SAP PRESS

# ABAP® Cookbook

## Practical Recipes for Modern Programming

- Walk through practical tutorials using modern ABAP tools and techniques
- Get solutions for simple and complex programming tasks
- Work with the ABAP RESTful application programming model, ABAP Cloud, business configuration maintenance objects, and more

Fabian Lupa
Sven Treutler

Rheinwerk Publishing

# Contents

# Chapter 2
# The Application Scenario

*In this chapter, you'll get to know the sample application that will accompany you throughout this book. We'll add new functions to it in each chapter. This chapter first presents the basic structure of the application.*

We've chosen a recipe portal as our application scenario and basis for the programming examples in this book. It provides various functions. In the following chapters, we'll use this application scenario to explain the new features in ABAP Cloud and demonstrate their use in real-life scenarios.

**Structure of this chapter**

Section 2.1 begins by describing the technical requirements that are placed on our recipe portal. Based on these requirements, we then present the technical framework and implementation. In Section 2.2, we describe how the domain, data element, and database table repository objects can be created, and we explain in Section 2.3 how an initial, simple application based on the ABAP RESTful application programming model can be generated from them. In Section 2.4, we expand this application to include associations that can be used to add ingredients to a recipe and write reviews. Finally, the recipe portal is presented from a user's perspective in Section 2.5.

**Using the Associated Sample Application**

All programming examples used in this book can be found in the download material for this book at *www.sap-press.com/6198* and in our Git repository at *http://s-prs.co/v619800*. The objects created in this chapter are located in the `DATAMODEL` subpackage.

## 2.1 Concept of the Sample Application

In this section, we present both the functional and the technical concept of our sample application—a recipe portal.

**Use case diagram**

Figure 2.1 presents the requirements of a recipe portal in the form of a Unified Modeling Language (UML) use case diagram. This diagram serves to provide a general overview of the interaction options between all actors and the system.

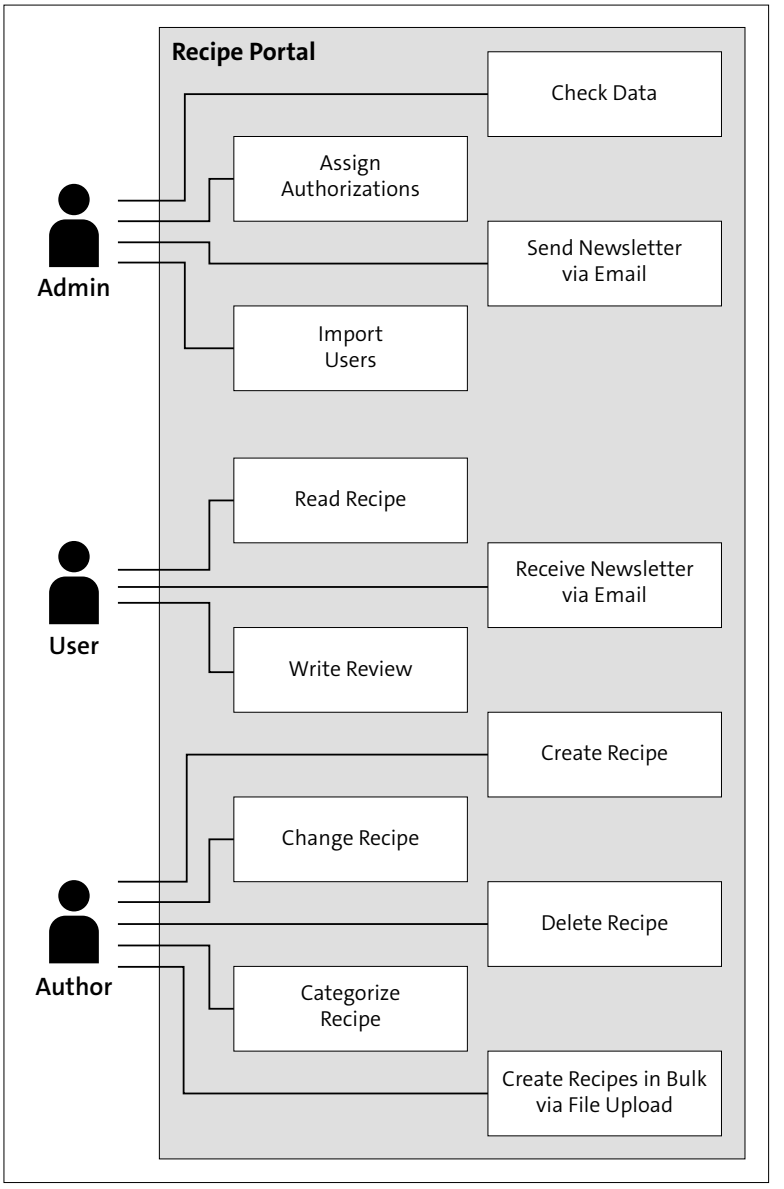



**Figure 2.1** Use Case Diagram for the Recipe Portal

In our application scenario, there are three actors:

- **User**
  This is a person who uses the recipe portal for research purposes and for finding new inspirations for dishes.
- **Author**
  This is a special user who has extended interaction options, such as creating recipes with ingredient lists.

- **Admin**
  This actor takes on technical tasks so that the recipe portal can be operated.

**User**

The user uses the application for the following tasks:

- **Reading a recipe**
  The user can read the instructions and ingredients for a dish.
- **Receiving an email newsletter**
  The user receives regular notifications about which recipes have been added.
- **Writing a review**
  The user can enter a rating for each recipe in the recipe portal.

**Author**

The following use cases are assigned to the author:

- **Creating a recipe**
  The author can enter a recipe with all the ingredients and a text about the recipe in the recipe portal.
- **Changing a recipe**
  The author can change the ingredients for a recipe, such as varying the quantity of an ingredient or deleting an ingredient.
- **Deleting a recipe**
  The author can delete the recipe. The associated ingredients and reviews will then be deleted as well.
- **Categorizing a recipe**
  The author can assign a recipe to different categories.
- **Uploading large numbers of recipes via file upload**
  The author has the option of uploading multiple recipes to the recipe portal via file upload.

**Admin**

The admin is responsible for the following use cases:

- **Sending an email newsletter**
  The admin can determine the recipes that are to be sent in a newsletter.
- **Importing users**
  The admin can import large numbers of users.
- **Checking data**
  The admin must check the consistency of the data in the database tables.
- **Assigning authorizations**
  The admin can assign authorizations to specific users so that they can use more or different functions than other users.

**Data model**

Based on the use case diagram in Figure 2.1, the technical concept for implementing the application scenario gets defined. In Figure 2.2, you can see the data model for the application scenario.

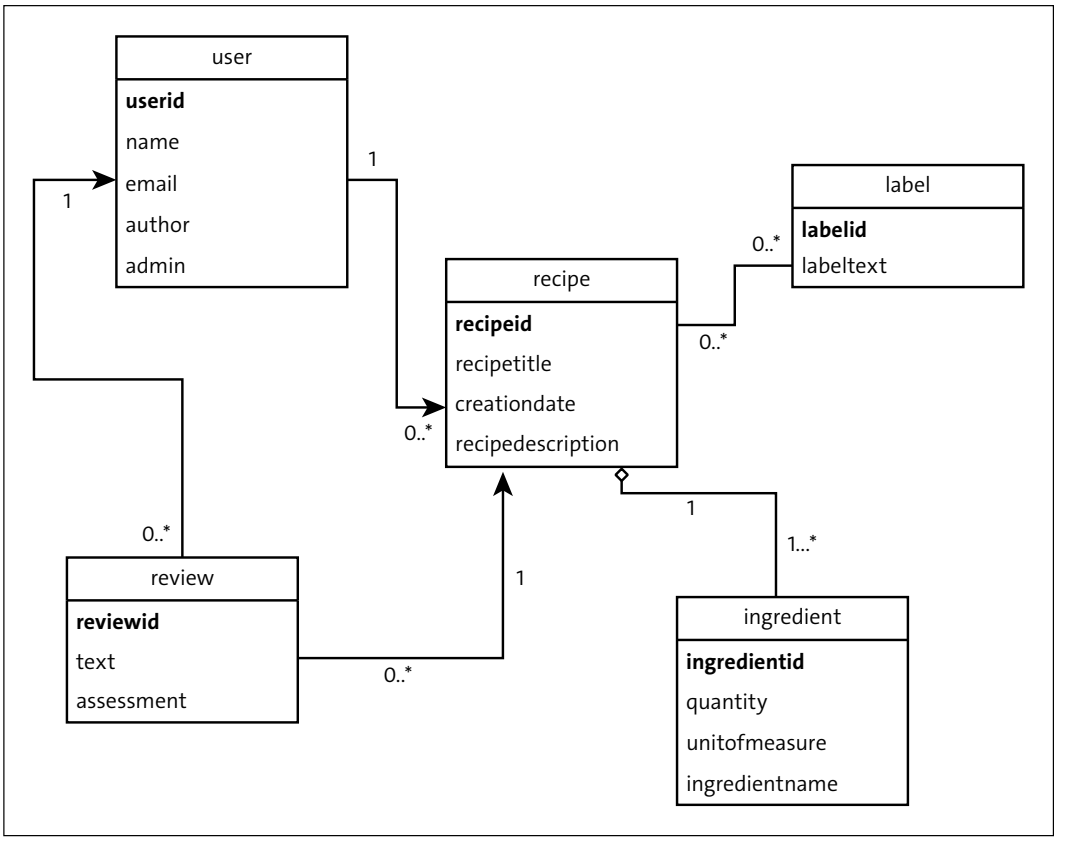



**Figure 2.2** Data Model for the Recipe Portal

In this data model, we use the following business objects:

- **User**
  The user is an actor in the system. It can be a regular user, an author, or an admin.
- **Recipe**
  This central object represents step-by-step instructions on how to prepare a particular dish. In addition to the title of the dish, it also contains the individual instructions for preparing the dish.
- **Ingredient**
  This object represents the components of a recipe that are required to implement it. The object consists of the number, a unit of measure (e.g., oz), and the actual ingredient (e.g., flour).

