S4D437

Transactional Apps with the ABAP RESTful Application Programming Model (RAP)

EXERCISES AND SOLUTIONS

Course Version: 24
Exercise Duration: 6 Hours 35 Minutes

SAP Copyrights, Trademarks and Disclaimers

Typographic Conventions

American English is the standard used in this handbook.

The following typographic conventions are also used.

This information is displayed in the instructor's presentation	🖥
Demonstration	➤
Procedure	1 2 3
Warning or Caution	⚠
Hint	💡
Related or Additional Information	»
Facilitated Discussion	💬
User interface control	*Example text*
Window title	*Example text*

Contents

Unit 1
Exercise 1

Define a CDS-Based Data Model

In this exercise, you create copies of the repository objects that define a CDS-based data model.

Note:
In this exercise, replace ## with your group number.

Table 1: Template

Repository Object Type	Repository Object ID
Database Table Definition	*/LRN/437T_TRAVEL*
Data Definition	*/LRN/437T_R_TRAVEL*
ABAP Class	*/LRN/CL_437T_TRAVEL_FILL*

Table 2: Solution

Repository Object Type	Repository Object ID
Database Table Definition	*/LRN/437A_TRAVEL*
Data Definition	*/LRN/437A_R_TRAVEL*

Task 1: Prepare the Project

In Eclipse, create an ABAP Cloud project and add package **`/LRN/S4D437_EXERCISE`** to the list of favorite packages. In the ABAP Cloud project, create a new package under the super package **`ZLOCAL`** (suggested name: **`ZS4D437_##`**) and add this package also to the favorite packages.

1. Open Eclipse, switch to the ABAP perspective and create an *ABAP Cloud* project.
2. Add package **`/LRN/S4D437_EXERCISE`** to the list of *Favorite Packages*.
3. In your ABAP Cloud project, create a new package with the attributes listed in the table. When prompted for a transport, choose the transport request in which you are involved. If no transport request is listed, create a new request.

Field	Value
Name	**`ZS4D437_##`**.
Description	**`RESTful Application Programming`**
Add to favorite packages	Checked

Field	Value
Superpackage	**`ZLOCAL`**
Package Type	**`Development`**
Software Component	**`ZLOCAL`**
Application Component	Leave this field blank.
Transport Layer	Leave this field blank.

Task 2: Copy a Database Table

Create a copy of the database table **`/LRN/437T_TRAVEL`** (suggested name: **`Z##_TRAVEL`**) and activate it.

1. In the *Project Explorer*, navigate to the **`/LRN/437T_TRAVEL`** database table in the **`/LRN/S4D437_TEMPLATE`** package.
2. Create a copy of the database table and place it in your package **`ZS4D437_##`**.
3. Activate the new database table.

Task 3: Fill the Database Table

Create a copy of ABAP class **`/LRN/CL_437T_TRAVEL_FILL`** (suggested name: **`ZCL_##_TRAVEL_FILL`**). Specify the name of your database table as the value of the **`c_travel_table`** constant. Then activate and execute the class.

1. Create a copy of ABAP class **`/LRN/CL_437T_TRAVEL_FILL`** and place it in your package **`ZS4D437_##`**.
2. Change the value of the **`c_travel_table`** constant to the name of your database table.
3. Activate the ABAP class and execute it as a console application.
4. Open your database table in the *Data Preview* tool, and confirm that it contains data.

Task 4: Copy a Root View Entity

Create a copy of data definition **`/LRN/437T_R_TRAVEL`** (suggested name: **`Z##_R_TRAVEL`**). Adjust the view definition to read from your database table **`Z##_TRAVEL`**.

1. Create a copy of data definition **`/LRN/437T_R_TRAVEL`** (suggested name: **`Z##_R_Travel`**), assign it to your own package and use the same transport request as before.
2. Edit the data definition. Adjust the view name to use a mixture of upper case and lower case (**`Z##_R_Travel`**) and make the view read from your own database table.
3. Activate the data definition.
4. Test the CDS view by opening it in the *Data Preview* tool.

Unit 1
Solution 1

Define a CDS-Based Data Model

In this exercise, you create copies of the repository objects that define a CDS-based data model.

Note:
In this exercise, replace ## with your group number.

Table 1: Template

Repository Object Type	Repository Object ID
Database Table Definition	*/LRN/437T_TRAVEL*
Data Definition	*/LRN/437T_R_TRAVEL*
ABAP Class	*/LRN/CL_437T_TRAVEL_FILL*

Table 2: Solution

Repository Object Type	Repository Object ID
Database Table Definition	*/LRN/437A_TRAVEL*
Data Definition	*/LRN/437A_R_TRAVEL*

Task 1: Prepare the Project

In Eclipse, create an ABAP Cloud project and add package **`/LRN/S4D437_EXERCISE`** to the list of favorite packages. In the ABAP Cloud project, create a new package under the super package **`ZLOCAL`** (suggested name: **`ZS4D437_##`**) and add this package also to the favorite packages.

1. Open Eclipse, switch to the ABAP perspective and create an *ABAP Cloud* project.

   a) Open Eclipse.

   b) Choose *Window* → *Perspective* → *Open Perspective* → *Other...*.

   c) In the dialog box, choose *ABAP* with a double-click.

   d) Choose *File* → *New* → *ABAP Cloud Project*.

   e) On the *New ABAP Cloud Project* dialog, choose the *Extract* link.

   f) Choose *Import...* and select the file containing the service key that you have been given.

   g) On the *Extract Service Instance URL* dialog, choose *Copy to Clipboard*.

h) Choose *Close*.

i) Paste the copied URL into the *ABAP Service Instance URL* field.

j) Choose *Next*.

k) Choose *Open Logon Page in Browser*.

l) On the browser page that opens, log on with the user and password that you have been given.

m) When you see the message *You have been successfully logged on*, close the browser window and return to Eclipse.

n) To finish creating the project, choose *Finish*.

2. Add package **`/LRN/S4D437_EXERCISE`** to the list of *Favorite Packages*.

a) In the *Project Explorer* on the left, expand your ABAP Cloud project.

b) Right-click *Favorite Packages* and choose *Add Package...*.

c) In the search field, enter **`/LRN/S4D437`**.

d) From the list of matching items, select */LRN/S4D437_EXERCISE* and choose *OK*.

3. In your ABAP Cloud project, create a new package with the attributes listed in the table. When prompted for a transport, choose the transport request in which you are involved. If no transport request is listed, create a new request.

Field	Value
Name	**`ZS4D437_##`**.
Description	**`RESTful Application Programming`**
Add to favorite packages	Checked
Superpackage	**`ZLOCAL`**
Package Type	**`Development`**
Software Component	**`ZLOCAL`**
Application Component	Leave this field blank.
Transport Layer	Leave this field blank.

a) In the *Project Explorer*, right-click on your ABAP Cloud project and choose *New → ABAP Package* .

b) Enter the package name **`ZS4D437_##`** where **`##`** is your group number.

c) Enter the description **`RESTful Application Programming`**.

d) Select the *Add to favorite packages* checkbox.

e) Enter the superpackage **`ZLOCAL`**.

f) Ensure that the *Package Type* is set to **`Development`**.

g) Choose *Next*.

h) Ensure that the *Software Component* is set to **`ZLOCAL`**.

i) Ensure that the *Application Component* is empty.

j) Ensure that the *Transport Layer* is empty.

k) Choose *Next*.

l) Check if there is a transport request listed under the option *Choose from requests in which I am involved*. If so, choose this option. If the list is empty, choose the option *Create a new request* and enter a request description, for example, **`ABAP Exercises`**.

m) Choose *Finish*.

Task 2: Copy a Database Table

Create a copy of the database table **`/LRN/437T_TRAVEL`** (suggested name: **`Z##_TRAVEL`**) and activate it.

1. In the *Project Explorer*, navigate to the **`/LRN/437T_TRAVEL`** database table in the **`/LRN/S4D437_TEMPLATE`** package.

   a) In the *Project Explorer*, expand node */LRN/S4D437_EXERCISE → /LRN/S4D437_TEMPLATE → Dictionary → Database Tables*.

2. Create a copy of the database table and place it in your package **`ZS4D437_##`**.

   a) Right-click **`/LRN/437T_TRAVEL`** and select *Duplicate...*.

   b) Enter the name of your own package and the suggested name for the new database table. Choose *Next*.

   c) Select the same transport request as before and choose *Finish*.

3. Activate the new database table.

   a) In the Eclipse toolbar, choose *Activate* or press **`Ctrl + F3`**.

Task 3: Fill the Database Table

Create a copy of ABAP class **`/LRN/CL_437T_TRAVEL_FILL`** (suggested name: **`ZCL_##_TRAVEL_FILL`**). Specify the name of your database table as the value of the **`c_travel_table`** constant. Then activate and execute the class.

1. Create a copy of ABAP class **`/LRN/CL_437T_TRAVEL_FILL`** and place it in your package **`ZS4D437_##`**.

   a) In the *Project Explorer* view, locate class **`/LRN/CL_437T_TRAVEL_FILL`** in package **`/LRN/S4D437_TEMPLATE`** and right-click it to open the context menu.

   b) From the context menu, select *Duplicate...*.

   c) Enter the name of your package and the name for the copy, then choose *Next*.

   d) Assign the new object to a transport request and choose *Finish*.

2. Change the value of the **`c_travel_table`** constant to the name of your database table.

   a) Scroll down to the code row starting with `CONSTANTS c_travel_table`.

   b) In the literal after VALUE, replace **`##`** with your group number.

3. Activate the ABAP class and execute it as a console application.

a) Press **Ctrl + F3** to activate the development object.

b) Press **F9** to execute the ABAP class as a console application.

4. Open your database table in the *Data Preview* tool, and confirm that it contains data.

   a) Right-click anywhere in the source code of the database table definition and choose *Open with → Data Preview*, or press **F8**.

Task 4: Copy a Root View Entity

Create a copy of data definition **/LRN/437T_R_TRAVEL** (suggested name: **Z##_R_TRAVEL**). Adjust the view definition to read from your database table **Z##_TRAVEL**.

