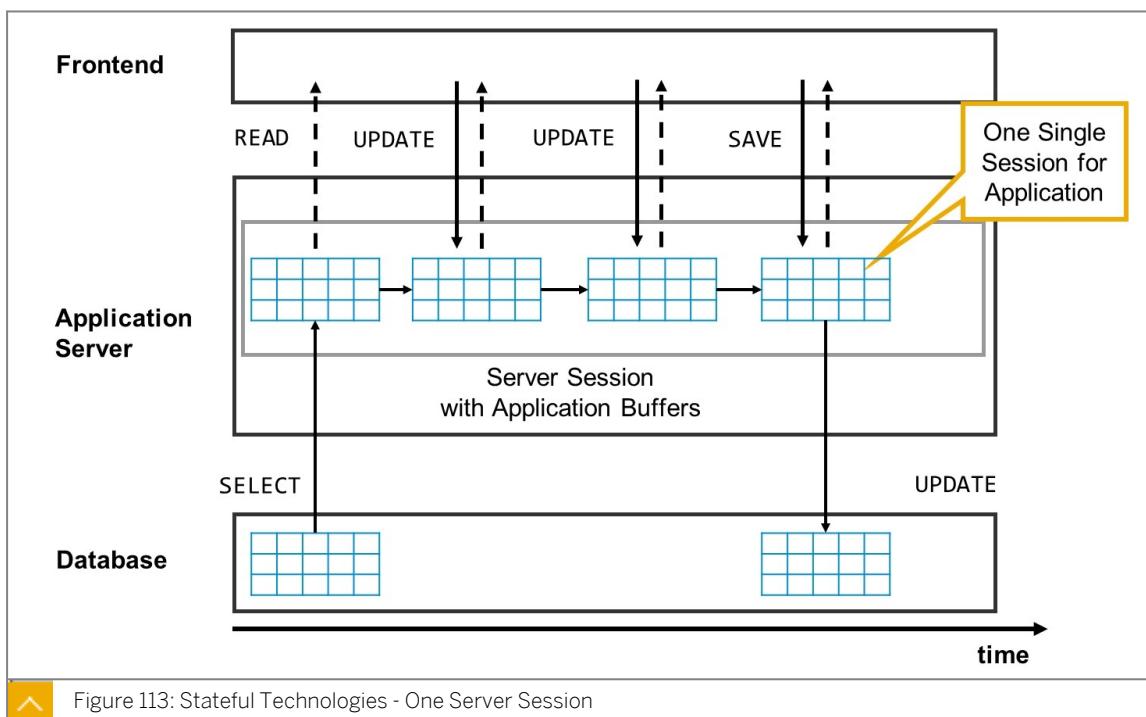


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain the need for draft in stateless applications
- Enable draft handling in the Business Object

Draft Motivation



SAP traditional applications are developed using stateful technologies, such as Floorplan Manager for Web Dynpro ABAP or the classic Dynpro technique.

These stateful transactional applications rely on a server session along with application buffers that can fulfill client requests (user interactions with multiple backend round trips) until the user has saved the data changes and finished their work. In stateful applications, data entry and data updates work on a temporary in-memory version of a business entity which is only persisted once it is sufficiently complete and consistent.

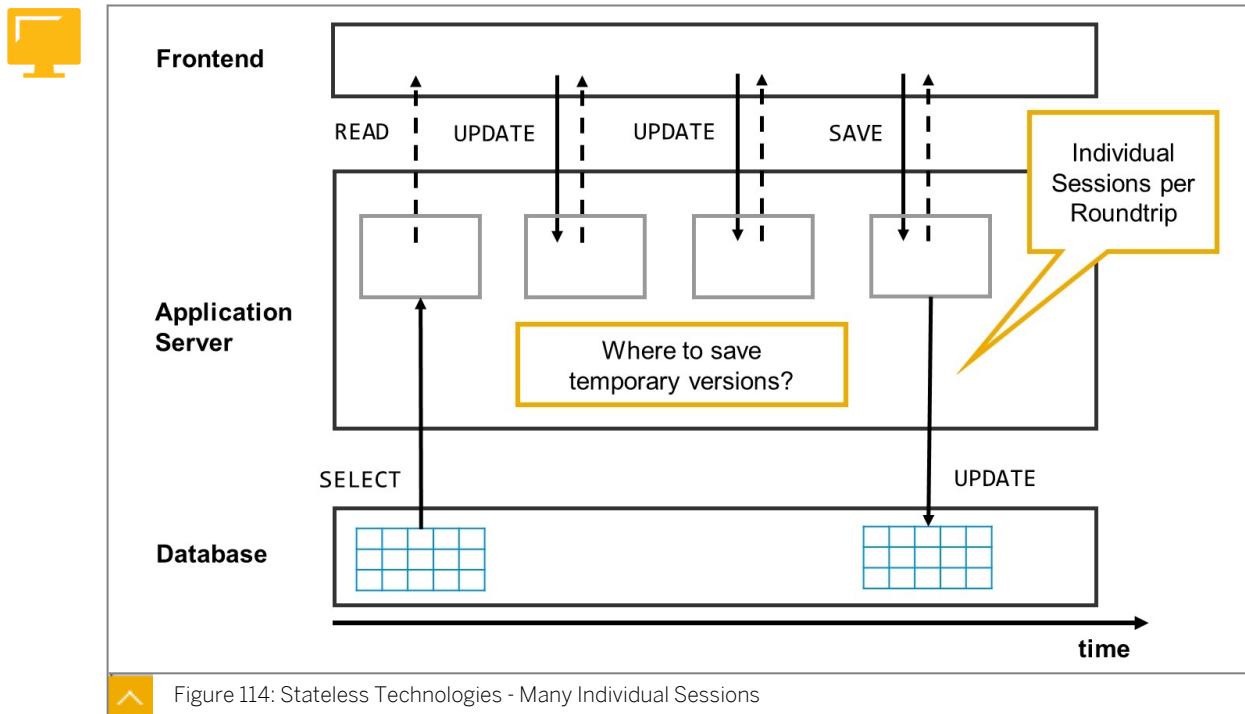


Figure 114: Stateless Technologies - Many Individual Sessions

Modern cloud-ready apps require a stateless communication pattern, for example, to leverage cloud capabilities like elasticity and scalability. Therefore, there is no fixed backend session resource along a business transaction for each user and the incoming requests can be dispatched to different backend resources, which supports load balancing. As a consequence, the application cannot save a temporary version of the business entity inside the application.

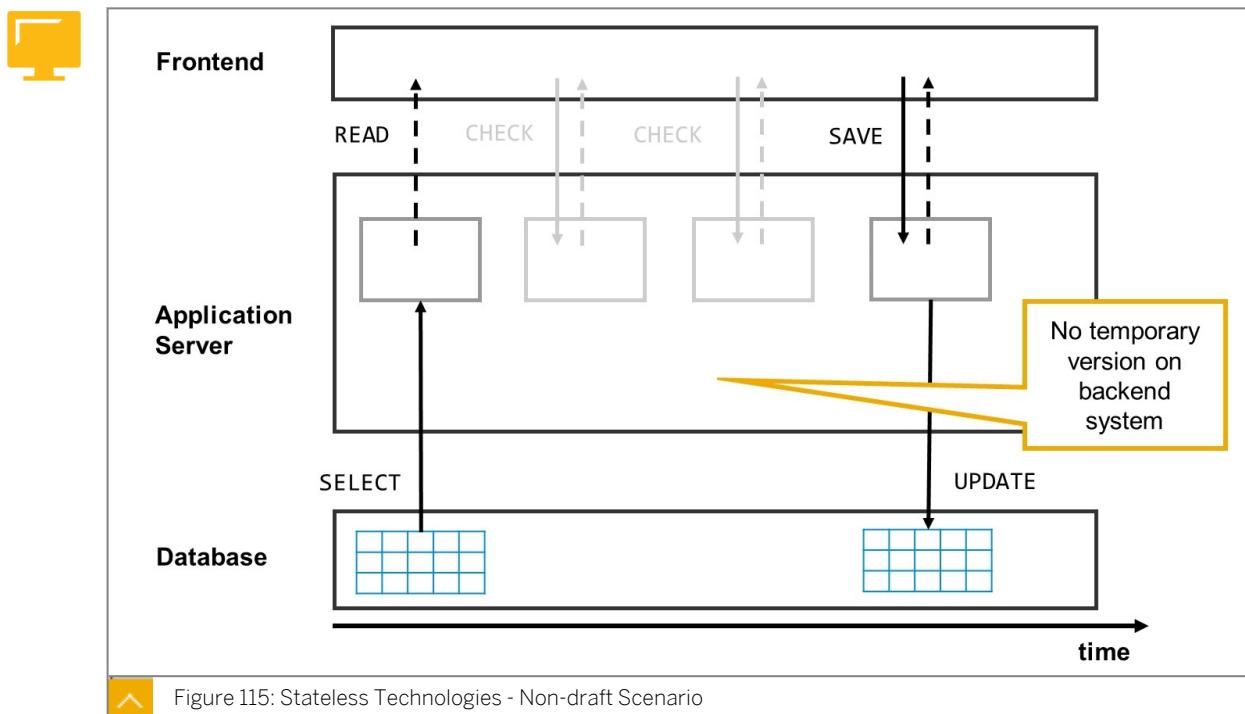
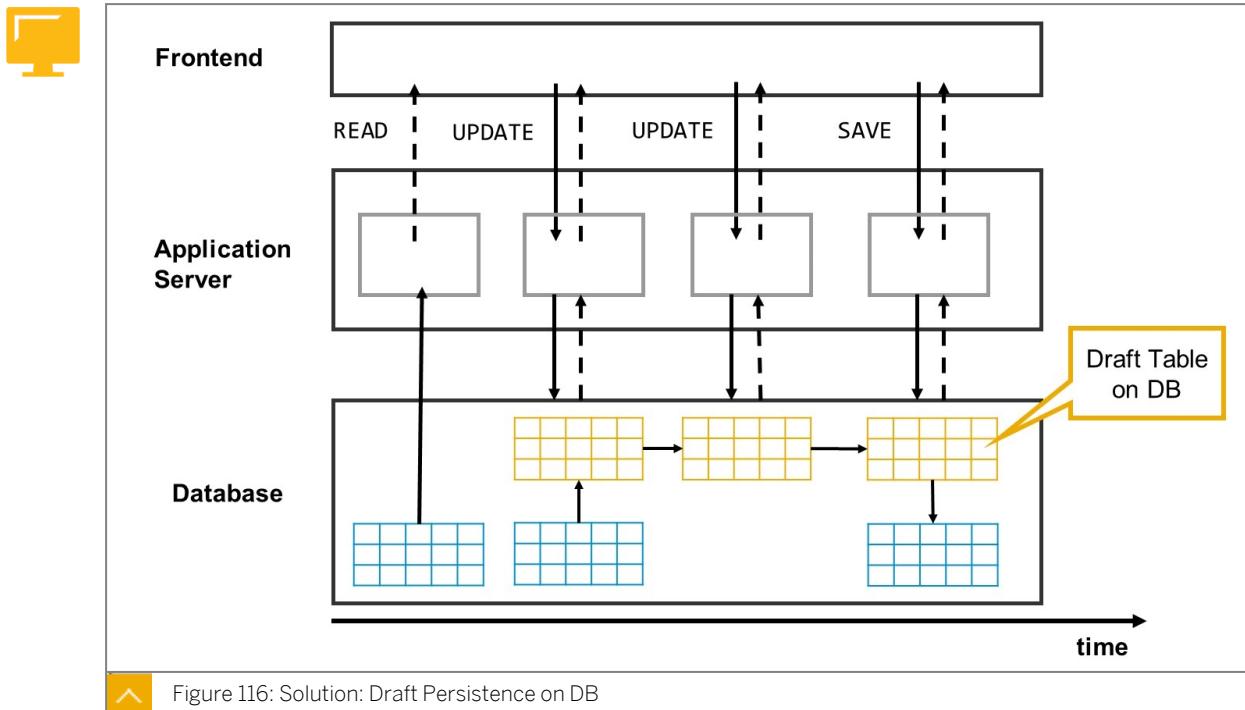


Figure 115: Stateless Technologies - Non-draft Scenario

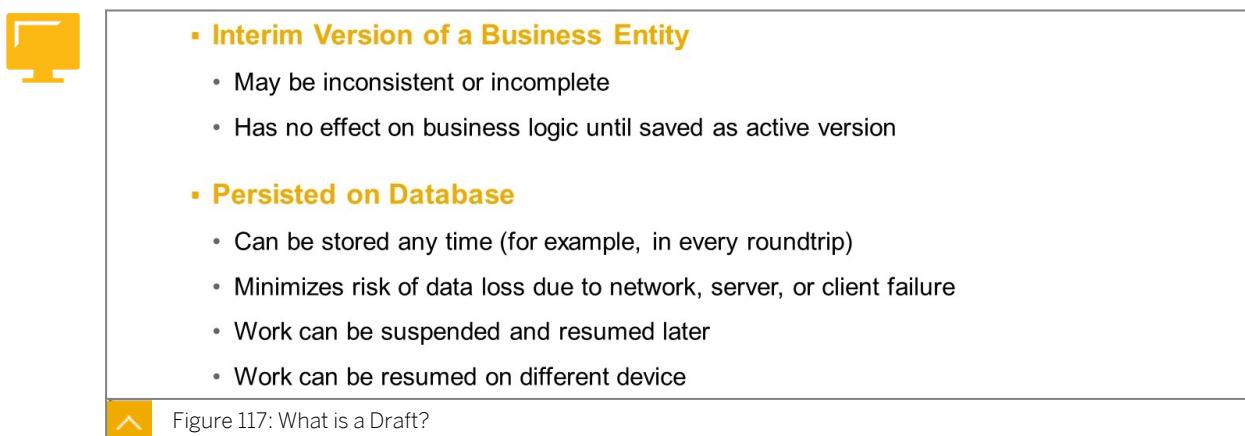
The ABAP Restful Application Programming Model follows such a stateless approach. The application does not save temporary versions on the application server. The application

performs checks, actions, and so on, but the changes to the data are only saved once the user chooses Save. This is referred to as the non-draft scenario.

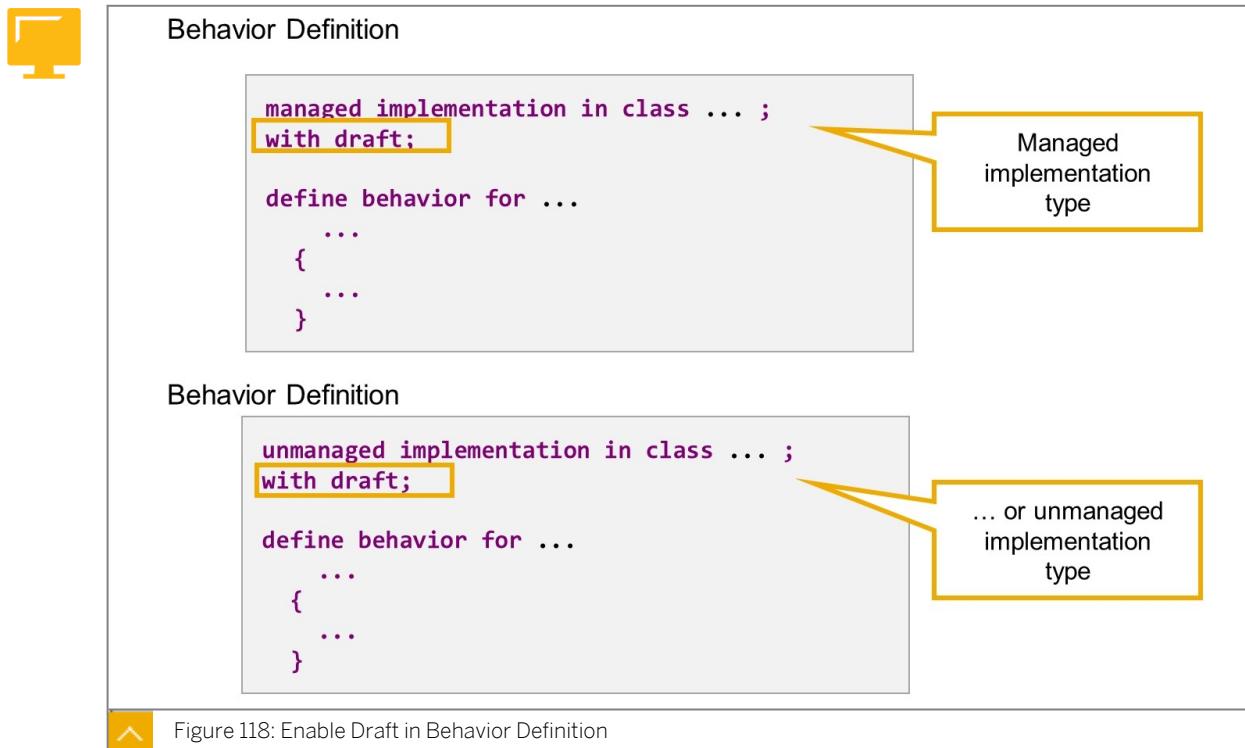


Both the stateful approach and the non-draft approach have one big disadvantage, the end user cannot store changed data that is inconsistent to continue at a later point in time or to recover this data, even if the application terminates unexpectedly.

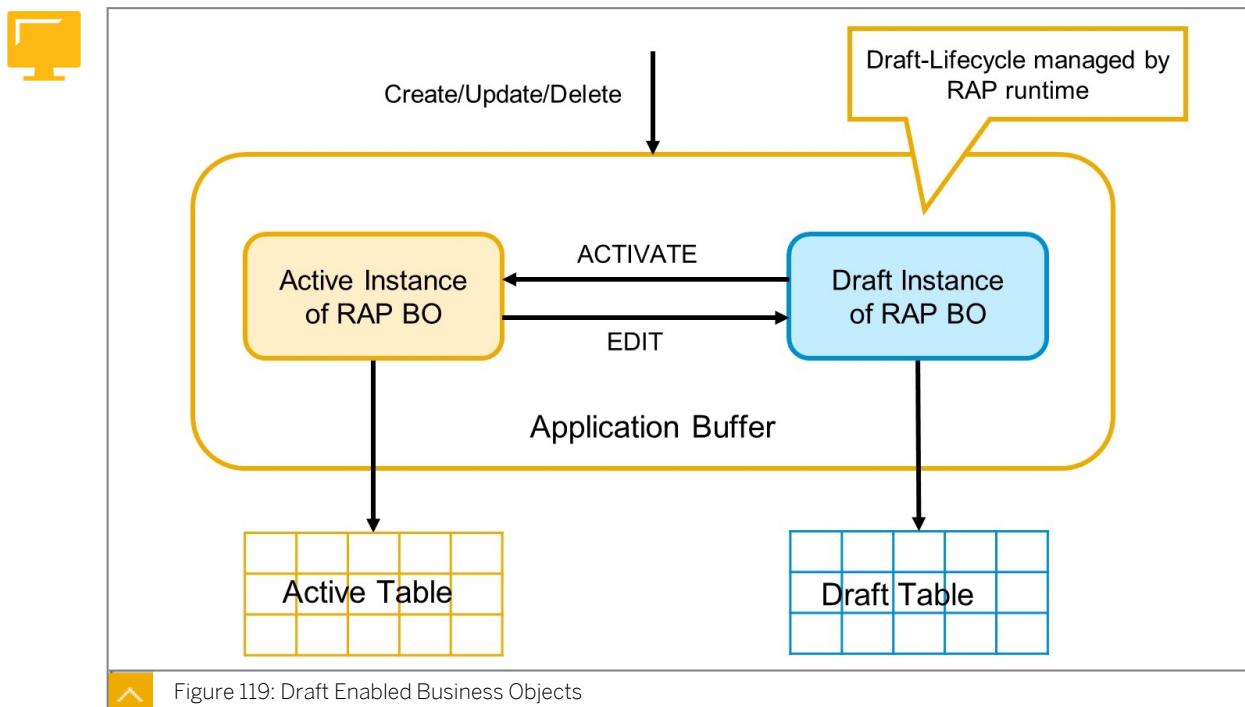
The draft scenario replaces the temporary in-memory version of the business entity with a persistent version on the database. This persistent temporary version is known as draft. It is not stored in the same database tables as the active versions but in special database tables, the Draft Tables. The draft represents the state and stores the transactional changes until they are persisted in the active table or discarded.



Draft Enabled RAP Business Objects

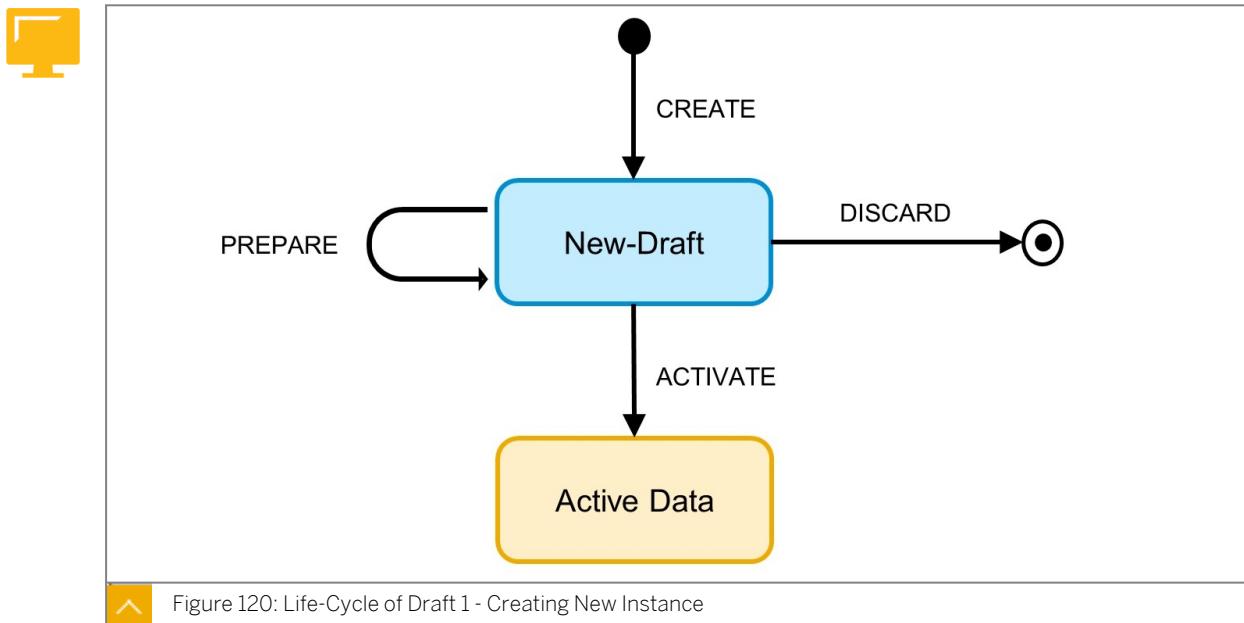


The draft capability for a draft business object is defined by adding the statement `with draft;` in the header part of the behavior definition. You can build draft business objects from scratch, or you can draft-enable existing business objects with both implementation types, managed or unmanaged.



In all scenarios, the draft is managed. This means that the draft life cycle is determined by the RAP draft-runtime as soon as the business object is draft-enabled. You, as an application developer, do not need to know how the draft instance is created, how draft data is written to the draft database table, or how the draft instance is activated.

Adding draft capabilities to your business object might imply changes in your business logic for the processing of active data that you are responsible for. In addition, RAP also offers implementation exits for cases in which you need business service-specific draft capabilities that impact the draft handling.



The fact that two database tables are involved in the runtime of a draft business object requires an elaborate life cycle of data. All data undergo several states and state transitions (actions) while being processed by a draft business object. There are two scenarios that need to be distinguished: New-Draft and Edit-Draft.

We speak of New-Draft instances if the data is initial data that is not yet persisted on the active database table. New-draft instances do not have a corresponding active instance.

The life cycle of a new-draft starts with the creation of a new draft instance. The new data is immediately stored in the draft persistence, regardless of its validity or completeness. Draft data can be enriched and checked for consistency by execution of the PREPARE action.

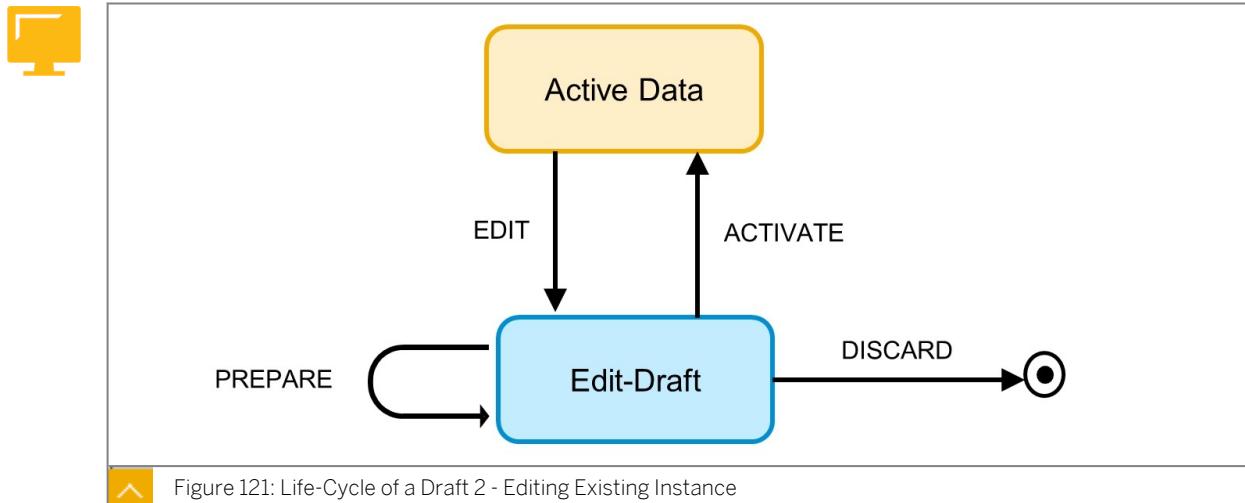
With the ACTIVATE action the draft data is copied into active data in the application buffer. ACTIVATE includes an implicit execution of PREPARE. Once the active instance is successfully created, the draft instance is discarded and the related data is deleted from the draft table.



Note:

The ACTIVATE action does not save the active instance on the database. The actual save is executed separately, either by COMMIT ENTITIES via EML or by calling the save sequence in case of OData.

Using the DISCARD action on a New-Draft will delete the related data from the application buffer and the draft table without creating an active instance.



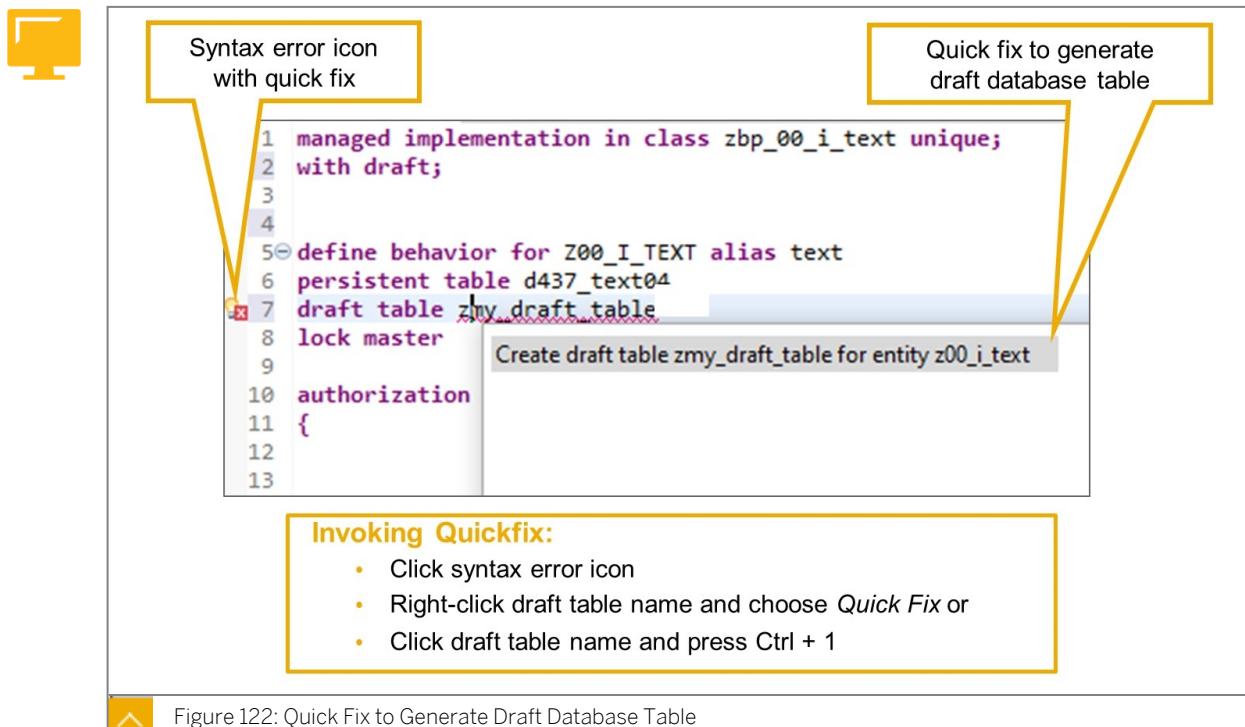
Edit-Drafts always exist in parallel to the corresponding active data. They are created by using the EDIT action on active instances. The whole active instance is copied to the draft table.

Like New-Drafts, Edit-Drafts can be enriched and validated using the PREPARE action.

With the ACTIVATE action the draft data is copied to overwrite the existing active data in the application buffer. As with New-Drafts, ACTIVATE includes an implicit execution of PREPARE.

Using the DISCARD action on an Edit-Draft will delete the draft data from the application buffer and the draft table, leaving the active data in the application buffer unchanged.

Draft Tables



Draft business objects need two separate database tables for each entity, one for the active persistence and one for storing draft instances. With using a separate database table for the draft information, it is guaranteed that the active persistence database table remains untouched and consistent for existing database functionality.

While the persistent table addition is used to specify the active table of a RAP BO entity, the draft table is assigned via the draft table addition. The draft table addition is mandatory in every behavior definition statement as soon as the RAP BO is draft-enabled.

The draft table can be generated automatically via a quick fix in the behavior definition. If the draft database table already exists, the quick fix completely overwrites the table.



Active Database Table			Draft Database Table		
Field	Key	Data Type	Field	Key	Data Type
CLIENT	<input checked="" type="checkbox"/>	CLNT	MANDT	<input checked="" type="checkbox"/>	CLNT
TEXT_UUID	<input checked="" type="checkbox"/>	RAW	TEXTUUID	<input checked="" type="checkbox"/>	RAW
TEXT_OWNER	<input type="checkbox"/>	CHAR	TEXTOWNER	<input type="checkbox"/>	CHAR
TEXT	<input type="checkbox"/>	CHAR	TEXT	<input type="checkbox"/>	CHAR
Include Structure (draft admin include)			.INCLUDE <u>DRAFTENTITYCREATIONDATETIME</u> <u>DRAFTENTITYLASTCHANGEDATETIME</u> <u>DRAFTADMINISTRATIVEUUID</u> <u>DRAFTENTITYOPERATIONCODE</u> <u>HASACTIVEENTITY</u> <u>DRAFTFIELDCHANGES</u>		

Figure 123: Draft Database Table Layout

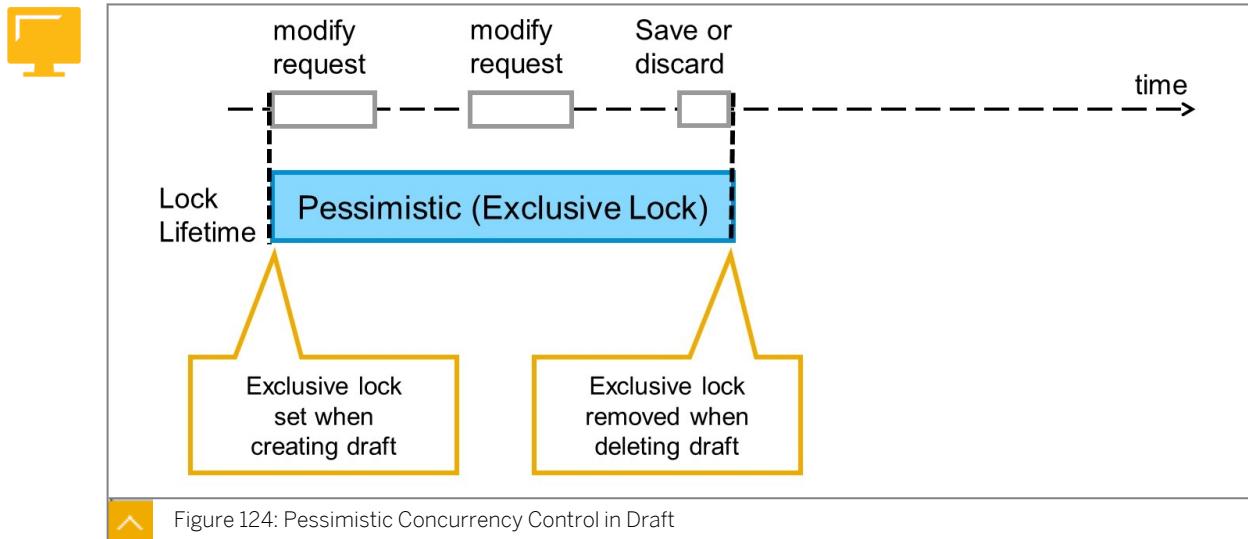
The draft database table contains exactly the same fields as the active database table plus some technical information the RAP runtime needs to handle draft. The technical information is added with the draft admin include SYCH_BDL_DRAFT_ADMIN_INC.



Note:

Although draft database tables are usual ABAP Dictionary database tables and there are no technical access restrictions, it is not allowed to directly access the draft database table via SQL, neither with reading access nor writing access. The access to the draft database table must always be done via EML, with which the draft metadata is updated automatically.

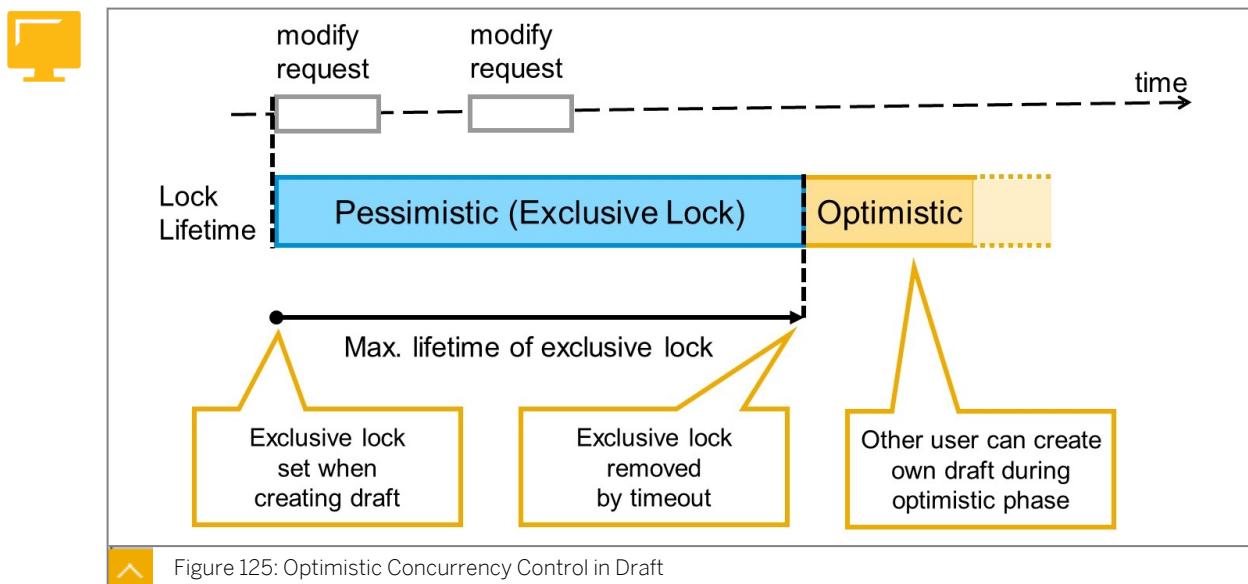
Concurrency Control in Draft



RAP uses a combination of pessimistic and optimistic concurrency control to ensure data consistency.

