ABAP RESTful Application Programming Model

Generated on: 2024-09-02 07:08:27 GMT+0000

ABAP Cloud | ABAP Cross-Product

PUBLIC

Original content: https://help.sap.com/docs/ABAP_Cloud/f055b8bf582d4f34b91da667bc1fcce6?locale=en-US&state=PRODUCTION&version=sap_cross_product_abap

Warning

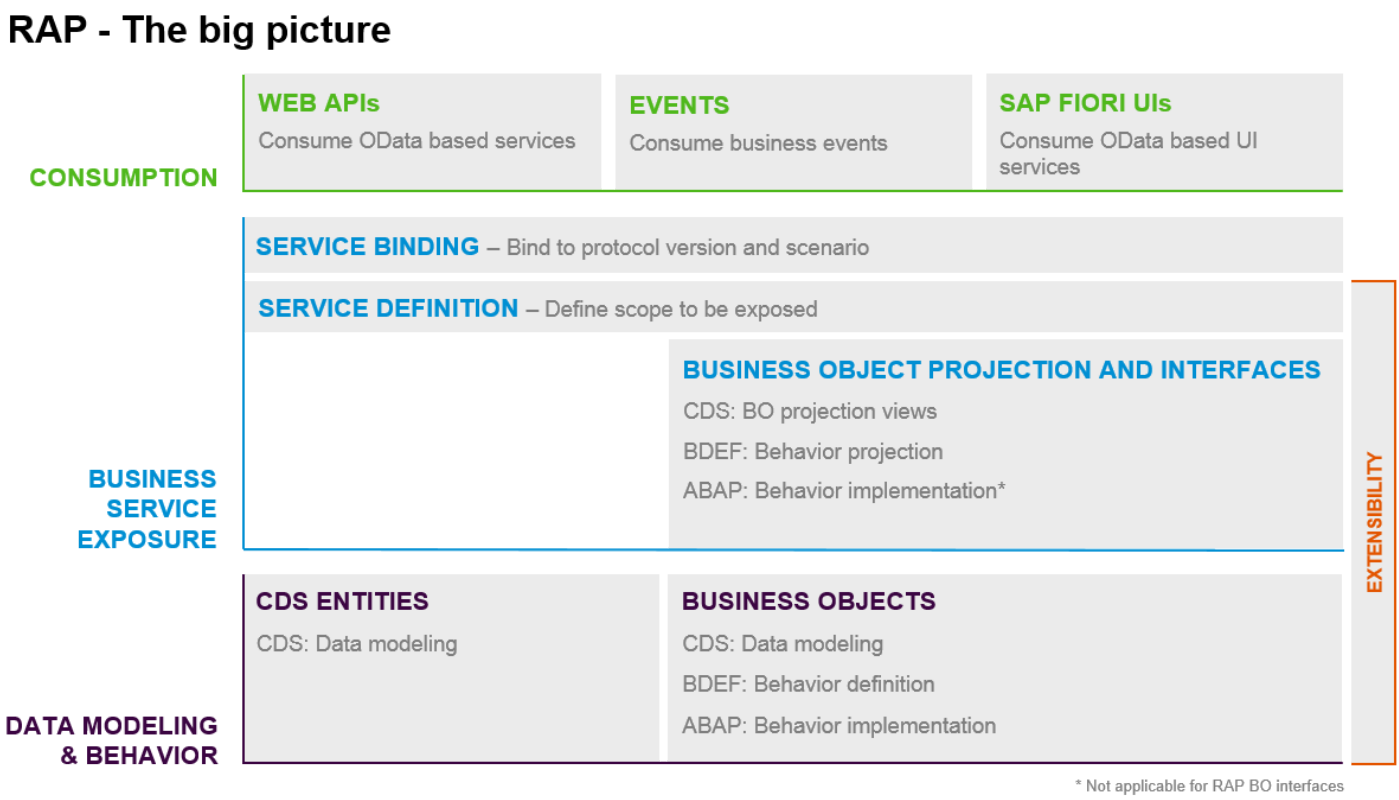
ABAP RESTful Application Programming Model

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

More information on how RAP relates to ABAP Cloud and on its role in the ABAP Cloud development model is available in the ABAP Cloud concepts guide.

Architecture Overview

This image is interactive. Hover over each area for a description. Click highlighted areas for more information.



Please note that image maps are not interactive in PDF output.

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

This image is interactive. Hover over each area for a description. Click highlighted areas for more information.

Please note that image maps are not interactive in PDF output.

ℹ For more information about the evolution of the ABAP programming model, read this blog on the community portal.

ABAP Language Version

The present documentation is based on ABAP for Cloud Development. All described features are applicable using the restricted set of language elements and repository objects that are released for cloud development. For cloud products, only the language version ABAP for Cloud Development is available. In classic ABAP development environments you can choose the ABAP language version. Developing RAP applications with ABAP for Cloud Development is recommended for all ABAP development environments as this is key for your developments to be upgrade-stable, cloud-ready, software-architecture-driven and thus future-proven.

For more information, about the ABAP language versions, see ABAP Language Versions (ABAP Keyword Documentation).

Validity of Documentation

The ABAP RESTful Application Programming Model is available in the following products:

- SAP BTP ABAP environment
- SAP S/4HANA Cloud
- SAP S/4HANA

Note

To highlight the specifics for on-prem releases, the icon is used.

To highlight the specifics for SAP BTP and SAP S/4HANA Cloud releases, the icon is used.

This documentation reflects the latest feature scope of ABAP Development Tools (ADT) that is shipped with the latest backend versions of the supported SAP products. Your actual available feature scope depends on your backend version of the respective SAP product.

Prerequisites

To start developing with the ABAP RESTful Application Programming Model, make sure you meet the prerequisites. See Prerequisites.

Constraints

The current version of the ABAP RESTful Application Programming Model still has some constraints for certain features. For a detailed list, see Development Constraints.

Learn

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

The ABAP RESTful Application Programming Model has unified the development of OData services with ABAP. It is based on three pillars that facilitate your development.

- **Tools:** The approach to integrate all implementation tasks in one development environment optimizes the development flow and offers an end-to-end experience in one tool environment. New development artifacts support the application developer to develop in a standardized way.

- **Language:** The ABAP language has been aligned and extended to support the development with the ABAP RESTful Application Programming Model, together with CDS. The application developer uses typed APIs for standard implementation tasks and benefits from auto-completion, element information, and static code checks.
- **Frameworks:** Powerful frameworks represent another important pillar of the ABAP RESTful Application Programming Model. They assume standard implementation tasks with options for the application developer to use dedicated code exits for application-specific business logic.

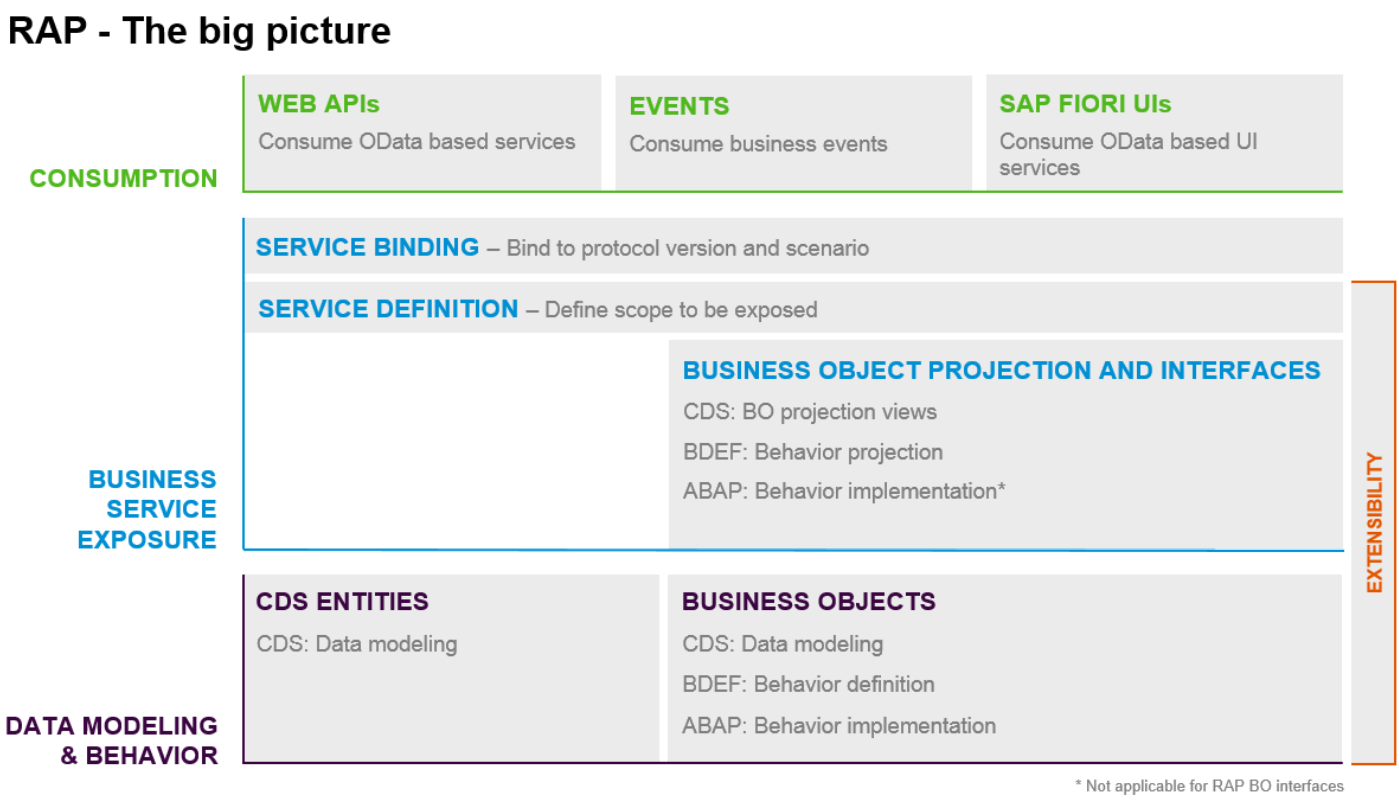
Learn how these pillars are incorporated into the architecture of the ABAP RESTful Application Programming Model in the following topics.

Design Time

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

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

This image is interactive. Hover over each area for a description. Click highlighted areas for more information.



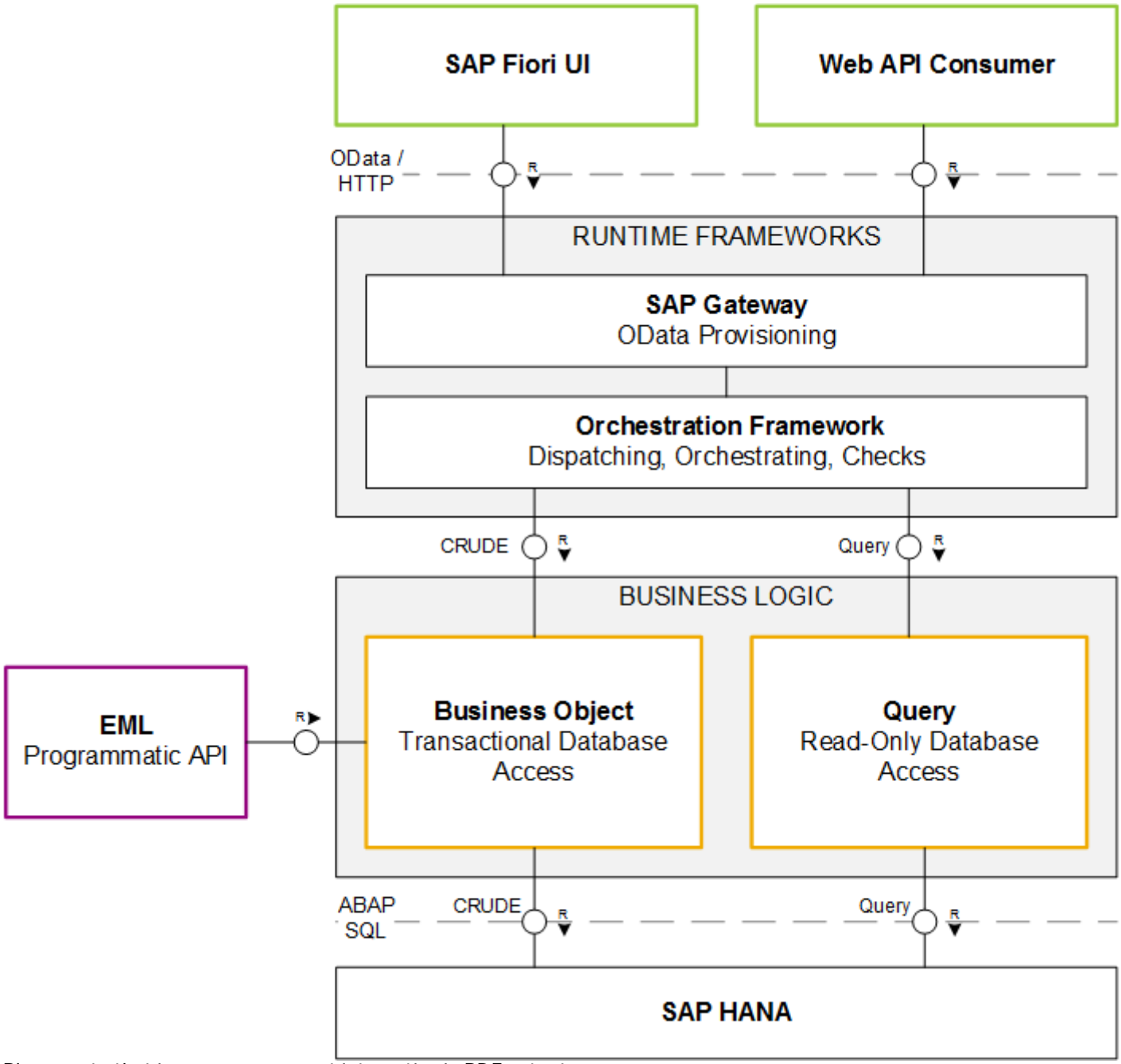
Please note that image maps are not interactive in PDF output.

Runtime

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

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

This image is interactive. Hover over each area for a description. Click highlighted areas for more information.



Please note that image maps are not interactive in PDF output.

A more detailed description is available for the following concepts:

- Data Modeling and Behavior
  - Business Object
  - Business Object Projection
  - Query
- Business Service
  - Service Definition
  - Service Binding
- OData Service Consumption
- Runtime Frameworks

- Entity Manipulation Language (EML)

Data Modeling and Behavior

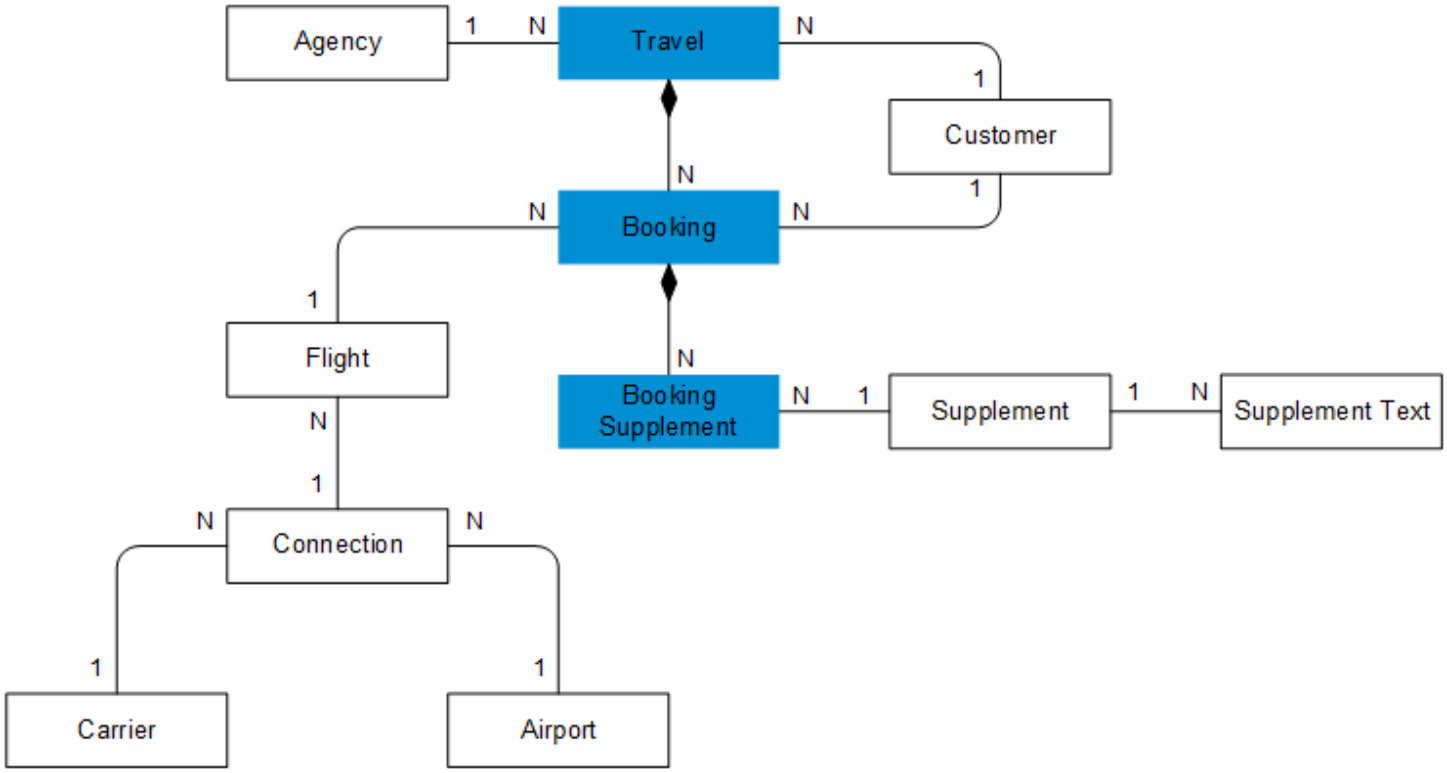
The layer of data modeling and behavior deals with data and the corresponding business logic.

Data Model

The data model comprises the description of the different entities involved in a business scenario, for example `travel` and `booking`, and their relationships, for example the parent-child relationship between `travel` and `booking`. The ABAP RESTful Programming Model uses CDS to define and organize the data model. CDS provides a framework for defining and consuming semantic data models. Every real-world entity is represented by one CDS entity. View building capabilities allow you to define application-specific characteristics in the data model. That means, CDS entities are the fundamental building blocks for your application. When using the CDS entity for a data selection, the data access is executed by the SQL-view, which is defined in the CDS entity.

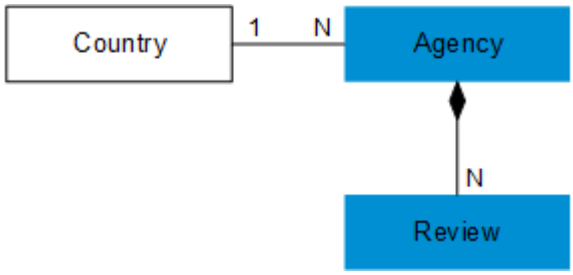
Depending on the use case, data models support transactional access or query access to the database. Thus, data models are used in business objects or queries respectively.

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



Data Model Used in the Development Guides of this Documentation

The following diagram shows the data model used in development guide Extend to showcase the possibilities for extending existing business objects.



Behavior

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

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

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

Business Object

Query

Runtime Frameworks

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

SAP Gateway

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

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

RAP Runtime Engine

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

If locks are implemented, the RAP runtime engine executes first instance-independent checks and sets locks. For the eTag handling, the framework calls the necessary methods before the actual request is executed.

ℹ Note

The RAP runtime engine is also known under the name SADL (Service Adaptation Description Language). Apart from the runtime orchestration, the SADL framework is also responsible for essential parts in the query and BO runtime.

Examples

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

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

Business Object

Introduction

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

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

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

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

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

Structure of a Business Object

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

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

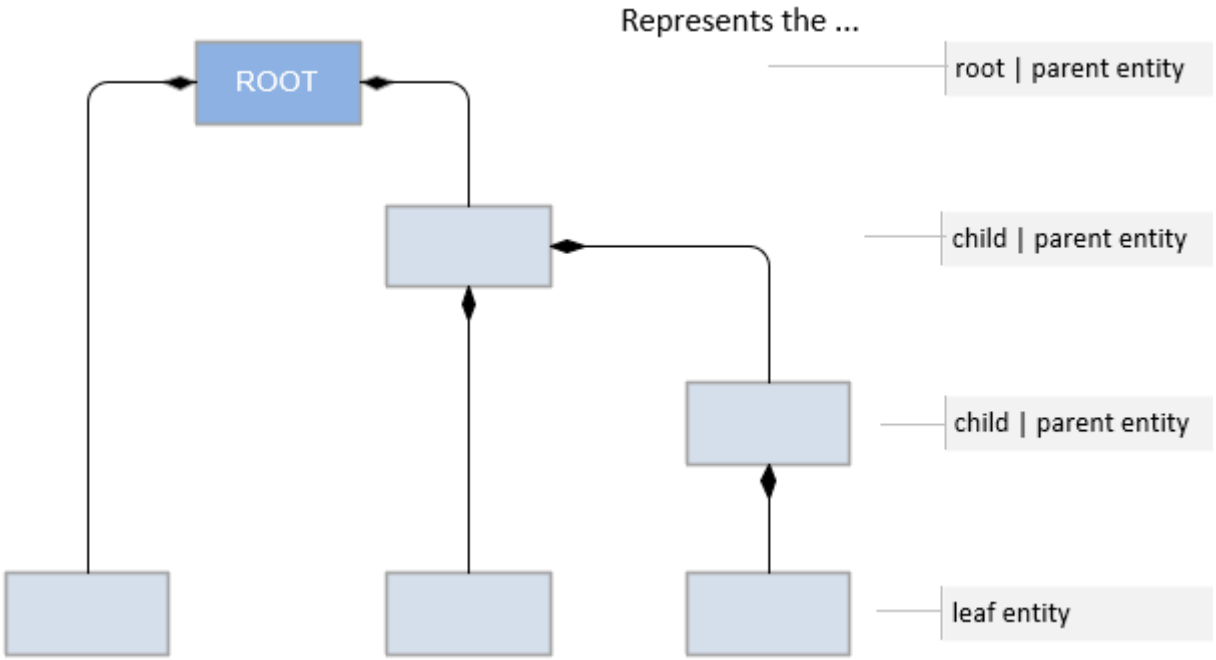
The root entity is of particular importance in a composition tree: The root entity serves as a representation of the business object and defines the top node within a hierarchy in a business object's structure. This is considered in the source code of the CDS data definition with the keyword `ROOT`.

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

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

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

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



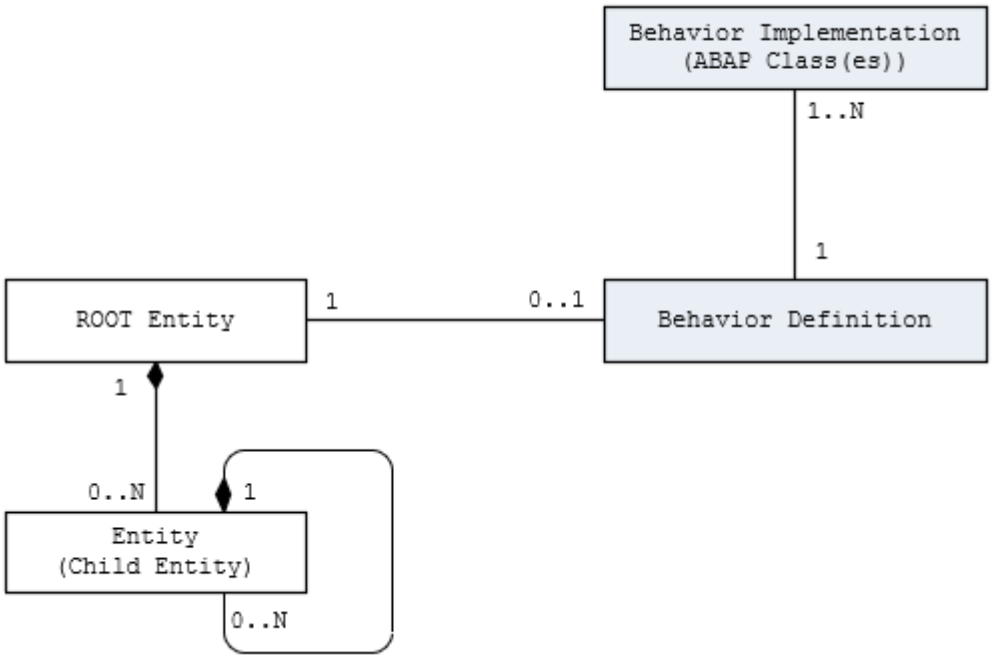
Composition Tree Reflects the Structure of a Business Object

Behavior of a Business Object

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

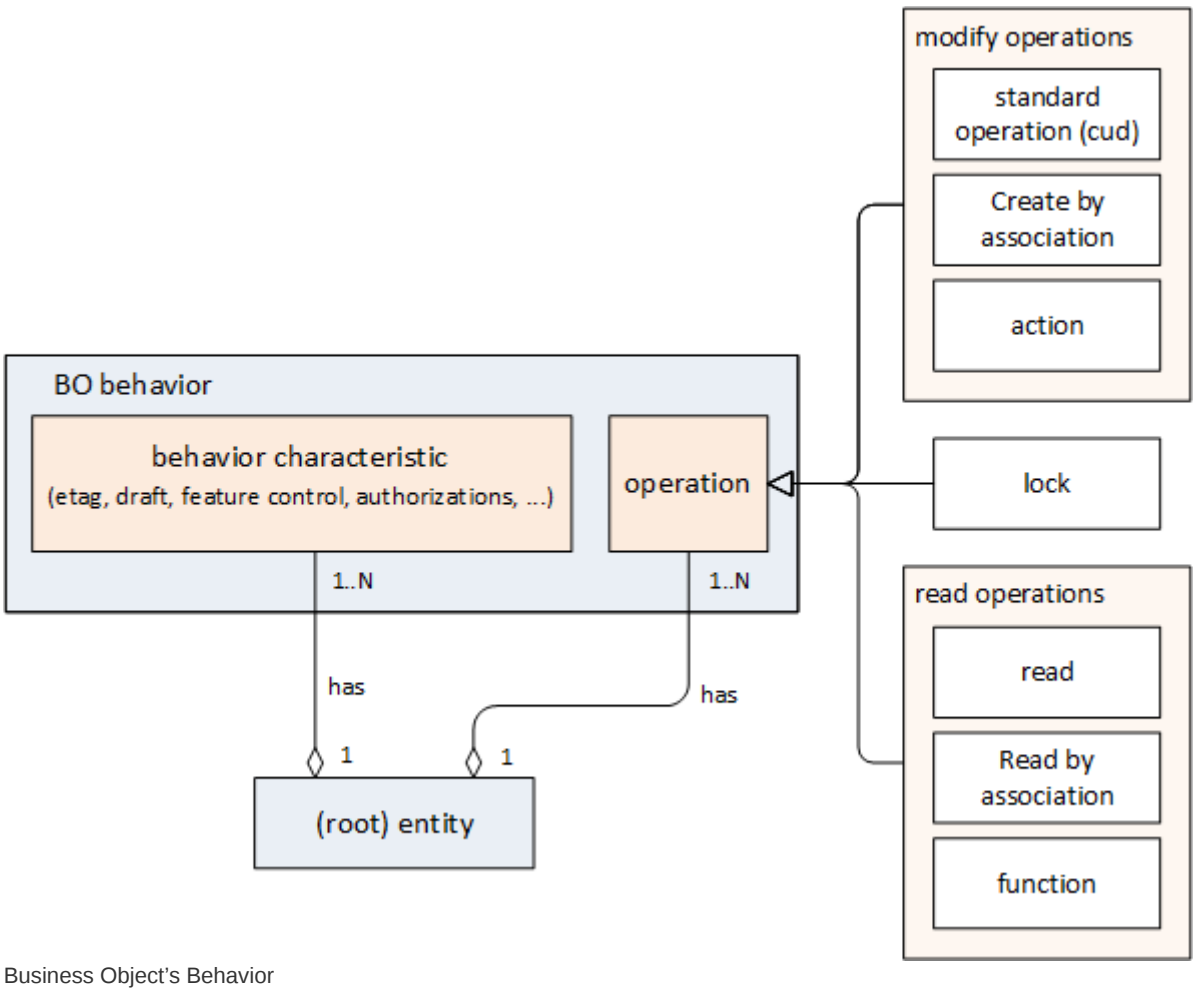
A behavior definition always refers to a CDS data model. As shown in the figure below, a behavior definition relies directly on the CDS root entity. One behavior definition refers exactly to one root entity and one CDS root entity has at most one behavior

definition (a 0..1 cardinality), which also handles all included child entities that are included in the composition tree. The implementation of a behavior definition can be done in a single ABAP class (behavior pool) or can be split between an arbitrary set of ABAP classes (behavior pools). The application developer can assign any number of behavior pools to a behavior definition (1..N cardinality).



Relationship Between the CDS Entities and the Business Object Behavior

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



Business Object's Behavior

Behavior Characteristic

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

ETag

Draft handling

Feature control

Numbering

Authorization Control

Apart from draft capabilities, these characteristics can be defined for each entity separately.

Operations

Each entity of a business object can offer a set of operations. They can cause business data changes that are performed within a transactional life cycle of the business object. As depicted in the diagram above, these modify operations include the standard operations create(), update() and delete() as well as lock implementations and application-specific operations with a dedicated input and output structure which are called actions. Another kind of operations are the read operations: they do not change any business data in the context of a business object behavior. Read operations include read, read by association, and functions (that are similar to actions, however, without causing any side effects).

For more information, see

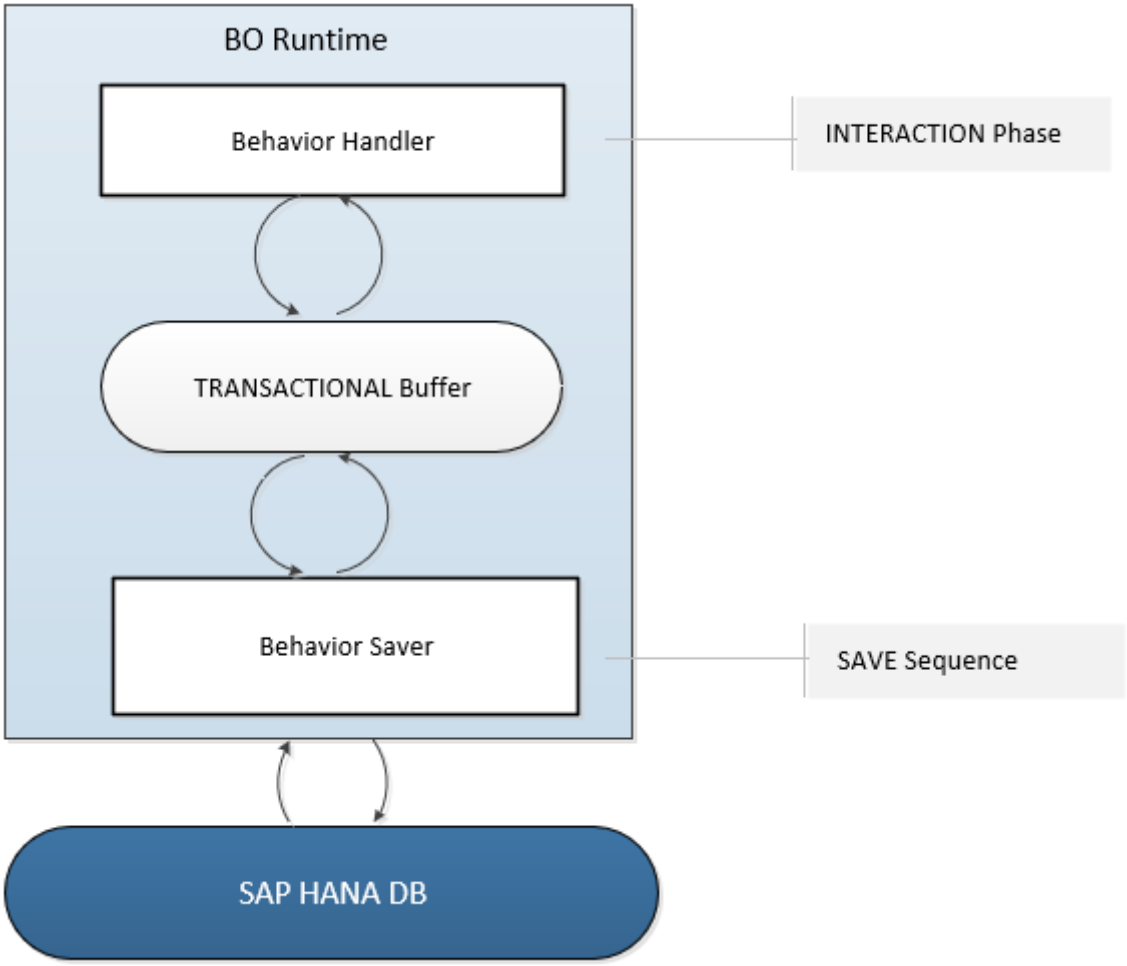
- Standard Operations
- Actions
- Locking

Business Object's Runtime

The business object runtime mainly consists of two parts:

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

For more information about the BO runtime within one LUW, see The RAP Transactional Model and the SAP LUW.



BO Runtime

For each operation the transactional runtime is described in detail in the respective runtime diagrams. See Operations.

The save sequence has the same structure for each operation. For more information, see Save Sequence Runtime.

Instantiation of Handler and Saver Classes

In general, the instantiation of handler and saver classes is tightly coupled to an ABAP session. Instances live as long as the ABAP session and they can exist across LUW borders. However, their life cycle depends on the way their methods are called. Nested method invocations always provoke the reinstantiation of the local handler and saver classes. For example, this is the case if a handler method executes an EML modify request in local mode to call another method of the same handler class, or if other BOs are involved in the invocation chain.

> **Note**
>
> Do not work with data in member variables for keeping data for subsequent method invocation. The data is lost once the handler class is reinstantiated.

The saver class is always instantiated once a corresponding handler method is called.

Released Business Objects

`Released Business Objects` are RAP business objects of which some components are released with a release contract that ensures lifecycle stability for the respective business object for consumers.

SAP releases business object parts for consumption in custom implementations with the `C1` release contract to ensure lifecycle stability and upgrade-safety. The details about consuming released BOs is described in Business Object Consumption.

Business Object Implementation Types

Different BO implementation types offer various options to create your development scenario with the ABAP RESTful Application Programming Model.

General

The RAP framework offers different implementation types that are optimized to match your needs as application developer. Depending on the initial situation of your development, you can choose the RAP development approach to easily exploit the many advantages of programming with RAP.

The main approaches are the two standard implementation types of RAP business objects, **managed** or **unmanaged**. Both approaches are based on a data model that is defined in CDS view entities. The difference between the two implementation types lies in the provisioning of the BO's transactional behavior. It is mainly the level of free-style development for your specific RAP application that determines which of the implementation type is the best option for you. Whereas the standard behavior is provided by a RAP managed provider when using the implementation type managed, it's the developer's task to implement the complete behavior in the unmanaged implementation type.

Business Objects with both implementation types can be exposed as OData services for UIs or Web APIs.

Managed Implementation Type

The managed implementation type addresses use cases where all essential parts of an application must be developed from scratch. However, these new applications can highly benefit from out-of-the-box support for transactional processing. Standard operations (create, update, delete) must only be specified in the behavior definition to obtain a ready-to-run business object. In addition the provisioning and handling of the transactional buffer is automatically done for you. The technical implementation aspects are taken over by the managed RAP BO provider. The interaction phase and the save sequence are implemented generically. The application developer can then focus on business logic that is implemented using actions, validations and determinations and user interaction. The corresponding transactional engine manages the entire life cycle of your business object and covers all aspects of your business application development.

The managed implementation type is aimed for building new applications from scratch without any already existing code for business logic. The standard functionality that is given in managed business objects is easily retrieved. The business logic is implemented using predefined implementation methods that are integrated during the runtime in the interaction phase and save sequence. On the other hand, integrating existing business logic in the standard functionality of managed business objects can hardly be achieved.

A complete developer guide on how to develop a new application from scratch is provided in Developing Managed Transactional Apps.

Managed Implementation Type with Unmanaged or Additional Save

Managed business objects fully provide the managed behavior during the interaction phase, as well as the saving procedure during the save sequence. In some cases, however, it might be necessary to implement a different saving option, be it because you need to save to a different database table, or you want to add an additional saving procedure. These use cases are supported in managed business objects that support **unmanaged** or **additional save**. Whereas the interaction phase is still completely managed by the RAP managed provider, the save sequence can be modified and business logic for saving can be integrated in the corresponding saver methods.

The developer guide for managed business objects provides a description on how to implement additional or unmanaged save options. See Integrating Additional Save in Managed Business Objects and Integrating Unmanaged Save in Managed Business Objects.

Unmanaged Implementation Type

In unmanaged business objects, the application developer must implement essential components of the REST contract. There is no standard functionality provided by the RAP framework. It is only the frame of a business object, that an application developer needs to adhere to. The behavior must be specified in the behavior definition but implemented manually in ABAP classes in the behavior pool. All existing business logic can be integrated, including standard behavior like create, update and delete. In addition, the transactional buffer must be provided and handled by the application developer in the ABAP behavior pool.

This implementation type is mainly aimed at application developers that already have their business logic encapsulated in functional modules. These function modules can then be integrated in the corresponding behavior methods in the RAP BO. Like this, you can reuse existing business logic in your RAP business objects and you still benefit from the standardized RAP runtime orchestration to easily create a RAP service.

A complete developer guide on how to develop an application which integrates existing legacy logic is given in Developing Unmanaged Transactional Apps.

Additional Implementation Options

Depending on the targeted RAP service that you want to create, there are several additional implementation options that can be included in the aforementioned implementation types.

Query Features

RAP services include sophisticated querying features, such as filtering, sorting, search capabilities, aggregation options, or value helps. Some standard features are provided by the RAP query engine and are given from scratch when building a RAP service. Business logic dependent features, such as search or value helps must be implemented using annotations in CDS and its strong data modeling capabilities.

Most of the read-only features are described in Developing Read-Only List Reporting Apps.

Draft Capabilities

To build end-user applications with stateless functionality that enables using the application independently of time or device, the RAP framework offers draft capabilities. Using the draft concept in RAP applications means persisting the state of the transactional buffer on a database table. This prevents data loss and stores all changes of a current user interaction until it is activated. The draft concept is enabled in RAP business objects in the behavior definition and is managed by the managed draft provider. The application developer does not have to handle draft processing.

A complete developer guide on how to develop an application with draft is given in Developing Transactional Apps with Draft Capabilities.

BOPF managed Implementation Type

For business objects built with the predecessor programming model a RAP integration is offered. This implementation type therefore addresses application developers who want to reuse and integrate BOs built with ABAP Programming Model for SAP Fiori. The CDS based BOPF BOs (Business Object Programming Framework) can be migrated using the Migration Tool, which is included in the ABAP Development Tools (ADT).

Once the CDS based BOPF BOs have been migrated, the application developer does not need to reimplement the essential components of the REST contract - comparable to managed BOs. Instead, the standard and application-specific operations delegated to the BOPF framework mostly remain in their former implementation classes and therefore can be reused.

Information about the migration tool and the migration process can be found in the Migration of CDS-based BOPF Business Objects guide.

RAP BO Behavior

Numbering

Concurrency Control

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

Draft

You can draft-enable a business object to automatically persist transactional data in the backend. This approach supports stateless communication for your applications.

Authorization Control

Authorization control in RAP protects your business object against unauthorized access to data.

Operations

This section describes the operations that are available for RAP business objects and their functionality.

Determinations

A determination is an optional part of the business object behavior that modifies instances of business objects based on trigger conditions.

Validations

A validation is an optional part of the business object behavior that checks the consistency of business object instances based on trigger conditions.

Determine Actions

Feature Control

This topic is about the concept of feature control for the ABAP RESTful application development.

Side Effects

Side effects are used to reload data, permissions, or messages or trigger determine actions based on data changes in UI scenarios with draft-enabled BOs.

Business Events

Save Options

Numbering

About Numbering

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

The primary key of a business object entity can be composed of one or more key fields, which are identified by the keyword `key` in the underlying CDS view of the business object. The set of primary key fields uniquely identify each instance of a business object. The primary key value of a business object instance cannot be changed after the `CREATE`.

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

Early and Late Numbering

In an early numbering scenario, the primary key value is set instantly after the modify request for the `CREATE` is executed. The key values can be passed externally by the consumer or can be set internally by the framework or an implementation of the `FOR`

NUMBERING method. For more information, see Early Numbering.

In a late numbering scenario, the key values are always assigned internally without consumer interaction after the point of no return in the interaction phase has passed, and the SAVE sequence is triggered. This ensures a gap free key value assignment like it is required for example for invoices. For more information, see Late Numbering.

Managed and Unmanaged Numbering

In a managed numbering scenario, the primary key values are assigned automatically by the framework without any additional implementation in the behavior pool, like with numbering: managed in UUID scenarios.

For more information, refer to Managed Internal Early Numbering.

In an unmanaged numbering scenario, an additional implementation is necessary for assigning the primary key values, like with early numbering or late numering that must be implemented in the respective saver methods in the behavior pool.

For more information, see:

- Unmanaged Internal Early Numbering
- Unmanaged Internal Late Numbering

Early Numbering
Late Numbering
Uniqueness Check for Primary Keys

Early Numbering

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

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

External Early Numbering
Internal Early Numbering
<method> FOR NUMBERING

Related Information

Numbering
Late Numbering

External Early Numbering

External Early Numbering

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

In this scenario, it must be ensured that the primary key value given by the consumer is uniquely identifiable. New instances with an already existing primary key value are rejected during the save sequence. The ABAP RESTful Application Programming Model supports uniqueness checks in early stages of the interaction phase to give the end user an early feedback.

For more information, see Uniqueness Check for Primary Keys.

Implementation

For external numbering, you must ensure that the primary key fields are not read-only at CREATE. Otherwise, the consumer cannot provide any value for the primary key field. The RAP framework offers dynamic primary key handling for this scenario. To ensure that the primary key fields are filled when creating new instances, but are read-only on further processing these instances, you can use operation-dependent field access restrictions in the behavior definition.

For more information, see Dynamic Feature Control.

```
...
define behavior for Entity [alias AliasedName]
{  ...
 field (mandatory:create| readonly:update);
  ...}
```

For more information about the syntax, see CDS-BDL - mandatory:create (ABAP Keyword Documentation) and CDS BDL - readonly:update.

Optional External Early Numbering

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

ℹ Note

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

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

Implementation

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

For details about the syntax, see CDS BDL - early numbering (ABAP Keyword Documentation).

Internal Early Numbering

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

Managed Internal Early Numbering

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

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

For more information, see Automatically Drawing Primary Key Values in Managed BOs.

Implementation

Managed numbering is defined in the behavior definition. Additionally, the key field must be defined as `readonly`.

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

For more information about the syntax, see CDS BDL - field numbering (ABAP Keyword Documentation).

Unmanaged Internal Early Numbering

Unmanaged Numbering in Managed Business Objects and Unmanaged Business Objects with managed Draft

Unmanaged Early Numbering in Unmanaged Implementation Scenarios

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

Implementation

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

For more information, see Implementing the CREATE Operation for Travel Instances.

Unmanaged Early Numbering in Managed and Unmanaged Implementation Scenarios with Draft Capabilities

Unmanaged Early Numbering in Managed and Unmanaged Implementation Scenarios with Draft Capabilities

Early numbering is defined in the behavior definition on entity level and the key assignment is implemented in the corresponding handler method which is called during the CREATE or CREATE-BY-ASSOCIATION. Unmanaged internal numbering is then

applied to all primary key fields of a BO entity and not on individual field level. Consequently, you must map all key fields in your implementation in the MAPPED result parameter of the FOR NUMBERING method .

Unmanaged numbering is defined in the behavior definition with early numbering on entity level. For scenarios in which all key fields are assigned with unmanaged early numbering, it is recommended to set the respective key fields to readonly like in the following generic example:

```
define behavior for Entity [alias AliasedName]
  lock (master|dependent() )
  early numbering
  ...
{  ...
  field ( readonly ) KeyField1 [, KeyField2, …, keyFieldn];
  ...}
```

You can then implement the FOR NUMBERING method in the behavior pool with the following method signature. Because the FOR NUMBERING method is executed during the CREATE, it can't be triggered externally via EML, but only implicitly with a create operation. Furthermore, you can't define any modify operations during a CREATE. The method receives all entities for which keys need to be assigned as importing parameter::

```
  METHODS earlyNumbering_Create method FOR NUMBERING
    [IMPORTING] entities FOR CREATE business_object_entity.
```

For more details regarding the FOR NUMBERING method, refer to <method> FOR NUMBERING.

For details about the syntax of unmanaged numbering, refer to CDS BDL - early numbering (ABAP Keyword Documentation).

For a specific implementation example of a managed business object with a number range object, refer to Developing Early Unmanaged Numbering.

<method> FOR NUMBERING

This method implements early unmanaged numbering for managed business objects. You implement the FOR NUMBERING method if you want to implement early unmanaged numbering for your business object entity. The implementation is called before a business object entity is instantiated during the CREATE operation. With this method, all primary key CIDs of a BO entity are mapped to the keys drawn in the method implementation.

ℹ Note

Early numbering can't be combined with numbering:managed.

Declaration of <method> FOR NUMBERING

Unmanaged early numbering is defined on BO entity level with early numbering. The implementation is applied to all keys of an entity, so that all key fields must be mapped in the implementation. The signature of this handler method is defined by the keyword FOR NUMBERING, followed by the input parameters entities, the implicit changing parameters reported, failed, and mapped:

```
  METHODS earlyNumbering_Create method FOR NUMBERING
    [IMPORTING] entities FOR CREATE business_object_entity.
```

The mapped structure specifies how the temporary keys are mapped to the keys drawn in the implementation. For more information about the `mapped` structure, refer to Derived Data Types.

Import Parameters

- `entities`

  The table type of `entities` includes all entitites for which keys must be assigned.

Implicit Return Parameters

The `FOR NUMBERING` method provides the implicit `CHANGING` parameters `failed` and `reported` and `mapped`. The `mapped` structure contains the mapping between temporary IDs and assigned keys from the implementation. For more information, see Implicit Response Parameters.

Late Numbering

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

Late numbering is a common concept for drawing gap-free numbers. In some cases, it can be business critical that identifier numbers are gap-free, for example invoice numbers.

Related Information

Numbering
Early Numbering

Unmanaged Internal Late Numbering

In scenarios with **internal numbering**, the runtime framework assigns the primary key value when creating the new instance. Late numbering is internal by default, since a consumer can't influence the SAVE sequence after the point of no return. To define `late numbering`, the method `ADJUST NUMBERS` must be implemented. For more information, see Unmanaged Internal Late Numbering Definition.

Late numbering is used for scenarios that need gap-free numbers. Since the final key value is set just before the SAVE to the database, everything is checked before the number is assigned.

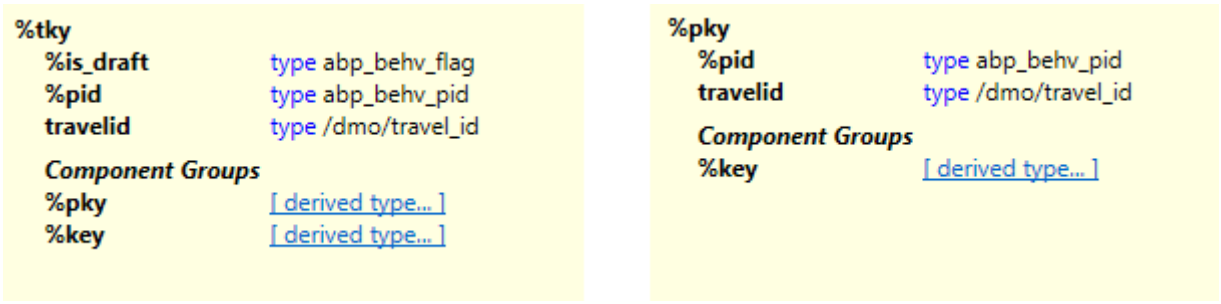
Late numbering available for managed and unmanaged implementation scenarios with and without draft. For specifics regarding RAP BOs with draft and `late numbering`, see Draft UUID for Draft Instances and Draft Instances and Associations with `late numbering` in Source or Target Entity.

For an overview as to when a key value is assigned in a late numbering scenario with the `ADJUST NUMBERS` method, see Save Sequence Runtime.

Key Handling in Late Numbering

A newly created instance must always be uniquely identifiable by its transactional key for internal processing during the interaction phase. In a RAP BO with late numbering, in which the final key value assignment only takes place during the save sequence, the transactional primary key needs an addition to fulfill the identification requirement during the interaction phase and the early save phases. This is done by extending the transactional primary key with a preliminary ID (`%PID`), in order to ensure the uniqueness also during more than one ABAP session (in draft scenarios).

The component group `%tky` then contains at least two elements: `%pid` and `%key` (where `%key` contains all key fields). For transactional processing during the interaction phase, `%tky` must be uniquely identifiable in total. This means that you can either fill `%pid` or `%key` with a preliminary value and leave the other one empty, or fill both for unique identification. The following screenshot illustrates the component groups `%tky` and `%pky` in a late numbering scenario. The example scenario is a draft scenario. Therefore, `%is_draft` is included in `%tky`.



`%tky` and `%pky` in Late Numbering Scenario with Draft

To assign the real and final key value to the instance, the method `ADJUST_NUMBERS` is called after `FINALIZE` and `CHECK_BEFORE_SAVE`, which is just before the actual SAVE happens. It returns the final key value, which is globally unique, which means that the instance doesn't need to be identified by the preliminary key anymore.

Retrieving Final Key Values with EML

In a late numbering scenario, the final key values are assigned after the point of no return, where no further consumer interaction is possible. However, in some use cases it might be required to retrieve the final key value, e.g., to trigger an operation of another BO, for which the respective key value is required as input parameter.

Example

BO A triggers a create of BO B when an instance of A is saved. To establish a foreign key relationship, the final key value of A is necessary.

As a consumer, you can use the `CONVERT KEY` statement on a root entity to retrieve the final key value assigned to a RAP BO instance in `ADJUST NUMBERS` and use it for further processing. If you use this statement on child entities outside of the behavior pool of the BO, no result is returned.

Note that you can't use `CONVERT KEY` for instances of your own RAP BO, only for foreign RAP BOs, e.g. in a cross-BO relationship. You can use `CONVERT KEY` only in the late save phase during the `COMMIT`, when the final key values are assigned.

For more information, see CONVERT KEY (ABAP- Keyword Documentation).

Late Numbering and Draft Instances

Draft UUID

Since the final key values are only assigned after the point of no return, the interaction phase operates on preliminary semantic key values. Thus, an additional draft identifier is required as PID to uniquely identify each draft instance. Hence, RAP BOs with draft capabilities and `late numbering` require an obligatory DRAFTUUID key field in the corresponding draft tables.

If late numbering is defined for a root entity, all child nodes must also contain a `parentdraftuuid` key field and their own draft identifier if they have late numbering assigned. The mapping of the draft UUIDs itself is managed by the RAP framework.

Note

If `late numbering` is added subsequently to a RAP BO, the draft tables must be regenerated to contain the required draft key fields.

As an example, the draft database table `/dmo/d_travel_d` of the `Travel` entity in a managed with draft scenario is extended as follows in case of `late numbering` on root entity level (assuming that the `Booking` entity also uses `late numbering` and `BookingSupplement` entity uses another numbering method):

Sample Code

```
define table /dmo/d_travel_d {
  key mandt          : mandt not null;
  key travel_id      : /dmo/travel_id not null;
  key draftuuid      : sdraft_uuid not null;
  ...}
```

As a direct child entity, the draft database table `/dmo/d_booking_d` of the `Booking` entity must also be extended with its own draft uuid and the draft uuid of the parent entity:

Sample Code

```
define table /dmo/d_booking_d {
  key mandt            : mandt not null;
  key travel_id        : /dmo/travel_id not null;
  key booking_id       : /dmo/booking_id not null;
  key draftuuid        : sdraft_uuid not null;
  parentdraftuuid      : sdraft_uuid;
  ...}
```

Sample Code

```
define table /dmo/a_bksuppl_d {
  key mandt                    : mandt not null;
  key travel_id                : /dmo/travel_id not null;
  key booking_id               : /dmo/booking_id not null;
  key booking_supplement_id    : /dmo/booking_supplement_id not null;
  parentdraftuuid              : sdraft_uuid;
  ...}
```

Draft Instances and Associations with `late numbering` in Source or Target Entity

Since final key values are only assigned during the late save phase, associations with late numbering in the source or target entity aren't supported, as well as foreign key associations with late numbering on the target entity and reverse foreign key associations with late numbering on the source entity. It's technically possible to define these associations, but they cause a syntax warning in the behavior definition. Executing these associations for draft instances results in a short dump.

Generally, instances with late numbering can only be defined as an association target if the instance was already saved and the final key values have been assigned.

The restrictions regarding associations aren't applicable for `late numbering` in the following scenarios:

- Compositions, ToParent associations, ToLockMaster associations, ToAuthorizationMaster associations, or ToEtagMaster associations.
- Foreign key associations with `early numbering` target and `late numbering` source.

- Reverse foreign key associations with `early numbering` source and `late numbering` target.

Unmanaged Internal Late Numbering Definition

You can define late numbering with the addition `late numbering` in the behavior definition header for each RAP entity in RAP BOs with managed and unmanaged implementation type with and without draft capabilities.

For more information about the syntax, see CDS BDL - late numbering (ABAP Keyword Documentation). As a prerequisite, you must implement the `ADJUST_NUMBERS` method for both implementation types.

For more information about the `ADJUST NUMBERS` method , see ADJUST_NUMBERS.

For an implementation example, refer to Implementing the Interaction Phase and the Save Sequence.

Uniqueness Check for Primary Keys

A uniqueness check ensures that new primary keys are unique before the saving of instances with these primary keys are rejected by a database table. The persistent database table rejects any entry with a key that already exists. To avoid that the work of an end user is in vain if all the changes are rejected by the database, the uniqueness of the primary key values of new or resumed instances must be checked as early as possible. Hence, on creating new draft data, the application must ensure that the primary keys are unique.

A uniqueness check must check the new primary key values against all already existing active instances and additionally all draft instances, that are still in their exclusive lock phase (for details about the locking phases of a draft instance, see Locking in Draft Scenarios). In case of a primary key value conflict with any of those existing instances, the creation of the new instance must be rejected. In case there is a conflict with the primary key values of the entity instance to be created and an already existing draft instance, that has reached its optimistic locking phase, the RAP runtime framework will however automatically discard the existing draft instance in order to allow creation of the new instance instead.

Internal Numbering

In scenarios with internal numbering, the uniqueness is usually given by the process of determining the number, for example by a number range object. In scenarios with internal managed numbering, the RAP runtime framework ensures the uniqueness as it provides unique UUID values.

External Numbering

In scenarios with external numbering, the uniqueness must be checked explicitly. In some cases, it is possible for the RAP runtime framework to do the check. For other cases, you must implement a precheck to check the keys before the actual instance is created and provide an implementation for the resume action in scenarios with draft support.

Responsibilities for the Uniqueness Check in Scenarios with External Numbering

Managed Scenario without Draft without Unmanaged Lock	RAP runtime framework
Managed Scenario with Draft without Unmanaged Lock	• for active instances: RAP runtime framework • for draft instances: Application developer needs to implement a precheck and the draft resume action.
Managed Scenario with/without Draft with Unmanaged Lock	Application developer needs to implement the uniqueness check in the precheck and the draft resume action.
Unmanaged Scenario with/without Draft	Application developer needs to implement the uniqueness check in the precheck and the draft resume action.

After the uniqueness is checked successfully, the RAP runtime framework exclusively reserves the primary key value for the corresponding draft instance for the time of exclusive lock phase. After the exclusive lock, once the draft is resumed, the uniqueness check must be executed again.

Related Information

Operation Precheck

Concurrency Control

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

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

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

- Optimistic Concurrency Control
- Pessimistic Concurrency Control (Locking)

Optimistic Concurrency Control

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

Pessimistic Concurrency Control (Locking)

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

Optimistic Concurrency Control

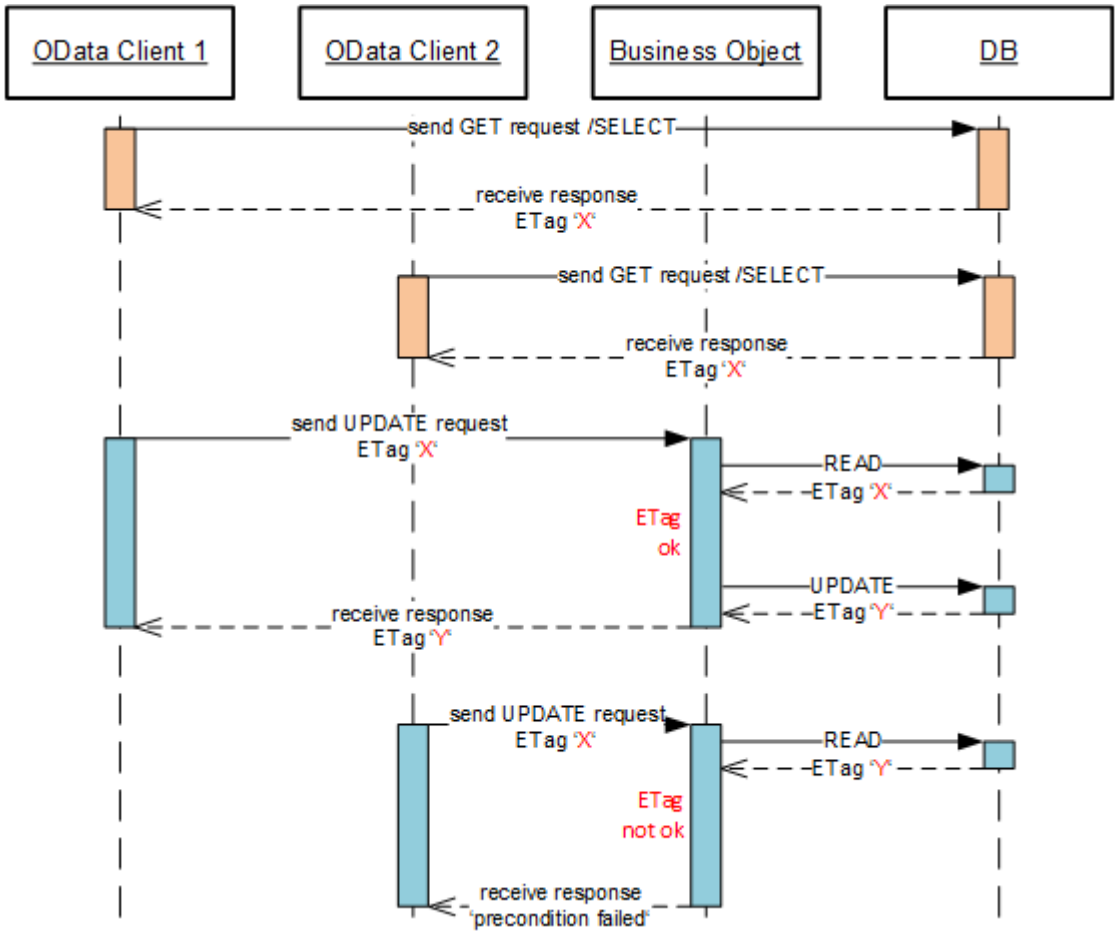
Optimistic concurrency control enables transactional access to data by multiple users while avoiding inconsistencies and unintentional changes of already modified data.

The approach of optimistically controlling data relies on the concept that every change on a data set is logged by a specified ETag field. Most often, the ETag field contains a timestamp, a hash value, or any other versioning that precisely identifies the version of the data set. Optimistic concurrency control is only relevant when consuming business objects via OData. That is why, the ETag is also referred to as OData ETag.

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

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

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



ETag Check in Update Operation

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

For more information about the ETag check during the runtime of a modify operation, see Update Operation Runtime.

ETag in Draft Scenarios

In draft scenarios, the ETag pursues the same goal as in non-draft scenarios. It ensures that the end user of an OData service only changes instances with the state that is known, for example displayed on the UI.

When active instances are requested (for example by the edit action), the value of the indicated ETag field on the active database table is compared to the ETag that is sent with the request.

When draft instances are requested (for example by an UPDATE on a draft instance), the value of the field on the draft table that logs the changes on the draft instance (`draftentitylastchangedatetime`) is used for value comparison. Like this, it is ensured that one user cannot make changes on the same draft instance on different UIs. In other words, a user can only make changes on the latest version of the draft instance.

Draft scenarios also require a total ETag that manages the transitions from active to draft and vice-versa. For more information, see Total ETag.

Related Information

ETag Definition
ETag Implementation

ETag Definition

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

ETag fields are dedicated administrative fields on the database table that log data access and modification. The RAP framework offers reuse data elements that can be used for the administrative data field definition in database tables. For more information, see RAP Reuse Data Elements.

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

For information about the syntax, see CDS BDL - ETag (ABAP Keyword Documentation).

ETag Master

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

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

ETag Dependent

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

Note

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

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

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

Projection Behavior Definition

```
projection;

 define behavior for ProjectionView [alias ProjectionViewAlias]
  use etag
 {
```

```
  ...
    use association _AssocToETagMaster
  }
```

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

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

Related Information

Optimistic Concurrency Control
ETag Implementation

ETag Implementation

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

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

If these prerequisites are fulfilled, the actual ETag check is performed by the RAP runtime engine, see Update Operation Runtime, for example.

Implementation for ETag Field Updates

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

Managed Scenario

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

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

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

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.

In scenarios where the managed RAP BO provider handles the ETag fields, it is recommended that you specify the respective fields as read-only using the field characteristic `readonly` in the behavior definition.

```
field ( readonly ) created_by, created_at, last_changed_by, LocalLastChangedAt;
```

Unmanaged Scenario

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

Implementation for Read Access to the ETag Field

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

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

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

The `READ by Association` must be defined in the behavior definition by the syntax `association AssocName`, see ETag Definition. It must be ensured that there is an implementation available for the `READ by Association` definition.

Managed Scenario

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

Unmanaged Scenario

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

For a complete example, see Implementing the READ Operation for Travel Data and Implementing the READ Operation for Associated Bookings.

Related Information

Optimistic Concurrency Control
ETag Definition

Pessimistic Concurrency Control (Locking)

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

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

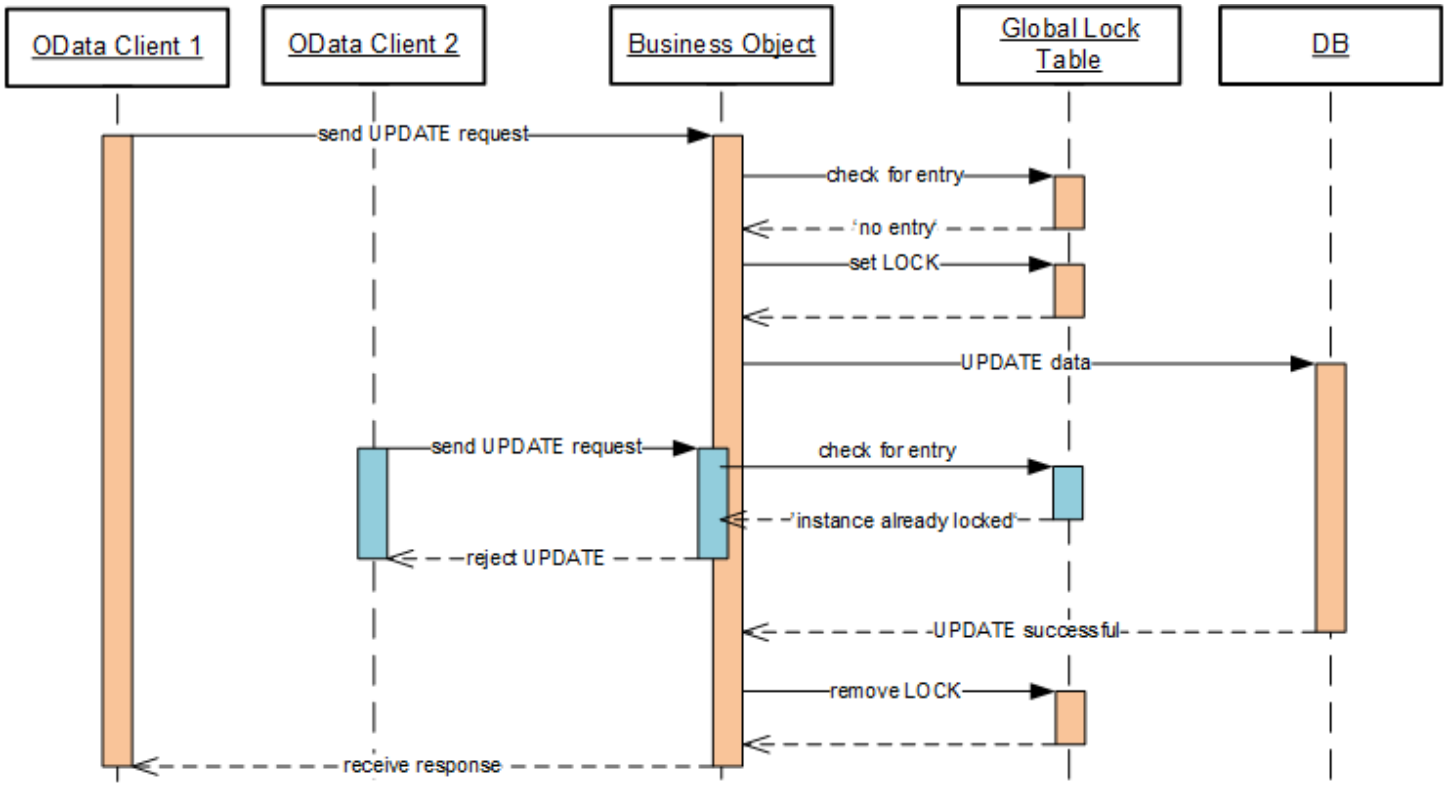
Technically, locking is ensured by using enqueue locks and global lock table entries. Before data is changed on the database, the corresponding data set receives a lock entry in the global lock table. Every time a lock is requested, the system checks the lock table to determine whether the request collides 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 multiple factors:

- **ABAP session**: The enqueue lock is released once the ABAP session is terminated.
- **LUW**: The lock is released, once the RAP LUW is terminated with a `COMMIT` or `ROLLBACK`.
- **Draft**: In draft scenarios, the lock is not coupled to the LUW or the ABAP session, but depends on the lifetime of the draft instance. The lock is released if the draft state for the instance expires, or, if the draft instance is activated and the LUW is terminated.

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



Locking Process During UPDATE Operation

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

For more information, see SAP Lock Concept.

Locking in Non-Draft Scenarios

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

- Standard Operations.
- Action.

The CREATE operation does not invoke the lock mechanism, as there is no active instance that can be locked during the runtime of CREATE, see Create Operation Runtime.

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

changed by other users or transactions.

ℹ Note

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

In the managed scenario, such a uniqueness check is executed by the managed BO provider. In the unmanaged scenario, the uniqueness check must be ensured by the application code provided by the application developer, see Uniqueness Check for Primary Keys.

Managed Scenarios

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

Unmanaged Scenarios

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

ℹ Note

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

Locking in Draft Scenarios

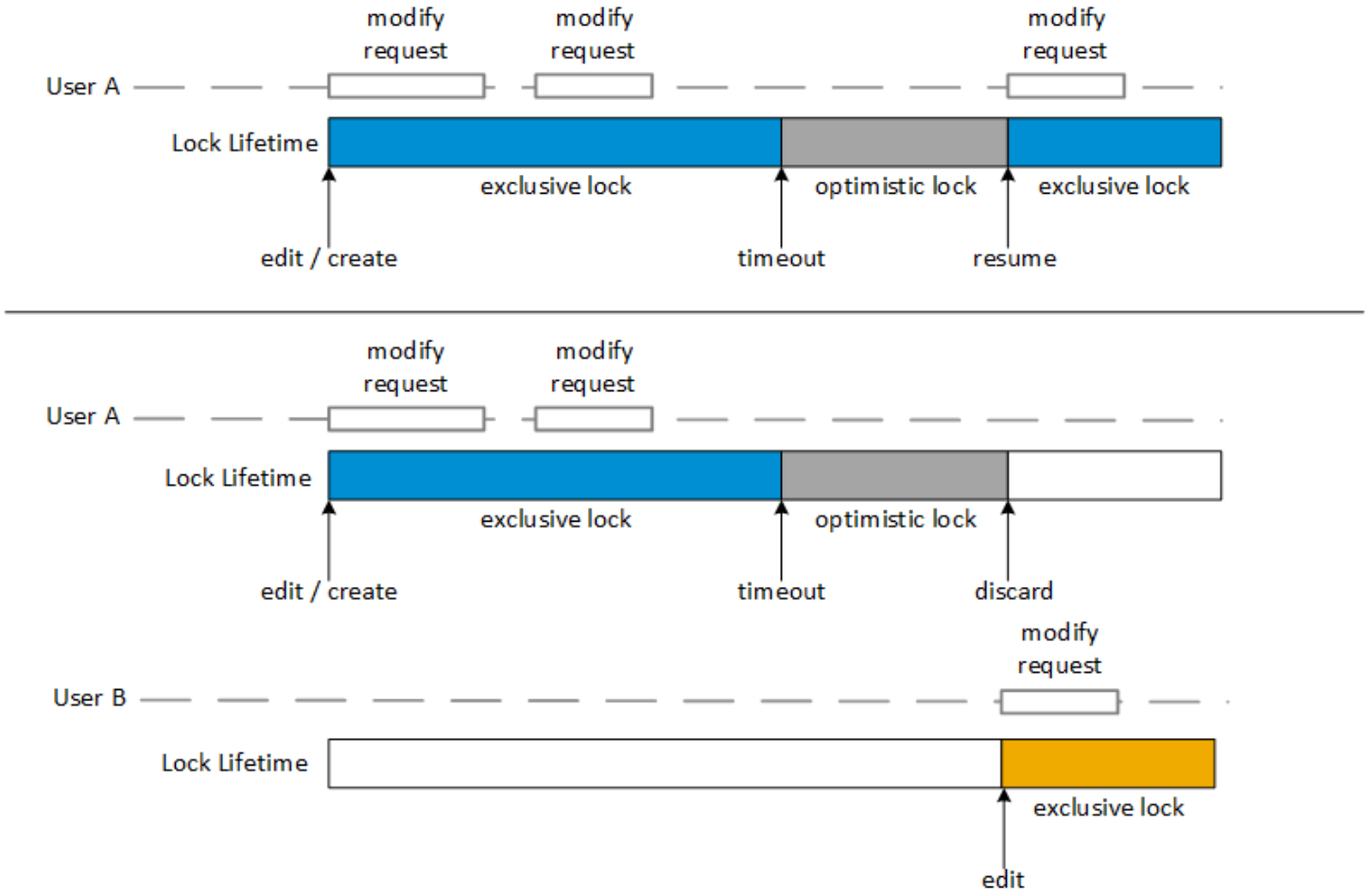
In scenarios with draft support, locking plays an even more crucial role during the draft business object life cycle.

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. This exclusive lock remains for a determined time, even if the ABAP session terminates. Once the exclusive lock expires after this duration time, the optimistic lock phase begins.

There are three cases that end the optimistic lock phase:

1. The user that created a draft instance for an active instance, and thus set an exclusive lock on the active instance, discards the draft explicitly. This can be the case, if the data changes are not relevant anymore.
2. The draft is discarded implicitly, when
    a. the draft remains untouched for a certain period of time. The lifetime of a draft is determined. If the draft is not used for a certain period of time, the draft is discarded automatically by the life-cycle service. By default this is done after 28 days.
    b. the corresponding active instance is changed directly without using the draft (by the draft owner or by a different user). This is possible during the optimistic lock phase. This change on the active instance invalidates the draft document. Invalid drafts are discarded automatically after a determined time by the draft life-cycle service.
    c. a second draft is created for the corresponding active document.

3. The draft is resumed by the draft owner. 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.



Lock Lifetime of a Draft

Lock Master and Lock Dependent

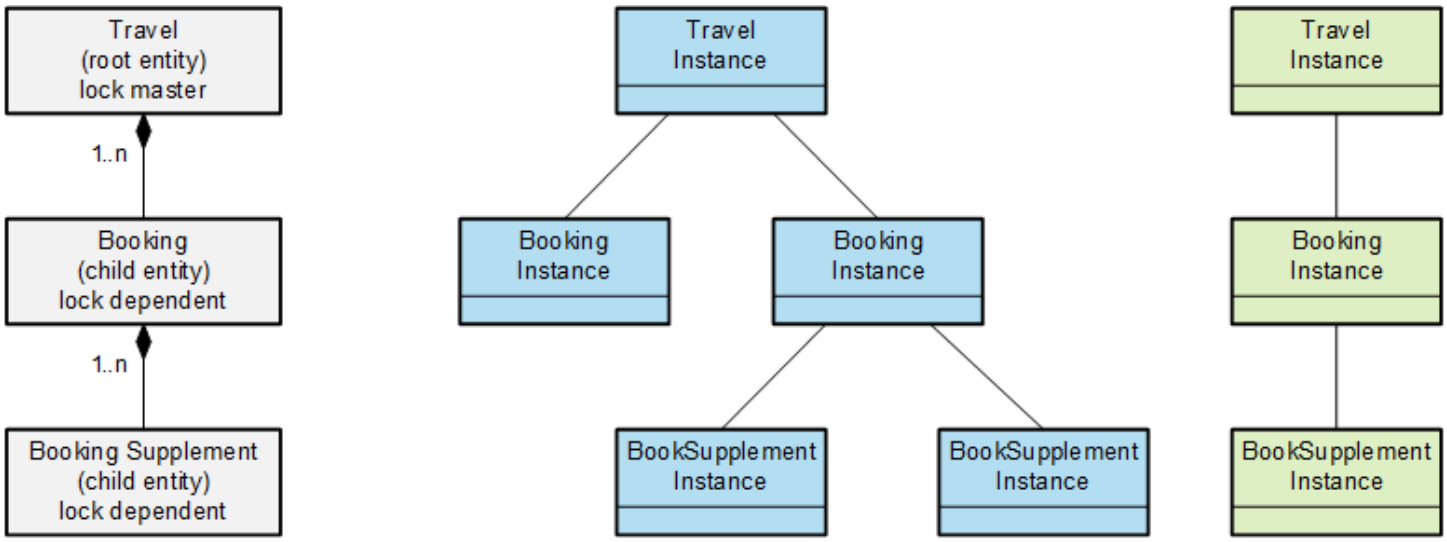
In RAP, locking is not only restricted to the entity instance that is being modified. All related entities in the business object structure are involved if one entity instance is getting locked. The locking structure is defined in the behavior definition with the keywords `lock master` and `lock dependent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

ℹ Note

Currently, only root entities can be lock masters.

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

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



Locking Structure in RAP BO

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

Related Information

Lock Definition
Lock Implementation

Lock Definition

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

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

ℹ Note

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

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

```
...
define behavior for CDSEntity [alias AliasedEntityName]
implementation in class ABAP_CLASS_NAME [unique]
...
lock master [unmanaged] | lock dependent by _AssocToLockMaster
...
{  ...
  association _AssocToLockMaster { }
}
```

Lock Master

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

> **Note**
>
> Currently, only root entities are allowed to be lock masters.
>
> Lock dependent entities must always have a lock master that is superior to them in the business object composition structure.

Lock Master Unmanaged

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

Lock Dependent

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

> **Note**
>
> In unmanaged business objects, the definition of lock dependent entities requires the implementation of the read-by-association method. As locking on lock dependent entities are delegated to the lock master, the read-by-association is used to get the authorization information from the master entity.
>
> In managed business objects, the read-by-association is provided by the RAP managed BO provider.

Related Information

Pessimistic Concurrency Control (Locking)
Lock Implementation

Lock Implementation

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

Managed Scenario

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

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

ℹ Note

In scenarios in which you have client-dependent database tables, but join client-independent fields from other database tables to your CDS view, the managed BO runtime locks the instances specific to the client. This means only the fields from the client-dependent database tables are locked. If you also want to lock the client-independent fields, you have to implement an `unmanaged lock`.

Unmanaged Scenario

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

For a complete example, see Implementing the LOCK Operation.

<method> FOR LOCK

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

The `FOR LOCK` method is automatically called by the orchestration framework before a changing (`MODIFY`) operation such as `update` is called.

Declaration of <method> FOR LOCK

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

ℹ Note

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

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

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

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

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

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

Import Parameters

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

- ID fields

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

ℹ Note

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

Changing Parameters

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

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

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

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

Implementation of `method FOR LOCK`

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

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

An example on how to implement the `method FOR LOCK` is given in Implementing the LOCK Operation.

Related Information

Implicit Response Parameters

Draft

You can draft-enable a business object to automatically persist transactional data in the backend. This approach supports stateless communication for your applications.

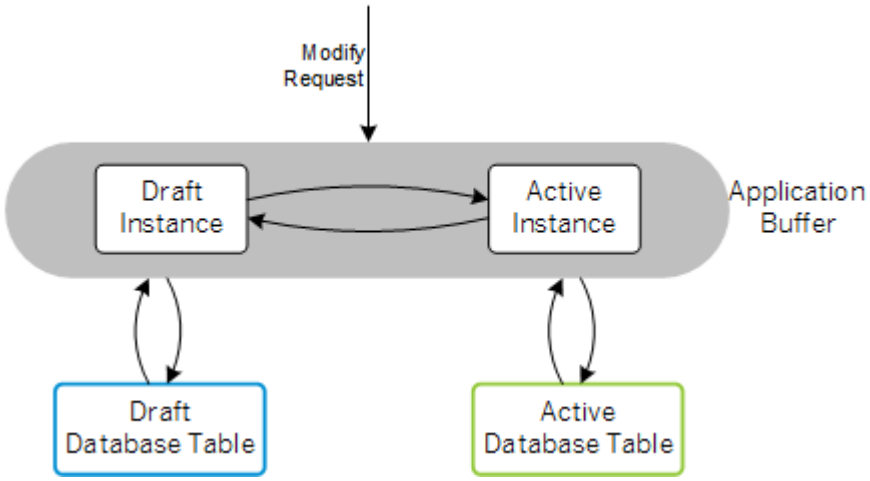
Modern cloud-ready apps require a stateless communication pattern, for example to leverage cloud capabilities like elasticity and scalability. Thus, there is no fixed backend session resource along a business transaction for each user and the incoming requests can be dispatched to different backend resources, which supports load balancing. On the other hand apps are stateful from end-user perspective. Business data that is entered by the end user needs to be locked, validated and enriched via ABAP business logic on backend side.

The draft concept fills the gap between a stateless communication pattern and a stateful application by applying REST principles:

- The draft represents the state and stores the transactional changes on the database in shadow tables. It is an addressable resource, the exact copy of the active data that is currently being edited.
- Between two backend roundtrips, there is no running ABAP session waiting for the next roundtrip. The execution might even be performed on different backend servers.