1. Create a copy of data definition **/LRN/437T_R_TRAVEL** (suggested name: **Z##_R_Travel**), assign it to your own package and use the same transport request as before.

   a) In the *Project Explorer* view, locate the data definition **/LRN/437T_R_TRAVEL** in the **/LRN/S4D437_TEMPLATE** package and right-click it to open the context menu.

   b) From the context menu, select *Duplicate...*.

   c) Enter the name of your package and the name for the copy. Choose *Next*.

   d) Assign the new object to a transport request and choose *Finish*.

2. Edit the data definition. Adjust the view name to use a mixture of upper case and lower case (**Z##_R_Travel**) and make the view read from your own database table.

   a) Adjust the code as follows:

```
define root view entity Z##_R_Travel
  as select from z##_travel
```

3. Activate the data definition.

   a) In the Eclipse toolbar, choose *Activate* or press **Ctrl + F3**.

4. Test the CDS view by opening it in the *Data Preview* tool.

   a) Right-click anywhere in the source code of the data definition and choose *Open With → Data Preview* or press **F8**.

Unit 1
Exercise 2

Define and Preview an OData UI Service

In this exercise, you create a projection of your CDS-based data model and enrich it with search metadata and UI metadata. Based on this projection, you define and bind an OData V2 UI Service and preview the result.

Caution:
This exercise uses version 2 of the OData protocol, as SAP Fiori does not currently support non-draft scenarios with the newer OData version 4.

Please note that SAP generally recommends implementing draft-enabled applications based on OData version 4. However, as the draft concept has not yet been covered in the course, OData version 2 is used here for didactic reasons.

Note:
In this exercise, replace ## with your group number.

Table 3: Solution

Repository Object Type	Repository Object ID
Data Definition (Projection)	*/LRN/437A_C_TRAVEL*
Metadata Extension	*/LRN/437A_C_TRAVEL*
Service Definition	*/LRN/437A_TRAVEL*
Service Binding	*/LRN/437A_UI_TRAVEL_O2*

Task 1: Create a CDS Projection View

Create a CDS projection view (suggested name: **`Z##_C_Travel`**), that reads all fields from your data model view **`Z##_R_Travel`**. Make sure that the projection view is classified as root view and that it specifies the provider contract for a **`transactional query`**.

1. Create a new data definition (suggested name: **`Z##_C_Travel`**) that defines a CDS view entity that is a projection on your data model view **`Z##_R_Travel`**. Assign it to your own package and use the same transport request as before.
2. Adjust the definition of the projection view. Add keyword **`root`**, which is needed because the projection view reads from a data model view that is a root view.
3. Use a quick fix to add a **`provider contract transactional_query`** clause.
4. Activate and test the CDS projection view.

Task 2: Enrich the CDS Projection View with Metadata

Enrich your projection view with search metadata and UI metadata. For the UI metadata, create a metadata extension (suggested name: **`Z##_C_TRAVEL`**) and copy the annotations from the template metadata extension **`/LRN/437T_C_TRAVEL`**.

1. Edit the data definition of your **`Z##_C_Travel`** projection view and enable the search functionality with **`Description`** and **`CustomerID`** as the default search elements.
2. For your projection view, allow extensions with metadata extensions.
3. Activate the data source.
4. Create a new metadata extension (suggested name: **`Z##_C_TRAVEL`**). Assign it to your own package and use the same transport request as before.
5. In the new metadata extension, set the metadata layer to **`#CORE`**.
6. Open the template metadata extension **`/LRN/437T_C_TRAVEL`** and copy the UI element annotations to your own metadata extension.
7. Similarly, copy the view annotation group starting with @*UI* from the template metadata extension to your own metadata extension.
8. Activate the metadata extension.

Task 3: Create an OData UI Service

Create a service definition (suggested name: **`Z##_TRAVEL`**) and a service binding (suggested name: **`Z##_UI_TRAVEL_O2`**), that exposes your **`Z##_C_Travel`** projection view as an OData V2 UI Service.

1. Create a service definition (suggested name: **`Z##_TRAVEL`**) for your **`Z##_C_Travel`** projection view. Assign it to your package **`ZS4D437_##`** and use the same transport request as before.
2. Activate the service definition.
3. Create a service binding (suggested name: **`Z##_UI_TRAVEL_O2`**) that binds your service **`Z##_TRAVEL`** as an OData V2 UI Service.
4. Activate the service binding.

Task 4: Publish and Test the OData UI Service

Publish the local service endpoint and preview the service as an SAP Fiori elements app.

1. In your service binding, publish the local service endpoint.
2. Preview the service as an SAP Fiori elements app.

Unit 1
Solution 2

Define and Preview an OData UI Service

In this exercise, you create a projection of your CDS-based data model and enrich it with search metadata and UI metadata. Based on this projection, you define and bind an OData V2 UI Service and preview the result.

Caution:
This exercise uses version 2 of the OData protocol, as SAP Fiori does not currently support non-draft scenarios with the newer OData version 4.

Please note that SAP generally recommends implementing draft-enabled applications based on OData version 4. However, as the draft concept has not yet been covered in the course, OData version 2 is used here for didactic reasons.

Note:
In this exercise, replace ## with your group number.

Table 3: Solution

Repository Object Type	Repository Object ID
Data Definition (Projection)	*/LRN/437A_C_TRAVEL*
Metadata Extension	*/LRN/437A_C_TRAVEL*
Service Definition	*/LRN/437A_TRAVEL*
Service Binding	*/LRN/437A_UI_TRAVEL_O2*

Task 1: Create a CDS Projection View

Create a CDS projection view (suggested name: **`Z##_C_Travel`**), that reads all fields from your data model view **`Z##_R_Travel`**. Make sure that the projection view is classified as root view and that it specifies the provider contract for a **`transactional query`**.

1. Create a new data definition (suggested name: **`Z##_C_Travel`**) that defines a CDS view entity that is a projection on your data model view **`Z##_R_Travel`**. Assign it to your own package and use the same transport request as before.

   a) In the *Project Explorer* view on the left, open the context menu for the data definition of your data model view.

   b) From the context menu, select *New Data Definition*.

   c) Enter the name of the new data definition and a description and choose *Next*.

   d) Select the same transport request as before and choose *Next*.

   e) From the list of templates, select *Projection View (creation) → defineProjectionView* and choose *Finish*.

2. Adjust the definition of the projection view. Add keyword **`root`**, which is needed because the projection view reads from a data model view that is a root view.

   a) Add keyword **`root`** between keywords **`define`** and **`view`**.

3. Use a quick fix to add a **`provider contract transactional_query`** clause.

   a) In the code row starting with `define root view entity`, place the cursor on the name of your projection view and press **`Ctrl + 1`**. Alternatively, choose the warning icon with light bulb left from that code row.

   b) From the list of available quick fixes, choose *Add PROVIDER CONTRACT TRANSACTIONAL_QUERY clause*.

4. Activate and test the CDS projection view.

   a) Press **`Ctrl + F3`** to activate the repository object.

   b) Press **`F8`** to open the *Data Preview* tool.

Task 2: Enrich the CDS Projection View with Metadata

Enrich your projection view with search metadata and UI metadata. For the UI metadata, create a metadata extension (suggested name: **`Z##_C_TRAVEL`**) and copy the annotations from the template metadata extension **`/LRN/437T_C_TRAVEL`**.

1. Edit the data definition of your **`Z##_C_Travel`** projection view and enable the search functionality with **`Description`** and **`CustomerID`** as the default search elements.

   a) In the data definition of your projection view, insert the `@Search.searchable: true` view annotation before the `define root view entity` statement.

   b) Insert the element annotation `@Search.defaultSearchElement: true` in front of the **`Description`** and the **`CustomerID`** view elements.

2. For your projection view, allow extensions with metadata extensions.

   a) In the data definition of your projection view, add the `@Metadata.allowExtensions: true` view annotation.

3. Activate the data source.

   a) Press **`Ctrl + F3`** to activate the development object.

4. Create a new metadata extension (suggested name: **`Z##_C_TRAVEL`**). Assign it to your own package and use the same transport request as before.

   a) In the *Project Explorer*, right-click the data definition **`Z##_C_Travel`** and select *New Metadata Extension*.

   b) Enter the name of the new development object and a description and choose *Next*.

   c) Select the same transport request as before and choose *Finish*.

5. In the new metadata extension, set the metadata layer to **`#CORE`**.

   a) Change the value of the *@Metadata.layer* annotation from **`layer`** to **`#CORE`**.

6. Open the template metadata extension **/LRN/437T_C_TRAVEL** and copy the UI element annotations to your own metadata extension.
   a) Choose **Ctrl + Shift + A**.
   b) Enter **/LRN/437T** in the *Search* field.
   c) From the list of matching items, choose */LRN/437T_C_TRAVEL (Metadata Extension)* and choose *OK*.
   d) Select the entire code that follows the **with** addition (including the curly brackets) and press **Ctrl + C** to copy it to the clipboard.
   e) Return to your own metadata extension, select the pair of curly brackets after the **with** addition and press **Ctrl + V** to replace it with the content of the clipboard.
7. Similarly, copy the view annotation group starting with *@UI* from the template metadata extension to your own metadata extension.
   a) Return to metadata extension **/LRN/437T_C_TRAVEL**.
   b) Select the code between the `@Metadata.layer` annotation and the `annotate view` statement and press **Ctrl + C**.
   c) Return to your own metadata extension, place the cursor above the `annotate view` statement, and press **Ctrl + V**.
8. Activate the metadata extension.
   a) Press **Ctrl + F3** to activate the development object.

Task 3: Create an OData UI Service

Create a service definition (suggested name: **Z##_TRAVEL**) and a service binding (suggested name: **Z##_UI_TRAVEL_O2**), that exposes your **Z##_C_Travel** projection view as an OData V2 UI Service.

