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

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

Course Overview

TARGET AUDIENCE

This course is intended for the following audiences:

- Development Consultant
- Developer

UNIT 1 The ABAP RESTful Programming Model (RAP)

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

Unit 1
Lesson 1

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)

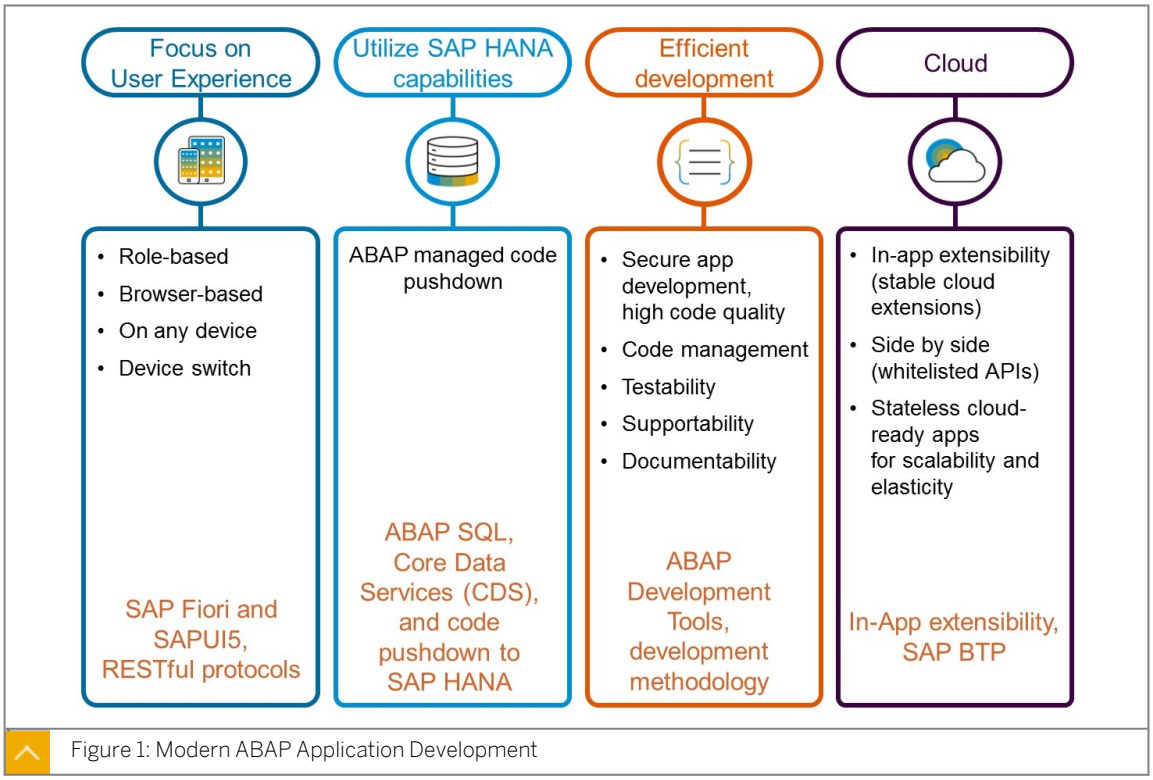

Figure 1: Modern ABAP Application Development

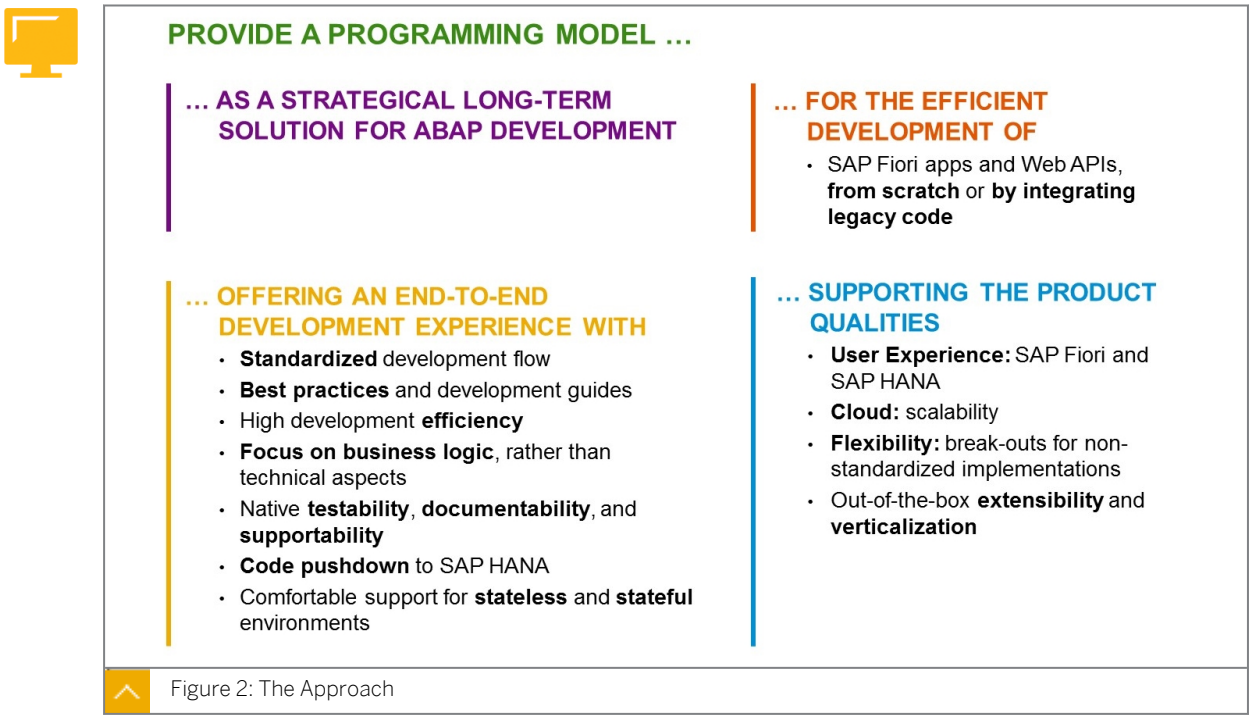

Figure 2: The Approach

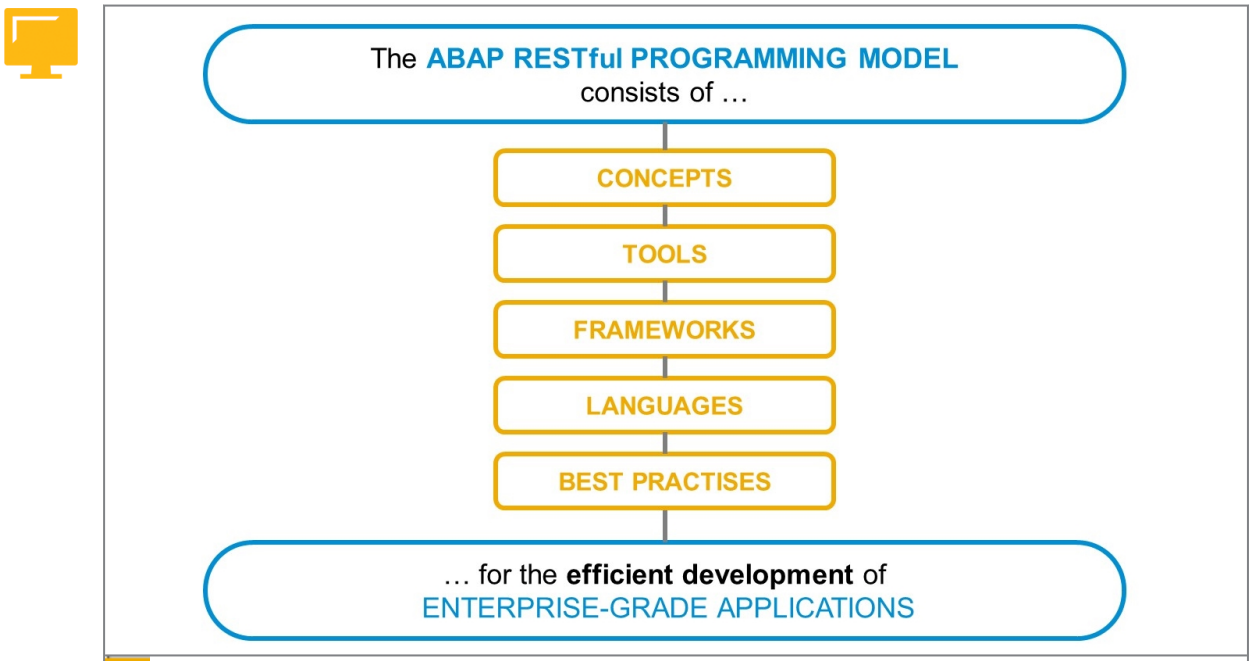

Figure 3: The ABAP RESTful Application Programming Model (RAP)

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.

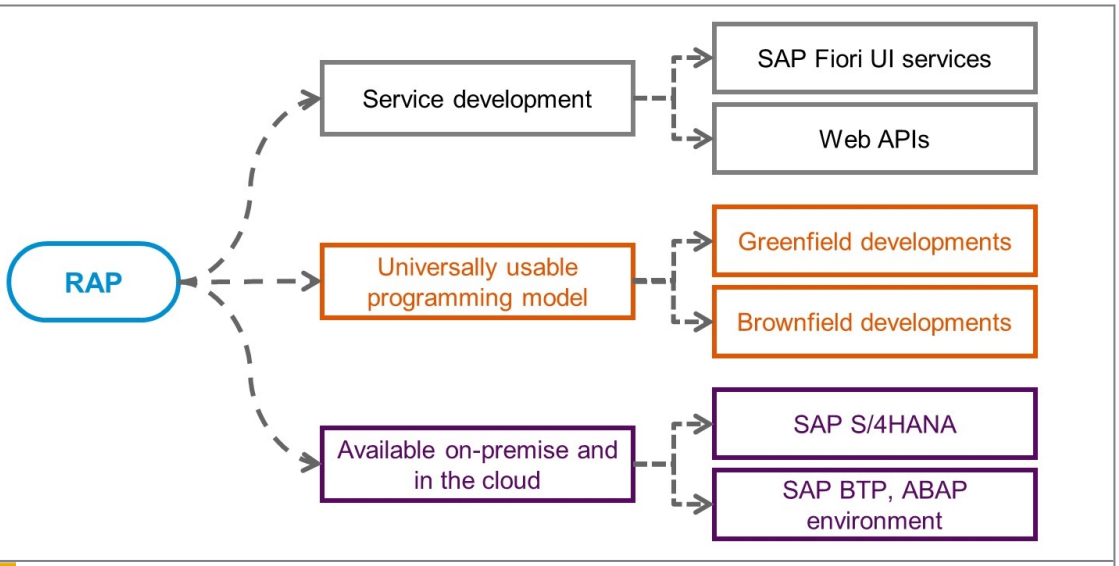

Figure 4: What and Where to Develop with RAP

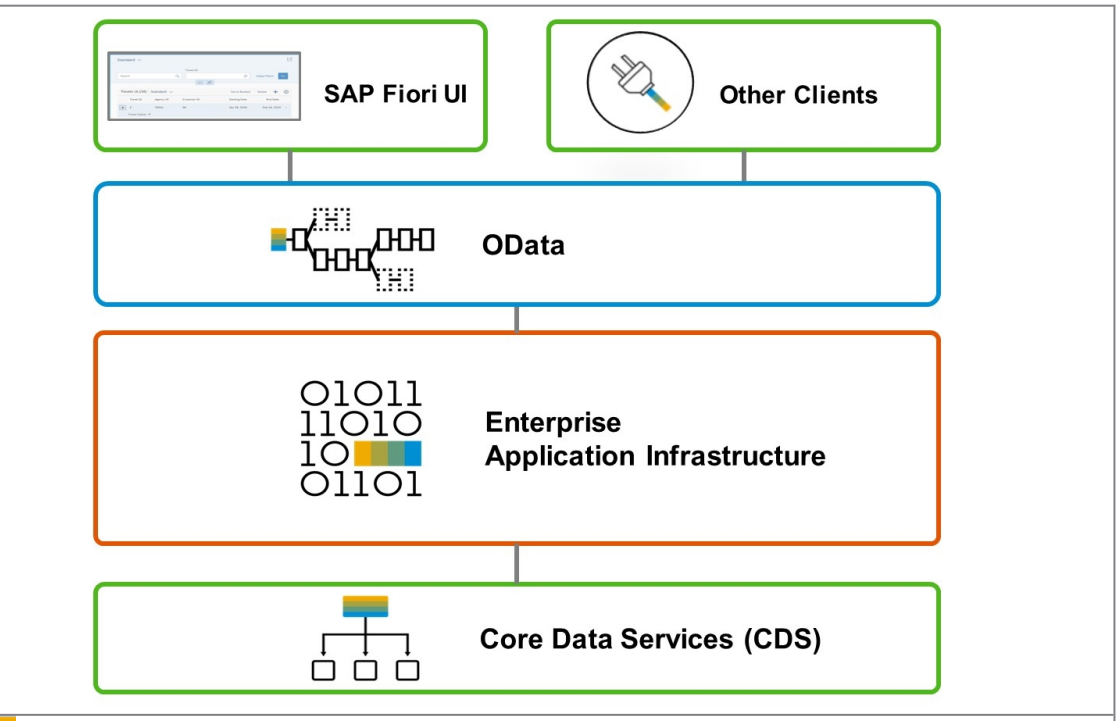

Figure 5: Overview of RAP

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.

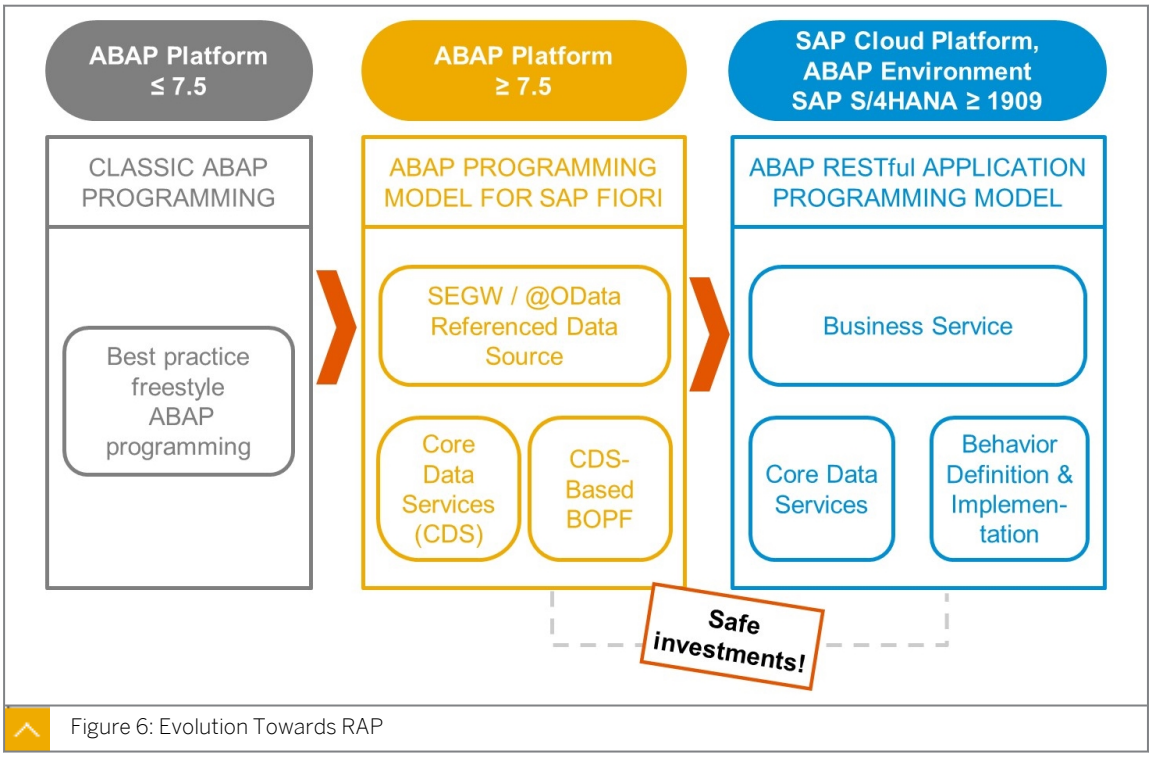

Figure 6: Evolution Towards RAP

ABAP Development Tools

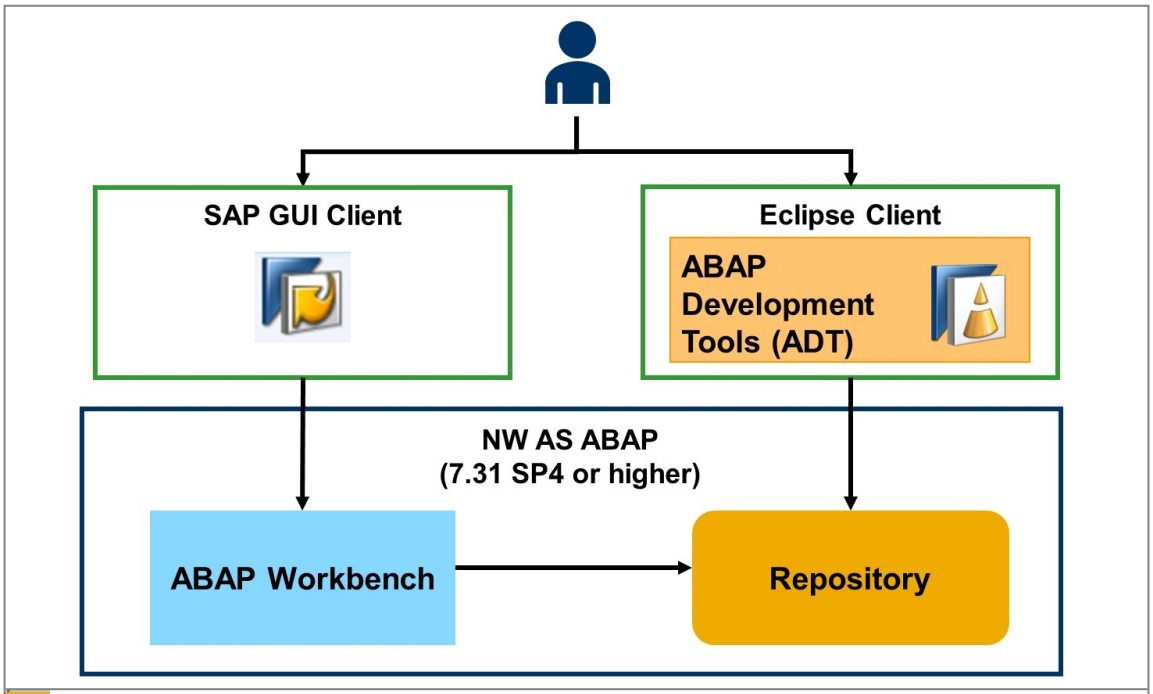

Figure 7: ABAP Development Tools

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

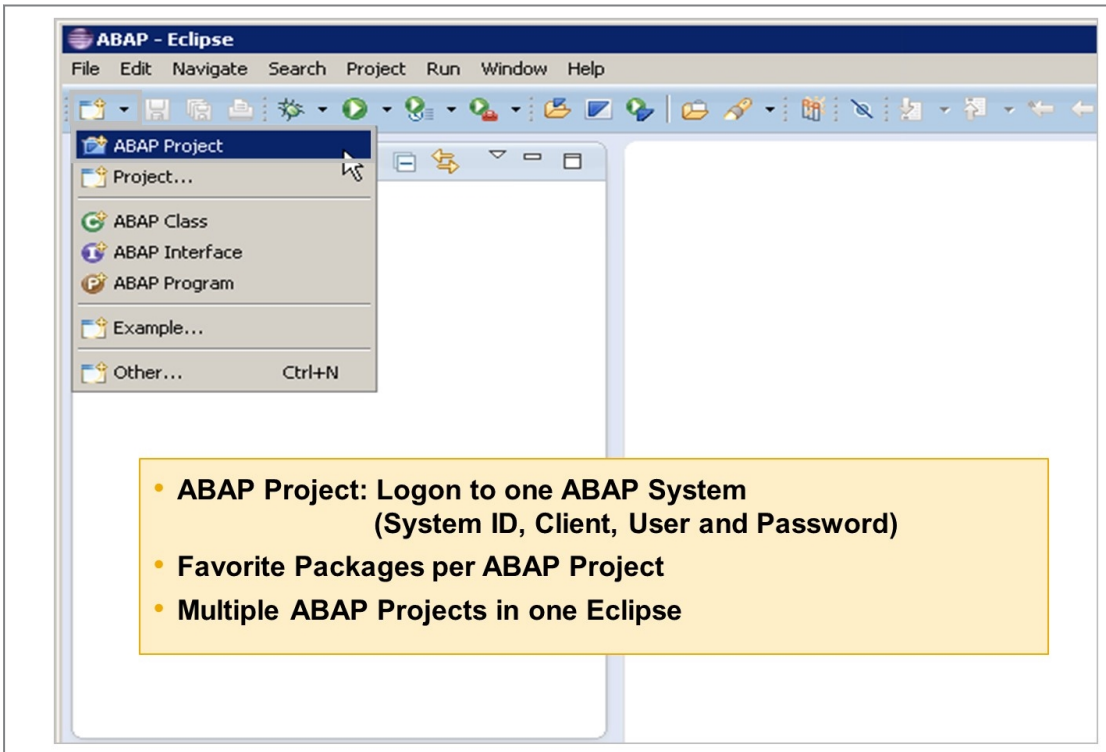

Figure 8: Creating an ABAP Project in ABAP Development Tools

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.

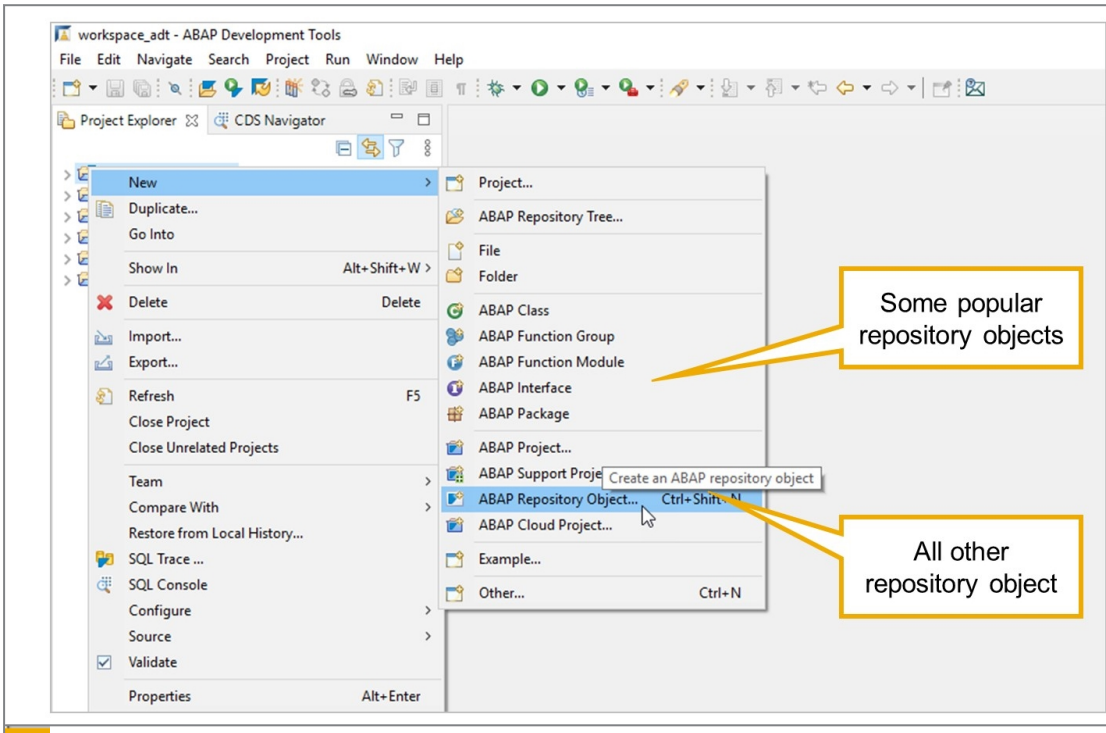

Figure 9: Creating a New Repository Object

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.

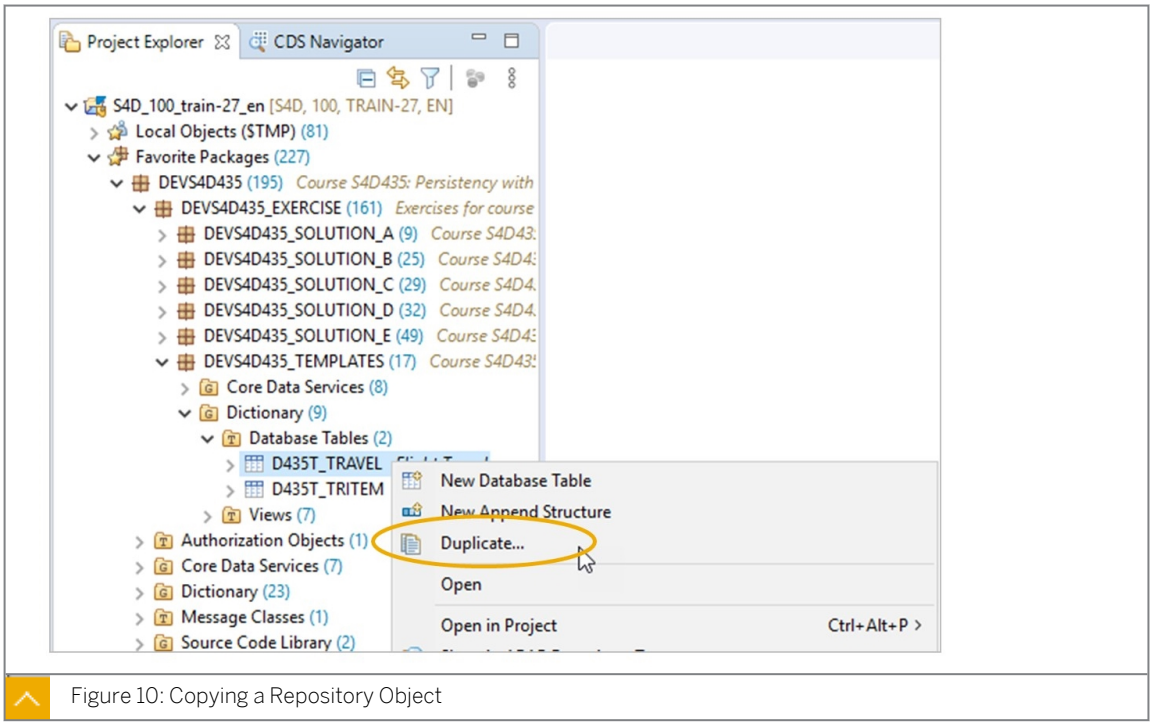

Figure 10: Copying a Repository Object

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.

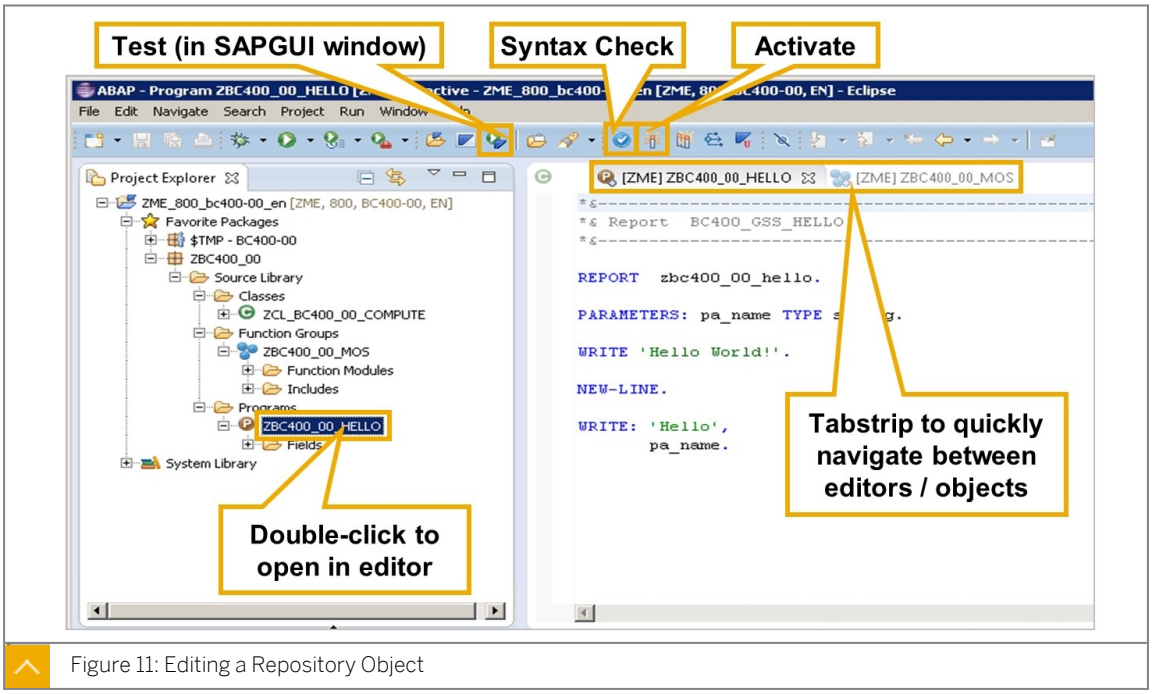

Figure 11: Editing a Repository Object

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

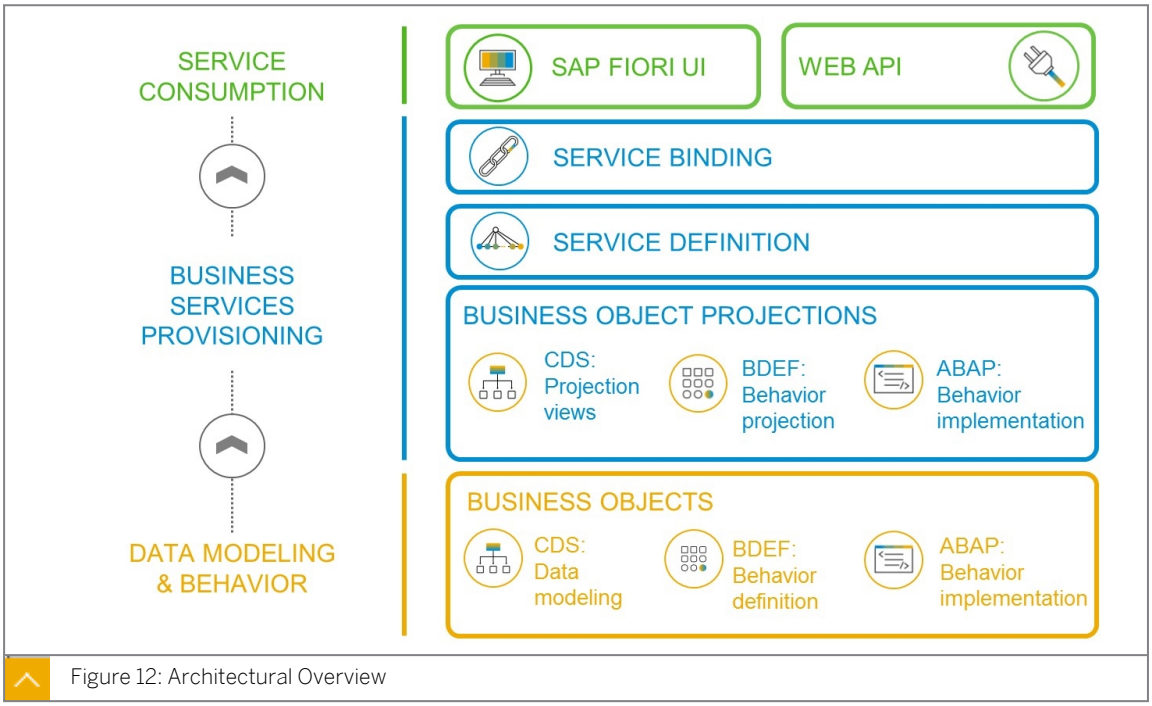

Figure 12: Architectural Overview

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.

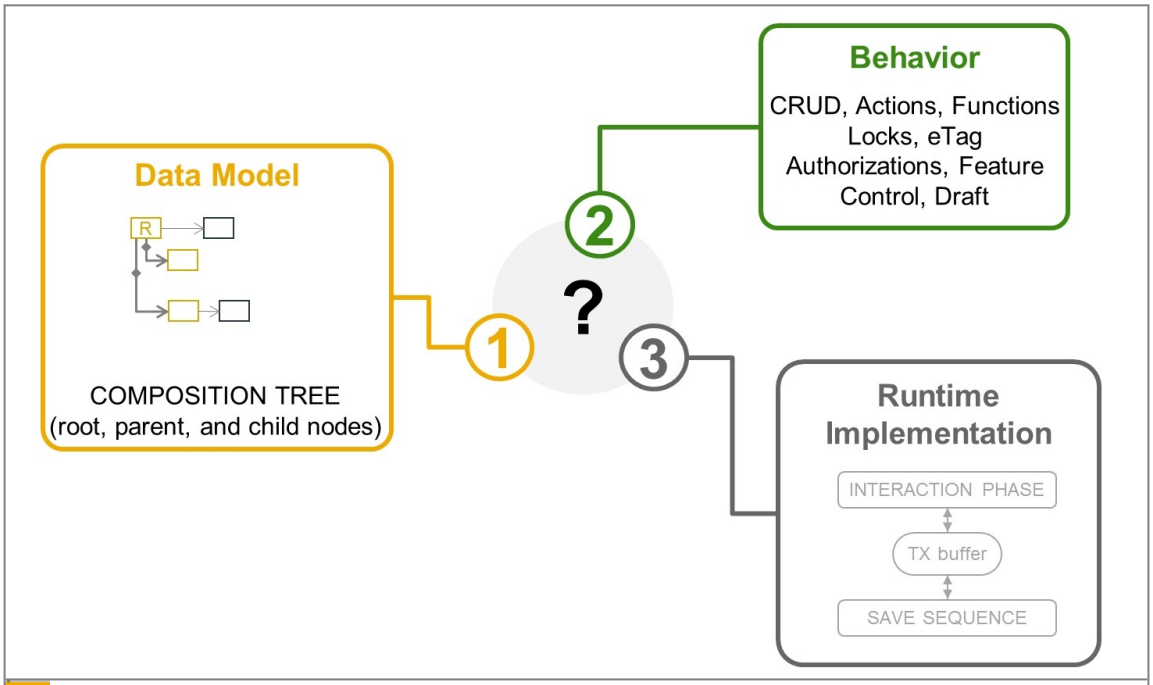

Figure 13: What is a Business Object?

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.

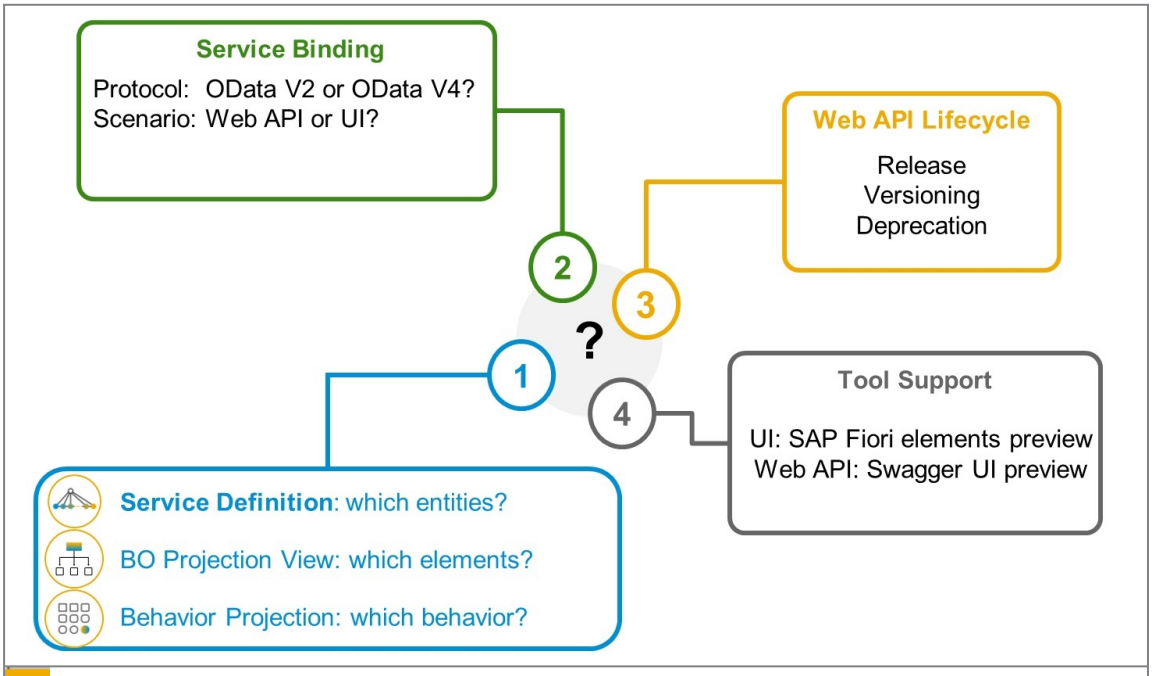

Figure 14: What is a Business Service?

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.

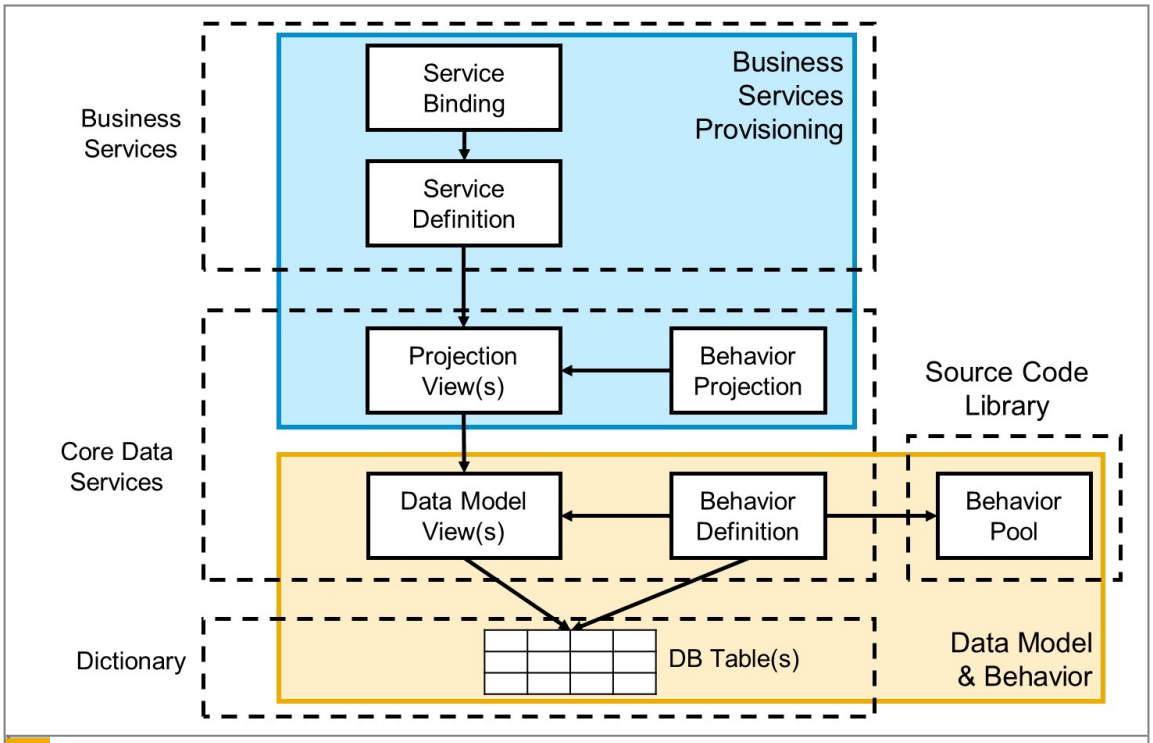

Figure 15: Development Flow

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

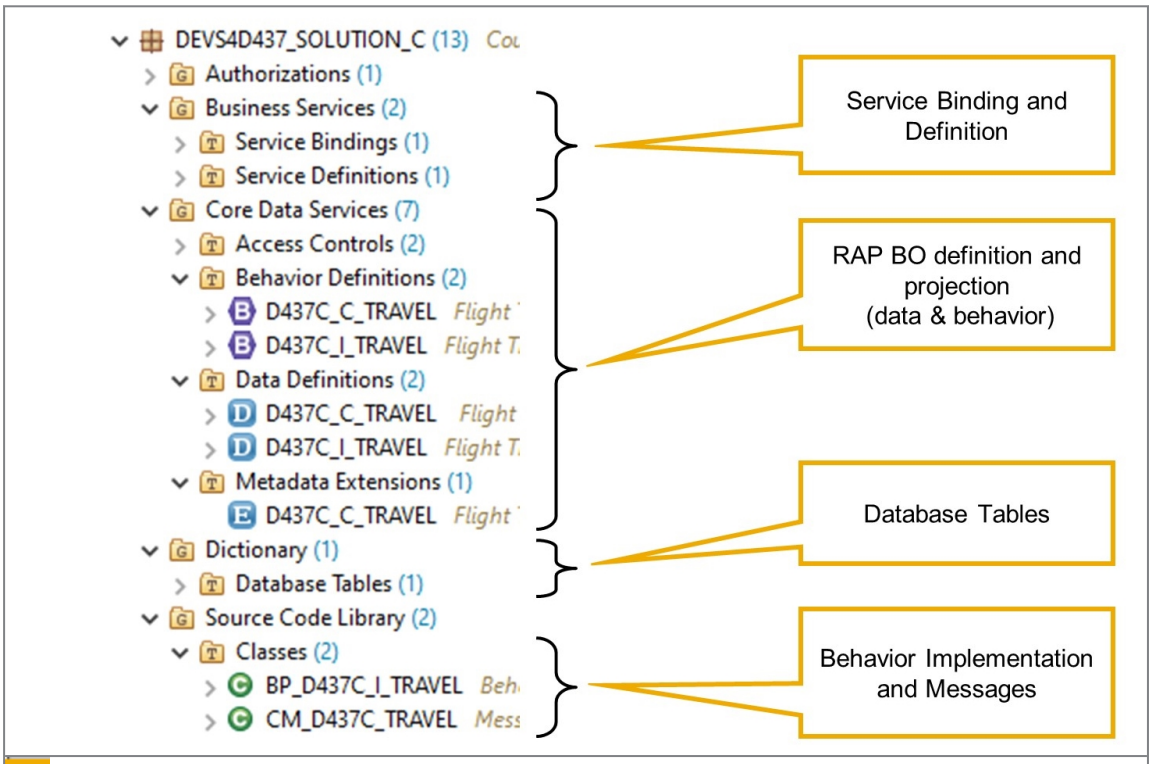

Figure 16: Repository Objects for RAP

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

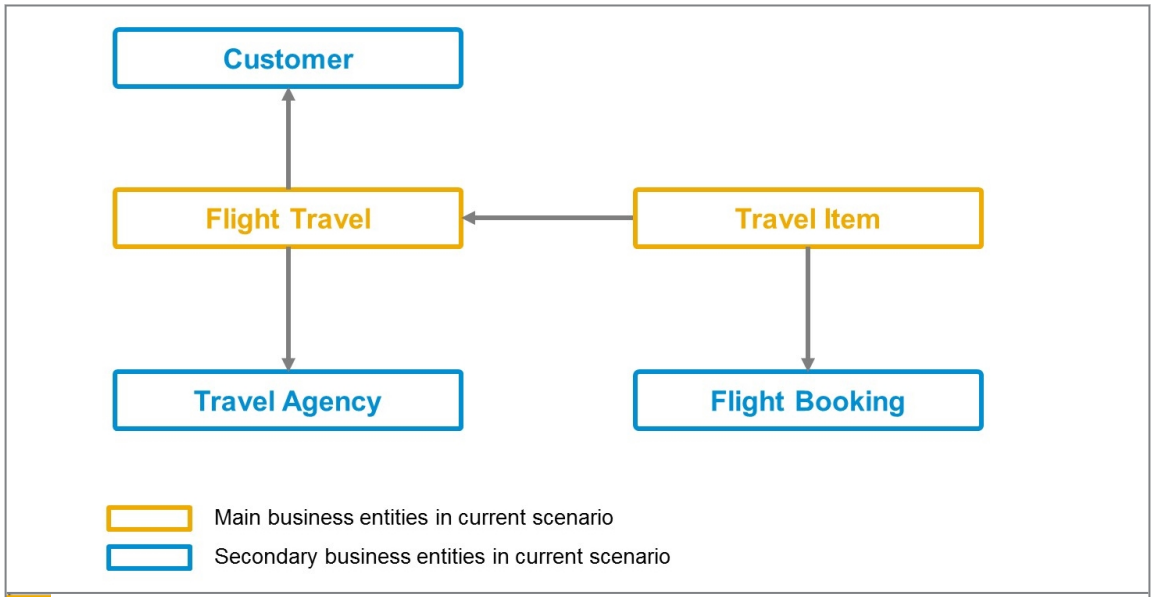

Figure 17: The Business Scenario

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.

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.

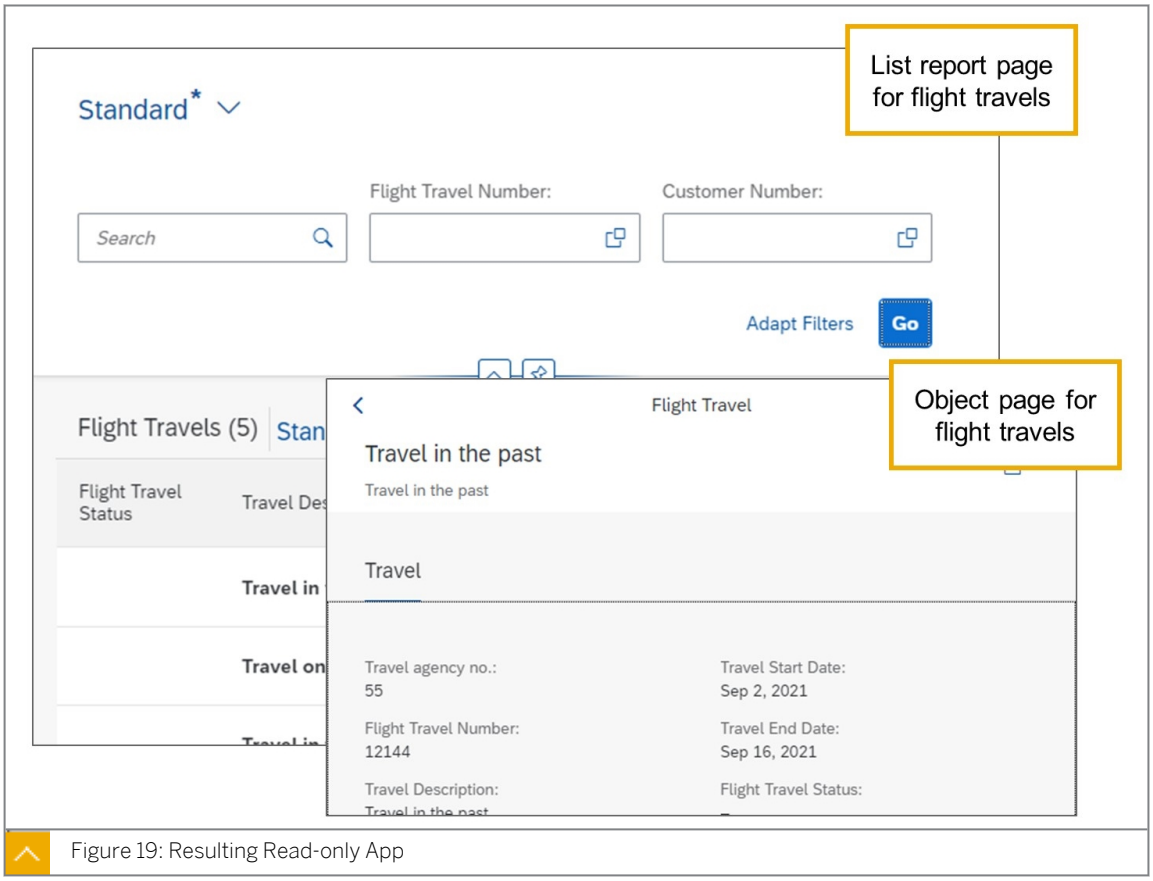

Figure 19: Resulting Read-only App

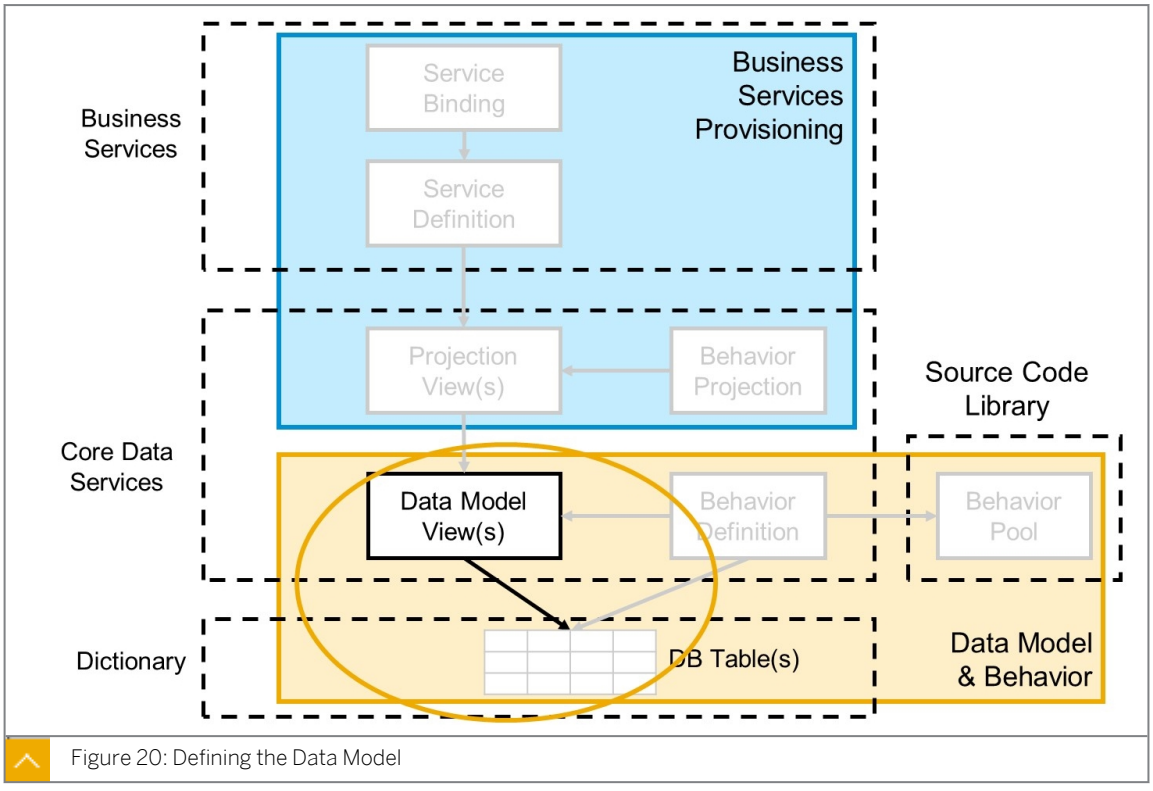

Figure 20: Defining the Data Model

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

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

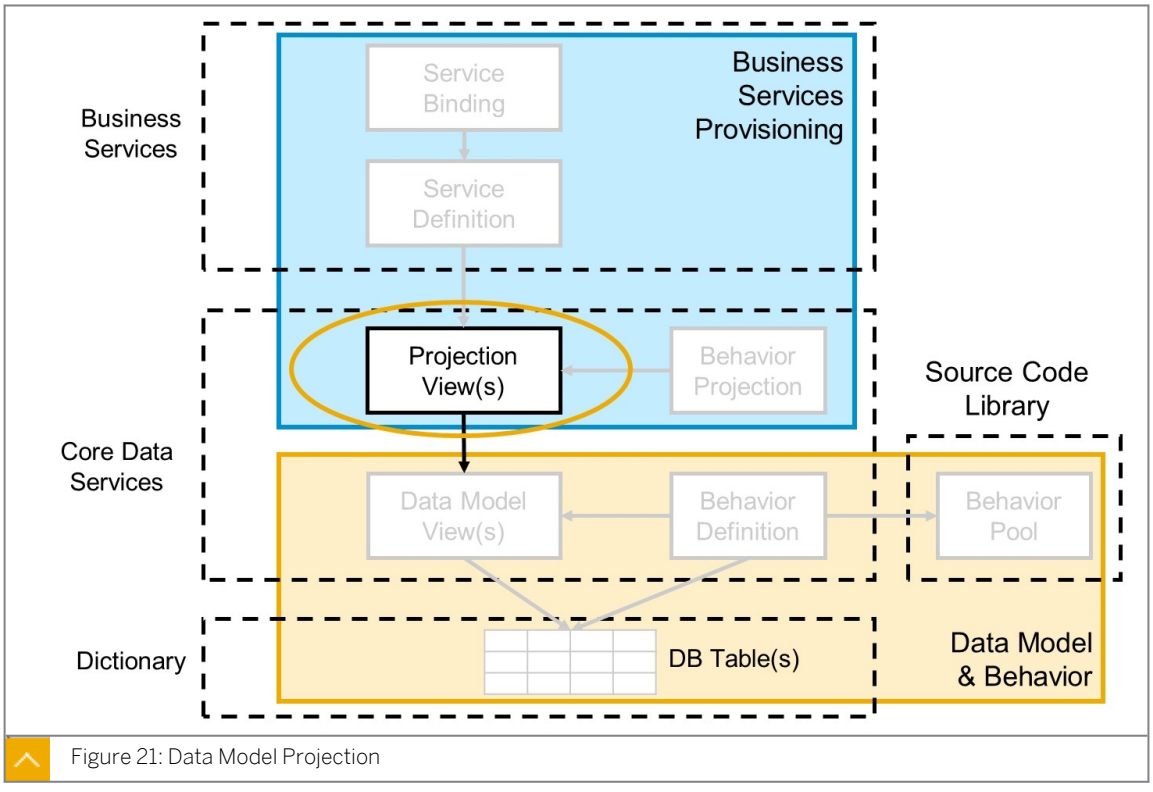

Figure 21: Data Model Projection

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.