- **Label**
  This object represents the categorization of recipes. A recipe can be assigned multiple labels, such as "dessert" and "vegan."
- **Review**
  This object represents the feedback for a dish. A user can enter rating texts for a recipe.

**The Label Object**

We'll look at the label in more detail in Chapter 5. In the following sections, we'll focus on the installation of the other objects.

## 2.2 Creating Dictionary Objects

This section describes the technical implementation of the individual objects of our sample application in the SAP system, based on the business concept from Section 2.1.

**Framework parameters**

The following framework parameters and naming conventions apply to the use of the recipe portal:

- Original language of the objects is EN.
- Technical designations are provided in English, but comments and language-dependent texts are in English.
- All object names in the application example contain the abbreviation ACB.

**Creating dictionary objects**

Once you've connected our development environment, ABAP development tools, to an SAP system, you can create the necessary objects in the *ABAP Dictionary* as described next. These objects are used to map the data model in the system. To do this, you must create domains, data elements, and database tables as global repository objects. The advantage of those global repository objects is that they can be used multiple times, while they need to be created only once. This means that changes to these objects only have to be made once, and all users can benefit from them.

**Creating a package**

Before the dictionary objects can be created, you must first create a package:

1. Select your project in the project explorer, right-click, and choose **New • ABAP Package** from the context menu (see Figure 2.3).

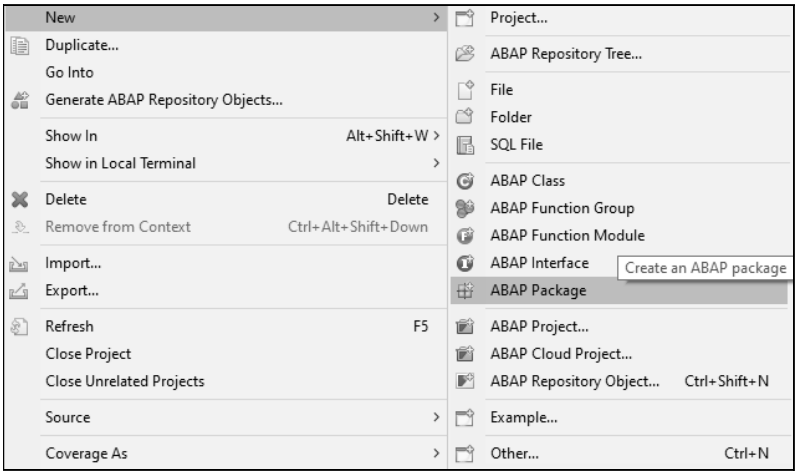



**Figure 2.3** Creating a Package

2. In the dialog that opens (see Figure 2.4), you need to define the project properties with the values from Table 2.1.

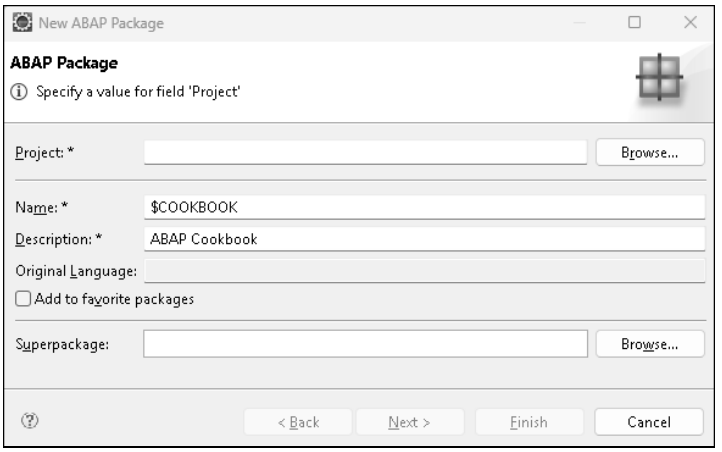



**Figure 2.4** Properties of a Package

| Field | Description | Value |
|---|---|---|
| **Project** | SAP system | This value will be entered automatically. |
| **Name** | Name of the package | $COOKBOOK |
| **Description** | Description of the package | Recipe Portal |
| **Package Type** | Type of package | Development |

**Table 2.1** Features of Our ABAP Package

3. Confirm the two dialog boxes that open next by clicking on the **Next** button. The $ character in the package name identifies our package as a local package that doesn't require any transport records.

**ABAP language version**

The $COOKBOOK package has now been created. It's now important to change the value in the **Default ABAP Language Version** field for this package to **ABAP for Cloud Development** (see Figure 2.5). This change ensures that all repository objects within the package must comply with the rules of the ABAP for cloud development language version setting.

The $COOKBOOK package must now be saved. It doesn't need to be activated. Now, we can create the repository objects.

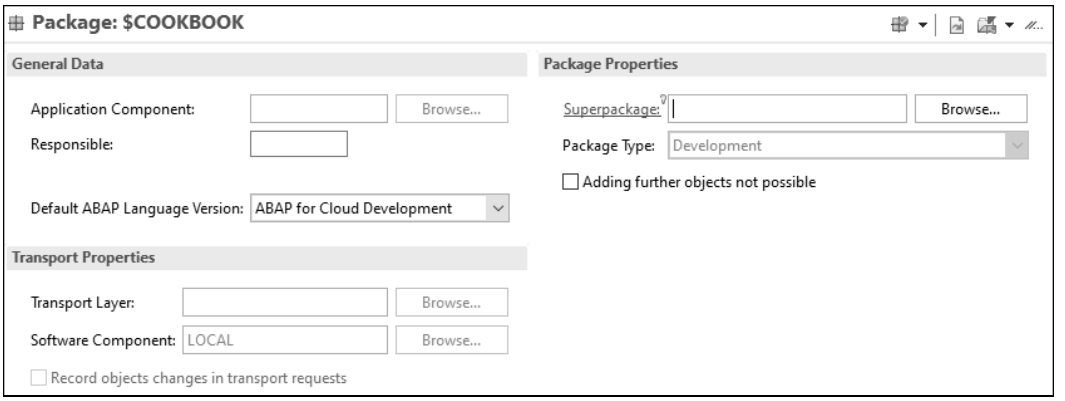



**Figure 2.5** Created $COOKBOOK Package

**Creating a domain**

First, we want to create the domains for the sample application. The domain dictionary object defines the technical and semantic properties of data types. At this point, we create a domain for the Recipe business object as an example. You can proceed in the same way for the other objects presented in the previous section:

1. Right-click on the package for our application in the ABAP development tools. In the context menu that opens, select the **New • Other ABAP Repository Object** path (see Figure 2.6).


**Figure 2.6** Creating a Domain via the Context Menu

2. In the search window that opens, search for the term "Domain", and then select it from the list (see Figure 2.7).


**Figure 2.7** Selecting ABAP Dictionary Object "Domain"

3. The dialog box shown in Figure 2.8 opens. Enter an appropriate name and the corresponding description for the domain here. For our example, we entered `ZACB_RECIPE_ID`.


**Figure 2.8** Popup Window for Creating a Domain

4. Click on the **Next** button.
5. The ABAP Dictionary editor opens where you can enter the technical properties of the domain (see Figure 2.9).

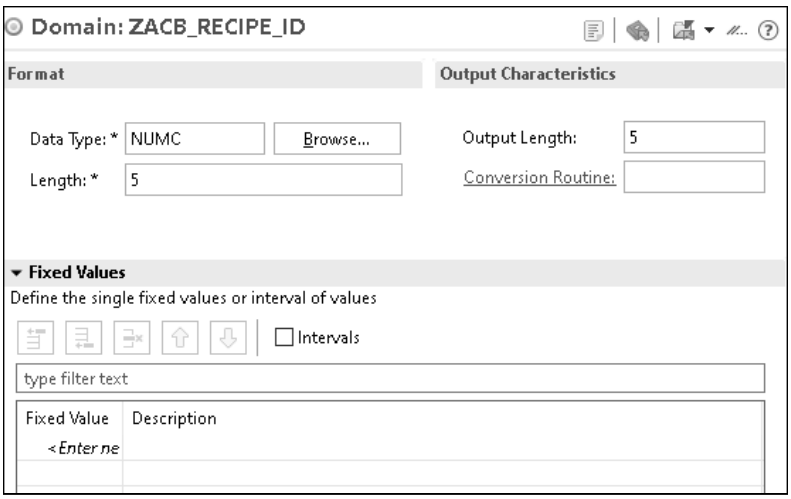



**Figure 2.9** Properties of the Domain

Enter the following details:

- **Data Type**: Here, you can choose between 13 different basic data types. They each represent the way in which numbers (integers and floating point representations), text fields, or date and time information will be displayed.
- **Length**: The length of a field value can either be fixed via the data type or specified manually for text fields.

