

ABAP® RESTful Application Programming Model

The Comprehensive Guide

- › 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

Lutz Baumbusch
Matthias Jäger
Michael Lensch



Rheinwerk
Publishing

Imprint

This e-book is a publication many contributed to, specifically:

Editor Hareem Shafi

Acquisitions Editor Hareem Shafi

German Edition Editor Janina Schweitzer

Translation Lemoine International, Inc.

Copyeditor Tanya Thamkruphat

Cover Design Bastian Illerhaus

Photo Credit Shutterstock: 295387826/@somsak nitimongkolchai

Layout Design Vera Brauner

Production E-Book Kyrsten Coleman

Typesetting E-Book SatzPro, Germany

We hope that you liked this e-book. Please share your feedback with us at the e-mail address: support@rheinwerk-publishing.com.

The Library of Congress Cataloging-in-Publication Control Number for the printed edition is as follows: 2022945836

© 2025 by:

Rheinwerk Publishing, Inc.

2 Heritage Drive, Suite 305

Quincy, MA 02171

USA

info@rheinwerk-publishing.com

+1.781.228.5070

Represented in the E.U. by:

Rheinwerk Verlag GmbH

Rheinwerkallee 4

53227 Bonn

Germany

service@rheinwerk-verlag.de

+49 (0) 228 42150-0

ISBN 978-1-4932-2379-4 (print)

ISBN 978-1-4932-2380-0 (e-book)

ISBN 978-1-4932-2381-7 (print and e-book)

1st edition 2023

1st German edition published 2022 by Rheinwerk Verlag

Chapter 10

Managed Scenario with Unmanaged Save: Integrating an Existing Application

In this chapter, we'll develop a RAP application using an existing standard SAP application with its data model and API. For this purpose, we'll use the managed scenario with unmanaged save. The focus is on the development of the save sequence with late numbering.

In Chapter 9, the goal was to build a standalone application using the ABAP RESTful application programming model. This application was not based on an existing application, but it had its own data model (greenfield approach). In this chapter, we'll show you how to build a RAP application that's based on the data model of an existing application and integrates it using its API. To integrate the API, you need to implement the `SAVE` phase of the save sequence and late numbering yourself.

The chapter starts with the description of the use case in Section 10.1. We'll then build the data model in Section 10.2 to create a behavior definition based on it in Section 10.3. In Section 10.4, we'll implement the create purchase order function and, in Section 10.5, we'll implement the delete purchase order function of the sample application. After that, we'll continue with Section 10.6 to expose the previously implemented function via the OData protocol. In Section 10.7, we'll implement authorization checks for read and write accesses. Finally, we'll create an SAP Fiori elements UI for the application in Section 10.8.

10.1 Description of the Use Case

This section describes the use case we'll implement in this chapter using the ABAP RESTful application programming model and its business requirements. Based on the technical requirements and the general conditions, we'll develop a suitable solution strategy and implement it step-by-step.

Fast entry of a purchase order We'll develop an application that provides the fast entry of a purchase order. Users should be able to order certain materials (e.g., a SAP PRESS book) on their own without having to know the details of the order creation process or having to contact their company's purchasing department. The application will be based on the purchasing module functionality of the MM-PUR application component, which is available in SAP S/4HANA.

- Functional requirements** The following list comprises the main functional requirements of the application:
- A purchase order can be created by entering a material and the order quantity.
 - In the fast entry mode, purchase orders can have only one item at a time.
 - Only materials approved by the purchasing department (valid products) can be ordered.
 - Materials that are released for ordering are stored in a database table for this purpose.
 - Purchase orders for materials that aren't valid can't be created.
 - The user sees only his or her own purchase orders.
 - Purchase orders can be deleted again.
 - Deleted purchase orders aren't displayed in the user interface of the application.

The application is implemented as SAP Fiori elements list report. In Figure 10.1, you can see the finished application.

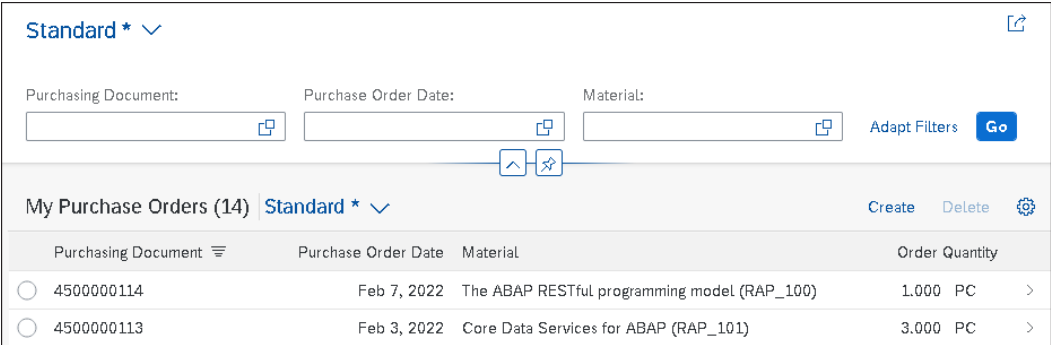


Figure 10.1 List Report for Fast Entry of Purchase Orders

Basic conditions An essential technical condition is the runtime environment of the application. It determines the technical and functional scope and general conditions or restrictions under which you can develop and operate software. The development of this application is for SAP S/4HANA 2021 FPS 1 (on-

premise). In the following sections, we'll elaborate on the impact of this essential condition on the design of the application.

Warning: Release-Dependent Implementation

Late numbering is included in SAP S/4HANA 2021 FPS 1 as an early development (see SAP Note 3060272). This means you can carry out the development with this release in the managed scenario with unmanaged save. If you have a lower release level, you must make the implementation using the unmanaged scenario.

Since the ABAP RESTful application programming model is also available in the cloud, see Chapter 12 for details on the technical environment for cloud applications.



In order to implement the business requirements, you first need a suitable public interface (API) for read and write access to purchase orders in SAP S/4HANA. For read access, you can use the CDS views of the virtual data model (VDM) provided in the standard SAP system.

APIs used

Business application programming interfaces (BAPIs) are also officially released APIs under SAP S/4HANA. You can therefore use the BAPI `BAPI_PO_CREATE1` to create a purchase order. The BAPI `BAPI_PO_CHANGE` enables you to semantically delete a purchase order by specifying a deletion indicator. Both of the function modules perform persistent database changes (i.e., not only changes in the memory during the interaction phase) and bundle them in a database logical unit of work (LUW). They must therefore be called in the save sequence.

To avoid duplicate data storage, our application doesn't implement its own persistence of purchase orders. Consequently, the purchase orders created in the standard SAP system represent the sole data source for the application. Alternatively, you could also implement persistence yourself so that purchase orders that are entered quickly are kept separately. This could be advantageous if, for example, the criteria for such purchase orders were complex to evaluate, could change foreseeably, or if additional fields could be necessary for such a process, and aren't included in the standard purchase order business object. In our example, however, this isn't the case. In Section 10.2.3, we'll describe in more detail the data model used here.

Persistence

Numbering is done by the API that's used, which is included in the save sequence (i.e., by the BAPI `BAPI_PO_CREATE1`). You must use late numbering for this. Details on the implementation for our application can be found in Section 10.4.1.

Numbering

- Validations

The BAPIs `BAPI_PO_CREATE1` and `BAPI_PO_CHANGE` have a test or simulation mode. You can use this mode to check the data in the transaction buffer and, based on the check results, abort the save sequence if necessary. You include these checks via validations within the RAP transaction (Section 10.4.6 and Section 10.5.2).
- Transaction or update control

Because you’re using BAPIs to create and change the purchase order, transaction control lies with the BAPI consumer (`BAPI_TRANSACTION_COMMIT` or `BAPI_TRANSACTION_ROLLBACK`). Here, the changes made will either be posted to the database or the LUW will be reset. The ABAP RESTful application programming model provides for transaction control, as with BAPIs, on the part of the consumer of the business object (i.e., either the EML consumer (via `COMMIT ENTITIES`) or the orchestration framework). So, you can safely use the mentioned BAPIs in the save sequence. The RAP framework determines the transaction boundaries. A `COMMIT ENTITIES` results in a `COMMIT WORK` being performed as well (which is behind the `BAPI_TRANSACTION_COMMIT`), so you don't have to call a `BAPI_TRANSACTION_COMMIT` yourself, nor are you allowed to do so in the behavior pool of a RAP business object. However, in other projects, you should note that there are exceptions among BAPIs that perform transaction control themselves. The integration of the selected BAPIs within the save sequence is shown in Figure 10.2.

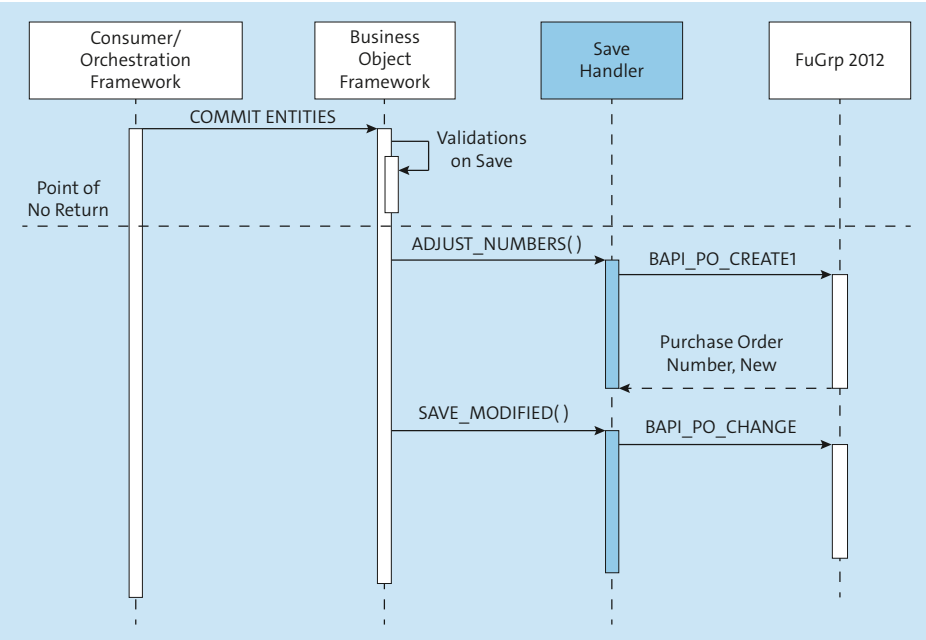


Figure 10.2 BAPI Call in the Save Sequence

You can use CDS entities of the VDM for read access. This allows you to define your own CDS access controls based on the CDS access controls defined in the VDM (Section 10.7). For write access, you can query authorization objects that use the BAPIs in the RAP business object.

Authorizations

Since users only need to enter a small amount of data when creating a purchase order, we can do without draft handling in this example, and the application status can be kept entirely on the client side (that is, typically in the SAPUI5 application within the web browser).

Draft handling

10.2 Building the Data Model

In the following sections, we'll describe the CDS data model for our sample application for the fast entry of a purchase order and show you how to create it in ABAP development tools (ADT). We'll start with an overview of the logical data model and explain which database tables are necessary in the ABAP dictionary area.

10.2.1 Overview of the Logical Data Model

In Figure 10.3, you can see the logical data model of the application as a Unified Modeling Language (UML) class diagram. The logical data model is initially independent of the way it's mapped in the repository using the ABAP dictionary or core data services.

Logical data model

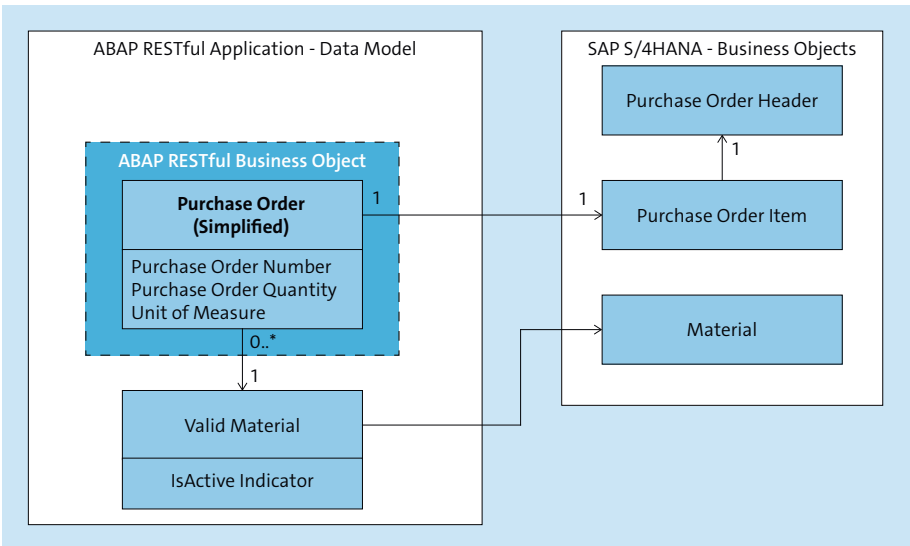


Figure 10.3 Logical Data Model of the Application

The UML class “purchase order (simplified)” is central and represents a purchase order whose counterpart in the standard SAP system is the purchase order item. It is simplified because it’s associated with only *one* instance of a purchase order item. As a result, the essential attributes of the class originate from both the purchase order header (e.g., the purchase order number) and the purchase order item (e.g., the purchase order quantity and the unit of measure). You’ll learn more about the other attributes in Section 10.2.3.