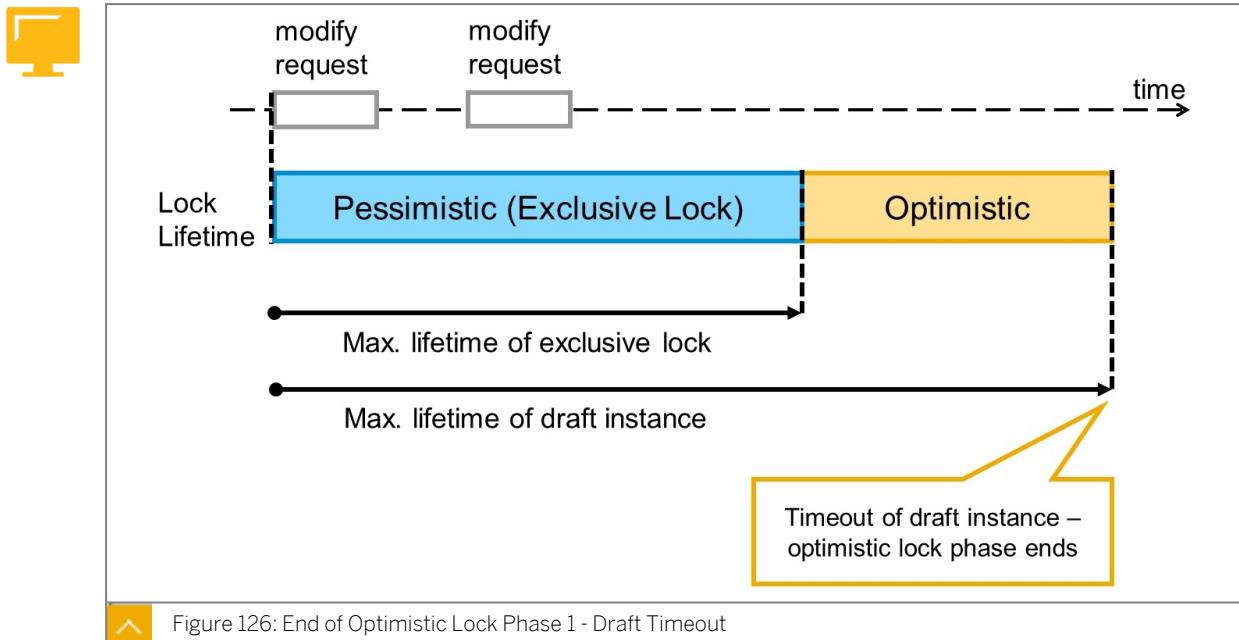
In scenarios with draft support, the pessimistic concurrency control (locking) plays an even more crucial role during the draft business object life cycle.

As soon as a draft instance is created for an existing active instance, the active instance receives an exclusive lock and cannot be modified by another user. The exclusive lock is not bound to the ABAP session. It remains intact between the different update requests from the same user. When the user saves or discards the changes, the draft is deleted and the exclusive lock is removed.

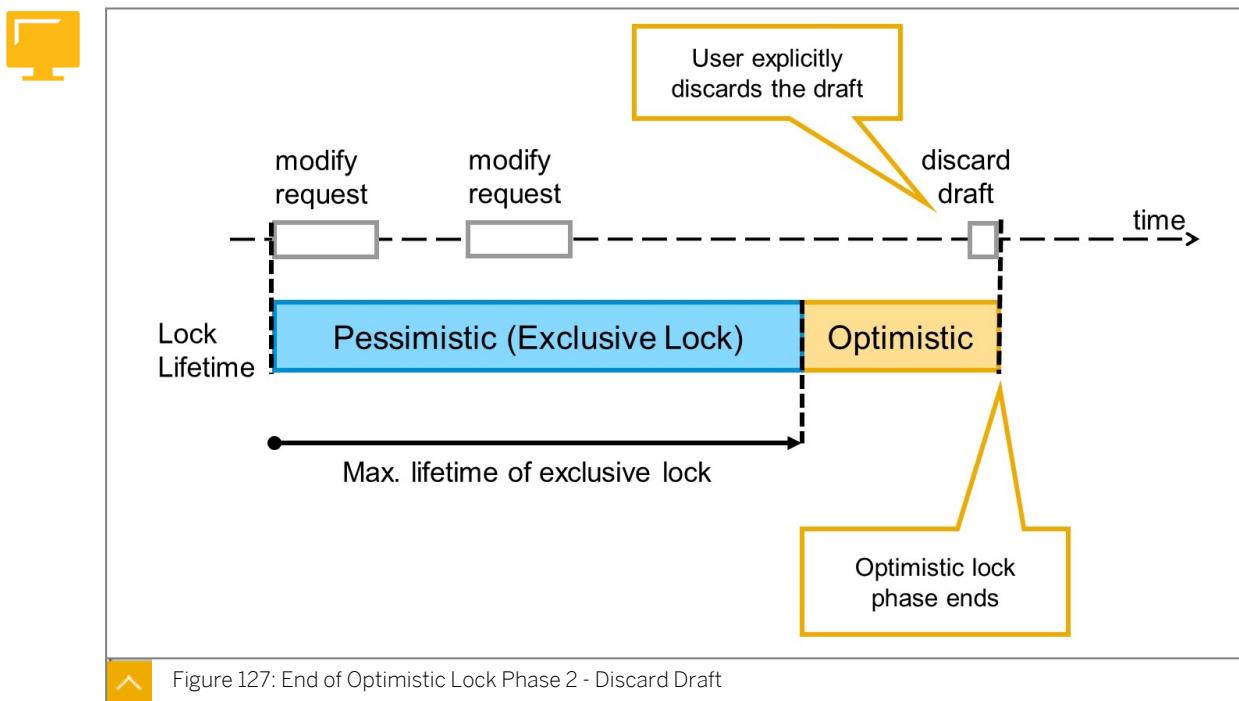


There is a maximum duration time for the exclusive lock. This duration time can be configured. When the timeout of the exclusive lock is reached, it is removed, even though the draft instance still exists because there was no explicit save or discard from the user. The pessimistic lock phase ends and the optimistic lock phase begins.

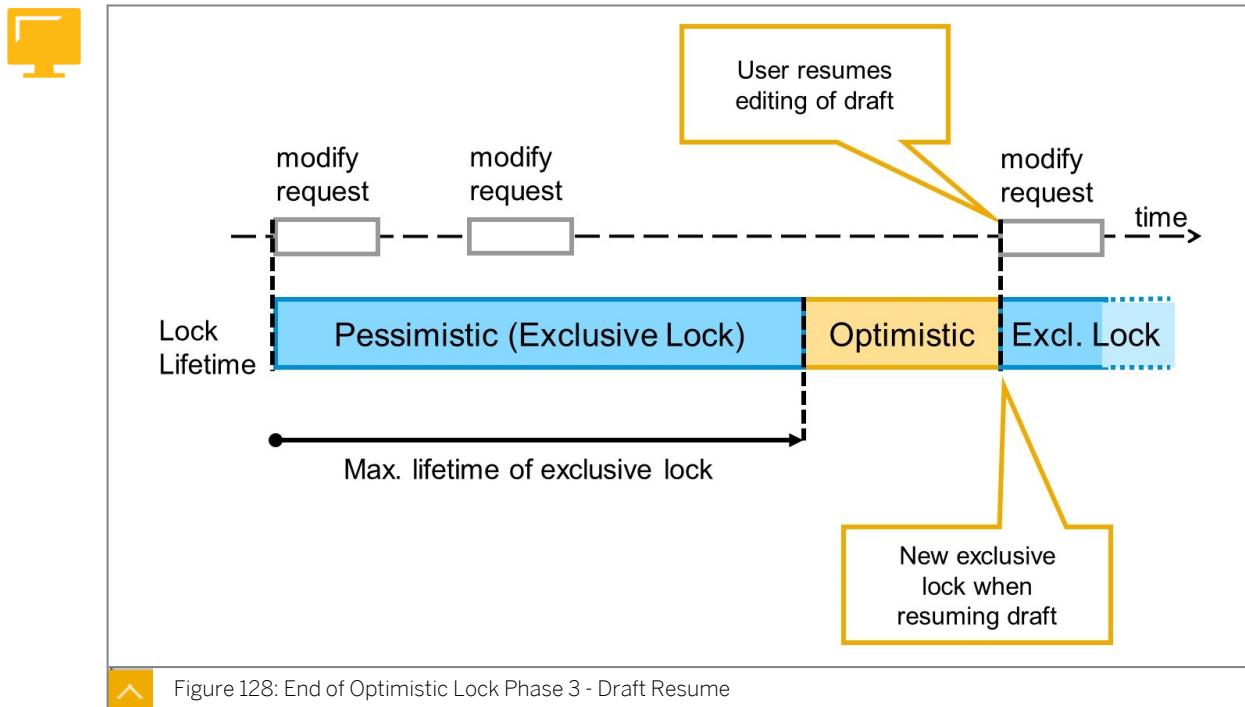
During the optimistic lock phase, another user can start editing the active instance of the business object, that is, set an exclusive lock and create their own draft instance.



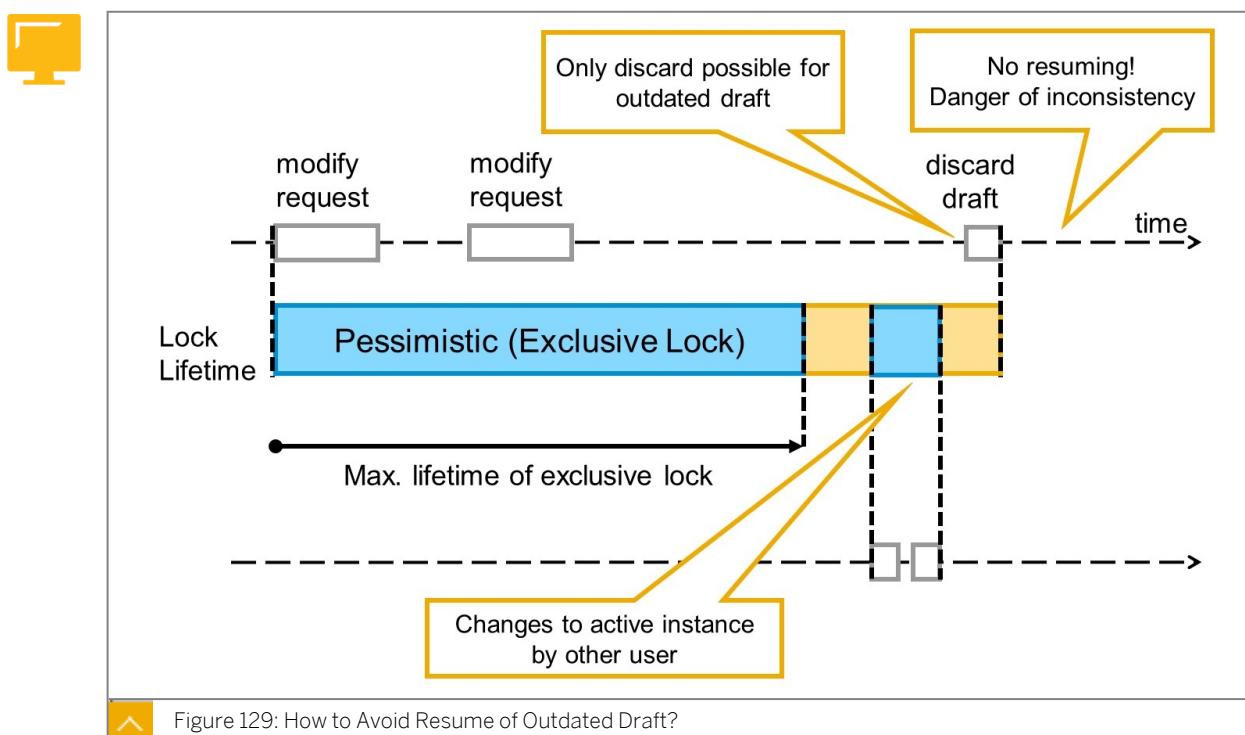
There is a configurable maximum lifetime for drafts. If the draft is not used for a certain period of time, the draft is discarded automatically by the life-cycle service. If no other draft exists at that moment, the optimistic lock phase ends.



If the user that created a draft instance for an active instance discards the draft explicitly, the optimistic lock phase ends. This can be the case if the data changes are no longer relevant.



If the user that created the draft continues to work on the draft instance after the exclusive locking phase has ended, the draft can be resumed and the changes are still available for the user. The optimistic locking phase ends as a new exclusive lock is set for the corresponding active document.

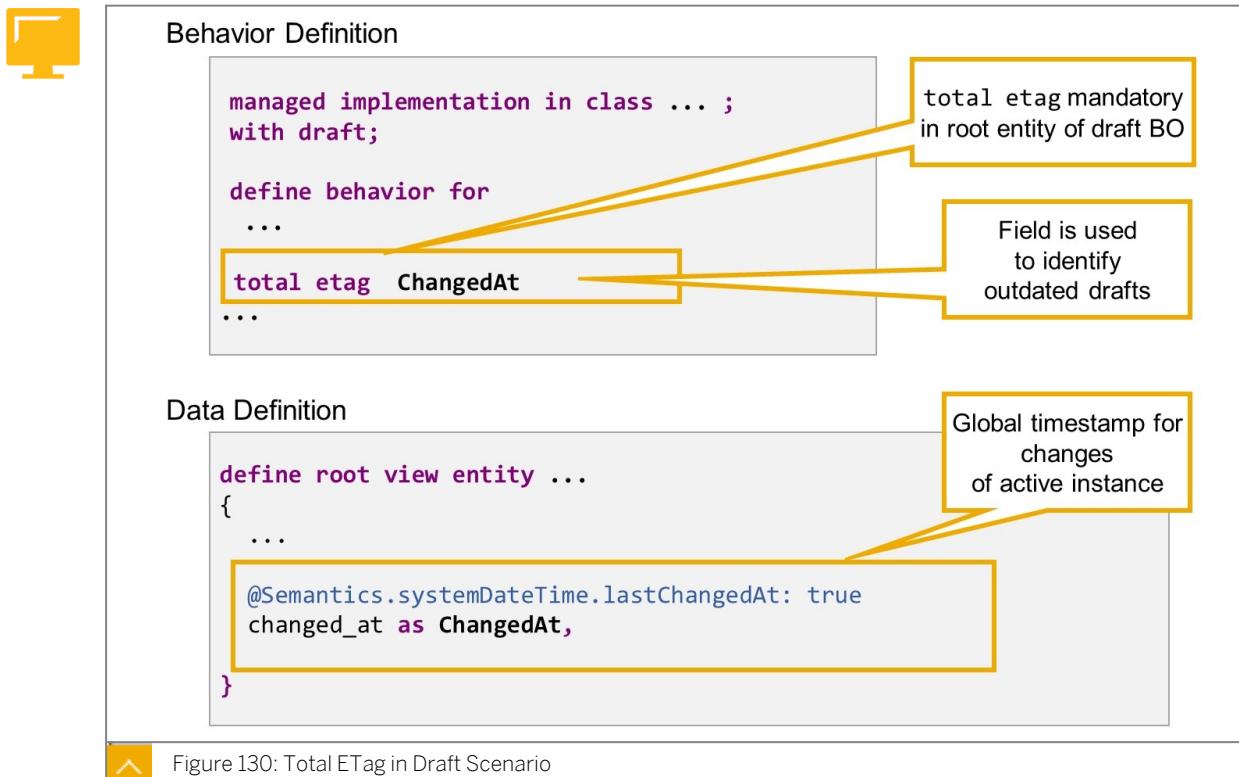


During the optimistic lock phase, it is possible that another user sets an exclusive lock, creates another draft instance, and saves the changes to the active instance. If there is legacy code accessing the same data, it is even possible that the active instance is changed directly without using a draft.

This makes the original draft outdated because it does not reflect the latest changes on the active instance. If the draft is not touched until it reaches its maximum lifetime, this is not an issue.

To avoid data inconsistencies, the framework has to ensure that the owner of the draft can only discard the changes. Resuming the draft must not be possible after the active instance was changed directly or via another draft instance.

ETag Fields in Draft



The RAP runtime framework uses an ETag field approach to identify outdated drafts. If the ETag field in the active instance and the draft are still the same, the draft is still valid and resuming is possible. If the value of the ETag field differs in the active instance and the draft instance, the active instance was changed since the exclusive lock expired and resuming the draft is no longer possible.

This ETag field is defined by adding `total etag` to the `define behavior` statement of the BO's root entity. The addition `total etag` is not supported in the behavior definition of child entities.



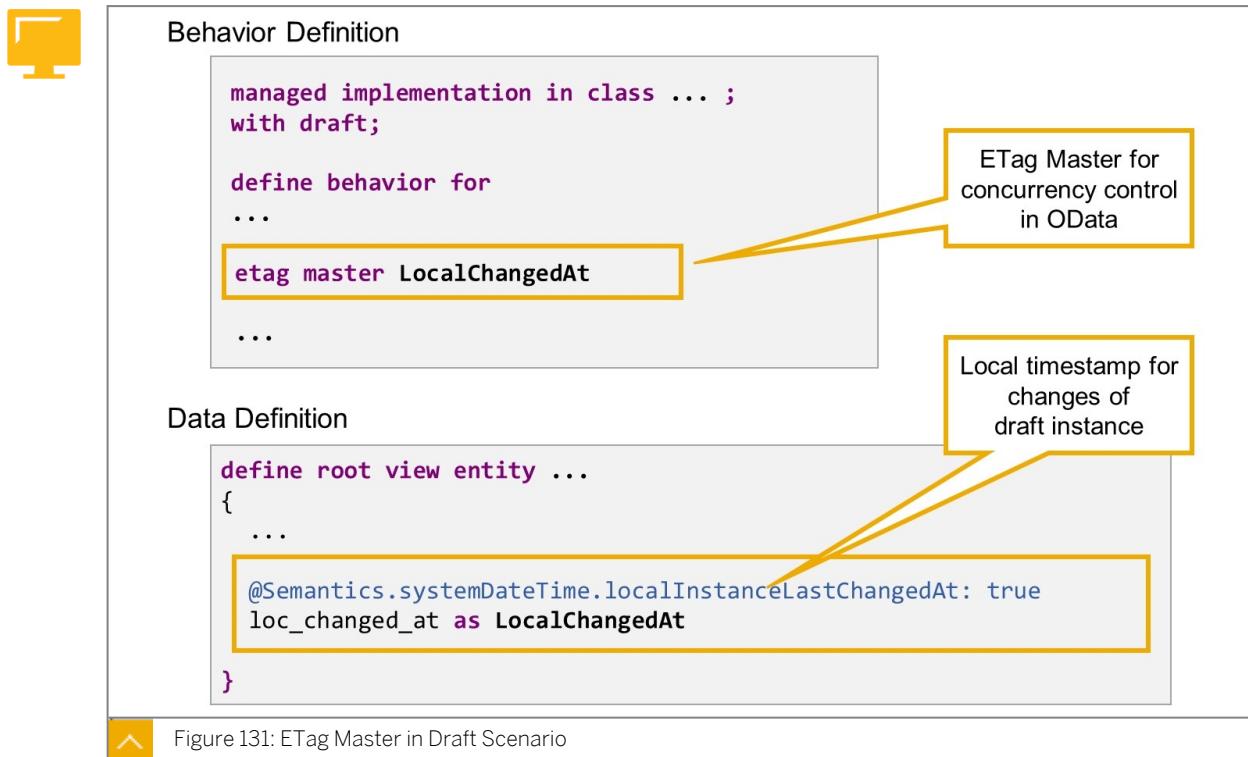
Note:

In draft-enabled RAP BOs, it is mandatory to define a total ETag field.

The field to be used as Total ETag field has to meet the following requirements:

- The Total ETag field value in the active version always changes when the active version is changed
- The Total ETag field value in a draft instance does not change during the draft lifetime

The administrative field annotated with `@Semantics.systemDateTime.lastchangedAt: true`, which we used as ETag master field earlier, meets these requirements.



On the other hand, the `lastChangedAt` field is not suitable as ETag master anymore if the BO is draft-enabled. For optimistic concurrency control in OData to work properly, the ETag master field has to receive a new value whenever there is an update of the draft instance of the related RAP BO entity. The `lastChangedAt` timestamp only changes when the draft is persisted.

To support OData concurrency control, SAP introduced a specific administrative field, the `LastChangedAt` timestamp for the local Instance. Any field annotated with `@Semantics.systemDateTime.localInstanceLastChangedAt: true` will be updated by the RAP runtime framework during every write access to the draft instance.,.

Draft Actions

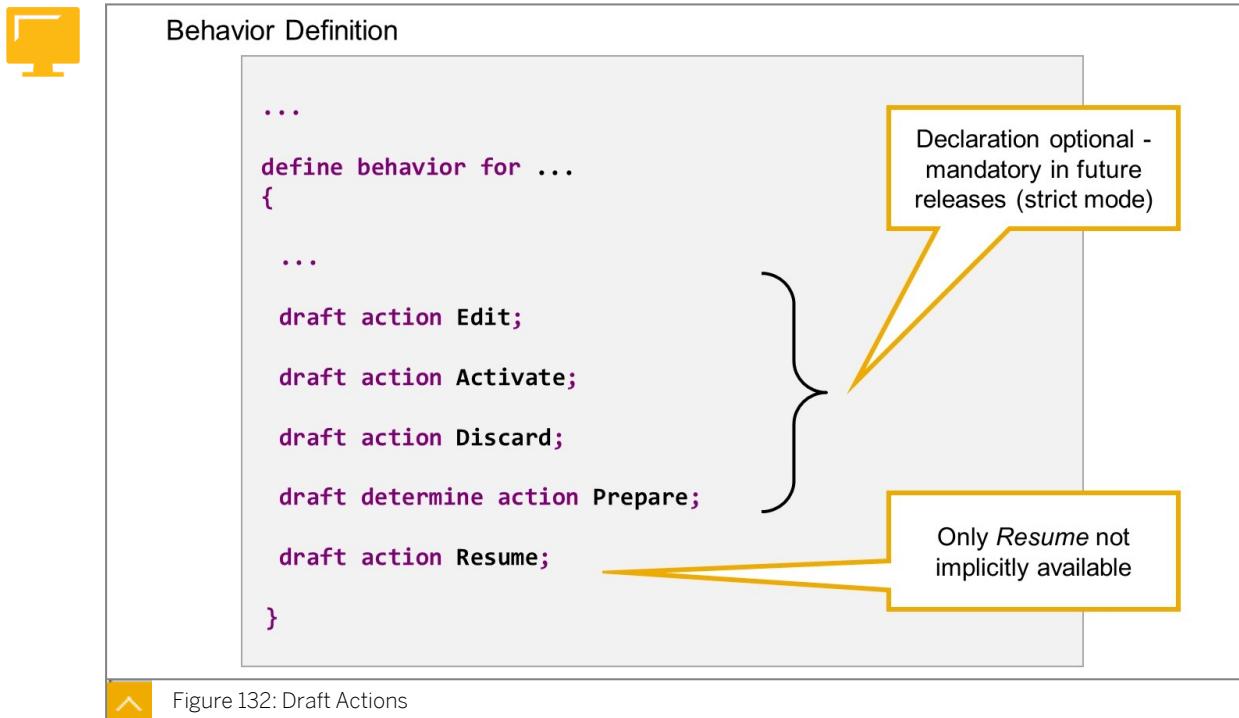


Figure 132: Draft Actions

Draft actions are actions that are implicitly available for draft business objects as soon as the business object is draft-enabled. They only exist for lock master entities, as they always refer to the whole lockable subtree of a business object.

All draft actions but one are automatically available in EML and exposed to OData, even without explicitly mentioning it in the behavior definition. The exception is draft action RESUME, which has to be declared in the behavior definition before it is available in OData and in EML.



Note:

In ABAP 7.55, draft actions can, but do not have to be, explicitly declared in the behavior definition. In future releases, their declaration will become mandatory when using strict mode for the syntax check of a behavior definition.

The following draft actions exist:

Draft Action EDIT

The draft action EDIT copies an active instance to the draft database table. Feature and authorization control is available for the EDIT, which you can optionally define to restrict the usage of the action.

Draft Action ACTIVATE

The draft action ACTIVATE is the inverse action to EDIT. It invokes the PREPARE and a modify call containing all the changes for the active instance in case of an edit-draft, or a CREATE in case of a new-draft. Once the active instance is successfully created, the draft instance is discarded.

In contrast to the draft action Edit, the Activate action does not allow feature or authorization control. Authorization is controlled later when the active instance is saved to the database.

Draft Action DISCARD

The draft action DISCARD deletes the draft instance from the draft database table. No feature or authorization control can be implemented.

Draft Determine Action PREPARE

The draft determine action PREPARE executes the determinations and validations that are specified for it in the behavior definition. The PREPARE enables validating draft data before the transition to active data.

In the behavior definition, you must specify which determinations and validations are called during the prepare action. Only determinations and validations that are defined and implemented for the BO can be used. No validations or determinations are called if there is nothing specified for the PREPARE.

Draft Action RESUME

The draft action RESUME is executed when a user continues to work on a draft instance whose exclusive lock for the active data has already expired. It re-creates the lock for the corresponding instance on the active database table. On an SAP Fiori elements UI, it is invoked when reopening and changing a draft instance whose exclusive lock is expired.

In case of a new draft, the same feature and authorization control is executed as defined for a CREATE. In case of an edit-draft, the same feature and authorization control is executed like in an Edit.

As the RESUME action is application-specific, it is only exposed to OData if it is explicitly declared in the behavior definition. You can only execute the RESUME action via EML if the action is explicitly made available in the behavior definition.

Unit 4

Exercise 13

Enable Draft Handling for a RAP Business Object

Business Scenario

In this exercise, you enable draft handling for your RAP Business Object and in your SAP Fiori elements App. To support optimistic concurrency control for draft instances, you extend the data model with a field that is updated during every save of a draft.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 13: Solution

Repository Object Type	Repository Object ID
Database Table	D437D_TRAVEL
CDS Data Definition (Model)	D437D_I_TRAVEL
CDS Data Definition (Projection)	D437D_C_TRAVEL
CDS Metadata Extension	D437D_C_TRAVEL
CDS Behavior Definition (Model)	D437D_I_TRAVEL

Task 1: Extend Data Model with Timestamp Field for Draft Changes

Extend your database table (`Z##_TRAVEL`) with an additional timestamp field (suggested name: `loc_changed_at`). Add it to your data model view (`Z##_I_Travel`) and annotate it with the necessary sub-annotation of annotation `@Semantics` to ensure the RAP run time automatically updates the field during every save of a draft. Add the field to your projection view (`Z##_C_Travel`) and its metadata extension, to keep it hidden in the SAP Fiori elements app.

1. Edit the definition of your transparent table (`Z##_TRAVEL`). At the end of the field list, add a field `loc_changed_at` and type it with data element `S_LOCCHANGEDAT`.
2. Activate the transparent table.
3. Edit the definition of your CDS view entity (`Z##_I_TRAVEL`) and add the new table field to the element list.



Note:
If you want to use a longer name in CDS and RAP (for example `LocalChangedAt`), specify the longer name after addition AS.

4. Add annotation `@Semantics.systemDateTime.localInstanceLastChangedAt: true` to the new view field.
5. Activate the data definition.
6. Edit the definition of your projection view (`Z##_C_TRAVEL`), add the new element to the element list and activate the data definition.
7. Edit the metadata extension for your projection view (`Z##_C_TRAVEL`), add the new element with annotation `@UI.hidden: true`, and activate the metadata extension.
8. Restart your SAP Fiori elements app and verify that everything still works.

Task 2: Enable Draft-Handling for the RAP Business Object

In the behavior definition for your data model view (`Z##_I_Travel`), add the necessary statement to enable draft handling for the Business Object. Define and generate a draft table for the travel data (suggested name: `Z##_TRAVEL_D` and define the view field you just created as total etag field.

1. Edit the behavior definition for your data model view (`Z##_I_TRAVEL`) and add statement with `draft;` immediately after statement `managed ... ;`
2. Scroll down to the `define behavior` statement and add addition `draft table`, followed by the name of the draft table you want to create.
3. Create the draft table.



Hint:
The editor offers a quick fix for this if you place the cursor on the draft table name.

4. Analyze the generated database table for draft data and open the structure that is included with name `%admin`.

Which fields of the draft table are not fields of the corresponding database table for active data?

5. Activate the definition of the draft table.
6. Return to the behavior definition. After addition `lock master`, add addition `total etag` followed by the name of the field that is still used in the `etag master` addition.
7. Edit the `etag master` addition and replace the field there with the name of the field you created in the previous task.

8. If the name of the new field is not identical in the database table and the CDS view, scroll down to the list of field mappings and add a mapping for the new Etag master field.
9. Explicitly declare the draft actions *Edit, Activate, Discard, Resume*.



Note:

This is not yet necessary in our training system, because the RAP framework implicitly declares the draft actions. However, in future releases, it will become mandatory to explicitly declare the draft actions when implementing the behavior definition in strict mode.

10. Activate the behavior definition.

Enable Draft Handling for a RAP Business Object

Business Scenario

In this exercise, you enable draft handling for your RAP Business Object and in your SAP Fiori elements App. To support optimistic concurrency control for draft instances, you extend the data model with a field that is updated during every save of a draft.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 13: Solution

Repository Object Type	Repository Object ID
Database Table	D437D_TRAVEL
CDS Data Definition (Model)	D437D_I_TRAVEL
CDS Data Definition (Projection)	D437D_C_TRAVEL
CDS Metadata Extension	D437D_C_TRAVEL
CDS Behavior Definition (Model)	D437D_I_TRAVEL

Task 1: Extend Data Model with Timestamp Field for Draft Changes

Extend your database table (`Z##_TRAVEL`) with an additional timestamp field (suggested name: `loc_changed_at`). Add it to your data model view (`Z##_I_Travel`) and annotate it with the necessary sub-annotation of annotation `@Semantics` to ensure the RAP run time automatically updates the field during every save of a draft. Add the field to your projection view (`Z##_C_Travel`) and its metadata extension, to keep it hidden in the SAP Fiori elements app.

1. Edit the definition of your transparent table (`Z##_TRAVEL`). At the end of the field list, add a field `loc_changed_at` and type it with data element `S_LOCCHANGEDAT`.
 - a) In the code based eclipse editor, add `loc_changed_at : s_locchangedat;` before the closing curly bracket.
2. Activate the transparent table.
 - a) Perform this step as in previous exercises.
3. Edit the definition of your CDS view entity(`Z##_I_TRAVEL`) and add the new table field to the element list.

**Note:**

If you want to use a longer name in CDS and RAP (for example `LocalChangedAt`), specify the longer name after addition AS.

- a) See the source code extract from the model solution.
- b) Add a comma after the last element of the element list.
- c) In a new code line, add `loc_changed_at` as `LocalChangedAt`.
4. Add annotation `@Semantics.systemDateTime.localInstanceLastChangedAt: true` to the new view field.
 - a) Add the annotation in the code line before the view field.
5. Activate the data definition.
 - a) Perform this step as in previous exercises.
6. Edit the definition of your projection view (`Z##_C_TRAVEL`), add the new element to the element list and activate the data definition.
 - a) Add a comma after the last element of the element list.
 - b) In a new code line, add `LocalChangedAt`.
7. Edit the metadata extension for your projection view (`Z##_C_TRAVEL`), add the new element with annotation `@UI.hidden: true`, and activate the metadata extension.
 - a) Before the closing curly brackets, add `@UI.hidden: true` followed by `LocalChangedAt;`
8. Restart your SAP Fiori elements app and verify that everything still works.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution.

Data Definition **D437D_I_TRAVEL**:

```
...
define root view entity D437d_I_Travel
  as select from d437d_travel
{
  ...
    @Semantics.user.lastChangedBy: true
    changed_by as ChangedBy,
    @Semantics.systemDateTime.localInstanceLastChangedAt: true
    loc_changed_at as LocalChangedAt
}
```

Data Definition **D435D_C_TRAVELTP**:

```
...
define root view entity D437d_C_Travel
```

```

        as projection on D437d_I_Travel

{
...
ChangedBy,
LocalChangedAt
}

```

Metadata Extension D437D_C_TRAVEL:

```

...
annotate view D437D_C_Travel
with
{
...
@UI.hidden: true
LocalChangedAt;
}

```

Task 2: Enable Draft-Handling for the RAP Business Object

In the behavior definition for your data model view (`Z##_I_Travel`), add the necessary statement to enable draft handling for the Business Object. Define and generate a draft table for the travel data (suggested name: `Z##_TRAVEL_D` and define the view field you just created as total etag field.

1. Edit the behavior definition for your data model view (`Z##_I_TRAVEL`) and add statement `with draft;` immediately after statement `managed ... ;`
 - a) See the source code extract from the model solution.
2. Scroll down to the `define behavior` statement and add addition `draft table`, followed by the name of the draft table you want to create.
 - a) See the source code extract from the model solution.
3. Create the draft table.



Hint:

The editor offers a quick fix for this if you place the cursor on the draft table name.

- a) Place the cursor on the name of the draft table.
- b) Open the context menu and choose *Quick Fix* or press `Ctrl + 1`.
- c) Double-click the proposed quick fix.
- d) Enter a description for the draft table and choose *Next >*.
- e) Select a transport request and choose *Finish*.

4. Analyze the generated database table for draft data and open the structure that is included with name %admin.
- a) The definition of the database table is already opened after the previous step.

Which fields of the draft table are not fields of the corresponding database table for active data?

The additional fields are the fields of named include %admin.

5. Activate the definition of the draft table.
- a) Perform this step as before.
6. Return to the behavior definition. After addition `lock master`, add addition `total etag` followed by the name of the field that is still used in the `etag master` addition.
- a) See the source code extract from the model solution.
7. Edit the `etag master` addition and replace the field there with the name of the field you created in the previous task.
- a) See the source code extract from the model solution.
8. If the name of the new field is not identical in the database table and the CDS view, scroll down to the list of field mappings and add a mapping for the new Etag master field.
- a) See the source code extract from the model solution.
9. Explicitly declare the draft actions *Edit,Activate,Discard,Resume*.



Note:

This is not yet necessary in our training system, because the RAP framework implicitly declares the draft actions. However, in future releases, it will become mandatory to explicitly declare the draft actions when implementing the behavior definition in strict mode.

- a) See the source code extract from the model solution.
10. Activate the behavior definition.
- a) Perform this step as before.
- b) Compare your code to the following extract from the model solution.

Behavior Definition **D437D_I_TRAVEL**:

```
managed implementation in class bp_d437d_i_travel unique;
with draft;

define behavior for D437d_I_Travel alias Travel
persistent table d437d_travel
draft table d437d_travel_d
lock master
total etag ChangedAt
etag master LocalChangedAt
authorization master ( instance )
{
```

```
...
draft action Edit;
draft action Activate;
draft action Discard;
draft action Resume;

mapping for d437d_travel corresponding
{
...
    LocalChangedAt = loc_changed_at;
}
}
```



LESSON SUMMARY

You should now be able to:

- Explain the need for draft in stateless applications
- Enable draft handling in the Business Object

Developing Draft-Enabled Applications



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Enable draft handling in a SAP Fiori elements app
- Explain the difference between transition messages and state messages
- Describe the draft-specifics in behavior implementations

Draft in SAP Fiori Elements



Behavior Definition (Projection)

```
projection;  
use draft;  
  
define behavior for ...  
...  
{  
...  
  
use action Edit;  
use action Activate;  
use action Discard;  
use action Prepare;  
use action Resume;  
}  
}
```

The diagram illustrates the projection of draft-handling in SAP Fiori Elements. It shows a code snippet for a Behavior Definition (Projection) with annotations explaining specific parts of the code:

- Include RAP draft-handling in projection**: Points to the `use draft;` statement.
- Total ETag and draft table automatically reused**: Points to the code block following `define behavior for ...`.
- Projection mandatory in future releases (strict mode)**: Points to the code block containing `use action` statements.

Figure 133: Projection of Draft-Handling

RAP draft handling can be reused with the syntax element `use draft`. As a prerequisite, the underlying RAP BO must be draft-enabled. The draft tables and the total ETag field are implementation details that are automatically reused and do not have to be explicitly specified.

Draft actions are reused implicitly, but it is recommended that they are listed explicitly, using the syntax element `use action`.



Note:

In releases after ABAP 7.55, it will become mandatory to explicitly specify all of the draft actions when using the strict mode.



Travel in the future

Travel in the future

Travel

Travel agency no.:

55

Flight Travel Number:

12130

Travel Description:

Travel in the future

Customer Number:*

1

Travel Start Date:*

MMM d, y

Travel End Date:*

Sep 29, 2021

Flight Travel Status:

-

Mandatory field missing

But draft with initial value saved

Draft saved

Save

Cancel

Figure 134: Editing a Draft Instance

When you edit data in a draft enabled SAP Fiori elements application, the framework will save the user entries in a draft instance - even if the data is inconsistent or incomplete.

The application indicates this at the bottom of the Object Page, next to the Save and Cancel buttons.

When the user chooses Save, the framework checks if the draft is consistent (determine action PREPARE) and, if it is, copy the draft to the active data (action ACTIVATE).

When the user chooses Cancel, the framework will discard the draft instance (action DISCARD).

When the user navigates back, closes the application, or in the case of any failure, the draft will remain and the user can pick up editing any time.