The semantic properties described are rudimentary properties. In addition to these, much more can be specified in the semantic properties of the domain, for example, that only lowercase letters or only specific values are allowed. These restrictions can be made using fixed values or by specifying a table.

6. Once you've made all the necessary entries, the domain still needs to be saved. To do this, press the Ctrl+S keyboard shortcut.
7. Finally, the domain must be activated by pressing Ctrl+F3.

**Creating a data element**

After creating the domains, you can create the *data elements* for the application. A data element is also a repository object and defines the semantic meaning of an object. It's used for database tables or for defining variables.

Here, we create a data element as an example to use it for fields in a database table:

1. Unlike when creating the domain, you must call the context menu for the **Dictionary** entry within the package structure (see Figure 2.10). Select the **New • Data Element** path.

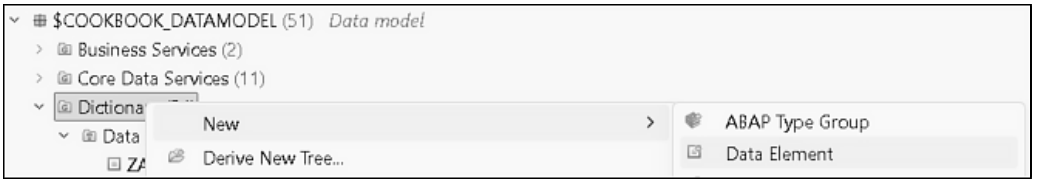



**Figure 2.10** Creating a Data Element via the Context Menu

2. In the dialog that opens next (see Figure 2.11), you must enter a name and a description for the data element. The name can have a maximum of 30 characters and can consist of letters, numbers, and underscores. It must be introduced with a letter or a namespace prefix. Click on the **Next** button.


**Figure 2.11** Creating a Data Element

3. Enter the following values in the ABAP Dictionary editor (see Figure 2.12):
   - **Category**: This field is used to specify the data type information.
   - **Type Name**: The name of the data element must be specified here. It can either be a domain provided by SAP or one that you've previously created yourself. In our example, we specify the `ZACB_RECIPE_ID` domain we created previously.
   - **Field Labels**: This area displays the texts that are displayed in the user interface (UI) when the data element is output. They are divided into four levels according to length. For our example, it's sufficient to enter the same value everywhere.

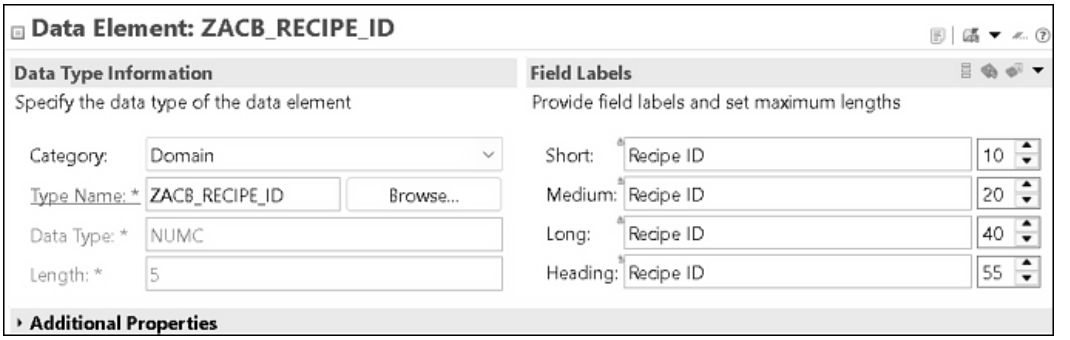



**Figure 2.12** Technical Properties of a Data Element

4. Save the data element by pressing the Ctrl+S shortcut.
5. In the final step, the data element must be activated (Ctrl+F3).

**Creating a database table**

Once the domain and the data element have been created, you can create the database tables for the individual business objects in accordance with the data model described in the previous section. We'll show you this here using the database tables for the recipe as an example:

1. Open the context menu for the **Dictionary** entry within our project structure, and select the following entry: **New • Database Table** (see Figure 2.13).


**Figure 2.13** Creating a Database Table via the Context Menu

2. Enter a name and a description in the following dialog. Enter "ZACB_RECIPE" as the name for our database table for the recipes. Once all mandatory fields have been filled in, click on the **Next** button (see Figure 2.14).


**Figure 2.14** Creating a Database Table

3. The ABAP Dictionary editor opens and generates an initial template for the database table in the form of annotations (see Listing 2.1):

```
@EndUserText.label : 'Recipes'
@AbapCatalog.enhancement.category : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table zacb_recipe {

  key client : abap.clnt not null;
}
```

**Listing 2.1** Generated Code for a Database Table

**Properties of the database table**

The name of the database table can be found in the following line:

```
define table <table name>
```

In the ABAP Dictionary editor, you can now add all other fields of the database table with the appropriate data element in the following format:

```
Field : data element
```

If a table field is a key, the `Key` attribute must also be added. Additional properties can be defined in the database table header via annotations (see Table 2.2).

| Annotation | Description |
|---|---|
| @EndUserText.label | Description of the database table |
| @AbapCatalog.enhancement.category | Whether the table be extended |
| @AbapCatalog.tableCategory | The table category involved |
| @AbapCatalog.deliveryClass | Whether it's an application or Customizing table |
| @AbapCatalog.dataMaintenance | How the table can be changed |

**Table 2.2** Annotations for a Database Table

Once all entries for the table have been made, the ZACB_RECIPE database table looks like Listing 2.2.

```
@EndUserText.label : 'Recipes'
@AbapCatalog.enhancement.category : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table zacb_recipe {
  key client            : abap.clnt not null;
  key recipe_id         : zacb_recipe_id not null;
  recipe_name           : zacb_recipe_name;
  recipe_text           : zacb_recipe_text;
}
```

**Listing 2.2** Database Table ZACB_RECIPE

The database table only needs to be saved (Ctrl+S) and activated (Ctrl+F3).

## 2.3 Generating an ABAP RESTful Application Programming Model Application

Now that the dictionary objects have been created, you can generate an ABAP RESTful application programming model application. For this purpose, the data model of the application is exposed via an Open Data Protocol (OData) service, on the basis of which an initial simple user UI can already be output.

**Using Generators**

The ABAP development tools provide multiple generators that help you create boilerplate coding and objects by generating code blocks based on parameters. This allows you to reach your goal much faster for new developments compared to a manual object creation. However, the generators can create only entirely new objects. You can't use them to edit existing objects. You can't restart the generation process based on a previous result. For this reason, you should check the suggested parameter values carefully to avoid manual reworking. In CDS views and the ABAP RESTful application programming model in particular, many interdependent or interrelated objects are used. It's difficult to fundamentally change, rename, or delete these at a later date.

### 2.3.1 Generating an OData Service and a Virtual Data Model

**Creating an OData UI service**

You can proceed as follows to generate the OData service and the virtual data model of your application:

1. Right-click on the **ZACB_RECIPE** database table you've just created in the project structure, and select **Generate ABAP Repository Objects** from the context menu (see Figure 2.15).

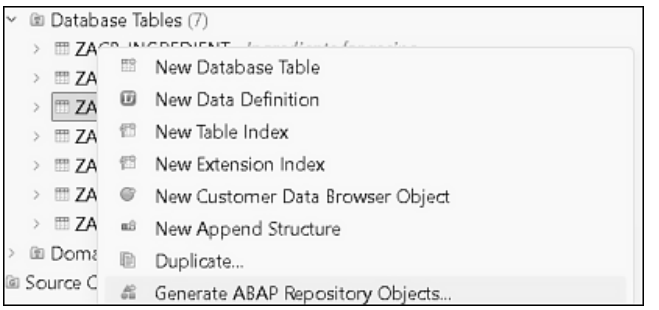



**Figure 2.15** Selecting the "Generate ABAP Repository Objects" Item from the Context Menu

2. In the generator that opens next, select the **OData UI Service** object type, and click the **Next** button (see Figure 2.16).

   You'll then receive the error message shown in Figure 2.17, which informs you that the database table is missing two fields with an administrative function, which are used to record its last change.