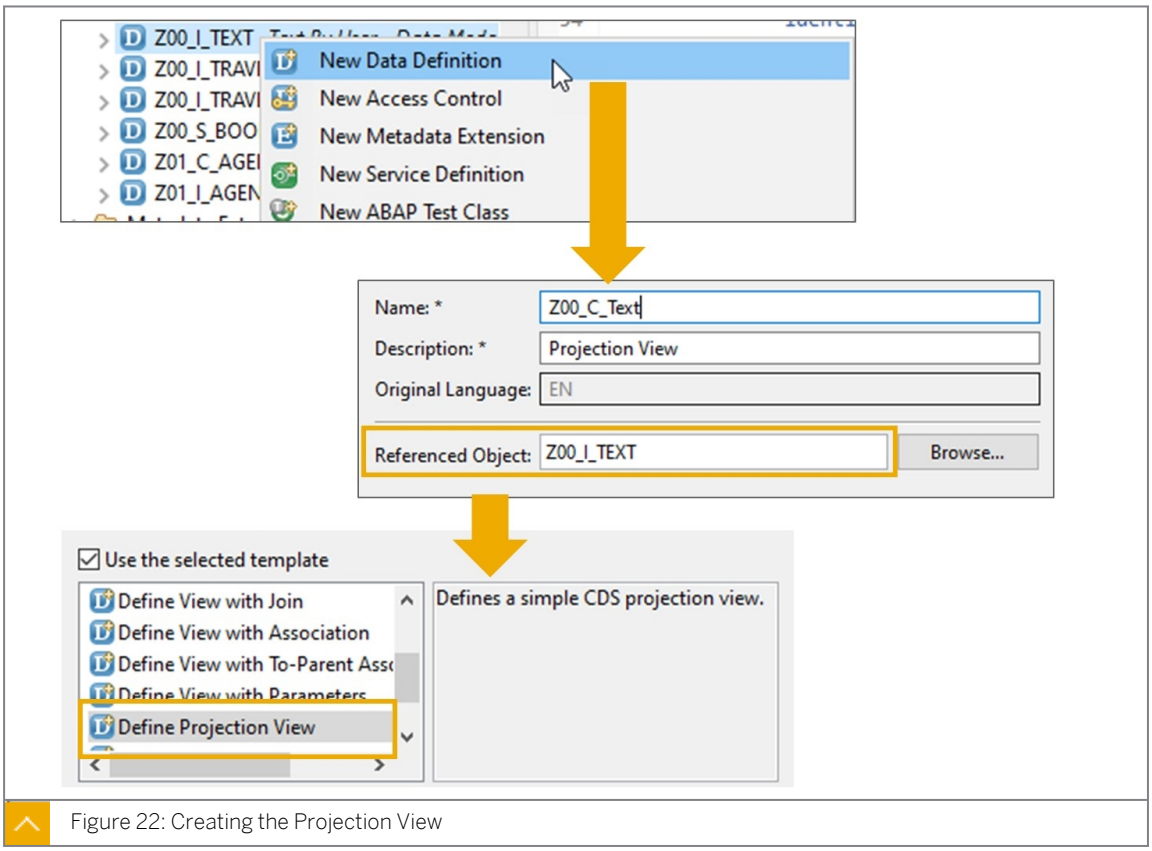

Figure 22: Creating the Projection View

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.

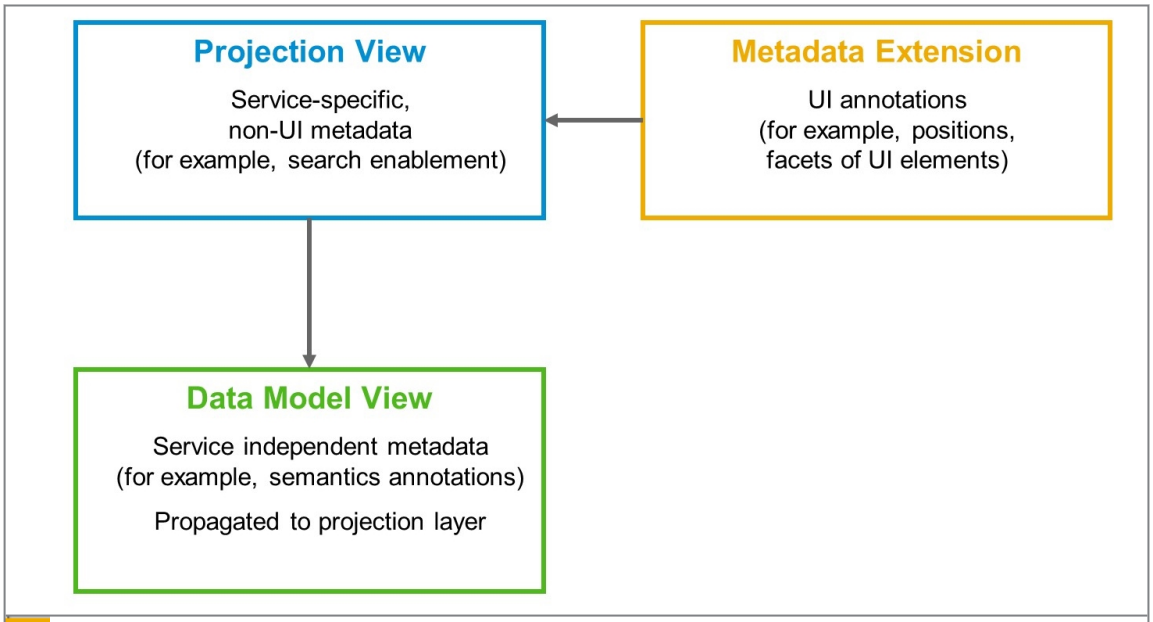

Figure 23: CDS Annotations for Service Metadata

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

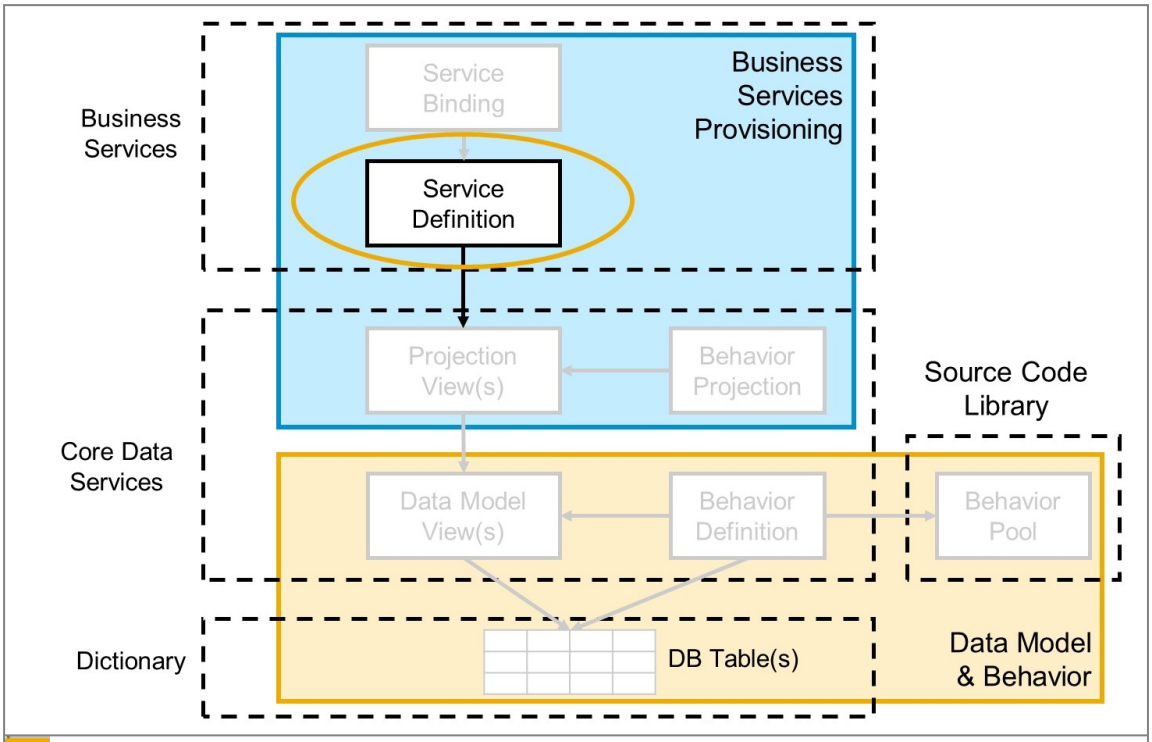

Figure 24: The Service Definition

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.

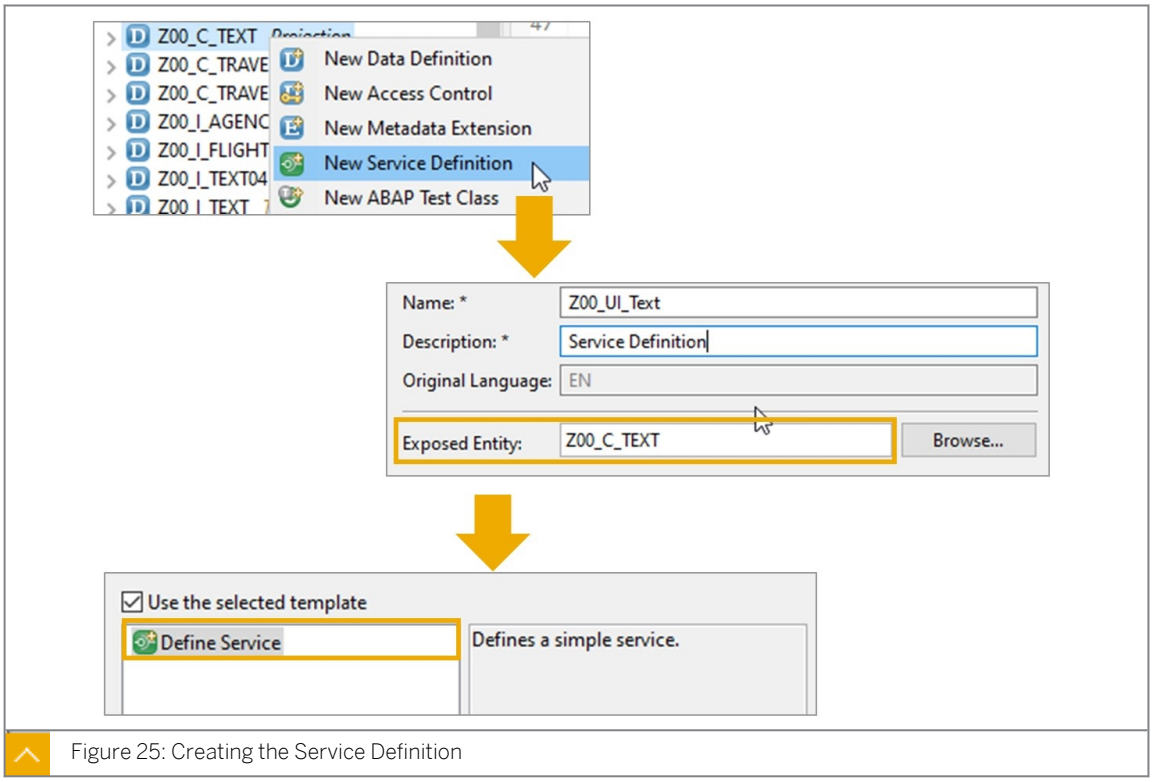

Figure 25: Creating the Service Definition

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.

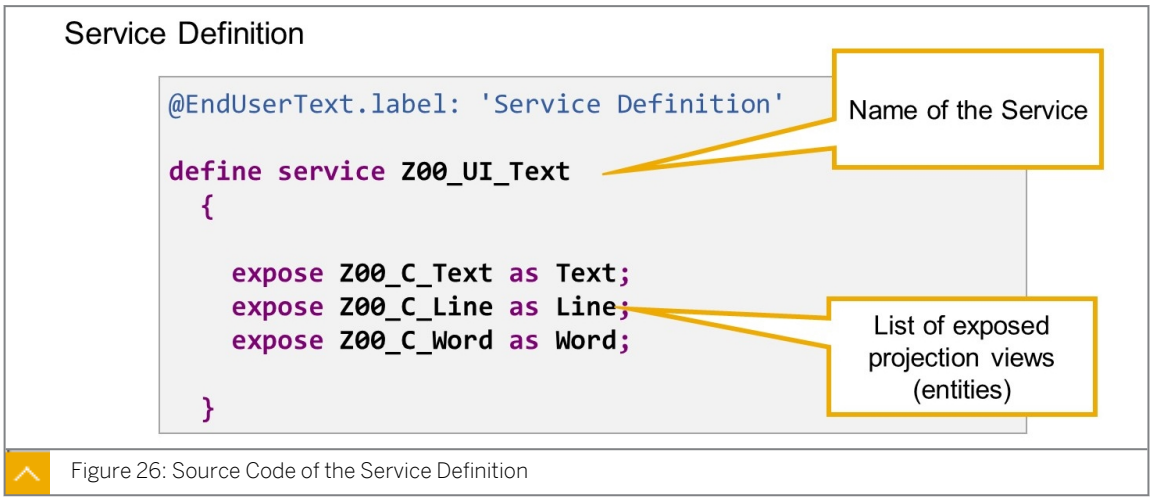

Figure 26: Source Code of the Service Definition

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

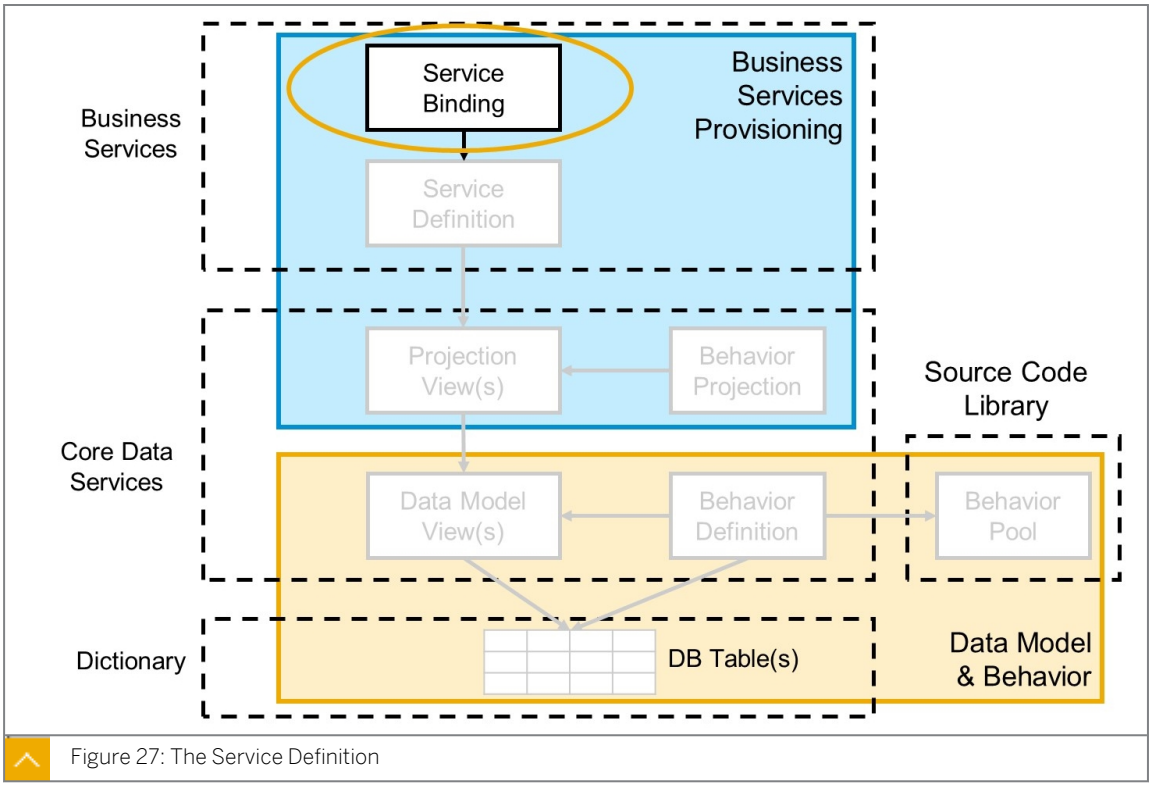

Figure 27: The Service Definition

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.

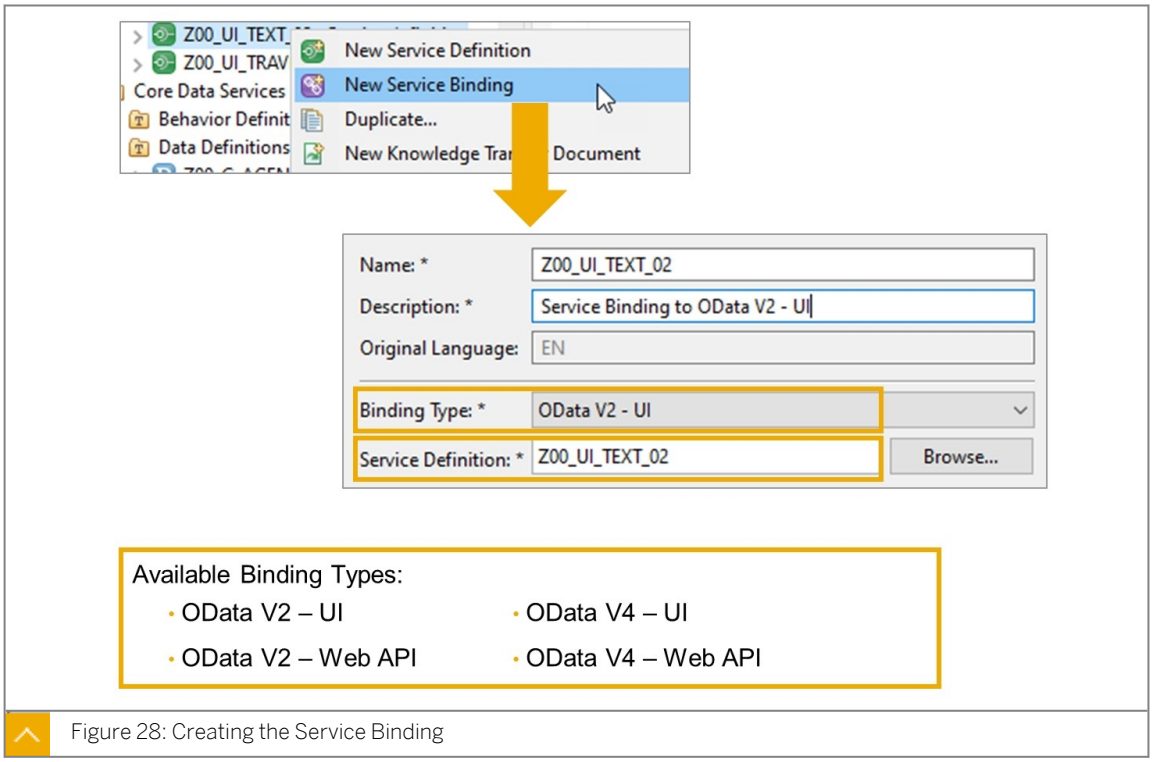

Figure 28: Creating the Service Binding

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.

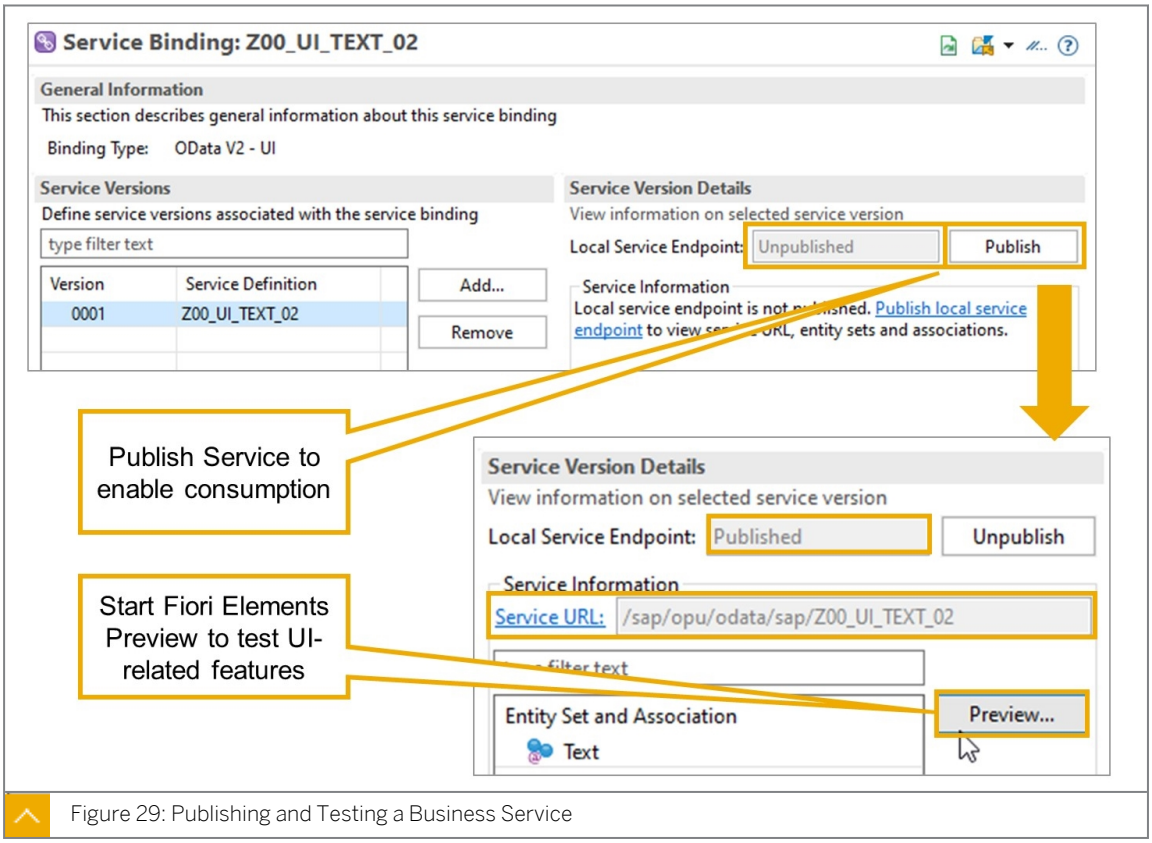

Figure 29: Publishing and Testing a Business Service

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.

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)

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

Unit 2
Lesson 1

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

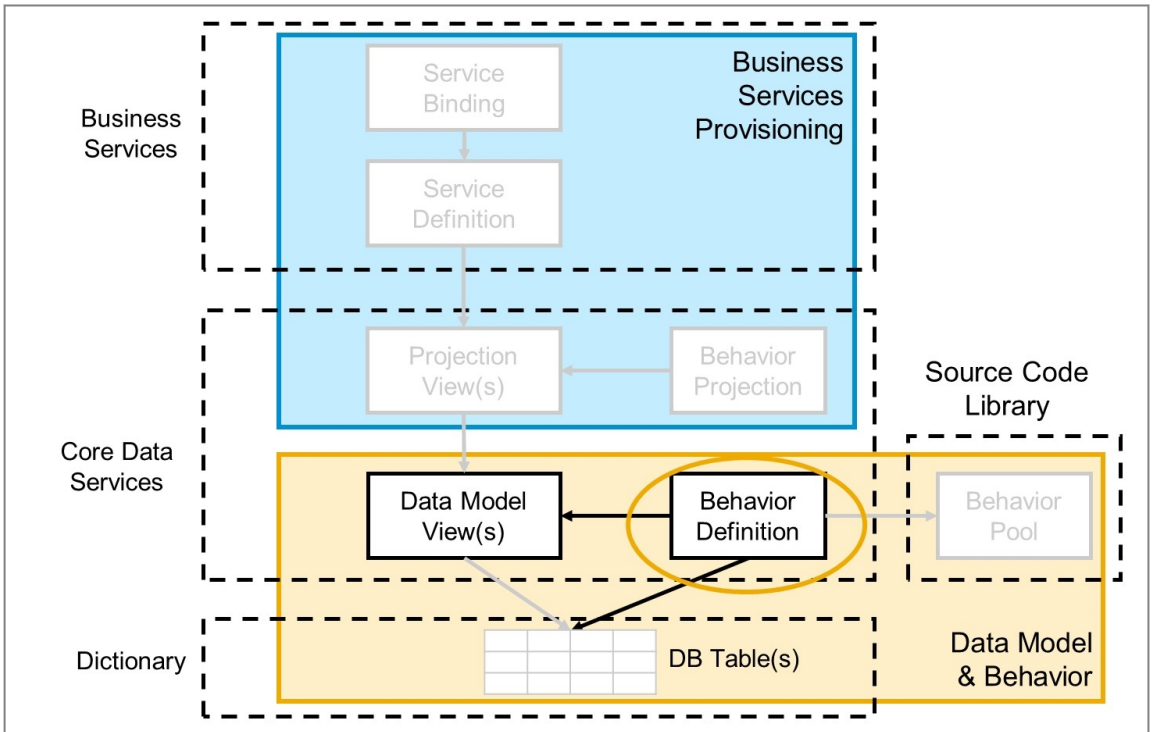

Figure 30: The Behavior Definition

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).

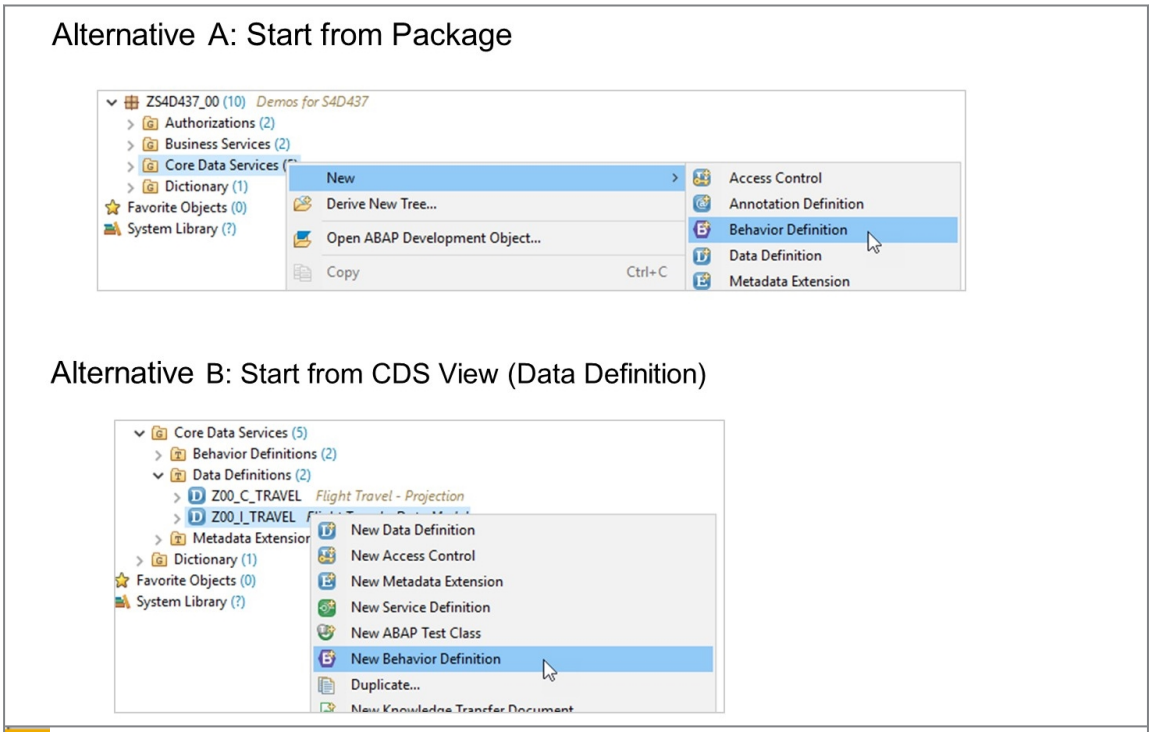

Figure 31: Create a New Behavior Definition (1)

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.

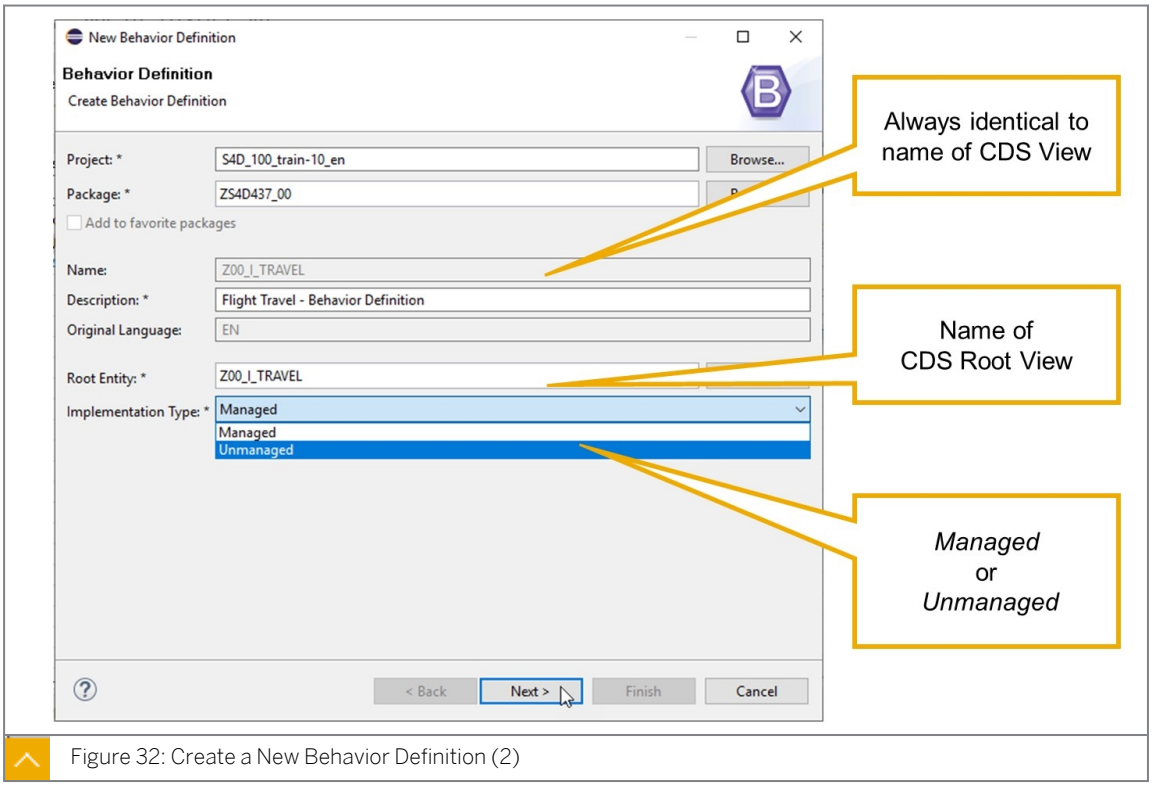

Figure 32: Create a New Behavior Definition (2)

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.

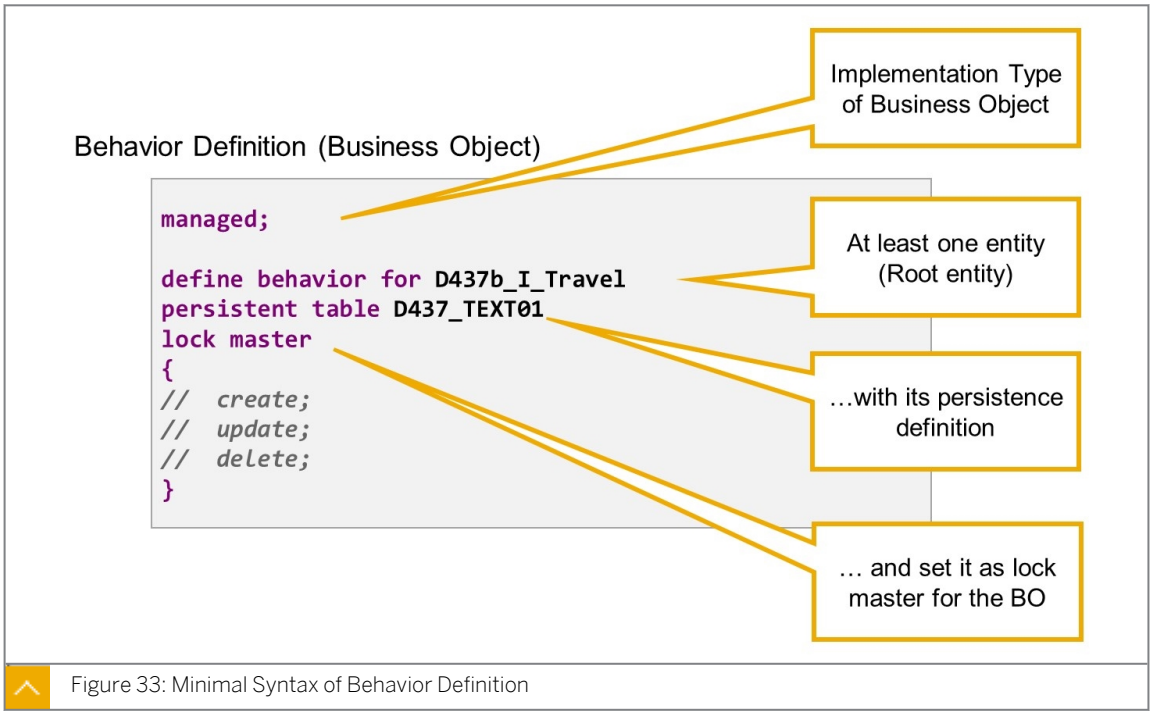

Figure 33: Minimal Syntax of Behavior Definition

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

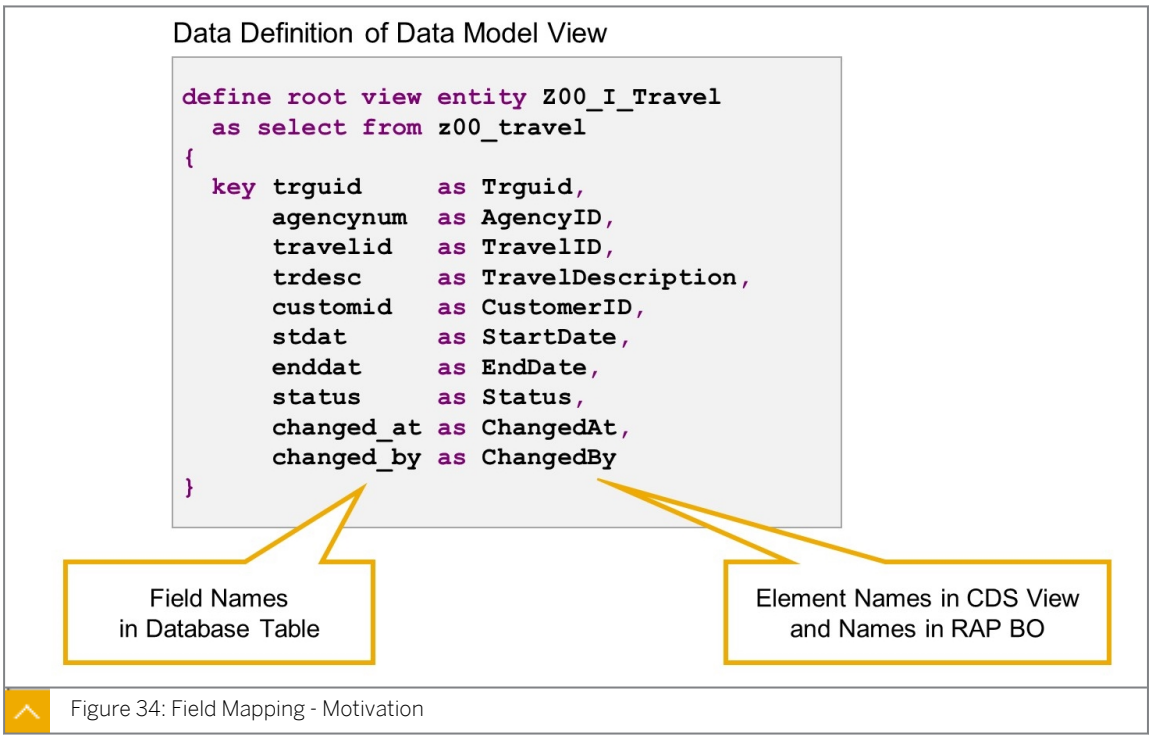

Figure 34: Field Mapping - Motivation

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.

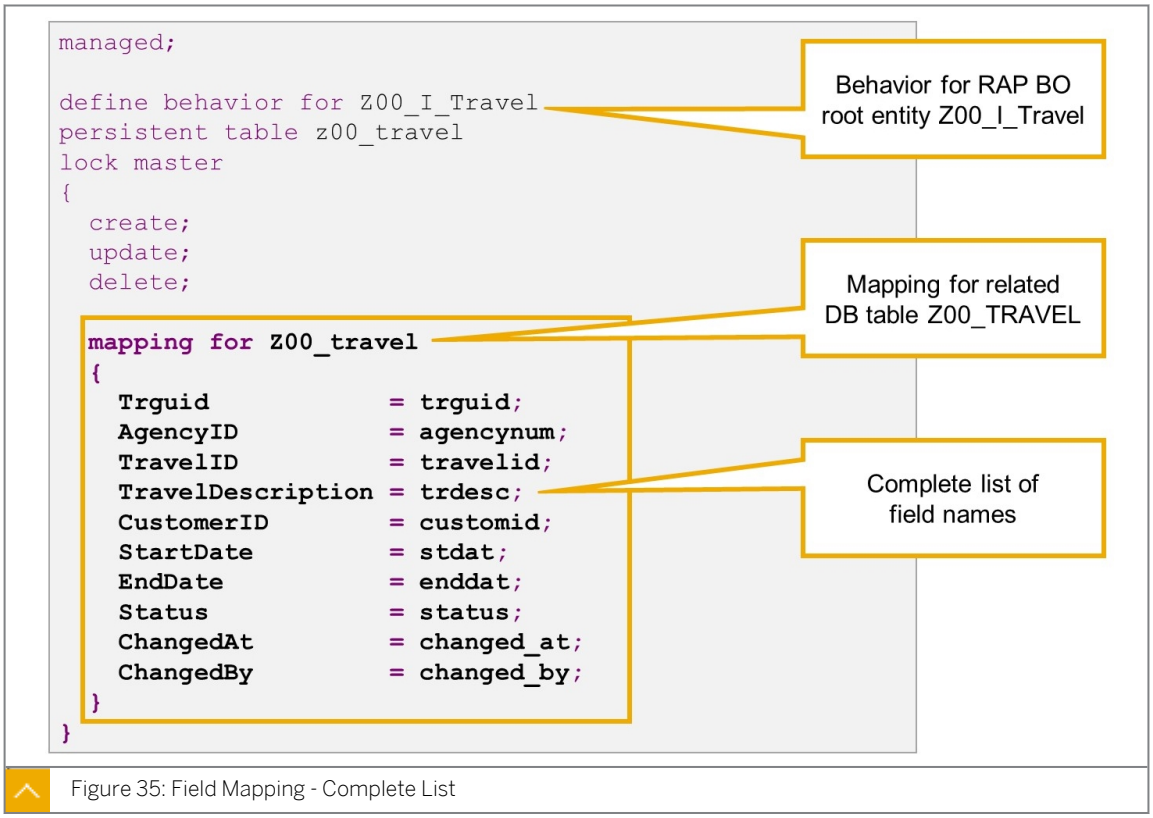

Figure 35: Field Mapping - Complete List

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.

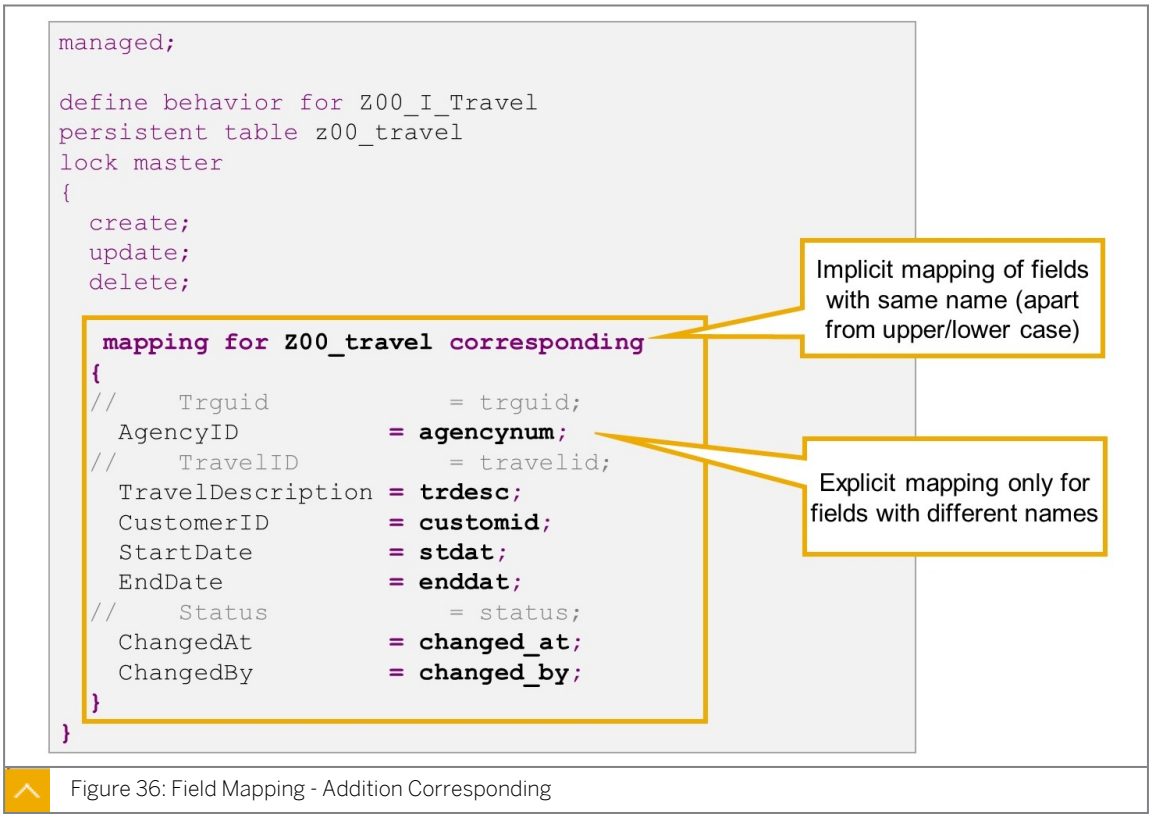

Figure 36: Field Mapping - Addition Corresponding

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

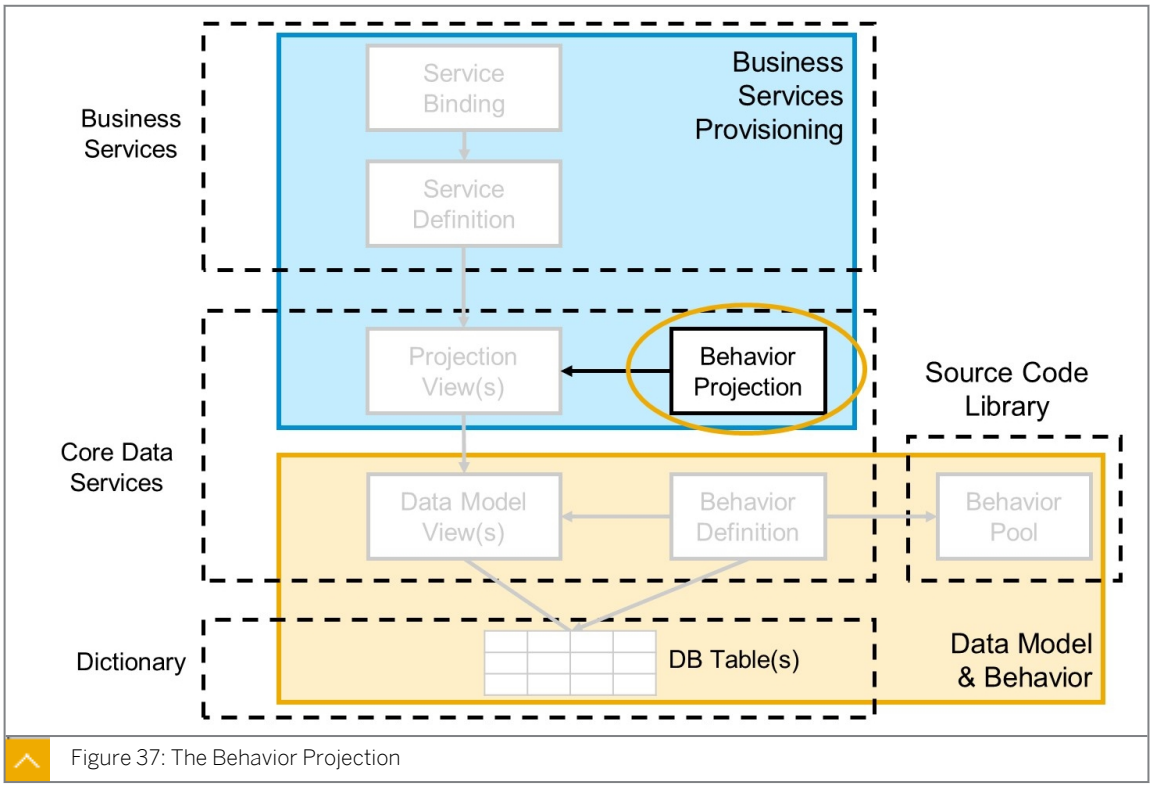

Figure 37: The Behavior Projection

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.

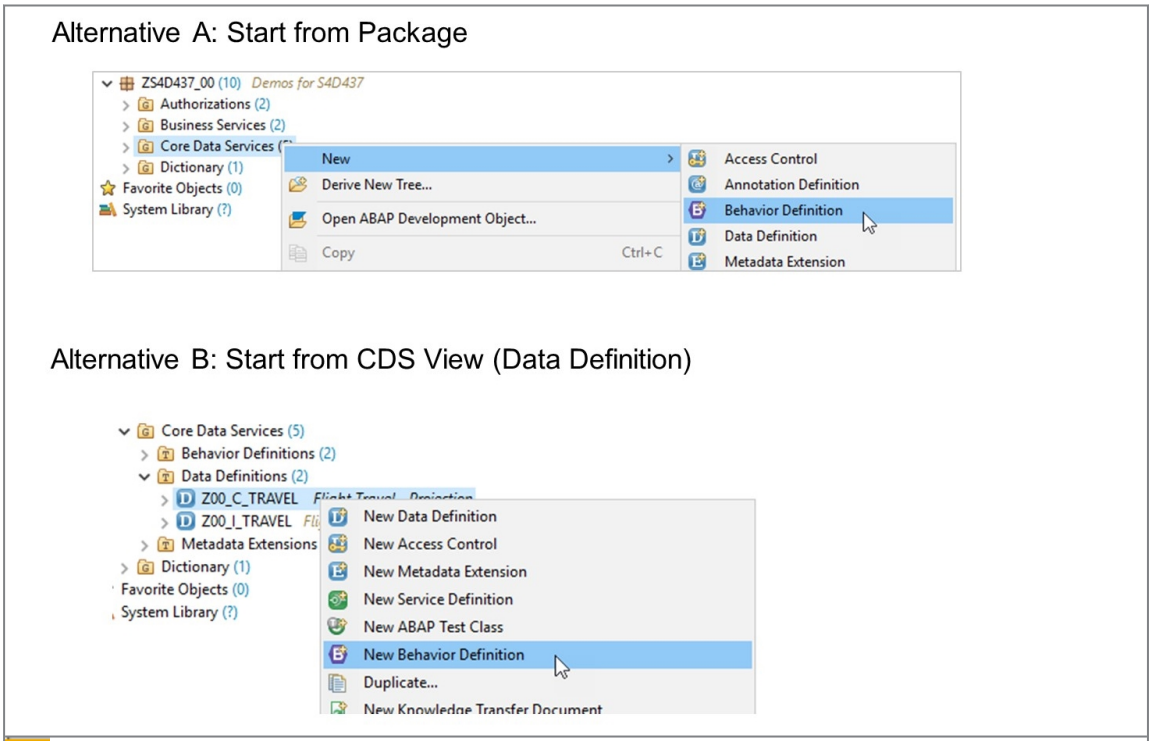

Figure 38: Create a New Behavior Projection (1)

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.

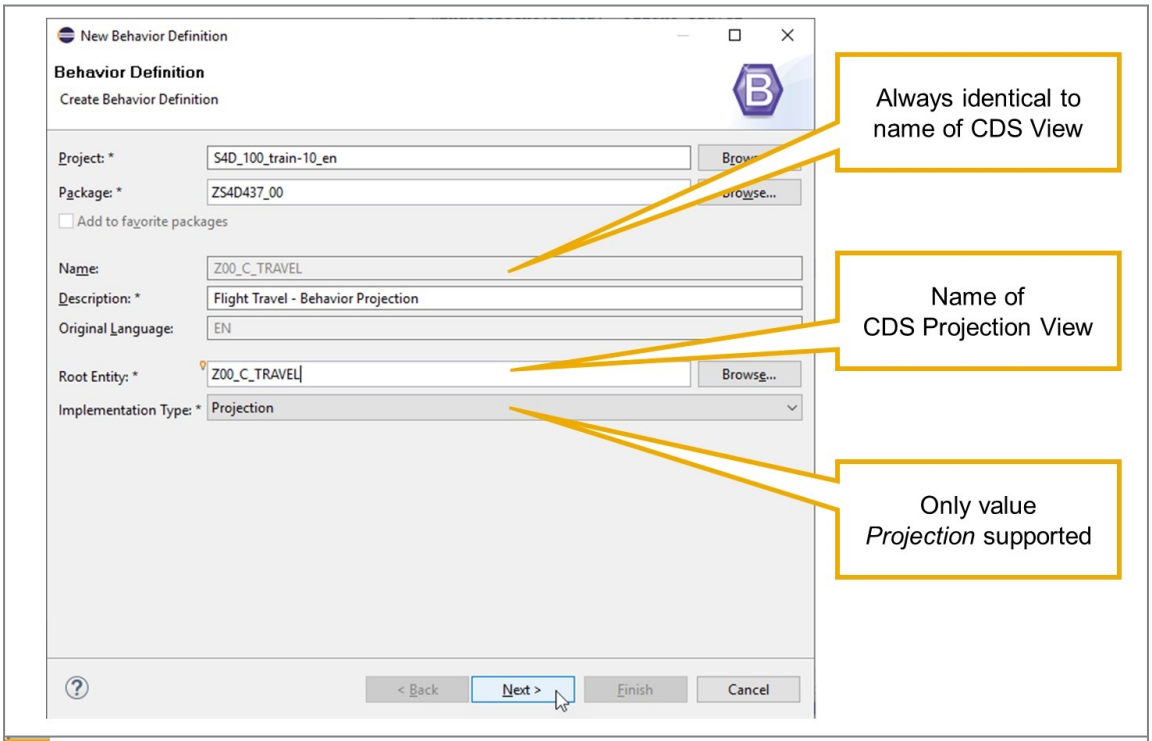

Figure 39: Create a New Behavior Definition (2)

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.

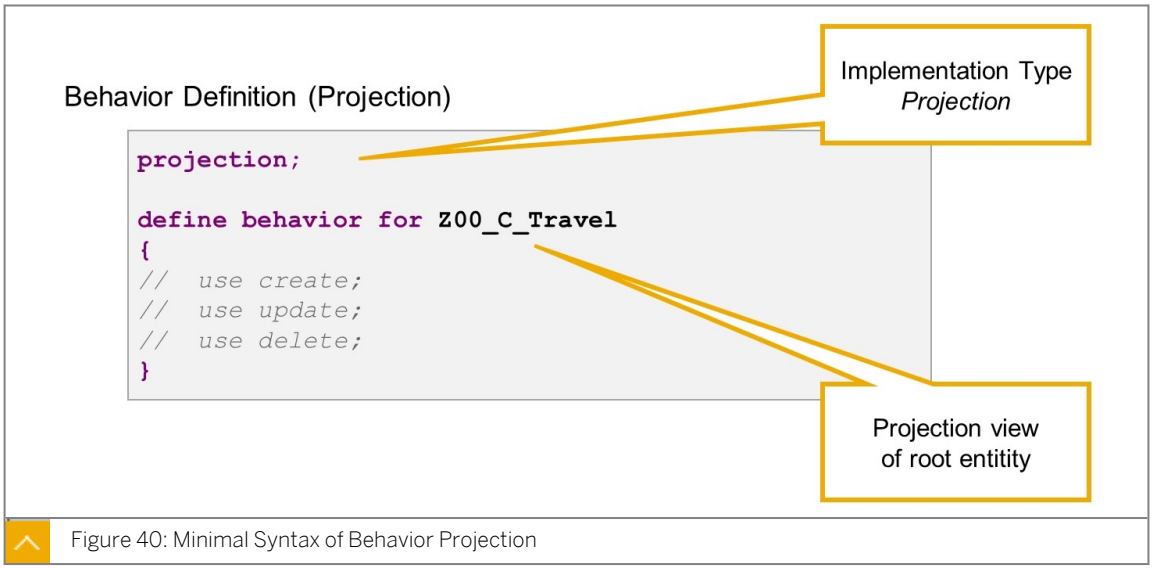

Figure 40: Minimal Syntax of Behavior Projection

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.

LESSON SUMMARY

You should now be able to:

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

Unit 2
Lesson 2

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.

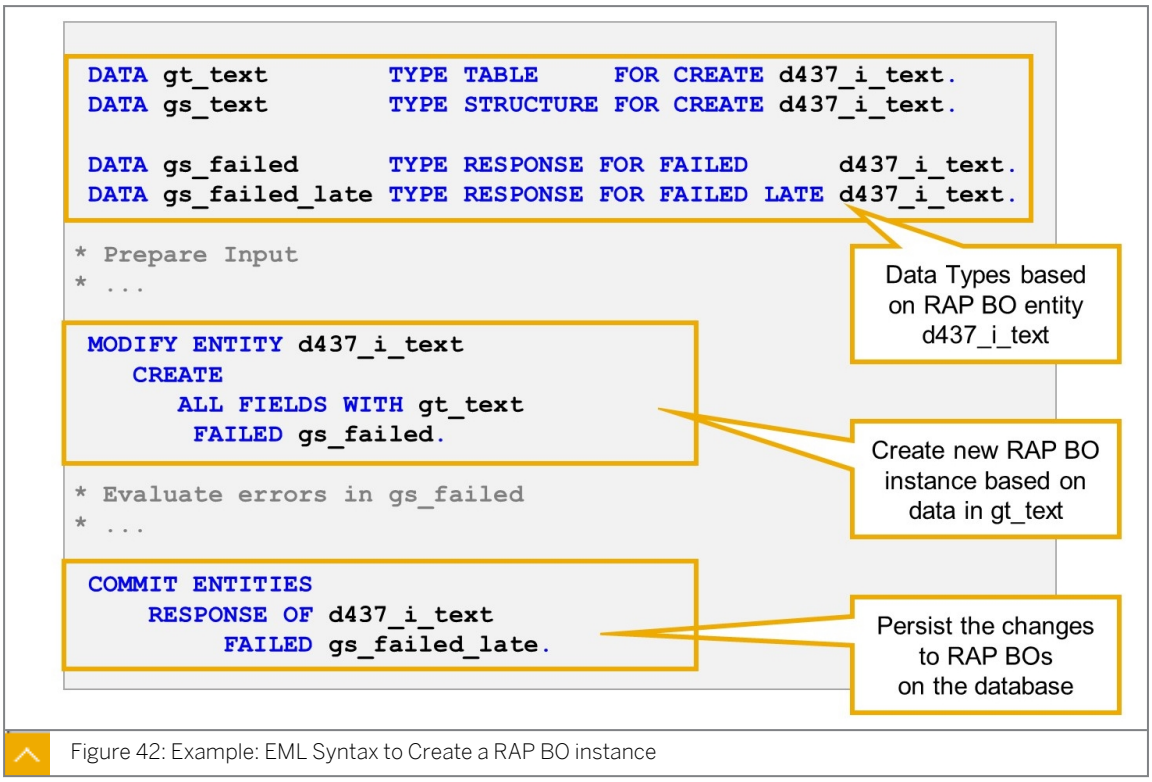

Figure 42: Example: EML Syntax to Create a RAP BO instance

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

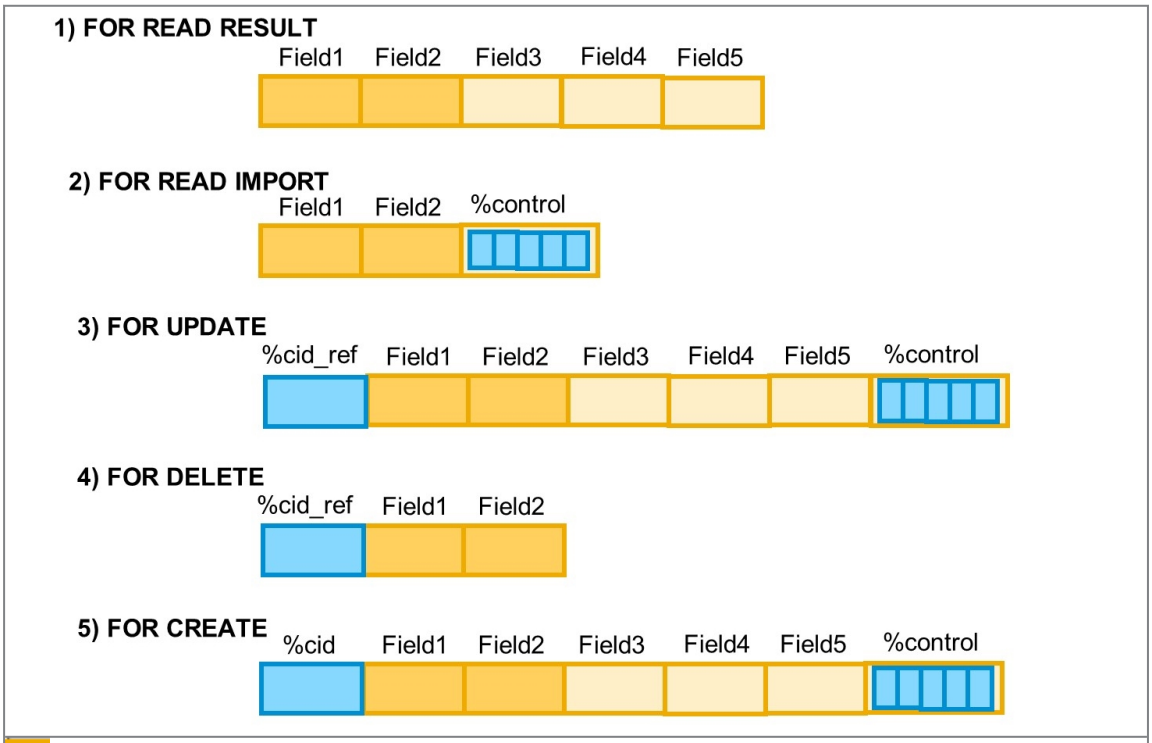

Figure 44: Overview of Derived Data Types for Import and Result

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.

