

ABAP® RESTful Application Programming Model

- › Develop ABAP applications for SAP S/4HANA and SAP BTP
- › Use key tools and technologies, including core data services and SAP Fiori
- › Get step-by-step guidance for modeling data, implementing behaviors, developing user interfaces, and more

2nd edition, updated and expanded

Lutz Baumbusch
Matthias Jäger
Michael Lensch



Rheinwerk
Publishing

Contents

| | |
|----------------|----|
| Foreword | 17 |
| Preface | 19 |

Part I Basic Concepts and Technical Components

| | |
|----------------------------------------------------------------------------------------------|-----------|
| 1 Introduction to the ABAP RESTful Application Programming Model | 25 |
| 1.1 What Is the ABAP RESTful Application Programming Model? | 26 |
| 1.1.1 The Purpose of the Programming Model | 26 |
| 1.1.2 The REST Architectural Style | 30 |
| 1.1.3 OData | 35 |
| 1.1.4 Technological Innovations with SAP S/4HANA | 37 |
| 1.1.5 Evolution of ABAP-Based Programming Models | 38 |
| 1.2 Architecture and Concepts of the ABAP RESTful Application Programming Model | 42 |
| 1.2.1 RAP Transaction Model | 43 |
| 1.2.2 Implementation Types | 43 |
| 1.2.3 Entity Manipulation Language | 45 |
| 1.2.4 Technical Context of a RAP Application and RAP Runtime Environment | 45 |
| 1.3 Development Objects of the ABAP RESTful Application Programming Model | 48 |
| 1.3.1 Data Modeling with Core Data Services | 48 |
| 1.3.2 Behavior Definition | 49 |
| 1.3.3 Behavior Implementation | 49 |
| 1.3.4 Projection Layer | 50 |
| 1.3.5 Business Services | 51 |
| 1.3.6 Interaction of the Artifacts | 51 |
| 1.4 ABAP Development Tools as a Development Tool | 52 |
| 1.5 Quality Attributes of the ABAP RESTful Application Programming Model | 54 |
| 1.5.1 Evolution Capability | 54 |
| 1.5.2 Development Efficiency | 55 |
| 1.5.3 Testability | 56 |

| | | |
|------------|-----------------------------------------------------------------------------|-----------|
| 1.5.4 | Separation of Business and Technology | 56 |
| 1.6 | Availability of the ABAP RESTful Application Programming Model | 57 |
| 1.6.1 | SAP BTP ABAP Environment | 57 |
| 1.6.2 | SAP S/4HANA Cloud ABAP Environment | 58 |
| 1.6.3 | ABAP Platform for On-Premise SAP S/4HANA | 59 |
| 1.7 | The Role of RAP in the ABAP Cloud Development Model | 60 |

2 Core Data Services: Data Modeling 65

| | | |
|------------|------------------------------------------------------------------|------------|
| 2.1 | What Are Core Data Services? | 66 |
| 2.2 | Structure and Syntax of CDS | 69 |
| 2.2.1 | Creating a Basic Interface View | 70 |
| 2.2.2 | Analyzing the Data Model | 74 |
| 2.2.3 | Using CDS Views | 77 |
| 2.2.4 | Extending the Data Model | 78 |
| 2.3 | Associations | 83 |
| 2.4 | Annotations | 86 |
| 2.5 | Access Controls | 91 |
| 2.6 | Extensibility of CDS Entities | 96 |
| 2.6.1 | CDS View Extensions | 97 |
| 2.6.2 | CDS Metadata Extensions | 100 |
| 2.7 | Additional CDS Functionality | 103 |
| 2.7.1 | Virtual Elements | 103 |
| 2.7.2 | CDS Custom Entities | 106 |
| 2.8 | Virtual Data Model | 110 |
| 2.9 | CDS Language Elements for Modeling Business Objects | 114 |

3 Behavior Definition 119

| | | |
|------------|----------------------------------------------------------------------|------------|
| 3.1 | What Is a Behavior Definition? | 119 |
| 3.1.1 | Context and Structure of a Behavior Definition | 120 |
| 3.1.2 | Syntax of a Behavior Definition | 123 |
| 3.1.3 | Possible Behavior | 124 |
| 3.2 | Editing a Behavior Definition in ABAP Development Tools | 131 |
| 3.2.1 | Creating a Behavior Definition | 131 |

| | | |
|-------------|-------------------------------------------------------------------------------------|------------|
| 3.2.2 | Changing and Activating a Behavior Definition | 135 |
| 3.2.3 | Finding and Opening a Behavior Definition | 136 |
| 3.2.4 | Documenting Behavior Definitions and Relationships | 137 |
| 3.3 | Implementation Types | 139 |
| 3.3.1 | Managed Scenario | 141 |
| 3.3.2 | Unmanaged Scenario | 143 |
| 3.4 | Strict Mode | 144 |
| 3.5 | Entity Behavior Definition | 145 |
| 3.6 | Defining a Behavior Pool | 146 |
| 3.6.1 | Behavior Pool for Behavior Definition | 146 |
| 3.6.2 | Behavior Pool for the CDS Entity | 147 |
| 3.6.3 | Behavior Pool for the Implementation Group | 147 |
| 3.7 | Numbering Assignment | 148 |
| 3.7.1 | Early, External Numbering Assignment | 150 |
| 3.7.2 | Early, Internal Numbering | 150 |
| 3.7.3 | Late Numbering | 151 |
| 3.8 | Field Properties | 152 |
| 3.8.1 | Mandatory Fields | 152 |
| 3.8.2 | Protection Against Write Access | 153 |
| 3.8.3 | Combination: Mandatory Field for Creation and Write Protection for Updates | 154 |
| 3.9 | Field Mappings | 155 |
| 3.10 | Standard Operations for a CDS Entity | 157 |
| 3.10.1 | Create, Read, Update, and Delete | 157 |
| 3.10.2 | Create and Read Operations by Association | 158 |
| 3.11 | Specific Operations for a CDS Entity | 160 |
| 3.11.1 | Actions | 161 |
| 3.11.2 | Functions | 168 |
| 3.11.3 | Functions for Defaulting | 170 |
| 3.12 | Concurrency and Locking Behavior | 173 |
| 3.12.1 | Pessimistic Locking | 173 |
| 3.12.2 | Optimistic Locking | 175 |
| 3.13 | Internal Business Logic | 177 |
| 3.13.1 | Determinations | 177 |
| 3.13.2 | Validations | 182 |
| 3.13.3 | Calling Determinations or Validations via an Action | 185 |
| 3.14 | Authorization Checks | 187 |
| 3.14.1 | Authorization Master | 188 |
| 3.14.2 | Authorization-Dependent | 190 |
| 3.14.3 | Delegating Authorization Checks | 191 |

| | | |
|-------------|--------------------------------------------------------|-----|
| 3.15 | Draft Handling | 192 |
| 3.15.1 | Enabling Draft Handling | 193 |
| 3.15.2 | Draft Handling in the Business Object Composition Tree | 193 |
| 3.15.3 | Draft Lifecycle and Draft Actions | 195 |
| 3.15.4 | Side Effects | 197 |
| 3.16 | Events | 204 |
| 3.16.1 | Manually Triggered Events | 204 |
| 3.16.2 | Derived Events | 206 |
| 3.17 | Overarching Concepts | 207 |
| 3.17.1 | Dynamic Feature Control | 207 |
| 3.17.2 | Preliminary Checks of Operations | 211 |
| 3.17.3 | Internal Visibility of Operations | 212 |

4 Entity Manipulation Language: Accessing Business Logic 215

| | | |
|------------|----------------------------------------------------------|-----|
| 4.1 | Data Types | 216 |
| 4.1.1 | Derived Data Types | 216 |
| 4.1.2 | Implicit Return Parameters | 218 |
| 4.2 | EML Operations | 219 |
| 4.2.1 | READ ENTITIES | 220 |
| 4.2.2 | MODIFY ENTITIES | 222 |
| 4.2.3 | GET PERMISSIONS | 226 |
| 4.2.4 | SET LOCKS | 227 |
| 4.2.5 | COMMIT ENTITIES | 228 |
| 4.2.6 | ROLLBACK ENTITIES | 229 |
| 4.3 | Using the EML Outside of Behavior Implementations | 229 |
| 4.3.1 | Use in the Context of an ABAP Report | 230 |
| 4.3.2 | Implementation in the Context of a Test Class | 231 |
| 4.4 | Concrete Use Cases | 232 |

5 Behavior Implementation 239

| | | |
|------------|---------------------------------------------------------------------------|-----|
| 5.1 | Business Object Provider API | 239 |
| 5.2 | Runtime Behavior of the ABAP RESTful Application Programming Model | 240 |
| 5.2.1 | Interaction Phase and Transactional Buffer | 241 |
| 5.2.2 | Save Sequence | 242 |

| | | |
|------------|--------------------------------------------------------------------------|------------|
| 5.3 | Interfaces for the Interaction Handler and the Save Handler | 243 |
| 5.4 | Interaction Handler | 244 |
| 5.4.1 | FOR MODIFY | 245 |
| 5.4.2 | FOR INSTANCE AUTHORIZATION | 248 |
| 5.4.3 | FOR GLOBAL AUTHORIZATION | 250 |
| 5.4.4 | FOR FEATURES | 252 |
| 5.4.5 | FOR GLOBAL FEATURES | 255 |
| 5.4.6 | FOR LOCK | 256 |
| 5.4.7 | FOR READ | 258 |
| 5.4.8 | FOR READ by Association | 259 |
| 5.4.9 | FOR DETERMINE | 261 |
| 5.4.10 | FOR VALIDATE | 262 |
| 5.4.11 | FOR NUMBERING | 263 |
| 5.4.12 | FOR PRECHECK | 264 |
| 5.5 | Save Handler | 265 |
| 5.5.1 | FINALIZE | 266 |
| 5.5.2 | CHECK_BEFORE_SAVE | 268 |
| 5.5.3 | ADJUST_NUMBERS | 269 |
| 5.5.4 | SAVE | 270 |
| 5.5.5 | CLEANUP | 272 |
| 5.5.6 | CLEANUP_FINALIZE | 273 |
| 5.6 | Events | 274 |
| 5.6.1 | Triggering Events | 275 |
| 5.6.2 | Consumption of Events | 275 |
| 6 | Business Services | 281 |
| 6.1 | Projection Layer | 282 |
| 6.1.1 | CDS Projection View | 283 |
| 6.1.2 | Projection Behavior Definition | 284 |
| 6.2 | Service Definition | 285 |
| 6.3 | Service Binding | 285 |
| 6.4 | Testing Business Services in SAP Gateway Client | 289 |
| 6.5 | Testing UI Services Using the SAP Fiori Elements Preview | 292 |
| 6.6 | Business Object Interfaces | 293 |
| 6.6.1 | Structure of a Business Object Interface | 293 |
| 6.6.2 | Using Business Object Interfaces | 296 |
| 6.6.3 | Using Business Object Interfaces as New BAPIs | 296 |

7 Extensibility of Business Objects 301

| | |
|------------------------------------------------------|-----|
| 7.1 Introduction to the Extensibility Concept | 301 |
| 7.2 Extension Options | 305 |
| 7.2.1 Extensions to the Data Model | 305 |
| 7.2.2 Extensions to the Behavior | 312 |
| 7.2.3 Extensibility with Additional CDS Entities | 316 |
| 7.3 Extending a Standard Business Object | 320 |
| 7.3.1 Description of the Use Case | 320 |
| 7.3.2 Extending the Data Model | 323 |
| 7.3.3 Extending the Behavior | 332 |
| 7.3.4 Derived Events | 342 |

8 Application Interfaces and SAP Fiori Elements 347

| | |
|--------------------------------------------------------|-----|
| 8.1 Development Tools | 347 |
| 8.1.1 SAP Business Application Studio | 347 |
| 8.1.2 Visual Studio Code | 350 |
| 8.2 SAP Fiori Elements UIs for RAP Applications | 351 |
| 8.2.1 Floorplans in SAP Fiori Elements | 351 |
| 8.2.2 Selected UI Annotations | 353 |
| 8.2.3 Defining UI Annotations in a CDS View | 355 |
| 8.2.4 Generating Annotations via the Service Modeler | 370 |

Part II Practical Application Development

9 Use Cases 385

| | |
|----------------------------------------------------------------------------|-----|
| 9.1 Areas of Use for the ABAP RESTful Application Programming Model | 385 |
| 9.2 Implementation Types | 386 |
| 9.3 Decision Criteria for Selecting the Implementation Type | 387 |

10 Managed Scenario: Developing an Application with SAP Fiori Elements 391

| | |
|-----------------------------------------------------------------|-----|
| 10.1 Description of the Use Case | 392 |
| 10.2 Building the Data Model | 392 |
| 10.2.1 Database Tables | 392 |
| 10.2.2 CDS Modeling | 396 |
| 10.3 Creating Behavior Definitions | 405 |
| 10.3.1 Creating Behavior Definitions for Certificate Management | 405 |
| 10.3.2 Enabling Draft Handling | 410 |
| 10.4 Defining a Business Service | 411 |
| 10.4.1 Creating a Service Definition | 412 |
| 10.4.2 Creating the Service Binding | 413 |
| 10.5 Creating an SAP Fiori Elements User Interface | 416 |
| 10.6 Enrichment with a Determination | 424 |
| 10.7 Enrichment with a Validation | 428 |
| 10.8 Enrichment with an Action | 432 |
| 10.9 Generation and Deployment of the App | 434 |
| 10.10 File Upload | 442 |

11 Managed Scenario with Unmanaged Save: Integrating an Existing Application 445

| | |
|-------------------------------------------------------------|-----|
| 11.1 Description of the Use Case | 446 |
| 11.2 Building the Data Model | 449 |
| 11.2.1 Overview of the Logical Data Model | 450 |
| 11.2.2 Database Tables | 452 |
| 11.2.3 CDS Modeling | 455 |
| 11.3 Creating a Behavior Definition | 460 |
| 11.4 Implementing the Create Purchase Order Function | 463 |
| 11.4.1 Declaring Managed Numbering | 463 |
| 11.4.2 Setting Field Properties | 463 |
| 11.4.3 Creating the Behavior Pool | 465 |
| 11.4.4 Implementing Determinations | 466 |

| | | |
|-------------|------------------------------------------------------------------------------|------------|
| 11.4.5 | Save Sequence: Implementing Creation via the Business Object Interface | 472 |
| 11.4.6 | Implementing Validations | 481 |
| 11.5 | Implementing the Delete Purchase Order Function | 487 |
| 11.5.1 | Save Sequence: Implementing Deletion via the Business Object Interface | 487 |
| 11.5.2 | Implementing a Validation | 492 |
| 11.6 | Defining Business Services | 492 |
| 11.6.1 | Setting Up the Projection Layer for the My Purchase Orders App | 492 |
| 11.6.2 | Creating a Service Definition | 494 |
| 11.6.3 | Creating a Service Binding | 495 |
| 11.7 | Implementing Authorization Checks | 495 |
| 11.7.1 | Access Controls for Read Access | 496 |
| 11.7.2 | Access Controls for Write Access | 497 |
| 11.8 | Creating an SAP Fiori Elements User Interface | 500 |
| 11.8.1 | Creating a Metadata Extension | 500 |
| 11.8.2 | Generating and Deploying the Application | 503 |

12 Unmanaged Scenario: Reusing Existing Source Code 505

| | | |
|-------------|------------------------------------------------------|------------|
| 12.1 | Description of the Use Case | 505 |
| 12.2 | Description of the Existing Application | 507 |
| 12.2.1 | Database Tables | 507 |
| 12.2.2 | Source Code of the Existing Application | 510 |
| 12.3 | Extending the Data Model | 513 |
| 12.4 | Creating a Behavior Definition | 519 |
| 12.5 | Creating a Behavior Implementation | 523 |
| 12.5.1 | Implementing the Interaction Phase | 525 |
| 12.5.2 | Implementing the Save Sequence | 533 |
| 12.6 | Defining a Business Service | 537 |

| | |
|-------------------------------------------------------------------|---------|
| 13 Special Features in the Cloud Environment | 541 |
| 13.1 Basic Technical Principles | 542 |
| 13.1.1 ABAP for Cloud Development | 545 |
| 13.1.2 Technical Infrastructure Components | 545 |
| 13.1.3 Migrating Legacy Code | 547 |
| 13.2 Identity and Access Management | 548 |
| 13.3 Deploying SAP Fiori Apps and Assigning Authorizations | 551 |
| 13.3.1 Creating an IAM App and Business Catalog | 552 |
| 13.3.2 Creating an IAM Business Role | 554 |
| 13.3.3 Integration in SAP Fiori Launchpad | 554 |
| 13.4 Consuming Business Services | 558 |
| Bibliography | 563 |
| The Authors | 565 |
| Index | 567 |

Chapter 3

Behavior Definition

In this chapter, you'll learn how to add a behavior definition to a data model defined with CDS and how to enrich it with the desired transactional behavior and business logic using the behavior definition language.