The UML class “purchase order (simplified)” is associated with the UML class “valid material” (i.e., the products that may be procured via the simplified purchase order). Procurement is only possible if the active indicator (attribute “IsActive”) is set for the “valid material.”

CDS data model In Figure 10.4, you can see how the data model of the application is mapped with the development objects of the ABAP dictionary and CDS. On the right, there are the VDM’s CDM entities, on which the application is based, to implement the desired functional requirements.

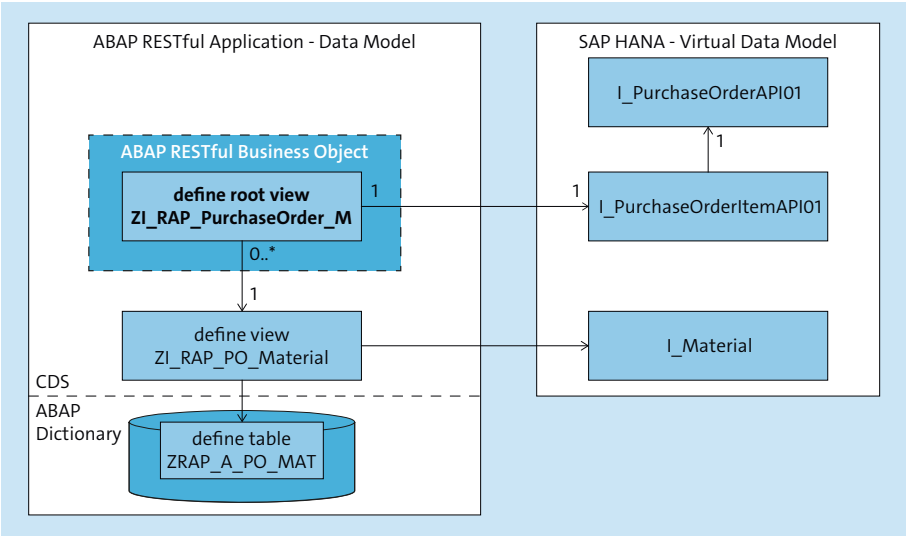


Figure 10.4 CDS Data Model of the Application

CDS root entity for the purchase order The functional requirements require read and write access to the standard SAP business object of the purchase order. Since we a) don’t need all the fields in the purchase order, b) the purchase orders placed through the application have only one internal purchase order line item, and c) we need transactional behavior for the purchase order (create and delete), we must model a separate CDS root entity named ZI_RAP_PurchaseOrder_M to represent the simplified purchase order. This is the central entity of the application. It’s marked as *root* so that you can later add behavior via a behavior definition and use it to define a RAP business object. So you should keep in

mind that, during CDS modeling, you want to implement transactional behavior and you will need a behavior definition to do so.

There's no composition tree for the simplified purchase order in this model (i.e., it has no child entities). Since the RAP part of the application doesn't have its own persistence, the simplified purchase order uses the standard SAP purchase order as a data source, accessing the CDS entity `I_PurchaseOrderItemAPI01` in read-only mode or the order header `I_PurchaseOrderAPI01` from there.

Structure of the RAP business object

In addition, the CDS entity `ZI_RAP_PurchaseOrder_M` is associated with the first purchase order item, which is determined by a private CDS view. For clarity, we have omitted the presentation of this CDS entity in Figure 10.4.

First purchase order item

The valid materials are stored in a separate database table named `ZRAP_A_PO_MAT`, which is mapped at the logical level via the CDS entity `ZI_RAP_PO_Material`. The simplified purchase order `ZI_RAP_PurchaseOrder_M` is associated with the material that's valid for fast entry `ZI_RAP_PO_Material`. The valid material in turn “knows” the material master `I_Material` by association.

Valid materials

Alternative Mapping of the Material Master

Instead of the CDS entity `I_Material`, you can also use the CDS entity `I_Product` to access material master data. `I_Product` has also been released for SAP S/4HANA Cloud (see https://api.sap.com/cdsvIEWS/I_PRODUCT). We've used the `I_Material` entity in the data model of our application because the business object “Material” is well known from classic ABAP development for SAP ERP.



Table 10.1 lists all CDS entities of the VDM that you'll use in this application.

CDS Entity	Meaning
<code>I_PurchaseOrderAPI01</code>	Purchase order header
<code>I_PurchaseOrderItemAPI01</code>	Purchase order item
<code>I_Material</code>	Material master
<code>I_Supplier</code>	Supplier

Table 10.1 Standard CDS Entities for Using the Fast Entry of Purchase Orders

10.2.2 Database Tables

As described, we want to allow the simplified entry of a purchase order only for certain materials. These materials are supposed to be offered to the

users via a search help in the user interface. In the implementation of the RAP business object, the system checks whether the materials are stored as released.

Structure of the database table Our application keeps these materials in a separate database table: ZRAP_A_PO_MAT. The structure of this database table is shown in Table 10.2.

Field Name	Meaning	Key?
client	Client	Yes
material	Material number	Yes
supplier	Supplier from whom the material is ordered	No
is_active	Active indicator; if it's set, the material can be ordered	No
created_by	Created by	No
created_at	Time of creation	No
last_changed_by	Last change by	No
last_changed_at	Time of last change	No

Table 10.2 Structure of Database Table ZRAP_A_PO_MAT

Creating a database table

- You can create the database table ZRAP_A_PO_MAT in ADT:
1. Select the package of your application in the **Project Explorer** and use the shortcut `[Ctrl] + [N]` to start the creation wizard.
 2. In the dialog that opens, you must select the **Database Table** entry under **ABAP • Dictionary**.
 3. Assign a name for the database table under **Name** (here it would be “ZRAP_A_PO_MAT”) and enter a description in the **Description** field.
 4. Click the **Next** button and, after selecting the transport request, finish the process by clicking the **Finish** button.



Naming Convention for Database Tables

In the ABAP RESTful application programming model, the prefix `_A` is used as a naming convention to distinguish database tables for active application data from database tables for draft instances (prefix `_D`).

5. Modify the generated source code to define the database table as shown in Listing 10.1.

```

@EndUserText.label : 'Valid materials for fast entry'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table zrap_a_po_mat {
  key client      : abap.clnt not null;
  key material    : matnr not null;
  supplier        : lifnr;
  is_active       : zrap_po_mat_active;
  created_by      : abp_creation_user;
  created_at      : abp_creation_tstmpl;
  last_changed_by : abp_lastchange_user;
  last_changed_at : abp_lastchange_tstmpl;
}

```

Listing 10.1 Source Code for the ZRAP_A_PO_MAT Database Table

6. Then, activate the database table. We provide the definition of the data element ZRAP_PO_MAT_ACTIVE, which is used to type the `is_active` column, with the download material for this book at www.sap-press.com/5642.

10.2.3 CDS Modeling

This section describes the development of the CDS-based data model. We'll move from the bottom up along the dependency relationships between CDS entities. That is, first we'll create the CDS view for wrapping the database table of the valid material and then we'll model the simplified order.

Valid Material

We want to logically map the database table ZRAP_A_PO_MAT in a CDS entity. This entity represents the valid material for fast order entry on a semantic level.

**CDS view for
material table**

1. To do this, you need to create a CDS view named `ZI_RAP_PO_Material` based on the database table ZRAP_A_PO_MAT. So, you must use the creation wizard again. Under **ABAP • Core Data Services**, select **Data Definition**.
2. Then fill in the dialog shown in Figure 10.5 to create the CDS entity. The package name (**Package** input field) may differ in your case from what we've used here.

Figure 10.5 Creating the ZI_RAP_PO_Material CDS View

3. Use the **Define View Entity** template and finish the process by clicking the **Finish** button.

The data definition language (DDL) source code editor opens with the source code generated based on the selected template.



Tip: Applying Pretty Printer

You should perform a formatting of the source code using the shortcut

Ctrl + **Shift** + **F**.

Maintaining associations with the VDM

Add the required associations to the CDS entities of the VDM to the source code as shown in Listing 10.2. To do this, you need to define an association to the material (association [0..1] to I_Material) and one to the supplier (association [0..1] to I_Supplier).

```
@AbapCatalog.viewEnhancementCategory: [#NONE]
@AccessControl.authorizationCheck: #CHECK
@endUserText.label: 'Gültiges Material für Schnellerfassung'
@Metadata.ignorePropagatedAnnotations: true
@ObjectModel.usageType:{
    serviceQuality: #X,
    sizeCategory: #S,
    dataClass: #MIXED
}
define view entity ZI_RAP_PO_Material
    as select from zrap_a_po_mat
```

```

association [0..1] to I_Material as _Material on
  _Material.Material = $projection.Material
association [0..1] to I_Supplier as _Supplier on
  _Supplier.Supplier = $projection.Supplier
{
  key material      as Material,
    supplier        as Supplier,
    is_active        as IsActive,
    created_by       as CreatedBy,
    created_at       as CreatedAt,
    last_changed_by  as LastChangedBy,
    last_changed_at  as LastChangedAt,
    _Material,
    _Supplier
}

```

Listing 10.2 CDS Entity for the Valid Material

Help View for Determining the First Purchase Order Item

In the next step, we need a CDS view that determines the number of the first purchase order item of each order. To do this, you must create the CDS view `ZP_RAP_PurchaseOrderItemCount`. By using the name prefix `P_` you communicate that this is a private CDS entity and therefore an implementation detail of the application. You can see the DDL source code of this view in Listing 10.3.

Determining the number of the first purchase order item

```

@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Help view for first purchase order item'
define view entity ZP_RAP_PurchaseOrderItemCount
  as select from I_PurchaseOrderItemAPI01
  {
    PurchaseOrder,
    min( PurchaseOrderItem ) as FirstPurchaseOrderItem
  }
group by
  PurchaseOrder
having
  count(*) = 1

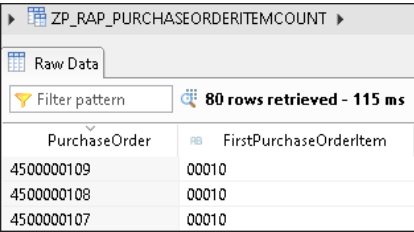
```

Listing 10.3 Help View for Determining the First Item Number

The view is based on CDS entity `I_PurchaseOrderItemAPI01` to select the purchase order items stored in the standard SAP system. What's crucial to this view is the bold highlighted line with the `min` aggregate expression to determine the lowest and therefore first item number of the order. The

FirstPurchaseOrderItem attribute contains this number. The aggregation expression `count(*) = 1` in the having clause determines the number of purchase order items per purchase order number (group by PurchaseOrder clause) and selects only purchase orders that have an item.

Testing the CDS view The editor of the CDS view enables you to execute the view via the function key `[F8]` and thus test the data selection. Alternatively, you can use the context menu of the CDS view and select the **Open With • Data Preview** menu item. After running the CDS view, the data preview opens in a new tab where the result set is displayed (see Figure 10.6).



PurchaseOrder	FirstPurchaseOrderItem
4500000109	00010
4500000108	00010
4500000107	00010

Figure 10.6 Result Set for the ZP_RAP_PurchaseOrderItemCount View in Data Preview

Purchase Order with a Purchase Order Item

CDS root entity Next, you can create the CDS entity `ZI_RAP_PurchaseOrder_M`. You should proceed in the same way as for the other CDS entities, but use the **Define Root View Entity** template. In Listing 10.4, you can see the declaration of the CDS root entity for our RAP business model with required join conditions.

```
@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Purchase order with one item'
define root view entity ZI_RAP_PurchaseOrder_M
  as select from I_PurchaseOrderItemAPI01 as item
    inner join ZP_RAP_PurchaseOrderItemCount as item_cnt
      on item.PurchaseOrder = item_cnt.PurchaseOrder
  and item.PurchaseOrderItem = item_cnt.FirstPurchaseOrderItem
    inner join ZI_RAP_PO_Material as po_mat
      on po_mat.Material = item.Material
  ...
```

Listing 10.4 CDS Root Entity of the Simplified Purchase Order with Join Conditions

Join conditions The CDS entity is based on the purchase order item `I_PurchaseOrderItemAPI01` of the standard SAP system. The inner join to the previously

created help view `ZP_RAP_PurchaseOrderItemCount` creates the link to the first purchase order item, while the inner join to the `ZI_RAP_PO_Material` view creates the link to the materials valid for the purchase order process. In this way, you select only first purchase order items that contain a material stored as valid. You don't evaluate the active indicator. This means that all purchase orders already placed with this material number will be selected, even if they have been set to inactive in the meantime.

Naming the CDS Entities of a RAP Business Object

The transaction behavior for the fast entry purchase order is represented through the behavior definition for the CDS root entity `ZI_RAP_PurchaseOrder_M`, which we'll create in Section 10.3. We'll choose the implementation type managed with unmanaged save. CDS entities of a RAP business object, whose runtime is managed (at least their interaction phase), get the suffix `_M`, according to the naming convention.