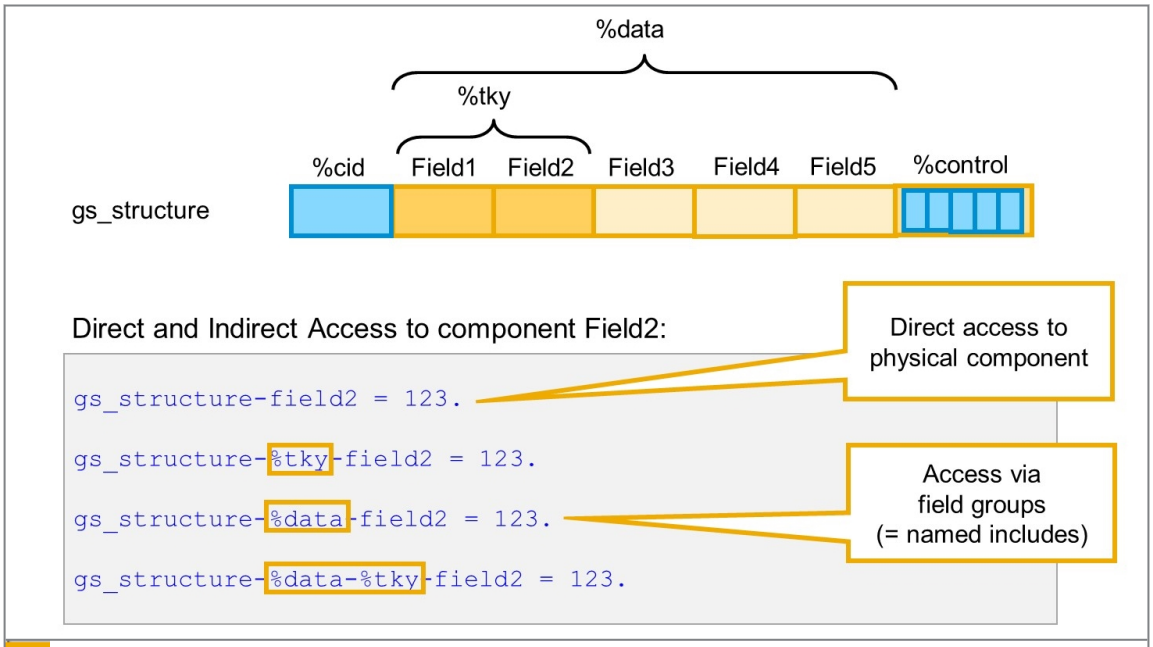

Figure 45: Component Groups in Derived Types

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

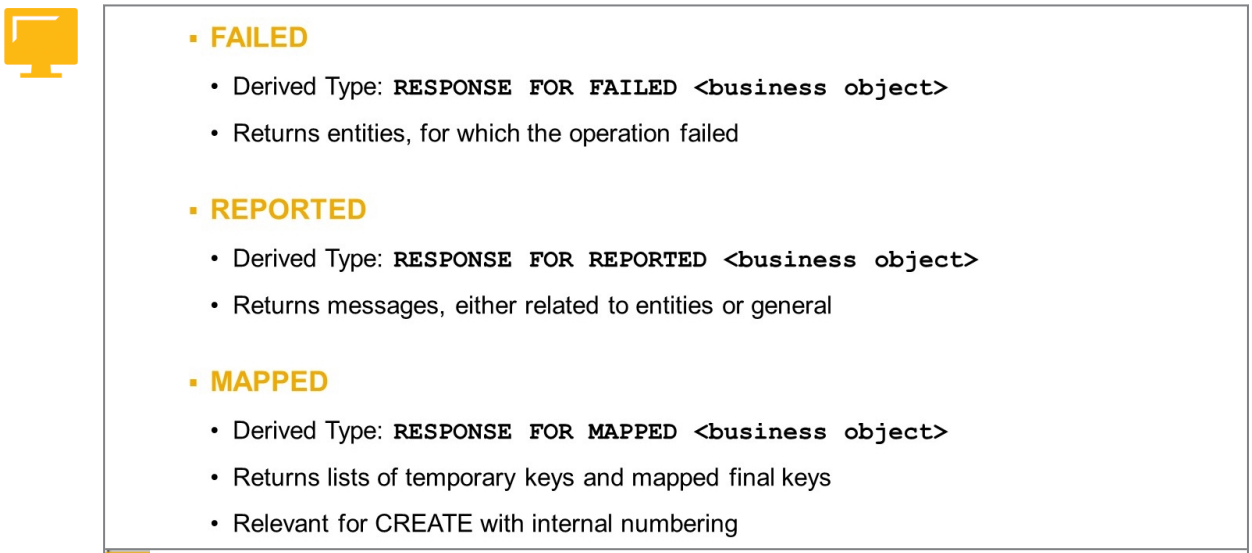

Figure 46: Response Operands

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.

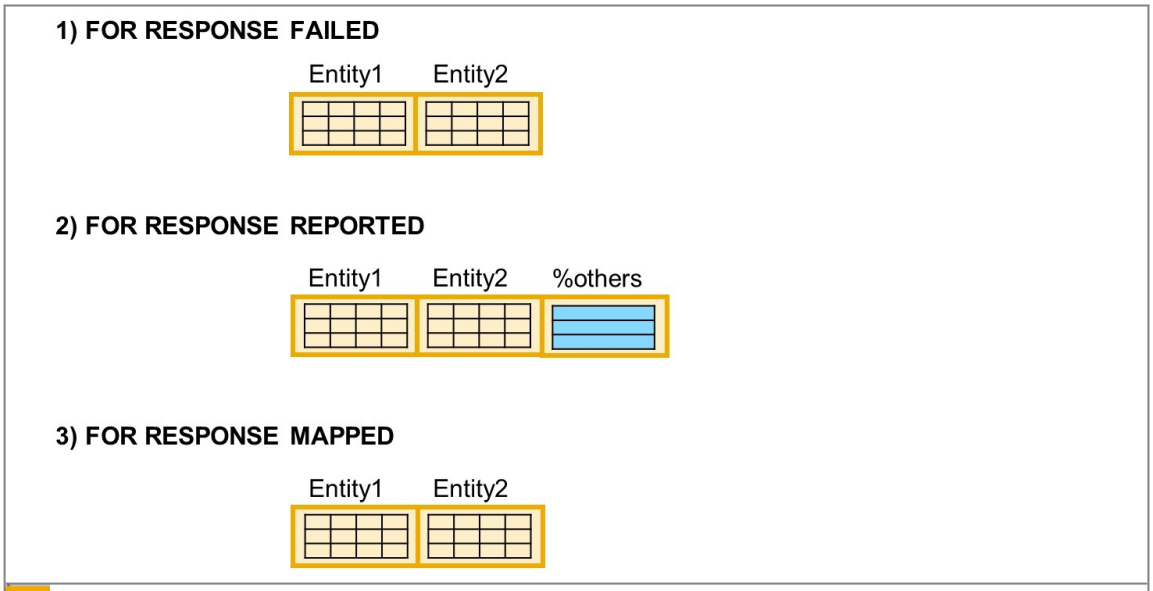

Figure 47: Overview of Derived Data Types for Response

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.

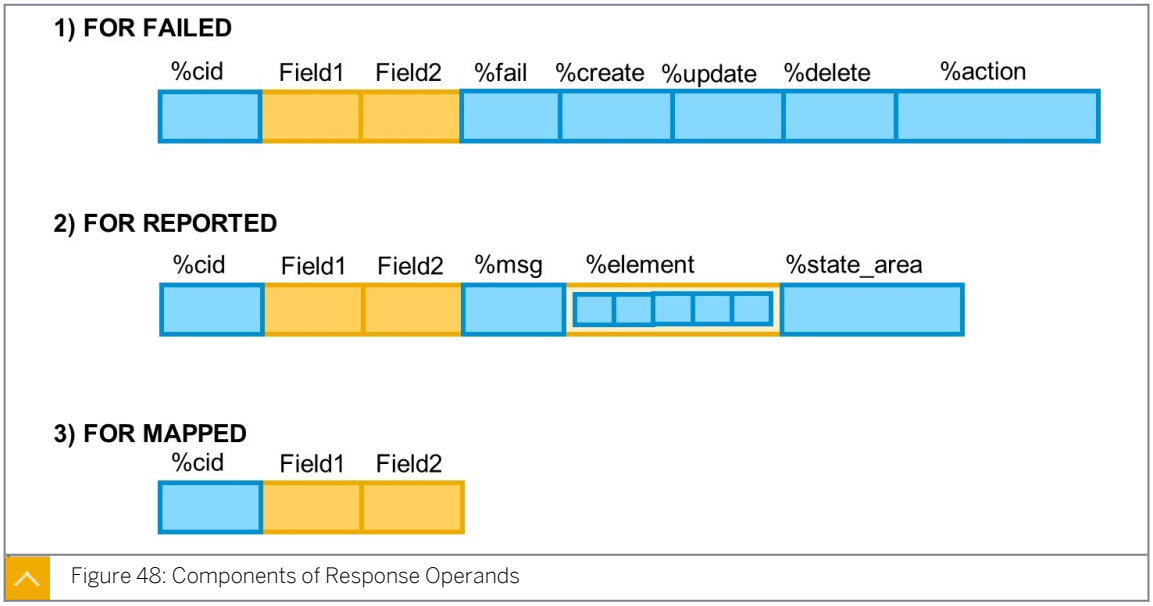

Figure 48: Components of Response Operands

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

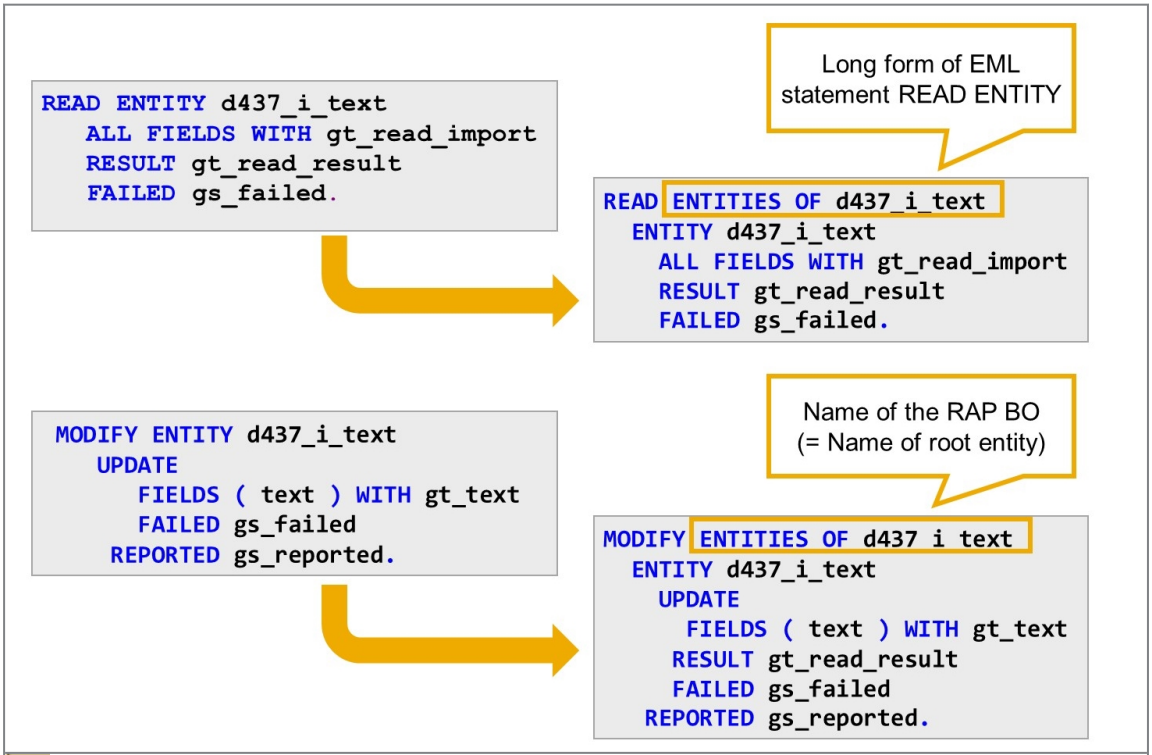

Figure 49: Short Form verus Long Form

Statements `READ ENTITY` and `MODIFY ENITITY` 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 on 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

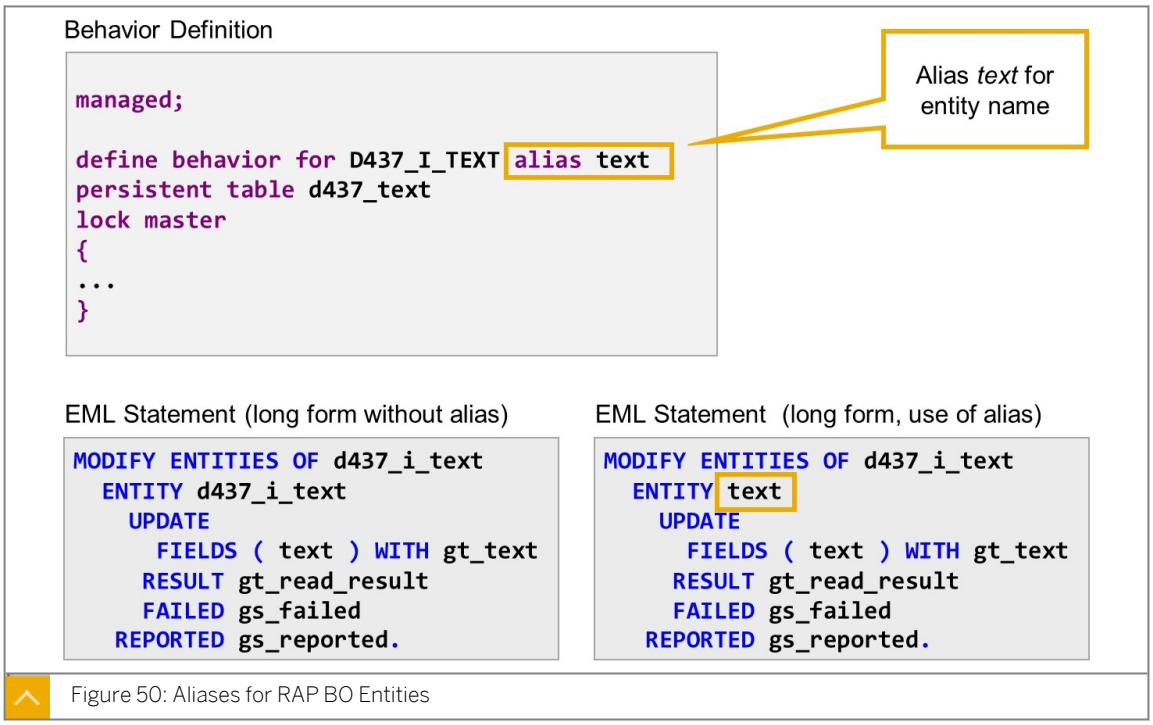

Figure 50: Aliases for RAP BO Entities

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.

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

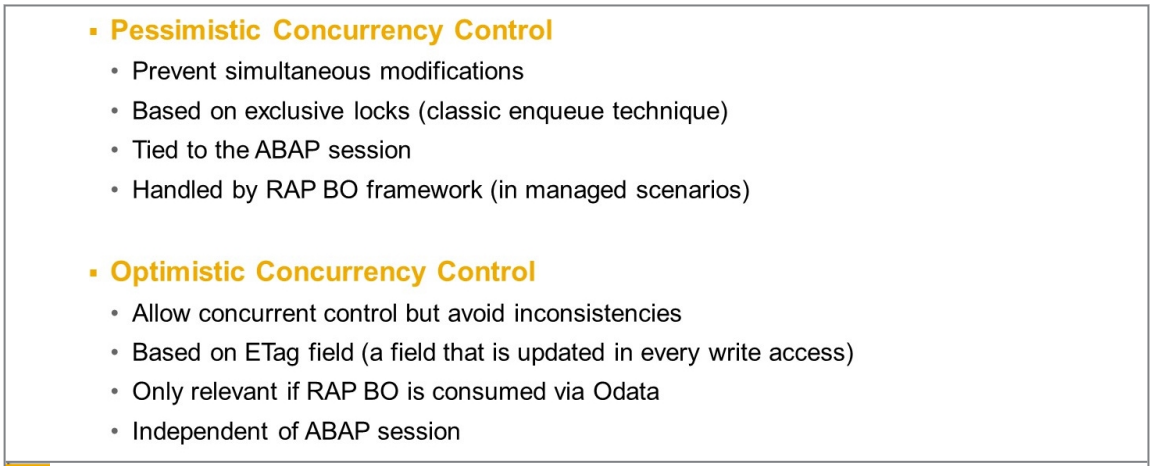

Figure 51: Concurrency Control in RAP

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 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

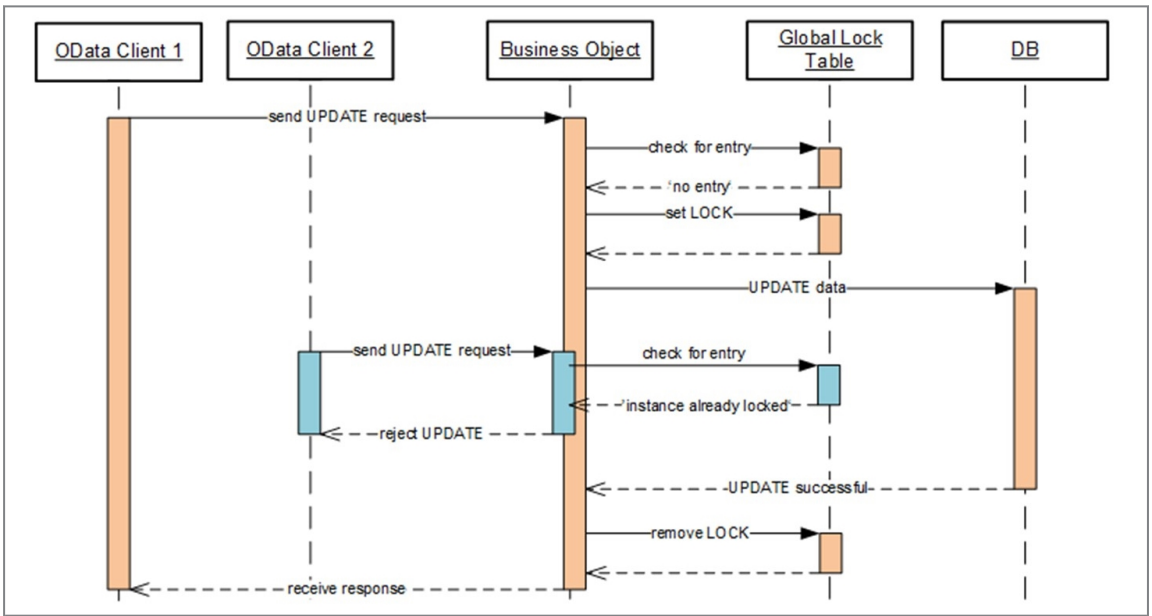

Figure 52: Locking During UPDATE Operation

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

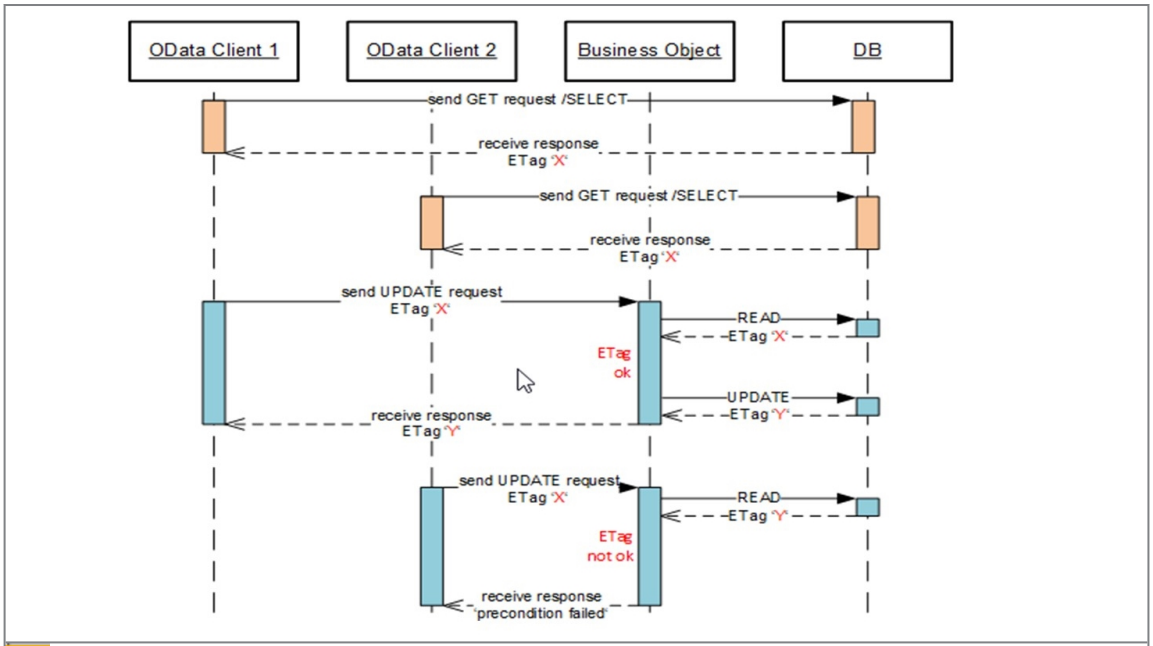

Figure 53: ETag Check in UPDATE Operation

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

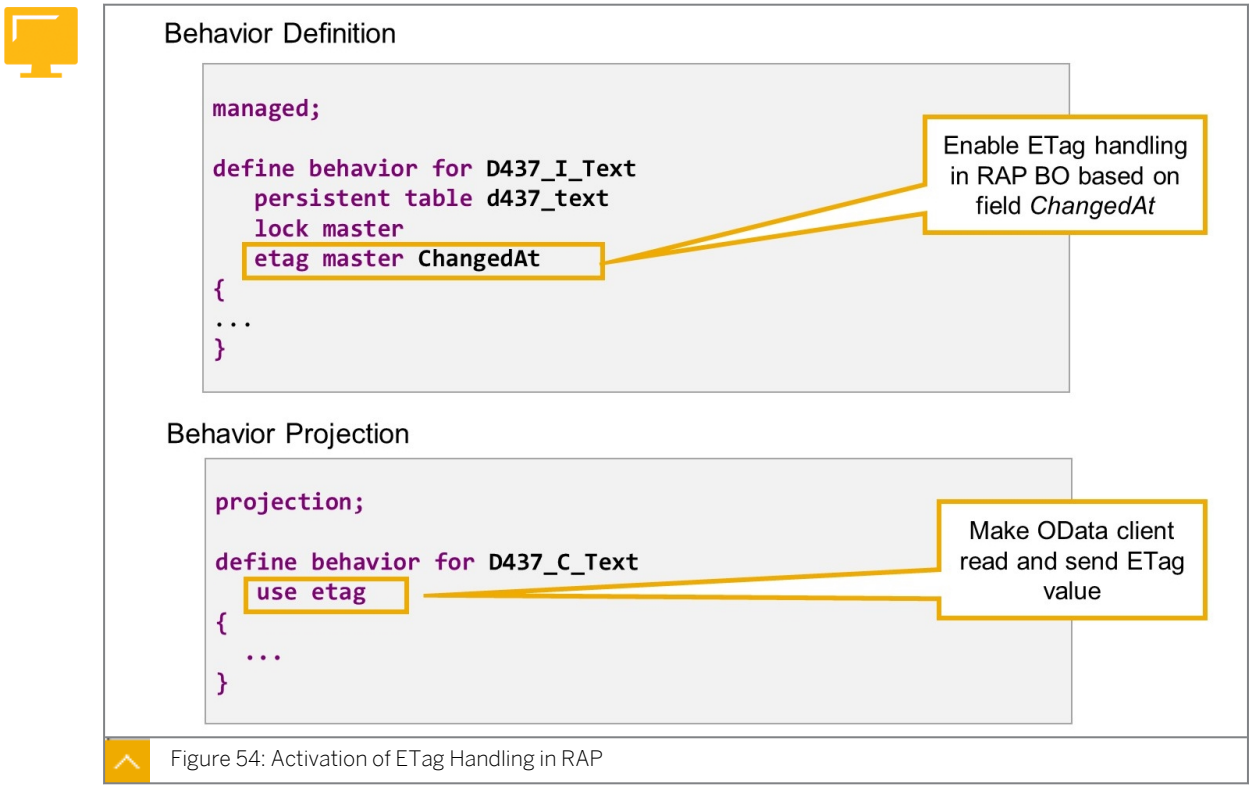

Figure 54: Activation of ETag Handling in RAP

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.

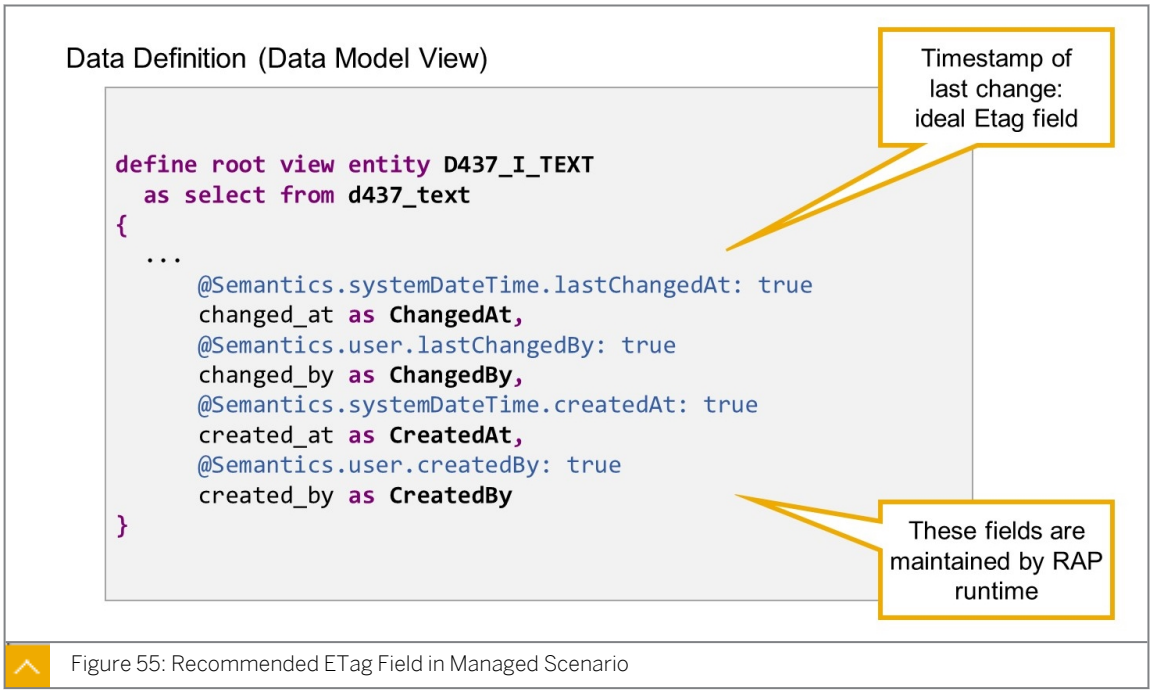

Figure 55: Recommended ETag Field in Managed Scenario

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.

LESSON SUMMARY

You should now be able to:

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

Unit 2
Lesson 4

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

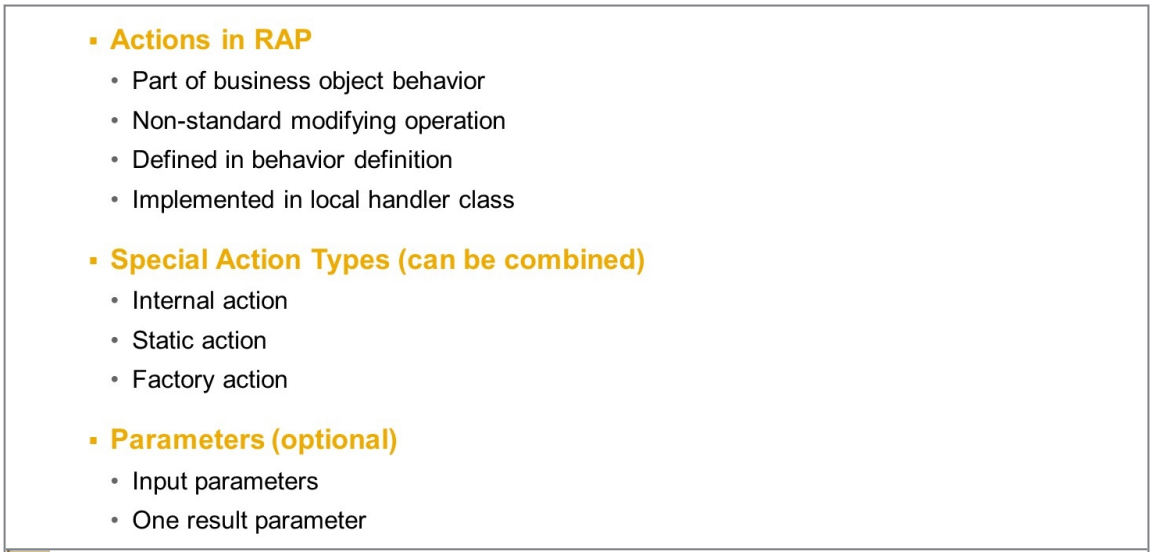

Figure 56: Actions in RAP

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.

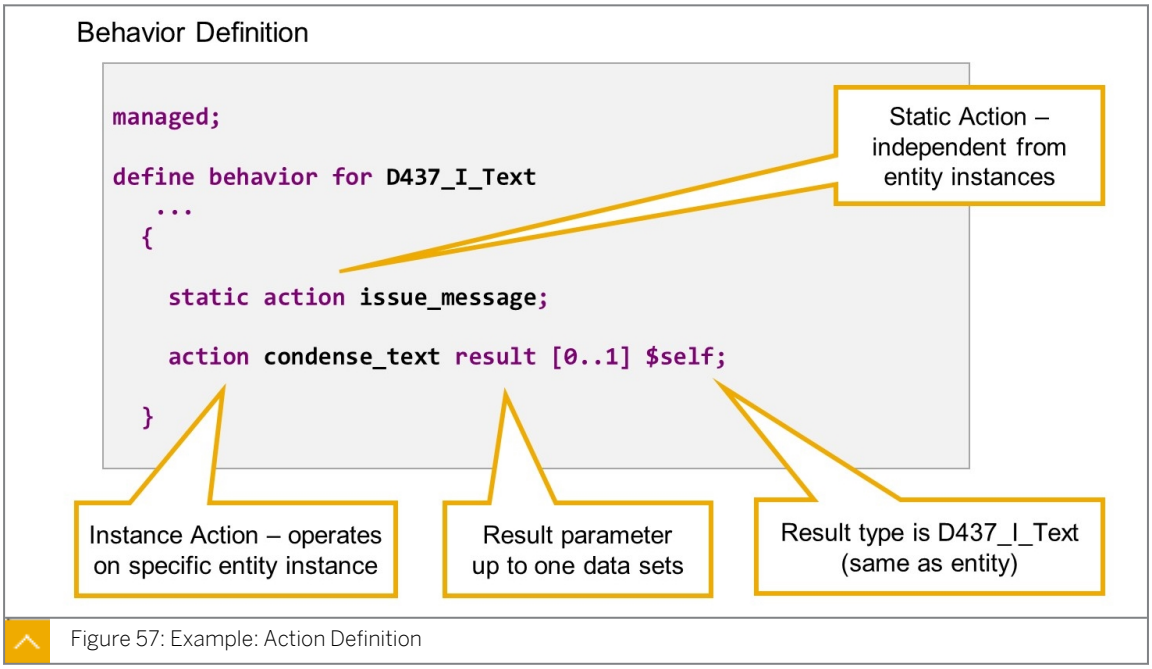

Figure 57: Example: Action Definition

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

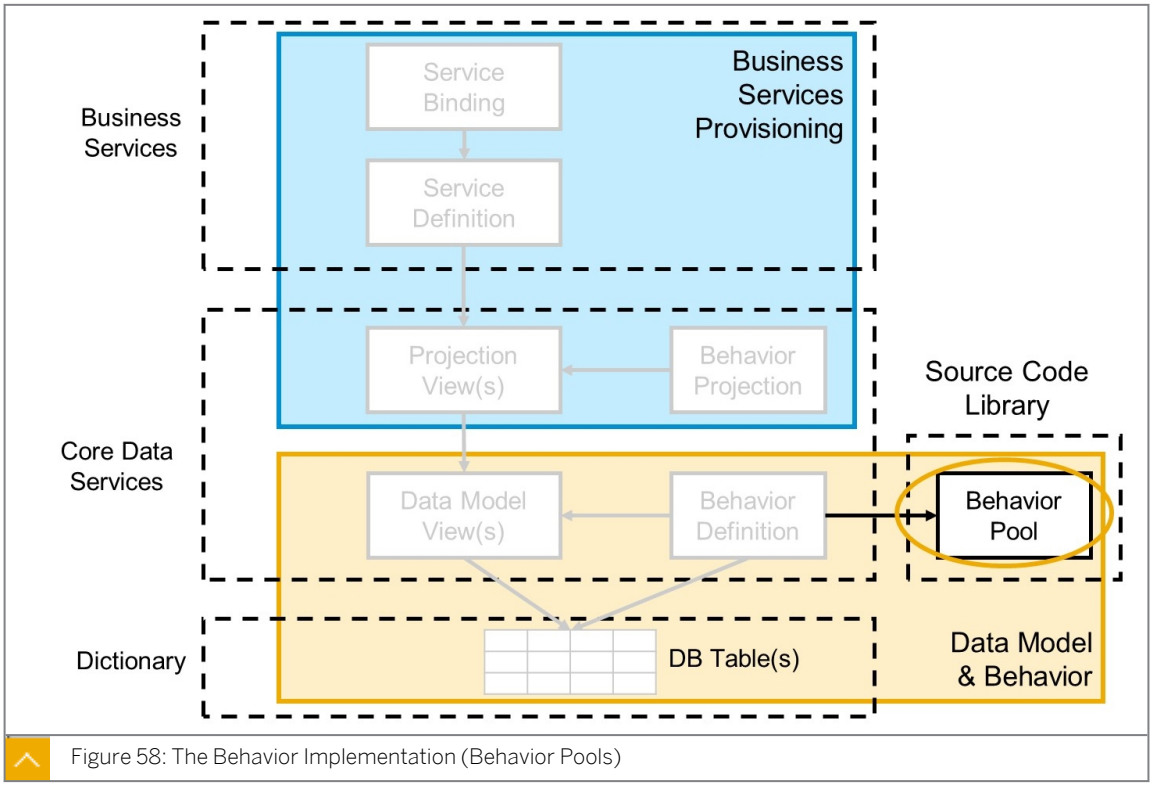

Figure 58: The Behavior Implementation (Behavior Pools)

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.

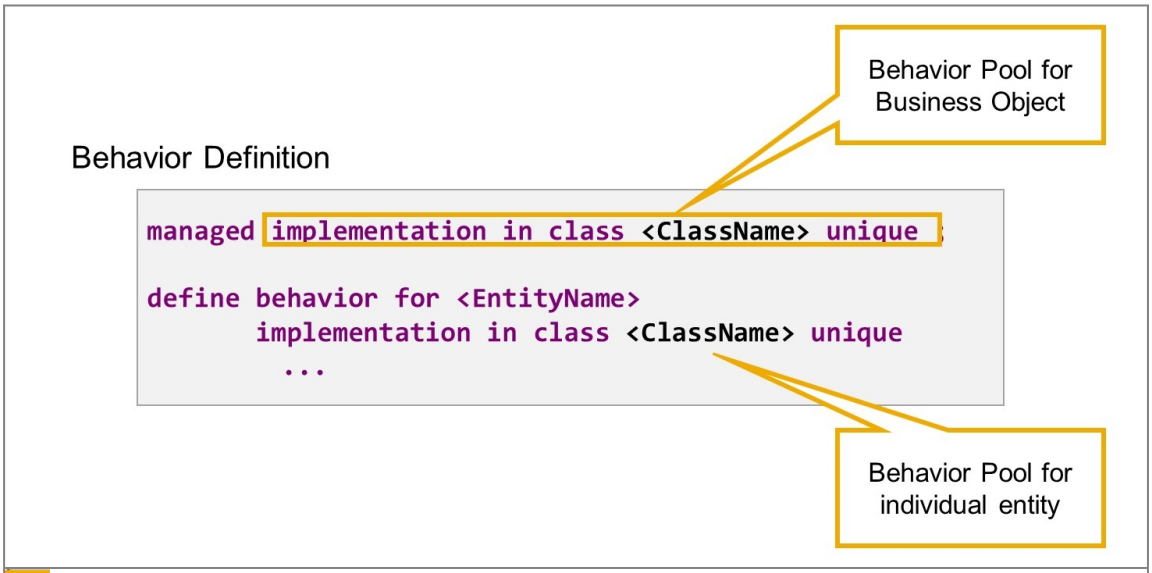

Figure 59: Definition of the Behavior Pool

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.

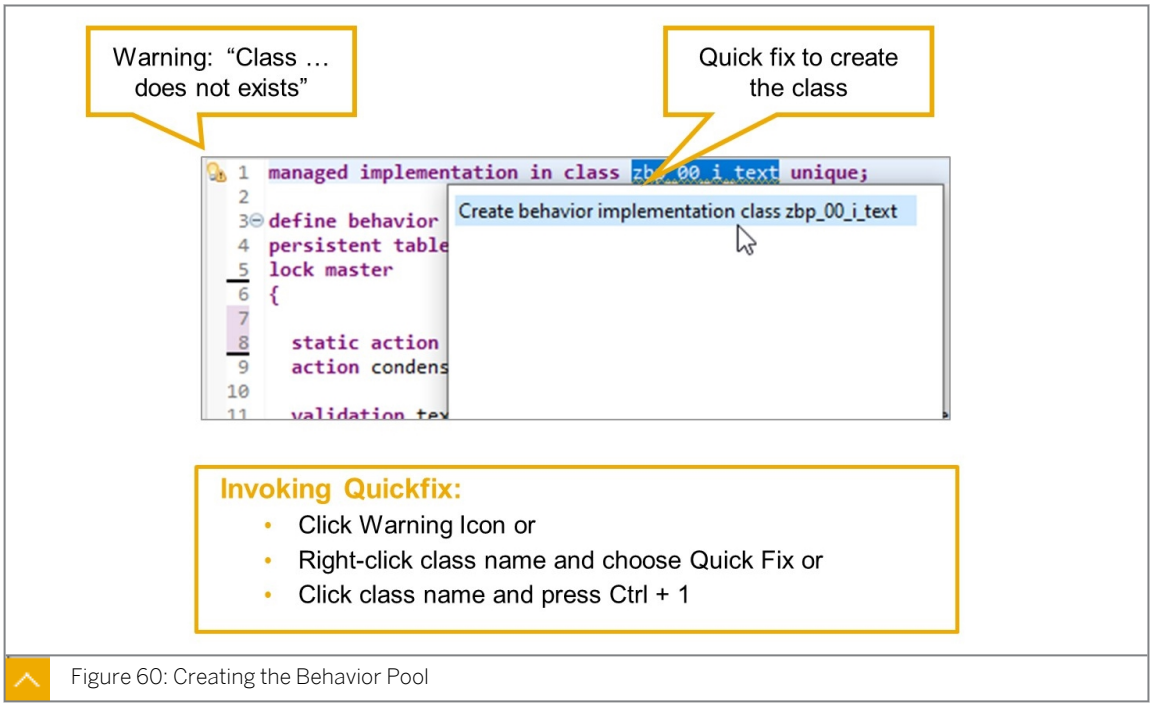

Figure 60: Creating the Behavior Pool

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.

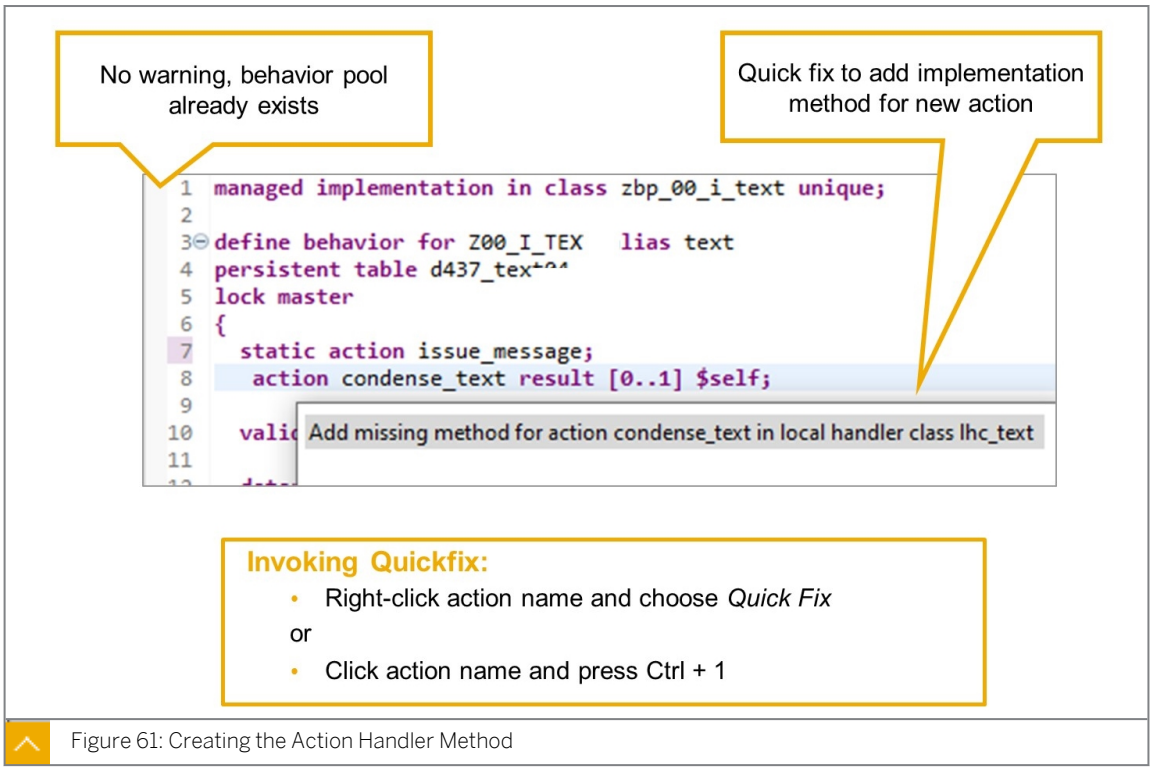

Figure 61: Creating the Action Handler Method

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.