Draft-enabled applications allow the end user to store changed data in the backend and continue at a later point in time or from a different device, even if the application terminates unexpectedly. This kind of scenario needs to support a stateless

communication and requires a replacement for the temporary in-memory version of the business entity that is created or edited. This temporary version is kept on a separate database table and is known as draft data. Drafts are isolated in their own persistence and do not influence existing business logic until activated.



Draft and Active Data

In general, the draft implies the following advantages for your application:

- **Save & Continue:**

  A draft allows you to stop processing and saving of business data at any time and to continue processing later on, no matter if the data is in a consistent state or not.

- **Staging:**

  The draft works like a staging area so that business data is isolated during processing and can finally be activated, that is, saved to the active database.

- **Device Switch:**

  As the draft persists the state of processing, it can be resumed with any device. For instance, you can edit data on a laptop and continue from a mobile device.

- **Collaboration:**

  With this release, only the exclusive draft is supported. This means that only the user who has created the draft is able to process and activate it. A shared draft, however, allows dispatching the ownership of the draft to a different user. Finally, the collaborative draft allows editing the same draft instance by multiple users at the same time.

State Handling

Applications usually have a transactional state that buffers all changes of a current transaction until it is committed by the application user. If the buffer state is consistent, the commit is accepted, if not, the whole transaction is rejected. This all-or-nothing-approach prevents inconsistent data on the database table.

For smaller applications, sending the whole transactional state to the backend is unproblematic. The state can be managed completely on the frontend side. We speak of **frontend state handling** in that case. However, in case of complex backend-located business logic (especially legacy code), configuration, and data volume, transferring everything to the frontend layer does not work for performance, maintenance, and security reasons. Thus, backend **state handling (draft)** is required.

Basic Principles

- The main business logic is implemented on the active entity, for example with actions or feature control. This behavior can be applied to draft entities in the same manner.
- There can be only up to one draft instance for each active instance at the same time.

- The primary key value of a draft and the corresponding active instance is the same.

Exclusive Draft

There are several approaches how draft instances that are created by different users can be accessed. The current version of the ABAP RESTful Application Programming Model only supports the **exclusive draft.** A draft exclusively belongs to the user that created the draft. Only this user is able to see and process the draft instance.

The exclusive draft goes hand in hand with the exclusive lock. As soon as a user starts working on a draft, it sets an exclusive lock. It is locked for other users. The exclusiveness is maintained for a defined period of time. After that period, the draft goes into an optimistic lock phase. In this phase, the draft can either be resumed by the same user or discarded, if other users take over the exclusive draft handling. For more information, see Locking in Draft Scenarios

Draft in the ABAP RESTful Application Programming Model

Draft is an option that you can use for application development with both, the `managed` and `unmanaged` implementation type and also with mixed scenarios, for example, `managed` with `unmanaged save`. In all scenarios, the draft is managed. That means, it is handled by the RAP runtime framework. You, as an application developer, do not need to care about how draft data is written to the draft database table. This is done for you. Of course, adding draft capabilities to your business object might imply changes in your business logic also for the processing of active data that you are responsible of. In addition, RAP also offers implementation exits for cases in which you need business service-specific draft capabilities that impact the draft handling.

Restrictions

Draft-Enabled Business Objects

- There can be locking conflicts for the active provider if an instance that was activated is locked again in the same transaction. This is because the durable lock of the active instance remains after the activation of a draft instance.
- Every child entity in a CDS composition hierarchy must be represented in the behavior definition if a behavior definition for the root entity exists and defines a draft-enabled RAP business object.

Unmanaged Business Objects with Draft

There are features that can be defined in the behavior definition of an unmanaged business object and thus used on active instances, but not on draft instances:

- You can't execute the following operations on draft instances:
  - Create-enabled associations that aren't modeled as part of a composition
  - Direct creates on child instances
- The aggregated admin data fields aren't updated automatically when a draft instance is saved.
- Associations that use `NOT` in the binding condition can't be draft-enabled.
- There's no support for transactional draft-enabled associations leading from draft to active instances.
- Client-independent CDS views aren't supported.

Draft Design Time

Given the fact that the runtime of a draft business object differs significantly from a non-draft business object, it comes with no surprise that there are also quite some differences with regard to the design time of a draft business object.

The addition `with draft` in the behavior definition defines a draft business object. As soon as the business object is draft-enabled with this syntax element, you have various other options to use draft capabilities on certain actions and operations. For more information, see Draft Business Object.

As draft instances are stored independently of active instances, a separate database table, the draft table, must be created. This draft table must be explicitly stated in the behavior definition and can be generated from there. For more information, see Draft Database Table.

To control the state of the active BO data, you use the total ETag, which must also be defined in the behavior definition. For more information, see Total ETag.

The operations create, update and delete need to be defined at least internally in the behavior definition. This ensures that operations performed on draft instances can be applied on active instances during the activate action by the framework. For more information, see Runtime Activate Action.

To enable that an association retrieves active data if it is followed from an active instance and draft data if it is followed from a draft source instance, the associations must be draft-enabled. For more information, see Draft-Enabled Associations.

Draft Business Object

The draft capability for a draft business object is defined in 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`. The draft-indicating syntax element `with draft` is added at the top of the behavior definition as it does not belong to a certain entity of the BO, but concerns the whole BO. You cannot implement draft capabilities for single BO entities.

For a detailed syntax description, see CDS BDL - with draft (ABAP Keyword Documentation).

In draft business objects, the handling of the draft instances is always managed by the RAP runtime framework, no matter if your business object is `unmanaged` or `managed`. That means, the draft life-cycle is determined by the RAP draft runtime and specific draft actions are implicitly available for the draft business object.

Runtime Aspects

If you use `%tky` to address the application key components of an entity, you do not have to change your business logic implementation when draft-enabling the business object. The derived type component `%tky` automatically includes the draft indicator `%IS_DRAFT` for draft business object to distinguish draft instances from active instances. Like this, the business functionality runs smoothly without adapting your code after draft-enabling your business object.

If you use `%key` in your business logic implementation, you have to revise the complete implementation when draft-enabling the business object. This derived type component only comprises the application key component. In this case, the runtime is not able to distinguish between draft and active instances.

ℹ Note

The recommendation is to only use `%tky` in your business logic implementation for both, active-only and draft business object.

For more information, about the components of derived types, see Components of BDEF Derived Types (ABAP Keyword Documentation).

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.

The draft data is stored on a separate draft table to match the different access patterns of draft and active data. While the active database table stores many instances, but rarely has access on these instances, it is vice versa for the draft table. With two separate tables, the performance can be adapted to the different approaches of the two tables more easily. During runtime, every request is marked whether it is intended for the draft table, or for the active with the draft indicator `%IS_DRAFT`.

With using a separate database table for the draft information, it is guaranteed that the active persistence database table remains untouched and thus consistent for existing database functionality.

ℹ 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 get updated automatically.

The fields of the draft database table are derived from the corresponding CDS view entity. In addition, it contains some technical information the RAP transactional engine needs to handle draft. The technical information is added with the draft admin include.

For more information, see CDS BDL - draft table (ABAP Keyword Documentation).

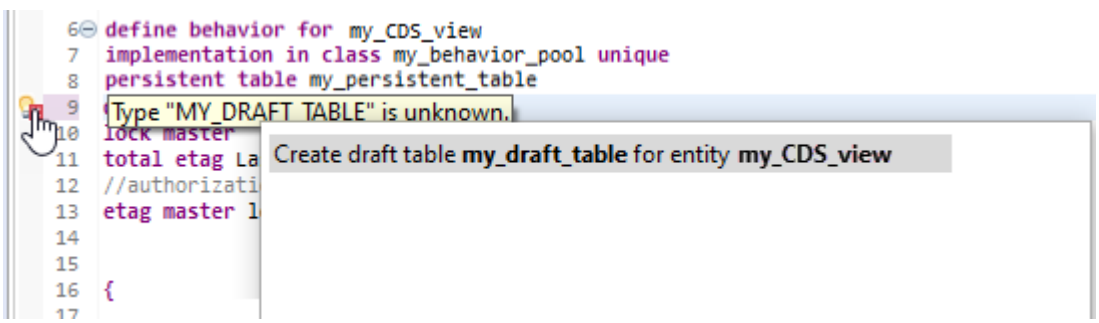
Draft Admin Include

```
...
define structure sych_bdl_draft_admin_inc {
  draftentitycreationdatetime   : sych_bdl_draft_created_at;
  draftentitylastchangedatetime : sych_bdl_draft_last_changed_at;
  draftadministrativedatauuid   : sych_bdl_draft_admin_uuid;
  draftentityoperationcode      : sych_bdl_draft_operation_code;
  hasactiveentity               : sych_bdl_draft_hasactive;
  draftfieldchanges             : sych_bdl_draft_field_changes;}
```

The draft table can be generated automatically via a quick fix in the behavior definition, which is offered as soon as the business object is draft-enabled with the syntax `with draft`.

For more information, about the syntax in the behavior definition, see CDS BDL - entity behavior characteristics (ABAP Keyword Documentation).

In case the draft database table does already exist, the quick fix completely overwrites the table.



Database Table Generation

Whenever you change the active data model in the active database table, you have to regenerate the draft table to align them. The behavior definition offers a quick fix for regeneration, but it is the responsibility of the application developer to keep the two database tables synchronized. When regenerating the draft database table it completely overwrites the existing one.

Total ETag

The total ETag is a designated field in a draft business object to enable optimistic concurrency checks during the transition from draft to active data.

Draft business objects require a total ETag. This designated field on the database table is updated by the RAP runtime framework as soon as an active instance is changed. The total ETag always refers to the complete BO. As soon as an instance of any BO entity is changed, the total ETag is updated. Its value is compared when resuming a draft instance to ensure that the active data has not been changed after the exclusive lock has expired. Thus, the total ETag is important for optimistic concurrency control during the transition from draft to active data. The total ETag is defined in the behavior definition.

Total ETag Definition

```
[implementation] unmanaged|managed|abstract [in class class_name unique];
with draft;

define behavior for CDSEntityName [alias AliasName]
implementation in class BehaviorImplementationClass unique
persistent table ActiveDBTable
draft table DraftDBTable
lock master
total etag TotalETagField
etag master LocalETagField
...
```

The total ETag field is defined on the lock master entity (currently identical to root entity) and optimistically controls the whole business object regarding concurrency on the active data.

ℹ Note

The definition of the total ETag is only possible directly after the lock master definition in the behavior definition.

The RAP managed framework is able to update the total ETag field automatically on saving a draft instance if the following conditions are fulfilled:

- The field is annotated in CDS with the annotation `@Semantics.systemDateTime.lastChangedAt: true` with the precondition that the field has a date-compatible type).
- The total ETag field is included in the field-mapping prescription in the behavior definition, if the names on the database table and in CDS are different. .

If you don't enable the framework to update the field, you have to include your own updating logic in the business logic of the BO.

ℹ Note

Total ETag and ETag master/dependent are two sides of the same medal. For a smoothly running application, you need both ETags.

The total ETag is used for edit-drafts, a draft that has a corresponding active instance. As soon as the exclusive lock expires and an edit-draft is resumed, the total ETag value of the draft instance is compared to the total ETag value on the active

instance. Only if the values coincide can the draft be resumed. The total ETag is compared for all entities of a BO. Draft business objects require a total ETag to ensure optimistic concurrency comparison.

The ETag master/dependent concept ensures that the end user of an OData service only changes instances with the state that is displayed on the UI. Hence, the ETag master/dependent prevents changes of the BO that are not noticed by the OData consumers. With ETag master, each BO entity can be checked independently. ETag master fields are required to provide optimistic concurrency locking for OData consumers. For more information, see Optimistic Concurrency Control.

Since the total ETag and the ETag master serve different purposes, it's recommended to use separate fields to make use of the different functionalities. If you use the same field for the total ETag and the ETag master for a BO with several child entities, you need to consider that the total ETag changes (and thus the ETag master) whenever a child entity is updated (and saved).

Draft Actions

Draft actions are actions that are implicitly available for draft business objects as soon as the business object is draft-enabled. In strict mode, they have to be explicitly declared in the behavior definition.

Draft actions can only be specified for lock master entities, as they always refer to the whole lockable subtree of a business object.

For syntax information, see CDS BDL - draft actions (ABAP Keyword Documentation).

Draft actions are needed for specific occasions in business services with draft. Draft actions can be completely handled by the RAP framework, in managed and in unmanaged implementation scenarios. Nevertheless, an application developer can add additional implementation for each draft action if the addition `with additional implementation` is used. In this case, the additional implementation is executed on the same instance version (draft or active), on which the default draft action is executed. For example, the additional implementation for Edit receives the transactional key with the draft indicator off, as the Edit action is executed on the active instance.

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.

For more information, see Edit 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` does not allow feature or authorization control. Authorization is not available as the Activate only transfers the state of the draft buffer to the active buffer. Authorization is controlled when the active instance is saved to the database.

For more information, see Edit Action.

The draft action `activate` is also available in an optimized variant which can improve the performance compared to the variant without optimization.

For more information, see Draft Action Activate Optimized.

Draft Action Discard

The draft action DISCARD deletes the draft instance from the draft database table. No feature or authorization control can be implemented.

For more information, see Discarding Root Draft Instances.

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.

> **Note**
>
> For performance reasons, determinations and validations are not executed during the draft determine action `Prepare`, if these determinations and validations have already been executed during a previous determine action that has been executed on the instance in the same transaction.
>
> This performance optimization is not applied in the following cases:
>
> - A modification performed after the first determine action triggers a determination/validation.
> - A validation in the first determine action reports a failed key.
> - A determination/validation has been assigned using the addition `always`.
>
> In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

For more information, see Preparing Draft Instances for Activation.

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 recreates the lock for the corresponding instance on the active database table. On a 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`.

If you want to implement your own logic for the resume action, it must be implemented by the application developer in the related behavior implementation class, just like any other action, see Action Implementation.

For more information about the `Resume`, see Resuming Locks for Draft Instances.

Draft Action Activate Optimized

The draft action `Activate` offers an optimized variant, which can improve the performance compared to the variant without optimization by reducing the number of determination and validation executions in a RAP transaction.

Prerequisite

Before using the optimized version of the draft action `Activate`, ensure that the determinations and validations of your business object follow the respective guidelines described in Determination and Validation Modelling. Using the optimized version without considerung these guidelines can impair the runtime behavior. The section Consequences of modelling errors exemplarily shows why the modelling guidelines are essential for the optimized `activate` action.

Enabling the optimized `Activate` action

In order to enable the optimized variant of the draft action `Activate`, add the keyword `optimized` to this action in the behavior definition:

```
define behavior for ...
{
  ...
  draft action Activate optimized;
  ...
}
```

For detailed information on the effects of the optimized `Activate` action on the runtime, see Runtime Activate Optimized Action.

Draft-Enabled Associations

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.

The intended behavior for all associations within the composition tree of a draft business object is to be draft-enabled, so that the associations always lead to the target instance with the same state (draft or active).

Example

On creating a child instance of a draft root instance by a create-by-association, you want to create a draft instance. Creating an active child instance would lead to BO-internal inconsistencies. That is why, all compositions are automatically draft-enabled.

As soon as you draft-enable a business object by adding `with draft` to the behavior definition, all BO-internal associations are automatically draft-enabled. To make this behavior explicit, the behavior prompts you to specify the compositions within a draft BO with `with draft`.

For syntax information, see > associations (ABAP Keyword Documentation).

Draft Query Views

Draft query views allow you to define READ access limitations for draft data based on the data control language (DCL).

About Draft Query Views

A DCL offers the possibility to restrict read access to the data that is returned from a CDS View. A RAP business object with draft capabilities is based on an active database table that reflects the state of the latest saved to the database and a draft table that stores the draft data until it's written to the persistent database.

In general, DCLs defined for the CDS views choosing from the active database table aren't applied to draft data. Draft query views allow you to define read access restrictions for draft data.

Draft Query View Definition

The following example is based on the sample scenario `/DMO/FLIGHT_REUSE_AGENCY`.

A draft query view is a CDS view that selects directly from a draft database table. This CDS view must always contain all fields contained in the draft table including administrative fields.

Defining the Draft Table

In the following example, the draft table defined in the base behavior definition is `/dmo/agency_d`. It contains the required fields for a travel agency like an agency name or address, a well as adminitrative data like `localcreatedby`:ℹ ***Expand the following code sample to view the source code of the draft table `/DMO/AGENCY_D`.***

```
@EndUserText.label : 'Draft table for entity /DMO/R_AGENCY'
...

define table /dmo/agency_d {

 key mandt     : mandt not null;
 key agencyid    : /dmo/agency_id not null;
 key draftuuid   : sdraft_uuid not null;
 name        : /dmo/agency_name;
 street       : /dmo/street;
 postalcode     : /dmo/postal_code;
 city        : /dmo/city;
 countrycode    : land1;
 phonenumber    : /dmo/phone_number;
 emailaddress    : /dmo/email_address;
 webaddress     : /dmo/web_address;
 attachment     : /dmo/attachment;
 mimetype      : /dmo/mime_type;
 filename      : /dmo/filename;
 localcreatedby   : abp_creation_user;
 localcreatedat   : abp_creation_tstmpl;
 locallastchangedby : abp_locinst_lastchange_user;
 locallastchangedat : abp_locinst_lastchange_tstmpl;
 lastchangedat   : abp_lastchange_tstmpl;
 "%admin"      : include sych_bdl_draft_admin_inc;
}
```

Defining a Draft Query View

The CDS View `/DMO/R_AgencyDraft` selects diretly from `/dmo/agency_d` and must contain all elements of the draft table. Also the fields from the `%admin` include are part of the CDS View. For more details about the elements of draft tables, see Draft Database Table. The `/DMO/R_AgencyDraft` can define read access controls like for other read use cases, which is then applied when data is read from the draft table.

ℹ ***Expand the following code sample to view the source code of the draft query view `/DMO/R_AgencyDraft`.***

```
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Draft Query View: Agency'


define view entity /DMO/R_AgencyDraft
  as select from /dmo/agency_d as Agency
```

```
{
  key agencyid                     as Agencyid,
  key draftuuid                    as Draftuuid,
      name                         as Name,
      street                       as Street,
      postalcode                   as Postalcode,
      city                         as City,
      countrycode                  as Countrycode,
      phonenumber                  as Phonenumber,
      emailaddress                 as Emailaddress,
      webaddress                   as Webaddress,
      attachment                   as Attachment,
      mimetype                     as Mimetype,
      filename                     as Filename,
      localcreatedby               as Localcreatedby,
      localcreatedat               as Localcreatedat,
      locallastchangedby           as Locallastchangedby,
      locallastchangedat           as Locallastchangedat,
      lastchangedat                as Lastchangedat,
      draftentitycreationdatetime  as Draftentitycreationdatetime,
      draftentitylastchangedatetime as Draftentitylastchangedatetime,
      draftadministrativedatauuid  as Draftadministrativedatauuid,
      draftentityoperationcode     as Draftentityoperationcode,
      hasactiveentity              as Hasactiveentity,
      draftfieldchanges            as Draftfieldchanges
}
```

Defining Draft Query Views in the Behavior Definition

The draft query view is included in the behavior definition with the keyword `query` after the draft table name. Then the respective draft query view is identified as draft query view by the RAP query engine. The behavior definition `/DMO/R_AgencyTP` specifies `/DMO/R_AgencyDraft` as draft query view for the Agency RAP BO.

ℹ ***Expand the following code sample to view the source code of the behavior definition `/DMO/R_AgencyTP`.***

```
define behavior for /DMO/R_AgencyTP alias /DMO/Agency
persistent table /dmo/agency
draft table /dmo/agency_d query /DMO/R_AgencyDraft
{...}
```

Draft Query Views in RAP Extensibility Scenarios

If the original BO is draft-enbaled and release contracs are used, using draft query views is mandatory during the extensibility-enablement.

For more information, see Extensibility-Enablement for CDS Data Model Extensions.

Draft Runtime

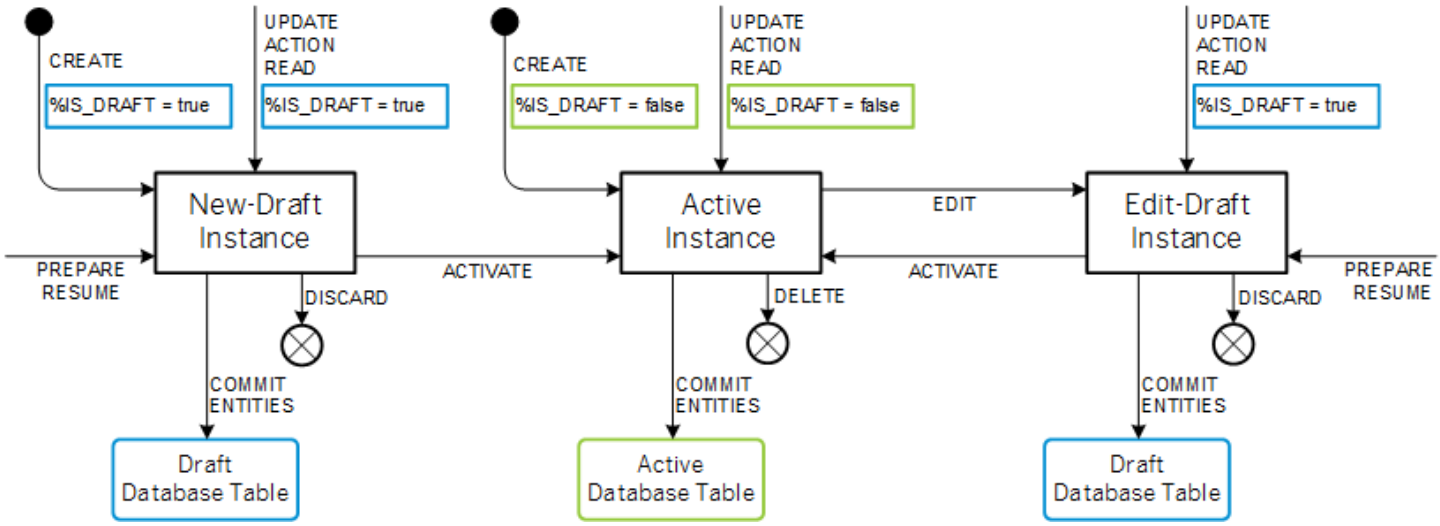
RESTful applications require a constant stateless communication with the backend to prevent that transient data is lost when a user session ends. Data changes are constantly persisted on the draft database tables and are therefore always retrievable from any device. The fact that two database tables are involved in the runtime of a draft business object requires an elaborate lifecycle of data. All data undergo several states while being processed by a draft business object. There are three distinct states that need to be distinguished.

For initial data that is not yet persisted on the active database table, we speak of **new-draft** instances. New-draft instances do not have a corresponding active instance. They are created using the modify CREATE operation having the draft indicator `%IS_DRAFT` set to `true`. The draft indicator is automatically available as derived type when working with the transactional key `%tky`.

As soon as a draft instance is transferred from the draft database table to the persistent table, the draft instance is activated. The data that is stored on the persistent database table is called **active data**. Active instances come into being by activating new-draft instances or by using the modify CREATE operation and setting the draft indicator `%IS_DRAFT` to `false`.

Edit-drafts always exist in parallel to the corresponding active data. Whenever active data is edited, the whole active instance is copied to the draft table. All modifications are saved on the draft database table before the changes are finally applied to the active data on the active database table again. Edit-draft instances are created by using the `EDIT` action on active instances.

The following diagram illustrates the lifecycle of draft and active data and their transitions from one state to the other.



Lifecycle of Draft and Active Data

When working with draft instances, you always set an exclusive lock for this instance. In case of an edit-draft, that means that the corresponding active instance cannot be modified by any other user. In case of a new-draft, the primary key number is locked exclusively to avoid duplicate key failure on the database. For the time of the exclusive lock phase, it is not possible to create neither an active, nor a draft instance with the same key. After the exclusive lock has expired, the draft is in an optimistic lock mode, in case the user starts processing again, the draft resume action is automatically invoked. It locks the corresponding active instance again, compares the total ETag and if successful, the draft can be continued in exclusive mode.

You can always send requests for draft instances or active instances. Accessing draft instances via EML or OData works exactly like accessing active instances. To distinguish if the requests are aimed at draft or active instances, the RAP runtime framework evaluates the draft indicator `%IS_DRAFT`. This draft indicator must be set for every request, which can be processed on active or draft instances. In OData, this technical primary key component is mapped to `isActiveInstance`.

On a UI, modifications of data in a draft BO always take place on the draft instance (except for actions). Only in case of activation via the activate action are the changes applied to the corresponding active instance. The transition from draft to active data and vice versa requires specific actions that process tasks that are not relevant in a non-draft business object. These actions are explained best when considering the lifecycle of a draft and active data in detail.

Creating Draft Data

Creating Active Data

Changing and Reading Instances of a Draft BO

Deleting Instances of a Draft BO

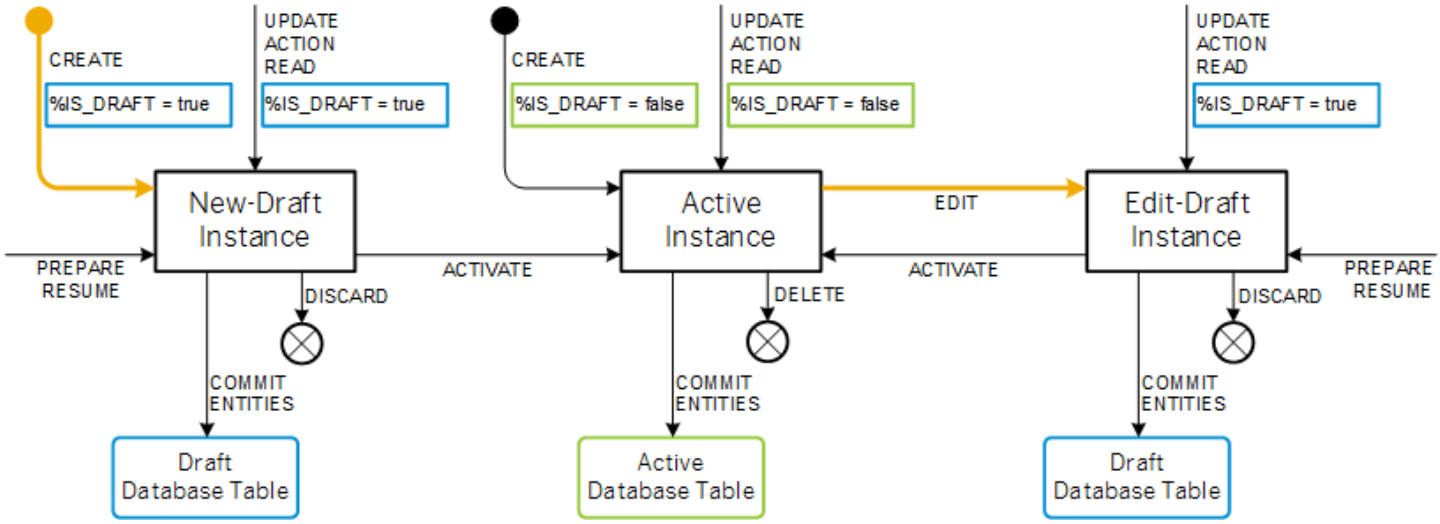
Preparing Draft Instances for Activation

Resuming Locks for Draft Instances

Saving Data to the Database

Creating Draft Data

You create draft instances by creating new-drafts or by editing active instances.



Lifecycle of Draft and Active Data

As the preceding diagram suggests, there are two options to create draft data:

Modify CREATE Request

By using the modify operation for CREATE, you create a new-draft instance, a draft instance that has no corresponding active instance. For such a create request, the draft indicator must be set to `true`. Like a CREATE for active instances, the RAP runtime sets an exclusive lock for the instance and triggers any determination on modify with trigger CREATE. On creating a new instance the uniqueness of the primary keys must be ensured. Otherwise, if duplicate keys reach the active database table, the whole request is denied. For more information about an early uniqueness check, see Uniqueness Check for Primary Keys.

On a Fiori Elements UI, using the **Create** button executes a CREATE request with the draft indicator set to true. It is not possible to directly create active instances by using the **Create** button on a Fiori UI.

Via EML, however, it is possible to distinguish requests to create draft data and requests to create active data. To create draft instances, the draft indicator must be set to `true`:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    CREATE FROM
    VALUE #( ( %is_draft          = if_abap_behv=>mk-on  "positive draft indicator
               %control-FieldName = if_abap_behv=>mk-on
               %data-FieldName    = 'Value') )
  REPORTED DATA(create_reported)
  FAILED DATA(create_failed)
  MAPPED DATA(create_mapped).
```

Edit Action

By executing the draft action `EDIT` on an active instance, you create an edit-draft instance, a draft instance that has a corresponding active instance on the persistent database. An edit action creates a new draft document by automatically copying the corresponding active instance data to the draft table. At the same time, the `EDIT` triggers an exclusive lock for the active instance. This lock is maintained until the durable lock phase of the draft ends, which is either when the draft is activated, or when the durable lock expires after a certain time.

The edit action has a parameter with which you can determine whether editing an active instance for which a draft already exists is possible or not. Having set the parameter `preserve_changes` to `true`, your action request is rejected if a draft already exists. If `preserve_changes` is set to `false` (default setting), the edit action is executed. That means, the active instance is copied to the draft database table and the existing draft is overwritten with the values of the active data. In that case, you lose the changes that anyone has done on the existing draft instance. Hence, it is recommended to always include the parameter in your action requests.

For edit action requests, the draft indicator does not have to be set, as the `EDIT` is always executed on the active instance.

Instance authority and feature control are both checked for the action. To implement application-specific controls, define the `EDIT` action in the behavior definition and implement the method for features or for authorization in the behavior pool.

On a Fiori Elements UI, the edit action is triggered when choosing the **Edit** button, which is only available on the object page of an active instance. Triggering the edit action from the UI always includes the parameter `preserve_changes` set to `true`.

In this case, OData sends a `POST` request for action execution of the `EDIT`, which is then carried out by the RAP runtime framework.

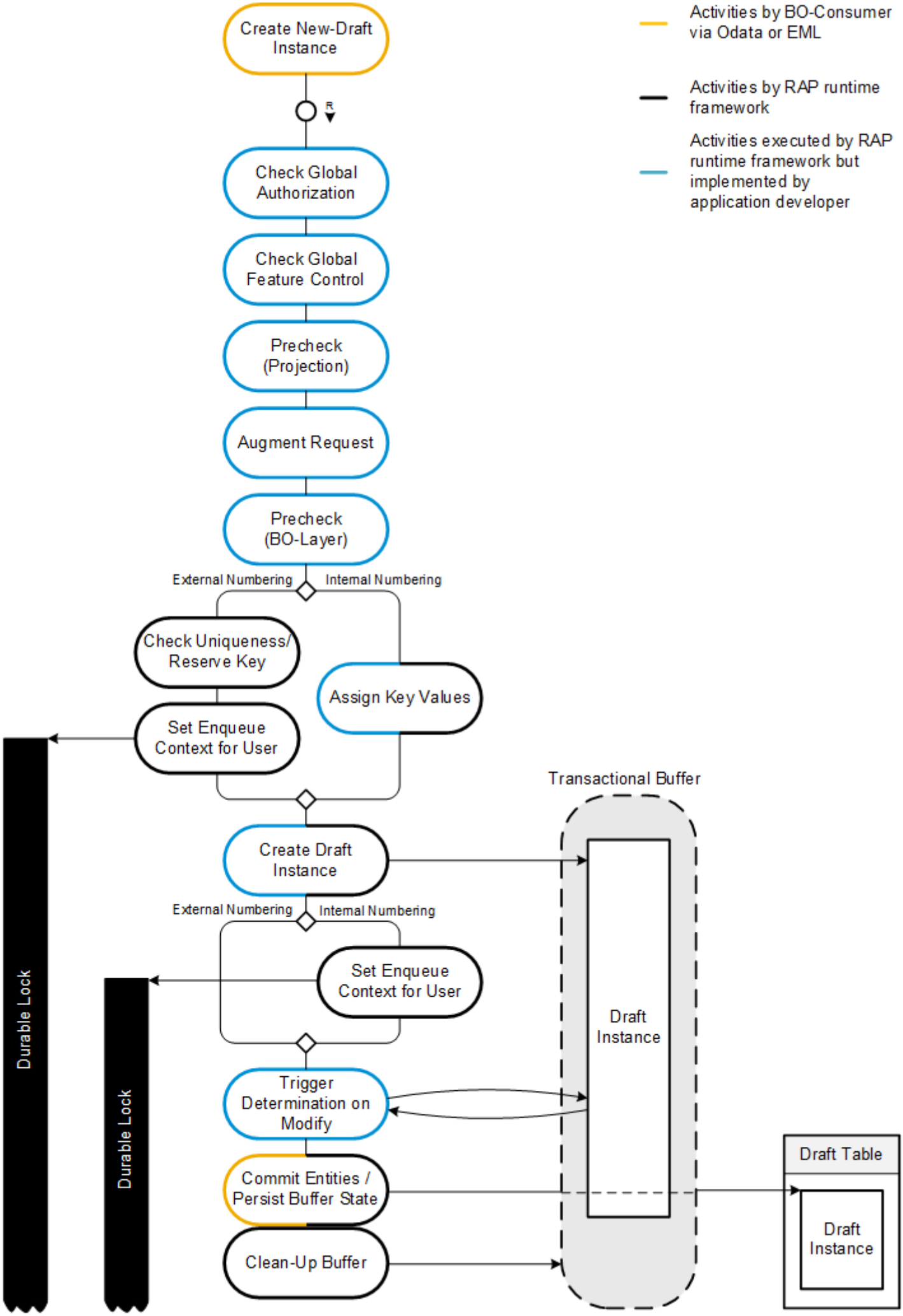
The edit action can be executed via EML, just like any other business logic action:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    EXECUTE Edit FROM
    VALUE #( ( %key-element1           = 'KeyValue'
               %param-preserve_changes = 'X' ) )
  REPORTED DATA(edit_reported)
  FAILED DATA(edit_failed)
  MAPPED DATA(edit_mapped).
```

Runtime CREATE Draft

This topic illustrates the execution order of possible runtime activities after executing a modify request for `CREATE`.

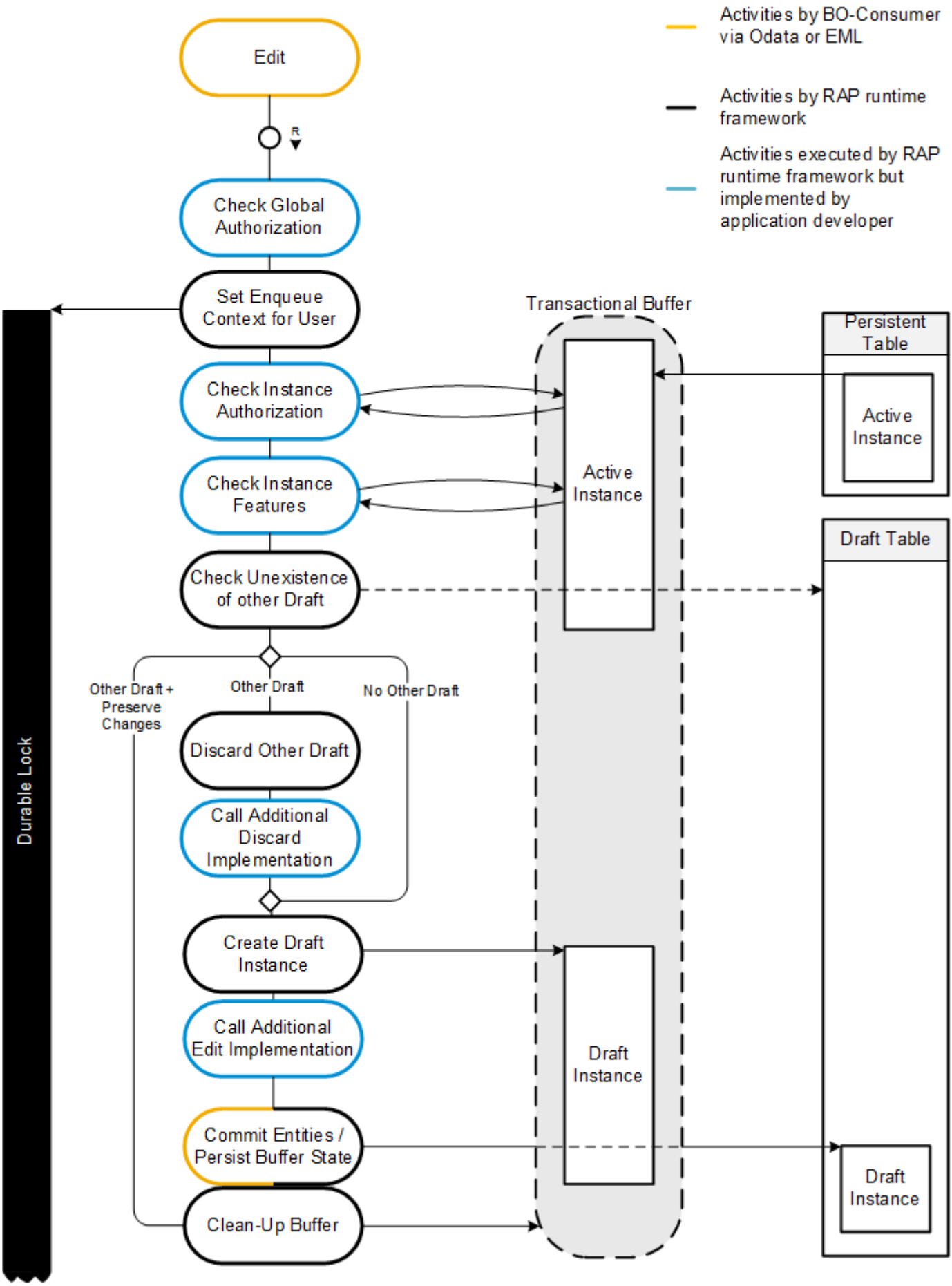
Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.


Activities by BO-Consumer via Odata or EML
Activities by RAP runtime framework
Activities executed by RAP runtime framework but implemented by application developer
Create New-Draft Instance
Check Global Authorization
Check Global Feature Control
Precheck (Projection)
Augment Request
Precheck (BO-Layer)
External Numbering
Internal Numbering
Check Uniqueness/ Reserve Key
Assign Key Values
Set Enqueue Context for User
Durable Lock
Transactional Buffer
Create Draft Instance
External Numbering
Internal Numbering
Set Enqueue Context for User
Durable Lock
Draft Instance
Trigger Determination on Modify
Commit Entities / Persist Buffer State
Clean-Up Buffer
Draft Table
Draft Instance

Runtime Edit Action

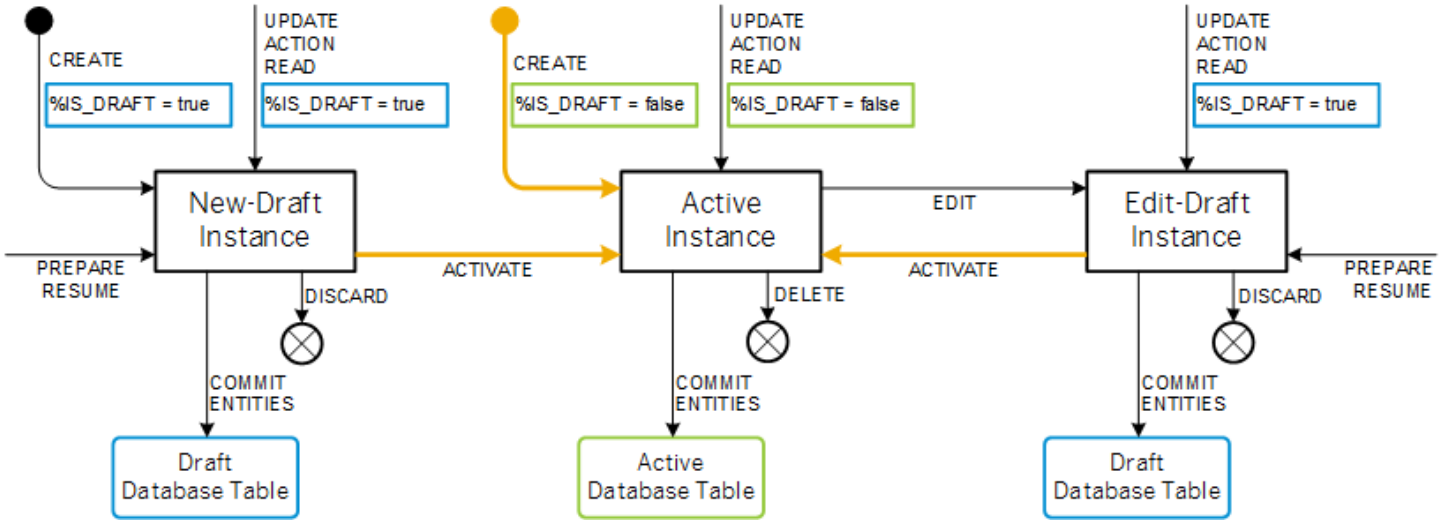
This topic illustrates the execution order of possible runtime activities after executing a modify request for `EDIT`.

Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.



Creating Active Data

You create active data by activating draft data or by directly creating an active instance with a modify CREATE request.



Lifecycle of Draft and Active Data

Modify CREATE Request

Creating a new active instance for a draft BO from scratch is not possible from a Fiori Elements UI. Every create request (OData POST) is sent without the OData draft indicator `IsActiveEntity`. This is interpreted as `false` and the RAP runtime framework creates a new-draft instance.

Via EML, however, it is possible to distinguish modify CREATE requests for active instances and for draft instances. The syntax for directly creating an active instance via EML is the following:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    CREATE FROM
    VALUE #( ( %is_draft          = if_abap_behv=>mk-off  "negative draft indicator
               %control-FieldName = if_abap_behv=>mk-on
               %data-FieldName    = 'Value') )
  REPORTED DATA(create_reported)
  FAILED DATA(create_failed)
  MAPPED DATA(create_mapped).
```

The runtime of a modify request for active instances can be seen in Create Operation Runtime.

ℹ Note

When saving the data from the application buffer on the persistent database table via `COMMIT ENTITIES` or BO-internal triggers of the save sequence, validation executions can prevent modification requests if the data is not consistent, see Save Sequence Runtime.

Activate Action

By executing the draft action ACTIVATE on a draft instance, you copy the draft instance to the active application buffer. It invokes the PREPARE and a modify request to change the active BO instance according to the state of the draft instance. Once the active instance is successfully created or updated, the draft instance is discarded. 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.

The activate action is only possible on draft instances.

On a Fiori Elements UI, the activate action is triggered when choosing the **Save** button after editing data. The **Save** button also triggers the save sequence after the activation.

For executing the `ACTIVATE` via EML, the following syntax is used:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    EXECUTE Activate FROM
    VALUE #( ( %key-FieldName = 'Value' ) )
  REPORTED DATA(activate_reported)
  FAILED DATA(activate_failed)
  MAPPED DATA(activate_mapped).
```

ℹ Note

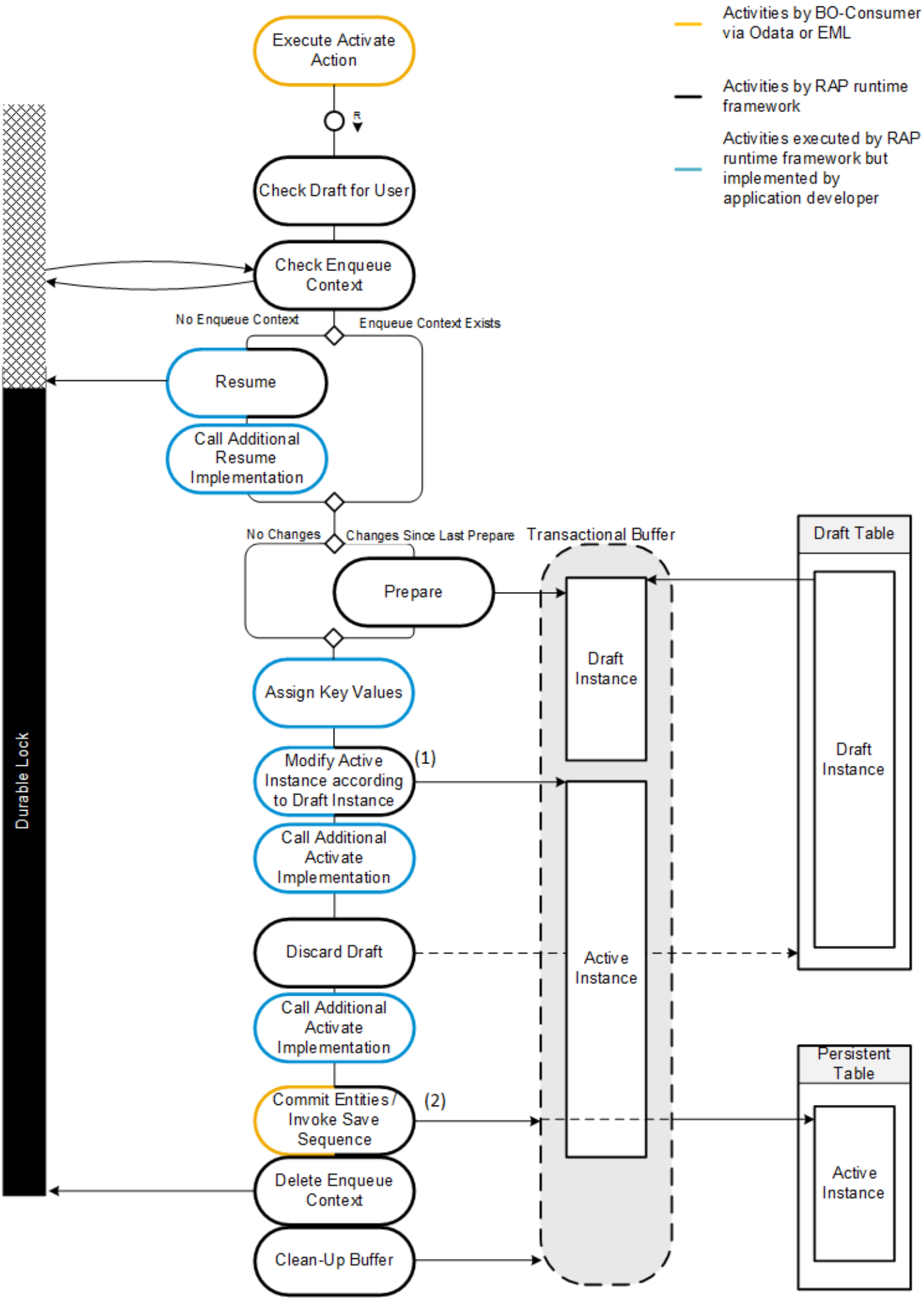
The modify request to adapt the active instance according to the state of the draft BO-instance entails every operation that was done on the draft instance.

In case of an unmanaged implementation scenario, it is just one modify request that is passed to the active unmanaged provider. This can lead to problems if a child draft instance is deleted; and in the same logical unit of work, a new child instance is created with the same key. As the order of the modify requests for delete and create by association is not determined, it can happen that the unmanaged provider executes the `CREATE_BY_ASSOCIATION` before the child instance with the same key is deleted. This scenario raises a short dump during runtime. To prevent this, combine the handler methods for delete and create by association in one handler method in unmanaged scenarios with draft. In this single handler method, strictly define the order so that first, the `DELETE` is executed and then the `CREATE_BY_ASSOCIATION`.

Runtime Activate Action

This topic illustrates the execution order of possible runtime activities after executing a modify request for `ACTIVATE`.

Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.


Activities by BO-Consumer via Odata or EML
Activities by RAP runtime framework
Activities executed by RAP runtime framework but implemented by application developer
Execute Activate Action
Check Draft for User
Check Enqueue Context
No Enqueue Context
Enqueue Context Exists
Resume
Call Additional Resume Implementation
No Changes
Changes Since Last Prepare
Transactional Buffer
Draft Table
Prepare
Draft Instance
Assign Key Values
Modify Active Instance according to Draft Instance
(1)
Call Additional Activate Implementation
Discard Draft
Active Instance
Call Additional Activate Implementation
Persistent Table
Commit Entities / Invoke Save Sequence
(2)
Active Instance
Delete Enqueue Context
Clean-Up Buffer
Durable Lock

i Note

- (1) The detailed runtime of this step is illustrated in the runtime diagrams of the operations in non-draft BOs. See

  Create Operation Runtime,

  Update Operation Runtime,

  Delete Operation Runtime,

  Create by Association Operation Runtime,

  Action Runtime.

- (2) The detailed runtime of this step is illustrated in the runtime diagram of the save sequence. See Save Sequence Runtime.

- The runtime of the `activate` action may vary depending on whether its optimized variant is used or not. See Draft Action Activate Optimized.

Runtime Activate Optimized Action

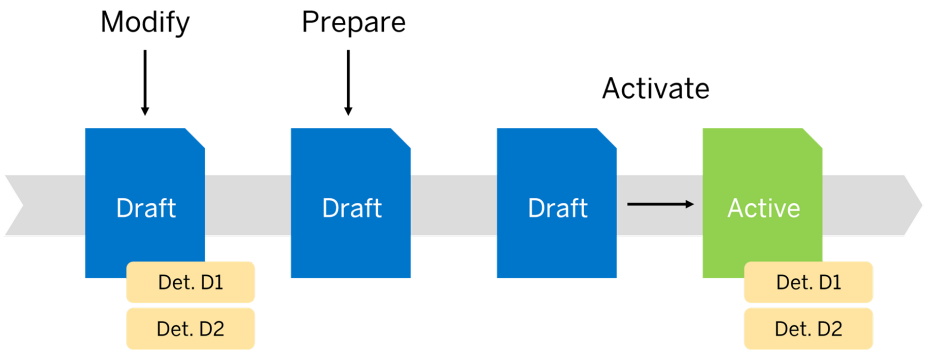
Optimization concept

Determinations and validations may be called multiple times in a RAP transaction. Since every execution of a determination or a validation costs performance, the optimized `Activate` action aims to reduce the number of executions to situations where they are really required. The following sections cover the individual changes in runtime resulting from the usage of the optimized `Activate` action:

Draft activation

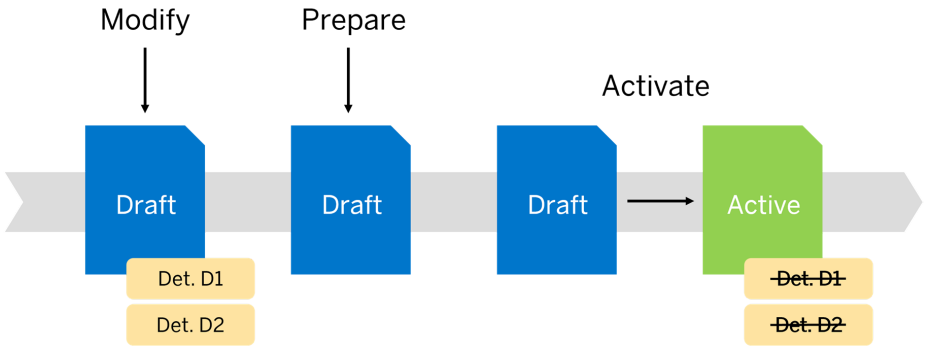
Without optimization

When draft instances are modified, determinations `on modify` are executed if their trigger conditions are met. In the `activate` variant without optimization, these determinations are executed again during draft activation, when the RAP runtime takes over the changes from the draft instances to the active instances.



With optimization

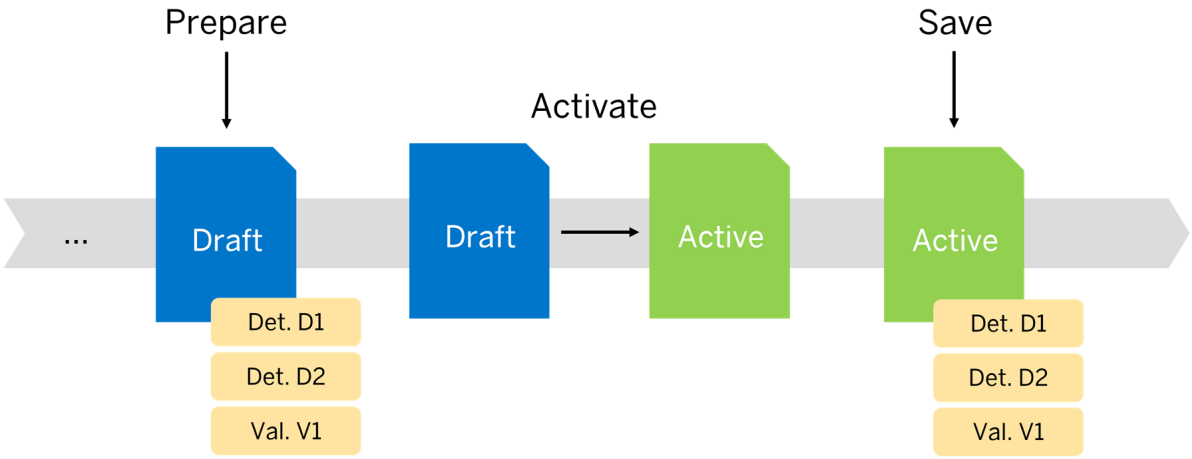
In the optimized variant, the determinations `on modify` are not executed again on the active instances during draft activation. Determinations `on modify` that are triggered by determinations `on save` during `Prepare` are skipped as well.



Save after draft activation

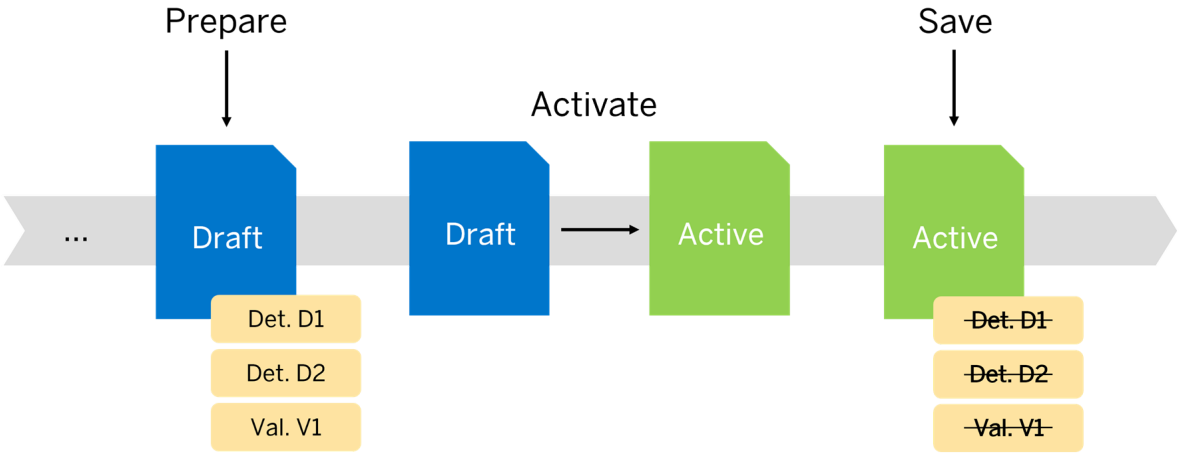
Without optimization

After activating a draft instance, determinations and validations with the trigger time `on save` are executed during the save sequence.



With optimization

Those determinations and validations that have been executed as part of the draft determine action `prepare` before activating the draft instance are not executed again when saving the active instance.



The optimizations outlined above are not applied in the following cases:

- A modification performed after the first determine action triggers a determination/validation.
- A validation in the first determine action reports a failed key.

Consequences of modelling errors

The optimizations outlined above only have the desired effects if the provider follows the best practices for Determination and Validation Modelling consequently.

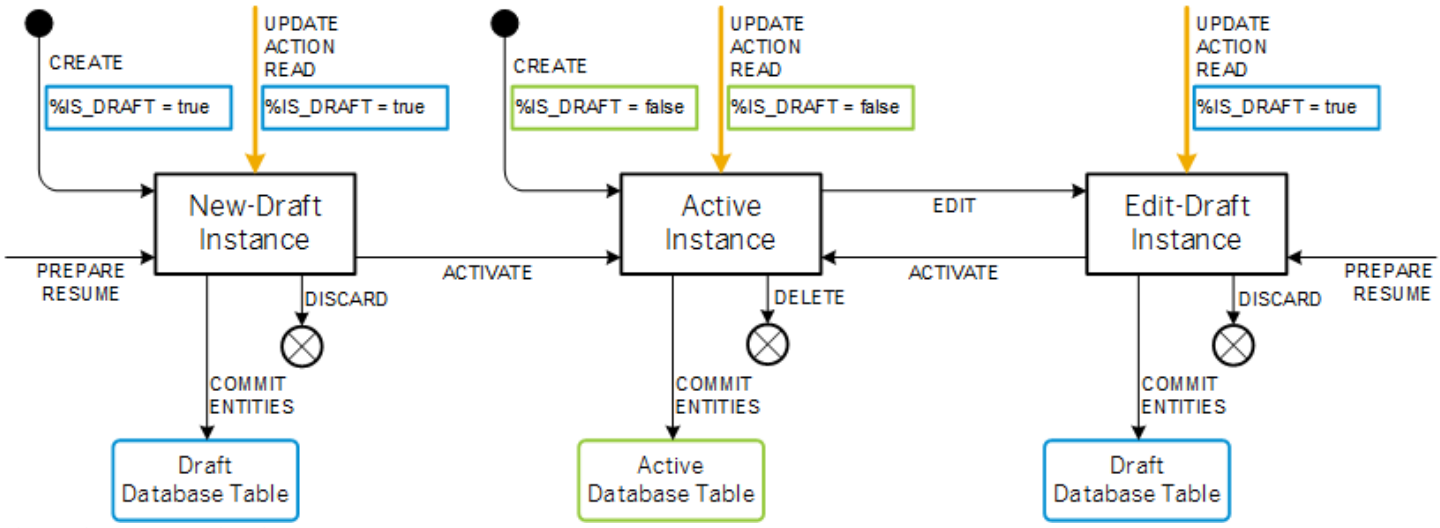
Example

A validation is used to check fields F1, F2 and F3. In order to increase the performance, the validation only validates field F1 during the `Prepare` action by restricting its processing on the instance kind `draft`. Fields F2 and F3 are then validated on active instances during the save sequence. If the BO provider now enables the optimized `activate` action, the execution of the validation during the save sequence is skipped, thus resulting in a missing validation of fields F2 and F3.

The example above shows why the behavior of determinations and validations should be implemented kind independently. Whereas the chosen validation modelling possibly produces the expected behavior and can even lead to performance improvements if the optimized `activate` action is not used, using the optimized variant in combination with the chosen validation modelling, however, leads to a significant change in behavior.

Changing and Reading Instances of a Draft BO

You execute actions on, update, or read instances of a draft BO by sending a modify or read request. The draft indicator decides whether the active or the draft instance is processed.



Life Cycle of Draft and Active Data

Updating Instances of a Draft BO

Fields of draft BO instances are changed by the modify request for UPDATE. Depending on the draft indicator, the request changes the active or the draft instance.

On a Fiori Element UI, the UPDATE with the draft indicator set to `true` is called whenever input fields on the UI are filled. After leaving the input field, the modify request for UPDATE is sent and the draft save sequence is triggered to save the draft on the draft database table.

The UPDATE with the draft indicator set to `false` is called by the activate action, after the prepare action is executed on an edit-draft instance. Hence, the modify UPDATE on active entities is hidden under the functionality of the Save button, see Activate Action.

Via EML you determine whether you want to update an active or a draft instance by setting the draft indicator `%IS_DRAFT = if_abap_behv=>mk-on/off`.

Each modify request for UPDATE, no matter if aimed at draft or active instances, goes through the same runtime process as in scenarios without draft, including locking etc. For more information about the runtime orchestrations, see Update Operation Runtime.

Executing Actions on a Draft BO

Actions on draft business objects are executed by modify requests for action execution. Depending on the draft indicator, the action is executed on the active or the draft instance.

A Fiori Elements UI sends requests for action execution whenever you choose an action button that is defined for an application-specific action. Depending on whether this is done on a draft or an active instance, the draft indicator is set to true or false.

In the EML syntax for executing actions, you can also set the draft indicator to indicate whether you want to execute the action on a draft or an active instance.

Each modify request for action execution, no matter if aimed at draft or active instances, goes through the same runtime process as in scenarios without draft, including locking etc. For more information about the runtime orchestrations of actions, see Action Runtime.

Static actions are always delegated to the active provider. Static actions contain an additional parameter that indicates whether the action produces a draft or an active instance.

Reading Instances of a Draft BO

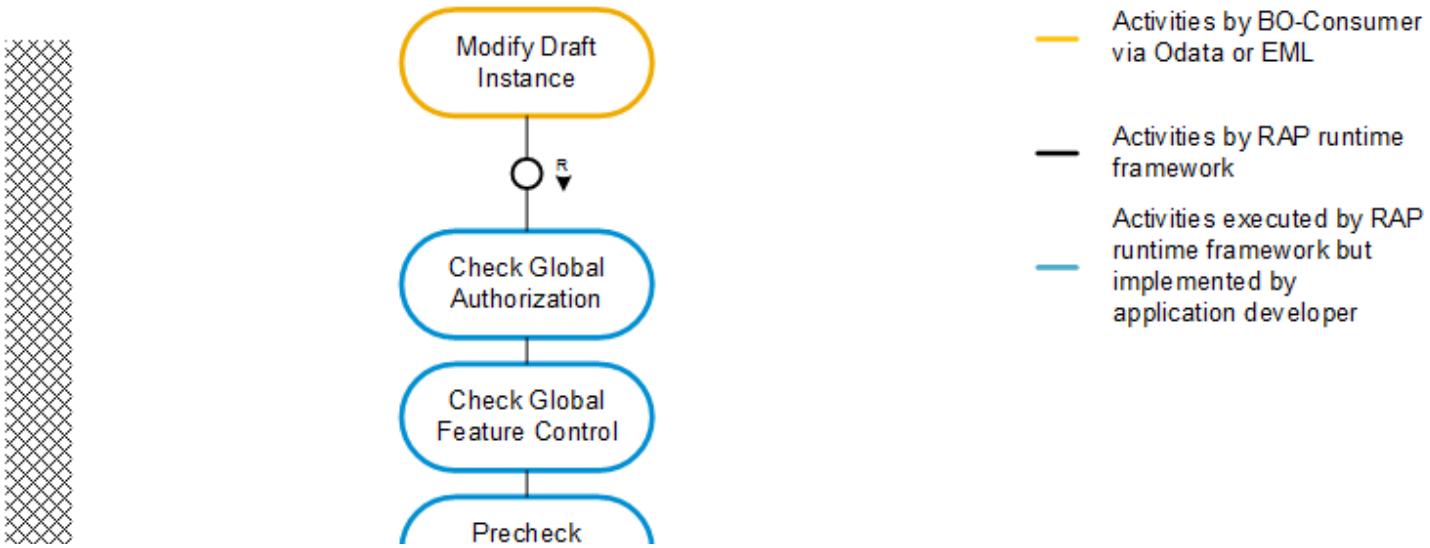
Business object instances of a draft BO are read by a read request. Depending on the draft indicator, the request reads the active or the draft instance.

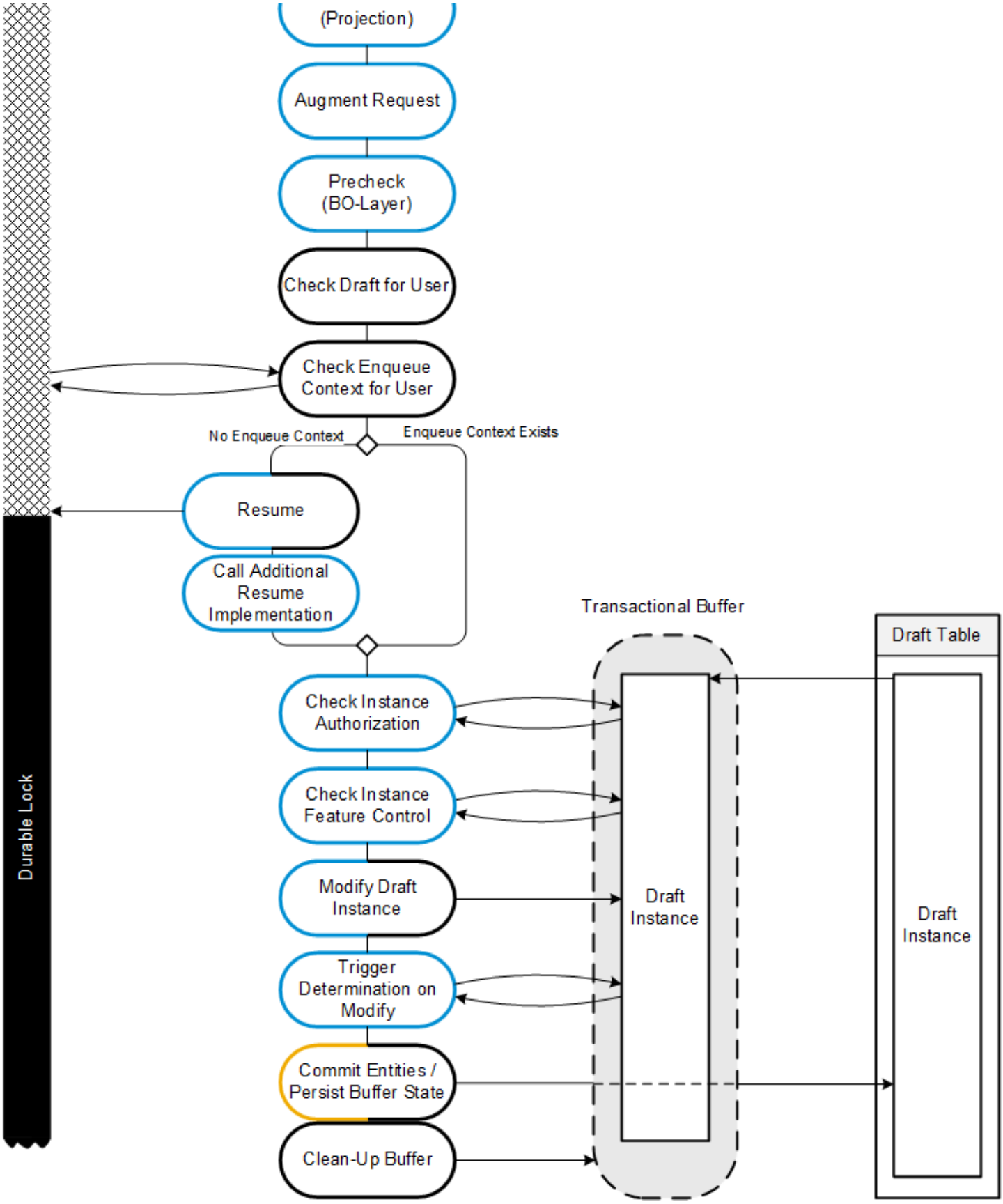
In the reading request via EML, set the draft indicator to determine whether active or draft instances are read. On a Fiori Elements UI, the transactional read operation is always used when buffer information is relevant for the next operation. The draft indicator decides whether the draft or the active instance is read from the buffer.

Runtime MODIFY Draft

This topic illustrates the execution order of possible runtime activities after executing a modify request for UPDATE, DELETE, actions, and create by association.

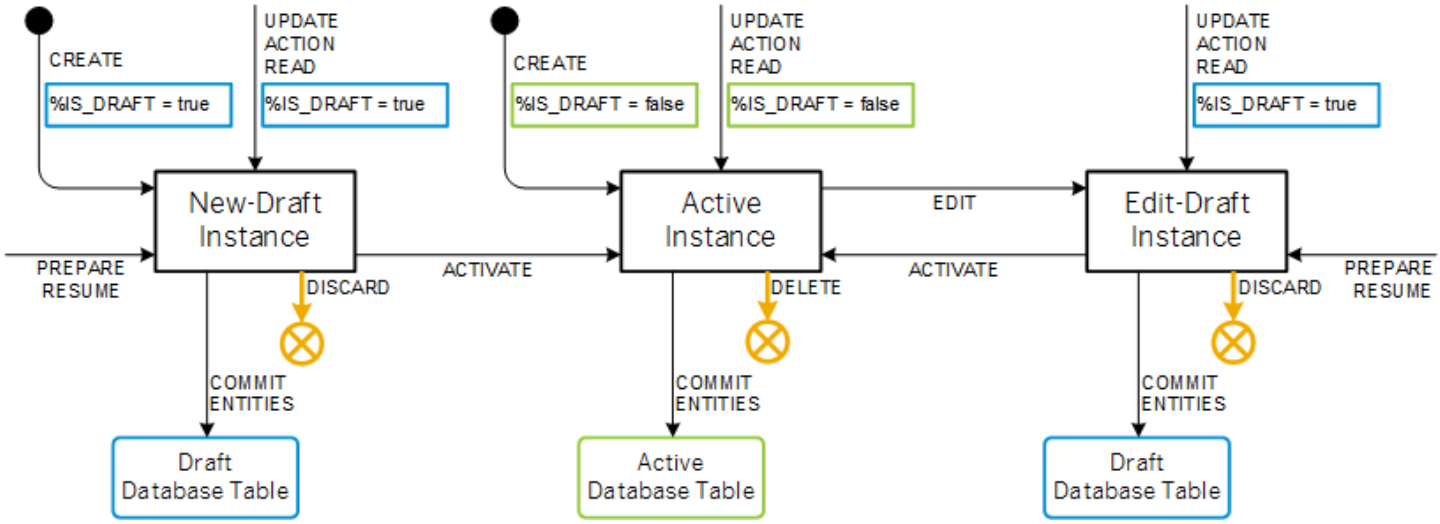
Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.





Deleting Instances of a Draft BO

You delete instances of a draft BO by using the modify request for delete on active instances or the discard action for draft instances.



Life Cycle of Draft and Active Data

Discarding Root Draft Instances

By executing the draft action `DISCARD`, you delete the draft instance from the draft database table. In addition, possible exclusive locks are released. The `DISCARD` can be executed on new-drafts and on edit-drafts. Apart from releasing the lock, the action has no impact on the active instance. It is just the draft data that is deleted.

On a Fiori Elements UI, the draft action `DISCARD` is invoked by choosing **Cancel**. The discard action is always enabled. Neither dynamic feature control, nor authority control can restrict it. A user can always make a draft undone.

Via EML you discard draft instances with the following syntax:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    EXECUTE Discard FROM
    VALUE #( ( %key-FieldName = 'Value' ) )
  REPORTED DATA(discard_reported)
  FAILED DATA(discard_failed)
  MAPPED DATA(discard_mapped).
```

You cannot delete a draft instance with a `DELETE` request. A delete request on a draft instance fails.

Deleting Root Active Instances

As in non-draft business objects, persisted data can be deleted by using the standard operation `DELETE`.

On a Fiori Elements UI, the `DELETE` is called when choosing **Delete** button. If you choose **Delete** for an instance that has an active and a draft representation, the UI sends a request for `DELETE` on the active instance and one for `DELETE` on a draft instance. The `DELETE` on the draft instance is BO-internally considered as a `DISCARD`.

The syntax for deleting instances is the following:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    DELETE FROM
    VALUE #( ( %key-FieldName = 'Value'
               %is_draft      = if_abap_behv=>mk-on|off  "draft indicator) )
  REPORTED DATA(delete_reported)
```

```
    FAILED DATA(delete_failed)
    MAPPED DATA(delete_mapped).
```

Each modify request for `DELETE` goes through the same runtime process as in scenarios without draft, including locking etc. For more information about the runtime orchestrations, see Delete Operation Runtime.

Deleting and Discarding Instances of Child Entities

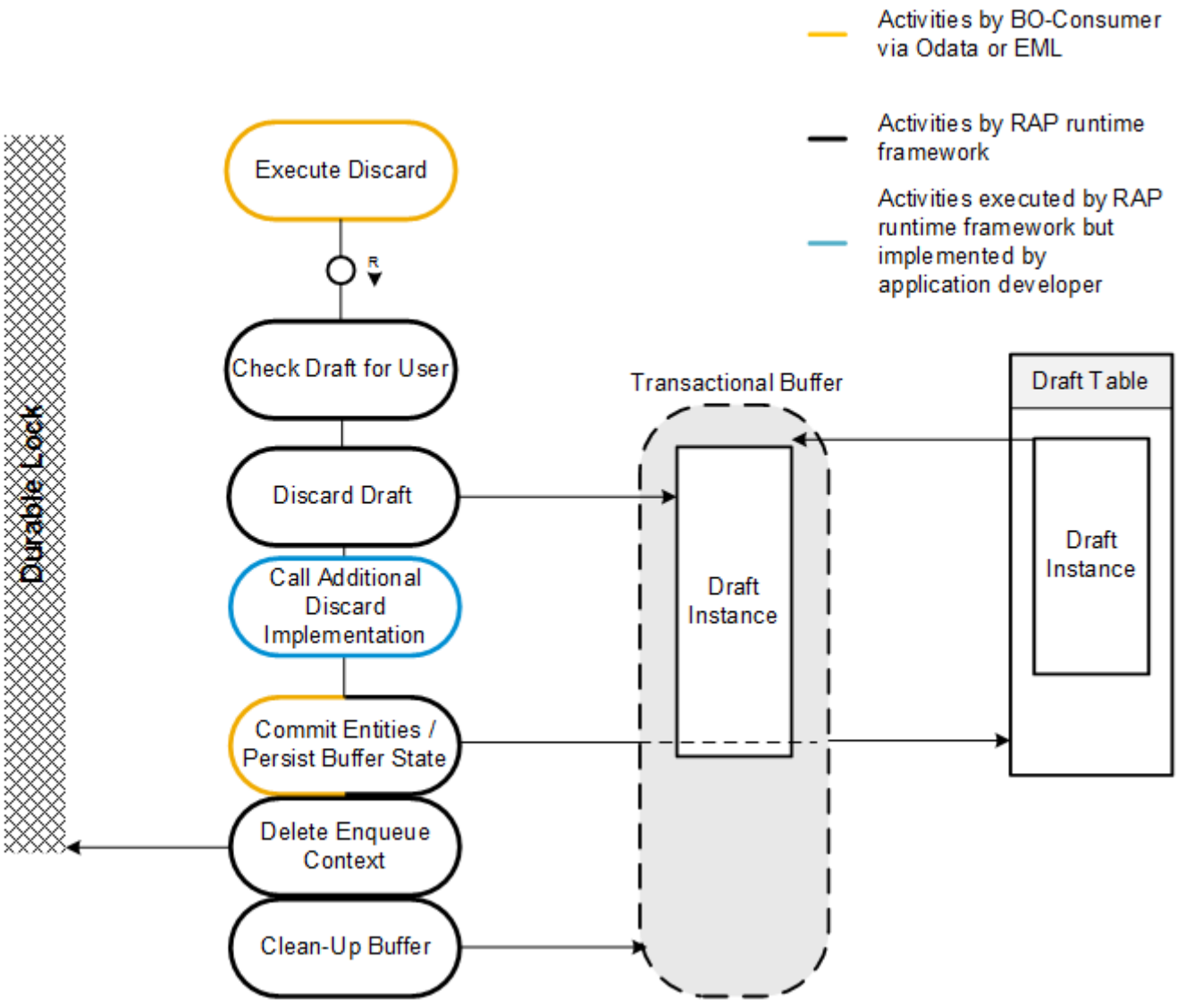
Deleting an active child instance, with a delete request that has the draft indicator set to true, directly deletes the active child instance without affecting the other entities of the business object.

Deleting draft child instances, with a delete request that has the draft indicator set to false, deletes the draft child instance. Once the root entity, and with it all related child entities, is activated, the active child instance is deleted as well. During activation, it is the state of the whole composition tree that is passed to the active buffer. The activate action provides the suitable modify request for the changes that are done on the draft BO.

Runtime Discard Action

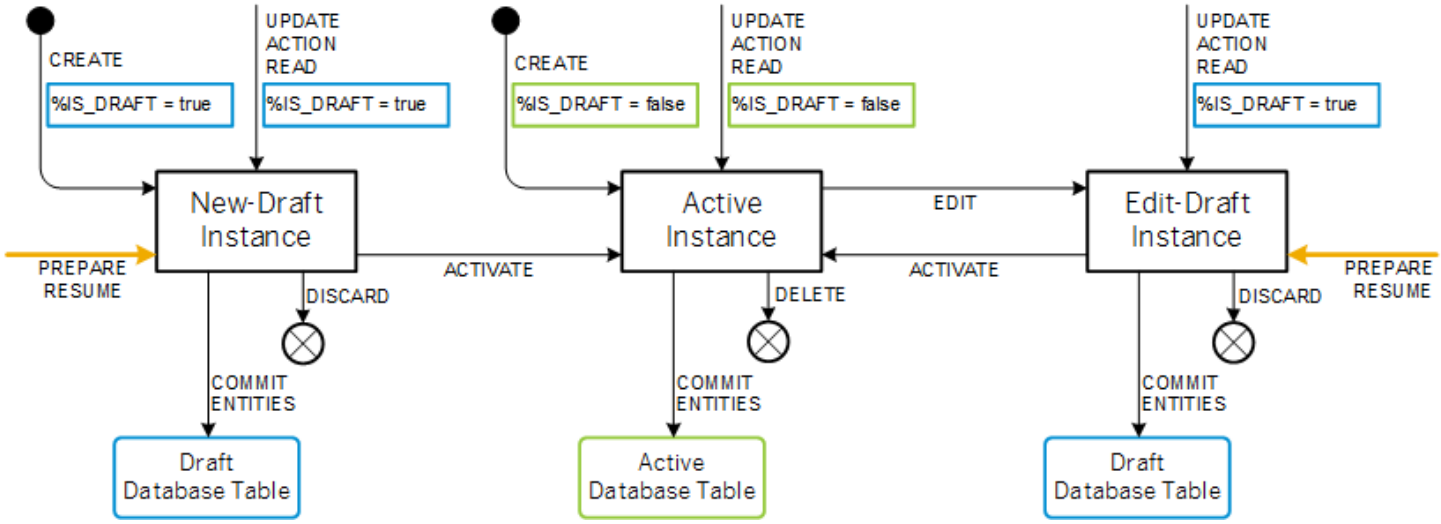
This topic illustrates the execution order of possible runtime activities after executing a modify request for `DISCARD`.

Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.



Preparing Draft Instances for Activation

Before draft instances become active, they are checked by the validations and determinations that are specified for the draft determine action prepare in the behavior definition.



Life Cycle of Draft and Active Data

The draft determine action PREPARE is executed automatically at least once before draft data is activated during the activate action. It ensures that the draft instance is consistent by calling the validations and determinations on save that are specified for the prepare action in the behavior definition. The prepare action can only be executed on draft instances.

ℹ Note

For performance reasons, determinations and validations are not executed during the draft determine action `Prepare`, if these determinations and validations have already been executed during a previous determine action that has been executed on the instance in the same transaction.

This performance optimization is not applied in the following cases:

- A modification performed after the first determine action triggers a determination/validation.
- A validation in the first determine action reports a failed key.
- A determination/validation has been assigned using the addition `always`.

In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

On a Fiori Elements UI, the PREPARE action is invoked by choosing **Save** on a draft instance. The UI then sends the request for PREPARE before executing the action ACTIVATE.