1. Create a service definition (suggested name: **Z##_TRAVEL**) for your **Z##_C_Travel** projection view. Assign it to your package **ZS4D437_##** and use the same transport request as before.
   a) In the *Project Explorer* view on the left, open the context menu for the data definition of your projection view.
   b) From the context menu, select *New Service Definition*.
   c) Enter the name of the service definition and a description and choose *Next*.
   d) Select the same transport request as before and choose *Next*.
   e) From the list of templates, select *defineService* and choose *Finish*.
2. Activate the service definition.
   a) Press **Ctrl + F3** to activate the development object.
3. Create a service binding (suggested name: **Z##_UI_TRAVEL_O2**) that binds your service **Z##_TRAVEL** as an OData V2 UI Service.
   a) In the *Project Explorer* view on the left, open the context menu for your service definition.
   b) From the context menu, select *New Service Binding*.

c) Enter the name of the service binding and a description.

d) Set the *Binding Type* to *OData V2 - UI* and choose *Next*.

e) Select the same transport request as before and choose *Finish*.

4. Activate the service binding.

a) Press **`Ctrl + F3`** to activate the development object.

Task 4: Publish and Test the OData UI Service

Publish the local service endpoint and preview the service as an SAP Fiori elements app.

1. In your service binding, publish the local service endpoint.

a) Open your service binding.

b) On the right, under *Service Versions Detail*, choose *Publish*.

2. Preview the service as an SAP Fiori elements app.

a) Under *Entity Set and Association*, choose the name of your projection view.

b) Choose *Preview...*.

The *Preview for Fiori Elements App* opens in a new browser window or browser tab.

> Note:
> You might need to enter your user and password for the ABAP application server, when you open the preview for the first time.

Unit 2
Exercise 3

Define a Business Object and Its Behavior

In this exercise, you define a Business Object (BO) by extending your CDS-based data model with a CDS behavior definition. You then extend your data model projection with a CDS behavior projection, to make the behavior of the BO visible to your OData UI Service.

Note:
In this exercise, replace ## with your group number.

Table 4: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437B_C_TRAVEL*
ABAP Class	*/LRN/BP_437B_R_TRAVEL*

Task 1: Create a CDS Behavior Definition
Create a CDS behavior definition that defines the behavior of a Business Object (BO) with your data model view entity **`Z##_R_Travel`** as the root entity. Define the field mapping and create the behavior implementation class.

Hint:
Use the *New Behavior Definition* wizard from the context menu of the *Project Explorer*. This creates the behavior definition based on a template.

1. Create a CDS behavior definition for your **`Z##_R_Travel`** data model view. Assign the new repository object to your own package and use the same transport request as before.
2. Perform a syntax check for the new behavior definition.

   Are there any syntax errors or warnings?

   ____

   ____

   ____

3. Enable persistence for all elements of your data model by defining the mapping between table field names and CDS view element names.
4. Use a quick fix to generate the behavior implementation class.

Hint:
To use this quick fix, it is mandatory that the behavior definition is saved and active.

5. Activate the behavior implementation class.

Task 2: Create a CDS Behavior Projection

Create a CDS behavior definition that defines the behavior of your projection view **`Z##_C_Travel`** and makes the behavior of the Business Object visible for your OData UI service.

Hint:
Use the *New Behavior Definition* wizard from the context menu of the *Project Explorer*. This creates the behavior definition based on a template.

1. Use the context menu in the *Project Explorer* to create a CDS behavior definition for your projection view *Z##_C_Travel*. Assign it to your own package and use the same transport request as before.
2. Activate the CDS behavior definition.
3. Refresh the preview of the service as SAP Fiori elements app.

   What is different on the *List Report Page*?

   What is new on the *Object Page*?

Hint:
If you want, play around with the new functions a bit, but remember that the behavior definition is still incomplete and that the Business Object does not yet ensure data consistency. For example, the BO does not yet provide a unique primary key when you create a new travel. If needed, execute ABAP class **`ZCL_##_TRAVEL_FILL`** again to delete and regenerate the sample data.

Unit 2
Solution 3

Define a Business Object and Its Behavior

In this exercise, you define a Business Object (BO) by extending your CDS-based data model with a CDS behavior definition. You then extend your data model projection with a CDS behavior projection, to make the behavior of the BO visible to your OData UI Service.

Note:
In this exercise, replace ## with your group number.

Table 4: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437B_C_TRAVEL*
ABAP Class	*/LRN/BP_437B_R_TRAVEL*

Task 1: Create a CDS Behavior Definition

Create a CDS behavior definition that defines the behavior of a Business Object (BO) with your data model view entity **`Z##_R_Travel`** as the root entity. Define the field mapping and create the behavior implementation class.

Hint:
Use the *New Behavior Definition* wizard from the context menu of the *Project Explorer*. This creates the behavior definition based on a template.

1. Create a CDS behavior definition for your **`Z##_R_Travel`** data model view. Assign the new repository object to your own package and use the same transport request as before.
   a) In the *Project Explorer* view on the left, open the context menu for the data definition of your data model view.
   b) From the context menu, select *New Behavior Definition*.
   c) Enter a description and confirm the suggested *Implementation Type*, **`Managed`**. Choose *Next*. Note that you cannot change the suggested name of the new behavior definition.
   d) Select the same transport request as before and choose *Finish*.
2. Perform a syntax check for the new behavior definition.
   a) Press **`Ctrl + F2`** and analyze the *Problems* tab below the source code editor.

Are there any syntax errors or warnings?

Yes, there are warnings about the none existing class **ZBP_##_R_TRAVEL** and missing field mappings.

3. Enable persistence for all elements of your data model by defining the mapping between table field names and CDS view element names.
   a) Within the curly brackets of the behavior definition, add the keyword `mapping for` followed by the name of your database table, addition `corresponding`, and a pair of curly brackets.
   b) Within the curly brackets, list all view elements and table fields that differ in more than just uppercase or lowercase.
   c) Add the following code:

```
mapping for z##_travel corresponding
    {
      AgencyID     = agency_id;
      TravelID     = travel_id;
      CustomerID   = customer_id;
      BeginDate    = begin_date;
      EndDate      = end_date;
      ChangedAt    = changed_at;
      ChangedBy    = changed_by;
    }
```

4. Use a quick fix to generate the behavior implementation class.

> Hint:
> To use this quick fix, it is mandatory that the behavior definition is saved and active.

   a) Press **Ctrl + F3** to activate the behavior definition.
   b) Right-click the name of the class and select *Quick Fix*. Alternatively, choose the warning icon next to statement `managed`.
   c) Double-click *Create behavior implementation class zbp_##_r_travel*.
   d) Enter a description and choose *Next*.
   e) Select the same transport request as before and choose *Finish*.
5. Activate the behavior implementation class.
   a) Press **Ctrl + F3** to activate the development object.

Task 2: Create a CDS Behavior Projection

Create a CDS behavior definition that defines the behavior of your projection view **Z##_C_Travel** and makes the behavior of the Business Object visible for your OData UI service.

> Hint:
> Use the *New Behavior Definition* wizard from the context menu of the *Project Explorer*. This creates the behavior definition based on a template.

1. Use the context menu in the *Project Explorer* to create a CDS behavior definition for your projection view *Z##_C_Travel*. Assign it to your own package and use the same transport request as before.

   a) In the *Project Explorer* view on the left, open the context menu for the data definition of your projection view.

   b) From the context menu, select *New Behavior Definition*.

   c) Enter a description, confirm the *Implementation Type* **`Projection`** and choose *Next*. Note that you cannot change the suggested name of the new behavior definition.

   d) Select the same transport request as before and choose *Finish*.

2. Activate the CDS behavior definition.

   a) Press **`Ctrl + F3`**.

3. Refresh the preview of the service as SAP Fiori elements app.

   What is different on the *List Report Page*?

   There are *Create* and *Delete* buttons in the toolbar of the table.

   What is new on the *Object Page*?

   There are *Edit* and *Delete* buttons in the header area.

   Hint:
   If you want, play around with the new functions a bit, but remember that the behavior definition is still incomplete and that the Business Object does not yet ensure data consistency. For example, the BO does not yet provide a unique primary key when you create a new travel. If needed, execute ABAP class **`ZCL_##_TRAVEL_FILL`** again to delete and regenerate the sample data.

   a) If the browser window with the preview of the SAP Fiori elements app is still open, reload the page, otherwise open your service binding, choose the name of your projection view and choose *Preview...*.

Unit 2
Exercise 4

Read and Update a RAP Business Object

In this exercise, you use the Entity Manipulation Language (EML) to read and update a Business Object.

Note:
In this exercise, replace ## with your group number.

Table 5: Template

Repository Object Type	Repository Object ID
ABAP Class	*/LRN/CL_437T_EML*

Table 6: Solution

Repository Object Type	Repository Object ID
ABAP Class	*/LRN/CL_437B_EML_S1*
Behavior Definition	*/LRN/437B_R_TRAVEL*
ABAP Class (Behavior Pool)	*/LRN/BP_437B_R_TRAVEL*

Task 1: Copy Template
Create a copy of ABAP class **/LRN/CL_437T_EML** (suggested name: **ZCL_##_EML**). Replace the values of constants **C_AGENCY_ID** and **C_TRAVEL_ID** with the key values of an existing record in your database table **Z##_TRAVEL**.

1. Create a copy of ABAP class **/LRN/CL_437T_EML**, assign it to your package **ZS4D437_##** and use the same transport request as before.

2. Open your database table **Z##_TRAVEL** in the *Data Preview* tool and make a note of the key values for any one of the data records.

3. Edit the ABAP Class. Replace the placeholder values for the constants **C_AGENCY_ID** and **C_TRAVEL_ID** with the values that you determined in the previous step.

Task 2: Read Data from a Business Object
In your ABAP class, use the EML statement **READ ENTITIES** to retrieve all fields of one root entity instance of your Business Object **Z##_R_TRAVEL**. Use the **C_AGENCY_ID** and **C_TRAVEL_ID** constants as input to identify the travel that you want to read. Write an error text to the console in case the read access failed.