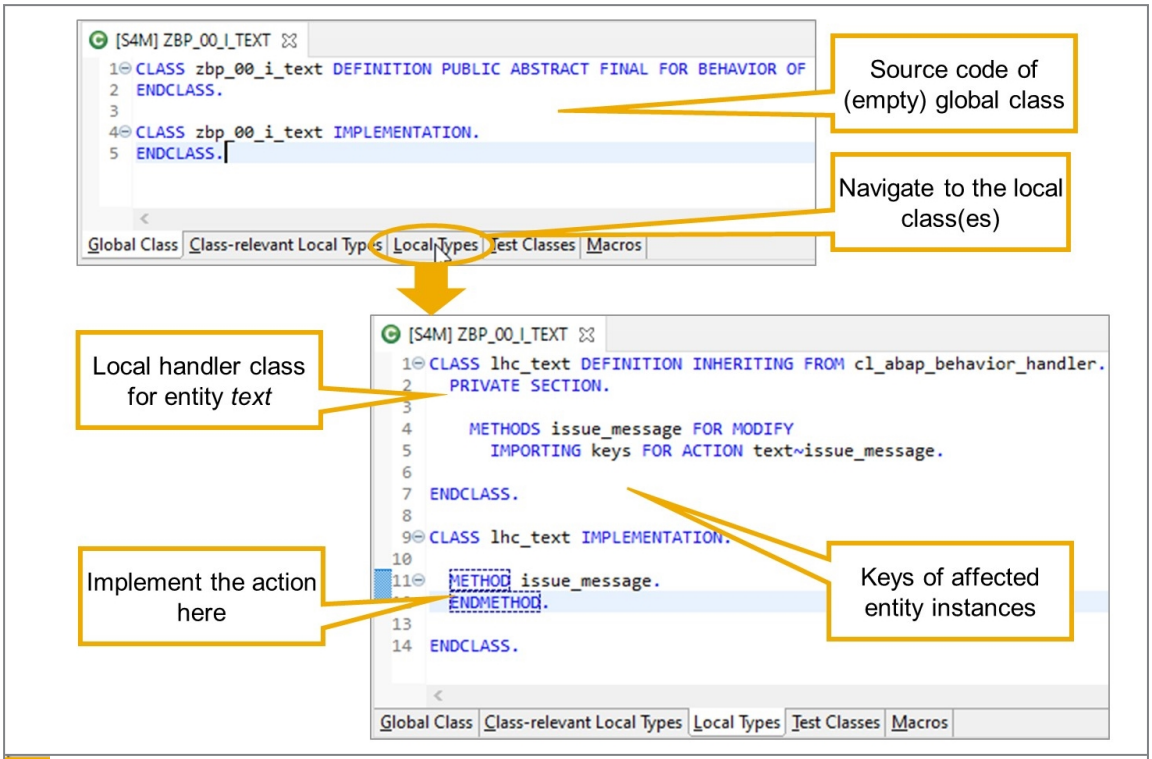

Figure 62: Navigating to the Action Handler Method

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

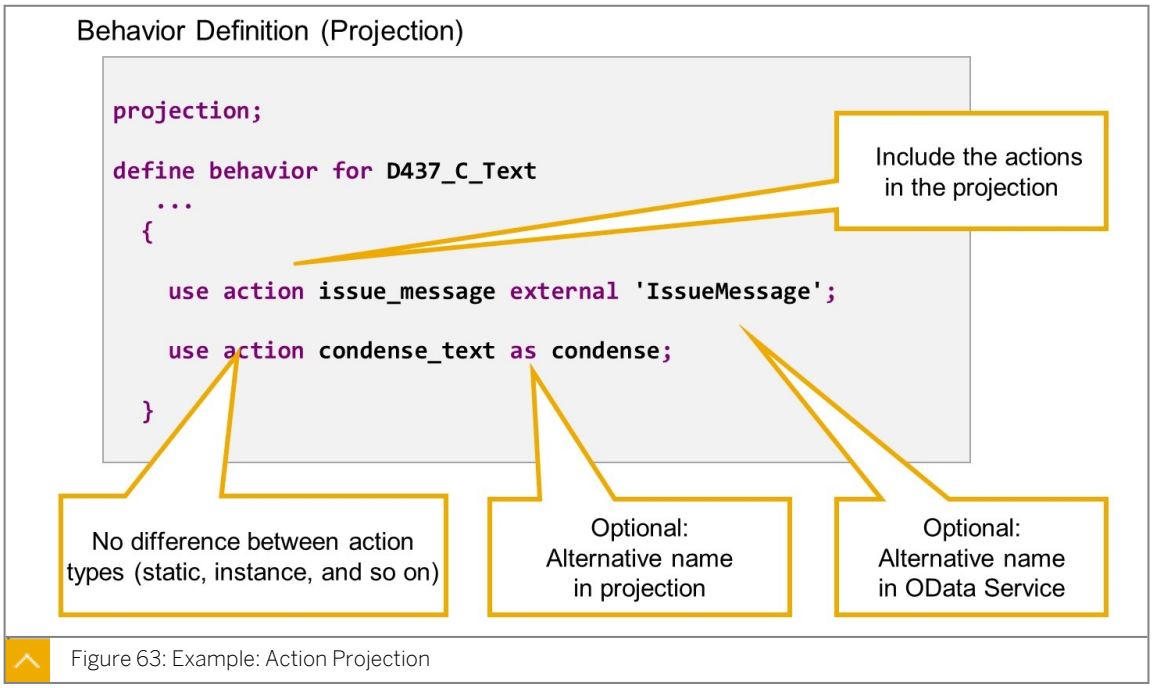

Figure 63: Example: Action Projection

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

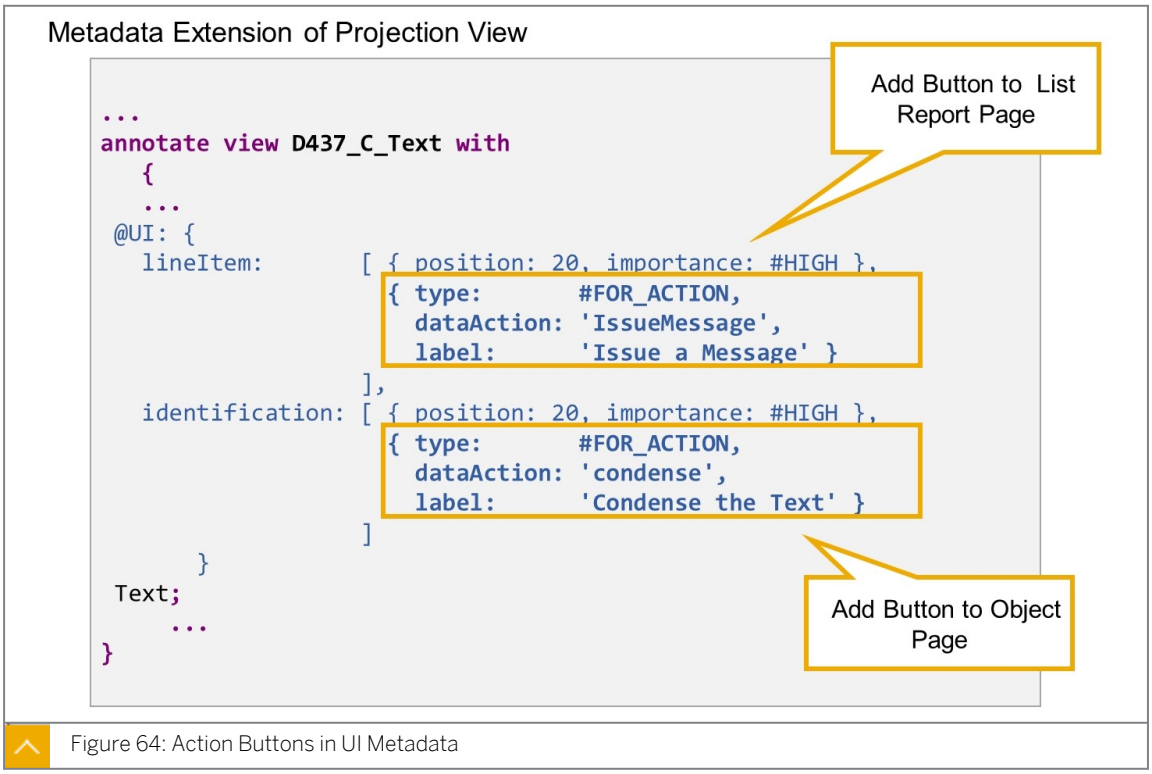

Figure 64: Action Buttons in UI Metadata

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

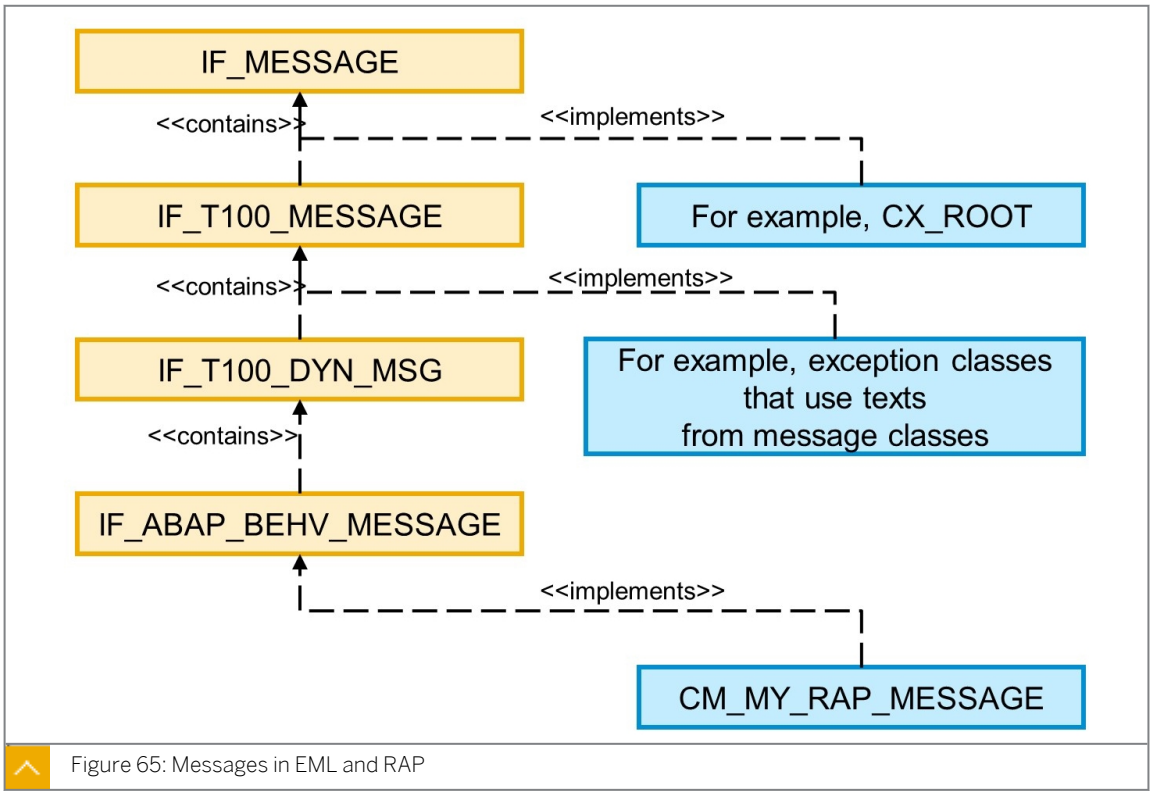

Figure 65: Messages in EML and RAP

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

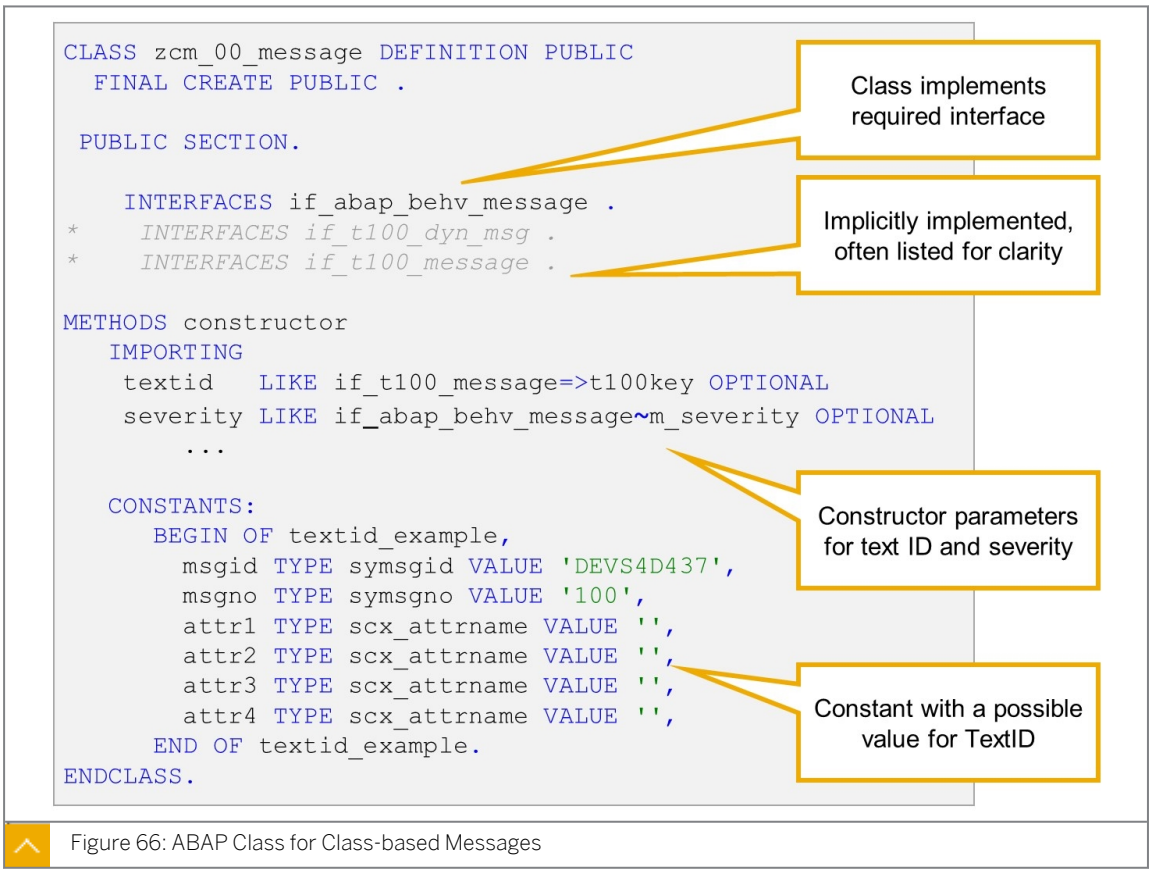

Figure 66: ABAP Class for Class-based Messages

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

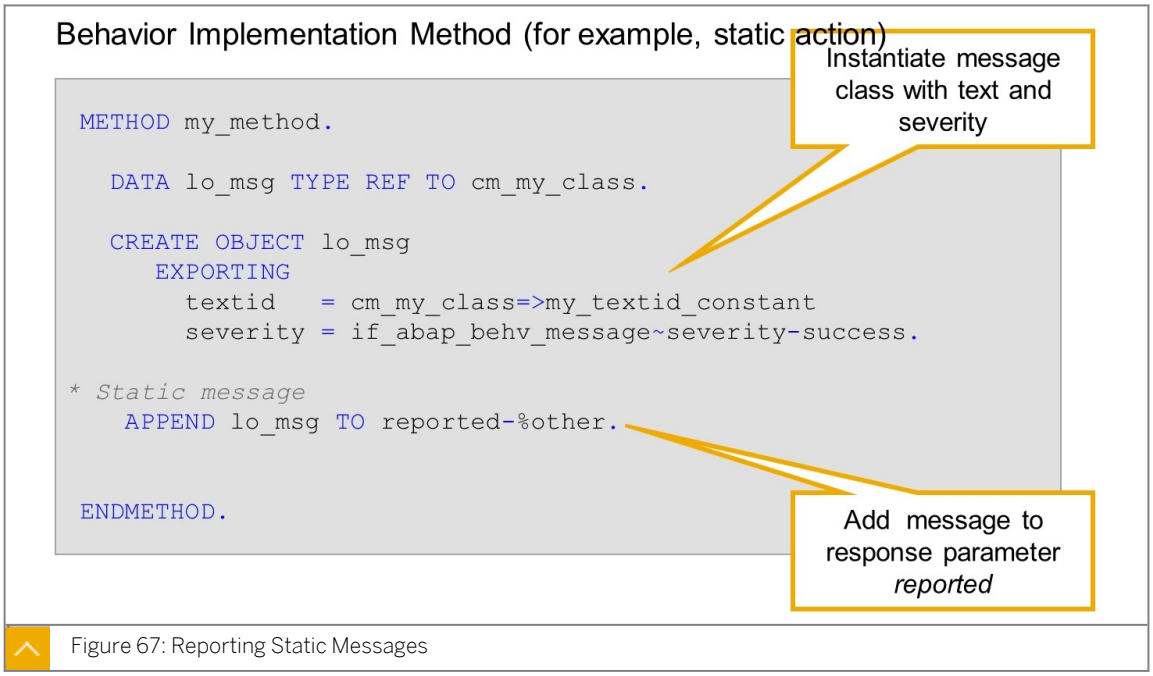

Figure 67: Reporting Static Messages

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

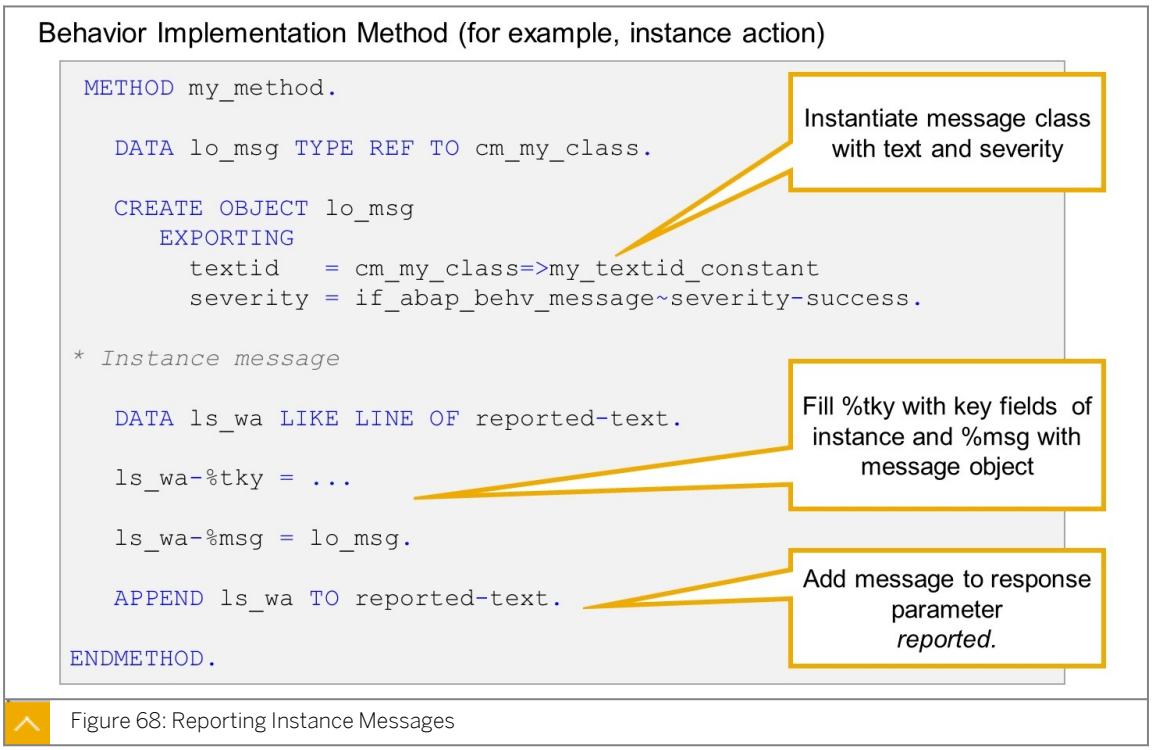

Figure 68: Reporting Instance Messages

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

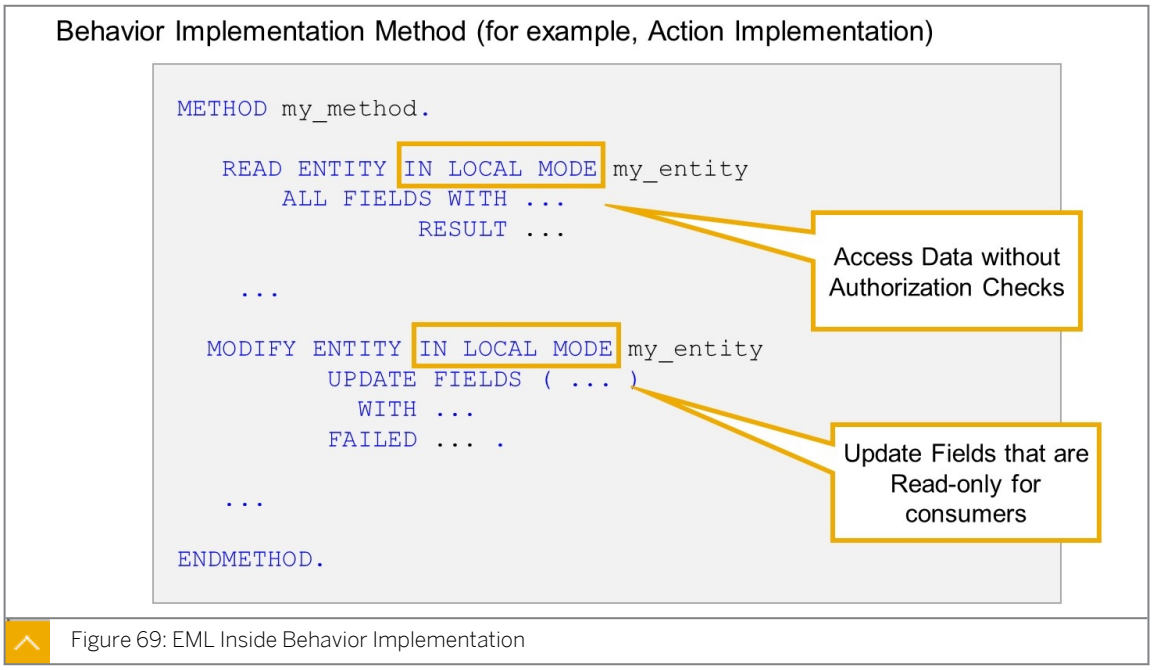

Figure 69: EML Inside Behavior Implementation

To access a RAP BOs data from inside its behavior implementation, EML statements are used. There is no mayor 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.

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

Unit 2
Lesson 5

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

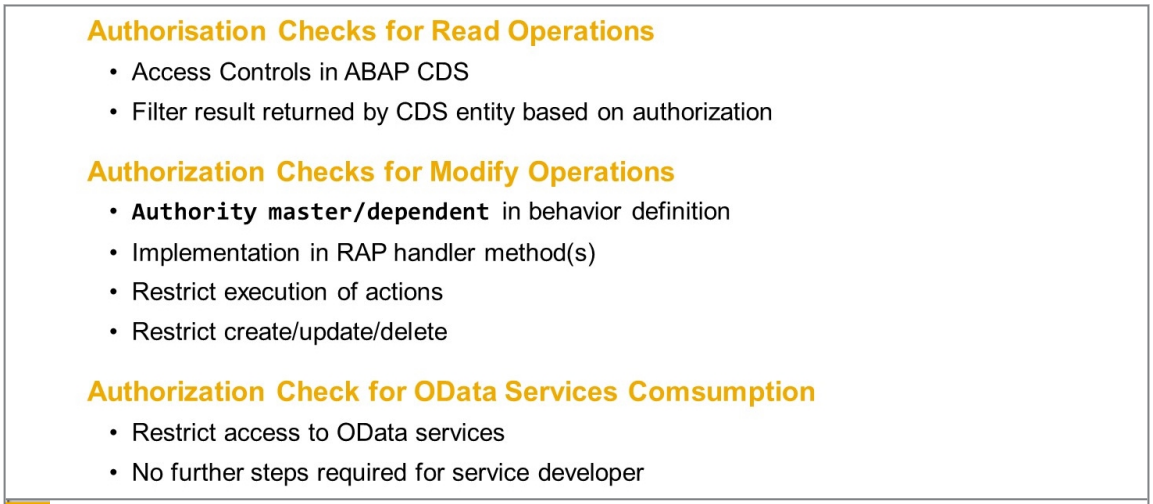

Figure 70: Authorization Control in RAP

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.

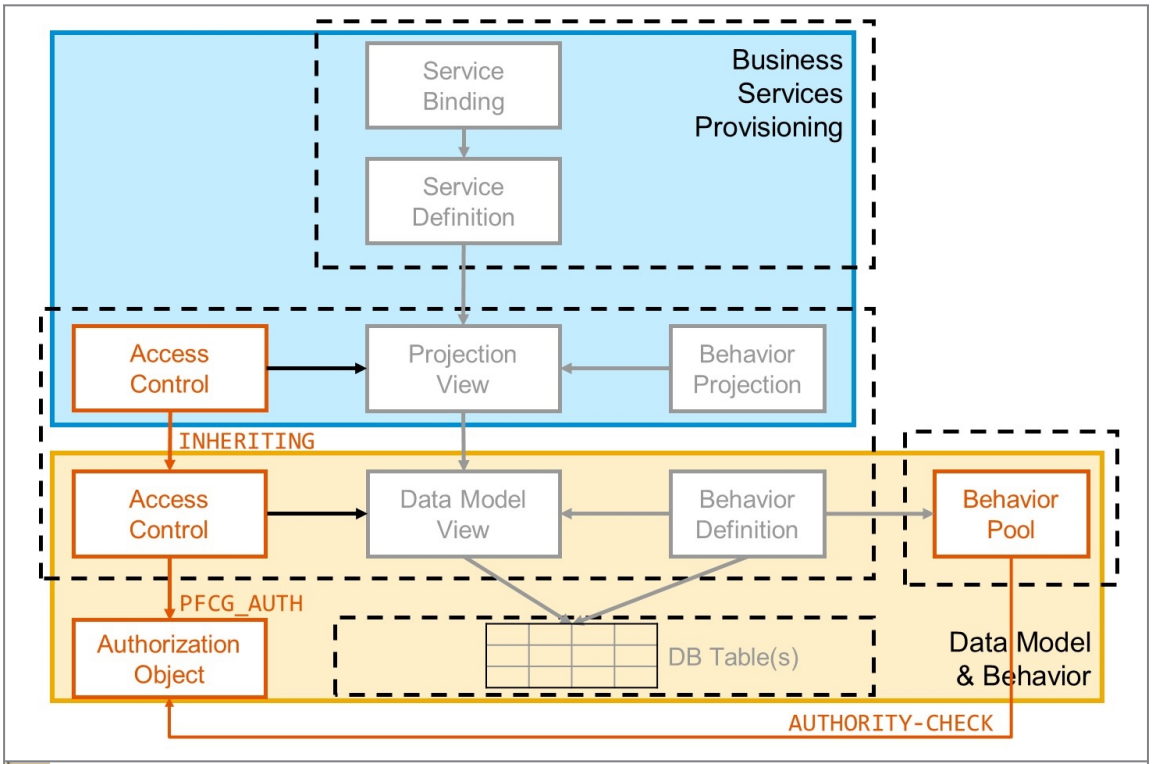

Figure 71: Overview of RAP Authorization Control

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

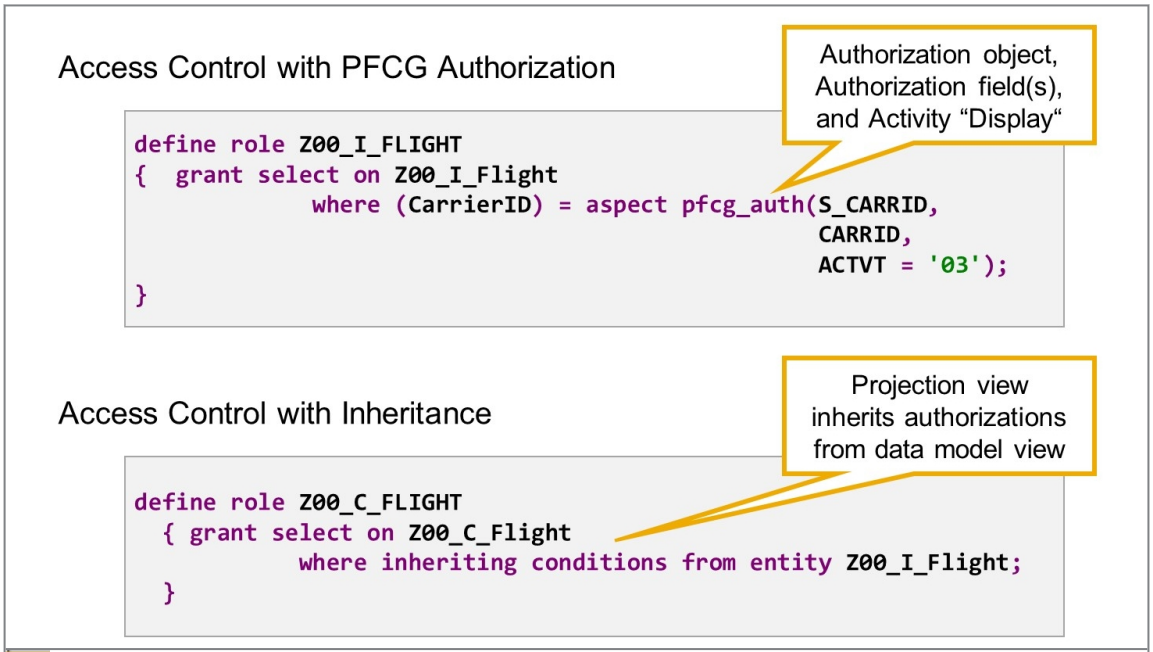

Figure 72: Creating Access Controls

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.

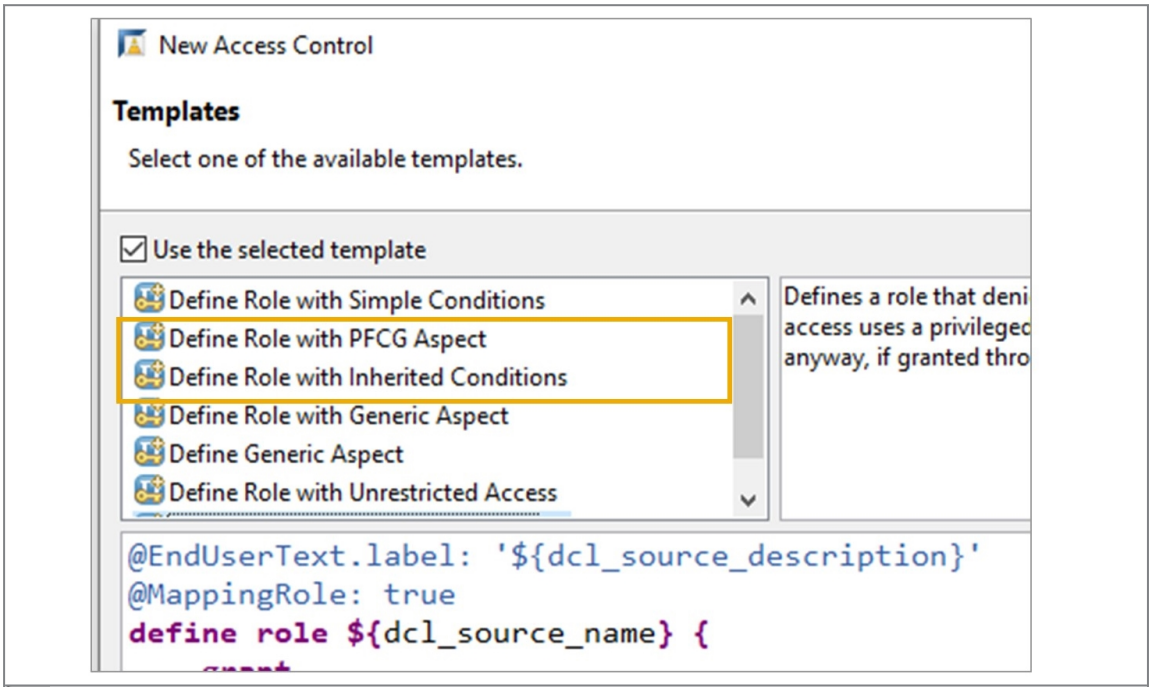

Figure 73: Templates for Creating Access Controls

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

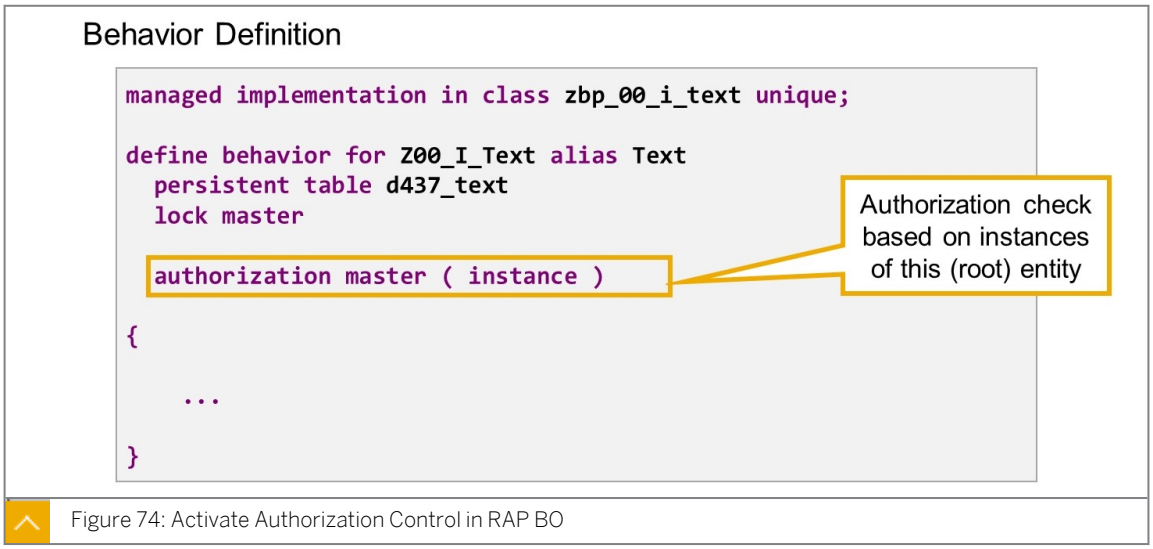

Figure 74: Activate Authorization Control in RAP BO

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.

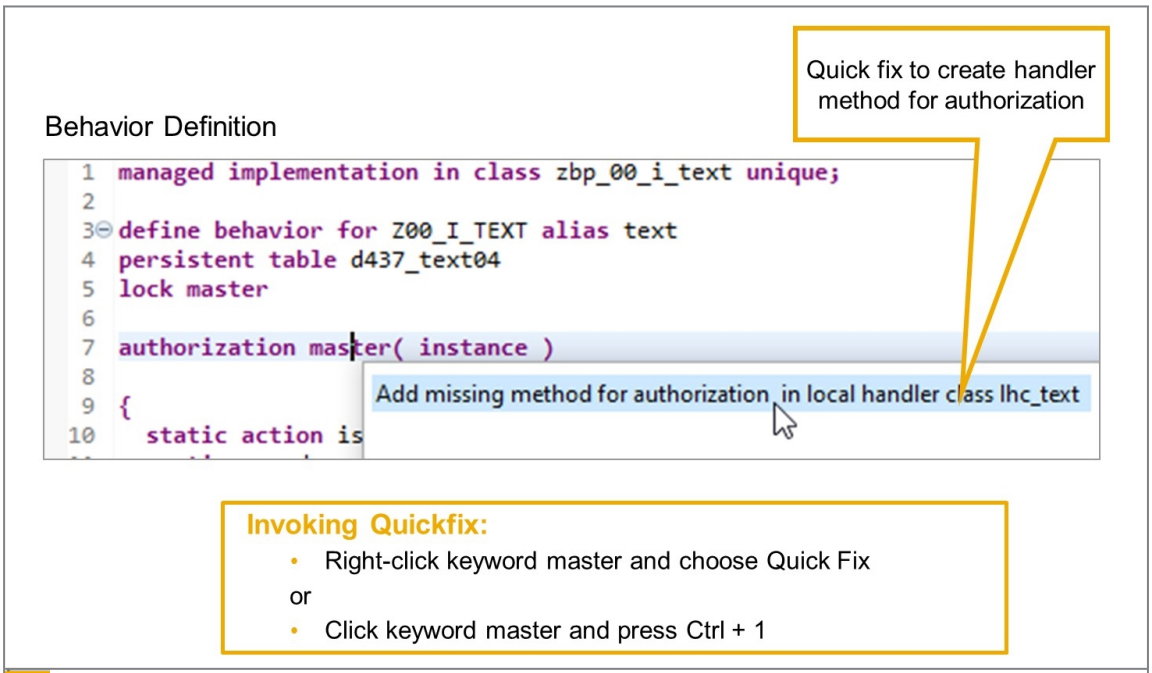

Figure 75: Creating the Authorization Handler Method

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.

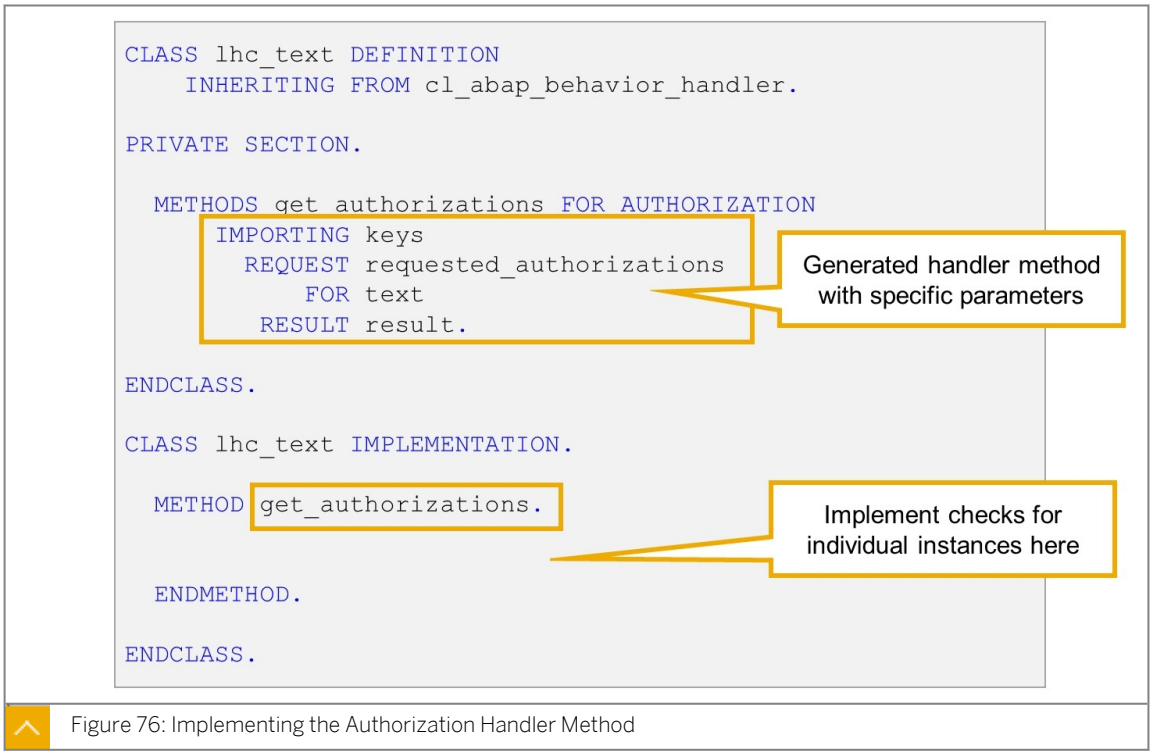

Figure 76: Implementing the Authorization Handler Method

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.

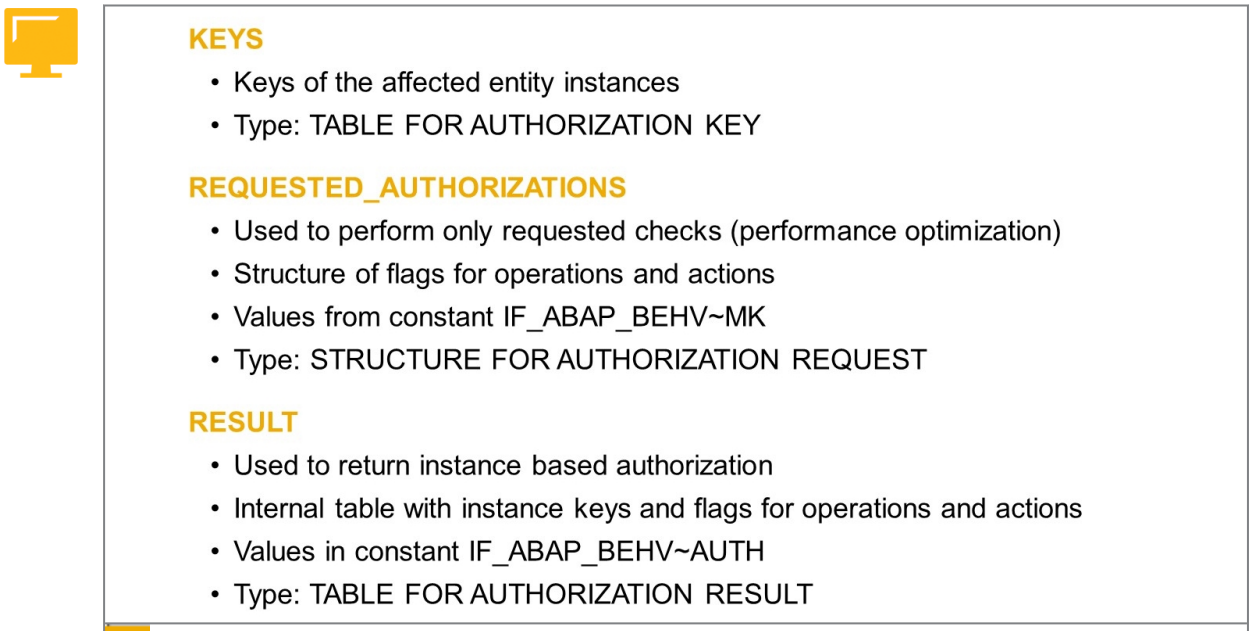

Figure 77: Parameters of the Authority Handler Method

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 I`F_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`

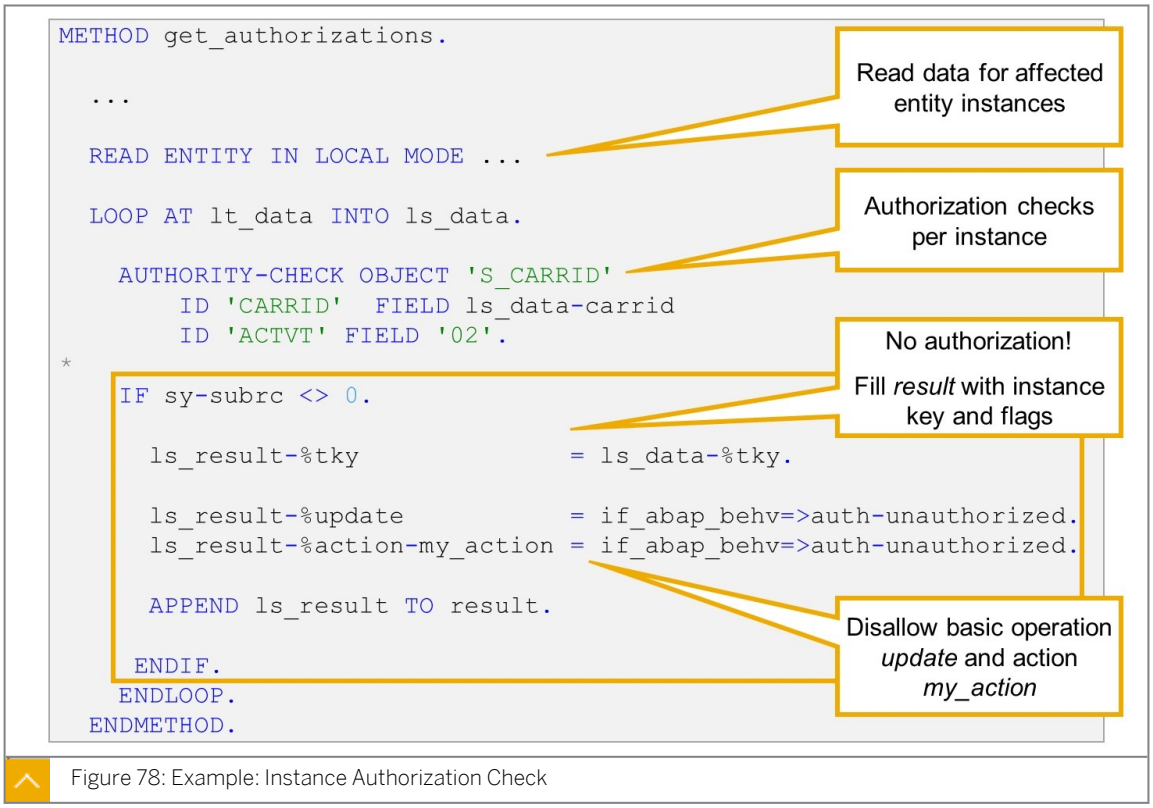

Figure 78: Example: Instance Authorization Check

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.

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

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

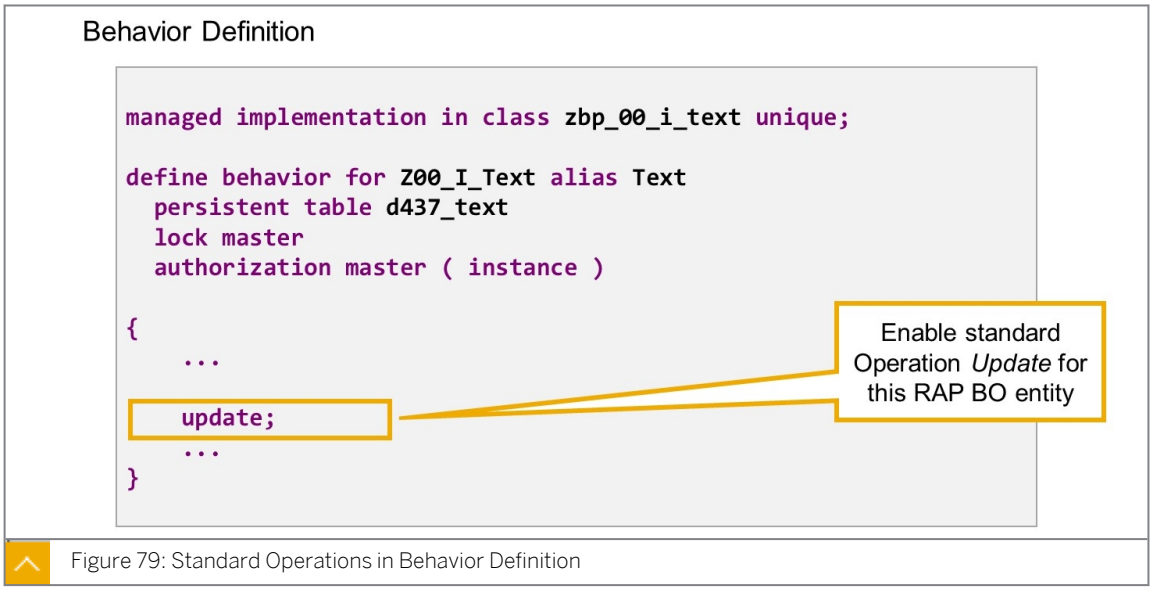

Figure 79: Standard Operations in Behavior Definition

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.

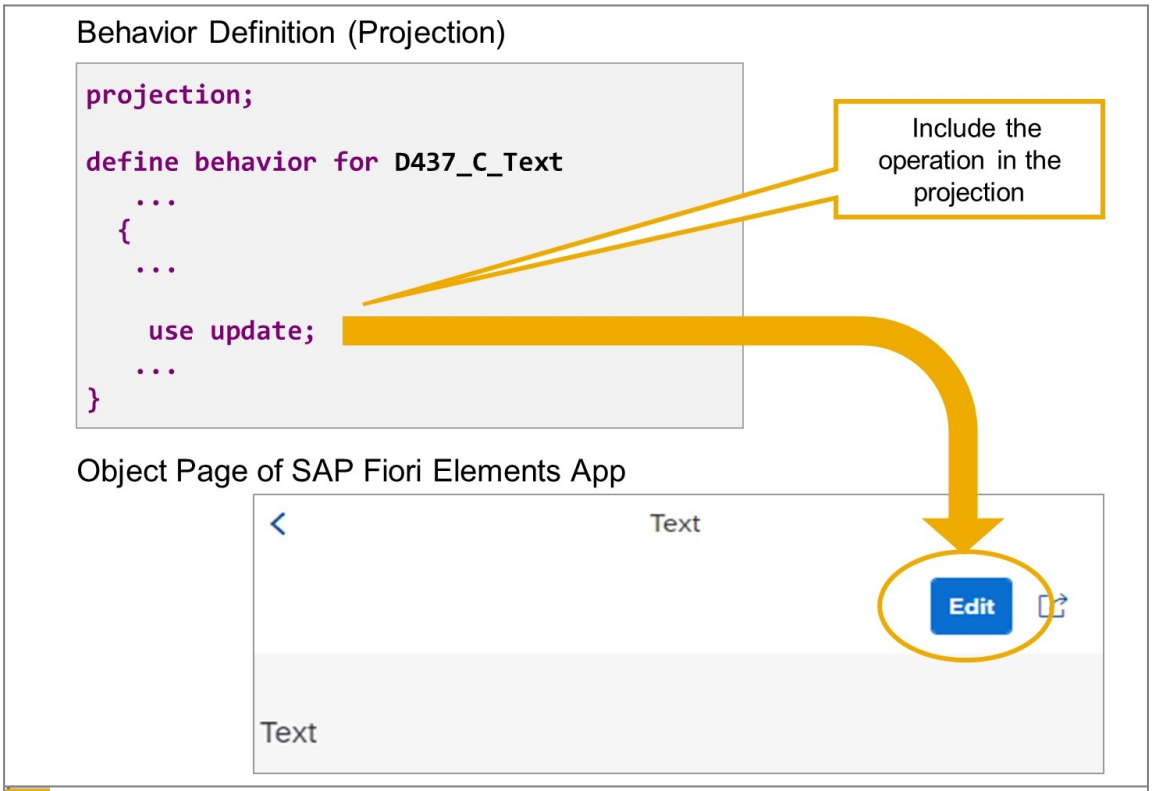

Figure 80: UPDATE in Behavior Projection

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

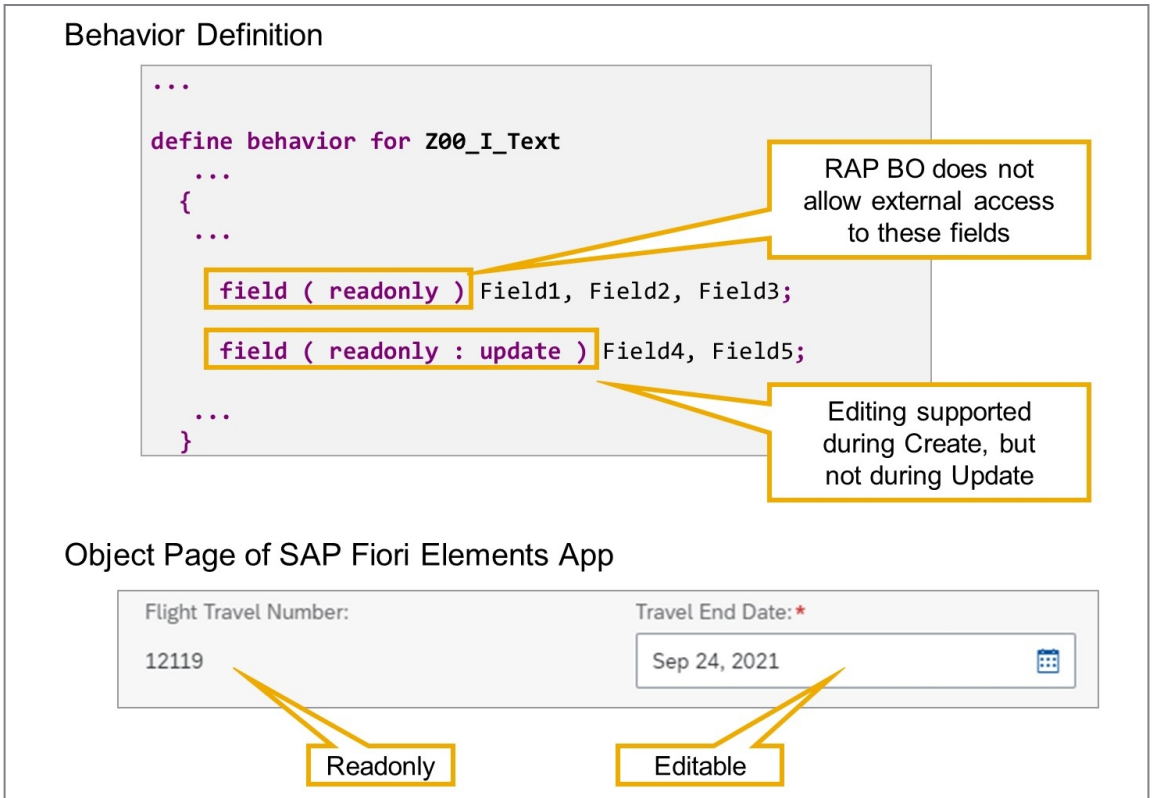

Figure 81: Read-only Input Fields

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.

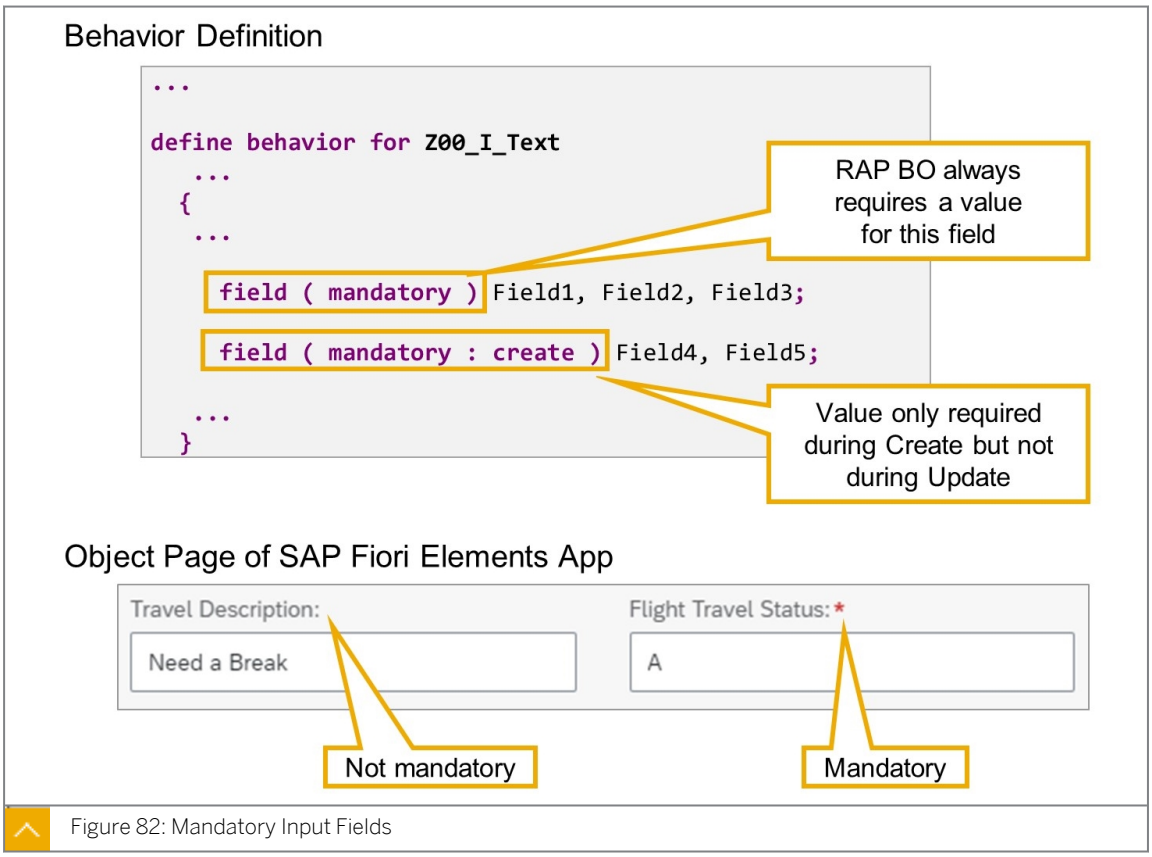

Figure 82: Mandatory Input Fields

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.

Note:
There is no runtime check for mandatory fields and no runtime error occurs if a mandatory field is not filled. If a runtime check is required, the application developer should implement it using a validation on save.

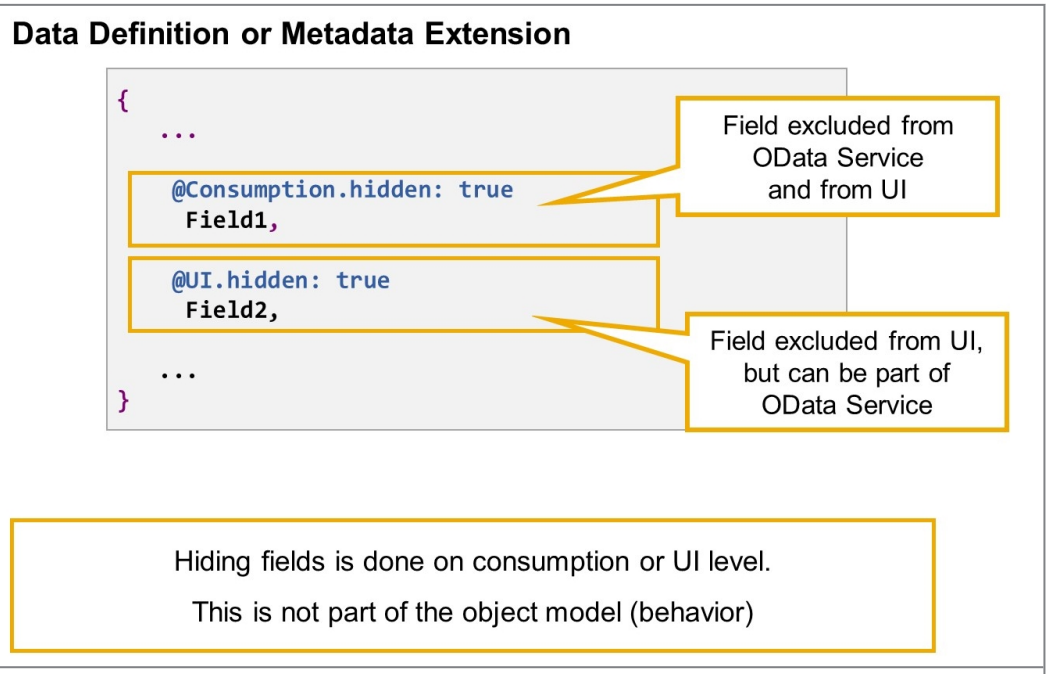

Figure 83: Hidden Fields

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

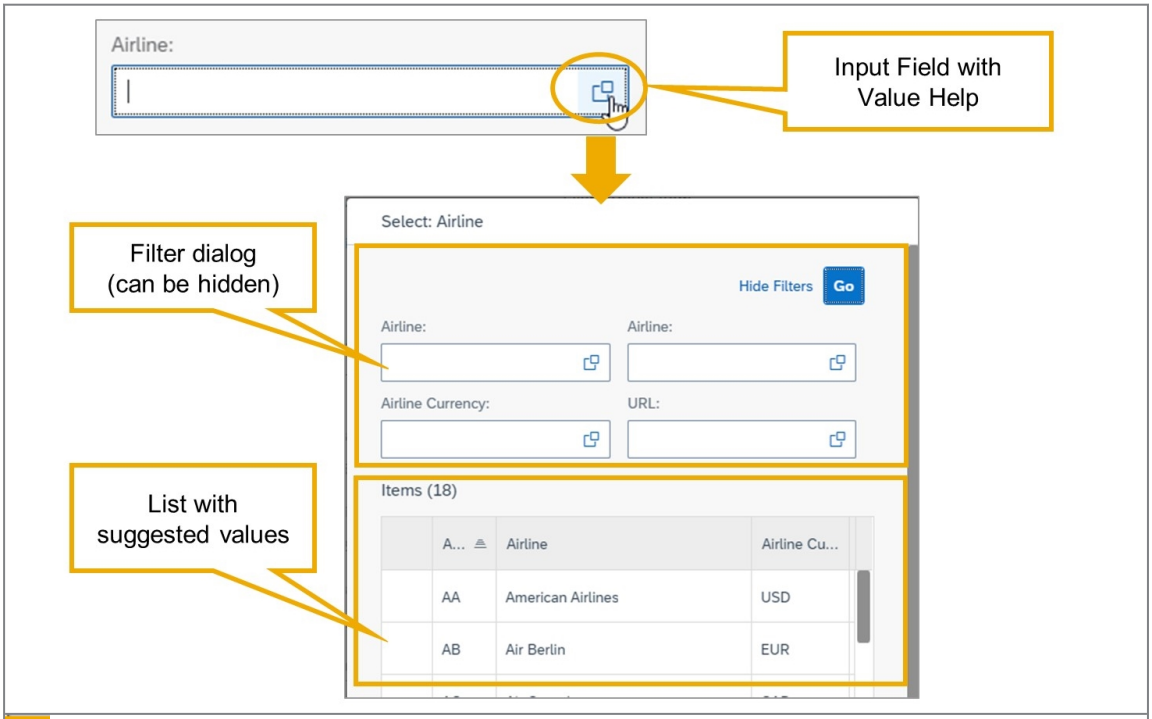

Figure 84: Value-Help in SAP Fiori App

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.

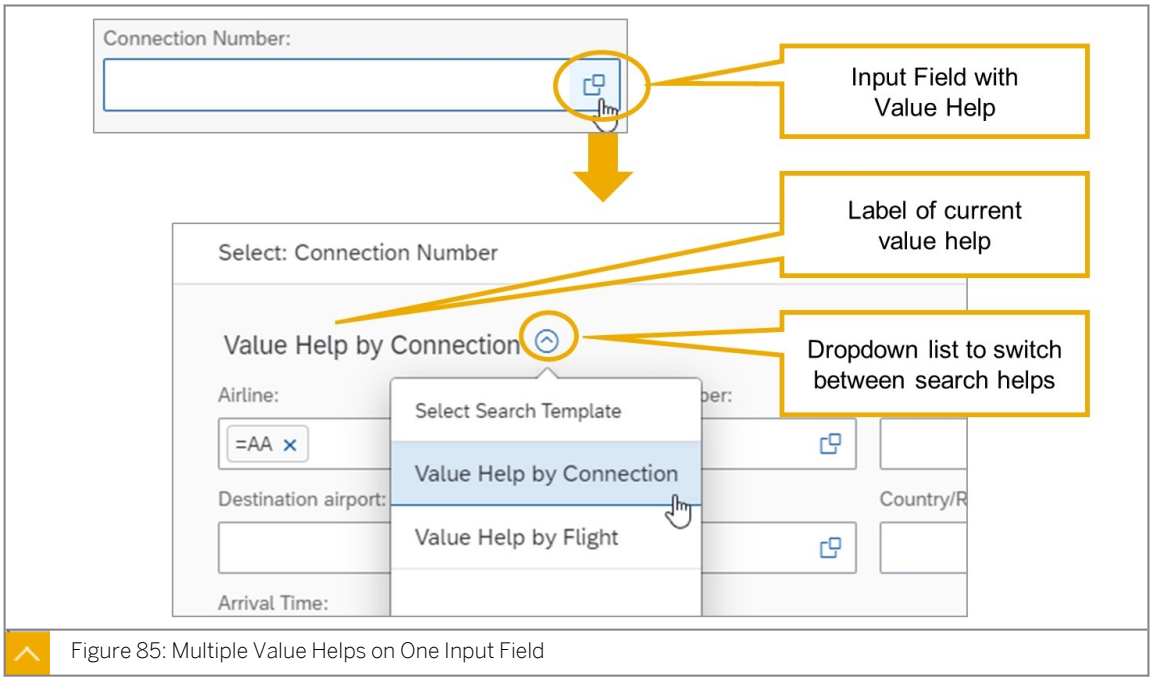

Figure 85: Multiple Value Helps on One Input Field

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.

Note:
This is similar to collective search help in classical ABAP dialog programming techniques (Dynpro and Web Dynpro), but that collective search helps used a tabstrip for visualization instead of a dropdown list.