ℹ Note

When choosing **Save** on the UI, the prepare action is executed at least twice. First, the UI requests the PREPARE directly, and then it is called during the execution of the activate action.

Via EML the prepare action is executed with the following syntax:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    EXECUTE Prepare FROM
    VALUE #( ( %key-FieldName = 'Value' ) )
```

```
    REPORTED DATA(prepare_reported)
    FAILED DATA(prepare_failed)
    MAPPED DATA(prepare_mapped).
```

As the PREPARE can only be executed on draft instances, the draft indicator cannot be set for the action.

The prepare action only fails if the instance with the given key is not found. If the validations that are called by the PREPARE detect inconsistencies, the issues are only written in the REPORTED table, but the FAILED table remains empty. A possible COMMIT ENTITIES after the prepare action succeeds if the PREPARE does not return any failed keys, even if the validations within the PREPARE return failed keys.

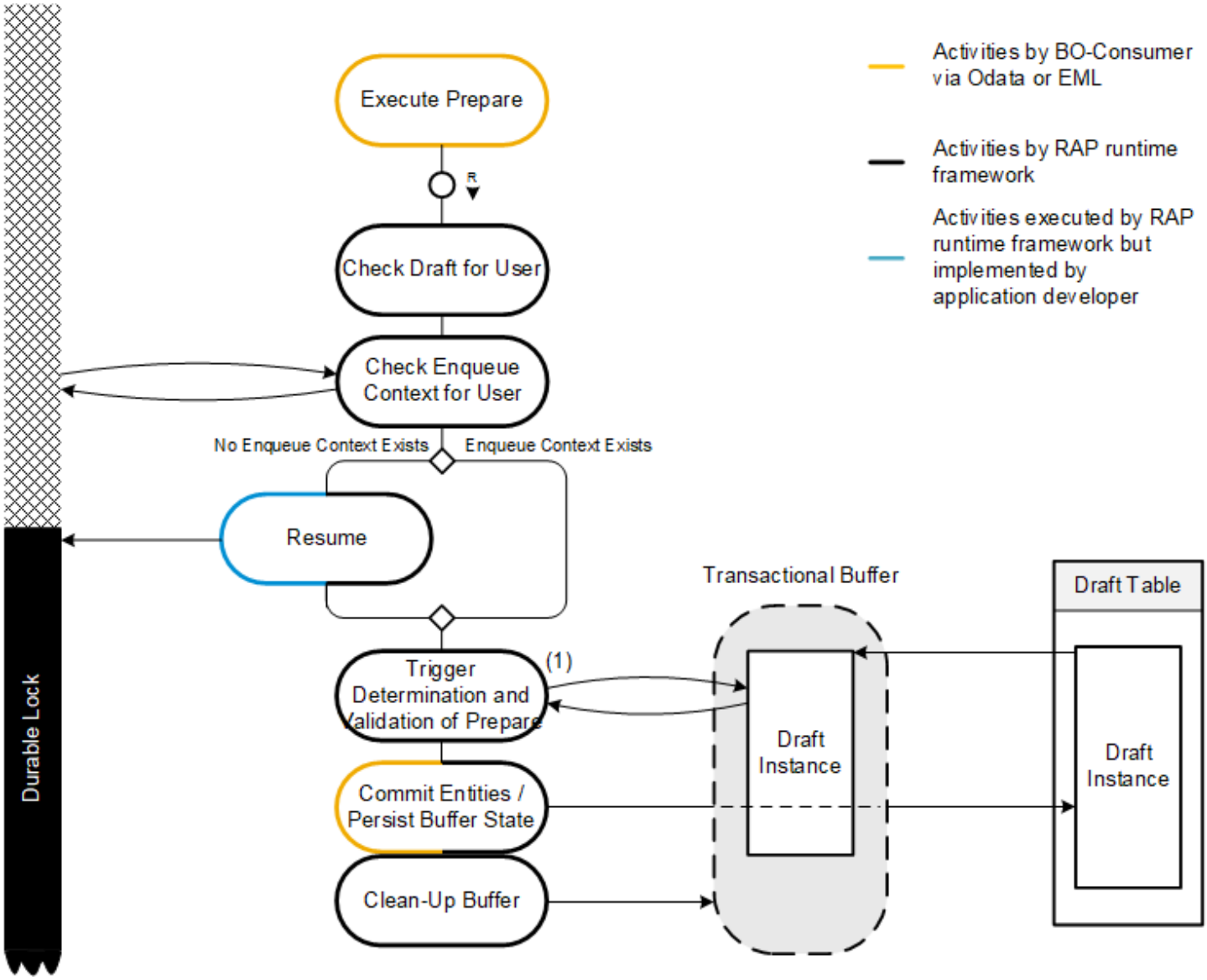
ℹ Note

Only transition messages from validations are handed over to the REPORTED table of the prepare action, state messages are not. State messages describe the state of an instance and saved together with the instance. They are returned if a READ is executed on the respective instance.

Runtime Prepare Action

This topic illustrates the execution order of possible runtime activities after executing a modify request for PREPARE.

Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.



(1):

Note

For performance reasons, determinations and validations are not executed during the draft determine action `Prepare`, if these determinations and validations have already been executed during a previous determine action that has been executed on the instance in the same transaction.

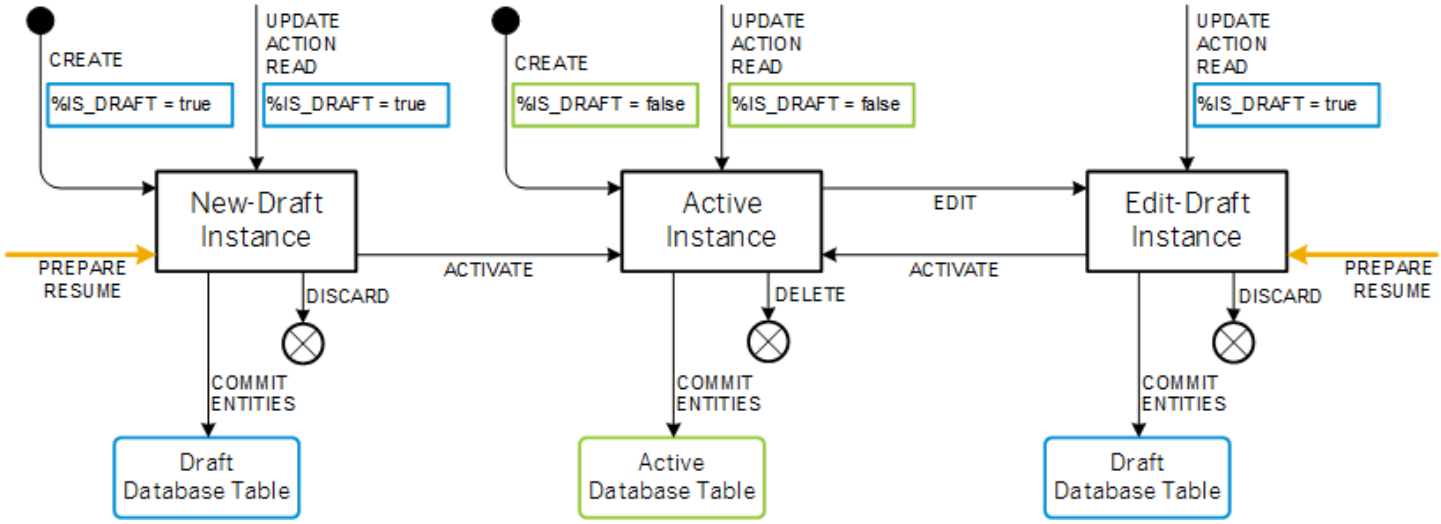
This performance optimization is not applied in the following cases:

- A modification performed after the first determine action triggers a determination/validation.
- A validation in the first determine action reports a failed key.
- A determination/validation has been assigned using the addition `always`.

In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

Resuming Locks for Draft Instances

The draft action resume is called automatically if the work on a draft instance with an expired lock is resumed.



Life Cycle of Draft and Active Data

The draft action RESUME is executed automatically whenever there is a modification on a draft instance whose exclusive lock has expired. It recreates the lock for the corresponding active instance on the active database table.

On a Fiori Elements UI, the action is invoked when the user chooses a draft instance and continues with editing data if the exclusive lock is expired. This is the case when the user chooses an own draft and continues with editing the input fields. An UPDATE on the draft instance is sent, which invokes the resume action if the lock has expired.

ℹ Note

If the user chooses a foreign draft and chooses the **Edit** button to edit any input fields, it is only possible to discard the foreign draft completely and start with a new draft based on the current active data. In this case, the user gets a pop-up with the warning that any changes done by the other user are lost.

Via EML the resume action is executed with the following syntax:

```
MODIFY ENTITIES OF BusinessObjectName
  ENTITY BO_Entity
    EXECUTE Resume FROM
    VALUE #( ( %key-FieldName = 'Value' ) )
  REPORTED DATA(resume_reported)
  FAILED DATA(resume_failed)
  MAPPED DATA(resume_mapped).
```

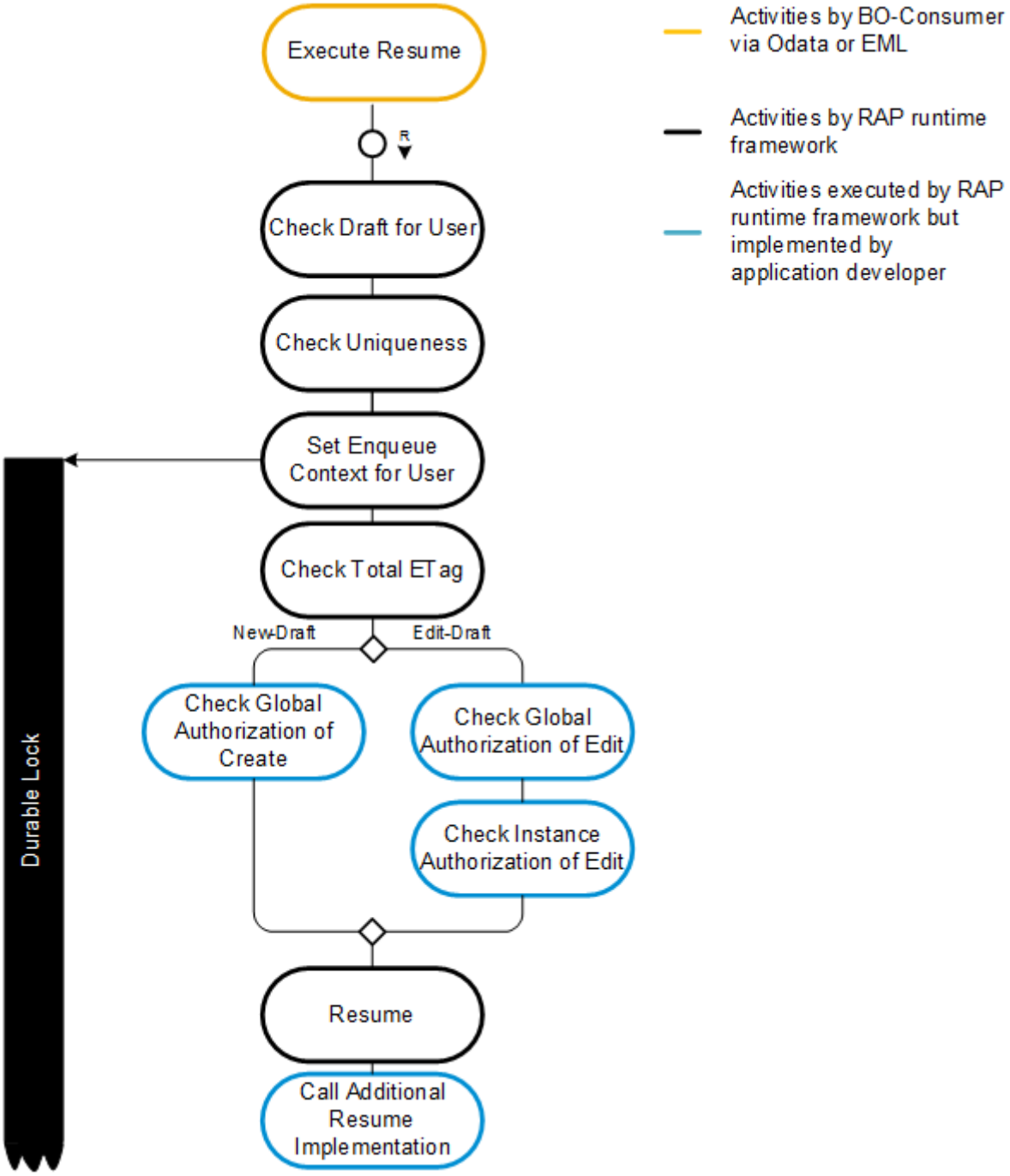
When resuming draft instances it must be ensured that the primary keys of the instance are still unique. It might have happened that other users created an instance with the same primary keys during the optimistic lock phase. For more information, see Uniqueness Check for Primary Keys.

During the execution of a resume action, the framework runs the same checks as for a `MODIFY CREATE` in case of a new-draft, and the same checks as for an `EDIT` in case of an edit-draft.

Runtime Resume Action

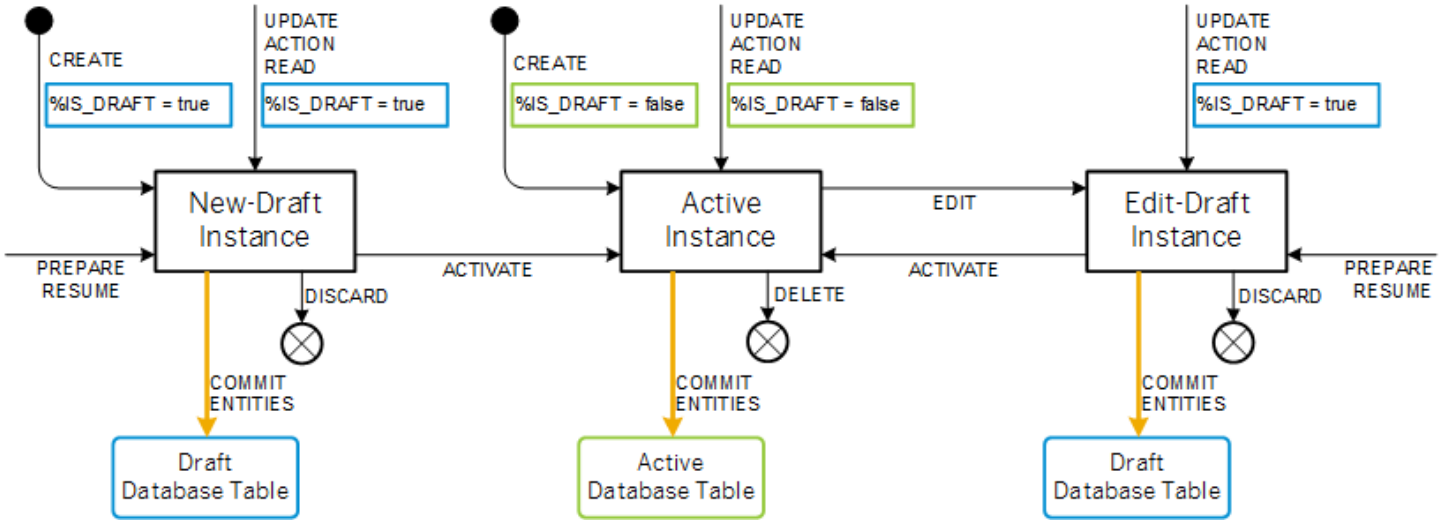
This topic illustrates the execution order of possible runtime activities after executing a modify request for RESUME.

Not all activities of the following runtime diagram are always present. It depends on the BO's behavior that is specified in the behavior definition.



Saving Data to the Database

Persisting business object instances is done during the save sequence. Draft instances are automatically saved on the draft database by the RAP runtime framework.



Life Cycle of Draft and Active Data

Depending on the state of the instance you want to save, the SAVE is done on the draft database table or the active database table. Active instances are saved on the active database tables; draft instances are saved on the draft database table.

Via EML, the statement `COMMIT ENTITIES` triggers the process to save the data on the database. It saves the state of the buffer, draft instances to the draft database table and active instances to the active database table. For active instances, the save sequence is triggered. For draft instances, only the actual save to the database is done. No validations or determinations on save are called.

Saving Active Instances

For saving active data to the persistent database table, the data of the instance must be consistent in the buffer. The actual SAVE is done during the save sequence, which is triggered after the interaction phase, see Save Sequence Runtime.

On a Fiori Elements UI, the save sequence is triggered by choosing the **Save** button on the UI. After the PREPARE and the `ACTIVATE` are executed successfully, the save sequence is triggered automatically.

ℹ Note

- The runtime of the save sequence may vary with respect to determinations and validations depending on whether the optimized variant of the `activate` action is used or not. For more information, see Draft Action Activate Optimized.
- For performance reasons, determinations and validations are not executed during the save sequence, if these determinations and validations have already been executed during a previous determine action that has been executed on the active instance in the same transaction. This performance optimization is not applied in the following cases:
  - A modification performed after the determine action triggers a determination/validation.
  - A validation in the determine action reports a failed key.

    In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

Saving Draft Instances

Draft data does not have to be in a consistent state to be saved. No validations are triggered when saving draft data to the draft database. The save sequence of the active provider is not triggered. That means, there is no `Finalize` and no `Check_Before_Save` when saving draft instances on the draft database table.

On a Fiori Element UI, the state of the application buffer is saved to the draft database table whenever the input fields are edited. This is done for each field separately. That means, after leaving one input field, the data is transferred to the draft database table. The UI displays the message **Draft saved** after each roundtrip.

Authorization Control

Authorization control in RAP protects your business object against unauthorized access to data.

Context

To define which consumers under which circumstances are allowed to read or change data of a business object, RAP offers an authorization concept for application developers to restrict access to the business object. Authorization control is always relevant when the permission to execute an operation depends on the role of the business object consumer.

This contrasts to feature control, where the permission for operations depends on the state of existing instances, or on BO-external factors that don't relate to the user.

The authorization for consumers is managed and maintained by the system administrator and grouped into consumer roles. Authorization objects define the authorization for the respective user roles for certain operations. In RAP each read or modify request can be checked via authorization objects against user roles before the request is finally executed and reaches data. The authorization check with authorization objects is called from CDS entities in case of read requests and from the behavior implementation in case of modify requests.

For more information, about using authorization objects in your implementation, see User and Role Administration of Application Server ABAP.

For more information, about the general authorization concept for RAP business services, see Authorization Basics in the Identity and Access Management (IAM) Guide.

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's 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 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.

! Restriction

You can't define authorization control or feature control for functions.

In addition, DCL is also automatically evaluated in case of transactional read access for managed business objects, that is when using EML-based read and read-by-association operations. In unmanaged business objects, the transactional read operation must be implemented by the application developer. Hence, DCL must be checked in this implementation.

Authorization Checks for Modify Operations

In RAP business objects, modifying operations, such as standard operations and actions can be checked against unauthorized access during runtime. To retrieve user authorizations for incoming requests, authorization objects are included in the behavior implementation for your business objects. Authorization objects return authorization values.

You have dedicated authorization implementation options in RAP business objects available. For them, authorization is defined in the behavior definition and implemented in the methods for authorization. These dedicated authorization methods are called during runtime at a specific point in time before the actual modify operation. Hence, you can prevent modify operations with authorization control before the actual modify operation is executed on the transactional buffer. Authorization checks in RAP authorization control methods can only check against the state of the BO before the request was executed, the so-called before image, because these methods are called before the actual modify request is executed. Hence, it isn't possible to check the authorization of a user against the incoming values. To implement authorization checks against incoming values, see Authorization Check against Incoming Values (Precheck).

To see when the authorization exits are called at runtime when updating a draft instance, see Runtime MODIFY Draft.

Global Authorization

Global authorization is used for all authorization checks that only depend on the user. You can define global authorization to check if users are allowed to execute an operation in general.

Example

Only HR representatives can create new instances in an employee master data BO.

During the runtime of the CREATE request, the authorization is checked against the role of the user.

Global authorization is the first check for incoming requests. With global authorization you can reject the request before it reaches any other method of the behavior handler classes.

Global authorization checks can be implemented for both, static, and instance-bound operations.

Instance Authorization

Instance authorization is used for all authorization checks that, in addition to the user role, depend on the state of the entity instance in question. With instance authorization, you can define authorization that depends on a field value of the instance.

Example

Managers can only change the salary amount of their own employees.

In this case, the employee entity must contain a field that refers to the manager. During the runtime of the UPDATE request, the value of the manager field is compared to the authorization of the user role. Only if for this value the user is authorized to update the instance, the update request is approved and can be executed. Otherwise, the request fails.

Instance authorization is only possible for instance-based operations. Operations that don't operate on a specific instance can't be checked with instance authorization. Operations that are excluded from instance authorization are CREATE and static actions. Technically speaking, the implementation method for instance authorizations requires the keys of the instance for the authorization check. Static operations don't relate to one specific instance, and thus, can't provide a key.

Authorization Check against Incoming Values (Precheck)

Authorization checks can be implemented in the corresponding precheck method for the operation to check against incoming values. In this case, the unwanted values don't even reach the transactional buffer.

⚠ Caution

Fiori Elements UIs don't fully support the behavior of authorization checks in precheck methods.

For more information, see Precheck and for an implementation example refer to Implementing Prechecks.

Authorization in Projection Behavior Definitions

Generally, CRUD operations and actions reused with `use` inherit the authorization control from the base behavior definition. If you define new actions on projection level as described in Actions and Functions in Projection Behavior Definitions and the projection behavior definition is defined as `strict`, you can add authorization control for these actions on projection level. Contrary to the base behavior definition, you must define each projection definition individually either as `authorization global` or `authorization instance`. There's no master/dependent relation between the different projection definitions. Additionally, you can define an action with `authorization:none` to exclude it from authorization checks.

For more information about syntax and examples, refer to CDS BDL - authorization, projection BDEF (ABAP Keyword Documentation).

Authorization in Draft Scenarios

Authorization checks in RAP authorization control methods are executed before the draft instance is modified. The authorization methods are called for the operations that operate on draft instances and for the draft actions `EDIT` and `RESUME`, see Runtime Edit Action and Runtime Resume Action. In case of `RESUME`, the authorization control for `CREATE` is checked when executed for a new-draft and it's called for `EDIT` when the `RESUME` is executed for edit-drafts.

Note

The authorization methods aren't called when activating the draft instance. If you want to prevent that a draft instance is activated, you must implement the authorization check in a validation.

Consumer Hints in UI Scenarios

In UI scenarios, consumer hints, which are retrieved from the backend, provide the information for the UI which operations are enabled for the current situation. On a Fiori Elements UI, action and other operation buttons are only clickable if authorization and feature control methods provide a positive result under the current conditions (the state of the retrieved BO instances).

The definition and implementation of authorization and feature control is important, as the consumer hints are based on the result of the authorization and feature control. The implementation methods for authorization and feature control are called on displaying the list report or the object page for a specific instance. By checking the authorization for all the exposed actions beforehand, users can't even start operations, for which they aren't authorized, as their triggers (action buttons) aren't available.

Example

The action `SetStatusBooked` can only be executed for instances, whose status isn't booked yet (feature control) and by users that are assigned to a specific role (authorization). When selecting instances, whose status is booked already, the button for this action isn't available on the UI and the user can't execute the action.

Related Information

Authorization Fields and Objects
Checking Quality of ABAP Code with ATC

Authorization Definition

Authorization is defined in the behavior definition of a RAP business object.

Syntax

Like other behavior characteristics, authorization is defined in the header of the behavior definition for a business object entity.

For details about the syntax of authorization in the behavior definition, see CDS BDL - authorization (ABAP Keyword Documentation).

Once you have defined authorization in the behavior definition you can use the quick fix to generate the corresponding methods in the specified behavior pool.

Authorization Master

An entity is defined as authorization master (`authorization master ()`) if the operations of this entity have their own authorization implementation. That means in the behavior implementation of this entity, the authorization control must be implemented in the corresponding method for authorization (global or instance). For the authorization master, you must define either global, or instance, or both.

Authorization Dependent

An entity is defined as authorization dependent (`authorization dependent by _Assoc`) if the authorization control from the authorization master entity shall also be applied for the operations of this entity. In this case, for the authorization check for the update, the delete, and the create-by-association operation on an authorization dependent entity, the authorization check for the update operation of the authorization master entity is applied.

For actions of authorization dependent entities, as well as for create-enabled associations that are not compositions, the authorization control must be implemented in separate methods for authorization in the behavior implementation class of the authorization dependent entity. It depends on the definition of the authorization master entity, if global or instance authorization must be implemented for these actions.

ℹ Note

For the definition of authorization dependent entities, you have to specify the association to the authorization master entity in the behavior definition, even though it is implicitly transaction-enabled due to internal BO relations.

```
{
  association _AssocToMaster;
}
```

ℹ Note

In unmanaged business objects, the definition of authorization dependent entities requires the implementation of the read-by-association method. As authorization checks on authorization dependent entities are delegated to the authorization master, the read-by-association is used to get the authorization information from the master entity.

In managed business objects, the read-by-association is provided by the RAP managed BO provider.

If you define an authorization master in a business object, it is recommended that all other entities are authorization dependent entities.

Global Authorization

By defining global authorization (`authorization master (global)`), you implement authority control for the following operations of the entity:

- Create
- Create-by-association
- Update
- Delete
- Static Actions
- Instance Actions

> **Note**
>
> Actions can define their own authorization control and overrule the authorization master with instance authorization or both, global and instance authorization.

The authorization check is implemented in the corresponding method in the behavior implementation. For more information, see Global Authorization Implementation.

The global authorization check is called during runtime before the request reaches the backend. In case of a scenario with draft, if a user is not authorized, a request is rejected before a draft is saved on the draft database table.

Instance Authorization

By defining instance authorization (`authorization instance ()`), the following operations of the entity can be checked against unauthorized access:

- Create-by-association
- Update
- Delete
- Instance Actions

> **Note**
>
> Actions can define their own authorization control and overrule the authorization master with global authorization or both, global and instance authorization.

The authorization check is implemented in the corresponding method in the behavior implementation. For more information, see Instance Authorization Implementation.

The instance authorization check is called during runtime before the actual execution of the requested operation. In draft scenarios this means that the draft instance already exists, but its activation is prevented.

You can define both, global and instance authorization. It is possible to check instance-based operations in the global and the instance authority check. The checks are executed during different points in time during runtime.

Authorization Context

A behavior definition can define authorization contexts that list multiple authorization objects.

Such an authorization context can be defined as own authorization context, which includes all authorization objects that are checked by the BO. In this case, the statement `define own authorization context` is used.

Furthermore, such an authorization context can be defined as an authorization context with a specific name, which names the authorization objects that can be disabled through privileged mode, using the statement `define authorization context`.

Additionally, you can use the notation `define own authorization context by privileged mode` when the own authorization context is identical to the context used for the privileged mode, meaning when privileged mode disables all authorization objects without exception.

For more information about syntax, see RAP - Authorization Context.

Privileged Mode

RAP business objects can offer privileged mode. With privileged mode in the behavior definition, RAP business object consumers can circumvent authorization checks performed by authorization objects that are called, for instance, by global- and instance authorization handlers.

You define privileged mode for the BO by disabling the authorization context using the clause `with privileged mode disabling`. As a result, authorization objects in this authorization context are not checked during privileged access.

This is an example for the provider BDEF that offers the privileged mode:

```
…
with privileged mode disabling NoCheckWhenPrivileged;
define authorization context NoCheckWhenPrivileged
{
  'AUTHOBJECT';
}
define own authorization context by privileged mode;
…
```

In the context of the example above, this means that if the BO is being accessed with privileged mode, the authorization object `AUTHOBJECT` of the authorization context `NoCheckWhenPrivileged` is not checked.

For more information about syntax, see RAP - Authorization Context .

For more information on how to access a BO which offers privileged mode through EML, see Entity Manipulation Language (EML)(Section `PRIVILEGED`).

Precheck

With a precheck implementation you can check against incoming values and deny incoming requests before data reaches the transactional buffer. For more information, see Operation Precheck.

The precheck requires an implementation in the behavior pool. For more information, see Precheck.

Authorization Exclusion

By using the syntax element `authorization: none` on an operation in the entity's behavior definition, you exclude the operation in question from authorization checks. You can disable operations from authorization checks in authorization master and dependent entities.

Authorization Delegation

By using the syntax `authorization: update` on an operation in the entity's behavior definition, you can use the authorization check that is implemented for the update operation for the annotated operation. The derived types in the augmentation implementation methods do not show the annotated operation in requested authorizations in this case.

Authorization can be delegated for the following operations:

- `delete`
- `create-by-association`
- actions

The authorization for internal actions cannot be delegated. Internal actions do not have authorization checks in general, as they are only invoked internally. Likewise, it is not possible to delegate the authorization for the draft action Edit .

You can delegate the authorization for operations to the update operation, even if the update operation is not enabled for this entity.

→ Remember

Standard operations in authorization dependent entities are automatically delegated to the update operation of the authorization master entity.

If you delegate the authorization for an action in an authorization dependent entity to the update operation, it will be delegated to the update operation of the authorization master entity in the end.

Authorization Implementation

Authorization checks are implemented in the behavior pool with the respective methods for global authorization, instance authorization or prechecks.

The authorization that is defined on a business object's entity must be implemented in the corresponding local handler class (`lhc_handler`) of the behavior pool. As depicted in the listing below, each such local class inherits from the base handler class `CL_ABAP_BEHAVIOR_HANDLER`. The authorization check is implemented in this handler class using the corresponding methods global authorization or instance authorization. To check authorization for incoming values, you implement the respective method for the precheck.

Authorization Implementation in the Behavior Pool

The method declarations can be generated via a quick fix when defining authorization in the behavior definition. Global and instance authorization need to be handled separately from provider perspective as they're consumed during different points in time.

Global Authorization

The global authorization method is called as the first method in the handler class, when a modify or create request is executed, see Update Operation Runtime.

As global authorization is independent of the state of the entity and just checks the authorization of the user that has executed the request, the global authorization handler is used to forbid certain operations for certain user groups in general. A typical use case is to only allow specific user groups to create new instances.

For a detailed implementation example, see Implementing Global Authorization.

For more information about the implementation in the behavior pool, see FOR GLOBAL AUTHORIZAITON (ABAP Keyword Documentation).

Instance Authorization

The method for instance authorization is called, right before the actual modification of the respective instance happens, see Update Operation Runtime. At this point in time, during the runtime execution, you can only make checks against the state of the instance as it was before the modification request was triggered. This state, the so-called before image is relevant for authorization checks in the instance authorization method. You don't have the option to check against incoming values.

Example

Travel instances with an agency ID that have a certain country/region code can only be changed by a specific user group. In this example, you implement the authorization check in the instance authorization method and select the agency data set from the database table `/DMO/agency` to check the value of the country/region code field of the before image instance to pass to the authorization object. The actual authorization check is done by the authorization object if you have defined authorization values.

For the detailed description of this example, see Implementing Instance Authorization.

For more information about the implementation in the behavior pool, see FOR INSTANCE AUTHORIZATION, AUTHORIZATION (ABAP Keyword Documentation).

Precheck

The precheck implementation checks incoming values against the before image of a RAP BO to determine whether certain values can be set in a request. The precheck implementation is called before the requested changes can reach the transactional buffer.

Example

A travel agency is only allowed to create or update travels for the country/region it's located in. Then the requests must be checked against the country-code of the respective travel agency to determine whether an agency has permissions to change or create the respective travels.

For a detailed implementation, see [SET LINK].

For more information about the implementation in the behavior pool, see FOR PRECHECK (ABAP Keyword Documentation).

Operations

This section describes the operations that are available for RAP business objects and their functionality.

Standard Operations

Standard operations in RAP business objects are operations that cover the standard functionality for creating, updating, deleting and locking RAP BO instances.

Nonstandard Operations

Nonstandard operations in RAP are operations that do not provide the canonical behavior of RAP BOs. Instead, they are used to provide customized, business-logic-specific behavior.

Operation Precheck

With a precheck implementation you can deny incoming requests before data reaches the transactional buffer.

Operation Augmentation

With an augmentation implementation you can add data or modify incoming requests on the projection layer before data reaches the transactional buffer.

Operation Defaulting

With a default values function you can default the input parameters for actions and functions as well as fields for create and create by-association operations in OData services. This way, default values functions serve as an input assistant for end users that trigger the respective operation on the UI.

Standard Operations

Standard operations in RAP business objects are operations that cover the standard functionality for creating, updating, deleting and locking RAP BO instances.

Create Operation Runtime

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

For more information about runtime specifics, see Create Operation Runtime.

Update Operation Runtime

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

For more information about runtime specifics, see Update Operation Runtime.

Delete Operation Runtime

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

For more information about runtime specifics, see Delete Operation Runtime.

Create by Association Operation Runtime

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

For more information about runtime specifics, see Create by Association Operation Runtime.

Lock Operation

In RAP, the `lock` operation is a standard operation that locks instances of a business object entity.

For details about the lock operation in RAP, see Pessimistic Concurrency Control (Locking).

Nonstandard Operations

Nonstandard operations in RAP are operations that do not provide the canonical behavior of RAP BOs. Instead, they are used to provide customized, business-logic-specific behavior.

Actions

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

Functions

A function in RAP is a custom read-operation that is part of the business logic.

Actions

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

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

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

Setting the Status to `booked` by an Action

In general, however, actions can have a much wider scope than just changing single values of an instance. You can create instances or implement more complex procedures in an action.

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

Triggers for Actions

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

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

! Restriction

There are some restrictions in exposing actions for OData V2 services:

- For importing parameters only simple types are supported
- Results with collection of complex type in collections are not supported.
- Result entity is only supported with cardinality `[0..1]` or `[1]`.

Related Information

Action Definition
Action Implementation
Action Runtime

Action Definition

You define actions for an entity in the behavior definition.

Syntax for Defining Actions

Actions are specified as non-standard operations in behavior definitions and are implemented in the ABAP behavior pool.

For details about the syntax, see CDS BDL - action (ABAP Keyword Documentation).

Input Parameter

For details about input parameters for actions, see CDS -BDL - input parameter (ABAP Keyword Documentation).

For details about modeling of parameters in RAP, see Modeling Parameters for Non-Standard Operations.

For more information about the modeling of actions, see Action deductDiscount in the development guide for Transactional Apps with Draft Capabilities.

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

For OData services, you can default the input parameters of actions by using default values functions. A default values functions serves as an input assistant for input parameters on a Fiori UI by calculating their default values and by providing them as its result for the respective action. The calcuated values are then display in a popup window for the input parameters which opens when the action is triggered by the end user. For conceptual information on defaulting input parameters of operations, see Operation Defaulting.

Output Parameters

For details, about output parameters, see CDS BDL - output parameter (ABAP Keyword Documentation).

For details about modeling of parameters in RAP, see Modeling Parameters for Non-Standard Operations.

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

Factory actions return their result in the `mapped` response parameter by mapping the keys to the `%cid` that was provided by the consumer.

By declaring the action result as `selective` you can define that the action consumer can decide whether the result shall be returned completely or only parts of it, for example the keys only. This can help to improve performance as performance consuming calculated fields can be excluded from the result. For more information about the implementation with selective result parameter, see Action Importing Parameter.A Fiori UI requests only the keys of the result when an action with selective result is executed.

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

You can return either a result entity or a result structure:

- **Result Entity:** By declaring the action with `result entity MyBOEntity`, the action returns one ore more instances of `MyBOEntity`. `result $self` returns one or more instances of the entity the action is assigned to.

  When you return a result entity with `$self` in a UI service, the current UI behavior is that it stays on the same page where they action is executed. For factory actions, a Fiori Elements UI currently navigates to the object page of the newly created instance.

  > **Note**
  > Only actions and functions having output entities that are included in the service definition are exposed in the service.

  In a projection behavior definition, result entities other than `$self` must be redefined with the projection result entity. For more information, see Actions in Projection Behavior Definitions.

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

Action Types

Internal Action

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

Static Action

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

Repeatable Action

The addition `repeatable` allows you to define an instance action that can be executed multiply on the same instance in the same EML request or OData changeset. The different executions of such an action can be distinguished using the content ID

`%CID`. The content ID is contained in the derived type of repeatable actions and needs to be provided in every call of a repeatable action.

> **Note**
>
> If bound non-factory actions or functions are executed multiply on the same instance (but maybe with distinct parameters) then the assignment of the operation result to the request operations is ambiguous for the consumer and for the provider. This is due to the missing operation ID. For OData consumption of such action sequence, this can lead to runtime errors. Use repeatable actions for this use case to avoid these errors.

Factory Action

With factory actions, you can create entity instances by executing an action. Factory actions can be instance-bound or static. Instance 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 prefilled default values.

In contrast to a non-factory action, the handler method of factory actions offers `%cid` handling for the result instance. The result is determined via the response parameter `mapped`. There is no `result` parameter. In contrast to non-factory actions, the `%cid`, which is provided by the consumer when executing factory actions, is mapped to the final key value. This helps the consumer identifying the newly created instance.

You can also define default factory actions that replace standard operations in certain scenarios when evaluated by the consuming framework, see Default Factory Action (ABAP Keyword Documentation).

Save Actions

Save actions are actions that can only be triggered during the RAP save sequence specifically during the `FINALIZE` or `ADJUST_NUMBERS`. Calling a save action defined as `save(finalize)` during the interaction phase like for example from a `determination on modify` results in a short dump.

For an overview of the RAP save sequence runtime, see Save Sequence Runtime.

With the addition `save (finalize| adjustnumbers)`, you can define the handler method in which the on save action can be called.

Action Additions

External Action Name

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

If you want to define an action button for an action with an external name, the external name must be used in the `@UI` annotation. For more information, see UI Consumption of Actions.

Instance Feature Control

Instance feature control for actions enables or disables the action depending on preconditions within the business object. For example, you might want to offer the action `accept_travel` only if the status is not *rejected* already.

Dynamic instance feature control for actions is defined with the syntax `features: instance`.

For more information, see Feature Control.

Global Feature Control

Global feature control for actions enables or disables the action depending on BO-external preconditions. For example, you might want to offer an action only during a specific time of the year.

Dynamic global feature control for actions is defined with the syntax `features: global`.

For more information, see Global Feature Control.

Authorization Control

Actions can be checked against unauthorized execution. If no authorization control is specified for the action, the authorization mode of the authorization master (`global` or `instance` or both) is used for the action.

You can replace the authorization control that is specified in the authorization master for every action, by defining the action with global or instance authorization or both: `( authorization: global| authorization: instance| authorization: global, authoarization: instance )`.

To exclude the action from authorization checks, you can use the syntax `authorization: none`.

You can also delegate the authorization for actions to the authorization for `update`.

For more information, see Authorization Definition.

Non-Locking Action

By default, instance actions lock the instance on which they are executed. To prevent that the locking mechanism is triggered when executing an action, you can define an instance action as non-locking with the syntax `lock: none`.

Copy actions are a use case for this. You might want to copy the data of an existing instance without locking the template instance. Since static actions are not related to a specific instance, they are non-locking by definition and this syntax element is not applicable.

> **Note**
>
> The locking mechanism is only prevented for the action itself. Possible modify calls in the action implementation are not affected by the non-locking specification in the action definition. Consequently, an instance is locked if it is modified by an action even if the action is defined as non-locking.

Actions and Functions in Projection Behavior Definitions

You can add new actions and functions on projection level, for example if an action is only relevant for one specific use case. For the definition, the same rules apply as for actions and functions defined on the base RAP business object. However, if an authorization concept is required, you must define it for each BO node and the projection BDEF must be defined as `strict`. For more information. see Authorization Definition. Actions and Functions added to the projection BDEF are automatically exposed with the OData service and don't need to be exposed separately.

Actions and Functions defined in base behavior definitions must be included in the projection behavior definition if you want to expose it for an OData service. The following syntax is used:

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

```
  use function FunctionName [result entity ProjResultEntity] [as FunctionAlias] [external ExtFuncNa
}
```

For more information about actions and functions in projection BDEFs, see CDS BDL - entity behavior definition (ABAP Keyword Documentation).

Related Information

Actions
Action Implementation
Action Runtime

Action Implementation

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

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

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

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

Implementing an Action in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.

PRIVATE SECTION.

  METHODS method_name FOR MODIFY
    IMPORTING keys FOR ACTION AliasedEntityName~ActionName
   [REQUEST requested_fields]
   [RESULT result].

ENDCLASS.

CLASS lhc_handler IMPLEMENTATION.
  METHOD method_name.

  // Implement method here!
  ENDMETHOD.
ENDCLASS.
```

Importing Parameter

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

Action Specifics	Importing Parameter Components
instance action	An instance action imports the key of the instance on which the action is executed.
	An instance action imports `%cid_ref`. This component can be filled by the action consumer if the action is executed on a newly created instance that does not have a final key yet. `%cid_ref` is then filled with the `%cid` of the instance the action is assigned to.
static action	A static action imports `%cid`. For static actions, the `%cid` works as an operation ID that identifies the operation uniquely.
action with parameter	An action with parameter imports the parameter structure `%param` for parameter input.
action with result type entity	If the action returns one or more entity instances, the action imports `%cid` to identify the new instance before the final key is set.
factory action	A factory action imports `%cid` and `%cid_ref`. As factory actions always create new instances, `%cid` is filled to identify the new instance(s) before the final key is set. If factory actions are instance-bound they also import `%cid_ref` to identify a possibly newly created instance to which they are assigned.

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

Result Parameter

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

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

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

UI Consumption of Actions

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

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

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

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

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

Example

For a fully implemented action, see Implementing Actions and Enabling Actions for UI Consumption.

Related Information

Action Runtime
Actions
Action Definition

Functions

A function in RAP is a custom read-operation that is part of the business logic.

Functions perform calculations or reads on business objects without causing any side effects. Functions don't issue any locks on database tables and you can't modify or persist any data computed in a function implementation.

⚠ Caution

You can't define authorization control or feature control for functions.

Example

If you have a business object that manages batch jobs and you want to administer the jobs simultaneously, you can use functions.

Functions and Queries

Similar to a query, a function offers a possibility for structured data retrieval and data filtering without modifying any data. A function is preferable to a query, if you don't want the consumer to operate on the entire result-set, but a prefiltered result-set instead. If the filter criteria are predefined and don't depend on the consumer, you can use functions to push the data filtering and structuring to the back end and only expose the final result set to the consumer.

While a query only allows one implementation, you can implement several instance-bound or entity-bound functions operating on the same result set, but with different filter conditions. You can develop more complex filtering semantics because you can use a combination of customized parameters reflecting your individual use case in addition to the query capabilities described in Query Capabilities.

Furthermore, you can create and expose transition messages for the consumer to offer more transparency if a request fails. For more information about messages, refer to Messages. Note that functions can only return transition messages and no state messages.

Triggers for Functions

For a function to be executed, a corresponding trigger is required. Functions can be called by:

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

Function Definition

Function Syntax

Functions are specified as nonstandard operations in behavior definitions and are implemented in the ABAP behavior pool.

For details about the syntax, see CDS BDL - function (ABAP Keyword Documentation).

Input Parameter

For details about input parameters for functions, see CDS -BDL - input parameter (ABAP Keyword Documentation).

For details about modeling of parameters in RAP, see Modeling Parameters for Non-Standard Operations.

For OData services, you can default the input parameters of functions by using default values functions. A default values functions serves as an input assistant for input parameters on a Fiori UI by calculating their default values and by providing them as its result for the respective function. The calcuated values are then display in a popup window for the input parameters which opens when the function is triggered by the end user. For conceptual information on defaulting input parameters of operations, see Operation Defaulting.

Output Parameters

For details, about output parameters, see CDS BDL - output parameter (ABAP Keyword Documentation).

For details about modeling of parameters in RAP, see Modeling Parameters for Non-Standard Operations.

The output parameter for actions and functions is defined with the keyword `result`. The result parameter is optional. However, if a result parameter is declared in the action definition, it must be filled in the implementation. If it isn't filled, the action doesn't return anything, even if the action is declared with result cardinality greater 0. In such a case, the OData service returns initial values.

By declaring the action result to as `selective` you can define that the action consumer can decide whether the result shall be returned completely or only parts of it, for example the keys only. This can help to improve performance as performance consuming calculated fields can be excluded from the result. For more information about the implementation with selective result parameter, see Action Importing Parameter. A SAP Fiori UI requests only the keys of the result when an action with selective result is executed.

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

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

> **Note**
>
> End-to-end support for RAP scenarios is only given with result entity cardinality `[0..1]`, or `[1]`.

You can return either a result entity or a result structure:

- **Result Entity:** By declaring the action with `result entity MyBOEntiy`, the action returns one ore more instances of `MyBOEntity`. `result $self` returns one or more instances of the entity the action is assigned to.

  When you return a result entity with `$self` in a UI service, the current UI behavior is that it stays on the same page where they action is executed. For factory actions, a Fiori Elements UI currently navigates to the object page of the newly created instance.

> **Note**
>
> Only actions and functions having output entities that are included in the service definition are exposed in the service.

In a projection behavior definition, result entities other than `$self` must be redefined with the projection result entity. For more information, see Actions in Projection Behavior Definitions.

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

Function Types

> **Caution**
>
> You can't define authorization control or feature control for functions.

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

Internal Function

By default, functions are executable by OData requests as well as by EML from another business object or from the same business object. To only provide a function for the same BO, the option `internal` can be set before the function names. An internal function can only be accessed from the business logic inside the business object implementation such as from a determination or from another function.

Static Function

By default, functions are related to instances of a business object's entity. The option `static` allows you to define a static function that isn't bound to any instance but relates to the complete entity.

Repeatable Function

The addition `repeatable` allows you to define an instance function that can be executed multiply on the same instance in the same EML request or OData changeset. The different executions of such a function can be distinguished using the content ID `%CID`. The content ID is contained in the derived type of repeatable functions and needs to be provided in every call of a repeatable function.

Actions and Functions in Projection Behavior Definitions

You can add new actions and functions on projection level, for example if an action is only relevant for one specific use case. For the definition, the same rules apply as for actions and functions defined on the base RAP business object. However, if an authorization concept is required, you must define it for each BO node and the projection BDEF must be defined as `strict`. For more information. see Authorization Definition. Actions and Functions added to the projection BDEF are automatically exposed with the OData service and don't need to be exposed separately.

Actions and Functions defined in base behavior definitions must be included in the projection behavior definition if you want to expose it for an OData service. The following syntax is used:

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

```
  use function FunctionName [result entity ProjResultEntity] [as FunctionAlias] [external ExtFuncNa
}
```

For more information about actions and functions in projection BDEFs, see CDS BDL - entity behavior definition (ABAP Keyword Documentation).

State Messages in Read Operations

Read operations, including functions, don't cause any changes to the business object state, but simply return data. Consequently, a read or function implementation can't issue new state messages.

In OData V4, a function with result type entity returns the respective entity including the respective state messages. In OData V2, the entity is returned without state messages as default behavior.

For more information about how messages behave in EML, refer to Message Behavior in EML (Entity Manipulation Language).

Function Implementation

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

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

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

Implementing a function in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.

PRIVATE SECTION.

  METHODS function_name FOR READ
    IMPORTING keys FOR FUNCTION AliasedEntityName~ActionName
   [REQUEST requested_fields]
   [RESULT result].

ENDCLASS.

CLASS lhc_handler IMPLEMENTATION.
  METHOD method_name.

  // Implement function here!
  ENDMETHOD.
ENDCLASS.
```

Importing Parameter

- Depending on the type of function, the importing parameter `keys` consists of the following components:

Action Specifics	Importing Parameter Components
instance function	An instance function imports the key of the instance on which the function is executed.
static function	A static function imports `%cid`. For static functions, the `%cid` works as an operation ID that identifies the operation uniquely.
function with parameter	A function with parameter imports the parameter structure `%param` for parameter input.

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

Result Parameter

The components of the result parameter depend on those of the importing structure. The imported values of `%cid` in case of a static function are returned if they are imported.

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

For functions with selective result, only the fields that are requested in `REQUEST` must be filled in `%param`.

Operation Precheck

With a precheck implementation you can deny incoming requests before data reaches the transactional buffer.

You can prevent illegal changes from reaching the transactional buffer by prechecking modify operations.

Use cases are

- Authorization checks against incoming values, see Authorization Control. For an implementation example, refer to Implementing Prechecks.
- Uniqueness checks in managed BO scenarios with an unmanaged lock implementation, or unmanaged BO scenarios, see Uniqueness Check for Primary Keys.

For details about the syntax for the **precheck** that can be defined for every modify operation in the behavior definition or in the projection behavior definition, see CDS BDL - standard operations (ABAP Keyword Documentation)

The implementation for the condition deciding whether the modify operation is executed for a certain instance must be implemented in the corresponding method in the behavior pool. If the precheck is used for an operation in the projection behavior definition, the method must be implemented in the behavior pool for the projection.

```
METHOD MethodName FOR PRECHECK
  IMPORTING Keys1 FOR CREATE...
            Keys2 FOR ACTION...
```

The precheck method is called during runtime before the assigned modify operation and removes all input from the modifying request for which the condition in the precheck is not fulfilled.

Depending on the use case, you can define a precheck for the operation in the BO-layer or in the projection layer. Also, it is possible to define for both layers.

Prechecks in UI Scenario

If a precheck fails and messages are returned in the `REPORTED` parameter, the messages are displayed on the UI. The end user has to resolve the issue by changing the entries, so that the precheck does not return any failed keys and messages.

Related Information

Standard Operations
Actions
Save Sequence Runtime

Operation Augmentation

With an augmentation implementation you can add data or modify incoming requests on the projection layer before data reaches the transactional buffer.

You can add data to a modify request or issue additional requests by augmenting the operation before the request is passed to the base business object.

Use cases for augmentation are

- Defaulting for incoming requests.

  The incoming original request for `CREATE` is augmented (modified) so that the request with prefilled values is passed to the base BO for processing. In this case, the augmented request is the same as the original request, just with different values.

- Behavior-enabling denormalized fields, for example enabling editing of language dependent fields, see Editing Language-Dependent Fields.

  The incoming original request for `CREATE` is augmented and a second request for `CREATE_BY_ASSOCIATION` is issued. In this case the original request is augmented with a second one.

Definition

Augmentation is defined in the projection behavior definition on the relevant operation with the following syntax:

```
define behavior for Entity [alias AliasedName]
  ..
{
  use create (augment);
  use update (augment);

  use association AssocName { create (augment); }
  ...
}
```

Implementation

The implementation of the augmentation is done in an ABAP handler class of the projection behavior pool for the augmented operation.

Method Declaration

```
METHODS augment_operation FOR MODIFY
  IMPORTING entities_create FOR CREATE ProjectionEntityAlias
  IMPORTING entities_update FOR UPDATE ProjectionEntityAlias
  IMPORTING entities_cba    FOR CREATE ProjectionEntityAlias\_Association
```

If you augment more than one operation, implement the logic in one method in the behavior implementation, to be able to work with `%cid_ref` if you need to reference another instance.

Example

One logical unit of work contains a create and an update request on the newly created instance. You need to be able to reference the newly created instance in the augmentation implementation. That's where the `%cid` of the newly created instance must be referenced in `%cid_ref` in the augmentation implementation for the update operation. If you handle `CREATE` and `UPDATE` in separate methods, you cannot work with `%cid` and `%cid_ref`.

A special form of the EML statement `MODIFY` is used to manipulate the request for the base BO in the handler implementation.

```
METHOD augment_operation.

  MODIFY AUGMENTING ENTITIES OF BaseBusinessObject
    ENTITY EntityAlias
     CREATE|UPDATE|EXECUTE MyAction FIELDS ( Field1 )
     WITH VALUE #( ( Field1 = 'DefaultValue' ) )
     RELATING TO OriginalInstance BY RelatedInstance.

ENDMETHOD.
```

Note

This form of EML can only be used by a BO-provider.

With this EML statement, you can modify entities of the base business object. All modify operations are allowed (including actions).

The statement variant has no `FAILED` or `REPORTED` addition. This is because of its special semantics. The operations in `modify augmenting` are not executed immediately, but are only merged with the operation of the original request. Thus, the processing of the augmented request by the base BO handler only begins after the termination of the augment method. Failures during the processing of the underlying base BO handlers cannot be reported back to the EML statement in the augmentation implementation. To trace possible errors during processing of the augmented request, the `relating` table is used to map messages to incoming requests. See RELATING Mechanism for New Instances.

However, the augment method itself has the changing parameters `FAILED` and `REPORTED`, which can be filled if errors in its input are detected. These responses are included in the overall response of the projection request. Furthermore, failed instances returned by the augment method are removed from the request before the remainder of it is passed to the base BO.

A `modify augmenting` statement can request original base instances, instances that are requested in the original request, or, it can modify other base instances that were not requested by the original request. The recognition of original instances is done either by the imported instance key, or by using `%CID_REF`.

The following regulations apply when an instance of the original request is augmented:

- To augment an original `UPDATE`, issue an augmenting operation `UPDATE` on the same instance.

  Set the values and `%CONTROL` flags for those fields only which are added by the augmentation.

- To augment an original `CREATE` (for example. for setting default values), issue an augmenting operation `CREATE` on the same instance.

  Do not issue an augmenting `UPDATE` (using the `%CID_REF`), as the base BO would then see the un-augmented `CREATE` operation, which it may refuse (for example because of a missing mandatory field) or handle in an unwanted manner.

- To augment an original `CREATE_BY_ASSOCIATION`, issue an augmenting operation `CREATE_BY_ASSOCIATION` on the same instance. The target set of instances may contain original target instances (with additional fields being set), as well as new instances. The same regulations apply as for `CREATE`

- It is not possible to set a field in augment which was already set for the original instance. The RAP runtime discards such augment fields. For example, values set in the original request cannot be changed. Only fields which are unset in the original request can be added.

`RELATING` Mechanism for New Instances

It can happen that the base business object returns entries in the `FAILED` and `REPORTED` tables for augmented requests. To relate these responses to the original request, if the augmented request contains new instances, the `MODIFY AUGMENTING` statement offers an addition which allows to relate augmented instances to original instances. The runtime uses this information to transform `FAILED` keys of new instances back to the keys of the related original instances.

If an augmented instance fails, the related original instance is included in the `FAILED` response of the overall request. If the failure occurs in locking, the related original instance is not passed to the base business object handler classes.

Content Identifiers for New Instances

Create requests on the projection layer that are augmented by a create or a create-by-association operation only contain the content IDs of the original create request in the `%cid` component of their `MAPPED` response. Content IDs that result from augmenting create and create-by-association operations are not included in the `MAPPED` response of the original create request.

Operation Defaulting

With a default values function you can default the input parameters for actions and functions as well as fields for create and create by-association operations in OData services. This way, default values functions serve as an input assistant for end users that trigger the respective operation on the UI.

Requirements for consuming OData Apps

If you use a Fiori Elements UI, default values functions work out of the box. They are called automatically for the respective operations and the resulting values are displayed in the parameter input window on the UI.

In non-Fiori Elements apps, however, you need to implement this orchestration yourself. In this case, the framework just provides the function as part of the OData metadata which needs to be handled by the consuming OData app respectively.

Defining Default Values Functions

Base Behavior Definition

Default values functions are defined in the base behavior definition by adding the keywords `{ default function GetDefaultsForOperation; }` behind the respective operation and parameter definition. Note that the name of the default values function needs to start with `GetDefaultsFor`. An external name for the OData exposure of the default values function can be defined. The following code block shows exemplary definitions of default values functions for existing operations:

```
// Create operations (name can be omitted):
create { default function; }

// Create-by-association-operations:
association _Item { with draft; create { default function GetDefaultsForCBA external 'GetDefaultsFo

//Actions:
action PlainAction parameter myPlainParameter { default function GetDefaultsForPlainAct; }

//Functions:
function DeepFunction deep parameter myDeepParameter result [0..*] MyEntity { default function GetD
```

For default values functions, the type of the function, instance or static, is derived from the operation to be defaulted respectively by the framework. A default values function that is defined for an instance action or function, for example, is implicitly defined as instance function as well. Hence, the function type cannot be defined manually in the behavior definition.

The importing parameter type of default values functions is derived by the framework as well. In case of an instance default values function, the importing parameter is typed with a table containing the key fields of the related entity. In case of a static default values function, the importing parameter is typed with a table containing the content identifier `%cid` indicating the respective operation.

The output parameter type of default values functions is derived according to the related operation kind: In case of a related create (by association) operation, the output parameter is typed with a table for mapping the input identifiers (key fields or `%cid`) to the fields of the related (target) entity. In case of related actions and functions, the output parameter is typed with a table for mapping the input identifiers to the input parameter fields of the related operation. This way, the provider of the default values function can assign default values for the respective (parameter) fields per instance in the result. Both flat and deep parameters can be defaulted.

Default values functions always provide one set of default parameter values per imported input identifier in the result table, hence their cardinality is always `[1]`.

Projection Behavior Definition

Default values functions need to be exposed on the BO projection layer analogously to non-defaulting functions using the keywords `use function`. Default values functions are not automatically exposed on the projection layer when the related operation is projected.

```
projection;
strict(2);

define behavior for ...
...
{
  ...
  use function GetDefaultsForCreate;
  use function GetDefaultsForCBA;
  use function GetDefaultsForPlainAct;
  use function GetDefaultsForDeepFct;
}
```

For operations defined on the projection layer, default values functions can be defined analogously to the base layer.

Implementing Default Values Functions

A default values function is implemented by assigning default values to the (parameter) fields for all imported input identifiers. The following code block shows the basic implementation structure for default values functions:

```
METHOD GetDefaultsForRelatedOperation.

* Calculate the default values for all imported input identifiers, e.g. by EML Read for imported ke
    READ ENTITIES OF ...
    RESULT DATA(read_results).

* Assign these default values to the respective (parameter) fields per input identifier in the resu
    LOOP AT read_results INTO DATA(read_result).

* Example for a related instance action/function/create-by-association:
          APPEND VALUE #( %tky                           = read_result-%tky
                          %param-field_to_be_defaulted   = [default_value] ) TO result.

* Example for a related create operation:
          APPEND VALUE #( %cid                           = read_result-%cid
                          %param-field_to_be_defaulted   = [default_value] ) TO result.

    ENDLOOP.

ENDMETHOD.
```

Errors can be written to the `failed` and `reported` changing parameters analogously to non-defaulting functions.

For an example implementation of a default values function, see Defaulting Input Parameters for Operations.

Determinations

A determination is an optional part of the business object behavior that modifies instances of business objects based on trigger conditions.

A determination is implicitly invoked by the business object's framework if the trigger condition of the determination is fulfilled. Trigger conditions can be `modify` operations and modified fields. The trigger condition is evaluated at the trigger time, a predefined point during the BO runtime. 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.

For detailed information on messages, see Messages.

Example

A determination is implemented to calculate the invoice amount based on a changed price or quantity of an item. As soon as the consumer creates a new item entity instance or updates the quantity or price of an existing one, the determination is executed and recalculates the invoice amount.

Note

When working with determinations, you have to consider the following runtime specifics:

- In unmanaged scenarios, determinations are only supported for draft instances, not for active instances.
- The determination result must not change if the determination is executed several times under the same conditions (idempotence).
- 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.
- Once a determination has been triggered, it must run independently from other determinations.
- If you create or update an instance and delete it with the same request, it can happen that an EML read operation in a determination on modify fails as instances with the given key cannot be found.

> **Note**
> Side effects can be used to trigger a call to the backend after a determination has been executed. This makes sense if the determined data are not automatically reread by the operation the determination belongs to. Side effects must be defined and annotated in the OData document. For more information, see Side effects.

Determination Definition

Syntax for Defining Determinations

For details about the syntax for defining determinations, see CDS BDL - determinations (ABAP Keyword Documentation)

Assigned Entity

A determination belongs to an entity stated in the behavior definition. The fields that are used for the trigger conditions must belong to the same entity the determination is assigned to. The determined fields and the determining fields may belong to the same entity or to other entities of the business object.

Reference to Implementation Class

Determinations are implemented in the behavior pool, which is referred in the behavior definition by the keyword `implementation in class ABAP_CLASS_NAME [unique]`.

Trigger Time

The trigger time defines at what time the trigger condition of a determination is evaluated. The following options are available:

- on modify: The determination is triggered during a `modify` operation.

  > **Note**
  > A determination on modify can also be triggered by the draft action `activate` as this action invokes a `modify` call including an `update` operation in case of an already existing active instance or a `create` operation in case of a new active instance. For more information, see Activate Action.

- on save:
  - The determination is triggered in the `finalize` phase of the save sequence.

- The determination can be called on request by executing a determine action, if the determination has been assigned to such an action in the behavior definition. For more information, see Determine Actions.

For more information on the different activities performed during the BO runtime, see Operations.

Trigger Condition

Determinations can be triggered by trigger operations or by trigger fields or by both.

Note

A determination can trigger itself, for example if `update` is defined as a trigger operation. To avoid an infinite loop of executions, the trigger conditions can be defined on field level. If the determination implementation changes other fields than the ones chosen as trigger conditions, the determination will not be triggered repeatedly. In case that a determination does trigger itself, for example because an `update` trigger is defined, the determination must stick to the rule of idempotence to avoid an infinite loop of executions, see Rules for determinations.

Trigger Operations

Determinations can be triggered by the operations `create`, `update` and `delete`. When one of these operations is executed for a draft instance or for an active instance, determinations with the respective trigger operations are triggered.

Note

The trigger operation `update` for determinations on save is only supported in combination with the trigger operation `create`.

Trigger Aggregations

Determinations on save are triggered according to the relation between the operations performed on the current transactional buffer and the state of the database before the transaction. This applies to draft and to non-draft scenarios.

In draft scenarios, the transactional buffer is represented by the draft instance. In order to determine which operation triggers a determination on save, all operations performed on the draft instance are aggregated across its whole lifetime. Then, when a determination on save is called, these operations are evaluated relatively to the state of the **active** database. This does not apply to determinations on modify.

Example

A new draft instance is created. Then, it is updated. After that, a determine action containing a determination on save is called. The trigger for the determination is the operation `create` and not the operation `update`, because from the active database's point of view, this instance is new and has hence been created, not updated.

If a delete operation is involved as a subsequent operation, the delete operation is the one which triggers the determination on save. This enables the business object consumer to revert changes that might have been performed before the delete operation by implementing a determination which triggers on that delete.

Example

A new draft instance is created as the child of an existing draft root instance. While entering data for this child instance, a determine action containing a determination on save is called that modifies the root instance. After that, the child instance is deleted. A determination which is triggered by the `delete` operation can now ensure that the changes done to the root instance are reverted.

The table below shows, which operation triggers a determination on save in case of different operation aggregations. The trigger operation which is respectively effective must be stated in the behavior definition for the determination so that it is executed. The following rules apply to the standalone execution of determinations on save as well as to their execution via determine actions.

Operation aggregation	Effective trigger operation
Create + Update	Create
Create + Delete	Delete
Update + Update	Update
Update + Delete	Delete
Delete + Create	Create

Trigger Fields

Determinations can be triggered by fields belonging to the assigned entity. When one or more fields are changed by a `create` or by an `update` operation, the determination is executed.

Input Parameter

A determination imports the keys of the instances on which the determination is executed. The name of the input parameter must be declared in the signature of the corresponding method, see Determination Implementation.

> **Note**
> If imported keys are not available anymore during the runtime of a triggered determination on modify, this determination is ignored by the framework.

Output Parameter

Messages can be returned to the consumer by writing them into the implicitly declared `REPORTED` structure.

Determination Implementation

The implementation of a determination is contained in a local handler class as part of the behavior pool. As depicted in the listing below, 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 importing parameter. The type of the importing parameter is an internal table containing the keys of the instances the determination will be executed on. Lastly the signature contains the affected entity followed by the name of the determination stated in the behavior definition.

It is possible to implement multiple determinations for multiple entities in a single method, if these determinations use the same trigger time.

Listing 1: Implementing a Determination in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM CL_ABAP_BEHAVIOR_HANDLER.

PRIVATE SECTION.
```

```
  METHODS method_name1 FOR DETERMINE ON MODIFY
    IMPORTING keys FOR AliasedEntityName~DetOnModify.

  METHODS method_name2 FOR DETERMINE ON SAVE
        IMPORTING keys FOR AliasedEntityName~DetOnSave
                          keys2 FOR AliasedEntityName2~DetOnSave2.


ENDCLASS.

CLASS lhc_handler IMPLEMENTATION.
  METHOD method_name1.

  // Implement method for determination here!
  ENDMETHOD.

...


ENDCLASS.
```

Validations

A validation is an optional part of the business object behavior that checks the consistency of business object instances based on trigger conditions.

A validation is implicitly invoked by the business object's framework if the trigger condition of the validation is fulfilled. Trigger conditions can be `modify` operations 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.

For detailed information on messages, see Messages.

Example

A validation is implemented to check if the customer ID contained in travel instances is valid. This validation is assigned to the entity `travel` and contains the trigger field `customer_ID`. As soon as the field for the customer ID is updated by the consumer, the validation checks whether the customer ID is valid or not. If the customer ID is not valid, the validation prevents the instance data from being saved in the save sequence and returns a warning message.

Note

When working with validations, you have to consider the following runtime specifics:

- In unmanaged scenarios, validations are only supported for draft instances, not for active instances.
- 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.
- It is not allowed to use EML modify statements in validations.
- In a managed scenario with draft, it is strongly recommended to assign validations to the `PREPARE`, so that state messages from the validations are returned correctly to the consumer. For more information, see State Messages.

Validation Definition

For details about the syntax for defining validations, see CDS BDL - validations (ABAP Keyword Documentation)

Assigned Entity

A validation belongs to an entity stated in the behavior definition. The fields that are used for the trigger conditions must belong to the same entity the validation is assigned to. The validated fields may belong to the same entity or to other entities of the business object.

Reference to Implementation Class

Validations are implemented in the behavior pool, which is referred in the behavior definition by the keyword `implementation in class ABAP_CLASS_NAME [unique]`.

Trigger Time

The trigger time defines at what time the trigger condition of a validation is evaluated. For validations, only the trigger time `on save` can be stated. Validations on save are executed:

- In the `checkBeforeSave` method of the save sequence
- On request by executing a determine action, if the validation has been assigned to such an action in the behavior definition. For more information, see Determine Actions.

For more information on the different activities performed during the BO runtime, see Operations.

Trigger Condition

Validations can be triggered by trigger operations or by trigger fields or by both.

Trigger Operations

Validations can be triggered by the operations `create`, `update` and `delete`. When one of these operations is executed for a draft instance or for an active instance, validations with the respective trigger operations are triggered.

Note

The trigger operation `update` for validations is only supported in combination with the trigger operation `create`.

Trigger Aggregations

Validations are triggered according to the relation between the operations performed on the current transactional buffer and the state of the database before the transaction. This applies to draft and to non-draft scenarios.

In draft scenarios, the transactional buffer is represented by the draft instance. In order to determine which operation triggers a validation, all operations performed on the draft instance are aggregated across its whole lifetime. Then, when a validation is called, these operations are evaluated relatively to the state of the **active** database.

Example

A new draft instance is created. Then, it is updated. After that, a determine action containing a validation is called. The trigger for the validation is the operation `create` and not the operation `update`, because from the active database's point of view, this instance is new and has hence been created, not updated.

If a delete operation is involved as a subsequent operation, the delete operation is the one which triggers the validation. This enables the business object consumer to check data determined by a determination that might have been executed before the delete operation.

Example

A new draft instance is created as the child of an existing draft root instance. While entering data for this child instance, a determine action containing a determination on save is called that modifies the root instance. After that, the child instance is deleted. A validation which is triggered by the delete operation can now validate the changes done to the root instance.

The table below shows, which operation triggers a validation in case of different operation aggregations. The trigger operation which is respectively effective must be stated in the behavior definition for the validation so that it is executed. The following rules apply to the standalone execution of validations as well as to their execution via determine actions.

Operation aggregation	Effective trigger operation
Create + Update	Create
Create + Delete	Delete
Update + Update	Update
Update + Delete	Delete
Delete + Create	Create

Trigger Fields

Validations can be triggered by fields belonging to the assigned entity. When one or more fields are changed by a `create` or by an `update` operation, the validation is executed.

Input Parameter

A validation imports the keys of the instances on which the validation is executed. The name of the input parameter must be declared in the signature of the corresponding method, see Validation Implementation.

Output Parameter

An invoked validation can reject inconsistent instance data from being saved by writing the keys of failed instances into the implicitly declared `FAILED` structure.

Additionally, messages can be returned to the consumer by writing them into the implicitly declared `REPORTED` structure.

Validation Implementation

The implementation of a validation is contained in a local handler class as part of the behavior pool. As depicted in the listing below, 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` 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. Lastly the signature contains the affected entity followed by the name of the validation stated in the behavior definition.

It is possible to implement multiple validations for multiple entities in a single method.

Listing 1: Implementing a Validation in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM CL_ABAP_BEHAVIOR_HANDLER.

PRIVATE SECTION.

  METHODS method_name1 FOR VALIDATE ON SAVE
    IMPORTING keys FOR AliasedEntityName~MyValidation.

  METHODS method_name2 FOR VALIDATE ON SAVE
        IMPORTING keys FOR AliasedEntityName~AnotherValidation
                          keys2 FOR AliasedEntityName2~AnotherValidation2.


ENDCLASS.

CLASS lhc_handler IMPLEMENTATION.
  METHOD method_name1.

  // Implement method for validation here!
  ENDMETHOD.

...


ENDCLASS.
```

Determine Actions

Determine Action

Determine actions allow the business object consumer to call determinations and validations on request. You can assign determinations on save and validations to a determine action and execute it like any other action. Whenever a determine action is called, the determinations and validations assigned to it are evaluated and then only those determinations and validations are executed whose trigger conditions are fulfilled.

Determine actions are primarily meant to be called by side effects in order to give the user immediate feedback after changing UI fields or field groups in draft-enabled applications. Combined with side effects, determine actions act as an early execution of parts of the save sequence that already runs determinations and validations before the draft instance is prepared and activated. Side effects must be defined and annotated in the OData document. RAP offers a way to define side effects in the behavior definition, which affects the OData annotations based on your application backend implementation. For more information, see Side Effects.

You cannot add `on modify` determinations to a determine action. Furthermore, feature and authorization control are not enabled for determine actions.

You cannot execute determine actions inside implementations of determinations and validations.

In unmanaged scenarios, determine actions must be implemented manually for active instances.

ℹ Note

For performance reasons, determinations and validations are not executed during a determine action, if these determinations and validations have already been executed during a previous determine action that has been executed on the same instance (draft or active) in the same transaction. This performance optimization is not applied in the following cases:

- A modification performed after the first determine action triggers a determination/validation.
- A validation in the first determine action reports a failed key.

- A determination/validation has been assigned using the addition `always`.

  In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

Syntax for Defining Determine Actions

Determine actions are specified in the behavior definition with the `DETERMINE ACTION DetActionName`.

For more information about the syntax, see CDS BDL - determine actions (ABAP Keyword Documentation).

Only determinations and validations that are defined and implemented in the BO can be assigned to a determine action.

You can include validations and determinations for child entities, if these validations or determinations do not include the trigger operation `delete`.

The `always` flag

When a determine action is called that contains a determination or validation with the flag `always`, this determination or validation is executed regardless of its trigger conditions. After a determination with the flag `always` has been executed, it can be triggered again by other determinations belonging to the same determine action.

Execution order

After calling a determine action, assigned determinations are executed before assigned validations. The execution order among determinations or validations themselves is defined by the framework and is independent of the specified order within the determine action.

Messages and failed keys

Determinations and validations assigned to determine actions can return messages to the `REPORTED` structure, but the failed keys of assigned validations are discarded.

The `draft determine action prepare` is a determine action that is implicitly available for all draft BOs. It is automatically called before a draft instance is activated and cannot be called for active instances. If a validation is assigned to the `draft determine action prepare` and detects failed keys, the subsequent `activate` action is not executed anymore. The `FAILED` structure is not filled in this case either.

For more information, see Draft Actions.

Feature Control

This topic is about the concept of feature control for the ABAP RESTful application development.

About Feature Control

With feature control, you can provide information to the service on how data has to be displayed for consumption in a SAP Fiori UI, for example if fields are mandatory or read-only.

You can implement feature control in a static or dynamic way. In a static case, you define which operations are available for each business object entity or which fields have specific access restrictions like being mandatory or ready-only. In a dynamic case, the access restrictions for fields or the enabling or disabling of methods depends on the state of the business object, for example on the value of a specific field.

Note

Note that you can only use instance feature control or global feature control on an operation or action. It's not possible to combine both.

For more information, see Instance Feature Control.

Consumer Hints in UI Scenarios

In UI scenarios, consumer hints, which are retrieved from the backend, provide the information for the UI which operations are enabled for the current situation. On a Fiori Elements UI, action and other operation buttons are only clickable if authorization and feature control methods provide a positive result under the current conditions (the state of the retrieved BO instances).

The definition and implementation of authorization and feature control is important, as the consumer hints are based on the result of the authorization and feature control. The implementation methods for authorization and feature control are called on displaying the list report or the object page for a specific instance. By checking the authorization for all the exposed actions beforehand, users can't even start operations, for which they aren't authorized, as their triggers (action buttons) aren't available.

Example

The action `SetStatusBooked` can only be executed for instances, whose status isn't booked yet (feature control) and by users that are assigned to a specific role (authorization). When selecting instances, whose status is booked already, the button for this action isn't available on the UI and the user can't execute the action.

Related Information

Global Feature Control

Instance Feature Control

You can define feature control on instance level for fields, operations, and actions. With this, you can control, for example, if a method is enabled or disabled when an instance of a business object entity has a certain state. You can define instance-based feature control for UPDATE and DELETE, as well as for actions. Since you need an active instance of a business object in this case, the CREATE operation can't have feature control on instance level.

Static Feature Control

Fields

You can define specific access restrictions for each field. You can implement the restrictions in a static way if the access restriction is always valid for each instance.

Within the behavior definition, you can specify individual fields of an entity that have certain access restrictions.

For details about the syntax, see CDS BDL - field characteristics, projection BDEF (ABAP Keyword Documentation) .

You can use the following field properties to define static field control:

The `field (read only)` property defines that values of the specified fields must not be created or updated by the consumer. You can set this property in the behavior definition.

ℹ Note

The BO runtime rejects external EML `MODIFY` requests that include read-only fields for update or create operations.

Read-only fields that are included in OData call to update or create instances are ignored while possible other fields are processed for update and create operations.

The `field (mandatory)` property defines that the specified fields are mandatory. The specified fields must be filled by the consumer when executing modifying requests. For the relevant fields, the value must be provided in CREATE operations. In update operations, it must not be given the null value.

ℹ Note

The mandatory property must be set in the behavior definition, not in the projection. This property isn't evaluated if you've defined it in a projection.

⚠ Caution

Note that there's no implicit validation by the business object framework. As an application developer, you must ensure that you've implemented a corresponding validation.

Operations

In a typical transactional scenario, you have to specify which operations are provided by the respective entity.

The transactional character of a business object is defined in the behavior definition where all supported transactional operations are specified for each node of the business object's composition tree. Whenever the corresponding root or child entity is going to be created, updated, or deleted, these operations must be declared in the behavior definition. In this way, you specify at the business object entity level whether each instance is enabled for creation, update, or deletion. For more general information about operations, refer to Operations.

For details about the syntax, see CDS BDL - standard operations (ABAP Keyword Documentation).

The following operations are available, if they are declared for an entity in the behavior definition:

- `create`
- `update`
- `delete`
- `internal *operation*`
- `_association { create;}`

Actions

For more information about the syntax, see CDS BDL - action (ABAP Keyword Documentation).

You can define actions and internal actions. If they're defined with static feature control, they're always available for the respective entity. For more general information, see Actions.

Operation	Effects

Operation	Effects
`action ActionName [...]`	If actions are defined in the behavior definition without dynamic feature control, they're always available for the respective entity. For general information about defining and implementing actions, refer to Actions.
`internal action ActionName [...]`	Specific operations of an entity of a business object can be defined using actions. Similar to standard operations, you can define internal actions in the behavior definition by adding the option internal to the operation name. Internal actions can only be accessed from the business logic inside the business object implementation such as from validations, determinations, or from other noninternal actions.

Dynamic Feature Control

Fields

You can define specific access restrictions for each field. You can implement the restrictions in a dynamic way if the restrictions depend on a certain operation or condition. With dynamic feature control, you can add access restrictions based on conditions or specify access restrictions for fields.

For details about the syntax, see CDS BDL - Field Characteristics, Projection BDEF .

You can implement dynamic feature control for fields in the behavior pool that decides upon the status for these fields. The fields are notated with `field (features: instance)` in the behavior definition. In the behavior pool, the following field statuses can be assigned:

- `UNRESTRICTED`– field has no restrictions
- `MANDATORY`– field is mandatory
- `READONLY`– field is read-only
- `ALL`– all restrictions are requested

The `field (mandatory:create)` triggers a check before the data is persisted. If the annotated field is not filled in, the SAVE is rejected.

The `field (readonly:update)` displays a field as read-only when a business object instance is edited and the value can't be changed via the user interface or via an external EML UPDATE request. If a field is only defined as `readonly:update` without `mandatory:create`, the fields can't be edited on the UI.

The combination of the `field (mandatory:create)` and the `field (readonly:update)` combines the effects of both annotations: In this case, a value must be filled in for new instances in CREATE requests and this value can't be changed via the user interface or an EML request during a MODIFY. A typical use case would be the definition of an ID using external numbering. The ID is defined once and isn't to be changed when the instance is modified.

Operations

For dynamic control of operations, the option `(features: instance)` must be added to the operation or association in question. This is also possible for the create-by-association operation. However, an implementation in the referenced class pool ABAP_CLASS is necessary for this. For each relevant operation, you can specify the following values in the implementation of FOR INSTANCE FEATURES:

- `ENABLED` - if the operation is enabled
- `DISABLED`- if the operation is disabled

Actions

You can implement specific operations for an entity of a business object with actions. Just like the CUD-operations, actions can be enabled or disabled using dynamic feature control. If you define actions in the behavior definition without feature control, they're always available for the respective business object entity they belong to.

- `action (features:instance)`

For dynamic control of actions, the option `(features: instance)` must be added to action in question. However, an implementation in the referenced class pool ABAP_CLASS is necessary for this. For each relevant operation, you can specify the following values in the implementation of FOR INSTANCE FEATURES:

- `ENABLED` - if the action is enabled
- `DISABLED` - if the action is disabled

For more information about the BDL syntax, see CDS BDL - feature control (ABAP Keyword Documentation).

For more information about the implementation syntax, see FOR INSTANCE FEATURES, FEATURES (ABAP Keyword Documentation).

Global Feature Control

You can define global feature control for feature control that is independent of the business object state, for example if you want to disable the CREATE depending on whether a certain business scope is active or not.

Global Feature Control

Operations, associations and actions that get Global Feature Control, are provided with the suffix `(features: global)`.