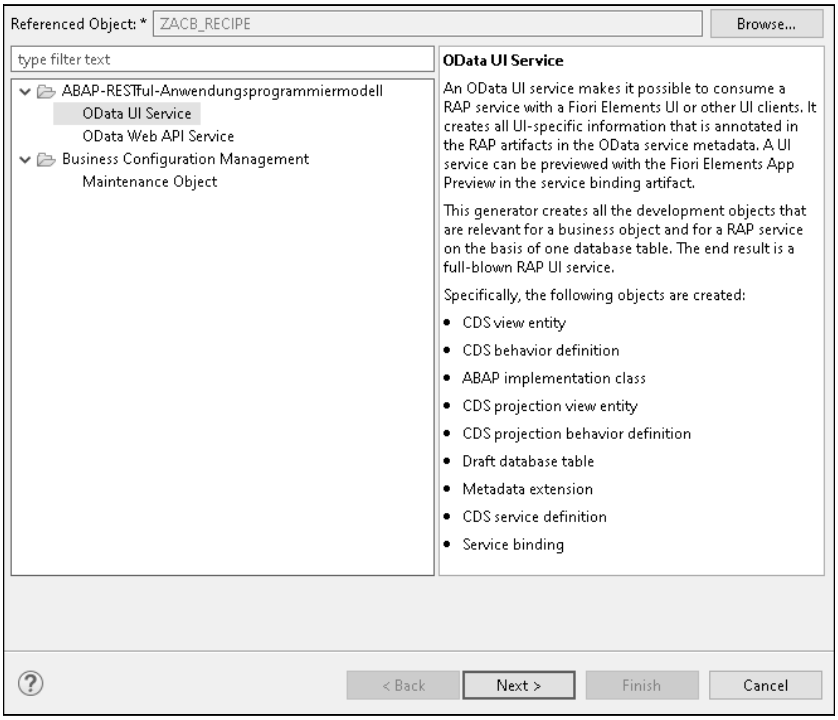



**Figure 2.16** Popup Window of the Generator for an OData UI Service

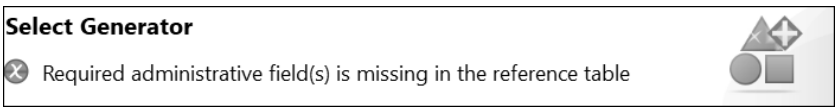



**Figure 2.17** Error Message When Generating the OData UI Service

These fields are required within an ABAP RESTful application programming model application and must be added to all underlying database tables. If you already have experience with ABAP RESTful application programming model applications, you can also complete this step directly when creating the database tables.

In Listing 2.3, we extend the database table with the aforementioned administration fields.

```
define table zacb_recipe {

  key client           : abap.clnt not null;
  key recipe_id        : zacb_recipe_id not null;
  recipe_name          : zacb_recipe_name;
  recipe_text          : zacb_recipe_text;
  created_by           : abp_creation_user;
  created_at           : abp_creation_tstmpl;
```

```
  local_last_changed_by : abp_locinst_lastchange_user;
  local_last_changed_at : abp_locinst_lastchange_tstmpl;
  last_changed_at       : abp_lastchange_tstmpl;
  last_changed_by       : abp_lastchange_user;

}
```

**Listing 2.3** Extension of Database Table ZACB_RECIPE

3. Activate the extended database table, and right-click to call the generator again.
4. In the dialog step that follows, you must specify a package. Select the package of our sample application, and confirm by clicking the **Next** button.
5. The subsequent dialog box (see Figure 2.18) displays the default names of the various objects in our ABAP RESTful application programming model application that are created when the OData UI service is generated. Check the suggested names on all layers listed in the **RAP Layers** section, and continue by clicking the **Next** button.

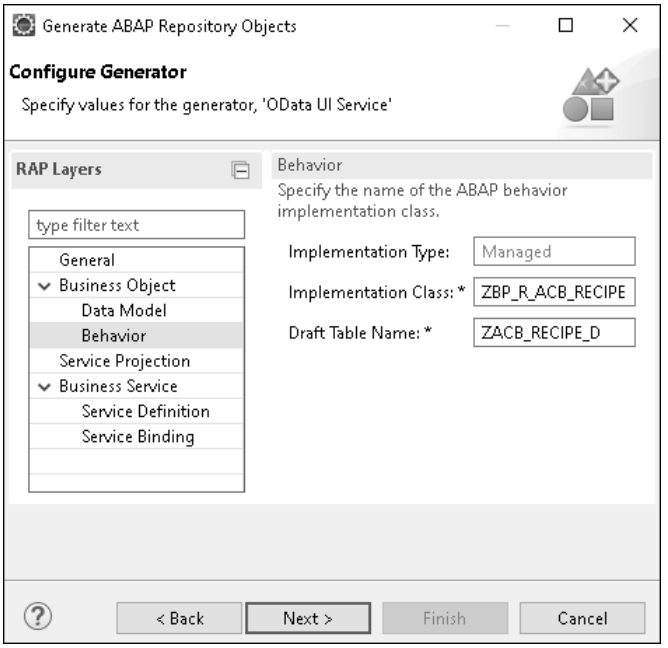



**Figure 2.18** ABAP RESTful Application Programming Model Application Objects Created by the Generator

6. Enter a transport request in the step that follows. After that, you can start generating the objects by clicking the **Finish** button.

**Naming the Objects**

Remember to check the names of the generated objects. They should match the conventions you've chosen. If that's not the case, you need to correct the names directly at this point. A subsequent correction can still be made, but this results in more work.

### 2.3.2 Generated CDS Entities of the ABAP RESTful Application Programming Model Application

**Generated CDS entities**

Table 2.3 lists the individual objects of the ABAP RESTful application programming model application that are created via the generator.

| Object Name | Explanation |
|---|---|
| Base entity ZACB_R_RECIPE | This data definition defines the data model of the root entity. |
| Basic behavior definition ZACB_R_RECIPE | This behavior definition describes the standard transactional behavior of the base entity. The behavior definition can be used directly in the managed scenario and also implements the draft concept of the ABAP RESTful application programming model (see Chapter 1, Section 1.6). |
| Behavior class ZBP_ACB_R_RECIPE | This ABAP class provides the implementation of the behavior defined in the behavior definition. |
| Draft table ZACB_RECIPE_D | This database table is used to temporarily store the draft data at runtime. It's managed by the ABAP RESTful application programming model framework. |
| Projection entity ZACB_C_RECIPE | This data definition is used to define the projected data model of the entity that is relevant for the current scenario. |
| Projection behavior definition ZACB_C_RECIPE | This behavior definition describes the behavior of the underlying base entity. |
| Metadata extension ZACB_C_RECIPE | The metadata extension is used to display the UI of the application via CDS annotations. |

**Table 2.3** Generated Objects Overview: ABAP RESTful Application Programming Model Application

| Object Name | Explanation |
|---|---|
| Service definition ZACB_UI_RECIPE | A service definition is used to define the relevant entity sets for our service and also to provide aliases if required. |
| Service binding ZACB_UI_RECIPE_04 | The service binding is used to make the generated service definition available as an OData UI service. |

**Table 2.3** Generated Objects Overview: ABAP RESTful Application Programming Model Application (Cont.)

**Relationships between the objects**

These objects were generated automatically. The relationships between the individual objects are illustrated in Figure 2.19. All objects are available to us as CDS entities.

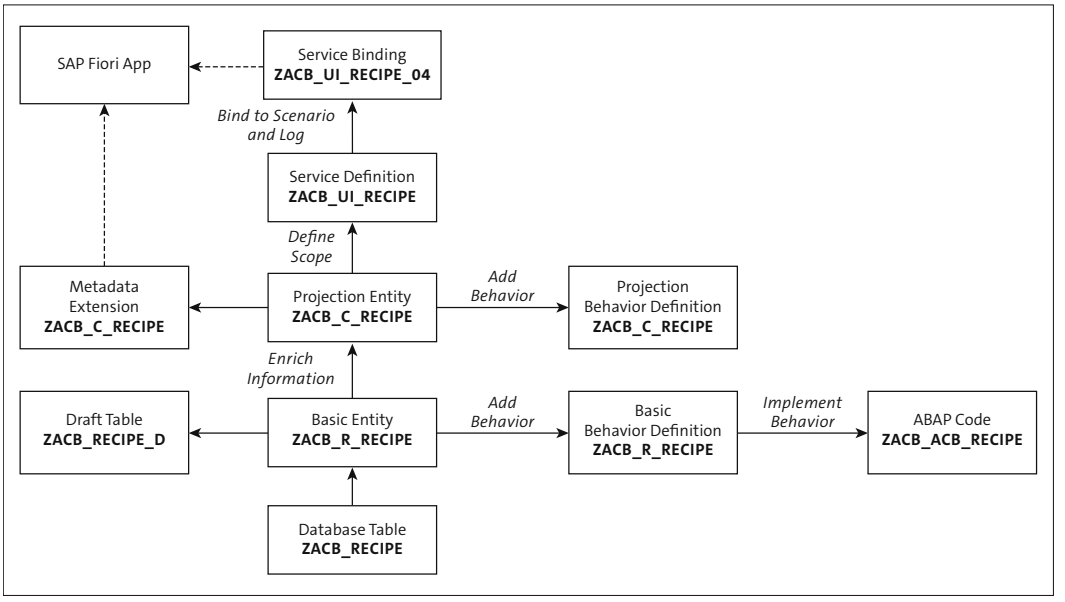



**Figure 2.19** CDS Entity Names of the Generated Objects and Relationship Diagram

**Base entity**

You can now view the individual CDS entities and expand them as required. We want to start with the ZACB_R_RECIPE base entity (see Listing 2.4). This *CDS view* defines the Recipe business object. A CDS view is a virtual structure. It can be used to combine complex and calculated data from different sources and merge them into a single view.

```
@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Base entity recipe'
```

```
define root view entity ZACB_R_Recipe
  as select from zacb_recipe

{
  key recipe_id as RecipeId,
  recipe_name as RecipeName,
  recipe_text as RecipeText,
  @Semantics.user.createdBy: true
  created_by as CreatedBy,
  @Semantics.systemDateTime.createdAt: true
  created_at as CreatedAt,
  @Semantics.user.localInstanceLastChangedBy: true
  local_last_changed_by as LocalLastChangedBy,
  @Semantics.systemDateTime.localInstanceLastChangedAt: true
  local_last_changed_at as LocalLastChangedAt,
  @Semantics.systemDateTime.lastChangedAt: true
  last_changed_at as LastChangedAt,
  @Semantics.user.lastChangedBy: true
  last_changed_by as LastChangedBy,
}
```

**Listing 2.4** Base Entity ZACB_R_RECIPE

The CDS code contains the following sections: First, the access control to the CDS view is described, and the label is specified. This is followed by the definition of a `root view entity` and access to a database table from which the data is to be imported (`as select from`). The last part describes the available data of the base entity. Here, it's specified which fields are `key` fields, and aliases are assigned for the field identifiers.