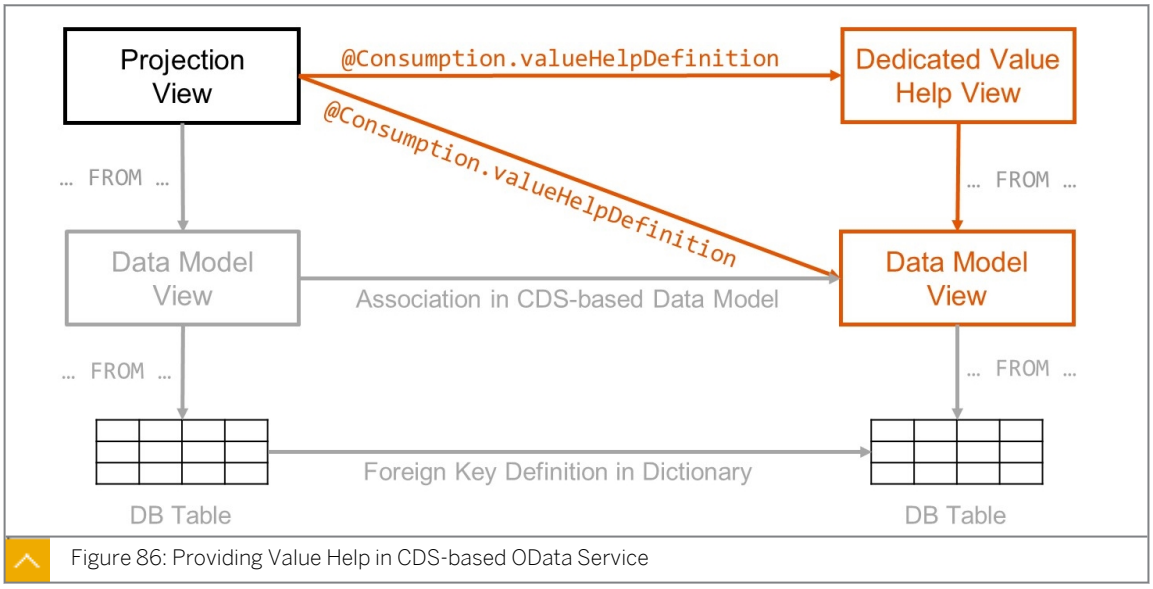

Figure 86: Providing Value Help in CDS-based OData Service

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.

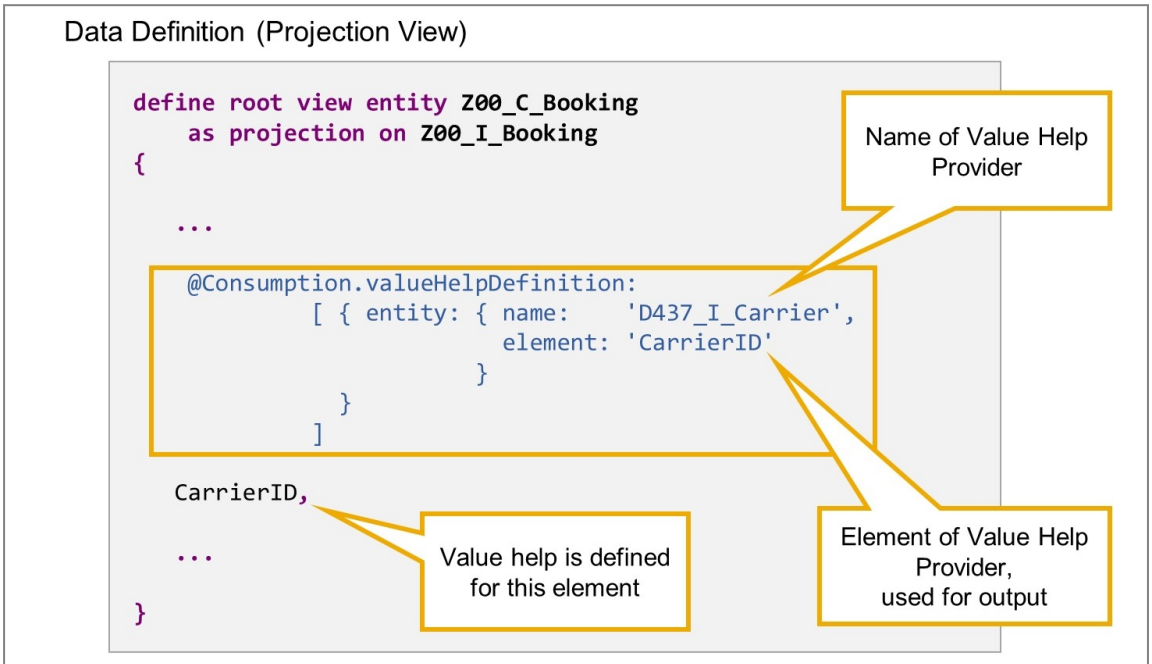

Figure 87: Simple Value Help

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.

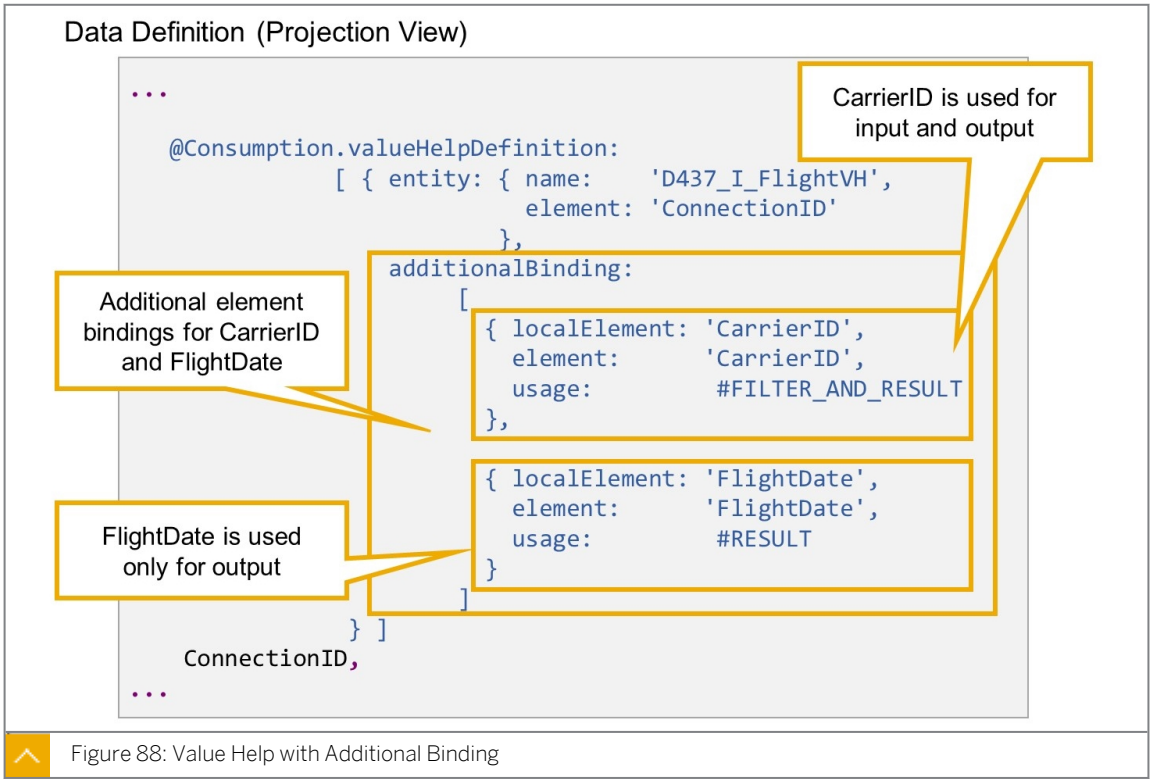

Figure 88: Value Help with Additional Binding

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 in 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.

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

Unit 3
Lesson 2

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

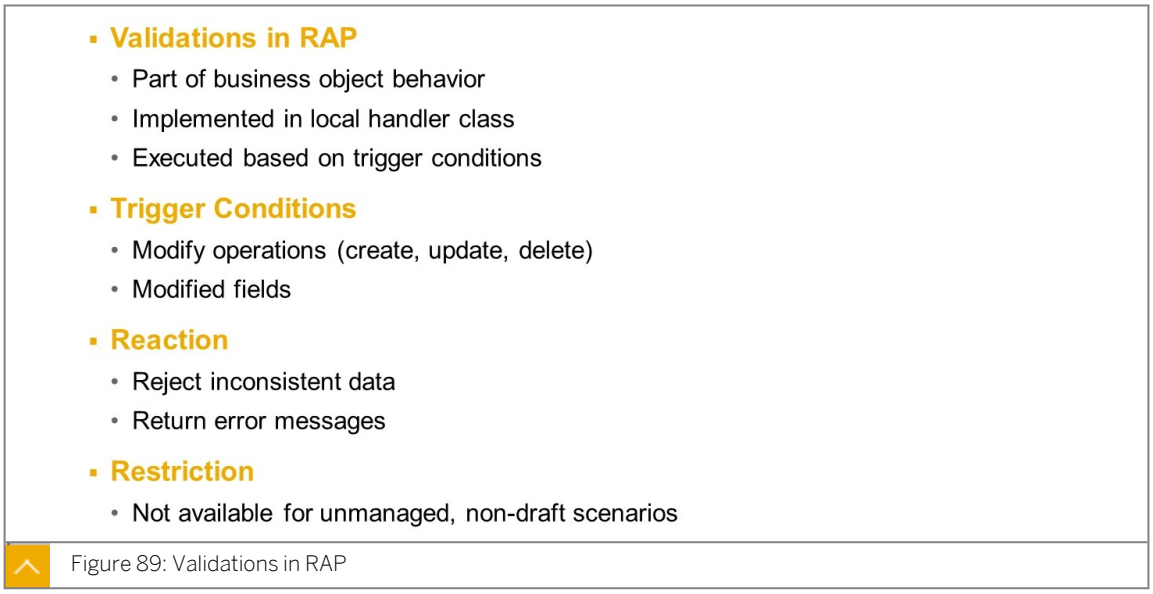

Figure 89: Validations in RAP

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.

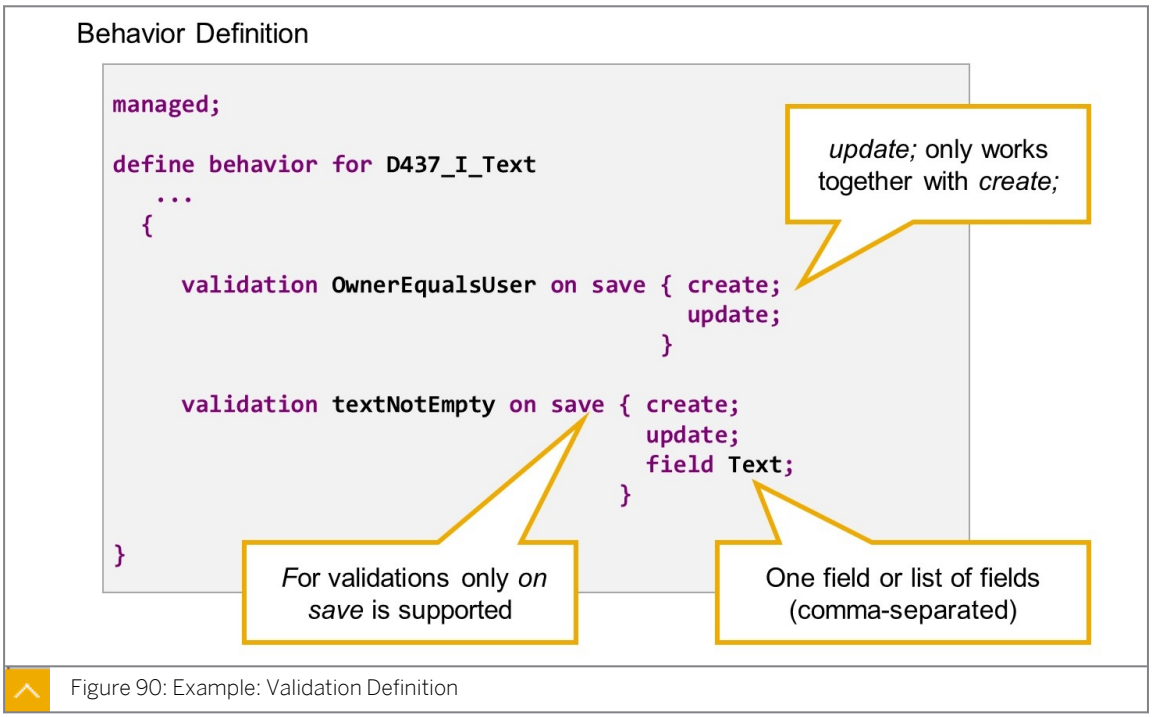

Figure 90: Example: Validation Definition

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

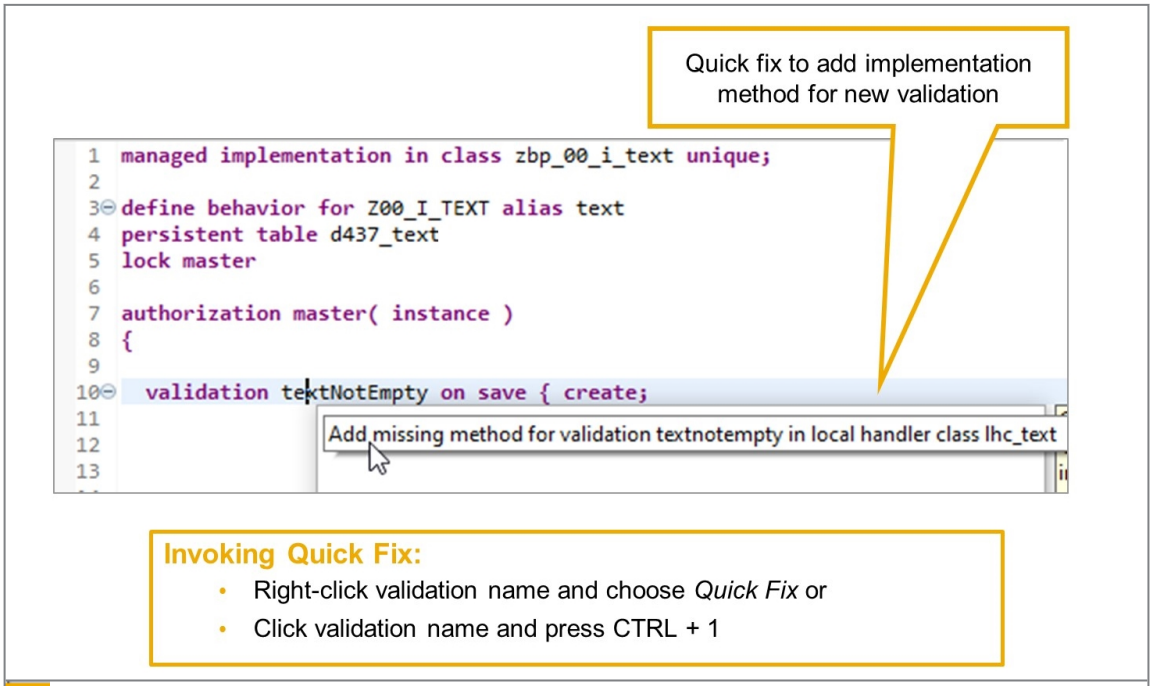

Figure 91: Creating the Validation Handler Method

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.

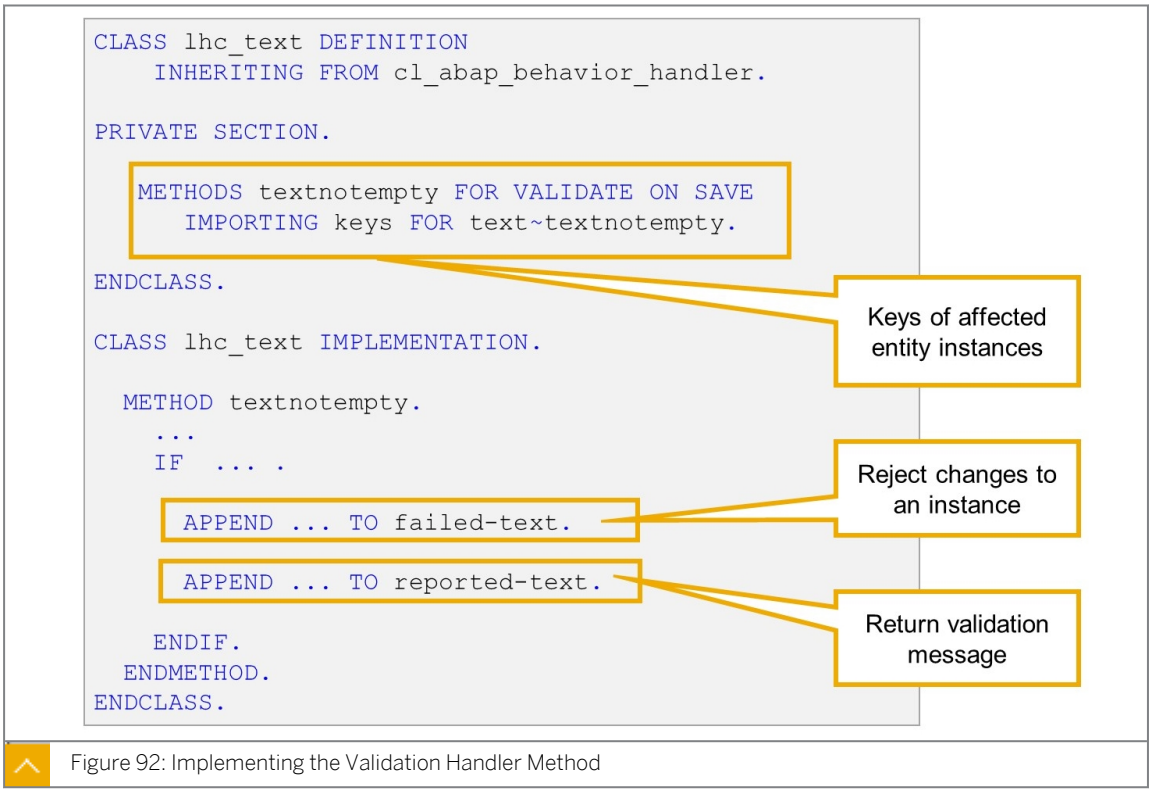

Figure 92: Implementing the Validation Handler Method

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

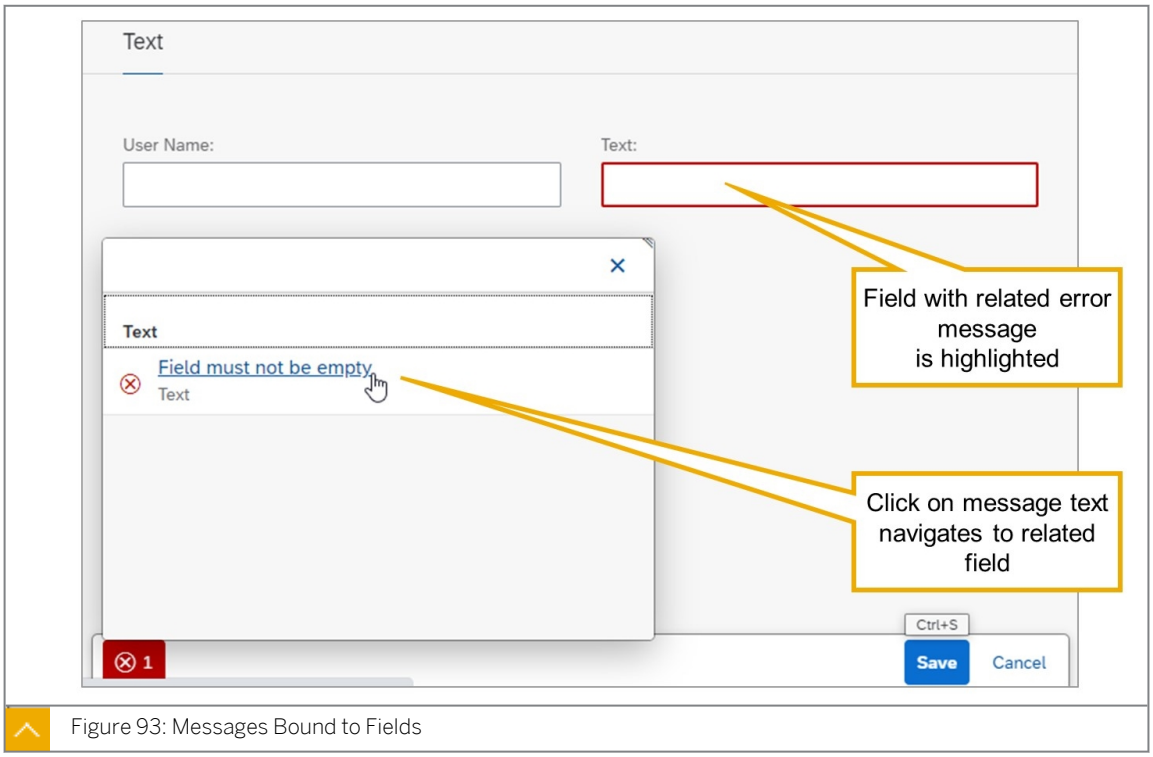

Figure 93: Messages Bound to Fields

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.

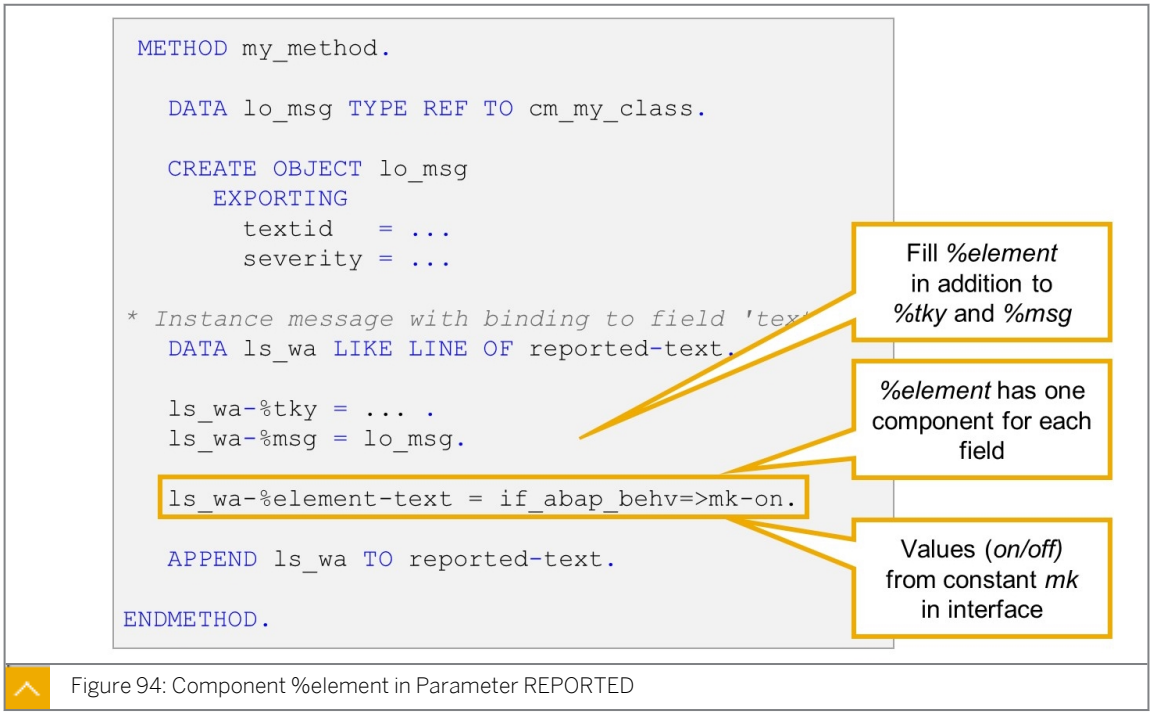

Figure 94: Component %element in Parameter REPORTED

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 `%tky` with the entity instance key.
4. Fill component `%msg` with a reference to the message object.
5. Il 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.

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.

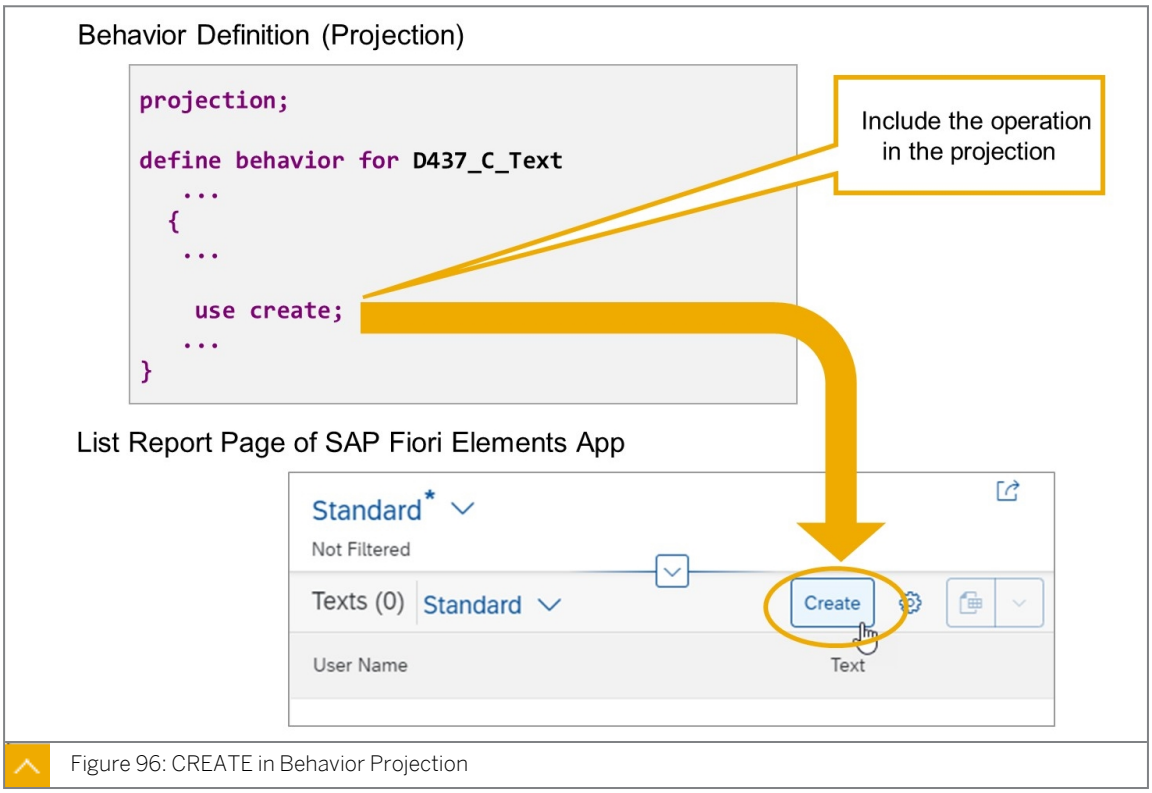

Figure 96: CREATE in Behavior Projection

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

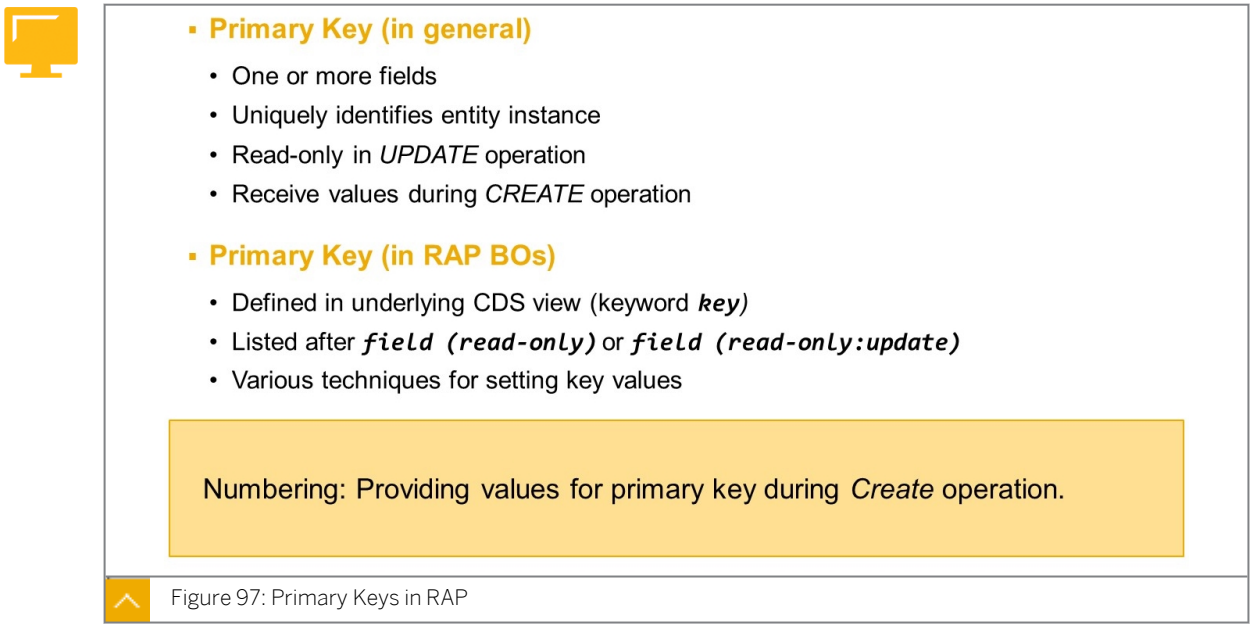

Figure 97: Primary Keys in RAP

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:

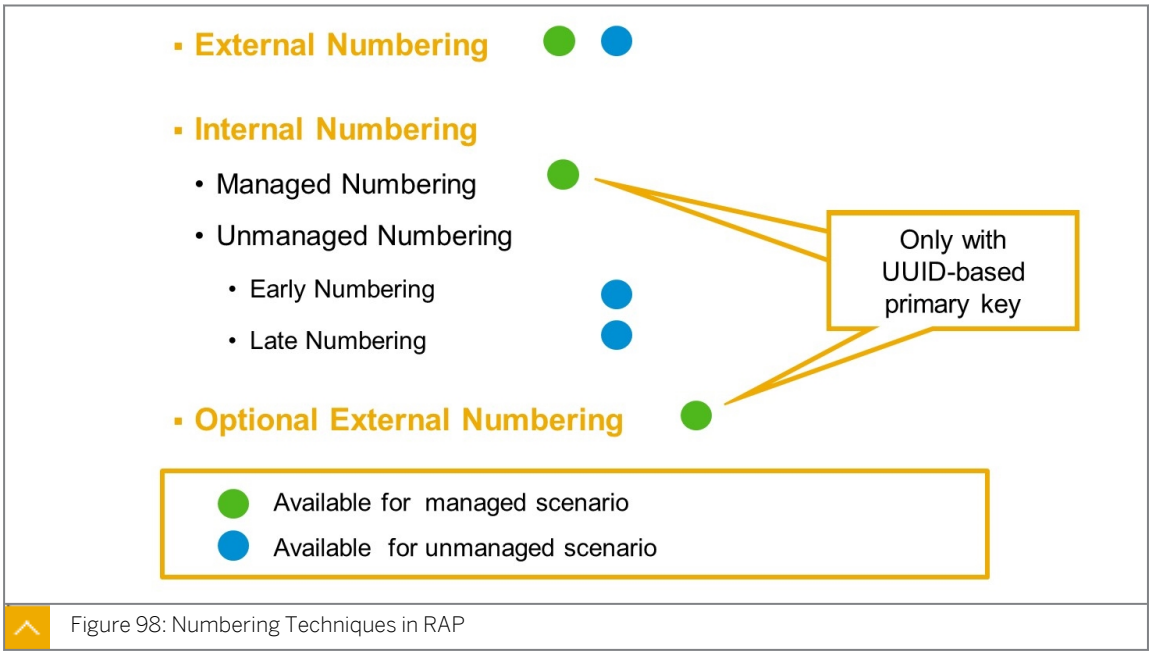

Figure 98: Numbering Techniques in RAP

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.

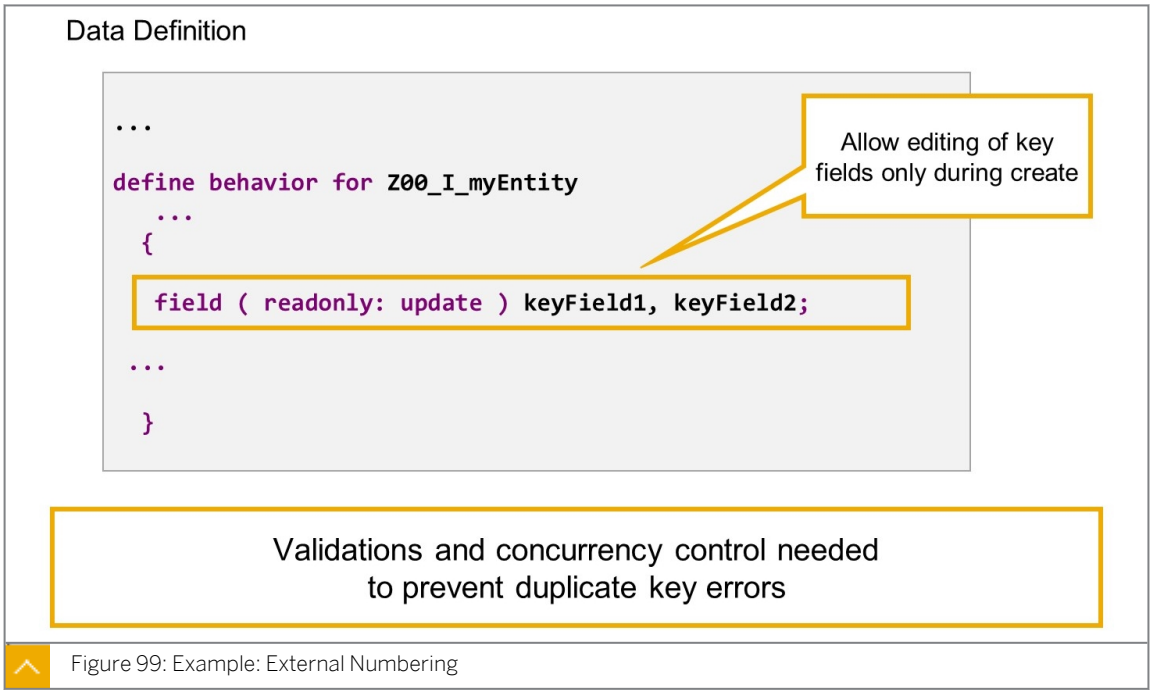

Figure 99: Example: External Numbering

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.

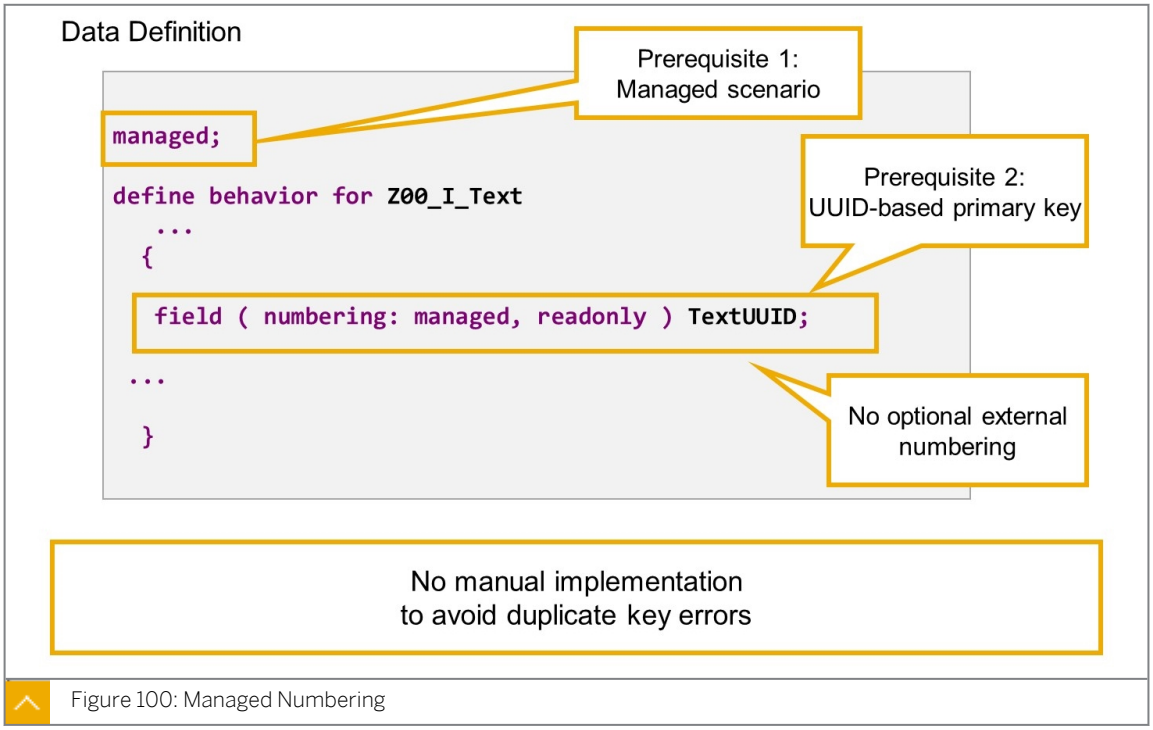

Figure 100: Managed Numbering

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.

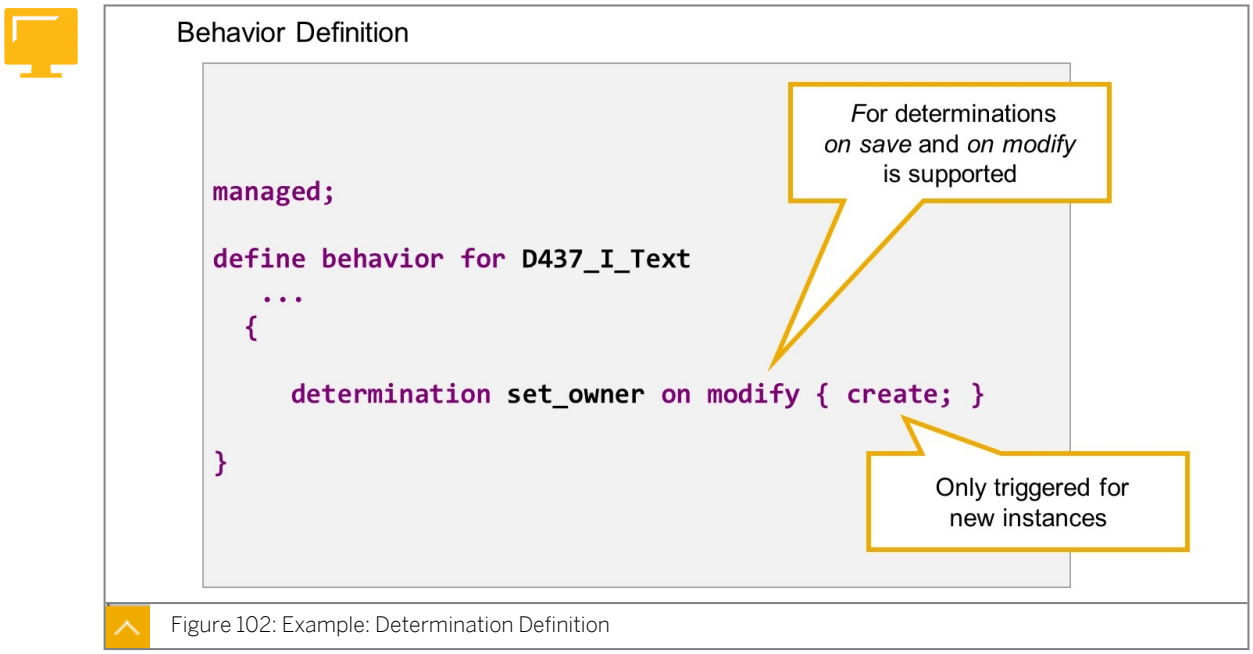

Figure 102: Example: Determination Definition

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 an 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

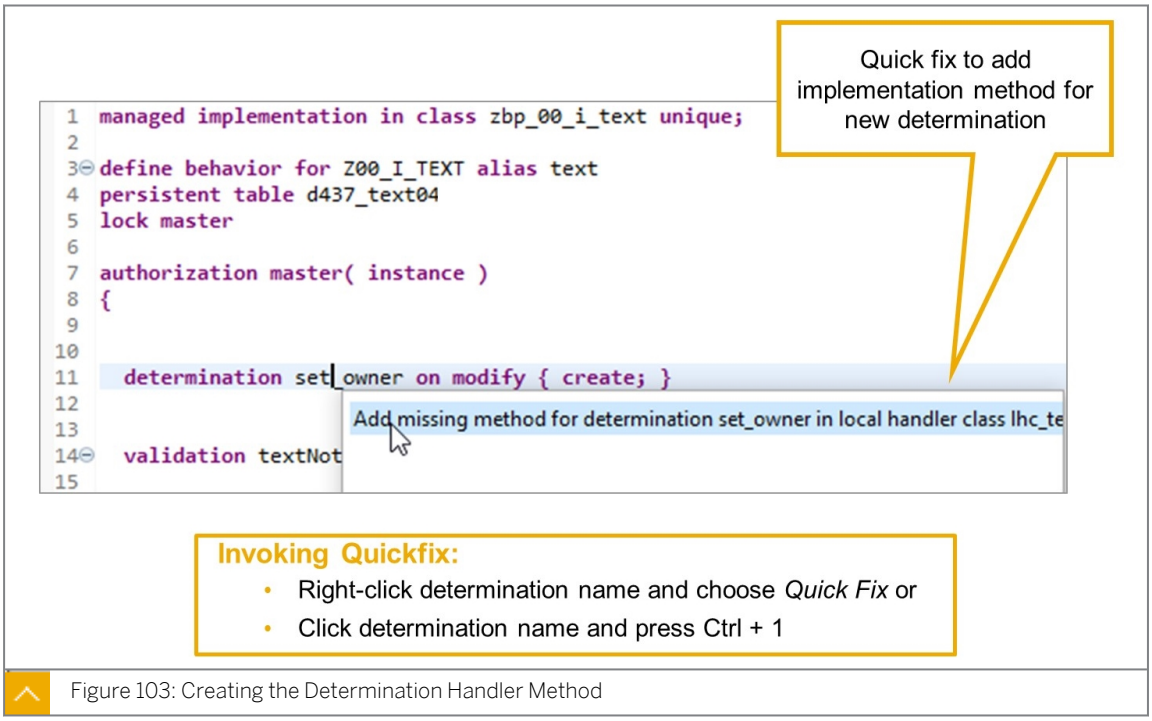

Figure 103: Creating the Determination Handler Method

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.

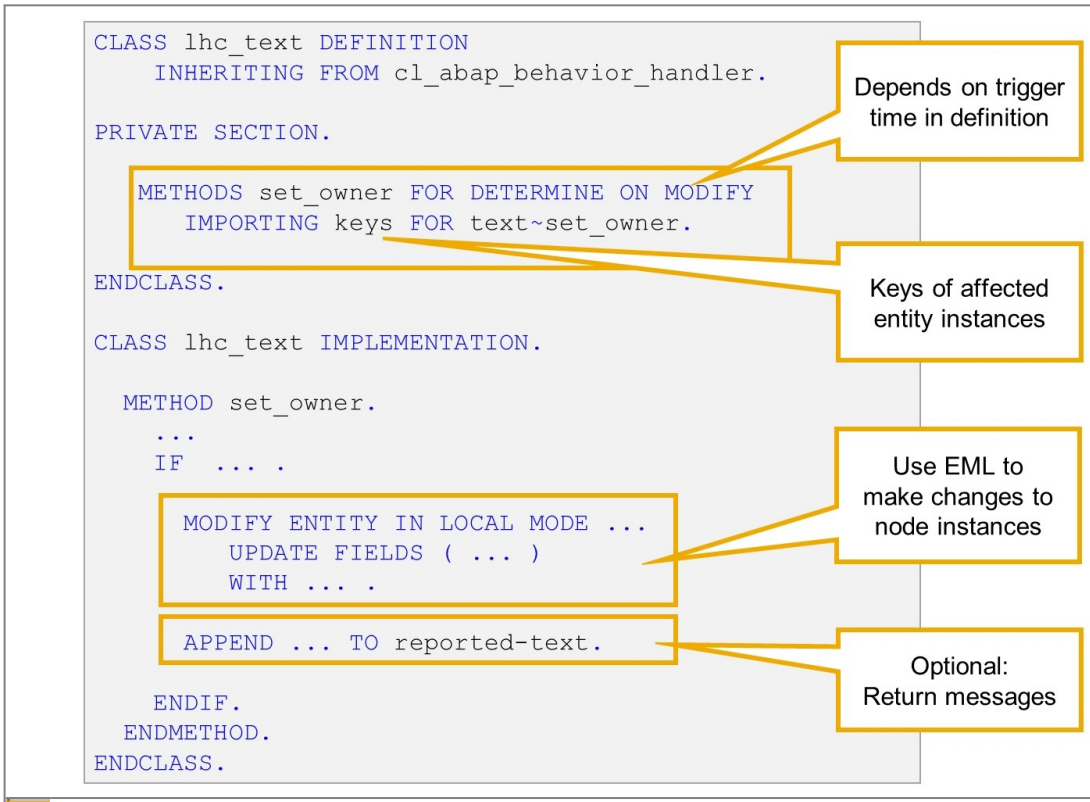

Figure 104: Implementing the Determination Handler Method

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.

LESSON SUMMARY

You should now be able to:

- Describe the numbering concepts in RAP
- Define and implement determinations

Unit 3
Lesson 4

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

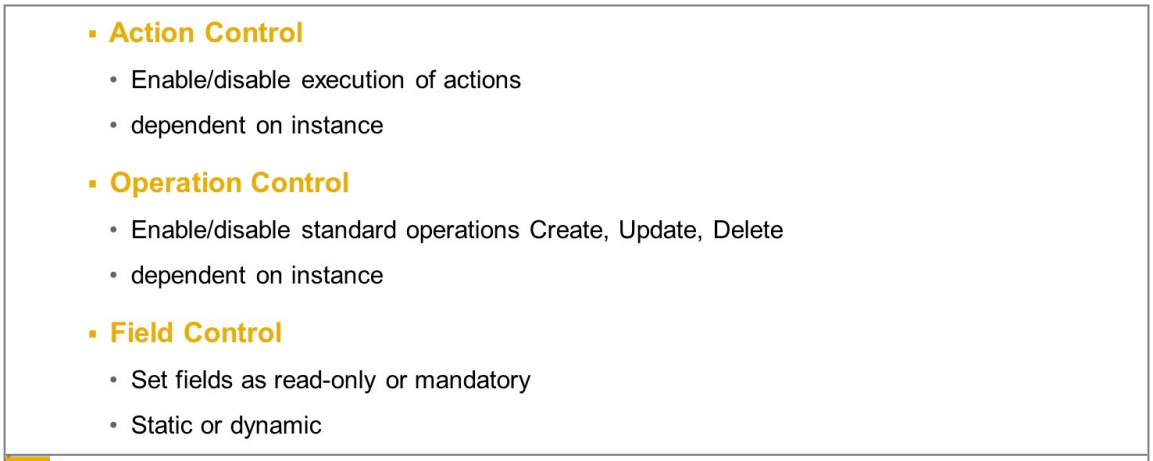

Figure 105: Dynamic Feature Control in RAP

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

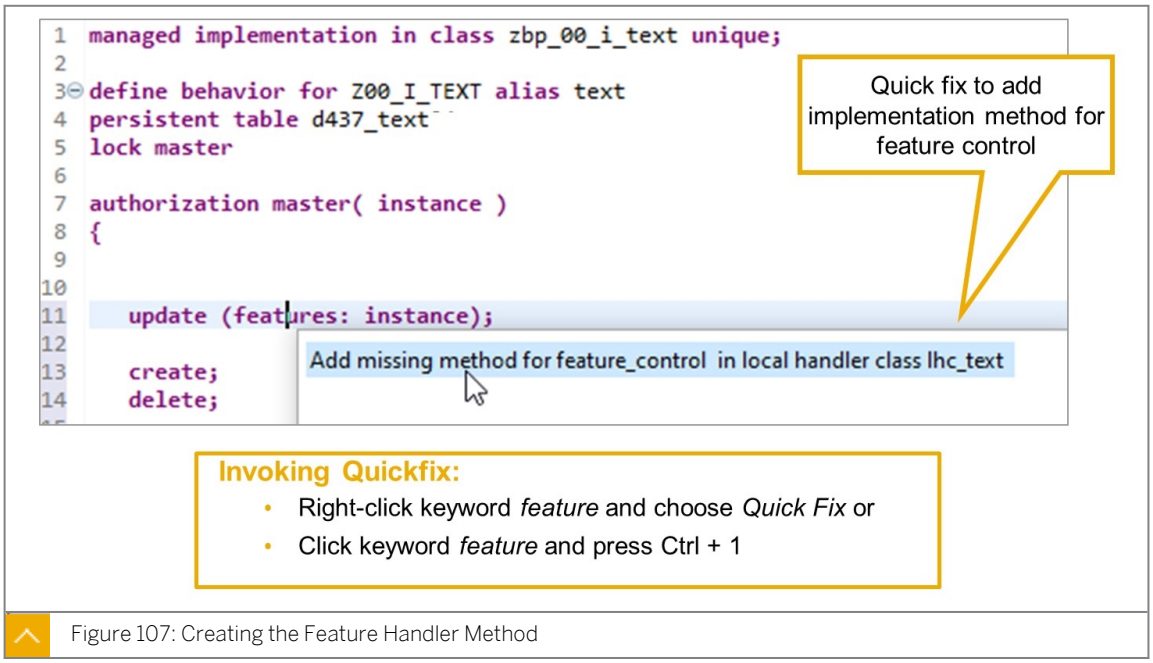

Figure 107: Creating the Feature Handler Method

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.