A series of associations follows in Listing 10.5 to establish the logical link to the purchase order item (`I_PurchaseOrderItemAPI01`), to the purchase order (`I_PurchaseOrderAPI01`), and to the ordered valid material (`ZI_RAP_PO_Material`).

Associations

```
association [1] to I_PurchaseOrderAPI01 as _PurchaseOrder on
    _PurchaseOrder.PurchaseOrder = item.PurchaseOrder
association [1] to I_PurchaseOrderItemAPI01 as
    _PurchaseOrderItem on _PurchaseOrderItem.PurchaseOrder =
    $projection.PurchaseOrder and
    _PurchaseOrderItem.PurchaseOrderItem =
    $projection.PurchaseOrderItem
association [1] to ZI_RAP_PO_Material as _POMaterial on
    _POMaterial.Material = $projection.Material
{
...
}
```

Listing 10.5 Associations within `ZI_RAP_PurchaseOrder_M` View Definition

Now, the question arises regarding which fields of the CDS entity are relevant for the respective use case (i.e., which fields are to be accessed from *outside* the business object in read and write mode within the `create` operation). The CDS entity `ZI_RAP_PurchaseOrder_M` requires the attributes listed in Table 10.3 from the business object of the purchase order so that the described requirements can be implemented (administrative fields are not shown).

Determining relevant fields

Attribute	Meaning (Origin)	Type of Operation
key PurchaseOrder	Purchase order number (purchase order header); key field	Read
PurchaseOrderType	Purchase order type (purchase order header)	Read
PurchaseOrderItem	Item number (purchase order item)	Read
PurchaseOrderDate	Purchase order date (purchase order header)	Read
PurchasingOrganization	Purchasing organization (purchase order header)	Read
PurchasingGroup	Purchasing group (purchase order header)	Read
Plant	Plant (purchase order item)	Read
Supplier	Supplier number (purchase order header)	Read
Materials	Material number (purchase order header)	Read and write
OrderQuantity	Purchase order quantity	Read and write
PurchaseOrderQuantityUnit	Unit of measure	Read
LastChangeDateTime	Timestamp of the last change; also filled during creation	Read

Table 10.3 Relevant Fields of ZI_RAP_PurchaseOrder_M CDS Entity

Denormalizing the CDS entity There’s only one CDS entity because we assume that the created purchase order has only one purchase order item. So, both fields from the purchase order item and those from the purchase order header data are found on one level in CDS view ZI_RAP_PurchaseOrder_M. From the 1-to-n relationship between purchase order and purchase order item, here you declare a 1-to-1 relationship. Thus, the data model is denormalized at this point.

Field declarations Now you need to declare the relevant fields in the selection list of the CDS entity. You can see the completion of the DDL source code with the selection list for field declaration (and the publication of the previously declared

associations) in Listing 10.6. Note that due to the `where` clause, you only see purchase order items that haven't been deleted (`item.PurchasingDocumentDeletionCode <> 'L'`).

```
...
{
  key item.PurchaseOrder,
    _PurchaseOrder.PurchaseOrderType,
    _PurchaseOrder.PurchaseOrderDate,
    _PurchaseOrder.PurchasingOrganization,
    _PurchaseOrder.PurchasingGroup,
    _PurchaseOrder.Supplier,
    _PurchaseOrder.CreatedByUser,
    _PurchaseOrder.CreationDate,
    _PurchaseOrder.LastChangeDateTime,
    item.PurchaseOrderItem,
    item.Material,
    item.OrderQuantity,
    item.PurchaseOrderQuantityUnit,
    item.Plant,
    _PurchaseOrder,
    _PurchaseOrderItem,
    _POMaterial
}
where
  item.PurchasingDocumentDeletionCode <> 'L'
```

Listing 10.6 Selection List and Final `where` Clause

Save and activate the class. The declaration of the key fields with `key` is essential for a RAP business object and the CDS entities of its composition tree. In our example, there's only one key field, `PurchaseOrder`, because our business object is a simplified purchase order. Since the separation between header and item data has been deliberately removed, the order number is sufficient as the sole key field. It's semantically correct and does not need to be supplemented by the number of the purchase order item.

Key field

10.3 Creating a Behavior Definition

In the previous section, you mapped the data model of the application using the ABAP dictionary and CDS views in the repository. It is thus already prepared in such a way that you can map the desired transactional

behavior. First of all, this involves basic declarations in the behavior definition before you can define the create purchase order and delete purchase order functions.

Choosing the implementation type

Before you create the behavior definition, you should choose the implementation type. Because we decided to use BAPIs for the purchase order, we don't need our own business logic during the interaction phase and we don't need to integrate an inventory API there. Accordingly, the interaction phase can be implemented by the managed business object provider. We can complement determinations and validations anyway.

Since the RAP business object doesn't have its own persistence, we don't need a fully managed business object that also implements the save sequence. On the contrary: We want to implement the save sequence ourselves to implement database changes through the BAPI calls. For these reasons, we opt for a managed with unmanaged save scenario. The interaction phase is thus managed, whereas the save sequence isn't. The latter can be implemented by using it in the behavior pool.

Creating the behavior definition

You can create a behavior definition for CDS root entity `ZI_RAP_PurchaseOrder_M` as follows:

1. Open the context menu for CDS entity `ZI_RAP_PurchaseOrder_M` and select the menu item **New Behavior Definition** there.
2. In the dialog that opens, keep the default setting **managed** for the value of the **Implementation Type** field and complete the creation process by clicking the **Finish** button.
3. Then, the editor with the proposed behavior definition language (BDL) source code will open.

Customizing the behavior definition

Customize the BDL source code as shown in Listing 10.7. Declare the save option with unmanaged save. This eliminates the need to specify a database table via persistent table, so you can remove the corresponding line from the generated source code. Assign the `PurchaseOrder` alias for the entity behavior definition. You can then use this descriptive name in the behavior pool.

```
managed with unmanaged save; // implementation in class zbp_i_rap_purchaseorder_m unique;
```

```
define behavior for ZI_RAP_PURCHASEORDER_M alias PurchaseOrder
lock master
//authorization master ( instance )
etag master LastChangeDateTime
{
    create;
```

```

    internal update;
    delete;
}

```

Listing 10.7 Basic Declarations in the BDL Source Code

Remove the comment in the lock declaration line in the entity behavior definition so that it is declared as `lock master`. Then, you should add `internal` to the update operation because the RAP business object only provides the standard create and delete operations externally, even if the deletion operation is implemented internally via an update of the purchase order (via the BAPI `BAPI_PO_CHANGE`). You need the update operation for changing instances within the behavior implementation.

As `etag master`, you should name the `LastChangeDateTime` field because the RAP business object also provides for a delete operation that might be performed on an outdated dataset. Finally, save and activate the behavior definition.

10.4 Implementing the Create Purchase Order Function

In this section, you'll implement the function for creating a purchase order. You have already taken a first step towards this, because the standard `create` operation has already been declared in the context of the behavior definition in the previous section.

10.4.1 Declaring Late Numbering

In the context of the `create` operation of a CDS entity of a RAP business object, you must always consider numbering and choose one of the options supported by the ABAP RESTful application programming model (see Chapter 3, Section 3.7).

Since in our example the instance of a purchase order is created, you need to consider in which way the numbering of the purchase order should be carried out. In the CDS data model, we've declared the order number `PurchaseOrder` as key. The purchase order numbering is done by the BAPI (we don't use a purchase order process in MM-PUR with external numbering here) and is therefore beyond our control. Thus, the number of the created purchase order will become available to us in the save sequence after calling the BAPI `BAPI_PO_CREATE1`. A numbering during the save sequence is supported by the ABAP RESTful application programming model via the late numbering option.

Selecting the numbering option

Declaring late numbering Provide the CDS entity `ZI_RAP_PurchaseOrder_M` with the late numbering function in the transactional properties of the entity behavior definition via the BDL keyword `late numbering` (see Listing 10.8).

```
managed with unmanaged save; // implementation in class zbp_i_rap_
purchaseorder_m unique;

define behavior for ZI_RAP_PURCHASEORDER_M alias PurchaseOrder
lock master
late numbering
//authorization master ( instance )
...
```

Listing 10.8 Declaring Late Numbering

This allows you to implement late numbering using the `ADJUST_NUMBERS` method in the save handler, which we'll implement in Section 10.4.5.

10.4.2 Setting Field Properties

If a behavior definition is created for a CDS data model, by default all fields of the contained CDS entities are also open for write accesses. However, this is often not desired. You should consider for each field whether it may be open for write access or not.



Tip: Lean Interface

Ensure that the interface of the business object and its operations is as lean as possible and therefore only allow write access to those fields for which it is actually necessary.

In relation to our use case of purchase order entry, this means that the `Material` and `OrderQuantity` fields must remain open for write access. For the remaining fields, only read access is allowed (see also Table 10.3). The `PurchaseOrderQuantityUnit` is determined from the base unit of measure of the material.

**Fields for
read access**

Thus, you need to provide the fields of CDS entity `ZI_RAP_PurchaseOrder_M` with the `field (readonly) ...` property so that they are protected against write access from outside the business object (see Listing 10.9).

```

...
{
    create;
    internal update;
    delete;

    field ( readonly )
        PurchaseOrder,
        PurchaseOrderDate,
        PurchasingOrganization,
        PurchasingGroup,
        Supplier,
        PurchaseOrderQuantityUnit,
        Plant,
        CreatedByUser,
        LastChangeDateTime,
        CreationDate;
...

```

Listing 10.9 Defining the readonly Field Property

As an additional field property, you can use the `field (mandatory)` ... addition to inform the UI level that `Material` and `OrderQuantity` are mandatory fields (see Listing 10.10).

Setting mandatory fields

```

...
    CreationDate;

    field ( mandatory ) Material, OrderQuantity;
...

```

Listing 10.10 Defining a Mandatory Field Property

These fields will now be displayed as mandatory fields in the user interface. You must check whether they've been populated using a validation (Section 10.4.6).

Using `mandatory:create` would be possible as an alternative, but makes it difficult to add draft handling later. In our example, we assume that a business object instance can basically be created without a given material and without a given quantity in the transaction buffer, even if it can't be saved in this inconsistent state.

Once you’ve declared the basic behavior (the implementation type and numbering) and the business object interface (e.g., `create` and `delete` operations, field properties), you can define the business-object-internal logic for implementing the purchase order creation and implement it with ABAP in the behavior pool.

10.4.3 Creating the Behavior Pool

Adding a behavior implementation

In this section, we’ll add a behavior implementation to the behavior definition. To do this, you need to remove the comment character (`//`) in the first line of the behavior definition from Listing 10.7 in order to declare the ABAP class `zbp_i_rap_purchaseorder_m` already stored there as a behavior pool. Since the RAP business object consists of only one CDS entity anyway, we’ll use a behavior pool for the entire behavior definition (see Listing 10.11).

```
managed with unmanaged save implementation in class zbp_i_rap_purchaseorder_m unique;

define behavior for ZI_RAP_PurchaseOrder_M alias PurchaseOrder
lock master
...

```

Listing 10.11 Declaration of the Behavior Pool in the Header of the Behavior Definition

Creating the behavior pool

Then, save and activate the behavior definition and create the behavior pool (i.e., the ABAP class) using ADT’s quick fix function. To do this, you must position the cursor on the name of the ABAP class in the source code and use the shortcut `[Ctrl] + [1]` (see Figure 10.7).

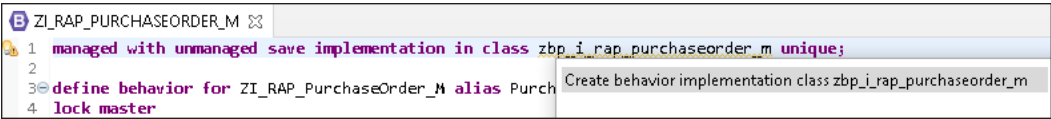


Figure 10.7 Creating the Behavior Pool via the Quick Fix Function

Complete the creation of the class via the dialog. Then, the source code editor for the created class will open. Based on the already declared behavior (late numbering and unmanaged save sequence), the save handler is generated and displayed directly on the **Local Types** tab (see Listing 10.8).

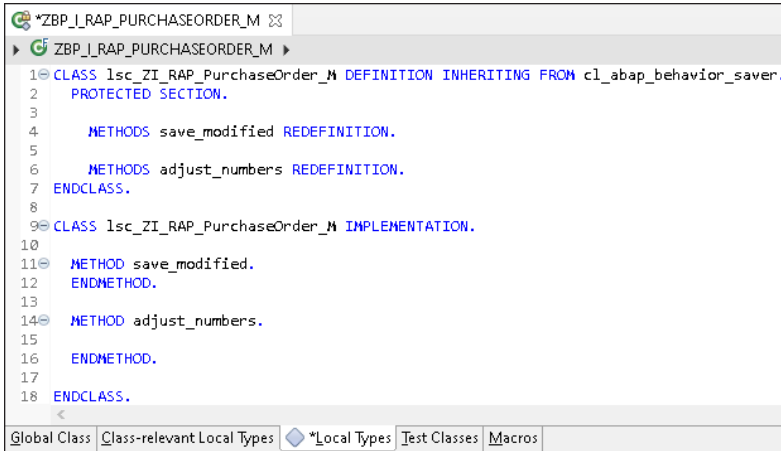


Figure 10.8 Creating the Behavior Pool with the Save Handler

10.4.4 Implementing Determinations

To create a purchase order via the BAPI, it's necessary to pass some central fields of the purchase order. This data is determined during the interaction phase and stored in the created instances in the transaction buffer. The purpose is to determine the following data:

Data to be determined

- **Data with organizational reference to the purchase order**
The fields with organizational reference are the fields for the purchasing organization, the purchasing group, and the plant.
- **Data with reference to the valid material**
These fields include the item number, supplier, and unit of measure of the valid material.