```
Create|Update|Delete|Action (features:global)
```

For global feature control, the option `(features: global)` must be added to the create, update, delete, or action operation in question. However, an implementation in the referenced class pool ABAP_CLASS is necessary for this. For each relevant operation, you can specify the following values in the implementation of FOR GLOBAL FEATURES:

- `ENABLED` - if the operation or action is enabled
- `DISABLED` - if the operation or action is disabled

```
_association {create(features:global);}
```

For global feature control of the create by association, the option `(features: global)` must be added to the create-by-association operation in question. However, an implementation in the referenced class pool ABAP_CLASS is necessary for this. For each relevant CREATE, you can specify in the implementing handler of the class pool the following values in FOR GLOBAL FEATURES:

- `ENABLED` - if the CREATE is enabled
- `DISABLED` - if the CREATE operation is disabled

For more information about the BDL syntax, see CDS BDL - feature control (ABAP Keyword Documentation).

For more information about the implementation syntax, see FOR GLOBAL FEATURES (ABAP Keyword Documentation).

Side Effects

Side effects are used to reload data, permissions, or messages or trigger determine actions based on data changes in UI scenarios with draft-enabled BOs.

About Side Effects

Side effects are useful in UI scenarios based on draft-enabled BOs to make a Fiori Elements UI aware that data changes of defined fields require the recalculation of other data values, permissions or messages.

Since draft-enabled scenarios use a stateless communication pattern, the UI doesn't trigger a reload of all BO-related properties for every user input. If only certain fields are changed by the end user in edit mode, that is on the draft BO instance, the user can't expect data consistency of displayed data on the UI at all times. In other words, when a UI user changes data of a draft instance, there is not necessarily a READ request for all fields that are displayed. Without side effects, this may lead to inconsistencies of displayed data on the draft instance, for example when data is calculated based on other fields.

Side effects solve this problem. They define the interdependency between fields and other BO characteristics to trigger a reload of affected properties. In particular, side effects are efficient since there is no full reload of all BO-properties, but only of those that are affected by particular user input.

Example

The total price in a travel BO is calculated based on different prices, such as booking fee, flight prices and supplement prices. If the UI user changes the booking fee, the total price needs to be recalculated. If the user doesn't directly save the draft BO instance, the modeled determination, with which the total amount is usually recalculated, is not triggered and the data on the UI in the total amount field doesn't change directly. That means, the user still sees the old total amount, which is an inconsistency and can be avoided by side effects. With a side effect, the application developer defines the interdependency between the total amount and the other amount fields, which triggers the reload to get the recalculated amount, whenever the UI user changes one of the amount fields.

For a detailed description of such a side effect implementation, see Developing Side Effects.

Side effects are defined in the RAP behavior definition. The definition provokes additional annotations in the OData metadata of the RAP service. Based on these annotations, the UI triggers the defined determine actions, reloads and sends READ requests for the respective properties. Like other behavior that is defined in the behavior definition, side effects must be reused for consumption in every projection behavior definition and interfaces. The syntax for resusing side effects is `use side effects`.

You can also add UI annotations via a UI project using the Fiori Tools, but with RAP side effects, as an application developer, you can influence the UI behavior from the backend implementation.

Types of Side Effects

Side effects can be defined in the base behavior definition for the following RAP BO properties:

- **field:** Whenever a defined field is changed on the UI, the side effect is triggered and the defined targets are reloaded.

  ```
  side effects { field MyField affects Targets }
  ```

- **action:** Whenever the action is executed on the UI, the side effect is triggered and the defined targets are reloaded.

  ```
  side effects { action MyAction affects Targets }
  ```

- **determine action:** Whenever the defined source is changed, the determine action is triggered and the defined targets are reloaded.

  ```
  side effects { determine action MyDetermineAction executed on Sources affects Targets }
  ```

- **self:** Whenever the entity the side effect is defined for is modified, the defined targets are reloaded. This type cannot affect (fields of) the entity itself.

  ```
  side effects { $self affects Targets }
  ```

Targets

The following BO properties can be used as side effect targets:

- **`field`:** The specified field is reloaded when the side effect is triggered `field MyField`. You can specify one or more fields as side effect target. The wildcard `field *` reloads all fields of the same entity instance. You can also use fields from other entity instances as targets. They must be defined via an association path `_assoc.Myfield`.
- **`permissions`:** The feature and authorization control of the specified properties are reloaded when the side effect is triggered. You can specify the permissions for
  - fields: `permissions(field MyField)`,
  - actions: `permissions(action MyAction)` and
  - for the standard operations: `permissions (create|update|delete)`.

  You can also specify permissions for elements and operations of associated entities via association path:
  - fields and actions: `permissions ( _assoc.{field|action} )`
  - operations: `permissions ( {create|update|delete} _assoc )`
- **`$self`:** The own entity is reloaded.
- **`entity`:** The specified associated entity is reloaded when the side effect is triggered: `entity _MyAssoc`.
- **`messages`:** All messages stored in `reported` are reloaded when the side effect is triggered.

  > **! Restriction**
  >
  > Messages as side effect target are only available with OData V4.

One type of side effect can affect multiple targets. The specifier determining the type of target must be enumerated as well.

Example

```
side effects { field MyField affects field Field1, field Field2, action Action1, action Acti
```

Sources

The following BO properties can be used as sources for determine action side effects.

- **`field`:** The determine action is triggered whenever the specified field is changed: `field MyField`. You can specify one or more fields as determine action source.

  > **i Note**
  >
  > By using more than one field as source for the determine action side effect, the fields are implicitly considered as field group. The side effect is only triggered if the cursor is set outside of the group of source fields after changing at least one of them.

  You can also use fields from other entity instances as sources. They must be defined via an association path: `_assoc.Myfield`.

- **`$self`:** The determine action is triggered whenever anything on the own entity instance is changed (fields, create child, delete child).
- **`entity`:** The determine action is triggered whenever anything on the specified entity is changed (fields, create, delete): `entity_assoc`.

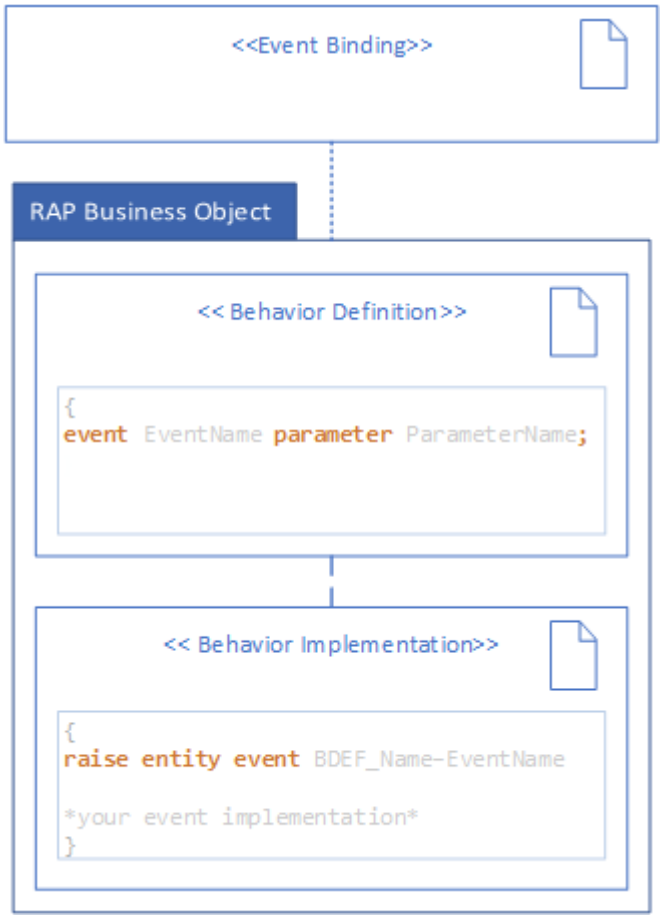
Business Events

About Business Events in RAP

RAP supports event-driven architecture natively. Event-driven architecture enables asynchronous communication between an event provider and an event consumer in use cases where no direct response from the event consumer is required. Events represent a significant change of state of the RAP business object, or of one of its children, that is relevant for follow-up processes. For example, when a new travel is created, you can enable consuming applications to trigger additional actions with the help of events. Most commonly, event-driven architecture is used in combination with a messaging service that notifies the event consumer when an event occurs. This follows a publisher-subscriber pattern where the event provider triggers the messaging when the specific event occurs. The event consumer is then notified and responsible to trigger the follow-up actions as required by the business use case. However, the event provider doesn't know about the event consumers and there is no two-sided communication pattern. This allows for asynchronous communication between event provider and event consumer.

RAP BO as Event Provider

As event provider, an event can be defined in the behavior definition of a RAP business object with the key word event `EventName` with or without parameters. After the event has been defined, it can be raised in the behavior implementation in the late save phase. Usually, an event with the respective message is raised after a change of state has been completed, e.g. if a BO entity like a travel or a booking has been created, updated or deleted. When the change is saved to the database, the event is raised during the SAVE sequence after the point of no return has passed. The `Event Binding` development artifact represents the design time definition of the event and maps the event defined in a RAP BO to a namespace, a business object and a business object operation like `modify`. The definition of the event in the behavior definition and the raising of the event in the SAVE sequence represents the runtime behavior of the respective event.



For more information on how to define a Business Event in a RAP BO, see: Develop Business Events

Related Information

Develop Business Events
Business Event Consumption

Save Options

Managed Scenarios

In managed scenarios, there are three options for setting up the save phase. You can use the built-in save sequence of the managed scenario, which is the default for any managed RAP BO. Based on the default save sequence, you can add additional implementations with the additional save. As an alternative to the save method of the managed save sequence runtime, you can implement your own save method as an unmanaged save.

Managed Save (Save Sequence Runtime)

The save sequence of the **managed save** is part of the business object runtime and is called after at least one successful modification was performed during the interaction phase. This is the default option for managed scenarios.

For more information, see Save Sequence Runtime.

Additional Save

You can add external functionality to the managed save sequence with the **additional save**. It is triggered after the managed runtime has gathered the changed data of business object's instances but before the final commit work has been executed. This feature is useful, for example, for reuse services like change documents or the application log.

For more background information about the additional save, see Additional Save. For more information about the implementation of the additional save, see Integrating Additional Save in Managed Business Objects.

Unmanaged Save

In a managed scenario, instead of using the save method of the managed save sequence runtime, you can implement an **unmanaged save**.

This option is useful if you want to bypass the save sequence of the managed business object's runtime and save business data and/or changes individually. By default, the managed runtime saves all changed instances of the business object's entity in the database table that is specified as persistent table DB_TABLE in the behavior definition (managed save).

For more background information about the unmanaged save, see Unmanaged Save. For more information about the implementation of the unmanaged save, see Integrating Unmanaged Save in Managed Business Objects.

Unmanaged Scenarios

In unmanaged business objects, you must create and implement your own save method for the save sequence.

For more information, see Implementing the Interaction Phase and the Save Sequence.

Related Information

Save Sequence Runtime
Additional Save
Unmanaged Save
Implementing the Interaction Phase and the Save Sequence

Additional Save

This section explains how to integrate the additional save within the transactional life cycle of managed business objects.

Use Case

In some application scenarios, an external functionality must be invoked during the save sequence, after the managed runtime has gathered the changed data of business object's instances, but before the final commit work has been executed.

For example, reuse services like **change documents** and the **application log** must be triggered during the save sequence and the changes of the current transaction must be written to change requests.

In real-life business applications, the data of business objects may change frequently. It is often helpful, and sometimes even necessary, to be able to trace or reconstruct changes for objects that are critical, for example for investigation or auditing purposes. The ABAP Application Server records changes to business data objects in change documents.

Application events can be centrally recorded in the application log. The entries of an application log contain information about who gave rise to a given event at what time and with which program.

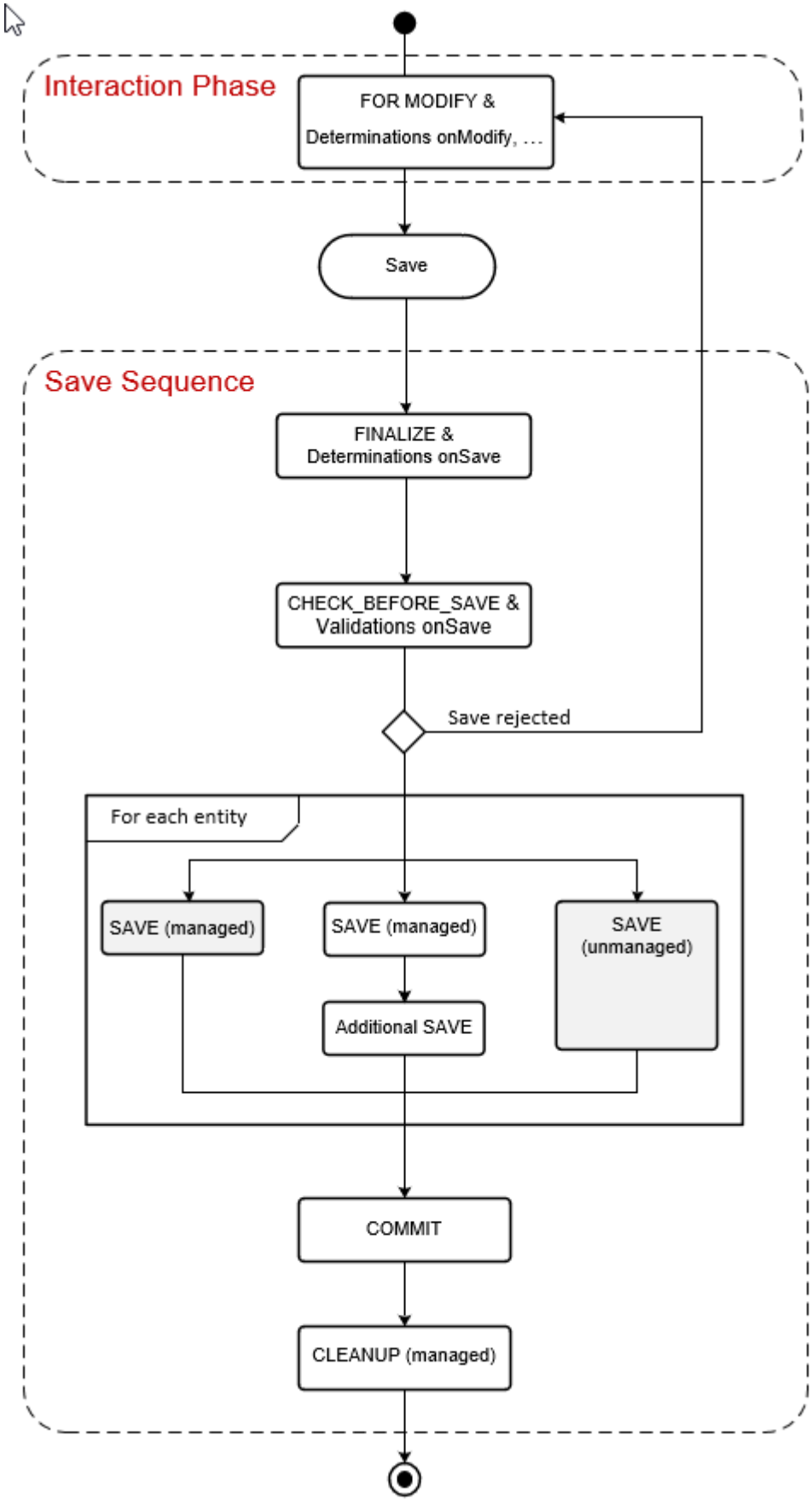
In order to integrate the additional save into the save sequence as a part of the managed runtime, you must first add the corresponding syntax to the behavior definition and then implement the saver handler method as a part of the behavior pool.

ℹ Note

If you would like to prevent the managed runtime from saving the entity's data and reuse your own save logic instead, you can integrate the unmanaged save instead. **More on this**: Integrating Unmanaged Save in Managed Business Objects.

Additional Save Within the Transactional Life Cycle

The following figure depicts the additional save within the transactional life cycle of a managed business object.



The save sequence is triggered for each business object after at least one successful modification (create, update, delete) was performed and saving data has been explicitly requested by the consumer. The save sequence starts with the `FINALIZE` processing step performing the final calculations and determinations before data changes can be persisted.

If the subsequent `CHECK_BEFORE_SAVE` call, including all onSave validations (validations with the trigger time `on save`), is positive for all transactional changes, the point-of-no-return is reached. From now on, a successful save is guaranteed by all involved BOs.

If, on the other hand, the result of the checks is negative at the time of `CHECK_BEFORE_SAVE`, a save is denied and the save sequence is interrupted. The consumer has now the option of modifying business object data and then trigger the save sequence again.

After the **point-of-no-return**, the save call persists all BO instance data from the transactional buffer into the database.

For each entity of an individual business object, the following options are available to execute the SAVE processing step:

- Managed save (default)
- Managed save in conjunction with additional save
- Unmanaged save (persisting the managed transactional buffer into the database is orchestrated by the application itself)

All change requests of the current LUW are committed. The actual save execution is finished by `COMMIT WORK`.

The final `CLEANUP` clears all transactional buffers of all business objects involved in the transaction by calling their corresponding cleanup implementations.

Activities Relevant to Developers

- Integrating Additional Save in Managed Business Objects
- Defining Additional Save in the Behavior Definition
- Implementing Additional Save

Unmanaged Save

This section explains how you can integrate unmanaged save within the transactional life cycle of managed business objects.

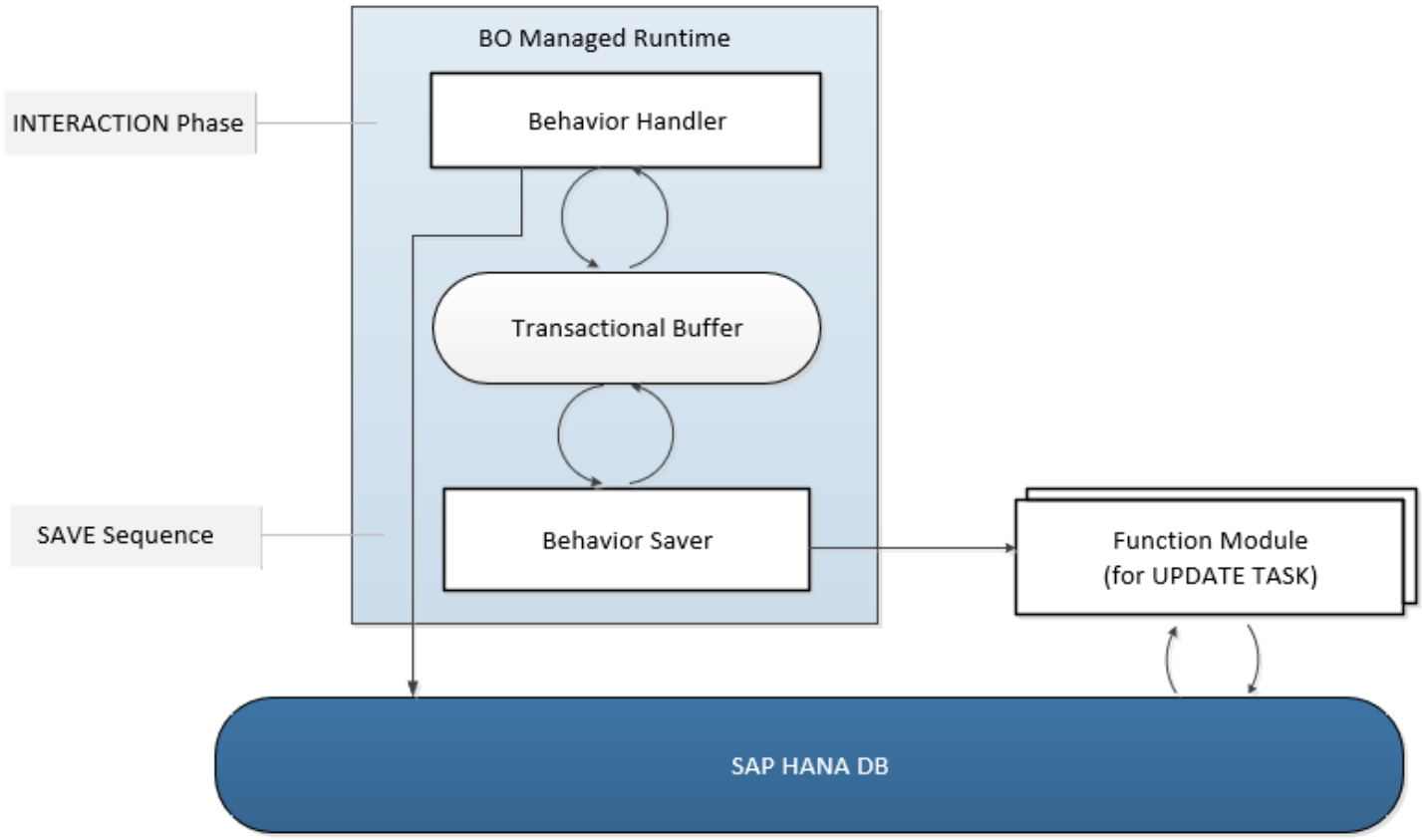
Use Case

In certain use cases you might be requested to prevent business object's managed runtime from saving business data (changes). By default, the managed runtime saves all changed instances of business object's entity in the database table that is specified as `persistent table DB_TABLE` in the behavior definition (managed save). However, you define for each entity of the business object or for the entire business object whether the complete save is done by the managed runtime or by the unmanaged save instead. This implementation flavor of a managed scenario may be relevant to you if you need to implement the interaction phase for your application anyway, but the update task function modules are already available.

The following figure outlines the main components of business objects managed runtime that, as an example, integrates function modules for persistent save of business data changes. Within the interaction phase, a consumer calls the business object operations to change business data and read instances with or without the transactional changes. The business object runtime keeps the changes in its internal transactional buffer which represents the state of instance data. After all changes on the related entity were performed, the instance data can be persisted. This is realized during the save sequence. To prevent the

managed runtime from saving the data, the function modules (for the update task) are called to save data changes of the relevant business object's entity (unmanaged save). In order to persist the business data changes, the function modules access the corresponding tables of the HANA database.

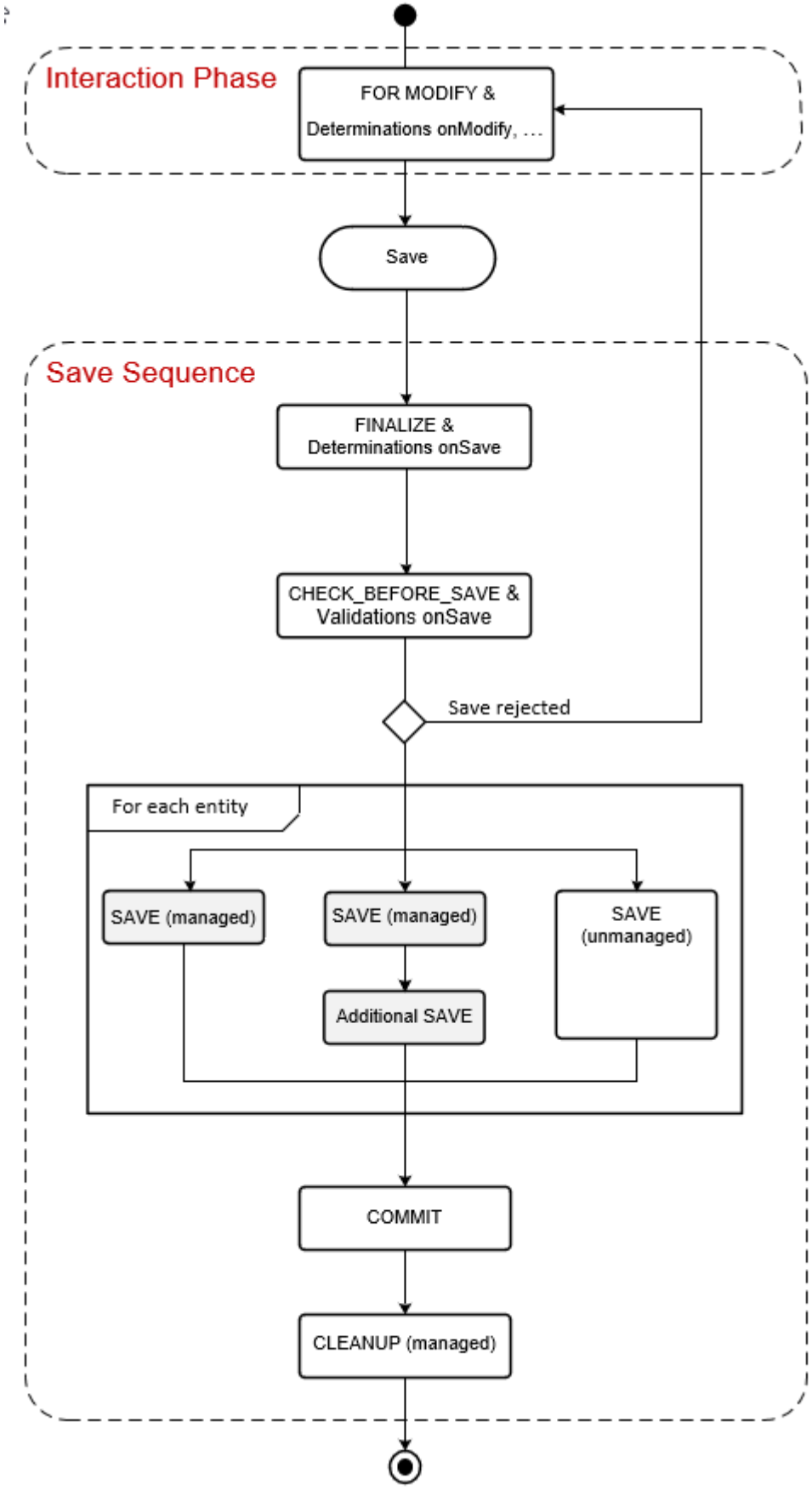
Note that the behavior handler can also directly access table data from the database during the interaction phase: Authorization checks, for example, require direct access to the table data on the database.



Runtime for the Managed Business Objects with Unmanaged Save

Unmanaged Save in Transactional Life Cycle

The following figure depicts the unmanaged save within the transactional life cycle of a managed business object.



Unmanaged Save within the Transactional Processing

The save sequence is triggered for each business object after at least one successful modification (create, update, delete) was performed and saving data has been explicitly requested by the consumer. The save sequence starts with the `FINALIZE` processing step performing the final calculations and determinations before data changes can be persisted.

If the subsequent `CHECK_BEFORE_SAVE` call, including all onSave validations (validations with the trigger time `on save`), is positive for all transactional changes, the point-of-no-return is reached. From now on, a successful save is guaranteed by all

involved BOs.

If, on the other hand, the result of the checks is negative at the time of `CHECK_BEFORE_SAVE`, a save is denied and the save sequence is interrupted. The consumer has now the option of modifying business object data and then trigger the save sequence again.

After the **point-of-no-return**, the save call persists all BO instance data from the transactional buffer in the database.

For each entity of an individual business object, the following options are available to execute the SAVE processing step:

- Managed save (default)
- Managed save in conjunction with additional save
- Unmanaged save (to prevent the managed runtime from saving the entities data)

All change requests of the current LUW are committed. The actual save execution is finished by `COMMIT WORK`.

The final `CLEANUP` clears all transactional buffers of all business objects involved in the transaction.

Activities Relevant to Developers

In order to integrate unmanaged save into the save sequence as a part of the managed runtime, you must first add the corresponding syntax to the behavior definition and then implement the saver handler method as a part of the behavior pool.

- Integrating Unmanaged Save in Managed Business Objects
- Defining Unmanaged Save in the Behavior Definition
- Implementing the Save Handler Method

Business Object Provider API

The implementation of business object functionality is done in a special type of class pool, which refers to the behavior definition, in which the behavior for the business object is defined. For the implementation type **unmanaged**, application developers must implement essential components of the REST contract themselves. For this, all desired operations (create, update, delete) must be specified in the corresponding behavior definition artifact by using the `Behavior Definition Language (BDL)` before they are implemented with ABAP. For the implementation type **managed**, standard operations, such as create, update and delete are assumed by the framework and application developers only have to implement the components of the business logic that are part of the specific application logic as non-standard behavior.

The implementation is carried out in a special type of class pool, the behavior pool, which refers to the behavior definition. The global class is defined with the following syntax:

```
CLASS ClassName DEFINITION
PUBLIC ABSTRACT FINAL
FOR BEHAVIOR OF BehaviorDefinition.
ENDCLASS.

CLASS ClassName IMPLEMENTATION.
ENDCLASS.
```

The concrete implementation of the business logic is based on the ABAP language and the Business Object Behavior API.

The implementation tasks are roughly divided into an **interaction phase** and a **save sequence**. The interaction phase is represented by the local handler class and the save sequence by the local saver class.

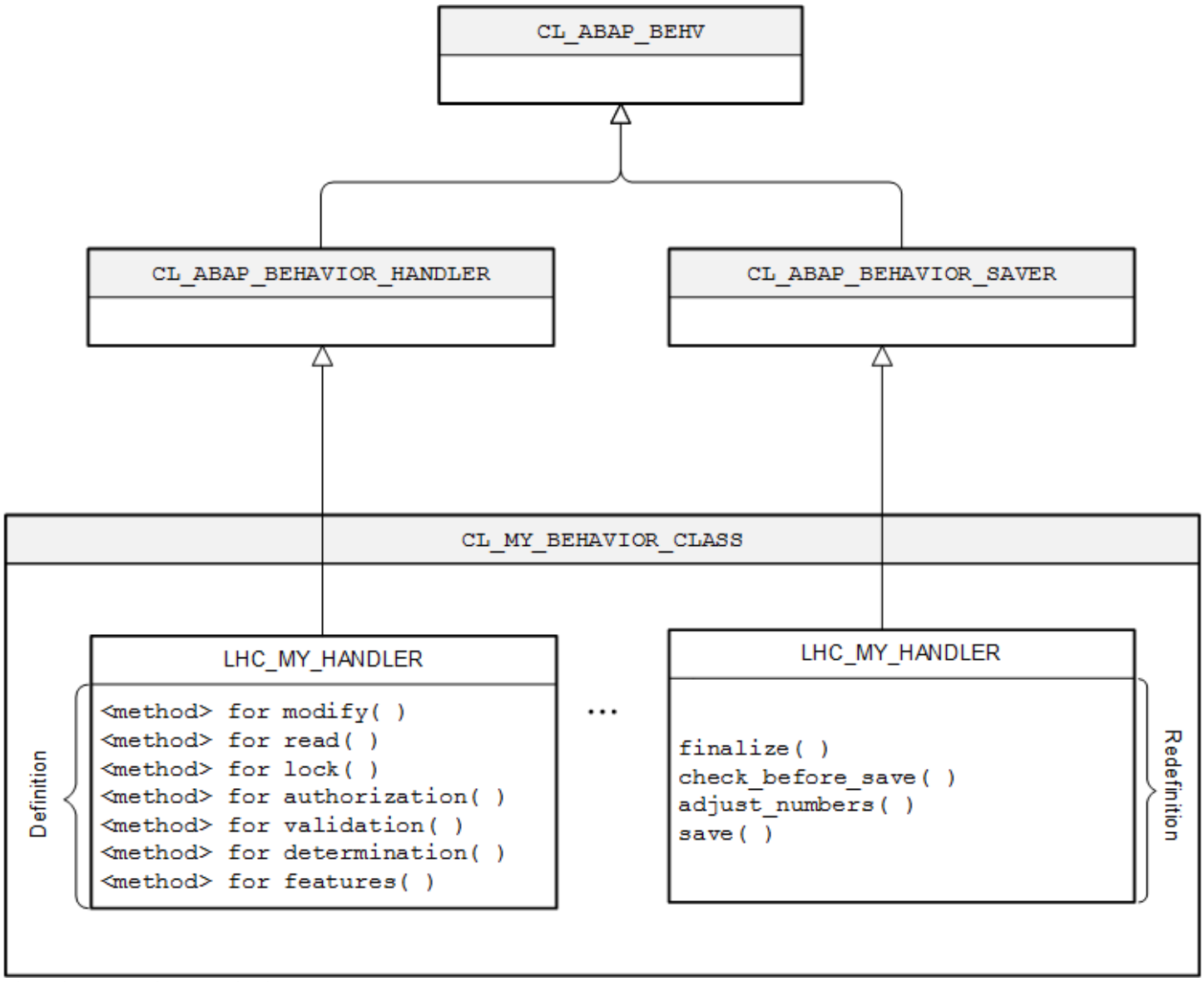
→ Remember

There are some specific rules for assigning names to local classes in a behavior pool. Both handler classes and saver classes are recognized by derivation from the respective system base class. The names LCL_HANDLER and LCL_SAVER are suggested by ADT when you create the class pool, but can be changed. We recommend applying naming conventions for behavior pools and local handler and saver classes corresponding to Naming Conventions for Development Objects.

At the top of the class hierarchy for the BO API is the class CL_ABAP_BEHV. This class is the foundation class for the handler and the saver class. It defines some fundamental data types to be used in the behavior processing (such as field names in derived type structures) and also provides message creation methods.

The classes that derive from this base class are:

- CL_ABAP_BEHAVIOR_HANDLER – This class is the base class for the handler.
- CL_ABAP_BEHAVIOR_SAVER – This class is the base class for the saver. It specifies the signature of all methods used to implement the save sequence of a business object provider.



Hierarchy Tree of the BO Behavior API

Detailed Information

- Implementing Handler Classes
- Implementing Saver Classes

- Using Implicit Response Parameters
- Derived Data Types

Handler Classes

To implement the behavior specified in the behavior definition, a special global ABAP class, the behavior pool is used. This global class is implicitly defined as ABSTRACT and FINAL. So, the behavior implementation cannot be found from outside the BO. A behavior pool can have static methods, CLASS-DATA, CONSTANTS and TYPES. The application may place common or even public aspects of its implementation here.

The real substance of a behavior pool is located in **Local Types**. Here you can define two types of special local classes: handler classes for the operations within the interaction phase and saver classes for the operations within the save sequence.

Within the global behavior pool one or multiple local handler classes are defined. Each such local class inherits from the base class CL_ABAP_BEHAVIOR_HANDLER. The signature of the handler methods are type-based on the entity that is defined by the keyword FOR *[*OPERATION*]* entity. If there is an alias defined in the behavior definition, the alias has to be used.

Syntax: Definition of the Local Handler Class

```
CLASS lcl_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.

  PRIVATE SECTION.

/*  FOR MODIFY method declaration               */
    METHODS modify_method FOR MODIFY
      [IMPORTING]
        create_import_parameter      FOR CREATE entity
        update_import_parameter      FOR UPDATE entity
        delete_import_parameter      FOR DELETE entity
        action_import_parameter      FOR ACTION entity~action_name
          [REQUEST requested-fields]
          [RESULT action_export_parameter]
        create_ba_import_parameter   FOR CREATE entity\_association.

/*  FOR AUTHORIZATION method declaration      */
    METHODS auth_method FOR AUTHORIZATION
      [IMPORTING] keys REQUEST requested_features FOR entity
      RESULT    result_parameter.

/*  FOR FEATURES method declaration  */
    METHODS feature_ctrl_method      FOR FEATURES
      [IMPORTING] keys REQUEST requested_features FOR entity
      RESULT    result_parameter.

/*  FOR GLOBAL FEATURES method declaration */
    METHODS get_global_features FOR GLOBAL FEATURES
      [IMPORTING]  REQUEST requested_features FOR entity
      RESULT result_parameter.

/*  FOR LOCK method declaration      */
    METHODS lock_method FOR LOCK
      [IMPORTING] lock_import_parameter FOR LOCK entity.

/*  FOR READ method declaration      */
    METHODS read_method FOR READ
      [IMPORTING] read_import_parameter FOR READ entity
      RESULT read_export_parameter.

/*  FOR READ by association method    */
    METHODS read_by_assoc_method_name  FOR READ
      [IMPORTING] read_ba_import_parameter FOR READ entity\_association
      FULL full_read_import_parameter
      RESULT read_result_parameter
      LINK read_link_parameter.
```

```
ENDCLASS.
```

Method Summary

Method	Description
`<method> FOR MODIFY`	Handles all changing operations (create, update, delete, and specific actions as they are specified in the behavior definition) of an entity
`<method> FOR AUTHORIZATION`	Implements authorization checks for accessing entities.
`<method> FOR FEATURES`	Implements the dynamic feature control of entities
`<method> FOR GLOBAL FEATURES`	Implements the global feature control of entities
`<method> FOR LOCK`	Implements the locking of entities corresponding to the lock properties in the behavior definition
`<method> FOR READ`	Handles the processing of reading requests

Method Details

- <method> FOR MODIFY
- <method> FOR LOCK
- <method> FOR READ
- Authorization Implementation
- FOR GLOBAL FEATURES (ABAP Keyword Documentation)
- FOR INSTANCE FEATURES, FEATURES (ABAP Keyword Documentation)

`<method> FOR MODIFY`

Handles all changing operations of an entity.

The `FOR MODIFY` method implements the standard operations create, update, delete, and application-specific actions, as they are specified in the behavior definition.

→ Tip

The `FOR MODIFY` method can handle multiple entities (root, item, sub item) and multiple operations during one processing step. In some cases, it might be useful to split the handler implementation into separate methods. Then, multiple behavior handlers, that is, multiple local behavior classes within one global behavior pool or even in multiple global behavior pools, can be defined.

Declaration of `<method> FOR MODIFY`

The declaration of the `<method> FOR MODIFY` expresses what changing operations this method is responsible for. In extreme cases, this is the total number of all changing operations that are possible according to the behavior definition.

Each individual specification within the declaration of `modify_method FOR MODIFY` consists of a combination of an operation with an entity or an entity part. To refer to the entities, the alias given in behavior definition is used - if there is any.

Each operation type has an import parameter `<operation>_import_parameter` for the incoming instance data and. Its name is freely selectable. The method includes an export parameter `action_export_parameter` if the operation type expects one. Action, for example, can have export parameters for their results.

The import parameters for `CREATE`, `UPDATE` and `CREATE by association` include the control structure `%control` to identify which fields have been filled by the caller.

You can declare all operations in a single `<method> FOR MODIFY`:

```
METHODS modify_method FOR MODIFY
  [IMPORTING]
    create_import_parameter      FOR CREATE entity
    update_import_parameter      FOR UPDATE entity
    delete_import_parameter      FOR DELETE entity
    action_import_parameter      FOR ACTION entity~action_name
      [REQUEST requested-fields]
      [RESULT action_export_parameter]
    create_ba_import_parameter   FOR CREATE entity\_association.
```

You can also declare a `<method> FOR MODIFY` for each operation. In many cases, it can be beneficial to implement the individual `MODIFY` operations in separate methods. This may be particularly the case if the implementations for the respective operations are more extensive.

```
METHODS:
  create_entity_method FOR MODIFY
    [IMPORTING] create_import_parameter    FOR CREATE entity,
  update_entity_method FOR MODIFY
    [IMPORTING] update_import_parameter    FOR UPDATE entity,
  delete_entity_method FOR MODIFY
    [IMPORTING] delete_import_parameter    FOR DELETE entity,
  action_method FOR MODIFY
    [IMPORTING] action_import_parameter    FOR ACTION entity~action_name
    RESULT    action_export_parameter,
  create_by_association FOR MODIFY
    [IMPORTING] create_ba_import_parameter FOR CREATE entity\_association.
```

For the sake of better readability, the keyword `IMPORTING` can be specified before the first import parameter.

The parameters can also be explicitly declared as `REFERENCE( ... )`; However, the declaration as `VALUE( ... )` is not allowed and therefore the importing parameters cannot be changed in the method.

ℹ Note

The data types with which the parameters are implicitly provided by the ABAP compiler are **derived types** resulting from the behavior definition. They usually contain at least the instance key according to the CDS definition, or even the full row type, as well as other components that result from the model (action parameter) or other features of the BO, such as `%pid` in case of late numbering. For more information, see Derived Data Types.

The method `FOR MODIFY` has three **implicit changing parameters** `failed`, `mapped`, and `reported`. These parameters can (but do not need to) be explicitly declared (developers may find this explicit declaration helpful), like this:

```
METHODS method_name FOR MODIFY
    [IMPORTING]
       create_import_parameter    FOR CREATE entity
        ...
     CHANGING  failed   TYPE DATA
```

```
              mapped   TYPE DATA
              reported TYPE DATA.
```

Since the derived types also come here into play, you cannot explicitly write them down. The ABAP compiler accepts the generic type DATA and replaces it with the respective derived types resulting from the behavior definition.

Each of FAILED, MAPPED, REPORTED is a structure type with one component per entity from the behavior definition (that is, per entity in a business object). The names of the components are the aliases defined in the behavior definition or else the original entity names.

All parameters and components of these structures are tables to allow mass processing. Together with the bundling of multiple operations in a method, it is possible to implement large modification requests in a single FOR MODIFY method call.

Implementation of <method> FOR MODIFY

When is the FOR MODIFY Method Called?

The FOR MODIFY method is called when the BO framework processes a change request that contains at least one of the operations defined in the method FOR MODIFY.

The FOR MODIFY method can determine which operations are specifically given, for example, in this way:

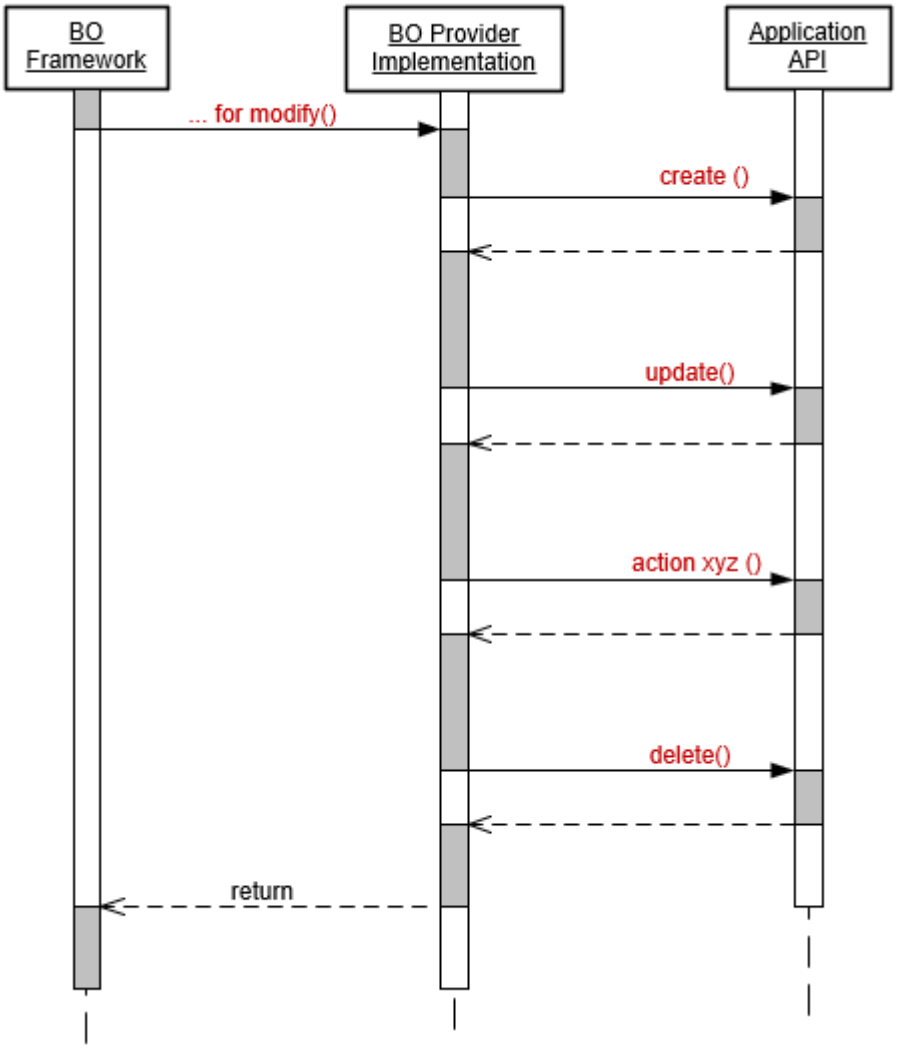
```
... parameter   IS [NOT] INITIAL.
```

Example

```
IF create_import_parameter IS NOT INITIAL.
... " code for creating entities
ENDIF.
```

Sequence of Processing of Individual Operations

The BO framework does not specify an order for the processing of individual operations within a FOR MODIFY call. It is therefore assumed that the application layer processes all the individual operations that are passed in a meaningful order for them. For example, it is usually useful to process create operations before update operations.



Sequence of Operations Processing within a `FOR MODIFY` Method call – An Example

Retrieving Results from Operation Processing

To get the output of an action call with a defined `RESULT`, the named export parameter `action_export_parameter` must be filled.

There are no explicit return parameters to be filled for all other operations. However, the three returning structures `failed`, `reported`, and `mapped` must be filled when the corresponding events happen. Their construction results in a fairly readable pattern, for example, to report failed instances or to store messages for instances:

```
APPEND ...  TO failed-Item.
APPEND ...  TO reported-Root.
```

All derived types also contain components that do not originate from the line type of the entity and begin with the character % to avoid naming conflicts with original components. For example, the row type of a `failed` table contains a component `%fail` to store the symptom for a failed instance; Also, an include structure `%key` that summarizes the primary key fields of the entity. `%key` is part of almost all derived types, including operation parameters. An overview of all derived types is given in

Thus, the above pattern can be as follows:

```
APPEND VALUE #( %KEY = <item>-%KEY %FAIL = IF_ABAP_BEHV=>CAUSE-... %CID = ...) TO failed-Item.
```

Related Information

Implicit Response Parameters

<method> FOR LOCK

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

The FOR LOCK method is automatically called by the orchestration framework before a changing (MODIFY) operation such as update is called.

Declaration of <method> FOR LOCK

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

ℹ Note

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

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

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

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

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

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

Import Parameters

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

- ID fields

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

ℹ Note

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

Changing Parameters

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

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

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

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

```
            reported TYPE DATA.
```

Implementation of `method FOR LOCK`

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

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

An example on how to implement the `method FOR LOCK` is given in Implementing the LOCK Operation.

Related Information

Implicit Response Parameters

<method> FOR READ

Implements a handler for processing reading requests.

The `FOR READ` method is used to return the data from the application buffer. If the buffer is empty, the data is read from the database (which typically populates the application buffer).

There are two options to read data from the application buffer:

- Direct READ, see Declaration of <method> FOR READ.
- READ by association, see Declaration of <method> FOR READ By Association.

Declaration of `<method> FOR READ`

Similar to `<method> FOR MODIFY`, the handler `<method> FOR READ` is also implemented to handle mass requests. It is also designed to bundle multiple operations.

```
METHODS method_name FOR READ
       [IMPORTING] read_import_parameter FOR READ entity
       RESULT read_result_parameter.
```

Again, for the sake of better readability, the keyword `IMPORTING` can be specified before the import parameter. The name of the import parameter `read_import_parameter` can be freely selected. It imports the key(s) of the instance entity to be read and indicates which elements are requested.

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

The parameter `RESULT` is a changing parameter. Its name can be freely selected.

Import Parameters

The row type of the import table `read_import_parameter` provides the following:

- ID fields

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

- `%CONTROL`

  The control structure reflects which elements are requested by the consumer.

Exporting Parameters

- `read_result_parameter`

  Returns the successfully read data.

  The row type of this table provides all elements that are specified in the element list of the entity that is read. Only the requested elements, which are indicated in the `%control` structure, must be filled.

Changing Parameters

In addition to the explicitly declared return parameter, the READ method also provides the implicit `CHANGING` parameters `failed`, `mapped` and `reported`.

- The `failed` parameter is used to log the entries that could not be read. You can specify the fail cause for the READ, for example `not_found`.
- The `mapped` parameter must not be filled in READ implementations.
- The `reported` parameter must not be filled in READ implementations.

For more information about the implicitly declared parameters, see Implicit Response Parameters.

Declaration of `<method>` `FOR READ` By Association

The syntax for the READ by association is similar to the one for the direct READ.

```
METHODS
     method_name   FOR READ
       [IMPORTING] read_ba_import_parameter FOR READ entity\_association
       FULL full_read_import_parameter
       RESULT read_result_parameter
       LINK read_link_parameter.
```

As for the other operation implementations, the keyword `IMPORTING` can optionally be specified explicitly. All parameter names can be freely selected.

The importing parameter `read_ba_import_parameter` imports the key(s) of the entity instance whose associated entity instance shall be read. In addition, it indicates which elements from the associated entity shall be read.

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

The placeholder `association` refers to the association along which you want to read data, for example `_booking` if you want to read all bookings associated to one travel instance.

The parameter `full_read_import_parameter` indicates whether the `RESULT` parameter must be filled or if only the `LINK` parameter must be filled. It has a boolean value.

The parameter `RESULT` is an exporting parameter. It returns the requested elements that are indicated in the importing parameter if the `FULL` parameter is set.

The parameter `LINK` is also an exporting parameter. It returns the key elements of the source and target entities no matter if the `FULL parameter` is set.

Import Parameters

- The row type of the import table `read_ba_import_parameter` provides the following data:
  - ID fields

    All elements that are specified as a key in the related CDS view.
  - `%CONTROL`

    The control structure reflects which elements of the associated entity are requested by the consumer.
- The type of the import parameter `full_read_import_parameter` is a character with boolean value.

Exporting Parameters

- `read_ba_import_parameter`: Returns the successfully read data from the associated entity if the `FULL` parameter is set.

  The row type of this table provides all elements that are specified in the element list of the entity that is read. Only the requested elements, which are indicated in the `%control` structure, must be filled.
- `read_link_parameter`: Returns the source and target key of the successfully read entity instances.

For more information about the implicitly declared parameters, see Implicit Response Parameters.

Changing Parameters

In addition to the explicitly declared parameters, the method for READ by association also provides the implicit `CHANGING` parameters `failed`, `mapped`, and `reported`.

- `read_ba_import_parameter`: Returns the successfully read data from the associated entity if the `FULL` parameter is set.

  The row type of this table provides all elements that are specified in the element list of the entity that is read. Only the requested elements, which are indicated in the `%control` structure, must be filled.
- `read_link_parameter`: Returns the source and target key of the successfully read entity instances.
- The `failed` parameter is used to log the entries that could not be read. You can specify the fail cause for the READ, for example `not_found`.
- The `mapped` parameter must not be filled in READ implementations.
- The `reported` parameter must not be filled in READ implementations.

For more information about the implicitly declared parameters, see Implicit Response Parameters.

<method> FOR FEATURES

This method implements the dynamic feature control for entities.

Dynamic feature control can be used for the standard operations `update`, `delete`, and `create_by_association`, for actions and on field level. Depending on the feature conditions is the operation executable or not. The `<method> FOR FEATURES` is called by the RAP runtime engine for every operation or field that is dynamically controlled.

Declaration of <method> FOR FEATURES

The operations or fields that are dynamically controlled are defined in the behavior definition with `features: instance`.

The dynamic feature control for an entity is implemented in a handler class using the method `feature_ctrl_method`. The signature of this handler method is defined by the keyword `FOR FEATURES`, followed by the input parameters `keys` and the `requested_features` of the `entity`.

```
METHODS feature_ctrl_method FOR FEATURES
  [IMPORTING] keys REQUEST requested_features FOR entity
  RESULT result.
```

Again, for the sake of better readability, the keyword `IMPORTING` can be specified before the import parameter.

ℹ Note

The name of the `<method> FOR FEATURES` can be freely chosen. Often `get_features` is used as method name.

Import Parameters

- `keys`

  The table type of `keys` includes all elements that are specified as a key for the related `entity`.

- `requested_features`

  The structure type of `requested_features` reflects which elements (fields, standard operations and actions) of the `entity` are requested for dynamic feature control by the consumer.

Export Parameters

The export parameter `result` is used to return the feature control values. It includes, besides the key fields, all the fields of the entity, standard operations and actions for which the features control was defined in the behavior definition.

Export Parameters

In addition to the explicitly declared export parameter, the `FOR FEATURE` method also provides the implicit `CHANGING` parameters `failed` and `reported`.

Related Information

Implicit Response Parameters

Saver Classes

The save sequence is called for each business object after at least one successful modification was performed using the BO behavior APIs in the current LUW.

For more information, see Save Sequence Runtime.

Method Details

- FINALIZE
- CHECK_BEFORE_SAVE
- ADJUST_NUMBERS
- SAVE
- CLEANUP

- CLEANUP_FINALIZE

Derived Data Types

In the behavior implementation of a business object, you make use of data types which the ABAP compiler implicitly derives from the involved CDS views and the behavior definition. These derived data types contain key and data fields of CDS entities and ensure the type-safe access to RAP BOs.

Derived Data Types in Method Signatures

The parameters of method signatures for RAP operations are mainly typed with derived data types.

Example

The following method signature originates from the managed flight reference scenario (`/DMO/FLIGHT_MANAGED`).

```
METHODS copyTravel FOR MODIFY IMPORTING keys FOR ACTION travel~copyTravel.
```

Based on this signature, the ABAP compiler determines the following import and changing parameters with the corresponding derived data types:

```
IMPORTING keys     TYPE TABLE    FOR ACTION IMPORT  /DMO/I_TRAVEL_M\\travel~copytravel
CHANGING  mapped   TYPE RESPONSE FOR MAPPED   EARLY /DMO/I_TRAVEL_M
          failed   TYPE RESPONSE FOR FAILED   EARLY /DMO/I_TRAVEL_M
          reported TYPE RESPONSE FOR REPORTED EARLY /DMO/I_TRAVEL_M
```

The parameters `mapped`, `failed` and `reported` are response parameters that are implicitly added during design time.

For more information on response parameters, see Implicit Response Parameters.

Explicit Declaration of Derived Data Types

Derived data types can also be used to explicitly declare variables. For example, an internal table can be declared with the type `TYPE TABLE FOR`:

```
travels TYPE TABLE FOR CREATE /DMO/I_Travel_M\\travel
```

This variable can then be used in EML to create instances of the travel entity.

For further information on EML, see Entity Manipulation Language (EML).

For further information on the explicit decaration with derived data types, see Declaration of Variables with BDEF Derived Types (ABAP-Keyword Documentation).

Components of Derived Data Types

Internal structures and tables declared with derived data types do not only contain key and data fields, but also other components that do not derive their line type from the entity. These components have special, tailor-made line types that provide additional information required in the context of transactional processing. The names of those RAP components begin with % to avoid naming conflicts with components of the CDS entities. Among the components, there are component groups available for summarizing groups of table columns under a single name. They begin with % as well and simplify the handling of

derived types for developers. For example, `%key` summarizes all primary keys. The use of the components depends on the RAP operation and/or context, for example, `%pid` is only available for late numbering scenarios.

Components of derived data types can also be used to type internal structures or tables:

```
TYPES: it_booking_update        TYPE TABLE FOR UPDATE /DMO/I_Travel_M\\travel,
       ltype_for_update         TYPE LINE OF it_booking_update,
       ltype_for_key            TYPE ltype_for_update-%key.
```

For further information on the components of derived data types, see Components of BDEF Derived Types (ABAP Keyword Documentation).

Implicit Response Parameters

When implementing a BO contract, you make use of implicit response parameters. These parameters do not have fixed data types and instead are assigned by the compiler with the types derived from behavior definition.

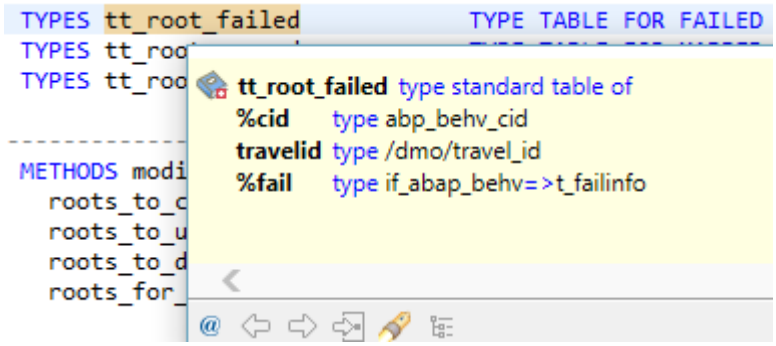
The implicit parameters can be declared explicitly as `CHANGING` parameters in the method signature of the handler classes by using the generic type `DATA`:

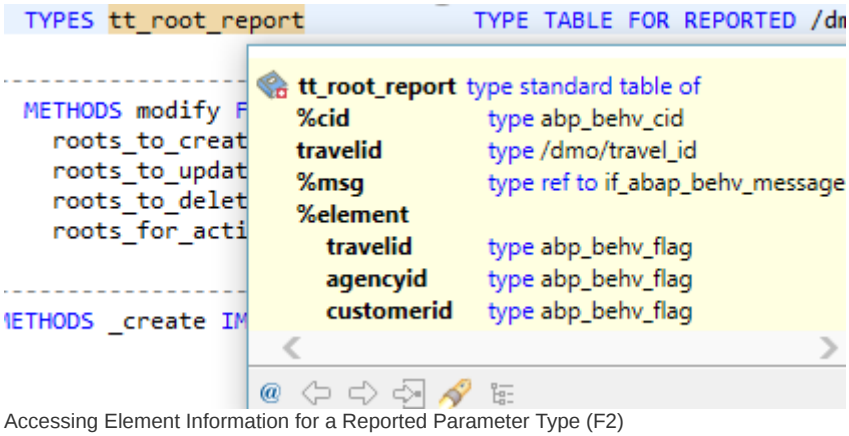
```
METHODS method_name FOR MODIFY | READ | LOCK
    [IMPORTING]

      <operation>_import_parameter    FOR <OPERATION> entity
      ...

     CHANGING  failed   TYPE DATA
              [mapped   TYPE DATA]  "Relevant for CREATE only
               reported TYPE DATA.
```

The ABAP compiler replaces the type `DATA` with the respective **derived types** resulting from the concrete behavior definition.

Implicit Parameters

Parameter	Description

Parameter	Description
FAILED	This exporting parameter is defined as a nested table which contains one table for each entity defined in the behavior definition. The **failed** tables include information for identifying the data set where an error occurred: • %CID and • ID of the relevant BO instance. The reason for the failure is specified by the predefined component: • %FAIL, which stores the symptom of the failure.  Accessing Element Information for Failed Parameter Type (F2)

Parameter	Description
REPORTED	This exporting parameter is used to return messages. It is defined as a nested table which contains one table for each entity defined in the behavior definition. **i Note** The termination at runtime due to errors must always be implemented depending on entries in the fail structure that indicate the fail cause. Messages as part of the reported structure only offer additional information about the respective circumstances, but aren't reliable indications for termination at runtime. It's always recommended to consider the fail cause in the fail structure for the program design. The **reported** tables include data for instance-specific messages. The data set for which the message is relevant is identified by the following components: • %CID • ID of the relevant instance • %MSG with an instance of the message interface IF_ABAP_BEHV_MESSAGE • %ELEMENT which refers to all elements of an entity.  Accessing Element Information for a Reported Parameter Type (F2) **i Note** Messages that are not related to a specific (entity) instance can be returned using the %OTHERS component.
MAPPED	This mapped parameter is defined as a nested table which contains one table for each entity defined in the behavior definition. The **mapped** parameters provide the consumer with ID mapping information. They include the information about which key values were created by the application for given content IDs. The BO runtime passes the created key values in any subsequent calls in the same request and in the response. The relevant data set is identified by the following components: • %CID • %KEY

(Copied Content from derived types topic)

- MAPPED [LATE] - The mapped result parameters provide the consumer with ID mapping information. By default, the mapping information is already available in the interaction phase (early mapped). The CID is then mapped to the real key or to the PID. Using the addition LATE, you specify that the mapping information is only available in the save sequence. This plays a role when providing the late numbering (see also: ADJUST_NUMBERS) where the PID is mapped to the real key.

- `FAILED [LATE]` - The failed parameters include information for identifying the data set where an error occurred. (Early) `FAILED` is provided during the interaction phase and contains the `CID` or the `KEY` to indicate instances for which an operation failed. `FAILED` with the additional specification `LATE` is only provided during the save sequence and contains the `PID` or the `KEY`, but not the `CID`.
- `REPORTED [LATE]` - The reported parameters are used to return messages in case of failure. (Early) `REPORTED` is provided during the interaction phase and contains the `CID` or the `KEY` to indicate instances for which an operation failed. `REPORTED` with the additional specification `LATE` is only provided during the save sequence and contains the `PID` or the `KEY`, but not the `CID`.
- `READ RESULT`

Components of Derived Data Types

All derived data types in the context of the ABAP RESTful programming model also contain components that do not originate from the row type of the entity and begin with the character `%` to avoid naming conflicts with original components. For example, the row type of a failed table contains a component `%fail` to store the symptom for a failed instance and also an include structure `%key` that contains all primary key fields of the entity.

EXAMPLE: Usage of `%...` components in a failed parameter