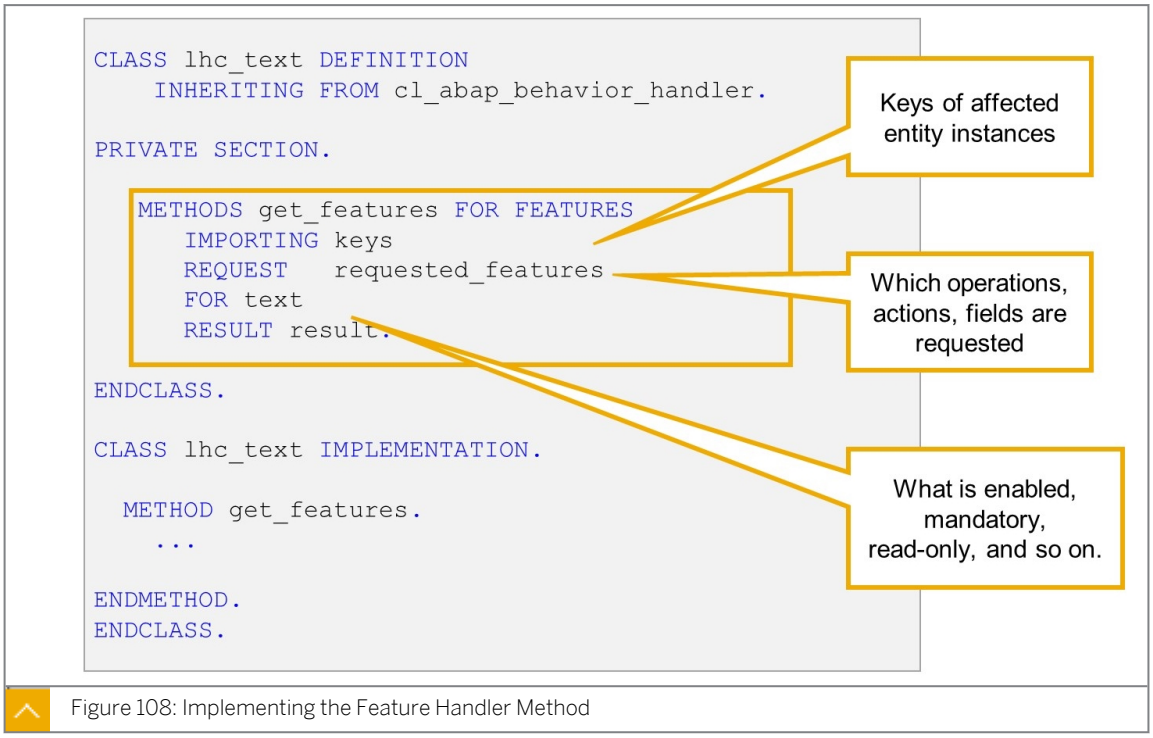

Figure 108: Implementing the Feature Handler Method

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

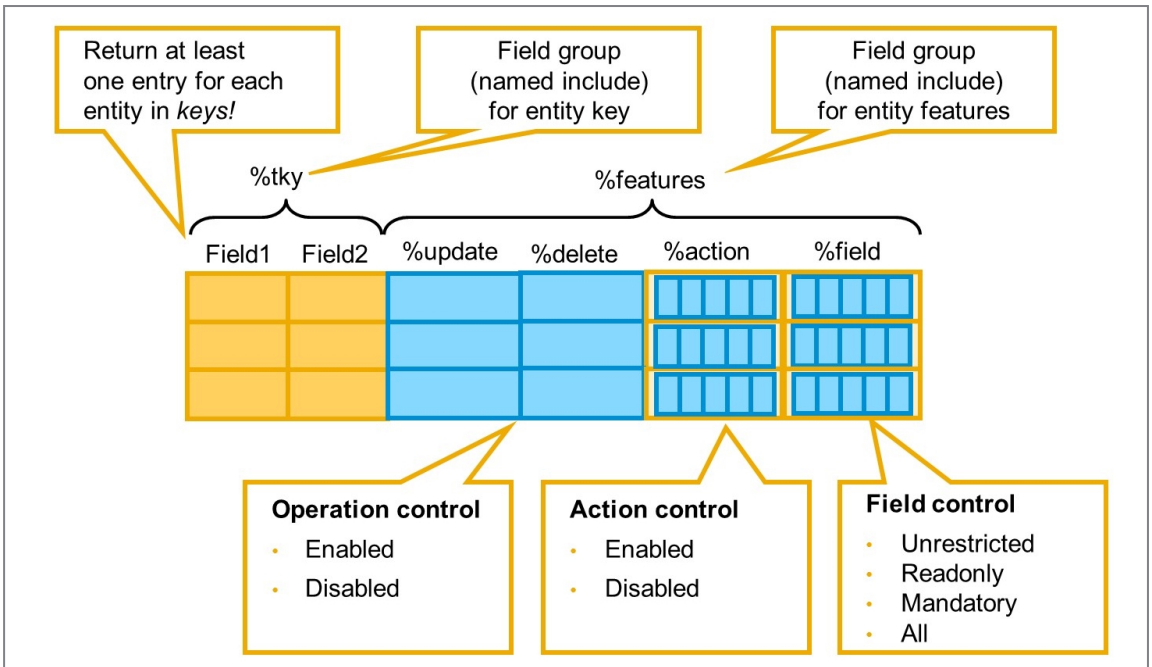

Figure 109: Result of Feature Handler Method

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.
  ENDLOOP.
ENDMETHOD.
```

Retrieve data for affected entity instances

Check if opertion should be allowed

Disable operation *update*

Fill *result* parameter

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.

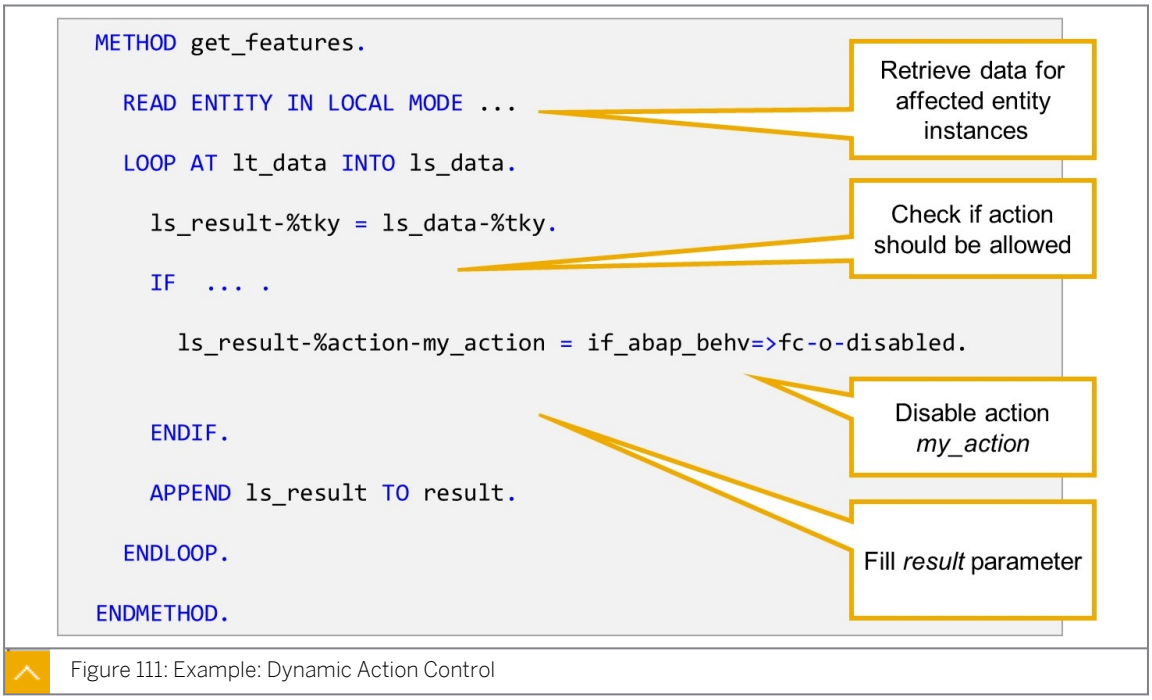

Figure 111: Example: Dynamic Action Control

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.

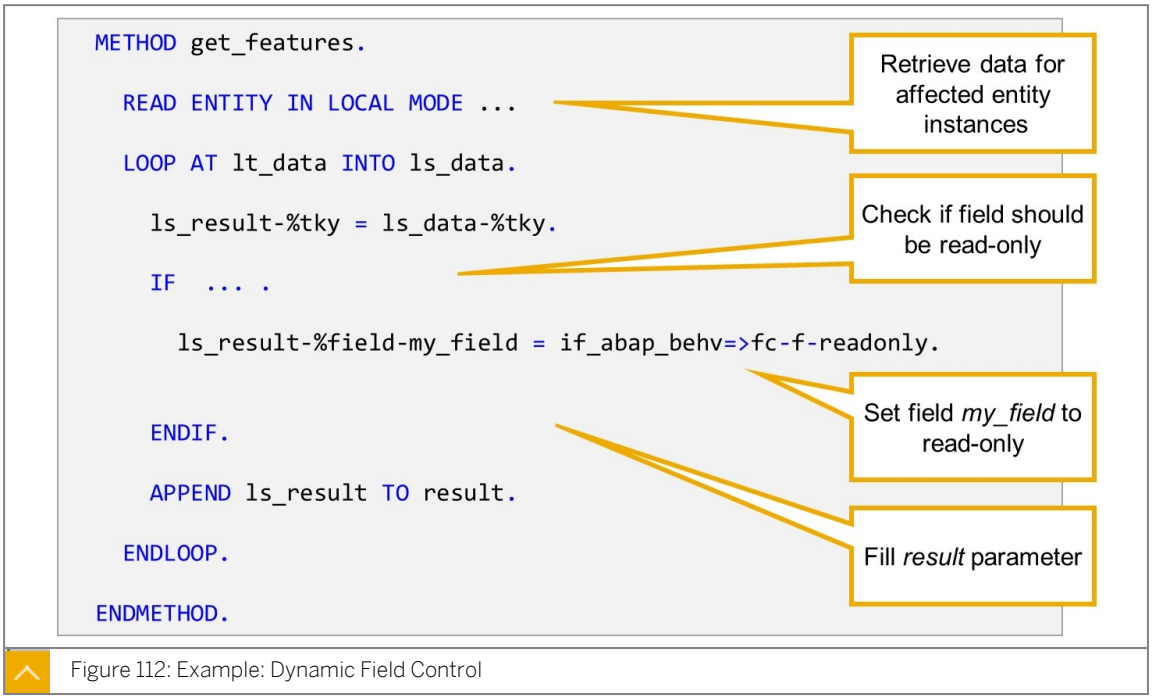

Figure 112: Example: Dynamic Field Control

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.

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

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

Unit 4
Lesson 1

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

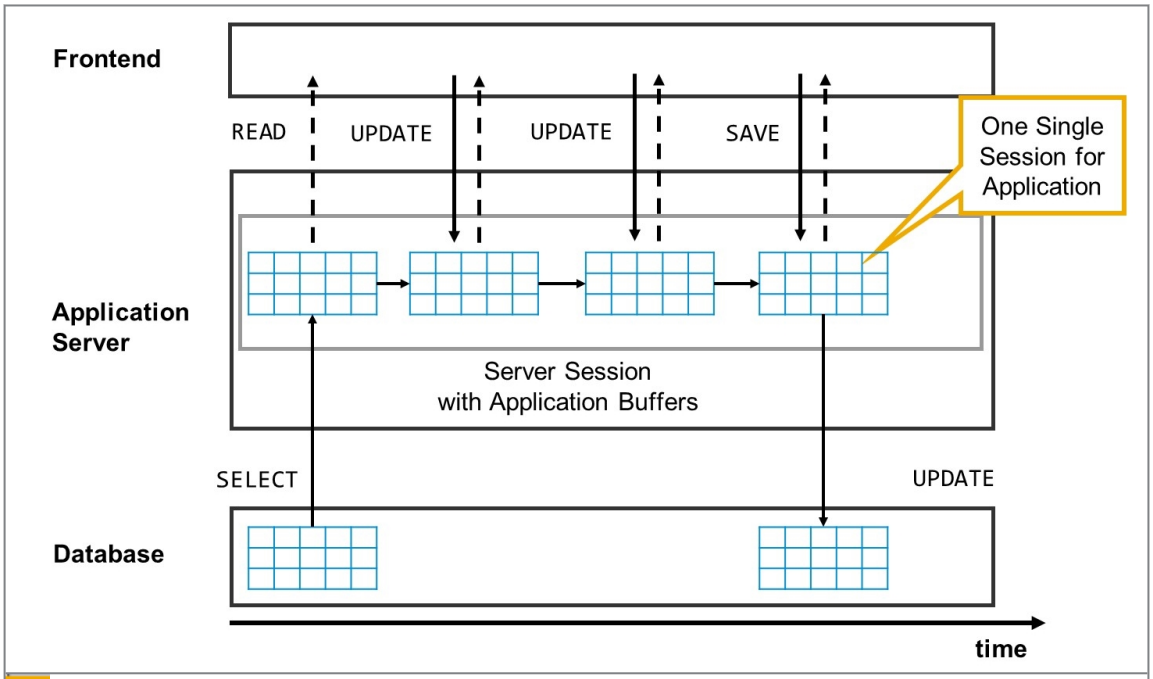

Figure 113: Stateful Technologies - One Server Session

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.

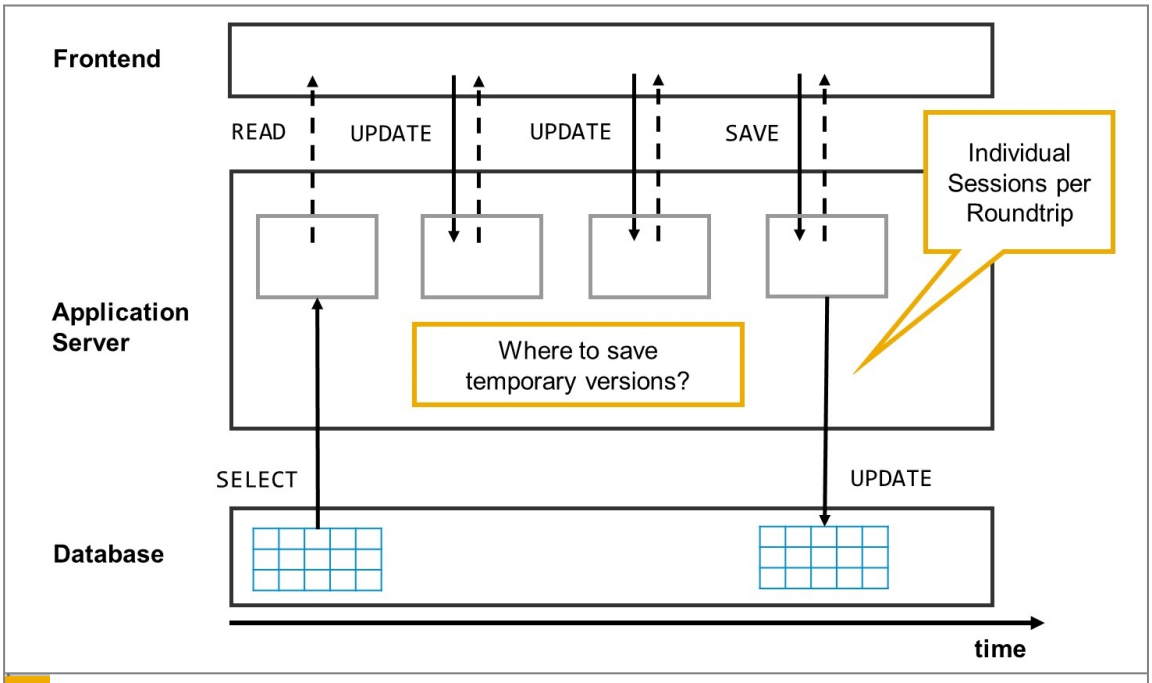

Figure 114: Stateless Technologies - Many Individual Sessions

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.

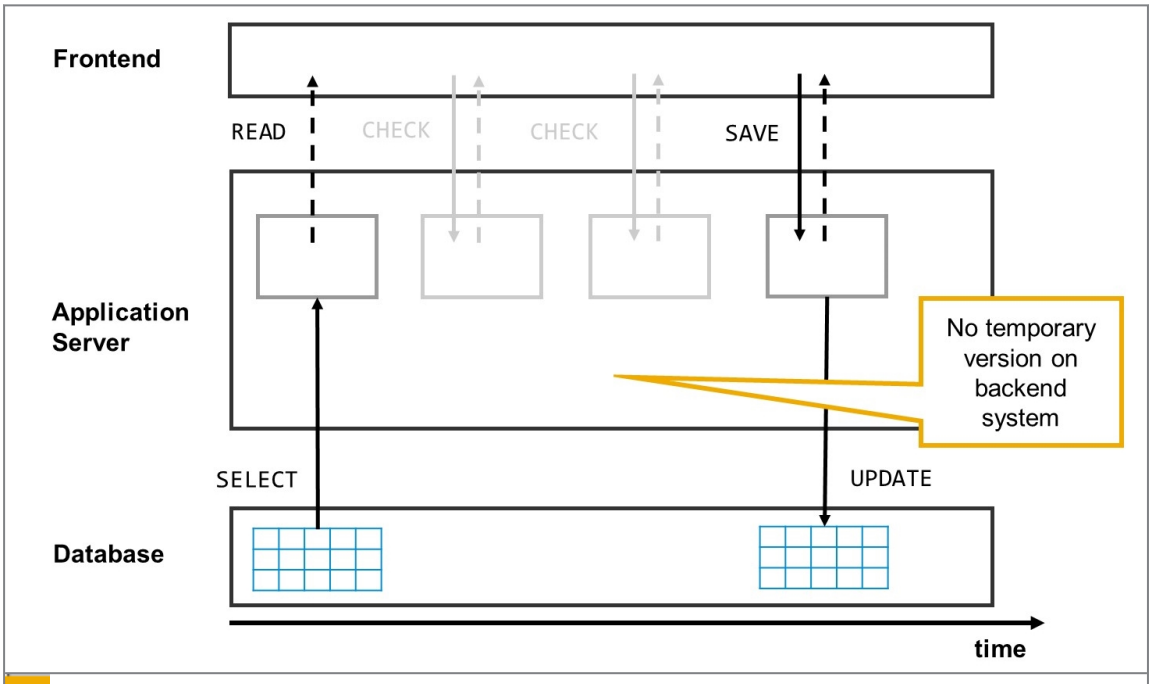

Figure 115: Stateless Technologies - Non-draft Scenario

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.

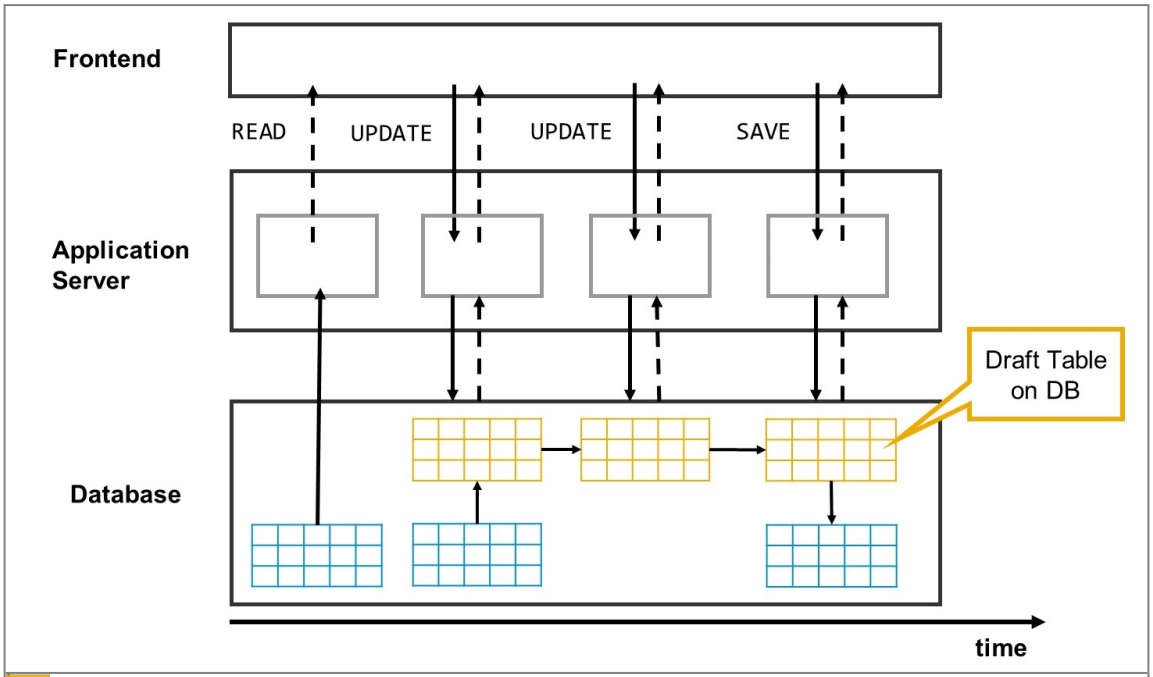

Figure 116: Solution: Draft Persistence on DB

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.

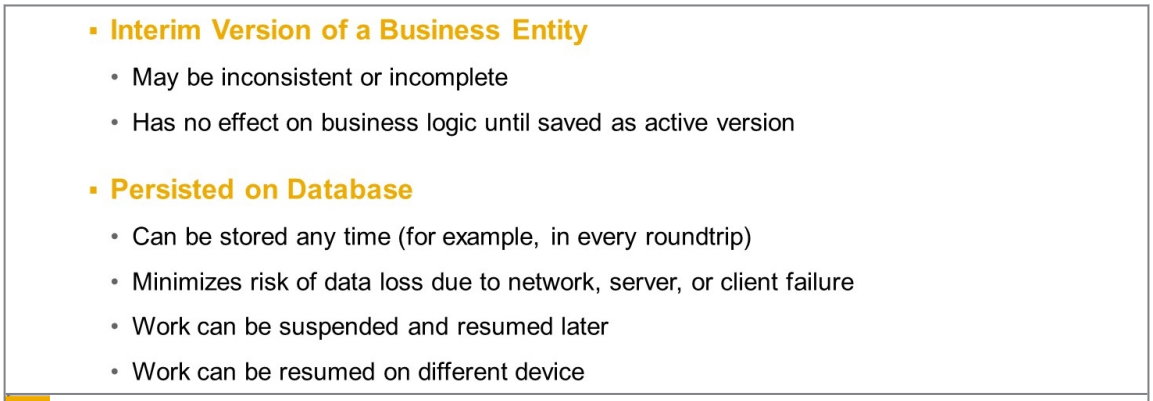

Figure 117: What is a Draft?

Draft Enabled RAP Business Objects

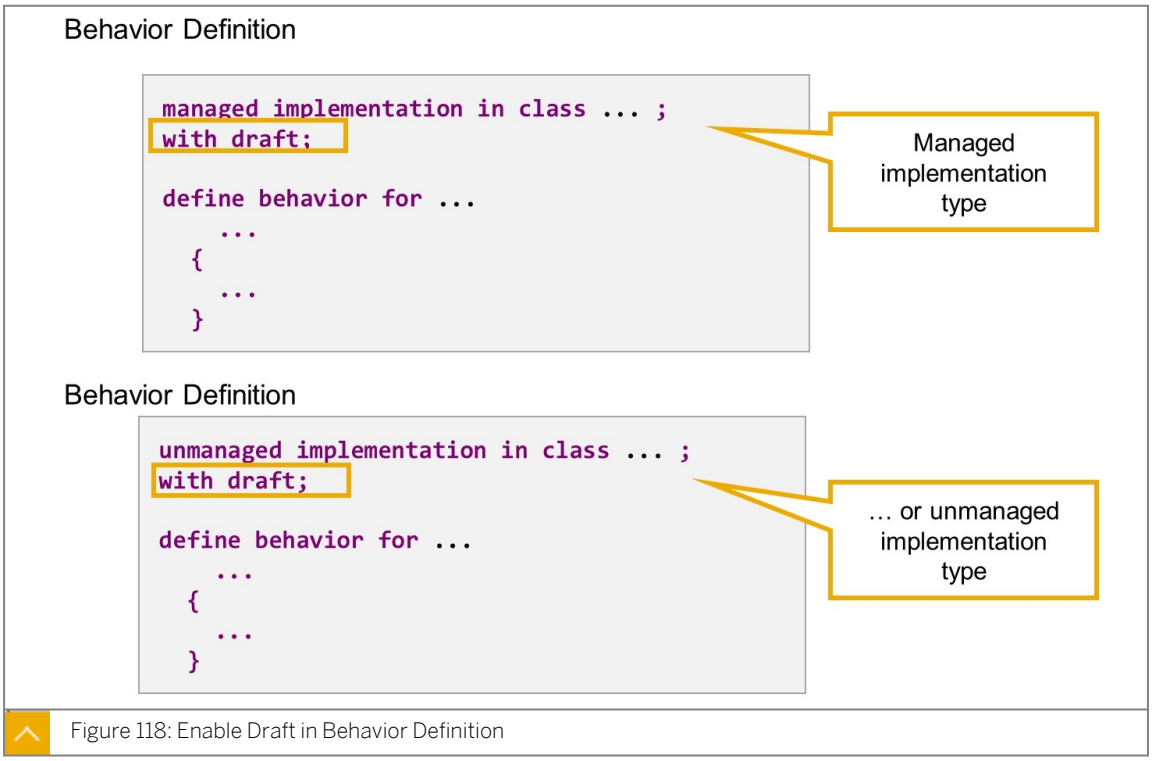

Figure 118: Enable Draft in Behavior Definition

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.

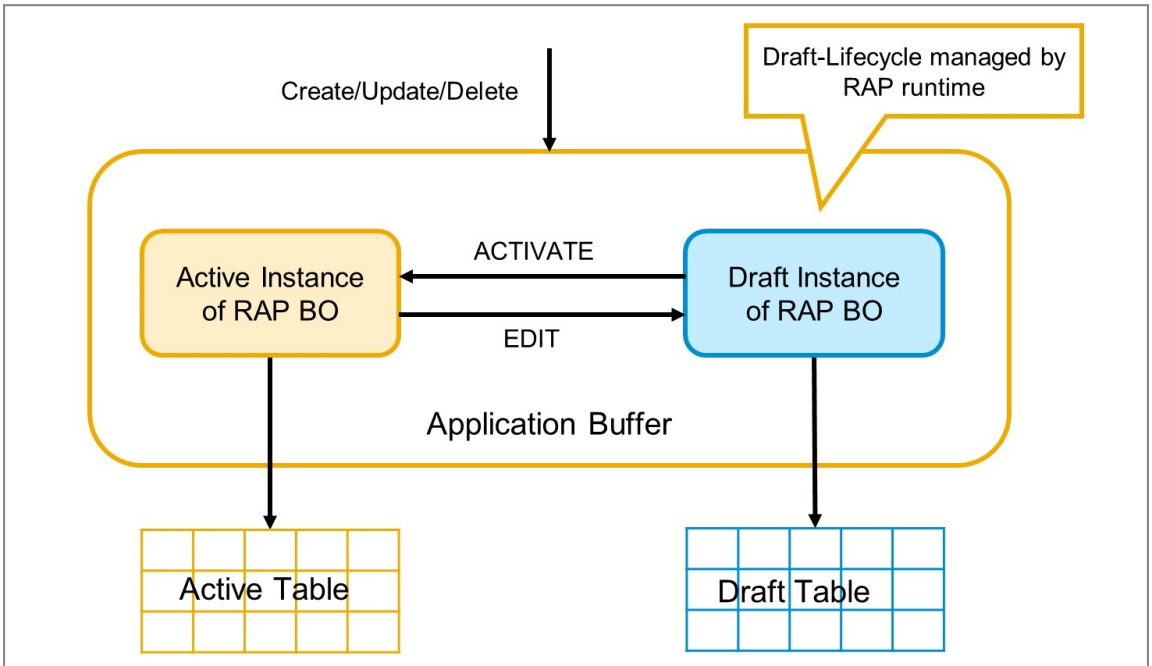

Figure 119: Draft Enabled Business Objects

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.

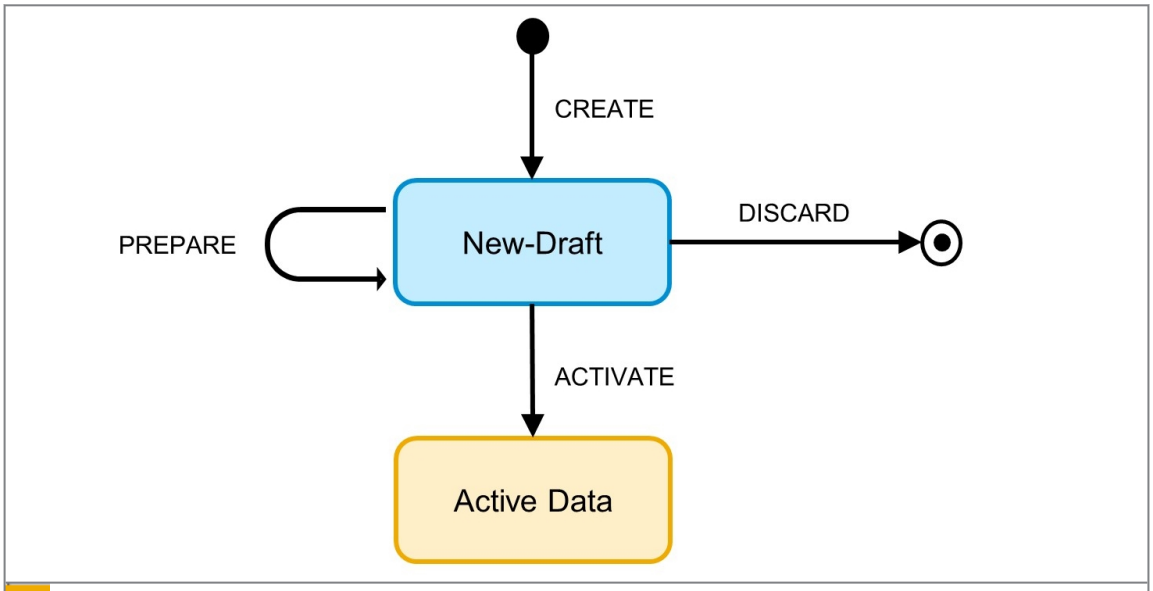

Figure 120: Life-Cycle of Draft 1 - Creating New Instance

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.

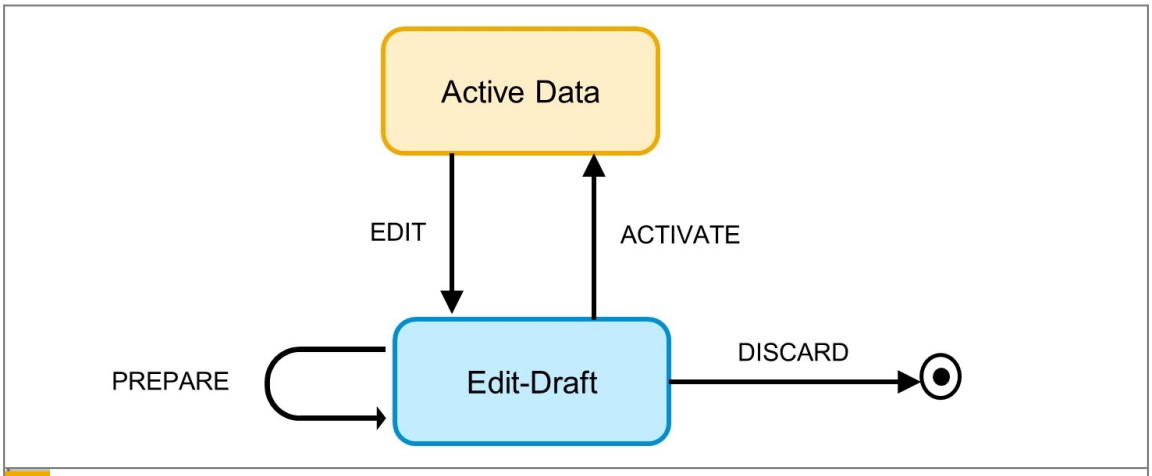

Figure 121: Life-Cycle of a Draft 2 - Editing Existing Instance

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

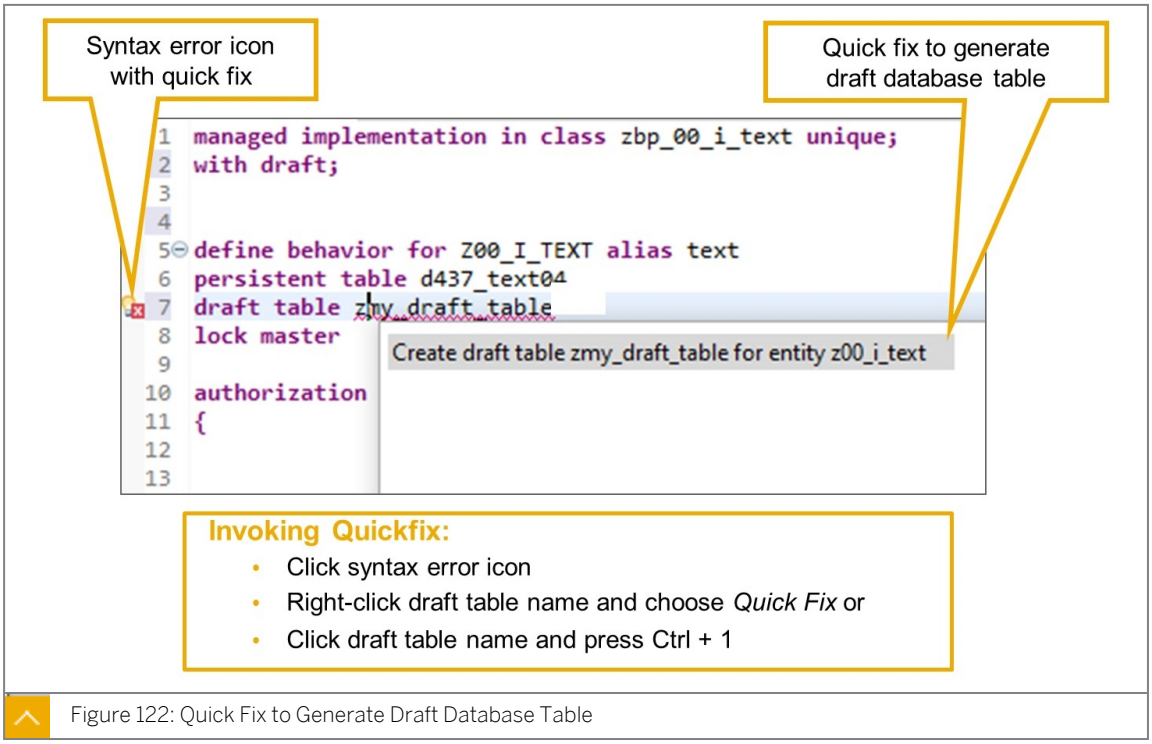

Figure 122: Quick Fix to Generate Draft Database Table

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.

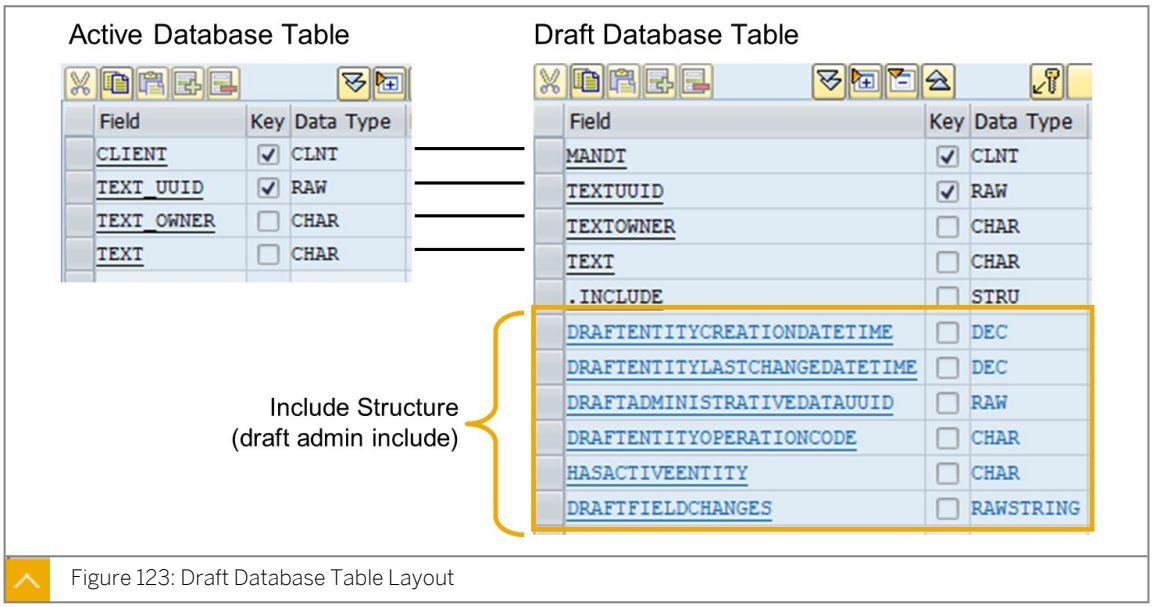

Figure 123: Draft Database Table Layout

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

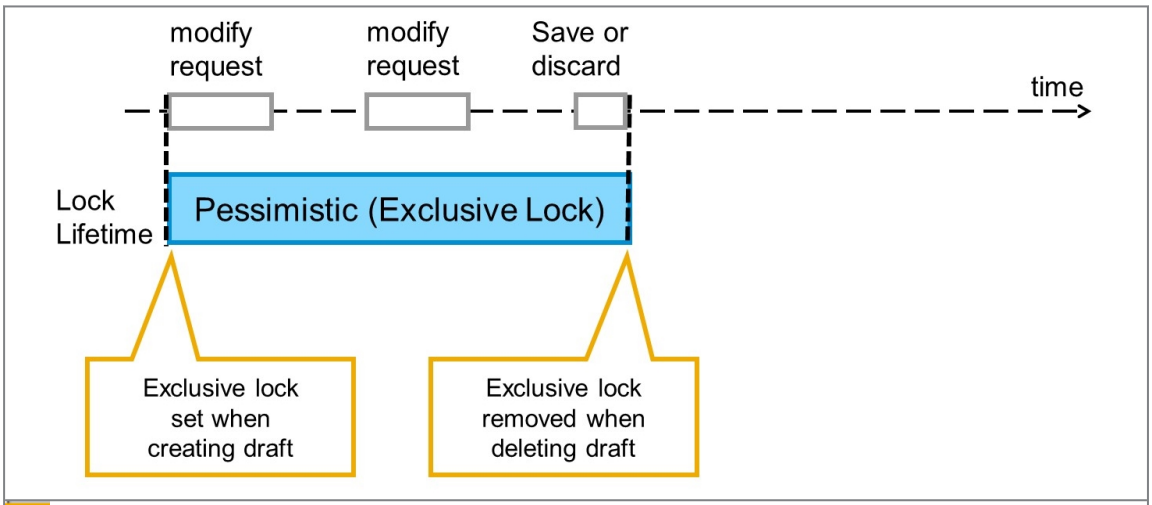

Figure 124: Pessimistic Concurrency Control in Draft

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.

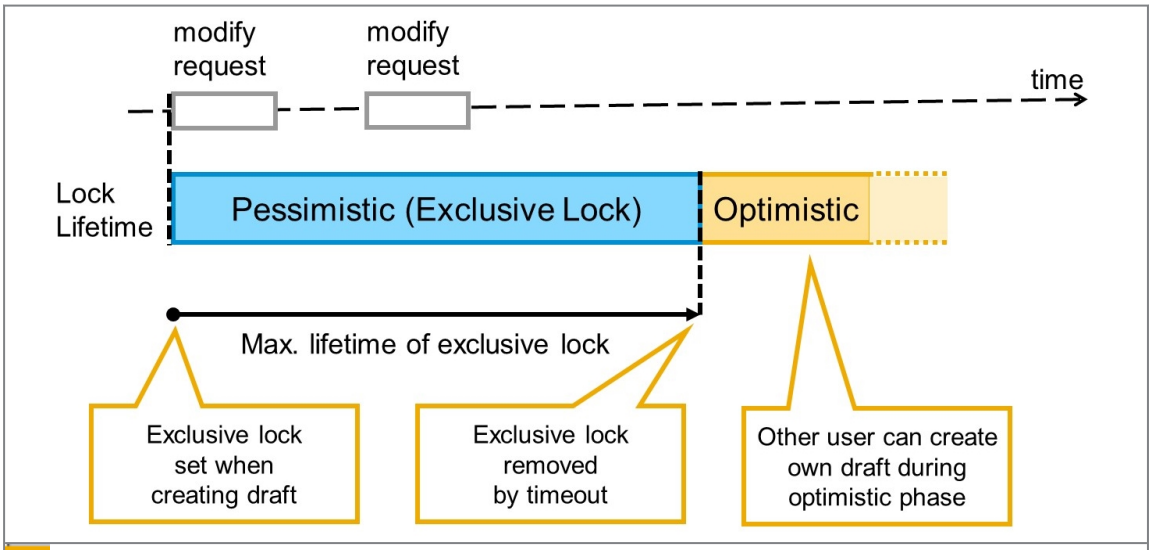

Figure 125: Optimistic Concurrency Control in Draft

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.

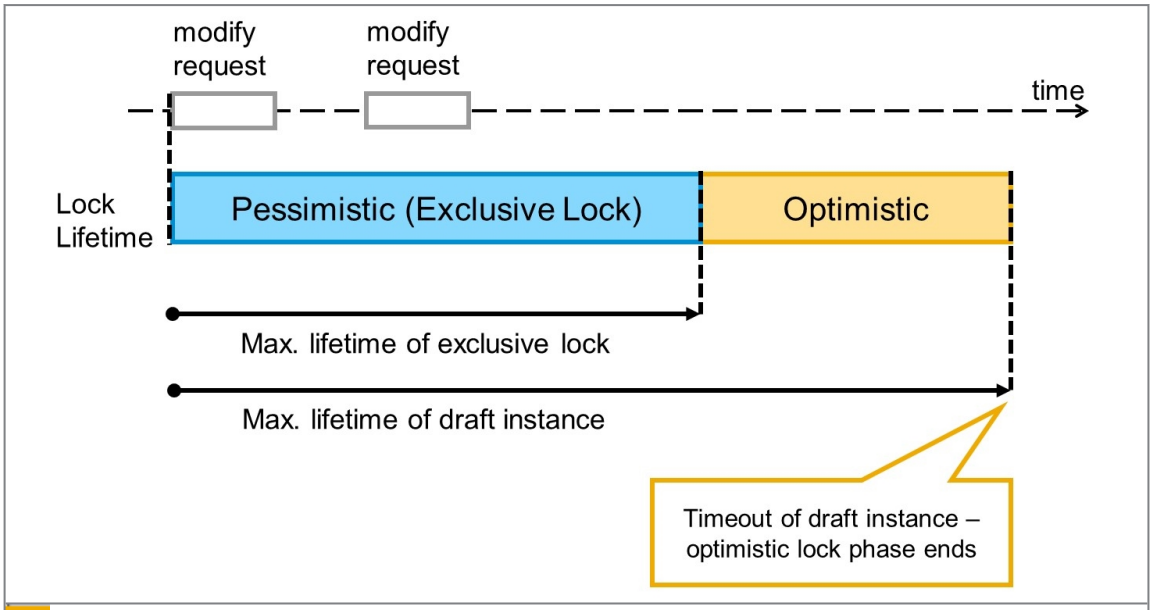

Figure 126: End of Optimistic Lock Phase 1 - Draft Timeout

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.

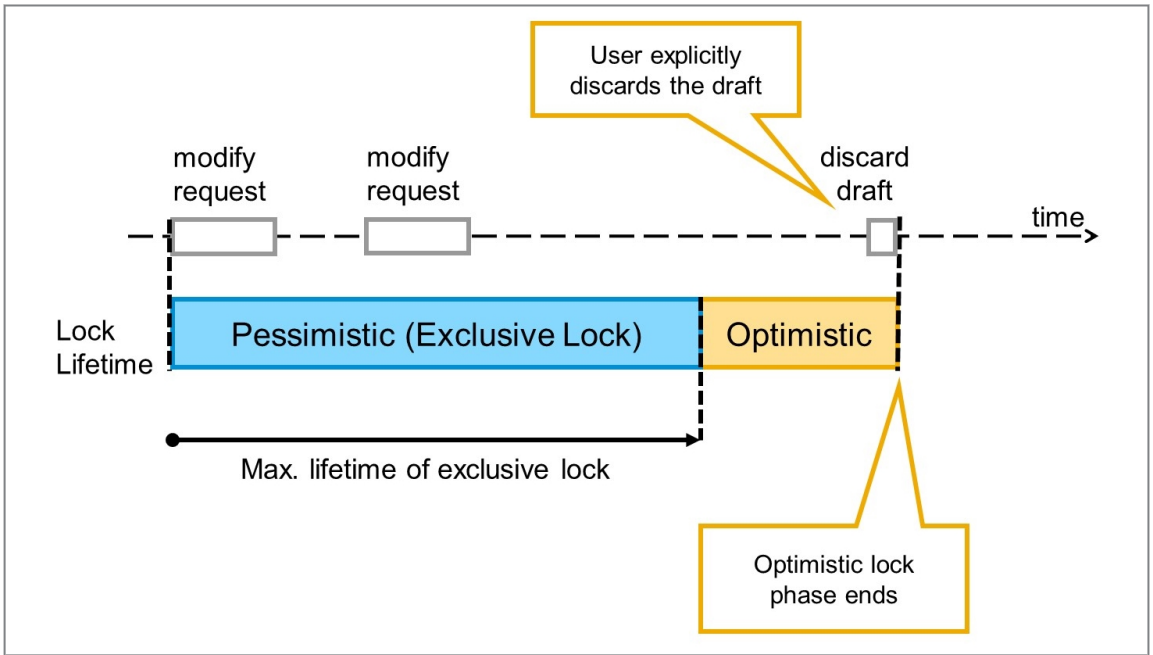

Figure 127: End of Optimistic Lock Phase 2 - Discard Draft

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.

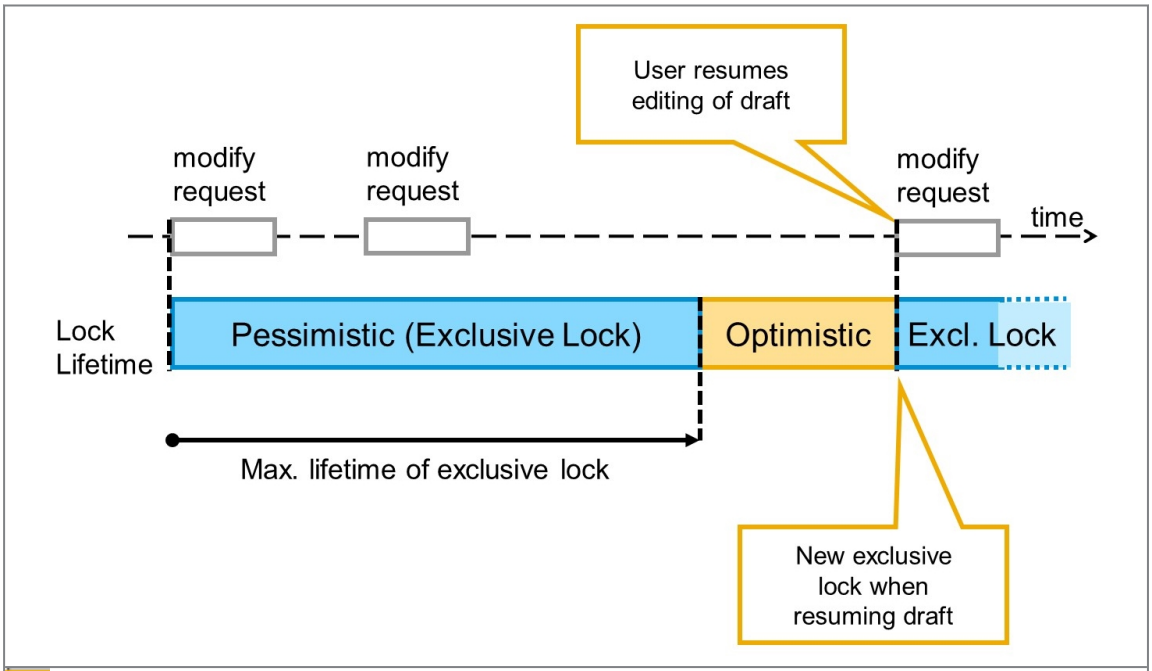

Figure 128: End of Optimistic Lock Phase 3 - Draft Resume

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.

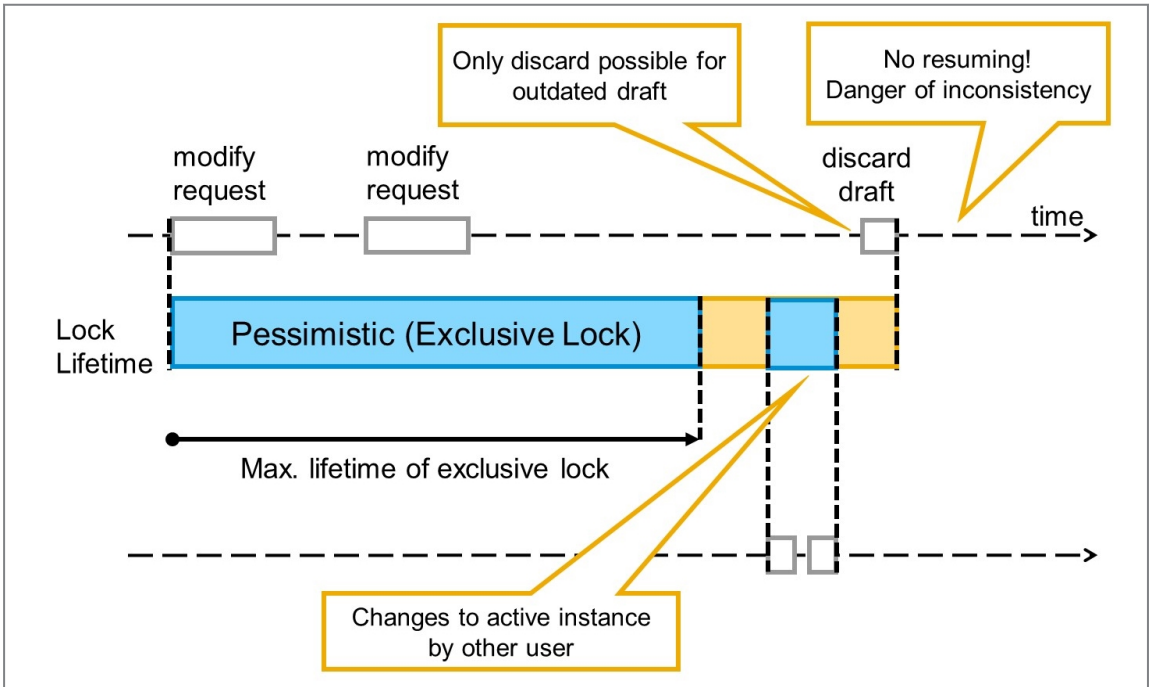

Figure 129: How to Avoid Resume of Outdated Draft?

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

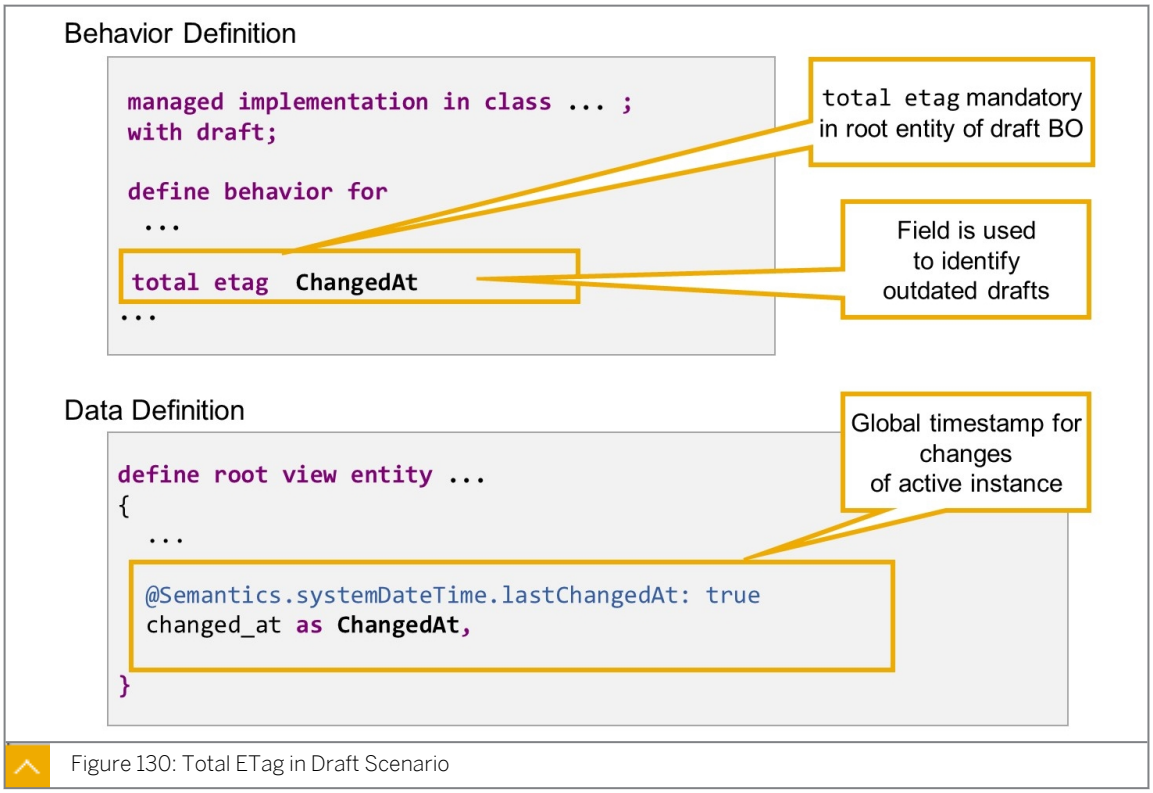

Figure 130: Total ETag in Draft Scenario

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.

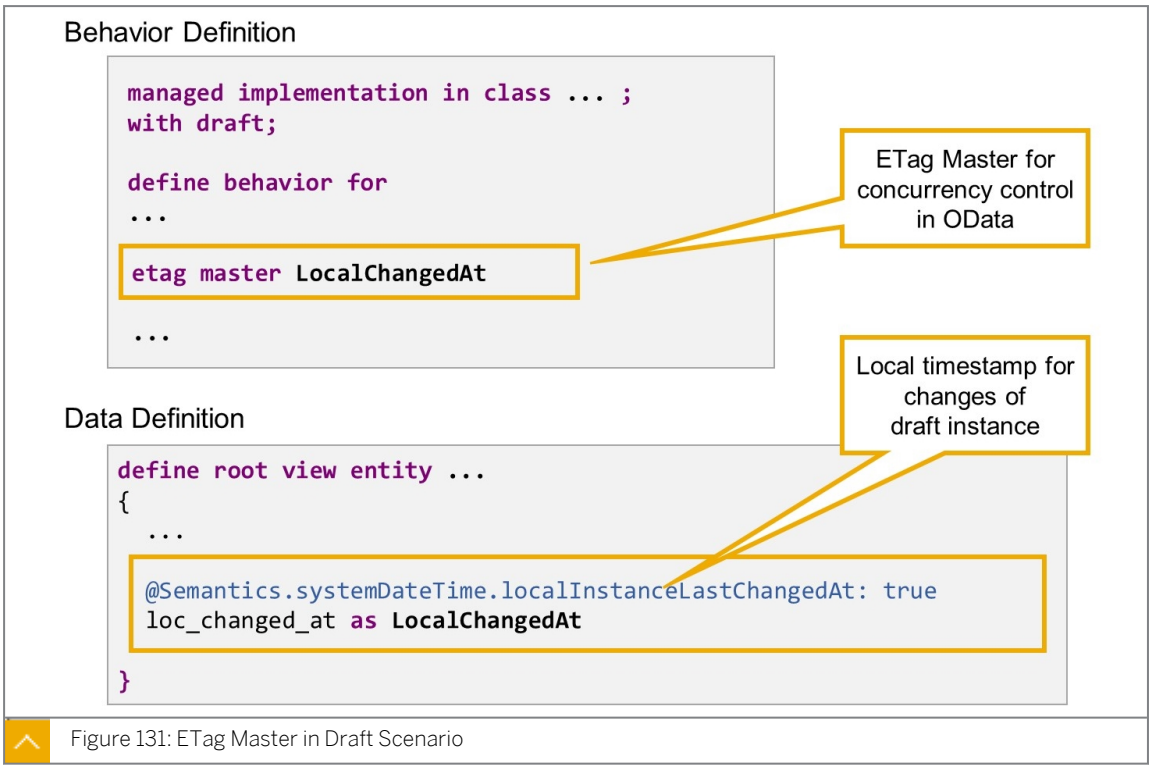

Figure 131: ETag Master in Draft Scenario

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

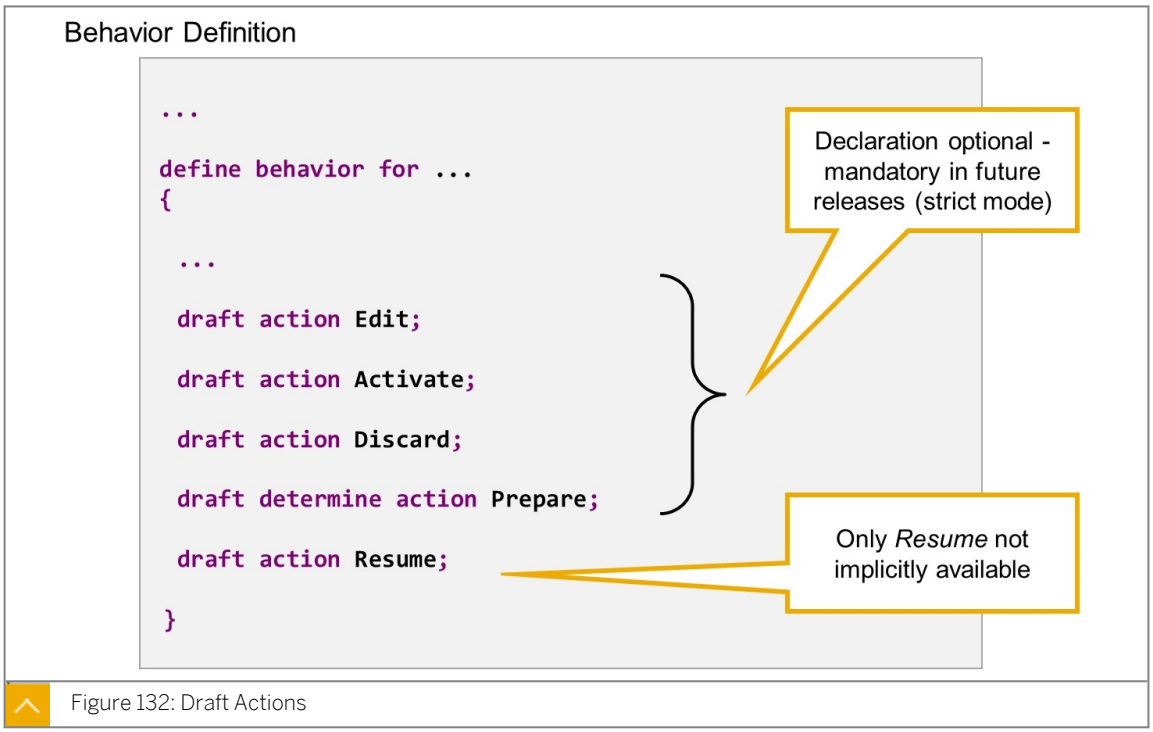

Figure 132: Draft Actions