**Behavior definition of the base entity**

Let's now take a look at the behavior definition for the base entity (see Listing 2.5). Behavior definitions are a crucial part of CDS data models because they determine what can be done with our data.

```
managed implementation in class zbp_acb_r_recipe unique;
strict ( 2 );
with draft;

define behavior for ZACB_R_Recipe alias Recipe
persistent table zacb_recipe
draft table zacb_recipe_d
etag master LocalLastChangedAt
total etag LastChangedAt
authorization master ( global )
{
```

```
  field ( mandatory : create )
  RecipeId;

  field ( readonly )
  CreatedAt,
  CreatedBy,
  LastChangedAt,
  LocalLastChangedAt,
  LocalLastChangedBy;

  field ( readonly : update )
  RecipeID;

  create;
  update;
  delete;

  draft action Edit;
  draft action Activate optimized;
  draft action Discard;
  draft action Resume;
  draft determine action Prepare;

  mapping for zacb_recipe
    {
      RecipeId           = recipe_id;
      RecipeName         = recipe_name;
      RecipeText         = recipe_text;
      CreatedBy          = created_by;
      CreatedAt          = created_at;
      LocalLastChangedBy = local_last_changed_by;
      LocalLastChangedAt = local_last_changed_at;
      LastChangedAt      = last_changed_at;
      LastChangedBy      = last_changed_by;
    }
}
```

**Listing 2.5** Behavior Definition ZACB_R_RECIPE for the Base Entity

**Components of the behavior definition**

In the behavior definition, the behavior is defined using the *Behavior Definition Language* (BDL). The behavior definition consists of the following information:

- **Type of scenario**
  The specification of the scenario determines the automatic availability

of operations. In the managed scenario, the method behavior is determined by the framework and can be supplemented by a custom logic. In the unmanaged scenario, you must implement the method.

- **Strict mode**
  In strict mode, additional syntax checks are applied to behavior definitions. This means that obsolete syntax can't be used, and implicit operations must be declared explicitly. The latest mode is always recommended (currently this is `strict(2)`).
- **Alias**
  An *alias* specifies a descriptive name for the business object so that the CDS view name doesn't always have to be used.
- **Draft mode (`with draft`)**
  The draft mode is a function that allows end users to start and pause their work on the entities and resume it later without having to save this data directly in the database. Following are the important objects for this mode:
  - `DraftTable`: Name of the draft table.
  - `Totaletag`: Timestamp for the draft.
  - `Draftaction`: Standard actions for the draft.
- **Lock**
  The `lock master` information is used to lock the instance during a changing action.
- **ETag**
  *ETag* prevents accidental overwriting when an object is edited concurrently. The `etag master <field name>` specification defines the field in which a timestamp is to be saved.
- **Authorization checks**
  The `authorization master ( instance )` definition calls the corresponding method in the implementation of the behavior class in which an authorization check is carried out.
- **Field properties (`field`)**
  The field properties define the behavior of a field. The following statements are used for this purpose:
  - `Field(readonly)`: Display only
  - `Field(readonly:update)`: Only display, except during creation
  - `Field(mandatory)`: Required field
  - `Field(suppress)`: Don't display metadata
  - `Field(numbering:managed)`: Automatic numbering

- **Standard actions**
  The standard actions to be implemented are specified. Possible actions are listed here:
  - `CREATE`: Adding a data record
  - `UPDATE`: Updating a data record
  - `DELETE`: Deleting a data record
- **Mapping**
  As part of the mapping action, the field names of the database table are assigned to those with field names in the CDS entity.

**Draft table**

Next, take a look at the draft table (see Listing 2.6). This is a separate database table for data in the design stage with the structure of the respective entity, that is, the field names correspond to those from the respective CDS entity.

```
@EndUserText.label : 'Draft recipe'
@AbapCatalog.enhancement.category : #EXTENSIBLE_ANY
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table zacb_recipe_d {

  key mandt          : mandt not null;
  key recipeid       : zacb_recipe_id not null;
  recipename         : zacb_recipe_name;
  recipetext         : zacb_recipe_text;
  createdby          : abp_creation_user;
  createdat          : abp_creation_tstmpl;
  locallastchangedby : abp_locinst_lastchange_user;
  locallastchangedat : abp_locinst_lastchange_tstmpl;
  lastchangedat      : abp_lastchange_tstmpl;
  lastchangedby      : abp_locinst_lastchange_user;
  "%admin"           : include sych_bdl_draft_admin_inc;
}
```

**Listing 2.6** Draft Table ZACB_RECIPE_D

In addition to the fields of the original database table, the draft table also contains the administration fields of the `SYCH_BDL_DRAFT_ADMIN_INC` structure.

**Projection entity**

Furthermore, the `ZACB_C_RECIPE` *projection view* was created (see Listing 2.7). This projection view serves as a link between the `ZACB_R_RECIPE` base entity and the application. With the help of such projection entities, the fields and functions of applications can be further restricted because the full range of

functions of the base entities may not be required for the application. Various projection entities can therefore be created from a base entity and made available to the application level. Compared to the base entity, our projection entity only contains the following addition:

```
provider contract transactional_query
as projection on ZACB_R_Recipe
```

This command is used to indicate that it's a projection of the `ZACB_R_Recipe` base entity. In addition, the behavior of the projection and the behavior of various functions are defined (`provider contract`, see Listing 2.7).

```
@AccessControl.authorizationCheck: #CHECK
@Metadata.allowExtensions: true
@EndUserText.label: 'Projection entity recipe'
@ObjectModel.semanticKey: [ 'RecipeID' ]
define root view entity ZACB_C_Recipe
  provider contract transactional_query
  as projection on ZACB_R_Recipe
{
  key RecipeId,
  RecipeName,
  RecipeText,
  LocalLastChangedAt,
}
```

**Listing 2.7** Projection Entity ZACB_C_Recipe

**Behavior definition of the projection entity**

In addition to the base entity with its behavior definition, a `ZACB_C_Recipe` behavior definition was also created for the projection entity (see Listing 2.8).

```
projection;
strict ( 2 );
use draft;
define behavior for ZACB_C_Recipe alias Recipe
use etag

{
  use create;
  use update;
  use delete;
  use action Edit;
  use action Activate;
  use action Discard;
  use action Resume;
```

```
  use action Prepare;
  use association _Ingredient { create; with draft; }
  use association _Review { create; with draft; }
}
```

**Listing 2.8** Behavior Definition of the Projection Entity

**Metadata definition**

In addition to the base and projection entities and their behavior definitions, a metadata extension was generated. A metadata extension uses annotations to map the graphical UI. Among other things, the following *UI annotations* can be found here:

- *Facets* are areas of the UI that are defined using the `@UI.facet` annotation. Listing 2.9 shows an example of such an annotation.

```
@UI.facet: [ {
 id: 'idIdentification',
 type: #IDENTIFICATION_REFERENCE,
 label: 'Recipe',
 position: 10
} ]
```

**Listing 2.9** Annotation for a Facet

A special facet is the one for the identification area, which is defined using the `@UI.identification` annotation. Fields in this area are intended to identify the object. This annotation (see Listing 2.10) must be assigned to all fields that are supposed to be displayed here.

```
@UI.identification: [ {
  position: 10 ,
  label: ''
} ]
RecipeId;
```

**Listing 2.10** Annotation for the Identification Area

- *General field annotations* can be used in different places. The following annotations are possible here:
  - `Hidden`: The field isn't displayed and can't be added during personalization.
  - `Position`: This specifies the order of the fields.
  - `Label`: This specifies the name of the field in the UI.

**Service definition**

The service definition (see Listing 2.11), which is also generated, describes which CDS entities of a data model are to be published. With the help of an alias name, the technical names for the service can be replaced by more descriptive names.

```
@EndUserText.label: 'Recipe'
define service ZACB_UI_RECIPE {
  expose ZACB_C_Recipe     as Recipe;
}
```

**Listing 2.11** Service Definition

**Service binding**

The service binding is generated based on the service definition. You can also take a closer look at this. Service bindings assign a protocol to the previously defined services. They can be published directly locally. The following binding types can be selected:

- **OData 2.0 (V2) or OData 4.0 (V4)**
  SAP recommends using the OData V4 version, which supports all functions such as the draft function.
- **UI**
  This transfers the UI annotations.
- **Web API**
  This creates a web service without UI annotations.
- **InA – UI**
  This is used for analytical data models.
- **SQL – Web API**
  This provides access with ABAP SQL.

### 2.3.3 Publishing the OData Service

**Manual or automatic publication**

An OData V4 service is made available with the generated service binding. If it has the **Unpublished** status, there are two ways to publish it:

- **Button click**
  You can click on the **Publish** button to publish the service automatically.
- **Manual activation via Transaction /IWFND/V4_ADMIN**
  Depending on the SAP S/4HANA version, automatic publishing may not yet be supported. If so, a manual activation via Transaction /IWFND/V4_ADMIN is necessary.