In this section, we'll declare and implement two determinations for this purpose.

For this purpose, you should create the determinations `initOrgData` and `initMaterialRelatedData` in the behavior definition, as shown in Listing 10.12. Save and activate the behavior definition afterwards.

Declaring determinations

```

...
field ( mandatory ) Material, OrderQuantity;

determination initOrgData on modify { create; }
determination initMaterialRelatedData on modify { create; }
...

```

Listing 10.12 Defining Determinations During the Create Operation

The determinations will be executed in the interaction phase after the `create` operation and require an implementation in the behavior pool.

Then, using ADT's quick fix feature, you must create the methods for the two determinations in the behavior pool. In each case, position the cursor on the determination name in the behavior definition.

Implementing the `initOrgData` Determination

initOrgData method You've created the `initOrgData` determination to determine organizational data for the purchase order creation. In Listing 10.13, you can see the signature of the `initOrgData` method for this determination in the interaction handler `lhc_PurchaseOrder`.

```
CLASS lhc_PurchaseOrder DEFINITION INHERITING FROM cl_abap_behavior_
handler.
...
PRIVATE SECTION.
    METHODS initOrgData FOR DETERMINE ON MODIFY
    IMPORTING keys FOR PurchaseOrder~initOrgData.
...
```

Listing 10.13 Signature of `initOrgData` Method for Determination

Implement the `initOrgData` method using EML. You can see the complete ABAP source code in Listing 10.14.

```
...
METHOD initOrgData.
    DATA(org_data) = read_org_data( ).
    MODIFY ENTITIES OF ZI_RAP_PurchaseOrder_M IN LOCAL MODE
    ENTITY PurchaseOrder
    UPDATE FROM
    VALUE #( FOR k IN keys
    ( %tky = k-%tky
    PurchasingOrganization = org_data-purch_org
    PurchasingGroup = org_data-pur_group
    Plant = org_data-plant
    %control-PurchasingOrganization = if_abap_behv=>mk-on
    %control-PurchasingGroup = if_abap_behv=>mk-on
    %control-Plant = if_abap_behv=>mk-on ) ).
ENDMETHOD.
```

Listing 10.14 Implementation of the `initOrgData` Determination Method

Because the interaction phase for this RAP business object is managed by the programming model, the implementation of the `create` operation is handled by the managed business object provider, and the created instances are available in the transaction buffer at runtime of the `initOrgData` determination without you needing to program anything for this. As a trigger condition for the determination, we have declared the `create` operation so that the implementing method `initOrgData` is executed after the creation of the instance(s) and the created instances are available in the transaction buffer at this time.

Instances created in the transaction buffer

Since handler methods in the ABAP RESTful application programming model are multi-instance capable, all key values of the affected (in this case, created) instance(s) are passed via the import parameter `keys`. A late numbering is declared for CDS entity `ZI_RAP_PurchaseOrder_M`, which is why the `keys` parameter is used to pass preliminary key values (`%TKY-%PID`) generated by the RAP framework, which are used to identify the created instances and are only valid during the RAP transaction.

Identifying created instances with %PID

Determinations perform calculations. In our example, they determine the appropriate organizational data to create the purchase order. This calculation is done using a separate method named `read_org_data`, which returns the purchasing organization, plant, and purchasing group independent of an instance. For this reason, reading concrete instances from the transaction buffer via `READ ENTITIES` isn't necessary in this implementation.

Running a determination

The `read_org_data` method is an implementation detail of our sample application and is representative of the "real" determination or calculation. It would have to be adapted or designed accordingly, depending on the requirements. For example, organizational data could be determined based on the user's organizational affiliation or derived from the ordered material.

You must now enter the organizational data determined in the `org_data` data object in the instances created within the RAP request. To do this, you can use the `MODIFY ENTITIES` statement. Note that you use an `update` operation when doing this, since the instances to be changed are already available in the transaction buffer.

Changing instances

Iterate over the key values `keys` of the affected instances in a `FOR` loop inside the constructor expression `VALUE#()`. The constructor expression creates an internal table that's passed as an actual parameter to the `UPDATE FROM` expression. You only need to provide the `PurchasingOrganization`, `PurchasingGroup` and `Plant` fields with the appropriate values and use the control structure `%CONTROL` to indicate that the field contents should be set. You can

Using a constructor expression

identify the key values of the instances using the key structure of the %TKY of the transactional key.

Implementing the `initMaterialRelatedData` Determination

Purpose of the determination The `initMaterialRelatedData` determination ensures that material-relevant data is determined on the basis of the transferred material number when an instance of the `ZI_RAP_PurchaseOrder_M` entity is created and the corresponding instances are updated with this data. The following field contents are determined for this purpose:

- **Item number (PurchaseOrderItem)**
The item number is assigned as a fixed value.
- **Supplier (Supplier)**
The supplier is determined via CDS entity `ZI_RAP_PO_Material` from the application table `ZRAP_A_PO_MAT`.
- **Purchase order quantity unit (PurchaseOrderQuantityUnit)**
The purchase order quantity unit is determined as the base unit of measure `MaterialBaseUnit` from the material master `I_Material`.

Access to material data Let's turn to the program logic for determining the above fields. Before you implement the determination of the supplier, you should do some preliminary work. The determination of the mentioned fields is based on CDS entity `ZI_RAP_PO_Material` and is swapped out to separate methods. In Listing 10.15, you can see the declarative part of the swapped out methods to implement the reading of the data from the application table.

```
CLASS lhc_PurchaseOrder DEFINITION INHERITING FROM cl_abap_behavior_handler.
...
PUBLIC SECTION.
    CLASS-METHODS class_constructor.
    ...
PRIVATE SECTION.
    "! Material data for fast purchase order entry
    CLASS-DATA gt_material_cust TYPE TABLE OF ZI_RAP_PO_Material.
    ...
    CLASS-METHODS read_cust_by_material
        IMPORTING
            VALUE(material) TYPE ZI_RAP_PurchaseOrder_M-Material
        EXPORTING
            VALUE(material_cust) TYPE ZI_RAP_PO_Material.

    CLASS-METHODS read_supplier_by_material
        IMPORTING
```

```

        VALUE(material) TYPE ZI_RAP_PurchaseOrder_M-Material
RETURNING
        VALUE(supplier) TYPE ZI_RAP_PO_Material-Supplier.
...

```

Listing 10.15 Declarative Part of the Swapped Out Methods for Reading the Valid Material

Listing 10.16 contains the method implementations.

```

CLASS lhc_PurchaseOrder IMPLEMENTATION.
    METHOD class_constructor.
        SELECT * FROM ZI_RAP_PO_Material
            INTO TABLE @gt_material_cust.
    ENDMETHOD.

    METHOD read_supplier_by_material.
        read_cust_by_material(
            EXPORTING
                material      = material
            IMPORTING
                material_cust = DATA(material_cust) ).
        supplier = material_cust-Supplier.
    ENDMETHOD.

    METHOD read_cust_by_material.
        READ TABLE gt_material_cust
            WITH KEY Material = material
            INTO material_cust.
    ENDMETHOD.
...

```

Listing 10.16 Method Implementations for Determining the Supplier

We don't want to go through the coding in detail, since it's standard ABAP. However, we'd like to point out that you can access all standard constructs within ABAP classes in the behavior pool. For example, you can use the class constructor to read data about the valid material once within the ABAP session (when loading the class into the memory). This allows you to access its contents at later points within the RAP transaction. We assume that the number of records in this table is small so that it can be completely loaded into the memory. Depending on the application, single record access to the table may be more advantageous.

For further modularization, we've implemented two small custom methods, `read_supplier_by_material` and `read_cust_by_material`. The `initMate-`

initMaterialRelated-Data method

rialRelatedData determination is also called at the time of the create operation. Accordingly, there are similarities with the implementation of the initOrgData determination. In the following sections, we'll therefore only explain the main differences in the implementation. In Listing 10.17, you can see the signature of the corresponding method for determining initMaterialRelatedData.

```
...
PRIVATE SECTION.

METHODS initOrgData FOR DETERMINE ON MODIFY
IMPORTING keys FOR PurchaseOrder~initOrgData.

METHODS initMaterialRelatedData FOR DETERMINE ON MODIFY
IMPORTING keys FOR PurchaseOrder~initMaterialRelatedData.
```

Listing 10.17 Method Signature for Determining initMaterialRelatedData

Navigate to the empty method implementation of initMaterialRelatedData and begin its implementation based on Listing 10.18.

```
...
METHOD initMaterialRelatedData.
  READ ENTITY IN LOCAL MODE ZI_RAP_PurchaseOrder_M
  FIELDS ( Material )
  WITH CORRESPONDING #( keys )
  RESULT DATA(po_headers).
...
```

Listing 10.18 Reading Instances from the Transaction Buffer via Read Entity Command

You can use the key values (keys) passed to the method to access the instances of CDS entity ZI_RAP_PurchaseOrder_M already created in the transaction buffer via the READ ENTITY short form. With FIELDS (Material), you only read the material number because only this is necessary for further calculation. You can use the constructor expression CORRESPONDING #(keys) to pass the passed key values of the internal table keys converted to the READ ENTITY command. This includes relevant key fields such as the preliminary ID %PID during the interaction phase with late numbering.

Changing instances
via MODIFY
ENTITIES

You now need to determine the relevant fields for the transferred instances and change them using the MODIFY ENTITIES command in the transaction buffer. Listing 10.19 shows the further implementation of the method.

```

METHOD initMaterialRelatedData.
  READ ENTITY IN LOCAL MODE
  ...
  RESULT DATA(po_headers).

MODIFY ENTITIES OF ZI_RAP_PurchaseOrder_M IN LOCAL MODE
ENTITY PurchaseOrder
  UPDATE FROM
    VALUE #( FOR po IN po_headers
      ( %tky = po-%tky
        Supplier =
          read_supplier_by_material( material = po-material )
          PurchaseOrderItem = '10'
          PurchaseOrderQuantityUnit =
            read_baseunit_by_material( material = po-material )
            %control-Supplier = if_abap_behv=>mk-on
            %control-PurchaseOrderItem = if_abap_behv=>mk-on
            %control-PurchaseOrderQuantityUnit =
              if_abap_behv=>mk-on ) ).
ENDMETHOD.

```

Listing 10.19 Changing Instances via Modify Entities Command

The Material field has been declared as a mandatory field in the behavior definition, which is why you don't need to consider the case of a material number that hasn't been transferred here. Again, you must use a constructor expression (VALUE #(...)) to create the internal table to pass data to the update operation of MODIFY ENTITIES. In this constructor expression, a row is created in the internal table for each instance read (FOR po IN po_headers). In each iteration over po_headers, the previously implemented methods are called with the material (po-material) as actual parameters to determine the supplier or the appropriate unit of measure and set it in the internal table.

Swapping out a Determination

You can also use mass determination of the data to be updated *before* the EML statement, MODIFY ENTITIES ... UPDATE and, if necessary, return messages via the return parameter REPORTED if, for example, the supplier for a material couldn't be determined.

The determinations implemented in this way ensure that the appropriate data for creating the purchase order is stored in the instances to be created.



You have thus created the starting point on the basis of which you can implement the save sequence and thus the actual creation of the purchase order via BAPI. (A short jump back to the interaction phase will still be necessary, however, but more on that in the next section.)