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.

LESSON SUMMARY

You should now be able to:

- Explain the need for draft in stateless applications
- Enable draft handling in the Business Object

Unit 4
Lesson 2

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

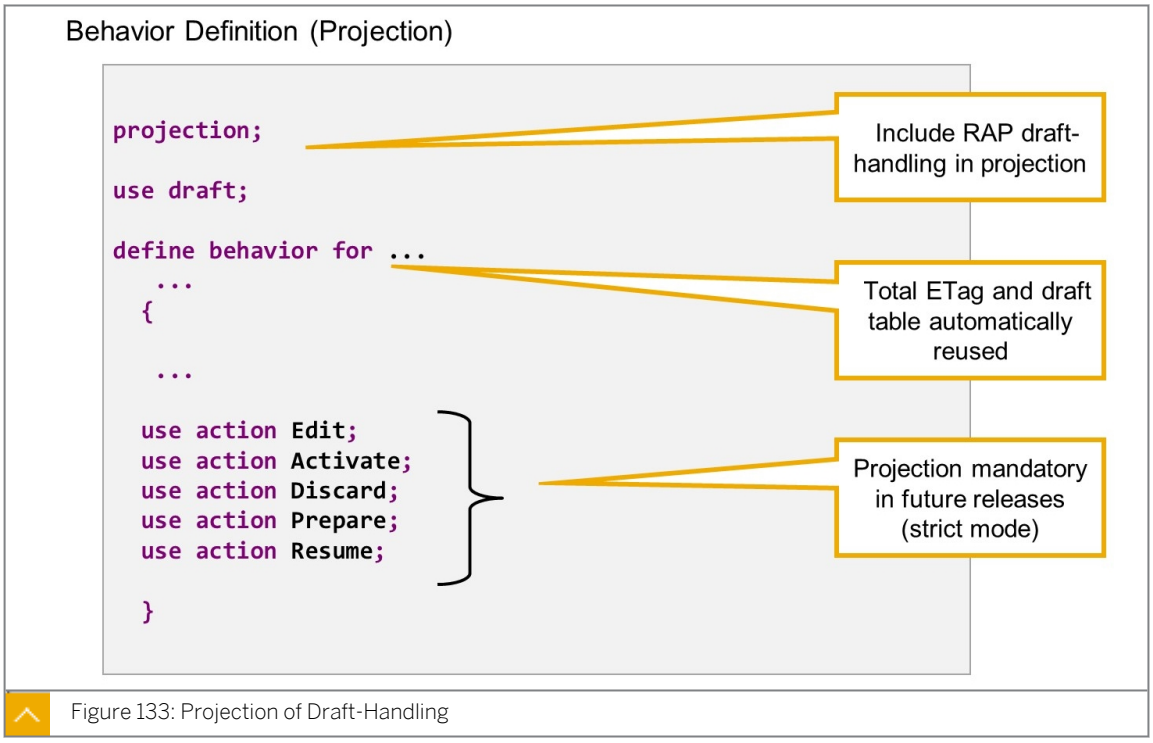

Figure 133: Projection of Draft-Handling

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.

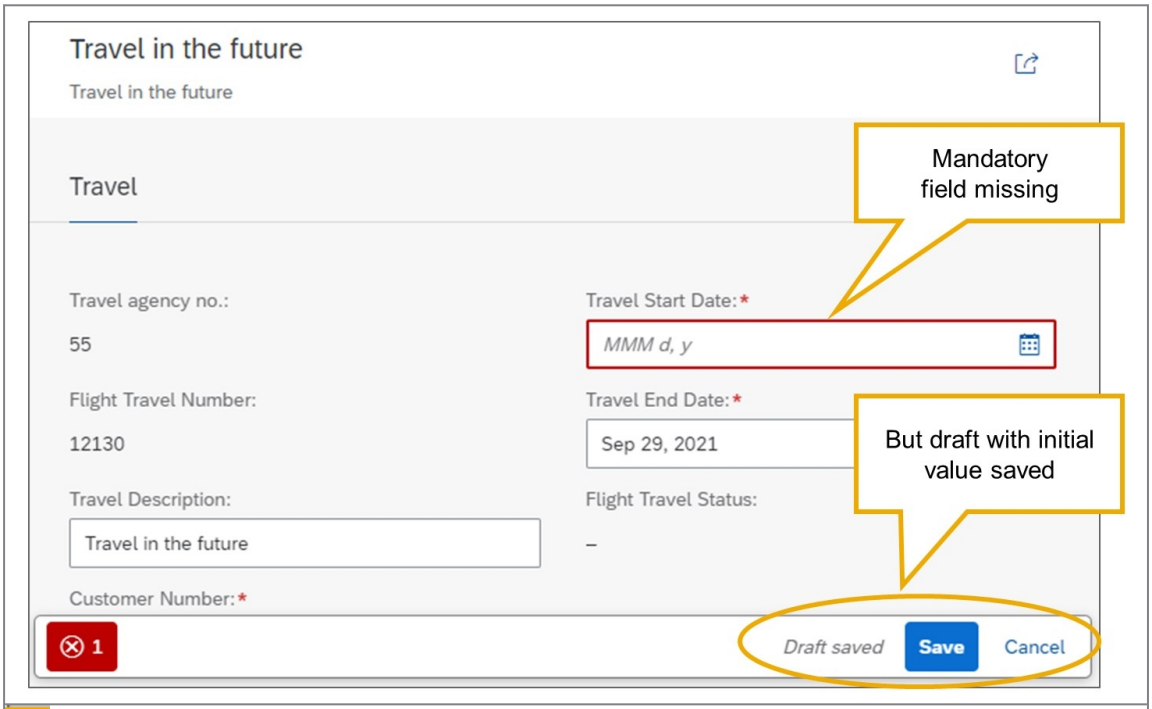

Figure 134: Editing a Draft Instance

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.

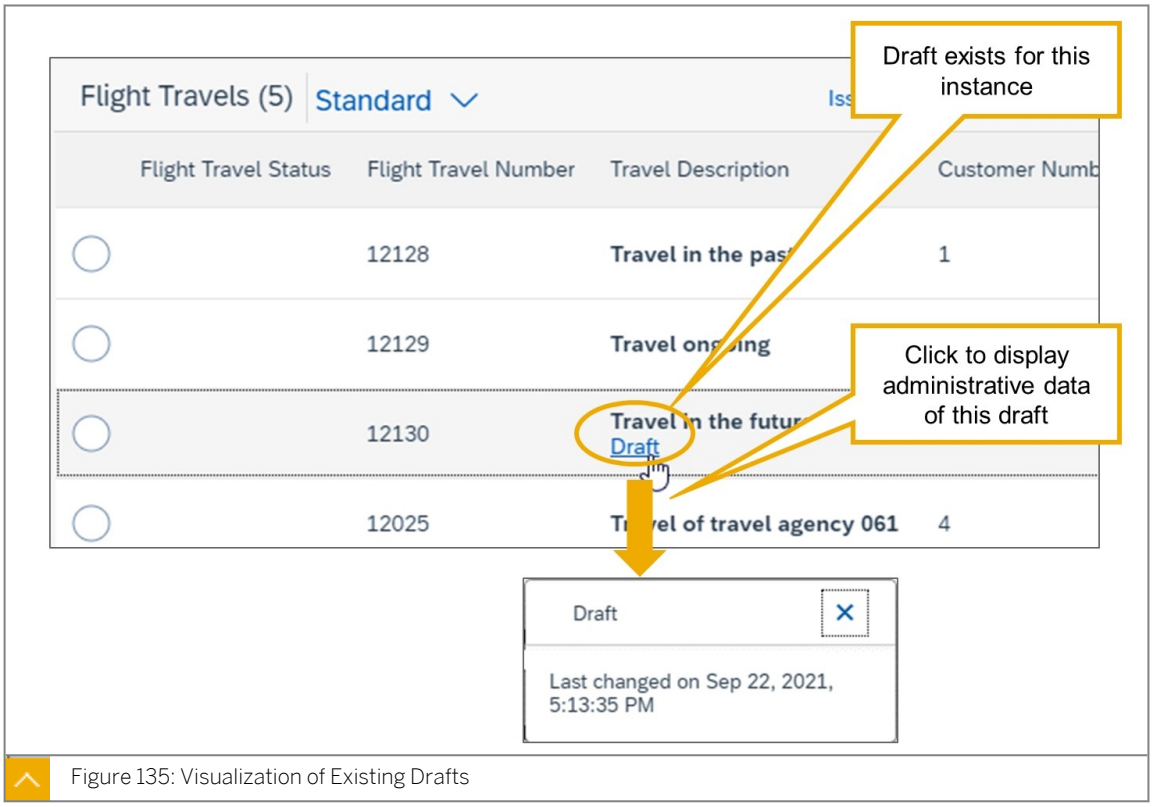

Figure 135: Visualization of Existing Drafts

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.

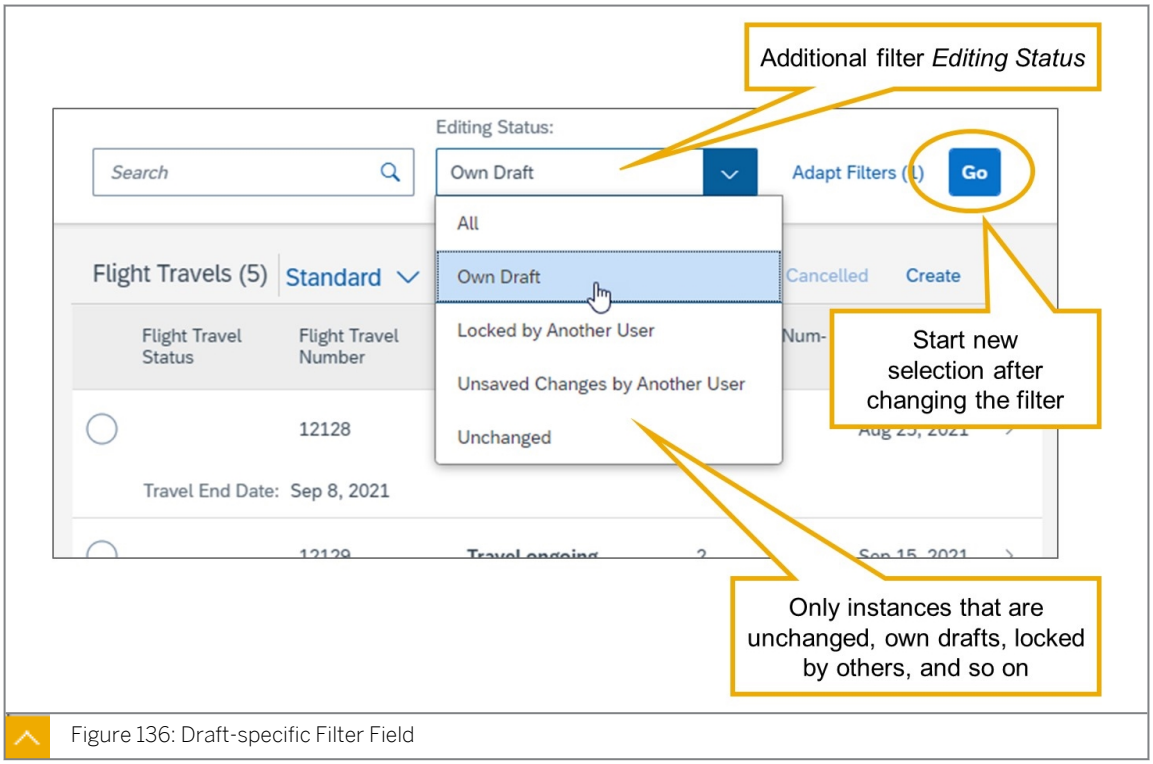

Figure 136: Draft-specific Filter Field

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

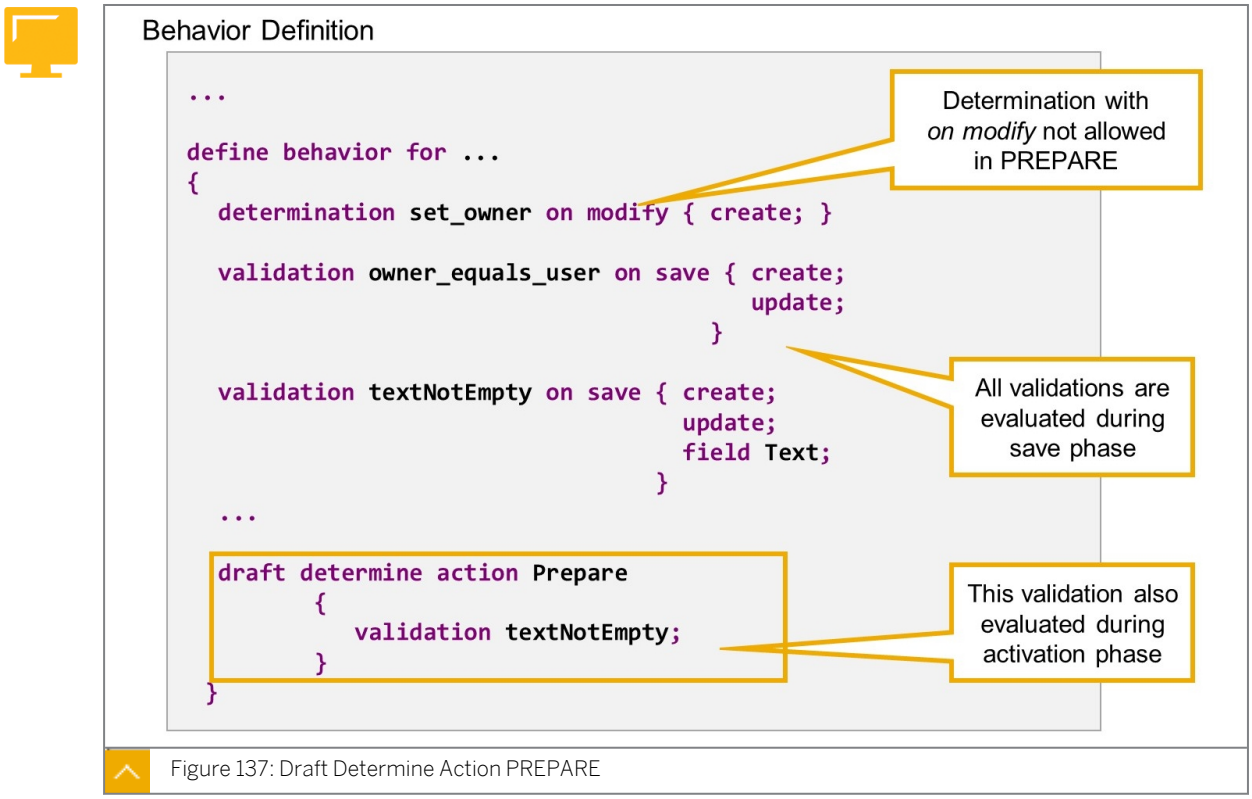

Figure 137: Draft Determine Action PREPARE

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	`%state_area` initial	`%state_area` not initial
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 are 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.

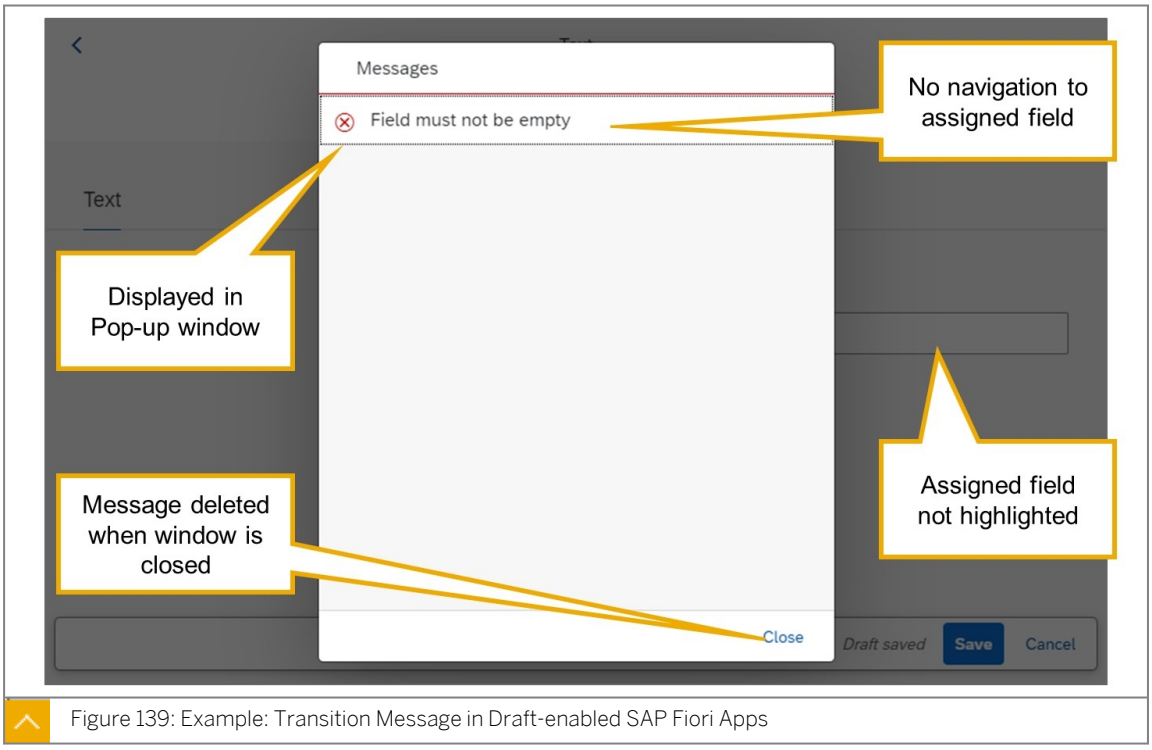

Figure 139: Example: Transition Message in Draft-enabled SAP Fiori Apps

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.

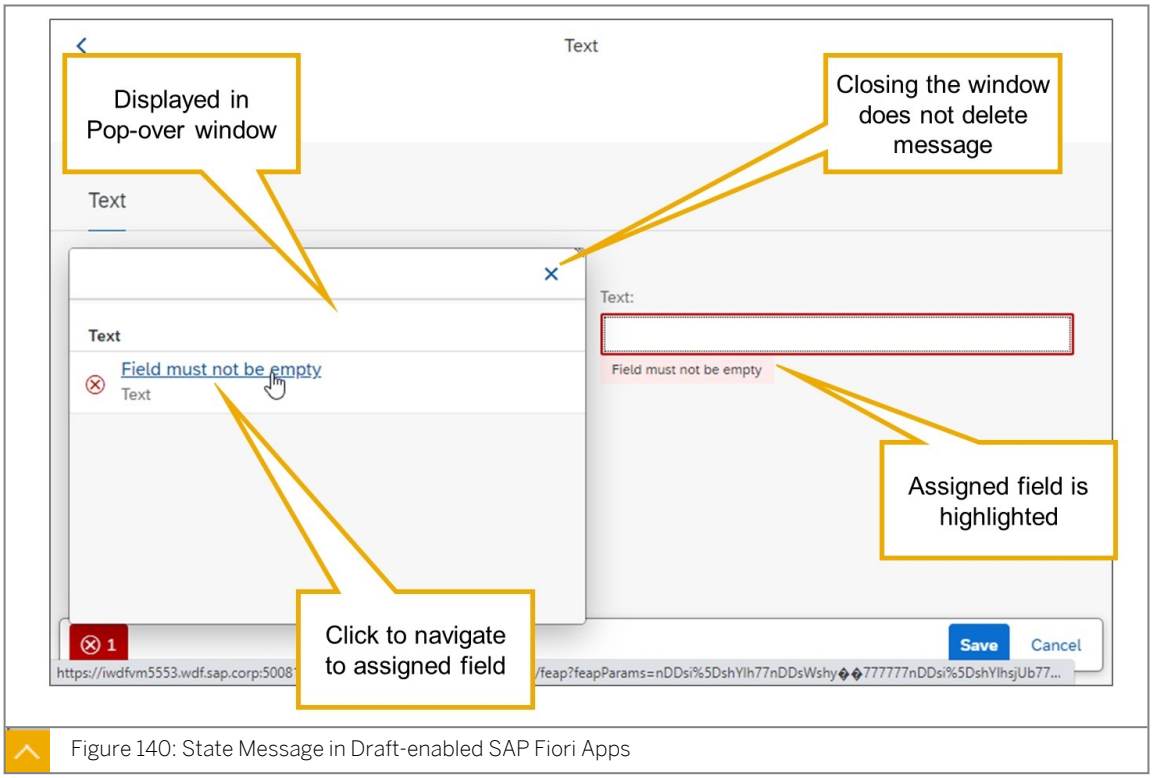

Figure 140: State Message in Draft-enabled SAP Fiori Apps

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.

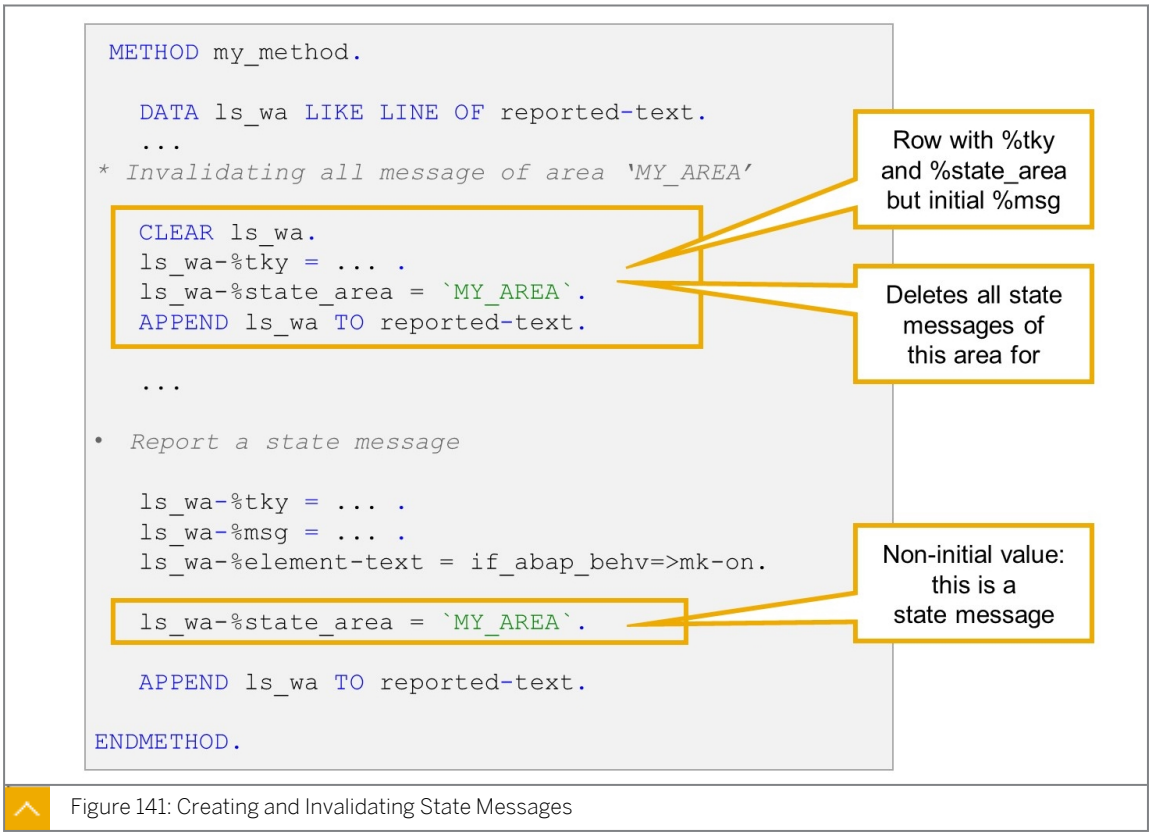

Figure 141: Creating and Invalidating State Messages

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

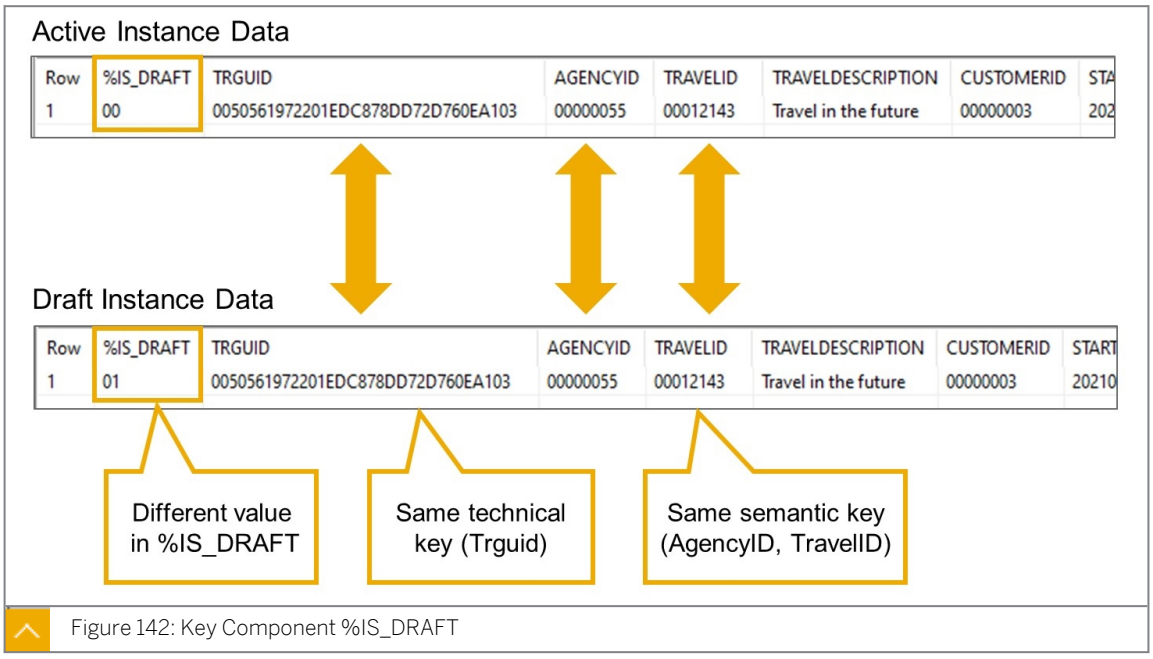

Figure 142: Key Component %IS_DRAFT

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`.

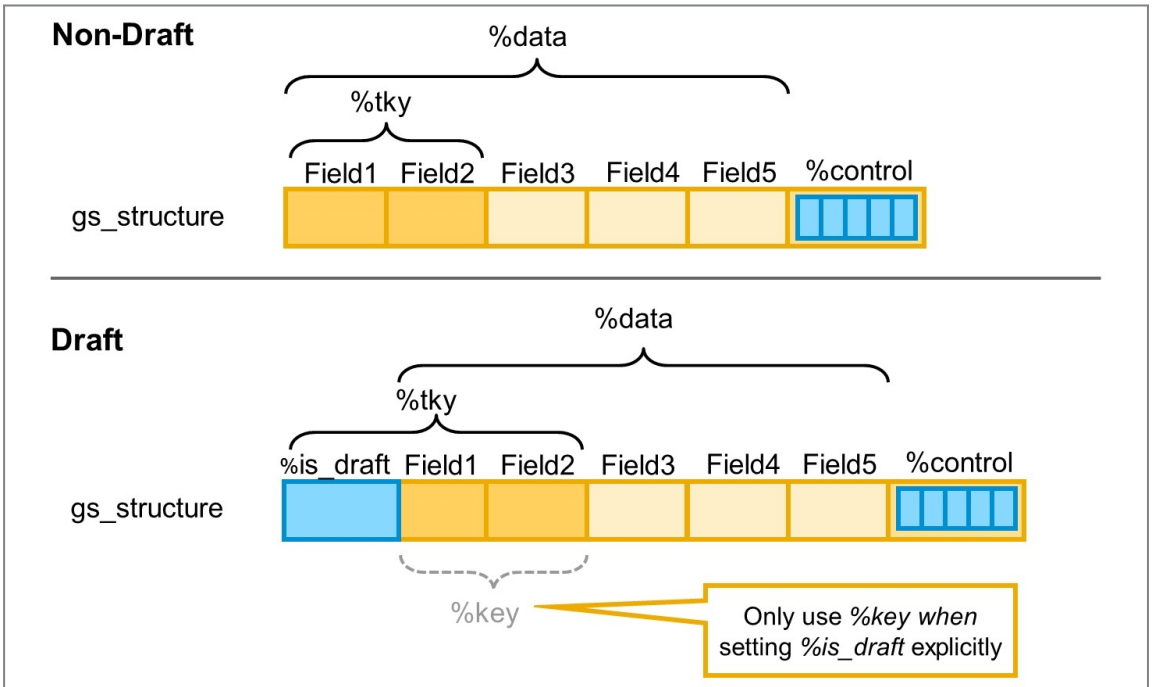

Figure 143: Component Groups %tky and %key

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

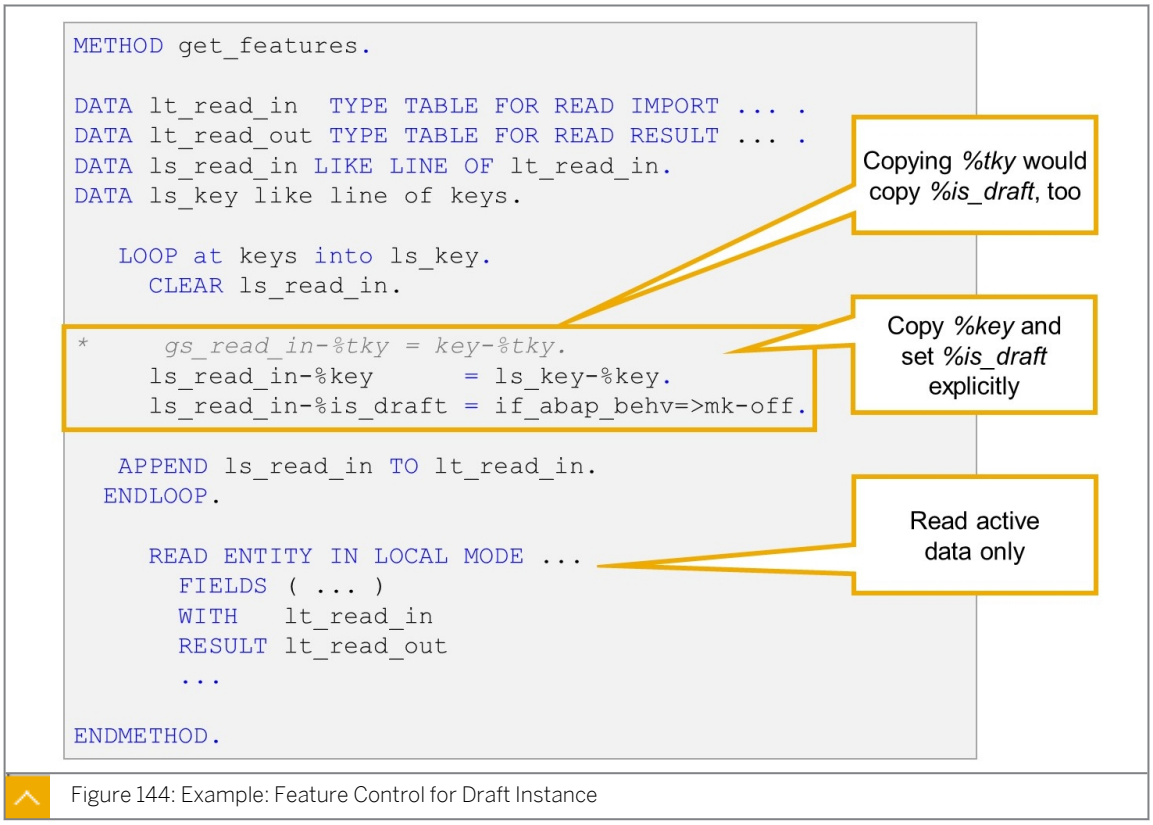

Figure 144: Example: Feature Control for Draft Instance

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.

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

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

Unit 5
Lesson 1

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

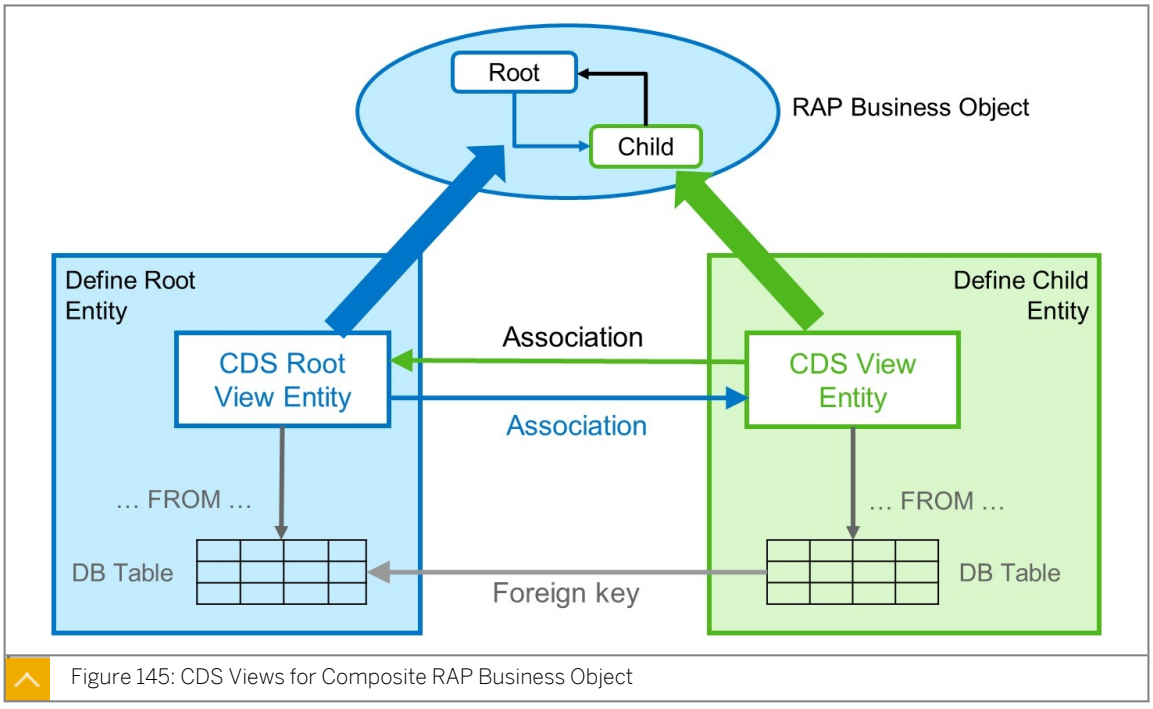

Figure 145: CDS Views for Composite RAP Business Object

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.

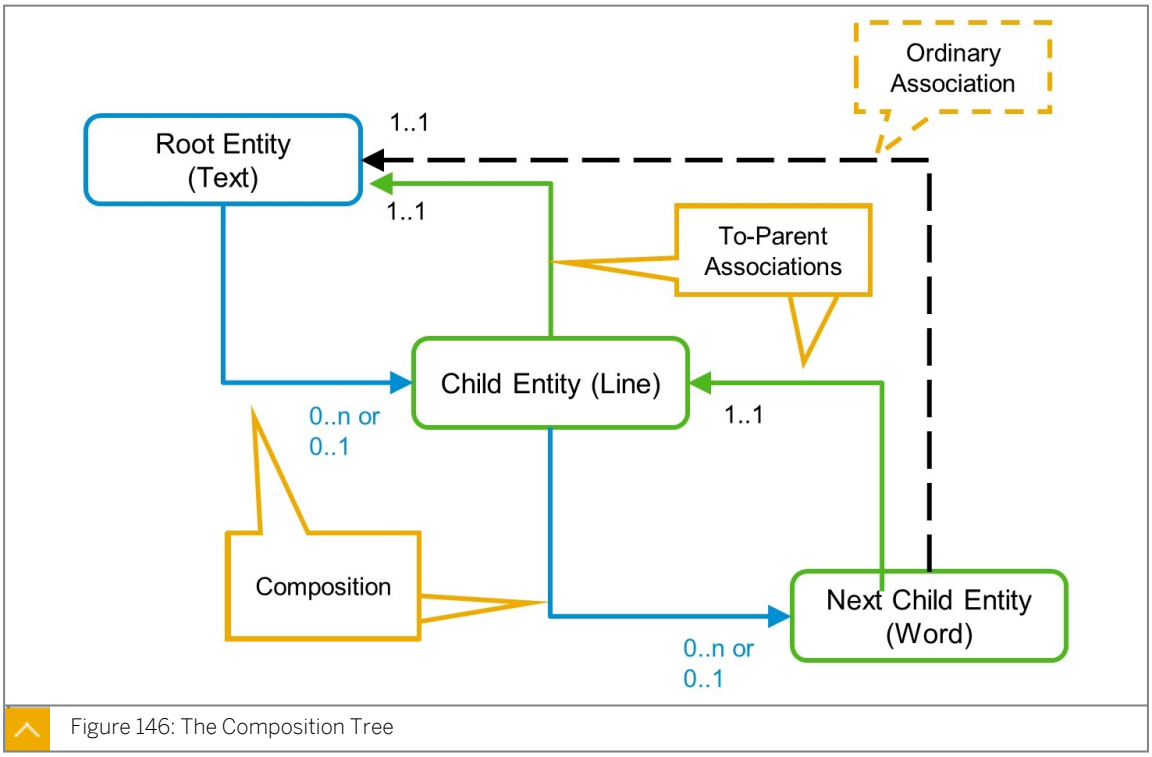

Figure 146: The Composition Tree

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

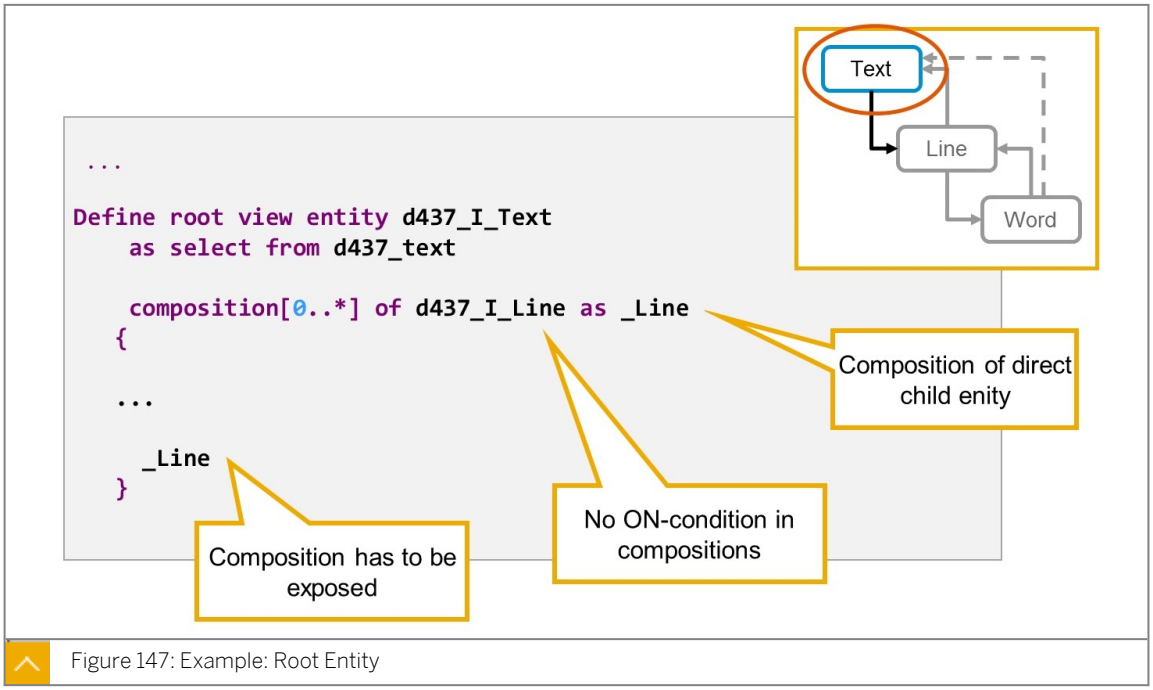

Figure 147: Example: Root Entity

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.

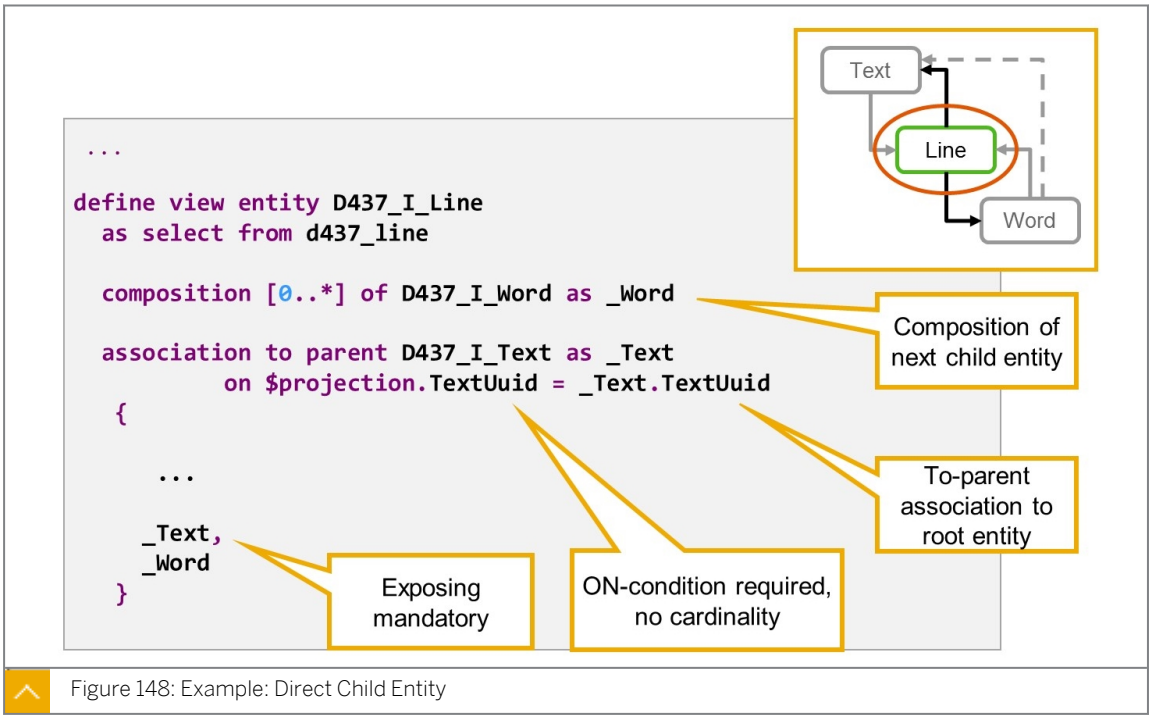

Figure 148: Example: Direct Child Entity

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.

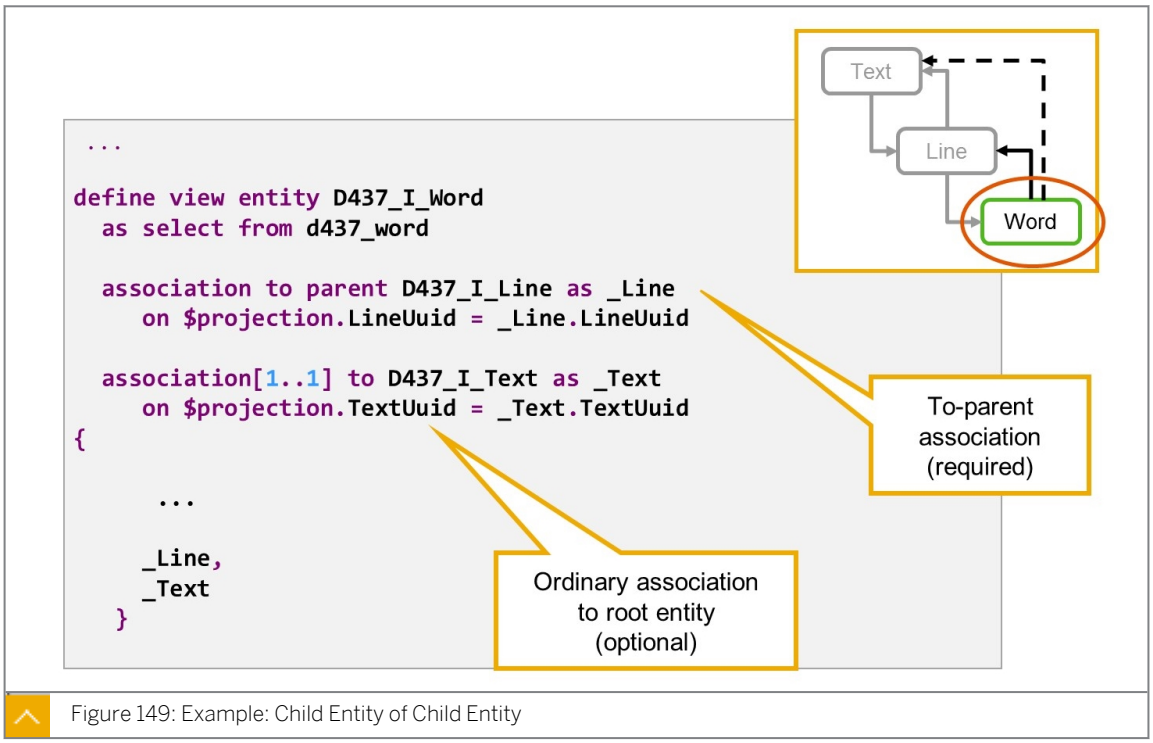

Figure 149: Example: Child Entity of Child Entity

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

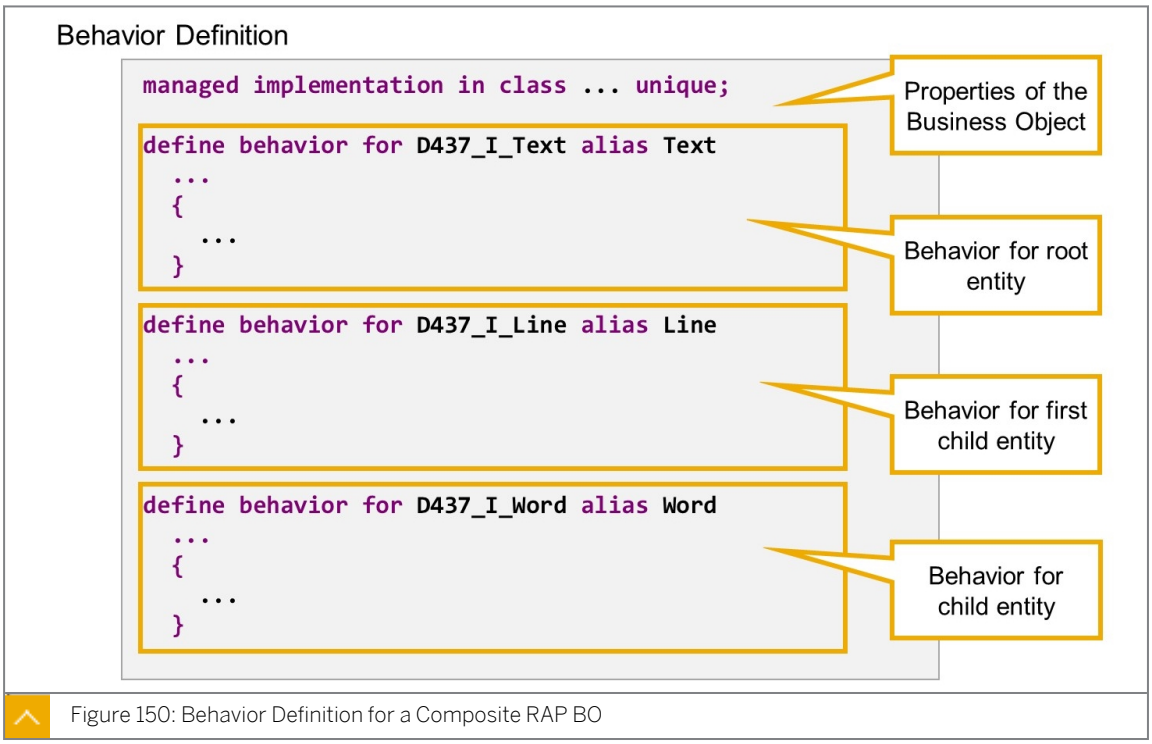

Figure 150: Behavior Definition for a Composite RAP BO

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.

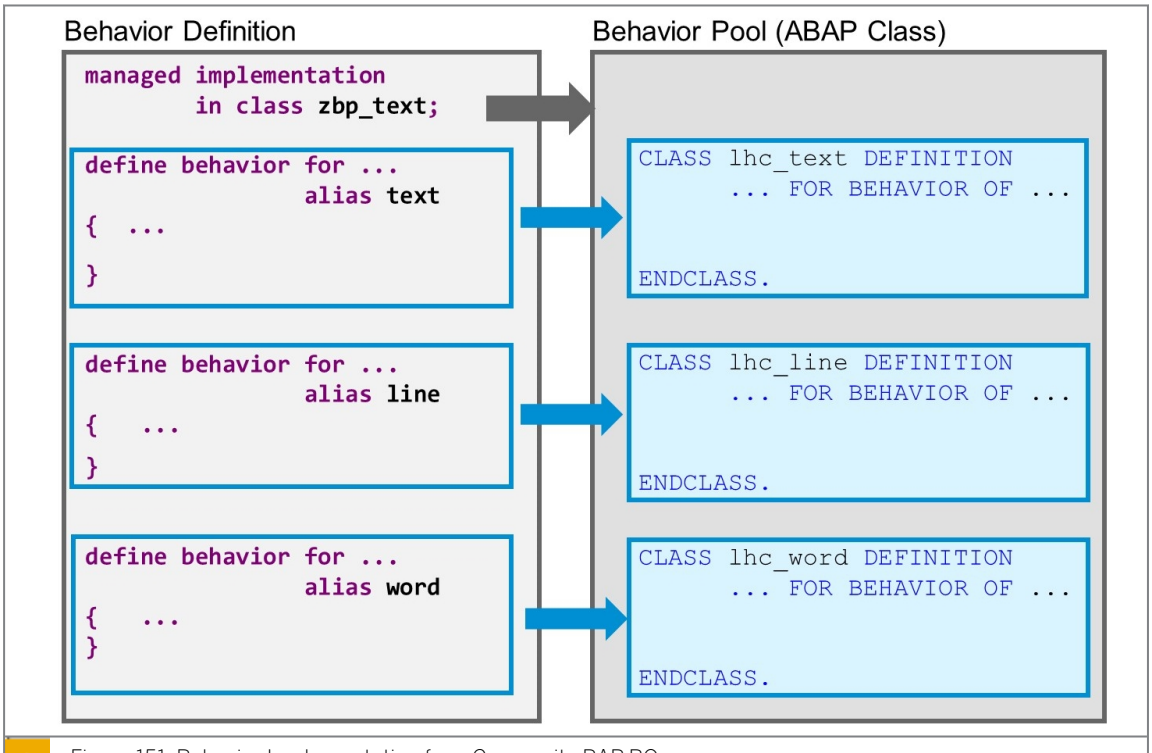

Figure 151: Behavior Implementation for a Composite RAP BO

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.

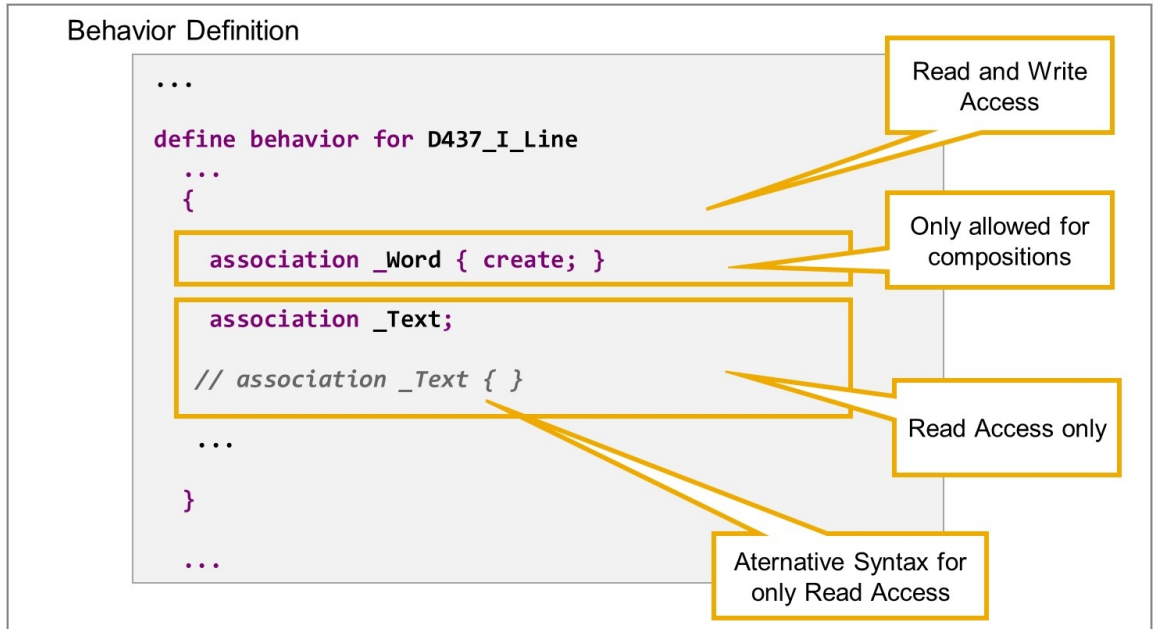

Figure 152: Associations in the Behavior Definition

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.

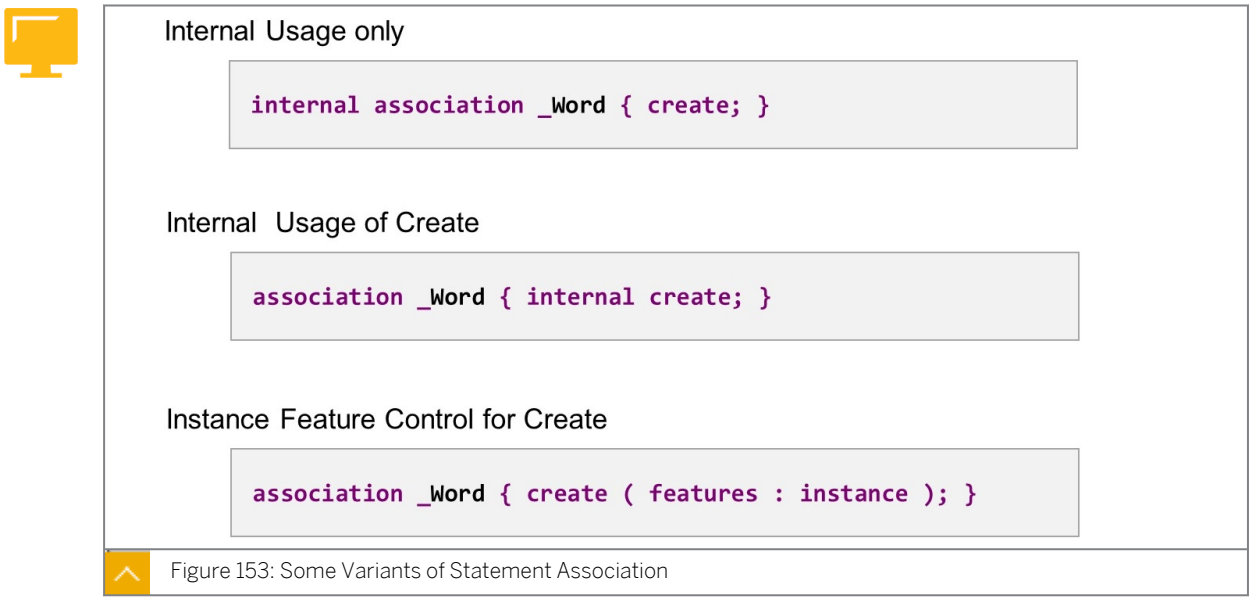

Figure 153: Some Variants of Statement Association

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`.

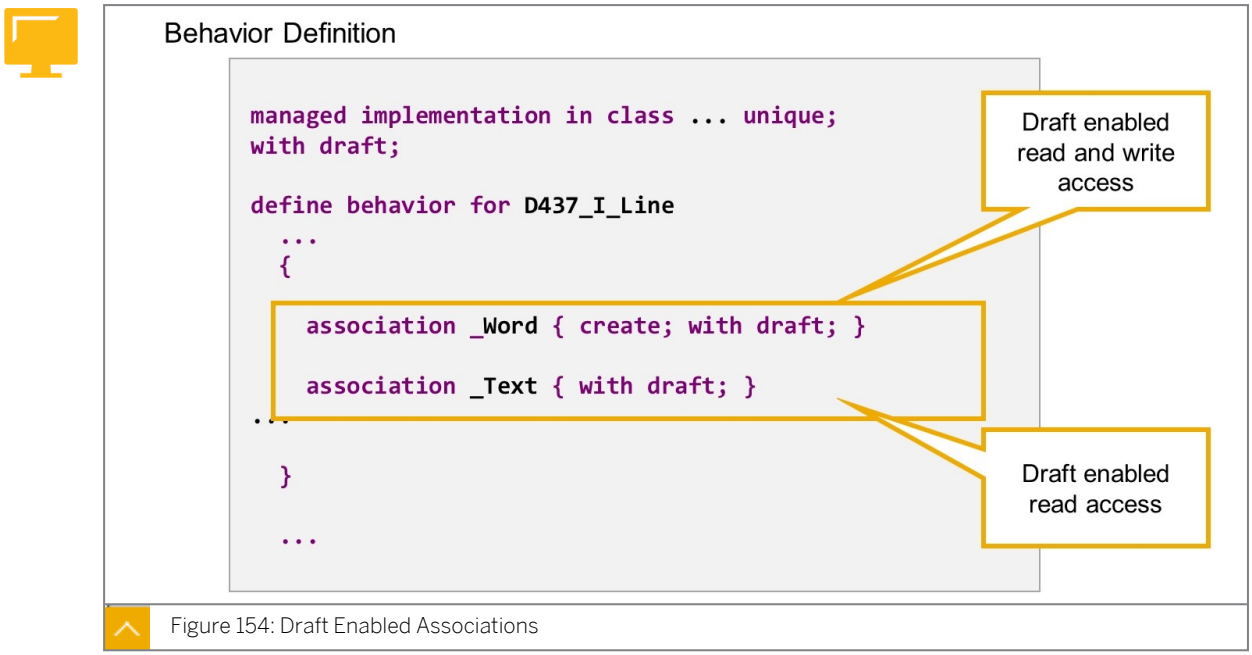

Figure 154: Draft Enabled Associations

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;`.

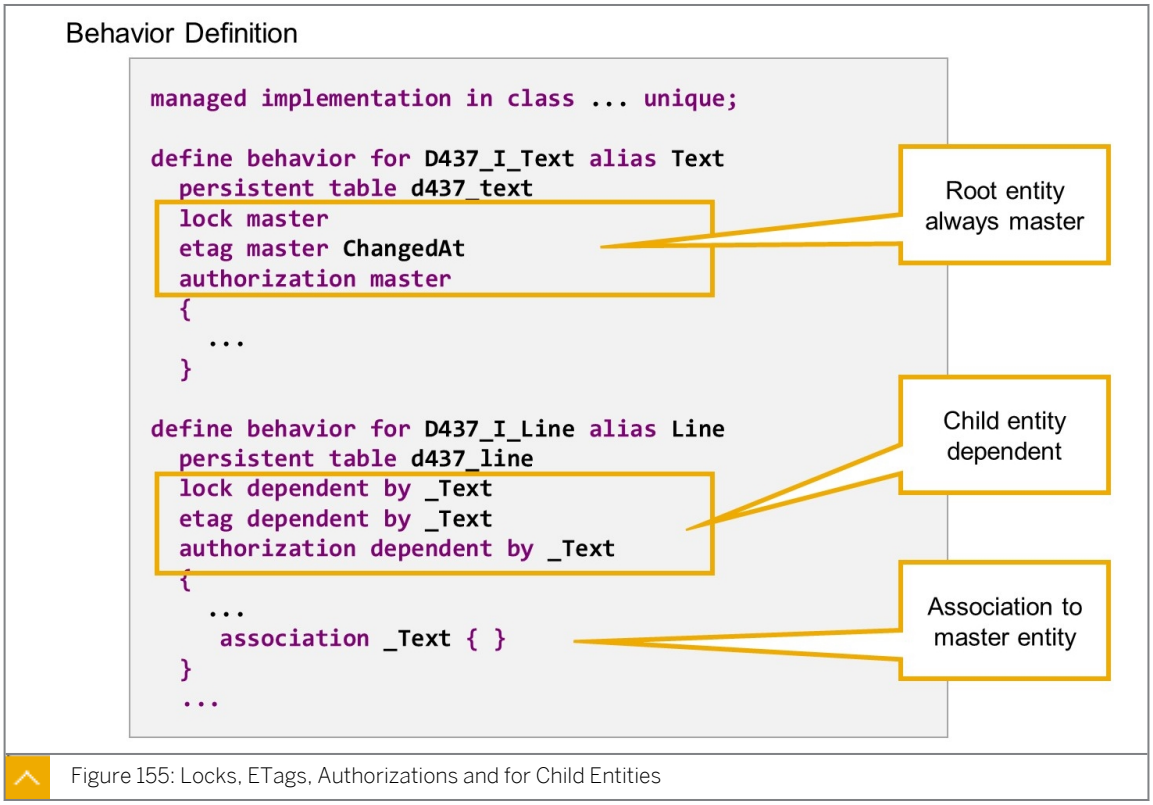

Figure 155: Locks, ETags, Authorizations and for Child Entities

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.

LESSON SUMMARY