**OData**

*Open Data Protocol* (OData) is an HTTP-based protocol for the exchange of data between systems. OData allows you to request and write data to resources using familiar operations such as `GET`, `POST`, `PUT`, `DELETE`, and `PATCH`.

**Publishing in Transaction /IWFND/V4_ADMIN**

To publish the OData service manually, follow these steps:

1. Log on to the SAP system, and call Transaction /IWFND/V4_ADMIN. This transaction enables OData V4 services to be published.
2. Click on the **Publish Service Groups** button.
3. Search for the service to be published using the **System Alias** and **Service Group ID** fields. You can also use wildcards. Click on the **Get Service Groups** button to start the search (see Figure 2.20).

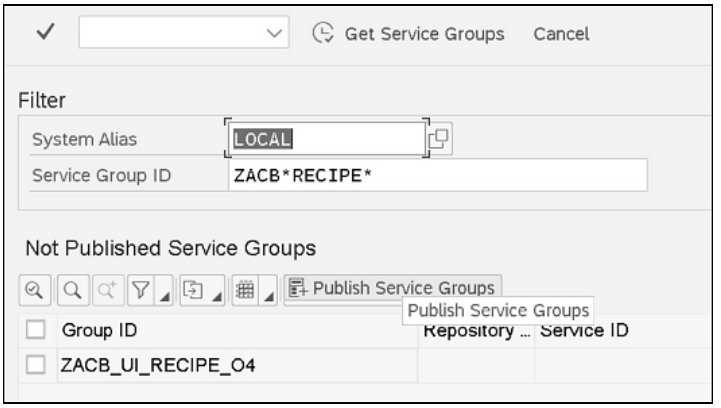



**Figure 2.20** Search Filters and List of All Service Groups Found

4. As a result, all service groups matching the search criteria will be displayed. In our example, this list only contains one entry.
5. Select the entry with the appropriate **Group ID**, and click on the **Publish Service Groups** button.
6. You can change the description of the service group in the dialog box that opens.
7. In the information window that opens next, click on ☑ (**Execute**), and then click on the **Back** button.

At this point, the service has been published in a service group and can be used.

### 2.3.4 Testing the Application

**"Published" status**

Using the preview function of the ABAP development tools, you can now check directly in the development environment whether the publication of the OData UI service was successful. To do this, you need to call the service binding. If the **Published** status is displayed here in the **Local Service Endpoint** field (see Figure 2.21), you can open a preview of the SAP Fiori app UI.

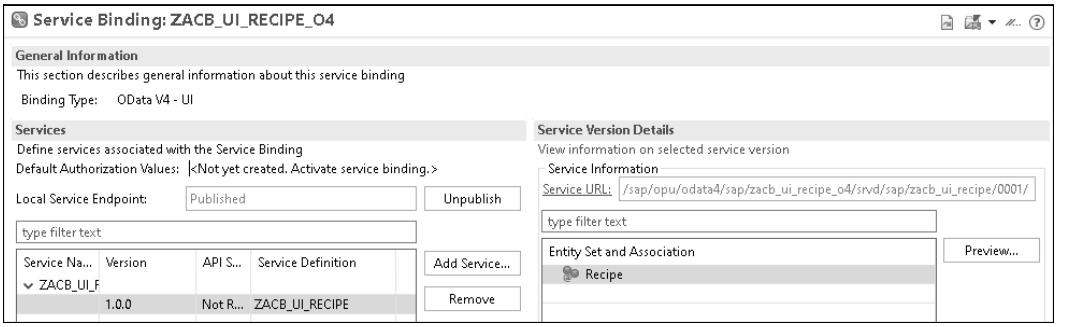



**Figure 2.21** Publication Status of the Service Endpoint in the Service Binding

**Preview of the SAP Fiori app**

Now select the appropriate entity (here, **Recipe**) in the **Service Binding** view and click on the **Preview** button to open the preview. You can see the UI of the generated SAP Fiori app in preview mode in Figure 2.22.


**Figure 2.22** Preview Mode of the SAP Fiori App

## 2.4 Creating Associations

**Adding more entities**

Before we look at the functions of the generated app from the user's perspective in the following section, we need to add the Ingredient and Review entities and link them to the Recipe root entity. In this section, we'll show you how to create an *association* for the root entity using the Ingredient entity. You can proceed in the same way for all other associated entities, in our example for the Review entity.

**Creating a data definition**

First, you must create a base entity for the associated entity. To do so, follow these steps:

1. Open the context menu of the **Data Definitions** folder (see Figure 2.23), and select the **New Data Definition** item.


**Figure 2.23** Context Menu Item: "New Data Definition"

2. In the dialog that opens, enter a name and description for the data definition. For our example, enter "ZACB_R_INGREDIENT" as the name. In the **Referenced Object** field, you must also enter a referenced object; in our example, this is the database table for the **Ingredient** object (see Figure 2.24). Then, click the **Next** button.


**Figure 2.24** Creating the Data Definition

3. In the subsequent step, select a transport request and confirm this by clicking the **Next** button.
4. Select a template. In our case, we want to create our own entity and therefore select **defineViewEntity** as the template (see Figure 2.25). This template generates the most important fields for the data definition.
5. Confirm the selection of the template by clicking the **Finish** button.

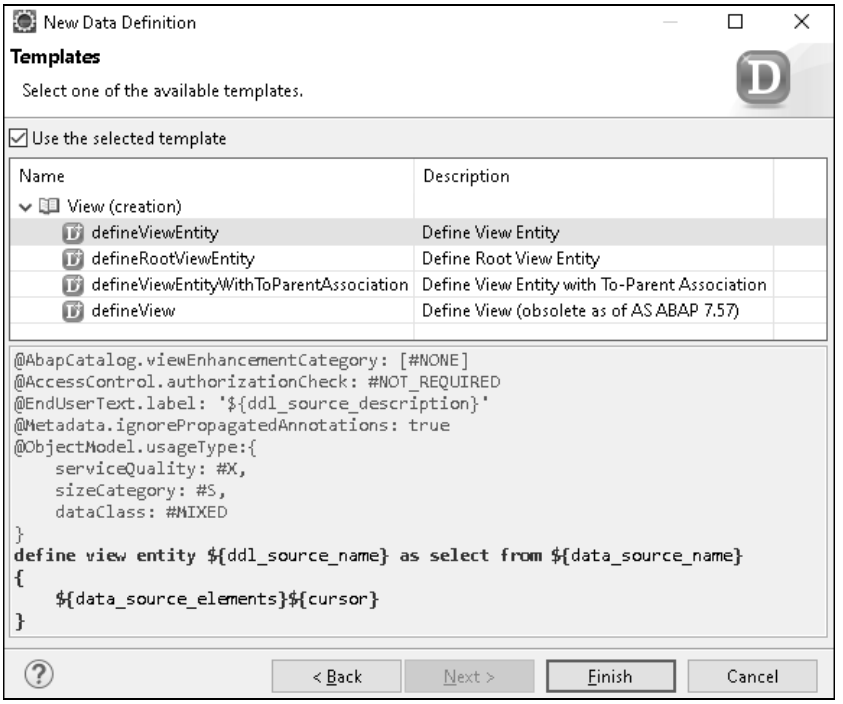



**Figure 2.25** Selecting a Template for the Data Definition

The CDS coding will then be generated from Listing 2.12 based on the template and the referenced object.

```
define view entity ZACB_R_Ingredient select from zacb_ingredient
{
    key recipe_id as RecipeId,
    key ingredient_id as IngredientId,
    name as Name,
    quantity as Quantity,
    unit as Unit,
    created_by as CreatedBy,
    created_at as CreatedAt,
    local_last_changed_by as LocalLastChangedBy,
    local_last_changed_at as LocalLastChangedAt,
    last_changed_at as LastChangedAt,
    last_changed_by as LastChangedBy
}
```

**Listing 2.12** Generated Base Entity ZACB_R_Ingredient

**Creating a projection entity**

Now you need to repeat steps 1–4 to create a projection entity. However, this time you don't select a template (see Figure 2.26), but click directly on the **Finish** button.

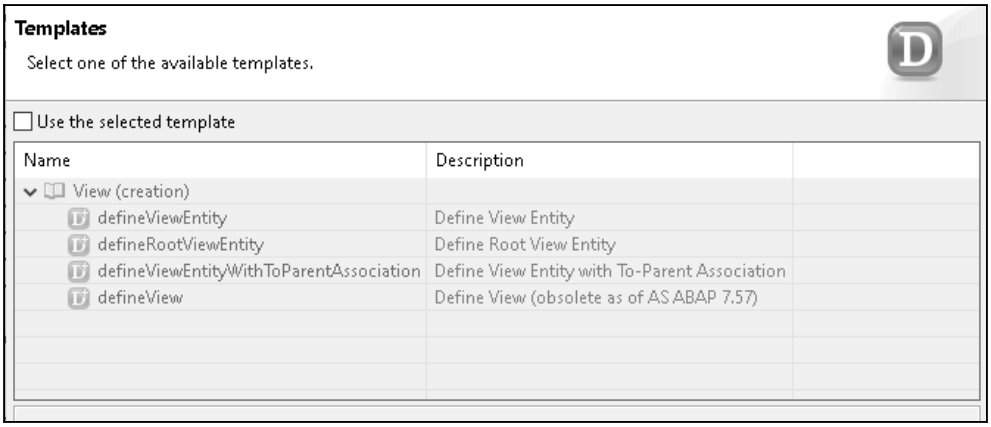



**Figure 2.26** Don't Select a Template for the Projection Entity

An empty data definition gets created. You can use the **defineProjectionView** pattern to trigger an automatic code generation. For this purpose, you should enter "defineProje..." in the empty editor window. The quick fix function of the ABAP development tools will then suggest the appropriate template (see Figure 2.27).