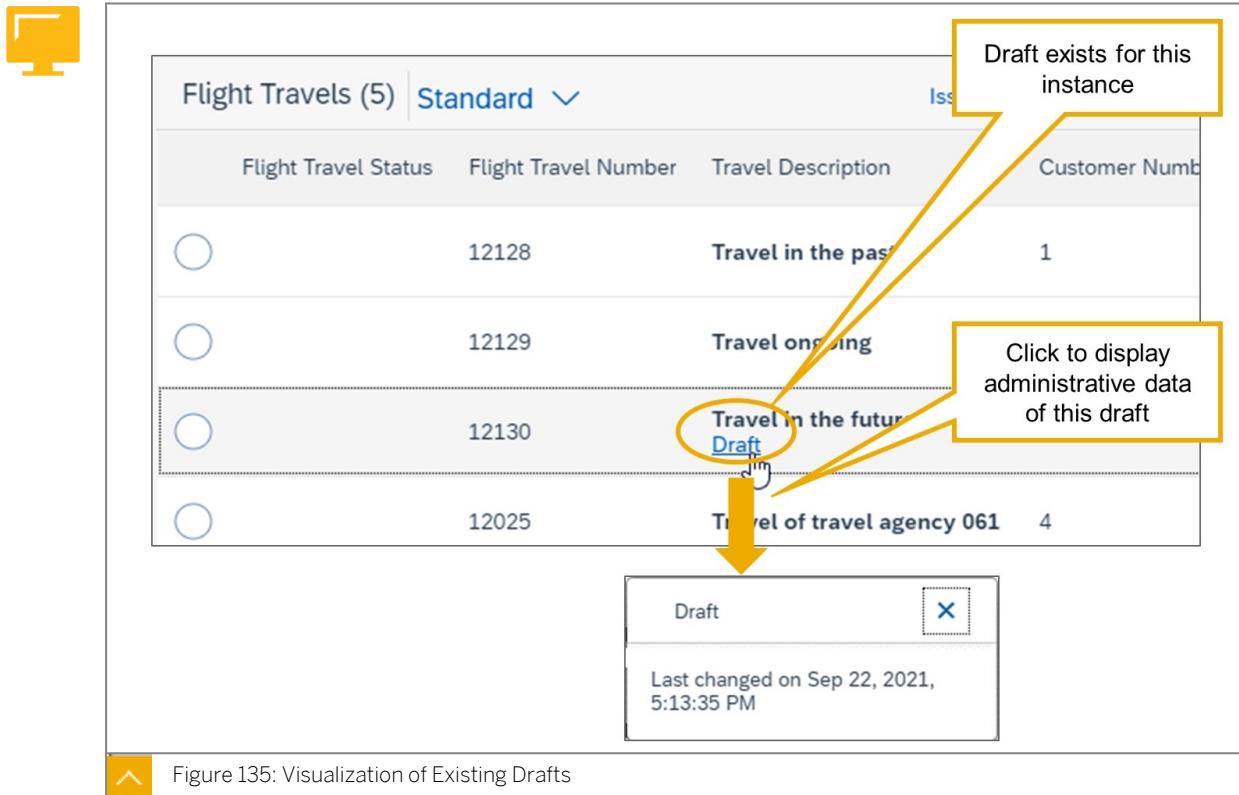


Figure 135: Visualization of Existing Drafts

On the list, a *Draft* link below the text field of a node indicates that this entry is a draft instance. Choosing the link displays a dialog window with administrative data of the draft.

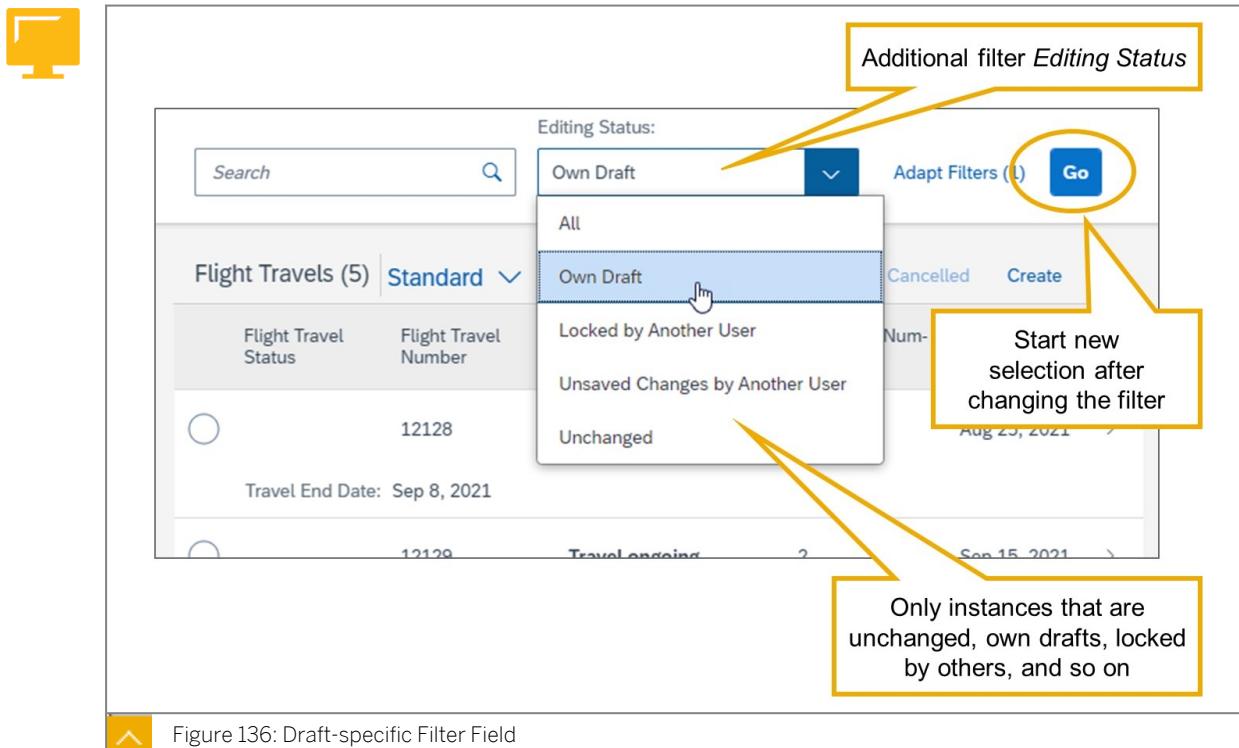


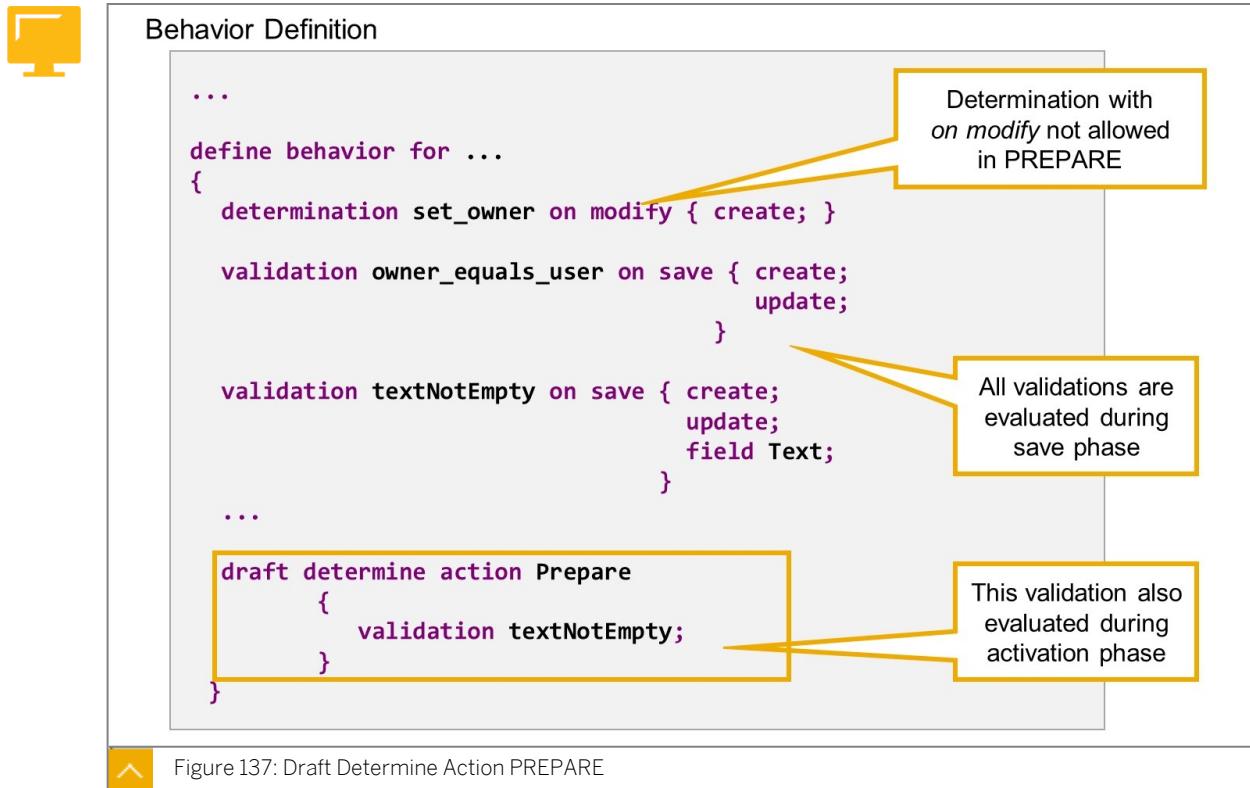
Figure 136: Draft-specific Filter Field

A draft enabled SAP Fiori app displays an additional filter field, *Editing Status*, by which the user can select draft instances or active, unchanged instances, only.

Note:

Changing the filter does not trigger a new selection, directly. You have to choose Go to change the displayed data.

Validations During Prepare



All validations, and the determinations defined as `on save`, are automatically evaluated during the save phase. This means that, just before the active data from the application buffer is written to the persistent database table, those validations and determinations are executed, for which the trigger conditions are fulfilled.

A developer can allow the RAP BO consumer to execute determinations and validations on request by defining a determine action and assigning determinations and validations to it.

Then, the validations and determinations assigned to the determine action are already evaluated whenever the determine action is executed.

The purpose of the implicitly defined draft determine action PREPARE is to validate draft data not only when they are persisted on the database but already before the transition to active data. It is implicitly executed by draft action ACTIVATE.

Note:

An RAP BO consumer can explicitly execute PREPARE any other time to check the consistency of the draft instance.

Like all other draft actions, PREPARE is implicitly enabled as soon as the business object is draft enabled, but, in this case, no determinations and validations are assigned to it. The

assignment of determinations and validations must be done explicitly in the behavior definition. To assign validations and determinations, add a pair of curly brackets after the action name and list the validations and determinations there.

The following restrictions apply:

- Only determinations and validations that are defined and implemented for the BO can be used.
- Only determinations defined as on save can be assigned.

Transition Messages and State Messages



	Transition Message	State Message
Refers to	Trigger request	BO instance and data
Definition	<code>%state_area initial</code>	<code>%state_area not initial</code>
Binding	Entity Instance or general	Always Entity Instance
Lifetime	While displayed	Until BO state changes
Visualization with Draft	Pop-up message, no highlighting of fields	Pop-over message, highlight related fields (if any)
Visualization without Draft	Pop-over message, highlight related fields (if any)	

Figure 138: State Messages and Transition Messages

RAP distinguishes Transition Messages and State Messages. While transition messages refer to a triggered request, state messages refer to a business object instance and its values.

A typical example for a transition message could be "Business Object is locked by user &1", which relates to a triggered request (Edit) and to an (unsuccessful) transition from display to edit mode.

A typical example for a state message could be "The order date &1 lies in the past" which relates to an invalid value in a field and an inconsistent state of the business object instance.

State messages are defined when the `%state_area` component in the REPORTED structure is filled with a string value. Messages with an empty `%state_area` are treated as transition messages. Note that this means all our messages so-far were transition messages.

Transition messages can either be bound to an RAP BO entity instance or be more general, that is, entered in component `%others` of the REPORTED structure. State messages must always be bound to an entity instance. They are not allowed in the component `%others`.

The most important difference between state messages and transition messages is the message lifetime and the visualization on the UI in draft scenarios.

In draft scenarios, a transition message appears as a pop-up message and is gone once the pop-up window is closed.

State messages are displayed in a message pop-over until the state of the business object changes. If a message is assigned to a field in %ELEMENT, the respective field is framed in the severity color to illustrate the link between the field values and a message in order to improve the user experience. For a business object with draft capabilities, state messages are persisted until the state that caused the message is changed and in a managed scenario, the messages are buffered until the end of the session.



Note:

In non-draft scenarios, SAP Fiori makes no difference in the visualization of transition messages and state messages.

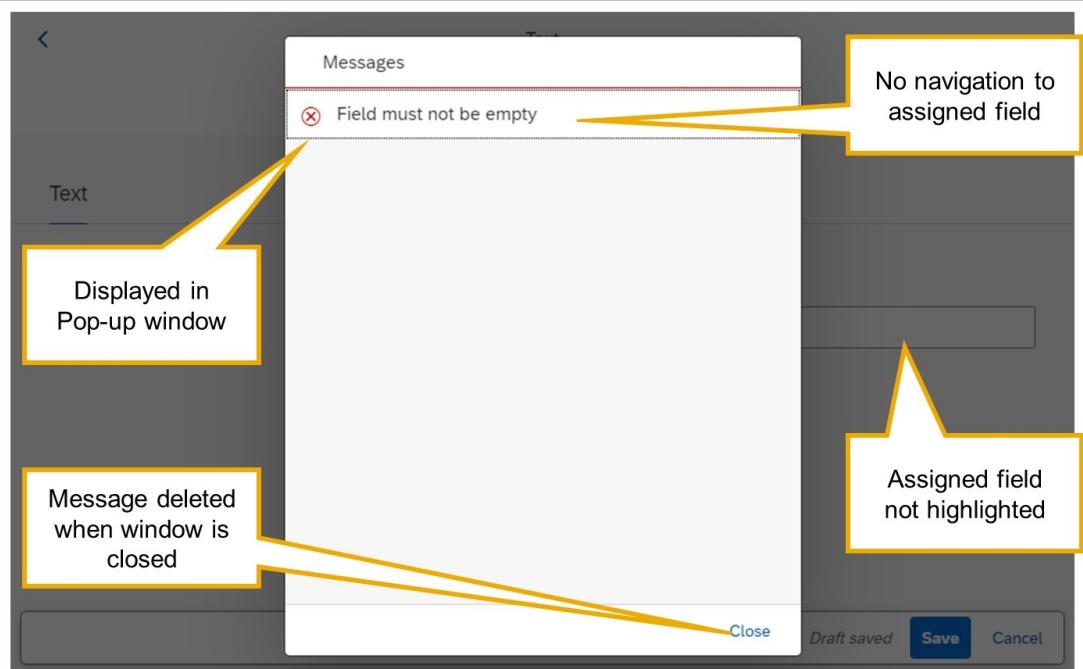


Figure 139: Example: Transition Message in Draft-enabled SAP Fiori Apps

The example shows the display of a transition message in a draft-enabled SAP Fiori application. The message is displayed in a pop-up window that blocks the application until closed by the user. When closing the window, the messages are deleted. Even though the message is connected to a field, because the application logic reported it with a non-initial structure %element, this connection is not visualized, neither is there a link to navigate from the message to the field nor is the field highlighted, for example with a red border.

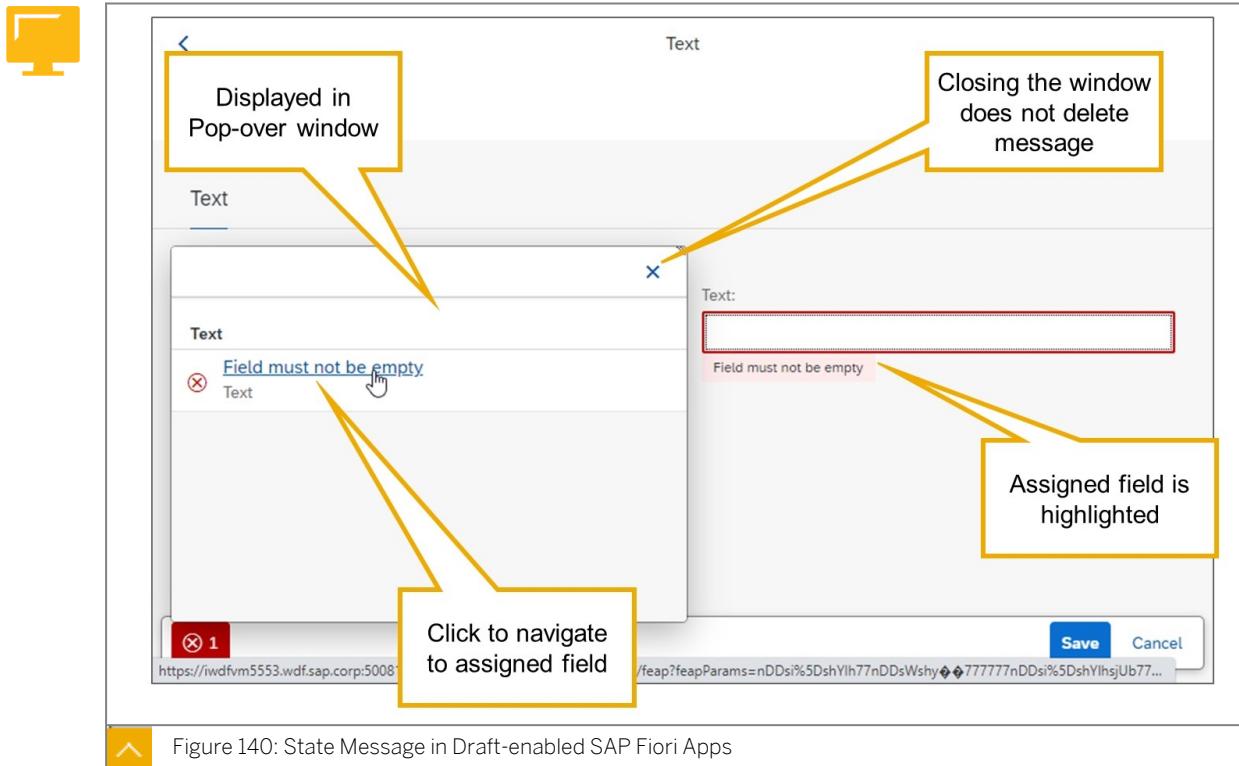


Figure 140: State Message in Draft-enabled SAP Fiori Apps

This next example shows the display of the same message, but this time it was reported as a state message.

The message is displayed in a pop-over window that does not block the application. The user can close the window but this does not delete the message. The connection to the assigned field is visualized by a navigation link on the message text and a red border to highlight the field.



Note:

In the current release of our training system (ABAP 7.55), there is a bug with the display of state messages coming from validations assigned to draft determine action PREPARE. Each message is displayed twice, once in the pop-over window, as described above, and additionally in a pop-up window like a transition message. The issue is reported and under investigation. For the moment, simply close the pop-up window with the superfluous messages.



```

METHOD my_method.

DATA ls_wa LIKE LINE OF reported-text.
...
* Invalidating all message of area 'MY_AREA'
CLEAR ls_wa.
ls_wa-%tky = ....
ls_wa-%state_area = `MY_AREA`.
APPEND ls_wa TO reported-text.
...
• Report a state message
ls_wa-%tky = .... .
ls_wa-%msg = .... .
ls_wa-%element-text = if_abap_behv=>mk-on.
ls_wa-%state_area = `MY_AREA` .
APPEND ls_wa TO reported-text.
ENDMETHOD.

```

Row with %tky
and %state_area
but initial %msg

Deletes all state
messages of
this area for

Non-initial value:
this is a
state message

Figure 141: Creating and Invalidating State Messages

A message becomes a state message when the %state_area component in the REPORTED structure is filled with a non-initial value. You can choose any string value but it is recommended that you stick to ASCII characters.

In draft scenarios, state messages are persisted with the draft data and, in managed scenarios, they are buffered until the end of the session. If the same request, for example, a validation, is triggered multiple times on the same instance, the same messages will be added to the message table again and again. To avoid this, you have to invalidate state messages explicitly.

In managed scenarios, it is sufficient to add a special row to the related component of REPORTED. This row should only contain a value for the key (%tky) of the entity instance and the state area ID (%state_area). All other components like %msg, %element, and so on, remain initial. With this entry, you delete all messages of the same state area for the specified entity instance.



Note:

In unmanaged scenarios, additional coding is needed in the implementation of the DELETE operation, to make sure that the related state messages are removed when deleting a draft instance.

The value for %state_area is only used to group state messages that are related and should be invalidated together. The value is not displayed on the UI nor is it contained in the OData metadata.

For the sake of readability, we recommend choosing a name that uniquely identifies the condition that the message originates from. For example, if a validation checks if a customer ID is valid, the %state_area 'Invalid_Customer' can be helpful in characterizing the condition

because of which the validation failed. Alternatively, you can choose the name of the operation a message is thrown in as `%state_area`, for example `'Validate_Customer'`.



Hint:

Define constants for the state area IDs to avoid typos and facilitate refactoring.

Implementation Aspects of Draft



Active Instance Data

Row	<code>%IS_DRAFT</code>	TRGUID	AGENCYID	TRAVELID	TRAVELDESCRIPTION	CUSTOMERID	STA
1	00	0050561972201EDC878DD72D760EA103	00000055	00012143	Travel in the future	00000003	202



Draft Instance Data

Row	<code>%IS_DRAFT</code>	TRGUID	AGENCYID	TRAVELID	TRAVELDESCRIPTION	CUSTOMERID	START
1	01	0050561972201EDC878DD72D760EA103	00000055	00012143	Travel in the future	00000003	20210

Different value
in `%IS_DRAFT`

Same technical
key (Trguid)

Same semantic key
(AgencyID, TravelID)

Figure 142: Key Component `%IS_DRAFT`

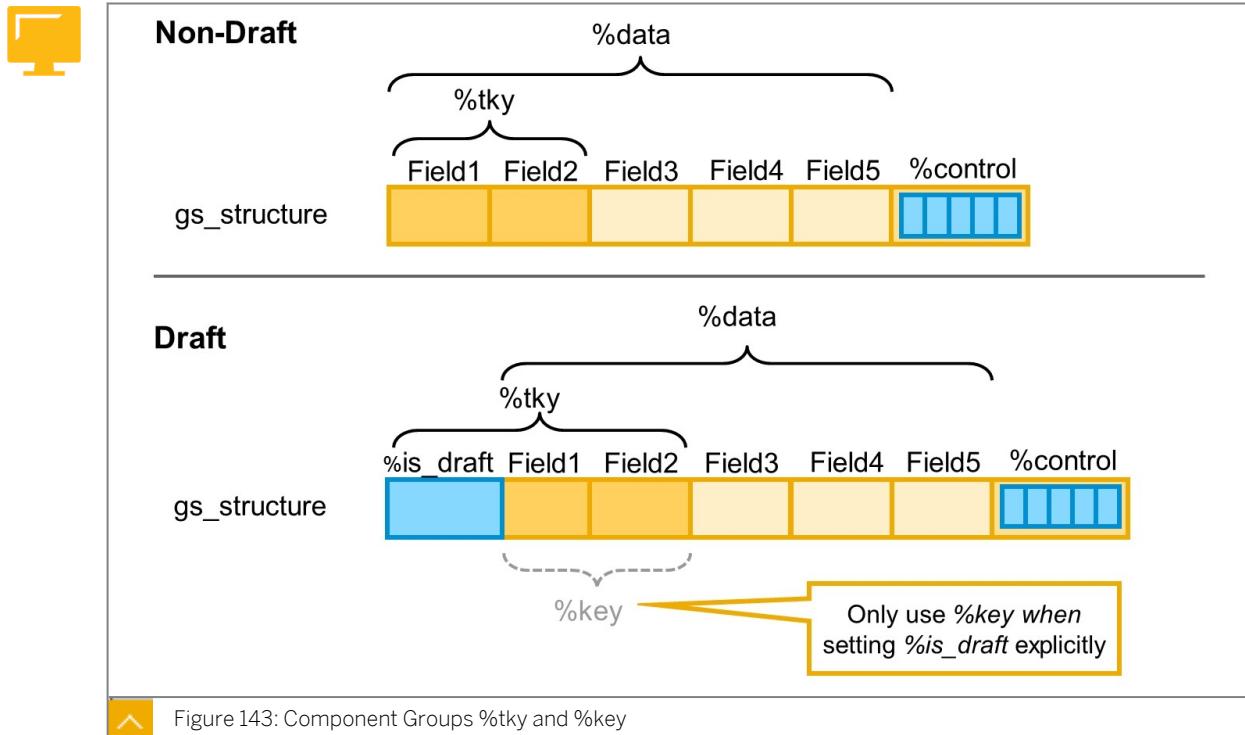
If a RAP Business Object is draft enabled, all derived types for its entities contain an additional component `%IS_DRAFT` that is used to distinguish between active instances and draft instances.

The example shows screenshots from the table display tool in the ABAP debugger.

The draft indicator `%IS_DRAFT` is typed with data element `abp_behv_flag` (technical type `X(1)`) and can assume two different values, which can be found in constant structure `if_abap_behv=>mk`. For a draft instance, `%IS_DRAFT` equals `if_abap_behv=>mk-on (#01)`, and `if_abap_behv=>mk-off (#00)` for active instances.

In RAP, an edit-draft is created by copying all fields of an active instance. In particular, the primary key fields have identical values in a draft instance and the corresponding active instance. The only way to distinguish between draft and active data is the value of `%IS_DRAFT`.

Because of this, `%IS_DRAFT` must be treated like an additional key field that is mandatory when accessing data via EML and in RAP implementations. The framework supports this by automatically including component `%IS_DRAFT` in the component group `%tky`.

Figure 143: Component Groups `%tky` and `%key`

If you use `%tky` to address the primary key field of an entity, you do not have to change your business logic implementation when draft-enabling the business object. The business functionality runs smoothly without adapting your code after draft-enabling your business object.

If you used field group `%key` in your business logic implementation, or addressed the key fields directly via their individual component names, you have to revise the implementation when draft-enabling the business object.



Note:

The recommendation is to only use `%tky` in your business logic implementation, unless you want to read the active instance for a draft instance



```

METHOD get_features.

DATA lt_read_in  TYPE TABLE FOR READ IMPORT ....
DATA lt_read_out TYPE TABLE FOR READ RESULT ....
DATA ls_read_in LIKE LINE OF lt_read_in.
DATA ls_key like line of keys.

LOOP at keys into ls_key.
  CLEAR ls_read_in.

*   gs_read_in-%tky = key-%tky.
*   ls_read_in-%key    = ls_key-%key.
*   ls_read_in-%is_draft = if_abap_behv=>mk-off.

  APPEND ls_read_in TO lt_read_in.
ENDLOOP.

READ ENTITY IN LOCAL MODE ...
  FIELDS ( ... )
  WITH  lt_read_in
  RESULT lt_read_out
  ...

ENDMETHOD.

```

Copying %tky would copy %is_draft, too

Copy %key and set %is_draft explicitly

Read active data only

Figure 144: Example: Feature Control for Draft Instance

In the behavior implementation for a draft-enabled business object, import parameter `keys` always contains the technical key fields and the draft indicator `%is_draft`. When you use component group `%tky` to setup the input for a `READ ENTITY` statement, you read draft data for draft instances and active data for active instances.

There can be situations where it becomes necessary to read the active data for a draft instance and not the draft data itself. A good example is the implementation of instance feature control.

Let's consider a sales order that becomes read-only when having a certain status (cancelled, delivered, and so on). When feature control is based on the draft data, the draft becomes read-only as soon as the user changes the status in the draft. If the status change was done accidentally, the user has no chance to undo it in the current edit process. The only remaining option is to cancel the draft and start editing again. But if feature control is based on the active instance, the draft data remain editable until the active data is updated.

To read the related active data for draft instance, use component group `%key` instead of `%tky` and set the draft indicator `%is_draft` explicitly.



Note:

For readability reasons, we recommend setting the draft indicator to `if_abap_behv=>mk-off` instead of leaving it initial, even though the result is the same.

Unit 4 Exercise 14

Enable Draft Handling in SAP Fiori Elements App and Adjust Implementations

Business Scenario

In this exercise, you expose the draft capabilities of your RAP Business Object to your OData UI Service and test the draft functionality in your SAP Fiori elements app. You then make necessary adjustments to the behavior definition and implementation.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 14: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Projection)	D437D_C_TRAVEL
CDS Behavior Definition (Model)	D437D_I_TRAVEL
ABAP Class	BP_D437D_I_TRAVEL

Task 1: Draft Handling in SAP Fiori Elements App

Make the draft capabilities of your RAP Business Object visible for your SAP Fiori elements app. To achieve this, add the draft functionality to the behavior project, then test the preview of the draft enabled app.

1. Edit the behavior projection, that is, the behavior definition for your projection view (`Z##_C_Travel`) and add the statement `use draft;` immediately after the statement `projection;`.
2. Explicitly include the five draft actions into the projection to make them part of the OData Service.



Note:

This is not yet necessary in our training system, because the RAP framework implicitly includes the draft actions into the projection. However, in future releases, it will become mandatory to explicitly include the draft actions when implementing the behavior projection in strict mode.

3. Activate the behavior definition.

4. Restart your SAP Fiori elements app and create a new flight. Leave all input fields empty and return to the *Report List* page without choosing *Create* or *Cancel*.
5. Display the content of the draft table (`Z##_TRAVEL_D`) in the data preview tool and verify that it contains the draft data of the new travel.

Which fields have already been filled?

Why are administrative fields `CHANGEDAT` and `CHNAGEDBY` not filled, yet?

6. Go back to your SAP Fiori elements app and resume editing the new flight travel.
7. Leave all input fields empty but this time choose *Save*.

What is the problem with the error messages from the validations?

What needs to be done to fix the display of the validation messages?

Task 2: Adjust Validations to Draft

Adjust the behavior definition of your RAP Business Object. Attach all validations for the draft determine action `Prepare` to ensure that they are called before draft instances are transferred to active instances. Adjust the implementation of all validations to report messages of type `state` instead of messages of type `transition`.

1. Edit the behavior definition for your data model view (`Z##_I_TRAVEL`). Within the curly brackets after the statement `define behavior`, add draft determine action `Prepare { ... } .`
2. Within the curly brackets of the new statement, list all validations of your RAP Business Object.
3. Activate the behavior definition.
4. Navigate to the implementation of method `validatecustomer()` in the local handler class.
5. Edit the implementation and fill the component of response parameter `reported`. Make sure to fill component `%state_area` with a not initial value, for example `CUSTOMER`.

**Note:**

To increase robustness and supportability of your code, we recommend defining a local constant (suggested name: `c_state`) with this value.

6. Scroll up to the beginning of the loop. Before the actual check, add a row to the same component of response parameter `reported` with only the total transactional key (`%tky`) and the same value for the state area (`%state_area`). Make sure that the component `%msg` is empty, so no message is reported with this entry.

**Note:**

This entry removes all previous state messages that were reported for the same draft instance and with the same value for `%state_area`.

7. Edit the implementation of method `validatestartdate()`. For all error messages that are reported by this validation, fill component `%state_area` with a suitable value, for example `STARTDATE`.
8. At the beginning of the loop, before the actual check, remove all previous state messages that were reported for the same draft instance and with the same value for `%state_area`.
9. Adjust the implementation of methods `validateenddate()` and `validatesequence` accordingly. Use a different value for `%state_area` in each method.
10. Activate the global class with the behavior implementation.

Task 3: Adjust Dynamic Feature Control

Adjust the implementation of method `get_features()` to draft. For edit draft instances (not new draft instances), the action and field control should be based on start date and end data from the corresponding active instance and not on the values from the draft instance itself.

1. Edit the implementation of method `get_features()`. Locate the loop over the affected flight travels. At the beginning of the loop, check whether the current flight travel is an active instance or a draft instance.

**Hint:**

Compare the key field `%is_draft` with corresponding values from the constant structure `mk` in the interface `of_abap_behv`.

2. If it is an active instance, let a new field symbol (suggest name: `<ls_for_check>`) point to the same data as `<ls_travel>`.
3. If the current flight travel is a draft instance, read the corresponding active instance.

**Hint:**

Use the statement `READ ENTITY` or `READ ENTITIES` with the same value for `%key` but with the initial value for `%is_draft`.

4. If the draft is a new draft, that is, if there is no related active instance, let the field symbol `<ls_for_check>` point to the same data as `<ls_travel>`.
5. If the draft is an edit draft, that is, if there is a related active instance, let the field symbol `<ls_for_check>` point to the active data.
6. In the logical expressions for feature control, replace the field symbol `<ls_travel>` with `<ls_for_check>`.



Caution:

Make sure that you still use the key in the field symbol `<ls_travel>` to set the key fields in the new entry for response parameter `result`.

7. Activate the behavior implementation.

Enable Draft Handling in SAP Fiori Elements App and Adjust Implementations

Business Scenario

In this exercise, you expose the draft capabilities of your RAP Business Object to your OData UI Service and test the draft functionality in your SAP Fiori elements app. You then make necessary adjustments to the behavior definition and implementation.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 14: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Projection)	D437D_C_TRAVEL
CDS Behavior Definition (Model)	D437D_I_TRAVEL
ABAP Class	BP_D437D_I_TRAVEL

Task 1: Draft Handling in SAP Fiori Elements App

Make the draft capabilities of your RAP Business Object visible for your SAP Fiori elements app. To achieve this, add the draft functionality to the behavior project, then test the preview of the draft enabled app.

1. Edit the behavior projection, that is, the behavior definition for your projection view (`Z##_C_Travel`) and add the statement `use draft;` immediately after the statement `projection;`.
 - a) See the source code extract from the model solution.
2. Explicitly include the five draft actions into the projection to make them part of the OData Service.



Note:

This is not yet necessary in our training system, because the RAP framework implicitly includes the draft actions into the projection. However, in future releases, it will become mandatory to explicitly include the draft actions when implementing the behavior projection in `strict mode`.

- a) See the source code extract from the model solution.

3. Activate the behavior definition.
 - a) Perform this step as before.
4. Restart your SAP Fiori elements app and create a new flight. Leave all input fields empty and return to the *Report List* page without choosing *Create* or *Cancel*.
 - a) Perform this step as before.
5. Display the content of the draft table (*Z##_TRAVEL_D*) in the data preview tool and verify that it contains the draft data of the new travel.
 - a) Locate the draft table in the *Project Explorer*.
 - b) Open the context menu for the table and choose *Open With ... → Data Preview*.

Which fields have already been filled?

The client field (*MANDT*), the technical key (*TRGUID*), the fields related to a determination (*AGENCYID*, *TRAVELEID*), and the timestamp field for changes to the local instance (*LOCALCHANGEGAT*).

Why are administrative fields *CHANGEDAT* and *CHNAGEDBY* not filled, yet?

They will be filled when the draft is activated, that is, when the draft data are copied to the table for active data.

6. Go back to your SAP Fiori elements app and resume editing the new flight travel.
 - a) Perform this step as before.
7. Leave all input fields empty but this time choose *Save*.

What is the problem with the error messages from the validations?

The error messages are displayed, but they are not related to the input fields.

What needs to be done to fix the display of the validation messages?

You have to attach the validations to the prepare action for drafts and the messages have to be classified as state messages.

- a) Perform this step as before.
- b) Compare your code to the following extract from the model solution.
Behavior Definition **D437D_C_TRAVEL**:

```
projection;  
use draft;
```

```

define behavior for D437d_C_Travel
use etag
{
  ...
  use action Edit;
  use action Activate;
  use action Discard;
  use action Resume;
}

```

Task 2: Adjust Validations to Draft

Adjust the behavior definition of your RAP Business Object. Attach all validations for the draft determine action *Prepare* to ensure that they are called before draft instances are transferred to active instances. Adjust the implementation of all validations to report messages of type `state` instead of messages of type `transition`.

1. Edit the behavior definition for your data model view (`Z##_I_TRAVEL`). Within the curly brackets after the statement `define behavior`, add `draft determine action Prepare { ... } .`
 - a) See the source code extract from the model solution.
2. Within the curly brackets of the new statement, list all validations of your RAP Business Object.
 - a) See the source code extract from the model solution.
3. Activate the behavior definition.
 - a) Perform this step as before.
4. Navigate to the implementation of method `validatecustomer()` in the local handler class.
 - a) Press Ctrl and click the name of the validation `validateCustomer`.
5. Edit the implementation and fill the component of response parameter `reported`. Make sure to fill component `%state_area` with a not initial value, for example `CUSTOMER`.



Note:

To increase robustness and supportability of your code, we recommend defining a local constant (suggested name: `c_state`) with this value.

- a) See the source code extract from the model solution.
6. Scroll up to the beginning of the loop. Before the actual check, add a row to the same component of response parameter `reported` with only the total transactional key (`%tky`) and the same value for the state area (`%state_area`). Make sure that the component `%msg` is empty, so no message is reported with this entry.



Note:

This entry removes all previous state messages that were reported for the same draft instance and with the same value for `%state_area`.

- a) See the source code extract from the model solution.