```
APPEND #VALUE(%KEY = ... %FAIL = ...) TO failed-entity.
```

The following list provides you with a description of the most common %... components:

Component	Description
`%CID`	The content ID `%CID` is a temporary primary key for an instance, as long as no primary key was created by the BO runtime. The content ID is always provided by the SADL framework. It is only needed in case of internal numbering and/or late numbering. The content ID provides the reference between the related entity instances. A good example is a ***DEEP INSERT*** for multiple parent/child instances with internal numbering and/or late numbering. In this case, the references between the child and parent instances are established using the content ID `%CID`.
`%CID_REF`	A reference to the content ID `%CID`.
`%tky`	Contains all key elements of an entity (CDS view) including the derived key components, for example `%IS_DRAFT` in draft scenarios. `%tky` is part of almost all derived types, including trigger parameters in the `for modify( )` method.
`%PID`	Defines the preliminary ID, before the final key is set in the `ADJUST_NUMBERS` method.. The preliminary ID is only available when `LATE NUMBERING` is defined in the behavior definition without the addition `IN PLACE`.

Component	Description
%CONTROL	Reflects which elements are requested by the consumer. The fields of the %CONTROL structure provide information, depending on the operation, about which elements of the entity are supplied in the request (for CREATE and UPDATE operations) or which elements are requested in the read request (for READ operations). For each entity element, this control structure contains a flag which indicates whether the corresponding field was provided/requested by the consumer or not. The element names of the entity have the uniform type ABP_BEHV_FLAG. **ℹ Note** The possible constants are defined in the basis handler interface `if_abap_behv=>mk-<...>`. For example, the elements that have the value `if_abap_behv=>mk-on` in the %CONTROL structure are used to handle delta updates within the UPDATE operation.
%DATA	Contains all data elements of an entity (CDS view).
%FAIL	Stores the symptom for a failed data set (BO instance). **ℹ Note** The possible values (`unspecific`, `unauthorized`, `not_found`, and so on) are defined by the ENUM type `IF_ABAP_BEHV=> T_FAIL_CAUSE`.
%MSG	Provides an instance of the message interface IF_ABAP_BEHV_MESSAGE. **→ Tip** The component %MSG of type `REF TO IF_ABAP_BEHV_MESSAGE` includes IF_T100_DYN_MSG. If you do not need your own implementation of this interface, then you can benefit from the provided standard implementation by using the inherited methods `new_message( )` or `new_message_with_text( )`.
%ELEMENT	Refers to all elements of an entity.
%PARAM	Holds the import/result type of actions.

RAP BO Best Practices

Here you can find best practices for RAP BO providers and consumers. Best practices also include recommended guidelines about how to model your RAP business object.

RAP Business Object Contract

The RAP business object contract defines a set of rules for the business object provider and consumer implementation to ensure consistency and reliability.

Strict Mode

The strict mode ensures that a RAP business object is lifycycle-stable and upgrade-save.

Determination and Validation Modelling

RAP Business Object Contract

The RAP business object contract defines a set of rules for the business object provider and consumer implementation to ensure consistency and reliability.

A RAP business object represents a typed API that is integrated in the ABAP language. When consuming a RAP BO, either via EML or via service binding (eg OData), the consumer must be sure that the result of any request is reliable and consistent and that every request returns the same results under the same circumstances. It's the provider's responsibility to ensure these correct results, no matter if the RAP BO provider is managed or unmanaged.

Example

Given the situation that a RAP consumer executes a create request. It is expected that the RAP BO either returns a new instance (represented by the primary key of this instance), or reports a fail cause in case the create request was not executable.

The RAP business object contract defines the expectations for a BO consumer and at the same time provides guidelines for the provider of what needs to be implemented to fulfill these expectations. It is defined by a set of rules, which is illustrated in the following topics. These rules will help you, as an application developer, to implement the RAP handler methods correctly.

Some of the contract rules are enforced by runtime checks. For example, using an `EML MODIFY` statement in a read-only implementation results in a runtime error. Other contract rules can be checked by running the ABAP test cockpit. For your applications to be upgrade-stable and sustainable, it is absolutely indispensable that you stick to the guidelines that are presented in the following sections.

You will find general RAP BO provider rules as well as information for each RAP implementation method.

General RAP BO Implementation Contract

This topic explains general rules that apply to all behavior implementation methods of a RAP business object.

Each RAP BO provides a typed API with which the BO provider and consumer can communicate. Dedicated methods are provided by the RAP framework to implement the behavior for a RAP business object. The RAP runtime framework ensures stability and consistency of RAP business objects and thus expects certain implementation in the dedicated handler and saver methods. Consequently, there are statements that do not coincide with the predefined functionality of these RAP implementation methods and which are therefore forbidden.

Scope	Forbidden Statements

Forbidden ABAP statements in all RAP handler methods	• `EML COMMIT ENTITIES`
	• `EML ROLLBACK ENTITIES`
	• `COMMIT WORK`
	• `ROLLBACK WORK`
	• 🏠 `CALL FUNCTION function IN UPDATE TASK`
	• 🏠 `CALL FUNCTION IN BACKGROUND UNIT`
	• 🏠 `SUBMIT AND RETURN`
	• 🏠 Any SCREEN- / SAP GUI-related statement, for example
	  ◦ `CALL SCREEN`
	  ◦ `SET SCREEN`
	  ◦ `MESSAGE`
	  ◦ `CALL DIALOG`
	  ◦ `CALL TRANSACTION`

Forbidden ABAP statements in read-only RAP handler methods	• `EML MODIFY ENTITIES`
Forbidden ABAP statements in all RAP saver methods	• 🏠 `SUBMIT AND RETURN`
Forbidden ABAP statements in late save phases (ADJUST_NUMBERS, SAVE)	• 🏠 `CALL FUNCTION function IN UPDATE TASK` • 🏠 `CALL FUNCTION IN BACKGROUND UNIT`

For a detailed handler restriction list, see Restrictions in RAP Handler and Saver Methods.

Importing Parameters

The importing parameters differ among the RAP handler and saver classes. Some methods require a complete set of instance data, other don't operation on instances at all. There are, however, common components of these importing parameters.

Instance-Bound Operations

Implementation methods for instance-bound operations always import the **instance identifier** to identify the instance on which the operation is to be executed. The instance identifier can either be `%tky` or `%cid_ref`. Both instance identifiers need derived types components such as `%is_draft` or `%pid` to uniquely identify an instance if more than one version can be available on the transactional buffer. Whereas `%tky` contains these components, they must accompany `%cid_ref` in referencing operations.

Static Operations

Implementation methods for static (non-instance bound) methods do not import an identifier of an instance. Instead, it is indispensable that they import the **content identifier** `%cid` as an identifier of the operation on a specific instance.

Input Identifiers

The identifying components, which are either instance identifier or content identifier, of the importing parameter are called **input identifiers** in the following.

The following tables give an overview of rules and guidelines for components of importing parameters.

Importing Parameter Components	Expected Provider Implementation
`%tky`	It is recommended to use the transactional key `%tky` for all implementation methods. It contains the actual key fields along with other components that are needed for unique identification of a certain entity instance depending on the scenario. In draft scenarios, the transactional key contains the draft indicator `%is_draft` to differentiate between draft instances and active instances on the buffer.
content identifier `%cid`	For operations that require a content identifier on consumption, `%cid` must always be reflected in the response structures, even if the `%cid` was replaced with an instance identifier during the interaction phase.
	An imported content identifier is always unique across all operations of one modify request. The RAP runtime engine checks this before calling the RAP handler method.
`%cid_ref`	The derived type component `%cid_ref` is used in operations that reference an instance that is identified with `%cid` in prior operations in the same LUW.
	The component `%cid_ref` always refers to a content identifier in the same request. In draft-enabled BOs, the component `%is_draft` must always correspond to the instance of the referenced content identifier. Only the combination of `%cid_ref` and `is_draft` can clearly identify an instance of a draft BO.
	For operations that use `%cid_ref` to identify a new instance, `%cid_ref` must always be reflected in the response structures, even if the `%cid_ref` was replaced with an instance identifier during the interaction phase by the RAP runtime engine. This is the case if creating operation and referencing operation are implemented in separate handler methods.
	In a handler method that returns `failed` for a referencing operation (one that uses `%cid_ref`), the implementation returns `%cid_ref` in `failed-%cid` and the determined `%tky` in `failed-%tky`. If the creating operation (one that uses `%cid`) and the referencing operation are implemented in separate handler methods, the determined `%tky` is imported into the referencing `%cid_ref`-operation by the RAP runtime engine. If the operations are implemented in one handler method, the implementation determines the `%tky` in the creating operation and sets it as the `failed-%tky` of the referencing operation.

Response Parameters

The RAP implementation methods have a common response pattern for which general rules apply. The provider makes use of these predefined and typed response structures to return information about the executed operation back to the consumers. The guidelines and rules on how to implement RAP handler and saver methods and how to fill these response parameters are subject to this documentation. The following table contains general provider information and rules for the common response parameters and their components.

Response Parameters

Response Parameter	Expected Provider Implementation
`failed`	The `failed` response parameter only contains a subset (0 - n) of input identifiers.
	The component `%fail` must contain the correct fail cause. The following fail causes are available: **Fail Cause** / **Usage** `not_found` — For instance operation handler methods if the requested instance does not exist. `locked` — For the lock handler method if the requested instance is already locked by another user. `unauthorized` — • Is set by the RAP runtime engine if an authorization handler method returns `unauthorized` in `result`. • In modify handler methods of unmanaged BOs if authorization is not implemented in authorization handler methods. `disabled` — • Is set by the RAP runtime engine if a feature control handler method returns `disabled` for operations in `result`. • In modify handler methods of unmanaged BOs if feature control is not implemented in feature control handler methods.

<table>
<tr><th>Response Parameter</th><th>Expected Provider Implementation</th></tr>
<tr><td></td><td>
<table>
<tr><th>Fail Cause</th><th>Usage</th></tr>
<tr><td><code>readonly</code></td><td>Is set by the RAP runtime engine if a feature control handler method returns <code>disabled</code> for fields in <code>result</code>.
In modify handler methods of unmanaged BOs if feature control is not implemented in feature control handler methods.</td></tr>
<tr><td><code>dependency</code></td><td>Is set by the RAP runtime engine for dependent operations with <code>%cid_ref</code> if the corresponding operation with <code>%cid</code> fails.
In the create-by-association handler method for the target instance if the CBA fails on the source instance.</td></tr>
<tr><td><code>conflict</code></td><td>Is only set by the RAP runtime engine.</td></tr>
<tr><td><code>unspecific</code></td><td>Any other failures.</td></tr>
</table>

For other fail causes than <code>not_found</code> or <code>locked</code>, the failed operation in <code>%op</code> must be provided.

Exception:

No operation is provided for
failed output instances in an instance factory action.
failed target instances in a create-by-association operation.
Example
If a create-by-association operation fails with fail cause <code>not_found</code> on the source instance, the target instances are listed in the target entity's <code>failed</code> parameter with fail cause <code>dependency</code>. But <code>%create</code> is only used for direct creates, and not for create-by-association.</td></tr>
<tr><td><code>mapped</code></td><td>The <code>mapped</code> response parameter only contains a subset (0 - n) of input identifiers.

The <code>mapped</code> response parameter does not contain duplicate entries.</td></tr>
</table>

Response Parameter	Expected Provider Implementation
	All input identifiers that are not part of `mapped` must be part of `failed` in the relevant implementation methods.
	If the `mapped` response parameter is not relevant, it stays empty. For example in instance actions.
`reported`	The `reported` response parameter can always be filled by the implementation to return messages.
	For severe fail causes the severity `error` is provided.
	For the following fail causes, the `reported` response parameter must be filled: • `unspecific` • `locked` • `unauthorized` • `disabled` • `readonly` • `dependency`
	The component %op must always be filled to indicate the operation the message relates to.
	The `reported` response parameter contains a subset (0 - n) of input identifiers. **Exception:** For entity instances of a lock dependent or an authorization dependent entity, the instance identifier provided in the reported structure is not the input instance identifier. In this case, the reported structure provides the instance identifier of every entity instance of the composition structure up to the root. **Example** If the fail cause of an operation on a child instance (lock dependent) is `locked`, the `reported` response parameter contains the key of the root instance (lock master).
	State messages in `reported` must always refer to an instance with an instance identifier. They cannot refer to a content identifier.

Response Parameter	Expected Provider Implementation
	State messages are not allowed in the response of the following behavior implementation methods: • read • functions • authorization control • feature control • lock • precheck • early numbering • any behavior implementation of newly defined behavior in the projection behavior definition.
`result`	The `result` response parameter always contains a subset of input identifiers (o..n).
	The `result` response parameter does not contain duplicate entries.
	All input identifiers that are not part of `result` must be part of `failed` (if result cardinality is greater or equal than 1).
	Input identifiers may only appear as often as the highest value of the result cardinality unless there are duplicates in the input. **Example** If result cardinality is 1 or `0..1`, one input instance identifier can only be listed once at most.
	The `result` response parameter must contain at least all requested fields. It may contain others.

Implementation Contract: READ

This topic explains the rules for implementing the read operation in a RAP business object.

The read operation is used to read data from the transactional buffer.

The implementation is done in the ABAP behavior pool in the method <method> FOR READ.

A BO consumer triggers the read operation with an ABAP EML read request by specifying the key value of the requested instance. The BO consumer expects the data of the requested instance to be returned by the read operation, if an instance with the given key exists. It is the BO provider's responsibility to return consistent and reliable results for any valid request. For more information, see READ.

Input

The following list defines which components are imported into the read handler method.

Obligatory

- Instance identifier: `%tky`

Output

The following table summarizes the provider rules for the read implementation.

Read Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful read operation.	Implementation returns exactly one row for each existing instance identifier in `result`. It does not return duplicate entries. At least the requested fields are returned, optionally more. At least the key fields of the entity are returned, even if not requested.	Existing instance identifiers. Duplicate existing instance identifiers.	`result`
Read operation as check.	Implementation returns exactly one row for each non-existing instance identifier in `failed`. It does not return duplicate entries or `result`.	Check if an instance for a respective key exists	`failed`
Unsuccessful read operation.	Implementation returns one row for each non-existing instance identifier in `failed`. It lists the instance identifier with the correct fail cause and initial %op. (Duplicates may appear.)	Non-existing (duplicate) instance identifiers. (Fail cause `not_found`.) Unauthorized read access. (Fail cause `unauthorized`.)	`failed`

Implementation Contract: READ-BY-ASSOCIATION

This topic explains the rules for implementing the read-by-association (RBA) operation in a RAP business object.

The read-by-association operation is used to read data from the transactional buffer, or from the CDS entity if the relevant entity is not yet present in the transactional buffer, by following an association from a source entity instance to all associated target instances.

The implementation is done in the ABAP behavior pool in the method <method> FOR READ.

A BO consumer triggers the RBA operation with an ABAP EML read request by specifying the key value of the source entity instance. The BO consumer expects that the read-by-association implementation returns the data of the associated instances, if a source instance with the given instance identifier exists. It is the BO provider's responsibility to return consistent and reliable results for any valid request. For more information, see READ BY Association.

Input

The following list defines which components are imported into the RBA handler method.

Obligatory

- Instance identifier: `%tky`

- Flag for full result

Output

The following table summarizes the provider rules for the RBA implementation.

Read-By-Association Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful RBA operation.	**If full result is requested:** Implementation returns exactly one row for each found target instance of an existing source instance identifier in `result`. It lists all existing target instances. It does not return duplicate entries. At least the requested fields of the target instances are returned, optionally more. For source and target instances that do not have any changes on the transactional buffer, the implementation returns exactly the same result as an ABAP SQL `select` on the corresponding CDS entities returns form the database. The result may only differ in BOs with suppressed or modify-enabled fields. (`field (suppress)` and `field (modify)`) **If link table is requested:** Implementation returns exactly one row for each combination of source and target instance. In every row it lists the source instance identifier with the associated target instance identifier.	Existing source instance identifiers. Duplicate existing source instance identifiers.	`result`
Unsuccessful RBA operation. Failure on source instance.	Implementation returns one row for each non-existing source instance identifiers in `failed`. It lists the source instance identifiers with the correct fail cause and initial `%op`. (Duplicates may appear.)	Non-existing (duplicate) instance identifiers. (Fail cause `not_found`.) `not_found`.)	`failed`

Implementation Contract: CREATE

This topic explains the rules for implementing the create operation in a RAP business object.

The create operation is used to create new entity instances of a RAP BO on the transactional buffer.

The implementation is done in the ABAP behavior pool in the method <method> FOR MODIFY.

A BO consumer triggers the create operation with an ABAP EML create request. The BO consumer expects that the keys of successfully created instances are returned by the create operation. For more information, see MODIFY.

Input

The following list defines which components are imported into the create handler method.

Obligatory

- Content identifier for output instances: `%cid`
- Any target field that is mandatory on create.

Optional

- Instance identifier for output instances: `%tky`
- Any target field that is not read-only.

Output

The following table summarizes the provider rules for the create implementation.

Create Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful create operation.	Implementation returns exactly one row for every created instance in `mapped`. It lists the content identifier with the corresponding mapped instance identifier of the new instance.	Distinct content identifiers. Non-existing instance identifiers for output instances.	`mapped`

Description	Expected Provider Implementation	Example	Further information
Unsuccessful create operation.	Implementation returns one row for each failed instance identifier in `failed`. It lists the content identifier with the correct fail cause and flags the `%create` component. For every input identifier listed in `failed`, there is one related error message returned in `reported`.	Existing instance identifiers for output instances. (Fail cause `unspecific`.) Duplicate instance identifiers: For duplicate instance identifiers with distinct content identifiers, the implementation can decide if all instances fail, or all except one. (Fail cause `unspecific`.) Unauthorized for create operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized`.) Feature-control disabled field/operation. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled` or `readonly`.)	`failed`

Implementation Contract: CREATE-BY-ASSOCIATION

This topic explains the rules for implementing the create-by-association operation in a RAP business object.

The create-by-association (CBA) operation is used to create new entity instances on the transactional buffer by following an association from a source entity in a RAP BO.

The implementation is done in the ABAP behavior pool in the method <method> FOR MODIFY.

A BO consumer triggers the create-by-association operation with an ABAP EML create request. The BO consumer expects that the keys of successfully created associated instances are returned by the CBA operation. For more information, see MODIFY.

Input

The following list defines which components are imported into CBA handler method.

Obligatory

- Source instance identifier: `source-%tky` and `source-%cid_ref`
- Target content identifier: `target-%cid`
- Any target field that is mandatory on create.

Optional

- Target instance identifier: `target-%tky`

- Any target field that is not read-only.

Output

The following table summarizes the provider rules for the CBA implementation.

Create-By-Association Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful CBA operation.	Implementation returns exactly one row for every created target instance in the target entity's `mapped` structure. It lists the content identifier of every created target instance mapping it to the target instance identifier.	Existing source instance identifiers. Non-existing target instance identifiers.	`mapped`
Unsuccessful CBA operation. Failure on source instance.	Implementation returns one row for each failed source input identifier in the source entity's `failed` structure. It fills the correct fail cause and flags the `%assoc` component on the correct association. Implementation returns one row for each target instance that was to be created in `failed` with fail cause `dependency`. For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing source instance identifier. (Fail cause `not_found` on source instance, fail cause `dependency` on target instances.) Unauthorized for CBA operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized`.) Feature-control disabled field/operation. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled` or `readonly`.)	`failed`

Description	Expected Provider Implementation	Example	Further information
Unsuccessful CBA operation. Failure on target instance.	Implementation returns one row for each target instance that was to be created in the `failed` structure of the target entity. It lists the target content identifiers with an adequate fail cause. For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Existing instance identifiers for target instances. (Fail cause `unspecific` `unspecifc`). (Duplicate target content identifier cannot reach implementation method. There will be a runtime error before.)Duplicate target instance identifiers: For duplicate target instance identifiers with distinct content identifier, the implementation can decide if all target instances fail, or all except one. (Fail cause Feature-control disabled field. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `readonly`.)	`failed`
Unsuccessful dependent CBA operation in the same handler. (CBA that uses `%cid_ref` in the same handler as the corresponding operation with `%cid`.)	The RAP runtime engine adds the entries for `failed` and `reported` for the target instances. The implementation must not fill them.	Duplicate target instance identifiers: For duplicate target instanceThe create and the CBA operation are implemented in the same handler and the create operation fails. The dependent operation CBA fails accordingly.	`failed`

Implementation Contract: UPDATE

This topic explains the rules for implementing the update operation in a RAP business object.

The update operation is used to change entity instances of a RAP BO on the transactional buffer.

The implementation is done in the ABAP behavior pool in the method <method> FOR MODIFY.

A BO consumer triggers the update operation with an ABAP EML update request. The BO consumer expects the update is successful or that the keys of unsuccessful updates are returned by the update operation implementation in the failed response parameters. For more information, see MODIFY UPDATE .

Input

The following list defines which components are imported into the update handler method.

Obligatory

- Instance identifier: `%tky` and `%cid_ref`

Optional

- Any field that is not read-only.

Output

The following table summarizes the provider rules for the update implementation.

Update Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful update operation.	Implementation changes data for instances on the transactional buffer.	Existing instance identifiers.	
Unsuccessful update operation.	Implementation returns one row for each failed instance identifier in `failed`. It lists the instance identifiers with the correct fail cause and flags the `%update` component. (Duplicates may appear.) For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing (duplicate) instance identifiers. (Fail cause `not_found`.) Unauthorized for update operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized`.) Feature-control disabled field/operation. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled` or `readonly`.)	`failed`

Implementation Contract: DELETE

This topic explains the rules for implementing the delete operation in a RAP business object.

The delete operation is used to delete entity instances of a RAP BO on the transactional buffer.

The implementation is done in the ABAP behavior pool in the method <method> FOR MODIFY.

A BO consumer triggers the update operation with an ABAP EML delete request. The BO consumer expects that the delete is successful or that the keys are returned by the delete operation implementation in the `failed` response parameter. For more information, see DELETE .

Input

The following list defines which components are imported into the delete handler method.

Obligatory

- Instance identifier: `%tky` and `%cid_ref`

Output

The following table summarizes the provider rules for the delete implementation.

Delete Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful delete operation.	Implementation deletes the requested entity instance on the transactional buffer.	Existing instance identifiers.	
Unsuccessful delete operation.	Implementation returns one row for each failed instance identifier in `failed`. It lists the instance identifiers with the correct fail cause and flags the `%delete` component. (Duplicates may appear.) For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing (duplicate) instance identifiers. (Fail cause `not_found`.) Unauthorized for delete operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized`.) Feature-control disabled field/operation. This is applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled` or `readonly`.) Delete operation on a draft instance. Draft instances can only be deleted via the draft action Discard. A delete on a draft instance fails with fail cause `disabled`.	`failed`

Implementation Contract: LOCK

This topic explains the rules for implementing the lock operation in a RAP business object.

The lock operation is used to lock entity instances of a RAP BO.

The implementation is done in the ABAP behavior pool in the method <method> FOR LOCK.

A BO consumer triggers the lock operation with an ABAP EML lock request. The BO consumer expects that the lock is successful or that the keys are returned by the lock operation implementation in the `failed` response parameter. For more information, see SET LOCKS.

Input

The following list defines which components are imported into the lock handler method.

Obligatory

- Instance identifier: %key

Output

The following table summarizes the provider rules for the lock implementation.

Lock Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful lock operation.	Implementation locks the requested entity instance.	Existing instance identifiers. Duplicate existing instance identifiers. They are merged by the RAP runtime engine before they reach the lock handler. Non-existing instance identifiers. (Fail cause `not_found` or `locked`.) The lock operation can, but does not have to check the existence of the input instances.	
Unsuccessful lock operation.	Implementation returns one row for each failed instance identifier in `failed`. It lists the instance identifier with the correct fail cause and initial `%op`. (Duplicates may appear.)	Non-existing instance identifiers. (Fail cause `not_found` or `locked`.) The lock operation can, but does not have to check the existence of the input instances. Instances are already locked by another user. (Fail cause `locked`.)	`locked`

Implementation Contract: Action

This topic explains the rules for implementing an action operation in a RAP business object.

The action operation is used to modify entity instances of a RAP BO on the transactional buffer based on custom logic.

The implementation is done in the ABAP behavior pool in a `method FOR MODIFY`, see Action Implementation.

A BO consumer triggers the action operation with an ABAP EML modify request executing an action. The BO consumer expects that the action is successful or that the input identifier is returned by the action operation implementation in the `failed` response parameter. For more information, see MODIFY ACTION.

Input

The following table defines which components are imported into the action handler method.

Input Components for Actions

Action Type	Obligatory Input Components
Instance non-factory action	Instance identifier: `%tky` and `%cid_ref`
Static non-factory action	Content identifier: `%cid`

Action Type	Obligatory Input Components
Instance factory action	Instance identifier: `%tky` and `%cid_ref` Content identifier: `%cid`
Static factory action	Content identifier: `%cid`

Output

The following table summarizes the provider rules for the action implementation.

Action Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful non-factory action.	If the action has a result parameter, the implementation fills the response parameter `result`. For every output parameter instance, it lists the input identifier and the calculated action result in `%param`. The parameter `mapped` must not be filled for non-factory actions.	Existing instance identifiers on instance actions. Distinct content identifiers of static actions.	`result`
Successful factory action.	Implementation returns exactly one row for every created instance in `mapped`. It lists the content identifier with the corresponding mapped instance identifier of the new instance.	Existing instance identifiers on instance actions. Distinct content identifiers.	`mapped`
Unsuccessful non-factory action.	Implementation returns one row for each failed input identifier in `failed`. It lists the input identifier with the correct fail cause and flags the `%action` component on the correct action. (Duplicates may appear) For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing instance identifier for instance actions. (Fail cause `not_found`) Unauthorized for action operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized`) Feature-control disabled field/operation. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled` or `readonly`).	`failed`

Description	Expected Provider Implementation	Example	Further information
Unsuccessful factory action. Failure on input instance. (Only applicable for instance factory actions.)	Implementation returns one row for each failed input identifier in `failed`. It lists the input identifier with the correct fail cause and flags the `%action` component on the correct action. It also lists the content identifier of the output instance in `failed` with fail cause `dependency`. The `%action` component is not flagged for the output instance, as the action belongs to the input instance. For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing instance identifier. (Fail cause `not_found` for input instance.) Unauthorized for action operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized` for input instance.) Feature-control disabled field/operation. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled` or `readonly` for input instance.)	`failed`
Unsuccessful factory action. Failure on output. (Applicable for instance and static factory actions.)	Implementation returns one row for the failed output instance in `failed`. It lists the content identifier with the correct fail cause. For static factory actions: Implementation flags the `%action` component on the correct action. (Instance factory actions do not flag the `%action` component on the output.) For every content identifier listed in `failed`, there is one related error message returned in `reported`.		`failed`

Implementation Contract: Function

This topic explains the rules for implementing a function operation in a RAP business object.

The function operation is used to return calculated information based on reading entity instances of a RAP BO on the transactional buffer.

The implementation is done in the ABAP behavior pool in a `method FOR READ`, see Function Implementation.

A BO consumer triggers the function operation with an ABAP EML read request executing a function. The BO consumer expects that the function is successful or that the instance identifier is returned by the function operation implementation in the `failed` response parameter.

For more information, see READ ENTITY, ENTITIES (ABAP Keyword Documentation) .

Input

The following table defines which components are imported into the function handler method.

Input Components for Functions

Function Type	Obligatory Input Components
Instance function	Instance identifier: `%tky`
Static function	Content identifier: `%cid`

Output

The following table summarizes the provider rules for the function implementation.

Function Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful function operation.	If the function has a result parameter, the implementation fills the response parameter `result`. For every output parameter instance, it lists the input identifier and the calculated function result in `%param`.	Existing instance identifiers on instance functions. Distinct content identifiers of static functions.	`result`
Unsuccessful function operation.	Implementation returns one row for each failed input identifier in `failed`. It lists the input identifier with the correct fail cause and flags the `%action` component on the correct function. (Duplicates may appear) For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing instance identifier for instance functions. (Fail cause `not_found`.) Unauthorized for function operation. This is only applicable if authorization is implemented in modify handlers in an unmanaged BO. (Fail cause `unauthorized`.) Feature-control disabled operation. This is only applicable if feature control is implemented in modify handlers in an unmanaged BO. (Fail cause `disabled`.)	`failed`

Implementation Contract: Feature Control

This topic explains the rules for implementing the RAP behavior methods for feature control.

Feature control is used to return permission values for operation, field, action and function control. Depending on the permission value a RAP consumer is allowed to execute an operation, action or function, or change a field on the transactional buffer of a RAP BO.

The implementation is done in the ABAP behavior pool in the `method FOR FEATURES` and `method FOR GLOBAL FEATURES`, see FOR INSTANCE FEATURES, FEATURES (ABAP Keyword Documentation) and FOR GLOBAL FEATURES (ABAP

Keyword Documentation).

A BO consumer triggers the feature control with an ABAP EML get permissions request. The BO consumer expects that the feature control implementation returns the requested permission values for existing instance identifiers.

For more information, see GET PERMISSIONS.

Input

The following list defines which components are imported into the feature control handler method.

Input Components for Feature Control

Feature Control Type	Obligatory Input Components
Instance feature control	Instance identifier: `%tky` Requested features
Global feature control	Requested features

Output

The following table summarizes the provider rules for the feature control implementation.

Feature Control Implementation Specific Rules

Description	Expected Provider Implementation	Example	Further information
Successful instance feature control.	Implementation returns exactly one row for each existing instance identifier in `result`. It lists the input identifier and the feature control permission values. At least the requested features are returned, optionally more.	Existing instance identifiers. Duplicate existing instance identifiers.	`result`
Successful global feature control.	Implementation returns the feature control permission values in `result`. At least the requested features are returned, optionally more.		`result`

Description	Expected Provider Implementation	Example	Further information
Unsuccessful feature control.	For instance feature control, the implementation returns one row for each failed instance identifier in `failed`. It lists the source instance identifiers with the correct fail cause and initial `%op`. (Duplicates may appear.) The global feature control implementation does not return `failed`. For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`.	Non-existing (duplicate) instance identifiers. (Fail cause `not_found`.)	`failed`

Implementation Contract: Authorization Control

This topic explains the rules for implementing the RAP behavior methods for authorization control.

Authorization control is implemented in RAP BOs to manage authorizations for executing a standard operation, an action or a function. The authorization can either be based on the status of a BO instance (instance authorization), or it can be instance-independent (global authorization).

The implementation is done in the ABAP behavior pool in the `method FOR INSTANCE AUTHORIZATION` and `method FOR GLOBAL AUTHORIZATION`, see Authorization Control.

A BO consumer triggers authorization control with an ABAP EML get permissions request. The BO consumer expects that the authorization handler methods return the requested permission values for existing instance identifiers in case of instance authorization, or permission values that are valid globally.

For more information, see GET PERMISSIONS.

Input

The following list defines which components are imported into the authorization control handler method.

Input Components for Authorization Control

Action Type	Obligatory Input Components
Instance authorization control	Instance identifier: `%tky` Requested features
Global authorization control	Requested features

Output

The following table summarizes the provider rules for the authorization control implementation.

Description	Expected Provider Implementation	Example	Further information
Successful instance authorization control.	Implementation returns exactly one row for each existing instance identifier in `result`. It lists the input identifier and the authorization control permission values. At least the requested features are returned, optionally more.	Existing instance identifiers. Duplicate existing instance identifiers.	`result`
Successful global authorization control.	Implementation returns the authorization control permission values in `result`. At least the requested features are returned, optionally more.		`result`
Unsuccessful authorization control.	For instance authorization control, the implementation returns one row for each failed instance identifier in `failed`. It lists the source instance identifiers with the correct fail cause and initial `%op`. (Duplicates may appear.) For every input identifier listed in `failed` with fail cause other than `not_found`, there is one related error message returned in `reported`. The global authorization control implementation does not return `failed`.	Non-existing (duplicate) instance identifiers. (Fail cause `not_found`.)	`failed`

Strict Mode

The strict mode ensures that a RAP business object is lifycycle-stable and upgrade-save.

About Strict Mode

ℹ Note

It is best practice to use `strict(2)` for all RAP BOs, except for:

- Business Configuration (BC) BOs
- BOs that consume a BC API or one of the following:
  - CL_EXCHANGE_RATES
  - CL_NUMBERRANGE_INTERVALS
  - CL_UOM_DIM_MAINTENANCE

- CL_UOM_MAINTENANCE
- XCO I18N APIs
- XCO Transport APIs

The strict mode represents the best practices regarding modeling and implementing a RAP business object. When a BO is defined as strict, this set of best-practice rules is technically enforced with additional syntax checks on specific design and implementation requirements to make sure that a BO is lifecycle-stable, upgrade-safe, and best-practice compliant.

Strict mode exists in different versions. It's recommended to always use the hightest version to ensure the strictest checks are applied to your RAP BO. The highest version always applies all checks of the previous versions and additional own checks.

When strict mode is defined, some syntax checks that lead to warnings without strict mode then lead to runtime errors.

The syntax checks in strict mode ensure that:

- the syntax in the behavior definition is up to date.
- all available operations are declared explicitly including draft actions.
- RAP best practices are enforced, like, for example, authorization implementation is mandatory.

Strict mode is currently mandatory and a prerequisite for BOs that are released with a release contract. For more information strict mode implementation requirements, refer to Strict Mode - Implementation Requirements for an overview.

Definition

You enable the strict mode with the addition `strict` directly after the implementation type in the behavior definition. The strict mode is available for the implementation types `managed`and `unmanaged`. For more information about `strict`, see CDS BDL - strict (ABAP Keyword Documentation).

Strict Mode - Implementation Requirements

Defining `strict` or `strict(2)` for your base or projection behavior definition results in specific implementation and modeling requirements that reflect the RAP best practices. If you define strict mode for an already existing RAP business object, your implementations or modeling may need to be adapted. To support the transition, the following list provides details about all implementation and modeling requirements that come along with `strict`or `strict(2)` mode:

- Strict Mode: Transactional Behavior Definition
- Strict Mode: Abstract Behavior Definition

The table columns offer the following information:

- **Requirement**: Indicates the best practice implementation expected by the `strict` mode
- **Implementation Type**: Indicates for which implementation type a corresponding requirement is valid.
- **Strict Mode Version**: Indicates whether an implementation requirement result from `strict` or `strict(2)`
- **Affects**: Indicates which RAP artifact is affected by errors once `strict` and additional syntax checks are active.
- **More Information**: Links to further information in the documentation (column can be added with **Switch/Hide Column**).

Transactional Behavior Definitions

Strict Mode Implementation: Transactional BDEFs

Requirement	Implementation Type	`Strict Mode` Version	Affects	More Information
Define `strict` for the base behavior definition, if you want to define `strict` in the projection behavior definition.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Projection Behavior Definition	
Define `strict(2)` for the base and projection behavior definition, if you want to release the BO for the `Extend (C0)` or ABAP `for Cloud Development (C1)` release contract.	• Managed • Unmanaged	• Strict mode version 2: `strict(2)`	• Base Behavior Implementation • Projection Behavior Implementation	• RAP Extensibility-Enablement
With `strict(2)`, %ASSOC is not part of the respective derived types, when child nodes define `authorization:update`.	• Managed • Unmanaged	• Strict mode version 2: `strict(2)`	• Base Behavior Implementation • Projection Behavior Implementation	
Define the root entity as lock master and all child nodes as lock dependent.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition Root • Base Behavior Definition Child Nodes	• Concurrency Control
Define the root entity as `authorization master` and all child nodes as `authorization dependent`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition Root • Base Behavior Definition Child Nodes	• Authorization Control
Define each BO node either as etag master or etag master/etag dependent unless a node is defined as `abstract`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition Root • Base Behavior Definition Child Nodes	• ETag Definition

Requirement	Implementation Type	`Strict Mode` Version	Affects	More Information
In `strict` mode, derived types and ready-results are only compatible with themselves in the implementation. **Note** Without `strict`, derived types and ready-results were implicitly compatible with structurally identical data types in some implementation scenarios. Now, the compatibility must be defined explicitly in the implementation with `CORRESPONDING`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Implementation • Projection Behavior Implementation	
Access the derived types for the following operation types only from provider side: • Authorizations • Features • Determinations • Validations • Additional save	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Methods in the RAP behavior pool: • `FOR FEATURES` • `FOR INSTANCE AUTHORIZATION` • Determinations • Validations	• Derived Data Types
Build a projection behavior definition on a BDEF with implementation type managed or unmanaged. **Note** Projection BDEFs can't be based on abstract base behavior definitions.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Projection Behavior Definition Nodes	
Create child instances only with explicitly stated `Creates-By-Association`. A direct `CREATE` on a subnode isn't allowed in most cases.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition Child Nodes	• Create by Association Operation Runtime
Implement the `ADJUST_NUMBERS` method in the behavior pool, if you've defined `late numbering`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition Nodes	• ADJUST_NUMBERS

Requirement	Implementation Type	`Strict Mode` Version	Affects	More Information
Explicitly enable all expected behavior in the behavior definition. **Note** `Strict` mode enforces explicit definition of all expected behavior and any implicit enablement is suppressed, like, for example, implicit read-enablement in toParent compositions or implicit enablement of CUD operations for a managed RAP business object.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition Nodes • Projection Behavior Definition	
Use only derived types from explicitly enabled operations in your implementation.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Implementation • Projection Behavior Implementation	
Use the newest syntax in your behavior definition and implementation. **Note** `Strict` mode result in syntax errors if deprecated syntax is used in the behavior definition or implementation.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition and Implementation • Projection Behavior Definition and Implementation	For EML: • Examples for Accessing Business Objects with EML
Only use EML invocations for explicitly enabled standard operations. **Note** You can only implement an EML delete for a RAP BO, if the DELETE is enabled in the corresponding behavior definition.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition and Implementation • Projection Behavior Definition and Implementation	
Ensure that feature control isn't used in combination with `readonly`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition • Projection Behavior Definition	

Requirement	Implementation Type	`Strict Mode` Version	Affects	More Information
Explicitly define all draft actions if your BO is draft-enabled: • draft action resume; • draft action Edit; • draft action Activate; • draft action Discard; • draft determine action Prepare { }	• Managed (only if draft-enabled) • Unmanaged (only if draft-enabled)	• Strict mode version 1: `strict`	• Base Behavior Definition	• Draft Actions
Explicitly display all draft actions in your projection BDEF, if the projection is defined as `strict`: • use action Resume; • use action Edit; • use action Activate; • use action Discard; • use action Prepare;	• Managed (only if draft-enabled) • Unmanaged (only if draft-enabled)	• Strict mode version 1: `strict`	• Projection Behavior Definition	• Draft Actions
Define the draft determine action `Prepare` without authorization control.	• Managed (only if draft-enabled) • Unmanaged (only if draft-enabled)	• Strict mode version 1: `strict`	• Base Behavior Definition	
Define functions as `FOR FUNCTION` in the method definition of your behavior implementation. **Note** The definition `FOR ACTION` for a function results in a syntax error.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Implementation • Projection Behavior Implementation	
Rename actions in the BDEF projection with the addition `as *ActionName*`, if another action has the same identifier.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Projection Behavior Definition	
Use view entities as data sources for a RAP business objects. CDS views aren't supported in `strict`mode.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Projection Behavior Definition	

Requirement	Implementation Type	`Strict Mode` Version	Affects	More Information
Use `late numbering` instead of `late numbering in place`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition and Implementation	
Don't use the `MAPPED` parameter in your `READ/LOCK` implementation and don't use `IMAGE` in your `READ` implementation .	• Unmanaged	• Strict mode version 1: `strict`	• Base/Projection Behavior Implementation	
Define EML calls of static actions or functions explicitly as `in local mode` if necessary.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Implementation • Projection Behavior Implementation	
Ensure that toParent/toChild associations aren't defined as `internal` and are explicitly defined in the behavior definition.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base/Projection Behavior Definition	
Only use CDS abstract entities or classic DDIC types in your action or function implementation.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base/Projection Behavior Implementation	
Ensure that your action or function implementation has either `$self`, `entity`, CDS abstract entities, or a DDIC type as returning parameter.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base/Projection Behavior Implementation	
Define an `implementation in class *ImplementationClass* unique` for each BO node where an implementation is required.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition • Projection Behavior Definition	
Ensure that your composition hierarchy is modeled consistently and without any gaps: That means that parent entities must always be defined in the behavior definition if their child entity is defined. However, you can define a parent entity without its dependent entities.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition • Projection Behavior Definition	

Requirement	Implementation Type	`Strict Mode` Version	Affects	More Information
Ensure that you define the associations in the behavior definition if the respective entity is defined in the behavior definition: **Example** In a composition with a `travel` entity as root node and a `booking` entity as child entity the following applies: If `booking` is defined with `Define behavior for *booking_entity*`, the `travel` entity must contain `association _Booking {...}`.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition • Projection Behavior Definition	
If you've defined `late numbering` for a parent entity, it recommended use late numbering also for its child entit'ses. However, the `%PID` isn't inherited implicitly by child entities in `strict` mode.	• Managed • Unmanaged	• Strict mode version 1: `strict`	• Base Behavior Definition	

Abstract Behavior Definition

Strict Mode Implementation: Abstract BDEFs

Requirement	Implementation Type	Affects	More Information
Use abstract CDS entities as data sources , if you want to define an abstract behavior definition.	• Abstract	• (Abstract) Base Behavior Definition	

Requirement	Implementation Type	Affects	More Information
Define an abstract behavior definition without any transactional behavior like: • Feature Control • Standard or Non-Standard Operations • Locks • Etag handling • Foreign Entity • ... Only define behavior definition elements such as associations and compositions , or mappings for an abstract BDEF.	• Abstract	• (Abstract) Base Behavior Definition	

Related Information

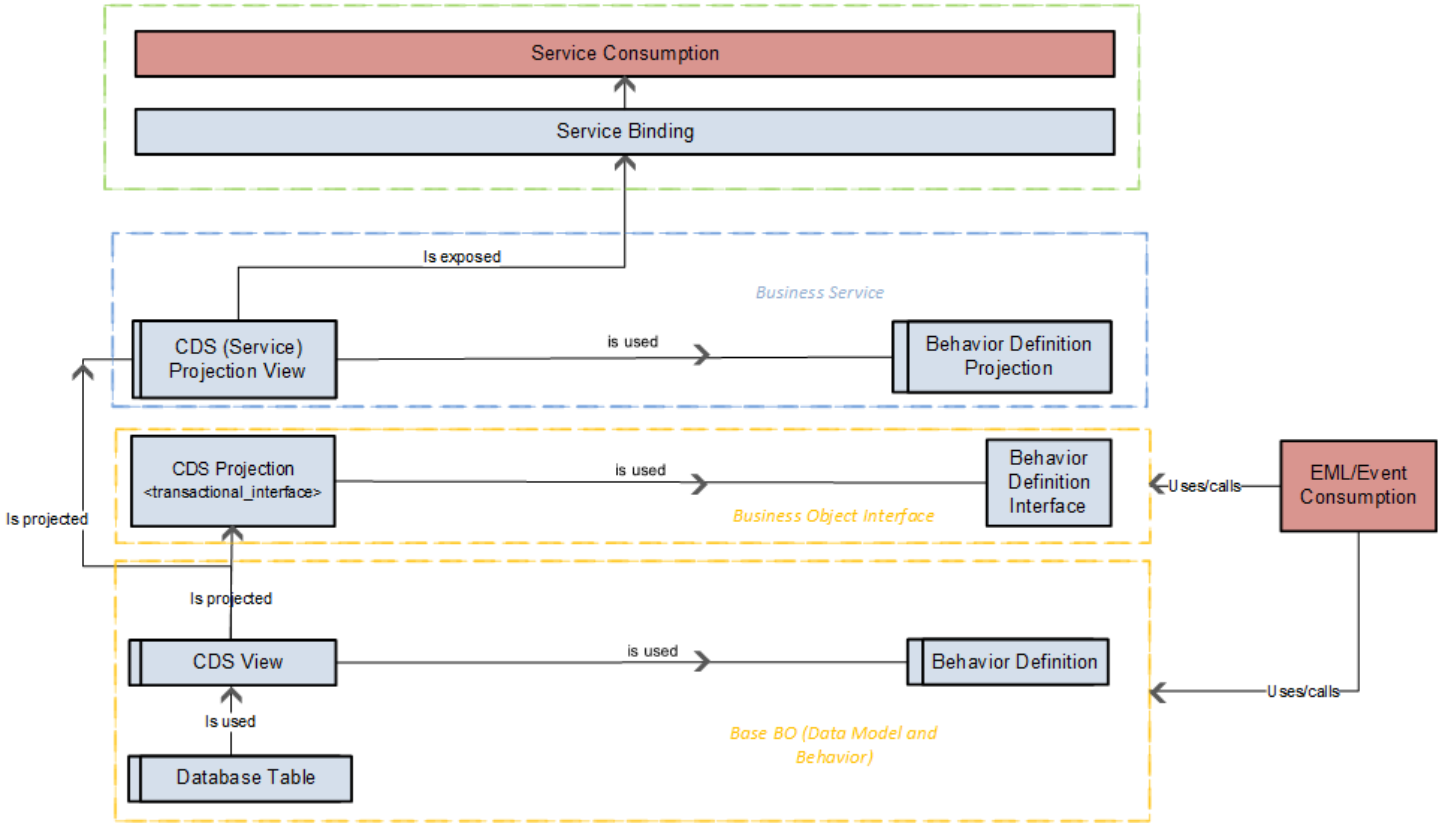
Strict Mode

Business Object Interface

A business object interface is required if you want to release your business object for e.g. usage/extensibility in other software components via stability contracts. Business object interfaces are not required if no cross component access is intended. For more information on stability contracts for business object interfaces, see What are RAP Business Object Interfaces?.

From a technical perspective, a BO interface layer consists of a CDS projection view with the provider contract `transactional_interface` and a `Behavior Definition Interface` with the keyword `interface` in its header. A RAP BO interface is a specific BDEF type without a runtime handler and thus without implementation class and behavior pool, that specifies a subset of elements from a RAP base BO.

The interface layer is built on top of the base BO layer as single point of access to the base BO:



Each RAP base BO can have multiple interfaces ([0,*]) to comply with different consumer requirements and use cases, for example one BO interface that exposes draft capabilities and another interface with the same RAP base BO that doesn't expose the draft capabilities and only allows access to active instances.

For more information about the syntax, refer to CDS DDL - DEFINE VIEW ENTITY AS PROJECTION ON, Transactional Interface and for more information about the provider contract, refer to CDS DDL - PROVIDER CONTRACT transactional_interface (ABAP Keyword Documentation).

What are RAP Business Object Interfaces?

A RAP `Business Object Interface` is a projection layer that technically decouples requirements for stable consumption from the behavior and data model of the underlying RAP base BO. The interface layer distinctly specifies a subset of elements or behaviors from a RAP base BO like fields or actions. The interface ensures lifecycle-stable consumption for RAP BO consumers : To ensure lifecycle-stability, development objects are released for specific API states, e.g. for `Use System-Internally (C1)`. Released development objects must abide by specific stability criteria, change, and development restrictions to guarantee lifecycle stability and upgrade-safety. With an interface layer, you can avoid unnecessary development restrictions on the RAP base BOs. When the BO interface is consequently released for `Use System-Internally (C1)`, e.g., for extensions, the restrictions resulting from the API state only apply to elements specified for consumption in the interface layer. Other elements that are only contained in the RAP base BO remain unaffected and can be changed without influencing the runtime of applications that consume the base BO via the RAP BO interface.

Since a RAP BO interface only acts as a consolidated consumption view for a RAP base BO, it doesn't have its own runtime handler. So, all incoming requests are delegated to the underlying RAP base BO and its respective behavior pool and implementation.

You can only newly define or redefine static feature control on the interface layer for elements from the RAP base BO. For example, you can define a fields as `readonly` in a BO interface when the same field doesn't have any static feature control in the underlying base BO. Other than static feature control, it's not possible to introduce new behavior or change existing behavior on interface level.

For an example about how to implement RAP BO interfaces, refer to Develop APIs.

An overview of the consumption possibilites of business objects and their interfaces can be found in chapter Consume.

Releasing RAP BO Interfaces as APIs

To make a RAP BO interface available to consumers in other software components, you can release the interface using the release contract `Use System-Internally (C1)` and visibility `Use in Cloud Development`.

The layer beneath a released SAP interface can't be accessed or consumed directly, but must always be accessed via the released interface layer of the RAP base BO.

BO Interfaces and Extensions

The BO interface layer is the basis for data model and behavior extensions. The BO interface is implicitly extended with every behavior extension that applies the respective BO interface.

Behavior Extensions and BO Interfaces

A BDEF extension is always based on a `Business Object Interface` (BO Interface) through which the implementation of the original RAP BO is accessed. A BDEF extension to the original BO technically extends the base BDEF, but is consumed via the BO interface during the runtime.

For more information about extensions, see Extend.

Annotation propagation

The interface projection layer is used to decouple requirements for the stable consumption of a business object from its underlying base data model and behavior. With respect to annotations, this means that annotation changes in the base data model must have no impact on the data model in the interface projection layer and above. Hence it is best practice to use the annotation `@Metadata.ignorePropagatedAnnotations: true` in the header section of the interface projection view. This annotation prevents annotations from the underlying base data model from being propagated further through the CDS view stack, thus possibly impacting projections that are built on top of the RAP BO interface. The annotations that are supposed to be effective in superjacent projections should be defined explicitly in the interface projection view.

Business Service

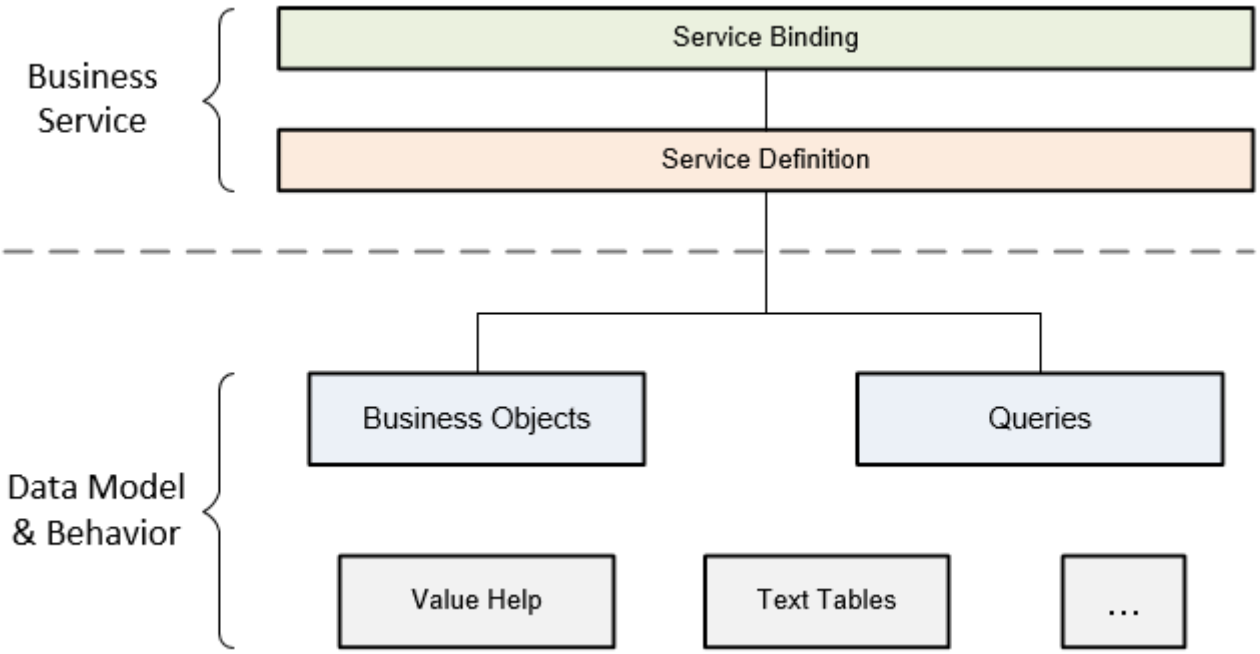
Definition

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

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

Business Services in the ABAP RESTful Application Programming Model

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



Business Service

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

Separation Between the Service Definition and the Service Binding

In a SAP Fiori UI, many role-based and task-oriented apps are based on the same data and related functionality must be created to support end users in their daily business and in their dedicated roles. This is implemented by reusable data and behavior models, where the data model and the related behavior is projected in a service-specific way. The **service definition** is a projection of the data model and the related behavior to be exposed, whereas the **service binding** defines a specific communication protocol, such as OData V2 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 hassle of re-implementation.

Example

Let us assume that a business object SalesOrder is defined and implemented in the data model and the behavior layer with the related value help and authorization management. The service definition might expose the SalesOrder and several additional business objects such as the Product and the BusinessPartner as they are included in a service binding for an OData V2 service.

The service requires the following artifacts:

- The service definition and the related projection views that project the service relevant parts of the data model implemented in CDS and the behavior definition where it projects the operations that should be exposed. For example, the SalesOrder BO might offer the operations: create, update, delete, and 10 different application-specific actions. However, for a concrete role-specific list report, only two actions are required, so the remaining 8 actions and three standard operations are not included in the service projection.
- If the service is used to create a user interface, additional UI semantics are required. These are implemented by CDS UI annotations that are regularly stored in CDS metadata extensions (MDEs).
- The service binding that uses the package of artifacts that is defined in the service definition to bind the package to a service type (Web API, UI service, INA service) and a protocol type (OData V2, OData V4).

Related Information

Service Definition

Service Binding

Working with Business Services

Business Object Projection

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

Introduction

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

Why Using Projections?

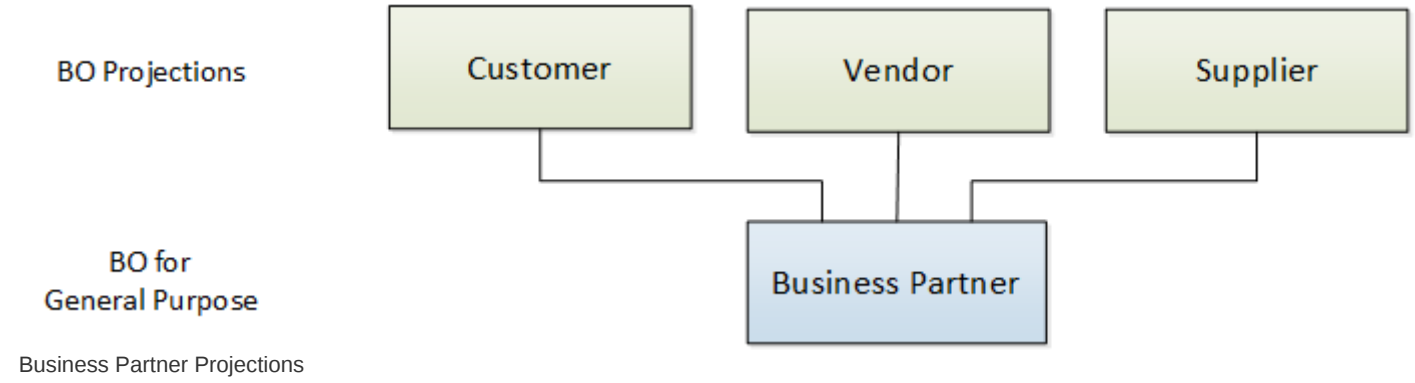
By using a projection layer for your business object, you gain flexibility in the service consumption. The general business object can be extended without affecting the already existing business service. This layering with projections enables robust application programming. The projection layer exposes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

The projection layer also enables one business object to be exposed in an OData service for a Fiori UI and for a stable Web API. The service-specific differences can then be implemented in the respective projection layers. For example, UI specifications are defined only in the BO projection that is exposed for the UI service. Furthermore, with projections, you cannot only define the type of the service, but you can also design role-based services. One business object for general purpose is exposed for more than one context-specific projection as specialized business object. The most prominent example is the business partner BO, which is exposed as customer, vendor, or supplier. In the projection, you can use that subset of the business partner BO that is relevant for the respective specialization.

Example

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

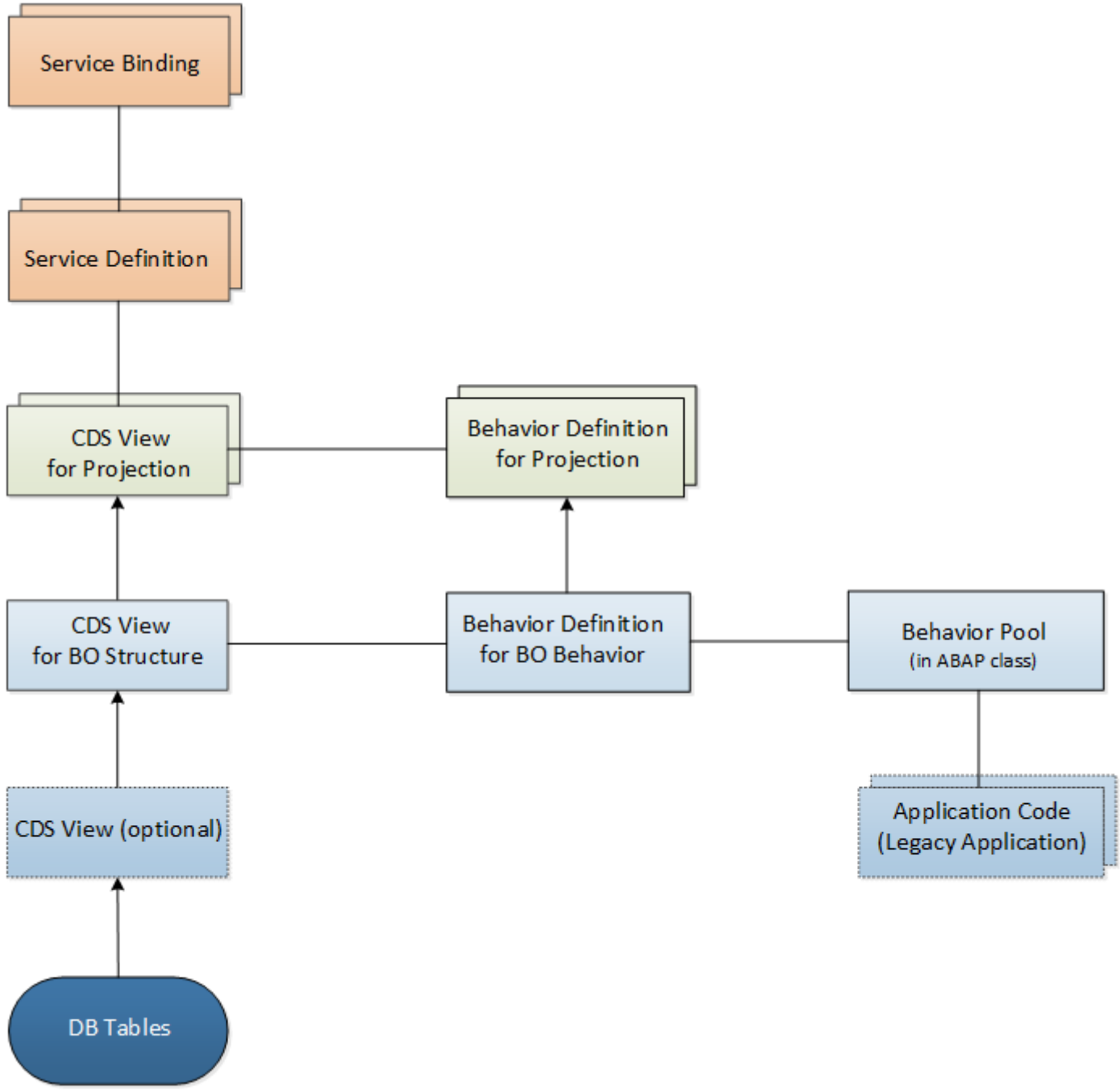
Imagine the business partner is enriched with characteristics for a new role of a business partner, for example a supplier. You can add the necessary additional elements, for example delivery information, to the data model and the behavior implementation in the business partner BO without affecting the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



Business Partner Projections

How to Use BO Projections?

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



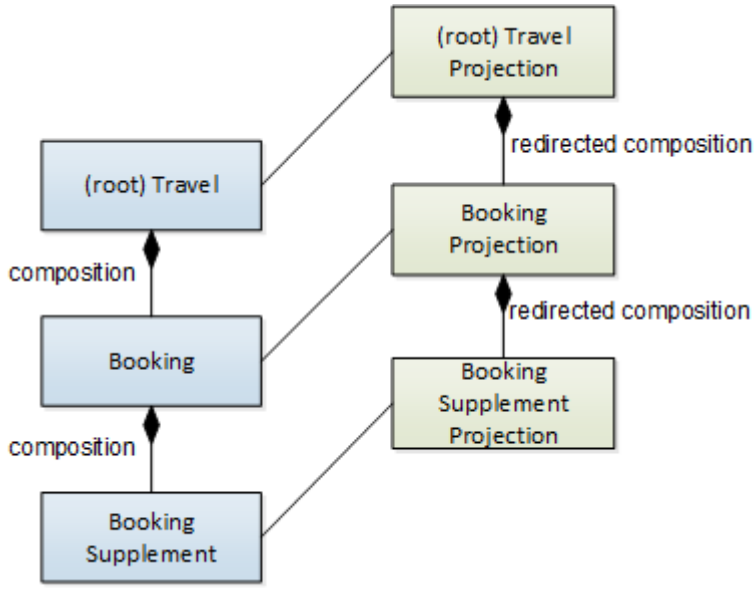
Involved Development Objects

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

- **CDS Projection Views**

  The projection of the data model is done in one or more CDS projection views, depending on the number of nodes of the underlying BO. The CDS projection views use the syntax element `as projection on <ProjectedEntity>` to mark the relationship to the underlying projected entity. As opposed to the former consumption views, they do not create another SQL view. Since they only provide the consumption representation of the projected entity, they do not need an ABAP Dictionary representation.

  If one BO entity is projected, the root and all parent entities must be projected as well. The root entity has to stay the root entity and must be defined as root projection view. The compositions are redirected to the new target projection entity.



Travel BO Projection

  For a detailed description on CDS projection views and their syntax, see CDS Projection View.

- **Projection Behavior Definition**

  The projection of the behavior is done in a behavior definition of type `projection`, which is declared in the header of the behavior definition. According to this type, only syntactical elements for projections can be used.

  For more information on projection behavior definitions and their syntax, see Projection Behavior Definition.

CDS Projection View

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

CDS projection views are defined in data definition development objects. The wizard for data definitions provides a template for projection views.

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

You can add new read-only associations in the projection view. This can be relevant to display additional information on the UI, like charts etc. It is not possible, however, to denormalize fields from new associated entity in the projection view. New associated entities cannot be accessed transactionally. Associations, including compositions, that are defined in the projected

CDS view can be used in the projection CDS view. However, associations or compositions might change their target, if the target CDS view is also projected. This is especially relevant for compositions as the complete Data Model is projected and therefore the composition target changes. In case of a changed target, the association or composition must be redirected to the new target. The projection view comes with a new syntax element to express the target change.

For more details about the projection view syntax, see Define View Entity as Projection (ABAP Keyword Documentation).

Annotation Propagation to Projection Views

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

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

Example

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

```
define root view /DMO/I_Travel
...

{
  key travel_id,
...
    @Semantics.amount.currencyCode: 'currency_code'
    total_price,
    @Semantics.currencyCode: true
    currency_code,
...}
```

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

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

```
define root view entity /DMO/C_Travel as projection on /DMO/I_Travel
...

{
  key travel_id,
...
      @Semantics.amount.currencyCode: 'CurrencyCode'
      total_price        as TotalPrice,

      currency_code      as CurrencyCode,
...}
```

Defining UI Specifics in the Projection Views

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

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

! Restriction

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

Related Information

Providing a Data Model for Projections

Projection Behavior Definition

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

The behavior definition with type `projection` is created equally to other types of behavior definitions. When creating a behavior definition based on a CDS projection view, the syntax template directly uses the projection type.

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

For details about the syntax of a projection behavior definition, see CDS BDL - implementation type (ABAP Keyword Documentation).

Explanation

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

- ETag
- standard operations
- actions
- functions
- create_by_association

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

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

- locking
- authorization
- feature Control

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

Related Information

Providing Behavior for Projections

Service Definition

Definition

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

Use

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

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

→ Remember

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

ⓘ Note

You cannot expose an OData service that includes abstract entities for Web APIs or UI services. You can expose abstract entities in a service definition, but only if they are created using OData client proxy tools (service consumption use case). The publishing of a service via a service binding causes a dump error.

For details about the syntax to define a service, see CDS SDL - DEFINE SERVICE (ABAP Keyword Documentation)

Explanation

The source code of the actual service definition is preceded by the optional CDS annotation `@EndUserText.label` that is available for all objects which can contain CDS annotations. The annotation value is a character string with a maximum of 60 characters. The specified `text` value should consist of a meaningful short text that describes the service in the original language of the source code.

Depending on the needs of your scenario, further optional annotations `@<Annotation_1>` ... `@<Annotation_n>` can be specified. If you expose a complete business object structure with more than one CDS entity for a service you can use the annotation `@ObjectModel.leadingEntity.name:` to define the root entity of the business service and hence the primary entrance point to the business service. The annotation provokes that the entity is marked in the entity tree in the service binding editor and appears as the first entity in the list.

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

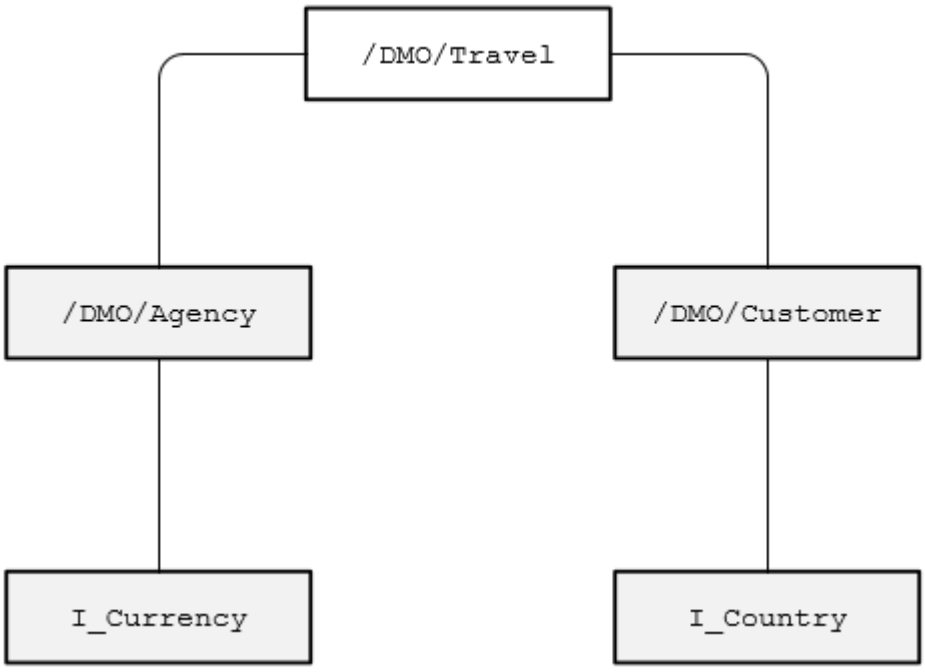
The source code of a service definition is created within a single bracket { ... } that is used to group all the related CDS entities (including their associations with the relevant entities) which are to be exposed as part of an individual service.

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

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

Example

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



Data model for the TRAVEL service

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

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

Note

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

Related Information

Service Binding
Creating Service Definitions

Service Definition Extension

A RAP `Service Definition Extension` extends an existing service definition to include extension nodes in the service binding.

A `Service Definition Extension` contains extension nodes you want to expose in your service binding. If you add a new extension BO node to a RAP BO, you must extend an existing `Service Definition` to expose the new BO entity to the consumer in the service binding. For more information about creating an extension node, see Node Extensions.

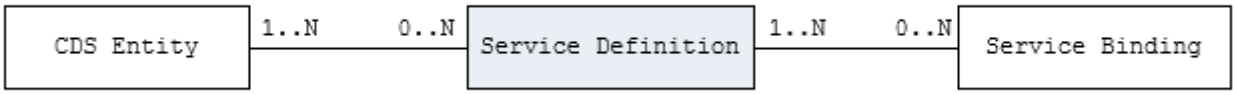
Service Binding

Definition

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

Use

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



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

Parameters

The following parameters are used to characterize a service binding:

Service Name

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

Tip

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

Binding Type

The binding type specifies the service type and the specific protocol which is implemented with the service binding.

The CDS data models can be exposed with the following protocols:

- **OData version 2.0** (ODATA V2)

  For more information, see https://www.odata.org/documentation/odata-version-2-0/ .

  For more information about Fiori Elements apps based on OData V2 Models, see OData V2 Model.

  > **Note**
  > In OData V2 services, unit and currency types are automatically added to the service metadata as built-in entity sets. With this feature, unit and currency values are always displayed correctly and validated regarding decimals on a UI.

- **OData version 4.0** (ODATA V4)

  For more information, see https://www.odata.org/documentation/ .

  OData V4 services have a wider scope than OData V2 services. Use OData V4 wherever possible for transactional services.

  > **Note**
  > Full support for Fiori Elements UIs based on OData V4 services is only granted for draft-enabled scenarios.

  For more information about Fiori Elements app based on OData V4 Models, OData V4 Model.

- **InA** (Information Access)

  Analytical data models are exposed with InA for live data access.

- **SQL**

  Access published ABAP-managed database API objects with open SQL.

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

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

`UI`	**UI service** A UI service makes it possible to add a SAP Fiori elements UI or other UI clients to the service. Currently, UI services are supported for OData and InA services.
`Web API`	**Service exposed as Web API** A service that is exposed as Web API is used for all other use cases apart from UIs. Web APIs can be consumed by an unknown consumer via OData. Web APIs can be extended. Currently, Web APIs are supported for OData and SQL services.

Service Version

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

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

Publishing

The **local service endpoint** of an OData service must be published via 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.

Service URL

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

Preview

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.

OData Exposure Comparison

Business services bound to the OData protocol differ depending on the type and the version chosen respectively in the service binding. The following table outlines differences between these types and versions:

Feature	WebAPI V2	WebAPI V4	UI Service V2	UI Service V4
UI annotations	No UI specific information contained in metadata	No UI specific information contained in metadata	UI specific information contained in metadata	UI specific information contained in metadata
Value help	Not available		Available	
Side Effetcs	Not available		Available	
Feature control	Not exposed as part of service		Exposed as part of service	
PDF Export	Not available		Available via entity sets	
Release for contracts	C2 possible in **ADT** > **Service Binding** > **context menu** > **API state**. See also: Releasing Development Objects		C1 possible in **ADT** > **Service Binding** > **context menu** > **API state**. See also: Releasing Development Objects	
Fiori Elements preview	Not available		Available in service binding	
Code lists for units of measure / currencies	Not available	Contained in service group	Contained in entity sets `SAP__Currency` and `SAP__UnitOfMeasure`	Contained in service group

Feature	WebAPI V2	WebAPI V4	UI Service V2	UI Service V4
ISO conversion of currency code format	Available		Not available	
ISO conversion of unit of measure	Conversion disabled for SAP Unit	ISO / SAP Unit validation	Conversion disabled for SAP Unit	ISO / SAP Unit validation
Conversion Exit CUNIT	Removed for the ISO unit of measure property		Available	

More information as well as examples for the business service exposure of business objects is available in the section Develop.

Service Transport

In an on-premise system, you can only use the service binding to enable your service locally and test it in the development system. Transporting the service binding with all the related artifacts that are generated locally is not possible. To enable your business service in a qualification or productive system, you have to disable the service in your local system (**Unpublish** button). Then use Gateway tools to activate and maintain your service.

For more information, see Activate the Service (SAP Gateway Foundation) for OData V2 scenarios or Service Development in Hub System (V4) for OData V4 scenarios.

Service Errors

The service binding shows errors that are related to the complete service. They are shown on creating the service binding or when reopening the artifact in ADT. The errors that are shown in the service binding are also appear in ATC checks.

Related Information

Service Definition

RAP Reuse Data Elements

You can use specific data elements in all RAP application for typing administrative fields.

The following list of reuse data elements can be used for the administrative fields in RAP BOs of all implementation types. These data elements are available for reuse in all systems.

Data Element	Definition
`ABP_CREATION_USER`	Identifier of the user who created the data.
`ABP_CREATION_DATE`	Date on which the data was created.
`ABP_CREATION_TIME`	Time at which the data was created.
`ABP_CREATION_TSTMPL`	Timestamp at which the data was created.
`ABP_CREATION_UTCL`	Timestamp at which the data was created.
`ABP_LASTCHANGE_USER`	Identifier of the user who changed the local instance itself or its compositional subtree last time.
`ABP_LASTCHANGE_DATE`	Date on which the data of the local instance or its compositional subtree was last changed.

Data Element	Definition
ABP_LASTCHANGE_TIME	Time at which the data of the local instance or its compositional subtree was last changed.
ABP_LASTCHANGE_TSTMPL	Timestamp at which the data of the local instance or its compositional subtree was last changed.
ABP_LASTCHANGE_UTCL	Timestamp at which the data of the local instance or its compositional subtree was last changed.
ABP_LOCINST_LASTCHANGE_USER	Identifier of the user who changed the local instance itself (without considering its compositional subtree) last time.
ABP_LOCINST_LASTCHANGE_DATE	Date on which the data of the local instance itself (without considering its compositional subtree) was last changed.
ABP_LOCINST_LASTCHANGE_TIME	Time at which the data of the local instance itself (without considering its compositional subtree) was last changed.
ABP_LOCINST_LASTCHANGE_TSTMPL	Timestamp at which the data of the local instance itself (without considering its compositional subtree) was last changed.
ABP_LOCINST_LASTCHANGE_UTCL	Timestamp at which the data of the local instance itself (without considering its compositional subtree) was last changed.

Administrative data fields are relevant in RAP application to log data access and modification. This is especially important when working with an OData ETag for optimistic concurrency control, or for the total ETag in draft applications.

For more information about the OData ETag, see Optimistic Concurrency Control.

For more information about the total ETag, see Total ETag.

Managed Business Objects

In managed business objects, the relevant administrative fields are updated automatically by the managed BO provider if the fields are annotated with the corresponding annotation in CDS. See Semantics Annotations.

Unmanaged Business Objects

In unmanaged business object, updating the administrative fields on data modification must be implemented by the unmanaged BO provider in all relevant RAP handler methods.

Example

When creating a new instance, the RAP handler method for create must ensure that the fields for the creation user and the creation time are filled correctly.

Using Groups in Large Development Projects

This section introduces the concept of groups that can be used to divide operations, actions and other implementation-relevant parts of the business logic into several groups for behavior implementation.

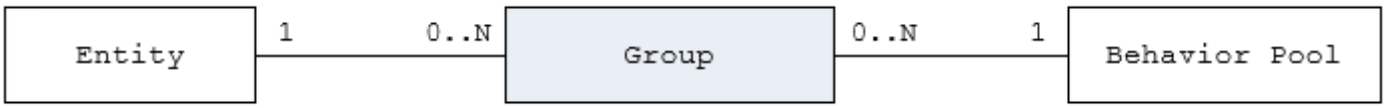
Use Case

Generally, the implementation of business object entity's operations and actions is done in the `Local Types` include of the behavior pool (ABAP class that implements business object's behavior) associated with that entity – unless the control of the implementation classes using the `IMPLEMENTATION IN CLASS` syntax (at entity level) has been completely dispensed.

Since the **Local Types** include can only be changed by one developer at a time, the efficiency of development would be significantly reduced in case of larger implementations. Let us think about large business objects with extensive business logic implementations, with many entities, each of which may contain a variety of elements.

As a solution, the operations and actions of an entity can be divided into several groups, whereby each of these groups can then be assigned a different behavior pool as implementation class.

Especially in large development projects, the responsibilities for the implementation of application logic are assigned to different members of a development team. To support this approach technically, we introduce the concept of groups. This approach enables that a team of developers can implement parts of business logic independently from each other. In addition, the group concept allows to tailor the functionality of business objects according to semantic considerations.



Relationship Between Entities, Groups and Implementing Classes

For details about the syntax for defining groups for unmanaged business objects, see CDS BDL - implementation grouping (ABAP Keyword Documentation)

Syntax: Defining Groups for Managed Business Objects

Groups can be defined within a behavior definition for a business object of type managed by using the following syntax:

```
[implementation] managed [implementation in class ABAP_ClASS [unique]];
define behavior for Entity [alias AliasedName]
  persistent table DB_TABLE
  ...
{
  [create;]
  [update;]
  [delete;]
  [read;]
  [association AssociationName [abbreviation AbbreviationName] {[create;] } ]


  group groupName_1 implementation in class ABAP_CLASS_1 unique
  {
    // Implementation-relevant content of the entity
  }

  group groupName_2 implementation in class ABAP_CLASS_2 unique
  {
    // Implementation-relevant content of the entity
  }

  // It is possible to assign the same behavior pool as the implementation class in different group
  group groupName_3 implementation in class ABAP_CLASS_1 unique
  {
    // Implementation-relevant content of the entity
  }

  group ...

}
```

Explanatory Notes

(1) The group name is defined locally within the entity's behavior definition and must not conflict with actions, determinations, or validations of the same name.

(2) The `implementation in class` syntax can only be used on groups, but no longer on individual entity's behavior definition itself. A group requires a behavior pool ABAP_CLASS_* and the addition of unique.

> **Note**
> With the addition `implementation in class` ABAP_ClASS in the header of the behavior definition, you have the option to implement the remaining functionality for all entities in a common behavior pool. For example, the save sequence for all entities of a business object could be implemented in a single behavior pool ABAP_CLASS.

It is possible to specify the same behavior pool as the implementation class in different groups. In the syntax above, the implementation-relevant content of `groupName_1` and of `groupName_3` must be implemented in `ABAP_CLASS_1`.

(3) The implementation-relevant content of an entity can be:

- Actions
- Instance-based feature control
- Determinations - for managed implementation type only
- Validations - for managed implementation type only.

> **Note**
> In the case of **unmanaged** implementation type, the standard operations (`CREATE, UPDATE, DELETE`) as well as READ and CREATE by association must be assigned to a group. In the **managed** case however, the standard operations (that are implemented by the framework) and READ and CREATE for associations can be specified either inside or outside groups.

(4) Information on mapping (which is never relevant to implementation) must always be specified outside of groups.

(5) Implicit standard operations (defined automatically and don't have to be explicitly specified) must be explicitly specified within one and the same group in the unmanaged case (where they are implementation-relevant):

- `read;` - for READ operations of an entity
- `lock;` - for LOCK operations of an entity that is defined as lock master.

Examples

Listing: Groups in the behavior definition of an unmanaged business object

```
implementation unmanaged implementation in class /DMO/BP_TRAVEL_U;

// behavior definition for the TRAVEL root entity
define behavior for /DMO/I_Travel_U alias travel
  etag LastChangedAt
  lock master

{
  group travel_cud implementation in class /dmo/bp_travel_cud unique
  {
    field ( read only ) TravelID;
    field ( mandatory ) AgencyID, CustomerID, BeginDate, EndDate;
    field(features:instance)   overall_status;

    create;
    update(features:instance);
```

```
    delete;

    read; // read and lock must be assigned explcitly to a group
    lock;

  }

  group travel_cba implementation in class /dmo/bp_travel_cba unique
  {
    association _Booking { create; }
  }

  group travel_main_actions implementation in class /dmo/bp_travel_main_a unique
  {
    action(features : instance) set_status_booked result [1] $self;
    action                      getTravel result [1] $self;
    action                      copyTravel result [1] $self;
  }

  group travel_aux_actions implementation in class /dmo/bp_travel_aux_a unique
  {
    action(features:instance) getMaxDate  result [1] $self;
    action(features:instance) getminDate  result [1] $self;
  }

  mapping for /dmo/travel
  {
    ...

  }
}

// behavior defintion for the BOOKING child entity
define behavior for /DMO/I_Booking_U alias booking
  lock dependent ( travel_id = travel_id )
{

  group booking_rud implementation in class /dmo/bp_booking_rud unique
  {
    read;
    update;
    delete;
  }

  group booking_fc implementation in class /dmo/bp_booking_fc unique
  {
    field ( read only ) TravelID, BookingID;
    field ( mandatory ) CustomerID, AirlineID, ConnectionID, FlightDate;
    action(features:instance) confirmBooking  result [1] $self;
  }

  mapping for /dmo/booking
  {
    ...
  }

}
```

Listing: Groups in the behavior definition of a managed business object

```
implementation unmanaged implementation in class /DMO/BP_TRAVEL_M;

// behavior definition for the TRAVEL root entity
define behavior for /DMO/I_Travel_M alias travel
  persistent table /dmo/travel_m
  with additional save
  lock master
  authorization master ( instance )
  etag LastChangedAt
{
  create;
  delete;
```

```
    association _Booking { create; }

    group travel_fc implementation in class /dmo/bp_travel_fc unique
    {
      field ( read only )      TravelID;
      field ( mandatory )      AgencyID, CustomerID, BeginDate, EndDate;
      field(features:instance) overall_status;
    }

    group travel_cba implementation in class /dmo/bp_travel_cba unique
    {
      association _Booking { create; }
    }

    group travel_actions implementation in class /dmo/bp_travel_a unique
    {
      action(features : instance)     set_status_booked result [1] $self;
      action                          getTravel result [1] $self;
      action ( authorization : none ) copyTravel result [1] $self;
      action(features:instance)       getMaxDate  result [1] $self;
      action(features:instance)       getminDate  result [1] $self;
    }

    group booking_det_val implementation in class /dmo/bp_booking_det_val unique
    {
      determination determineDiscount on modify { create; }
      validation validateAgency       on save   { field Agency_ID; }
      validation validateCustomer     on save   { field Customer_ID; }
      validation validateDates        on save   { field Begin_Date, End_Date; }
      validation validateStatus       on save   { field overall_Status; }
    }

    mapping for /dmo/travel
    {
      ...

    }
  }

  // behavior definition for the BOOKING child entity
  define behavior for /DMO/I_Booking_M alias booking
  {
    read;
    update;
    delete;

    mapping for /dmo/booking
    {
      ...
    }

    group booking_fc implementation in class /dmo/bp_booking_fc unique
    {
      field ( read only ) TravelID, BookingID;
      field ( mandatory ) CustomerID, AirlineID, ConnectionID, FlightDate;
      action(features:instance) confirmBooking  result [1] $self;
    }

    group booking_det_val implementation in class /dmo/bp_booking_det_val unique
    {
      determination totalBookingPrice on modify { field Flight_Price; }
      determination determineCustomerStatus on modify { create; }
      // internal action: triggered by determination
      internal action SetCustomerStatus;
    }

  }
```

Implementation-Related Aspects

(1) The name of the group and which operations are associated with this group do not matter to external users (and can therefore be changed retrospectively by the developer). This means that external operations and actions are still accessed by the usual syntax, in which only the name of the entity, the operation, and, if applicable, the action/determination/validation or association plays a role, but not the name of the group.

However, there are exceptions: the name of the group is relevant for the implementation of instance-based feature control - the corresponding implementations then control only those features that are associated with their respective group. (The framework merges the information for the external consumers.) The corresponding syntax `entity-group` can only be used within the implementation class associated with that group. Specifically, the following declarations are concerned:

Syntax for `methods ... for features`

```
methods method_name for features key_param
       request request_param for entity~group_name
       result result_parameter.
```

Syntax for `types/data ... for features`

```
type|data ... type table|structure for features key|request|result entity~group_name.
```

Note

This declaration can also be done in the `public section` of the implementation class to make the group-dependent type public outside. Because it makes the changes to group assignment incompatible with external users, such publishing is not recommended.

Example: Declaration of local handler for feature control implementation

Within the implementation class, the syntax `methods ... for features` for instance-based feature control can only be defined by specifying the group name:

```
class lhc_travel_fc definition inheriting from cl_abap_behavior_handler.
  private section.
    methods get_features for features
      importing keys request requested_features for travel~group_name result result.

endclass.
```

(2) Because associations with the usual association syntax can only be assigned as a whole to a group, it is not possible to implement the association's CREATE operation in an implementation class other than the READ operation.

Example: Local behavior implementation of create by association

```
class lhc_travel_cba definition inheriting from cl_abap_behavior_handler.
  private section.
    methods create_bookings for modify
      importing entities for create travel\_Booking.

    methods read_bookings for read
      importing keys for read travel\_Booking full result_requested
      result result link association_links.
endclass.

class lhc_travel_cba implementation.

  method create_bookings.
```

```
    ...
  endmethod.
  method read_bookings.
    ...
  endmethod.
endclass.
```

Using Client-Independent Database Tables

The RAP managed BO runtime supports transactional scenarios with client-independent database tables in scenarios with and without draft capabilities. This topic provides information about things you need to consider when using client-independent tables.

Creating Client-Independent Database Tables

The procedure for creating client-independent database tables is the same as described in Procedure: Creating Tables. You can simply remove the client field in the template for database tables.

The following code block shows the travel database table without a client field. The following example depicts a table for an active instance, but the respective draft table has an identical structure and can only be differentiated by the table name.