You should now be able to:

- Define compositions in RAP BOs

Unit 5
Lesson 2

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

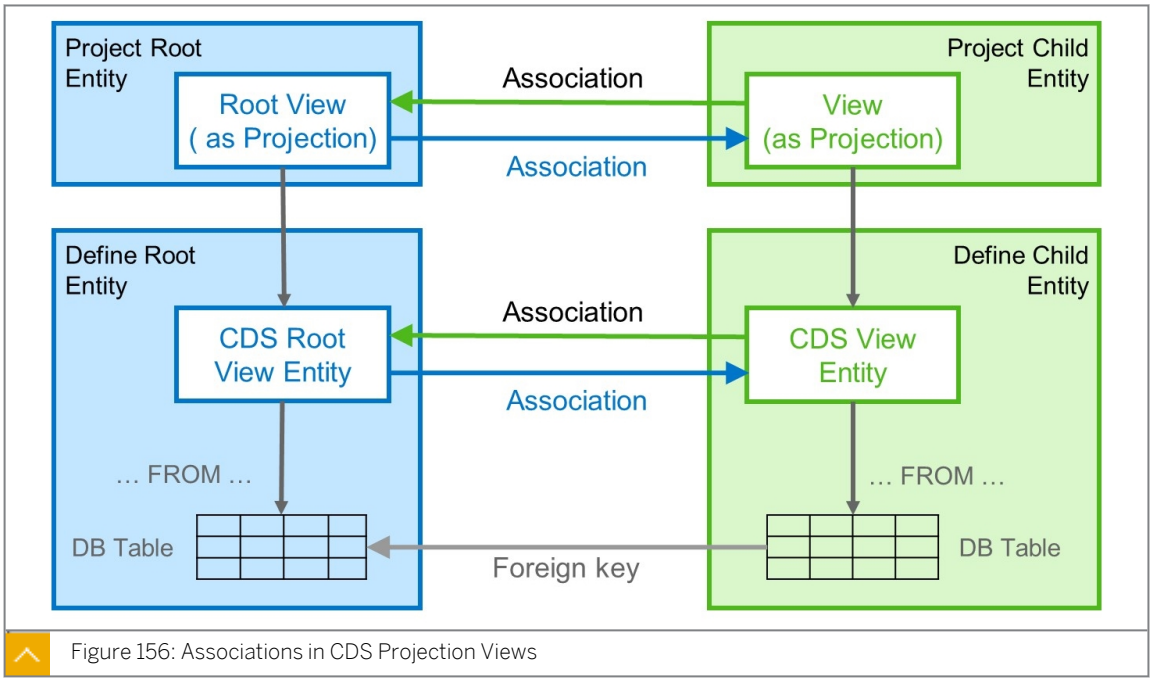

Figure 156: Associations in CDS Projection Views

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.

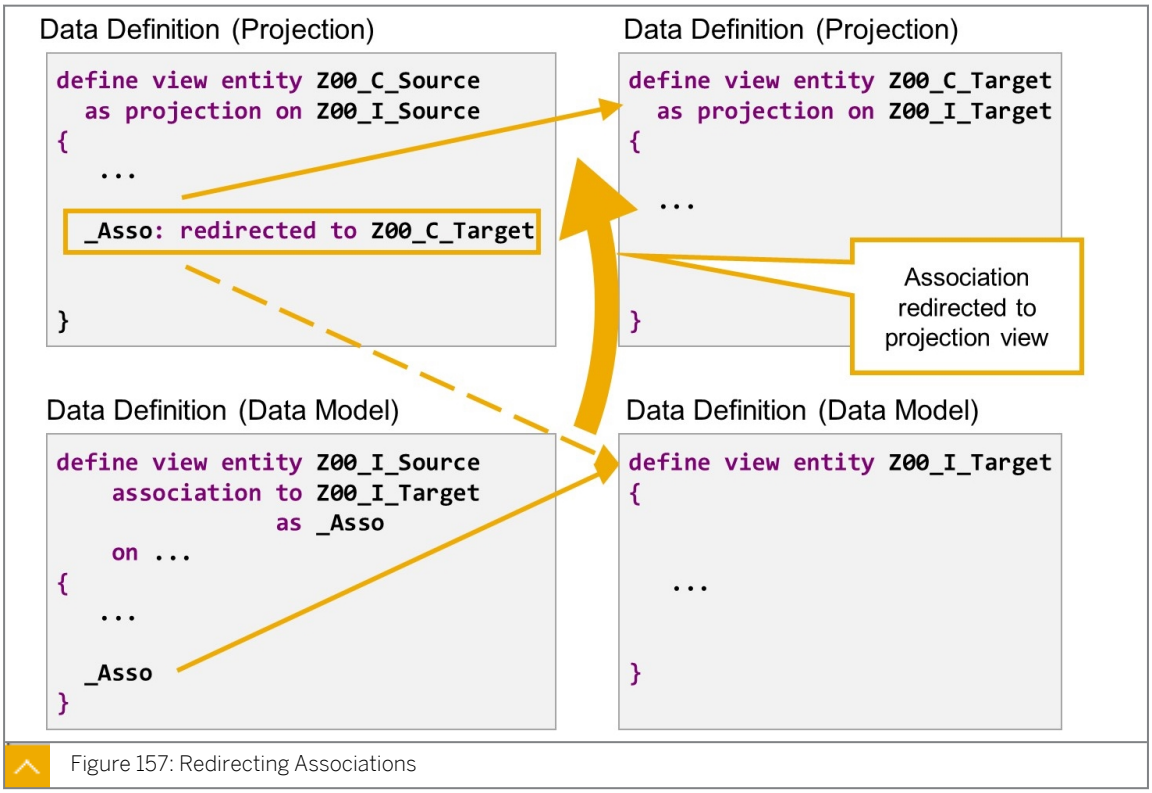

Figure 157: Redirecting Associations

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.

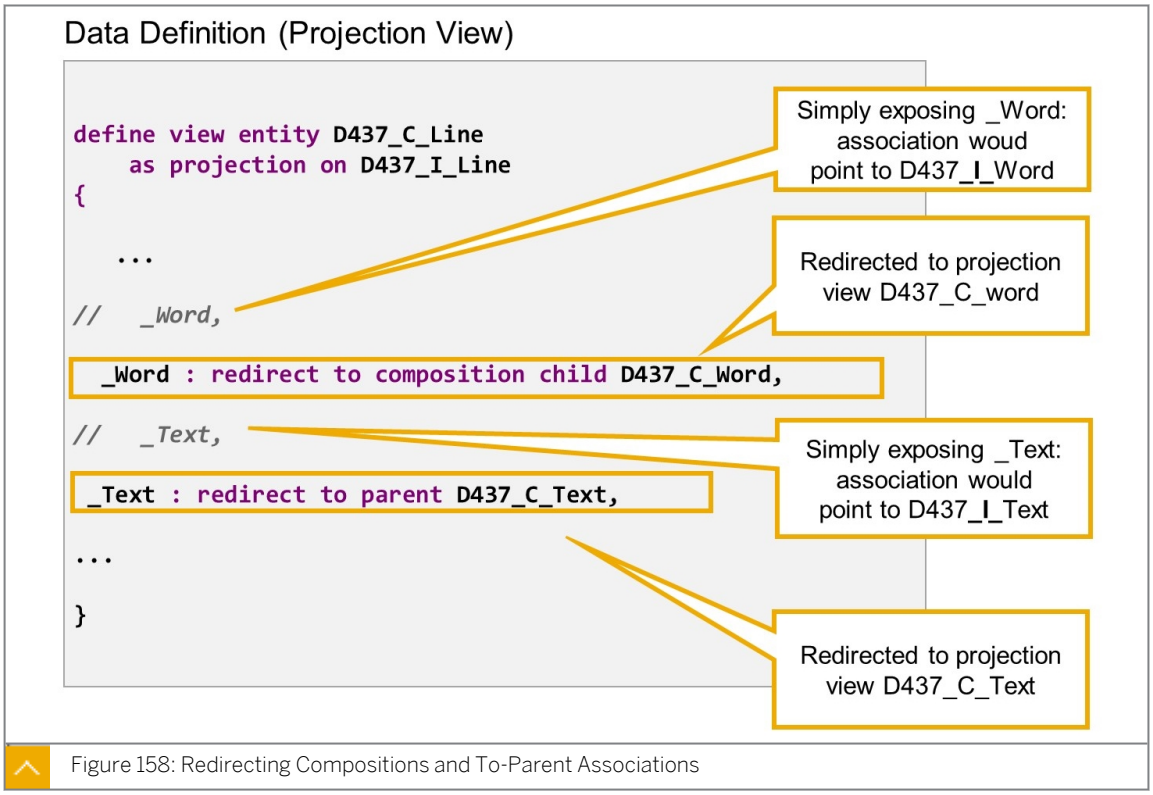

Figure 158: Redirecting Compositions and To-Parent Associations

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

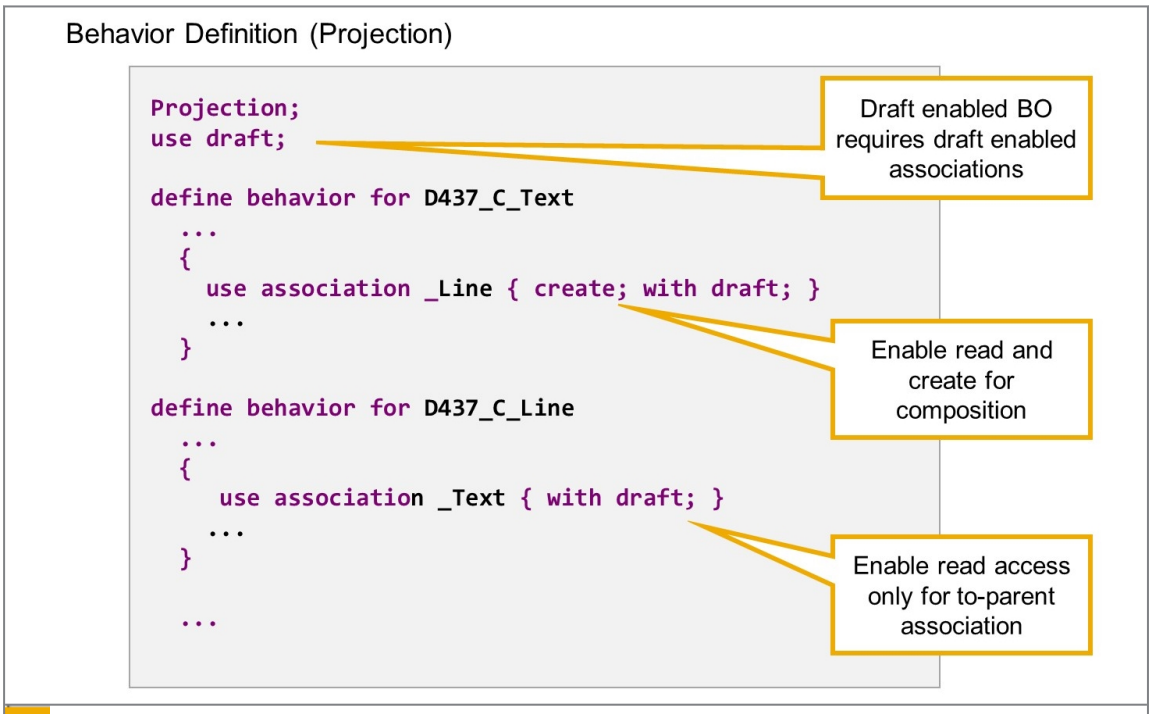

Figure 159: Behavior Projection of Composite Business Object

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

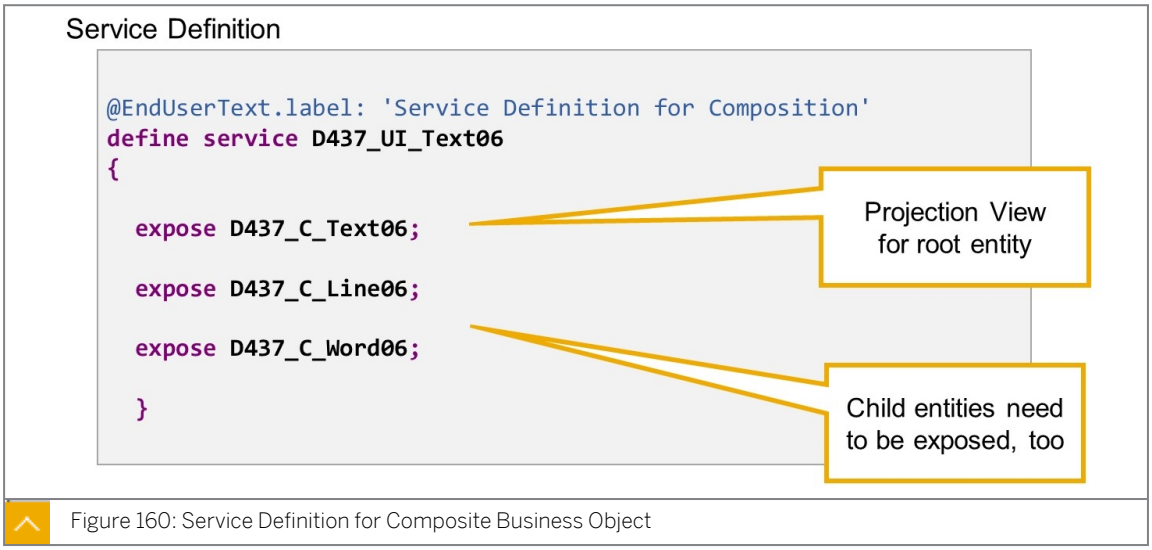

Figure 160: Service Definition for Composite Business Object

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.

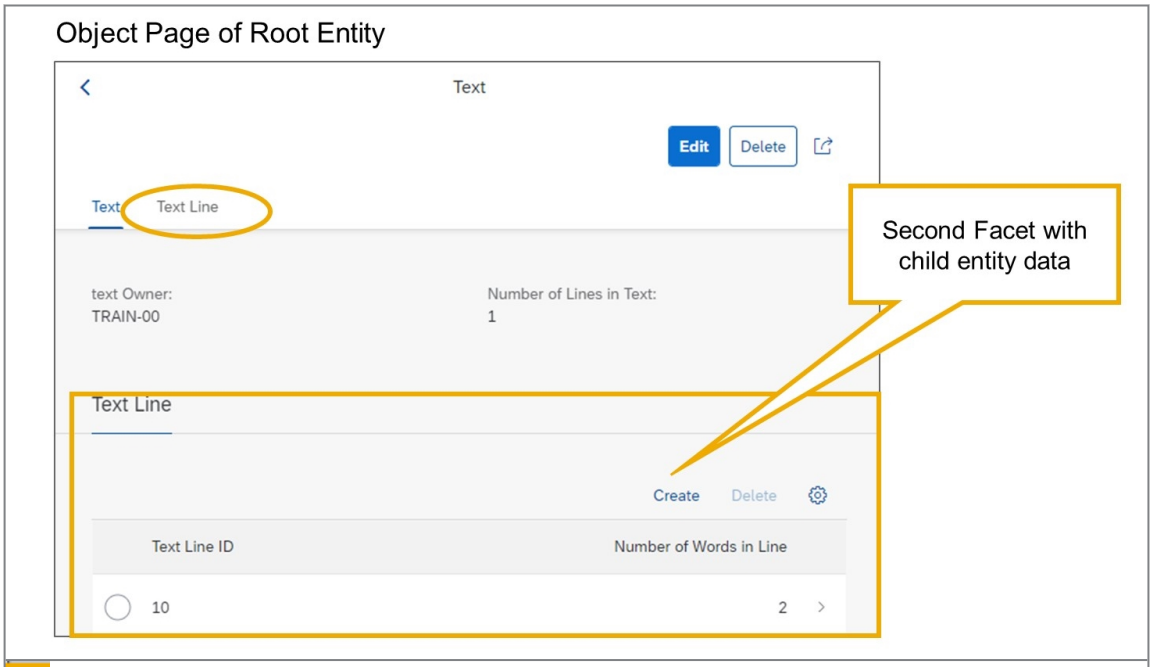

Figure 161: Visualization of Child Entity in Second Facet

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*.

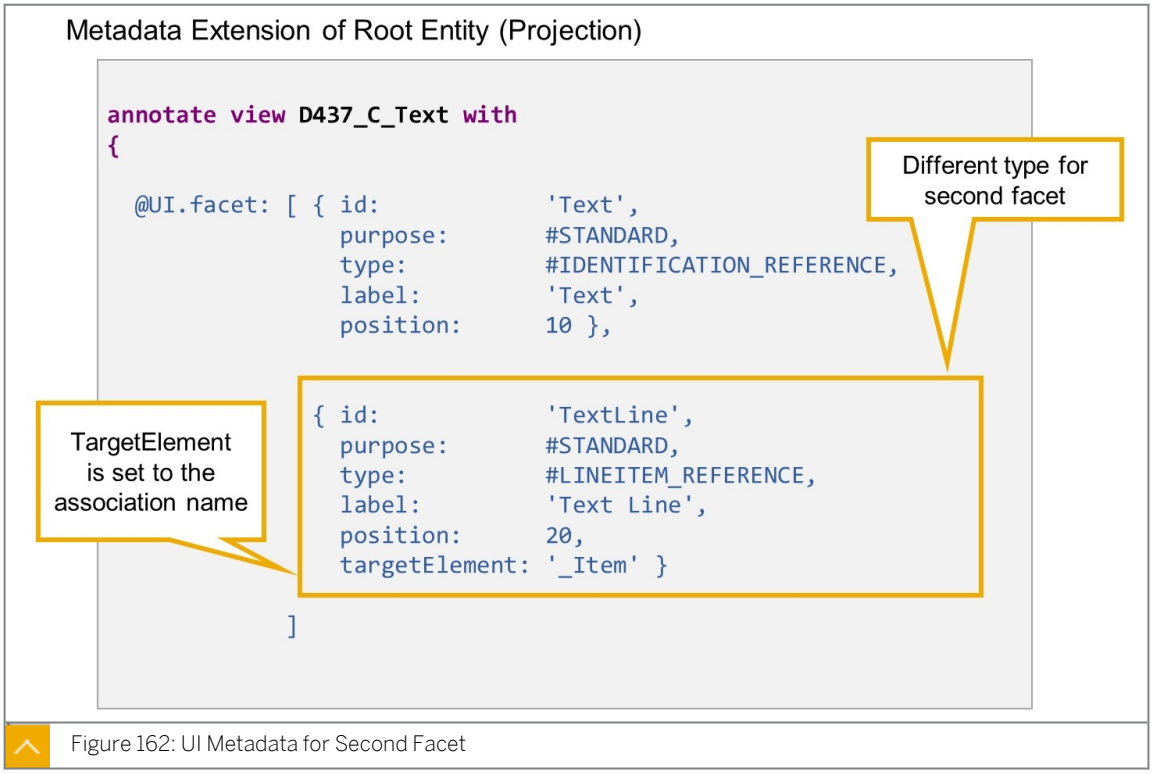

Figure 162: UI Metadata for Second Facet

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.

LESSON SUMMARY

You should now be able to:

- Expose compositions to OData services
- Enable navigation in SAP Fiori elements apps

Unit 5
Lesson 3

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

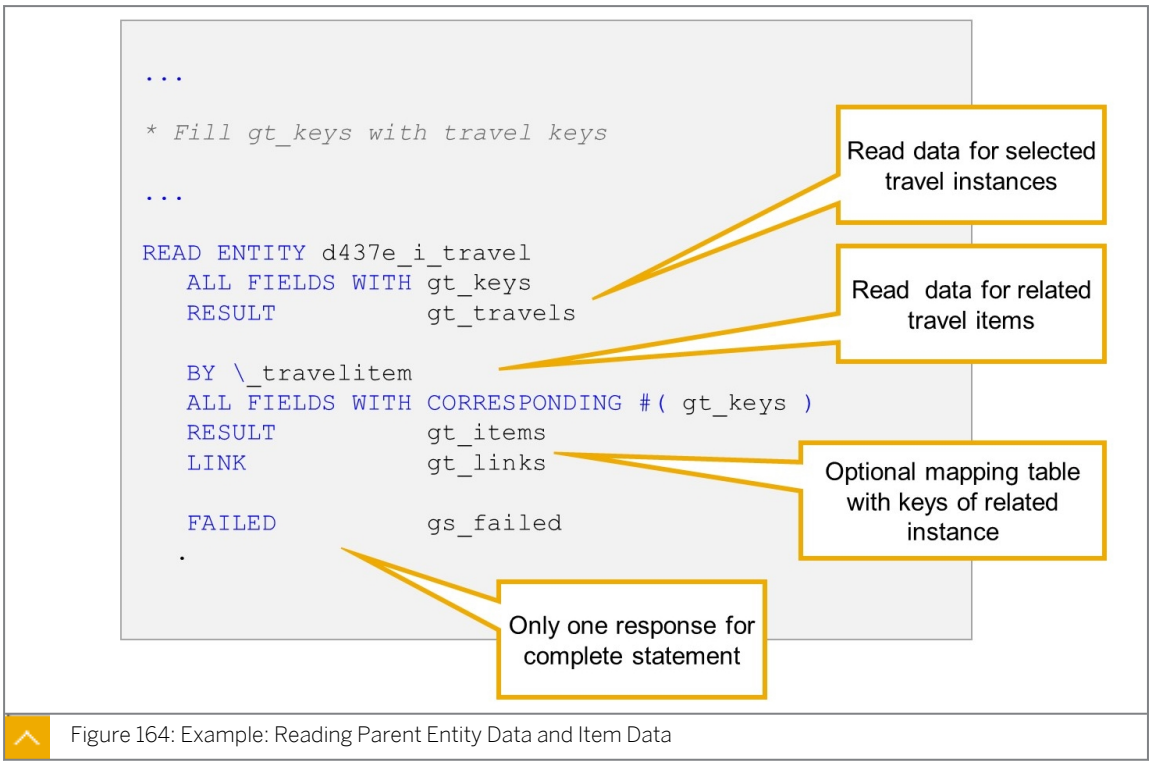

Figure 164: Example: Reading Parent Entity Data and Item Data

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 TravelItem 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 combing 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

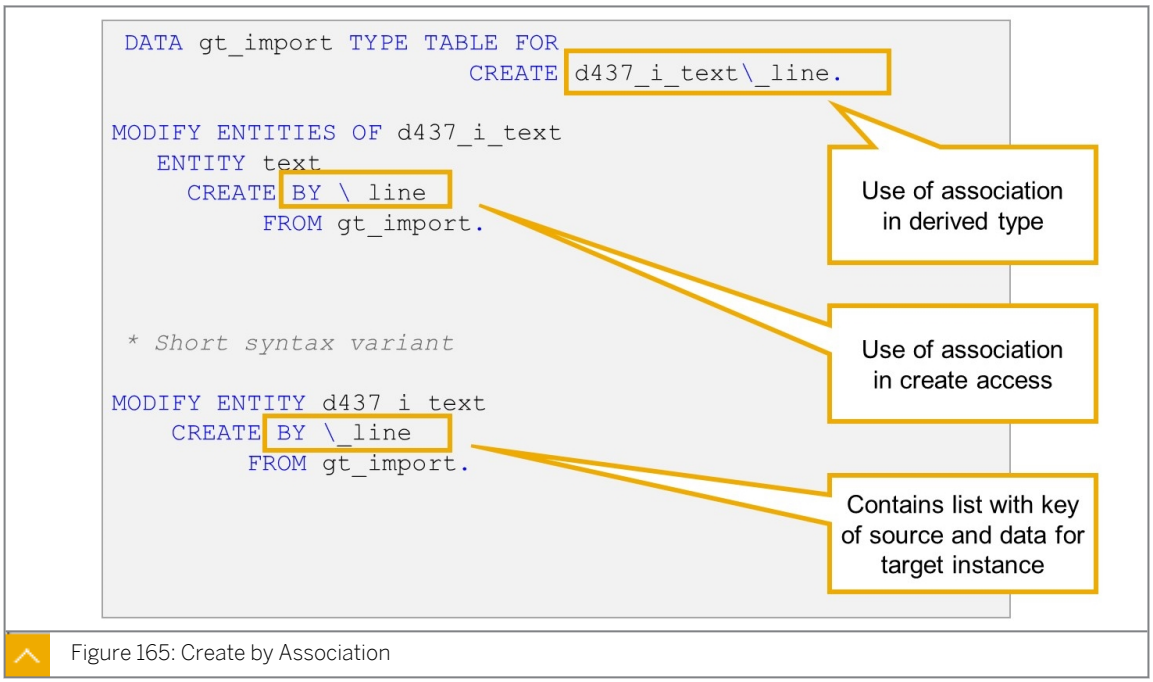

Figure 165: Create by Association

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.

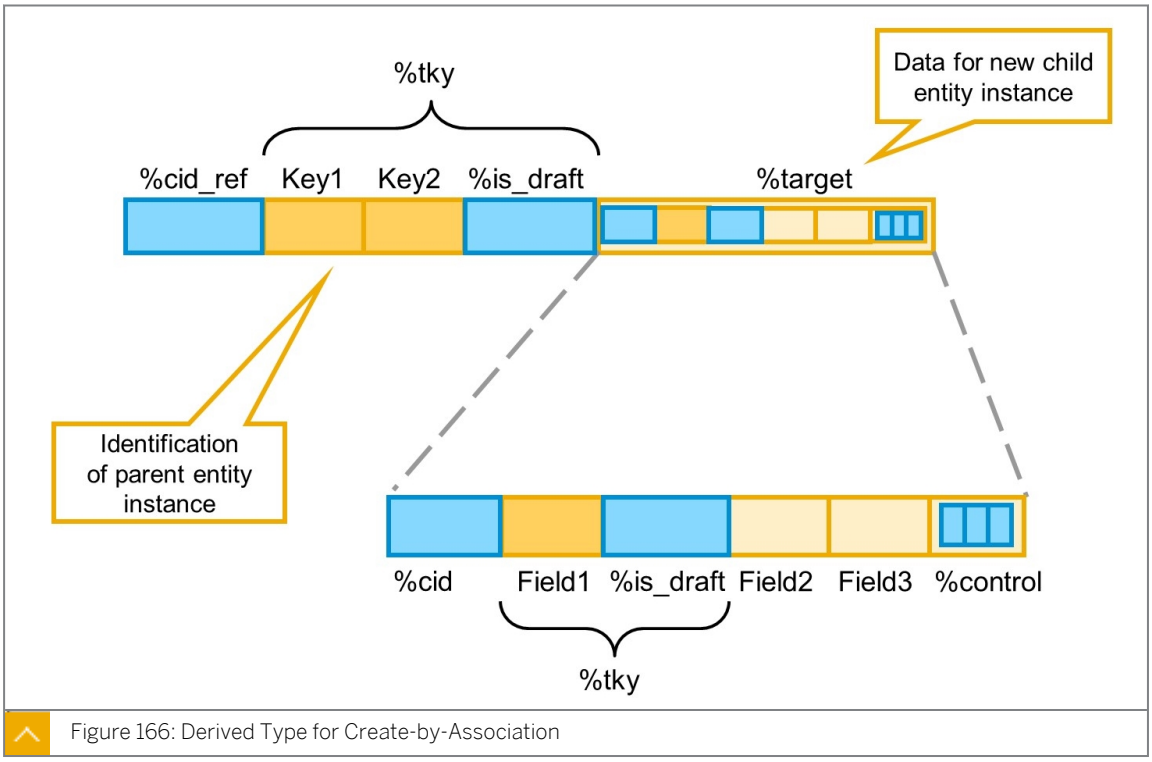

Figure 166: Derived Type for Create-by-Association

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.

LESSON SUMMARY

You should now be able to:

- Access composite business objects with EML

UNIT 6 Transactional Apps with Unmanaged Business Object

UNIT OBJECTIVES

- Define the behavior for an unmanaged Business Object
- Implement data access of an unmanaged Business Object

Unit 6
Lesson 1

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

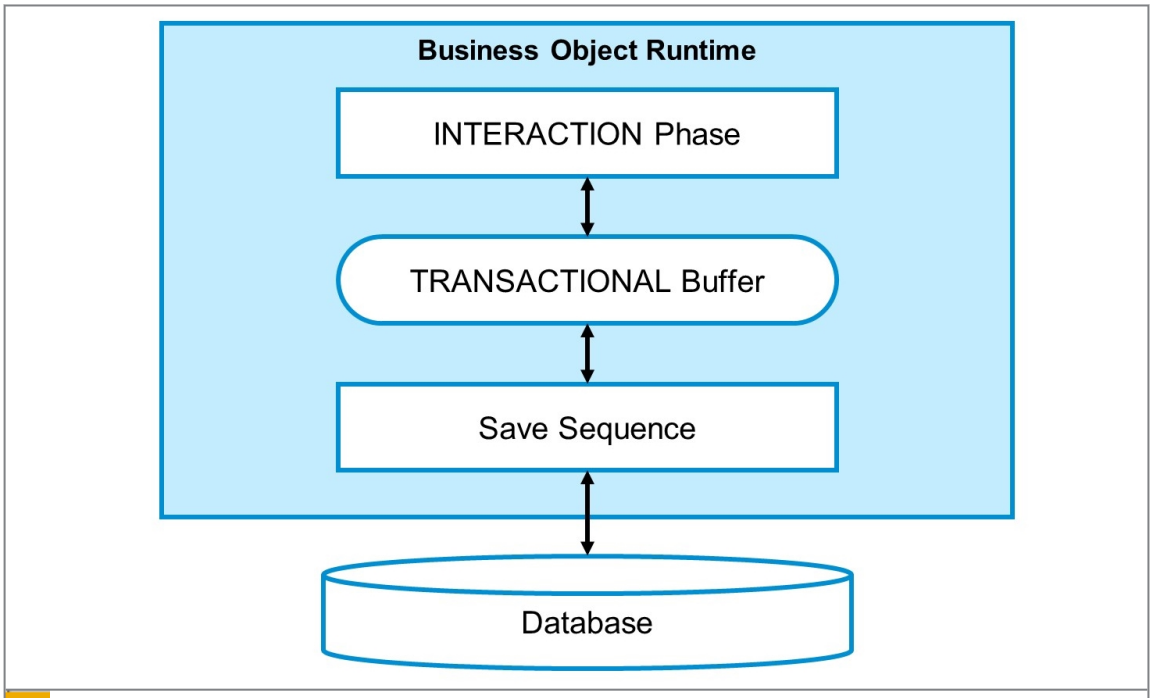

Figure 167: Interaction Phase and Save Sequence

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

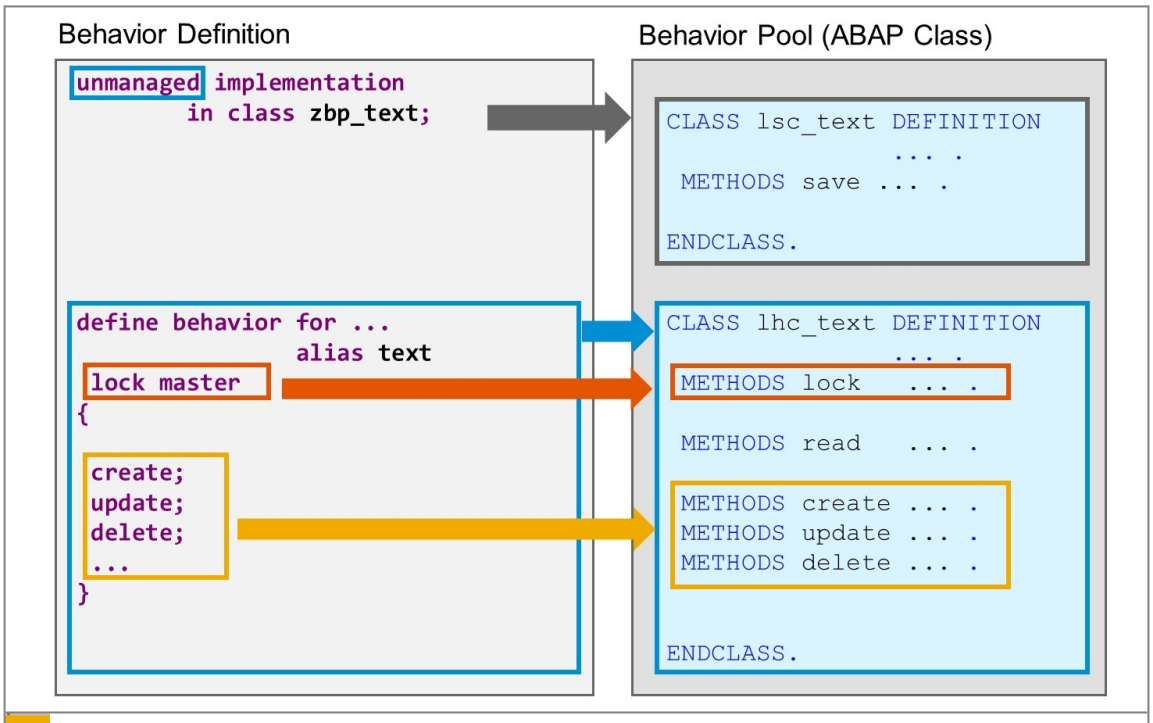

Figure 169: The Behavior Pool of an Unmanaged RAP BO

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.

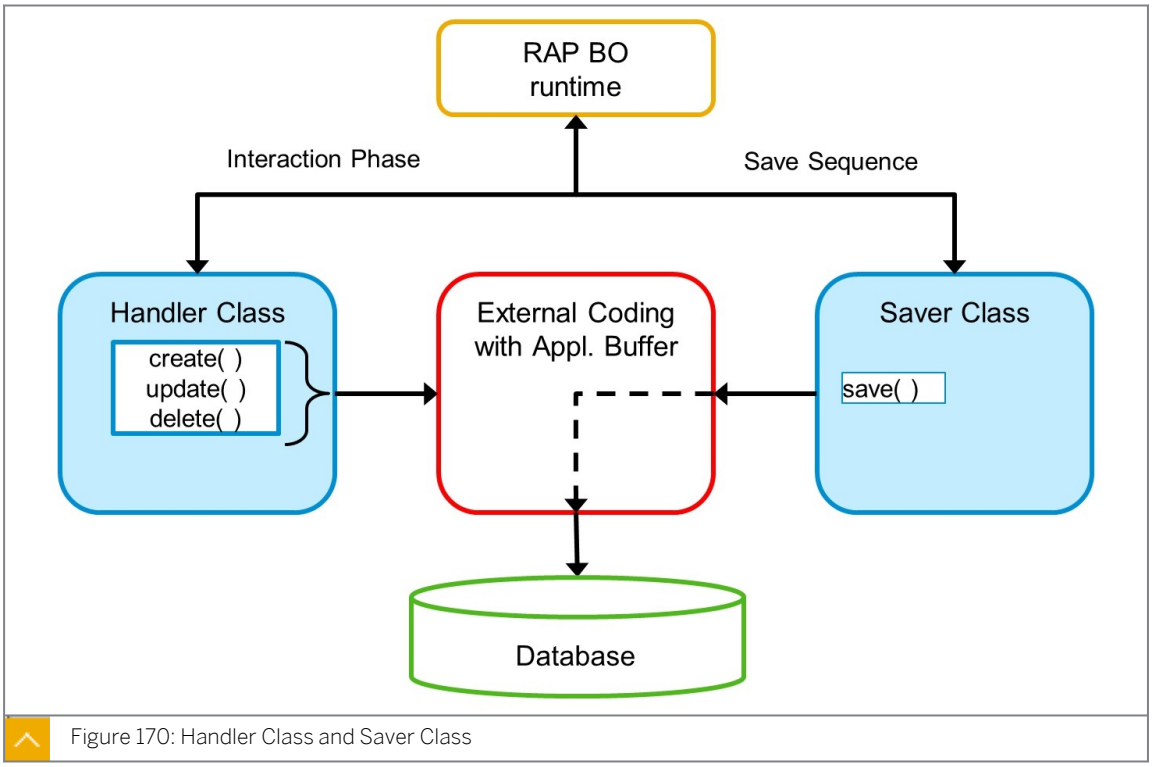

Figure 170: Handler Class and Saver Class

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

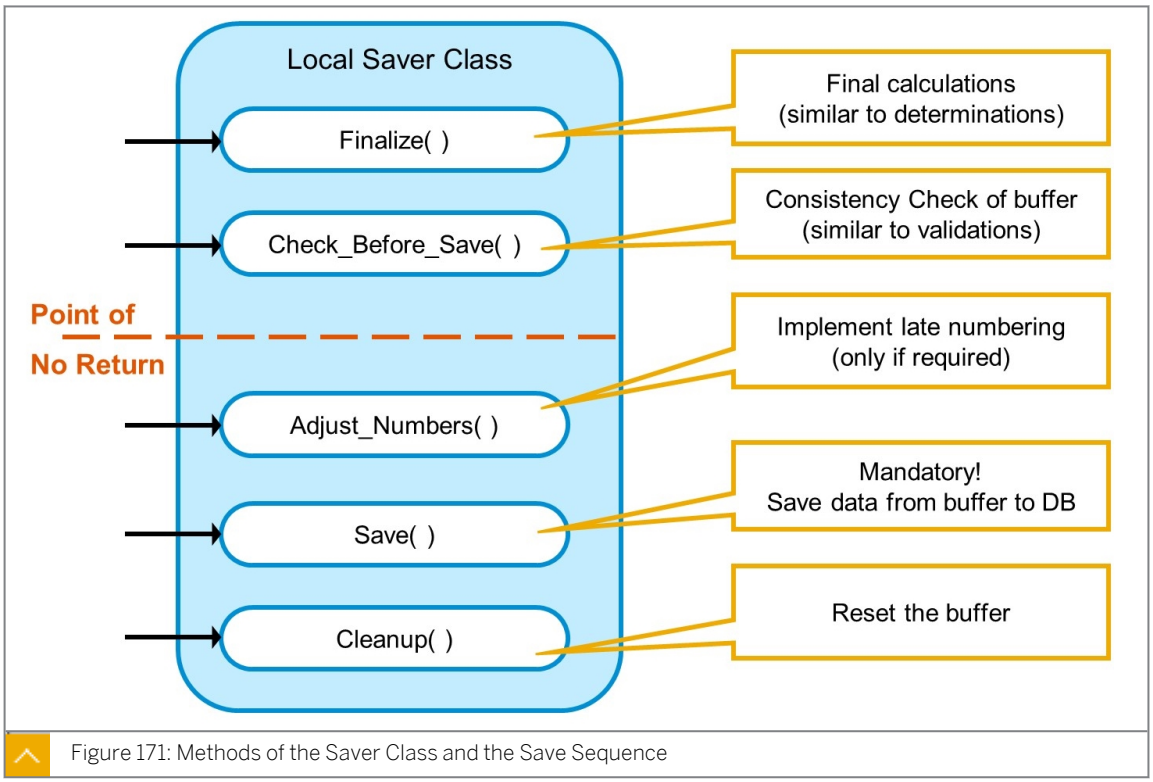

Figure 171: Methods of the Saver Class and the Save Sequence

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.

LESSON SUMMARY

You should now be able to:

- Define the behavior for an unmanaged Business Object

Unit 6
Lesson 2

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

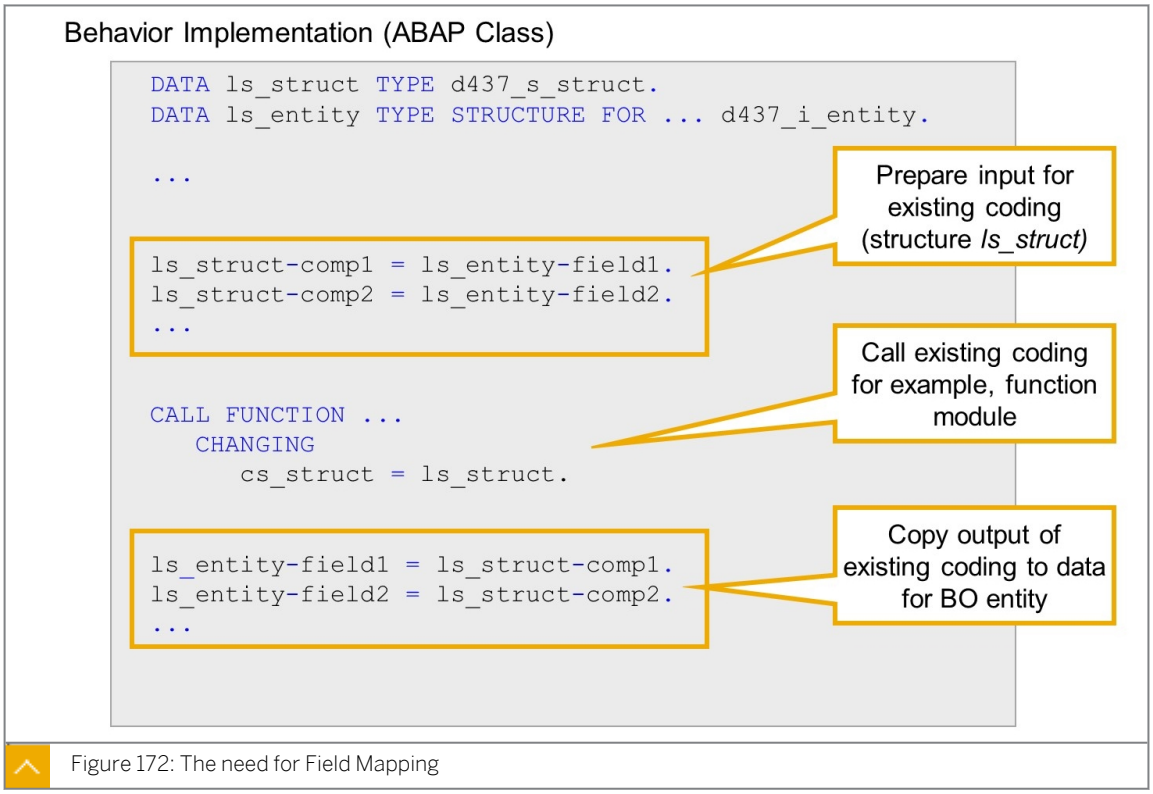

Figure 172: The need for Field Mapping

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.

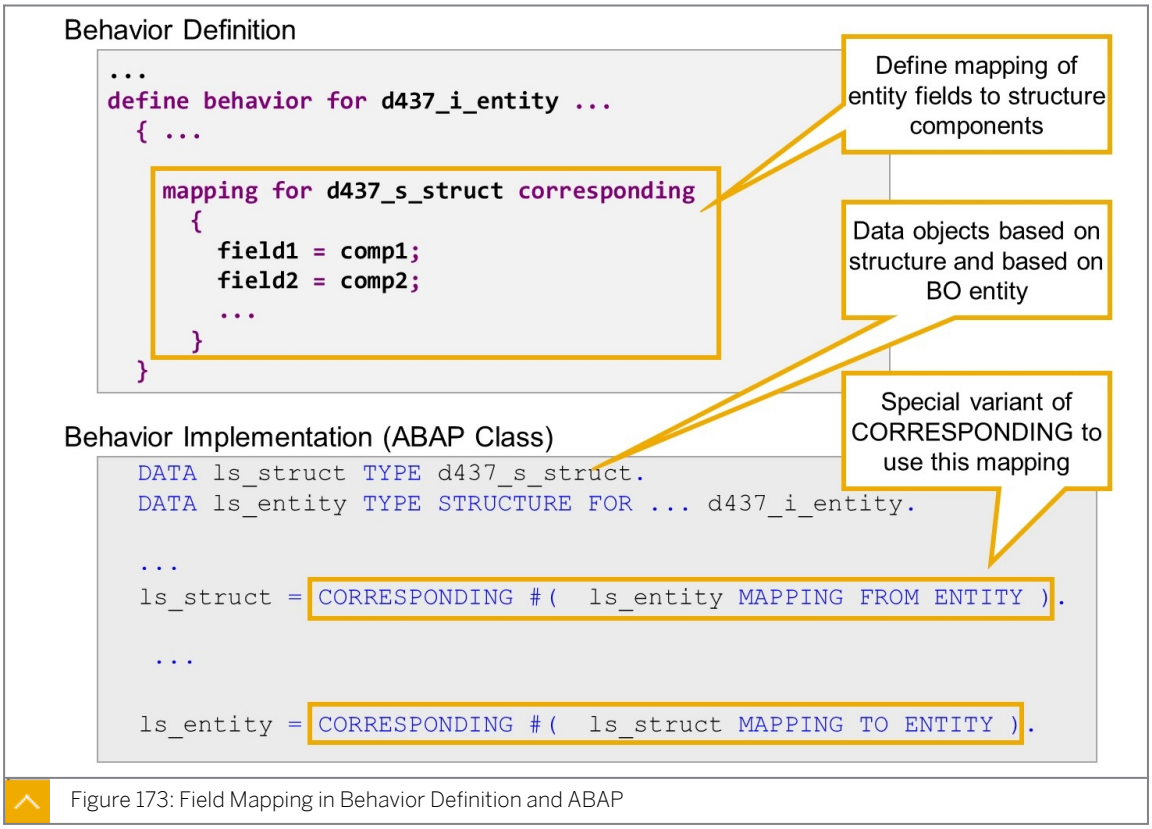

Figure 173: Field Mapping in Behavior Definition and ABAP

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

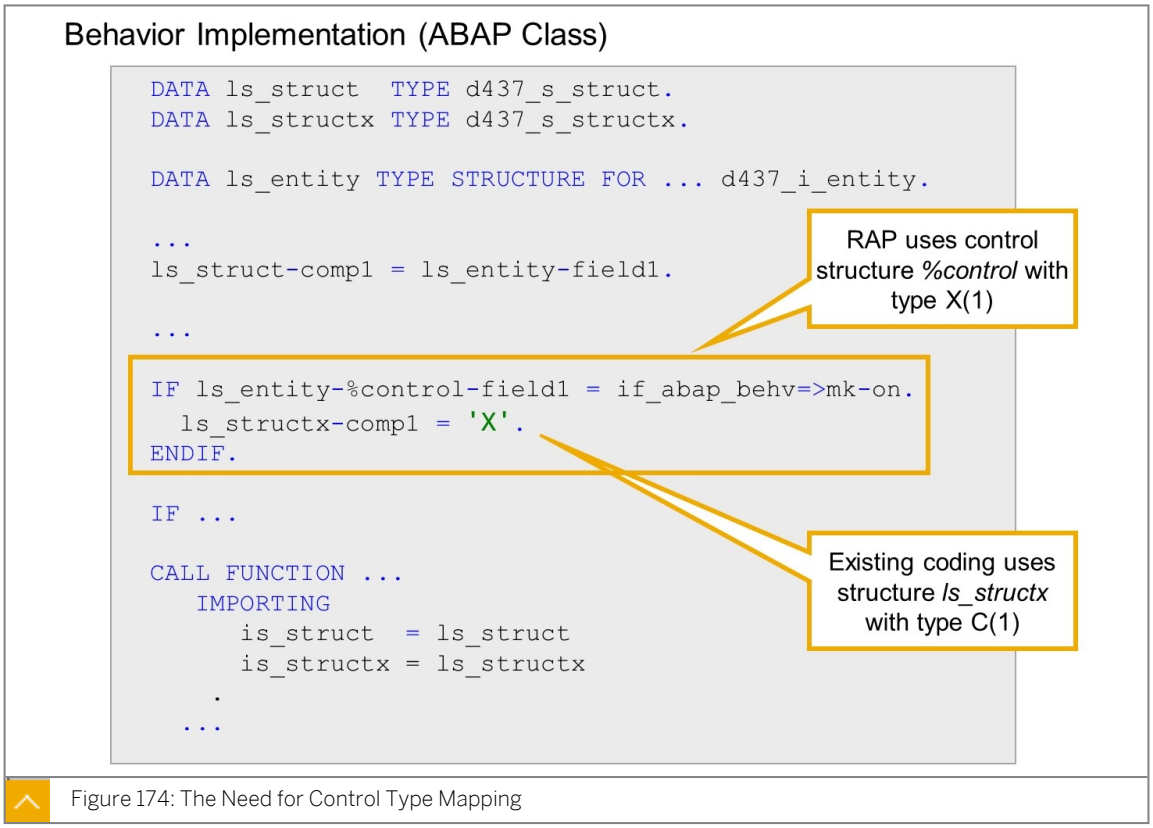

Figure 174: The Need for Control Type Mapping

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 `%control` is mapped to the first component of the legacy control structure `ls_structx`.

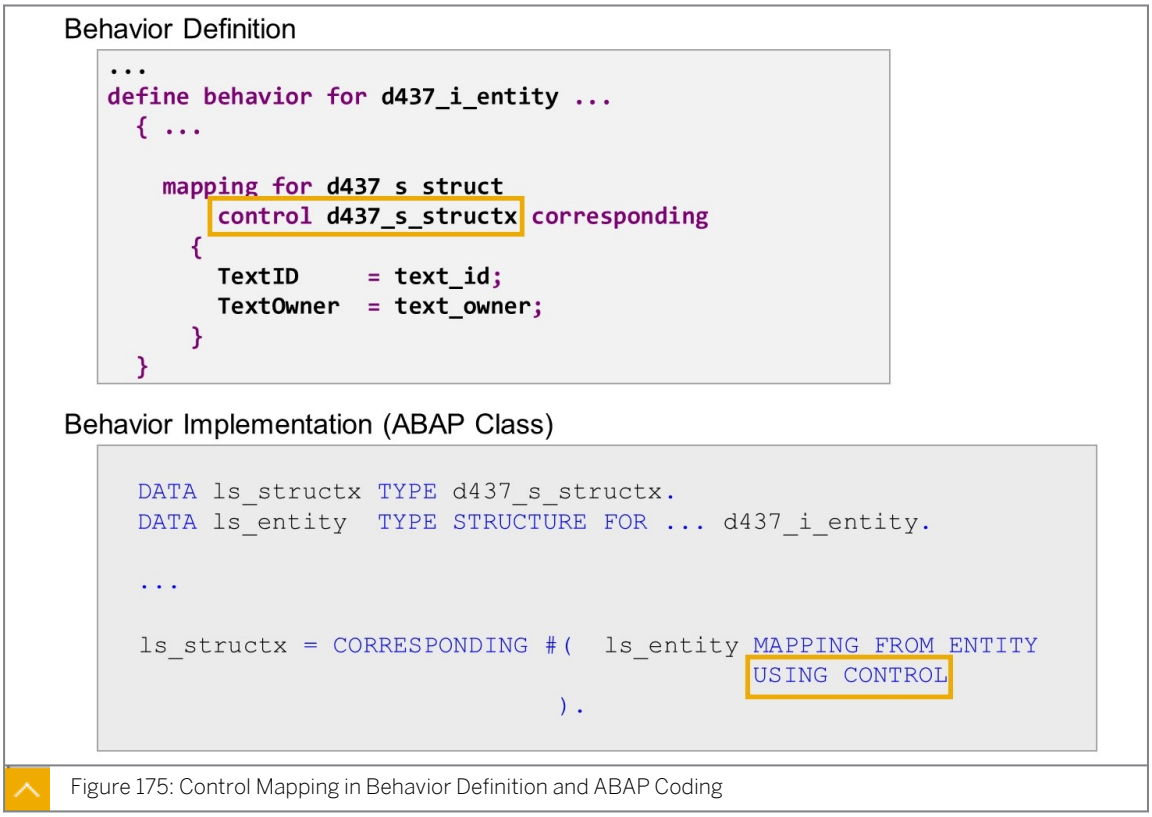

Figure 175: Control Mapping in Behavior Definition and ABAP Coding

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 ).`

LESSON SUMMARY

You should now be able to:

- Implement data access of an unmanaged Business Object