7. Edit the implementation of method `validatestartdate()`. For all error messages that are reported by this validation, fill component `%state_area` with a suitable value, for example `STARTDATE`.
 - a) See the source code extract from the model solution.
8. At the beginning of the loop, before the actual check, remove all previous state messages that were reported for the same draft instance and with the same value for `%state_area`.
 - a) See the source code extract from the model solution.
9. Adjust the implementation of methods `validateenddate()` and `validatesequence` accordingly. Use a different value for `%state_area` in each method.
 - a) See the source code extract from the model solution.
10. Activate the global class with the behavior implementation.
 - a) Perform this step as before.
 - b) Compare your code to the following extract from the model solution.

Behavior Definition **D437D_I_TRAVEL**:

```
...
define behavior for D437d_I_Travel
persistent table d437d_travel
draft table d437d_travel_d
lock master
total etag ChangedAt
etag master LocalChangedAt
authorization master ( instance )
{
    ...
    draft determine action Prepare
    {
        validation validateCustomer;
        validation validateStartDate;
        validation validateEndDate;
        validation validateSequence;
    }
    ...
}
```

Class **BP_D437D_I_TRAVEL**:

```
METHOD validatecustomer.

* for message object
  DATA lo_msg TYPE REF TO cm_devs4d437.

* work areas for response parameters
  DATA ls_reported_travel LIKE LINE OF reported-travel.
  DATA ls_failed_travel   LIKE LINE OF failed-travel.

* Constant for state area (needed for validation messages in draft)
  CONSTANTS c_state TYPE string VALUE `CUSTOMER`.

* read required data
```

```
*****
READ ENTITY IN LOCAL MODE d437d_i_travel
  FIELDS ( customerid ) WITH CORRESPONDING #( keys )
  RESULT DATA(lt_travel).

  LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

* New for Draft: Add new line to to reported
*           to delete previous messages of same state area
***** 
  CLEAR ls_reported_travel.
  MOVE-CORRESPONDING <ls_travel> TO ls_reported_travel.
  ls_reported_travel-%state_area = c_state .
  APPEND ls_reported_travel TO reported-travel.

  "expression-based alternative
*   APPEND VALUE #( %tky      = <ls_travel>-%tky
*                   %state_area = c_state )
*   TO reported-travel.

* validate data and create message object in case of error
***** 

  IF <ls_travel>-customerid IS INITIAL.

  ...

  ENDIF.

* report message and mark flight travel as failed
***** 
  IF lo_msg IS BOUND.

    CLEAR ls_failed_travel.
    MOVE-CORRESPONDING <ls_travel> TO ls_failed_travel.
    APPEND ls_failed_travel TO failed-travel.

    CLEAR ls_reported_travel.
    MOVE-CORRESPONDING <ls_travel> TO ls_reported_travel.
    ls_reported_travel-%element-customerid = if_abap_behv=>mk-on.
    ls_reported_travel-%msg = lo_msg.
    ls_reported_travel-%state_area = c_state.
    APPEND ls_reported_travel TO reported-travel.

    " expression-based alternative without helper variables
*   APPEND CORRESPONDING #( <ls_travel> )
*     TO failed-travel.
*
*   APPEND VALUE #(
*     %tky      = <ls_travel>-%tky
*     %element = VALUE #( customerid = if_abap_behv=>mk-on )
*     %msg     = NEW cm_devs4d437(
*                   textid      = cm_devs4d437=>customer_not_exist
*                   customerid = <ls_travel>-customerid
*                   severity    = cm_devs4d437=>severity-error
*                 )
*     %state_area = c_state
*   )
*   TO reported-travel.
*
*   CLEAR lo_msg.
  ENDIF.
```

```
ENDLOOP.
```

```
ENDMETHOD.
```

```
METHOD validatestartdate.

CONSTANTS c_area TYPE string VALUE `STARTDATE`.

READ ENTITY IN LOCAL MODE d437d_i_travel
FIELDS ( startdate ) WITH CORRESPONDING #( keys )
RESULT DATA(lt_travel).

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

APPEND VALUE #( %tky      = <ls_travel>-%tky
                %state_area = c_area )
TO reported-travel.

IF <ls_travel>-startdate IS INITIAL.

APPEND CORRESPONDING #( <ls_travel> )
TO failed-travel.
APPEND VALUE #(
    %tky      = <ls_travel>-%tky
    %element = VALUE #( startdate = if_abap_behv=>mk-on )
    %msg     = NEW cm_devs4d437(
        textid   = cm_devs4d437=>field_empty
        severity = cm_devs4d437=>severity-error
    )
    %state_area = c_area
) TO reported-travel.

ELSEIF <ls_travel>-startdate < sy-datum.
" or use cl_abap_context_info=>get_system_date( )

APPEND CORRESPONDING #( <ls_travel> )
TO failed-travel.
APPEND VALUE #(
    %tky      = <ls_travel>-%tky
    %element = VALUE #( startdate = if_abap_behv=>mk-on )
    %msg     = NEW cm_devs4d437(
        textid   = cm_devs4d437=>start_date_past
        severity = cm_devs4d437=>severity-error
    )
    %state_area = c_area
)
TO reported-travel.

ENDIF.

ENDLOOP.
ENDMETHOD.
```

```
METHOD validateenddate.
```

```
CONSTANTS c_area TYPE string VALUE `ENDDATE`.

READ ENTITY IN LOCAL MODE d437d_i_travel
FIELDS ( enddate ) WITH CORRESPONDING #( keys )
RESULT DATA(lt_travel).
```

```

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

APPEND VALUE #( %tky      = <ls_travel>-%tky
                %state_area = c_area )
               TO reported-travel.

IF <ls_travel>-enddate IS INITIAL.

   APPEND CORRESPONDING #( <ls_travel> )
                           TO failed-travel.
   APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element = VALUE #( enddate = if_abap_behv=>mk-on )
      %msg      = NEW cm_devs4d437(
                     textid   = cm_devs4d437=>field_empty
                     severity = cm_devs4d437=>severity-error
                     )
      %state_area = c_area
      )
      TO reported-travel.

ELSEIF <ls_travel>-enddate < sy-datum.
   " or use cl_abap_context_info=>get_system_date( )

   APPEND CORRESPONDING #( <ls_travel> )
                           TO failed-travel.
   APPEND VALUE #(
      %tky      = <ls_travel>-%tky
      %element = VALUE #( enddate = if_abap_behv=>mk-on )
      %msg      = NEW cm_devs4d437(
                     textid   = cm_devs4d437=>end_date_past
                     severity = cm_devs4d437=>severity-error
                     )
      %state_area = c_area
      )
      TO reported-travel.

ENDIF.

ENDLOOP.
ENDMETHOD.

```

```

METHOD validateSequence.

CONSTANTS c_area TYPE string VALUE `SEQUENCE`.

READ ENTITY IN LOCAL MODE d437d_i_travel
  FIELDS ( startdate enddate ) WITH CORRESPONDING #( keys )
  RESULT DATA(lt_travel).

LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

  APPEND VALUE #( %tky      = <ls_travel>-%tky
                  %state_area = c_area )
                 TO reported-travel.

  "Sequence of Dates
  -----
  IF <ls_travel>-startdate IS INITIAL
  OR <ls_travel>-enddate IS INITIAL.
    " ignore empty fields, already covered above

```

```

ELSEIF <ls_travel>-enddate < <ls_travel>-startdate.

APPEND CORRESPONDING #( <ls_travel> )
    TO failed-travel.
APPEND VALUE #(
    %tky      = <ls_travel>-%tky
    %element = VALUE #( startdate = if_abap_behv=>mk-on
                        enddate   = if_abap_behv=>mk-on )
    %msg     = NEW cm_devs4d437(
        textid   = cm_devs4d437=>dates_wrong_sequence
        severity = cm_devs4d437=>severity-error
    )
    %state_area = c_area
)
    TO reported-travel.

ENDIF.

ENDLOOP.
ENDMETHOD.

```

Task 3: Adjust Dynamic Feature Control

Adjust the implementation of method `get_features()` to draft. For edit draft instances (not new draft instances), the action and field control should be based on start date and end date from the corresponding active instance and not on the values from the draft instance itself.

1. Edit the implementation of method `get_features()`. Locate the loop over the affected flight travels. At the beginning of the loop, check whether the current flight travel is an active instance or a draft instance.



Hint:
Compare the key field `%is_draft` with corresponding values from the constant structure `mk` in the interface `of_abap_behv`.

- a) See the source code extract from the model solution.
2. If it is an active instance, let a new field symbol (suggest name: `<ls_for_check>`) point to the same data as `<ls_travel>`.
 - a) See the source code extract from the model solution.
 3. If the current flight travel is a draft instance, read the corresponding active instance.



Hint:
Use the statement `READ ENTITY` or `READ ENTITIES` with the same value for `%key` but with the initial value for `%is_draft`.

- a) See the source code extract from the model solution.
4. If the draft is a new draft, that is, if there is no related active instance, let the field symbol `<ls_for_check>` point to the same data as `<ls_travel>`.
 - a) See the source code extract from the model solution.
 5. If the draft is an edit draft, that is, if there is a related active instance, let the field symbol `<ls_for_check>` point to the active data.

- a) See the source code extract from the model solution.
6. In the logical expressions for feature control, replace the field symbol <ls_travel> with <ls_for_check>.

**Caution:**

Make sure that you still use the key in the field symbol <ls_travel> to set the key fields in the new entry for response parameter result.

- a) See the source code extract from the model solution.
7. Activate the behavior implementation.
- a) Perform this step as before.
- b) Compare your code to the following extract from the model solution.

ABAP class **BP_D437D_I_TRAVEL**:

```

METHOD get_features.

* work area for parameter result
  DATA ls_result LIKE LINE OF result.

* helper objects to shorten the code
  CONSTANTS c_enabled   TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-o-enabled.
  CONSTANTS c_disabled  TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-o-disabled.
  CONSTANTS c_read_only TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-f-read_only.
  CONSTANTS c_mandatory TYPE if_abap_behv=>t_xflag
                VALUE if_abap_behv=>fc-f-mandatory.

  DATA lv_today TYPE cl_abap_context_info=>ty_system_date.

***** 
* Get system date
  lv_today = cl_abap_context_info=>get_system_date( ).

* Read data of all affected
  READ ENTITY IN LOCAL MODE d437d_i_travel
    FIELDS ( status startdate enddate )
    WITH CORRESPONDING #( keys )
    RESULT DATA(lt_travel).

  LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<ls_travel>).

* for draft: distinguish between active and draft

  IF <ls_travel>-%is_draft = if_abap_behv=>mk-off.
    " active instance
    ASSIGN <ls_travel> TO FIELD-SYMBOL(<ls_for_check>).
  ELSE.
    " draft instance
    READ ENTITY IN LOCAL MODE d437d_i_travel
    FIELDS ( status startdate enddate )
    WITH VALUE #( ( %key      = <ls_travel>-%key
                  %is_draft = if_abap_behv=>mk-off
                ) )
  ENDIF.

```

```

        RESULT DATA(lt_travel_active).

        IF lt_travel_active IS INITIAL.
          " new draft
          ASSIGN <ls_travel> TO <ls_for_check>.
        ELSE.
          " edit draft
          READ TABLE lt_travel_active INDEX 1 ASSIGNING <ls_for_check>.
        ENDIF.
      ENDIF.

* Transfer complete key to result table
  ls_result-%tky = <ls_travel>-%tky.

* Dynamic action control
  IF <ls_for_check>-status = 'C'.  "already cancelled
    ls_result-%features-%action-set_to_cancelled = c_disabled.
  ELSEIF <ls_for_check>-enddate IS NOT INITIAL
    AND <ls_for_check>-enddate <= lv_today.
    ls_result-%features-%action-set_to_cancelled = c_disabled.
  ELSE.
    ls_result-%features-%action-set_to_cancelled = c_enabled.
  ENDIF.

* dynamic operation control (update)
  IF <ls_for_check>-status = 'C'.  "already cancelled
    ls_result-%features-%update = c_disabled.
  ELSEIF <ls_for_check>-enddate IS NOT INITIAL
    AND <ls_for_check>-enddate <= lv_today.
    ls_result-%features-%update = c_disabled.
  ELSE.
    ls_result-%features-%update = c_enabled.
  ENDIF.

* dynamic field control (Customer, StartDate)
  IF <ls_for_check>-startdate IS NOT INITIAL
    AND <ls_for_check>-startdate <= lv_today.
    ls_result-%features-%field-startdate = c_read_only.
    ls_result-%features-%field-customerid = c_read_only.
  ELSE.
    ls_result-%features-%field-startdate = c_mandatory.
    ls_result-%features-%field-customerid = c_mandatory.
  ENDIF.

  APPEND ls_result TO result.
ENDLOOP.

ENDMETHOD.

```

Alternative with even more expression-based syntax:

```

METHOD get_features.

DATA(lv_today) = cl_abap_context_info->get_system_date( ).

READ ENTITY IN LOCAL MODE d437d_i_travel
  ALL FIELDS WITH CORRESPONDING #( keys )
  RESULT DATA(lt_travel).

```

```

READ ENTITY IN LOCAL MODE d437d_i_travel
  ALL FIELDS WITH VALUE #( FOR key IN keys
    ( %key = key-%key
      %is_draft = if_abap_behv=>mk-off
    ) )
  RESULT DATA(lt_travel_active)
FAILED DATA(ls_failed).

* for new drafts
  result =
    VALUE #(  FOR row IN ls_failed-travel
      (
        %tky = keys[ KEY entity %key = row-%key ]-%tky
      )
    ).

* for edit draft and active
  result =
    VALUE #( BASE result
      FOR <travel> IN lt_travel_active
      (
        "key
        %tky = keys[ KEY entity %key = <travel>-%key ]-%tky
        "action control
        %features-%action-set_to_cancelled
        = COND #( WHEN <travel>-status = 'C'
          THEN if_abap_behv=>fc-o-disabled
          WHEN <travel>-enddate IS NOT INITIAL
          AND <travel>-enddate <= lv_today
          THEN if_abap_behv=>fc-o-disabled
          ELSE if_abap_behv=>fc-o-enabled
        )
        "operation control
        %features-%update
        = COND #( WHEN <travel>-status = 'C'
          THEN if_abap_behv=>fc-o-disabled
          WHEN <travel>-enddate IS NOT INITIAL
          AND <travel>-enddate <= lv_today
          THEN if_abap_behv=>fc-o-disabled
          ELSE if_abap_behv=>fc-o-enabled
        )
        "field control
        %features-%field-startdate
        = COND #( WHEN <travel>-startdate IS NOT INITIAL
          AND <travel>-startdate <= lv_today
          THEN if_abap_behv=>fc-f-read_only
          ELSE if_abap_behv=>fc-f-mandatory
        )
        %features-%field-customerid
        = COND #( WHEN <travel>-startdate IS NOT INITIAL
          AND <travel>-startdate <= lv_today
          THEN if_abap_behv=>fc-f-read_only
          ELSE if_abap_behv=>fc-f-mandatory
        )
      )
    ).

ENDMETHOD.

```



LESSON SUMMARY

You should now be able to:

- Enable draft handling in a SAP Fiori elements app
- Explain the difference between transition messages and state messages
- Describe the draft-specifics in behavior implementations

UNIT 5

Transactional Apps with Composite Business Object

Lesson 1

Defining Composite RAP Business Objects	268
Exercise 15: Define a Composite RAP Business Object	279

Lesson 2

Defining Compositions in OData UI Services	290
Exercise 16: Define a Composite OData UI Service with RAP	297

Lesson 3

Implementing the Behavior for Composite RAP BOs	307
Exercise 17: Implement the Behavior of a Composite RAP Business Object	311

UNIT OBJECTIVES

- Define compositions in RAP BOs
- Expose compositions to OData services
- Enable navigation in SAP Fiori elements apps
- Access composite business objects with EML

Defining Composite RAP Business Objects



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define compositions in RAP BOs

Composite Business Objects in RAP

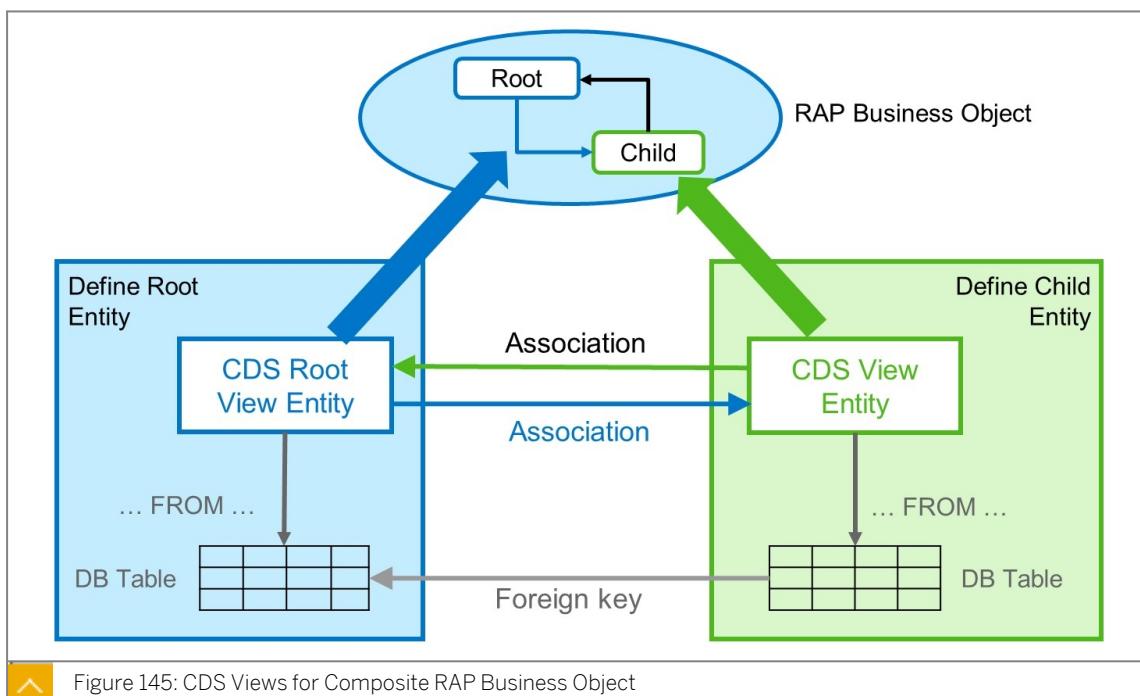


Figure 145: CDS Views for Composite RAP Business Object

Up to now, we worked with RAP Business Objects that consisted of one single node, the root entity. More generally, a Business Object (BO) consists of a hierarchical tree of nodes where each node of is an element that is modeled with a CDS entity and arranged along a composition path.

At runtime, one instance of the BO consists of exactly one instance of the root entity and a variable number of instances of the child entities. A sales order, for example, consists of exactly one header (the root entity instance) and several items (child entity instances).

The hierarchy of entities is defined through two special kinds of associations, namely by Compositions and To-Parent Associations.



Note:

It is not necessary to define foreign key relations between the underlying tables. They are included in the picture to illustrate that the associations in the CDS data model correspond to relations in the relational data model on ABAP Dictionary level.

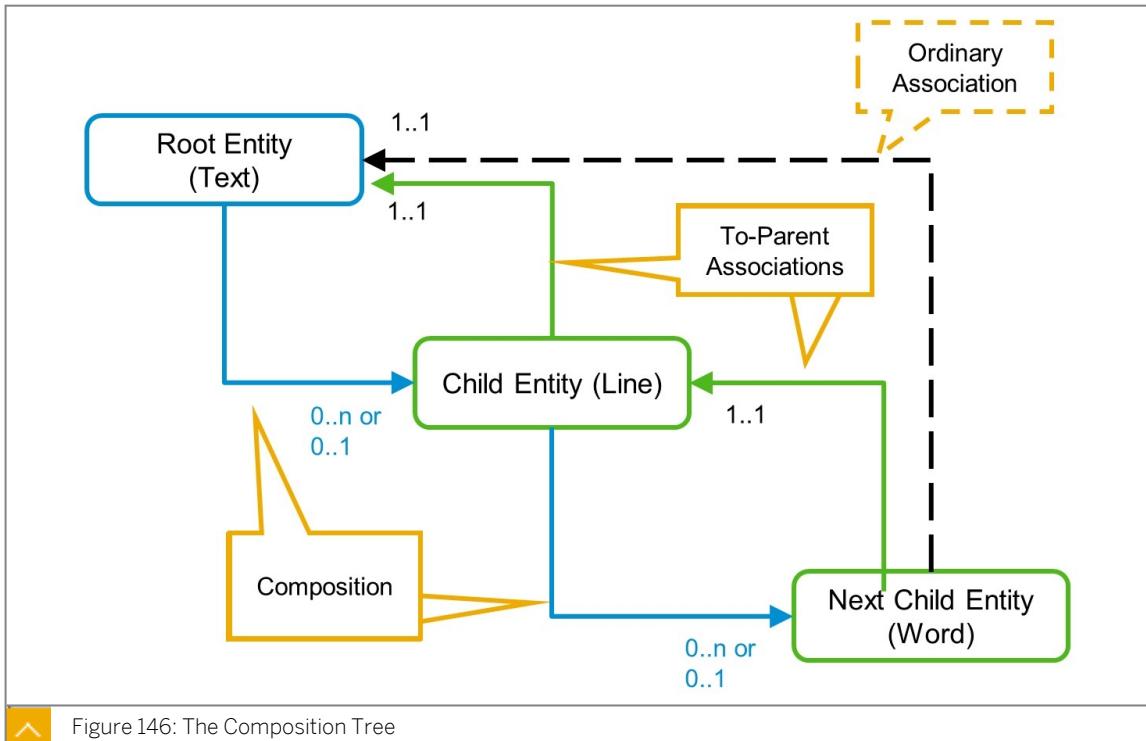


Figure 146: The Composition Tree

In RAP, a composition is a specialized association that defines a whole-part relationship and always leads from the parent to the direct child. A child entity (composite part) only exists together with its parent entity (whole). The definition of a composition always requires the definition of a corresponding to-parent association that leads from the child entity to the direct parent entity.

In the example, the hierarchy consists of three entities, the root entity (Text), its direct child entity (Line) and an indirect child entity (Word), which has the first child entity as its parent. The text is a composition of text lines and each line is a composition of words. For each of the two compositions, there is a corresponding to-parent association.

As well as the compositions and to-parent associations, it is possible to define other relations within the composition tree, for example, from a child entity's child to the root entity. Such relations are defined with ordinary associations. They are not mandatory in general, but might be needed in certain circumstances, for example, to establish a lock master/ lock dependent relation over more than two layers.

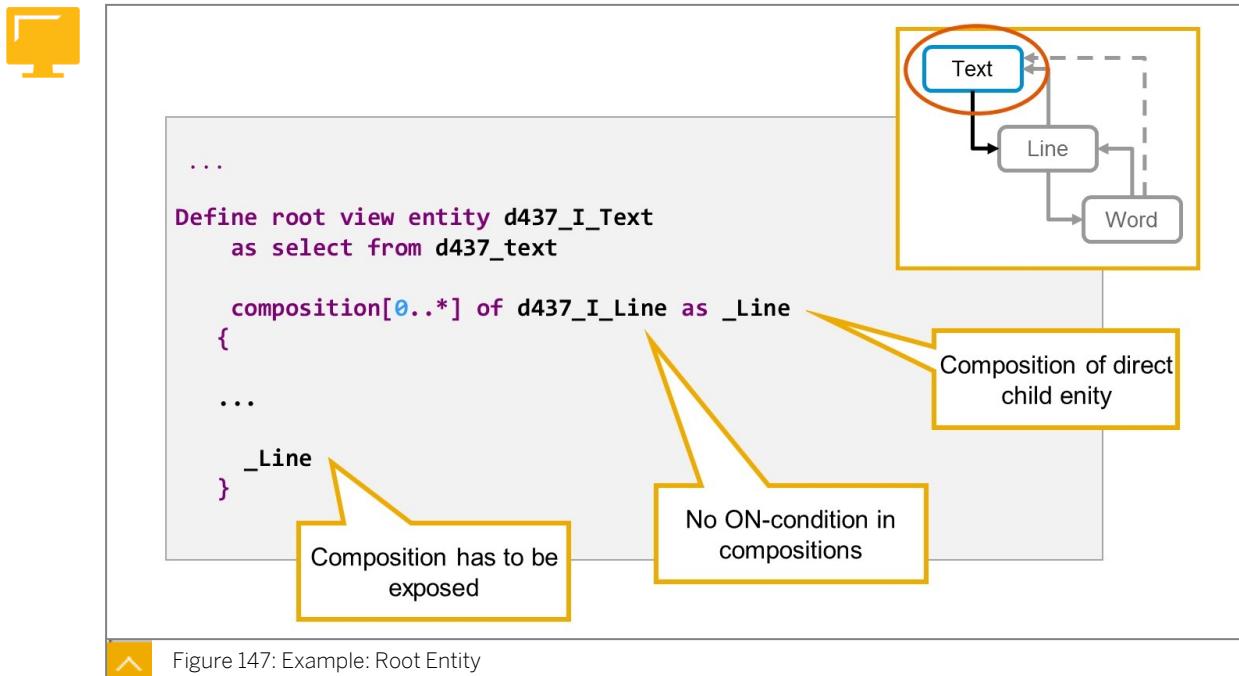
The following restrictions apply when modeling the composition tree of a RAP Business Objects:

- There is exactly one root entity.
- The root entity may be the source but must not be the target of a composition.

- Source and target of a composition are never the same entity

The cardinality of an association expresses how many instances of an entity may be involved in the relationship. It specifies the number of entity instances that are connected to a single source instance and is expressed with a lower bound and an upper bound in the form: x..y (lower_bound..upper_bound). In a RAP BO, the cardinality of the composition can be 0..1 or 0..n, but the cardinality of a to-parent association always has to be 1..1.

CDS Compositions and To-Parent Associations



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 for D437_I_Text with the keyword ROOT.

The root entity (D437_I_Text) serves as the source of a composition which is defined using the keyword COMPOSITION in the corresponding data definition. The target of this composition (D437_I_Line) defines the direct child entity.

CDS compositions are defined similarly to CDS associations. The same rules apply for the cardinality and the name of the composition. The main difference is that for a composition no ON-condition is defined explicitly. The ON condition is generated automatically using the ON condition of the to-parent association of the composition target.

The name of the composition must be added exactly once to the select_list of the CDS view entity it is defined in, without attributes and alias. If no name is defined for the composition, the name of the composition is the name of the target entity target and this name must be made available in the SELECT list.



Caution:

Fields from a composition target can't be used locally in the SELECT list, WHERE clause, or any other position of the view entity in which it is defined.

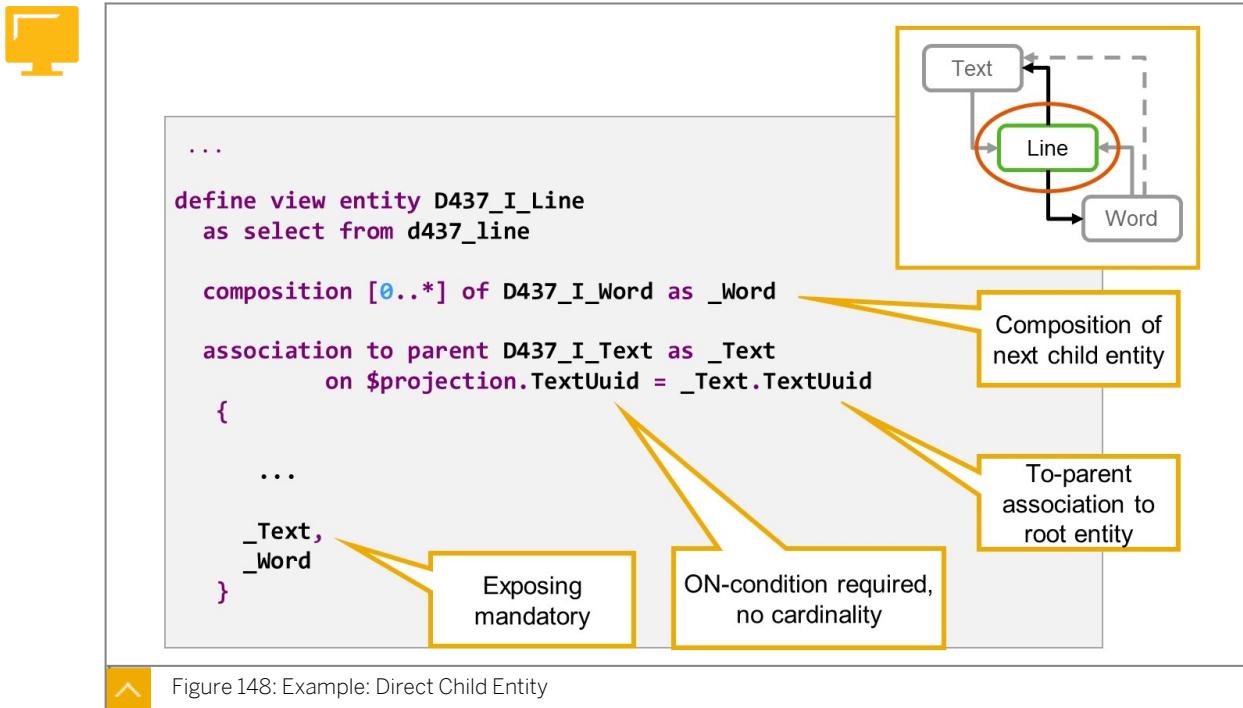


Figure 148: Example: Direct Child Entity

If CDS view entity is the target of a composition, it has to define a CDS to-parent association. The to-parent association is defined using the special syntax ASSOCIATION TO PARENT.

The direct child entity (D437_I_Line) servers as target of a composition and therefore defines a to-parent association to the direct parent entity (D437_I_Text).

CDS to-parent associations are defined similarly to CDS associations. The same rules apply for the name of the association. An ON -condition has to be defined for which some certain rules apply.

The main difference is that for a to-parent association the cardinality cannot be defined explicitly for to-parent associations and is generated as [1..1].

A child entity cannot have more than one to-parent associations but itself be a parent entity and define further compositions. Child entity D437_I_Line, for example, is parent of child entity D437_I_Word.

The name of the to-parent association must be added exactly once to the select list of the CDS view entity it is defined in, without attributes and alias. If no name is defined for the composition, the name of the composition is the name of the target entity target and this name must be made available in the SELECT list.

The following rules apply to the operands and syntax of the ON condition of a to-parent association:

- Only key fields of the parent entity
- All key fields of the parent entity
- Each field of child entity only once
- Fields on child entity with prefix \$projection
- Only comparison with "="
- No OR or NOT

**Hint:**

To avoid syntax errors, it is recommended to define the to-parent association first and the corresponding composition after.



```

...
define view entity D437_I_Word
  as select from d437_word

  association to parent D437_I_Line as _Line
    on $projection.LineUuid = _Line.LineUuid

  association[1..1] to D437_I_Text as _Text
    on $projection.TextUuid = _Text.TextUuid
{
  ...
  _Line,
  _Text
}

```

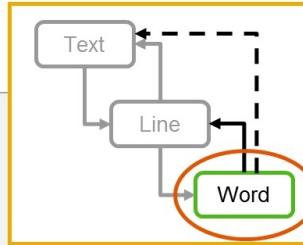


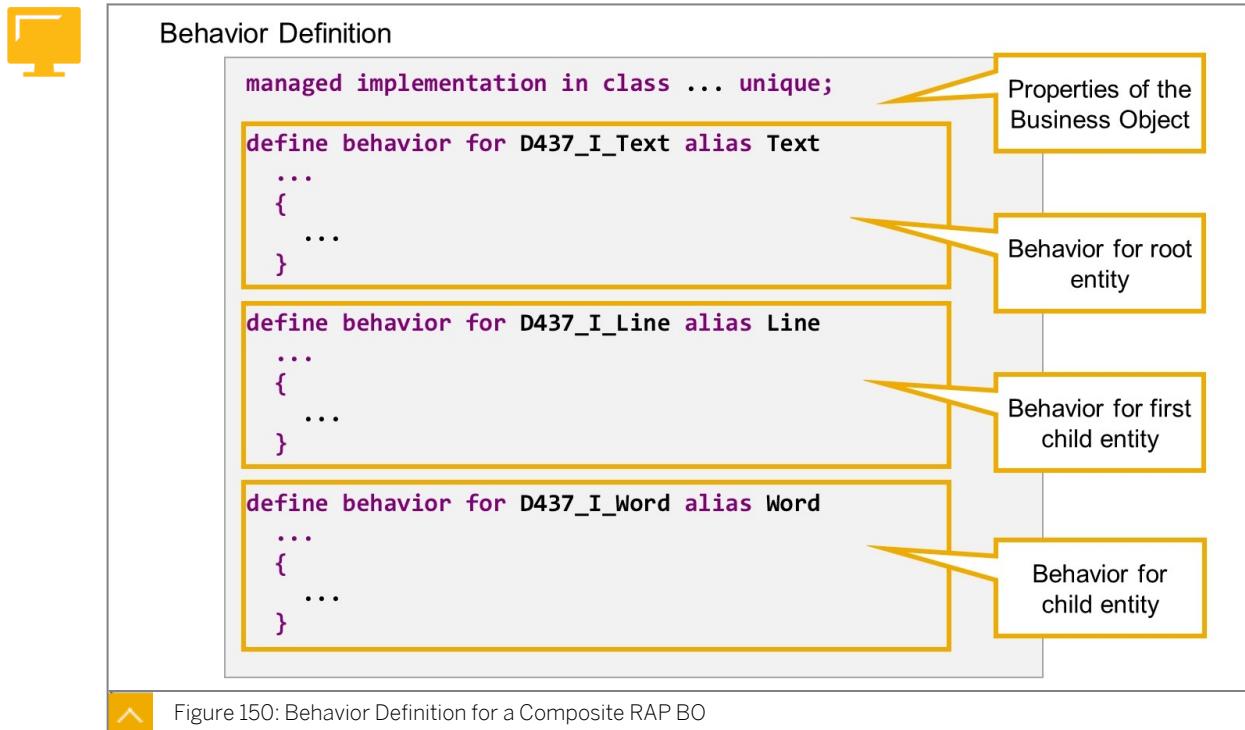
Figure 149: Example: Child Entity of Child Entity

The child entity of a child entity only requires the to-parent association to its direct parent. It is not mandatory to define a direct association to the root entity and there is no special association type for that purpose. The child entity D437_I_Word of D437_I_Line does not necessarily require an association to the root entity (D437_I_Text).

We will see later that, when adding the behavior for the RAP BO, such an association can be helpful, for example, to reference the root entity as lock master or authorization master.

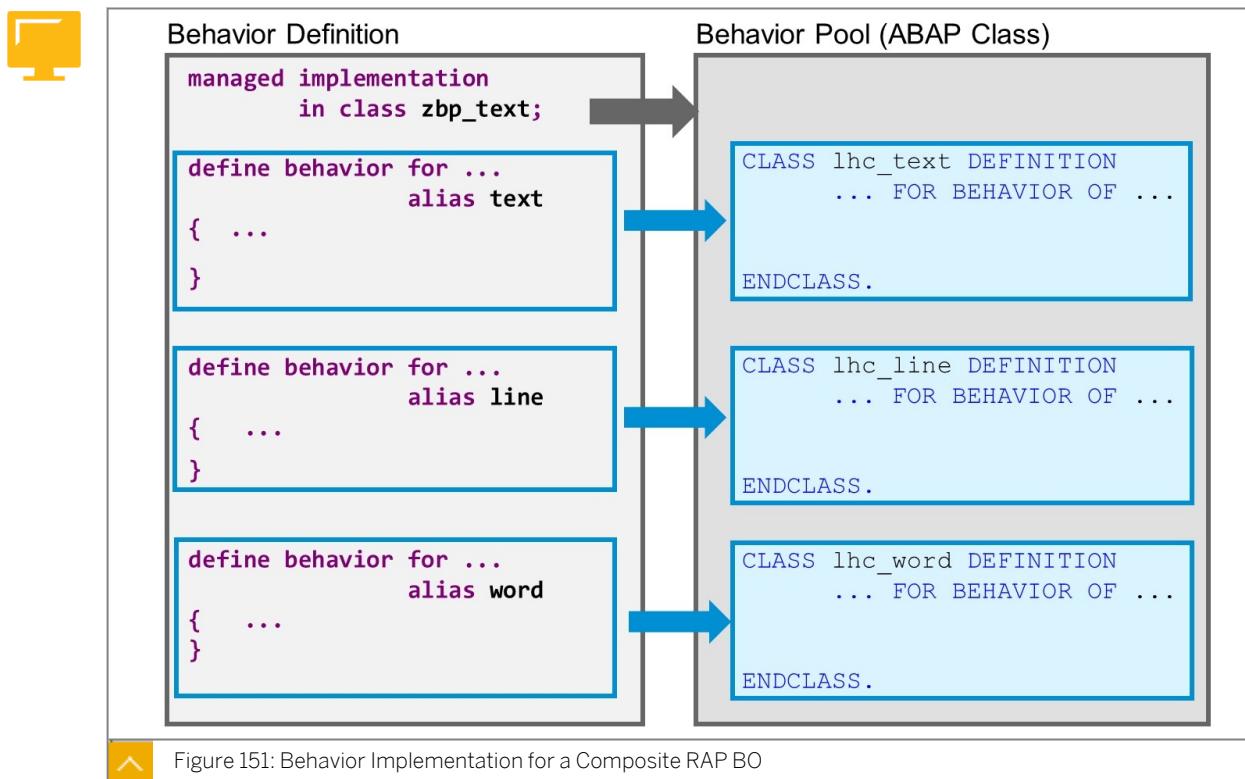
If an association to the root entity is needed, it is defined as an ordinary association with cardinality (1..1).