In the previous chapter, you learned how to model the structure of a business object in core data services (CDS). For read-only operations (queries) on a business object, this modeling is already sufficient. But what happens if you want to add write operations or business logic, that is, behavior, to this business object and its child entities? This chapter describes how you can create a behavior definition for that purpose.

After a short introduction in Section 3.1, we'll show you how to create behavior definitions in ABAP Development Tools (ADT) in Section 3.2. In Section 3.3, Section 3.4, and Section 3.5, we'll go into more detail about some general concepts of behavior definition, such as the definition of the different implementation types, strict mode, and entity behavior definitions. In Section 3.6, you'll learn how to define a behavior pool, which you'll then flesh out later in the behavior implementation.

The subsequent sections deal in greater detail with concrete behavior that you can define. Section 3.7 explains the method of numbering assignment. In Section 3.8, you'll learn about field properties for the CDS data model. Section 3.9 is about field mappings, Section 3.10 describes standard operations, and Section 3.11 deals with specific operations. Section 3.12 covers concurrent access and locking behavior, Section 3.13 describes the definition of internal business logic, Section 3.14 covers authorization checks, and Section 3.15 describes draft handling. Section 3.16 describes the events of a RAP business object. Section 3.17 concludes this chapter with some overarching concepts of behavior definition.

3.1 What Is a Behavior Definition?

The *CDS behavior definition* (hereafter referred to as *behavior definition*) is the central development object for declaring the behavior of a business object via the behavior definition language (BDL). When you add a behavior definition to a CDS root entity, you're defining a RAP business object.

3.1.1 Context and Structure of a Behavior Definition

In Figure 3.1, you can see the artifacts in which a behavior definition is embedded. A RAP business object consists of a CDS data model and a behavior. Using `define root view`, you can create a behavior definition for a CDS root entity so you can add transactional behavior. There can only be *one* such definition for a CDS root entity. No behavior definition can be created for other CDS entities that aren't root.

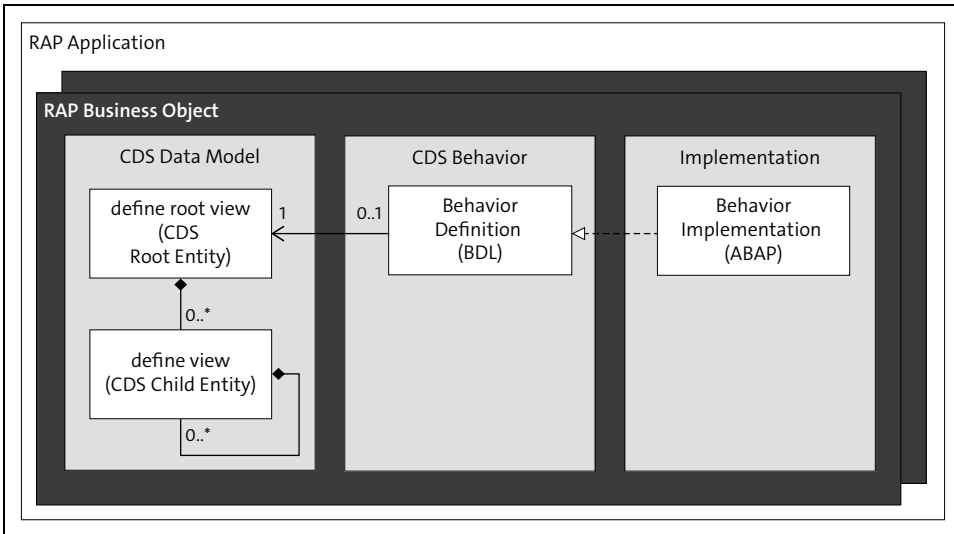


Figure 3.1 Artifacts for the Definition of a RAP Business Object

The behavior declared via BDL requires a suitable behavior implementation in ABAP in any case. Either the ABAP RESTful application programming model provides the corresponding behavior (e.g., a standard operation) or the respective application implements it. Depending on the application, mixed forms are also frequently encountered. For more information on this, see Section 3.3.

For its part, a RAP application can consist of more than one RAP business object if the data model of the application provides for this and the business mapping requires it. Different RAP business objects may also refer to each other or invoke an appropriately exposed behavior (e.g., an update operation or an action).

As an example, let's take a look at the behavior definition of the `Travel` business object from the ABAP Flight Reference Scenario. It represents a trip that consists of several bookings (CDS child entity `Booking`). The `Travel` business object is represented by the CDS root entity `/DMO/R_Travel_D`. Because there can only be one behavior definition for a CDS root entity, its name is also simply `/DMO/R_Travel_D`. In ADT, this behavior definition can be found in **Project Explorer** under **Core Data Services • Behavior Definitions** (see Figure 3.2).

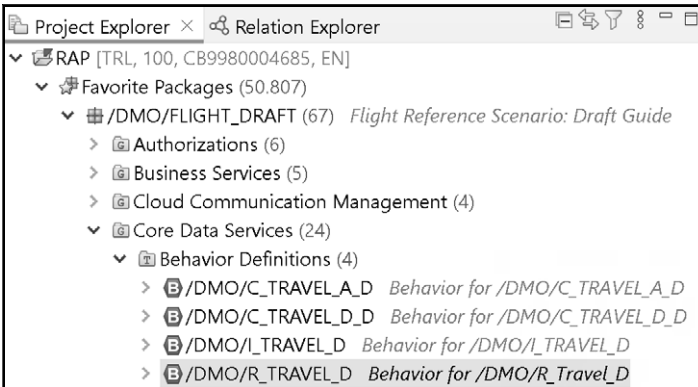


Figure 3.2 Behavior Definition “/DMO/R_TRAVEL_D” in Project Explorer

Note: Projection Behavior Definition

The development objects /DMO/C_TRAVEL_A_D, /DMO/C_TRAVEL_D_D and /DMO/R_TRAVEL_D shown in Figure 3.2 are *projection behavior definitions*, which represent a special type of behavior definition. For more information, see Chapter 6, Section 6.1.2. The development object /DMO/I_TRAVEL_D is a business object interface. You can find out more about business object interfaces in Chapter 6, Section 6.6.

Double-clicking on the behavior definition /DMO/R_Travel_D takes you to the ADT editor, where the BDL source code is displayed (see Figure 3.3).

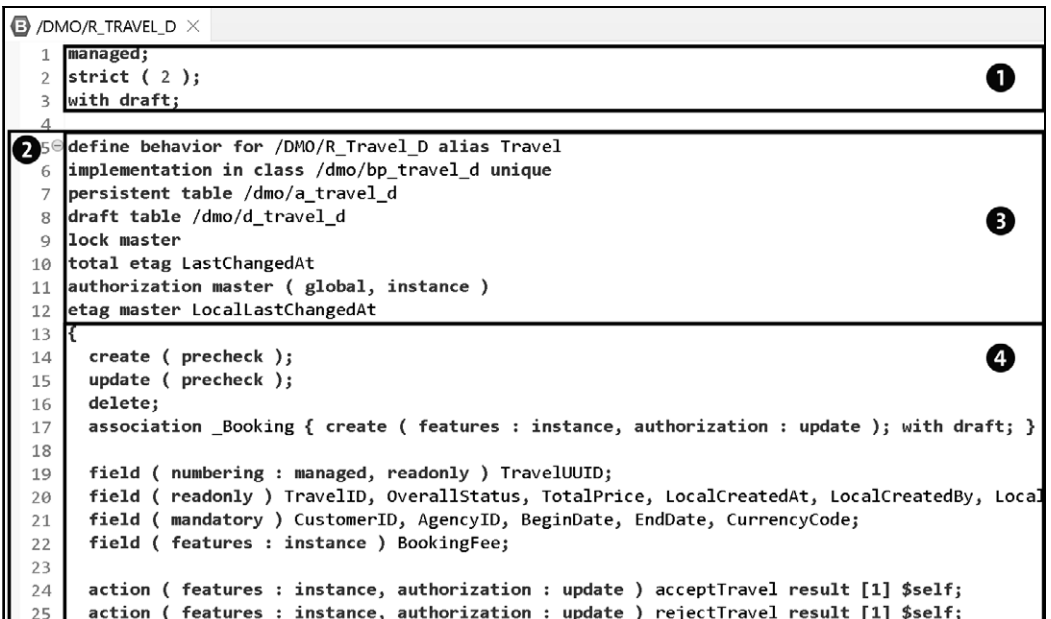


Figure 3.3 BDL Source Code of Behavior Definition “/DMO/R_Travel_D”

A behavior definition is divided into four areas:

- ❶ Header of the behavior definition
- ❷ Area of the entity behavior definition
- ❸ Header of the entity behavior definition with transactional properties
- ❹ Body of the entity behavior definition

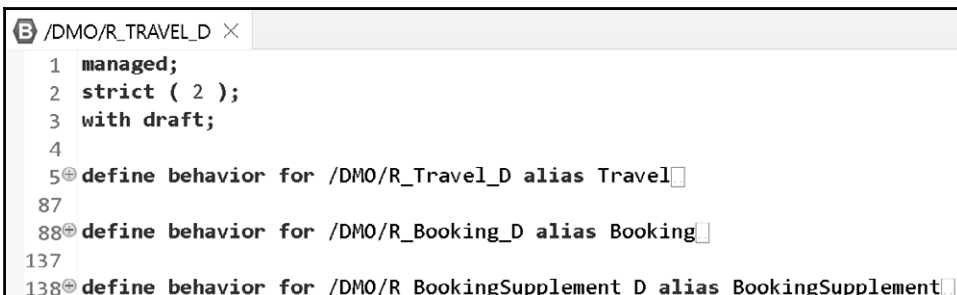
The header of the behavior definition starts with the choice of the implementation type. In this example, `managed` declares that the managed scenario should be used for the RAP business object. This area also defines properties that apply to the entire composition tree of the business object, such as draft handling (`with draft`).

For each CDS entity of the composition tree, an *entity behavior definition* can now follow. Thus, there may be one or more entity behavior definitions within a behavior definition. An entity behavior definition consists of a head and a body.

The entity behavior definition is introduced with the `define behavior for` statement and forms the beginning of the head section. Then, you can define a behavior pool via `implementation in class` (this is also possible at the behavior definition level; see Section 3.6). You can also declare other transactional properties of the CDS entity. For example, in the managed scenario, the database table for the CDS entity is set via `persistent table`.

The head section is followed by the body of the entity behavior definition. This is enclosed in curly brackets `{ ... }` and contains additional keywords that specify the behavior of the CDS entity. For example, the BDL statements `create`, `update`, and `delete` specify that this CDS entity can be created, updated, and deleted. With `action`, you can declare an action (i.e., a specific operation, on the CDS entity), which you can implement later in the behavior pool (here: `action ... acceptTravel` or `action ... rejectTravel`).

In Figure 3.3, due to space constraints, we've shown only one entity behavior definition of the `Travel` business object, namely the mandatory definition for the CDS root entity. However, the `Travel` business object consists of the additional CDS child entities `/DMO/R_Booking_D` and `/DMO/R_BookingSupplement_D`, which in turn have their own entity behavior definition, and, together with the CDS root entity, they form the composition tree of the business object (see Figure 3.4).



```

❷ /DMO/R_TRAVEL_D ×
1  managed;
2  strict ( 2 );
3  with draft;
4
5  define behavior for /DMO/R_Travel_D alias Travel
87
88  define behavior for /DMO/R_Booking_D alias Booking
137
138  define behavior for /DMO/R_BookingSupplement_D alias BookingSupplement

```

Figure 3.4 Additional Entity Behavior Definitions of the Travel Business Object

The relationship between a business object's CDS entities, that business object's behavior definition, and the entity behavior definitions it contains is illustrated in Figure 3.5.

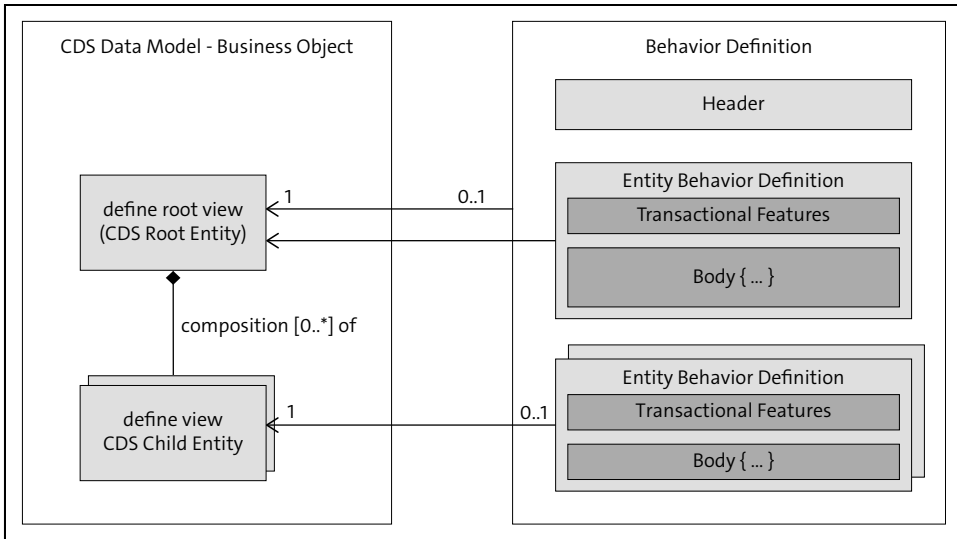


Figure 3.5 Relationship Between Behavior Definition and CDS Entities

3.1.2 Syntax of a Behavior Definition

In Listing 3.1, you can see the behavior definition of the `Travel` business object. We've shortened it considerably to illustrate the general BDL syntax with this example.

```
managed;
strict ( 2 );
with draft;

define behavior for /DMO/R_Travel_D alias Travel
implementation in class /dmo/bp_travel_d unique
persistent table /dmo/a_travel_d
...
{
    create ( precheck );
    update ( precheck );
    delete;

    association _Booking { create; }
    validation validateCustomer on save { create; field CustomerID; }
}
```

Listing 3.1 BDL Syntax Based on the Behavior Definition for the “/DMO/R_Travel_D” Object

The general syntax of the BDL is based on the CDS syntax. The following rules apply:

- **Keywords**

BDL keywords, such as `managed` or `define behavior for`, are written in lowercase. A mixture of lowercase and uppercase letters isn't allowed.