1. In the **if_oo_adt_classrun~main** method, implement a **READ ENTITIES** statement for your Business Object **Z##_R_TRAVEL** to read instances of the root entity.

2. Read all fields of the instance with `AgencyId = c_agency_id` and `TravelId = c_travel_id`.

   Hint:
   Use a VALUE expression to identify the instance that you want to read.

3. Store the result in a suitably typed data object (suggested name: **`travels`**).

   Hint:
   Use an inline declaration to make sure the data object has the correct type.

4. Retrieve the **`FAILED`** response, store it in a suitably typed data object (suggested name: **`failed`**) and end the ABAP statement.

   Hint:
   Use an inline declaration to make sure the data object has the correct type.

5. Evaluate the response. If `failed` is not initial, write a suitable error text to the console.
6. Use the *Tooltip Description* (**`F2`**) to analyze the type of the data object **`travels`**.

   What is the type of data object **`travels`**?

   ____
   ____
   ____

7. Use the *Tooltip Description* to analyze the type of the data object **`failed`**.

   What is the type of data object **`failed`**?

   ____
   ____
   ____

   How many components does **`failed`** have and what are the component names?

   ____
   ____
   ____

Task 3: Update the Data of a Business Object

If the read access was successful, use the EML statement MODIFY ENTITIES to change the description of the travel. Roll back your changes and write an error text to the console in case the update fails. Commit your changes if the update was successful.

1. Add an ELSE branch to the IF...ENDIF control structure and implement a MODIFY ENTITIES statement for the root entity of your business object **`Z##_R_TRAVEL`**.

2. Update the description of the instance with `AgencyId = c_agency_id` and `TravelId = c_travel_id`.

   > Hint:
   > Use a VALUE expression to identify the instance that you want to update and to specify the new description.

3. Retrieve the **`FAILED`** response and store it in existing data object **`failed`**. Then end the ABAP statement.

4. Evaluate the response. If **`failed`** is initial, execute the statement COMMIT ENTITIES and write a success text to the console. If **`failed`** is not initial, execute the statement ROLLBACK ENTITIES and write an error text to the console.

5. Activate and test your code as a console application. Display the content of your database table **`Z##_TRAVEL`** in the *Data Preview* tool to confirm that the changed description is stored on the database.

Task 4: Use Entity Alias

Define an alias for the root entity of your business object, adjust the code where necessary, and analyze the impact on the data object **`failed`**. Use the refactoring capabilities of ADT to rename the local class in the behavior pool.

1. Navigate to your behavior definition **`Z##_R_TRAVEL`** and define an alias for the root entity (suggested name: **`Travel`**).

2. Activate the behavior definition.

3. Return to the ABAP class **`ZCL_##_EML`** and perform a syntax check.

   Are there any new syntax warnings?

   ____

   ____

   ____

4. Adjust the EML statements as suggested by the syntax warnings and activate the ABAP class.

5. Display the *Tooltip Description* for data object **`failed`**.

   How did the alias definition change the type of the data object **`failed`**?

   ____

   ____

   ____

6. Edit the behavior pool and rename the local class to **`lhc_travel`**.

7. The definition of the *get_instance_authorizations* method in the behavior pool still uses the name *Z##_R_TRAVEL*. Replace this name with the alias **`Travel`**.

8. Activate the behavior pool.

Unit 2
Solution 4

Read and Update a RAP Business Object

In this exercise, you use the Entity Manipulation Language (EML) to read and update a Business Object.

Note:
In this exercise, replace ## with your group number.

Table 5: Template

Repository Object Type	Repository Object ID
ABAP Class	*/LRN/CL_437T_EML*

Table 6: Solution

Repository Object Type	Repository Object ID
ABAP Class	*/LRN/CL_437B_EML_S1*
Behavior Definition	*/LRN/437B_R_TRAVEL*
ABAP Class (Behavior Pool)	*/LRN/BP_437B_R_TRAVEL*

Task 1: Copy Template

Create a copy of ABAP class **/LRN/CL_437T_EML** (suggested name: **ZCL_##_EML**). Replace the values of constants **C_AGENCY_ID** and **C_TRAVEL_ID** with the key values of an existing record in your database table **Z##_TRAVEL**.

1. Create a copy of ABAP class **/LRN/CL_437T_EML**, assign it to your package **ZS4D437_##** and use the same transport request as before.
   a) In the *Project Explorer* view, locate class **/LRN/CL_437T_EML** in package **/LRN/S4D437_TEMPLATE** and right-click it to open the context menu.
   b) From the context menu, select *Duplicate...*.
   c) Enter the name of your package and the name for the copy. Then choose *Next*.
   d) Assign the new object to a transport request and choose *Finish*.
2. Open your database table **Z##_TRAVEL** in the *Data Preview* tool and make a note of the key values for any one of the data records.
   a) Open the database table definition in the editor.
   b) Press **F8** to open the *Data Preview* tool.

c) Make a note of the values in columns **AGENCY_ID** and **TRAVEL_ID** for any one of the displayed records.

3. Edit the ABAP Class. Replace the placeholder values for the constants **C_AGENCY_ID** and **C_TRAVEL_ID** with the values that you determined in the previous step.

   a) Edit the class as usual.

Task 2: Read Data from a Business Object

In your ABAP class, use the EML statement **READ ENTITIES** to retrieve all fields of one root entity instance of your Business Object **Z##_R_TRAVEL**. Use the **C_AGENCY_ID** and **C_TRAVEL_ID** constants as input to identify the travel that you want to read. Write an error text to the console in case the read access failed.

1. In the **if_oo_adt_classrun~main** method, implement a **READ ENTITIES** statement for your Business Object **Z##_R_TRAVEL** to read instances of the root entity.

   a) Insert the following code:

```
READ ENTITIES OF Z##_R_Travel
    ENTITY Z##_R_Travel
```

2. Read all fields of the instance with `AgencyId = c_agency_id` and `TravelId = c_travel_id`.

   Hint:
   Use a VALUE expression to identify the instance that you want to read.

   a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel
    ENTITY Z##_R_Travel
     ALL FIELDS WITH
      VALUE #( ( AgencyId = c_agency_id
                 TravelId = c_travel_id ) )
```

3. Store the result in a suitably typed data object (suggested name: **travels**).

   Hint:
   Use an inline declaration to make sure the data object has the correct type.

   a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel
    ENTITY Z##_R_Travel
     ALL FIELDS WITH
      VALUE #( ( AgencyId = c_agency_id
                 TravelId = c_travel_id ) )
      RESULT DATA(travels)
```

4. Retrieve the **FAILED** response, store it in a suitably typed data object (suggested name: **failed**) and end the ABAP statement.

> Hint:
> Use an inline declaration to make sure the data object has the correct type.

a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel
    ENTITY Z##_R_Travel
     ALL FIELDS WITH
      VALUE #( ( AgencyId = c_agency_id
                 TravelId = c_travel_id ) )
      RESULT DATA(travels)
      FAILED DATA(failed).
```

5. Evaluate the response. If `failed` is not initial, write a suitable error text to the console.

   a) After the READ ENTITIES statement, add the following code:

```
IF failed IS NOT INITIAL.
    out->write( `Error retrieving the travel` ).
ENDIF.
```

6. Use the *Tooltip Description* (**`F2`**) to analyze the type of the data object **`travels`**.

   What is the type of data object **`travels`**?

   `travels` is an internal table of derived type **`TABLE FOR READ RESULT z##_r_travel`**.

   a) To display the *Tooltip Description*, place the cursor on `travels` and press **`F2`**.

7. Use the *Tooltip Description* to analyze the type of the data object **`failed`**.

   What is the type of data object **`failed`**?

   `failed` is a structure of type **`RESPONSE FOR FAILED EARLY z##_r_travel`**.

   How many components does **`failed`** have and what are the component names?

   `failed` has just one component of name **`z##_r_travel`**. The **`z##_r_travel`** component is an internal table.

   a) To display the *Tooltip Description*, place the cursor on `failed` and press **`F2`**.

Task 3: Update the Data of a Business Object

If the read access was successful, use the EML statement MODIFY ENTITIES to change the description of the travel. Roll back your changes and write an error text to the console in case the update fails. Commit your changes if the update was successful.

1. Add an ELSE branch to the IF...ENDIF control structure and implement a MODIFY ENTITIES statement for the root entity of your business object **`Z##_R_TRAVEL`**.

a) Adjust the code as follows:

```
IF failed IS NOT INITIAL.
    out->write( `Error retrieving the travel` ).
ELSE.
    MODIFY ENTITIES OF Z##_R_Travel
     ENTITY Z##_R_Travel

ENDIF.
```

2. Update the description of the instance with `AgencyId = c_agency_id` and `TravelId = c_travel_id`.

> Hint:
> Use a VALUE expression to identify the instance that you want to update and to specify the new description.

a) Adjust the code as follows:

```
MODIFY ENTITIES OF Z##_R_Travel
    ENTITY Z##_R_Travel
     UPDATE FIELDS ( Description )
      WITH VALUE #( ( AgencyId    = c_agency_id
                      TravelId    = c_travel_id
                      Description = `My new Description` ) )
```

3. Retrieve the **`FAILED`** response and store it in existing data object **`failed`**. Then end the ABAP statement.

a) Adjust the code as follows:

```
MODIFY ENTITIES OF Z##_R_Travel
    ENTITY Z##_R_Travel
     UPDATE FIELDS ( Description )
      WITH VALUE #( ( AgencyId    = c_agency_id
                      TravelId    = c_travel_id
                      Description = `My new Description` ) )
      FAILED failed.
```

4. Evaluate the response. If **`failed`** is initial, execute the statement COMMIT ENTITIES and write a success text to the console. If **`failed`** is not initial, execute the statement ROLLBACK ENTITIES and write an error text to the console.

a) After the MODIFY ENTITIES statement, add the following code:

```
IF failed IS INITIAL.
    COMMIT ENTITIES.
    out->write( `Description successfully updated` ).
ELSE.
    ROLLBACK ENTITIES.
    out->write( `Error updating the description` ).
ENDIF.
```

5. Activate and test your code as a console application. Display the content of your database table **`Z##_TRAVEL`** in the *Data Preview* tool to confirm that the changed description is stored on the database.

a) Press **`Ctrl + F3`** to activate the development object.

b) Press **`F9`** to execute the ABAP class as a console application.

c) Open the database table and press **`F8`** to open the *Data Preview* tool.

Task 4: Use Entity Alias

Define an alias for the root entity of your business object, adjust the code where necessary, and analyze the impact on the data object **failed**. Use the refactoring capabilities of ADT to rename the local class in the behavior pool.

1. Navigate to your behavior definition **Z##_R_TRAVEL** and define an alias for the root entity (suggested name: **Travel**).

   a) In either the READ ENTITIES or the MODIFY ENTITIES statement, place the cursor on `Z##_R_Travel` and press **F3**. Alternatively, hold down the **Ctrl** key and choose `Z##_R_Travel`.

   b) In the DEFINE BEHAVIOR FOR statement, remove the comment sign (//) before the ALIAS addition.

   c) Replace the placeholder after `alias` with **Travel**.

2. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

3. Return to the ABAP class **ZCL_##_EML** and perform a syntax check.

   Are there any new syntax warnings?

   Yes, there are new warnings: *The alias "Travel" from the behavior definition "Z##_R_TRAVEL" should be used instead of "Z##_R_TRAVEL".*

   a) Return to the ABAP class, press **Ctrl + F2**.

   b) The new warnings are listed on the *Problems* view. You also find warning icons left from the ENTITY additions in the EML statements.

4. Adjust the EML statements as suggested by the syntax warnings and activate the ABAP class.

   a) After the ENTITY additions in the READ ENTITIES and MODIFY ENTITIES statements, replace **Z##_R_Travel** with **Travel**.

   b) Press **Ctrl + F3** to activate the development object.

5. Display the *Tooltip Description* for data object **failed**.

   How did the alias definition change the type of the data object **failed**?

   The component name changed from **z##_r_travel** to **travel**.

   a) Place the cursor on the data object `failed` (not on the keyword `FAILED`) and press **F2** to display the *Tooltip Description*.

6. Edit the behavior pool and rename the local class to **lhc_travel**.

   a) In the managed statement, place the cursor on the name of the behavior pool **zbp_##_r_travel** and press **F3**.

b) From the tabstrip below the editor, choose *Local Types* to navigate to the source code of the local class.

c) Right-click the name of the local class and choose *Quick Fix*. Alternatively, place the cursor on the name of the local class and press **`Ctrl + 1`**.

d) In the list of available quick fixes, double-click *Rename lhc_z##_r_travel*.

e) While the name of the local class is still highlighted, change the name to **`lhc_travel`**.

7. The definition of the *get_instance_authorizations* method in the behavior pool still uses the name *Z##_R_TRAVEL*. Replace this name with the alias **`Travel`**.

   a) Adjust the code as follows:

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

8. Activate the behavior pool.

   a) Press **`Ctrl + F3`** to activate the development object.

Unit 2
Exercise 5

Enable Optimistic Concurrency Control

In this exercise, you enable optimistic concurrency control in your business object. To do this, you specify one view element as the *ETag* field and make sure that the run time changes the value of this element during every write access to the business object.

Note:
In this exercise, replace ## with your group number.

Table 7: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	*/LRN/437B_R_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437B_C_TRAVEL*

Task 1: Enable Automatic Update of Administrative Fields
Make sure that the runtime automatically updates the administrative fields *ChangedAt* and *ChangedBy* when writing changes to the database.

1. In the definition of your data model view **`Z##_R_Travel`**, add the annotation *@Semantics.systemDateTime.lastChangedAt: true* to the **`ChangedAt`** view element.
2. Add annotation *@Semantics.user.lastChangedBy: true* to the **`ChangedBy`** view element.
3. Activate the data definition.
4. Execute your ABAP class *ZCL_##_EML* as a console application again to set the description of one of your flights.
5. Use the *Data Preview* tool to confirm that the **`ChangedAt`** and **`ChangedBy`** view elements return updated values.

Task 2: Define ETag Field
Enable optimistic concurrency control for your business object by making a suitable view element the *ETag* field of your entity. Then expose the *ETag* definition to the OData service.

1. In your behavior definition for the data model view **`Z##_R_Travel`**, add the **`etag master`** addition, followed by the field name **`ChangedAt`**.
2. Activate the behavior definition.
3. In your behavior projection, that is, in the behavior definition for the projection view **`Z##_C_Travel`**, add the **`use etag`** addition.
4. Activate the behavior projection.

Enable Optimistic Concurrency Control

In this exercise, you enable optimistic concurrency control in your business object. To do this, you specify one view element as the *ETag* field and make sure that the run time changes the value of this element during every write access to the business object.

Note:
In this exercise, replace ## with your group number.

Table 7: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	*/LRN/437B_R_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437B_C_TRAVEL*

Task 1: Enable Automatic Update of Administrative Fields

Make sure that the runtime automatically updates the administrative fields *ChangedAt* and *ChangedBy* when writing changes to the database.

1. In the definition of your data model view **Z##_R_Travel**, add the annotation *@Semantics.systemDateTime.lastChangedAt: true* to the **ChangedAt** view element.

   a) Adjust the code as follows:

```
@Semantics.systemDateTime.lastChangedAt: true
changed_at  as ChangedAt,
changed_by  as ChangedBy
```

2. Add annotation *@Semantics.user.lastChangedBy: true* to the **ChangedBy** view element.

   a) Adjust the code as follows:

```
@Semantics.systemDateTime.lastChangedAt: true
changed_at  as ChangedAt,
@Semantics.user.lastChangedBy: true
changed_by  as ChangedBy
```

3. Activate the data definition.

   a) Choose *Activate* or press **Ctrl + F3**.

4. Execute your ABAP class *ZCL_##_EML* as a console application again to set the description of one of your flights.

   a) Open the source code of the class, and press **F9** to execute it.

5. Use the *Data Preview* tool to confirm that the **ChangedAt** and **ChangedBy** view elements return updated values.

a) Right-click anywhere in the definition of your data model view *Z##_R_Travel* and choose *Open With → Data Preview*.

Result

In the data set that you changed with EML, the values displayed in columns **`ChangedAt`** and **`ChangedBy`** must be different than in the other data sets.

Task 2: Define ETag Field

Enable optimistic concurrency control for your business object by making a suitable view element the *ETag* field of your entity. Then expose the *ETag* definition to the OData service.

1. In your behavior definition for the data model view **`Z##_R_Travel`**, add the **`etag master`** addition, followed by the field name **`ChangedAt`**.

   a) Adjust the code as follows:

```
define behavior for Z##_R_Travel alias Travel
persistent table z##_travel
lock master
authorization master ( instance )
etag master ChangedAt
```

2. Activate the behavior definition.

   a) Press **`Ctrl + F3`** to activate the development object.

3. In your behavior projection, that is, in the behavior definition for the projection view **`Z##_C_Travel`**, add the **`use etag`** addition.

   a) Adjust the code as follows:

```
define behavior for Z##_C_Travel //alias <alias_name>
use etag
{
  use create;
  use update;
  use delete;
}
```

4. Activate the behavior projection.

   a) Press **`Ctrl + F3`** to activate the development object.

Unit 2
Exercise 6

Define and Implement an Action

In this exercise, you extend the UI metadata of your OData service with the definition of a button. By choosing this button, the user can cancel selected flight travels. To achieve this, you first define an action in the behavior definition of your business object. Then you make the action available in the behavior projection and extend the UI metadata with a button to trigger the action. Finally, you implement the action.

Note:
In this exercise, replace ## with your group number.

Table 8: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
ABAP Class	*/LRN/BP_437B_R_TRAVEL*
Behavior Definition (Projection)	*/LRN/437B_C_TRAVEL*
Metadata Extension	*/LRN/437B_C_TRAVEL*
ABAP Class (for messages)	*/LRN/CM_437B_TRAVEL*

Task 1: Define an Instance Action

For the root node of your business object, define a new action (suggested name: **`cancel_travel`**) and use a quick fix to generate the method for the action implementation.

1. Open the behavior definition for your data model view **`Z##_R_TRAVEL`** and add the statement **`action`**, followed by the name of the new action (suggested name: **`cancel_travel`**) and **`;`** as the statement delimiter.

   Does the editor display a new syntax warning for the new code row?

   ___

   ___

   ___

2. Use the quick fix that is related to the syntax warning to add the implementation method to the handler class.

   Note:
   To use this quick fix, the behavior definition must be saved and active.

3. Activate the behavior implementation class.

Task 2: Link the Action to a Button

Add the action to the behavior projection to make it available in the OData UI service and comment out the exposure of the basic operations **`Create`**, **`Update`**, and **`Delete`**. In the metadata extension for your projection view **`Z##_C_TRAVEL`**, add the necessary annotations to define a button on the list report page that is linked to the action. Set a breakpoint in the action implementation method and test the action by choosing the button in the preview for the OData UI service.

1. Edit the behavior definition for the projection view (behavior projection) and comment out the statements that make the standard operations visible for the OData service.
2. Add the statement that makes the action visible for the OData service.
3. Activate the behavior definition.
4. Open the metadata extension for your projection view and scroll down to the annotations for the **`Status`** view element.
5. In the annotation array for annotation **`@UI.lineItem`**, add an array element for the sub annotations **`type`**, **`dataAction`**, and **`label`**.

> Note:
> An annotation array is a comma-separated list of array elements in square brackets. Now, the annotation array of annotation**`@UI.lineItem`** consists of just one array element **`{ position: 10, importance: #HIGH }`**. To add another array element, add a comma and a pair of curly brackets.

6. Inside the annotation array element, add three sub annotations with the following values:

Sub annotation	Value
type	**`#FOR_ACTION`**
dataAction	**`'cancel_travel'`**
label	**`'Cancel the Travel'`**

> Note:
> If your action has a different name, you must adjust the value for annotation `dataAction` accordingly.

7. Activate the metadata extension.
8. Close and reopen the OData service preview and make sure that the new button is visible.
9. Return to the behavior pool and set a breakpoint in the action implementation method.

> Hint:
> There is no need to place any code between METHOD and ENDMETHOD. You can set the breakpoint at either the **`METHOD`** or the **`ENDMETHOD`** statement.

10. Return to the browser window where the app is running, cancel a travel, and confirm that the breakpoint is hit.

Task 3: Implement the Action

In the handler method for the action, use EML to retrieve the data of the selected flight travel. Loop at the data to check the current status of the travel. If the travel is not already canceled (the value of **`STATUS`** does not equal **`C`**), update the status of this travel with value **`C`**. If the travel is already canceled, add the key of the travel to the **`FAILED`** structure and a suitable error message to the **`REPORTED`** structure.

1. In the implementation of method **`cancel_travel`**, implement a READ ENTITIES statement for the root entity of your business object **`Z##_R_Travel`**. Read all fields of all travels that the user selected before triggering the action and store the result in an inline-declared data object (suggested name: **`travels`**).

   Note:
   The import parameter **`keys`** contains the keys of the selected travels. Use a CORRESPONDING expression to convert the content of **`keys`** to the type that is needed as input of the READ ENTITIES statement.

2. Analyze the content of the *Problems* view.

   Is there a new syntax warning?

   ___

   ___

   ___

3. Add **`IN LOCAL MODE`** to the EML statement.

4. Implement a loop over the result, using either an inline declared work area or an inline declared field symbol.

5. Inside the loop, evaluate the value of component **`status`** for the current row. If the value does not equal **`C`**, implement a MODIFY ENTITIES statement (with addition IN LOCAL MODE) to update the root entity of your business object **`Z##_R_Travel`** and set the value of **`status`** for the current travel to **`C`** .

   Hint:
   We recommend that you use the component group **`%tky`** to identify the current travel instance. This way no extra adjustment is needed later, when you draft-enable your business object.

6. If the current travel is already canceled, add its key values to the **`FAILED`** structure and a suitable error message to the **`REPORTED`** structure.

   Note:
   For the error message, create an instance of ABAP class **`/LRN/CM_S4D437`** with a suitable constant for the **`TEXTID`** parameter of the constructor.

7. Activate the behavior implementation class and test your OData service.

Task 4: Optional: Define a Wrapper Class for Messages

Define your own ABAP class for messages (suggested name: **ZCM_##_TRAVEL**). Let it implement the interface **IF_ABAP_BEHV_MESSAGE** and inherit from super class **CX_STATIC_CHECK**. In the class, define a public structured constant for **TEXTID** that refers to message **130** of message class **/LRN/S4D437** (suggested name for the constant: **already_canceled**). Make sure that the constructor offers an optional parameter to set the attribute **if_abap_behv_message~m_severity**.

1. Create a new ABAP class inheriting from class **CX_STATIC_CHECK** and implementing the interface **IF_ABAP_BEHV_MESSAGE**.

2. Check the implementation part of the class. If you find method bodies without implementation for the methods *if_message~get_longtext* and/or *if_message~get_text*, delete them.

3. In the public section of the class, define a structured constant (suggested name: **already_canceled**).

   Note:
   Use the source code template *textIdExceptionClass*.

4. Use a quick fix to rename the constant.

5. Edit the component values of the structured constant. Make the constant refer to message **130** of message class **/LRN/S4D437**.

6. Edit the definition of the constructor of your exception class. Remove or comment parameter **PREVIOUS** and add an optional parameter (suggested name: **severity**) with the same type that is used for attribute **if_abap_behv_message~m_severity**.

7. Edit the implementation of the constructor. Remove the optional parameter **PREVIOUS** from the call of the super constructor.

8. At the end of the method, add a statement to set the value of attribute **if_abap_behv_message~m_severity** to the value of the **severity** parameter, but only if the value of that parameter is not initial. Otherwise, set the attribute to **if_abap_behv_message~severity-error**.

   Hint:
   You can use the logic for attribute **if_t100_message~t100key** as a role model.

9. Activate the wrapper class for messages.

10. In your behavior implementation, replace **/LRN/CM_S4D437** with your own class **ZCM_##_TRAVEL** and activate the behavior pool.

Define and Implement an Action

In this exercise, you extend the UI metadata of your OData service with the definition of a button. By choosing this button, the user can cancel selected flight travels. To achieve this, you first define an action in the behavior definition of your business object. Then you make the action available in the behavior projection and extend the UI metadata with a button to trigger the action. Finally, you implement the action.

Note:
In this exercise, replace ## with your group number.

Table 8: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
ABAP Class	*/LRN/BP_437B_R_TRAVEL*
Behavior Definition (Projection)	*/LRN/437B_C_TRAVEL*
Metadata Extension	*/LRN/437B_C_TRAVEL*
ABAP Class (for messages)	*/LRN/CM_437B_TRAVEL*

Task 1: Define an Instance Action

For the root node of your business object, define a new action (suggested name: **`cancel_travel`**) and use a quick fix to generate the method for the action implementation.

1. Open the behavior definition for your data model view **`Z##_R_TRAVEL`** and add the statement **`action`**, followed by the name of the new action (suggested name: **`cancel_travel`**) and **`;`** as the statement delimiter.

   Does the editor display a new syntax warning for the new code row?

   Yes, there is a syntax warning that *Action Z##_R_TRAVEL ~ CANCEL_TRAVEL is not implemented*.

   a) Adjust the code as follows:

```
{
  create;
  update;
  delete;
  field ( readonly ) AgencyId, TravelId;

  action cancel_travel;
```

2. Use the quick fix that is related to the syntax warning to add the implementation method to the handler class.

> Note:
> To use this quick fix, the behavior definition must be saved and active.

   a) Press **Ctrl + F3** to activate the behavior definition.

   b) Either choose the warning icon next to the `action` statement or right-click the name of the action and select *Quick Fix*.

   c) Double-click *Add method for action cancel_travel of entity z##_r_travel in local class lhc_travel*.

      Result

      The quick fix adds a new method **cancel_travel** to the local class **lhc_travel** in your global class **ZBP_##_R_TRAVEL**.

3. Activate the behavior implementation class.

   a) Press **Ctrl + F3** to activate the development object.

Task 2: Link the Action to a Button

Add the action to the behavior projection to make it available in the OData UI service and comment out the exposure of the basic operations **Create**, **Update**, and **Delete**. In the metadata extension for your projection view **Z##_C_TRAVEL**, add the necessary annotations to define a button on the list report page that is linked to the action. Set a breakpoint in the action implementation method and test the action by choosing the button in the preview for the OData UI service.

1. Edit the behavior definition for the projection view (behavior projection) and comment out the statements that make the standard operations visible for the OData service.

   a) Adjust the code as follows:

```
{
  //  use create;
  //  use update;
  //  use delete;
}
```

2. Add the statement that makes the action visible for the OData service.

   a) Adjust the code as follows:

```
{
  //  use create;
  //  use update;
  //  use delete;

  use action cancel_travel;
}
```

3. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

4. Open the metadata extension for your projection view and scroll down to the annotations for the **Status** view element.

   a) Locate the metadata extension in the *Project Explorer* and double-click it.

b) Scroll down to view the **Status** element.

5. In the annotation array for annotation **@UI.lineItem**, add an array element for the sub annotations **type**, **dataAction**, and **label**.

> Note:
> An annotation array is a comma-separated list of array elements in square brackets. Now, the annotation array of annotation**@UI.lineItem** consists of just one array element **{ position: 10, importance: #HIGH }**. To add another array element, add a comma and a pair of curly brackets.

a) Adjust the code as follows:

```
@UI: {
    lineItem:       [ { position: 10, importance: #HIGH },
                      {                                 }
                    ],
    identification: [ { position: 70, importance: #HIGH } ]
     }
Status;
```

6. Inside the annotation array element, add three sub annotations with the following values:

Sub annotation	Value
type	**#FOR_ACTION**
dataAction	**'cancel_travel'**
label	**'Cancel the Travel'**

> Note:
> If your action has a different name, you must adjust the value for annotation `dataAction` accordingly.

a) Adjust the code as follows:

```
@UI: {
    lineItem:       [ { position: 10, importance: #HIGH },
                      { type:        #FOR_ACTION,
                        dataAction: 'cancel_travel',
                        label:      'Cancel the Travel'
                      }
                    ],
    identification: [ { position: 70, importance: #HIGH } ]
     }
Status;
```

7. Activate the metadata extension.

a) Press **Ctrl + F3** to activate the development object.

8. Close and reopen the OData service preview and make sure that the new button is visible.

a) If the browser window with the preview of the OData service is still open, close it.

b) Open your service binding, select the name of the projection view and choose *Preview...*.

9. Return to the behavior pool and set a breakpoint in the action implementation method.

   Hint:
   There is no need to place any code between METHOD and ENDMETHOD. You can set the breakpoint at either the **METHOD** or the **ENDMETHOD** statement.

   a) Return to the editor with the source code of ABAP class **ZBP_##_R_TRAVEL**.

   b) Find code row `METHOD cancel_travel` and double-click the column left from the row number.

10. Return to the browser window where the app is running, cancel a travel, and confirm that the breakpoint is hit.

    a) Return to the OData service preview and execute the query by choosing *Go*.

    b) Select a travel and choose *Cancel the Travel* from the header toolbar of the table.

       Result

       This opens the debugger.

    c) In the debugger, press **F8** to resume program execution.

Task 3: Implement the Action

In the handler method for the action, use EML to retrieve the data of the selected flight travel. Loop at the data to check the current status of the travel. If the travel is not already canceled (the value of **STATUS** does not equal **C**), update the status of this travel with value **C**. If the travel is already canceled, add the key of the travel to the **FAILED** structure and a suitable error message to the **REPORTED** structure.

1. In the implementation of method **cancel_travel**, implement a READ ENTITIES statement for the root entity of your business object **Z##_R_Travel**. Read all fields of all travels that the user selected before triggering the action and store the result in an inline-declared data object (suggested name: **travels**).

   Note:
   The import parameter **keys** contains the keys of the selected travels. Use a CORRESPONDING expression to convert the content of **keys** to the type that is needed as input of the READ ENTITIES statement.

   a) Adjust the code as follows:

```
METHOD cancel_travel.