Behavior Definition for Composite RAP BO



While each entity of a composite RAP BO has its own data definition, there is only one behavior definition source per business object.

After some general properties of the business object, for example the implementation type of draft/non-draft enabled implementation, the behavior definition source contains exactly one define behavior statement for each entity of the hierarchy.



If a behavior definition source contains more than one DEFINE BEHAVIOR FOR statements, the corresponding behavior pool, that is the global ABAP class specified after IMPLEMENTATION IN CLASS, contains one local handler class for each of the entities.

We recommend that the name of the local handler class is lhc_<entity_name> where <entity_name> is the name of the CDS view entity or, if provided, the alias name for the entity from the behavior definition.



Hint:

When you use the available quick fix to generate the local handler classes, the name will automatically follow this guideline.



Behavior Definition

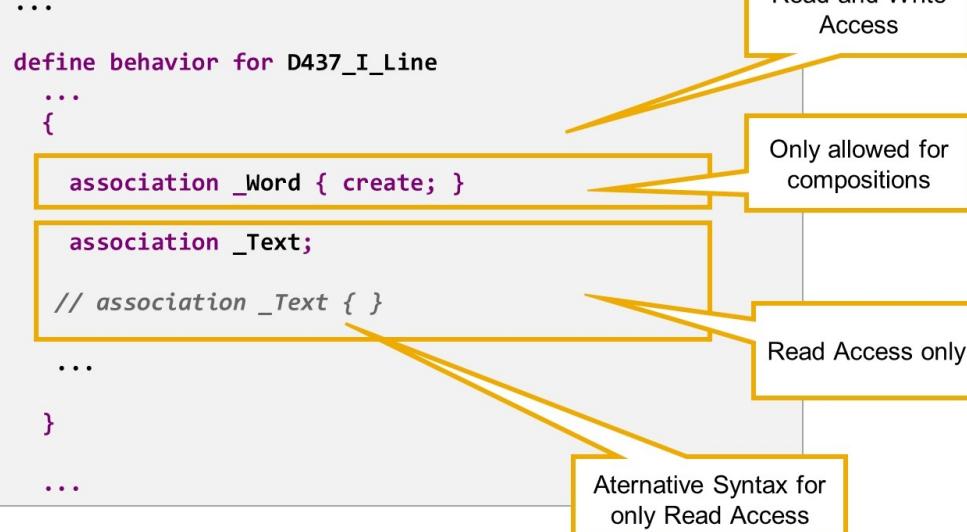


Figure 152: Associations in the Behavior Definition

By adding your associations to the behavior definition, you explicitly enable read access and create access for your associations. This means that you allow a RAP BO consumer to read data from related entity instances or to create new instances of the association target entity.

Read and create access is defined with the statement `association _Assoc { create; }`. Create access is only allowed for compositions. It is not allowed for to-parent associations. This means that child nodes can be created via their parent node, but parents can't be created via their child nodes.

Read access only is defined with `association _Assoc;` or with the alternative syntax variant, `association _Assoc { }.` This is allowed for any association defined in the CDS view entity.



Note:

To-parent associations are automatically read enabled by default and compositions are read and create enabled by default. We still recommend to specifying the read- and create-by-association operations explicitly. In future releases, this will be enforced when using strict-mode.



Internal Usage only

```
internal association _Word { create; }
```

Internal Usage of Create

```
association _Word { internal create; }
```

Instance Feature Control for Create

```
association _Word { create ( features : instance ); }
```

Figure 153: Some Variants of Statement Association

Several operation additions are available to restrict the usage of an association. If internal is placed before keyword association, read and create access are forbidden for an outside consumer of the business object. If it is placed within the curly brackets, before the keyword create, the create access is restricted, but read access is available for outside consumers.

As for the standard operations, update and delete, you can implement instance feature control for the create operation. To do so, add (features : instance) within the curly brackets, after keyword create.



Behavior Definition

```
managed implementation in class ... unique;  
with draft;
```

```
define behavior for D437_I_Line
```

```
...  
{
```

```
    association _Word { create; with draft; }
```

```
    association _Text { with draft; }
```

```
...  
}
```

Draft enabled
read and write
access

Draft enabled
read access

Figure 154: Draft Enabled Associations

By adding with draft; inside the curly brackets, you specify that the association is draft-enabled. A draft-enabled association retrieves active data if it is followed from an active instance and draft data if it is followed from a draft source instance (for details about the draft concept, see CDS BDL - managed, with draft).

If a business object is draft-enabled, then all associations should be draft-enabled, so that the associations always lead to the target instance with the same state (draft or active).

**Note:**

As soon as you draft-enable a BO by adding `with draft`, all BO-internal associations are automatically draft-enabled. To make this behavior explicit, the editor prompts you to specify the compositions within a draft BO with `with draft;`.

**Behavior Definition**

```
managed implementation in class ... unique;

define behavior for D437_I_Text alias Text
  persistent table d437_text
    lock master
    etag master ChangedAt
    authorization master
  {
    ...
  }

define behavior for D437_I_Line alias Line
  persistent table d437_line
    lock dependent by _Text
    etag dependent by _Text
    authorization dependent by _Text
  {
    ...
    association _Text { }
  }
  ...
}
```

Root entity
always master

Child entity
dependent

Association to
master entity



Figure 155: Locks, ETags, Authorizations and for Child Entities

The root entity of a business object is always defined as lock master and, if etag or authorization are specified, this is always with addition master.

For child entities, syntax options lock dependent by etag dependent by and authorization dependent by are available, each followed by the name of an association, that points to the related master entity.

The following rules apply:

lock:

- Currently, only root entities are allowed as lock master,
- Lock dependent is mandatory for child entities in managed scenarios,
- The association always points to root entity.

etag:

- Child entities can be dependent on master.
- Child entities with etag master have to define an own etag field.
- Association can point to non-root entity that is higher in the hierarchy.

authorization:

- Currently, only root entities are allowed as authorization master.
- Association always points to root entity.



Note:

If an entity is authorization dependent, the authorization check for update, delete, and create-by-association operations is done as authorization check for update of the master entity. The authorization check for actions, and create-enabled associations that are not compositions, is done in separate methods in the handler class for the dependent entity.

Unit 5

Exercise 15

Define a Composite RAP Business Object

Business Scenario

In this exercise, you extend your RAP Business Object for flight travels. Each flight travel becomes a composition of header data (*root entity*) and a number of flight travel items (*child entities*). First you copy and adjust template repository objects for the flight travel items. Then you define the composition of the RAP BO. All template repository objects are located in ABAP package *DEVS4D437TEMPLATES*.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 15: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model, Root)	D437E_I_TRAVEL
CDS Data Definition (Model, Child)	D437E_I_TRAVELITEM
CDS Behavior Definition (Model)	D437E_I_TRAVEL

Task 1: Copy Template for Data Model (Child Entity)

Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace *D437T* with *Z##*. Make sure that the copies refer to each other and not to the template objects.

Table 16: Template

Repository Object Type	Repository Object ID
Transparent Table	D437T_TRITEM
CDS Data Definition (Model, Child)	D437T_I_TRAVELITEM

1. Copy transparent table *D437T_TRITEM* to transparent table *Z##_TRITEM*. Adjust the foreign key definition for the table field *trguid* and activate the new repository object.
2. Create a copy of data definition *D437T_I_TRAVELITEM* (suggested name: *Z##_I_TRAVELITEM*). Replace the database table in the *FROM* clause with your own database table.
3. Remove the keyword *root* and activate the new repository object.

Task 2: Define Composition in Data Model

In the data definitions for your data model views (`Z##_L_Travel` and `Z##_L_TravellItem`), add the required associations to establish a parent-child relation between flight travels and flight travel items.

1. Edit the data definition for data model view `Z##_Travel` and add the addition `composition of`, followed by the name of your data model view for the child entity (`Z##_L_TravellItem`).



Hint:

Template *Define Root View Entity*, with which you created the data definition, already contained this addition. If you commented it out, rather than deleting it, you can re-activate it now.

2. Add a suitable cardinality and a meaningful association name (suggested name: `_TravellItem`).
3. Add the *composition* association to the element list of the view.
4. Perform a syntax check for the data definition.

Can you activate the data definition for the root view entity?

5. Edit the data definition of your child view entity `Z##_TravellItem`. Add the addition `association to parent`, followed by the name of your root view entity (`Z##_L_Travel`).
6. Add a meaningful association name (suggested name: `_Travel`) and the *ON* condition for the association.

Why are you not able to add a cardinality?

7. Add the `to parent` association to the element list of the view.
8. Perform a syntax check for the data definition.

Can you activate the data definition for the child view entity?

9. Activate the two data definitions in the correct sequence.

Task 3: Define Composition in the Behavior Definition

Add the child entity to the behavior definition for your RAP Business Object (`Z##_L_TRAVEL`). Complete the behavior definition to remove all syntax errors and warnings. In particular, link

the child entity to your transparent table (`Z##_TRITEM`) and a (generated) draft table (suggested name: `Z##_TRITEM_D`). Define the authorization handling and lock handling of the child entity dependent of the root entity, but define a specific etag field for the child entity. Define the field mapping, static field control and managed internal numbering. Finally, explicitly add the `composition` association and the `to parent` association to the behavior definitions.



Note:

Explicitly adding the associations is not yet mandatory in our training system because the RAP framework implicitly declares these associations. However, in future releases, it will become mandatory to explicitly declare the composition associations when implementing the behavior definition in `strict mode`.

1. Open the behavior definition for your data model view (`Z##_I_TRAVEL`) and perform a syntax check.

Is there a syntax error related to the child entity of the data model?

2. Edit your behavior definition (`Z##_I_TRAVEL`). At the end of the source code, add a new `define behavior` statement, followed by the name of the data model view for the child entity (`Z##_I_TravelItem`), an alias (suggested name: `Item`), and a pair of curly brackets (`{...}`).
3. Use the `addition persistent table` to link the behavior definition to your transparent table for travel items (`Z##_TRITEM`).
4. Use the `addition draft table` to link the behavior definition to a (not yet existing) draft table for travel items (suggested name for the new draft table: `Z##_TRITEM_D`).
5. Use the available quick fix to generate the draft table.
6. Return to the behavior definition. Use the `additions authorization dependent by` and `lock dependent by` to make lock and authorization handling for the child entity dependent from the root entity.
7. Use the `addition etag master` to enable optimistic concurrency control based on the time stamp of the last change (field `LocalChangedAt`).
8. To enable persistence for all elements of your child entity, define the mapping between table field names and CDS view element names.
9. Enable managed internal numbering for the technical key field (`Itguid`) and set all fields that are part of the semantic key or that are used in the definition of the `to_parent` association (`AgencyID`, `TravelID`, `ItemID`, `Trguid`) to `read-only`.
10. Add the `to parent` association (`_Travel`) to the behavior definition of the child entity and the `composition` association (`_TravelItem`) to the behavior definition of the root entity.



Note:

Remember that your Business Object is draft-enabled. Do not forget to draft-enable the associations, too.

11. Activate the behavior definition.

Define a Composite RAP Business Object

Business Scenario

In this exercise, you extend your RAP Business Object for flight travels. Each flight travel becomes a composition of header data (*root entity*) and a number of flight travel items (*child entities*). First you copy and adjust template repository objects for the flight travel items. Then you define the composition of the RAP BO. All template repository objects are located in ABAP package *DEVS4D437TEMPLATES*.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 15: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model, Root)	D437E_I_TRAVEL
CDS Data Definition (Model, Child)	D437E_I_TRAVELITEM
CDS Behavior Definition (Model)	D437E_I_TRAVEL

Task 1: Copy Template for Data Model (Child Entity)

Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace *D437T* with *Z##*. Make sure that the copies refer to each other and not to the template objects.

Table 16: Template

Repository Object Type	Repository Object ID
Transparent Table	D437T_TRITEM
CDS Data Definition (Model, Child)	D437T_I_TRAVELITEM

1. Copy transparent table *D437T_TRITEM* to transparent table *Z##_TRITEM*. Adjust the foreign key definition for the table field *trguid* and activate the new repository object.
 - a) Perform this step as before.
2. Create a copy of data definition *D437T_I_TRAVELITEM* (suggested name: *Z##_I_TRAVELITEM*). Replace the database table in the *FROM* clause with your own database table.
 - a) Perform this step as before.

3. Remove the keyword `root` and activate the new repository object.
 - a) Perform this step as before.

Task 2: Define Composition in Data Model

In the data definitions for your data model views (`Z##_I_Travel` and `Z##_I_TravelItem`, add the required associations to establish a parent-child relation between flight travels and flight travel items.

1. Edit the data definition for data model view `Z##_Travel` and add the addition `composition of`, followed by the name of your data model view for the child entity (`Z##_TravelItem`).



Hint:

Template `Define Root View Entity`, with which you created the data definition, already contained this addition. If you commented it out, rather than deleting it, you can re-activate it now.

- a) See the source code extract from the model solution.
2. Add a suitable cardinality and a meaningful association name (suggested name: `_TravelItem`).
 - a) See the source code extract from the model solution.
3. Add the `composition` association to the element list of the view.
 - a) See the source code extract from the model solution.
4. Perform a syntax check for the data definition.
 - a) Press `Ctrl + F2`.

Can you activate the data definition for the root view entity?

No, because the `to parent` association is missing from the data definition of the child view entity.

5. Edit the data definition of your child view entity `Z##_TravelItem`. Add the addition `association to parent`, followed by the name of your root view entity (`Z##_I_Travel`).
 - a) See source code extract from the model solution.
6. Add a meaningful association name (suggested name: `_Travel`) and the `ON` condition for the association.

Why are you not able to add a cardinality?

The cardinality of `to parent` association is always `[1..1]` and must not be specified manually.

- a) See the source code extract from the model solution.
7. Add the `to parent` association to the element list of the view.
 - a) See the source code extract from the model solution.

8. Perform a syntax check for the data definition.

- a) Press Ctrl + F2.

Can you activate the data definition for the child view entity?

Yes, even though the *composition* association is missing from the (active) data definition of the root view entity, this is just a warning (as opposed to error in the data definition in the root view entity).

9. Activate the two data definitions in the correct sequence.

- a) Place the cursor somewhere in the source code of *Z##_I_TravelItem* and press Ctrl + F3.
- b) Then place the cursor somewhere in the source code of *Z##_I_Travel* and press Ctrl + F3.



Hint:

Alternatively, you can activate both repository objects at the same time by pressing Ctrl + F3.

- c) Compare your code to the following extract form the model solution.

Data Definition **D437E_I_TRAVEL**:

```
...
define root view entity D437e_I_Travel
  as select from d437e_travel
  composition[0..*] of D437e_I_TravelItem as _TravelItem
{
  ...
  ...
  _TravelItem
}
```

Data Definition **D437E_I_TRAVELITEM**:

```
...
define view entity D437e_I_TravelItem
  as select from d437e_tritem
  association to parent D437e_I_Travel as _Travel
    on $projection.Trguid = _Travel.Trguid
{
  ...
  ...
  _Travel
}
```

Task 3: Define Composition in the Behavior Definition

Add the child entity to the behavior definition for your RAP Business Object (*Z##_I_TRAVEL*). Complete the behavior definition to remove all syntax errors and warnings. In particular, link

the child entity to your transparent table (`Z##_TRITEM`) and a (generated) draft table (suggested name: `Z##_TRITEM_D`). Define the authorization handling and lock handling of the child entity dependent of the root entity, but define a specific etag field for the child entity. Define the field mapping, static field control and managed internal numbering. Finally, explicitly add the composition association and the to parent association to the behavior definitions.



Note:

Explicitly adding the associations is not yet mandatory in our training system because the RAP framework implicitly declares these associations. However, in future releases, it will become mandatory to explicitly declare the composition associations when implementing the behavior definition in strict mode.

1. Open the behavior definition for your data model view (`Z##_I_TRAVEL`) and perform a syntax check.

Is there a syntax error related to the child entity of the data model?

Yes. The syntax error reads: “Target ‘`Z##_I_TRAVELITEM`’ of ‘`Z##_I_TRAVEL` _`TRAVELITEM`’ must be defined in the same BDEF as ‘`Z##_I_TRAVEL`’.”

- a) Perform this step as before.
2. Edit your behavior definition (`Z##_I_TRAVEL`). At the end of the source code, add a new `define behavior` statement, followed by the name of the data model view for the child entity (`Z##_I_TravelItem`), an alias (suggested name: `Item`), and a pair of curly brackets (`{...}`).
 - a) See the source code extract from the model solution.
3. Use the addition `persistent table` to link the behavior definition to your transparent table for travel items (`Z##_TRITEM`).
 - a) See the source code extract from the model solution.
4. Use the addition `draft table` to link the behavior definition to a (not yet existing) draft table for travel items (suggested name for the new draft table: `Z##_TRITEM_D`).
 - a) See the source code extract from the model solution.
5. Use the available quick fix to generate the draft table.
 - a) Open the context menu on the name of the draft table and choose *Quick Fix*.
 - b) Choose the offered quick fix, enter a description for the draft table, and choose *Next >*
 - c) Press `Ctrl + F3` to activate the new transparent table.
6. Return to the behavior definition. Use the additions `authorization dependent by` and `lock dependent by` to make lock and authorization handling for the child entity dependent from the root entity.
 - a) Specify the dependence with the name of the to parent association in the data model (`_Travel`).
 - b) See the source code extract from the model solution.

7. Use the addition `etag master` to enable optimistic concurrency control based on the time stamp of the last change (field `LocalChangedAt`).
 - a) See the source code extract from the model solution.
8. To enable persistence for all elements of your child entity, define the mapping between table field names and CDS view element names.
 - a) Within the curly brackets of the behavior definition, add the keyword `mapping for` followed by the name of your database table, the addition `corresponding`, and a pair of curly brackets.
 - b) Within the curly brackets, list all view elements and table fields that differ in more than just uppercase or lowercase.

**Hint:**

To reduce typing effort, you may copy the `mapping` statement from the model solution, that is, behavior definition `D437E_I_TRAVEL`.

- c) See the source code extract from the model solution.
9. Enable managed internal numbering for the technical key field (`Itguid`) and set all fields that are part of the semantic key or that are used in the definition of the `to_parent` association (`AgencyID`, `TravelID`, `ItemID`, `Trguid`) to `read-only`.
 - a) See the source code extract from the model solution.
10. Add the `to_parent` association (`_Travel`) to the behavior definition of the child entity and the `composition` association (`_TravelItem`) to the behavior definition of the root entity.

**Note:**

Remember that your Business Object is draft-enabled. Do not forget to draft-enable the associations, too.

- a) See the source code extract from the model solution.
11. Activate the behavior definition.
 - a) Choose `Activate` or press `Ctrl + F3`.
 - b) Compare your source code to the following extract from the model solution.

Behavior Definition **D437E_I_TRAVEL**:

```
/***** Business Object Travel *****/
managed implementation in class bp_d437e_i_travel unique;
with draft;

/***** Root Entity Travel *****/
define behavior for D437e_I_Travel alias Travel
  persistent table d437e_travel
  draft table d437e_travel_d
  lock master
  total etag ChangedAt
```

```
etag master LocalChangedAt
authorization master ( instance )
{

...
association _TravelItem { with draft; }

...
}

/***** Child Entity Travel Item *****/
define behavior for D437e_I_TravelItem alias Item
persistent table d437e_tritem
draft table d437e_tritem_d
lock dependent by _Travel
authorization dependent by _Travel
etag master LocalChangedAt
{
    field ( readonly, numbering : managed ) Itguid;
    field ( readonly ) AgencyID, TravelId, ItemID, Trguid;

    association _Travel { with draft; }

mapping for d437e_tritem
{
    Itguid          = itguid;
    AgencyID       = agencynum;
    TravelId        = travelid;
    ItemID         = tritemno;
    Trguid          = trguid;
    CarrierId      = carrid;
    ConnectionId   = connid;
    FlightDate     = fldate;
    BookingId      = bookid;
    FlightClass    = class;
    PassengerName  = passname;
    CreatedAt      = created_at;
    CreatedBy      = created_by;
    ChangedAt      = changed_at;
    ChangedBy      = changed_by;
    LocalChangedAt = loc_changed_at;
}
}
```



LESSON SUMMARY

You should now be able to:

- Define compositions in RAP BOs

Defining Compositions in OData UI Services

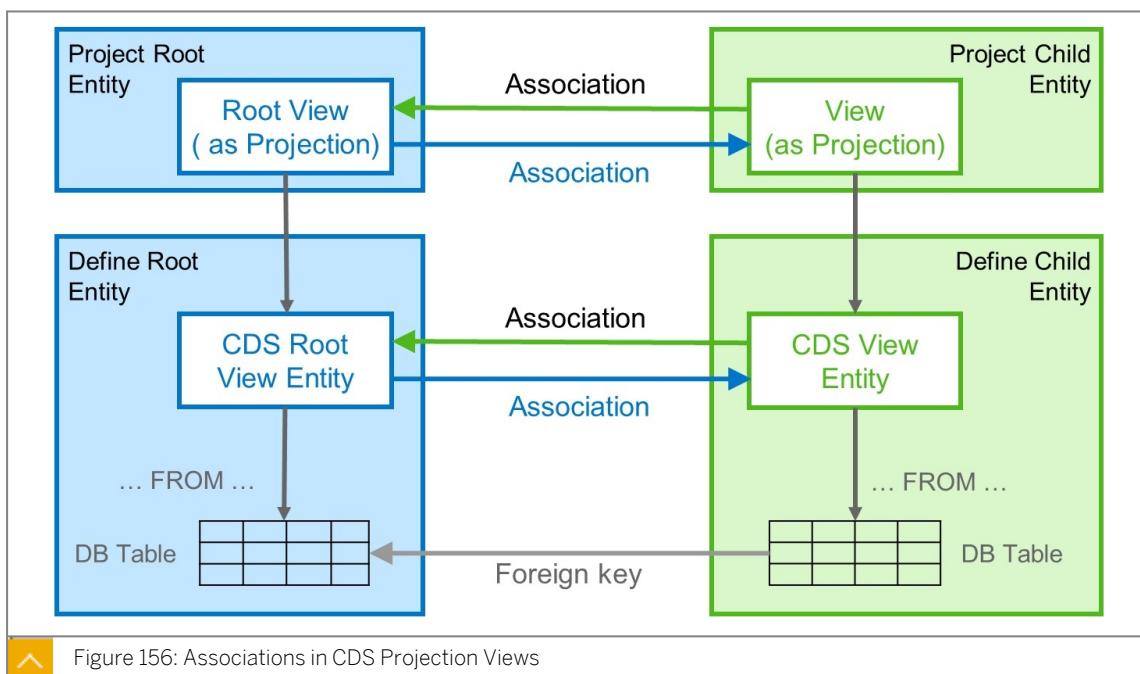


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Expose compositions to OData services
- Enable navigation in SAP Fiori elements apps

Composition in Data Model Projection



Just like the data definition of composite RAP Business object consists of several CDS view entities, its projection consists of several CDS projection views, one projection view for each of the data definition views. To build an OData UI service, each projection view is enriched with the UI metadata, preferably in a CDS metadata extension.

To make the structure of the business object available in the OData Service too, you have to establish the hierarchy on the projection layer. To do so, you need compositions and to-parent associations that link the projection views.



Note:

It would not be sufficient to simply expose the associations defined in the underlying data definition views. The targets of those associations are a data definition views and not projection views. By following such an association, the consumer would not have access to the required metadata.



Data Definition (Projection)

```
define view entity Z00_C_Source
  as projection on Z00_I_Source
{
  ...
  _Asso: redirected to Z00_C_Target
}
```

Data Definition (Projection)

```
define view entity Z00_C_Target
  as projection on Z00_I_Target
{
  ...
}
```

Association redirected to projection view

Data Definition (Data Model)

```
define view entity Z00_I_Source
  association to Z00_I_Target
    as _Asso
    on ...
{
  ...
  _Asso
}
```

Data Definition (Data Model)

```
define view entity Z00_I_Target
{
  ...
}
```



Figure 157: Redirecting Associations

Instead of defining completely new associations on projection level, we recommend reusing the associations from the underlying data model and redirecting them to a new target.

In the example, the data model view on the left (Z00_I_Source), defines and exposes an association `_Asso` that uses the data model view on the right as its target (Z00_I_Target).

The view on the upper left (Z00_C_Source) is a projection on of Z00_I_Source. It has access to the exposed association `_Asso` and can expose it further. But, by doing so, the association `_Asso` would still point to the data definition view Z00_I_Target.

The association is redirected by adding a colon, the keyword `redirected to`, and the name of the new target.

This syntax can be used for any kind of association, general associations, compositions, and to-parent associations. However, when using `redirected to <target>`, the special characteristics of the compositions and to-parent associations will be lost.



Data Definition (Projection View)

```

define view entity D437_C_Line
  as projection on D437_I_Line
{
  ...
  // _Word,
  _Word : redirect to composition child D437_C_Word,
  ...
  // _Text,
  _Text : redirect to parent D437_C_Text,
  ...
}

```

Simply exposing _Word:
association would point to D437_I_Word

Redirected to projection view D437_C_word

Simply exposing _Text:
association would point to D437_I_Text

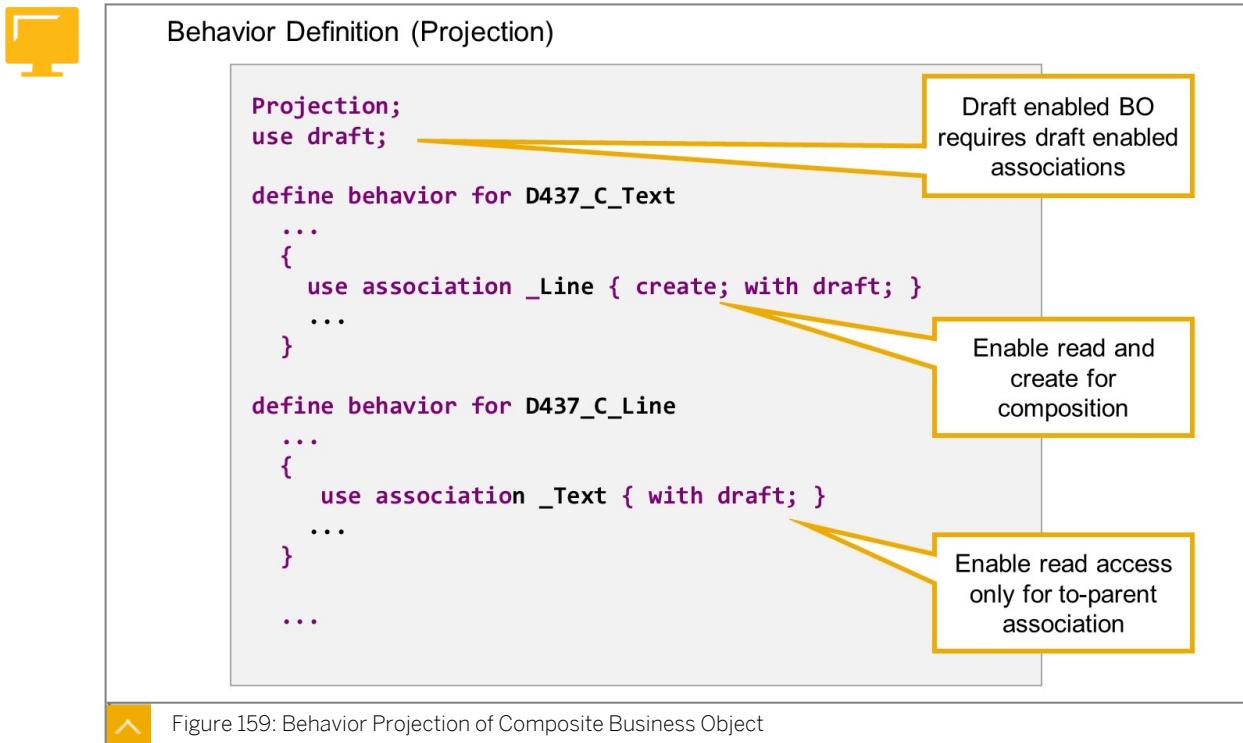
Redirected to projection view D437_C_Text

Figure 158: Redirecting Compositions and To-Parent Associations

For redirecting compositions and to-parent associations, ABAP CDS offers the dedicated syntax elements `redirected to composition child` and `redirected to parent`. By using these variants, the special characteristics of compositions and to-parent associations are kept.

When using `redirected to composition child`, the original association has to be a composition and the new target has to be a projection of the original target. When using `redirected to parent`, the original association has to be a to-parent association. The new target should be a projection of the original target.

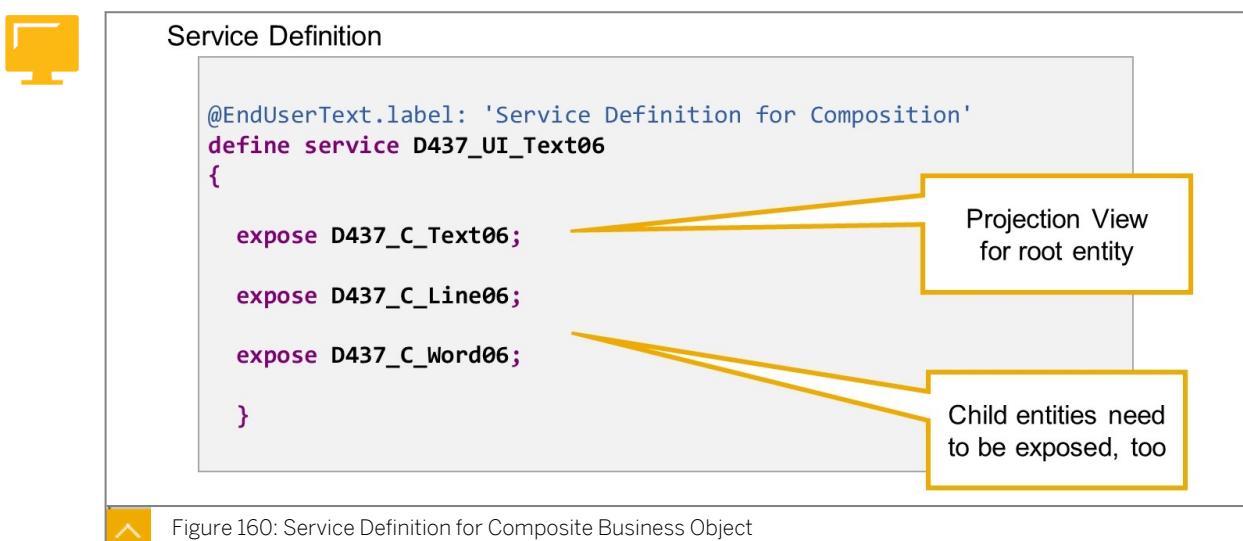
Composition in Behavior Projection



To make the transactional enabling of the associations available in the OData service, we have to include it in the behavior projection. Similar to `use create`, `use delete`, or `use action`, a statement `use association` exists for this purpose.

If RAP draft handling is enabled in the behavior projection (`use draft`), the associations must be draft enabled using the syntax addition `with draft;`.

Facets and Additional Object Page in SAP Fiori



If a RAP BO projection consists of several entities, each entity has to be exposed in the service definition explicitly to make the hierarchy available in the service.

Object Page of Root Entity

Text

Text Line

Text Owner: TRAIN-00

Number of Lines in Text: 1

Text Line

Create Delete

Text Line ID Number of Words in Line

10 2 >

Second Facet with child entity data

Figure 161: Visualization of Child Entity in Second Facet

In an SAP Fiori elements app, the composition can be displayed by adding additional facets to the object page. The facet can then contain a list of the related child entity instances. In the example, the object page for root entity *Text* contains a second facet, which displays a list of *Text Lines*.

Metadata Extension of Root Entity (Projection)

```
annotate view D437_C_Text with
{
  @UI.facet: [ { id: 'Text',
    purpose: #STANDARD,
    type: #IDENTIFICATION_REFERENCE,
    label: 'Text',
    position: 10 },
    { id: 'TextLine',
    purpose: #STANDARD,
    type: #LINEITEM_REFERENCE,
    label: 'Text Line',
    position: 20,
    targetElement: '_Item' } ]
}
```

TargetElement is set to the association name

Different type for second facet

Figure 162: UI Metadata for Second Facet

The additional facet is defined in the metadata extension of the parent entity. The first facet, which was already there, displays the data of the parent entity itself. It is of type `#IDENTIFICATION_REFERENCE`.

The facet for the child entity data has to be of type #LINEITEM_REFERENCE. Facets of this type require a value for subannotation targetElement. Here, you specify the name of the association that links the child entity to the parent entity. Most of the time, this association will be a composition, or, more precisely, an association that is redirected to a composition child.

Unit 5 Exercise 16

Define a Composite OData UI Service with RAP

Business Scenario

In this exercise you add the child entity of your RAP Business Object to the definition and meta data of the ODATA service to display associated flight travel items in your SAP Fiori elements app.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 17: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	D437E_C_TRAVELITEM
CDS Behavior Definition (Projection)	D437E_C_TRAVEL
Service Definition	D437E_UI_TRAVEL
CDS Metadata Extension	D437E_C_TRAVELITEM

Task 1: Copy Template for Data Model Projection (Child Entity)

Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace D437T with Z##. Make sure the copies refer to your own repository objects and not to the template objects.

Table 18: Template

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	D437E_C_TRAVELITEM
CDS Metadata Extension	D437E_C_TRAVELITEM

1. Create a copy of data definition D437T_C_TRAVELITEM (suggested name: Z##_C_TRAVELITEM). Replace the CDS View after `projection on` with the name of your own data model view for travel items (Z##_I_TravelItem).
2. Remove the keyword `root` and activate the new repository object.
3. Create a copy of metadata extension D437T_C_TRAVELITEM (suggested name: Z##_C_TRAVELITEM). Replace the CDS view after `annotate view` with your own projection view for travel items (Z##_C_TravelItem) and activate the new repository object.

Task 2: Add Composition to the Data Model Projection

In the data definitions for your projection views (`Z##_C_Travel` and `Z##_C_TravelItem`), expose the associations from the underlying data model views. Make sure that the associations do not point to the data model views but that they are redirected to point at the respective projection views. Make sure the associations keep their special character as composition association and `to parent` association.

1. Edit the data definition of your projection view for travel items (`Z##_C_TravelItem`). At the end of the projection list, add the name of the `to parent` association, which you defined in the underlying data model view.
2. Redirect the association to the projection view for flight travels.
3. Classify the redirected association as a `to parent` association.
4. Activate the data definition.
5. Edit the data definition of your projection view for travels (`Z##_C_Travel`). At the end of the projection list, add the name of the `composition` association, which you defined in the underlying data model view.
6. Redirect the association to the projection view for flight travel items and classify the redirected association as a `composition` association.
7. Activate the data definition.

Task 3: Add Composition to the Behavior Projection

Edit the behavior projection for your RAP Business Object (`Z##_C_TRAVEL`) and add a behavior projection for the child entity. Expose its etag definition and add the associations that establish the composition.

1. Edit your behavior projection (`Z##_C_TRAVEL`). At the end of the source code, add a new `define behavior` statement, followed by the name of the projection view for the child entity (`Z##_C_TravelItem`), an alias (suggested name: `Item`) and a pair of curly brackets (`{...}`).
2. Add the etag definition to the behavior projection.
3. Add the `to parent` association (`_Travel`) to the behavior projection of the child entity and the `composition` association (`_TravelItem`) to the behavior projection of the root entity.



Note:

Remember that the projection of your Business Object is draft-enabled. Do not forget to draft-enable the associations, too.

4. Activate the behavior projection.

Task 4: Add Composition to the OData UI Service

Add the child entity of your RAP BO to your service definition (`Z##_UI_TRAVEL`) and extend the UI metadata of the service to display a list of child entities in a second facet on the object page for flight travels.

1. Edit your service definition `Z##_UI_TRAVEL` and add the projection view for flight travel items.

2. Activate the service definition.
3. Open your service binding and analyze the information under *Entity Set and Association*.

**Note:**

There is no *Refresh* button available in service bindings. If your service binding is already opened, you have to close and re-open it to see the additional information.

What new information do you see?

4. Open the metadata extension for flight travels (`Z##_C_TRAVEL`) and locate the annotation `@UI.facet`. Add a comma sign (,) after the closing curly bracket.
5. Copy the existing facet definition (the pair of curly brackets and its content), and insert the copy after the comma sign.
6. Adjust the second facet definition according to the following table:

Property	Value
<code>id</code>	<code>'TravelItem'</code>
<code>purpose</code>	<code>#STANDARD</code>
<code>type</code>	<code>#LINEITEM_REFERENCE</code>
<code>label</code>	<code>'Travel Items'</code>
<code>position</code>	<code>20</code>

7. At the end of this second facet definition, add the property `targetElement` with the name of the `composition` association as value.
8. Activate the metadata extension and retest the preview for the SAP Fiori Elements app. Make sure that the *Object page* for flight travels displays an (empty) list of flight travel items.