- **Ending of statements**

Statements usually end with a semicolon (;). This isn't the case for statements in the context of the entity behavior definition and its transactional properties.

- **Names or identifiers**

Names or identifiers, such as the name of the validation `validateCustomer`, aren't case sensitive. Here, the name of the validation is given in camel case notation for better readability. Because of this rule, another validation called `ValidateCustomer` would result in a syntax error because it has already been declared. Numbers and underscores in the name are allowed.

- **Comments**

A line comment is introduced by a double slash `//`. This can be placed anywhere within a line. All characters following the double slash are treated as comments. A multiline comment is introduced by the string `/*` and ends with `*/`. Text that's between these characters is treated as a comment.

3.1.3 Possible Behavior

This section provides an overview of the possible behaviors you can equip a RAP business object with. In the ABAP RESTful application programming model, the behavior is also summarized under the term *features*, which means the functions of a business object. In addition, in this section, we'll look at the effects of declaring behavior using the BDL in the behavior definition. This way, you can assess what behavior is available to you when designing a business object. Secondly, you'll see the impact BDL statements have, not only in initial design but also when you want to make changes to the behavior definition and when there are already business object consumers.

Overview of Features

We assign the behavior of a RAP business object to the following categories:

- Transactional behavior and properties
- Standard operations
- Specific operations
- Internal business logic
- Fields
- Events
- UI-related behavior
- Overarching elements

If a business object supports write operations, other aspects become technically relevant, such as the persistence of the business data or the handling of concurrent accesses in the event of changes. The ABAP RESTful application programming model summarizes these aspects via the terms *transactional behavior* and *properties*. In the context of application development, you might think about how authorization checks (authorization), lock behavior (lock, etag, etag total), numbering assignment (numbering), or persistence of data (persistent table) are implemented. Furthermore, you can activate draft handling (with draft) and thereby use the draft mode built into the ABAP RESTful application programming model for a business object (see Table 3.1).

| Behavior | BDL Keyword | Properties |
|----------------------------------------------------------------------|------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Authorization checks | authorization master authorization dependent by ... | Behavior implementation required |
| Lock behavior (pessimistic method) | lock master lock dependent by ... | Managed by the RAP framework; behavior implementation required in the unmanaged scenario |
| Lock behavior (optimistic method) | etag master etag dependent ... total etag | Managed by the RAP framework |
| Numbering assignment | early numbering late numbering | Behavior implementation required |
| Numbering assignment (universally unique identifier [UUID]) | field k1(numbering:managed) | Managed by the RAP framework |
| Persistence | persistent table ... | Managed by the RAP framework |
| Draft handling | with draft; draft table ... draft action ...; draft determine action ...; | Managed by the RAP framework; behavior implementation for draft action possible |

Table 3.1 Overview of Transactional Behavior and Properties

Standard operations are operations that create, update, or delete instances of a CDS entity. These operations make changes to the CDS entities of the business object and thus count as write operations. Standard operations also include reading the business object's CDS entities and locking, in the context of a pessimistic locking procedure (see Table 3.2).

| Behavior | BDL Keyword | Properties |
|----------------------------|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Create Change Delete | create; update; delete; | Business object externally visible, callable |
| Read Locks | read; lock; | Business object externally visible, callable Explicit specification of read and lock only for implementation groups and the unmanaged scenario |
| Creation via association | association_a1 { create; } | Business object externally visible, callable |
| Reading via association | association_a1 { } | Business object externally visible, callable |

Table 3.2 Overview of Standard Operations

Specific operations on a RAP business object are operations that go beyond the general CRUD operations and have a functional reference to the business object. These include actions (action) and functions (function, see Table 3.3).

| Behavior | BDL Keyword | Properties |
|----------|---------------------------------------------|------------------------------------------------------------------------------|
| Action | action a1 ...; static action a2; | Business object externally visible, callable; write operation |
| Function | function f1 ...; static function f2 ...; | Business object externally visible, callable; operation without side effects |

Table 3.3 Overview of Specific Operations



Note: Two Meanings of “Side Effect”

We use the term *side effect* in two different contexts in this book. In the context of the operations of a RAP business object, it refers to the effects the operation has on the data of a business object instance (e.g., see Table 3.3). In this sense, an operation is without side effects if it does not or may not perform any changes (like a function). However, it does have side effects when it creates, changes, or deletes a business object instance.

Side effect also refers to a very specific feature that is defined using the *side effects* keyword within the behavior definition. In the narrower sense, you must use this keyword to formally define a side effect and thus express that a previously made change to a business object instance results in the reloading of data via the UI (through an

operation or determination that has a side effect in the broader sense). We take a closer look at this feature in Section 3.15.4.

Internal business logic refers to all functionally motivated program logic within business objects that's used to implement business processes. While this is very broad and also applies to actions, for example, we explicitly refer to internally visible business logic here, that is, logic that isn't visible and consumable from outside the business object. The ABAP RESTful application programming model provides validation and determination for this purpose, as well as the `determine` action, which makes both callable together (see Table 3.4).

| Behavior | BDL Keyword | Properties |
|----------------------|--------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Determination | <code>determination d1 ... on modify;</code> <code>determination d2 ... on save;</code> | Business object internal; calculation during interaction phase or save sequence, with side effects |
| Validation | <code>validation v1 ... on save;</code> | Business object internal; data check of the transactional buffer without side effects |
| Determination action | <code>determine action da1 {...}</code> | Business object external; determinations or validations declared as externally callable |

Table 3.4 Overview of Internal Business Logic

The *fields* of a business object are determined by the fields of the respective CDS entity. In the context of the behavior definition and the standard write operations already described, you can decide which fields are mandatory (`field(mandatory:create)`) or which fields can't be changed (`field(readonly)`). You can also define a field mapping (`mapping`) and thus make field mappings between structured ABAP Dictionary objects and a CDS entity and then use them in the behavior implementation (see Table 3.5).

| Behavior | BDL Keyword | Properties |
|------------------|-------------------------------------|-----------------------------------------------------------------------------------------|
| Field properties | <code>field(...) f1, f2, fn;</code> | Business object external; declare fields as mandatory or protect against write accesses |
| Field mappings | <code>mapping ... {...}</code> | Business object internal; use field mappings in the behavior pool |

Table 3.5 Overview of Fields

Events of a CDS entity of a RAP business object enable event-based, asynchronous communication with event consumers. In this way, the trigger of the event and its consumer are only very loosely coupled. A consumer can be the local system or a remote system. A CDS entity can declare an event without parameters (*event*) or with parameters (*event ... parameter ...*). Furthermore, derived events (*managed events*) can be declared on the basis of existing events (see Table 3.6).

| Behavior | BDL Keyword | Properties |
|-------------------------------------------|---------------------------------------------------|----------------------------------------------------------------|
| Event (manually triggered) | <code>event e1;</code> | External to business objects; behavior implementation required |
| Event (manually triggered with parameter) | <code>event e2 parameter p1;</code> | External to business objects; behavior implementation required |
| Derived event | <code>managed event e3 on e2 parameter p2;</code> | External to business objects; managed by RAP framework |

Table 3.6 Overview of Events

UI-related behavior is part of the behavior definition. A UI can be based on its declaration and integrate the corresponding behavior. UI-related behavior is defined in the behavior definition independently of the specific UI technology. Therefore, the specific UI technology is tasked with integrating the defined features in a suitable way. The features for UI-related behavior include functions for field preallocation (default function) and side effects with which the UI can reload data (see Table 3.7).

| Behavior | BDL Keyword | Properties |
|----------------------------------|------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Function for field preallocation | <code>... default function GetDefaultsFor...;</code> | External to business objects, behavior implementation required; integration in SAP Fiori elements |
| Side effects | <code>side effects { ... }</code> | External to business objects; integration in SAP Fiori elements |

Table 3.7 Overview of UI-Related Features

In the last category, we have grouped BDL keywords that have an overarching character. That is, these keywords can't stand alone, but add certain properties to standard or specific operations. For example, you can mark certain elements as visible only internally (*internal*) or use a *dynamic feature control* to determine at runtime whether a field is protected against write access or not (*features*). Additionally, you can declare a preliminary check (*precheck*) for an operation performed on the RAP business object (see Table 3.8).

| Behavior | BDL Keyword | Properties |
|---------------------------------|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| Dynamic feature control | features:instance features:global | Dynamic determination of field properties, standard operations and actions; behavior implementation required |
| Preliminary check of operations | precheck | Check for feasibility of standard operations or actions before reaching the transactional buffer; behavior implementation required |
| Visibility | internal | BDL feature only visible internally |

Table 3.8 Overview of Cross-Sectional BDL Keywords

Effects of Behavior Declarations

If you declare behavior in the behavior definition, this has various effects on both business object–internal and business object–external artifacts of a RAP application (see Figure 3.6).

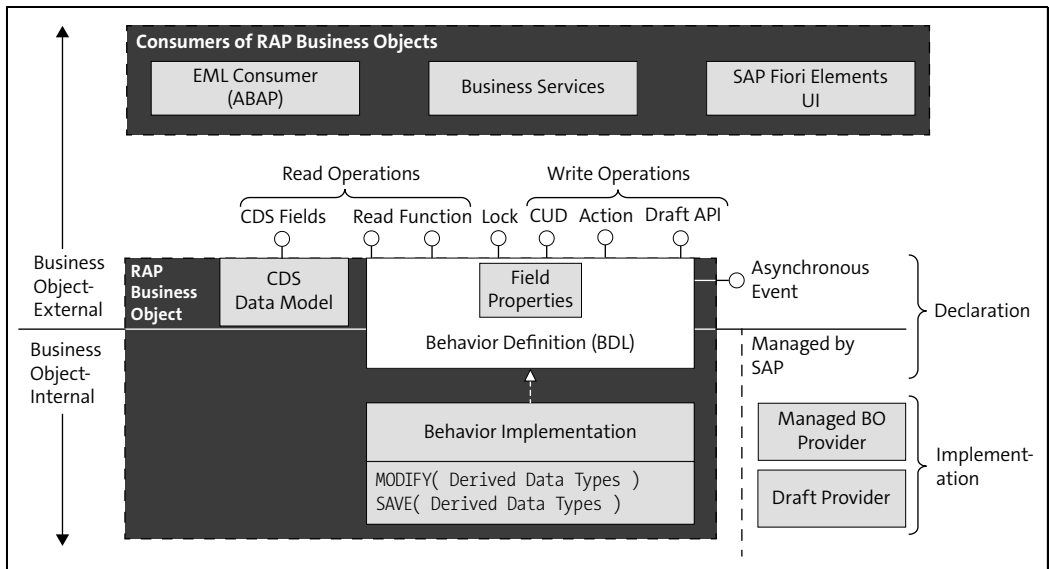


Figure 3.6 Features of the Behavior Definition and Surrounding Artifacts

The BDL keywords impact the following areas:

■ Business object interface

Using the features in the behavior definition, you can design the external interface

of the RAP business object, consisting of read and write operations. This can comprise the standard create, read, update, delete (CRUD) operations (and a possible locking operation in the context of pessimistic locking), actions, and functions. If a RAP business object supports only the create and update operations, but not the delete operation, you can declare only the corresponding standard operations `create` and `update`, but not `delete`. The business object interface also includes the fields of the respective CDS entities defined via the CDS data model and any field properties (`field`) specified in the behavior definition. Also included are events for which interested recipients can register.



Note: Differentiation from the Term Business Object Interface

We use the term *business object interface* here in a broader, logical sense and refer to the parts of a RAP business object that are visible to consumers. The ABAP RESTful application programming model also provides a separate development object called the business object interface, which you can use to explicitly model the public interface of a RAP business object. We'll discuss the business object interface development object in Chapter 6, Section 6.6.

■ Provision of the behavior implementation

The behavior definition also allows you to influence the way in which the RAP business object is implemented, and thus how the operations provided in the business object interface will come into effect. This concerns, for example, the basic choice of the implementation type. For example, you can use the managed business object provider (`managed`) or the draft handler of the RAP framework (`with draft`).

When you declare behavior via the BDL that requires a method implementation (e.g., a validation or action), you enable the corresponding method in the behavior implementation and can implement it. The behavior definition can thus be viewed as an interface implemented by one or more classes. In addition, BDL behavior also affects the derived data types in the context of behavior implementation and entity manipulation language (EML) consumers.

■ Business services

The business object interface has external characteristics, but initially can only be called locally via the EML. The RAP business object with its remote interface, which was first exposed to OData via a business service, is then found there in the metadata of the OData service on the one hand, and in the concretely usable operations of the OData service on the other.

So, for example, if you've exposed a web API based on a RAP business object for integration purposes, then it's important to keep this external interface compatible and know that the behavior definition or artifacts of the projection layer can have a direct impact on it, like when a field that was previously modifiable is now protected against write access.

■ SAP Fiori elements UI

Even though the behavior definition in isolation has no direct impact on an SAP Fiori elements UI, the default operations, for example, are still exposed via the OData metadata and represented as buttons in an SAP Fiori elements UI. Draft handling also affects the SAP Fiori elements UI via the declaration in the behavior definition and represents UI elements that implement draft handling.

Tip: Lean Interface for a RAP Business Object

Select only the behavior that is useful and necessary for the functionality of the business object because all features are components of the business object interface (in the broader sense). If a standard operation can only be used internally in the business object (e.g., the `create` or `update` operation), you can use the additional `internal` operation (Section 3.17.3).



3.2 Editing a Behavior Definition in ABAP Development Tools

In this section, we'll show you how to use ADT to create, update, activate, and search for a behavior definition. The behavior definition is an independent development object in the ABAP Repository. It's assigned to a package and connected to the transport system. It can be maintained exclusively via ADT.

3.2.1 Creating a Behavior Definition

In Figure 3.2 shown earlier, you've already seen how a behavior definition is displayed in ADT's **Project Explorer**. There are now two ways you can create a behavior definition in ADT:

■ By direct reference to the CDS root entity

You must use the context menu to select the CDS root entity for which you want to create the behavior definition.