```
@EndUserText.label : 'Flight Reference Scenario: Managing Travels'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table /dmo/travel_m {
  key travel_id   : /dmo/travel_id not null;
  agency_id       : /dmo/agency_id;
  customer_id     : /dmo/customer_id;
  begin_date      : /dmo/begin_date;
  end_date        : /dmo/end_date;
  @Semantics.amount.currencyCode : '/dmo/travel_data.currency_code'
  booking_fee     : /dmo/booking_fee;
  @Semantics.amount.currencyCode : '/dmo/travel_data.currency_code'
  total_price     : /dmo/total_price;
  currency_code   : /dmo/currency_code;
  description     : /dmo/description;
  overall_status  : /dmo/overall_status;
  @AbapCatalog.anonymizedWhenDelivered : true
  created_by      : syuname;
  created_at      : timestampl;
  @AbapCatalog.anonymizedWhenDelivered : true
  last_changed_by : syuname;
  last_changed_at : timestampl;
}
```

Modeling Business Objects with Client-Independent Tables

A business object must be consistent with regard to the client fields in the database tables for the active and the draft instance. This means that CDS views that are used for the data model in scenarios with client-independent tables must not contain a client element and the same applies to draft tables: If a managed business object has client-independent data sources, the

automatically created draft tables are created client-independently. For more information about how to generate draft database tables, refer to Draft Database Table.

In addition, the created SQL-view of a CDS V1 view must not contain such a client element. When using CDS V2, the client handling of the CDS view is implicitly defined by the underlying data source.

There must be client-consistency among the entities in the business object composition structure. You can't to have one entity that is client-dependent and another one that is client-independent with a compositional relationship between them.

Business Object Runtime with Client-Independent Tables

The client field in database tables influences the runtime of business objects during lock. While locks are set on entity instances only for one specific client in client-dependent tables, locks in client-independent tables lock instances for all clients.

> **Note**
> In scenarios in which you have client-dependent database tables, but join client-independent fields from other database tables to your CDS view, the managed BO runtime locks the instances specific to the client. This means only the fields from the client-dependent database tables are locked. If you also want to lock the client-independent fields, you have to implement an `unmanaged lock`.

> **Note**
> In scenarios, in which you use an unmanaged save in the managed scenarios, the managed BO runtime always sets a client-specific lock. If you want to lock client-independently, you have to use an `unmanaged lock`.

Related Information

Lock Definition
Integrating Unmanaged Save in Managed Business Objects

Determination and Validation Modelling

This chapter outlines the best practices for modelling determinations and validations.

For basic information on determinations and validations, see Determinations and Validations.

Trigger conditions

Determinations and validations are commonly used to first read instance data and then to perform the modification/validation based on the read data. In order to ensure, that the determinations and validations can react on changes in fields that are read repectively, these fields should be used as their trigger conditions.

Kind independent behavior

The behavior of determinations and validations must coincide for draft and active instances. This includes the state messages reported by determinations and validations.

In order to implement determinations and validations kind independently, use the derived type component `%tky`. This component includes the draft indicator `%is_draft` which is set depending on the context by the framework. Before you set the draft indicator `%is_draft` manually, ensure that this does not lead to an unwanted difference in behavior between draft and active instances.

For more information on the draft indicator, see Draft.

Runtime independent behavior

The behavior of determination and validation executions must be identical, irrespective from the runtime phase they are executed in. A validation, for instance, that is part of a determine action, must provide the same reported result in the interaction phase and in the save sequence.

Determination usage

Application logic intended to notify about saved changes must be implemented in a provider implementation that is called after the point of no return of the save sequence, e.g. in the additional save implementation. It must not be implemented in a determination on save.

Failed entries from determinations

Determinations don't have a failed response. In determinations where fields need to be checked before their calculation can be performed, the application needs to store negative results of these checks. Validations then need to consider these negative results when returning their failed response during CHECK_BEFORE_SAVE. In cases where determinations are executed multiple times due to retriggered conditions, the check result can change during subsequent determination executions. This needs to be considered by the determination logic that maintains the failed entries for validations.

RAP BO Runtime

The runtime of a RAP business object (BO) consists of two main parts: The interaction phase and the save sequence.

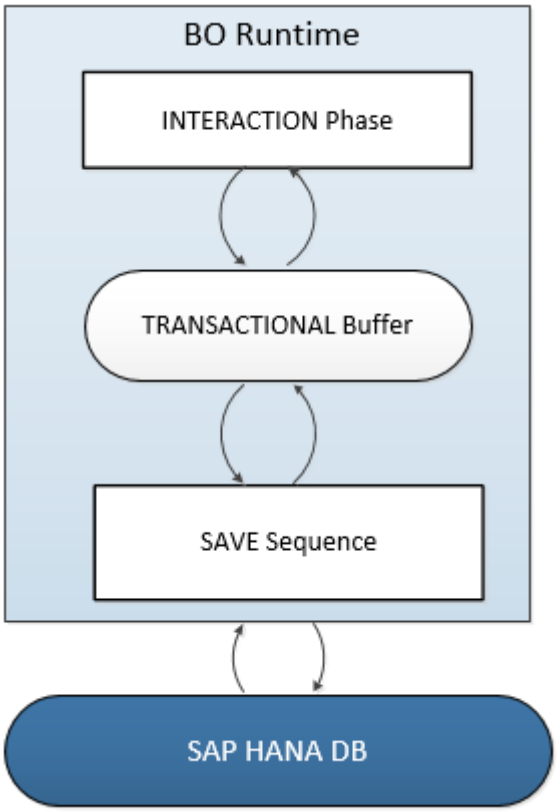
Interaction Phase

The interaction phase is the first part of the BO runtime. In this phase, a consumer can call business object operations to change data and read instances with or without the transactional changes. The business object keeps the changes in its internal transactional buffer, which represents the state. This transactional buffer is always required for a business object, regardless of how it is implemented.

Save Sequence

When all changes are done, the data must be persisted. This happens in the second part of the BO runtime, the save sequence.

This image is interactive. Hover over each area for a description. Click highlighted areas for more information.



Please note that image maps are not interactive in PDF output.

Implementation

The implementation of the interaction phase and the save sequence is carried out in a special type of class pool, the behavior pool, which refers to the behavior definition. The interaction phase is represented by the local handler class and the save sequence by the local saver class. For more information on the implementation of business object functionality, see also : Business Object Provider API.

Operations Runtime

The RAP Transactional Model and the SAP LUW

This topic describes how RAP supports the transactional model of a controlled SAP LUW. The logical unit of work (LUW) is the sum of all operations and work processes that are used to transfer data from one consistent state on the database to another.

Save Sequence Runtime

The save sequence is part of the business object runtime and is called after at least one successful modification was performed during the interaction phase.

Operations Runtime

This section describes the operations that are available for RAP business objects and their functionality.

Standard Operations Runtime

Nonstandard Operations Runtime

Nonstandard operations in RAP are operations that do not provide the canonical behavior of RAP BOs. Instead, they are used to provide customized, business-logic-specific behavior.

Standard Operations Runtime

Standard operations in RAP business objects are operations that cover the standard functionality for creating, updating, deleting and locking RAP BO instances.

Create Operation Runtime

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

Update Operation Runtime

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

Delete Operation Runtime

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

Create by Association Operation Runtime

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

Create Operation Runtime

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

ℹ Note

In case of a managed business object, instances for child entities can only be created by a create-by-association.

The following runtime diagram illustrates the main agents' activities during the interaction phase of a create operation when the BO consumer sends a create request. The save sequence is illustrated in a separate diagram, see Save Sequence Runtime.

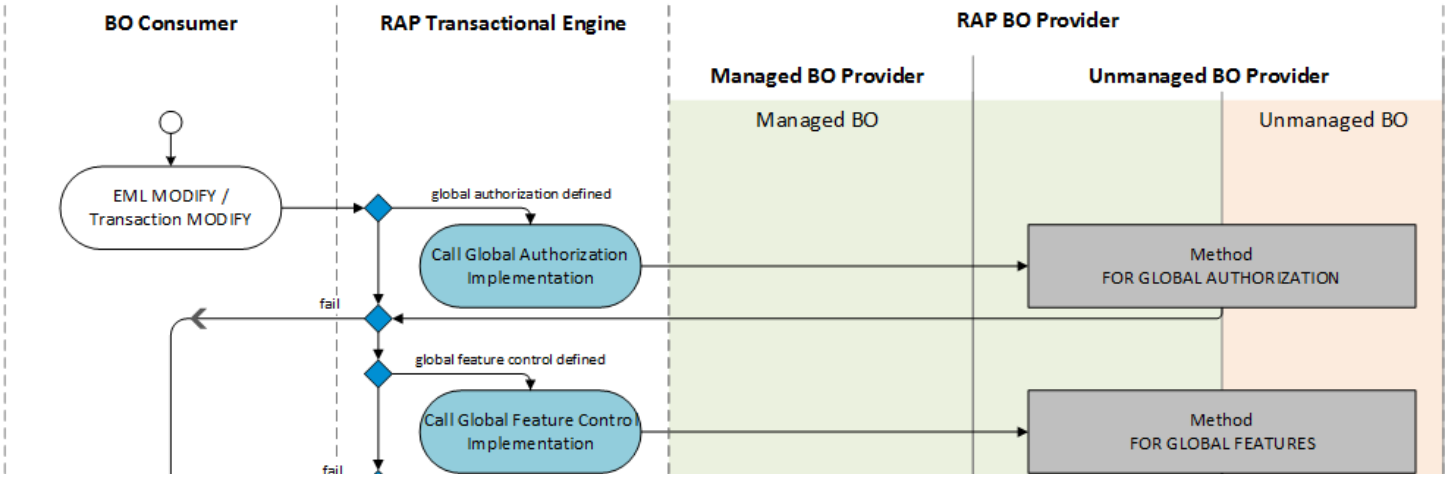
Instantiation of Handler Classes in the ABAP Behavior Pool

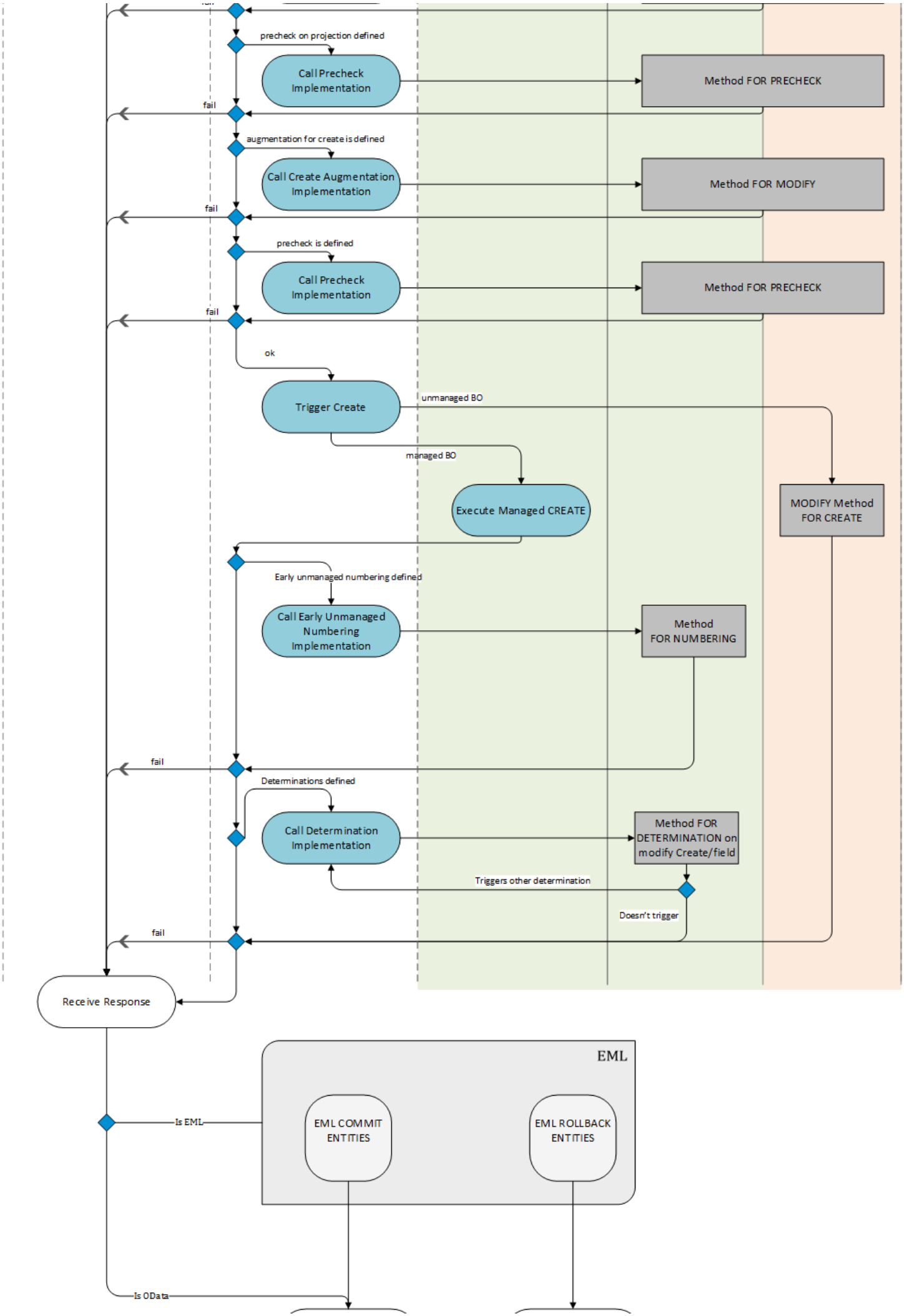
In general, the instantiation of handler and saver classes is tightly coupled to an ABAP session. Instances live as long as the ABAP session and they can exist across LUW borders. However, their life cycle depends on the way their methods are called. Nested method invocations always provoke the reinstantiation of the local handler and saver classes. For example, this is the case if a handler method executes an EML modify request in local mode to call another method of the same handler class, or if other BOs are involved in the invocation chain.

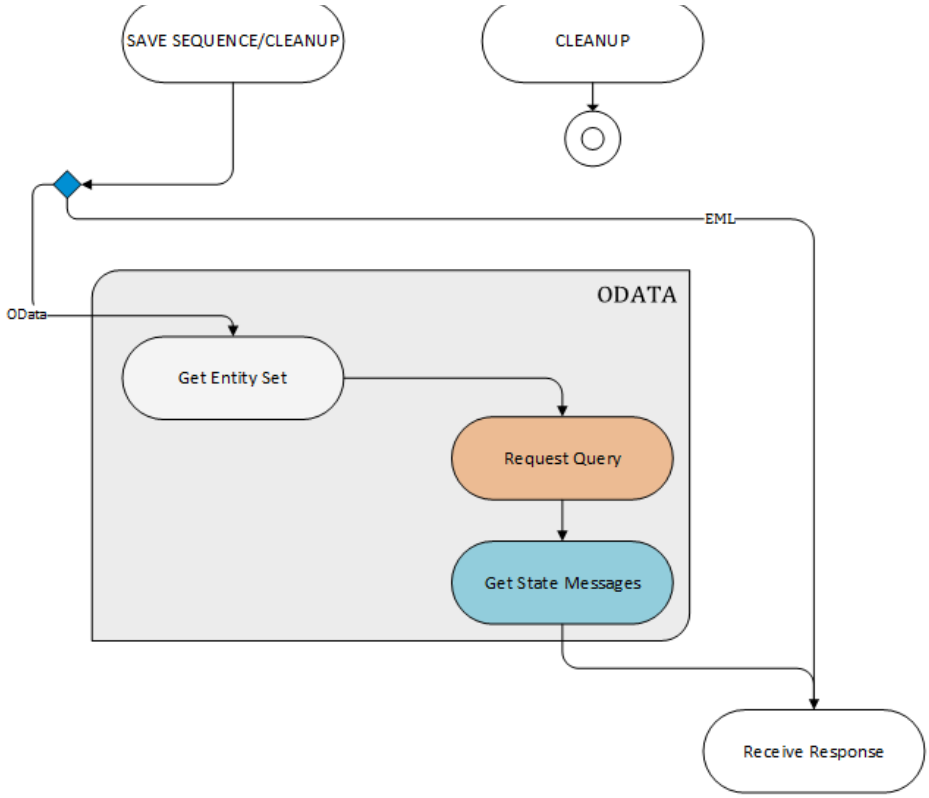
ℹ Note

Do not work with data in member variables for keeping data for subsequent method invocation. The data is lost once the handler class is reinstantiated.

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




precheck on projection defined
Call Precheck Implementation
Method FOR PRECHECK
fail
augmentation for create is defined
Call Create Augmentation Implementation
Method FOR MODIFY
fail
precheck is defined
Call Precheck Implementation
Method FOR PRECHECK
fail
ok
Trigger Create
unmanaged BO
managed BO
Execute Managed CREATE
MODIFY Method FOR CREATE
Early unmanaged numbering defined
Call Early Unmanaged Numbering Implementation
Method FOR NUMBERING
fail
Determinations defined
Call Determination Implementation
Method FOR DETERMINATION on modify Create/field
Triggers other determination
Doesn't trigger
fail
Receive Response
EML
Is EML
EML COMMIT ENTITIES
EML ROLLBACK ENTITIES
Is OData



Please note that image maps are not interactive in PDF output.

Parent topic: Standard Operations Runtime

Related Information

Update Operation Runtime
Delete Operation Runtime
Create by Association Operation Runtime

Update Operation Runtime

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

The following runtime diagram illustrates the main agents' activities during the interaction phase of an update operation when a BO consumer sends an update request. The save sequence is illustrated in a separate diagram, see Save Sequence Runtime.

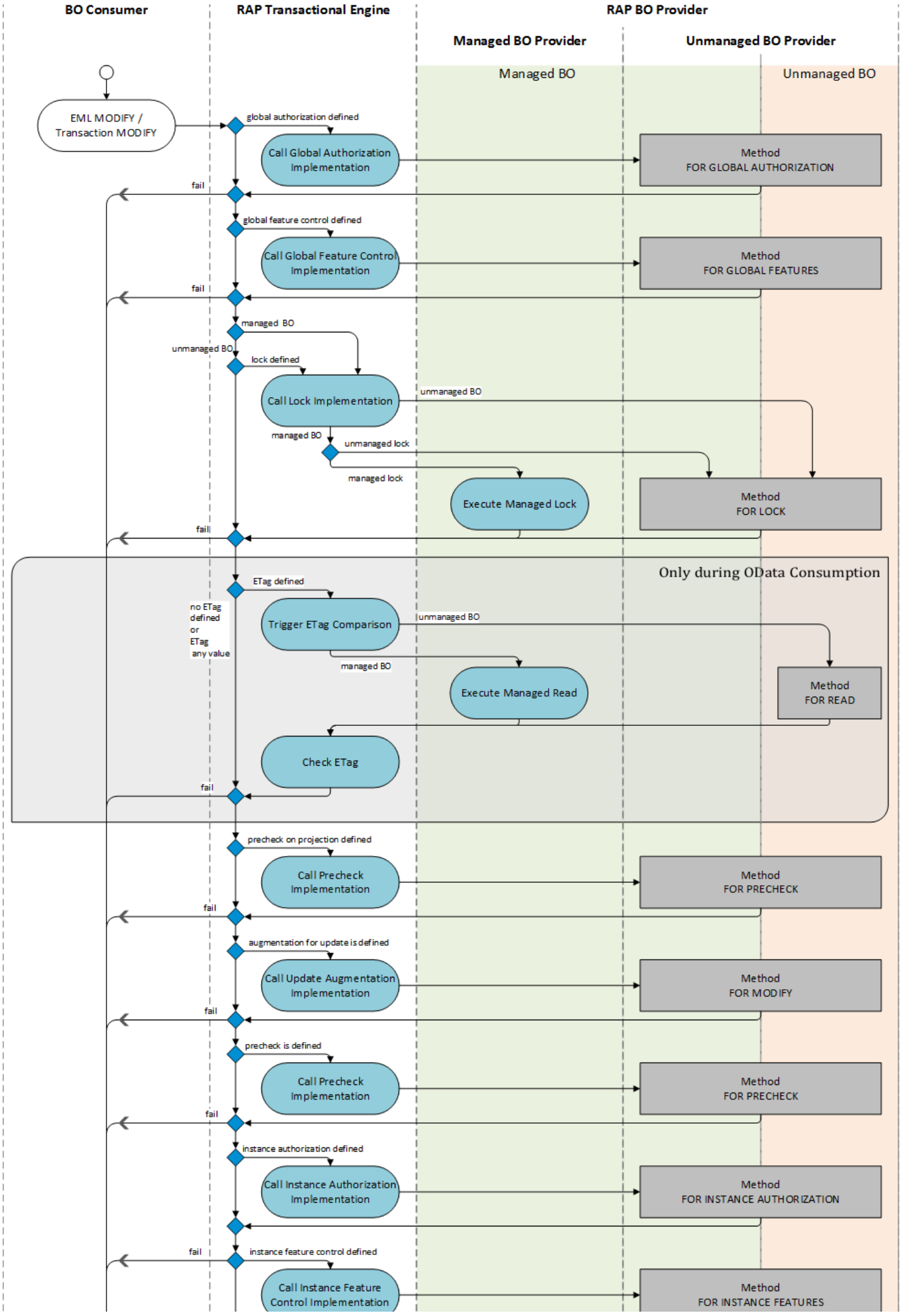
Instantiation of Handler Classes in the ABAP Behavior Pool

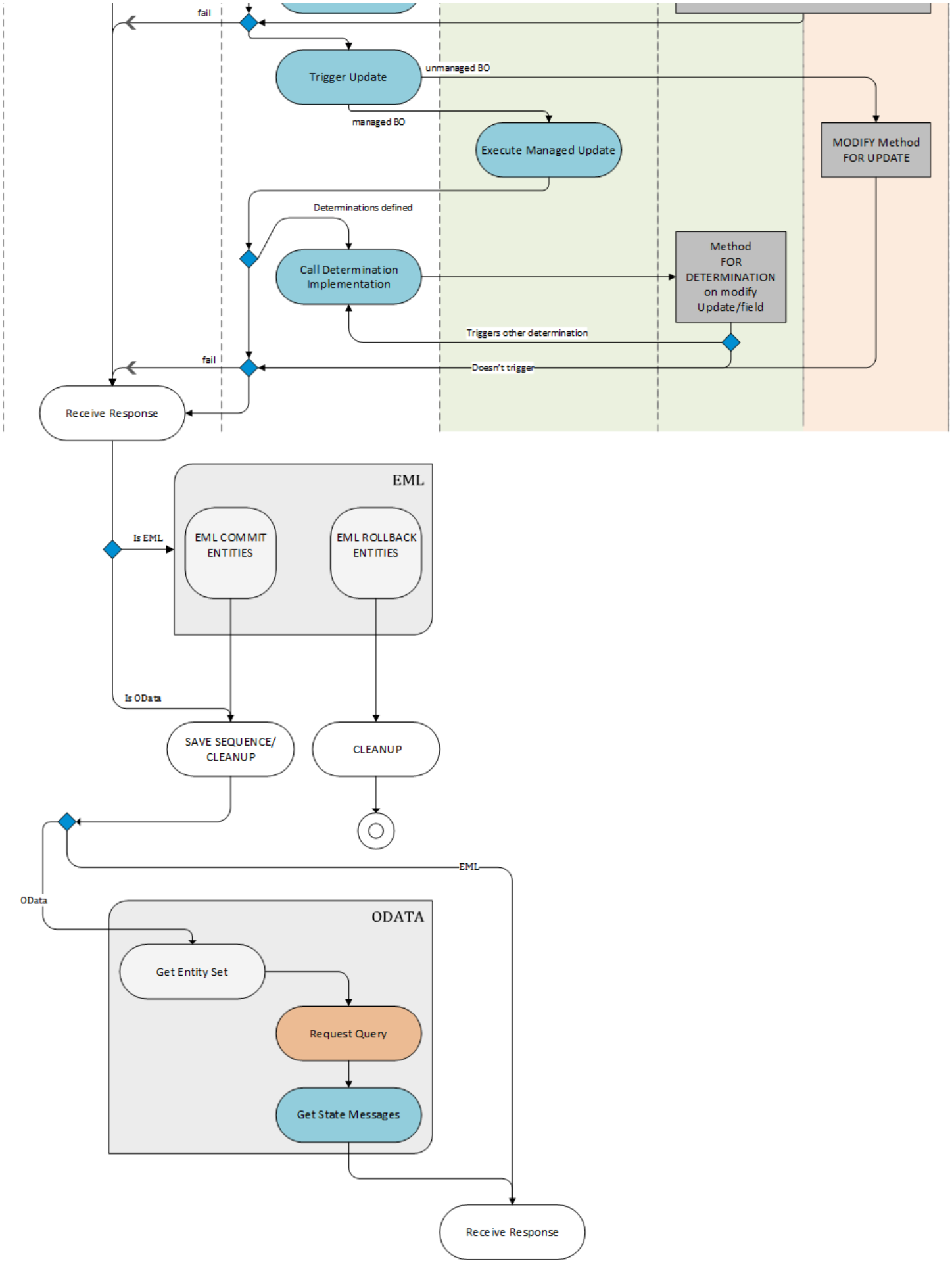
In general, the instantiation of handler and saver classes is tightly coupled to an ABAP session. Instances live as long as the ABAP session and they can exist across LUW borders. However, their life cycle depends on the way their methods are called. Nested method invocations always provoke the reinstantiation of the local handler and saver classes. For example, this is the case if a handler method executes an EML modify request in local mode to call another method of the same handler class, or if other BOs are involved in the invocation chain.

ℹ Note

Do not work with data in member variables for keeping data for subsequent method invocation. The data is lost once the handler class is reinstantiated.

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


BO Consumer
RAP Transactional Engine
RAP BO Provider
Managed BO Provider
Unmanaged BO Provider
Managed BO
Unmanaged BO
EML MODIFY / Transaction MODIFY
global authorization defined
Call Global Authorization Implementation
Method FOR GLOBAL AUTHORIZATION
fail
global feature control defined
Call Global Feature Control Implementation
Method FOR GLOBAL FEATURES
fail
managed BO
unmanaged BO
lock defined
Call Lock Implementation
unmanaged BO
managed BO
unmanaged lock
managed lock
Execute Managed Lock
Method FOR LOCK
fail
Only during OData Consumption
ETag defined
no ETag defined or ETag any value
Trigger ETag Comparison
unmanaged BO
managed BO
Execute Managed Read
Method FOR READ
Check ETag
fail
precheck on projection defined
Call Precheck Implementation
Method FOR PRECHECK
fail
augmentation for update is defined
Call Update Augmentation Implementation
Method FOR MODIFY
fail
precheck is defined
Call Precheck Implementation
Method FOR PRECHECK
fail
instance authorization defined
Call Instance Authorization Implementation
Method FOR INSTANCE AUTHORIZATION
fail
instance feature control defined
Call Instance Feature Control Implementation
Method FOR INSTANCE FEATURES



Please note that image maps are not interactive in PDF output.

Parent topic: Standard Operations Runtime

Related Information

Create Operation Runtime
Delete Operation Runtime
Create by Association Operation Runtime

Delete Operation Runtime

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

The following runtime diagram illustrates the main agents' activities during the interaction phase of a delete operation when a BO consumer sends a delete request. The save sequence is illustrated in a separate diagram, see Save Sequence Runtime.

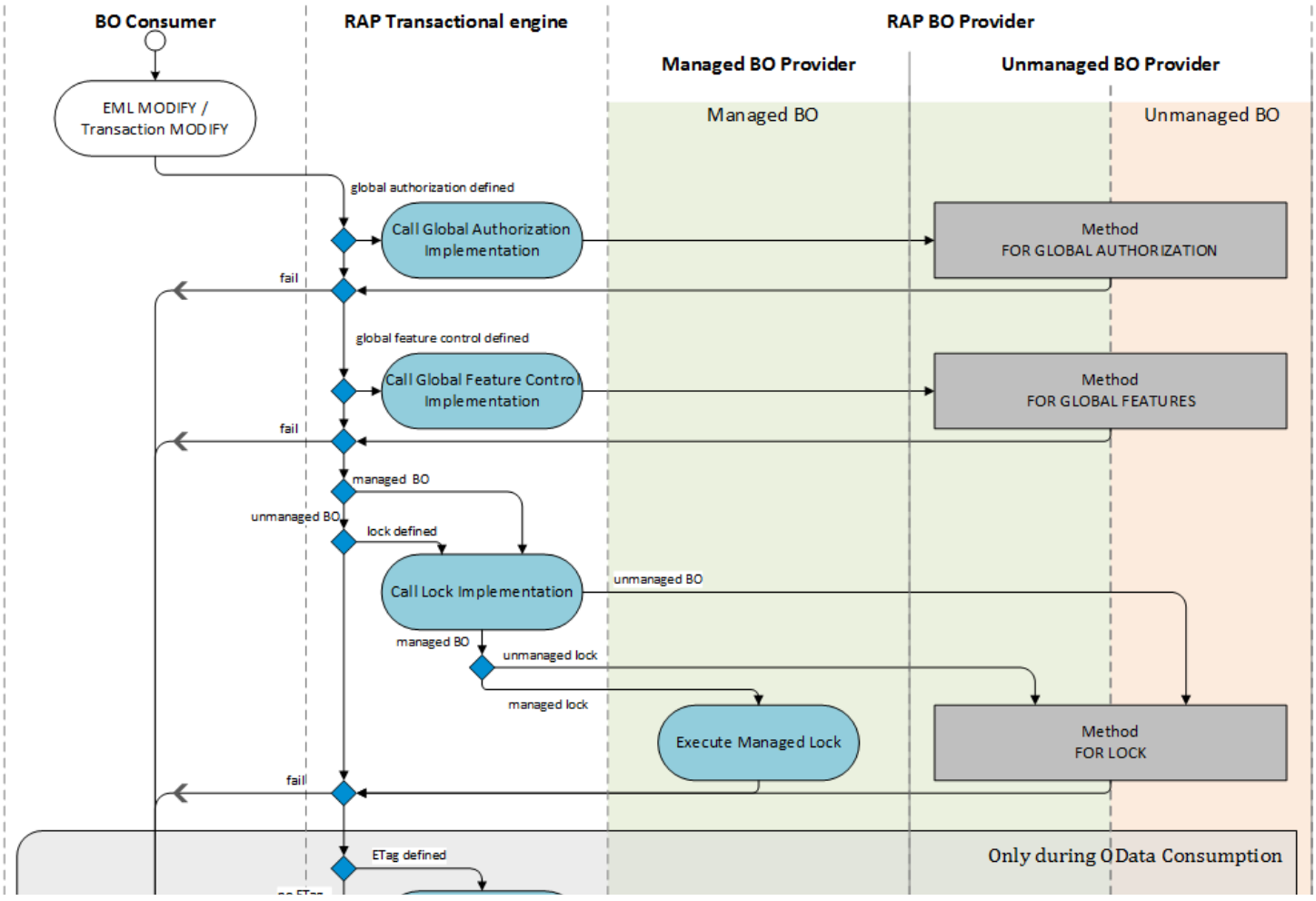
Instantiation of Handler Classes in the ABAP Behavior Pool

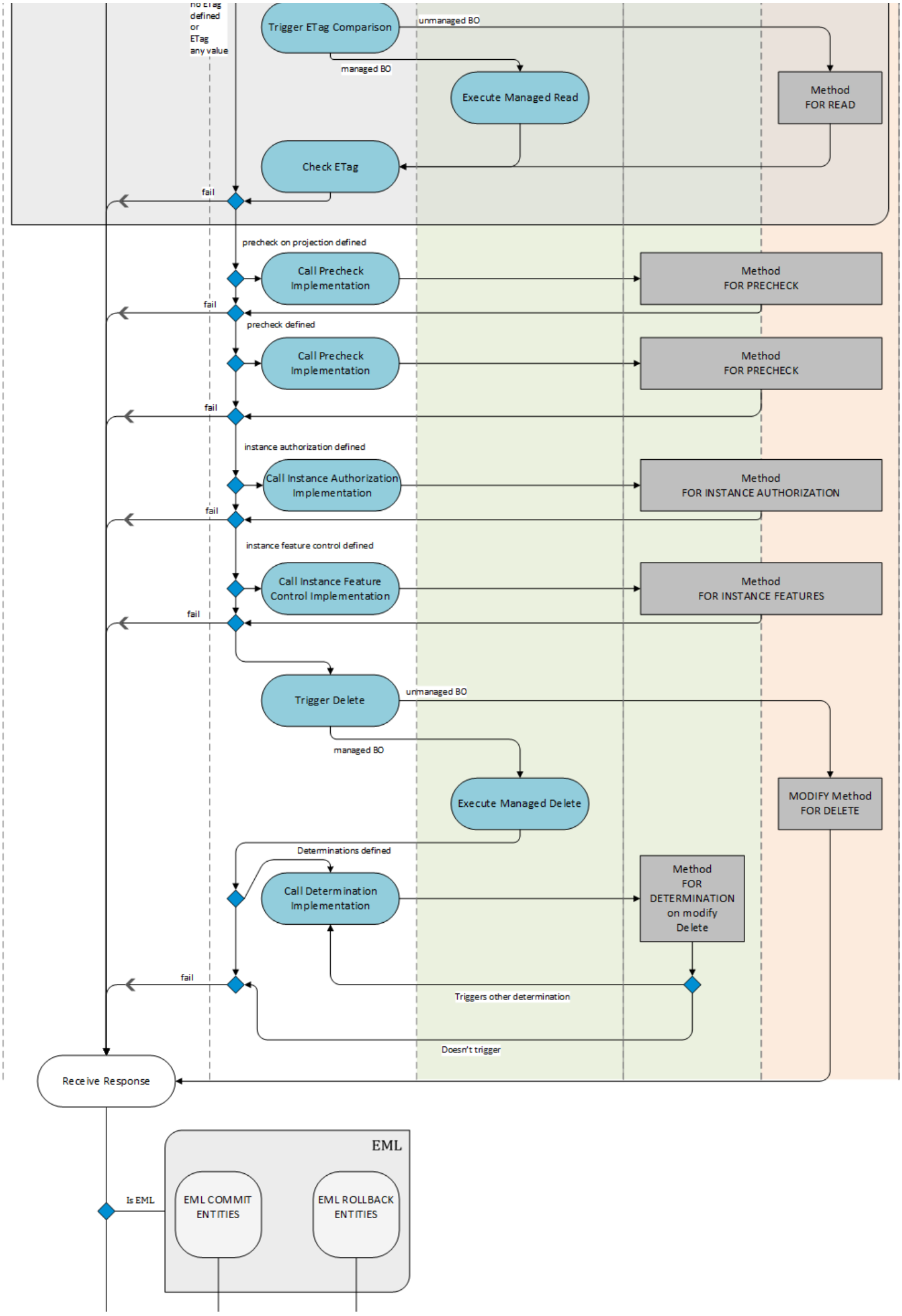
In general, the instantiation of handler and saver classes is tightly coupled to an ABAP session. Instances live as long as the ABAP session and they can exist across LUW borders. However, their life cycle depends on the way their methods are called. Nested method invocations always provoke the reinstantiation of the local handler and saver classes. For example, this is the case if a handler method executes an EML modify request in local mode to call another method of the same handler class, or if other BOs are involved in the invocation chain.

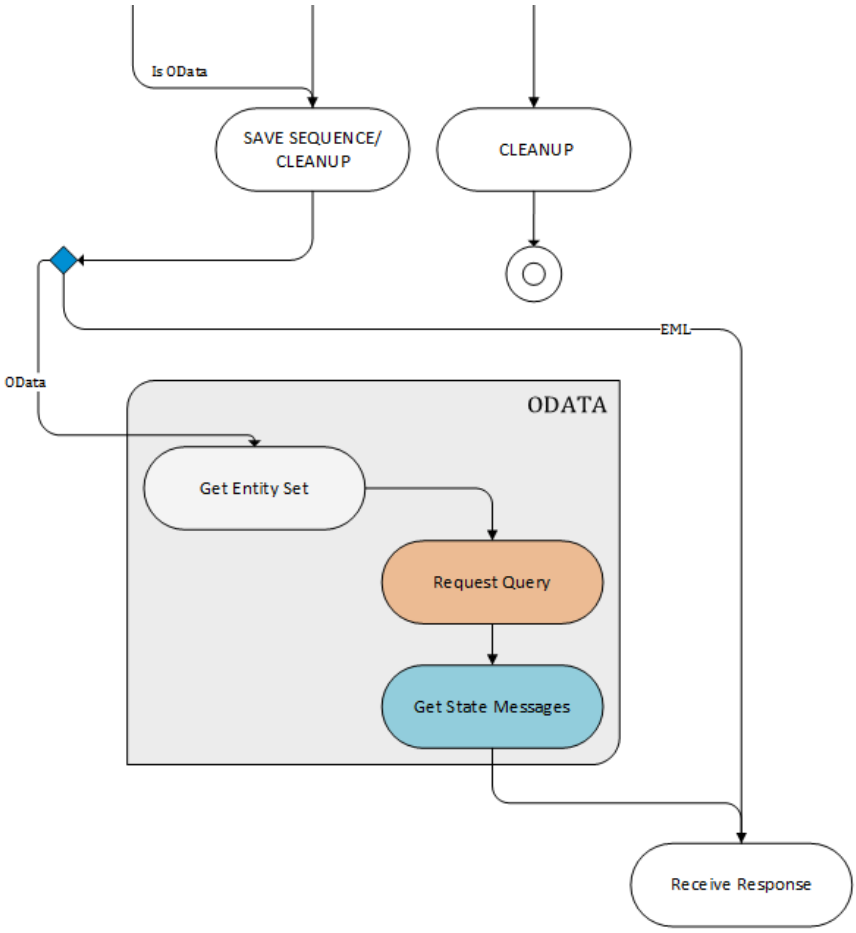
ℹ Note

Do not work with data in member variables for keeping data for subsequent method invocation. The data is lost once the handler class is reinstantiated.

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




no ETag defined or ETag any value
Trigger ETag Comparison
unmanaged BO
managed BO
Execute Managed Read
Method FOR READ
Check ETag
fail
precheck on projection defined
Call Precheck Implementation
Method FOR PRECHECK
fail
precheck defined
Call Precheck Implementation
Method FOR PRECHECK
fail
instance authorization defined
Call Instance Authorization Implementation
Method FOR INSTANCE AUTHORIZATION
fail
instance feature control defined
Call Instance Feature Control Implementation
Method FOR INSTANCE FEATURES
fail
Trigger Delete
unmanaged BO
managed BO
Execute Managed Delete
MODIFY Method FOR DELETE
Determinations defined
Call Determination Implementation
Method FOR DETERMINATION on modify Delete
fail
Triggers other determination
Doesn't trigger
Receive Response
EML
Is EML
EML COMMIT ENTITIES
EML ROLLBACK ENTITIES



Please note that image maps are not interactive in PDF output.

Parent topic: Standard Operations Runtime

Related Information

Create Operation Runtime
Update Operation Runtime
Create by Association Operation Runtime

Create by Association Operation Runtime

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

The following runtime diagram illustrates the main agents' activities during the interaction phase of a `create by association` operation when a BO consumer sends a create-by-association request. The save sequence is illustrated in a separate diagram, see Save Sequence Runtime.

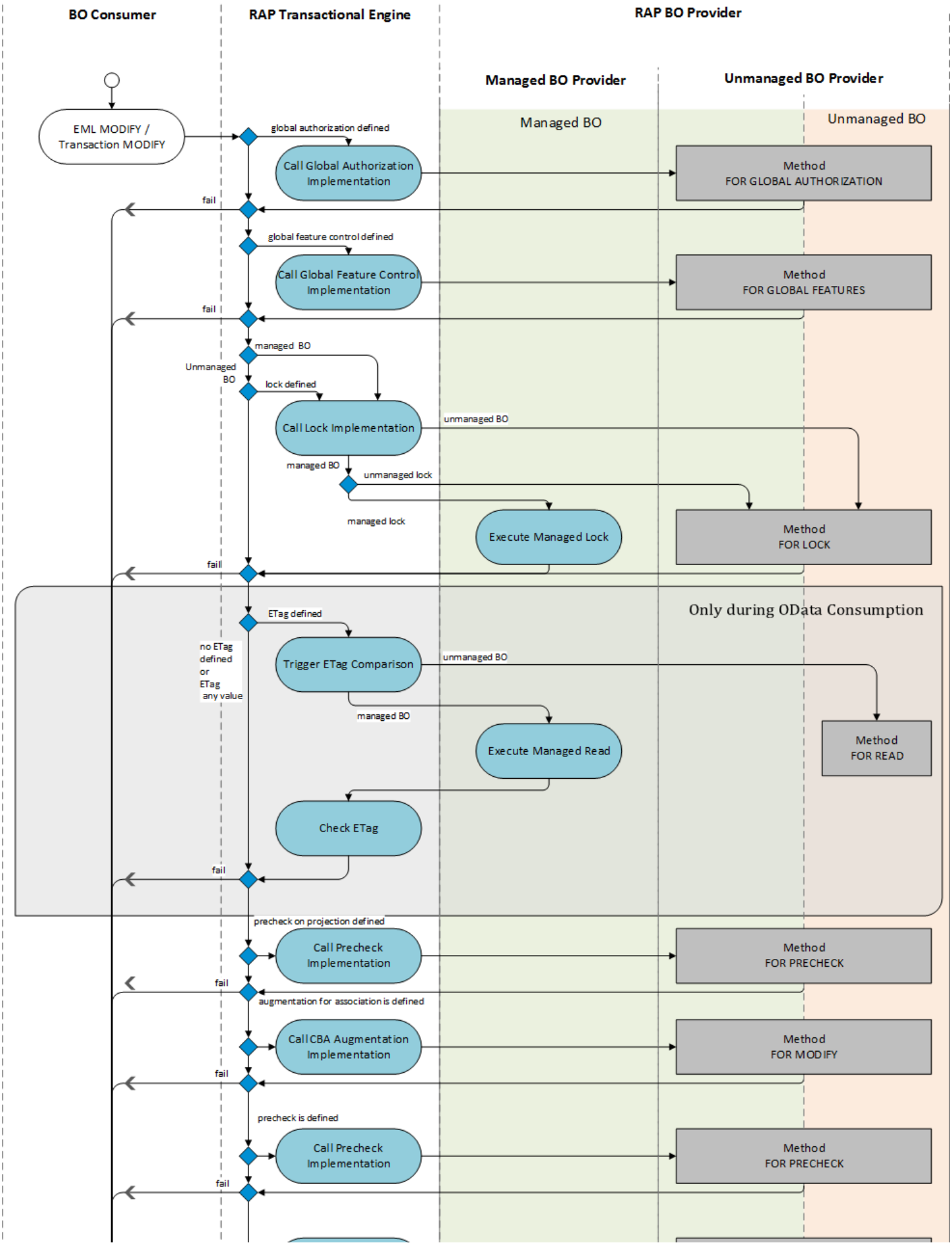
Instantiation of Handler Classes in the ABAP Behavior Pool

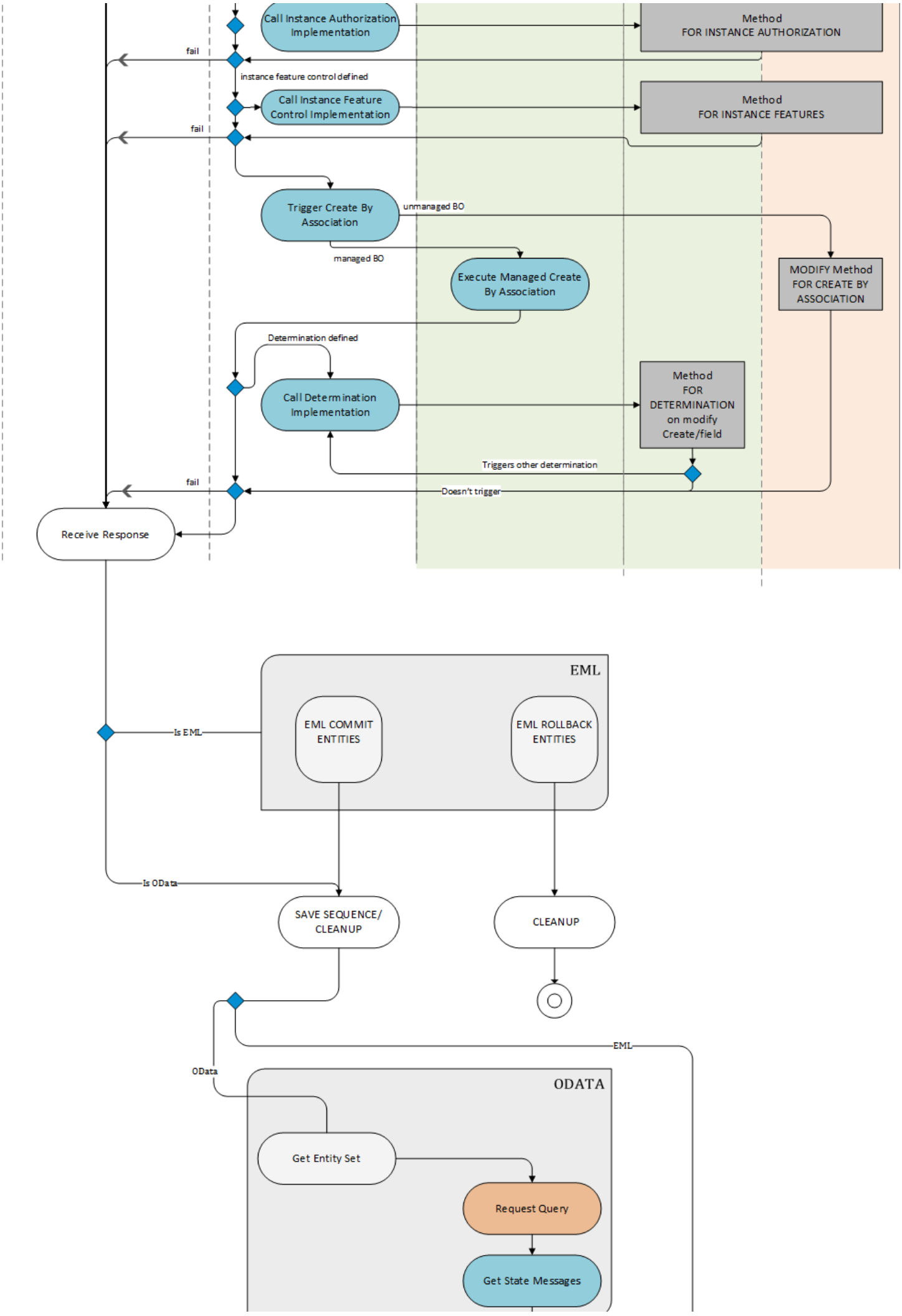
In general, the instantiation of handler and saver classes is tightly coupled to an ABAP session. Instances live as long as the ABAP session and they can exist across LUW borders. However, their life cycle depends on the way their methods are called. Nested method invocations always provoke the reinstantiation of the local handler and saver classes. For example, this is the case if a handler method executes an EML modify request in local mode to call another method of the same handler class, or if other BOs are involved in the invocation chain.

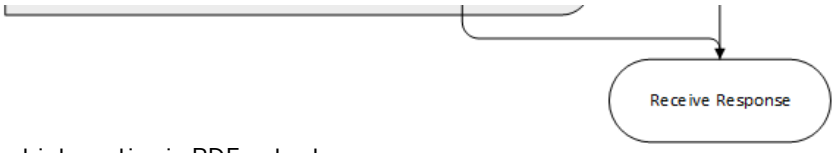
ℹ Note

Do not work with data in member variables for keeping data for subsequent method invocation. The data is lost once the handler class is reinstantiated.

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




Call Instance Authorization Implementation
Method FOR INSTANCE AUTHORIZATION
fail
instance feature control defined
Call Instance Feature Control Implementation
Method FOR INSTANCE FEATURES
fail
Trigger Create By Association
unmanaged BO
managed BO
Execute Managed Create By Association
MODIFY Method FOR CREATE BY ASSOCIATION
Determination defined
Call Determination Implementation
Method FOR DETERMINATION on modify Create/field
Triggers other determination
fail
Doesn't trigger
Receive Response
EML
EML COMMIT ENTITIES
EML ROLLBACK ENTITIES
Is EML
Is OData
SAVE SEQUENCE/ CLEANUP
CLEANUP
EML
OData
ODATA
Get Entity Set
Request Query
Get State Messages



Please note that image maps are not interactive in PDF output.

Parent topic: Standard Operations Runtime

Related Information

Create Operation Runtime
Update Operation Runtime
Delete Operation Runtime

Nonstandard Operations Runtime

Nonstandard operations in RAP are operations that do not provide the canonical behavior of RAP BOs. Instead, they are used to provide customized, business-logic-specific behavior.

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

Action Runtime

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

The following runtime diagram illustrates the main agents' activities during the interaction phase of an action when a BO consumer requests the execution of an action. The save sequence is illustrated in a separate diagram, see Save Sequence Runtime.

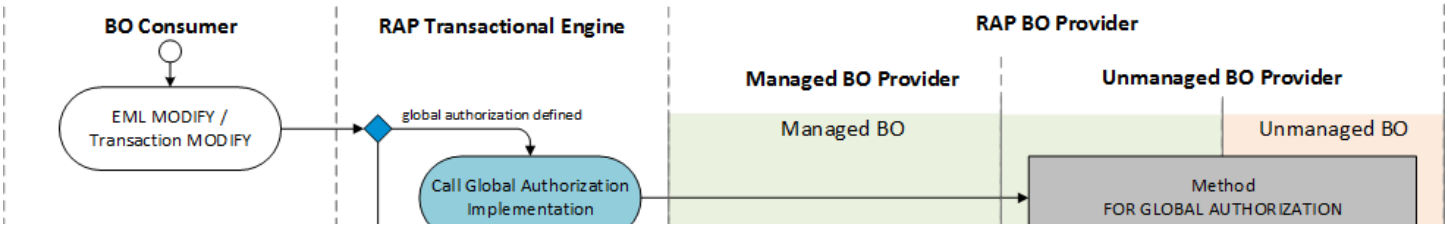
Instantiation of Handler Classes in the ABAP Behavior Pool

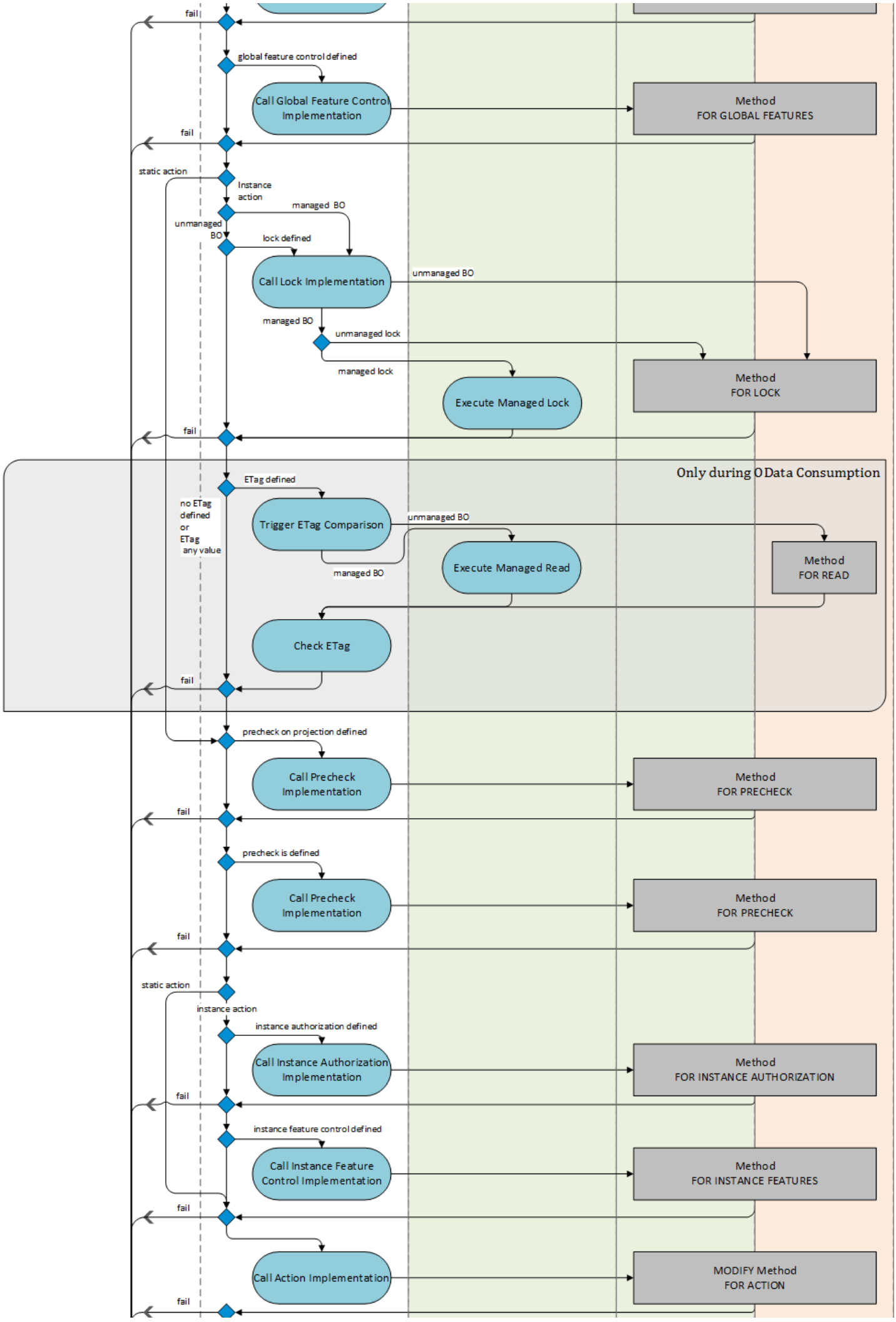
In general, the instantiation of handler and saver classes is tightly coupled to an ABAP session. Instances live as long as the ABAP session and they can exist across LUW borders. However, their life cycle depends on the way their methods are called. Nested method invocations always provoke the reinstantiation of the local handler and saver classes. For example, this is the case if a handler method executes an EML modify request in local mode to call another method of the same handler class, or if other BOs are involved in the invocation chain.

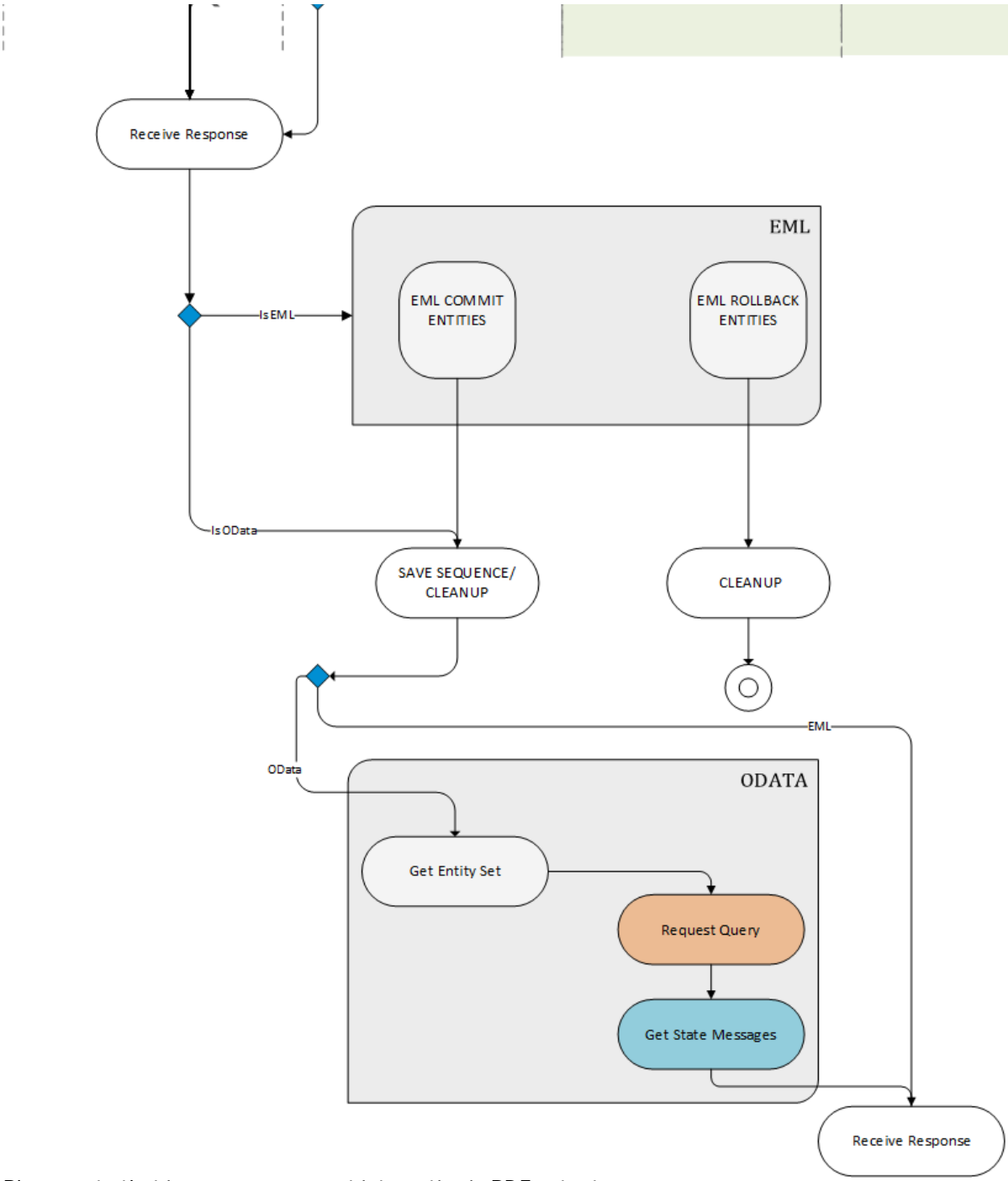
ℹ Note

Do not work with data in member variables for keeping data for subsequent method invocation. The data is lost once the handler class is reinstantiated.

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




fail
global feature control defined
Call Global Feature Control Implementation
Method FOR GLOBAL FEATURES
fail
static action
Instance action
managed BO
unmanaged BO
lock defined
Call Lock Implementation
unmanaged BO
managed BO
unmanaged lock
managed lock
Execute Managed Lock
Method FOR LOCK
fail
Only during OData Consumption
ETag defined
no ETag defined or ETag any value
Trigger ETag Comparison
unmanaged BO
managed BO
Execute Managed Read
Method FOR READ
Check ETag
fail
precheck on projection defined
Call Precheck Implementation
Method FOR PRECHECK
fail
precheck is defined
Call Precheck Implementation
Method FOR PRECHECK
fail
static action
instance action
instance authorization defined
Call Instance Authorization Implementation
Method FOR INSTANCE AUTHORIZATION
fail
instance feature control defined
Call Instance Feature Control Implementation
Method FOR INSTANCE FEATURES
fail
Call Action Implementation
MODIFY Method FOR ACTION
fail



Please note that image maps are not interactive in PDF output.

Parent topic: Nonstandard Operations Runtime

Related Information

Actions
Action Definition
Action Implementation

The RAP Transactional Model and the SAP LUW

This topic describes how RAP supports the transactional model of a controlled SAP LUW. The logical unit of work (LUW) is the sum of all operations and work processes that are used to transfer data from one consistent state on the database to another.

The ABAP RESTful Application Programming Model (RAP) does not introduce its own LUW concept, but uses the rules and concepts of the SAP LUW. RAP strictly adheres to the rules of a SAP LUW by inherently checking transactional contexts and only allowing developers to use or call implementations that match the transactional contexts. In that sense, the **controlled SAP LUW** is inherently built into RAP. For more information about transactional consistency during the SAP LUW, see Controlled SAP LUW.

To make sure that an application and all involved parties (for example other business objects) work consistently and aligned with all LUW prerequisites, the transactional modeling of highly complex scenarios can become cumbersome and unforeseeable. When developing with RAP, you achieve a consistent transactional modeling that includes all involved parties by default. This is ensured by adhering to the framework RAP provides, which entails checks for all different use cases, exactly like the controlled SAP LUW checks the contexts. The RAP-inherent checks ensure that your application is consistent and stable, also when more than one BO is involved.

RAP divides the LUW into the **interaction phase** and the **save sequence**. Data can be changed and created in an inconsistent state on the transactional buffer during the interaction phase. The save sequence ensures that the new state of data is consistent and ready to be written to the database. The final database commit is then also executed during the save sequence. An LUW is terminated when the database commit is executed successfully and the transactional buffer is cleared, or, if the database commit is unsuccessful. In the latter case, all work processes of the current LUW must be completely rolled back. That means, the LUW must not leave any remains, neither on any database, nor on the transactional buffer. This is also true for BO-external procedures. The RAP framework offers options to trigger this clean-up work. See The Save Sequence.

The RAP transactional model is seamlessly embedded in a SAP LUW. The statements to terminate a SAP LUW `COMMIT WORK` and `ROLLBACK WORK` trigger `COMMIT ENTITIES` and `ROLLBACK ENTITIES` and thus terminate the LUW with RAP procedures.

ℹ Note

The following violations against the SAP LUW principles will always lead to a runtime error in a RAP context:

- Explicit use of `COMMIT WORK` or `ROLLBACK WORK`. The transaction handling and database commit is exclusively handled by RAP.
- Calling the update in background tasks (bgPF) in the interaction phase or the early save phase.

Some other operations only lead to runtime errors, if the RAP behavior definition is implemented in ABAP for Cloud Development and the `strict(2)`-mode is applied:

- Invoking authorization checks during the save sequence
- Modifying database operation in the interaction phase or early-save phase
- Raising business events in the interaction phase or early-save phase
- (Implicit or explicit) database commits for the primary database connection in the late-save phase.

For more information, see General RAP BO Implementation Contract.

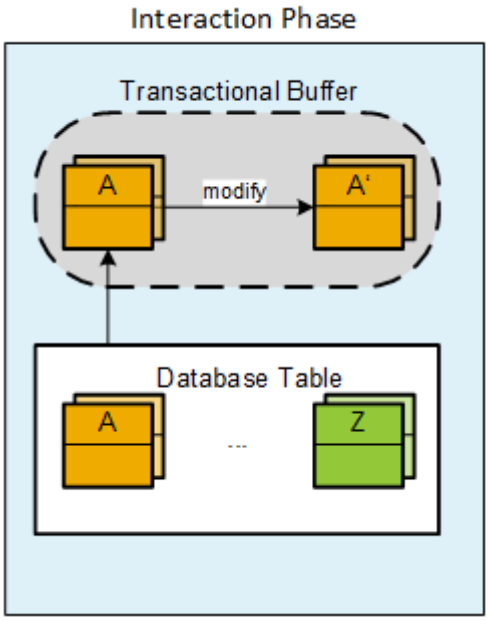
Every BO entity instance that is involved in one LUW must be locked to avoid conflicting access. Locking is tightly coupled to the lifetime of the LUW. A lock for a BO entity instance lasts as long as the LUW. With every database commit or with a rollback, the lock is released and the same BO instance can be used in a new LUW. For more information about the locking concept in RAP, see Pessimistic Concurrency Control (Locking).

The Interaction Phase

Working with data of a RAP business object implies manipulating data on the database: You can create new data, or you can modify existing data. Both approaches require an additional memory area to store the data that is currently being worked on.

This area is called transactional buffer. In managed BOs, the transactional buffer is fully handled by the managed BO provider. For unmanaged BOs, the transactional buffer is implemented by the application developer within the unmanaged BO provider. Newly created data, that is, new BO instances are created in the transactional buffer and are further processed from there. Existing data is retrieved from the database and loaded into the transactional buffer for further processing, for example changing values of the BO instance. During this time, the data on the database remains in a consistent state. In other words, while the BO instance is consistent on the database table, the state of the BO instance on the transactional buffer can become inconsistent.

The following diagram shows how data is retrieved from the database table and modified on the transactional buffer.



The Interaction Phase

Data modification on the transactional buffer is done via RAP operations during the interaction phase, see Operations and Determinations. During this phase the data does not have to be in a consistent state.

⚠ Caution

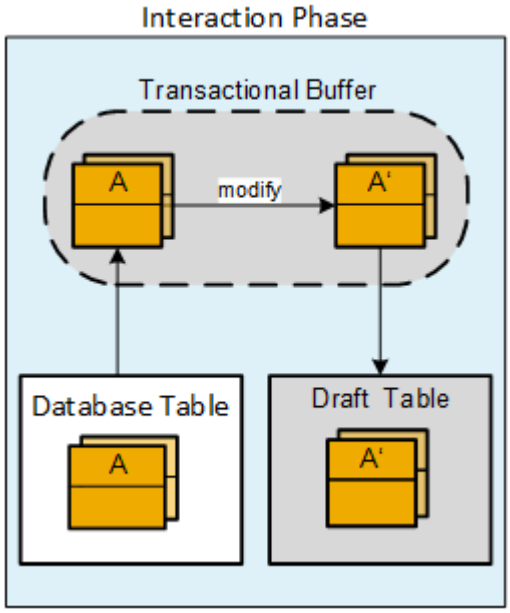
There is no consistency check during the interaction phase. That means, the data that is being worked on is always potentially inconsistent. Hence, it is not allowed to trigger any procedure that directly changes data on the database during that phase. This would be based on a potentially inconsistent data state and there is no mechanism that checks the data before saving.

If any work process during the interaction phase fails, the consumer is notified and is able to restart the request.

The Persisted Transactional Buffer in Draft Scenarios

Due to the stateless approach of draft scenarios, the logical LUW can be split into more than one ABAP session in such scenarios. Technically, one LUW is then divided into several database LUWs. The state of the transactional buffer must be stored intermediately and there must be durable locks that survive a termination of the ABAP session. When working with draft-enabled business objects, every change on the transactional buffer, which is the set of changes within one technical LUW, is persisted to the draft database table. This ensures that draft data don't get lost. Like this, inconsistent data can reach the draft database table. The content of this database table, however, is not process-relevant. Writing the latest transactional buffer to the draft persistence is part of the interaction phase and therefore the save sequence is not relevant for the draft persistence. For more information, about the interaction between the transactional buffer and the draft database table, see Draft Runtime.

The following diagram shows the persistence of the transactional buffer on the draft table.



The Interaction Phase

The Save Sequence

The data changes during the interaction phase are saved to the persistent database table during the save sequence. It is invoked with the EML request `COMMIT ENTITIES`, which can be invoked by an EML consumer or by the RAP transactional engine, for example when processing OData changesets. For more detailed information about the save sequence and its phases, see Save Sequence Runtime.

The save sequence ensures that the BO instances of all involved BOs on the transactional buffer are in a consistent state and ready to be saved to the database. If this is not the case, the early phases (`FINALIZE` or `CHECK_BEFORE_SAVE`) of the save sequence fail and the `COMMIT ENTITIES` request returns `sy-subrc=4`. In this case, the saver method `CLEANUP_FINALIZE` for all touched BOs is called, which ensures the clearing of the transactional buffer for all BOs that are involved in the current LUW. Then, the save sequence is terminated without any database commit and the BO instance can be further modified to reach a consistent state.

If the early save phases are successful, the actual saving to the database is triggered and must be executed successfully. The responsible saver methods are `ADJUST_NUMBERS` and `SAVE`. The former draws final numbers in late numbering scenarios. The latter method finally saves the data and executes the database commit. Numbers of other referenced objects can be fetched via `CONVERT_KEY`. This successful `SAVE` terminates the SAP LUW. The `COMMIT ENTITIES` request returns `sy-subrc=0`. External work processes that rely on a successful database commit can be triggered.

RAP BOs can also fail during the late save phase (`SAVE`). A `COMMIT ENTITIES` request returns `sy-subrc=8`. In this case the current LUW is inconsistent and all changes must be rolled back to the last consistent state. Hence, it is not allowed to trigger any procedure that cannot be withdrawn by `ROLLBACK ENTITIES` at any point in time during the LUW, but especially not during the late save phase. If such a fail happens, the ABAP session can be rescued with an explicit subsequent `ROLLBACK ENTITIES` for EML, and subsequent OData change sets will be processed regularly.

Transactional Phases of the SAP LUW in RAP

The SAP LUW consists of two transactional phases:

- modify
- save

For more information, see Controlled SAP LUW.

These phases and their entailed checks are also used in RAP. However, they do not coincide with the two main parts in RAP, the interaction phase and the save sequence. The following table illustrates the mapping of the transactional phases of a SAP LUW to RAP.

Transactional Phases

Transactional Phase	Phase in an SAP LUW in RAP
modify	***RAP interaction phase***: During the RAP interaction phase, standard operations such as create, read, update, delete, or other operations can be triggered. Changes are stored in the transactional buffer.
	Early save phase (as part of the RAP save sequence): Final checks and changes can be made to data in the transactional buffer. There may be multiple parties implementing a business object, or even multiple business objects involved. Therefore, it is important that all of them can report if they are in a consistent state and if the changes can be persisted. In case of errors, it is possible to return to the interaction phase so that errors can be corrected. This is true, for example, for UI scenarios involving end users. It is expected that no user- or business-related errors will occur after the early save phase.
save	***Late save phase*** (as part of the RAP save sequence): The main task in the late save phase is to persist the consistent data in the transactional buffer to the database. In ABAP Cloud, this is done by direct database updates using ABAP SQL. In classic ABAP, you can also use the update task. Reaching the late save phase (if there are no errors in the early save phase) marks a point of no return. Unlike the early save phase, when you reach the late save phase, you cannot return to the interaction phase. The late save phase either ends with a successful commit, or the changes are rolled back and a runtime error occurs. The late save phase is followed by a cleanup phase that clears the transactional buffer.

Information Flow in RAP during a SAP LUW

Sometimes it is necessary to pass information related to a BO instance from the interaction phase to the save sequence, without the information being relevant for the data model. Such an information can be relevant to evaluate if external or asynchronous processes must be triggered during the late save phases.

Example

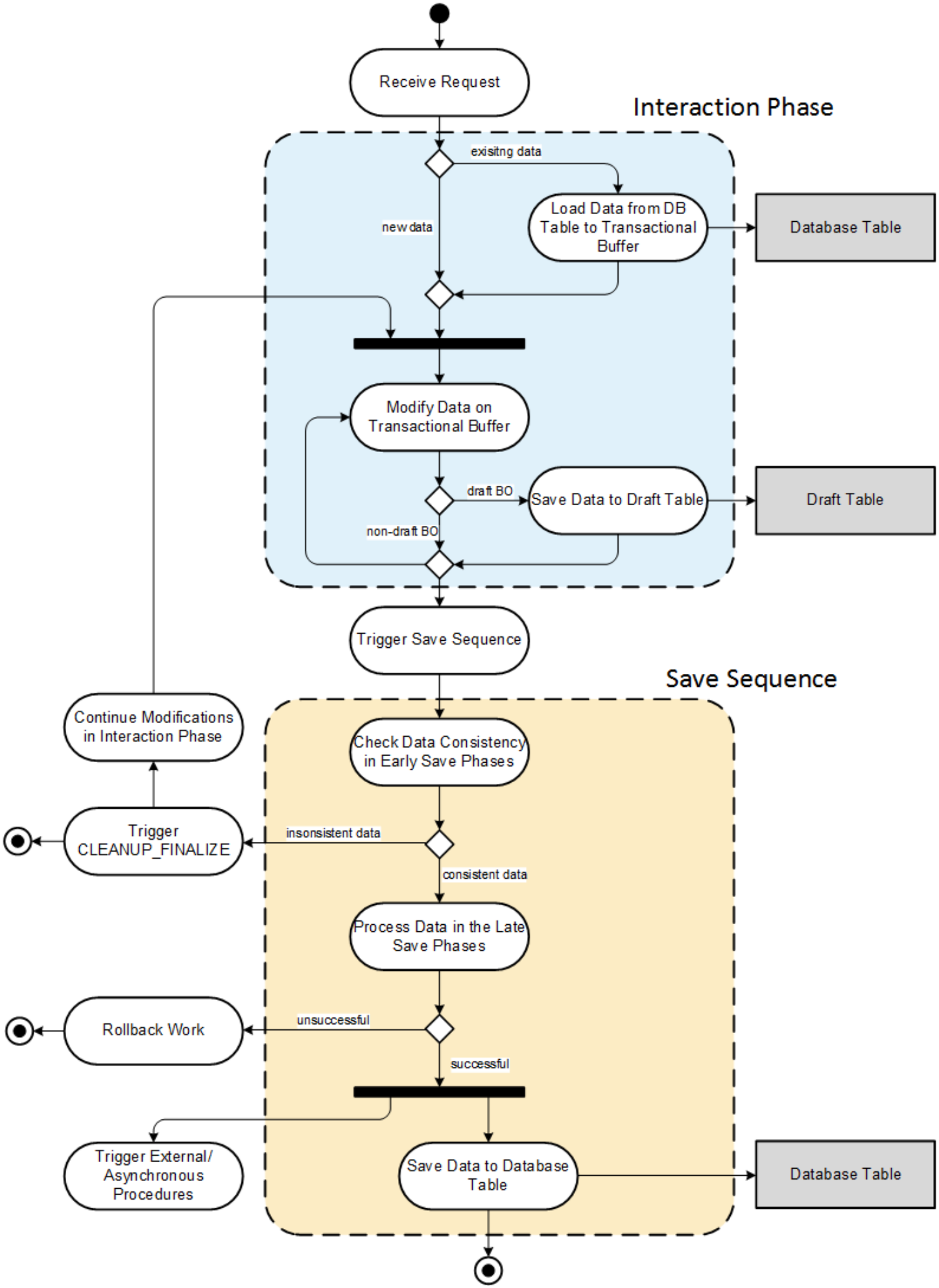
It might be necessary to inform an external RAP BO consumer about a change on the database. This notification process can only be triggered on saving the data to the database, because any step before can fail in the LUW and the consumer might be informed about changes that effectively did not happen. Therefore such processes or notifications can only be triggered during the actual save step or the additional save in the save sequence.

To evaluate if such a process must be triggered during the save phase in the save sequence, an auxiliary field can be added to the transactional data model in CDS. This field does not need to have a persistence on the database, but is only added in CDS as a transient field (technically done by a cast). This auxiliary field can be filled during the interaction phase with information that is relevant during the save phase in the save sequence and evaluated when the actual save happens.

> ⚠ **Caution**
>
> It must always be ensured that a procedure that is triggered during the late save phases can be rolled back entirely without any leftovers in the current SAP LUW.

RAP Activities during an SAP LUW

The following diagram shows the activities during an SAP LUW in RAP.



RAP Activities during a SAP LUW

Save Sequence Runtime

The save sequence is part of the business object runtime and is called after at least one successful modification was performed during the interaction phase.

The save sequence starts with `FINALIZE`, in which final calculations involving all BOs in the current LUW are executed before data can be persisted. These changes are done by determinations for managed implementation scenarios, or in the corresponding implementation method for unmanaged implementation scenarios. The `FINALIZE` step is the last option to change data on the transactional buffer. After the `FINALIZE` step, EML modify requests result in runtime errors. For more information, see FINALIZE.

ℹ Note

The SAVE sequence is always triggered, when a `MODIFY` request on a BO is executed, even when the BO buffer is not changed and solely delegates the request, for instance to another BO.

If the subsequent `CHECK_BEFORE_SAVE` step is positive for all transactionally involved business objects, the point-of-no-return is reached. From now on, a successful SAVE must be guaranteed for all involved BOs. The checks are performed via validations in managed implementation scenarios, or in the corresponding method for unmanaged implementation scenarios. For more information, see CHECK_BEFORE_SAVE.

After the point-of-no-return, the `ADJUST_NUMBERS` call can occur to take care of late numbering. For more information, see ADJUST_NUMBERS.

The SAVE step persists all BO instance data from the transactional buffer for all involved business objects in the database. For more information, see SAVE.

The `CLEANUP` step clears the transactional buffer. For more information, see CLEANUP.

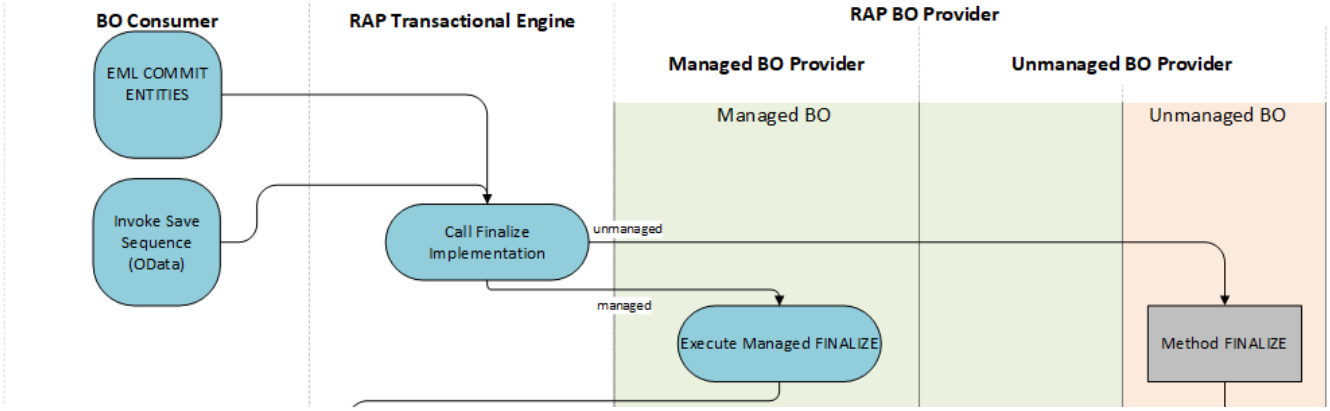
You can use the `CLEANUP_FINALIZE` method to clear the transactional buffer if the `FINALIZE` or the `CHECK_BEFORE_SAVE` fails in any BO that is involved in the current LUW. For more information, see CLEANUP_FINALIZE.

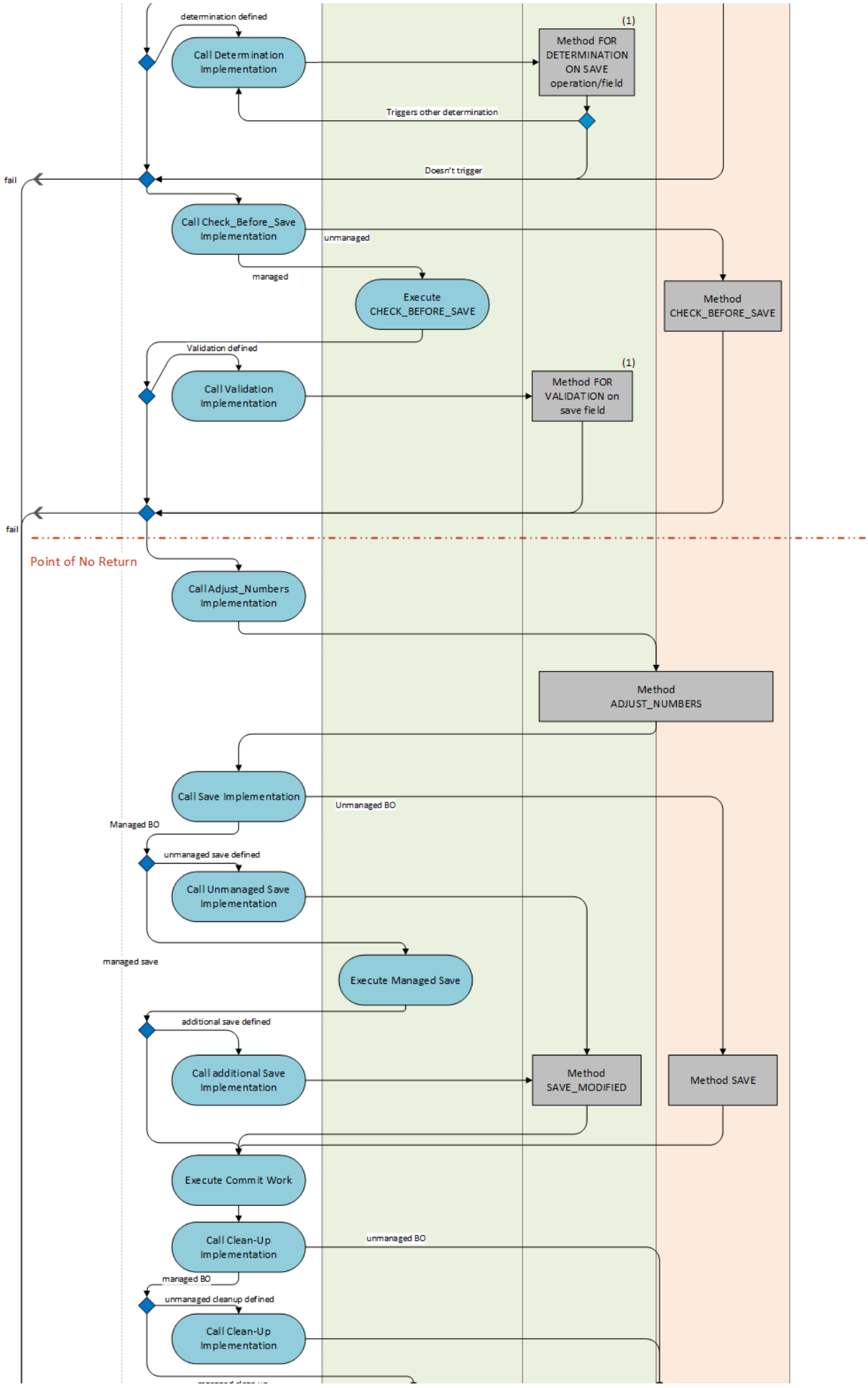
ℹ Note

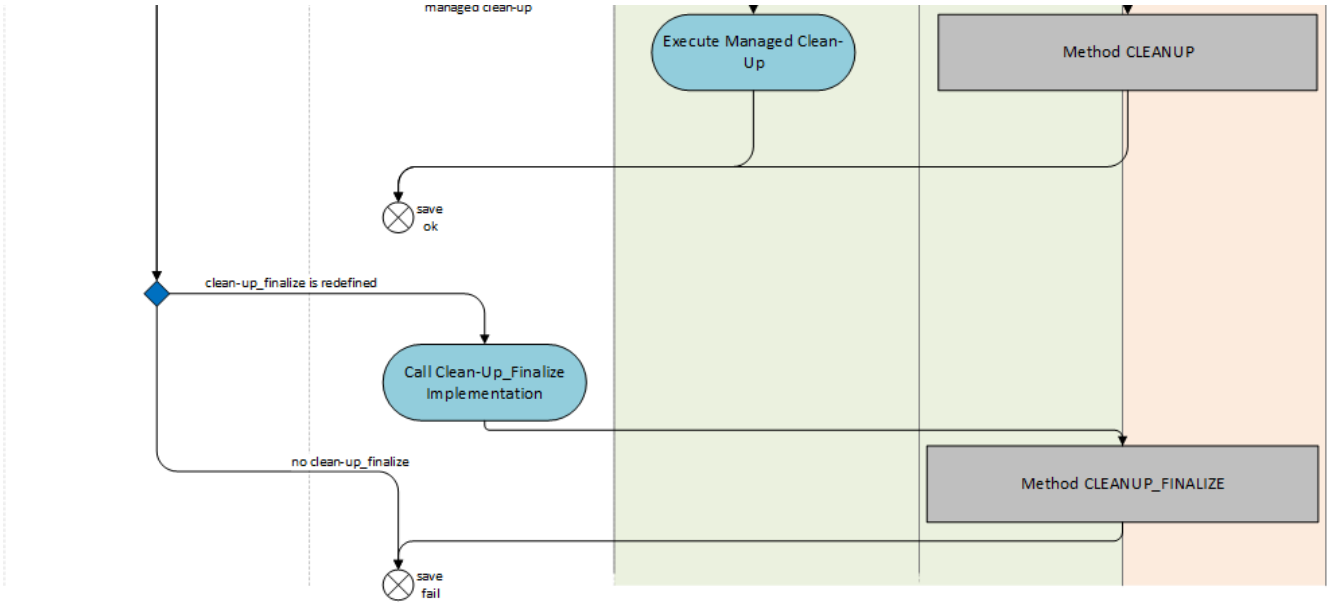
In transactions where multiple business objects are involved, the phases `FINALIZE` and `CHECK_BEFORE_SAVE` are called for all business objects that are registered for the save sequence. This also applies if one of these business objects fails in `FINALIZE` or `CHECK_BEFORE_SAVE`. The `CLEANUP_FINALIZE` then ensures for all involved business objects that their transactional buffers are cleared.

The following runtime diagram illustrates the main agents' activities during the save sequence.

This image is interactive. Hover over each area for a description. Click highlighted areas for more information.




determination defined
Call Determination Implementation
(1)
Method FOR DETERMINATION ON SAVE operation/field
Triggers other determination
Doesn't trigger
fail
Call Check_Before_Save Implementation
unmanaged
managed
Execute CHECK_BEFORE_SAVE
Method CHECK_BEFORE_SAVE
Validation defined
Call Validation Implementation
(1)
Method FOR VALIDATION on save field
fail
Point of No Return
Call Adjust_Numbers Implementation
Method ADJUST_NUMBERS
Call Save Implementation
Unmanaged BO
Managed BO
unmanaged save defined
Call Unmanaged Save Implementation
managed save
Execute Managed Save
additional save defined
Call additional Save Implementation
Method SAVE_MODIFIED
Method SAVE
Execute Commit Work
Call Clean-Up Implementation
unmanaged BO
managed BO
unmanaged cleanup defined
Call Clean-Up Implementation



Please note that image maps are not interactive in PDF output.

(1):

- The runtime of the save sequence may vary with respect to determinations and validations depending on whether the optimized variant of the `activate` action is used or not. For more information, see Draft Action Activate Optimized.
- For performance reasons, determinations and validations are not executed during the save sequence, if these determinations and validations have already been executed during a previous determine action that has been executed on the active instance in the same transaction. This performance optimization is not applied in the following cases:
  - A modification performed after the determine action triggers a determination/validation.
  - A validation in the determine action reports a failed key.

    In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

A detailed description of the activities during the interaction is illustrated in separate diagrams, see

- Create Operation Runtime
- Update Operation Runtime
- Delete Operation Runtime
- Action Runtime

Related Information

FINALIZE
CHECK_BEFORE_SAVE
ADJUST_NUMBERS
SAVE
CLEANUP
CLEANUP_FINALIZE
Determinations
Validations

FINALIZE

Finalizes data changes before they can be persisted on the database.

The `FINALIZE` implementation is called as a first step in the save sequence. This step executes final calculations based on the results of the interaction phase for all involved BOs in the current LUW. `FINALIZE` is the last chance to change BO instances on the transactional buffer via EML modify calls. After this step, EML modify requests result in runtime errors.

Managed BOs

In managed BOs, final modifications are done via determinations, see Determinations.

- The runtime of the save sequence may vary with respect to determinations and validations depending on whether the optimized variant of the `activate` action is used or not. For more information, see Draft Action Activate Optimized.
- For performance reasons, determinations and validations are not executed during the save sequence, if these determinations and validations have already been executed during a previous determine action that has been executed on the active instance in the same transaction. This performance optimization is not applied in the following cases:
  - A modification performed after the determine action triggers a determination/validation.
  - A validation in the determine action reports a failed key.

    In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

Unmanaged BOs

In unmanaged BOs, final calculations are done in the corresponding method in the behavior pool.

Method `FINALIZE`

The method `FINALIZE` must be redefined in the local saver class to implement it.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS finalize REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD finalize.
// ** implement finalize **//
ENDMETHOD.
```

Changing Parameters

- `FAILED`

  The parameter `FAILED` is filled to log the entities for which `FINALIZE` went wrong.
- `REPORTED`

  You can fill the parameter `REPORTED` to return messages in case of failure.

Related Information

CHECK_BEFORE_SAVE

Checks the transactional buffer for consistency.

The CHECK_BEFORE_SAVE implementation is called during the save sequence before the point-of-no-return. This step checks the consistency of the transactional buffer to ensure a successful save.

If the check for all involved BOs returns positive feedback based on all transactional changes, the **point-of-no-return** is reached. From now on, a successful SAVE must be guaranteed for all involved BOs and the data is persisted.

If on the other hand, errors are reported in the changing parameter FAILED, the save sequence is aborted.

Managed BOs

In managed BOs, the final check for all involved BOs is done via validations, see Validations.

- The runtime of the save sequence may vary with respect to determinations and validations depending on whether the optimized variant of the `activate` action is used or not. For more information, see Draft Action Activate Optimized.
- For performance reasons, determinations and validations are not executed during the save sequence, if these determinations and validations have already been executed during a previous determine action that has been executed on the active instance in the same transaction. This performance optimization is not applied in the following cases:
    - A modification performed after the determine action triggers a determination/validation.
    - A validation in the determine action reports a failed key.

      In order to ensure data consistency, make sure that your determinations and validations follow the respective modelling guidelines described in Determination and Validation Modelling.

Unmanaged BOs

In unmanaged BOs, the final check for all involved BOs is done in the corresponding method in the behavior pool.

Method CHECK_BEFORE_SAVE

The method CHECK_BEFORE_SAVE must be redefined in the local saver class to implement it.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS check_before_save REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD check_before_save.
// ** implement check_before_save here **//
ENDMETHOD.
```

Changing Parameters

- FAILED

  The parameter FAILED is filled to log the entities for which there is no positive feedback.

- REPORTED

  You can fill the parameter REPORTED to return messages in case of failure.

Related Information

Implicit Response Parameters

ADJUST_NUMBERS

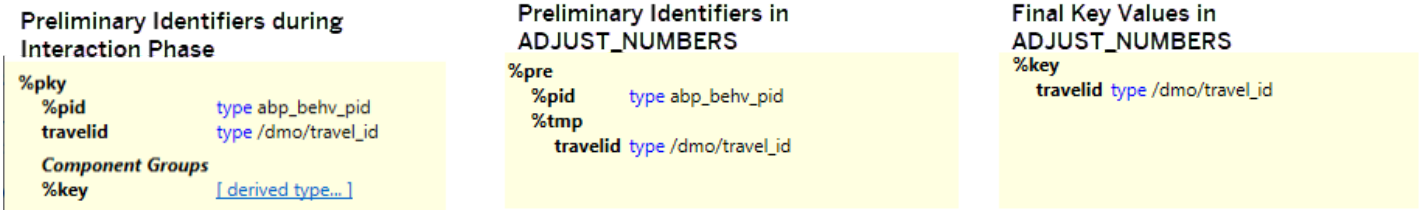
Implements late numbering.

The ADJUST_NUMBERS implementation is called during the save sequence after the point-of-no-return. At this point in time, the transactional buffer is in a consistent state and all instances can receive their final number.

The implementation of ADJUST_NUMBERS is only possible if late numbering is modeled in the behavior definition. For more information, see Late Numbering. The method must be implemented by the application developer in late numbering scenarios. Usually the final key value is assigned by number range objects and the adjusted number is then mapped in the mapped response parameter.

In scenarios with late numbering, BO instances are identified via the transactional key (%tky) during the interactions phase. In late numbering scenarios %tky contains %pky, which contains at least two components: %pid and %key, the actual key fields. Both components might contain preliminary key values, which must be mapped to the final keys in ADJUST_NUMBERS.

The values of the component group %pky, which is used during the interaction phase for preliminary identification, are assigned to the component group %pre in ADJUST_NUMBERS. To differentiate potential preliminary key values in the key fields from the actual final key values, the preliminary in-place key values are assigned to the component group %tmp in ADJUST_NUMBERS. The final keys in %key must be mapped to the preliminary identifiers in %pre in ADJUST_NUMBERS. The following compilation illustrates the identifiers during the interaction phase and in ADJUST_NUMBERS.



Identifiers in Late Numbering Scenarios

The mapping in ADJUST_NUMBERS is done from %pre to %key in the mapped response parameter. The following example shows a mapping, where both components of %pky are used for unique identification during the interaction phase. This isn't mandatory, you can also use only %pid or the key fields.

Example

```
  METHOD adjust_numbers.
...
* Fill %key-<key_field> with the adjusted number
    APPEND VALUE #( %pre-%pid               = 'MyPID'
                    %pre-%tmp-<key_field>     = 'MyPreliminaryKey_InPlace'
                    %key-<key_field>          = 'MyFinalKey'
                  ) TO mapped-<entity>.