Can you create new flight travel items?

Unit 5

Solution 16

Define a Composite OData UI Service with RAP

Business Scenario

In this exercise you add the child entity of your RAP Business Object to the definition and meta data of the ODATA service to display associated flight travel items in your SAP Fiori elements app.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 17: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	D437E_C_TRAVELITEM
CDS Behavior Definition (Projection)	D437E_C_TRAVEL
Service Definition	D437E_UI_TRAVEL
CDS Metadata Extension	D437E_C_TRAVELITEM

Task 1: Copy Template for Data Model Projection (Child Entity)

Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace D437T with Z##. Make sure the copies refer to your own repository objects and not to the template objects.

Table 18: Template

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	D437E_C_TRAVELITEM
CDS Metadata Extension	D437E_C_TRAVELITEM

1. Create a copy of data definition D437T_C_TRAVELITEM (suggested name: Z##_C_TRAVELITEM). Replace the CDS View after *projection on* with the name of your own data model view for travel items (Z##_L_TravelItem).
 - a) Perform this step as before.
2. Remove the keyword `root` and activate the new repository object.
 - a) Perform this step as before.
3. Create a copy of metadata extension D437T_C_TRAVELITEM (suggested name: Z##_C_TRAVELITEM). Replace the CDS view after `annotate view` with your own

projection view for travel items (`Z##_C_TravelItem`) and activate the new repository object.

- a) Perform this step as before.

Task 2: Add Composition to the Data Model Projection

In the data definitions for your projection views (`Z##_C_Travel` and `Z##_C_TravelItem`), expose the associations from the underlying data model views. Make sure that the associations do not point to the data model views but that they are redirected to point at the respective projection views. Make sure the associations keep their special character as composition association and to parent association.

1. Edit the data definition of your projection view for travel items (`Z##_C_TravelItem`). At the end of the projection list, add the name of the to parent association, which you defined in the underlying data model view.
 - a) See the source code extract from the model solution.
2. Redirect the association to the projection view for flight travels.
 - a) After the association name, add a colon (:), followed by the keywords redirected to and the name of the redirection target (`Z##_Travel`).
 - b) See the source code extract from the model solution.
3. Classify the redirected association as a to parent association.
 - a) Between the keyword to and the name of the redirection target, add the keyword parent.
 - b) See the source code extract from the model solution.
4. Activate the data definition.
 - a) Choose Activate or press STRG + F3.
5. Edit the data definition of your projection view for travels (`Z##_C_Travel`). At the end of the projection list, add the name of the composition association, which you defined in the underlying data model view.
 - a) See the source code extract from the model solution.
6. Redirect the association to the projection view for flight travel items and classify the redirected association as a composition association.
 - a) After the association name, add a colon (:), followed by the keywords redirected to composition child and the name of the redirection target (`Z##_Travel`)
 - b) See the source code extract from the model solution.
7. Activate the data definition.
 - a) Choose Activate or press Ctrl + F3.
 - b) Compare your code to the following extract form the model solution.

Data Definition **D437E_C_TRAVEL**:

```
...
define root view entity D437e_C_Travel
as projection on D437e_I_Travel
```

```
{
...
    _TravelItem: redirected to composition child D437e_C_TravelItem
}
```

Data Definition **D437E_C_TRAVELITEM**:

```
...
define view entity D437e_C_TravelItem
    as projection on D437e_I_TravelItem
{
...
    _Travel : redirected to parent D437e_C_Travel
}
```

Task 3: Add Composition to the Behavior Projection

Edit the behavior projection for your RAP Business Object (*Z##_C_TRAVEL*) and add a behavior projection for the child entity. Expose its etag definition and add the associations that establish the composition.

1. Edit your behavior projection (*Z##_C_TRAVEL*). At the end of the source code, add a new `define behavior` statement, followed by the name of the projection view for the child entity (*Z##_C_TravelItem*), an alias (suggested name: `Item`) and a pair of curly brackets (`{...}`).
 - a) See the source code extract from the model solution.
2. Add the etag definition to the behavior projection.
 - a) Before the opening curly bracket, add keywords `use etag`.
 - b) See the source code extract from the model solution.
3. Add the `to parent` association (`_Travel`) to the behavior projection of the child entity and the `composition` association (`_TravelItem`) to the behavior projection of the root entity.



Note:

Remember that the projection of your Business Object is draft-enabled. Do not forget to draft-enable the associations, too.

- a) See the source code extract from the model solution.
4. Activate the behavior projection.
 - a) Choose *Activate* or press `Ctrl + F3`.
 - b) Compare your source code to the following extract from the model solution.

Behavior Definition **D437E_C_TRAVEL**:

```
***** Business Object Travel ***** /
```

```

projection;
use draft;

***** Root Entity Travel *****

define behavior for D437e_C_Travel alias Travel
use etag
{

    ...

    use association _TravelItem { with draft; }

    ...

}

***** Child Entity Travel Item *****

define behavior for D437e_C_TravelItem alias Item
use etag
{

    use association _Travel { with draft; }

}

```

Task 4: Add Composition to the OData UI Service

Add the child entity of your RAP BO to your service definition (`Z##_UI_TRAVEL`) and extend the UI metadata of the service to display a list of child entities in a second facet on the object page for flight travels.

1. Edit your service definition `Z##_UI_TRAVEL` and add the projection view for flight travel items.
 - a) Within the curly brackets, add another `expose` statement, followed by the name of your projection view for flight travel items (`Z##_C_TravelItem`).
 - b) See the source code extract from the model solution.
2. Activate the service definition.
 - a) Choose `Activate` or press `Ctrl + F3`.
3. Open your service binding and analyze the information under *Entity Set and Association*.



Note:

There is no *Refresh* button available in service bindings. If your service binding is already opened, you have to close and re-open it to see the additional information.

What new information do you see?

The child entity `Z##_C_TravelItem`, the associations between root entity and child entity and the CDS views that are used in the child entity for value helps.

4. Open the metadata extension for flight travels (`Z##_C_TRAVEL`) and locate the annotation `@UI.facet`. Add a comma sign (,) after the closing curly bracket.
 - a) See the source code extract from the model solution.

5. Copy the existing facet definition (the pair of curly brackets and its content), and insert the copy after the comma sign.
 - a) See the source code extract from the model solution.
6. Adjust the second facet definition according to the following table:

Property	Value
<i>id</i>	'TravelItem'
<i>purpose</i>	#STANDARD
<i>type</i>	#LINEITEM_REFERENCE
<i>label</i>	'Travel Items'
<i>position</i>	20

- a) See the source code extract from the model solution.
7. At the end of this second facet definition, add the property `targetElement` with the name of the `composition` association as value.
 - a) See the source code extract from the model solution.
8. Activate the metadata extension and retest the preview for the SAP Fiori Elements app. Make sure that the *Object page* for flight travels displays an (empty) list of flight travel items.

Can you create new flight travel items?

No, the basic operations for the child entities (create, update, delete) are not yet enabled and projected to the UI service.

- a) Perform this step as before.
- b) Compare your source code to the following extract from the model solution.

Service Definition D437E_UI_TRAVEL:

```
@EndUserText.label: 'Flight Travel Service Definition'
define service D437E_UI_TRAVEL {
  expose D437e_C_Travel;
  expose D437e_C_TravelItem;
}
```

Metadata Extension D437E_C_TRAVEL

```
...
annotate view Z00_C_TRAVEL with
{
  @UI.facet: [ { id:           'Travel',
                 purpose:       #STANDARD,
                 type:          #IDENTIFICATION_REFERENCE,
                 label:         'Travel',
```

```
        position:      10 },  
  
    { id:          'TravelItem',  
      purpose:     #STANDARD,  
      type:        #LINEITEM_REFERENCE,  
      label:       'Travel Items',  
      position:    20,  
      targetElement: '_TravelItem'  
    ]  
  
    ...  
}
```



LESSON SUMMARY

You should now be able to:

- Expose compositions to OData services
- Enable navigation in SAP Fiori elements apps

Implementing the Behavior for Composite RAP BOs



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Access composite business objects with EML

Read-by-Association Operations



```
DATA gt_import TYPE TABLE FOR
      READ IMPORT d437_i_text\_line.

DATA gt_result TYPE TABLE FOR
      READ RESULT d437_i_text\_line.

READ ENTITIES OF d437_i_text
  ENTITY text
    [ BY \ line ]
    ALL FIELDS WITH gt_import
  RESULT gt_result.

* Short syntax variant

READ ENTITY d437_i_text
  [ BY \_line ]
  ALL FIELDS WITH gt_import
  RESULT gt_result.
```

Use of association
in derived types

Use of association
in read access

gt_import contains
keys for instances of
entity *text*

Figure 163: Read by Association

When accessing a RAP business object via EML, you can use the associations defined in its behavior definition to read data from associated child entities that are part of the composition tree.



Note:

It is also possible to read parent entities via the child entities, however, only from within the implementation class.

The read-by-association operation is implemented by adding `BY \<association_name>` between the entity name and the field specification in `READ ENTITIES` or `READ ENTITY`. In the example, the association name is `_line`.

The derived types for the read-by-association operation are defined by using `<entity_name>\<association_name>` instead of the entity name

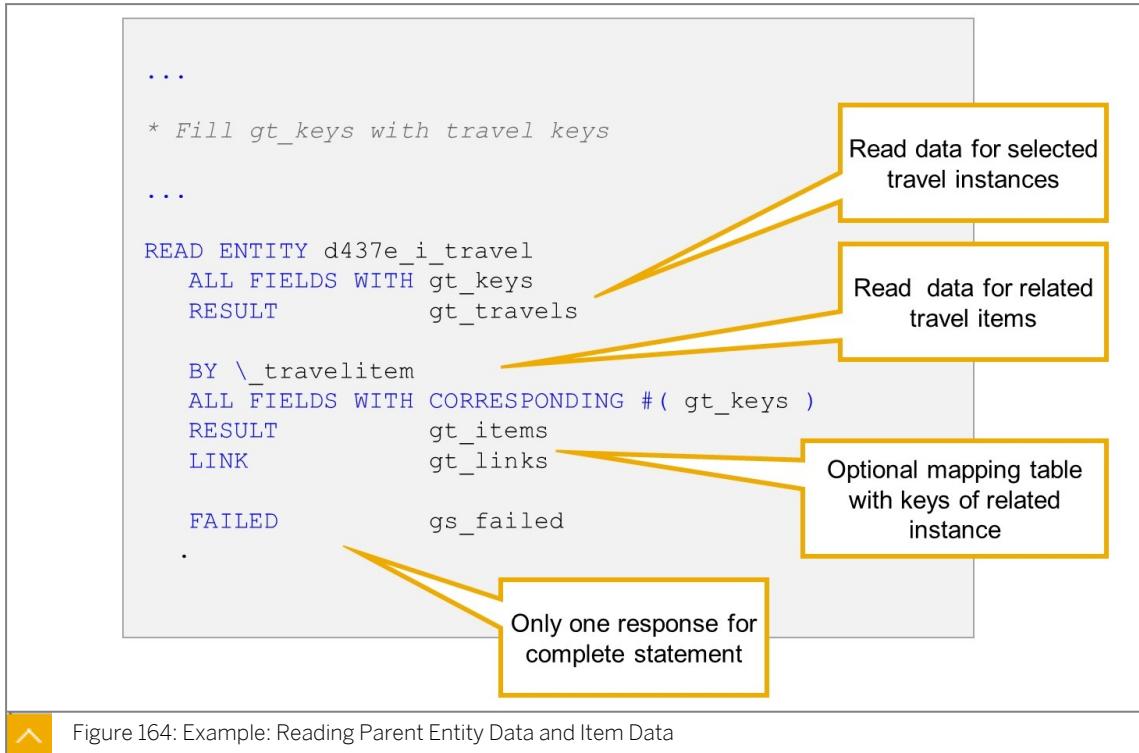


Figure 164: Example: Reading Parent Entity Data and Item Data

It is possible to read parent entity instances and the related child entity instances in the same EML statement, even when using the short form, READ ENTITY. In the example, `gt_keys` is filled with one or several keys for instances of root entity `d437_i_travel`. The first part of the READ ENTITY statement retrieves the data for the travel instances.

The second part reads data of all TravellItem instances that are related to the root entity instances via association `_travelitem`.

The addition `LINK` is only available in read-by-association operations and returns an internal table with keys of source entity instances and target entity instance. It can be used later to map the travels to the related items if more than one travel has been read.



Hint:

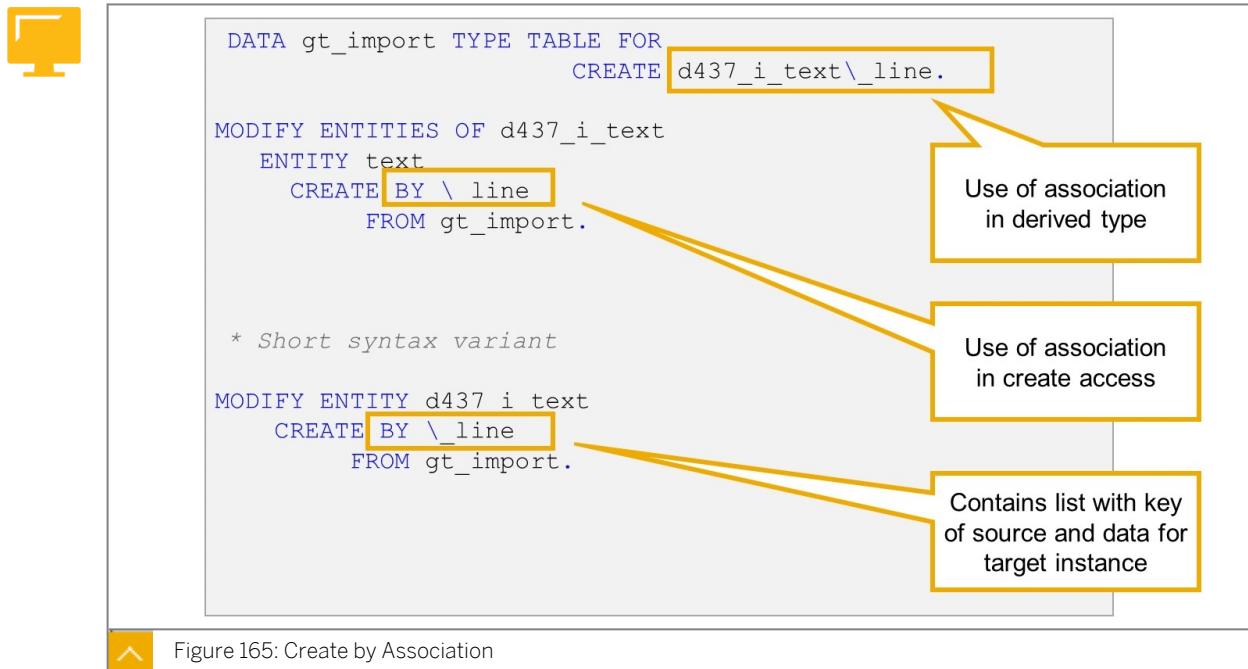
If all you need are the keys of the travel items, you can omit the `RESULT` addition and only specify the `LINK` addition. This can help to improve performance.



Note:

When combining several read and read-by-association operations in one statement, the response parameters, like, for example the `FAILED` parameter, only exist once.

Create-by-Association Operations



If an association is create-enabled in the behavior definition, you can use this association to create instances of the associated entity.



Note:

Up to now, only compositions can be create-enabled. This means that the target of the association is always a composition child of the entity for which you execute the operation.

The create-by-association operation is implemented by adding `BY \<association_name>` after the keyword `CREATE` in `MODIFY ENTITIES` or `MODIFY ENTITY`. In the example, the association name is `_line`.

The derived type for the create-by-association operation is defined by using `<entity_name>\<association_name>` instead of the entity name alone.

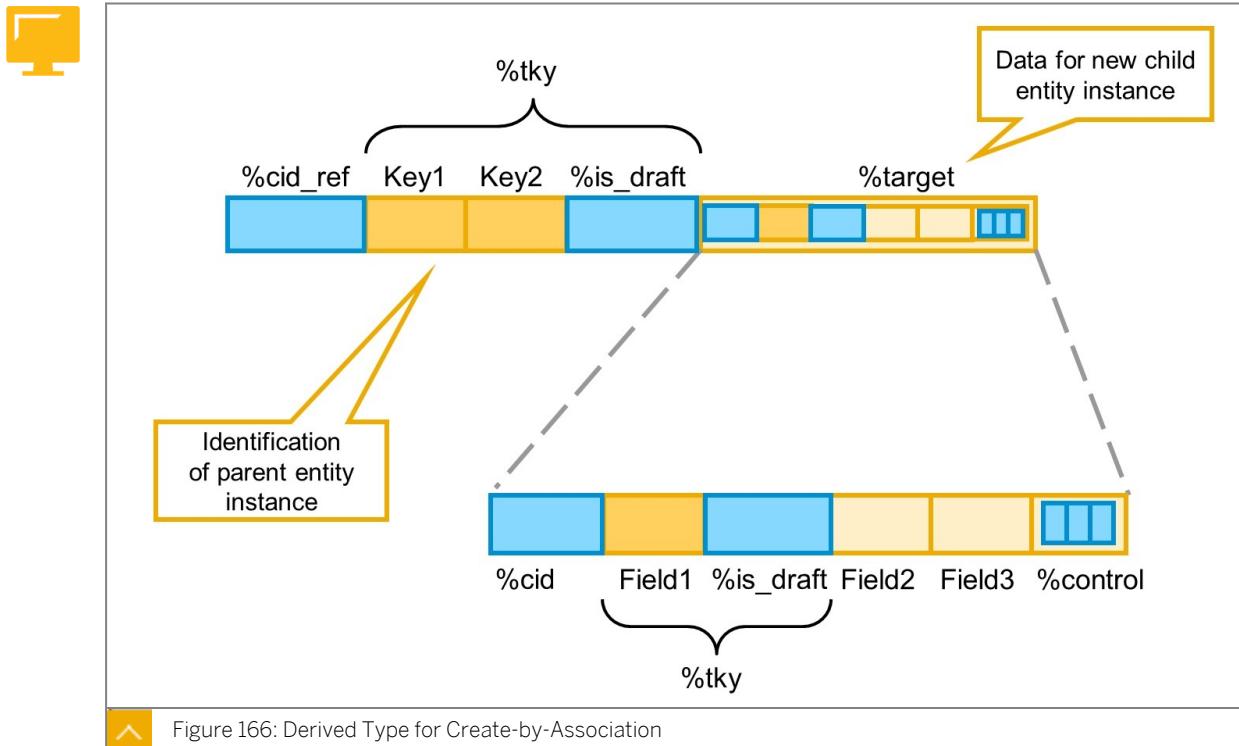


Figure 166: Derived Type for Create-by-Association

The line type of the internal table that serves as import for EML operation Create-By-Association consists of two parts:

- Elementary components to identify the instance of the parent entity. These fields are summarized in component group **%tky**.
- %target** is a structured component to specify the data for the new child entity instance, including a **%control** structure to specify which components are supplied

The component **%cid_ref** is needed to identify the parent entity instance in situations where the actual key is not yet available. This is the case if, for example, internal numbering is used and the parent entity instance has just been created, or is created in the same EML statement.

Similarly, the component **%cid** in the sub-structure **%target** is used to set a temporary key for the new child entity instance, by which it can be identified until internal numbering provides the actual key.

Unit 5 Exercise 17

Implement the Behavior of a Composite RAP Business Object

Business Scenario

In this exercise, you extend the behavior definition for the child entity with basic operations (create, update, delete) and you define and implement some determinations and validations.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 19: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Definition)	D437E_I_TRAVEL
CDS Behavior Definition (Projection)	D437E_C_TRAVEL
ABAP Class	BP_D437E_I_TRAVEL

Task 1: Basic Operations for the Child Entity

Enable the basic operations (create, update, delete) for the child entity (`Z##_I_TravelItem`), in both, the RAP Business Object and in the SAP Fiori elements app.

1. Edit the behavior definition for your RAP Business Object (`Z##_I_TRAVEL`). Scroll down to the behavior definition for the child entity (`Z##_I_TravelItem`) and add the basic operations (update, delete).

Why do you get a syntax error when you add the basic operation `create`, too?

2. scroll up to the behavior definition of the root entity (`Z##_I_Travel`) and enable the association to the child entity for `create`.
3. Edit the behavior projection for your RAP Business Object (`Z##_C_TRAVEL`) and make the new basic operations available in the OData service.
4. Similarly, add the `create` property of the `composition` association to the behavior projection.

5. Activate the behavior projection and refresh the preview for the SAP Fiori elements app. Make sure the second facet of the *Object page* offers a *Create* button and *Delete* button.

Task 2: Determination for Semantic Key of Child Entity

Define a determination for the child entity of your RAP Business Object (suggested name: *determineSemanticKey*). Implement the determination to set the values of fields *AgencyID* and *TravelID* to the same values as in the related instance of the root entity (*Z##_L_Travel*).

To set the value of field *ItemID*, find the largest item number used in the same flight travel so far and increase it by 10. Remember to evaluate both the active travel items and the draft travel items.

1. Open the behavior definition for your RAP Business Object and scroll down to the behavior definition for your child entity (*Z##_L_TravelItem*)
2. Define a new determination for this child entity (suggested name: *determineSemanticKey*) and make sure that it is only executed during creation of a new instance.
3. Edit the implementation method for the determination. use EML statement *READ ENTITY* or *READ ENTITIES* to read the data of all affected flight travel items and implement a loop over the resulting internal table.
4. If either *AgencyID* or *TravelID* is initial, read the data of the related parent entity.



Hint:

Use addition *BY* of EML statement *READ ENTITY* or *READ ENTITIES*, followed by the name of the to parent association.

5. Retrieve the values of *AgencyID* and *TravelID* from the related flight travel and update the current flight travel instance with these values .
6. If *ItemID* is initial, read all travel items of the related flight travel.



Hint:

Use addition *BY* of EML statement *READ ENTITY* or *READ ENTITIES*, followed by the name of the composition association.

7. In a loop over all travel instances of the same flight travel, find the largest value of *ItemID*, increase it by 10, and update the value for the current flight travel.
8. Use EML statement *MODIFY ENTITY* or *MODIFY ENTITIES* to update all affected flight travel items with the new values for the semantic key fields.
9. Activate your behavior pool (global class *ZBP_##_L_TRAVEL*) and re-test the preview of your SAP Fiori elements app.

Task 3: Implement Validation for Flight Class

For the child entity (*Z##_L_TravelItem*), define a validation for the flight class. Add the validation to the *draft prepare* action and generate the implementation method. Issue suitable error messages from message class *DEVS4D437* if the value is initial or if it does not match the fixed values of domain *S_CLASS*.



Hint:

Instead of hard coding the three possible values, we recommend to read from CDS view entity *D437_I_FlightClass*.



Caution:

Remember to issue the messages as `state` messages and do not forget to supply the path to the related root entity.

1. For the child entity (*Z##_I_TravelItem*), define a new validation (suggested name: `validateFlightClass`).
2. Specify basic operation `Create` and changes to field `FlightClass` as triggers for the validation.
3. Add the new validation to the list of validations that are to be performed during the `draft prepare` action of the root entity.



Hint:

Address the validation in the following form: <child entity alias>~<validation name>.

4. Use the available quick fix to generate the implementation method for the validation in the behavior handler class for flight travel items.
5. Edit the validation implementation method. Read the flight class and the technical key of the related root entity (flight travel).
6. Loop over all affected flight travel items and issue a suitable error message from message class *DEVS4D437* if the value is initial.



Caution:

Remember to provide a value for parameter `%state_area` of the message object constructor and to delete previous messages with the same value for `%state_area`. Also make sure you supply parameter `%path` with the complete key of the related root entity, including the `%is_draft` flag.

7. If the flight class is not initial, implement an existence check, reading from CDS view entity *D437_I_FlightClass* and issue a suitable error message from message class *DEVS4D437* if the value is not valid.
8. Activate your behavior pool (ABAP class *BP_D437E_I_TRAVEL*) and re-test the preview for your SAP Fiori elements app.

Task 4: Optional: Implement More Validations

For the child entity (*Z##_I_TravelItem*), define and implement a validation for the flight data (*CarrierID*, *ConnectionID*, and *FlightDate*). Also make sure that the *FlightDate* lies in the future.

1. For the child entity (*Z##_I_TravelItem*), define a new validation (suggested name: *validateFlightDate*) and issue a suitable error messages from message class *DEVS4D437* if the flight date is initial or if it lies in the past.
2. For the same entity, define a new validation (suggested name: *validateFlight*) and issue suitable error messages from message class *DEVS4D437* if any of the fields and *CarrierID*, *ConnectionID*, or *FlightDate* are initial or do not correspond to an existing flight in database table *SFLIGHT*.



Note:

Instead of reading from the database table directly, we recommend to read from CDS view entity *D437_I_Flight*.

3. Activate your behavior pool (ABAP class *BP_D437E_I_TRAVEL*) and re-test the preview for your SAP Fiori elements app.

Implement the Behavior of a Composite RAP Business Object

Business Scenario

In this exercise, you extend the behavior definition for the child entity with basic operations (create, update, delete) and you define and implement some determinations and validations.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 19: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Definition)	D437E_I_TRAVEL
CDS Behavior Definition (Projection)	D437E_C_TRAVEL
ABAP Class	BP_D437E_I_TRAVEL

Task 1: Basic Operations for the Child Entity

Enable the basic operations (create, update, delete) for the child entity (`Z##_I_TravelItem`), in both, the RAP Business Object and in the SAP Fiori elements app.

1. Edit the behavior definition for your RAP Business Object (`Z##_I_TRAVEL`). Scroll down to the behavior definition for the child entity (`Z##_I_TravelItem`) and add the basic operations (update, delete).

Why do you get a syntax error when you add the basic operation `create`, too?

For child entities, the activation and deactivation of operation `create` takes place in the root entity using the `create` addition to the composition association.

2. scroll up to the behavior definition of the root entity (`Z##_I_Travel`) and enable the association to the child entity for `create`.
 - a) Within the curly brackets after the name of the association, add `create`;
 - b) See source code extract from the model solution.

3. Edit the behavior projection for your RAP Business Object (`Z##_C_TRAVEL`) and make the new basic operations available in the OData service.
 - a) Add `use update` and `use delete` to the behavior projection.
 - b) See the source code extract from the model solution.
4. Similarly, add the `create` property of the `composition` association to the behavior projection.
 - a) Add `create;` within the curly brackets of the `use association` statement in the behavior projection of the root entity.
 - b) See source code extract from the model solution.
5. Activate the behavior projection and refresh the preview for the SAP Fiori elements app. Make sure the second facet of the *Object page* offers a *Create* button and *Delete* button.
 - a) Perform this step as before.
 - b) Compare your source code to the following extract from the model solution:

Behavior Definition **D437E_I_TRAVEL**

```
***** Business Object Travel *****/
managed implementation in class bp_d437e_i_travel unique;
with draft;

***** Root Entity Travel *****/
define behavior for D437e_I_Travel alias Travel
persistent table d437e_travel
draft table d437e_travel_d
lock master
total etag LocalChangedAt
etag master ChangedAt
authorization master ( instance )
{
  ...
  association _TravelItem { create; with draft;  }
  ...
}

***** Child Entity Travel Item *****/
define behavior for D437e_I_TravelItem alias Item
persistent table d437e_tritem
draft table d437e_tritem_d
lock dependent by _Travel
authorization dependent by _Travel
etag master ChangedAt
{
  ...
  update;
  delete;
}
```

```

...
}

Behavior Definition D437E_C_TRAVEL

/********************* Business Object Travel *****/
projection;
use draft;

/****************** Root Entity Travel *****/
define behavior for D437e_C_Travel alias Travel
use etag
{

}

/****************** Child Entity Travel Item *****/
define behavior for D437e_C_TravelItem alias Item
use etag
{

    use update;
    use delete;

    ...
}

```

Task 2: Determination for Semantic Key of Child Entity

Define a determination for the child entity of your RAP Business Object (suggested name: *determineSemanticKey*). Implement the determination to set the values of fields *AgencyID* and *TravelID* to the same values as in the related instance of the root entity (*Z##_L_Travel*).

To set the value of field *ItemID*, find the largest item number used in the same flight travel so far and increase it by 10. Remember to evaluate both the active travel items and the draft travel items.

1. Open the behavior definition for your RAP Business Object and scroll down to the behavior definition for your child entity (*Z##_L_TravelItem*)
 - a) Perform this step as before.
2. Define a new determination for this child entity (suggested name: *determineSemanticKey*) and make sure that it is only executed during creation of a new instance.
 - a) See the source code extract from the model solution.
3. Edit the implementation method for the determination. use EML statement *READ ENTITY* or *READ ENTITIES* to read the data of all affected flight travel items and implement a loop over the resulting internal table.
 - a) See the source code extract from the model solution.
4. If either *AgencyID* or *TravelID* is initial, read the data of the related parent entity.



Hint:

Use addition BY of EML statement *READ ENTITY* or *READ ENTITIES*, followed by the name of the to parent association.

- a) See the source code extract from the model solution.
5. Retrieve the values of *AgencyID* and *TravelID* from the related flight travel and update the current flight travel instance with these values .
- a) See source code extract from the model solution.
6. If *ItemID* is initial, read all travel items of the related flight travel.



Hint:

Use addition BY of EML statement *READ ENTITY* or *READ ENTITIES*, followed by the name of the composition association.

- a) See the source code extract from the model solution.
7. In a loop over all travel instances of the same flight travel, find the largest value of *ItemID*, increase it by 10, and update the value for the current flight travel.
- a) See the source code extract from the model solution.
8. Use EML statement *MODIFY ENTITY* or *MODIFY ENTITIES* to update all affected flight travel items with the new values for the semantic key fields.
- a) See the source code extract from the model solution.
9. Activate your behavior pool (global class ZBP_##_I_TRAVEL) and re-test the preview of your SAP Fiori elements app.
- a) Perform this step as before.
 - b) Compare your source code to the following extract from the model solution:

Behavior Definition D437E_I_TRAVEL

```
***** Business Object Travel *****/
managed implementation in class bp_d437e_i_travel unique;
with draft;

***** Root Entity Travel *****/
define behavior for D437e_I_Travel alias Travel
persistent table d437e_travel
draft table d437e_travel_d
lock master
total etag LocalChangedAt
etag master ChangedAt
authorization master ( instance )
{
  ...
}
```

```

***** Child Entity Travel Item *****

define behavior for D437e_I_TravelItem alias Item
  persistent table d437e_tritem
  draft table d437e_tritem_d
  lock dependent by _Travel
  authorization dependent by _Travel
  etag master ChangedAt
{
  ...
  determination determineSemanticKey on modify { create; }
  ...
}

```

ABAP Class BP_D437E_I_TRAVEL

```

METHOD determinesemantickey.

* read semantic key data of all affected travel items
***** READ ENTITY IN LOCAL MODE d437e_i_travelitem
      FIELDS ( agencyid travelid itemid )
      WITH CORRESPONDING #( keys )
      RESULT DATA(lt_items).

* Loop over all affected travel items
***** LOOP AT lt_items ASSIGNING FIELD-SYMBOL(<ls_item>).

* Retrieve AgencyID and TravelID if not initial
***** IF <ls_item>-agencyid IS INITIAL
      OR <ls_item>-travelid IS INITIAL.

      " Read parent entity data (travel)
      "for this child entity (travel item) by association
      READ ENTITY IN LOCAL MODE d437e_i_travelitem
          BY \_travel FIELDS ( agencyid travelid )
          WITH value #( ( %tky = <ls_item>-%tky ) )
          RESULT DATA(lt_travels).

      <ls_item>-agencyid = lt_travels[ 1 ]-agencyid.
      <ls_item>-travelid = lt_travels[ 1 ]-travelid.

    ENDIF.

* Retrieve ItemID if not initial
***** IF <ls_item>-itemid IS INITIAL.

      " read all child entities (travel items)
      " assigned to the same parent entity (travel) by association

      READ ENTITY IN LOCAL MODE d437e_i_travel
          BY \_travelitem FIELDS( itemid )
          WITH VALUE #( ( %tky = lt_travels[ 1 ]-%tky ) )
          RESULT DATA(lt_other_items).

```

```

    " find maximum item number
LOOP AT lt_other_items ASSIGNING FIELD-SYMBOL(<ls_other_item>).
    IF <ls_other_item>-itemid > <ls_item>-itemid.
        <ls_item>-itemid = <ls_other_item>-itemid.
    ENDIF.
ENDLOOP.
"
<ls_item>-itemid = <ls_item>-itemid + 10.
ENDIF.

ENDLOOP.

* Update flight travel items with new data
*****MODIFY ENTITY IN LOCAL MODE d437e_i_travelitem
    UPDATE FIELDS ( agencyid traveldid itemid )
    WITH CORRESPONDING #( lt_items ).

ENDMETHOD.
```

Task 3: Implement Validation for Flight Class

For the child entity (*Z##_I_TravelItem*), define a validation for the flight class. Add the validation to the *draft prepare* action and generate the implementation method. Issue suitable error messages from message class *DEVS4D437* if the value is initial or if it does not match the fixed values of domain *S_CLASS*.



Hint:

Instead of hard coding the three possible values, we recommend to read from CDS view entity *D437_I_FlightClass*.



Caution:

Remember to issue the messages as state messages and do not forget to supply the path to the related root entity.

1. For the child entity (*Z##_I_TravelItem*), define a new validation (suggested name: *validateFlightClass*).
 - a) Add statement *validation* to the behavior definition, followed by the name of the new validation.
 - b) See the source code extract from the model solution.
2. Specify basic operation *Create* and changes to field *FlightClass* as triggers for the validation.
 - a) See the source code extract from the model solution.
3. Add the new validation to the list of validations that are to be performed during the *draft prepare* action of the root entity.



Hint:

Address the validation in the following form: <child entity alias>~<validation name>.

- a) See the source code extract from the model solution.
4. Use the available quick fix to generate the implementation method for the validation in the behavior handler class for flight travel items.
 - a) Place the cursor on the name of the validation, open the context menu and choose *Quick Fix*.
 - b) Choose the available quick fix and press Enter.
 5. Edit the validation implementation method. Read the flight class and the technical key of the related root entity (flight travel).
 6. Loop over all affected flight travel items and issue a suitable error message from message class *DEVS4D437* if the value is initial.



Caution:

Remember to provide a value for parameter `%state_area` of the message object constructor and to delete previous messages with the same value for `%state_area`. Also make sure you supply parameter `%path` with the complete key of the related root entity, including the `%is_draft` flag.

- a) See source code extract from the model solution.
7. If the flight class is not initial, implement an existence check, reading from CDS view entity *D437_I_FlightClass* and issue a suitable error message from message class *DEVS4D437* if the value is not valid.
 - a) See the source code extract from the model solution.
 8. Activate your behavior pool (ABAP class *BP_D437E_I_TRAVEL*) and re-test the preview for your SAP Fiori elements app.
 - a) Perform this step as before
 - b) Compare your code to the following extract from the model solution:

Behavior Definition **D437E_I_TRAVEL**

```
/***** Business Object Travel *****/
managed implementation in class bp_d437e_i_travel unique;
with draft;

/***** Root Entity Travel *****/
define behavior for D437e_I_Travel alias Travel
persistent table d437e_travel
draft table d437e_travel_d
lock master
total etag LocalChangedAt
etag master ChangedAt
authorization master ( instance )
{
  ...
  draft determine action Prepare
}
```

```

{
    validation validateCustomer;
    validation validateStartDate;
    validation validateEndDate;
    validation validateSequence;

    validation Item~validateFlightClass;
}

...
}

/***** Child Entity Travel Item *****/
define behavior for D437e_I_TravelItem alias Item
persistent table d437e_tritem
draft table d437e_tritem_d
lock dependent by _Travel
authorization dependent by _Travel
etag master ChangedAt
{
...
validation validateFlightClass on save
{ create;
  field FlightClass;
}
...
}

```

ABAP Class **BP_D437E_I_TRAVEL**

```

METHOD validateflightclass.

CONSTANTS c_area TYPE string VALUE `FLIGHTCLASS`.

READ ENTITY IN LOCAL MODE d437e_i_travelitem
FIELDS ( flightclass trguid )
WITH CORRESPONDING #( keys )
RESULT DATA(lt_items).

LOOP AT lt_items ASSIGNING FIELD-SYMBOL(<ls_item>).

APPEND VALUE #( %tky      = <ls_item>-%tky
                %state_area = c_area )
TO reported-item.

IF <ls_item>-flightclass IS INITIAL.

APPEND CORRESPONDING #( <ls_item> )
TO failed-item.
APPEND VALUE #(
    %tky      = <ls_item>-%tky
    %element = VALUE #( flightclass = if_abap_behv=>mk-on )
    %msg      = NEW cm_devs4d437(

```

```

        textid    = cm_devs4d437=>field_empty
        severity = cm_devs4d437=>severity-error
    )
    %state_area = c_area
    %path       = VALUE #( travel-%is_draft = <ls_item>-
%is_draft
                                travel-trguid     = <ls_item>-
trguid )
                                )
TO reported-item.

ELSE.

"existence check for flight class
SELECT SINGLE @abap_true
      FROM d437_i_flightclass
      WHERE flightclass = <ls_item>-flightclass
      INTO @DATA(lv_exists)
.

IF lv_exists <> abap_true.

APPEND CORRESPONDING #( <ls_item> )
      TO failed-item.
APPEND VALUE #(
      %tky      = <ls_item>-%tky
      %element = VALUE #( flightclass = if_abap_behv=>mk-
on )
      %msg      = NEW cm_devs4d437(
          textid    = cm_devs4d437=>class_invalid
          severity = cm_devs4d437=>severity-error
          flightclass = <ls_item>-flightclass
      )
      %state_area = c_area
      %path       = VALUE #( travel-%is_draft = <ls_item>-
%is_draft
                                travel-trguid     = <ls_item>-
trguid )
                                )
TO reported-item.

ENDIF.
ENDIF.

ENDLOOP.

ENDMETHOD.
```

Task 4: Optional: Implement More Validations

For the child entity (*Z##_I_TravelItem*), define and implement a validation for the flight data (*CarrierID*, *ConnectionID*, and *FlightDate*). Also make sure that the *FlightDate* lies in the future.