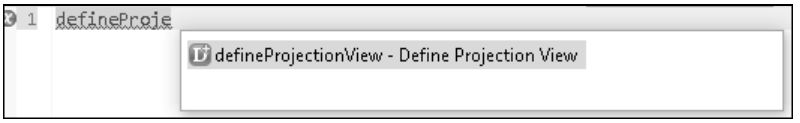



**Figure 2.27** Selecting a Code Template for a Projection View

When you select the template, the code will automatically be generated from Listing 2.13.

```
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Ingredient'
@Metadata.ignorePropagatedAnnotations: true
define view entity ZACB_C_Ingredient as projection on
  data_source_name
{

}
```

**Listing 2.13** Generated Code for the Projection View

**Adding entity names and fields**

Add the name of the base entity here as `data_source_name`. You can also specify the fields of the base entity and use `key` to define which fields are key fields (see Listing 2.14).

```
define view entity ZACB_C_Ingredient
  as projection on ZACB_R_Ingredient as Ingredient
{
  key RecipeId,
  key IngredientId,
      Name,
      Quantity,
      Unit,
      LocalLastChangedAt,
      LocalLastChangedBy,
      LastChangedAt,
      LastChangedBy,
}
```

**Listing 2.14** Completed Code of the Projection Entity

**Linking the base entities**

Now, the two base entities must first be linked to each other via an association. To do this, we first create a *composition* from the root entity in the direction of the linked entity (see Listing 2.15). A composition defines an existential dependency between two entities, whereby the child entity can't exist without the parent entity.

```
define root view entity ZACB_R_Recipe
  as select from zacb_recipe
  composition [0..*] of ZACB_R_Ingredient as _Ingredient
{
  key recipe_id             as RecipeId,
      recipe_name           as RecipeName,
…
```

**Listing 2.15** Specifying a Composition Link

At the level of the associated entity, you must use `association to parent` to add an association to the root entity. This creates an upward relationship with the root entity (see Listing 2.16).

```
@EndUserText.label: 'CDS entity Ingredient'
define view entity ZACB_R_Ingredient
  as select from zacb_ingredient
  association to parent ZACB_R_Recipe as _Recipe
    on $projection.RecipeId = _Recipe.RecipeID
{
  key recipe_id     as RecipeId,
  key ingredient_id as IngredientId,
…
```

**Listing 2.16** Adding "association to parent"

**Linking the projection entities**

A link between the two entities must also be created at the level of the projection entities. In the root entity, you must add the `redirected to composition child` expression (see Listing 2.17).

```
…
define root view entity ZACB_C_Recipe
  provider contract transactional_query
  as projection on ZACB_R_Recipe
{
  …
      _Ingredient : redirected to composition child
         ZACB_C_Ingredient,
}
```

**Listing 2.17** Adding the Link to the Child Entity

A similar link is made from the associated entity to the root entity. Here, you need to enter the `redirected to parent` addition (see Listing 2.18). The relationship defined in this way is used to navigate between the entities in the service binding.

```
…
define view entity ZACB_C_Ingredient
  as projection on ZACB_R_Ingredient as Ingredient
{
…
      _Recipe : redirected to parent ZACB_C_Recipe
}
```

**Listing 2.18** Redirected to Parent

**Extending the behavior definition**

You must define a behavior for each entity. To do this, you want to extend the behavior definition as shown in Listing 2.19:

- Associations are published in the entities.
- Data records of associated entities such as the ingredient are created via the root entity.
- The behavior of locks and authorizations takes place via the root entity.

```
managed implementation in class zbp_acb_r_recipe unique;
strict ( 2 );
with draft;

define behavior for ZACB_R_Recipe alias Recipe
…
  association _Ingredient { create; with draft; }
…
define behavior for ZACB_R_Ingredient alias Ingredient
persistent table zacb_ingredient
lock dependent by _Recipe
authorization dependent by _Recipe
draft table zacb_ingredien_d
{
  update;
  delete;
  field ( readonly ) RecipeId;
  field ( readonly ) IngredientId;

  association _Recipe { with draft; }
  mapping for zacb_ingredient
    {
   …
    }
}
```

**Listing 2.19** Link in the Behavior Definition of the Base Entities

These associations must also be defined in the behavior definition of the projection entities (see Listing 2.20).

```
projection;
strict ( 2 );
use draft;
define behavior for ZACB_C_Recipe alias Recipe
…
  use association _Ingredient { create; with draft; }
…
define behavior for ZACB_C_Ingredient alias Ingredients
…
  use association _Recipe { with draft; }
```

**Listing 2.20** Link in the Behavior Definition of the Projection Entities

**Extending the metadata definition**

To ensure that the linked entities are displayed in the SAP Fiori app, both the metadata extension of the root entity must be extended and a separate metadata extension must be created for the child entities. A facet must be added to the metadata extension of the root entity (see Listing 2.21).

```
{
id          : 'controlSection',
type        : #LINEITEM_REFERENCE,
position    : 20,
targetElement: '_Ingredient'
},
```

**Listing 2.21** Metadata Extension of the Root Entity

The `targetElement` annotation is used to indicate that the metadata extension of the associated entity is supposed to be called and displayed in this section.

**Extending the service definition**

Finally, the association must be specified in the service definition (see Listing 2.22).

```
@EndUserText.label: 'Recipe'
define service ZACB_UI_RECIPE {
  expose ZACB_C_Recipe as Recipe;
  expose ZACB_C_Ingredient as Ingredient;
}
```

**Listing 2.22** Extending the Service Definition

Once all created and modified development artifacts have been successfully activated, the link between the two entities can be seen in the service binding (see Figure 2.28).

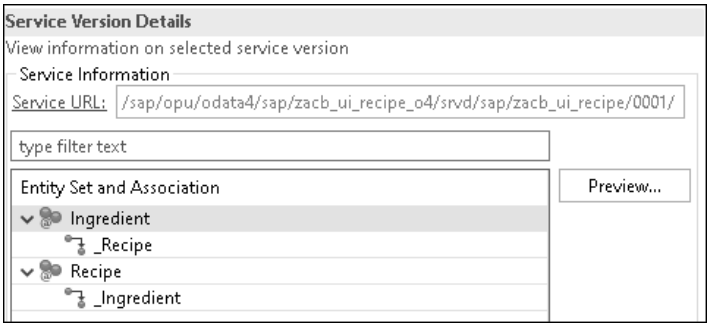



**Figure 2.28** Service Binding with Associations

Repeat the steps shown here for all associated entities.

## 2.5 Application Scenario from a User's Perspective

We now want to explain the basic functions of our SAP Fiori app. This app serves as the basis for all the functions we'll successively add to the application in this book. For this purpose, you should display the application using the **Preview** function of the service binding.

**List view of the recipes**

The recipe portal is displayed as an SAP Fiori app in a browser. The individual recipes are displayed in a list view. Our application doesn't yet contain any recipes (see Figure 2.29).


**Figure 2.29** Homepage of the Recipe Portal

The **Go** button is used to import all recipes in the recipe portal from the SAP backend and display them in the list (see Figure 2.30).

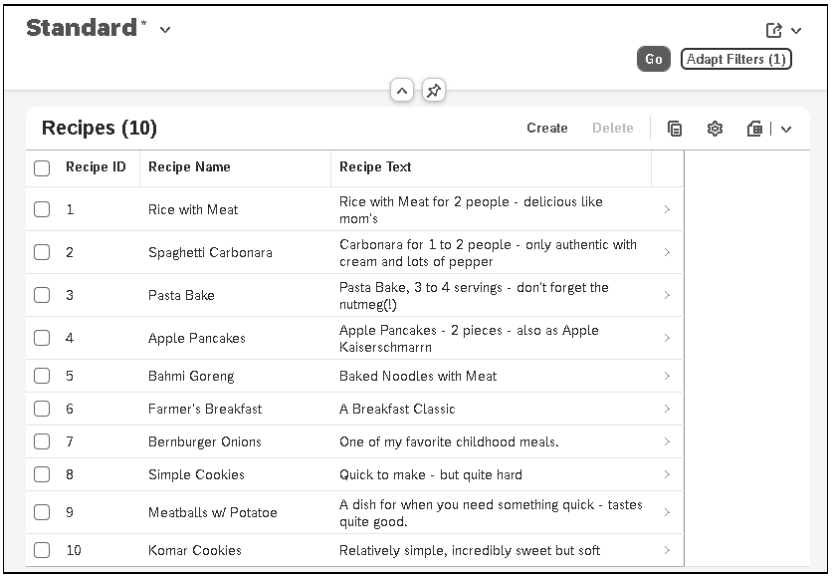



**Figure 2.30** Display of All Recipes in the SAP Fiori App

**Single view of a recipe**

If a specific recipe is selected, the user is taken to the individual view of this recipe (see Figure 2.31). Information such as the recipe name and the recipe text as well as all ingredients and reviews are displayed here in table format.