■ By using the creation wizard for the behavior definition development object type

Here, you manually select the CDS root entity during the creation process.

If you want to create a behavior definition via the context menu, proceed as follows:

1. In the **Project Explorer**, select the CDS root entity for which you want to create the behavior definition.
2. Right-click on the CDS root entity to open the context menu, and select the **New Behavior Definition** entry (see Figure 3.7).

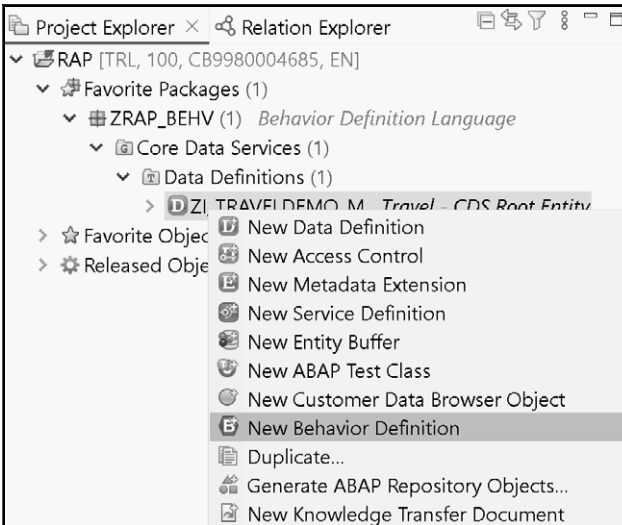


Figure 3.7 Creating a Behavior Definition with Reference to the CDS Root Entity

3. The dialog box shown in Figure 3.8 opens. Here, the **Name** and **Root Entity** fields are already predefined by the CDS root entity selection (here, ZI_TRAVELDEMO_M).
4. Enter an appropriate description for your behavior definition under **Description**, and select the desired implementation type under **Implementation Type**. For our example, we choose **Managed**. Confirm your entries by clicking **Next**.

 The screenshot shows the 'New Behavior Definition' dialog box. The title bar says 'New Behavior Definition'. Below the title bar, there is a 'Behavior Definition' section with a 'Create Behavior Definition' button. The dialog contains the following fields and controls:

- Project: * (text field with 'RAP', 'Browse...' button)
- Package: * (text field with 'ZRAP_BEHV', 'Browse...' button)
- ☐ Add to favorite packages
- Name: (text field with 'ZI_TRAVELDEMO_M')
- Description: * (text field with 'Travel - CDS Root Entity')
- Original Language: (text field with 'EN')
- Root Entity: * (text field with 'ZI_TRAVELDEMO_M')
- Implementation Type: * (dropdown menu with 'Managed' selected)
- At the bottom, there is a help icon (?), a '< Back' button, a 'Next >' button (highlighted), a 'Finish' button, and a 'Cancel' button.

Figure 3.8 Dialog Box for Creating a Behavior Definition

- Then, create a new transport request, or select an existing one. Confirm your entry by pressing **Enter**.

The **Project Explorer** display will refresh so you'll see the newly created behavior definition there (see Figure 3.9).

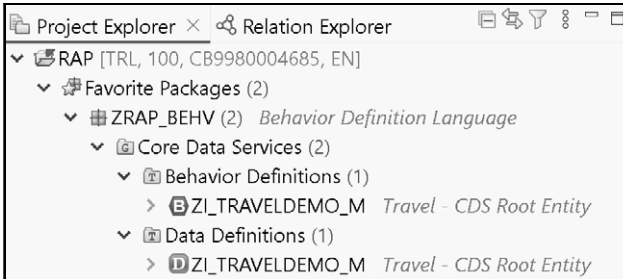


Figure 3.9 The New Behavior Definition “ZI_TRAVELDEMO_M” in Project Explorer


After the creation, the editor for the BDL source code of the behavior definition ZI_TRAVELDEMO_M opens (see Figure 3.10). The coding is already suggested here based on the selected implementation type. For example, you'll find the managed keyword in the behavior definition head, the entity behavior definition for the selected CDS root entity ZI_TRAVELDEMO_M, and the standard operations create, update, and delete. Furthermore, the alias addition (`//alias <alias_name>`) is suggested in the entity behavior definition when initially created. We recommend directly assigning a suitable, descriptive alias name.

```

1 managed implementation in class zbp_i_traveldemo_m unique;
2 strict ( 2 );
3
4 define behavior for ZI_TravelDemo_M //alias <alias_name>
5 persistent table /dmo/travel_m
6 lock master
7 authorization master ( instance )
8 //etag master <field_name>
9 {
10 create;
11 update;
12 delete;
13 field ( readonly ) TravelId;
14 }

```

Figure 3.10 BDL Source Code Editor for the New Behavior Definition “ZI_TRAVELDEMO_M”

The gray diamond icon on the  icon of the editor tab indicates that the behavior definition is still in an inactive state.

You can also select the creation via the wizard for the object type **Behavior Definition**. Here, proceed as follows:

1. Open the creation wizard via the menu path **File • New • Other** or using the keyboard shortcut **[Alt] + [N]**.
2. Select the **ABAP • Business Services** node, and then choose the **Behavior Definition** development object type (see Figure 3.11). Confirm your selection by clicking **Next**.
3. The already familiar dialog window for creating the behavior definition opens (refer back to Figure 3.8). Click the **Browse** button next to the **Root Entity** input field (which is not shown in our example, but will appear when there is no given reference to a CDS root entity), and select the CDS root entity for which you want to create the behavior definition. Confirm your entries by clicking **Next**.

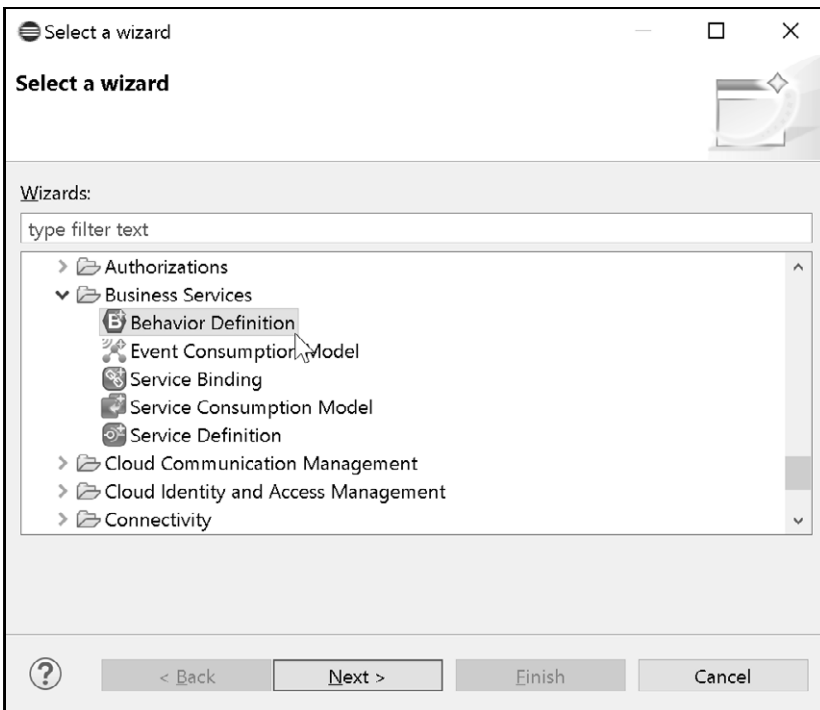


Figure 3.11 Selecting the Behavior Definition via the Creation Wizard



Note: Behavior Definition Name Can't Be Changed

The behavior definition name always corresponds to the CDS root entity name because there can only be one behavior definition for a CDS root entity. For this reason, you can't change the name.

4. Then, create a new transport request, or select an existing one. Confirm your entry with **[Enter]**.

Next, let's look at how you can change and activate a behavior definition.

3.2.2 Changing and Activating a Behavior Definition

Follow these steps if you want to change a behavior definition:

1. Double-click the behavior definition in the **Project Explorer**.

The editor for the behavior definition with the BDL source code opens.

2. Update the BDL source code. In addition to manual entries, you can also use the auto-complete function (see Figure 3.12). To do this, position the cursor where you want to add code, for example, in the body of the entity behavior definition, and press the shortcut `[Ctrl] + [Space]`.

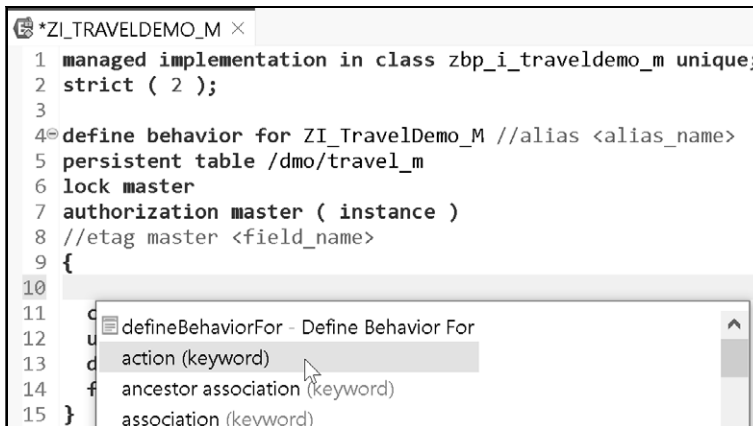




Figure 3.12 Using Auto-Completion in the BDL Editor

3. Choose a suitable keyword, and confirm with `[Enter]`. Make necessary further additions so that the syntax of the keyword is correct.
4. Save the BDL source code via the **Save** button  or the keyboard shortcut `[Ctrl] + [S]`.

You can activate the behavior definition via the **Activate** button  or the keyboard shortcut `[Ctrl] + [F3]`.

As is the case with many other development objects, you can view details about your behavior definition in the **Properties** view. The activation status (here, **Inactive**) can be seen next to **Version** (see Figure 3.13).

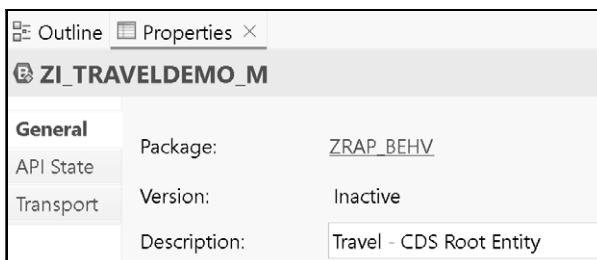



Figure 3.13 Activation State of the Behavior Definition

3.2.3 Finding and Opening a Behavior Definition

Proceed as follows if you want to search for a behavior definition that has already been created:

1. Open the dialog for searching development objects via the **Open**  icon for opening development objects in the toolbar or via the shortcut **Ctrl** + **Shift** + **A**. We recommend using the shortcut. (The general use of shortcuts can speed up the handling of ADT. There are corresponding shortcuts available for many development tasks.)
2. In the search box, enter “type:”, and then press the **Ctrl** + **Space** keys. Select the development object type **BDEF (Behavior Definition)** to restrict the search to this type (see Figure 3.14).

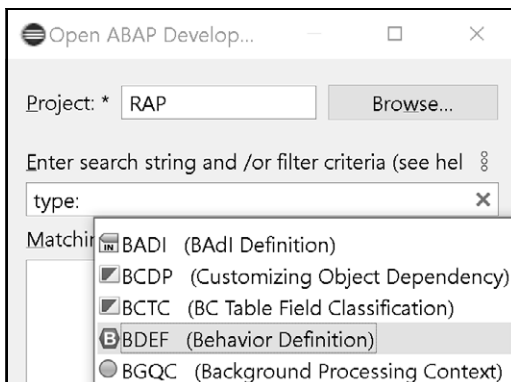


Figure 3.14 Restricting a Search Run to Behavior Definitions

3. Then, enter the name of the behavior definition you want to find in the search field. You can use wildcards (e.g., **z*tra*demo**).
4. The results list for the entered search query appears (see Figure 3.15). Double-click the relevant entry to open the behavior definition.

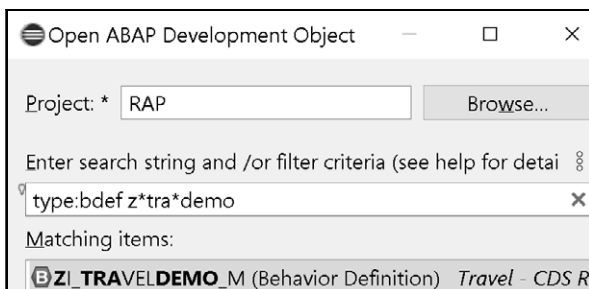


Figure 3.15 Searching for a Behavior Definition by Wildcard with Result List

3.2.4 Documenting Behavior Definitions and Relationships

You can document various development objects via *Knowledge Transfer Documents* (KTDs), which also includes the behavior definition. With KTDs, you document various elements of the behavior definition, such as the `create` operation or an action, using *Markdown syntax*, a simplified markup language.

Note: Knowledge Transfer Documents

Until now, many development object types have been documented using SAPScript-based technologies. The KTD is a relatively new option for documenting ABAP development objects. The documentation is directly linked to the respective development object. SAP's focus when providing new KTDs is on the object types of the ABAP RESTful application programming model. You also have a new, more elegant option for documenting your own RAP development objects. The KTD is available from SAP Business Technology Platform (SAP BTP), ABAP environment 2008, or from ABAP platform 7.55.

To create a KTD, you right-click to open the context menu of your behavior definition and select **New Knowledge Transfer Document** (see Figure 3.16).

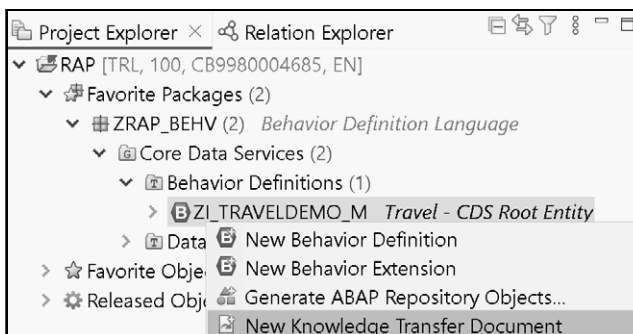


Figure 3.16 Creating a KTD for a Behavior Definition

You can maintain Markdown-based documentation in the **Documentation** section on the **Source** tab. In the **Object Structure** section, you must select the behavior definition element you want to document (here, the `create` operation). To display a preview of the documentation, select the **Output** tab (see Figure 3.17).

If a KTD exists, the behavior definition editor displays a corresponding **Open Documentation** option (see Figure 3.18).

Note: Documenting Other RAP Artifacts

You can also document a CDS entity or a projection behavior definition with a KTD.