10.4.5 Save Sequence: Implementing the Creation via BAPI

**ADJUST_NUMBERS
method**

The development of the solution strategy (Section 10.1) has shown that we need a late numbering for the RAP business object, but the numbering is carried out by the BAPI `BAPI_PO_CREATE1` in the course of the purchase order creation. To correctly implement the RAP contract for late numbering, you must implement the purchase order creation in the `ADJUST_NUMBERS` method provided for late numbering in the save handler, not in the `SAVE_MODIFIED` method. Due to the automatic creation of the behavior pool, this method has already been declared in the save handler (see Listing 10.20).

```
CLASS lsc_ZI_RAP_PurchaseOrder_M DEFINITION INHERITING FROM cl_abap_
behavior_saver.
  PROTECTED SECTION.
    METHODS adjust_numbers REDEFINITION.
    ...
ENDCLASS.
```

Listing 10.20 Declaration of the `ADJUST_NUMBERS` Method in the Save Handler

**Reading created
instances from the
transaction buffer**

First, you should read the instances created during the interaction phase. Use the preliminary key values assigned by the RAP framework from the internal table `mapped-purchaseorder`, which has been passed to the method (see Listing 10.21).

```
CLASS lsc_ZI_RAP_PurchaseOrder_M IMPLEMENTATION.
  METHOD adjust_numbers.
    " Read purchase orders based on key values from interaction phase
    READ ENTITIES OF ZI_RAP_PurchaseOrder_M
      ENTITY PurchaseOrder
      ALL FIELDS
      WITH CORRESPONDING #( mapped-purchaseorder )
      RESULT DATA(pos_to_create).
    ...
```

Listing 10.21 Reading Created Purchase Order Instances From the Transaction Buffer

Now iterate over the read instances and create a purchase order for each entry via BAPI. We've swapped out the BAPI call to a separate method: `create_purchase_order` (see Listing 10.22).

```
...
LOOP AT pos_to_create INTO DATA(po_to_create).
  " Create a new purchase order
  lhc_purchaseorder=>create_purchase_order(
    EXPORTING
      as_test_run      = abap_false
      po_entity        = CORRESPONDING #( po_to_create )
    IMPORTING
      po_header_created = DATA(po_header_created)
      return            = DATA(return) ).
...
ENDLOOP.
```

Listing 10.22 Creating a Purchase Order Within the `Adjust_Numbers` Method

For the `create_purchase_order` method, you need to first create the type declaration `tt_return` and the method declaration (see Listing 10.23).

**Method for the
BAPI call**

```
CLASS lhc_PurchaseOrder DEFINITION INHERITING FROM cl_abap_behavior_
handler.
```

```
  PUBLIC SECTION.
    TYPES tt_return TYPE STANDARD TABLE OF bapiret2.
    CLASS-METHODS create_purchase_order
      IMPORTING
        VALUE(po_entity) TYPE ZI_RAP_PurchaseOrder_M
        VALUE(as_test_run) TYPE abap_bool DEFAULT abap_true
      EXPORTING
        VALUE(return) TYPE tt_return
        VALUE(po_header_created) TYPE bapimepoheader.
...

```

Listing 10.23 Signature of the `create_purchase_order` Method for the Purchase Order Creation

The method uses CDS root entity `ZI_RAP_PurchaseOrder_M` as the import parameter so that the data can be passed according to the type. The `as_test_run` flag calls the BAPI in test mode. Implement the method using the ABAP source code from Listing 10.24:

```
METHOD create_purchase_order.
  DATA: ls_po_header  TYPE bapimepoheader,
         ls_x_po_header TYPE bapimepoheaderx,
         lt_po_item    TYPE STANDARD TABLE OF bapimepoitem,
         ls_po_item    LIKE LINE OF lt_po_item,
         lt_x_po_item  TYPE STANDARD TABLE OF bapimepoitemx.

  " Header data for the purchase order
  ls_po_header = CORRESPONDING #( po_entity MAPPING FROM
    ENTITY ).

  ls_x_po_header-vendor      = abap_true.
  ls_x_po_header-purch_org  = abap_true.
  ls_x_po_header-pur_group  = abap_true.

  " Item data, create only one item
  " Map PurchaseOrderItem, Material, Plant, OrderQuantity
  ls_po_item = CORRESPONDING #( po_entity MAPPING FROM ENTITY ).
  ls_po_item-net_price = '1.0'.
  APPEND ls_po_item TO lt_po_item.

  lt_x_po_item = VALUE #( ( po_item    = ls_po_item-po_item
                           net_price = abap_true
                           material  = abap_true
                           plant     = abap_true
                           quantity  = abap_true ) ).

  CALL FUNCTION 'BAPI_PO_CREATE1'
    EXPORTING
      poheader      = ls_po_header
      poheaderx     = ls_x_po_header
      testrun       = as_test_run
    IMPORTING
      expheader     = po_header_created
    TABLES
      return        = return
      poitem        = lt_po_item
      poitemx       = lt_x_po_item.
ENDMETHOD.
```

Listing 10.24 Implementation of the create_purchase_order Method

This is an ordinary BAPI call that takes the data from the passed instance (po_entity) and creates a purchase order from it.

In the implementation, we use assignments via the constructor expression `CORRESPONDING #()` using field mappings via `MAPPING FROM ENTITY`. In this way, the mapping between fields of structured legacy data types (here the data types of the BAPI) to the respective CDS entity of the RAP business object can be used for assignments. For this purpose, a corresponding field mapping must exist in the behavior definition. Add and activate the two necessary field mappings from Listing 10.25 in the behavior definition.

Using field mappings

mapping for bapimepoheader corresponding

```
{
  PurchaseOrder = po_number;
  PurchasingOrganization = purch_org;
  PurchasingGroup = pur_group;
  Supplier = vendor;
}
```

mapping for bapimepoitem corresponding

```
{
  PurchaseOrderItem = po_item;
  Material = material;
  OrderQuantity = quantity;
}
```

Listing 10.25 Declared Field Mappings for the BAPI Data Types

After the BAPI call has been moved to its own method, you can turn your attention to implementing the `ADJUST_NUMBERS` method. Report an error situation via `dump` and populate the `MAPPED` parameter (see Listing 10.26).

Returning purchase order numbers

```
...
LOOP AT pos_to_create INTO DATA(po_to_create).
  " Create a new purchase order
  lhc_purchaseorder=>create_purchase_order(
    EXPORTING
      as_test_run      = abap_false
      po_entity        = CORRESPONDING #( po_to_create )
    IMPORTING
      po_header_created = DATA(po_header_created)
      return            = DATA(return) ).

  READ TABLE return WITH KEY type = 'E'
    INTO DATA(return_err).
  IF sy-subrc EQ 0.
    RAISE SHORTDUMP NEW zcx_rap_purchaseorder( ... ).
  ENDIF.
```

```
" MAPPED: Return key values
APPEND INITIAL LINE TO mapped-purchaseorder
    ASSIGNING FIELD-SYMBOL(<ls_mapped>).
<ls_mapped>-%pid = po_to_create-%pid.
<ls_mapped>-PurchaseOrder = po_header_created-po_number.
...
ENDLOOP.
```

Listing 10.26 Returning Purchase Order Numbers via the MAPPED Parameter

The essential responsibility of the `ADJUST_NUMBERS` method is to generate and return a permanent key value for each `%PID`. The purchase order creation returns the created purchase order data including the purchase order number via the actual parameter `po_header_created`. This is why we don't create the purchase order in the `SAVE_MODIFIED` method; we create it at this point. Via a new line in the internal table `mapped-purchaseorder`, you assign the corresponding purchase order number from the previously performed creation process to the currently processed `%PID`.

Returning messages Complete the implementation of the `ADJUST_NUMBERS` method with a return of messages. These are passed in the `REPORTED` parameter. Listing 10.27 contains the ABAP source code.

```
LOOP AT pos_to_create INTO DATA(po_to_create).
...
reported-purchaseorder = VALUE #(
    BASE reported-purchaseorder
    FOR r IN return WHERE ( type = 'I' OR type <> 'W' )
    ( %tky = po_to_create-%tky
      PurchaseOrder = po_header_created-po_number
      %msg = me->new_message(
        id      = r-id
        number  = r-number
        severity = COND #(
            WHEN r-type = 'I'
            THEN if_abap_behv_message=>severity-information
            WHEN r-type = 'W'
            THEN if_abap_behv_message=>severity-warning )
      v1 = r-message_v1
      v2 = r-message_v2
      v3 = r-message_v3
      v4 = r-message_v4 )
    ) ).
ENDLOOP.
```

Listing 10.27 Populating the Reported Parameter from the `BAPIRET2` Structure

Note that the `ADJUST_NUMBERS` method is already executed after the point of no return within the save sequence and therefore no error messages can be reported via the `REPORTED` parameter. If an error does occur at this point, you must use `RAISE SHORTDUMP` to create a dump.

10.4.6 Implementing Validations

To ensure that the creation of the purchase order with the given data is possible, you'll implement validations in this section that check the data consistency before saving. In our use case, we want to examine the following issues:

- Have the mandatory fields for the purchase order quantity and material been filled?
- Is it generally possible to create a purchase order with the given data?
- Is the material number contained in the purchase order item valid?

To do this, you need to create two different validations—`validatePurchaseOrder` and `validateMaterial`—in the behavior definition for the `ZI_RAP_PurchaseOrder_M` entity with the alias name `PurchaseOrder` (see Listing 10.28).

Declaring
validations

```
define behavior for ... alias PurchaseOrder
{
    ...
    validation validatePurchaseOrder on save { create; }
    validation validateMaterial on save { create; }
}
```

Listing 10.28 Declaring Validations for CDS Entity `PurchaseOrder`

The `validatePurchaseOrder` validation performs the entire check of the purchase order creation. You can use the test indicator of the BAPI `BAPI_PO_CREATE1` for this. Since we can't influence the call sequence of the validations on the part of the RAP framework, we don't swap out the check for mandatory fields to a separate validation, but additionally check the mandatory fields in the `validatePurchaseOrder` validation. We call the BAPI only if these mandatory fields have been filled.

The `validateMaterial` validation checks whether the specified material is valid. Since the RAP business object doesn't support an `update` operation, you should use `create` to store the attachment operation as a trigger condition for these validations.

For the error messages you also create a message class named `ZRAP_PO`. To do this, you can use the shortcut `[Ctrl] + [N]` to open the creation dialog in ADT and select the **Message Class** entry under **ABAP**. Run through the

Creating a message
class with messages

creation wizard, providing the message class with the messages listed in Table 10.4 in the process. Save and close the message class afterwards.

Number	Message Text
001	Material &1 is not intended for fast entry.
002	Material &1 is not active.
003	Please enter a purchase order quantity.
004	Please enter a material.
005	No authorization to create purchase orders.

Table 10.4 Messages of the ZRAP_PO Message Class

Implementing the validatePurchaseOrder Validation

Declaring a method

Generate the method declarations again using the quick fix function for the validation names in the behavior pool. Since the validatePurchaseOrder validation has the create trigger condition, you can use the keys parameter to read data about the created instances from the transaction buffer (see Listing 10.29).

```
METHOD validatePurchaseOrder.  
  DATA mandatory_field_missing TYPE abap_bool.  
  READ ENTITIES OF ZI_RAP_PurchaseOrder_M IN LOCAL MODE  
    ENTITY PurchaseOrder  
      ALL FIELDS WITH CORRESPONDING #( keys )  
      RESULT DATA(pos_to_create).  
  ...
```

Listing 10.29 Reading Instances to be Checked from the Transaction Buffer

Checking mandatory fields

Then, iterate over the instances to be checked, and check whether the mandatory field of the purchase order quantity OrderQuantity (Section 10.4.2) has been filled. If that's not the case, output the validation error via the FAILED parameter with reference to the respective instance (<po_to_create>-%tky), marking the create operation as incorrectly executed. For this purpose, you should fill the REPORTED parameter with a suitable error message and refer to the incorrect field (see Listing 10.30).

```
...  
LOOP AT pos_to_create ASSIGNING FIELD-SYMBOL(<po_to_create>).  
  
  CLEAR mandatory_field_missing.
```

```

" Checking mandatory fields
IF <po_to_create>-OrderQuantity IS INITIAL.
    failed-purchaseorder = VALUE #( BASE failed-purchaseorder
        ( %tky = <po_to_create>-%tky
          %create = if_abap_behv=>mk-on ) ).

    reported-purchaseorder = VALUE #( BASE reported-purchaseorder
        ( %tky = <po_to_create>-%tky
          %element-orderquantity = if_abap_behv=>mk-on
          %msg = me->new_message( severity =
                                if_abap_behv_message=>severity-error
                                id = 'ZRAP_PO'
                                number = '003' ) ) ).

    mandatory_field_missing = abap_true.
ENDIF.

```

Listing 10.30 Checking a Mandatory Field for the Purchase Order Quantity

After that, you want to check if the mandatory field `Material` is filled by supplying the `FAILED` or `REPORTED` parameter, as shown in Listing 10.30. In this case, the `REPORTED` parameter receives message 004 of the previously created message class and refers to the `Material` field instead of the `OrderQuantity` field (see Listing 10.31).

```

...
IF <po_to_create>-Material IS INITIAL.
    failed-purchaseorder = ...
    reported-purchaseorder = ...
    mandatory_field_missing = abap_true.
ENDIF.

IF mandatory_field_missing = abap_true.
    CONTINUE.
ENDIF.
...

```

Listing 10.31 Checking a Mandatory Field for the Material

The current loop pass will be interrupted by `CONTINUE` if a mandatory field hasn't been filled so that the subsequent BAPI call won't take place.

If the mandatory fields `Material` and `OrderQuantity` have been filled, you can call the swapped out method `create_purchase_order` described in Section 10.4.5, but this time for validation purposes in test mode, using the import parameter `as_test_run = abap_true` (see Listing 10.32).