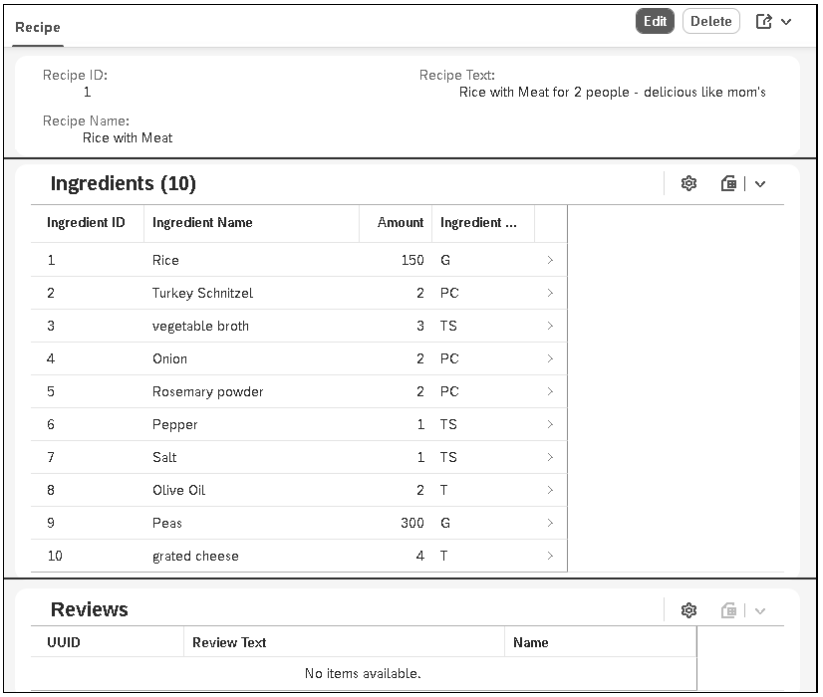



**Figure 2.31** Single View of a Recipe

**Adding a recipe**

To add a new recipe, the user must click on the **Create** button in the initial view (refer to Figure 2.30). A popup window appears in which a recipe ID of up to five digits must be entered (see Figure 2.32).

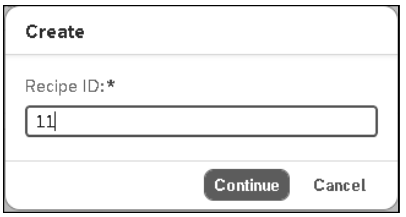



**Figure 2.32** Specifying a Recipe ID

A name (**Recipe Name** field) and preparation instructions (**Recipe Text** field) must then be entered (see Figure 2.33).


**Figure 2.33** Entering the Recipe Data

### Numbering

An explicit numbering by the user is required in the generated application. In Chapter 9, we show you how you can improve this using internal numbering.

**Adding an ingredient**

Next, you can add the ingredients for the recipe. To do this, click on the **Create** button. This opens the creation view for an ingredient (see Figure 2.34). There you can enter the required information for the ingredient, such as the name, quantity, and unit.

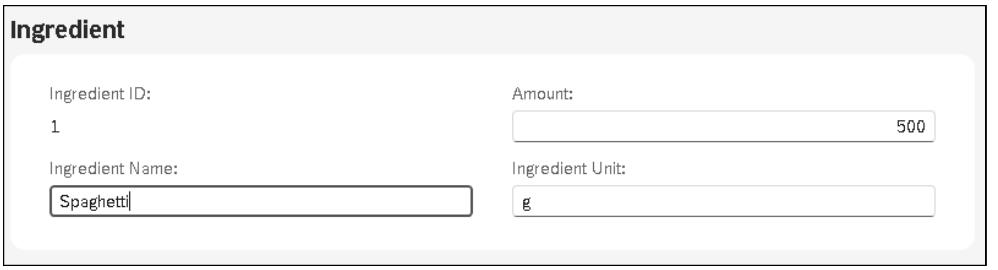



**Figure 2.34** Entering an Ingredient

After confirming your entries by clicking the **Apply** button, the ingredient will be added to the recipe but not yet saved in the database. The data gets saved when you click on the **Create** button in the **Recipe** view (see Figure 2.35).

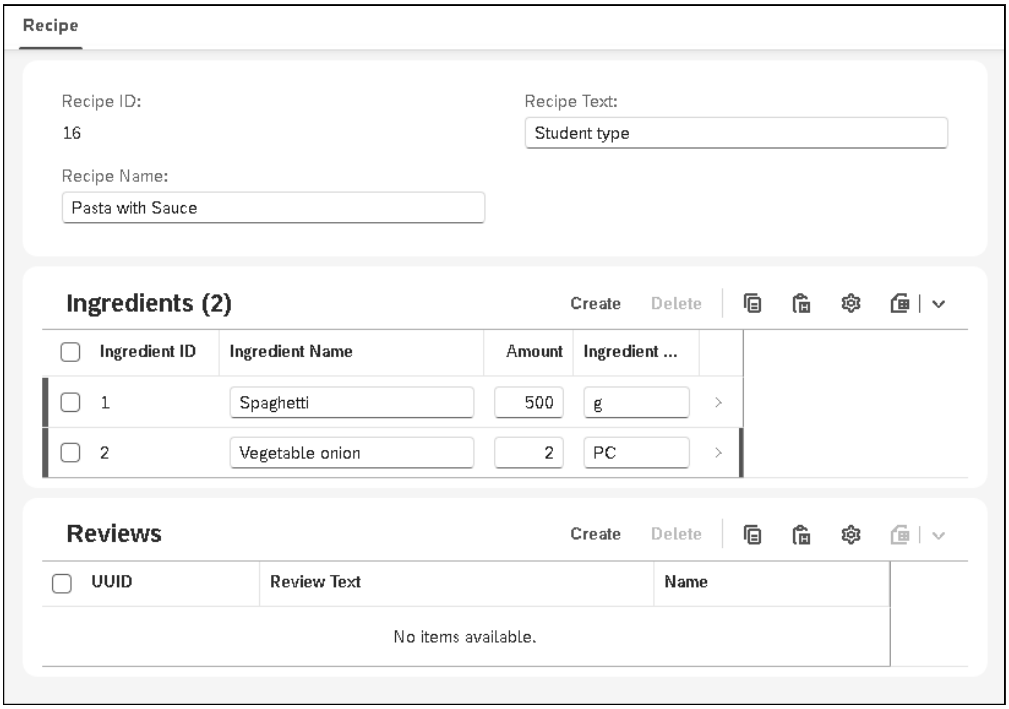


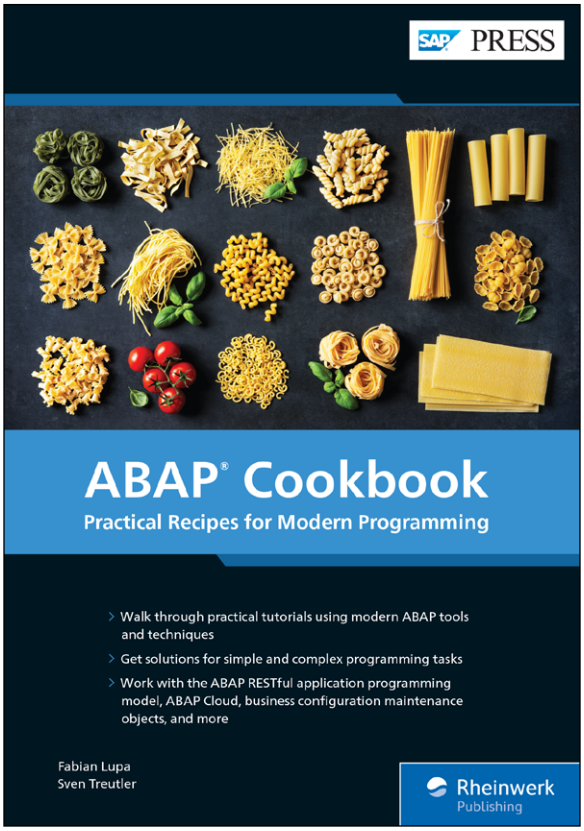

**Figure 2.35** Recipe View with Details of All Ingredients

## 2.6 Summary

In this chapter, we've presented the sample application of a recipe portal that we'll use for the examples in this book. We derived the technical concepts of the application from the use case. The repository objects domain, data element, and database table were then created.

Based on the database table for the recipes, we generated an ABAP RESTful application programming model application and then described it step-by-step. Among other things, we discussed the difference between CDS base entities and projection entities. We've also introduced important components of the ABAP RESTful application programming model such as the behavior and metadata definition, the service definition, and the service binding.

For a recipe portal, it doesn't suffice to enter recipe names and recipe texts because it must also be possible to maintain ingredients. To do this, we've manually extended the automatically generated application. It's now also possible to add reviews to a recipe. We've shown you how to build relationships between multiple associated CDS entities and their behavior definitions.




Fabian Lupa, Sven Treutler

# ABAP Cookbook

## Practical Recipes for Modern Programming

- Walk through practical tutorials using modern ABAP tools and techniques
- Get solutions for simple and complex programming tasks
- Work with ABAP RESTful application programming model, ABAP Cloud, business configuration maintenance objects, and more

**www.sap-press.com/6198**



Fabian Lupa is a senior software engineer and trainer at adesso, where he's responsible for employee training and ABAP development. Sven Treutler is an ABAP developer at rku.it GmbH, where he focuses on new technologies and quality assurance in the ABAP environment.

**ISBN 978-1-4932-2777-8 • 444 pages • 12/2025**

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

Rheinwerk Publishing