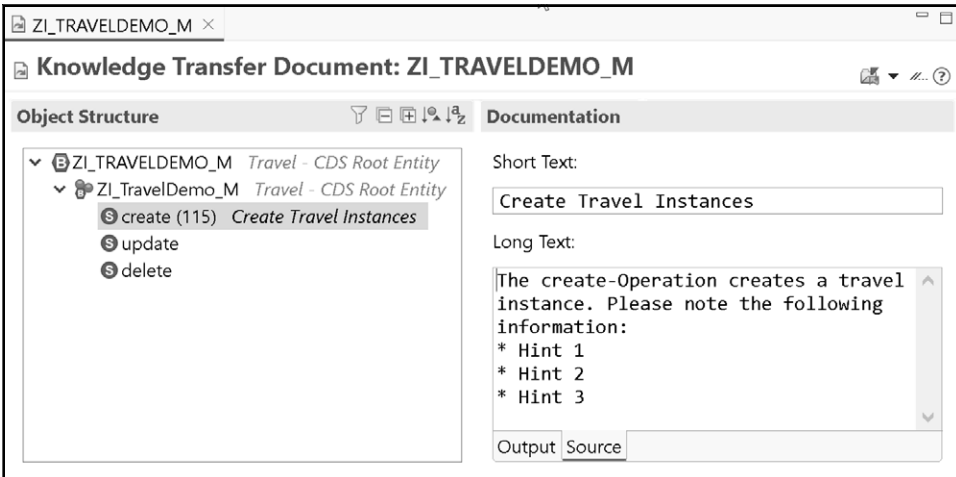


Figure 3.17 Documenting the Create Operation in Markdown Syntax

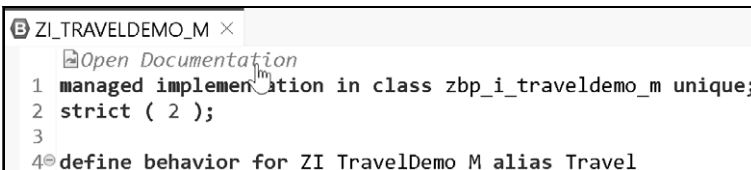


Figure 3.18 Jump to Documentation in the Behavior Definition

The **Relation Explorer** is a separate view in ADT that allows you to view contextual information about a business object (or other development object). You can choose whether you want to see the development object in question in the overview, in the context of its users or in the context of the development objects used. You can open the **Relation Explorer** view (see Figure 3.19) in the editor of the behavior definition via the context menu path **Show In • Relation Explorer**.

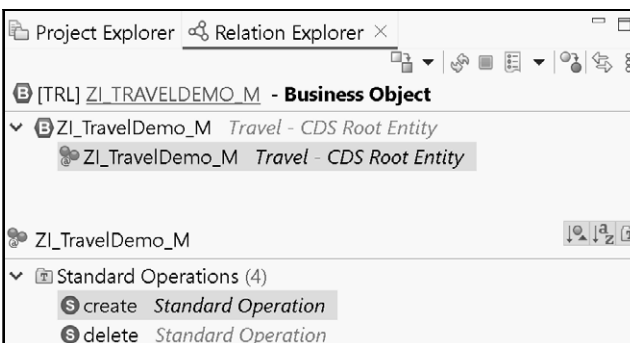


Figure 3.19 Business Object “ZI_TRAVELDEMO_M” in the Relation Explorer View

The documentation created for an element in the previous section can also be viewed in the **Relation Explorer** (see Figure 3.20). To do this, select the **Show Element Information** entry in the context menu or press the **[F2]** function key.

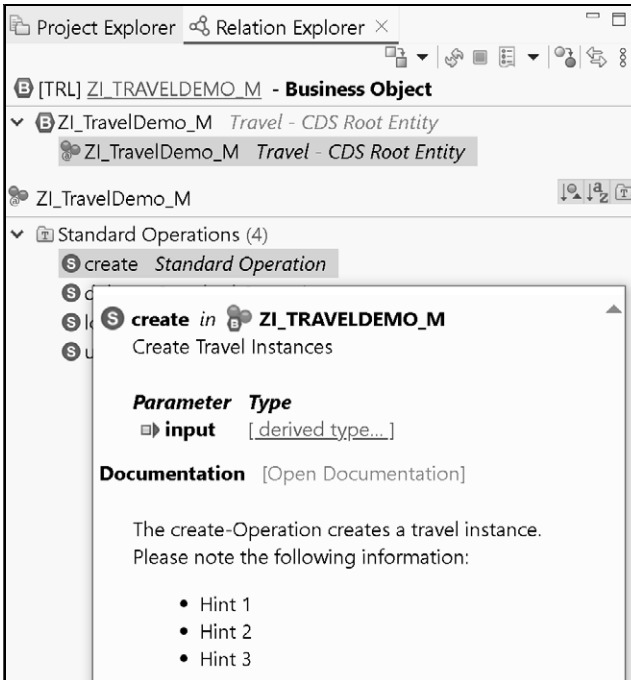


Figure 3.20 Documentation for the Create Operation in the Relation Explorer View

3.3 Implementation Types

The implementation type is your first specification in the behavior definition. You use it to specify the source of the RAP business object implementation. There are two sources (see Figure 3.21):

■ Managed

SAP provides the implementation through the RAP framework. The implementation can be done by a business object provider, the draft handling provider, or both in combination.

■ Unmanaged

The RAP business object itself provides the implementation via the behavior pool (unmanaged business object provider).

The implementation type applies to all CDS entities of the business object composition tree. It isn't possible to declare one CDS entity as managed and another as unmanaged.

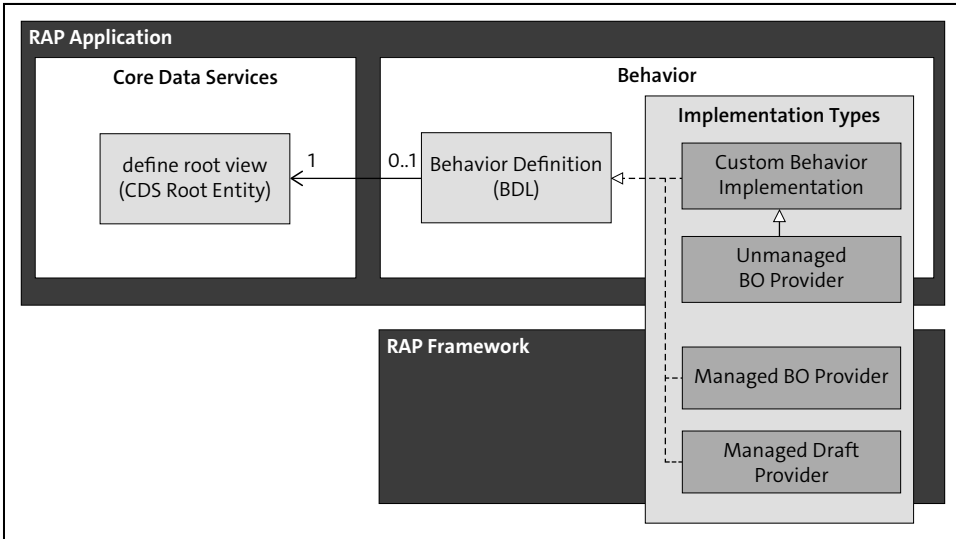


Figure 3.21 Implementation Types of a Behavior Definition

In Table 3.9, we've summarized each implementation type of a RAP business object and indicated in each case the responsibility for providing the implementation.

| Implementation Scope | Managed | Managed with Additional Save | Managed with Unmanaged Save | Unmanaged |
|------------------------------------------------------------------------------|---------|------------------------------|-----------------------------|-----------------------------------------------------------------------|
| Standard operations in the interaction phase and in the transactional buffer | RAP | RAP | RAP | Custom implementation |
| Complete save sequence | RAP | RAP | RAP* *except SAVE | Custom implementation |
| SAVE phase of the save sequence (see Chapter 5, Section 5.5.4) | - | Custom | Custom | - |
| Specific operations and internal business logic | Custom | Custom | Custom | Custom implementation; validations/ determinations only in draft mode |

Table 3.9 RAP Business Object Implementation Deployment by Implementation Type

Note: Projection Behavior Definition

You can use `projection` to declare the implementation type of a projection behavior definition that's based on a behavior definition. Chapter 6 provides more information about this type.

**3.3.1 Managed Scenario**

You can use the `managed` keyword if you want to use the managed business object provider to implement the RAP business object (see Listing 3.2). The managed business object provider implements CRUD operations across the interaction phase and save sequence.

`managed;`

```
define behavior for /DMO/I_Travel_M alias Travel
persistent table /DMO/TRAVEL_M...
```

Listing 3.2 “Managed” Implementation Type for a RAP Business Object

In the managed scenario, the ABAP RESTful application programming model takes care of saving changes to the transactional buffer for you and stores them in the database. For this purpose, you must specify the database table in the entity behavior definition using `persistent table`, followed by the database table name. This specification is only relevant in the managed scenario.

Note: Using Field Mappings

If you use `persistent table`, specifying a field mapping between the database table and the CDS entity with `mapping` is necessary or recommended (Section 3.9).



The following save options are available in the managed scenario:

- **Unmanaged save**
You can implement the save process yourself.
- **Additional save**
You can implement additional save logic.

This allows you to influence the save sequence, but all other properties of the managed implementation type are still valid.

In the following sections, we'll first describe the behavior definition level specification that applies to all CDS entities of a business object's composition tree. Finally, we'll demonstrate how you can also declare a save option specifically for a CDS entity.



Note: Other Combinations of the Save Options

It's also possible to combine the save options at the behavior definition level with those at the level of the respective CDS entity.

Unmanaged Save

By using the managed scenario with unmanaged save, you can implement the save process within the save sequence yourself (see the “SAVE_MODIFIED Method” subsection in Chapter 5, Section 5.5.4). The default save implementation of the managed business object provider (see the “SAVE Method” subsection in Chapter 5, Section 5.5.4) isn't run in this case. You can specify this using the addition `with unmanaged save`:

managed with unmanaged save;

```
define behavior for ZI_SalesOrder alias SalesOrder
...
```

If you specify the save option in the behavior definition head, it applies to all CDS entities. However, you can also define the save option at the level of the respective CDS entity (see the “Save Options at the CDS Entity Level” subsection coming up).

You can add `and cleanup` to the unmanaged save option. This allows you to implement the `cleanup` method in the behavior pool later:

managed with unmanaged save and cleanup;

```
define behavior for ZI_SalesOrder alias SalesOrder
...
```



Note: Using "persistant table" Not Permitted

If you use the managed scenario with unmanaged save, the specification of `persistant table` isn't allowed because you implement the save process within the save sequence yourself. The specification can therefore no longer be meaningfully evaluated by the RAP framework.

Using the further addition `with full data`, you can specify that you want to be supplied with the contents of *all* fields of the respective instances during the save sequence—regardless of whether they have been changed in the transactional buffer or not:

managed with unmanaged save with full data;

```
define behavior for ZI_SalesOrder alias SalesOrder
...
```

This means that you no longer have to read the complete field contents in the save sequence yourself because the RAP runtime does this for you. This makes it easier, for example, to set up a legacy application programming interface (API) for backing up instances whose interface requires you to transfer all field contents of the data to be saved in full.

Additional Save

If you want to add your own save logic to the process in the save sequence (whether with managed or with unmanaged save), you can use the `addition` with `additional save`:

managed with additional save;

```
define behavior for ZI_SalesOrder alias SalesOrder
...
```

This allows you to implement the updating of simple update documents at a later stage, for example. Here, you also have the option of using the `additions` and `cleanup` or `with full data`. The same rules for the specification in the behavior definition apply for the `additional save` option as for the `unmanaged save` option.

Save Options at the CDS Entity Level

You can also declare the `additional` or `unmanaged save` individually for one or more CDS entities at the entity behavior definition level (see Listing 3.3).

```
managed implementation in class ... unique;
strict ( 2 );

define behavior for ZI_SalesOrder alias SalesOrder
with additional save
...
```

Listing 3.3 Additional Save at the Level of a CDS Entity

The declaration of the `unmanaged save` would look as follows at this point:

```
...
define behavior for ZI_SalesOrder alias SalesOrder
with unmanaged save
...
```

3.3.2 Unmanaged Scenario

You can use the `unmanaged` keyword to declare the `unmanaged` implementation type for a RAP business object. In the behavior pool, you make the full implementation of the

interaction phase and save sequence in this case. Here, you need to specify a global ABAP class as the behavior pool in the head of the behavior definition using the keyword `implementation in class` (see Listing 3.4).

```
unmanaged implementation in class /DMO/BP_TRAVEL_U unique;
strict ( 2 );

define behavior for /DMO/I_Travel_U alias Travel
...
```

Listing 3.4 Using the Unmanaged Scenario

We'll describe the implementation process for this case in Chapter 5, Section 5.5. In Chapter 12, we've implemented a use case for the unmanaged scenario.



Note: Abstract Implementation Type

You can alternatively declare an abstract implementation type using the `abstract` keyword. Such a behavior definition is based on an abstract CDS entity. This implementation type is intended for typing purposes of parameters (of actions or functions). Only a severely limited set of syntactic elements is available. You can't declare transactional behavior there.

3.4 Strict Mode

The keyword `strict` allows you to specify that a stricter syntax check of the BDL source code should be performed. There are two versions of the strict mode available:

- Strict mode version 1 with a specification of `strict`
- Strict mode version 2 with a specification of `strict (2)`

When you create a behavior definition, `strict (2)` is used by default (see Listing 3.5).

```
managed implementation in class /DMO/BP_TRAVEL_M unique;
strict ( 2 );

define behavior for /DMO/I_Travel_M alias Travel
...
```

Listing 3.5 Strict Mode Definition in the Behavior Definition Head

The stricter syntax checks are based on ABAP RESTful application programming model best practices, and they ensure that the RAP business object remains stable against changes in the programming model. For example, if the strict mode is used in the unmanaged scenario, you must define the locking and authorization behavior for each

CDS entity. Otherwise, the behavior definition can't be activated. Using strict mode also ensures that you must declare implicit behavior explicitly in the BDL source code. We recommend that you use the strict mode by default for your behavior definitions.

3.5 Entity Behavior Definition

You can use `define behavior for`, followed by the CDS entity name, to initiate an entity behavior definition. You can specify any of the CDS entities in the business object composition tree, but you must specify the CDS root entity at a minimum (see Listing 3.6).

```
managed implementation in class /DMO/BP_TRAVEL_M unique;
strict ( 2 );
```

```
define behavior for /DMO/I_Travel_M alias Travel
...
```

```
define behavior for /DMO/I_Booking_M alias Booking
...
```

```
define behavior for /DMO/I_BookSuppl_M alias Booksuppl
...
```

Listing 3.6 Different Entity Behavior Definitions within a Behavior Definition

The `alias` addition enables you to use a descriptive name for the CDS entity. This is in contrast to the technical name of the CDS entity, which must be unique within the ABAP Dictionary, satisfy certain naming conventions, and be created in a specific namespace. The alias name is used as an identifier in derived data types to refer to the respective CDS entity. This facilitates access to the business object via EML.

Warning: Alias Name Is Visible Outside the Business Object

Note that the alias name is an externally visible identifier. It can be used both internally on the part of behavior implementation and by external consumers via EML. Changing the alias name can therefore lead to syntax errors on the consumer side. Therefore, you should assign the alias name wisely and change it only with caution.