Calling the BAPI
for checking

```
...
create_purchase_order(
    EXPORTING
        as_test_run = abap_true
        po_entity   = CORRESPONDING #( <po_to_create> )
    IMPORTING
        return      = DATA(return) ).
...
```

Listing 10.32 Calling the create_purchase_order Method for Validation

Filling FAILED The essential part of this validation is to evaluate the result (here, the parameter `return`) and to report possible error situations. In the event of an error, you can thus ensure that the RAP framework will abort the save sequence. You must evaluate the result of the BAPI call in the `return` parameter (see Listing 10.33).

```
...
" Report error situation
READ TABLE return
    WITH KEY type = 'E'
    TRANSPORTING NO FIELDS.
IF sy-subrc EQ 0.
    " Report create operation as failed
    failed-purchaseorder = VALUE #( BASE failed-purchaseorder
        ( %tky = <po_to_create>-%tky
          %create = if_abap_behv=>mk-on ) ).
...
```

Listing 10.33 Reporting the Create Operation for an Instance as Failed

Filling REPORTED Subsequently, you can still return corresponding messages via the `REPORTED` parameter (see Listing 10.34).

```
...
" Return error messages
reported-purchaseorder = VALUE #( BASE reported-purchaseorder
    FOR r IN return WHERE ( type = 'E' )
    ( PurchaseOrder = <po_to_create>-PurchaseOrder
      %msg = me->new_message( id = r-id
        number = r-number
        severity = if_abap_behv_message=>severity-error
        v1 = r-message_v1
        v2 = r-message_v2
        v3 = r-message_v3
```

```

        v4 = r-message_v4 ) ) ).
    ENDIF.
  ENDLOOP.
ENDMETHOD.

```

Listing 10.34 Returning Messages via the Reported Parameter

Implementing the validateMaterial Validation

The validateMaterial validation checks whether the transferred material is valid for fast order entry. If that's not the case, the validation will report an error.

Open the behavior pool and navigate to the implementation class on the **Local Types** tab. Start with the implementation of the validateMaterial method. Again, first use EML to read all instances relevant to the check from the transaction buffer (see Listing 10.35).

Declaring a method

```

METHOD validateMaterial.
  READ ENTITIES OF ZI_RAP_PurchaseOrder_M IN LOCAL MODE
  ENTITY PurchaseOrder
    ALL FIELDS WITH CORRESPONDING #( keys )
    RESULT DATA(pos).
...

```

Listing 10.35 Reading Instances from the Transaction Buffer for Validation

Then, iterate over the instances you've read and perform the data validation. We assume that only instances with filled material number are to be checked, since you have already implemented the mandatory field check in the validatePurchaseOrder validation.

You should first read the validity of the material via the read_cust_by_material method so that you can check the material stored in the instances (see Listing 10.36).

```

...
LOOP AT pos INTO DATA(po) WHERE NOT Material IS INITIAL.
  " Access data from table ZRAP_BO
  read_cust_by_material(
    EXPORTING
      material      = po-Material
    IMPORTING
      material_cust = DATA(material_cust) ).
...

```

Listing 10.36 Reading Data for the Valid Material

Running the validation Now, you need to implement the actual validation of the data. To do so, you need to check whether the material in the instance is a valid material at all (that is, whether it has been stored in table ZRAP_A_PO_MAT). In addition, you must use the IsActive field to check whether the material is active, provided it has been stored. The data for the valid material is located in the material_cust data object.

Reporting an error via REPORTED You can report error situations via the REPORTED parameter using the previously created messages in the ZRAP_PO message class. Use the %element-material field to create a reference between the error message and the actual field (see Listing 10.37). This mapping can be evaluated by the SAP Fiori elements UI.

```
...
  IF NOT material_cust IS INITIAL.
    IF material_cust-IsActive NE abap_true.
      reported-purchaseorder = VALUE #( BASE reported-purchaseorder
        ( %tky = po-%tky
          %element-material = if_abap_behv=>mk-on
          %msg = me->new_message( severity =
            if_abap_behv_message=>severity-error
            id = 'ZRAP_PO'
            number = '002'
            v1 = |{ po-Material ALPHA = OUT }| ) ) ).
    ENDIF.
  ELSE.
    reported-purchaseorder = VALUE #( BASE reported-purchaseorder
      ( %tky = po-%tky
        %element-material = if_abap_behv=>mk-on
        %msg = me->new_message( severity =
          if_abap_behv_message=>severity-error
          id = 'ZRAP_PO'
          number = '001'
          v1 = |{ po-Material ALPHA = OUT }| ) ) ).
    ENDIF.
  ENDOLOOP.
ENDMETHOD.
```

Listing 10.37 Running the Validation and Returning Error Messages

Now, save and activate the behavior pool. Thus, you have implemented the necessary behavior to implement the create purchase order function via a behavior definition and the behavior pool.

10.5 Implementing the Delete Purchase Order Function

In this section, we'll implement the delete purchase order function. To do this, we implement the `SAVE_MODIFIED` method of the save handler "unlocked" by the `unmanaged` save implementation type. In Section 10.3, you declared the standard operation `delete` for CDS root entity `ZI_RAP_PurchaseOrder_M` and selected the save option with `unmanaged` save. Both declarations are a prerequisite for implementing the delete purchase order function now.

10.5.1 Save Sequence: Implementing the Deletion via BAPI

The delete operation acts on an existing instance of a purchase order, so you have to deal with competing data accesses here as well. For example, one user may change a purchase order via SAP's own SAP Fiori app or transaction, while another user may want to delete the same purchase order via your RAP application. In this case, deletion may not be possible due to the lock that has already been set. Although the SAP standard takes care of processing the purchase order within the application integrated via BAPIs, this only takes effect in the save sequence of the RAP transaction model. However, the lock has already been set by the ABAP RESTful application programming model in the interaction phase.

For this reason, you must implement your own lock behavior for the RAP business object and *not* use the generic lock behavior through the managed business object provider. It's necessary to add the `unmanaged` keyword to the lock master declaration in the behavior definition (see Listing 10.38).

```
define behavior for ZI_RAP_PurchaseOrder_M alias PurchaseOrder
lock master unmanaged
late numbering
...
```

Listing 10.38 Declaration of an Unmanaged Lock

Create the relevant FOR LOCK handler method using the quick fix function and then implement it. To do this, you should use the special function modules for lock management (*lock modules*) that belong to the purchase order (see Listing 10.39).

```
METHOD lock.
  LOOP AT keys ASSIGNING FIELD-SYMBOL(<k>).
    CALL FUNCTION 'ENQUEUE_EMEKKOS'
      EXPORTING
```

Implementing a custom locking behavior

Implementing a custom locking behavior

```
        ebeln          = <k>-PurchaseOrder
EXCEPTIONS
    foreign_lock      = 1
    system_failure    = 2
    OTHERS             = 3.
CASE sy-subrc.
    WHEN 1. " Purchase order is already locked
        failed-purchaseorder = VALUE #( BASE
            failed-purchaseorder
                ( purchaseorder = <k>-PurchaseOrder
                    %fail-cause = if_abap_behv=>cause-locked
                ) ).
        reported-purchaseorder = VALUE #(
            BASE reported-purchaseorder
                ( purchaseorder = <k>-PurchaseOrder
                    %msg      = new_message(
                        id      = sy-msgid
                        number  = sy-msgno
                        severity = if_abap_behv_message=>severity-
                                                                    error
                    )
                )
                v1 = sy-msgv1
                v2 = sy-msgv2
                v3 = sy-msgv3
                v4 = sy-msgv4 ) ) ).
    WHEN OTHERS.
        RAISE SHORTDUMP NEW zcx_rap_purchaseorder( ... ).
ENDCASE.
ENDLOOP.
ENDMETHOD.
```

Listing 10.39 Implementation of the For Lock Method

If the lock couldn't be set for the particular <k>-PurchaseOrder order number, you should populate the `FAILED` parameter with the corresponding order number (<k>-PurchaseOrder) and the cause of the error (`if_abap_behv=>cause-locked`). In addition, you should return a message for the respective purchase order via the `REPORTED` parameter. You can use the helper method `new_message` for this purpose. If the lock couldn't be set for technical reasons, you must trigger a dump (`RAISE SHORTDUMP`).

Implementing the delete operation

Having implemented the setting of the lock, you can swap out the deletion of a purchase order via BAPI to a separate method (as we've already done with the creation of the purchase order). You can declare this method in the implementation class in the behavior pool (see Listing 10.40).

```

CLASS lhc_PurchaseOrder DEFINITION INHERITING FROM cl_abap_behavior_
handler.
  PUBLIC SECTION.
  ...
  CLASS-METHODS delete_purchase_order
    IMPORTING
      VALUE(po_number)   TYPE ZI_RAP_PurchaseOrder_M-PurchaseOrder
      VALUE(as_test_run) TYPE abap_bool DEFAULT abap_true
    EXPORTING
      VALUE(return)      TYPE tt_return.

```

Listing 10.40 Method Declaration for Deleting a Purchase Order

Listing 10.41 contains the implementation of this method.

```

...
METHOD delete_purchase_order.
  DATA: ls_po_header   TYPE bapimepoheader,
        ls_x_po_header TYPE bapimepoheaderx.
  * Delete means to set a deletion indicator in the purchase order
  ls_po_header-delete_ind = abap_true.
  ls_x_po_header-delete_ind = abap_true.
  CALL FUNCTION 'BAPI_PO_CHANGE'
    EXPORTING
      purchaseorder = po_number
      poheader      = ls_po_header
      poheaderx     = ls_x_po_header
      testrun       = as_test_run
  TABLES
    return          = return.
ENDMETHOD.
...

```

Listing 10.41 Method Implementation for Deleting a Purchase Order

Warning: Considering the Semantics of the Delete Operation

Even though deleting a purchase order is implemented by means of updating the purchase order by setting a deletion indicator (`delete_ind`) in the BAPI `BAPI_PO_CHANGE`, the RAP business object will provide the `delete` operation. On a logical level, this is a deletion of a purchase order, not an update. The `delete` operation occurs external to the business object, while the implementation of this function occurs internally. The semantics of the interface and the actual implementation are thus separated from each other.



**Implementing
SAVE_MODIFIED**

Next, you want to implement the `SAVE_MODIFIED` method in the save handler of the behavior pool. During the creation of the behavior pool, the method has already been redefined there (see the subsection about implementing the `initOrgData` determination in Section 10.4.4). You can see the redefinition in Listing 10.42.

```
CLASS lsc_ZI_RAP_PurchaseOrder_M DEFINITION INHERITING FROM cl_abap_
behavior_saver.
  PROTECTED SECTION.
    METHODS adjust_numbers REDEFINITION.
    METHODS save_modified REDEFINITION.
ENDCLASS.
```

Listing 10.42 Redefinition of the `SAVE_MODIFIED` Method in the Save Handler

You can check the signature of the `SAVE_MODIFIED` method by placing the cursor on the method name and pressing the `[F2]` function key. Then, you'll see the signature as shown in Figure 10.9.

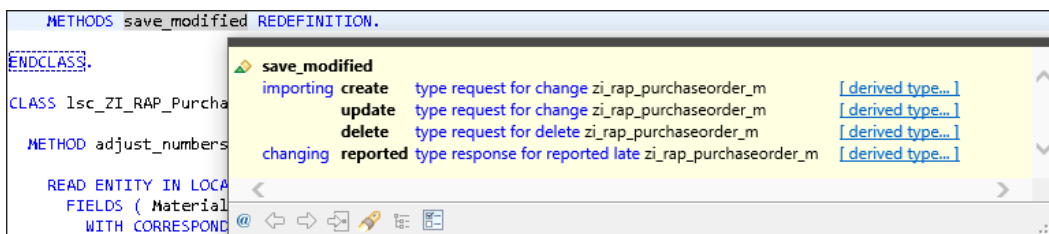


Figure 10.9 Signature of the `SAVE_MODIFIED` Method

You can access the purchase orders to be deleted using the `delete` parameter with the derived data type for deleting RAP entities (TYPE REQUEST FOR DELETE). In Listing 10.43, you can see the implementation of the `SAVE_MODIFIED` method. Iterate over the purchase orders marked for deletion in the transaction buffer (LOOP AT `delete-purchaseorder`) and call the previously implemented method `delete_purchase_order` to delete the respective purchase order.

```
...
METHOD save_modified.
  LOOP AT delete-purchaseorder ASSIGNING FIELD-SYMBOL(<po_to_delete>).
    lhc_purchaseorder=>delete_purchase_order(
      EXPORTING
        po_number      = <po_to_delete>-PurchaseOrder
        as_test_run    = abap_false
```

```

IMPORTING
    return      = DATA(return) ).
...

```

Listing 10.43 Implementation of the SAVE_MODIFIED Method

After that, you can return messages about the saving process via the REPORTED parameter. Since we're in the late save sequence (i.e., the point in time is after the point of no return, as is the case with the ADJUST_NUMBERS method), no more error messages can be returned here. If an error does occur at this point, you must create a dump (RAISE SHORTDUMP) (see Listing 10.44).

```

...
READ TABLE return WITH KEY type = 'E'
    INTO DATA(return_err).
IF sy-subrc EQ 0.
    RAISE SHORTDUMP NEW zcx_rap_purchaseorder( ... ).
ENDIF.
...