1. For the child entity (*Z##_I_TravelItem*), define a new validation (suggested name: *validateFlightDate* and issue a suitable error messages from message class *DEVS4D437* if the flight date is initial or if it lies in the past.
 - a) See source code extract from the model solution.
2. For the same entity, define a new validation (suggested name: *validateFlight*) and issue suitable error messages from message class *DEVS4D437* if any of the fields and *CarrierID*, *ConnectionID*, or *FlightDate* are initial or do not correspond to an existing flight in database table *SFLIGHT*.

**Note:**

Instead of reading from the database table directly, we recommend to read from CDS view entity *D437_I_Flight*.

- a) See the source code extract from the model solution.
3. Activate your behavior pool (ABAP class *BP_D437E_I_TRAVEL* and re-test the preview for your SAP Fiori elements app.
- a) Perform this step as before
 - b) Compare your code to the following extract from the model solution:

Behavior Definition D437E_I_TRAVEL

```

/***** Business Object Travel *****/
managed implementation in class bp_d437e_i_travel unique;
with draft;

/***** Root Entity Travel *****/
define behavior for D437e_I_Travel alias Travel
persistent table d437e_travel
draft table d437e_travel_d
lock master
total etag LocalChangedAt
etag master ChangedAt
authorization master ( instance )
{

...
draft determine action Prepare
{
    validation validateCustomer;
    validation validateStartDate;
    validation validateEndDate;
    validation validateSequence;

    validation Item~validateFlightClass;

    validation Item~validateFlightDate;
    validation Item~validateFlight;
}

...
}

/***** Child Entity Travel Item *****/
define behavior for D437e_I_TravelItem alias Item
persistent table d437e_tritem
draft table d437e_tritem_d
lock dependent by _Travel

```

```

authorization dependent by _Travel
etag master ChangedAt
{
  ...
  validation validateFlightDate on save
  { create;
    field FlightDate;
  }

  validation validateFlight on save
  { create;
    field CarrierId, ConnectionId, FlightDate;
  }

  ...
}

```

ABAP Class BP_D437E_I_TRAVEL

```

METHOD validateflightdate.

CONSTANTS c_area TYPE string VALUE `FLIGHTDATE`.

READ ENTITY IN LOCAL MODE d437e_i_travelitem
  FIELDS ( flightdate trguid )
  WITH CORRESPONDING #( keys )
  RESULT DATA(lt_items).

LOOP AT lt_items ASSIGNING FIELD-SYMBOL(<ls_item>).

  APPEND VALUE #( %tky      = <ls_item>-%tky
                  %state_area = c_area
                )
  TO reported-item.

  IF <ls_item>-flightdate IS INITIAL.

    APPEND CORRESPONDING #( <ls_item> )
    TO failed-item.
    APPEND VALUE #(
      %tky      = <ls_item>-%tky
      %element = VALUE #( flightdate = if_abap_behv=>mk-on )
      %msg     = NEW cm_devs4d437(
        textid   = cm_devs4d437=>field_empty
        severity = cm_devs4d437=>severity-error
      )
      %state_area = c_area
      %path    = VALUE #( travel-%is_draft = <ls_item>-
        %is_draft
                                travel-trguid      = <ls_item>-
        trguid )
    )
    TO reported-item.

  ELSEIF <ls_item>-flightdate < sy-datum.
    " or use cl_abap_context_info=>get_system_date( )

    APPEND CORRESPONDING #( <ls_item> )

```

```

        TO failed-item.
APPEND VALUE #(
    %tky      = <ls_item>-%tky
    %element = VALUE #( flightdate = if_abap_behv=>mk-on )
    %msg     = NEW cm_devs4d437(
        textid   = cm_devs4d437=>flight_date_past
        severity = cm_devs4d437=>severity-error
    )
    %state_area = c_area
    %path     = VALUE #( travel-%is_draft = <ls_item>-
%is_draft
                                travel-trguid = <ls_item>-
trguid )
)
TO reported-item.

ENDIF.

ENDLOOP.

ENDMETHOD.

METHOD validateflight.

CONSTANTS c_area TYPE string VALUE `FLIGHT`.

READ ENTITY IN LOCAL MODE d437e_i_travelitem
FIELDS ( carrierid connectionid flightdate trguid )
WITH CORRESPONDING #( keys )
RESULT DATA(lt_items).

LOOP AT lt_items ASSIGNING FIELD-SYMBOL(<ls_item>).

APPEND VALUE #( %tky      = <ls_item>-%tky
                %state_area = c_area )
TO reported-item.

IF <ls_item>-carrierid IS INITIAL.

APPEND CORRESPONDING #( <ls_item> )
TO failed-item.
APPEND VALUE #(
    %tky      = <ls_item>-%tky
    %element = VALUE #( carrierid = if_abap_behv=>mk-on )
    %msg     = NEW cm_devs4d437(
        textid   = cm_devs4d437=>field_empty
        severity = cm_devs4d437=>severity-error
    )
    %state_area = c_area
    %path     = VALUE #( travel-%is_draft = <ls_item>-
%is_draft
                                travel-trguid = <ls_item>-
trguid )
)
TO reported-item.
ELSEIF <ls_item>-connectionid IS INITIAL.

APPEND CORRESPONDING #( <ls_item> )
TO failed-item.
APPEND VALUE #(
    %tky      = <ls_item>-%tky
    %element = VALUE #( connectionid = if_abap_behv=>mk-on )
)

```

```

        %msg      = NEW cm_devs4d437(
                      textid   = cm_devs4d437=>field_empty
                      severity = cm_devs4d437=>severity-error
                      )
        %state_area = c_area
        %path       = VALUE #( travel-%is_draft = <ls_item>-
%is_draft
                                         travel-trguid     = <ls_item>-
                                         trguid )
                                         )

        TO reported-item.

ELSE.

"existence check for flight
SELECT SINGLE @abap_true
  FROM d437_i_flight
 WHERE carrierid = @<ls_item>-carrierid
   AND connectionid = @<ls_item>-connectionid
   AND flightdate    = @<ls_item>-flightdate
 INTO @DATA(lv_exists)
 .

IF lv_exists <> abap_true.

APPEND CORRESPONDING #( <ls_item> )
  TO failed-item.
APPEND VALUE #(
  %tky      = <ls_item>-%tky
  %element  = VALUE #( flightclass = if_abap_behv=>mk-
on )
  %msg      = NEW cm_devs4d437(
                      textid   = cm_devs4d437=>flight_not_exist
                      severity = cm_devs4d437=>severity-error
                      carrierid = <ls_item>-carrierid
                      connectionid = <ls_item>-connectionid
                      flightdate = <ls_item>-flightdate
                      )
  %state_area = c_area
  %path       = VALUE #( travel-%is_draft = <ls_item>-
%is_draft
                                         travel-trguid     = <ls_item>-
                                         trguid )
                                         )

        TO reported-item.

ENDIF.
ENDIF.

ENDLOOP.

ENDMETHOD.
```



LESSON SUMMARY

You should now be able to:

- Access composite business objects with EML

UNIT 6

Transactional Apps with Unmanaged Business Object

Lesson 1

Understanding Data Access in Unmanaged Implementations	330
Exercise 18: Define an Unmanaged Business Object	335

Lesson 2

Implementing Unmanaged Business Objects	343
Exercise 19: Implement an Unmanaged Business Object	347

UNIT OBJECTIVES

- Define the behavior for an unmanaged Business Object
- Implement data access of an unmanaged Business Object

Understanding Data Access in Unmanaged Implementations

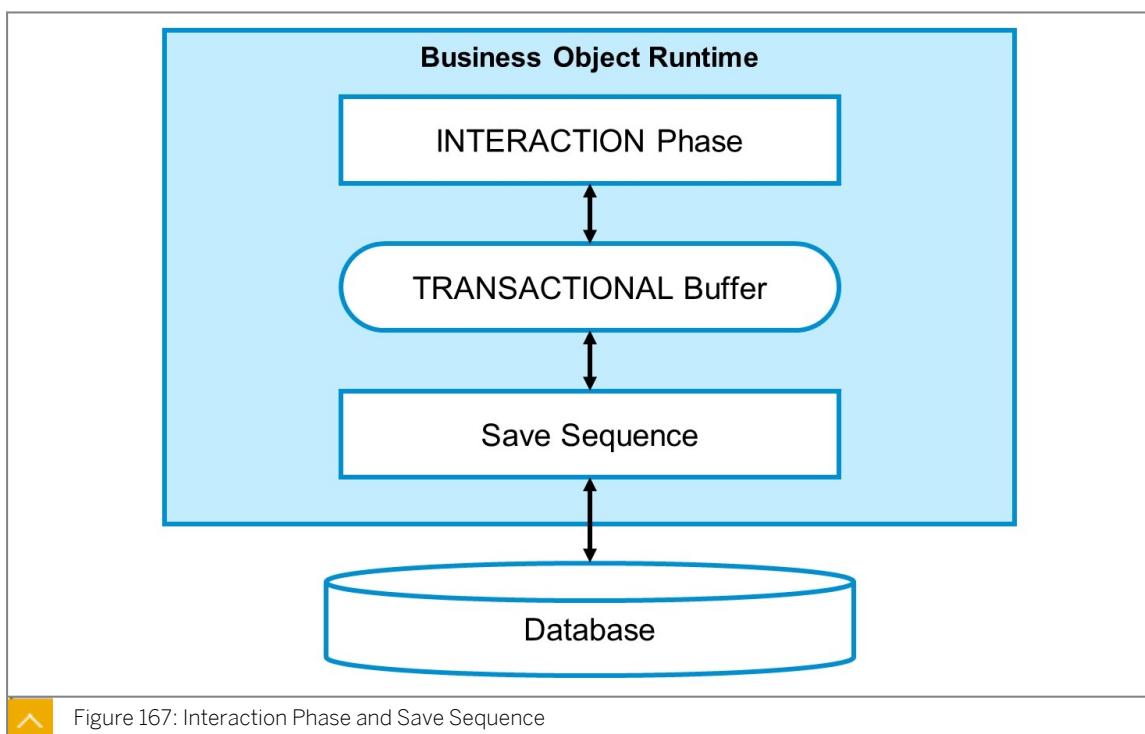


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define the behavior for an unmanaged Business Object

Interaction Phase and 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 transactional buffer, which represents the state.
- After all changes are performed, the data should be persisted. This is implemented within the save sequence.

In a managed implementation scenario for RAP business objects, the RAP provisioning framework defines and manages the transactional buffer. During interaction phase, it automatically fills the buffer with data from the persistent tables, handles write access to the

buffer during standard operations (create, update, and delete), and takes care of writing the changed buffer content back to the persistent tables during the save sequence.

Unmanaged versus Managed



	Managed BO	Unmanaged BO
Recommended for	New development without existing code	Reuse of existing business logic
Transactional Buffer and Standard BO operations	Handled by RAP framework	Implemented in ABAP behavior pool
Persisting of application buffer during Save	Handled by RAP framework	Implemented in ABAP behavior pool
Non-Standard BO Operations	Implemented in Actions	
Determinations and Validations	Available (draft and non-draft)	Only if BO is draft-enabled
Pessimistic Concurrency Control (Locks)	Handled by framework (implementation optional)	Implemented in ABAP behavior pool
Optimistic Concurrency Control (ETag Handling)	Handled by RAP framework	Requires implementation of READ operation

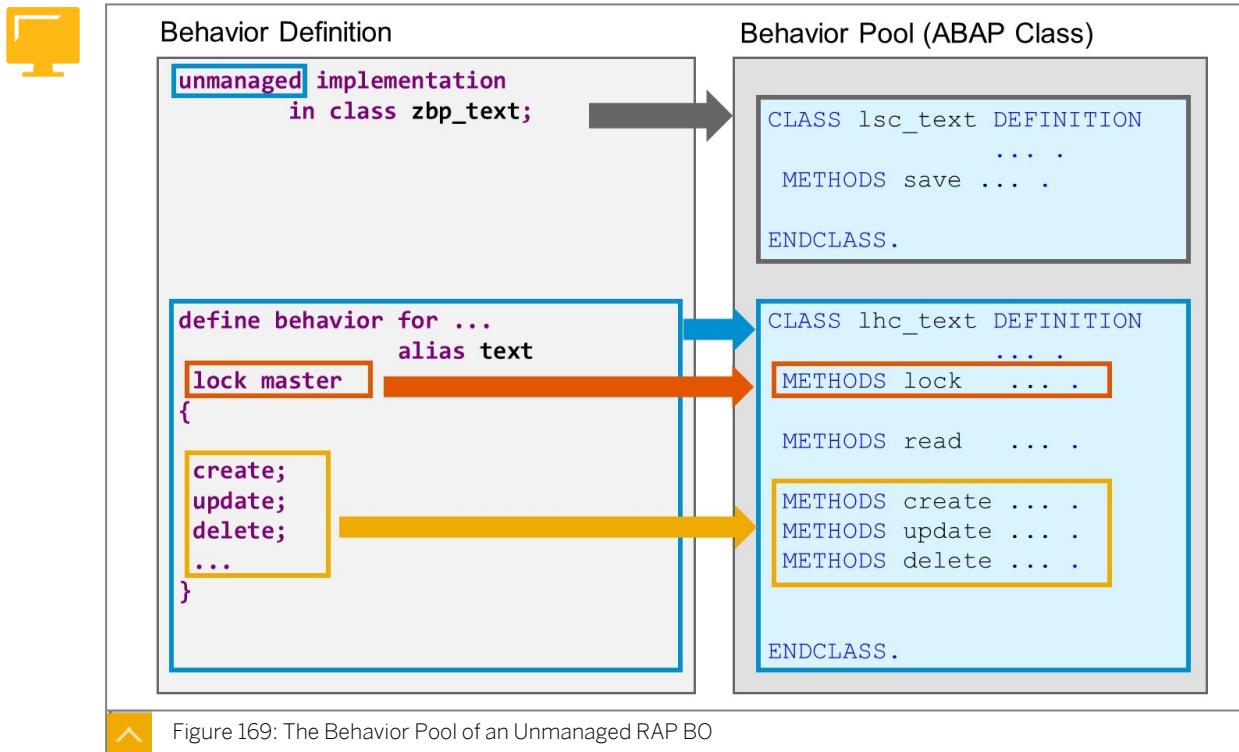
 Figure 168: Comparison of Managed and Unmanaged Business Objects

In the unmanaged implementation type, the transactional buffer, the standard BO operations, and the database access must be implemented in the ABAP behavior pool. Unmanaged implementation is recommended for development scenarios in which business logic already exists and is intended to be reused. If your start your development from scratch, or if not much more than the persistent database tables exist, we recommend that you follow the managed approach.

Some additional restrictions apply for the unmanaged implementation scenario:

- Determinations and Validations are not available, unless the business object is draft enabled.
- Pessimistic concurrency control (locks) are not mandatory. If they are required, they have to be implemented manually.
- Optimistic concurrency control (ETag handling) requires the manual implementation of a READ method that provides the latest version of the entity instance.

Handler Class and Saver Class



If a behavior definition uses the unmanaged implementation type, it is mandatory to specify a behavior pool.

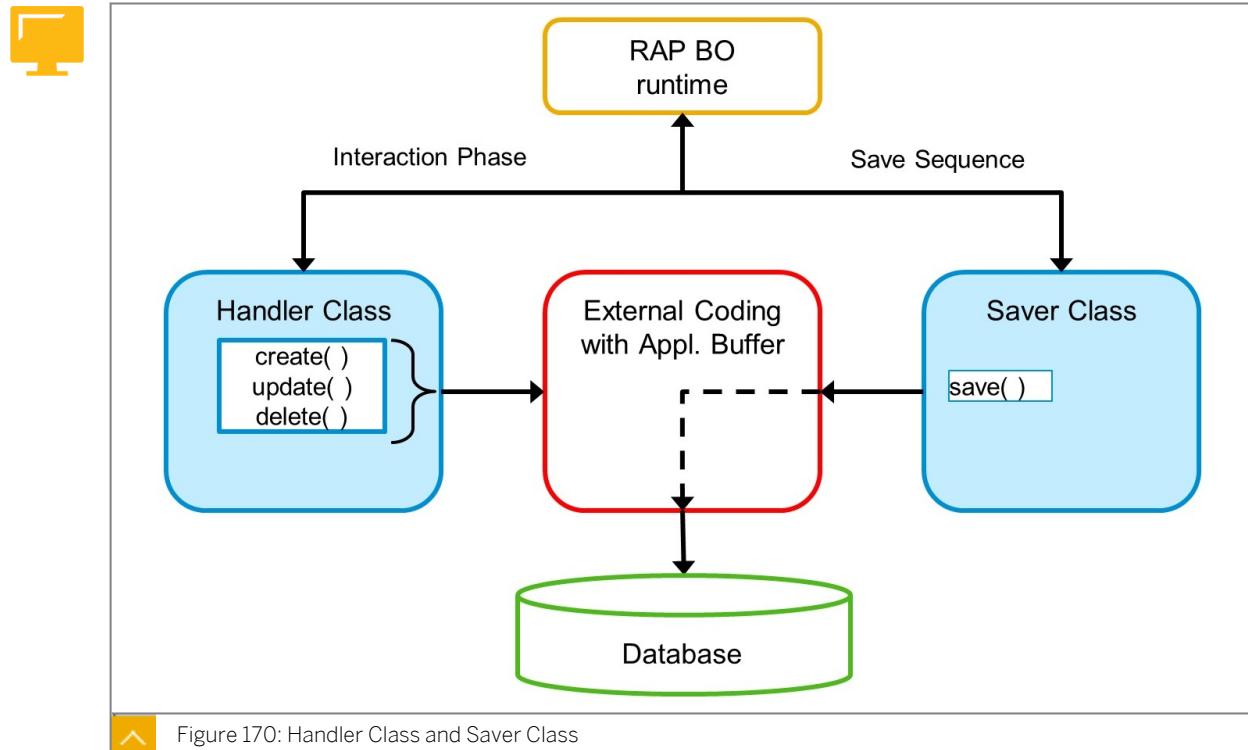
Like in the managed scenario, this behavior pool contains one local handler class (`lhc`) for each entity of the business object. The methods of the local handler classes are triggered during the interaction phase.

In addition to the local handler classes, the behavior pool of an unmanaged RAP BO contains a local saver class (`lsc`). There is not a saver class for each entity but only one saver class for the business object as a whole. The framework calls the methods of the local saver class during the save sequence.

The ABAP syntax check issues a warning if the saver class does not at least define a FOR SAVE method. Similarly, it issues warnings if a FOR READ method is missing in the local handler classes.

In the unmanaged scenario, it is optional to set the root entity as lock master. If the addition `lock master` is present, the corresponding handler class has to define and implement a FOR LOCK method.

Similarly, corresponding FOR MODIFY methods are required if the behavior definition enables the standard operations create, update, and delete.

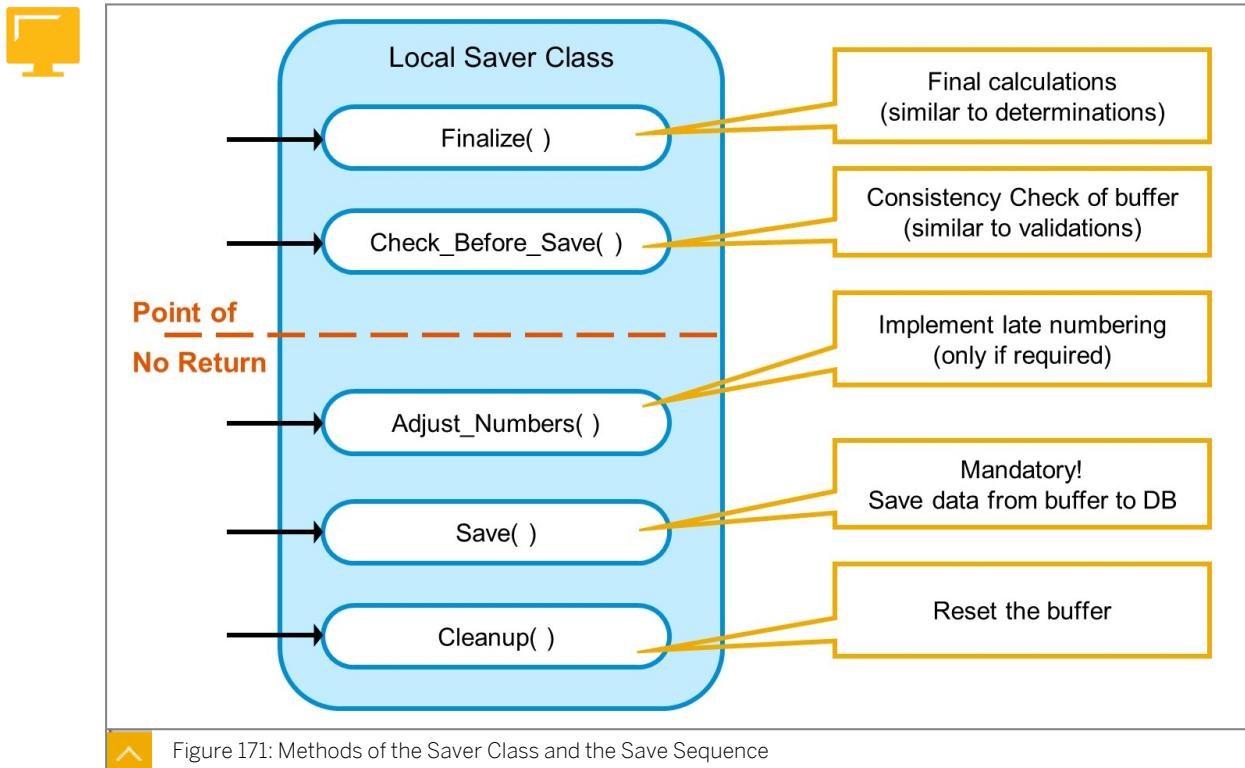


If a RAP BO consumer, for example an OData service, sends a modify request to the RAP BO, the framework triggers the execution of the respective create, update, and delete methods in the handler classes.

The implementation of these methods validates the data changes and stores them in an application buffer. Usually, the validation logic and the application buffer are implemented somewhere outside the behavior pool, for example, in an already existing function module or global class.

When, during the save sequence, the RAP BO runtime triggers the `save()` method of the saver class, the implementation of this class triggers the copying of the application buffer content to the database.

The Save Sequence



As well as the mandatory FOR SAVE method, you can define and implement several other methods in the local saver class. They are executed in a given sequence during the save phase, that is, after at least one successful modification was performed during the interaction phase.

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. The `save()` call persists all BO instance data from the transactional buffer in the database. The final `cleanup()` call resets the transactional buffer.

Unit 6

Exercise 18

Define an Unmanaged Business Object

Business Scenario

In this exercise, you copy the relevant repository objects to define and project a CDS-based data model for travel agencies. Then you define, preview and test an OData UI service.

When you add the behavior definition and behavior projection, you choose the *unmanaged* approach to re-use the logic of the existing function group *D437AGENCY*.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 20: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	D437F_I_AGENCY
CDS Behavior Definition (Projection)	D437F_C_AGENCY

Task 1: Copy Templates for Data Model

Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace *D437T* with *Z##*. Make sure the copies refer to each other and not to the template objects.

Table 21: Template

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	D437T_I_AGENCY
CDS Data Definition (Projection)	D437T_C_AGENCY
CDS Metadata Extension	D437T_C_AGENCY

1. Create a copy of data definition *D437T_I_AGENCY* (suggested name: *Z##_I_AGENCY*) and activate the new data definition.
2. Create a copy of data definition *D437T_C_AGENCY* (suggested name: *Z##_C_AGENCY*). Replace the CDS View after *projection on* with the name of your own data model view for travel agencies (*Z##_I_Agency*) and activate the new data definition.
3. Create a copy of metadata extension *D437T_C_AGENCY* (suggested name: *Z##_C_AGENCY*). Replace the CDS view after *annotate view* with your own projection view for travel agencies (*Z##_C_AGENCY*) and activate the new metadata extension.

Task 2: Define and Preview OData UI Service

Create a service definition (suggested name: Z##_UI_AGENCY) and a service binding (suggested name: Z##_UI_AGENCY_02) that exposes your projection view, Z##_C_Agency, as an OData V2 UI Service.

1. Create a service definition (suggested name: Z##_UI_AGENCY) for your projection view Z##_C_Agency and activate it.
2. Create a service binding (suggested name: Z##_UI_AGENCY_02) that binds your service, Z##_UI_AGENCY, as an OData V2 UI Service.
3. Publish the local service endpoint.
4. Preview the service as SAP Fiori elements app.

Task 3: Define and Project Behavior (unmanaged)

Create a behavior definition for your data model view, Z##_I_Agency, with *Implementation Type* set to *unmanaged*. Generate the behavior pool (suggested name: ZBP##_AGENCY) and analyze it.

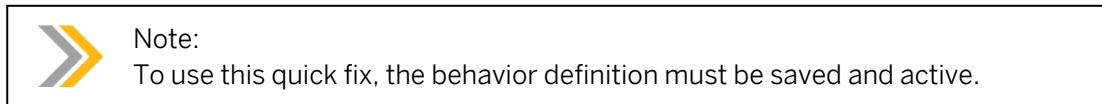
Create a behavior definition for your projection view Z##_C_Agency. For the moment, only project the basic operation *update*.

Test the SAP Fiori elements app and use debugging to analyze when the RAP framework executes various methods of the generated handler class and saver class.

1. Create a behavior definition for your data model view Z##_I_Agency. When prompted for the *Implementation Type*, change the default value to *unmanaged*.
2. Perform a syntax check for the new behavior definition.

Are there any syntax errors or warnings?

3. Define a suitable alias for the root entity of the RAP Business Object (suggested name: Agency).
4. Activate the behavior definition and use the *Quick Fix* function of the editor to generate the behavior implementation class.



5. Analyze the behavior implementation class.

Is there more than one local class?

What are the methods in those classes?

6. Activate the implementation class (behavior pool) and set a break-point at the beginning of each predefined method.
7. Create a behavior definition for your projection view `Z##_C_Agency`.
8. Define a suitable alias for projection of the root entity of your RAP Business Object (suggested name: `Agency`) and exclude the basic operations `create` and `delete` from the projection.
9. Activate the behavior projection and test your SAP Fiori elements app. Open a travel agency on the *Object* page, change to edit mode, make some changes, and save them.

Which methods are executed and in what sequence?

Define an Unmanaged Business Object

Business Scenario

In this exercise, you copy the relevant repository objects to define and project a CDS-based data model for travel agencies. Then you define, preview and test an OData UI service.

When you add the behavior definition and behavior projection, you choose the *unmanaged* approach to re-use the logic of the existing function group *D437AGENCY*.



Note:

In this exercise, replace ## with the number that your instructor assigned to you.

Table 20: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	<i>D437F_I_AGENCY</i>
CDS Behavior Definition (Projection)	<i>D437F_C_AGENCY</i>

Task 1: Copy Templates for Data Model

Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace *D437T* with *Z##*. Make sure the copies refer to each other and not to the template objects.

Table 21: Template

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	<i>D437T_I_AGENCY</i>
CDS Data Definition (Projection)	<i>D437T_C_AGENCY</i>
CDS Metadata Extension	<i>D437T_C_AGENCY</i>

1. Create a copy of data definition *D437T_I_AGENCY* (suggested name: *Z##_I_AGENCY*) and activate the new data definition.
 - a) Perform this step as before.
2. Create a copy of data definition *D437T_C_AGENCY* (suggested name: *Z##_C_AGENCY*). Replace the CDS View after *projection on* with the name of your own data model view for travel agencies (*Z##_I_Agency*) and activate the new data definition.
 - a) Perform this step as before.

3. Create a copy of metadata extension *D437T_C_AGENCY* (suggested name: *Z##_C_AGENCY*). Replace the CDS view after *annotate view* with your own projection view for travel agencies (*Z##_C_AGENCY*) and activate the new metadata extension.
 - a) Perform this step as before.

Task 2: Define and Preview OData UI Service

Create a service definition (suggested name: *Z##_UI_AGENCY*) and a service binding (suggested name: *Z##_UI_AGENCY_02*) that exposes your projection view, *Z##_C_Agency*, as an OData V2 UI Service.

1. Create a service definition (suggested name: *Z##_UI_AGENCY*) for your projection view *Z##_C_Agency* and activate it.
 - a) Perform this step as before.
2. Create a service binding (suggested name: *Z##_UI_AGENCY_02*) that binds your service, *Z##_UI_AGENCY*, as an OData V2 UI Service.
 - a) Perform this step as before.
3. Publish the local service endpoint.
 - a) Perform this step as before.
4. Preview the service as SAP Fiori elements app.
 - a) Perform this step as before.

Task 3: Define and Project Behavior (unmanaged)

Create a behavior definition for your data model view, *Z##_I_Agency*, with *Implementation Type* set to *unmanaged*. Generate the behavior pool (suggested name: *ZBP##_AGENCY*) and analyze it.

Create a behavior definition for your projection view *Z##_C_Agency*. For the moment, only project the basic operation *update*.

Test the SAP Fiori elements app and use debugging to analyze when the RAP framework executes various methods of the generated handler class and saver class.

1. Create a behavior definition for your data model view *Z##_I_Agency*. When prompted for the *Implementation Type*, change the default value to *unmanaged*.
 - a) In the *Project Explorer* view, open the context menu for the data definition of your data model view.
 - b) From the context menu, choose *New Behavior Definition*.
 - c) Enter a description, change the *Implementation Type* to *unmanaged* and choose *Next >*.
Note that you cannot change the suggested name of the new behavior definition.
 - d) Select the same transport request as before and choose *Finish*.
2. Perform a syntax check for the new behavior definition.
 - a) Press *Ctrl + F2* and analyze the *Problems* tab below the source code editor.

Are there any syntax errors or warnings?

There are no syntax errors. Unlike in the *managed* case, the lock master/lock dependent flag and the persistence definition are not mandatory. However, there is a syntax warning, because an existing implementation class must be specified in the *unmanaged* case, and this implementation class does not exist yet.

3. Define a suitable alias for the root entity of the RAP Business Object (suggested name: *Agency*).
 - a) See the source code extract from the model solution.
4. Activate the behavior definition and use the *Quick Fix* function of the editor to generate the behavior implementation class.



Note:

To use this quick fix, the behavior definition must be saved and active.

- a) Press **Ctrl + F3** to activate the repository object.
 - b) Either right-click somewhere on the name of the class and choose *Quick Fix* or choose the warning icon next to the *unmanaged* statement.
 - c) Double-click *Create behavior implementation class ZBP_##_I_AGENCY*.
 - d) Enter a description for implementation class and choose *Next >*.
 - e) Select the same transport request as before and choose *Finish*.
5. Analyze the behavior implementation class.
 - a) In the behavior pool (global class *ZBP_##_I_AGENCY*), navigate to the *Local Types* tab.

Is there more than one local class?

Yes, in addition to the handler class (*LHC_<Alias_Name>*), there is a saver class (*LSC_<Alias_Name>*).

What are the methods in those classes?

The handler has predefined methods `create()`, `delete()`, `update()`, and `read()`. The saver class has predefined methods `check_before_save()`, `finalize()`, and `save()`.

6. Activate the implementation class (behavior pool) and set a break-point at the beginning of each predefined method.
 - a) Press **Ctrl + F3**.
 - b) In the line with the `ENDMETHOD` statement, double-click the area left of the line number to set a break-point. Alternatively, you can use statement `BREAK-POINT` in the method implementation.
7. Create a behavior definition for your projection view *Z##_C_Agency*.

- a) Perform this step as before.
8. Define a suitable alias for projection of the root entity of your RAP Business Object (suggested name: *Agency*) and exclude the basic operations `create` and `delete` from the projection.
 - a) See the source code extract from the model solution.
 9. Activate the behavior projection and test your SAP Fiori elements app. Open a travel agency on the *Object page*, change to edit mode, make some changes, and save them.

Which methods are executed and in what sequence?

update() → finalize() → check_before_save() → save().

- a) Perform this step as before.
- b) Compare your source code to the following extract from the model solution.

Behavior Definition **D437F_I_AGENCY**:

```
unmanaged implementation in class zbp_00_i_agency unique;

define behavior for Z00_I_Agency alias Agency
{
  create;
  update;
  delete;
}
```

Behavior Definition **D437F_I_AGENCY**:

```
projection;

define behavior for Z00_C_Agency alias Agency
{
  // use create;
  // use update;
  // use delete;
}
```



LESSON SUMMARY

You should now be able to:

- Define the behavior for an unmanaged Business Object

Implementing Unmanaged Business Objects



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement data access of an unmanaged Business Object

Type Mapping



Behavior Implementation (ABAP Class)

```
DATA ls_struct TYPE d437_s_struct.  
DATA ls_entity TYPE STRUCTURE FOR ... d437_i_entity.
```

```
...
```

```
ls_struct-comp1 = ls_entity-field1.  
ls_struct-comp2 = ls_entity-field2.  
...
```

Prepare input for existing coding (structure *ls_struct*)

```
CALL FUNCTION ...  
CHANGING  
    cs_struct = ls_struct.
```

Call existing coding for example, function module

```
ls_entity-field1 = ls_struct-comp1.  
ls_entity-field2 = ls_struct-comp2.  
...
```

Copy output of existing coding to data for BO entity

Figure 172: The need for Field Mapping

Whenever existing code and its data types are to be reused in behavior pools of business objects, you need to perform a mapping between CDS field names and types and the corresponding legacy field names and types.

In the example, the existing code is a function module with a changing parameter. The parameter of this function module is typed with *d437_s_struct*, which is a structure type with components *comp1*, *comp2*, and so on. The RAP BO defines an entity *d437_i_entity* with fields named *field1*, *field2*, and so on.

Inside the handler methods, the RAP BO data is available in data objects that are based on derived types for the entity. To hand the information over to the function module, it has to be copied into a data object based on structure type *d437_s_struct* and, because the component types are not identical, normal *CORRESPONDING* does not help here.

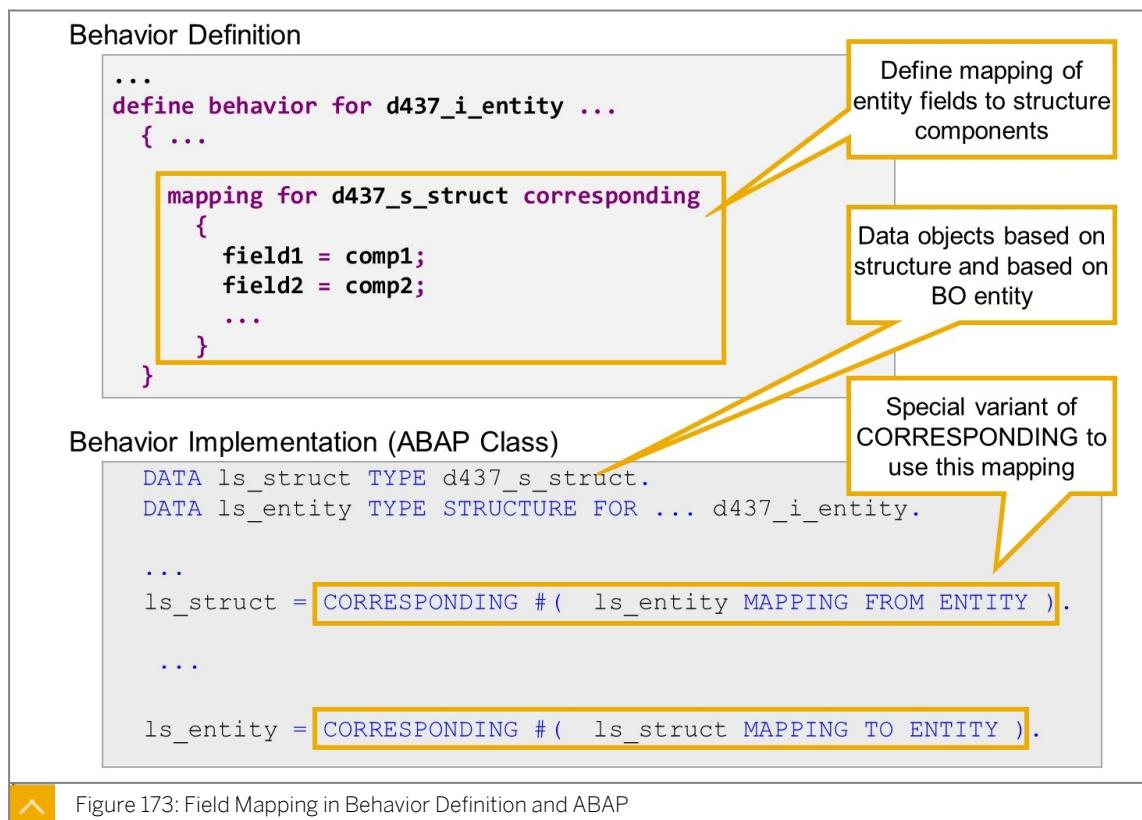


Figure 173: Field Mapping in Behavior Definition and ABAP

In RAP, it is possible to define a central, declarative mapping in the behavior definition and to use this mapping in the behavior implementation with a special variant of the CORRESPONDING expression.