By the way, this fact affects not only the alias name but also all other declarations of the behavior definition that belong to the externally visible part of the RAP business object.



With the addition `external '<external name>'`, you can assign another meaningful name for the CDS entity whose length isn't limited to 30 characters. This external name isn't visible in ABAP programs, but it can be found in the metadata of the OData service in which the CDS entity was exposed.

3.6 Defining a Behavior Pool

Depending on the implementation type and declared behavior, you need a separate implementation for this (e.g., a separate numbering assignment, action, determination, etc.). You can do this in the behavior pool, a special ABAP class of its own. You can declare one or more behavior pools at different levels of the behavior definition:

- **RAP business object**
Level of the behavior definition.
- **CDS entity**
Head of the entity behavior definition.
- **CDS entity**
Body of the entity behavior definition.

Thus, it's also possible to distribute the implementation to several behavior pools for a RAP business object and to combine the specified levels according to certain rules. This allows for parallel work on a RAP business object and also leads to better comprehensibility if there's a direct mapping of the CDS entity to the behavior pool.

3.6.1 Behavior Pool for Behavior Definition

You can declare a behavior pool in the head of the behavior definition. This way, you can implement all the necessary behavior for all CDS entities of the RAP business object in one place. You make the declaration with `implementation in class`, followed by the name of the implementation class, and end it with `unique` (see Listing 3.7).

```
managed implementation in class /DMO/BP_TRAVEL_M unique;
strict ( 2 );

define behavior for /DMO/I_Travel_M alias Travel
...
```

Listing 3.7 Declaring a Behavior Pool in the Head of the Behavior Definition

The addition `unique` is mandatory and ensures that implemented behavior is implemented without overlap.

Specifying a behavior pool at the behavior definition level is optional. However, if you need to implement the save sequence yourself because the keywords `unmanaged`, `with additional save`, or `with unmanaged save` are specified, you absolutely need a behavior pool at the behavior definition level for this. This also applies when the save option is defined at the CDS entity level, as shown earlier in Listing 3.3.

3.6.2 Behavior Pool for the CDS Entity

You can declare a behavior pool for each CDS entity within the behavior definition and implement the associated behavior there (except for a save option defined there; see Listing 3.8).

```
define behavior for /DMO/R_Travel_D alias Travel
implementation in class /dmo/bp_travel_d unique
...

define behavior for /DMO/R_Booking_D alias Booking
implementation in class /dmo/bp_booking_d unique
...
```

Listing 3.8 Behavior Pool at the CDS Entities Level

You can also combine the specification of a behavior pool at the behavior definition level with the specification at the CDS entity level. In this case, you must implement the save sequence in the behavior pool of the behavior definition and specific behavior of the CDS entities in the associated behavior pools (see Listing 3.9).

```
managed with additional save implementation in class zbp_i_salesorder_save unique;
strict ( 2 );

define behavior for ZI_SalesOrder alias SalesOrder
implementation in class zbp_i_salesorder unique
...
define behavior for ZI_SalesOrderItem alias Item
implementation in class zbp_i_salesorderitem unique
...
```

Listing 3.9 Behavior Pools for the Behavior Definition and the CDS Entities

Specifying a behavior pool at the CDS entity level is optional. If no behavior pool is specified here, you make the implementation of the CDS entity's behavior in the behavior pool of the behavior definition.

3.6.3 Behavior Pool for the Implementation Group

Implementation groups enable you to distribute implementation-relevant behavior of a CDS entity to separate behavior pools. If you use implementation groups for a CDS entity, you mustn't declare a behavior pool for the CDS entity. The global specification at the behavior definition level, however, is still permitted. For this reason, if you use implementation groups, each implementation-relevant behavior must be associated with an implementation group.

You use the keyword `group` followed by the name of the implementation group (here, it's `grpCancelation`) and the specification of the responsible behavior pool to declare an implementation group. The behavior you want to implement in the specified behavior pool must be enclosed in parentheses. In Listing 3.10, for example, the action `cancel` is supposed to be implemented.

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    create;
    update;
    delete;

    group grpCancelation implementation in class zbp_i_salesorder_cancel unique
    {
        action cancel result [1] $self;
    }
    group ...
    { ... }
}
```

Listing 3.10 Defining an Implementation Group for an Action



Tip: Meaningful Use of Implementation Groups

You should use implementation groups only if you want to distribute behavior across more than one implementation group. This is necessary if you want to enable parallel working within a CDS entity and thus have to use different behavior pools. For example, you can group technically related behavior into an implementation group.

The name of the implementation group must be different from the name of the respective behavior. For example, there mustn't be a `cancel` action if there's an implementation group with the same name, and vice versa. Implementation groups act only business object internally, so you can change the name and structure of implementation groups without violating the interface contract with consumers.

3.7 Numbering Assignment

The type of numbering assignment is a central aspect of an application that plays an important role already in the data modeling phase. In the following sections, we'll describe the basic principles of numbering assignment in the ABAP RESTful application programming model and show you the options provided for this by the programming model in the behavior definition.

Using the numbering assignment, you specify in the behavior definition for the respective CDS entity how its key fields are supplied with values when an instance of the CDS entity gets created. The numbering is thus relevant for the standard operations `create` and `create via association`. They are applied in the interaction phase as well as in the save sequence.

It's useful to distinguish between temporary and permanent key values:

- A *temporary key value* uniquely identifies a *newly created* instance only in the transactional buffer. This is necessary, for example, to access the instances or to establish associations between instances of different CDS entities of the business object composition tree in the transactional buffer.
- A *permanent key value* is the final key value with which the instance is persistently stored and permanently identifiable.

Using the RAP numbering assignment, you can define the way in which a *permanent* key value is assigned. The time of the numbering and the origin of the permanent key value are important in this context:

- **Time**

The time determines when the numbering assignment takes place. Here, we distinguish between early or late numbering assignment. The *early numbering assignment* takes place within the interaction phase, and the *late* one takes place within the save sequence.

- **Origin**

The origin defines whether the key value is set business object externally or business object internally. For *external numbering assignment*, the key value needs to be specified by the consumer (e.g., an end user via the UI or an EML consumer). *Internal numbering assignment* means that the key value is assigned business object internally.

In the following sections, we'll take a closer look at the possible combinations of time and origin and show how you can specify them in the behavior definition. The possible combinations are as follows:

- Early, external numbering assignment
- Early, internal numbering assignment
- Late, and thus internal, numbering assignment

In the context of numbering assignment, it's also important to check and report (e.g., to the user) as early as possible whether the selected key value is unique or not. This is called the *uniqueness check* in the ABAP RESTful application programming model. Its implementation depends on the type of numbering assignment:

- **Implementation for internal numbering assignment**

In the case of internal (early or late) numbering, the implementation of the check (e.g., via a number range) must ensure the uniqueness of the assigned key value. For UUID key values, a check for uniqueness isn't necessary.

■ Implementation for external numbering assignment

For external numbering assignment, the check can be performed by the RAP framework in the managed scenario without draft handling and for active instances with draft handling. Otherwise, you must provide for the implementation of the uniqueness check via a `precheck` (Section 3.17.2) or, for draft instances, via the `resume` action (Section 3.15.3).

3.7.1 Early, External Numbering Assignment

Early external numbering is the default numbering behavior for the CDS entity. You don't need to explicitly specify anything in the behavior definition for this. Consumers set the permanent key value during the interaction phase. This type of numbering is useful for meaningful key values or if you want to explicitly allow the consumer to assign external numbers, for example, in interface scenarios.



Example: Assignment of Material Numbers

The CDS root entity `Material` with a key field `Material number` can provide for early external numbering. In this way, a user can assign a material number according to company-specific rules.

3.7.2 Early, Internal Numbering

You can use the `early numbering` keyword to declare early numbering for a CDS entity (see Listing 3.11).

```
define behavior for ZI_SalesOrder alias SalesOrder
early numbering
...
{
  ...
}
```

Listing 3.11 Declaring Early, Internal Numbering Assignment

You can use early, internal numbering in the managed and in the unmanaged scenario with draft handling and implement numbering yourself.



Note: Method in Behavior Implementation

Using the `early numbering` keyword requires an implementation of the `FOR NUMBERING` method in the handler class (see Chapter 5, Section 5.4.11).

A variant of early, internal numbering is the use of UUIDs to identify the created instances. The RAP framework provides this feature by default, so this is a form of managed numbering.

To declare early, internal numbering, you must use the `field` statement with the field property `numbering : managed` and specify the corresponding key field (here, it's `SalesOrderUuid`; see Listing 3.12).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    field ( numbering : managed ) SalesOrderUuid;
    ...
}
```

Listing 3.12 Early, Internal Numbering with UUIDs

For more details on setting field properties with the `field` statement (e.g., to protect key fields against business object–external write access), see Section 3.8.

Tip: Using UUID Numbering

In Chapter 10, you can find a real-life example of how to use UUID numbering.



3.7.3 Late Numbering

Late numbering occurs as part of the save sequence, just before the instances are saved to the database. This can be used, for example, to ensure that instances are numbered without gaps.

You usually use number range intervals to implement a late numbering, for example, for an invoice or purchase order number. Another use case could be the assignment of a meaningful key according to a certain system. A new material number, for example, could be derived from certain values of the instance to be saved, such as the material type, the business field and the version status. These values aren't known at the `create` time, but only at the time of saving.

You can use the `late numbering` keyword in the head of each entity behavior definition to declare late numbering for a CDS entity (see Listing 3.13).

```
define behavior for ZI_SalesOrder alias SalesOrder
late numbering
...
{
    // Set key field to "read only"
    field ( readonly ) SalesOrderId;
    ...
}
```

Listing 3.13 Declaring Late Numbering

For the `SalesOrder` entity, late numbering is declared here. The `SalesOrderId` key field can be provided with a value within the save sequence via a number range, for example.



Note: Method in Behavior Implementation

When you specify `late numbering`, you must implement the `ADJUST_NUMBERS` method of the save handler in the behavior implementation (see Chapter 5, Section 5.5.3).

Late numbering is available in both the managed and unmanaged scenario. This is true irrespective of whether or not you use draft handling for the RAP business object. However, late numbering in the managed scenario is only available with SAP BTP ABAP environment 2111 and with SAP S/4HANA 2021 FPS01.



Note: Composite Keys of CDS Child Entities

If a CDS child entity has a composite key and thus includes the key of the CDS parent entity, late numbering must also be used for the CDS child entity.

3.8 Field Properties

When you add behavior to a CDS data model, all fields of the entity are usable for the standard operations `create`, `create-by-association`, and `update`. That's not always what you want. Usually, you want to define precisely which fields should be accessible in the context of these operations. This way, you can define the *interface* of the respective operation. These field properties are declared using the BDL keyword `field`.



Note: Checks for External Consumers Only

Note that restrictions you've made for certain fields apply only to *external* consumers of the RAP business object. Within the behavior implementation itself, corresponding checks, such as those specified by `field(mandatory:create)`, aren't performed.

Managed numbering assignment via UUIDs is also specified via a `field` property. For details, see Section 3.7.2.

3.8.1 Mandatory Fields

With `field(mandatory) f1, ..., fn;`, you can define one or more fields as mandatory fields for write operations. This concerns the operations `create`, `create-by-association`, and `update`. The respective field will then be marked as mandatory in the UI. However, you must initiate the check to determine whether it has actually been maintained (e.g., in the form of a validation; see Section 3.13.2).

With `field(mandatory:create) f1, ..., fn;`, you specify whether the fields are mandatory fields in the context of the operations `create` and `create-by-association` (see Listing 3.14).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  field ( numbering : managed ) SalesOrderUuid;
  // Set key to read only
  field ( readonly ) SalesOrderUuid;
  field ( mandatory : create ) SalesOrderType;
  ...
}
```

Listing 3.14 Mandatory Fields for the Create Operation

For the fields provided with the `mandatory:create` field property, the RAP framework checks whether they have been filled. If that's not the case, a runtime error (`BEHAVIOR_MANDATORY_FIELD`) will be thrown. Thus, no testing takes place within the behavior implementation.

You can generally use this field property for fields that *must be* provided with values by the consumer in the creation process, for example, a key value for external numbering assignment. It also makes sense to assign this field property to fields that must be provided with data as part of a creation operation so that the business object instance can be run in a meaningful way, for example, a document type `SalesOrderType` of a `SalesOrder` entity.

3.8.2 Protection Against Write Access

With `field(readonly) f1, ..., fn;`, you protect the specified fields from external write access; that is, their values can't be set by the operations `create`, `create-by-association`, or `update`. If an external consumer tries to change a read-only field, a runtime error will be triggered.

In Listing 3.15, you see the example of a `Person` entity that has a `PersonUuid` key field, and administrative fields `CreatedAt` and `CreatedBy`. The key fields or administrative fields are only open for read access.

```
define behavior for ZRAP_I_Person_M alias Person
...
{
  field ( numbering : managed ) PersonUuid;
  field ( readonly ) PersonUuid;
  field ( readonly ) CreatedAt, CreatedBy,
    LastChangedAt, LastChangedBy;
  ...
}
```

Listing 3.15 Setting Key Fields or Administrative Fields to “readonly”

In Listing 3.16, you can see an example of a foreign key field (here, it's `PersonUuid`, to which the `Address` entity belongs) that is protected against write access via `readonly`.

```
define behavior for ZRAP_I_Address_M alias Address
...
{
  field ( numbering : managed ) AddressUuid;
  field ( readonly ) PersonUuid, AddressUuid;
  ...
}
```

Listing 3.16 Setting Foreign Key Fields to “readonly”

Read-only fields are used in the following cases, for example:

- Key fields for internal numbering
- Fields for creating (composition) associations (foreign key fields)
- Administrative fields, such as the creation or modification time stamp
- Other calculated fields, such as a status or an amount calculated via a determination

With `field (readonly:update) f1, ..., fn;`, you specify that the specified fields are protected against write access only during update operations. The field is thus open during the create and create-by-association operations.

3.8.3 Combination: Mandatory Field for Creation and Write Protection for Updates

With `field (mandatory:create, readonly:update) f1, ..., fn;`, you can combine a field both as a mandatory field in the create case and as a read-only field in the update case (see Listing 3.17).

```
define behavior for ZRAP_I_Person_K_U alias Person
...
{
  field( mandatory : create, readonly : update ) PersonId;
  ...
}
```

Listing 3.17 Mandatory Field Combined with Write Protection for Updates

A frequent use case for this combination is controlling fields (e.g., for a document type) that have to be maintained at the time of creation and can't be updated later. Key fields in the context of external numbering also play an important role.



Note: Setting Field Properties Dynamically

You can also use the ABAP RESTful application programming model to determine the field properties at runtime. To learn how to do this, see Section 3.17.1.

3.9 Field Mappings

With *field mappings*, you can map the fields of a CDS entity to fields of a structured data type (a database table or structure) from the ABAP Dictionary. The following use cases exist for this:

- Field mapping between database table and CDS entity
- Field mapping between structure and CDS entity
- Field mapping between control structure and CDS entity
- Field mapping between structure and input parameter of an action

In the managed scenario, you can use `persistent table` to specify the database table where instances of the respective CDS entity are stored (Section 3.3.1). We recommend that you assign meaningful field names in the CDS entity, so that the names are different from the names of the respective columns in the database table. For this reason, you must use `mapping for` to declare a field mapping for the CDS entity between the CDS entity and the database table, which has been declared via `persistent table` in the behavior definition (see Listing 3.18).