```

Listing 10.44 Reporting an Error Situation via Dump After the Point of No Return

Evaluate the return parameter, which contains the messages of the BAPI call, and return a corresponding message with reference to the respective purchase order instance (<po_to_delete>-PurchaseOrder) for each entry in the return parameter. Listing 10.45 contains the ABAP source code for this.

```

...
" Return messages
reported-purchaseorder = VALUE #( BASE reported-purchaseorder
    FOR r IN return WHERE ( type = 'I' OR type <> 'W' )
    ( PurchaseOrder = <po_to_delete>-PurchaseOrder
        %msg = me->new_message(
            id      = r-id
            number   = r-number
            severity = COND #( WHEN r-type = 'I' THEN
                if_abap_behv_message=>severity-information
                WHEN r-type = 'W' THEN
                if_abap_behv_message=>severity-warning )
            v1 = r-message_v1
            v2 = r-message_v2
            v3 = r-message_v3
            v4 = r-message_v4 ) ) ).

```

```
ENDLOOP.  
ENDMETHOD.
```

Listing 10.45 Returning Messages in SAVE_MODIFIED via Reported Method

Then, save and activate the behavior pool.

10.5.2 Implementing a Validation

The delete purchase order function also includes a validation that checks whether a deletion is possible at all. The procedure for implementing this validation is identical to that for implementing the `validatePurchaseOrder` validation from Section 10.4.6.

**Declaring the
validateOnDelete
validation**

First, you need to declare the `validateOnDelete` validation in the behavior definition:

```
...  
validation validateMaterial on save { create; }  
validation validateOnDelete on save { delete; }  
...
```

It has the trigger condition `delete`, so it gets called for instances deleted in the transaction buffer during the save sequence.

**Implementing the
validation**

Create and implement an appropriate method in the behavior pool using the quick fix function. You can see the ABAP source code for this validation in Listing 10.46.

```
METHOD validateOnDelete.  
  LOOP AT keys ASSIGNING FIELD-SYMBOL(<k>).  
    delete_purchase_order(  
      EXPORTING  
        po_number    = <k>-PurchaseOrder  
        as_test_run  = abap_true  
      IMPORTING  
        return       = DATA(return) ).  
  ...
```

Listing 10.46 Checking Whether it is Possible to Delete a Purchase Order

For deletion, only the reference to the respective instance via the key value is necessary. The key values are available via the `keys` parameter. Reading from the transaction buffer via `READ ENTITIES` is therefore not necessary.

For this validation, you also report for each purchase order instance to be deleted via the **FAILED** parameter whether or not the delete operation can be performed (see Listing 10.47).

```
...
" Report error situation
READ TABLE return
  WITH KEY type = 'E'
  TRANSPORTING NO FIELDS.
IF sy-subrc EQ 0.
  " Report delete operation as failed
  failed-purchaseorder = VALUE #( BASE
                                failed-purchaseorder
                                ( PurchaseOrder = <k>-PurchaseOrder
                                  %delete = if_abap_behv=>mk-on ) ).
...

```

Listing 10.47 Filling the Failed Parameter for the Delete Operation

After that, you want to pass the error messages from the `return` parameter to the `REPORTED` parameter again (see Listing 10.48). Finally, save and activate the behavior pool.

```
...
" Return error messages
reported-purchaseorder = VALUE #( BASE reported-purchaseorder
  FOR r IN return WHERE ( type = 'E' )
  ( PurchaseOrder = <k>-PurchaseOrder
    %msg = me->new_message( id = r-id
      number = r-number
      severity = if_abap_behv_message=>severity-error
      v1 = r-message_v1
      v2 = r-message_v2
      v3 = r-message_v3
      v4 = r-message_v4 ) ) ).
ENDIF.
ENDLOOP.
ENDMETHOD.
...

```

Listing 10.48 Filling the Reported Parameter

10.6 Defining Business Services

My Purchase Orders application In the previous sections, you’ve implemented a RAP business object named ZI_RAP_PurchaseOrder_M with data model and transactional behavior. Now we want to provide users with an SAP Fiori app called My Purchase Orders, in which they can view their placed purchase orders based on the RAP business object and create new purchase orders. You’ll use the projection layer to provide a service-specific view of the RAP business object. In turn, you’ll expose the projection layer artifacts as an OData service via a service definition and a service binding so that the SAP Fiori app can consume them.

10.6.1 Setting up the Projection Layer for the My Purchase Orders Application

In the following steps, you’ll create a CDS projection view and a projection behavior definition to implement the projection layer.

Creating a CDS projection view First, you need to create a CDS projection view based on CDS root entity ZI_RAP_PurchaseOrder_M:

- 1. Open the context menu of the ZI_RAP_PurchaseOrder_M entity in the **Project Explorer** and select the menu item **New Data Definition**.
- 2. The name of the CDS root entity is preset in the **Referenced Object** field in the following creation dialog. Name the CDS projection view “ZC_RAP_PurchaseOrderOwn_M” and enter a description (see Figure 10.10).

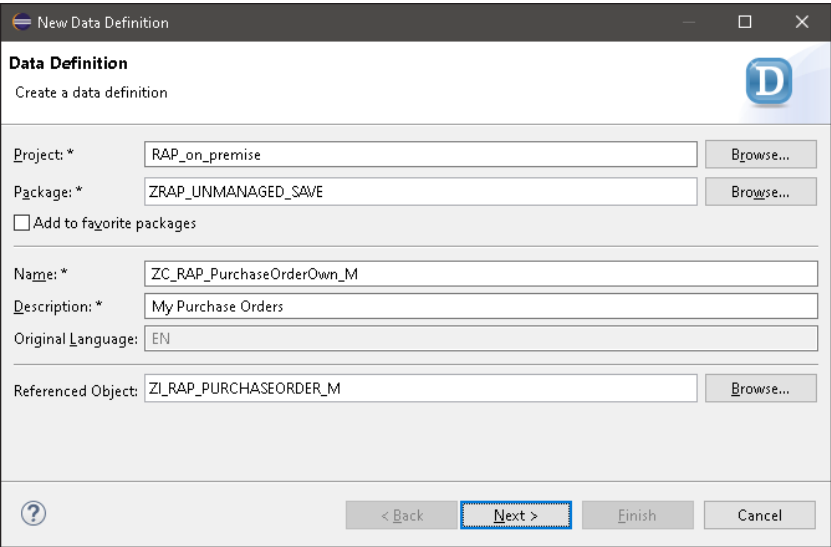


Figure 10.10 Creating a CDS Projection View

3. In the dialog that follows next, you must select the **Define Projection View** template. Complete the process by clicking the **Finish** button.

The DDL source editor for the CDS projection view will open. Perform a source code formatting using the shortcut `[Ctrl] + [Shift] + [F]`. Then, add the root keyword to the CDS entity, as shown in Listing 10.49.

Defining the CDS projection view

```
@EndUserText.label: 'My Purchase Orders'
@AccessControl.authorizationCheck: #CHECK
define root view entity ZC_RAP_PurchaseOrderOwn_M
  as projection on ZI_RAP_PurchaseOrder_M
{
  key PurchaseOrder,
    PurchaseOrderType,
    ...
}
```

Listing 10.49 Defining a CDS Projection View as Root Entity

The investment process generates the selection list of the CDS projection view from the projected CDS entity `ZI_RAP_PurchaseOrder_M`. Remove the `PurchasingOrganization`, `PurchasingGroup`, `Supplier`, `CreatedByUser` and `CreationDate` fields from this selection list.

Add another field named `MaterialName` to the selection list of the CDS projection view and an annotation named `ObjectModel.text.element` to provide the material number with a language-dependent descriptive text (the material short text) (see Listing 10.50).

Selecting language-dependent text

```
...
  PurchaseOrderDate,
  @ObjectModel.text.element: ['MaterialName']
  Material,
  _POMaterial._Material._Text.MaterialName as
    MaterialName : localized,
  OrderQuantity,
  ...
```

Listing 10.50 Defining a Descriptive Text for Material Number

The `MaterialName` field is a text element. The `localized` keyword is used in the CDS projection view to resolve the text association depending on the current logon language.

In the `My Purchase Orders` application, only the purchase orders that the respective logged-in user has created are relevant. So, you should add the `CreatedByUser = $session.user` filter criterion to the CDS projection view via the `where` clause:

Adding a filter criterion

```
...
    _PurchaseOrderItem
}
where
    CreatedByUser = $session.user
```

Save and activate the CDS projection view afterwards.

Creating a projection behavior definition

Based on the previously created CDS projection view, you can now create a projection behavior definition so that you can expose the desired transactional behavior. To do this, select **New Behavior Definition** from the context menu of CDS projection view `ZC_RAP_PurchaseOrderOwn_M`. You can optionally enter a new description in the creation dialog; the other input fields are already prefilled and can't be modified. Go through the dialog and finish the process by clicking the **Finish** button.

A generated BDL source code will open; in there, you should add the alias name `PurchaseOrder` (see Listing 10.51). Save and activate the projection behavior definition afterwards.

```
projection;

define behavior for ZC_RAP_PurchaseOrderOwn_M alias PurchaseOrder
{
    use create;
    use delete;
}
```

Listing 10.51 Projection Behavior Definition for the My Purchase Orders Application

10.6.2 Creating a Service Definition

Next, you need to create a service definition named `ZRAP_PO_Own_M` with the description “Service definition, My Purchase Orders” based on the previously created CDS projection view `ZC_RAP_PurchaseOrderOwn_M`. Proceed as described in Chapter 9, Section 9.4.1. Specify the semantic alias `PurchaseOrder` for the exposed CDS entity `ZC_RAP_PurchaseOrderOwn_M` using `as`. In Listing 10.52 you can see the complete SDL source code.

```
@EndUserText.label: 'Service definition, My Purchase Orders'
define service ZRAP_PO_Own_M {
    expose ZC_RAP_PurchaseOrderOwn_M as PurchaseOrder;
}
```

Listing 10.52 Service Definition `ZRAP_PO_Own_M`

10.6.3 Creating a Service Binding

Then, create a service binding named `ZUI_RAP_PO_OWN_M_O2` for service definition `ZRAP_PO_Own_M`. Proceed as described in Chapter 9, Section 9.4.2. Use the value **OData V2 - UI** as the **Binding Type**.

Then, publish the service binding using the **Publish** button. You can see the already published service binding in Figure 10.11.

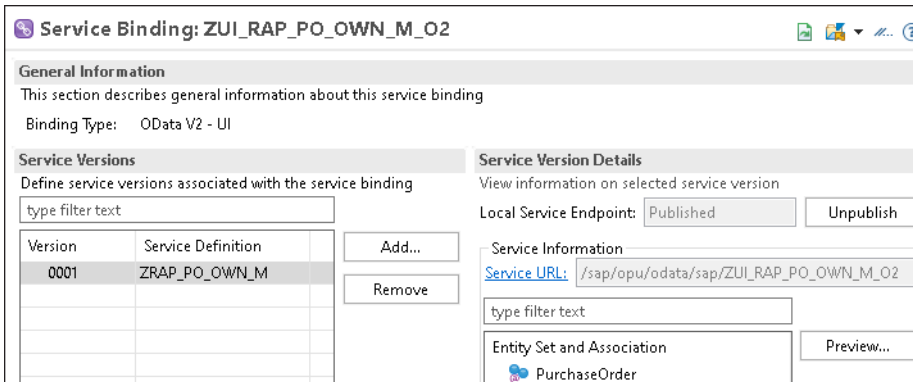


Figure 10.11 Published Service Binding for the Purchase Order Entity Set

10.7 Implementing Authorization Checks

In this section, you'll implement authorization checks for the fast purchase order entry application.

10.7.1 Access Controls for Read Access

CDS access controls protect the records of a CDS entity from unauthorized access. However, this access protection is only effective *locally* (see Chapter 2, Section 2.5). The CDS projection view `ZC_RAP_PurchaseOrderOwn_M` created in Section 10.6.1 uses the CDS view `ZI_RAP_PurchaseOrder_M` as a data source, which in turn is based on the standard view `I_PurchaseOrderItemAPI01`. Thus, the CDS access control of the same name for the order item in the VDM does *not* take effect for our entities, `ZI_RAP_PurchaseOrder_M` or `ZC_RAP_PurchaseOrder_M`. You need your own access control. In contrast, the respective access controls of the associated CDS entities take effect for read accesses defined via associations.

Custom CDS access control

Since our RAP business object is based on standard CDS entities of the VDM, you can use the CDS access controls defined there for read accesses. You can check directly in the system if there's an access control for the used CDS entity `I_PurchaseOrderItemAPI01`:

Searching for standard access controls

- 1. Open the CDS entity `I_PurchaseOrderItemAPI01` in ADT.
- 2. Run a where-used list using the shortcut `[Ctrl] + [Shift] + [G]`.
- 3. In the search result, you can see the CDS access control of the same name `I_PurchaseOrderItemAPI01 (Access Control)` (see Figure 10.12).