  ENDMETHOD.
```

For more information about derived type components, see Components of BDEF Derived Types (ABAP - Keyword Documentation).

Implementation

The method ADJUST_NUMBERS must always be implemented when late numbering is used.

Method ADJUST_NUMBERS

The method for ADJUST_NUMBERS must be redefined in the local saver class to implement it.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS adjust_numbers REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD adjust_numbers.
// ** implement adjust_numbers here **//
ENDMETHOD.
```

Changing Parameter

- MAPPED
- REPORTED

  Messages can be reported via the implicit changing parameter REPORTED. As consumer errors must not appear after CHECK_BEFORE_SAVE, REPORTED should only contain success or information messages, such as *Material stock is low*.

Note

The method must not fail and thus needs no FAILED parameter, as the exchange of temporary IDs takes place after the point-of-no-return. If the application needs to stop the transaction, it can only produce a short dump.

Related Information

Implicit Response Parameters

SAVE

Saves the data from the transactional buffer to the database.

The SAVE implementation is called during the save sequence to save the current state of the transactional buffer to the database. All business object instances (including instances from cross-BO relationships) that are involved in the current LUW are saved.

Managed BOs

In managed business objects, the SAVE is done by the managed BO provider.

If the business scenario requires an alternative save implementation, it is possible to define an additional or an unmanaged save in the managed scenario. This is done by adding the following syntax in the behavior definition.

```
define behavior for Entitiy [alias Alias]
implementation in class ABAP_CLASS unique
...
with additional | unmanaged save [with full data]
...
{ … }
```

The option is available for each BO entity separately.

The implementation is done in the corresponding method `save_modified` in the behavior pool. If the addition `with full data` is used in the behavior definition, once an instance is modified, then not only the changed fields but all fields are provided into the `save_modified` method.

It is also possible to self-implement a cleanup, if an additional or unmanaged save is defined, see CLEANUP.

Method SAVE_MODIFIED

The method for SAVE_MODIFIED must be redefined in the local saver class to implement it.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS save_modified REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD save|save_modified..
// ** implement save here **//
ENDMETHOD.
```

Importing Parameters

The method imports a parameter for all defined modify operations.

- CREATE

  This parameter entails information of instances that were created during the LUW.

- UPDATE

  This parameter entails information of instances that were updated during the LUW.

- DELETE

  This parameter entails information of instances that were deleted during the LUW.

Changing Parameter

- REPORTED

  Messages can be reported via the implicit returning parameter REPORTED. As consumer errors must not appear after CHECK_BEFORE_SAVE, REPORTED should only contain success or information messages, such as *Booking has been*

saved.

> **i Note**
>
> The method must not fail and thus needs no `FAILED` parameter. If the application needs to stop the transaction, it can only raise a short dump.

Unmanaged BOs

For unmanaged business objects, the SAVE must always be implemented by the application developer.

The implementation is done in the corresponding method in the behavior pool.

Method SAVE

The method SAVE must be redefined in the local saver class to implement it.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS save REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD save.
// ** implement save here **//
ENDMETHOD.
```

Changing Parameter

The changing parameter `REPORTED` behaves like the one in `SAVE_MODIFIED`, see Changing Parameters in SAVE.

Related Information

Implicit Response Parameters

CLEANUP

Clears the transactional buffer.

The cleanup implementation is called after the save sequence has run, or after a `ROLLBACK ENTITIES` statement for any kind of instance access during the interaction phase. The cleanup empties the transactional buffer from any remaining instance data to terminate the LUW. It is called whenever instance data was written to the application buffer. That means, it is called for all touched BOs during the interaction phase, no matter if failed or successful.

After the data is persisted, or the interaction has failed, it is expected that the transactional buffer is cleared, since the same ABAP session might be used for more than one LUW and any remaining changes in the transactional buffer could lead to inconsistencies.

Managed BOs

In managed business objects, the `CLEANUP` is done by the managed BO provider.

If an additional or an unmanaged save is implemented in a managed BO, the cleanup step can also be implemented by the application developer. This is done by adding the following syntax in the behavior definition.

```
define behavior for Entitiy [alias Alias]
implementation in class ABAP_CLASS unique
...
with additional | unmanaged save and cleanup
...
{ … }
```

The implementation is done in the corresponding method in the behavior pool.

Unmanaged BOs

For unmanaged business objects, the CLEANUP must always be implemented by the application developer.

The implementation is done in the corresponding method in the behavior pool.

Method CLEANUP

The CLEANUP method must be redefined in the local saver class to implement it.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS save|save_modified REDEFINITION.
    METHODS cleanup REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD cleanup.
// ** implement clean up here **//
ENDMETHOD.
```

This method does not have any parameters.

CLEANUP_FINALIZE

Clears the transactional buffer if the phase FINALIZE or CHECK_BEFORE_SAVE fails.

The method CLEANUP_FINALIZE can be redefined in the saver class of an ABAP behavior pool to implement the buffer cleanup. It is only called if the phase FINALIZE or CHECK_BEFORE_SAVE of any BO that is involved in the current LUW returns failed keys.

Typically, CLEANUP_FINALIZE is used in cross-BO consumption scenarios with dependencies in the early save phases. You can use this method to revoke changes done during FINALIZE if necessary. In most cases changes done during FINALIZE do not harm if the save is rejected and the transaction returns to the interaction phase

Method CLEANUP_FINALIZE

To use the method CLEANUP_FINALIZE, redefine the method in the saver class of the ABAP behavior pool. It does not have to be defined in the behavior definition. If this method is not explicitly redefined, the buffer clean-up only happens after the save

sequence by OData explicitly or by an explicit `COMMIT ENTITIES` via EML.

```
CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.

  PROTECTED SECTION.

    METHODS cleanup_finalize REDEFINITION.

ENDCLASS.


CLASS lcl_save IMPLEMENTATION.
METHOD cleanup_finalize.
// ** implement cleanup_finalize  here **//
ENDMETHOD.
```

This method does not have any parameters.

Entity Manipulation Language (EML)

The Entity Manipulation Language (EML) is a part of the ABAP language that enables access to RAP business objects.

Because the consumption of business objects via the OData protocol requires a Fiori UI or a web API, EML enables a type-safe access to business objects directly by using ABAP. EML interacts with business objects by triggering their operations for specified entities. An operation can only be triggered by EML if the operation is specified for the relevant entity in the behavior definition and if it is implemented accordingly.

Use Cases

EML can be used to provide behavior for business objects. For example, you can implement an action that first triggers a create and then an update operation using EML. This action enables the creation of instances with default values and is implemented in the behavior pool of the business object for which the behavior is to be provided.

EML can also be used to consume business objects. Typical scenarios are unit's tests that are implemented in a class separate from the business objects to be tested. These unit tests use EML to check the transactional behavior of business objects by triggering their operations. Other scenarios involve cross BO applications, where business objects use each others' operations to implement their behavior or the consumption of released business objects.

ℹ Note

Apart from the standard EML API, which is described in this guide, there's also a generic EML API. The generic EML API enables the generic integration of business objects in other frameworks and isn't covered by this guide.

EML Scenarios

The following table gives you an overview of the different EML scenarios, what to take into account and where to find examples and further information:

Scenario	Description	Relevant Statements

Scenario	Description	Relevant Statements
Develop (BO with in `IN LOCAL MODE` in its own behavior pool)	In an EML develop scenario, you use EML to provide functionality for your own business object in its own behavior pool. When the EML accesses instances of the respective BO with `READ/READ-BY-ASSOCIATION`, `CREATE/CREATE-BY-ASSOCIATION`, `UPDATE`, `DELETE`, and `EXECUTE`, you can add `IN LOCAL MODE` to bypass access controls, as well as authorization and feature control implementations. That means , for example, that you can update a field that is defined as `READONLY` in an EML develop scenario.	• READ • READ BY Association • CREATE and CREATE BY Association • UPDATE • DELETE • EXECUTE • AUGMENTING BY Association
Consume (Business Object from other behavior pool/released business objects)	In an EML consumption scenario, you can use EML to consume business objects in cross BO relationships or you can consume released business objects. In consumption scenarios, you can't add `IN LOCAL MODE` to the statements to bypass access controls, as well as authorization and feature control implementations.	• READ • READ BY Association • CREATE and CREATE BY Association • UPDATE • DELETE • EXECUTE • GET PERMISSIONS • SET LOCKS
Test	In an EML test scenario, you use EML to test a business object outside of an ABAP behavior pool. That requires using `COMMIT ENTITIES` to end the RAP LUW. When your're implementing inside the behavior pool, this is handled by the framework.	• READ • READ BY Association • CREATE and CREATE BY Association • UPDATE • DELETE • EXECUTE • GET PERMISSIONS • SET LOCKS • COMMIT ENTITIES • ROLLBACK ENTITIES

EML Statements

This section provides an overview on operations that can be triggered using the respective EML statements. For detailed information as well as code examples for each statement, see Examples for Accessing Business Objects with EML.

For more information about the syntax, see ABAP for Consuming RAP Business Objects (ABAP Keyword Documentation).

READ

The READ statement includes all operations that don't change data of entity instances (read-only access). It's possible to read data from the selected entities and from associated entities. You can trigger multiple read operations within one READ statement.

For an implemented example, see READ.

For more information about the syntax, see READ ENTITY, ENTITIES (ABAP Keyword Documentation) .

MODIFY

The MODIFY statement includes all operations that change data of entity instances. You can trigger the standard operations CREATE, UPDATE, and DELETE as well as the nonstandard operations actions and functions. For BOs containing a projection layer, you can furthermore augment incoming requests by adding data or modifying operations before the requests are passed to the base business object. An example use case is augmenting a create request with instance data in order to set default values for new instances in the augmentation implementation. It's possible to trigger multiple modify operations by one MODIFY statement. If these operations don't depend on each other (e.g. multiple update operations on different entities), their execution sequence isn't determined. To ensure a certain execution sequence in this case, use separate MODIFY statements.

> **Note**
>
> If a MODIFY statement triggers a create and a create-by-association operation using `%cid_ref` in the same call, instance feature and instance authorization implementations are not executed. Depending on the business logic, you need to implement respective checks by means of prechecks and/or validations.

For an implemented example, see MODIFY.

For more information about the syntax, see MODIFY ENTITY, ENTITIES (ABAP Keyword Documentation) .

COMMIT ENTITIES

The `COMMIT ENTITIES` statement triggers the persistence of data from the transactional buffer to the database for all business objects that were changed within the LUW.

In the context of an application consisting of RAP and non-RAP implementations, the statement `COMMIT WORK` implicitly triggers `COMMIT ENTITIES` and with it the RAP save sequence to persist all the data from the transactional buffer to the database. If the saving of data from the RAP implementation fails, for example because validations triggered during the save sequence, the commit is aborted and all changes committed until this point are rolled back to ensure data consistency. The `COMMIT WORK` is aborted when the `COMMIT ENTITIES` fails. To avoid this, you can use `COMMIT ENTITIES IN SIMULATION MODE` to check if the save sequence runs without a problem. The simulation mode triggers the save sequence, but doesn't persist any data.

For an example for `COMMIT ENTITIES`, see COMMIT ENTITIES.

For more information about the syntax, see COMMIT ENTITIES (ABAP Keyword Documentation).

For an example for `COMMIT ENTITIES in SIMULATION MODE` refer to ABAP EML - COMMIT ENTITIES IN SIMULATION MODE (ABAP Keyword Documentation).

ROLLBACK ENTITIES

The `ROLLBACK ENTITIES` statement rolls back all changes since the last commit and resets the transactional buffer.

For an example, see ROLLBACK ENTITIES.

For more information about the syntax, see ROLLBACK ENTITIES (ABAP Keyword Documentation).

GET PERMISSIONS

The GET PERMISSIONS statement is used to retrieve information about feature control and authorization permissions from business objects on global and instance level.

For an example, see GET PERMISSIONS.

For more information about the syntax, see GET PERMISSIONS (ABAP Keyword Documentation) .

SET LOCKS

The SET LOCKS statement locks entity instances and thus prevents their modification by other users.

For an example, see SET LOCKS.

For more information about the syntax, see SET LOCKS (ABAP Keyword Documentation) .

EML Operands

IN LOCAL MODE

Implementing IN LOCAL MODE is only possible if the EML implementation consumes instances of a RAP BO in its own behavior pool. The addition IN LOCAL MODE ensures that a BO is accessed directly without checking access control, authorization control or feature control or prechecks. For more information about the syntax, see IN LOCAL MODE (ABAP Keyword Documentation) and EML Operands .

IN LOCAL MODE has the following effects for the different operations:

Operation	IN LOCAL MODE Effect	Affected Methods in Behavior Pool	Example

• READ
• READ BY ASSOCIATION
• EXECUTE FUNCTION
	IN LOCAL MODE bypasses::
	• Access Controls
	• Authorization Checks
		• **Access Controls:** Independent of Behavior Pool
		• **Authorization Checks:**
		  ◦ Method FOR GLOBAL AUTHORIZATION
		  ◦ Method FOR INSTANCE AUTHORIZATION
			• If an EML READ acceses data sources with defined access controls and the operand IN LOCAL MODE, all data sets are returned independently of the defined access controls.

Operation	`IN LOCAL MODE` Effect	Affected Methods in Behavior Pool	Example

• `CREATE`
• `UPDATE`
• `DELETE`
• `EXECUTE ACTION`
	`IN LOCAL MODE` bypasses:
	• Authorization Checks
	• Feature Control
	• Precheck on projection and base BO level
		• **Authorization Checks:**
		  ◦ Method `FOR GLOBAL AUTHORIZATION`
		  ◦ Method `FOR INSTANCE AUTHORIZATION`
		• **Feature Control:**:
		  ◦ Method `FOR GLOBAL FEATURES`
		  ◦ Method `FOR INSTANCE FEATURES`
		• **Precheck**:
		  ◦ Method `FOR PRECHECK` (Projection Layer)
		  ◦ Method `FOR PRECHECK` (Base BO Layer)
			• You can update a `read only` field via EML with `IN LOCAL MODE`.

For an example implementation with `IN LOCAL MODE`, see READ

`PRIVILEGED`

Implementing `PRIVILEGED` enables the consumer for privileged access, e.g. to circumvent authority checks.

As a consumer, you can use privileged access to a RAP BO to circumvent authority checks for a BO. The privileged mode must have been defined by the provider using authority contexts in the RAP BO interface that you want to consume. `PRIVILEGED` can only be used, if the RAP BO entity you want to consume is defined with `with privileged mode`.

For more information about syntax, see ABAP EML - PRIVILEGED (ABAP Keyword Documentation).

Basic Statement Components

Despite variety of functions, EML statements usually contain a set common components.

```
MODIFY ENTITIES OF Business_Object
ENTITY BO_Entity
Operation FIELDS ( RelevantField1 RelevantField2 ) WITH internal_table | FROM internal_table
[FAILED failed]
[MAPPED mapped]
[REPORTED reported].
```

Every EML statement starts with a keyword for the statement type indicating the purpose of the EML statement, e.g. or specifiestheof theEMLREADor one or multiple entities whose instances are to be worked on. For statements like `MODIFY` covering different operation types, the actual operation of the EML statement is indicated after the entity specification part.

Depending on the operation, you must pass relevant values to identify the related instance or to pass values for the instances. EML expects that you explicitly mark the fields that are relevant for the operation and for which the values should be used. These fields can be either specified with the fields statement or by filling the control structure.

The responses of the operations triggered by an EML statement are stored in typed response parameters. The structure `failed` for example is used to store the keys of instances for which the triggered operations have failed whereas the table `result` contains the results returned by a READ operation or by an action.

Examples for Accessing Business Objects with EML

This topic offers code examples to demonstrate how you can access business objects using the Entity Manipulation Language (EML).

Reference Business Object

The following sections cover different EML statements that you can use for interacting with business objects. Unless otherwise stated, the described EML statements derive from or refer to the draft business object described in the development guide Developing Transactional Apps with Draft Capabilities.

EML can be used to trigger operations specifically for draft instances. Moreover, EML can trigger draft actions that consider the specific characteristics of draft enabled business objects. For information on the draft-specific usage of EML, see Draft.

READ

General Information: READ

The READ statement provides read access to entity instances and returns the requested instances, fields, and state messages. This statement can be used as a basic operation that provides necessary data to work with in subsequent business logic. In case of draft-enabled business objects, the READ returns the state messages belonging to an instance.

For more information about the syntax, see READ ENTITY, ENTITIES (ABAP Keyword Documentation) .

For more information regarding the implementation contract, refer to Implementation Contract: READ.

Implementation with `IN LOCAL MODE`

Implementing `IN LOCAL MODE` is only possible if the EML implementation consumes instances of a RAP BO in its own behavior pool. The addition `IN LOCAL MODE` ensures that a BO is accessed directly without checking access control, authorization control or feature control or prechecks. For more information about the syntax, see IN LOCAL MODE (ABAP Keyword Documentation) and EML Operands .

Consumption

BO external consumption includes scenarios like unit tests that are implemented in a class separate from the business objects to be tested. Other scenarios are cross BO applications, where business objects use each others' operations to implement their behavior or the consumption of released business objects in the behavior pool of a custom RAP BO. In these cases, access controls, authorization control and feature control implementations are always triggered during a request and can't be bypassed. When consuming released business objects, only the released part of the BO (projection) can be consumed in the behavior pool of a custom RAP BO.

Example: READ

The following statement reads the begin date, the end date, and the semantic ID of travel instances identified by the importing parameter `keys`. In the behavior pool of the reference business object, this `READ` statement is used in the validation `validateDates` to provide the data to be validated in subsequent checks. For further contextual information and for the complete code of this validation, see Validate Dates.

```
READ ENTITIES OF /DMO/R_Travel_D IN LOCAL MODE
    ENTITY Travel
      FIELDS (  BeginDate EndDate TravelID )
      WITH CORRESPONDING #( keys )
    RESULT DATA(travels)
   FAILED DATA(failed)
   REPORTED DATA(reported).
```

The `READ` statement is introduced by the keywords `READ ENTITIES OF` followed by the behavior definition of the business object the statement refers to.

Since this statement is contained in the behavior pool of the business object it refers to, the keywords `IN LOCAL MODE` are used. They ensure priviledged access to the underlying CDS views that returns all data sets even if access controls are defined in a DCL.

The statement at hand reads the data of travel instances, hence the `ENTITY Travel` is specified. Here, we use the alias specified for the entity in the behavior definition.

The next line states the fields whose values are to be included in the read result.

The keywords `WITH CORRESPONDING #( keys )` indicate that the read operation will be performed on instances with the primary keys contained in the internal table `keys`. This internal table is provided as an importing parameter of the validation method.

The internal table specified after the keyword `RESULT` acts as a response parameter that contains the result of the read operation. The type of this response parameter is derived from the specified entity and from the triggered read operation by the framework.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

READ BY Association

General Information: READ BY Association

The `READ BY association` statement uses an association of the entity specified in the EML statement in order to read instances of an associated entity.

For more information about the syntax, see READ ENTITY, ENTITIES (ABAP Keyword Documentation) .

For more information about the implementation contract, refer to Implementation Contract: READ-BY-ASSOCIATION.

Implementation with `IN LOCAL MODE`

Implementing `IN LOCAL MODE` is only possible if the EML implementation consumes instances of a RAP BO in its own behavior pool. The addition `IN LOCAL MODE` ensures that a BO is accessed directly without checking access control, authorization control or feature control or prechecks. For more information about the syntax, see IN LOCAL MODE (ABAP Keyword Documentation) and EML Operands .

Consumption

BO external consumption includes scenarios like unit tests that are implemented in a class separate from the business objects to be tested. Other scenarios are cross BO applications, where business objects use each others' operations to implement their behavior or the consumption of released business objects in the behavior pool of a custom RAP BO. In these cases, access controls, authorization control and feature control implementations are always triggered during a request and can't be bypassed. When consuming released business objects, only the released part of the BO (projection) can be consumed in the behavior pool of a custom RAP BO.

Example: `READ BY` Association

The following statement uses the to-parent-association of booking instances identified by the importing parameter `keys` in order to read the UUID of their travel instance. In the behavior pool of the reference business object, this `READ` statement is used in the determination `calculateTotalPrice` of the booking entity to determine that travel UUID, for which a travel price recalculation needs to be executed. For further contextual information and for the complete code of this determination, see Calculate Total Price.

```
  READ ENTITIES OF /DMO/R_Travel_D IN LOCAL MODE
      ENTITY Booking BY \_Travel
        FIELDS ( TravelUUID  )
        WITH CORRESPONDING #(  keys  )
      RESULT DATA(travels)
LINK DATA(link)
FAILED DATA(failed)
REPORTED DATA(reported).
```

The read statement is introduced by the keywords `READ ENTITIES OF` followed by the behavior definition of the business object the statement refers to.

Since this read statement is contained in the behavior pool of the business object it refers to, the keywords `IN LOCAL MODE` are used. They ensure priviledged access to the underlying CDS views that returns all data sets even if access controls are defined in a DCL.

The statement at hand reads the data of travel instances, but the association specified for the booking entity is used. Hence, the `ENTITY Booking` is specified. Here, we use the alias specified for the entity in the behavior definition.

To indicate that the read operation is to be performed by an association, the syntax `BY \` is used. After that, the name of the association specified in the according CDS entity is used.

The next line states that only the value of the field `TravelUUID` is to be included in the read result.

The keywords `WITH CORRESPONDING #( keys )` indicate that the read operation will be performed for instances with those primary keys that are contained in the internal table `keys`. This internal table is provided as an importing parameter of the determination method.

The internal table specified after the keyword `RESULT` acts as a response parameter which contains the result of the read operation. The type of this response parameter is derived from the specified entity and from the triggered read operation by the framework.

The internal table declared after the keyword `LINK` acts as another response parameter. It contains the primary keys of the source and the target entity instances involved in the read-by-association operation. Since the association `_Travel` is used in this case, the source is a booking instance and the target is a travel instance. The type of this response parameter is derived from the specified entity and from the triggered read operation by the framework. The response parameter `LINK` is not used in the determination mentioned above.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

MODIFY

The `MODIFY` statement is used to change data of entity instances. The following sections provide examples for operations that can be triggered using this statement.

For more information about the syntax, see MODIFY ENTITY, ENTITIES (ABAP Keyword Documentation) .

Implementation with `IN LOCAL MODE`

Implementing `IN LOCAL MODE` is only possible if the EML implementation consumes instances of a RAP BO in its own behavior pool. The addition `IN LOCAL MODE` ensures that a BO is accessed directly without checking access control, authorization control or feature control or prechecks. For more information about the syntax, see IN LOCAL MODE (ABAP Keyword Documentation) and EML Operands .

Consumption

BO external consumption includes scenarios like unit tests that are implemented in a class separate from the business objects to be tested. Other scenarios are cross BO applications, where business objects use each others' operations to implement their behavior or the consumption of released business objects in the behavior pool of a custom RAP BO. In these cases, access controls, authorization control and feature control implementations are always triggered during a request and can't be bypassed. When consuming released business objects, only the released part of the BO (projection) can be consumed in the behavior pool of a custom RAP BO.

General Information: CREATE and CREATE BY Association

The `CREATE` statement enables the creation of entity instances. The `CREATE BY association` statement uses an association of the entity specified in the EML statement in order to create instances of its child entity.

For more information about the implementation contract, refer to Implementation Contract: CREATE and Implementation Contract: CREATE-BY-ASSOCIATION.

Example: `CREATE` and `CREATE BY` Association

The `MODIFY` statement below involves two create operations. The first one creates an instance of the entity travel using the prepopulated internal table `create`. The second one triggers a create-by-association operation that creates two booking instances based on the created travel instance. The following code can be used in a separate unit test class to test the create operations of the reference business object.

```
*Declare internal table using derived type
    DATA create TYPE TABLE FOR CREATE /DMO/R_Travel_D.

*Select valid flight data
    SELECT SINGLE AirlineID, ConnectionID, FlightDate FROM /DMO/I_Flight INTO @DATA(flight).

*Populate internal table for travel instance
```

```
    create = VALUE #( ( %cid                = 'create_travel'
                        %is_draft           = if_abap_behv=>mk-off
                        CustomerID          = '1'
                        AgencyID            = '70006'
                        BeginDate           = flight-FlightDate
                        EndDate             = flight-FlightDate
                        Description         = 'Travel 1'
 ) ).


*Create a travel instance and two associated booking instances
    MODIFY ENTITIES OF /DMO/R_Travel_D
        ENTITY Travel
          CREATE FIELDS ( CustomerID AgencyID BeginDate EndDate Description ) WITH create
          CREATE BY \_Booking
          FIELDS ( CustomerID AirlineID ConnectionID FlightDate ) WITH

                  VALUE #( ( %cid_ref = 'create_travel'

                                        %target = VALUE #( ( %cid         = 'create_booking_1'

                                         %is_draft    = if_abap_behv=>mk-off

                                         CustomerID   = '1'

                                                AirlineID    = flight-AirlineID

                                                                                ConnectionID = fl

                                                                                FlightDate   = fl

                                                                              ( %cid         = 'c

                                                                                %is_draft    = if

                                                                                CustomerID   = '1

                                                                                AirlineID    = fl

                                                                                ConnectionID = fl

                                                                              FlightDate   = flig

                                                                               ) ) )

    MAPPED DATA(mapped)
    REPORTED DATA(reported)
    FAILED DATA(failed).


*Persist transactional buffer
    COMMIT ENTITIES.
```

Before an internal table can be used for creating the travel instance, the type of this internal table needs to be derived from the corresponding CDS entity and its behavior definition. The modify operation the internal table is used for needs to be stated in its type declaration as well. As a result, the declared internal table contains all CDS elements and components necessary to perform the intended modify operation.

To ensure the usage of valid flight data, we now select the airline ID, the connection ID, and the flight date from an arbitrary row of the flight database and save its values in the local structure `flight`.

In the next step, the internal table is populated with the values for the travel instance using the `value` operator. Here, we also use the values from the structure `flight`. Since the target business object supports managed internal numbering, the primary key values don't need to be specified in the internal table. Instead, the content ID for the travel instance provided, which can be freely selected. A content ID is used as preliminary identifier as long as no primary key for the instance exists. In the example, it is referred to in the subsequent create-by-association operation. Using the draft indicator `%is_draft` we can determine whether we want to create active or draft instances. Since we want to create active instances, we set the draft indicator to false.

The `MODIFY` statement is introduced by the keywords `MODIFY ENTITIES OF` followed by the behavior definition of the business object the statement refers to.

The statement at hand triggers the create and the create-by-association operation of the travel entity, hence the `ENTITY Travel` is specified. Here, we use the alias specified for the entity in the behavior definition.

The keywords `CREATE FIELDS` are followed by the names of the fields to be populated. After the keyword `WITH` we specify the name of the internal table based on which the travel instance is to be created. The derived type assigned to the internal table ensures the type-safe creation of the travel instance. It's best practise to use this syntax as it provides direct feedback when trying to override fields that are defined as readonly. Furthermore, it´s not possible to set initial values for fields when the syntax `SET FIELDS WITH` is used instead.

The create-by-association operation is specified by the keywords `CREATE BY \` followed by the name of the association.

After specifying the fields to be populated for the associated entity, the keyword `WITH` introduces the specification of input parameters that are expected in the form of an internal table. This internal table is constructed using the value operator containing data for two booking instances. Since we already specified the fields to be modified, we don´t need to additionally activate them in the `%control` structure. As the primary key of the travel instance for which the booking instances are to be created doesn't exist yet, the travel instance is referred to using its content id `create_travel`.

The structure specified after the keyword `MAPPED` acts as a response parameter. It contains the information on which primary keys were provided by the create operations for the given content IDs. The type of this response parameter is derived from the specified entity and from the triggered operation by the framework.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

Finally, the persistence of data to the database is triggered via the `COMMIT ENTITIES` statement. For more information on this statement, see COMMIT ENTITIES.

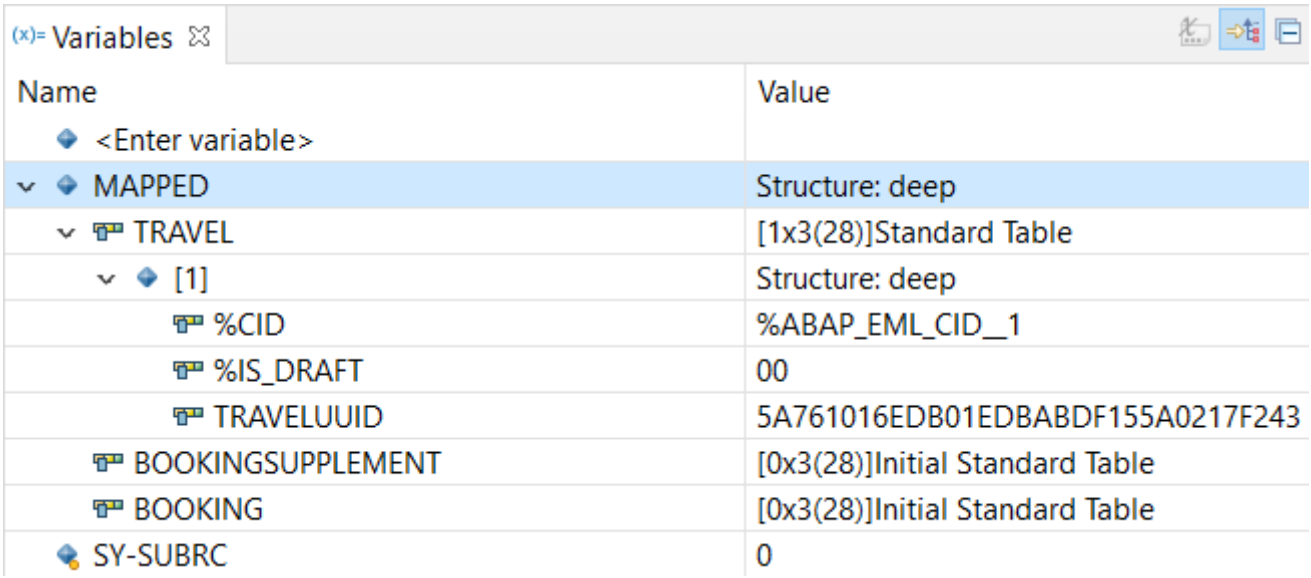
AUTO FILL CID

For static operations like `CREATE` and `CREATE BY ASSOCIATION`, the addition `AUTO FILL CID` can be used to ensure that `%cid` is filled with a content ID (CID). Note that, if `%cid_ref` is to be used, then `AUTO FILL CID` must not be used for the referenced instance as `%cid_ref` can't refer to automatically created content IDs. In the example below, a travel instance is created using an internal table, for which no content ID is specified. The content ID is generated automatically during the creation of the travel instance using `AUTO FILL CID`.

```
*Declare internal table using derived type
DATA create TYPE TABLE FOR CREATE /DMO/R_Travel_D.
```

```
*Select valid flight data
SELECT SINGLE FlightDate FROM /DMO/I_Flight INTO @DATA(flight).

*Populate internal table for travel instance, %cid does not need to be specified
create = VALUE #( (   %is_draft            = if_abap_behv=>mk-off
                      CustomerID           = '1'
                      AgencyID             = '70006'
                      BeginDate            = flight
                      EndDate              = flight
                      Description          = 'Travel 1'
                ) ).

*Create a travel instance using AUTO FILL CID
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Travel
    CREATE AUTO FILL CID FIELDS ( CustomerID AgencyID BeginDate EndDate Description ) WITH create
MAPPED DATA(mapped)
REPORTED DATA(reported)
FAILED DATA(failed).

*Persist transactional buffer
COMMIT ENTITIES.
```

The mapping information between the generated content ID and the key of the instance is stored in the MAPPED structure.



General Information: UPDATE

The UPDATE statement enables the editing of entity instances.

For more information about the implementation contract, refer to Implementation Contract: UPDATE.

Example: UPDATE

The following UPDATE statement sets the status of two booking instances to 'Accepted'. Unlike on a UI, with EML you've the choice between modifying draft or active instances. In this case, we want to update the active instances directly. You can use the following code in a separate unit test class to test the update operation of the reference business object. Note that the placeholders for the booking UUIDs need to be replaced before according to the existing data.

```
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Booking
    UPDATE FIELDS ( BookingStatus )
    WITH VALUE #( ( BookingUUID   = '<Booking_UUID>'
                    %is_draft     = if_abap_behv=>mk-off
                    BookingStatus = 'A' )
```

```
                    ( BookingUUID   = '<Booking_UUID>'
                      %is_draft     = if_abap_behv=>mk-off
                      BookingStatus = 'A' ) )
  FAILED DATA(failed)
  REPORTED DATA(reported).

  COMMIT ENTITIES.
```

The `MODIFY` statement is introduced by the keywords `MODIFY ENTITIES OF` followed by the behavior definition of the business object the statement refers to.

The statement at hand triggers the update operation of the booking entity, hence the `ENTITY Booking` is specified. Here, we use the alias specified for the entity in the behavior definition.

The next line states that the values of the field `BookingStatus` are to be updated.

The keywords `WITH VALUE` introduce the construction of an internal table that indicates which instances are to be updated with which values. Within the constructor operator `VALUE`, the instances are first identified by its booking UUIDs, then their values for the booking status field are specified respectively. Using the draft indicator `%is_draft` we can determine whether we want to update active or draft instances. Since we want to update the active instances directly, we set the draft indicator to false.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

Finally, the persistence of data to the database is triggered via the `COMMIT ENTITIES` statement. For more information on this statement, see COMMIT ENTITIES.

General Information: `DELETE`

The `DELETE` statement enables the deletion of entity instances.

For more information about the implementation contract, refer to Implementation Contract: DELETE.

Example: `DELETE`

The following statement deletes an instance of the entity travel. It can be used in a separate unit test class to test the delete operation of the reference business object. Note that the placeholder for the travel UUID needs to be replaced before according to the existing data.

```
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY TRAVEL
    DELETE FROM VALUE #( ( TravelUUID = '<Travel_UUID>'
                           %is_draft  = if_abap_behv=>mk-off ) )
FAILED DATA(failed)
REPORTED DATA(reported).

COMMIT ENTITIES.
```

The `MODIFY` statement is introduced by the keywords `MODIFY ENTITIES OF` followed by the behavior definition of the business object the statement refers to.

The statement at hand uses the delete operation of the travel entity, hence the `ENTITY Travel` is specified. Here, we use the alias specified for the entity in the behavior definition.

The value operator specifies the key of the instance that is to be deleted. Using the draft indicator `%is_draft` we can determine whether we want to delete active or draft instances. Since we want to delete an active instance, we set the draft indicator to false.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

Finally, the persistence of data to the database is triggered via the `COMMIT ENTITIES` statement. For more information on this statement, see COMMIT ENTITIES.

General Information: `EXECUTE` Action

This statement enables the execution of actions.

For more information about the implementation contract, refer to Implementation Contract: Action.

Example: `EXECUTE` Action

The following statement triggers an action that recalculates the total travel price based on changed values. The action is executed for those instances whose keys have been determined by the `READ BY association` statement. In the behavior pool of the reference business object, the following `EXECUTE` statement is used in the determination `calculateTotalPrice` for the entities booking and booking supplement. For further contextual information and for the complete code of this determination, see Calculate Total Price.

```
   MODIFY ENTITIES OF /DMO/R_Travel_D IN LOCAL MODE
       ENTITY Travel
         EXECUTE reCalcTotalPrice
           FROM CORRESPONDING  #( travels )
 RESULT DATA(result)
 FAILED DATA(failed).
```

The `MODIFY` statement is introduced by the keywords `MODIFY ENTITIES OF` followed by the behavior definition of the business object the statement refers to.

Since this statement is contained in the behavior pool of the business object it refers to, the keywords `IN LOCAL MODE` are used. They ensure that authorization and feature control implementations aren't triggered.

The statement at hand executes the action for instances of the travel entity, hence the `ENTITY Travel` is specified. Here, we use the alias specified for the entity in behavior definition.

The keyword `EXECUTE` is followed by the name of the action that is to be executed.

The action is executed only for the respective parent travel instances. Hence, the field expression `FROM` selects the internal table `travels` has been filled with the required travel UUIDs by the `READ BY association` statement described above.

The table declared after the keyword `RESULT` contains the result returned by the action. Since the triggered action doesn't have a result, `RESULT` is filled in this case.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

General Information: `AUGMENTING`

With the statement `AUGMENTING` you can add data or modify incoming modify requests on the projection layer by augmenting requested modify operations before the requests are passed to the base business object.

For more information about the syntax, see MODIFY AUGMENTING ENTITY (ABAP Keyword Documentation).

Example: AUGMENTING

The following method enables the maintenance of the entity Supplement contains a denormalized field Description. This field is defined in the associated child entity SupplementText. Without this augmentation method, incoming modify requests on the supplement entity can't consider the description field since it derives from another entity. The AUGMENTING statement is used in this case to augment an incoming create request by a create-by-association operation. This ensures that incoming create requests do not only lead to the creation of supplement instances but also to the creation of corresponding supplement text instances which contain the changed description field. The augmentation of incoming update requests isn't considered in the code sample shown below. For further information on this augmentation scenario, see Editing Language-Dependent Fields.

```
METHOD augment.
...

"Handle create requests including SupplementDescription
    LOOP AT entities_create INTO DATA(supplement_create).
      "Count Table index for uniquely identifiably %cid on creating supplementtext
      APPEND sy-tabix TO relates_create.

      "Direct the Supplement Create to the corresponding SupplementText Create-By-Associatio
      APPEND VALUE #( %cid_ref           = supplement_create-%cid
                      %is_draft          = supplement_create-%is_draft
                      %key-supplementid  = supplement_create-%key-SupplementID
                      %target            = VALUE #( ( %cid              = |CREATETEXTCID{ sy
                                                      %is_draft         = supplement_create-
                                                      %key-supplementid = supplement_create-
                                                      languagecode      = sy-langu
                                                      description       = supplement_create-
                                                      %control          = VALUE #( supplement
                                                                                   languageco
                                                                                   descriptio
                                                   ) )
                    ) TO suppltext_for_new_suppl.
    ENDLOOP.

*Augment Create by a Create by Association
MODIFY AUGMENTING ENTITIES OF /DMO/I_Supplement
      ENTITY Supplement
        CREATE BY \_SupplementText
        FROM suppltext_for_new_suppl
        RELATING TO entities_create BY relates_create.

...
ENDMETHOD.
```

The augmentation method receives the supplement instances to be created via the importing parameter table entities_create. For each of these instances the current table index is appended to the internal table relates_create, which will be used by the subsequent AUGMENTING statement. Moreover, the required instance data for the subsequent create-by-association operation are stored in the internal table suppltext_for_new_suppl. As a result, each row of this table contains the data of the supplement instance referenced using the component cid_ref as well as the data for the associated supplement text instance that is to be created. This table is used in the following AUGMENTING statement.

The AUGMENTING statement is introduced by the keywords MODIFY AUGMENTING ENTITIES OF followed by the behavior definition of the business object the statement refers to. Since AUGMENTING statements can only refer to entities of base business objects, the business object /dmo/i_supplement is specified.

The statement at hand uses the create-by-association operation of the supplement entity, hence the `ENTITY Supplement` is specified. Here, we use the alias specified for the entity in the behavior definition.

The create-by-association operation is specified by the keywords `CREATE BY \` followed by the name of the association.

The operation is performed using the prepared internal table `suppltext_for_new_suppl`.

It can happen that the base business object returns entries in the `FAILED` and `REPORTED` structures for augmented requests. To relate these responses to the original request, if the augmented request contains new instances, the `AUGMENTING` statement offers the addition `RELATING TO` allows to relate augmented instances to original instances. The runtime uses this information to transform `FAILED` keys of new instances back to the keys of the related original instances. If an augmented instance - in this case a supplement text instance - fails, the related original instance - in this case the supplement instance - is included in the `FAILED` response of the overall request.

GET PERMISSIONS

General Information: `GET PERMISSION`

This statement is used to check whether permissions in terms of authorization and feature control exist on global and instance level. Information about permissions can be retrieved for the operations create, create-by-association, update, and delete as well as for actions and fields. The following sections cover dedicated EML statements for different permission types.

For general information about authorization and feature control, see Authorization Control and Feature Control.

For more information about the syntax, see GET PERMISSIONS (ABAP Keyword Documentation) .

For more information about the implementation contract for authorization and feature control, see Implementation Contract: Feature Control and Implementation Contract: Authorization Control.

Example: Global Authorization

Global authorization is used for all authorization checks that only depend on the user. You can define global authorization to check if users are allowed to execute an operation in general.

The following code is used to check whether the logged on user has the permission for performing create operations on the reference business object.

```
*Declare derived type for authorization request
DATA: request_ga TYPE STRUCTURE FOR PERMISSIONS REQUEST /DMO/R_Travel_D.

*Activate check for create operation
request_ga-%create = if_abap_behv=>mk-on.

*Perform authorization request
GET PERMISSIONS ONLY GLOBAL AUTHORIZATION ENTITY /DMO/R_Travel_D
  REQUEST request_ga
RESULT DATA(result)
FAILED DATA(failed)
REPORTED DATA(reported).
```

Before we perform the actual authorization request, we prepare a local structure containing the requested permission. At first, we declare this local structure using the derived type for permission requests on the reference business object. Then we activate the create component within the structure by means of the ABAP interface `if_abap_behv`.

The `GET PERMISSIONS` statement is introduced by the keywords `GET PERMISSIONS ONLY GLOBAL AUTHORIZATION ENTITY` followed by the entity the statement refers to.

The keyword REQUEST is followed by the local structure containing the requested operation permission.

The result is saved in a local structure named `result`.

For information on the response parameters FAILED and REPORTED, see Common Response Parameters.

Example: Instance Authorization

Instance authorization is used for all authorization checks that, in addition to the user role, depend on the state of the entity instance in question. With instance authorization, you can define authorization that depends on a field value of the instance.

The following code is used to check whether performing update operations on a given travel instance of the reference business object is possible. According to the instance authorization logic of the business object, this depends on whether the logged on user has the update permission for the country code of the instance. The country code of the instance is defined by the `CountryCode` of the agency instance that is associated with the travel instance. Note that the placeholder for the travel UUID needs to be replaced according to the existing data.

```
*Declare derived type for authorization request
DATA: request_ia TYPE STRUCTURE FOR PERMISSIONS REQUEST /DMO/R_Travel_D.

*Activate check for update operation
request_ia-%update = if_abap_behv=>mk-on.

*Perform authorization request
GET PERMISSIONS ONLY INSTANCE AUTHORIZATION ENTITY /DMO/R_Travel_D
  FROM  VALUE #( ( TravelUUID = '<Travel_UUID>'
                   %is_draft  = if_abap_behv=>mk-off ) )
  REQUEST request_ia
RESULT DATA(result)
FAILED   DATA(failed)
REPORTED DATA(reported).
```

Before we perform the actual authorization request, we prepare a local structure containing the requested permission. At first, we declare this local structure using the derived type for permission requests on the reference business object. Then we activate the update component within the structure by means of the ABAP interface `if_abap_behv`.

The `GET PERMISSIONS` statement is introduced by the keywords `GET PERMISSIONS ONLY INSTANCE AUTHORIZATION ENTITY` followed by the entity the statement refers to.

The value operator specifies the key of the instance whose permissions are to be checked. Using the draft indicator `%is_draft` we can determine whether we want to check the permissions for the active or for the draft instance. Since we want to check the active instance, we set the draft indicator to false.

The keyword REQUEST is followed by the local structure containing the requested operation permission.

The result is saved in a local structure named `result`.

For information on the response parameters FAILED and REPORTED, see Common Response Parameters.

Example: Instance Feature Control

You can use instance feature control to define which fields, operations, and actions have access restrictions. These restrictions can depend on the state of the instance.

The following code is used to check the access restrictions of the field `BookingFee` that is defined for the entity `Travel` in the reference business object. The access restrictions for the field depend on the value of the field `OverallStatus`. If the value is 'A' (accepted), the field `BookingFee` is read-only. Else it's unrestricted. Note that the placeholder for the travel UUID needs to be replaced according to the existing data.

```
*Declare derived type for feature control request
DATA: request_ifc TYPE STRUCTURE FOR PERMISSIONS REQUEST /DMO/R_Travel_D.

*Activate check for field
request_ifc-%field-BookingFee = if_abap_behv=>mk-on.

*Perform feature control request
GET PERMISSIONS ONLY INSTANCE FEATURES ENTITY /DMO/R_Travel_D
  FROM  VALUE #( ( TravelUUID = '<Travel_UUID>'
                   %is_draft  = if_abap_behv=>mk-off ) )
  REQUEST request_ifc
RESULT DATA(result)
FAILED DATA(failed)
REPORTED DATA(reported).
```

Before we perform the actual feature control request, we prepare a local structure containing the requested permission. At first, we declare this local structure using the derived type for permission requests on the reference business object. Then we activate the check for the field using the field component within the structure by means of the ABAP interface `if_abap_behv`.

The `GET PERMISSIONS` statement is introduced by the keywords `GET PERMISSIONS ONLY INSTANCE FEATURES ENTITY` followed by the entity the statement refers to.

The value operator specifies the key of the instance whose permissions are to be checked. Using the draft indicator `%is_draft` we can determine whether we want to check the permissions for the active or for the draft instance. Since we want to check the active instance, we set the draft indicator to false.

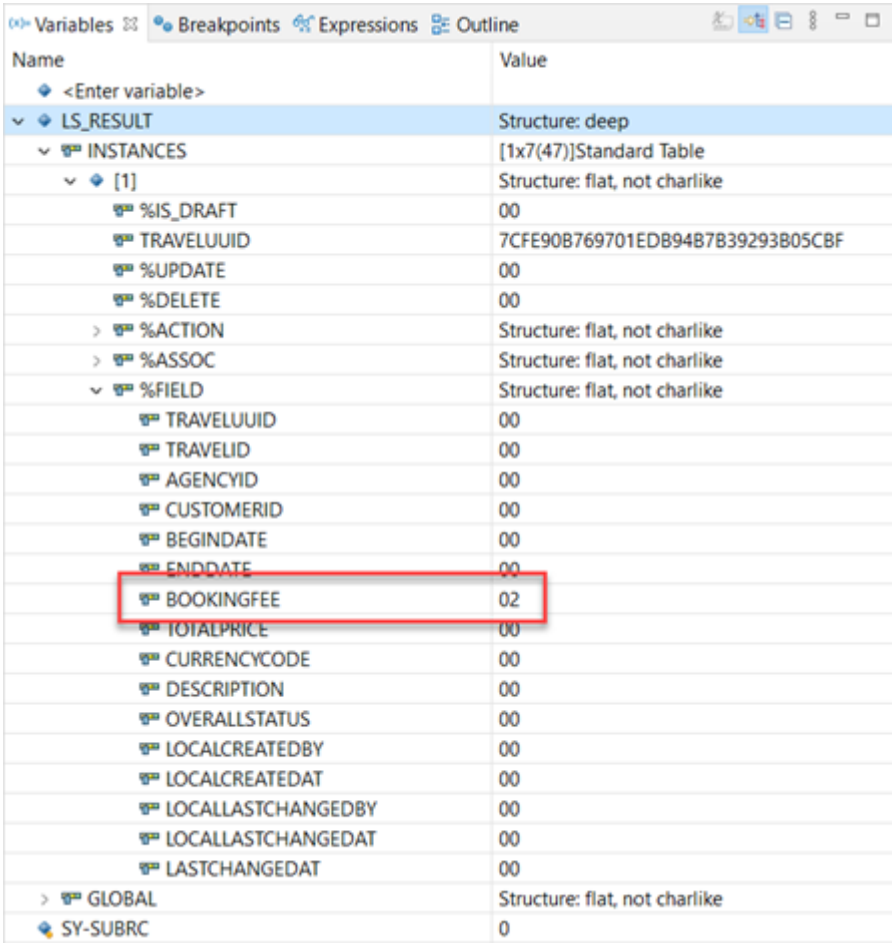
The keyword `REQUEST` is followed by the local structure containing the requested field permission.

The result is saved in a local structure named `result`.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

Checking Results

The following screenshot shows the structure `result` resulting from the preceding instance feature control permission request in the debugger. The permission request has been performed for a travel instance with the overall status `Accepted`.



To interpret the returned permission value, we consult the ABAP interface `if_abap_behv` in the element info. It contains a legend explaining the meaning for returned values. In the section containing the constants for feature control, there's a subsection for fields. This subsection contains the meaning of field value '02': **read_only**.

```
fc
    f
      unrestricted      type if_abap_behv=>t_xflag value '00'
      mandatory         type if_abap_behv=>t_xflag value '01'
      read_only         type if_abap_behv=>t_xflag value '02'
      all               type if_abap_behv=>t_xflag value 'FF'
    o
      enabled           type if_abap_behv=>t_xflag value '00'
      disabled          type if_abap_behv=>t_xflag value '01'
image
    transactional
    before
mk
    off                 type if_abap_behv=>t_xflag value '00'
    on                  type if_abap_behv=>t_xflag value '01'
op
    m
      create            type if_abap_behv=>t_char01 value 'C'
      update            type if_abap_behv=>t_char01 value 'U'
      delete            type if_abap_behv=>t_char01 value 'D'
      action            type if_abap_behv=>t_char01 value 'A'
      create_ba         type if_abap_behv=>t_char01 value 'O'
```

The values returned for the other get permissions statements shown above can be evaluated accordingly. Note that the value returned for operations or fields that haven't been requested is always '00'.

SET LOCKS

General Information: SET LOCKS

Before entity instances can be modified, they need to be locked for the corresponding user. When performing modify operations, this step is taken over implicitly by the framework. It isn't necessary to lock an instance explicitly before performing a modify operation. However, to prevent other users from modifying an instance, the `SET LOCKS` statement can be used. The instance is then locked for the corresponding user and can't be modified by other users until the lock is released at the end of the transaction.

For more information about the syntax, see SET LOCKS (ABAP Keyword Documentation) .

For more information about the implementation contract, see Implementation Contract: LOCK.

Example: SET LOCKS

The following statement locks an instance of the reference business object. Note that the placeholder for the travel UUID needs to be replaced according to the existing data.

```
SET LOCKS OF /DMO/R_Travel_D
  ENTITY TRAVEL
    FROM VALUE #( ( TravelUUID = '<Travel_UUID>' ) )
FAILED DATA(failed)
REPORTED DATA(reported).
```

The `SET LOCKS` statement is introduced by the keywords `SET LOCKS OF` followed by the entity the statement refers to.

The statement at hand locks an instance of the travel entity, hence the `ENTITY TRAVEL` is specified.

The next line specifies the primary key of the instance that is to be locked.

For information on the response parameters `FAILED` and `REPORTED`, see Common Response Parameters.

COMMIT ENTITIES

COMMIT ENTITIES

General Information: `COMMIT ENTITIES`

The `COMMIT ENTITIES` statement triggers amongst others the save sequence, which persists data from the transactional buffer to the database. The `COMMIT ENTITIES` statement triggers the persistence of data from the transactional buffer to the database for all business objects that were changed within the LUW. `COMMIT ENTITIES` needs to be specified explicitly for modify operations that are triggered outside of a behavior pools to terminate the LUW. For modify request from within a behavior pool, `COMMIT ENTITIES` and subsequently the save sequence is triggered by the framework.

For more information about the syntax, see COMMIT ENTITIES (ABAP Keyword Documentation).

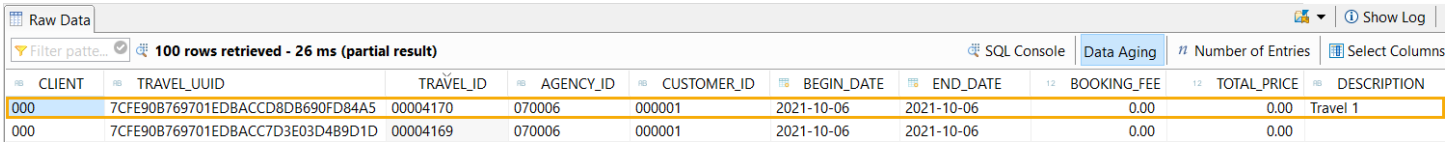
Example: `COMMIT ENTITIES`

The following example code shows a `MODIFY CREATE` statement followed by a `COMMIT ENTITIES` statement, which are executed on the reference business object.

```
"Declare internal table using derived type
DATA create TYPE TABLE FOR CREATE /DMO/R_Travel_D.

"Select valid flight data
```

```
  SELECT SINGLE FlightDate FROM /DMO/I_Flight INTO @DATA(flight).

  "Populate internal table for travel instance
  create = VALUE #( ( %cid                = 'create_travel'
                      %is_draft           = if_abap_behv=>mk-off
                      CustomerID          = '1'
                      AgencyID            = '70006'
                      BeginDate           = flight
                      EndDate             = flight
                      Description         = 'Travel 1'
   ) ).

  "Create travel instance
  MODIFY ENTITIES OF /DMO/R_Travel_D
    ENTITY Travel
       CREATE FIELDS ( CustomerID AgencyID BeginDate EndDate Description ) WITH create
     MAPPED DATA(mapped_modify)
     REPORTED DATA(reported_modify)
     FAILED DATA(failed_modify).

  "Persist travel instance
  COMMIT ENTITIES
    RESPONSE OF /DMO/R_Travel_D
     FAILED DATA(failed_commit)
     REPORTED DATA(reported_commit).
```

The created travel instance is persisted and visible in the database `/DMO/A_TRAVEL_D`.



Raw Data | Show Log | Filter patte... | 100 rows retrieved - 26 ms (partial result) | SQL Console | Data Aging | Number of Entries | Select Columns

CLIENT	TRAVEL_UUID	TRAVEL_ID	AGENCY_ID	CUSTOMER_ID	BEGIN_DATE	END_DATE	BOOKING_FEE	TOTAL_PRICE	DESCRIPTION
000	7CFE90B769701EDBACCD8DB690FD84A5	00004170	070006	000001	2021-10-06	2021-10-06	0.00	0.00	Travel 1
000	7CFE90B769701EDBACC7D3E03D4B9D1D	00004169	070006	000001	2021-10-06	2021-10-06	0.00	0.00	

Persisted Travel Instance

Simulation Mode

General Information: Simulation Mode

If the `COMMIT ENTITIES` statement is used with the addition `IN SIMULATION MODE`, the save sequence is executed without actually saving any data. This means that in simulation mode, the methods `finalize`, `check_before_save`, and `cleanup_finalize` are executed, but the methods `adjust_numbers`, `save` and `cleanup` are not. Since determinations and validations are executed by the methods `finalize` and `check_before_save`, the simulation mode can be used to check whether the finalized data could be persisted, if a `COMMIT ENTITIES` statement without simulation mode was executed.

Example: Simulation Mode

The following example code shows a `MODIFY CREATE` statement followed by a `COMMIT ENTITIES` statement in simulation mode, which are executed on the reference business object. The `COMMIT ENTITIES` statement in simulation mode checks whether the created travel instance can be persisted.

```
  "Declare internal table using derived type
  DATA create TYPE TABLE FOR CREATE /DMO/R_Travel_D.

  "Select valid flight data
  SELECT SINGLE FlightDate FROM /DMO/I_Flight INTO @DATA(flight).

  "Populate internal table for create operation
  create = VALUE #( ( %cid       = 'create_travel'
                      %is_draft  = if_abap_behv=>mk-off
                      CustomerID = '1'
                      AgencyID   = '70006'
                      BeginDate  = flight
                      EndDate    = flight ) ).

  "Create travel instance
```

```
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Travel
    CREATE FIELDS ( CustomerID AgencyID BeginDate ) WITH create
   FAILED DATA(failed_modify)
   REPORTED DATA(reported_modify).

"Check whether saving is possible
COMMIT ENTITIES IN SIMULATION MODE
  RESPONSE OF /DMO/R_Travel_D
   FAILED DATA(failed_commit)
   REPORTED DATA(reported_commit).
```

In order to determine whether the instance can be persisted, the response parameters `FAILED` and `REPORTED` of the `COMMIT ENTITIES` statement can be checked. These parameters are populated by validations in case of inconsistencies.

FAILED_COMMIT	Structure: deep
TRAVEL	[1x8(40)]Standard Table
[1]	Structure: flat, not charlike
%IS_DRAFT	00
TRAVELUUID	7CFE90B769701EEBACCA436D38FE4E77
%FAIL	Structure: flat, not charlike
%CREATE	00
%UPDATE	00
%DELETE	00
%ACTION	Structure: flat, not charlike
%ASSOC	Structure: flat, not charlike
BOOKINGSUPPLEMENT	[0x6(28)]Initial Standard Table
BOOKING	[0x6(28)]Initial Standard Table
SY-SUBRC	4

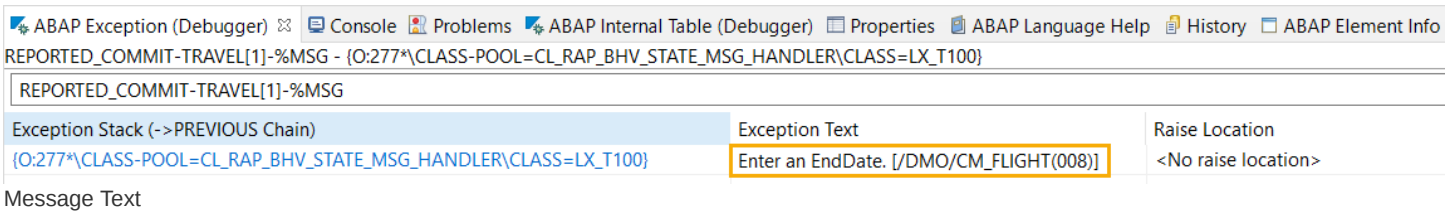
Failed Key for Travel Instance

The `FAILED` parameter contains an entry in its travel table for the created travel instance, which means that this instance cannot be persisted. In order to determine the reason for the failed commit operation, the response parameter `REPORTED` can be checked.

REPORTED_COMMIT	Structure: deep
TRAVEL	[1x10(72)]Standard Table
[1]	Structure: deep
%IS_DRAFT	00
TRAVELUUID	7CFE90B772401EDBADE093D11881CAD3
%MSG	{O:277*\CLASS-POOL=CL_RAP_BHV_STATE_MSG_HANDLER\CLASS=LX_T100}
%CREATE	00
%UPDATE	00
%DELETE	00
%ACTION	Structure: flat, not charlike
%ELEMENT	Structure: flat, not charlike
%STATE_AREA	
%GLOBAL	00
BOOKINGSUPPLEMENT	[0x9(88)]Initial Standard Table
BOOKING	[0x9(76)]Initial Standard Table
%OTHER	[0x1(8)]Initial Standard Table
SY-SUBRC	4

Reported Message

The %MSG component contains a reference to the corresponding message. Its text is displayed in the ABAP Exception view after double clicking the %MSG component in the Variables tab.



The failed key and the message have been returned by the validation `validateDates`, which prevents the persistence of instance data in case of inconsistent dates. In this example, the end date is missing as the corresponding field has not been specified in the `FIELDS` expression of the `MODIFY` statement. After adding the field `EndDate` to the `FIELDS` expression, the instance data is consistent and can be persisted using a `COMMIT ENTITIES` statement.

```
"Declare internal table using derived type
DATA create TYPE TABLE FOR CREATE /DMO/R_Travel_D.

"Select valid flight data
SELECT SINGLE FlightDate FROM /DMO/I_Flight INTO @DATA(flight).

"Populate internal table for create operation
create = VALUE #( ( %cid       = 'create_travel'
                    %is_draft  = if_abap_behv=>mk-off
                    CustomerID = '1'
                    AgencyID   = '70006'
                    BeginDate  = flight
                    EndDate    = flight ) ).

"Create travel instance
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Travel
   CREATE FIELDS ( CustomerID AgencyID BeginDate EndDate ) WITH create
    FAILED DATA(failed_modify)
    REPORTED DATA(reported_modify).

"Persist travel instance
COMMIT ENTITIES.
```

ROLLBACK ENTITIES

General Information: `ROLLBACK ENTITIES`

The `ROLLBACK ENTITIES` statement resets the transactional buffer. It is used outside of behavior pools to roll back all changes done since the last `COMMIT ENTITIES` operation.

For more information about the syntax, see ROLLBACK ENTITIES (ABAP Keyword Documentation).

Example: `ROLLBACK ENTITIES`

The following code demonstrates the effect of the `ROLLBACK ENTITIES` statement. After declaring and populating internal tables which will be used for subsequent `MODIFY` statements, a first instance is created using a `MODIFY` statement and persisted using a `COMMIT ENTITIES` statement. Then, after the creation of the second instance, the transactional buffer is reset using the `ROLLBACK ENTITIES` statement. The following `COMMIT ENTITIES` statement causes no change in the database as the transactional buffer has been emptied by the `ROLLBACK ENTITIES` statement. At the end of the execution, the database only contains one travel instance.

For more information on the `COMMIT ENTITIES` statement, see COMMIT ENTITIES.

Expand the following code sample to view the source code

Sample Code

```
*Declare internal tables using derived type
DATA travel1 TYPE TABLE FOR CREATE /DMO/R_Travel_D .
DATA travel2 TYPE TABLE FOR CREATE /DMO/R_Travel_D .

*Select valid flight data
SELECT SINGLE FlightDate FROM /DMO/I_Flight INTO @DATA(flight1).

*Populating the internal tables
travel1 = VALUE #( ( %cid       = 'create_travel_1'
                     %is_draft  = if_abap_behv=>mk-off
                     CustomerID = '1'
                     AgencyID   = '70006'
                     BeginDate  = flight1
                     EndDate    = flight1 ) ).

travel2 = VALUE #( ( %cid       = 'create_travel_2'
                     %is_draft  = if_abap_behv=>mk-off
                     CustomerID = '2'
                     AgencyID   = '70007'
                     BeginDate  = flight1
                     EndDate    = flight1 ) ).


*Create first travel instance
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Travel
    CREATE FIELDS ( CustomerID AgencyID BeginDate EndDate ) WITH travel1.

*Persist transactional buffer
COMMIT ENTITIES
RESPONSE OF /DMO/R_Travel_D
FAILED DATA(failed_commit1)
REPORTED DATA(reported_commit1).

*Create second travel instance
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Travel
    CREATE FIELDS ( CustomerID AgencyID BeginDate EndDate ) WITH travel2.

*Reset transactional buffer
ROLLBACK ENTITIES.

*Persist (empty) transactional buffer
COMMIT ENTITIES
RESPONSE OF /DMO/R_Travel_D
FAILED DATA(failed_commit2)
REPORTED DATA(reported_commit2).
```

Common Response Parameters

EML statements have their own response parameters. These response parameters receive data from the operations triggered by the respective EML statement. Response parameters that can be used in many EML statements are the structures `FAILED` and `REPORTED`. These structures contain nested tables for each entity of the BO. The `FAILED` structure logs the keys of instances for which a triggered operation has failed. The `REPORTED` structure logs messages that have been returned from a triggered operation. The types of both response parameters are derived from the involved entity and operation by the framework.

The following example code shows a `CREATE` and a `COMMIT ENTITIES` statement which are executed on the reference business object.

```
*Declare internal table using derived type
DATA create TYPE TABLE FOR CREATE /DMO/R_Travel_D.

*Select valid flight data
SELECT SINGLE FlightDate FROM /DMO/I_Flight INTO @DATA(flight).

*Populate internal table for create operation
create = VALUE #( ( %cid       = 'create_travel'
                    %is_draft  = if_abap_behv=>mk-off
                    CustomerID = '1'
                    AgencyID   = '70006'
                    BeginDate  = flight ) ).

*Create travel instance
MODIFY ENTITIES OF /DMO/R_Travel_D
  ENTITY Travel
    CREATE FIELDS ( CustomerID AgencyID BeginDate ) WITH create
FAILED DATA(failed_modify)
REPORTED DATA(reported_modify).

*Try to persist travel instance
COMMIT ENTITIES
RESPONSE OF /DMO/R_Travel_D
FAILED DATA(failed_commit)
REPORTED DATA(reported_commit).
```

The create operation is executed without errors and hence does not return any keys to the `FAILED` structure. The commit operation however cannot be executed because the end date has not been provided for the travel instance. The validation `validateDates`, which is executed as part of the commit operation, detects the missing end date and returns the key of the failed instance to the `FAILED` structure of the `COMMIT ENTITIES` statement. Moreover, a message indicating the reason for the failed persistence is returned to the `REPORTED` structure.

Response parameters dedicated to specific operations like `MAPPED` or `LINK` are covered in the sections for the respective EML statements.

Messages

This topic explains the basic message concepts relevant for the ABAP RESTful Application Programming Model.

About Messages

Messages offer an important way to guide and validate consumer and user actions, and help to avoid and resolve problems. Thus, messages are important to communicate problems to a consumer or user. Well-designed messages help to recognize, diagnose, and resolve issues. That's why it's important to always use messages consistently and optimize the interaction as a whole. Consequently, errors and warnings that require action should be clearly stated and described in a way that helps to resolve the issue quickly and efficiently. It's recommended to provide a message for each entry in the fail structure to provide additional information.

There are different types of messages depending on whether they refer to the state of a business object instance or only to the current request. **State messages** refer to a business object instance and **transition messages** refer to a request. State messages must always be bound to a business object instance (bound), whereas transition messages can either be bound or unbound (not related to a business object instance).

ℹ Note

The termination at runtime due to errors must always be implemented depending on entries in the fail structure that indicate the fail cause. Messages as part of the reported structure only offer additional information about the respective circumstances, but aren't reliable indications for termination at runtime. It's always recommended to consider the fail cause in the fail structure for the program design.

Messages in EML

When you execute a modify request, the keys of the failed instances are returned in the `failed` structure. Messages for the related failed instance are returned with the `reported` structure whose components are derived at runtime by the compiler depending on the returned values. The following components of the `reported` structure are relevant for the message handling:

- `REPORTED`
  - `%CID`: ID of the relevant instance
  - `%MSG`: Filled with an instance of the message-wrapper class
  - `%ELEMENT`: Lists all fields or associations of an entity the message relates to.
  - `%STATE_AREA`: If this component of type `String` is filled in, the framework interprets a message as state message.
  - `%OP`: This component indicates to which executed operation the message relates to. This component is only relevant for transition messages.
  - `%OTHER`: The reported structure contains a table for each entity defined and in addition `%OTHER` for all messages that aren't entity-related. The `%OTHER` component is filled with an instance of the message-wrapper class when a message isn't related to a business object entity (Unbound messages)
  - `%path` (only relevant for child entities): The path component maps a child entity to its parent. If there's a business object with several child entities, the `%PATH` component is extended to map the child entity to its parent and the business object root.

With the `%ELEMENT` component, you can assign messages to one or several target fields. These targets are interpreted by the client and result in an improved user experience, because the target establishes a visible link between the message and the target field that also enables navigation if there are many error messages. The `%STATE_AREA` component determines whether a message is interpreted as state message. If this component is left empty, a message is interpreted as a transition message. If the component is filled in, the message is interpreted as a state message. The component `%OP` becomes important to distinguish to which operation the message relates to if many operations or actions are requested for the same instance.

For more general information about the `reported` and `failed` structure, see:

- Reported Structure.
- Failed Structure.

For more information about how messages behave in EML, see Message Behavior in EML (Entity Manipulation Language).

.

Related Information

State Messages
Transition Messages

Generic Message Implementation

Generic Message Implementation

Generally, the message creation and allocation to a reported structure is identical for every scenario:

```
APPEND VALUE #(
      %tky   = instance-%tky

      %msg   = NEW message_exception(
                   textid               =  message_exception=>message_exception_constant
                   severity             =  if_abap_behv_message=>severity-severity
                   message_variable     =  instance-field )
      %element-field1                   =  if_abap_behv=>mk-on
      %element-field2                   =  if_abap_behv=>mk-on
      %element-_association             =  if_abap_behv=>mk-on
      %op-%create                       =  if_abap_behv=>mk-on
      %op-%action-action1               =  if_abap_behv=>mk-on
      %state_area                       =  'state_area'
      %path                             =  VALUE #( <root>-%is_draft = <child>-%is_draft
                                                    <root>-<key>     = <child-<parent_key_in_chil
```

- APPEND: Appends the message to the reported structure
- `%tky`: The %TKY makes it possible to uniquely identify to which instance a message belongs. A %TKY is always required for state messages - if state messages are accessed via EML, the respective messages can only be returned if the %TKY of the instance is known. For transitions messages, the %TKY is required for the framework to resolve a `%element-field1` target assignment. If a transition message doesn't have a %TKY, the target isn't resolved in an OData response and the message is interpreted as unbound transition message by the framework.
- `Message_Exception`: In this example, a message exception class is used to encapsulate the messages stored in a message class to handle the formatting of message variables types like dates or amounts via the exception class. You can adapt the message exception class to fit your specific message requirements if necessary.

  For an example implementation, see Creating a Message Exception Class.
- `Message_Exception_Constant`: A message constant is implemented for every message contained in the message class. The constant contains the message class from where the message is drawn and the message number to identify one specific message and its variables.
- `Severity`: The severity specifies if a message is a success, information, warning, or an error message. Depending on the severity, the message is displayed with different icons and colors on the UI. You can select between the following options:
  - **Success**: A success message informs the user that a process or action was completed successfully. No action is required from the user-side. Note that success messages are omitted in some cases.
  - **Information**: An information message offers additional information about a process or an action. No action is required from the user-side.
  - **Warning**: A warning indicates that an action may be required on the user-side.
  - **Error**: An error occurred and an action is required on the user interface.
- `Message_Variable`: You can pass one or more message variables, if they're required for the message you want to display. It's recommended to declare all CDS view fields as possible but optional input parameters in your message exception class, so you can use all fields as variables if necessary.
- `%element-field1/field2`: The referenced field is used as a target for the message. This improves the user experience, as it enables navigation and clear allocation of errors when there are multiple error messages.

  `%element` can also contain associations to child nodes (for example, Sales Order Header -> Sales Order Item). If the target refers to an association, the message references all subinstances of the association. This is useful, for example, if there are no subinstances to issue a message saying that at least one subitem must exist on parent level.

For more information, about how a target assignment is displayed in a UI use case, refer to State Messages on the UI.

- %op-create/%action-action1: The referenced standard or non-standard operation is used to indicate to which operation the message in reported refers to. If more than one operation is executed for the same entity instance in one EML or OData call, the %op-indicator is relevant to map the messages to the corresponding operations. Functions are also addressed via the component %op-%action.

  This component is only relevant for transition messages. State messages don't belong to operations.

- %state_area (only relevant for state messages): This message is identified as state message since this component is filled .For a transition message, this component isn't filled.

  For more information about state messages, see State Messages.

- %path (only relevant for child entities): The path component maps a child entity to its parent. If there's a business object with several child entities, the %PATH component is extended to map the child entity to its parent and the business object root.

- REPORTED-BUSINESS_OBJECT_ENTITY: This message is bound since it's allocated to a specific business object entity. Unbound messages are allocated to the %other component instead of a specific entity.

- Longtexts: If a message has a longtext, the longtext is automatically displayed on the UI together with the message and no additional implementation is required.

Transition Messages

Transition messages refer to a triggered request.

Transition Messages

Transition messages refer to a triggered request and are only valid during the runtime of the request. In contrast to state messages, they don't have any relation to the state of the business object itself, but instead to the transition between states. A typical example for a transition message could be: "Business Object is locked by user &1". As the example refers to a request relevant for a specific entity, it's classified as a bound transition message.

Optionally, a transition message can be bound to a business object entity by adding a %tky to identify the instance and adding the transition message to the REPORTED structure of a business object entity.

The %path Component for Child Entities

The %path component must be filled in for all child entities of a root entity to explicitly map child entities to the parents. For a direct child of a root, the %path component maps the child entity to the specific parent instance using the primary key and the draft-indicator. In the following example the draft parent (%is_draft) is mapped to the child draft via the instance keys:

```
%path = VALUE #( <root>-%is_draft = <child>-%is_draft
                 <root>-<key>     = <child-<parent_key_in_child_entity> ))
```

For a business object consisting of three entities, the second child entity is mapped to the direct parent entity and additionally to the root of the business object. The %path component allows efficient mapping between the entities at runtime.

```
%path = VALUE #( <root>-%is_draft   = <child_2>-%is_draft
                 <root>-<key>       = <child_2>-<root_key_in_child2_entity>
                 <parent>-%is_draft = <child_2>-%is_draft
                 <parent>-<key>     = <child_2>-<parent_key_in_child2_entity> )
```

For a specific implementation example, refer to Sample Implementation: `validateBookingDate` (Booking).

The `%tky` Component

The `%tky` defines whether a transition message refers to a specific business object instance or not. You can define entity-bound transition messages belonging to a business object entity without referring to specific instances with the following syntax:

```
APPEND VALUE #(  %msg = NEW messagewrapper(
                          textid = message_exception=>message_exception_constant
                          severity = if_abap_behv_message=>severity-severity
                          message_variable = read_result-field ) ) TO reported-business_object_enti
```

If no `%tky` is specified, the transition message doesn't refer to an instance of the business object, but the message is still semantically related to the respective entity the REPORTED structure belongs to. This syntax is required, for example, if you want to return a transition message in the context of **global authorizations** or **global feature control**. In global methods, no business object instance exists, but the message is related to a specific entity of the business object. Note that components that rely on the `%tky` to resolve assignments like `%element` are ignored in this context. Transition messages without instance reference are interpreted as unbound transition messages in the OData metadata, however defining an unbound transition message in the context of global authorization isn't possible. Note that the global flag is set automatically by the framework and can't be set manually.

Generic Message Implementation: Bound Transition Message

A message is interpreted as a transition message if the %STATE_AREA isn't filled in the implementation. The following generic implementation is an example of an instance-bound transition message (added to the reported structure of a business object entity and referencing a business object instance with the `%tky` component) with one message variable. It's recommended to have target fields assigned to the `(%ELEMENT-FIELDNAME)` component, if the transition message relates to a particular field value. This doesn't affect the UI, but the target is transmitted as part of the OData metadata.

Instance-bound transition messages require a %tky as instance reference to resolve assignments to the `%element` component. If no %tky is specified, these assignments are ignored during runtime. Instance-bound messages are returned from instance method implementations like instance feature control or instance authorization. An instance-bound transition message with a `%tky` as instance is generically implemented as follows:

```
APPEND VALUE #(
    %tky = instance-%tky
    %msg = NEW message_exception(
            textid           = message_exception=>message_exception_constant
            severity         = if_abap_behv_message=>severity-severity
            message_variable = instance-field ) ) TO reported-business_object_entity.
```

As this example message is attached to a business object entity, the assumption is that the content of the message is related to the request for a specific entity. If this isn't the case and the transition message has no link to any business object entity, a transition message can be bound to the %OTHER component.

For a specific implementation example, refer to Sample Implementation: `setToBooked` (Travel) .

Entity-bound transition messages are semantically only related to an entity, but not a specific instance. Hence, entity-bound transition messages are implemented without an instance reference and are returned from global method implementations like global feature control or global authorization. An entity-bound transition message is generically implemented as follows:

```
APPEND VALUE #(  %msg = NEW messagewrapper(
                          textid = message_exception=message_exception_constant
                          severity = if_abap_behv_message=>severity-severity
                          message_variable = read_result-field ) ) TO reported-business_object_e
```

This example is implemented without a `%tky` reference since global exits are called before a specific business object instance is created. The `%msg` component is filled exactly like an instance-bound transition message. If a `%element` is defined in this case, the assignment is ignored during the runtime. For more details, refer to the section **The `%tky` Component**.

Generic Message Implementation: Unbound Transition Message

An unbound transition message is allocated to the %OTHER component. This generic implementation represents a transition message that passes one message variable to the message-wrapper class and is allocated to the %OTHER component.