```
define behavior for ZRAP_I_Person_M alias Person
persistent table zrap_a_ph
...
{
  mapping for zrap_a_ph corresponding
  {
    PersonUuid = person_uuid;
    Surname = surname;
    GivenName = given_name;
    DateOfBirth = date_of_birth;
    CreatedAt = created_at;
    CreatedBy = created_by;
    LastChangedAt = last_changed_at;
    LastChangedBy = last_changed_by;
  }
  ...
}
```

Listing 3.18 Field Mapping Between the Database Table and CDS Entity

With the addition `corresponding`, you can ensure that fields with the same name are mapped, even if they aren't explicitly listed.

A field mapping is also useful if you call existing ABAP modularization units, such as function modules, in the behavior implementation, and the structures used in the interface are different from the fields of the corresponding CDS entity. This may be the

case, for example, if you want to integrate the API of an existing application into the ABAP RESTful application programming model.

Let's suppose you want to use a Business Application Programming Interface (BAPI) within a behavior implementation to create orders. You can define a field mapping between the BAPI interface data types (here, it's `bapimepoheader`) and the CDS entity (see Listing 3.19).

```
...
  mapping for bapimepoheader corresponding
  {
    PurchaseOrder = po_number;
    PurchasingOrganization = purch_org;
    PurchasingGroup = pur_group;
    Supplier = vendor;
  }
...
```

Listing 3.19 Field Mapping for the “bapimepoheader” Structure

This central definition in the behavior definition allows you to use field mapping both in behavior implementation and in value assignments between the structured data type and the corresponding CDS entity. This is possible in both directions. We'll first show you an assignment with the CDS entity as the source:

```
DATA ls_po TYPE bapimepoheader.
ls_po = CORRESPONDING #( po_entity MAPPING FROM ENTITY ).
```

In this next example, you can see the reverse case, namely the assignment of the mapped structured data type `bapimepoheader` with the CDS entity as the target:

```
DATA ls_po TYPE bapimepoheader.
DATA ls_po_entity TYPE zi_rap_purchaseorder_m.
ls_po_entity = CORRESPONDING #( ls_po MAPPING TO ENTITY ).
```

BAPIs often provide parameters consumers can use to define which fields are to be changed when the BAPI is called (called *checkbox structures*). A RAP business object provides these types of structures by default in the interface of standard operations (%CONTROL structure; see Chapter 4, Section 4.1.1), which can be consumed via the EML and evaluated within the behavior implementation. For this reason, you can add the data type of the control structure to a field mapping using the `control` addition (see Listing 3.20).

```
...
  mapping for bapimepoheader control bapimepoheaderx corresponding
  {
    PurchaseOrder = po_number;
```



```

    PurchasingOrganization = purch_org;
    PurchasingGroup = pur_group;
    Supplier = vendor;
}
...

```

Listing 3.20 Field Mapping for Control Structures

We usually assume that the fields of the actual structure and the associated control structure have the same name. However, if there are name differences, you can define the different field name from the control structure within the mapping using the `control` addition at field level (see Listing 3.21).

```

...
mapping for zsales_order control zsales_order_x corresponding
{
    SalesOrder = sales_order_id;
    Customer = customer_id;
    DeliveryDate = delivery_date control delivery_date_x;
}
...

```

Listing 3.21 Field Mapping for Different Field Names in Control Structures

3.10 Standard Operations for a CDS Entity

The standard operations for the CDS entities of a business object include the CRUD operations (`create`, `read`, `update`, and `delete`). We distinguish between standard operations that are executed directly on the CDS entity and those that are executed via associations.

Note: Business Object–Internal and Business Object–External Consumers

Operations can be consumed both business object internally and business object externally if they haven't been explicitly declared as `internal` (Section 3.17.3). When we refer to consumers in the following sections, we always mean both business object–internal consumers (the behavior implementation) and business object–external consumers.



3.10.1 Create, Read, Update, and Delete

Standard write operations on a CDS entity include the creation (`create`), update (`update`), and deletion (`delete`) of business object instances. These operations perform changes to the data of a business object and can basically be specified optionally. The `read` operation is always implicitly available and doesn't need to be explicitly declared.

Accordingly, the keywords `create`, `update`, and `delete` allow you to specify which of the standard write operations the CDS entity supports, that is, which operations must be implemented depending on the implementation type (see Listing 3.22).

```
define behavior for /DMO/I_Travel_M alias Travel ...
...
{
  // Standard operations for the travel entity
  create;
  update;
  delete;
  ...
}
```

Listing 3.22 Example of Declaring Standard Operations

You can use `create` to specify that the CDS entity supports the `create` operation. Thus, consumers can create new instances of the CDS entity. The CDS entity is used to declare one or more key fields that must be populated during the creation operation. Numbering to uniquely identify the created instances across a RAP transaction plays an essential role (Section 3.7). The non-key fields of the CDS entity are populated as part of the creation operation. You can use field properties to specify which fields are mandatory or open for write access (Section 3.8).



Note: “Create” Operation Only for CDS Root Entity

The `create` operation can be declared only for CDS root entities. CDS child entities are created using the `create-by-association` operation (Section 3.10.2).

With `update`, you define that instances of the respective CDS entity can be updated. The operation passes corresponding non-key fields to the CDS entity for modification. The instance to be changed is identified by the values of the key fields. These values are set as part of the `create` operation and by their very nature can’t be changed.

You can use `delete` to specify that instances of the respective CDS entity can be deleted. The instance to be deleted is identified in each case by the values of the key fields. During the interaction phase, the instances to be deleted are flagged in the transactional buffer; during the save sequence, the instance is persistently deleted from the database.

3.10.2 Create and Read Operations by Association

Composition associations (`composition of`) and to-parent associations (`association to parent`) link CDS entities to each other and form the composition tree of a RAP business object. In the behavior definition, you can define these associations using the `create-`

by-association and read-by-association standard operations. The update and delete operations are declared directly for the respective CDS child entity.

Tip: Explicitly Specifying Associations

If you don't explicitly specify the create-by-association and read-by-association operations in the behavior definition, they will still be implicitly available for composition and to-parent associations. However, we always recommend declaring the operations explicitly in the behavior definition.



Read-by-Association

With the read-by-association operation, you grant consumers read access to the instances of CDS entities of the association target. You can read instances of the subordinate CDS child entities and the instance of the CDS parent entity.

You do this using the `association` keyword followed by the association name and empty curly brackets (see Listing 3.23). This example defines the reading of instances of the CDS child entity (i.e., in the direction of the composition association from the CDS data model).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  association _Item { }
}
```

Listing 3.23 Read-by-Association in the Direction of the Composition

Let's suppose there is a `SalesOrder` entity (sales order) that consists of `Item` instances (items). The `Item` instances for the respective `SalesOrder` instance can then be read via the linked `_Item` association.

Note: Read-by-Association Operation as a Prerequisite

You need the read-by-association operation as a prerequisite for declaring authorization checks (Section 3.14) or locking behavior (Section 3.12). The RAP framework uses these operations to read the respective master instance.



If you want to support reading the respective parent instance, you should define the already mentioned association operation in the CDS child entity (see Listing 3.24). This example defines reading the instance of the CDS parent entity (i.e., in the direction of the `to parent` association from the CDS data model).

```

define behavior for ZI_SalesOrderItem alias Item
...
{
    association _SalesOrder { }
}

```

Listing 3.24 Read-by-Association Operation in the Direction of a “to parent” Association

With this declaration, it's possible to read the corresponding `SalesOrder` instance starting from an `Item` instance.

Create-by-Association

With the create-by-association operation, you allow consumers to create one or more instances of a CDS child entity from its parent. This assumes an existing instance of the CDS parent entity. The operation can be performed only in the direction of the CDS child entity.

You can use the `association` keyword to declare the operation, followed by the association name and supplemented by the `create` operation (see Listing 3.25).

```

define behavior for ZI_SalesOrder alias SalesOrder
...
{
    association _Item { create; }
}

```

Listing 3.25 Declaration of Create-by-Association and Read-by-Association Operations

This allows you to create one or more `Item` instances based on a `SalesOrder` instance, which are associated with each other. It means that a foreign key relationship is established. The declaration of the read-by-association operation is thus supplemented by the `create` operation. The create-by-association operation can only be declared together with the read-by-association operation.

3.11 Specific Operations for a CDS Entity

Beyond the standard operations, additional specific operations can be declared for a CDS entity in the ABAP RESTful application programming model. These operations must then be implemented in the behavior pool. The following specific operations are possible:

- Actions
- Functions

The use of actions and functions is optional.

3.11.1 Actions

An *action* is a specific operation of a CDS entity that can perform business object–internal or –external change operations. An action is therefore always a write access. You can use it to implement your own business logic in the behavior pool. An action is declared in the BDL with the keyword `action`, followed by the name of the action. It's always assigned to a concrete CDS entity of the business object composition tree.

By default, an action is *instance-based*—it applies to concrete instances of the respective entity. For example, a `SalesOrder` entity that provides an instance-based `cancel` action (to cancel the `SalesOrder` instance) is conceivable here. When this action gets executed, it performs all the technically defined changes that make up the cancellation of a sales order, such as a change in status. Listing 3.26 shows this by using the instances of the `SalesOrder` entity as an example.

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    action cancel;
}
```

Listing 3.26 Declaration of an Instance-Based Action

In addition to instance-based actions, there are also *static actions*. A static action isn't performed on concrete instances of the entity, but applies to the entire entity. It maps cross-instance logic and is introduced with the keyword `static`. For example, an `Address` entity could have a `markDuplicates` action that identifies duplicates among all `Address` instances and marks them as such (see Listing 3.27).

```
define behavior for ZI_Address alias Address
...
{
    ...
    static action markDuplicates;
}
```

Listing 3.27 Declaration of a Static Action

Static actions are often factory actions. We'll describe this particular type of action in the upcoming "Factory Actions" subsection.

Input Parameter

You can define exactly one *input parameter* for actions and thus parameterize the behavior of the action. This applies to both instance-based and static actions.

You can use the `parameter` keyword followed by the appropriate data type to provide the action with an input parameter. The input parameter can have one of the following structured data types:

- Structured data type from the ABAP Dictionary
- Abstract CDS entity
- CDS entity for which the action was defined

The last option is relevant only for static actions because actions are instance-based by default.



Note: Advantages of Structured Parameters

You may wonder why RAP actions (and functions) support only one input or return parameter and why you can't declare a set of parameters in the way you're used to, from classical modularization units such as function blocks or methods. The parameter data types are structured data types (e.g., from the ABAP Dictionary or abstract CDS entities) in which you can include a set of fields. This way, you can map multiple parameters. If new fields are added, you only have to maintain them in one place, namely in the data type. Wherever the data type in question is used (e.g., in other methods within the behavior implementation), the new field is then also present, without you having to adapt the method signature(s) and calls, and add additional parameters. This facilitates the adaptation of existing coding.

For example, an action to cancel a sales order might receive a parameter of type `zrap_s_cancellation_opts` (see Listing 3.28). In addition, the consumer is able to pass the cancellation options to be applied (e.g., the type of remittance or whether the cancellation should skip certain checks).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    action cancel parameter zrap_s_cancellation_opts;
}
```

Listing 3.28 Input Parameter Typed with Structure

You can also use an abstract CDS entity to type the input parameter of an action. The definition of the abstract CDS entity can be found in Listing 3.29. Nonabstract CDS entities can't be used for typing.

```
@EndUserText.label: 'Cancellation options'
define abstract entity ZRAP_A_CancellationOpts
{
```

```

    payment_return_type : abap.char(3);
    force_cancellation  : abap_boolean;
}

```

Listing 3.29 Abstract CDS Entity for Typing the Input Parameter of an Action

In Listing 3.30, the abstract CDS entity `ZRAP_A_CancellationOpts` is used to type the input parameter.

```

define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    action cancel parameter ZRAP_A_CancellationOpts;
}

```

Listing 3.30 Input Parameter Typed with Abstract CDS Entity

Warning: “\$self” as Input Parameter

In strict behavior definitions (Section 3.4), explicitly specifying `$self` as an input parameter for instance-based actions isn’t supported and results in a syntax error. In nonstrict behavior definitions, the definition doesn’t result in an error until runtime. However, `$self` can be used as a data type for an input parameter in static actions.



Return Parameter

Not only can actions be provided with an input parameter, but they also can be provided with exactly one *output parameter* (or return parameter). This specification is optional. You can use the following data types for this purpose:

- CDS entity for which the action was defined
- Nonabstract CDS entity with behavior definition
- Abstract CDS entity
- Structured data type from the ABAP Dictionary

You can use the `result` keyword, followed by the cardinality and data type to define the return parameter of an action. You specify the cardinality in square brackets `[]`, declaring how many instances of the return parameter are returned by the action. For instance-based actions, the cardinality specifies how many return values are returned per instance passed to the action (via keys in the implementation) (see Table 3.10).

| Cardinality | Meaning |
|-------------|---------------------------------------------------------------------------------------------------------|
| [0..1] | The action returns, at most, one return value per passed instance. |
| [1] | The action returns exactly one return value per passed instance. |
| [0..*] | The action returns any number of return values per passed instance. |
| [1..*] | The action returns any number of return values per passed instance, but at least one for each instance. |

Table 3.10 Cardinalities for Return Parameters

Let's suppose there is a `Customer` entity associated with an `Address` entity via a composition relationship. A `Customer` instance can consist of multiple `Address` instances. The `Address` entity provides a `setDefault` action via the behavior definition (see Listing 3.31).

```
define behavior for ZI_Address alias Address
...
{
    ...
    action setDefault result [1] $self;
}
```

Listing 3.31 Custom CDS Entity as Type of the Return Parameter

If the action is executed on the respective passed `Address` instances, the corresponding instances are set as default addresses for the respective `Customer` instance, whereby the implementation ensures that there's only one default address per `Customer` instance. You can use `$self` to type the return parameter with the CDS entity for which the action is defined, that is, with `ZI_Address`. The cardinality of `[1]` indicates that there's exactly one return value for each instance passed to the action, which means that one `Address` instance will be returned. The action returns the data of the `Address` instance that was set as the default address.



Note: Cardinalities for Static Actions

Because static actions are executed without reference to one or more instances of the respective CDS entity, the cardinality for static actions declares how many instances will be returned when the action is called.