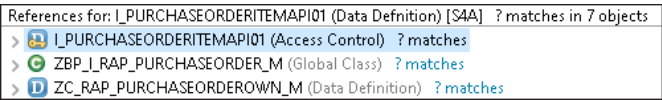


Figure 10.12 Result of Searching for a CDS Access Control for the `I_PurchaseOrderAPI01` CDS Entity

Creating access control

You can create your own access control for the CDS entity `ZI_RAP_PurchaseOrder_M` based on this standard access control:

- 1. Open the context menu of the CDS entity in the **Project Explorer** and select the **New Access Control** entry.
- 2. In the **Name** field, you should use the same name as the CDS entity you want to protect (here, `ZI_RAP_PurchaseOrder_M`).
- 3. Your access control should inherit from CDS entity `I_PurchaseOrderItemAPI01` (inheriting conditions from entity) so that you can use the authorization checks defined there directly and don't need to copy them. The prerequisite for this is that the fields relevant for the authorization check must be part of the inheriting CDS entity, such as the `Plant` field, for example. You can see the data control language (DCL) source code for your access control in Listing 10.53.

```
@EndUserText.label: 'CDS access control'
@MappingRole: true
define role ZI_RAP_PurchaseOrder_M {
    grant select on ZI_RAP_PurchaseOrder_M
    where
    inheriting conditions from entity I_PurchaseOrderItemAPI01;
}
```

Listing 10.53 Access Control for `ZI_RAP_PurchaseOrder_M` CDS Entity

- 4. Secure and activate the access control.

Creating access control for the projection view

Now, you should create another CDS access control for the CDS projection view `ZC_RAP_PurchaseOrderOwn_M`. This access control inherits the authorization check of the projected CDS view `ZI_RAP_PurchaseOrder_M`. You can see the DCL source code in Listing 10.54. Save and activate this access control as well.

```
@EndUserText.label: 'CDS access control'
@MappingRole: true
define role ZC_RAP_PurchaseOrderOwn_M {
    grant select on ZC_RAP_PurchaseOrderOwn_M
    where
        inheriting conditions from entity ZI_RAP_PurchaseOrder_M;
}
```

Listing 10.54 CDS Access Control for CDS Projection View ZC_RAP_PurchaseOrderOwn_M

10.7.2 Access Controls for Write Access

Now, open the behavior definition of the CDS root entity ZI_RAP_PurchaseOrder_M and add the statement `authorization master (instance)`. This way you can enable instance-based authorization checking (see Listing 10.55). Save and activate the behavior definition afterwards.

Declaring an authorization check

```
...
define behavior for ZI_RAP_PURCHASEORDER_M alias PurchaseOrder
lock master
late numbering
authorization master ( instance )
etag master LastChangeDateTime
...
```

Listing 10.55 Declaring Authorization Checks

In the implementation class in the behavior pool, you can create the method to run the authorization check using the quick fix function. Listing 10.56 shows the method declaration.

Method for the authorization check

```
...
METHODS get_instance_authorizations
FOR INSTANCE AUTHORIZATION
IMPORTING keys REQUEST requested_authorizations FOR PurchaseOrder
RESULT result.
...
```

Listing 10.56 Method Declaration for the Instance-Based Authorization Check

You can determine the authorization objects used with the respective activities from the BAPI documentation. In our example, these are the following authorization objects: M_BEST_BSA, M_BEST_EKO, and M_BEST_EKG. Even if the BAPIs themselves perform authorization checks, you should implement

these checks in the RAP business object. This way, you can use the capabilities of the RAP framework and ensure, for example, that no other person can make unauthorized changes to the transaction buffer.



Tip: Using an Authorization Trace

To find out which authorizations are being used, you can also activate an authorization trace in Transaction STAUTHTRACE and reproduce the desired behavior in the standard application, for example, creating a purchase order. In the SAP BTP, ABAP environment, an SAP Fiori app is available for activating and evaluating authorization checks (see Chapter 12).

Authorization for the delete operation

Now, open the behavior pool and implement the `get_instance_authorizations` method to check the specified authorization objects (see Listing 10.57). Then activate the behavior pool.

```
CLASS lhc_PurchaseOrder IMPLEMENTATION.
...
METHOD get_instance_authorizations.
  READ ENTITY ZI_RAP_PurchaseOrder_M
    ALL FIELDS
    WITH CORRESPONDING #( keys )
    RESULT DATA(pos_for_auth_check).
  " 02 - Change => Is deletion allowed
  LOOP AT pos_for_auth_check ASSIGNING FIELD-SYMBOL(<po_for_auth_
check>).
    AUTHORITY-CHECK OBJECT 'M_BEST_EKO'
      ID 'EKORG' FIELD <po_for_auth_check>-PurchasingOrganization
      ID 'ACTVT' FIELD '02'.
    IF sy-subrc <> 0.
      APPEND VALUE #( PurchaseOrder = <po_for_auth_check>-
        PurchaseOrder
          %delete = if_abap_behv=>auth-
            unauthorized ) TO result.
    CONTINUE.
  ENDIF.
  " ... Check authorization object M_BEST_EKG, M_BEST_BSA
ENDLOOP.
ENDMETHOD.
...
```

Listing 10.57 Implementation of Instance-Based Authorization Checks

Authorization for the create operation

In the ABAP RESTful application programming model, you can protect the create operation from unauthorized access using the global authorization

check. In our use case, however, the check must be dependent on the purchasing organization, and data for the instances is not available in the global authorization check. For this reason, you *can't* use the global authorization check here, but must provide the `create` operation in the behavior definition with a `precheck` and perform the authorization check there (see Listing 10.58).

```
define behavior for ZI_RAP_PurchaseOrder_M alias PurchaseOrder
...
{
  create ( precheck );
  internal update;
  delete;
...
}
```

Listing 10.58 Create Operation with the Precheck Option

Now, save and activate the behavior definition and create a `FOR PRECHECK` method in the implementation class using the quick fix function. Implement them as shown in Listing 10.59.

**Implementing the
FOR PRECHECK
method**

```
METHOD precheck_create.
  DATA(org_data) = read_org_data( ).
  LOOP AT entities ASSIGNING FIELD-SYMBOL(<po>).
    AUTHORITY-CHECK OBJECT 'M_BEST_EKO'
      ID 'EKORG' FIELD org_data-purch_org
      ID 'ACTVT' FIELD '01'.
    IF sy-subrc <> 0.
      failed-purchaseorder = VALUE #( BASE failed-purchaseorder
        ( %cid = <po>-%cid
          %create = if_abap_behv=>auth-unauthorized ) ).
      reported-purchaseorder = VALUE #(
        BASE reported-purchaseorder
        ( %cid = <po>-%cid
          %msg = me->new_message( severity =
            if_abap_behv_message=>severity-error
            id = 'ZRAP_PO'
            number = '005' ) ) ).
      CONTINUE.
    ENDIF.
    " Check authorization object M_BEST_EKG
  ...
```

```
ENDLOOP.  
ENDMETHOD.
```

Listing 10.59 Implementation of the FOR PRECHECK Method for Authorization Checks

Then, save and activate the behavior pool again.

10.8 Creating an SAP Fiori Elements User Interface

Based on the previously created projection layer and the published service binding, in this section we'll create an SAP Fiori elements application that allows users to select, create, view, and delete their purchase orders.

10.8.1 Creating a Metadata Extension

Allowing metadata extensions

UI annotations are best placed in a metadata extension to swap out the presentation layer of the application in the backend into a separate artifact. To make this possible, you first need to add the annotation `@Metadata.allowExtensions: true` to the CDS projection view (see Listing 10.60). Then, activate the CDS entity.

```
@AccessControl.authorizationCheck: #CHECK  
@EndUserText.label: 'My Purchase Orders'  
@Metadata.allowExtensions: true  
define root view entity ZC_RAP_PurchaseOrderOwn_M  
...
```

Listing 10.60 Allowing Metadata Extensions to the CDS Projection View



Allowing Metadata Extensions

We'd like to point out here that adding this annotation isn't done only out of technical necessity, but is done deliberately. By using it, you allow the addition of a metadata extension to a CDS entity and thus to open this CDS entity for adaptations "from outside." If the RAP application is made available outside the sphere of influence of your organization or organizational unit, the metadata extension can be used there as well, which is why you must support the technical contract throughout the application lifecycle.

Creating the metadata extension

Now, you can create a metadata extension for CDS projection view `ZC_RAP_PurchaseOrderOwn_M`. Proceed as described in Chapter 9, Section 9.5. You can

see the CDS source code for the metadata extension with the UI annotations in Listing 10.61.

```
@Metadata.layer: #CORE
@UI.headerInfo: { typeName: 'My purchase order',
                  typeNamePlural: 'My purchase order' }
annotate view ZC_RAP_PurchaseOrderOwn_M with
{
  @UI.facet: [{ id: 'Identification',
                type: #IDENTIFICATION_REFERENCE,
                label: 'Bestellung',
                position: 10,
                purpose: #STANDARD }]

  @UI.selectionField: [{ position: 10 }]
  @UI.lineItem: [{ position: 10 }]
  @UI.identification: [{ position: 10 }]
  PurchaseOrder;

  @UI.selectionField: [{ position: 20 }]
  @UI.lineItem: [{ position: 20 }]
  PurchaseOrderDate;

  @UI.selectionField: [{ position: 30 }]
  @UI.lineItem: [{ position: 30 }]
  @UI.identification: [{ position: 20 }]
  Material;

  @UI.lineItem: [{ position: 40 }]
  @UI.identification: [{ position: 40 }]
  OrderQuantity;
}
```

Listing 10.61 Metadata Extension with UI Annotations

Then, activate the metadata extension. After that, you can return to the service binding and test the UI annotations using the preview function for CDS entity ZC_RAP_PurchaseOrderOwn_M. You'll notice when creating a purchase order using the SAP Fiori elements user interface that there's no search help for the **Material** input field. However, we want to give users the opportunity to directly select those materials that are valid for fast order entry via that type of a search help.

To do this, create a new CDS entity based on CDS entity ZI_RAP_PO_Material,

Testing the creation
of a purchase order

Creating a CDS
entity for the search
help

must open the context menu for CDS entity `ZI_RAP_PO_Material` and select **New Data Definition**. Choose “`ZI_RAP_PO_MaterialActive_VH`” as the name and assign the description “Valid materials, search help.” You can see the source code of the search help view in Listing 10.62.

```
...
define view entity ZI_RAP_PO_MaterialActive_VH
  as select from ZI_RAP_PO_Material
{
  @ObjectModel.text.association: '_MaterialText'
  key Material,
    _Material.MaterialBaseUnit as MaterialBaseUnit,
    @Consumption.hidden: true
    _Material.Material          as MaterialForText,
    _Material._Text            as _MaterialText
}
where
  IsActive = 'X'
```

Listing 10.62 CDS Entity as a Search Help for Valid Materials

The material short text as descriptive text for the material number gets implemented via the annotation `@ObjectModel.text.association: '_MaterialText'`. Matching the use case of the search help, the CDS entity selects only valid materials and also implements the condition `IsActive = 'X'` in the `where` clause. Save and activate the CDS entity created for the search help.

**Including a UI
annotation for the
search help**

Include the CDS entity in the metadata extension via the UI annotation `@Consumption.valueHelpDefinition` as a search help for the `Material` field (see Listing 10.63). This is comparable to the classic search help binding in structured data types in the ABAP dictionary.

```
...
@Consumption.valueHelpDefinition: [{ entity: {
  name : 'ZI_RAP_PO_MaterialActive_VH',
  element: 'Material' },
  additionalBinding: [{ element: 'MaterialBaseUnit',
    localElement:
      'PurchaseOrderQuantityUnit',
    usage: #RESULT }] } ]
Material;
...
```

Listing 10.63 Search Help Binding for the Material Field

Save and activate the metadata extension afterwards. You can now use the preview function for the `PurchaseOrder` entity set in the service binding again to test the search help.

CDS Entity for Search Help in Service Binding

If you open the service binding again, you'll notice that the CDS entity for the search help is now also automatically exposed there, without you having to add it manually. This is necessary because the user interface must also display the search help remotely via OData and perform the data selection.



10.8.2 Generating and Deploying the Application

Now you can create an SAP Fiori elements application based on the published service binding using SAP Fiori tools in Visual Studio Code. Proceed as described in Chapter 9, Section 9.9.

Open Visual Studio Code and start the application generator. In the course of the wizard, select the parameters from Table 10.5 for generating the application. Based on this information, the SAP Fiori tools will generate your application.

Generating the application

Range • Parameters	Value
Floorplan Selection • Floorplan	List Report Object Page
Data Source and Service Selection • Service	ZUI_RAP_PO_OWN_M_02
Entity Selection • Main entity	PurchaseOrder
Project Attributes • Module name	purchaseorderui
Project Attributes • Application title	Fast order entry
Deployment Configuration • Name	ZPO_OWN_UI

Table 10.5 Values for Generating the SAP Fiori Elements Application

Deploying the application in the ABAP backend is also done as described in Chapter 9, Section 9.9.

Deployment

Tip: Possible Expansion Stages

If you wish, you can further extend the application you developed in this chapter. Possible expansion stages could be:



- Using draft handling (special feature for late numbering: additional key field `DRAFTUUID` in draft table necessary)
- Creating an EML consumer to create purchase orders based on the RAP business object.
- Representing the valid material as a separate RAP business object and generating a separate SAP Fiori elements application for this object.
- Inserting another projection layer for the backend administration of orders, in which, for example, more fields are displayed for agents or in which there's an additional "Release purchase order" action.
- Creating a new SAP Fiori elements application based on this projection layer.