```
    reported-%other = VALUE #( ( NEW message_exception(
                                 textid           = message_exception=>message_exception_constant
                                 severity         = if_abap_behv_message=>severity-severity
                                 message_variable = instance-field ) ) ).
```

Unbound and Bound Transition Messages on the UI

This topic shows how bound and unbound transition messages are displayed with UI5.

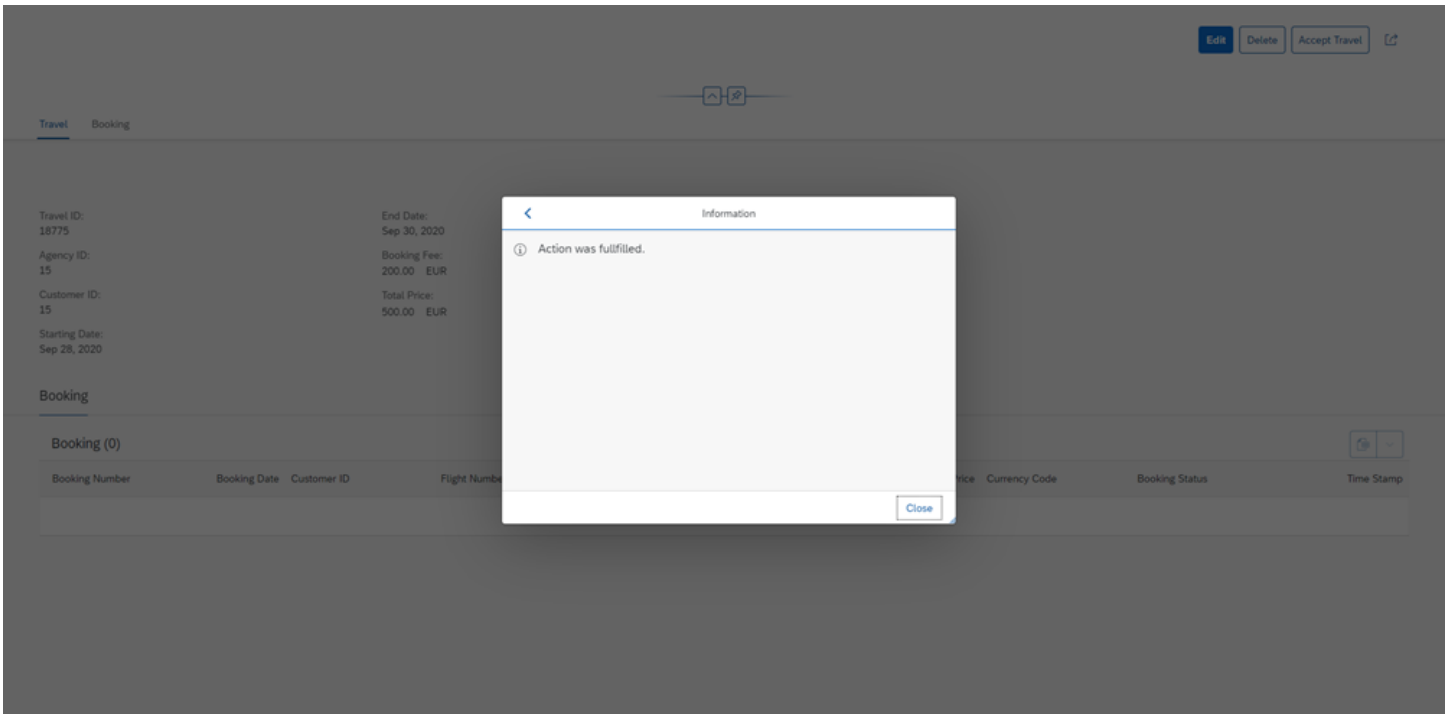
Unbound and Bound Transition Messages on the UI

> **Note**
> The display of messages depends on the OData version and the UI technology, so the display may vary.

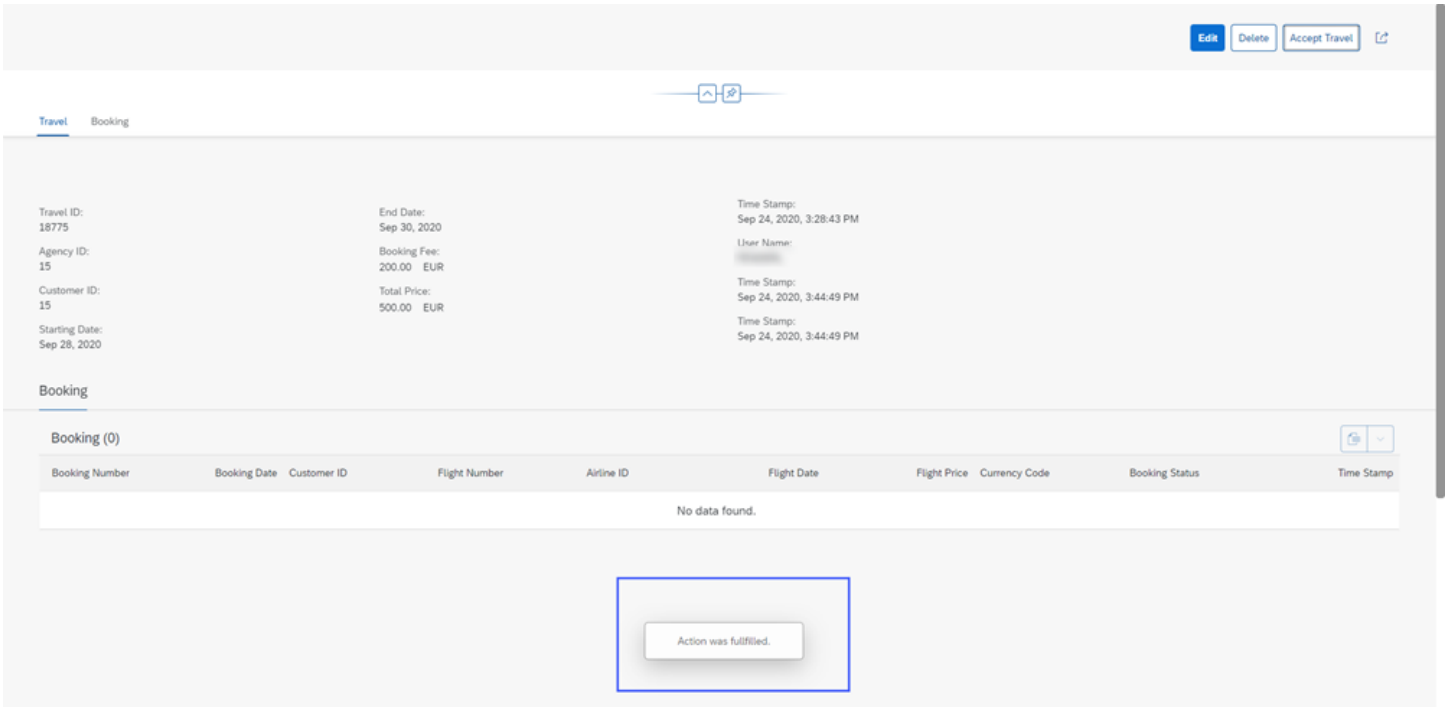
A transition message appears as a pop-up message and is gone once the pop-up window is closed. The rendering is the same for bound and unbound transition messages. The following example refers to a bound transition message in an action implementation. The action `AcceptTravel` modifies the travel entity and changes the overall status (field overall_status) to 'A' (accepted). To make sure that the state transition was successful, an information message is thrown after the action was called to confirm that the overall status was indeed changed. Because this change triggers a change of the business object state and is directly related to the travel entity, the message is defined as a bound transition message and is allocated to the `reported-travel` structure. Furthermore, since the request is related to the field `overall_status`, this field is added as element target. This doesn't directly affect the UI, but the information is contained in the OData Metadata. This example doesn't pass any message variables, but they can be passed by filling the defined import parameters of the /DMO/MESSAGEWRAPPER class. For a generic example of a message with message variables, you can refer to State Messages. :

```
APPEND VALUE #( %MSG = NEW /DMO/MESSAGEWRAPPER(
                                                TEXTID   = DMO/MESSAGEWRAPPER=>ACTION_APPROVAL
                                                SEVERITY = IF_ABAP_BEHV_MESSAGE=>SEVERITY-INFORMATI
                                           %ELEMENT-OVERALL_STATUS = IF_ABAP_BEHV=>MK-ON
) TO REPORTED-Travel.
```

This transition message is rendered as a pop-up on the UI with OData V4 once the respective action is triggered and completed. The pop-up depends on the message severity - messages with severity information, warning and error are displayed in a pop-up that can be closed by the user:



Since usually no action is required if there's a success message, a transition message with this severity appears as a brief pop-up and disappears on its own without any user interaction:



State Messages

State messages refer to a business object instance and its values.

State Messages

⚠ Caution

A state message must always be bound to a business object entity and can't be allocated to the %OTHER component. You musn't define state messages in an action or function added on projection level. If a state message is issued in a projection, this leads to a short dump.

⚠ Caution

If a rollback is triggered in the context of an exposed RAP OData service, the state of a business object is returned to the state it had before the request as executed. Since state messages always reflect the current state of the persisted entity, state messages triggered after the initial request and before the rollback are invalid with regards to the persisted entity. As a consequence, the framework converts state messages triggered during this time frame to transitions messages that are then allocated to the `REPORTED` structure of the respective `MODIFY` request.

If a business object is consumed via EML in a non-RAP scenario, and the SAVE is canceled in the `CheckBeforeSave` phase, the business object remains in its state and the state messages are preserved.

State messages refer to a business object instance and its values. For a business object with draft capabilities, they're 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. Messages in validations and determinations (that are part of a determine action or the Prepare action in a draft scenario) can typically be considered as state messages. Validations usually check the business object values for inconsistencies, thus reflecting the business object state, whereas determinations trigger changes to the business object state. Depending on your scenario, state messages may be converted to transition messages.

Example

If a validation returns an error message regarding an incorrect value in a field, the state message is persisted until the value is changed and the save sequence is triggered again.

Using state messages is only recommended for the following cases:

- **Business Object with Draft Capabilities**: Determinations and Validation that are allocated to the PREPARE or a determine action
- **Unmanaged Business Object**: Finalize/Check Before Save Code Exit
- **Managed Scenario**: Determinations/Validation on Save

The `%state_area` Component

State messages are defined as such when the `%state_area` component is filled in with a string in the `REPORTED` structure. Regarding the naming, it's recommended to choose a name that uniquely identifies the condition that the message originates from. For example, if a validation checks if a **CustomerID** the user entered is consistent with customers stored in a customer table, 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` . This value isn't displayed on the UI nor is it contained in the OData metadata - the `%state_area` is only used to clear state messages from the corresponding message table.

Consequently, you need to define a `%state_area` for each unique condition you're checking against and want to be able to invalidate your messages for.

The `%path` Component for Child Entities

The `%path` component must be filled in for all child entities of a root entity to explicitly map child entities to the parents. For a direct child of a root, the `%path` component maps the child entity to the specific parent instance using the primary key and the draft-indicator. In the following example the draft `parent` (%is_draft) is mapped to the `child` draft via the instance keys:

```
%path       = VALUE #( <root>-%is_draft    = <child>-%is_draft
                       <root>-<key>        = <child>-<parent_key_in_child_entity> )
```

For a business object consisting of three entities, the second child entity is mapped to the direct parent entity and additionally to the root of the business object. The `%path` component allows efficient mapping between the entities at runtime.

```
%path       = VALUE #( <root>-%is_draft   = <child_2>-%is_draft
                       <root>-<key>       = <child_2>-<root_key_in_child2_entity>
                       <parent>-%is_draft = <child_2>-%is_draft
                       <parent>-<key>     = <child_2>-<parent_key_in_child2_entity> )
```

For a specific implementation example, refer to Sample Implementation: `validateBookingDate` (Booking).

Invalidating State Messages

State messages must be invalidated so that the messages aren't continuously added to a REPORTED structure if the same request is triggered multiple times on the same instance. You can only invalidate messages belonging to the same `%state_area` in one statement. Each unique `%state_area` needs to be invalidated separately. Use the following syntax to invalidate state messages in context of a **Managed Business Object**:

```
APPEND VALUE #( %tky        = instance-%tky
                %state_area = 'state_area' ) TO REPORTED-BUSINESS_OBJECT_ENTITY.
```

The `%state_area` component invalidates all messages that were added to the REPORTED structure with the same state area property for the instance with the `%tky` component. Since the `%msg` is undefined, all `%msg` that were added beforehand with the respective `%state_area` are removed by the framework from the message table:

```
"Clear state area for instance
APPEND VALUE #( key       = instance-key
                %state_area = if_abap_behv=>state_area_all ) TO REPORTED-BUSINESS_OBJECT_ENTITY.
```

Generic Message Implementation: State Message

A message is interpreted as state message once the `%state_area` component is filled in. The following generic implementation is a state message with two target fields that is allocated to a business object entity passing one field as a message variable:

```
APPEND VALUE #(
      %tky   = instance-%tky

      %msg   = NEW message_exception(
                    textid               =  message_exception=>message_exception_constant
                    severity             =  if_abap_behv_message=>severity-severity
                    message_variable     =  instance-field )
      %element-field1                    =  if_abap_behv=>mk-on
      %element-field2                    =  if_abap_behv=>mk-on
      %element-_association              =  if_abap_behv=>mk-on
      %op-%create                        =  if_abap_behv=>mk-on
      %op-%action-action1                =  if_abap_behv=>mk-on
      %state_area                        =  'state_area'
      %path                              =  VALUE #( <root>-%is_draft = <child>-%is_draft
                                                     <root>-<key>     = <child-<parent_key_in_chil
```

For more information, about how state messages are displayed on the UI and a more specific implementation example, refer to State Messages on the UI.

For an example implementation with state and transition messages, refer to Creating a Message Exception Class and Exposing Messages for a Sample Business Object with Draft Capabilities.

State Messages in Read Operations

Read operations, including functions, don't cause any changes to the business object state, but simply return data. Consequently, a read or function implementation can't issue new state messages.

In OData V4, a function with result type entity returns the respective entity including the respective state messages. In OData V2, the entity is returned without state messages as default behavior.

For more information about how messages behave in EML, refer to Message Behavior in EML (Entity Manipulation Language).

State Messages on the UI

This topic shows that state messages are displayed with UI5.

State Messages on the UI

The representation of messages depends on the UI technology and the following screenshots are UI5-specific and the message representation may vary in other cases.

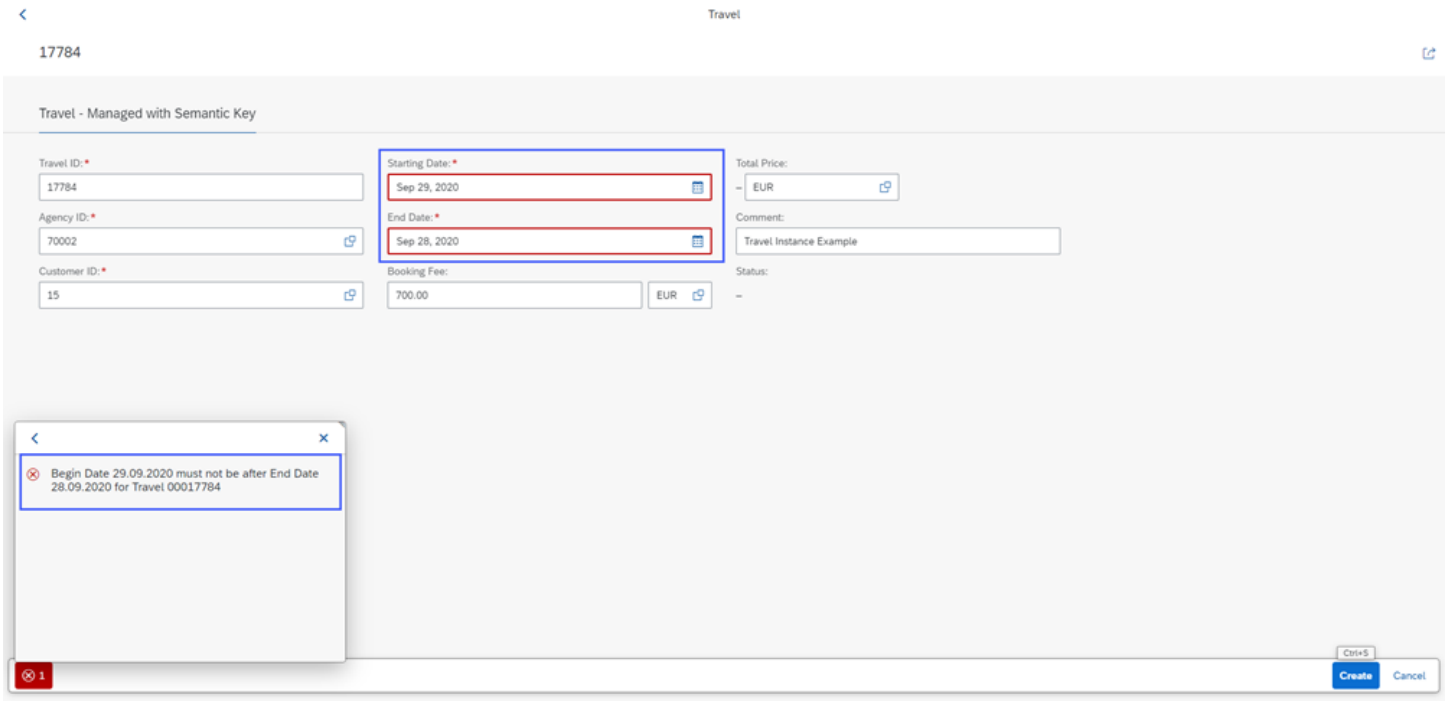
State messages are displayed in a message pop-over and they're persisted until the state of the business object changes. If a message is assigned to field in `%ELEMENT`, the respective field is framed in the severity color to illustrate the link between the field values and a message in order to improve the user experience.

The following example is extracted from the implementation of the `validateDates` method from the managed scenario. This validation checks if the start date is earlier than the enddate. Since a validation refers to the state of business object, the `%state_area` component is filled in with `'VALIDATE_DATES'`. If you implement several state messages within the same implementation, it is recommended to use the same value for all `%state_area` definitions.

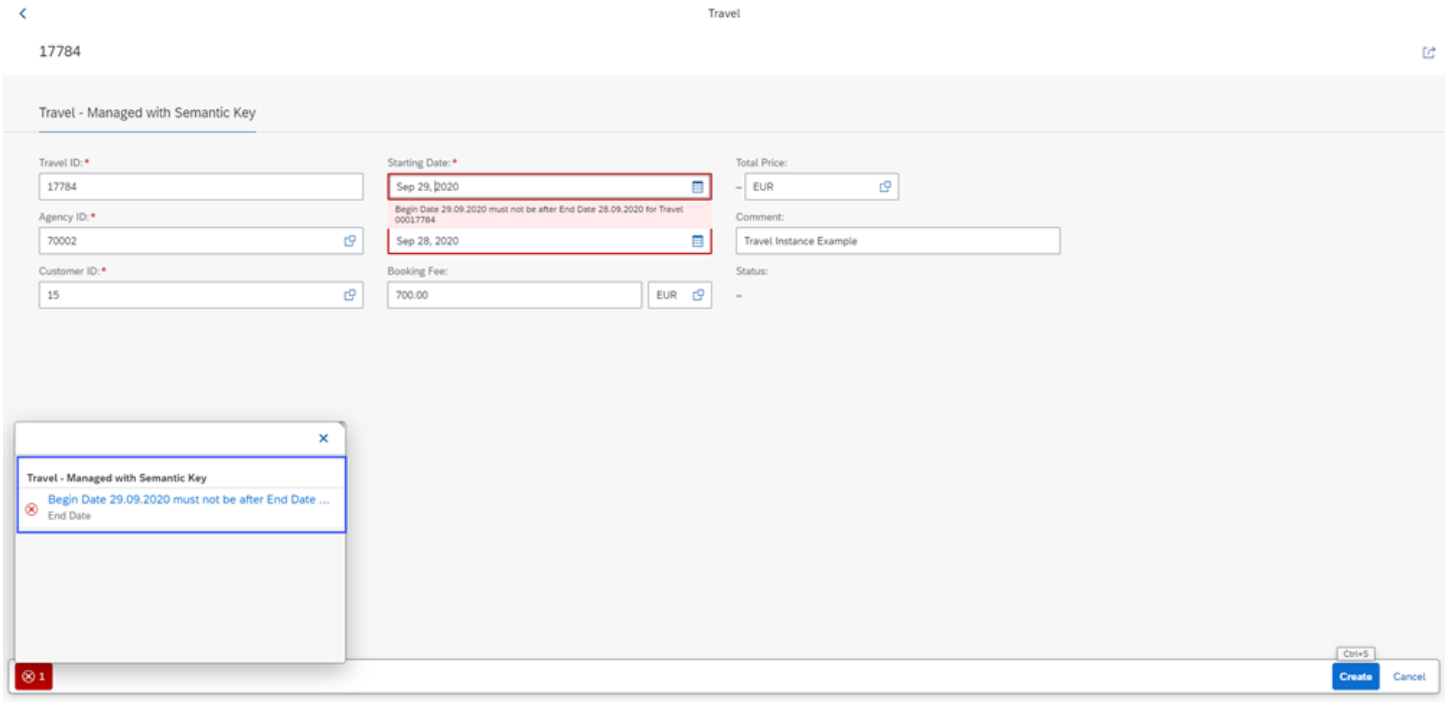
For this validation, two target elements `%element-BeginDate` and `%element-EndDate` are defined, since these field values are checked in the validation. The class `/dmo/cm_flight_home` is used as a message-wrapper class in this case. The specific message id and the respective variables necessary for the message text are contained in the `end_date_before_begin_date` constant. For this message, the `begin_date` (type DATS). `end_date` (type DATS) and `travel_id` (type String) are passed as variables for the message. The message severity is defined as **Error** indicating that the date must be changed by the user for incorrect values. The message is allocated to the reported structure of the travel entity:

```
APPEND VALUE #( %tky        = ls_travel_result-%tky
                %state_area =      'VALIDATE_DATES'
                %msg        = NEW /dmo/cm_flight_home( textid = /dmo/cm_flight_home=>end_date_b
                                                       begin_date = ls_travel_result
                                                       end_date   = ls_travel_result
                                                       travel_id  = ls_travel_result
                                                       severity   = if_abap_behv_mes
                %element-begindate  = if_abap_behv=>mk-on
                %element-enddate    = if_abap_behv=>mk-on ) TO reported-travel.
```

If a date is incorrect, the state message is rendered as follows after the user tries to save the travel instance:



The defined target elements are framed in red to indicate the link between the message and the respective fields. As a state message, the validation result appears in the message box on the lower left. The defined message length exceeds the maximum length for a short text and is automatically displayed in the longtext view so that the complete text is readable for the user. The message is allocated to the travel entity that has the label **Managed- Travel with Semantic Key** - the heading for the message is always derived from the business object label to whose reported structure the message was allocated. Furthermore, the state messages enable the user to navigate between the messages and the affected field. The respective message then appears on the bottom of the affected fields:



Related Information

[State Messages]
[Creating a Message Exception Class]
[Exposing Messages for a Sample Business Object with Draft Capabilities]

Message Mapping

In cross BO scenarios, message mapping ensures that messages reported by a foreign business object can be adapted to the message context of your own business object.

Foreign Entity Definition

To be able to map messages from a foreign business object to your own business object, you need to define the entities that are expected to raise messages as foreign entities in the header of your behavior definition.

```
…
foreign entity myForeignEntityA;
foreign entity myForeignEntityB;
…
define behavior …
```

This adds the defined foreign entities to the derived type of the `reported` parameter structure.

For more information about the syntax of foreign entities, refer to CDS BDL - foreign entity (ABAP Keyword Documentation) .

Map_Messages Handler

The map_messages handler is a method inherited from the class cl_abap_behavior_handler which you can redefine in your saver class. The messages that are raised by the defined foreign entities are available in the changing parameter reported of this method.

Use the map messages handler to assign messages from foreign entity instances to the corresponding entity instances of your own business object. Remove the messages from the respective foreign entity tables of the reported structure afterwards. You can also use the handler to filter messages beforehand, for instance, based on the assigned severity.

If the map_messages handler is used to assign messages from foreign entities to own entities, the provider needs to ensure the semantically correct message relation from foreign entity instances to own entity instances.

Multiple Map_Messages handlers

The map_messages handler can be implemented both on the base and on the projection layer of a business object. Furthermore, multiple map_messages handlers from different business objects can be involved in the reporting of messages. Also extensions can add map_messages handlers to business objects. The result of messages processed by multiple map_messages handlers depends on the particular operations these handlers perform:

- A message is deleted if all handlers either delete the respective reported entry or clear its %MSG component.
- A message is kept if at least one handler leaves the respective reported entry untouched. An empty map_messages handler is treated as an instruction to keep all messages.
- A message that is replaced multiple times by different handlers results in one of these messages. Which message remains, is not defined.
- Handlers can express neutrality towards a message by clearing the respective %MSG component. This leaves a message unchanged but allows changes by other handlers. If a message is cleared, operations that are performed on other messages for the same instance key (delete, keep or replace) are ignored.

Related Information

Mapping Messages Between Business Objects

Message Behavior in EML (Entity Manipulation Language)

This topic describes how transition and state messages behave in EML.

Transition Messages

As transition messages are semantically related to the current request and not a business object state, transition messages are returned with the `REPORTED` structure of the respective EML statement.

For example, if an action throws a transition message and the actions is triggered with a `MODIFY` statement, the transition message is returned with the `REPORTED` structure of the same statement. Transition messages are bound to a request, so the message can't be accessed at a later point in time, meaning it cannot be returned by a subsequent `READ` statement.

State Messages

As state messages are semantically related to the state of business object, state messages aren't returned with the `REPORTED` structure of the EML request that changes the state of the business object, but can instead only be accessed via a `READ`. All thrown state messages are pooled during the `MODIFY` operations and are returned with a `READ` on the business object entity for which the `MODIFY` requests were triggered.

For example, if a `MODIFY - CREATE` triggers a validation during the save sequence that throws a state message, this message isn't contained in the `REPORTED` structure of the same request, but in the `REPORTED` structure of the next `READ` on the same instance.

> ⚠ **Caution**
>
> If a rollback is triggered in the context of an exposed RAP OData service, the state of a business object is returned to the state it had before the request as executed. Since state messages always reflect the current state of the persisted entity, state messages triggered after the initial request and before the rollback are invalid with regards to the persisted entity. As a consequence, the framework converts state messages triggered during this time frame to transitions messages that are then allocated to the `REPORTED` structure of the respective `MODIFY` request.
>
> If a business object is consumed via EML in a non-RAP scenario, and the SAVE is canceled in the `CheckBeforeSave` phase, the business object remains in its state and the state messages are preserved.

Example: Managed Business Object with Draft

The following example illustrates the message behavior of state and transition messages:

A validation belonging to a business object throws one transition message and one state message. This example is only used for message behavior comparison - generally speaking a validation would rather throw state messages than transition messages. First two draft instances are created via an EML `CREATE`. Then the two entities are committed to the draft table. On the draft table, the validation is triggered which check for the value of `sample_field`.

The generic validation implementation follows the usual action implementation pattern. The validation is triggered during the `Prepare` and checks if the content of the field `sample_field` is valid:

```
[implementation] unmanaged|managed|abstract [in class class_name unique];
with draft;

...
define behavior for CDSEntity travel
implementation in class travel_implementation [unique]
```

```
...
{
  validation sampleValidation on save { field sample_field; }
  draft determine action Prepare { validation sampleValidation; }

}
```

Within the `sampleValidation`, there are two messages implemented. The state message is triggered if an incorrect value was entered in the `sample_field` and has the severity error, the transition message confirms that the contained value is correct and has the severity success:

ℹ *Expand the following code sample to view the source code of the method* ***`sampleValidation`****.*

Sample Code

```
METHOD sampleValidation.

//State message is invalidated
      APPEND VALUE #(  %tky        = k-%tky
                       %state_area = 'sampleValidation' ) TO reported-travel.

[...Code to check value ]

    IF value_is_incorrect.

      APPEND VALUE #(  %tky = key-%tky ) TO failed-travel.
      APPEND VALUE #(  %tky                 = key-%tky
                       %state_area          = 'sampleValidation'
                       %msg               = NEW messagewrapper( textid = messagewrapper=>validat
                                                                travel_id = ls_travel-travelid
                                                                severity  = if_abap_behv_message=
                       %element-sample_field  = if_abap_behv=>mk-on ) TO reported-travel.

    ELSE.
          //value is correct

      APPEND VALUE #(  %tky     = key-%tky
                       %msg     = new messagewrapper( textid = messagewrapper=>validation_value_
                                                      travel_id = ls_travel-travelid
                                                      severity  = if_abap_behv_message=>severity
                       %element-sample_field  = if_abap_behv=>mk-on ) to reported-travel.


    ENDIF.



ENDMETHOD.
```

Now, two CREATE operations are triggered via EML. In the first CREATE, an incorrect value is passed for the `sample_field`. In the second CREATE, a correct value is provided. The subsequent `Prepare` actions for both created instances trigger their validation. Hereby only the prepare for the instance with the correct value provides an entry in reported as the corresponding message has been implemented as transition message. Next, an EML read is performed on both instances. Hereby only the read for the instance with the incorrect value provides an entry in reported as the corresponding message has been implemented as state message. Provided that this state message is persisted using COMMIT ENTITIES, it can even be retrieved in a later session for the instance.ℹ *Expand the following code sample to view the source code*

Sample Code

```
// Trigger two creates via EML

//Returns state message
    MODIFY ENTITIES OF /dmo/BusinessObject
    ENTITY Bo_Entity
    CREATE SET FIELDS WITH
    VALUE #( ( %tky      = VALUE #( ID        = '1234'
                                    %is_draft = if_abap_behv=>mk-on  )
               %data     = VALUE #(

                           sample_field = 'incorrectValue'
                           [...fill in other required fields for CREATE]
                ) )
    FAILED  DATA(create_failed)
    MAPPED  DATA(create_mapped)
    REPORTED  DATA(create_reported).


//Returns transition message in reported
    MODIFY ENTITIES OF /Dmo/BusinessObject
    ENTITY Bo_Entity
    CREATE SET FIELDS WITH
    VALUE #( ( %tky  = VALUE # ( ID        = '12345'
                                 %is_draft  = if_abap_behv=>mk-on  )

               %data = VALUE #(

                           sample_field = 'correctValue'
                           [...fill in other required fields for CREATE]
                ) )
    FAILED  DATA(create_failed_2)
    MAPPED  DATA(create_mapped_2)
    REPORTED  DATA(create_reported_2).

//Commit created instances to draft table

    COMMIT ENTITIES RESPONSE OF /Dmo/BusinessObject
    FAILED DATA(COMMIT_FAILED)
    REPORTED DATA(COMMIT_REPORTED).

//Execute prepare to trigger state message - state message not contained in reported

    MODIFY ENTITIES OF /Dmo/BusinessObject
    ENTITY  Bo_Entity
    EXECUTE prepare from  VALUE #( ( id = '1234' ) )

    FAILED   DATA(prepare_failed)
    MAPPED   DATA(prepare_mapped)
    REPORTED DATA(prepare_reported).

//Execute prepare to trigger messages to trigger transition message - transition message contain
```

```
      MODIFY ENTITIES OF /Dmo/BusinessObject
      ENTITY  Bo_Entity
      EXECUTE prepare from  VALUE #( ( id = '12345' ) )

      FAILED   DATA(prepare_failed_2)
      MAPPED   DATA(prepare_mapped_2)
      REPORTED DATA(prepare_reported_2).



  // Read on draft travel entity 1234 - returns the state message for this instance

      READ ENTITIES OF /Dmo/BusinessObject
      ENTITY Bo_Entity
      FIELDS (ID )
      WITH VALUE #( (   ID = '1234'
                        %is_draft = if_abap_behv=>mk-on ) )
      RESULT DATA(READ_RESULT)
      REPORTED DATA(READ_REPORTED_STATEM)
      FAILED DATA(READ_FAILED_STATEM).

  //Read on draft travel entity 12345 - reported is empty


      READ ENTITIES OF /Dmo/BusinessObject
      ENTITY Bo_Entity
      FIELDS ( ID )
      WITH VALUE #( (  ID        = '12345'
                      %is_draft = if_abap_behv=>mk-on  ) )
      RESULT DATA(READ_RESULT_)
      REPORTED DATA(READ_REPORTED)
      FAILED DATA(READ_FAILED).

  //In new session - state message was persisted for draft and is returned with READ_REPORTED_STATI

      READ ENTITIES OF /Dmo/BusinessObject
      ENTITY Bo_Entity
      FIELDS ( ID )
      WITH VALUE #( (   ID        = '1234'
                        %is_draft = if_abap_behv=>mk-on ) )
      RESULT DATA(READ_RESULT)
      REPORTED DATA(READ_REPORTED_STATEM)
      FAILED DATA(READ_FAILED_STATEM).
```

Result

The two messages are allocated as follows:

- State Message Instance 1234: The persisted state message is allocated in the READ_REPORTED_STATEM strcuture. Even if a new session is started, the READ on instance 1234 returns the state message until the value is changed and the PREPARE is triggered again. When the business object is consistent in the draft instance, the business object instance can be persisted on the data base.

- Transition Message Instance 12345: The transition message is returned with the `prepare_reported_2` structure.

OData V2

This topic describes how messages are modeled in OData V2.

Messages in OData V2

Messages in OData V2 aren't modeled as entities, but are returned together with the business data.

Successful Request (http response 2xx)

If the request is successful (http response 2xx), messages are contained in the custom response header `sap-message` with the following structure:

- **Message Code**: A machine-readable code
- **Message Text**: Message text defined in the T100 message class.
- **Message Targets**: Message targets are defined with the `%element` component. A message can have one or multiple targets. If multiple targets are defined, they're modeled as an array of additionalTargets. In this case, each item in this array is a string with the same syntax as target.
- **Severity**: The severity reflects the severity defined in the `%msg` component with the statement `SEVERITY = IF_ABAP_BEHV_MESSAGE=>SEVERITY-SEVERITY`.
- **Transition indicator**: An optional transition indicator - transition messages originate during transition from one backend state to another backend state, for example, during execution of an action. Transition messages are flagged as `transition:true` and state messages are flagged as `transition:false`.
- **Detail messages:** Zero or more details, each with a code, message, severity, target, and optional an additionalTargets array and a transition indicator.

The content of the SAP-Message header uses the same format (Atom/XML or JSON) as the response body.

The `sap-message` header is structured as follows:

```
sap-message: {
  "code": "DMO_BUSINESSOBJECT_MESSAGES/002",
  "message": "Message text as defined in T100 message class",
  "severity": "info",
  "transition": true,
  "target": "to_Target"
  "details": [

  ]
}
```

The code is composed of the T100 message class the message originates from and the message identifier. This message example has the severity info and was defined as transition message that has the target `_Target`.

Not Successful (Http Response Code 4xx [Client Error]/ 5xx [Server Error])

If a request isn't successful, messages are returned with the http body. If the response contains multiple messages, they're arranged hierarchically below the first returned message. The first message is described with the following properties:

- **Lang**: Language in which the server returned the message (response language is derived from request language).
- **Message**: Message text as defined in the respective T100 message class.

The properties within the response are structured as follows:

```
"error": {
  "code": "DMO_BUSINESSOBJECT_MESSAGES/001",
  "message": {
    "lang": "en",
    "value": "Message Text as defined in T100 message class."
  }
```

Details about all messages are contained in the error details block:

```
      "errordetails": [
        {
          "code": "DMO_BUSINESSOBJECT_MESSAGES/001",
          "message": "Message text as defined in T100 message class.",
          "propertyref": "",
          "severity": "error",
          "transition": true,
          "target": ""
        },
        {
          "code": "DMO_BUSINESSOBJECT_MESSAGES/002",
          "message": "Second text as defined in T100 message class.",
          "propertyref": "",
          "severity": "success",
          "transition": true,
          "target": "to_Target"
        },
      ]
```

In this example, both messages have the property `transition:true` indicating that they were defined as transition messages (no %state_area defined in the implementation). The `code` indicates which message class the message was created in, the numeric value reflects the message identifier defined in the T100 message class. The first example wasn't bound to target because of which the property is undefined. The second message was assigned to the target `_Target` with the `%element` component.

OData V4

This topic describes how messages are modeled in OData V4.

Messages in OData V4

Bound Messages in OData V4 are modeled as a complex type named `sap__messages` and unbound messages (messages allocated to `%other`) are transported with the response header.

Unbound Messages

Unbound messages are transported with the response header property `sap__messages` that has the following structure:

- **Message Code**: A machine-readable code
- **Message Text**: Message text defined in the message class.
- **Numeric Severity**: The numeric severity corresponds to the severity defined in the `%msg` component with the statement `SEVERITY = IF_ABAP_BEHV_MESSAGE=>SEVERITY-SEVERITY`. The mapping is as follows:
  - **Success**: 1 (Success - no action required)
  - **Info**: 2 (Information - no action required)
  - **Warning**: 3 (Warning - action may be required)
  - **Error**: 4 (Error - action is required)
- **Optional longtextUrl**: Contains the URL to the longtext, if the message was defined with a longtext in the message class.
- **Optional Target**: The target relates a detail message to (a part of) an OData resource, or a related OData resource. This link required for errors resulting from validation to establish a visual link between the message and the affected fields. It's possible to define one or multiple targets.
- **Message Targets**: Message targets are defined with the `%element` component. A message can have one or multiple targets. If multiple targets are defined, they're modeled as an array of additionalTargets. In this case, each item in this array is a string with the same syntax as target.

The content of the `sap-messages` header uses JSON and is encoded according to the rules for HTTP header fields.

```
sap-messages: [
  {
    "code": "DMO_BUSINESSOBJECT_MESSAGES/002",
    "message": "Message text as defined in message class",
    "numericSeverity": 2,
    "longtextUrl": "..."
    "target": "to_Target"
  }
]
```

The code is composed of the message class the message originates from and the message identifier. This message example has the severity info (2) and was defined with a longtext and has the target `_Target`.

Successful Request (http response 2xx): Bound Message

If a request (http response 2xx) is successful, bound messages are contained in an explicitly modeled collection-valued message container property to avoid header size problems. This complex type has the following properties:

- **Message Code**: A machine-readable code
- **Message Text**: Message text defined in the message class.

- **Message Target**: Message targets are defined with the `%element` component. A message can have one or multiple targets. If multiple targets are defined, they're modeled as an array of additionalTargets. In this case, each item in this array is a string with the same syntax as target.
- **Numeric Severity**: The numeric severity corresponds to the severity defined in the `%msg` component with the statement `SEVERITY = IF_ABAP_BEHV_MESSAGE=>SEVERITY-SEVERITY`. The mapping is as follows:
  - **Success**: 1 (Success - no action required)
  - **Info**: 2 (Information - no action required)
  - **Warning**: 3 (Warning - action may be required)
  - **Error**: 4 (Error - action is required)
- **Transition indicator**: An optional transition indicator - transition messages originate during transition from one backend state to another backend state, for example, during the execution of an action. Transition messages are flagged as `transition:true` and state messages are flagged as `transition:false`.
- **Optional longtextUrl**: Contains the URL to the longtext, if the message was defined with a longtext in the message class.

The `sap__messages` is structured as follows:

```
sap-messages: {
  "code": "DMO_BUSINESSOBJECT_MESSAGES/002",
  "message": "Message text as defined in message class",
  "target": "to_Target"
  "numericSeverity": "3",
  "transition": true,
  "longtextUrl": "..."
}
```

The code is composed of the message class the message originates from and the message identifier. This message example has the severity warning (3) and was defined as transition message that has the target `_Target`. Furthermore, the message was defined with a longtext.

Not Successful (Http Response Code 4xx [Client Error]/ 5xx [Server Error]): Bound Message

If a request (http response 2xx) Code 4xx [Client Error]/ 5xx [Server Error]), the error response has the following structure:

- **Message Code**: A machine-readable code
- **Message Text**: Message text defined in the message class.
- **Message Target**: Message targets are defined with the `%element` component. A message can have one or multiple targets. If multiple targets are defined, they're modeled as an array of additionalTargets. In this case, each item in this array is a string with the same syntax as target.
- **Details**: Zero or more details, each with a code, message, and a target.

The error response is extended with instance annotations, to add a severity to detail messages, or an array of additionalTargets or the longtext URL:

```
{
  "error": {
    "code": "UF0",
    "message": " "Message text as defined in message class"",
    "target": "",
```

```
    "@Common.additionalTargets": [],
    "@Common.longtextUrl": "...",
    "details": [
      {
        "code": "UF1",
        "message": "Message text as defined in message class",
        "target": "$_Target",
        "@Common.additionalTargets": [],
        "@Common.numericSeverity": 4,
        "@Common.longtextUrl": "..."
      },
         ...
        ]
  }
}
```

The code is composed of the message class the message originates from and the message identifier. There was no target defined for this message. The second example was defined with a longtext, the target `_Target`, and the severity 4 (error).

Query

A query is the connecting interface for read-only access to the database in OData services. It is used for list reports or analytical reports to process data.

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

Data Model

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

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

Note

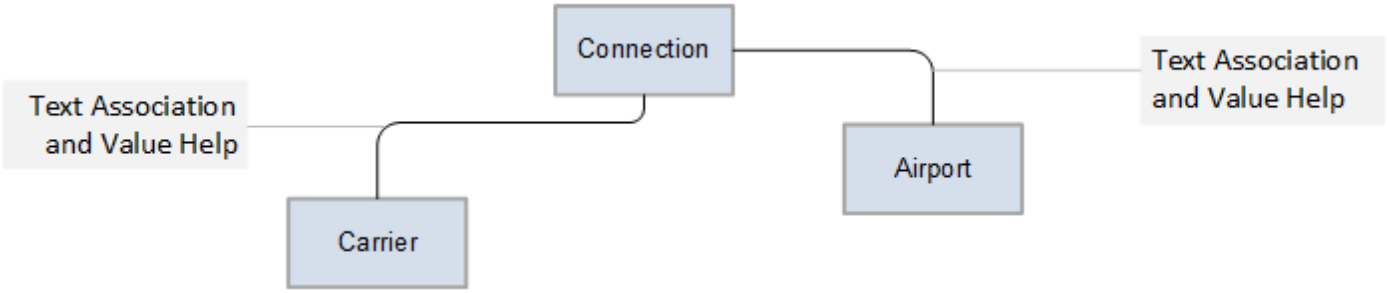
The RAP runtime engine can only handle associations to or from a custom entity with attribute bindings (A1 = A2 and B1 = B2), but no:

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

Example

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

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



Loose Combination of CDS Entities with Associations for Query Capabilities

Query Capabilities

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

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

The following table lists the available query capabilities.

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

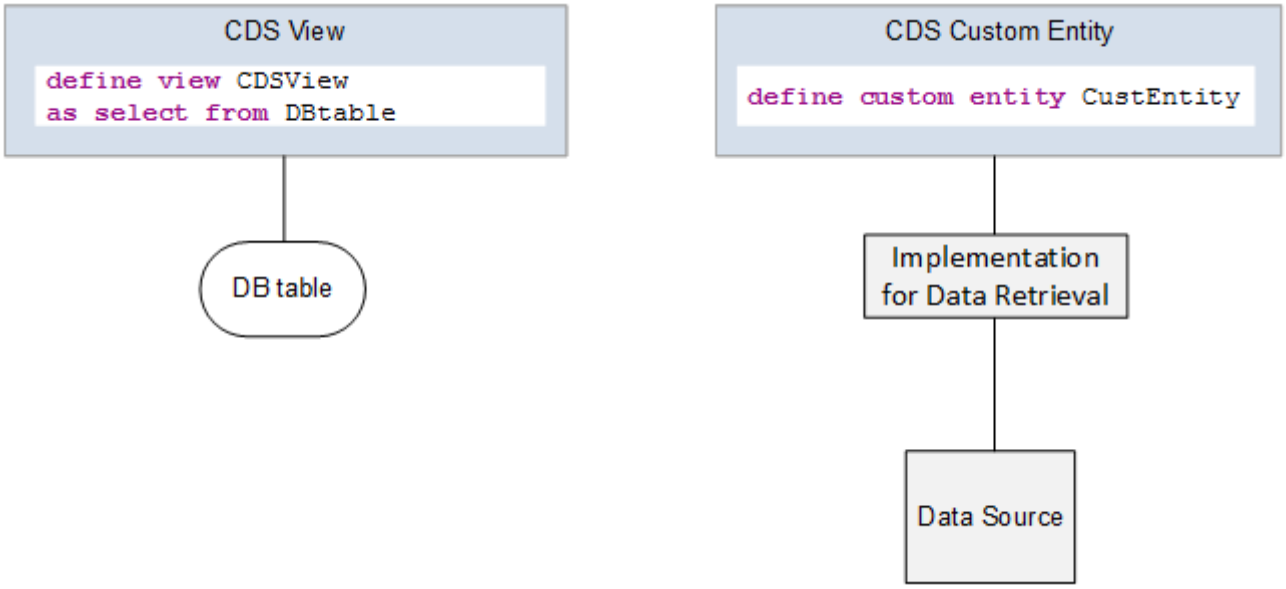
All of these features do not modify data on the database but process data to structure them for a certain output.

Query Runtime

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

There is also the option to handle the query manually. We speak of an unmanaged query in this case. An unmanaged query can be used, for example, if the data source of a query is not a database table. That means, the framework cannot provide the SQL statement to access the database. Instead, the application developer needs to implement every query capability to retrieve the data matching the OData request. For the unmanaged implementation type, the data model is manifested in a CDS view containing the annotation `ObjectModel.query.implementedBy:`. This annotation is used to reference a query implementation class which implements the data retrieval.

The following diagram exemplifies the runtime of a managed and an unmanaged query. In the shown example, a custom entity is used to structure the data coming from the referenced query implementation.



Managed and Unmanaged Query in Contrast

For more information about the query runtime, see Query Implementation Types.

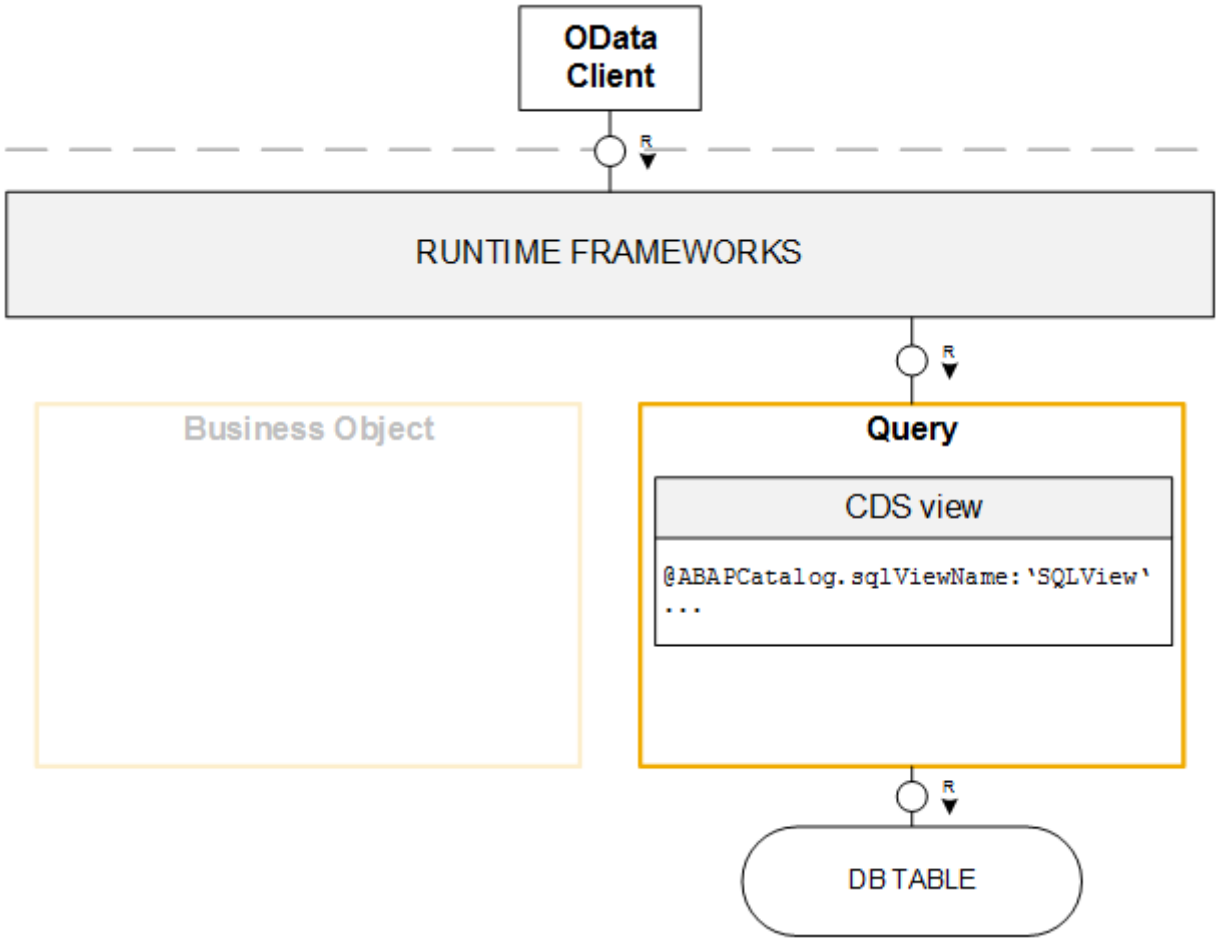
For a detailed runtime diagram, see Query Runtime.

Query Implementation Types

Managed Query

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

The following diagram illustrates the runtime of a query.



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

Managed Query - Runtime

Example

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

For a detailed runtime diagram, see Query Runtime.

Unmanaged Query

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

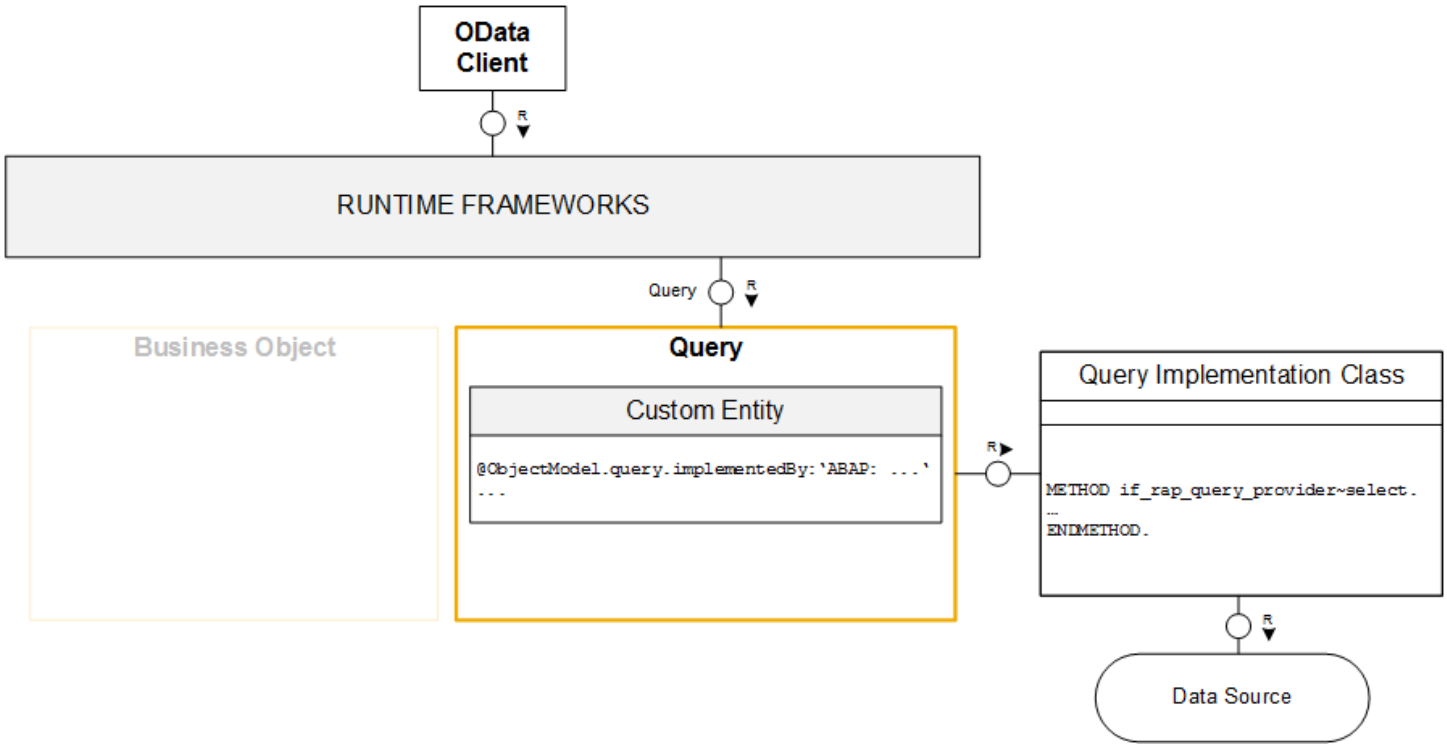
Use cases for unmanaged queries are

- the data source for an OData request is not a database table, but, for example another OData service, which is reached by an OData client proxy,
- performance optimization with application specific handling,
- using AMDPs with some query push-down parameters in the SQL script implementation,
- forwarding the call to the analytical engines, or

- enrichment of query result data on property or row level, for example when splitting rows for intermediate sums or condensing the filter result.

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

The following diagram illustrates the runtime of an unmanaged query.



Unmanaged Query - Runtime

The data model for an unmanaged query can be defined in a CDS custom entity. A custom entity defines the structure of the data returned by the query. This is done using CDS syntax in a CDS data definition (DDLS). A CDS custom entity does not have an SQL view to select data from a database. Instead, the custom entity specifies an ABAP class that implements the query. The entity annotation `@ObjectModel.query.implementedBy: 'ABAP:...'` is used to reference the query implementation class in the data definition of the CDS custom entity. This annotation is evaluated when the unmanaged query is executed whereby the query implementation class is called to perform the query. For every query request on the CDS custom entity, the implementation class is reinstantiated. The class instance is never cached.

Since no SQL artifact is generated for custom entities and the query is implemented in ABAP, custom entities cannot be used in ABAP SQL or in SQL joins in data definitions.

For details about the syntax of a CDS custom entity, see CDS DDL - DEFINE CUSTOM ENTITY (ABAP Keyword Documentation)

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

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

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

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

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

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

Example

An example on how to use a CDS custom entity and implement an unmanaged query with the interface `IF_RAP_QUERY_PROVIDER` in a query implementation class is given in Implementing an Unmanaged Query.

The use case of an unmanaged query in combination with the client proxy is explained in the develop scenario Developing a UI Service with Access to a Remote Service.

For more information about the interface `IF_RAP_QUERY_PROVIDER`, see Unmanaged Query API.

Note

Custom Entities cannot be projected in CDS projection views.

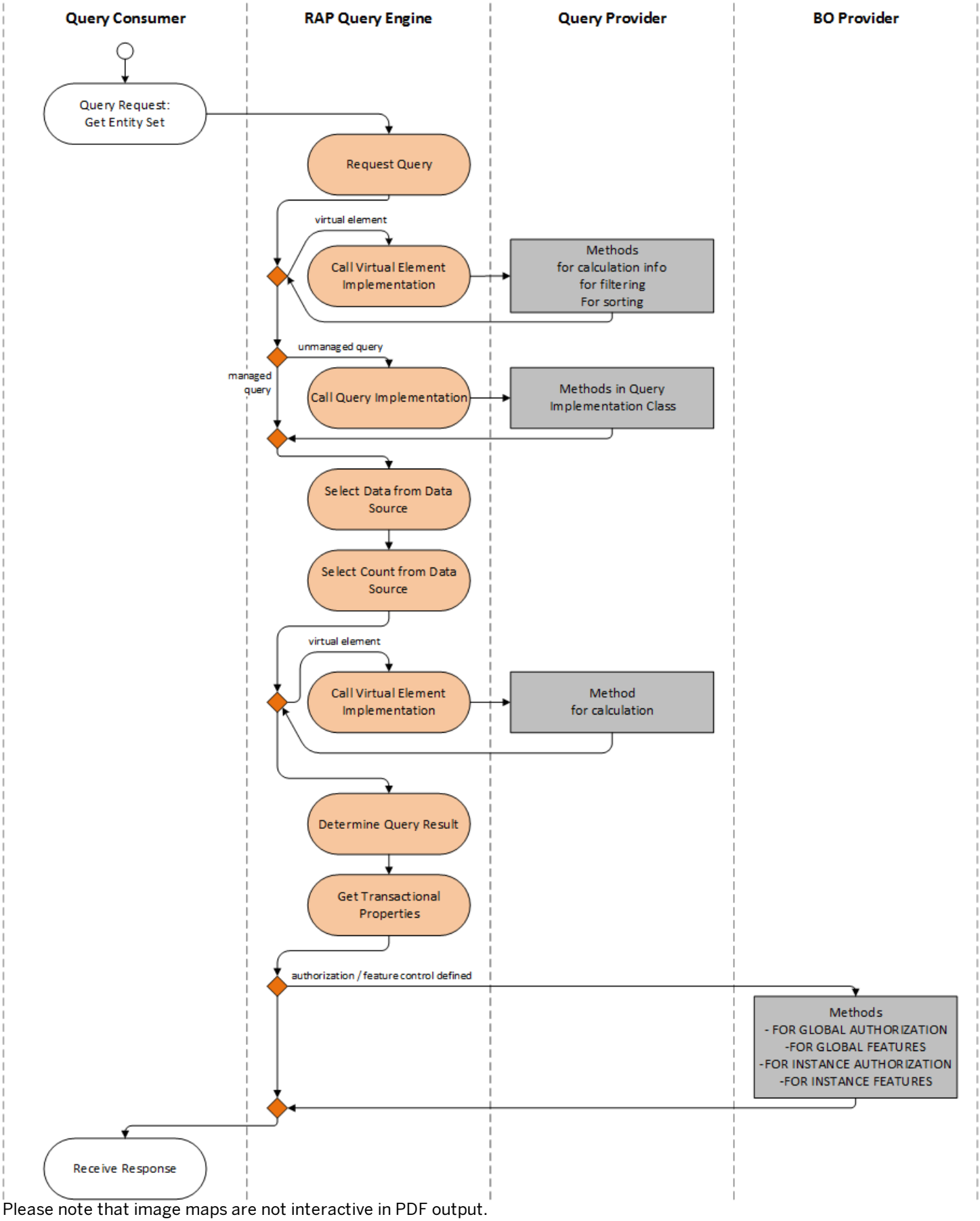
For a detailed runtime diagram, see Query Runtime.

Query Runtime

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

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

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



Please note that image maps are not interactive in PDF output.

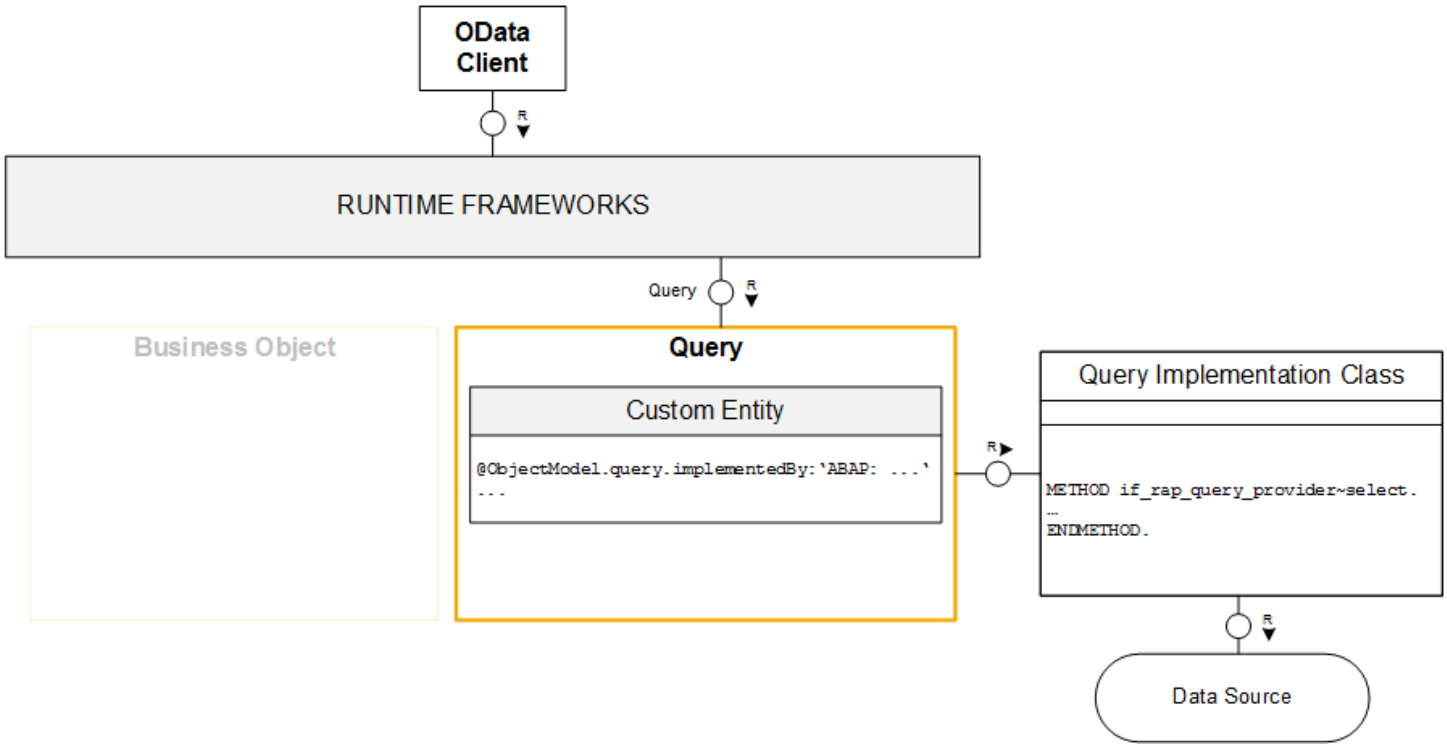
ℹ Note

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

Unmanaged Query API

In contrast to managed queries, in which a framework assumes the implementation tasks to select the requested data from the data source, the implementation of the unmanaged query must be done by the application developer. For this, all desired query capabilities (paging, filtering, sorting, counting, ...) must be implemented in a query implementation class, which is referenced in a CDS view, for example, in a custom entity.

The following diagram illustrates the runtime of an unmanaged query:



The query request is delegated to the query implementation class which must implement the `IF_RAP_QUERY_PROVIDER`. This API is described in following.

Interfaces

These interfaces define methods for the unmanaged query API.

- Interface IF_RAP_QUERY_PROVIDER
  - Interface IF_RAP_QUERY_REQUEST
    - Interface IF_RAP_QUERY_FILTER
    - Interface IF_RAP_QUERY_PAGING
    - Interface IF_RAP_QUERY_AGGREGATION
  - Interface IF_RAP_QUERY_RESPONSE

For more conceptual information about the unmanaged query, see Query Implementation Types.

For an example on how to implement an unmanaged query, see Implementing an Unmanaged Query.

☁ For an example on how to implement the unmanaged query contract in a development scenario, see Implementing the Query for Service Consumption.

ℹ Note

☁Before version 1908, `IF_A4C_RAP_QUERY_PROVIDER` and the related interfaces was the API to implement an unmanaged query. This API is deprecated as of 1908, but still available in ABAP Environment. However, it is recommended to

use only the new interface `IF_RAP_QUERY_PROVIDER`.

The interfaces differ in some aspects, for example the handling of filter requests. The new interface offers some more methods to reflect the query requests in more detail, for example `get_aggregation` or `get_parameters`, which facilitates the implementation.

Interface `IF_RAP_QUERY_PROVIDER`

This interface defines a method that is used for requesting and responding to OData query requests in an unmanaged query.

Method `select`

The method `select` must be implemented in custom entity scenarios. It replaces the SQL-`SELECT` of a CDS view to retrieve and return data. The `select` method must be called by the query implementation class, which is referenced in the custom entity annotation `@ObjectModel.query.implementedBy`.

The `select` imports an interface instance for the request data and one for the response data:

Signature

```
METHODS select IMPORTING io_request  TYPE REF TO if_rap_query_request
                                     io_response TYPE REF TO if_rap_query_response
               RAISING   cx_rap_query_provider.
```

Interface IF_RAP_QUERY_REQUEST

Interface IF_RAP_QUERY_RESPONSE

Parameter

`IO_REQUEST`	Interface instance for gathering request information that are used as input for the select implementation. The request interface provides methods for implementing query options, like filtering or sorting.
`IO_RESPONSE`	Interface instance for the result output of the select implementation.

Exception

`CX_RAP_QUERY_PROVIDER`	Exception that can be raised if there is an error during the query execution.

Example

See Implementing an Unmanaged Query.

Interface `IF_RAP_QUERY_REQUEST`

The interface defines methods to parametrize a query request in an unmanaged query. It is used to handle OData query options for data retrieval.

Method `get_entity_id`

This method returns the CDS entity name of the requested entity set of an OData request in an unmanaged query.

With this method, you can ensure that the query implementation is only executed if the correct entity for this query implementation set is called.

Signature

```
METHODS get_entity_id RETURNING VALUE(rv_entity_id) TYPE string.
```

`rv_entity_id`	CDS entity name of the requested entity set.

Example

See Returning Requested Entity in an Unmanaged Query.

Method `is_data_requested`

This method returns a boolean value to indicate if data is requested.

Note

If this method is used to indicate the request for data, the method `set_data` must be called.

Signature

```
METHODS is_data_requested RETURNING VALUE(rv_is_requested) TYPE abap_bool.
```

Parameter

`rv_is_requested`	If data needs to be returned, the value is `abap_true`. If no data needs to returned, the value is `abap_false`.

Example

See Requesting and Setting Data or Count in an Unmanaged Query.

Method `is_total_numb_of_rec_requested`

This method returns a boolean value to indicate if the total number of records is requested. The total number of records is requested by the query option `$inlinecount` or a `$count` request.

Note

If this method indicates the request for the total number of records, the total count needs to be returned by the method `set_total_number_of_records`.

Signature

```
METHODS is_total_numb_of_rec_requested RETURNING VALUE(rv_is_requested) TYPE abap_bool.
```

Parameter

`rv_is_requested`	If the total number of records needs to be returned the value is `abap_true`. If the total number of records is not requested the value is `abap_false`.

Example

See Requesting and Setting Data or Count in an Unmanaged Query.

Method `get_filter`

This method returns a filter object. This filter object is an interface instance of `IF_RAP_QUERY_FILTER`. If a filter is requested,its methods return the filter information. Only records that match this filter condition must be returned or counted.

Signature

```
METHODS get_filter RETURNING VALUE(ro_filter) TYPE REF TO if_rap_query_filter.
```

Interface IF_RAP_QUERY_FILTER

Parameter

`RO_FILTER`	Contains the filter condition.

Example

See Implementing Filtering in an Unmanaged Query.

Method `get_paging`

This method returns an object with paging information. The paging object is an interface instance of `IF_RAP_QUERY_PAGING`. It limits the number of records to be returned as response data with offset and page size.

Signature

```
METHODS get_paging RETURNING VALUE(ro_paging) TYPE REF TO if_rap_query_paging.
```

Interface IF_RAP_QUERY_PAGING

Parameter

`RO_PAGING`	Contains the paging information.

Example

See Implementing Paging in an Unmanaged Query.

Method `get_sort_elements`

This method returns the sort order for the sort elements.

Signature

```
METHODS get_sort_elements RETURNING VALUE(rt_sort_elements) TYPE tt_sort_elements.
```

Parameter

`rt_sort_elements`	Contains the elements to be sorted with their sort direction. It is an ordered list to define the ranking order, the first element being the primary sort criteria. The table indicates the names of the sort element and the sort order with a boolean value in the column `descending`. The following table illustrates how the returning value looks like. ***tt_sort_elements*** ELEMENT_NAME: ***string*** | DESCENDING: ***abap_bool***

Example

For a filter request like

```
<service_root_url>/<entity_set>?$orderby=Customer_ID desc
```

the method `get_sort_elements` returns the following entries in the returning table:

rt_sort_elements

ELEMENT_NAME	DESCENDING
CUSTOMER_ID	X

Example

See Implementing Sorting in an Unmanaged Query.

Method `get_parameters`

This method returns a list of the entity parameters and their values.

Signature

```
METHODS get_parameters RETURNING VALUE(rt_parameters) TYPE tt_parameters.
```

Parameter

`rt_parameters`	Contains a list of parameters and their given values. ***tt_parameters*** PARAMETER_NAME: ***string*** | VALUE: ***string***

Example

For a filter request like

```
<service_root_url>/<entity_set>(p_start_date=datetime'2016-07-08T12:34',p_end_date=datetime'2019
```

the method `get_parameters` returns the following table:

rt_parameters

PARAMETER_NAME	VALUE
P_START_DATE	20160708
P_END_DATE	20190708

Example

See Using Parameters in an Unmanaged Query.

Method `get_aggregation`

This method returns an aggregation object. This object is an interface instance of `IF_RAP_QUERY_AGGREGATION` which contains methods to indicate which elements need to be aggregated or grouped.

Signature

```
METHODS get_aggregation RETURNING VALUE(ro_aggregation) TYPE REF TO if_rap_query_aggregation.
```

Interface IF_RAP_QUERY_AGGREGATION

Parameter

`ro_aggregation`	Interface instance for information about aggregation and grouping.

Example

See Implementing Aggregations in an Unmanaged Query.

Method `get_search_expression`

This method returns the requested search string.

Signature

```
METHODS get_search_expression RETURNING VALUE(rv_search_expression) TYPE string.
```

Parameter

`rv_search_expression`	Contains a free search expression with unspecified format.

Example

See Implementing Search in an Unmanaged Query.

Method `get_requested_elements`

This method returns the requested elements, which need to be given to the response.

Signature

```
METHODS get_requested_elements RETURNING VALUE(rt_requested_elements) TYPE tt_requested_elements.
```

`rt_requested_elements`	Contains a list of the requested elements.

Example

See Considering Requested Elements in an Unmanaged Query.

Interface `IF_RAP_QUERY_FILTER`

This interface is a filter criteria provider for the unmanaged query. The methods provide different representations for the filter criteria.

Method `get_as_ranges`

This method returns the filter as a list of simultaneously applicable range tables. The table is initial if no filter is supplied.

Signature

```
METHODS get_as_ranges RETURNING VALUE(rt_ranges) TYPE tt_name_range_pairs
                      RAISING cx_rap_query_filter_no_range.
```

Parameter

`rt_ranges`	Contains a list of filter conditions in name-range-table pairs. That means, every requested filter element is related to a ranges table that indicates the filter conditions. The returning value is in a ranges-table-compatible format. The following table illustrates the list of name and ranges table. The columns of the ranges tables have the semantics of selection table criteria. They are defined as follows: • **SIGN**: Contains the values `I` for inclusive or `E` for exclusive consideration of the defined range • **OPTION**: Contains the operator values. Valid operators are `EQ`, `NE`, `GE`, `GT`, `LE`, `LT`, `CP`, and `NP`, if the column high is initial, and `BT`, `NB`, if column high is not initial. • **LOW**: Contains the comparison value or the lower interval limitation. • **HIGH**: Contains the upper interval limitation. ***tt_name_range_pairs***

NAME	RANGE
string	*tt_range_option* SIGN | OPTION | LOW | HIGH *c(1)* | *c(2)* | *string* | *string*

Example

For a filter request like

```
<service_root_url>/<entity_set>?$filter= Agency_ID eq '070031'
                                  and (Begin_Date ge datetime 2019-01-01T00 and Begin_Date le
```

the method `get_as_ranges` returns the following entries in the range table:

tt_name_range_pairs

NAME	RANGE
AGENCY_ID	*tt_range_option* SIGN | OPTION | LOW | HIGH I | EQ | 070031 |
Begin_Date	*tt_range_option* SIGN | OPTION | LOW | HIGH I | BT | 20190101 | 20191231

Exception

`cx_rap_query_filter_no_range`	This exception is thrown if the filter cannot be converted into a ranges table. In this case the developer can try to use the method `get_as_tree` or `get_as_sql_string` as a fall back or throw an error.

Example

See Implementing Filtering in an Unmanaged Query.

Method `get_as_sql_string`

This method returns the filter as an SQL string. The string is initial if no filter is supplied.

Signature

```
METHODS get_as_sql_string RETURNING VALUE(rv_string) TYPE string.
```

Parameter

`rv_string`	Contains the filter conditions as an ABAP SQL string. The variable can be used directly in the `WHERE` clause of an ABAP SQL statement to select data.

Example

For a filter request like

```
<service_root_url>/<entity_set>?$filter= Agency_ID eq '070031'
                                    and (Begin_Date ge datetime 2019-01-01T00 and Begin_Date le
```

the method `get_as_sql_string` returns
`BEGIN_DATE BETWEEN '20190101' AND '20191231' AND AGENCY_ID = '070031'`. This string has the correct syntax to be used in an SQL statement.

See Implementing Filtering in an Unmanaged Query.

The SQL filter can also be applied on data that have been prefiltered using e.g. a ranges table.

Method `get_as_tree`

This method returns a reference to an expression tree representing the filter conditions, which can be easily queried and manipulated due to the semantic tree structure. Every node in the filter tree has a type indicating the content of the node. The reference to the filter tree is unbound in case no filter is supplied.

Signature

```
METHODS get_as_tree RETURNING VALUE(ro_tree) TYPE REF TO if_rap_query_filter_tree.
```

Parameter `ro_tree`

The parameter `ro_tree` references the interface `if_rap_query_filter_tree`. Use the method `get_root_node` of this interface to get the root node of the expression tree. From there you can parse through the filter tree using the following methods of the interface `if_rap_query_filter_tree_node`:

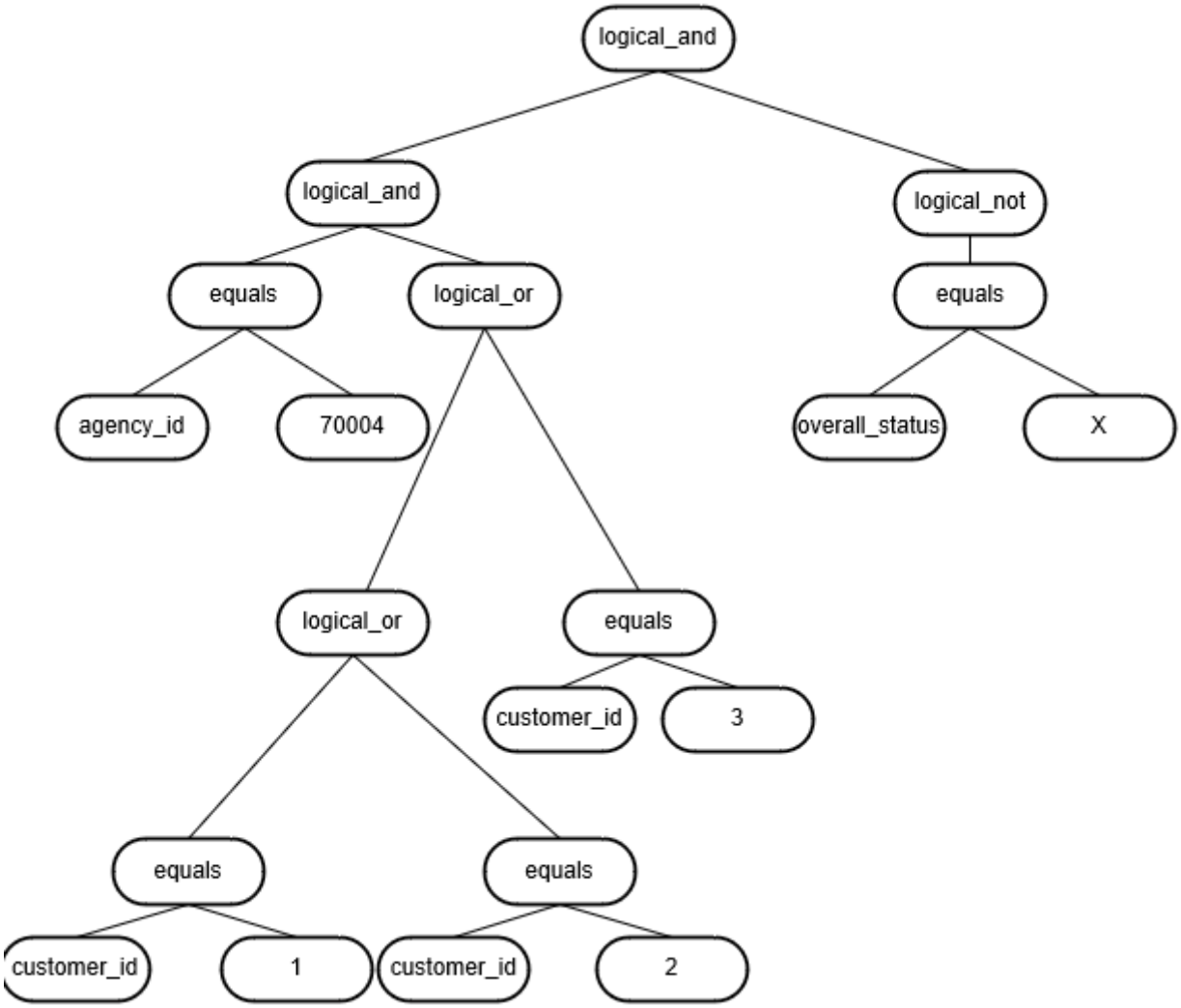
- get_children: Get the child nodes of the current node in the order of the user input.
- get_type: Get the type of the current node.
- get_value: For nodes of the type `identifier` this method returns a reference to a string containing the identifier, i.e. the field involved in the field condition. For nodes of the type `value` this method returns a reference to the raw value. For all other node types this method must not be called. If it is called regardless an uncatchable exception will be thrown.

Example

For a filter request like

```
AGENCY_ID = '070004' AND ( CUSTOMER_ID = '000001' OR CUSTOMER_ID = '000002' OR CUSTOMER_ID = '00(
```

the method `get_as_tree` returns a filter tree with the following semantics:



The structure of the filter tree as well as the order of the nodes within this structure are not guaranteed for the given condition and can change depending on the modifications done to the filter condition.

Consuming code must be prepared to deal with an extension of the available node types as further node types might be added in the future.

Example

See Implementing Filtering in an Unmanaged Query.

Interface `IF_RAP_QUERY_PAGING`

This interface provides the information for paging requests. The methods provide the offset and the page size for one OData request.

Method `get_offset`

This method indicates the number of records to drop from the list of data records in the data source. In an OData query request, the offset is requested by the query option `$skip`.

Signature

```
METHODS get_offset RETURNING VALUE(rv_offset) TYPE int8.
```

Parameter

`rv_offset`	Contains the number of records that are dropped from the result list. **Example** If `rv_offset` is 2, the first record in the result list is the data record on position 3.

Example

See Implementing Paging in an Unmanaged Query.

Method `get_page_size`

This method indicates the maximum number of records that are to be returned. In an OData query request, the page size is requested by the query option `$top`.

Signature

```
METHODS get_page_size RETURNING VALUE(rv_page_size) TYPE int8.
```

Parameter

`rv_page_size`	Contains the number of records that are returned. **Note** `rv_page_size` `if_rap_query_pagin=>page_size_unlimited` if no limit is requested.

Example

See Implementing Paging in an Unmanaged Query.

Interface `IF_RAP_QUERY_AGGREGATION`

This interface provides methods to receive information about the requested aggregation and grouping requests.

Method `get_aggregated_elements`

This method returns the requested aggregated elements with their aggregation method and the output elements in a string table. These values can then be extracted and used in the query implementation.

Signature

```
METHODS get_aggregated_elements RETURNING VALUE(rt_aggregated_elements) TYPE tt_aggregation_eleme
```

Parameter

`rt_aggregated_elements`	Contains the aggregation method, the input element, and the output element. The constants for the available predefined aggregation methods are: • **COUNT:** for returning the number of values of the input element in the output element. The constant `co_count_all_identifier` as value for `input_element` denotes the counting of all rows. • **COUNT_DISTINCT:** for returning the number of unique values of the input element in the output element. • **SUM:** for returning the sum of the input element in the output element. • **MIN:** for returning the minimum of the input element in the output element. • **MAX:** for returning the maximum of the input element in the output element. • **AVG:** for returning the average of the input element in the output element. The input element is the element whose values are aggregated and the output element is the element, which contains the aggregated value. The output element can be the same as the input element. **Example** rt_aggregated_elements **aggregation_method** | **input_element** | **result_element** SUM | BOOKING_FEE | TOTAL_BOOKING_FEE

Signature

Example

See Implementing Aggregations in an Unmanaged Query.

Method `get_grouped_elements`

This method returns the requested elements by which the result is to be grouped.

Signature

```
METHODS get_grouped_elements RETURNING VALUE(rt_grouped_elements) TYPE tt_grouped_elements.
```

Parameter

`rt_grouped_elements`	Returns an ordered list of the elements by which the result is to be grouped. The elements are listed in the order of grouping priority.

Example

See Implementing Aggregations in an Unmanaged Query.

Interface IF_RAP_QUERY_RESPONSE

This interface provides methods to return data and the count for the query response. The results of the methods of interface `IF_RAP_QUERY_REQUEST` are integrated in the response.

Method `set_data`

This method provides the response for the method `if_rap_query_request~is_data_requested`. If this method is called, the table of result data must be provided (empty if there is no result data).

Signature

```
METHODS set_data IMPORTING it_data TYPE STANDARD TABLE
                 RAISING   cx_rap_query_response_set_twic.
```

Parameter

`it_data`	Contains a table of the data records for the query response. Use the type of your custom entity for the response to be compatible with the request.

Exception

`cx_rap_query_response_set_twic`	Exception is raised when the result table is set more than once.

Example

See Requesting and Setting Data or Count in an Unmanaged Query.

Method `set_total_number_of_records`

This method provides the response for the method `if_rap_query_request~is_total_numb_of_rec_requested`. If this method is called, the count needs to be set for the response.

Signature

```
METHODS set_total_number_of_records IMPORTING iv_total_number_of_records TYPE int8
                                    RAISING   cx_rap_query_response_set_twic.
```

Parameter

`iv_total_number_of_records`	Contains the total number of records. If no records match the given request criteria, the value zero must be passed.

Exception

`cx_rap_query_response_set_twic`	Exception is raised when the number of records is set more than once.

Example

See Requesting and Setting Data or Count in an Unmanaged Query.