The `entity` keyword allows you to use a nonabstract CDS entity that has a behavior definition as the data type for the return parameter. Let's assume that there's a `SalesOrder` entity on the basis of which an `Invoice` entity can be created. The `Invoice` entity is a

standalone RAP business object. In this case, you can map the creation of the invoice as a `createInvoice` action, which returns the data for the created `Invoice` instance as a return parameter. The return parameter is typed with the corresponding CDS entity (in the example, `ZI_Invoice`) (see Listing 3.32).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    action createInvoice result [0..1] entity ZI_Invoice;
}
```

Listing 3.32 CDS Entity for Typing the Return Parameter

If you omit the `entity` keyword, you can only use a structured data type from the ABAP Dictionary or an abstract CDS entity as the data type of the return parameter, as opposed to a concrete entity from a RAP business object. Let's suppose you have a `SalesOrder` entity (sales order) that provides a `createInvoice` action, as in the previous example. The action creates an invoice based on the respective `SalesOrder` instance, now via an existing legacy API, rather than another RAP business object. In this case, the action could be provided with a data type for the invoice header data of the existing API as a return parameter (see Listing 3.33).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    action createInvoice result [0..1] zrap_legacy_invoice_head;
}
```

Listing 3.33 ABAP Dictionary Structure for Typing the Return Parameter

Note: Using the Selective Keyword

You can also equip the return parameter of an action with the `selective` keyword to allow the consumer to provide a choice of fields of the return parameter. Details about this keyword can be found in Section 3.11.2.



If you provide an action (or function) with an `entity` parameter, you must also include this entity in the respective service definition (see Chapter 6). The entity is exposed to OData as `EntityType` (in contrast to the structured data type), which is exposed as `ComplexType`. You can trace this in the metadata document for the OData service (via `$metadata`).

Factory Actions

A *factory action* is initiated with the `factory` keyword and creates new instances of the CDS entity to which it's assigned. A factory action can be instance-based or static. The data type of the return parameter is always implicitly `$self` and doesn't need to be specified explicitly. However, you must specify the cardinality explicitly with `[1]`.

You can use an instance-based factory action to enable the copying of a business object. For example, you can offer a specific instance of a `SalesOrder` entity to use as a copy template to create a new instance from. For this purpose, you can define a factory action named `copy` (see Listing 3.34).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    factory action copy [1];
}
```

Listing 3.34 Declaration of a Factory Action

You can also provide factory actions with an input parameter. This could, for example, control the copy process and determine which child CDS entities should and should not be copied. For example, a `SalesOrder` entity whose composition tree consists of `Partner` and `Item` entities could provide a factory action, `deepCopy`, whose input parameters allow the consumer to control whether the subordinate `Partner` or `Item` instances should be copied as well (see Listing 3.35).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    factory action deepCopy parameter zrap_s_so_copy_ops [1];
}
```

Listing 3.35 Factory Action with Input Parameter

You can see the definition of the `zrap_s_so_copy_ops` data type of the input parameter in Listing 3.36. The factory action implementation can evaluate the `copy_partner` or `copy_items` flags.

```
@EndUserText.label : 'SalesOrder, Copy Options'
@AbapCatalog.enhancement.category : #NOT_EXTENSIBLE
define structure zrap_s_so_copy_ops {
    copy_partner : abap_boolean;
    copy_items   : abap_boolean;
}
```

Listing 3.36 ABAP Dictionary Structure for Typing an Input Parameter

Factory actions can also be defined as static. In this case, they are applied at the CDS root entity level. For example, you can use a factory action to create and return instances with default values. Let's suppose you want to implement user management with a `User` entity. This entity has certain attributes, such as the user name or a flag indicating whether or not the `User` instance is locked. You could now declare a static factory action `createDefaultUser` that creates a new `User` instance (see Listing 3.37). In the implementation, the lock flag of the new instance is set by default.

```
define behavior for ZI_User alias User
...
{
    ...
    static factory action createDefaultUser [1];
}
```

Listing 3.37 Declaration of a Static Factory Action

Note: Static Factory Actions for CDS Child Entities

CDS child entities, by definition, require an existing CDS parent entity (Section 3.10.2). For this reason, you typically use instance-based factory actions at this level, even if static factory actions would be allowed.



Repeatable Actions

You can mark instance-based actions or functions (Section 3.11.2) as *repeatable* by using the addition `repeatable`. Such an action can be called multiple times within an EML call (or an OData change set) on an instance (with potentially different parameter values). The action interface provides the content ID `%CID` as a field with the `repeatable` addition so that the consumer and provider can differentiate between the individual action calls and their results (see Listing 3.38).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    repeatable action setDiscount parameter ZRAP_A_SetDiscount result [1] $self;
}
```

Listing 3.38 Declaration of a Repeatable Action

Save Actions

Save actions can be defined using the `save` addition and can be executed by external consumers or in the save sequence at the specified time. Save actions can be declared at two different times:

- As part of the finalization in the early memory sequence, that is, in the implementation of a determination (determination on save) or the `FINALIZE` method
- In the context of late numbering (Section 3.7.3), that is, in the `ADJUST_NUMBERS` method

To declare a save action for calling during finalization, you must use `save(finalize)`. During late numbering, you can declare a save action by using `save(adjustnumbers)`. You can also specify both times for an action in combination with `save(finalize, adjustnumbers)` (see Listing 3.39).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    save(finalize) action actionOnFinalize;
    save(adjustnumbers) action actionOnAdjustNumbers;
    save(finalize, adjustnumbers) action actionOnFinalizeAdjustNumbers;
}
```

Listing 3.39 Declaration of Save Actions

3.11.2 Functions

A *function* is a specific operation of a CDS entity with which you can implement calculations or complex selections. Unlike an action, a function is a read access and free of side effects for the business object. Thus, no lock is set when a function is processed. Functions allow you to implement your own business logic in the behavior pool. A function is declared in the BDL with the `function` keyword followed by the name of the function. It's always assigned to a CDS entity of the business object composition tree.

Like actions, functions can be instance based or static, and also have an input and a return parameter (Section 3.11.1). Note that in this context, unlike actions, the return parameter is a mandatory part of a function. This section includes some examples and different use cases of functions.

Let's suppose you've defined an `Address` entity and want to find duplicates among the address instances based on various criteria. You can implement this requirement using a static function that returns corresponding `Address` instances. The action could be named `findDuplicates`, for example. The return parameter of this function would be typed with the current CDS entity via `$self`, and thus with the CDS entity for which the function is declared (see Listing 3.40).

```
define behavior for ZI_Address alias Address
...
{
    ...
    static function findDuplicates result [0..*] $self;
}
```

Listing 3.40 Static Function with a Return Parameter Typed with “\$self”

You can also type the return parameter with a CDS entity of the business object composition tree. Let's suppose there's a `Project` entity defined in the context of a project management. This entity has a composition of tasks (`Task` entity). Task instances can have a "plan duration" attribute and predecessor-successor relationships among them. So, for example, a `Task` instance can't start until another `Task` instance has been completed. You now want to realize which `Task` instances are on the critical path of the project based on their plan duration and dependency relationship. You can implement this complex calculation using an instance-based function of the `Project` entity, which then internally calculates corresponding `Task` instances that lie on the critical path of the project and returns them as a result (see Listing 3.41).

```
define behavior for ZI_Project alias Project
...
{
    ...
    function calcCriticalPath result [0..*] entity ZI_Task;
}

define behavior for ZI_Task alias Task
...
```

Listing 3.41 Function with a Return Parameter Typed with a CDS Entity of the Business Object Composition Tree

You can also use an abstract CDS entity to type the return parameter. Let's suppose there's a `SalesOrder` entity with a composition of `Item` entities. Starting from a `SalesOrder` instance, you want to calculate the total of all discounts given based on associated `Item` instances. To do this, you can declare an instance-based function, `sumDiscounts`, that internally sums up the discounts and returns the result via a return parameter typed as an abstract CDS entity (see Listing 3.42).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
    ...
    function sumDiscounts result [1] ZRAP_A_SalesOrderDiscount;
}
```

Listing 3.42 Function with Abstract CDS Entity for Typing the Return Parameter

In Listing 3.43, you can see the definition of the abstract CDS entity.

```
@EndUserText.label: 'SalesOrder, Discounts'
define abstract entity ZRAP_A_SalesOrderDiscount
{
    @Semantics.amount.currencyCode : 'Currency'
```

```
DiscountSum : abap.dec(11,2);
Currency    : abap.cuky( 5 );
}
```

Listing 3.43 Declaration of an Abstract CDS Entity as Output Data Type of a Function

Using the `selective` keyword, you allow the consumer of the function to request only a certain selection of fields of the return parameter. The function can be implemented in a performance-optimized manner, taking into account the consumer's field selection. For example, you could add the `selective` keyword to the `calcCriticalPath` function mentioned earlier to read and return only specific fields of the `Task` instance requested by the consumer (see Listing 3.44). The `selective` keyword can also be used for actions.

```
define behavior for ZI_Project alias Project
...
{
    ...
    function calcCriticalPath result selective
        [0..*] entity ZI_Task;
}
```

Listing 3.44 Using the Selective Keyword in a Function

3.11.3 Functions for Defaulting

You can calculate default values in the behavior implementation for certain operations using a function for *field prefilling (defaulting)*. Such a function is only executed automatically from a UI based on SAP Fiori elements, not if you call the relevant operation directly via EML. Before an operation is performed, the function for defaulting is executed and the result is evaluated accordingly. For example, the input fields in the action popup are prefilled with the respective default values of the function. Functions for defaulting can also be called separately via EML, just like normal functions.

Defaulting is supported for the following operations:

- **Create and create-by-association**

When creating a business object instance, fillable fields can be prefilled.

- **Action and function**

The respective action or function must have an input parameter. The values of the input parameter can be stored with default values. The action or function can be static or instance based. Factory actions are also supported.

Note that the field preallocation function is a UI-relevant aspect in the behavior definition and is called automatically by UIs based on SAP Fiori elements. If, for example, you want to provide newly created instances in the transactional buffer with default values,

you should use a `determination` with the standard operation `create` as the trigger condition (Section 3.13.1). This would be pure business logic, completely independent of UI aspects.

You can use the `default function` keyword as an addition to the operation for which you want to calculate default values. Such a function is therefore not declared separately in the behavior definition, but for the operation that is to be defaulted. The name of the function must start with `GetDefaultsFor` or `GetDfltsFor`.

Let's suppose you want to preset the currency with a default value in the creation dialog on the UI when creating a `SalesOrder` instance to make it easier for users to enter it. To do this, add `default function` to the `create` operation of the CDS root entity, and implement the function accordingly (see Listing 3.45).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  create { default function; }
}
```

Listing 3.45 Function for Prefilling a Field for a Create Operation

The specification of a function name can be omitted here. However, when you specify a function name, this must be `GetDefaultsForCreate` for the `create` operation (see Listing 3.46).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  create { default function GetDefaultsForCreate; }
}
```

Listing 3.46 Explicit Specification of Function Name for Defaulting

If you want to prefill values during the creation of a CDS child entity, you must add a function for defaulting to the `create` operation within the association using `default function` (see Listing 3.47).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  association _Item { }
  { create { default function GetDefaultsForItem; } }
}
```

Listing 3.47 Function for Defaulting for Create-by-Association

If an action has declared an input parameter, you can prefill the values of the input parameter using a function. Let's suppose you want to use a `setReplacementMaterial` action for the CDS entity `Item` to calculate and set a suitable replacement material for the material available in the `Item` instance. The action has an input parameter to pass the number of the replacement material to the action. In this case, you need to add the default function **keyword** to the action and add the function name to the `GetDefaultsForReplacement` field prefilling (see Listing 3.48).

```
define behavior for ZI_SalesOrderItem alias Item
...
{
  action setReplacementMaterial parameter ZRAP_A_ReplacementMaterial
    result [1] $self { default function GetDefaultsForReplacement; }
}
```

Listing 3.48 Action with Function for Defaulting

Another use case would be if you wanted to prefill the input parameter of the `deepCopy` factory action. The factory action copies a `SalesOrder` instance, and the consumer uses the input parameter to control which CDS entities of the `SalesOrder` instance are to be copied and which ones aren't (refer to Listing 3.35 and Listing 3.36, respectively). For this purpose, you must add the default function **keyword** to the factory action, specifying the function name `GetDefaultsForDeepCopy` (see Listing 3.49).

```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  ...
  factory action deepCopy parameter zrap_s_so_copy_ops [1]
    { default function GetDefaultsForDeepCopy; }
}
```

Listing 3.49 Factory Action with Function for Defaulting

The prefilling of fields is also possible for the input parameter of a function (see Listing 3.50).

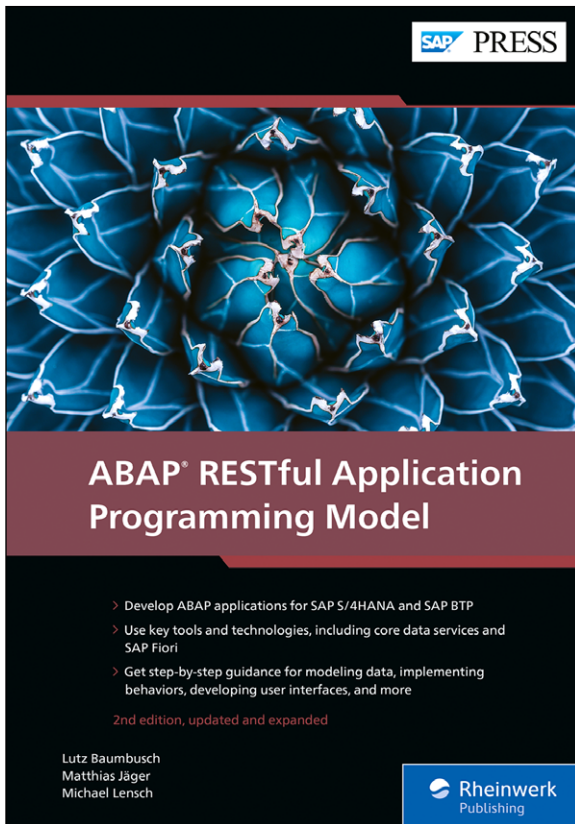
```
define behavior for ZI_SalesOrder alias SalesOrder
...
{
  ...
  function sumDiscounts parameter ZRAP_A_SalesOrderDiscountInput
```



```
    result [1] ZRAP_A_SalesOrderDiscount
  { default function GetDefaultsForSumDiscounts; }
}
```

Listing 3.50 Function with Function for Defaulting

Using the optional addition external '<external name>', you can define the externally used name of the function, which in this way becomes part of the OData model (refer to Section 3.5).



Baumbusch, Jäger, Lensch

ABAP RESTful Application Programming Model

- Develop ABAP applications for SAP S/4HANA and SAP BTP
- Use key tools and technologies, including core data services and SAP Fiori
- Get step-by-step guidance for modeling data, implementing behaviors, developing user interfaces, and more



www.sap-press.com/6161

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

Lutz Baumbusch, Matthias Jäger, and Michael Lensch are SAP developers with decades of combined experience with ABAP, SAP S/4HANA, cloud development, and more. Their current focus is ABAP application development using the latest programming model.

ISBN 978-1-4932-2752-5 • 576 pages • 07/2025

E-book: \$84.99 • Print book: \$89.95 • Bundle: \$99.99



Rheinwerk
Publishing