This technique is particularly interesting for the unmanaged implementation type, which essentially represents a kind of wrapper for existing legacy functionality. But, with the managed implementation type, it can happen, for example, that the code for a determination or validation already exists, but is based on "old" (legacy) data types.

In the example, the behavior definition contains a mapping statement for structure type `d437_s_struct` with the pairs of entity fields and structure components.

The ABAP coding in the behavior implementation defines a data object `ls_struct`, which is typed with `d437_s_struct` and a data object `ls_entity` typed with a derived structure type for entity `d437_i_entity`.

When copying information from `ls_entity` to `ls_struct`, the mapping is used in a CORRESPONDING expression with the addition `MAPPING FROM ENTITY`. When copying information from `ls_struct` to `ls_entity`, the addition `MAPPING TO ENTITY` is used instead.

Control Mapping

 Behavior Implementation (ABAP Class)

```

DATA ls_struct    TYPE d437_s_struct.
DATA ls_structx  TYPE d437_s_structx.

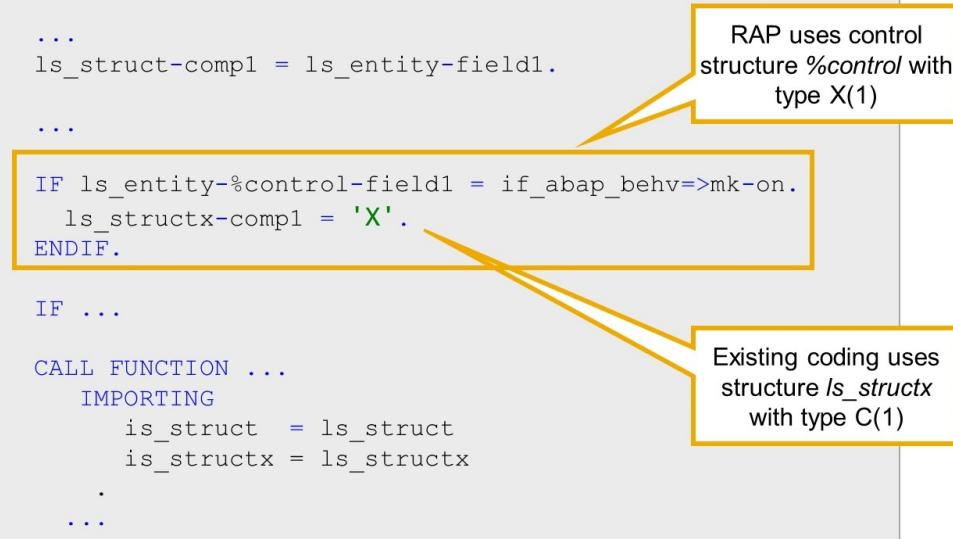
DATA ls_entity    TYPE STRUCTURE FOR ... d437_i_entity.

...
ls_struct-compl = ls_entity-field1.

...
IF ls_entity-%control-field1 = if_abap_behv=>mk-on.
  ls_structx-compl = 'X'.
ENDIF.

IF ...
CALL FUNCTION ...
  IMPORTING
    is_struct    = ls_struct
    is_structx   = ls_structx
  .
  ...

```



The diagram illustrates the mapping of control structures between RAP and legacy ABAP code. A yellow box highlights the RAP code: `ls_struct-compl = ls_entity-field1.` Another yellow box highlights the legacy code: `ls_structx-compl = 'X'.` A callout points from the RAP code to a note: "RAP uses control structure %control with type X(1)". A callout points from the legacy code to another note: "Existing coding uses structure ls_structx with type C(1)".

Figure 174: The Need for Control Type Mapping

In some legacy scenarios, as well as the dictionary type directly corresponding to the entity, there is a second dictionary type that contains the same components, but all of them have data type C(1).

This type is then used to indicate the fields in the main structure that are accessed by an operation (update, read, and so on).



Note:

Such type pairs are often used in BAPIs, for example, BAPIAD2VD/BAPIAD2VDX, where the control data element is BAPIUPDATE with the type C(1).

When calling such existing code, the actual parameter for such a control structure has to be filled with the values from the %CONTROL structure in the derived types.

As well as the possibly different field names, you have to consider the different concepts for bool-like types that are used in RAP and in legacy code. Where ABAP traditionally uses type C(1) with values 'X' and '' (Space), RAP uses the more modern approach of type X(1) with values hexadecimal values #01 and #00.

This could lead to rather lengthy coding. In the example, only the first component of %control1 is mapped to the first component of the legacy control structure ls_structx.



Behavior Definition

```
...
define behavior for d437_i_entity ...
{ ...

    mapping for d437_s_struct
        control d437_s_structx corresponding
    {
        TextID      = text_id;
        TextOwner   = text_owner;
    }
}
```

Behavior Implementation (ABAP Class)

```
DATA ls_structx TYPE d437_s_structx.
DATA ls_entity  TYPE STRUCTURE FOR ... d437_i_entity.

...
ls_structx = CORRESPONDING #( ls_entity MAPPING FROM ENTITY
                           USING CONTROL
                           ).
```



Figure 175: Control Mapping in Behavior Definition and ABAP Coding

It is possible to include a mapping definition for the control structure in the mapping for statement for the data structure. To do this, add keyword `control`, followed by the name of the legacy control structure.

In ABAP, add keywords `USING CONTROL` inside the `CORRESPONDING` expression if you want to populate a legacy control structure based on component `%control` of a derived data type.



Note:

For the opposite direction, use the following syntax: `ls_entity = CORRESPONDING #(ls_structx CHANGING CONTROL).`

Unit 6 Exercise 19

Implement an Unmanaged Business Object

In this exercise, you implement the behavior of your unmanaged RAP Business Object for flight travels.

You implement data access for basic operation `update` by calling the necessary function modules from function group `D437AGENCY`.

To improve convenience for the user, you make the RAP runtime request locks and trigger authority checks at an earlier stage. For the related implementations, you again call the relevant function modules from function group `D437AGENCY`.

Optionally, enable and implement the basic operations `create` and `delete`.



Note:

In this exercise, replace `##` with the number that your instructor assigned to you.

Table 22: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	<code>D437F_I_AGENCY</code>
CDS Behavior Definition (Projection)	<code>D437F_C_AGENCY</code>
ABAP Class	<code>BP_D437F_I_AGENCY</code>

Task 1: Analyze Function Group D437AGENCY

Analyze the functionality of function group `D437AGENCY`. Use the function modules of this function group to read and create travel agencies.

1. Open function group `D437AGENCY`.
2. Analyze the function group in *Project Explorer*.

How many function modules belong to this function group?

3. Analyze the interface of function module `D437_AGENCY_READ`.

What input does the function module expect?

What information does the function module return?

4. Perform a test of the function module to read details of travel agency 00000100.

What is the currency of the travel agency?

5. Perform a test of the function module to read details of a non-existing travel agency, for example *IV_AGENCYNUM* = 00000001.

What is the result?

6. Analyze the interface of function module *D437_AGENCY_UPDATE*.

What input does the function module expect?

7. Perform a test of function module *D437_AGENCY_UPDATE*. Supply the input parameters to change the currency of travel agency 00000100 from *EUR* to *XYZ*.

What is the result?

8. Repeat the test with new currency *USD*.

Do you receive an error message?

9. Repeat the test of function module *D437_AGENCY_READ* with the same travel agency number as before.

Why does the function module still return the old value for *CURRENCY*?



Hint:

In classical ABAP workbench, it is possible to test sequences of function module calls where the main program of the function group remains in the memory.

Task 2: Implement Method *update()*

Implement the method *update()* of the handler class (*lhc_Agency*) with a call of function module *D437_AGENCY_UPDATE*.

For each affected travel agency, map the input to the method to suitable actual parameters of the function module. If the function module raises an exception, mark the affected instance as failed and hand the messages over to the RAP runtime.

1. Edit the implementation of method *update()*. Define suitable actual parameters for all parameters of function module *D437_AGENCY_UPDATE*.
2. Implement a loop over all affected travel agencies (import parameter *entities*) and call function module *D437_AGENCY_UPDATE* for each of them.
3. Transfer the information from the current line of parameter *entities* to your actual parameter for formal parameter *IS_AGENCY*.

Why is it not sufficient to use MOVE-CORRESPONDING or CORRESPONDING #() in its standard form?

Is there a RAP-specific alternative to explicitly assigning the components with differing names? If yes, implement this alternative.

4. Transfer the information from substructure *%control* of the current line of parameter *entities* to your actual parameter for parameter *IS_AGENCYX*.

Why is it not possible to move the data between one structure and the other?

Is there a RAP-specific alternative to explicitly mapping the different technical representation of Boolean values? If yes, implement this alternative.

5. After the function module call, check `sy-subrc` to handle exceptions. If there are exceptions, add the complete key of the current travel agency to the relevant component of response structure `failed`.
6. Implement a loop over your actual parameter for export parameter `et_messages` and, for each message in turn, add a row to the relevant component of the response structure `reported`.



Note:

Use inherited method `new_message()` of your local handler class to create a message object based on the information in parameter `et_messages`.

Task 3: Implement Method `save()`

Implement method `save()` of the saver class (`lsc_Agency`) with a call of function module `D437_AGENCY_SAVE`.

If the function module raises an exception, add an error message to component `%other` of response parameter `reported`.



Hint:

Use inherited method `new_message()` of the local saver class to create a message object based on message 650 of message class `DEVS4D437`.

1. Edit the implementation of method `save()`. Implement a call of function module `D437_AGENCY_UPDATE`.
2. After the function module call, check `sy-subrc` to handle exceptions. In case of exceptions, add a new message object to component `%other` of response parameter `reported`.
3. Activate the implementation class.

Task 4: Implement Locks and Authorization Checks

Set the root entity of your RAP Business Objects for travel agencies as lock master and authorization master, with instances-based authorization checks. Generate the required additional methods of the handler class and implement them with calls of suitable function modules of function group `D437AGENCY`.

1. Edit the behavior definition for your data model view for travel agencies (`Z##_I_Agencies`). Add the additions `lock master` and `authorization master(instance)` to the statement `define behavior`.
2. Activate the behavior definition and generate the required additional methods in the behavior implementation class.
3. Edit the implementation of method `lock()` and implement a loop over all affected travel agencies (parameter `keys`).
4. For each travel agency in turn, call function module `D437_AGENCY_LOCK`. If the function module raises exception `already_locked`, add the key of the travel agency to the relevant component of structured response parameter `failed` and add the messages from parameter `et_messages` to the correct component of response parameter `reported`.



Note:

Exception system *failure* indicates a serious internal problem with the *Enqueue Server*. For this serious situation, we do not handle the exception but risk a short dump.

5. Edit the implementation of method `get_authorizations()` and implement a loop over all affected travel agencies (parameter `keys`).
6. Evaluate structured import parameter `requested_authorizations`. For each component with a value equal to `if_abap_behv=>mk-on`, call function module `D437_AGENCY_CHECK_AUTH` with the key of the travel agency and the related value for `iv_activity`.
7. For each affected travel agency, add a row to response parameter `result` with the key of the travel agency and the information about the operations for which the user is authorized or not authorized.



Note:

Use the components of structured constant `auth` in interface `if_abap_behv`.

8. Activate the behavior implementation and re-test your SAP Fiori elements app.

Task 5: Optional: Implement Create and Delete

Expose basic operations `create` and `delete` to your OData UI Service and implement the related methods of the local handler class.



Note:

When you enable basic operation `create`, don't forget to set key field `AgencyIDas` read-only. The function module that creates new travel agencies implements internal numbering for this field.

1. Edit the behavior definition for your data model view (`Z##_I_Agency`) and set the key field `Agencynum` to `readonly`.

2. Enable the creation and deletion of travel agencies in your SAP Fiori elements app.
3. Implement method `create()` of local handler class `lhc_Agency`. For each line in import parameter `entities`, map the data from this line to a suitable actual parameter for import parameter `is_agency`, call function module `D437_AGENCY_CREATE`, and, in case of errors, mark the travel agency as `failed` and handle the messages.



Note:

Because the new travel agency does not have a proper key value, yet, the RAP framework uses an intermediate key (component `%cid`). You have to use this intermediate key when filling response parameters `failed` and `reported`.

4. In case the creation of the new travel agency was successful, that is, the function module did not raise an exception, fill the relevant component of structured response parameter mapped with the `%cid` value of the current entity and the `agencynum` value returned by the function module.
5. Implement method `delete()` of local handler class `lhc_Agency`. For each line in import parameter `entities`, call function module `D437_AGENCY_DELETE` and, in case of errors, handle the messages.
6. Activate the behavior implementation and re-test your SAP Fiori elements app.



Caution:

Try not to delete travel agencies that you have not created yourself. You might not be alone in the system. However, if you properly implemented method `get_authorizations()` before, the function module should only allow you to delete the travel agencies you created yourself.

Implement an Unmanaged Business Object

In this exercise, you implement the behavior of your unmanaged RAP Business Object for flight travels.

You implement data access for basic operation `update` by calling the necessary function modules from function group `D437AGENCY`.

To improve convenience for the user, you make the RAP runtime request locks and trigger authority checks at an earlier stage. For the related implementations, you again call the relevant function modules from function group `D437AGENCY`.

Optionally, enable and implement the basic operations `create` and `delete`.



Note:

In this exercise, replace `##` with the number that your instructor assigned to you.

Table 22: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	<code>D437F_I_AGENCY</code>
CDS Behavior Definition (Projection)	<code>D437F_C_AGENCY</code>
ABAP Class	<code>BP_D437F_I_AGENCY</code>

Task 1: Analyze Function Group D437AGENCY

Analyze the functionality of function group `D437AGENCY`. Use the function modules of this function group to read and create travel agencies.

1. Open function group `D437AGENCY`.
 - a) In *ABAP Development Tools*, choose *Open ABAP Development Object* or press `Ctrl + Shift + A`.
 - b) Enter `D437AGENCY` and choose *OK*.
2. Analyze the function group in *Project Explorer*.

How many function modules belong to this function group?

Seven function modules.

- a) If the focus in the *Project Explorer* view is not yet on the function group, choose *Link with Editor* from the toolbar within the *Project Explorer* view.

b) Expand the function group to see its sub-objects.

3. Analyze the interface of function module *D437_AGENCY_READ*.

What input does the function module expect?

A travel agency number and a flag to indicate whether the buffer of the function group is to be used.

What information does the function module return?

A structure with travel agency data and an internal table with messages.

a) Double-click the function module name in the *Project Explorer* to open the editor.

4. Perform a test of the function module to read details of travel agency 00000100.

What is the currency of the travel agency?

European Euro (EUR)

a) Place the cursor anywhere on the source code of the function module and press F8.

b) Enter the number of the travel agency, leave the default value for *iv_use_buffer* and choose *Execute (F8)*.

c) Analyze the result in returning parameter *ES_AGENCY*.

5. Perform a test of the function module to read details of a non-existing travel agency, for example *IV_AGENCYNUM* = 00000001.

What is the result?

Parameter *ES_AGENCY* is empty and parameter *ET_MESSAGES* contains one line with the message 600 of message class *DEVS4D437*.

a) Perform this step as before.

6. Analyze the interface of function module *D437_AGENCY_UPDATE*.

What input does the function module expect?

A structure with travel agency data and structure with Boolean flags, one flag for each component of the data structure.

a) Double-click the function module name in the *Project Explorer* to open the editor.

7. Perform a test of function module *D437_AGENCY_UPDATE*. Supply the input parameters to change the currency of travel agency 00000100 from *EUR* to *XYZ*.
 - a) Place the cursor anywhere on the source code of the function module and press F8.
 - b) Open the edit dialog for input parameter *IS_AGENCY* and enter travel agency number and the new value for component *CURRENCY*.
 - c) Return to the list of parameters, open the edit dialog for input parameter *IS_AGENCYX* and enter value *x* for component *CURRENCY*.
 - d) Return to the list of parameters and choose *Execute (F8)*.

What is the result?

The function module raises an exception and export parameter *ET_MESSAGES* contains a line with message number 625 of message class *DEVS4D437*.

8. Repeat the test with new currency *USD*.

Do you receive an error message?

No, the value 'USD' is accepted.

- a) Perform this step as before.

9. Repeat the test of function module *D437_AGENCY_READ* with the same travel agency number as before.
 - a) Perform this step as before.

Why does the function module still return the old value for *CURRENCY*?

The new value is not yet stored on the database. It was only written to the buffer. Unfortunately, the main program of the function group is loaded individually for each function module test. That way, the data in the buffer is lost between the test of *D437_AGENCY_UPDATE* and *D437_AGENCY_READ*.



Hint:

In classical ABAP workbench, it is possible to test sequences of function module calls where the main program of the function group remains in the memory.

Task 2: Implement Method update()

Implement the method `update()` of the handler class (`lhc_Agency`) with a call of function module *D437_AGENCY_UPDATE*.

For each affected travel agency, map the input to the method to suitable actual parameters of the function module. If the function module raises an exception, mark the affected instance as failed and hand the messages over to the RAP runtime.

1. Edit the implementation of method `update()`. Define suitable actual parameters for all parameters of function module `D437_AGENCY_UPDATE`.
 - a) See the source code extract from the model solution.
2. Implement a loop over all affected travel agencies (import parameter `entities`) and call function module `D437_AGENCY_UPDATE` for each of them.
 - a) See the source code extract from the model solution.
3. Transfer the information from the current line of parameter `entities` to your actual parameter for formal parameter `IS_AGENCY`.

Why is it not sufficient to use `MOVE-CORRESPONDING` or `CORRESPONDING #()` in its standard form?

Some component names in the input structure of the function module differ from the field names in the RAP Business Object.

Is there a RAP-specific alternative to explicitly assigning the components with differing names? If yes, implement this alternative.

Yes, defining a mapping for structure type `d437_s_agency` in the behavior definition and using expression `CORRESPONDING #()` with the addition `MAPPING FROM ENTITY`.

- a) Add a mapping for `d437_s_agency` statement to your behavior definition for travel agencies.
 - b) Transfer the data with expression `CORRESPONDING #(<Is_agency> MAPPING FROM ENTITY)`.
 - c) See the source code extract from the model solution.
4. Transfer the information from substructure `%control` of the current line of parameter `entities` to your actual parameter for parameter `IS_AGENCYX`.

Why is it not possible to move the data between one structure and the other?

Because `%control` uses flags of type `raw(1)` with values '00' and '01', whereas `IS_AGENCY` uses flags of type `char(1)` with values '' and 'X'.

Is there a RAP-specific alternative to explicitly mapping the different technical representation of Boolean values? If yes, implement this alternative.

Yes, extending the mapping in the behavior definition with addition `control d437_s_agencyx` and using expression `CORRESPONDING #()` with addition `MAPPING FROM ENTITY USING CONTROL`.

- a) Add control d437_s_agencyx to your mapping statement in the behavior definition for travel agencies.
 - b) Transfer the data with expression CORRESPONDING #(... MAPPING FROM ENTITY USING CONTROL).
 - c) See the source code extract from the model solution.
5. After the function module call, check sy-subrc to handle exceptions. If there are exceptions, add the complete key of the current travel agency to the relevant component of response structure *failed*.
- a) See the source code extract from the model solution.
6. Implement a loop over your actual parameter for export parameter et_messages and, for each message in turn, add a row to the relevant component of the response structure *reported*.



Note:

Use inherited method new_message() of your local handler class to create a message object based on the information in parameter et_messages.

- a) See the source code extract from the model solution.
- b) Compare your source code to the following extract from the model solution.

Behavior Definition **D437F_I_AGENCY**

```
unmanaged implementation in class bp_d437f_i_agency unique;

define behavior for D437f_I_Agency alias Agency
{

    ...

    mapping for d437_s_agency
        control d437_s_agencyx corresponding
    {
        AgencyID      = agencynum;
        ZIPCode       = postcode;
        Language      = langu;
        CurrencyCode = currency;
    }
}
```

Local class **LHC_AGENCY** in **BP_D437F_I_AGENCY**

```
METHOD update.

    " actual parameters for function module call
DATA ls_agency    TYPE d437_s_agency.
DATA ls_agencyx   TYPE d437_s_agencyx.
DATA lt_messages  TYPE symsg_tab.
DATA lv_agencynum TYPE s_agency.

* Loop over affected travel agencies (entities)
*****
```

```

LOOP AT entities ASSIGNING FIELD-SYMBOL(<ls_agency>).

* Prepare call of legacy function module
***** ****
* map data structure from RAP field names to legacy field names

" a) manually

*     ls_agency = CORRESPONDING #( <ls_agency> ).
*     ls_agency-agencynum = <ls_agency>-agencyid.
*     ls_agency-postcode = <ls_agency>-zipcode.
*     ls_agency-langu = <ls_agency>-language.
*     ls_agency-currency = <ls_agency>-currencycode.

" b) with RAP-specific variant of CORRESPONDING #( )

ls_agency = CORRESPONDING #( <ls_agency>
                           MAPPING FROM ENTITY ).

* map control structure %control to legacy control structure

" a) manually (have fun!)

*     IF <ls_agency>-%control-agencyid = if_abap_behv=>mk-on.
*         ls_agencyx-agencynum = abap_true.
*     ENDIF.
*     IF <ls_agency>-%control-ZIPCode = if_abap_behv=>mk-on.
*         ls_agencyx-postcode = abap_true.
*     ENDIF.
*         IF <ls_agency>-%control-Language = if_abap_behv=>mk-on.
*             ls_agencyx-langu = abap_true.
*         ENDIF.

*     " ...

" b) With RAP-specific variant of CORRESPONDING #( )

ls_agencyx = CORRESPONDING #( <ls_agency>
                           MAPPING FROM ENTITY
                           USING CONTROL ).

* Call legacy function module for update in buffer
***** ****

CALL FUNCTION 'D437_AGENCY_UPDATE'
  EXPORTING
    is_agency      = ls_agency
    is_agencyx     = ls_agencyx
  IMPORTING
    et_messages    = lt_messages
  EXCEPTIONS
    wrong_input    = 1
    not_found      = 2
    already_locked = 3
    no_auth        = 4.
  IF sy-subrc <> 0.
    " in case of error
    " mark as failed
    APPEND VALUE #( %tky = <ls_agency>-%tky )
      TO failed-agency.

```

```

" and map messages
LOOP AT lt_messages ASSIGNING FIELD-SYMBOL(<ls_message>)
      WHERE msgty = 'E' OR msgty = 'A'.
APPEND VALUE #( %tky = <ls_agency>-%tky
      %msg = me->new_message(
          id      = <ls_message>-msgid
          number  = <ls_message>-msgno
          severity = me->ms-error
          v1      = <ls_message>-msgv1
          v2      = <ls_message>-msgv2
          v3      = <ls_message>-msgv3
          v4      = <ls_message>-msgv4
      )
  )
TO reported-agency.

ENDLOOP.
ENDIF.

ENDLOOP.

ENDMETHOD.

```

Task 3: Implement Method save()

Implement method `save()` of the saver class (`lsc_Agency`) with a call of function module `D437_AGENCY_SAVE`.

If the function module raises an exception, add an error message to component `%other` of response parameter `reported`.



Hint:

Use inherited method `new_message()` of the local saver class to create a message object based on message 650 of message class `DEVS4D437`.

1. Edit the implementation of method `save()`. Implement a call of function module `D437_AGENCY_UPDATE`.
 - a) See the source code extract from the model solution.
2. After the function module call, check `sy-subrc` to handle exceptions. In case of exceptions, add a new message object to component `%other` of response parameter `reported`.
 - a) See the source code extract from the model solution.
3. Activate the implementation class.
 - a) Press `Ctrl + F3`.
 - b) Compare your source code to the following extract from the model solution.

Local class `LSC_D437F_I_AGENCY` in `BP_D437F_I_AGENCY`

```
METHOD save.
```

```

CALL FUNCTION 'D437_AGENCY_SAVE'
  EXCEPTIONS
    error_in_create = 1
    error_in_update = 2

```

```

        error_in_delete = 3
        OTHERS          = 4.
IF sy-subrc <> 0.

    APPEND  me->new_message(
            id          = 'DEVS4D437'
            number     = '650'
            severity   = me->ms-error
            )
    TO reported-%other.

ENDIF.
ENDMETHOD.
```

Task 4: Implement Locks and Authorization Checks

Set the root entity of your RAP Business Objects for travel agencies as lock master and authorization master, with instances-based authorization checks. Generate the required additional methods of the handler class and implement them with calls of suitable function modules of function group *D437AGENCY*.

1. Edit the behavior definition for your data model view for travel agencies (*Z##_I_Agencies*).
Add the additions `lock master` and `authorization master(instance)` to the statement `define behavior`.
 - a) See the source code extract from the model solution.
2. Activate the behavior definition and generate the required additional methods in the behavior implementation class.
 - a) Press `Ctrl + F3`.
 - b) Place the cursor on the keyword `lock` and invoke the quick fix to create method `lock()`.
 - c) Place the cursor on keyword `master` after keyword `authorization` and invoke the quick fix to create method `get_authorizations()`.
3. Edit the implementation of method `lock()` and implement a loop over all affected travel agencies (parameter `keys`).
 - a) See source code extract from the model solution.
4. For each travel agency in turn, call function module *D437_AGENCY_LOCK*. If the function module raises exception `already_locked`, add the key of the travel agency to the relevant component of structured response parameter `failed` and add the messages from parameter `et_messages` to the correct component of response parameter `reported`.



Note:

Exception system failure indicates a serious internal problem with the *Enqueue Server*. For this serious situation, we do not handle the exception but risk a short dump.

- a) See the source code extract from the model solution.
5. Edit the implementation of method `get_authorizations()` and implement a loop over all affected travel agencies (parameter `keys`).
 - a) See the source code extract from the model solution.

6. Evaluate structured import parameter `requested_authorizations`. For each component with a value equal to `if_abap_behv=>mk-on`, call function module `D437_AGENCY_CHECK_AUTH` with the key of the travel agency and the related value for `iv_activity`.
 - a) See the source code extract from the model solution.
7. For each affected travel agency, add a row to response parameter `result` with the key of the travel agency and the information about the operations for which the user is authorized or not authorized.

**Note:**

Use the components of structured constant `auth` in interface `if_abap_behv`.

- a) See the source code extract from the model solution.
8. Activate the behavior implementation and re-test your SAP Fiori elements app.
 - a) Perform this step as before.
 - b) Compare your source code to the following extract from the model solution.

Behavior Definition D437F_I_AGENCY

```
unmanaged implementation in class zbp_00_i_agency unique;

define behavior for Z00_I_Agency alias Agency
lock master
authorization master (instance )
{
  ...
}
```

Local class LHC_AGENCY in BP_D437F_I_AGENCY**Method lock()**

```
METHOD lock.

  " actual parameters for function module call
  DATA lt_messages TYPE symsg_tab.

* Loop over keys of affected travel agencies
*****LOOP AT keys ASSIGNING FIELD-SYMBOL(<ls_key>).

* call legacy function module to set lock
*****CALL FUNCTION 'D437_AGENCY_LOCK'
  EXPORTING
    iv_agencynum    = <ls_key>-agencyid
  IMPORTING
    et_messages     = lt_messages
  EXCEPTIONS
    already_locked = 1
*           system_failure = 2    "ignore -> short dump
```

```

        IF sy-subrc <> 0.

        " in case of error
        " mark as failed
        APPEND VALUE #( agencyid = <ls_key>-agencyid )
                      TO failed-agency.

        " and map messages
        LOOP AT lt_messages ASSIGNING FIELD-SYMBOL(<ls_message>)
              WHERE msgty = 'E' OR msgty = 'A'.
        APPEND VALUE #( agencyid = <ls_key>-agencyid
                      %msg = me->new_message(
                        id      = <ls_message>-msgid
                        number  = <ls_message>-msgno
                        severity = me->ms-error
                        v1      = <ls_message>-msgv1
                        v2      = <ls_message>-msgv2
                        v3      = <ls_message>-msgv3
                        v4      = <ls_message>-msgv4
                      )
        )
        TO reported-agency.
      ENDLOOP.
    ENDIF.

  ENDLOOP.

ENDMETHOD.
```

Method *get_authorizations()*

```

METHOD get_authorizations.

" helper variable to fill response parameter result
DATA ls_result LIKE LINE OF result.
*****  

* Loop over keys of affected travel agencies
*****  

LOOP AT keys ASSIGNING FIELD-SYMBOL(<ls_key>).

  CLEAR ls_result.

  ls_result-%tky = <ls_key>-%tky.

  IF requested_authorizations-%update = if_abap_behv=>mk-on.

* call legacy function module to check authorization for update
*****  

  CALL FUNCTION 'D437_AGENCY_CHECK_AUTH'
    EXPORTING
      iv_agencynum = <ls_key>-agencyid
      iv_activity  = '02'
    EXCEPTIONS
      no_authority = 1.
    IF sy-subrc <> 0.
      ls_result-%update = if_abap_behv=>auth-unauthorized.
    ENDIF.
  ENDIF.
```

```

IF requested_authorizations-%delete = if_abap_behv=>mk-on.
  CALL FUNCTION 'D437_AGENCY_CHECK_AUTH'
    EXPORTING
      iv_agencynum = <ls_key>-agencyid
      iv_activity = '06'
    EXCEPTIONS
      no_authority = 1.
  IF sy-subrc <> 0.
    ls_result-%delete = if_abap_behv=>auth-unauthorized.
  ENDIF.
ENDIF.

* Add outcome to response parameter result
*****APPEND ls_result
TO result.

ENDLOOP.

ENDMETHOD.

```

Task 5: Optional: Implement Create and Delete

Expose basic operations `create` and `delete` to your OData UI Service and implement the related methods of the local handler class.



Note:

When you enable basic operation `create`, don't forget to set key field `AgencyID` as read-only. The function module that creates new travel agencies implements internal numbering for this field.

1. Edit the behavior definition for your data model view (`Z##_I_Agency`) and set the key field `Agencynum` to `readonly`.
 - a) Edit the behavior definition for your data model view (`Z##_I_Agency`) and add statement `field (readonly) AgencyID;`
 - b) See the source code extract from the model solution.
2. Enable the creation and deletion of travel agencies in your SAP Fiori elements app.
 - a) Edit the behavior definition for your projection view (`Z##_C_Agency`). Uncomment or add statements `use create;` and `use delete;`
 - b) See the source code extract from the model solution.
3. Implement method `create()` of local handler class `lhc_Agency`. For each line in import parameter `entities`, map the data from this line to a suitable actual parameter for import parameter `is_agency`, call function module `D437_AGENCY_CREATE`, and, in case of errors, mark the travel agency as `failed` and handle the messages.



Note:

Because the new travel agency does not have a proper key value, yet, the RAP framework uses an intermediate key (component `%cid`). You have to use this intermediate key when filling response parameters `failed` and `reported`.

- a) See the source code extract from the model solution.
4. In case the creation of the new travel agency was successful, that is, the function module did not raise an exception, fill the relevant component of structured response parameter mapped with the %cid value of the current entity and the agencynum value returned by the function module.
- a) See the source code extract from the model solution.
5. Implement method `delete()` of local handler class `lhc_Agency`. For each line in import parameter `entities`, call function module `D437_AGENCY_DELETE` and, in case of errors, handle the messages.
- a) See the source code extract from the model solution.
6. Activate the behavior implementation and re-test your SAP Fiori elements app.



Caution:

Try not to delete travel agencies that you have not created yourself. You might not be alone in the system. However, if you properly implemented method `get_authorizations()` before, the function module should only allow you to delete the travel agencies you created yourself.

- a) Perform this step as before.
- b) Compare your source code to the following extract from the model solution.

Behavior Definition D437F_I_AGENCY

```
unmanaged implementation in class bp_d437f_i_agency unique;
define behavior for D437f_I_Agency alias Agency
{
    field ( readonly ) AgencyID;
    ...
}
```

Behavior Definition D437F_C_AGENCY

```
projection;
define behavior for D437f_C_Agency alias Agency
{
    use create;
    use update;
    use delete;
}
```

Local class LHC_AGENCY in BP_D437F_I_AGENCY

Method create()

```
METHOD create.
    " actual parameters for function module call
```

```

DATA ls_agency TYPE d437_s_agency.
DATA lt_messages TYPE symsg_tab.
DATA lv_agencynum TYPE s_agency.

* Loop over affected travel agencies (entities)
***** ****
LOOP AT entities ASSIGNING FIELD-SYMBOL(<ls_agency>).

* Prepare call of legacy function module
***** ****

    ls_agency = CORRESPONDING #( <ls_agency>
                                MAPPING FROM ENTITY
                                ) .

* Call legacy function module for create in buffer
***** ****

    CALL FUNCTION 'D437_AGENCY_CREATE'
        EXPORTING
            is_agency          = ls_agency
        IMPORTING
            et_messages         = lt_messages
            ev_agencynum       = lv_agencynum
        EXCEPTIONS
            wrong_input         = 1
            no_external_number = 2
            error_in_number_get= 3
            already_locked     = 4.
    IF sy-subrc <> 0.
        " in case of error
        " mark as failed
        APPEND VALUE #( %cid = <ls_agency>-%cid )
                      TO failed-agency.

        " and map messages
        LOOP AT lt_messages ASSIGNING FIELD-SYMBOL(<ls_message>)
              WHERE msgty = 'E' OR msgty = 'A'.

            APPEND VALUE #( %cid = <ls_agency>-%cid
                            %msg = me->new_message(
                                id      = <ls_message>-msgid
                                number  = <ls_message>-msgno
                                severity = me->ms-error
                                v1      = <ls_message>-msgv1
                                v2      = <ls_message>-msgv2
                                v3      = <ls_message>-msgv3
                                v4      = <ls_message>-msgv4
                            )
            )
            TO reported-agency.

    ENDLOOP.

    ELSE.

        " map temporary key (%cid) to permanent key (agencyid)
        APPEND VALUE #( %cid = <ls_agency>-%cid
                        agencyid = lv_agencynum
                    )
    )

```

```

        TO mapped-agency.

      ENDIF.

    ENDLOOP.

ENDMETHOD.
```

Method *delete()*

```

METHOD delete.

" actual parameters for function module call
DATA lt_messages TYPE symsg_tab.

* Loop over affected travel agencies (entities)
***** ****
LOOP AT entities ASSIGNING FIELD-SYMBOL(<ls_agency>).

* Call legacy function module to delete in buffer
***** ****
CALL FUNCTION 'D437_AGENCY_DELETE'
  EXPORTING
    iv_agencynum = <ls_agency>-agencyid
  IMPORTING
    et_messages = lt_messages
  EXCEPTIONS
    wrong_input = 1
    not_found = 2
    already_locked = 3
    no_auth = 4.
  IF sy-subrc <> 0.
    " in case of error
    " mark as failed
    APPEND VALUE #( %tky = <ls_agency>-%tky )
      TO failed-agency.

    " and map messages
    LOOP AT lt_messages ASSIGNING FIELD-SYMBOL(<ls_message>)
      WHERE msgty = 'E' OR msgty = 'A'.
      APPEND VALUE #( %tky = <ls_agency>-%tky
        %msg = me->new_message(
          id      = <ls_message>-msgid
          number  = <ls_message>-msgno
          severity = me->ms-error
          v1      = <ls_message>-msgv1
          v2      = <ls_message>-msgv2
          v3      = <ls_message>-msgv3
          v4      = <ls_message>-msgv4
        )
      )
    TO reported-agency.

  ENDLOOP.

ENDIF.

ENDLOOP.
ENDMETHOD.
```



LESSON SUMMARY

You should now be able to:

- Implement data access of an unmanaged Business Object