    READ ENTITIES OF Z##_R_Travel
      ENTITY Travel
        ALL FIELDS
        WITH CORRESPONDING #( keys )
        RESULT DATA(travels).

ENDMETHOD.
```

2. Analyze the content of the *Problems* view.

Is there a new syntax warning?

Yes, there is a new warning about the *statement variant "IN LOCAL MODE"*.

3. Add **`IN LOCAL MODE`** to the EML statement.

   a) Adjust the code as follows:

```
METHOD cancel_travel.

    READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        ALL FIELDS
        WITH CORRESPONDING #( keys )
        RESULT DATA(travels).

ENDMETHOD.
```

4. Implement a loop over the result, using either an inline declared work area or an inline declared field symbol.

   a) Adjust the code as follows:

```
METHOD cancel_travel.

    READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        ALL FIELDS
        WITH CORRESPONDING #( keys )
        RESULT DATA(travels).

    LOOP AT travels INTO DATA(travel).

    ENDLOOP.

ENDMETHOD.
```

   b) Alternative with field symbol:

```
METHOD cancel_travel.

    READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        ALL FIELDS
        WITH CORRESPONDING #( keys )
        RESULT DATA(travels).

    LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

    ENDLOOP.

ENDMETHOD.
```

5. Inside the loop, evaluate the value of component **`status`** for the current row. If the value does not equal **`C`**, implement a MODIFY ENTITIES statement (with addition IN LOCAL MODE) to update the root entity of your business object **`Z##_R_Travel`** and set the value of **`status`** for the current travel to **`C`** .

Hint:
We recommend that you use the component group **%tky** to identify the current travel instance. This way no extra adjustment is needed later, when you draft-enable your business object.

a) Adjust the code as follows:

```
LOOP AT travels INTO DATA(travel).

  IF travel-status <> 'C'.
    MODIFY ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        UPDATE
        FIELDS ( status )
        WITH VALUE #( ( %tky   = travel-%tky
                        status = 'C' ) ).
  ENDIF.

ENDLOOP.
```

6. If the current travel is already canceled, add its key values to the **FAILED** structure and a suitable error message to the **REPORTED** structure.

Note:
For the error message, create an instance of ABAP class **/LRN/CM_S4D437** with a suitable constant for the **TEXTID** parameter of the constructor.

a) Adjust the code as follows:

```
LOOP AT travels INTO DATA(travel).

  IF travel-status <> 'C'.
    MODIFY ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        UPDATE
        FIELDS ( status )
        WITH VALUE #( ( %tky   = travel-%tky
                        status = 'C' ) ).
  ELSE.
    APPEND VALUE #( %tky = travel-%tky )
      TO failed-travel.

    APPEND VALUE #(
      %tky = travel-%tky
      %msg = NEW /LRN/CM_S4D437(
                textid =
                  /LRN/CM_S4D437=>already_canceled ) )
        TO reported-travel.
  ENDIF.

ENDLOOP.
```

7. Activate the behavior implementation class and test your OData service.

a) Press **Ctrl + F3** to activate the development object.

b) Return to the OData service preview, select a travel that is not yet canceled and choose *Cancel the Travel*.

Result

The content of the *Travel Status* column changes to *C*.

c) Select the same travel again and choose *Cancel the Travel*.

Result

You now see the error message.

Task 4: Optional: Define a Wrapper Class for Messages

Define your own ABAP class for messages (suggested name: **`ZCM_##_TRAVEL`**). Let it implement the interface **`IF_ABAP_BEHV_MESSAGE`** and inherit from super class **`CX_STATIC_CHECK`**. In the class, define a public structured constant for **`TEXTID`** that refers to message **`130`** of message class **`/LRN/S4D437`** (suggested name for the constant: **`already_canceled`**). Make sure that the constructor offers an optional parameter to set the attribute **`if_abap_behv_message~m_severity`**.

1. Create a new ABAP class inheriting from class **`CX_STATIC_CHECK`** and implementing the interface **`IF_ABAP_BEHV_MESSAGE`**.

   a) In the *Project Explorer* window, right-click your own package and select *New → ABAP Class*.

   b) Enter the name of the new class, and a description and super class **`CX_STATIC_CHECK`**.

   c) To add the interface, choose *Add...*.

   d) On the *Add ABAP Interface* window, enter the name of the interface and choose *OK* to close the dialog window.

   e) Choose *Next*, and *Finish* to confirm the preselected transport request.

2. Check the implementation part of the class. If you find method bodies without implementation for the methods *if_message~get_longtext* and/or *if_message~get_text*, delete them.

3. In the public section of the class, define a structured constant (suggested name: **`already_canceled`**).

   Note:
   Use the source code template *textIdExceptionClass*.

   a) Navigate to the definition part of the class and place the cursor after the statement PUBLIC SECTION.

   b) Enter **`text`** and press **`Ctrl + Space`**.

   c) On the suggestion list, place the cursor on *textIdExceptionClass* and press **`Enter`**.

4. Use a quick fix to rename the constant.

   a) After `begin of`, place the cursor on `ZCM_##_TRAVEL` and press **`Ctrl + 1`** to invoke the quick fixes.

   b) In the list of available quick fixes, double-click *Rename zcm_##_travel*.

   c) While the old name is still highlighted, enter **`already_canceled`** as the new name.

5. Edit the component values of the structured constant. Make the constant refer to message **130** of message class **/LRN/S4D437**.

   a) Adjust the code as follows:

```
CONSTANTS:
  BEGIN OF already_canceled,
    msgid TYPE symsgid VALUE '/LRN/S4D437',
    msgno TYPE symsgno VALUE '130',
    attr1 TYPE scx_attrname VALUE '',
    attr2 TYPE scx_attrname VALUE '',
    attr3 TYPE scx_attrname VALUE '',
    attr4 TYPE scx_attrname VALUE '',
  END OF already_canceled.
```

6. Edit the definition of the constructor of your exception class. Remove or comment parameter **PREVIOUS** and add an optional parameter (suggested name: **severity**) with the same type that is used for attribute **if_abap_behv_message~m_severity**.

   a) Adjust the code as follows:

```
METHODS constructor
  IMPORTING
    !textid  LIKE if_t100_message=>t100key OPTIONAL
*     !previous LIKE previous OPTIONAL .
    severity LIKE if_abap_behv_message~m_severity OPTIONAL.
```

7. Edit the implementation of the constructor. Remove the optional parameter **PREVIOUS** from the call of the super constructor.

   a) Adjust the code as follows:

```
CALL METHOD super->constructor
*   EXPORTING
*   previous = previous
  .
```

8. At the end of the method, add a statement to set the value of attribute **if_abap_behv_message~m_severity** to the value of the **severity** parameter, but only if the value of that parameter is not initial. Otherwise, set the attribute to **if_abap_behv_message~severity-error**.

   Hint:
   You can use the logic for attribute **if_t100_message~t100key** as a role model.

   a) Adjust the code as follows:

```
IF textid IS INITIAL.
  if_t100_message~t100key = if_t100_message=>default_textid.
ELSE.
  if_t100_message~t100key = textid.
ENDIF.

IF severity IS INITIAL.
  if_abap_behv_message~m_severity = if_abap_behv_message~severity-
error.
ELSE.
  if_abap_behv_message~m_severity = severity.
ENDIF.
```

9. Activate the wrapper class for messages.

   a) Press **Ctrl + F3** to activate the development object.

10. In your behavior implementation, replace **/LRN/CM_S4D437** with your own class **ZCM_##_TRAVEL** and activate the behavior pool.

   a) Return to your action implementation (method **CANCEL_TRAVEL** in local class **LHC_TRAVEL** of global class **ZBP_##_R_TRAVEL**).

   b) Adjust the code as follows:

```
APPEND VALUE #(
  %tky = travel-%tky
*   %msg = NEW /LRN/CM_S4D437(
*             textid =
*               /LRN/CM_S4D437=>already_canceled )
  %msg = NEW ZCM_##_TRAVEL(
            textid =
              ZCM_##_TRAVEL=>already_canceled )   )
    TO reported-travel.
```

   c) Press **Ctrl + F3** to activate the development object.

Implement Authority Checks

In this exercise, you implement authority checks for your application: read access in CDS access controls, write access by implementing a method of the behavior handler class.

Note:
In this exercise, replace ## with your group number.

Table 9: Solution

Repository Object Type	Repository Object ID
CDS Access Control (Model)	*/LRN/437B_R_TRAVEL*
CDS Access Control (Projection)	*/LRN/437B_C_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
ABAP class	*/LRN/BP_437B_R_TRAVEL*

Task 1: Analyze Authorization Object /LRN/AGCY

Open the definition of Authorization Object **/LRN/AGCY** and analyze it.

1. Open the definition of authorization object **/LRN/AGCY**.

2. Analyze the definition of the authorization object.

   Which are the authorization fields of this authorization object?

   ___

   ___

   ___

   Which are the permitted activities?

   ___

   ___

   ___

Task 2: Define CDS Access Controls

Create an access control for each of your CDS view entities. In the access control for your data model view entity **Z##_R_Travel**, grant access to the travels that were booked at the travel agencies for which the current user has display rights. In the access control for your projection view entity **Z##_C_Travel**, inherit the conditions from the data model view entity.

1. Create an access control for your data model view entity (suggested name: **Z##_R_TRAVEL**). When prompted for a template choose *Define Role with PFCG Aspect*.

2. In the brackets after the WHERE keyword, specify the view element that contains the travel agency number.

3. Supply the function `pfcg_auth( )` with the name of the authority object and its authorization fields. Use authorization field *ACTVT* as a filter field and assign the value for display authorization as filter value.

4. Activate the new access control.

5. Create an access control for your projection view entity (suggested name: **`Z##_C_TRAVEL`**). When prompted for a template, choose *Define Role with Inherited Conditions*. Let the access control inherit the conditions from your data model view entity.

6. After the keyword ENTITY, specify your data model view.

7. Activate the new access control.

8. Test the preview of your OData UI service to verify that now not all entries of your database table are displayed.

Task 3: Instance-Based Authorization Control

In the behavior implementation for your root entity, implement the **`get_instance_authorizations`** method. Make sure that the user needs the change authorization for the travel agency to update travels and to execute action **`cancel_travel`**.

Note:
Instead of using the statement AUTHORITY-CHECK directly, use method **`authority_check`** of the helper class **`/LRN/CL_S4D437_MODEL`**. This method simulates different authorizations for different users.

1. Navigate to the handler class for your root entity and implement the **`get_instance_authorizations`** method. Fill the **`result`** parameter with one row for each affected flight travel.

   Hint:
   Use a CORRESPONDING expression to transfer the content of the **`keys`** parameter to the **`result`** parameter.

2. Implement a loop over the result parameter assigning an inline-declared field symbol (suggested name: **`<result>`**).

3. Inside the loop, check whether the current user has change authorization for the respective travel agency.

   Note:
   Instead of implementing an AUTHORITY-CHECK statement yourself, call the **`authority_check`** method of helper class **`/LRN/CL_S4D437_MODEL`**. This method simulates different authorization profiles for different users.

4. Using the field symbol, update the current row of the **`result`**. Set the flags for the **`update`** operation and the **`cancel_travel`** action depending on the outcome of the **`authority_check`** method.

> Note:
> You find suitable constants in interface **`IF_ABAP_BEHV`**.

5. Activate your coding and test the authority check in the preview of the OData UI service.

Unit 2
Solution 7

Implement Authority Checks

In this exercise, you implement authority checks for your application: read access in CDS access controls, write access by implementing a method of the behavior handler class.

Note:
In this exercise, replace ## with your group number.

Table 9: Solution

Repository Object Type	Repository Object ID
CDS Access Control (Model)	*/LRN/437B_R_TRAVEL*
CDS Access Control (Projection)	*/LRN/437B_C_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437B_R_TRAVEL*
ABAP class	*/LRN/BP_437B_R_TRAVEL*

Task 1: Analyze Authorization Object /LRN/AGCY
Open the definition of Authorization Object **`/LRN/AGCY`** and analyze it.

1. Open the definition of authorization object **`/LRN/AGCY`**.
   a) In *ABAP Development Tools*, choose *Open ABAP Development Object* or press **`Ctrl + Shift + A`**.
   b) Enter **`/LRN/AGCY`**.
   c) From the list of matching items, select */LRN/AGCY (Authorization Object)* and choose *OK*.
2. Analyze the definition of the authorization object.

   Which are the authorization fields of this authorization object?

   `ACTVT` and **`/LRN/AGCY`**

   Which are the permitted activities?

   01 (Create or generate), *02 (Change)*, *03 (Display)*, and *06 (Delete)*

Task 2: Define CDS Access Controls
Create an access control for each of your CDS view entities. In the access control for your data model view entity **`Z##_R_Travel`**, grant access to the travels that were booked at the

travel agencies for which the current user has display rights. In the access control for your projection view entity **Z##_C_Travel**, inherit the conditions from the data model view entity.

1. Create an access control for your data model view entity (suggested name: **Z##_R_TRAVEL**). When prompted for a template choose *Define Role with PFCG Aspect*.
   a) In the *Project Explorer*, right-click the data definition of your data model view entity and select *New Access Control*.

   b) Enter a name and a description and choose *Next*.

   c) Confirm the preselected transport request and choose *Next*.

   d) In the list of available templates, select *Define Role with PFCG Aspect* and choose *Finish*.

2. In the brackets after the WHERE keyword, specify the view element that contains the travel agency number.
   a) Adjust the code as follows:

```
define role Z##_R_TRAVEL {
  grant
    select
      on
        Z##_R_TRAVEL
          where
            (AgencyId) = aspect pfcg_auth(...);
}
```

3. Supply the function `pfcg_auth( )` with the name of the authority object and its authorization fields. Use authorization field *ACTVT* as a filter field and assign the value for display authorization as filter value.
   a) Adjust the code as follows:

```
define role Z##_R_TRAVEL {
  grant
    select
      on
        Z##_R_TRAVEL
          where
            (AgencyId) = aspect pfcg_auth(
                                     /LRN/AGCY,
                                     /LRN/AGCY,
                                     ACTVT = '03');
}
```

4. Activate the new access control.
   a) Press **Ctrl + F3** to activate the development object.

5. Create an access control for your projection view entity (suggested name: **Z##_C_TRAVEL**). When prompted for a template, choose *Define Role with Inherited Conditions*. Let the access control inherit the conditions from your data model view entity.
   a) In the *Project Explorer*, right-click the data definition of your projection view entity and select *New Access Control*.

   b) Enter a name and a description and choose *Next*.

   c) Confirm the preselected transport request and choose *Next*.

   d) In the list of available templates, select *Define Role with Inherited Conditions* and choose *Finish*.

6. After the keyword ENTITY, specify your data model view.

   a) Adjust the code as follows:

```
define role Z##_C_TRAVEL {
  grant
    select
      on
        Z##_C_TRAVEL
          where
            inheriting conditions from entity Z##_R_Travel;
}
```

7. Activate the new access control.

   a) Press **Ctrl + F3** to activate the development object.

8. Test the preview of your OData UI service to verify that now not all entries of your database table are displayed.

   a) Return to the preview of your OData UI service and choose *Go*.

      Result

      You should see fewer entries than before.

Task 3: Instance-Based Authorization Control

In the behavior implementation for your root entity, implement the **get_instance_authorizations** method. Make sure that the user needs the change authorization for the travel agency to update travels and to execute action **cancel_travel**.

Note:
Instead of using the statement AUTHORITY-CHECK directly, use method **authority_check** of the helper class **/LRN/CL_S4D437_MODEL**. This method simulates different authorizations for different users.

1. Navigate to the handler class for your root entity and implement the **get_instance_authorizations** method. Fill the **result** parameter with one row for each affected flight travel.

   Hint:
   Use a CORRESPONDING expression to transfer the content of the **keys** parameter to the **result** parameter.

   a) Adjust the code as follows:

```
METHOD get_instance_authorizations.
  result = CORRESPONDING #( keys ).

ENDMETHOD.
```

2. Implement a loop over the result parameter assigning an inline-declared field symbol (suggested name: **<result>**).

   a) Adjust the code as follows:

```
METHOD get_instance_authorizations.
  result = CORRESPONDING #( keys ).
  LOOP AT result ASSIGNING FIELD-SYMBOL(<result>).

  ENDLOOP.
ENDMETHOD.
```

3. Inside the loop, check whether the current user has change authorization for the respective travel agency.

   > Note:
   > Instead of implementing an AUTHORITY-CHECK statement yourself, call the **authority_check** method of helper class **/LRN/CL_S4D437_MODEL**. This method simulates different authorization profiles for different users.

   a) Adjust the code as follows:

```
METHOD get_instance_authorizations.
  result = CORRESPONDING #( keys ).

  LOOP AT result ASSIGNING FIELD-SYMBOL(<result>).
    DATA(rc) = /lrn/cl_s4d437_model=>authority_check(
                             i_agencyid  = <result>-agencyid
                             i_actvt     = '02' ).

  ENDLOOP.
ENDMETHOD.
```

4. Using the field symbol, update the current row of the **result**. Set the flags for the **update** operation and the **cancel_travel** action depending on the outcome of the **authority_check** method.

   > Note:
   > You find suitable constants in interface **IF_ABAP_BEHV**.

   a) Adjust the code as follows:

      > Note:
      > The ELSE branch is optional because the value of **if_abap_behv=>auth-allowed** is identical to the initial value.

```
METHOD get_instance_authorizations.
  result = CORRESPONDING #( keys ).

  LOOP AT result ASSIGNING FIELD-SYMBOL(<result>).
    DATA(rc) =  /lrn/cl_s4d437_model=>authority_check(
                             i_agencyid  = <result>-agencyid
                             i_actvt     = '02' ).
    IF rc <> 0.
      <result>-%action-cancel_travel = if_abap_behv=>auth-unauthorized.
      <result>-%update               = if_abap_behv=>auth-unauthorized.
    ELSE.
      <result>-%action-cancel_travel = if_abap_behv=>auth-allowed.
      <result>-%update               = if_abap_behv=>auth-allowed.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

5. Activate your coding and test the authority check in the preview of the OData UI service.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) Restart the preview for the OData UI service.

   c) Select different travels and see if the *Cancel the Travel* button is displayed as active.

      Result

      You should find that for at least one of your travels the button is displayed inactive due to missing authorizations.

Unit 3
Exercise 8

Maintain Static Field Properties and Value Help

In this exercise, you enable direct editing of data in the preview of the OData UI service. You extend the behavior definition with static field control and use CDS annotations to provide value help.

Note:
In this exercise, replace ## with your group number.

Table 10: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437C_C_TRAVEL*
CDS Data Definition (Projection)	*/LRN/437C_C_TRAVEL*

Task 1: Enable Direct Editing of Data
Enable direct editing on the object page of your transactional SAP Fiori elements app. To achieve this, make sure that the **`update`** operation is part of your business object and your OData service.

1. Open the behavior definition for your business object **`Z##_R_Travel`** and make sure it defines the **`update`** operation. If the statement is missing, add it. If it is commented out, remove the comment signs.
2. Activate the behavior definition.
3. Open the behavior projection, that is, the behavior definition on projection level (**`Z##_C_Travel`**) and make sure it exposes the **`update`** operation to the OData service. If the statement is missing, add it. If it is commented out, remove the comment signs.
4. Activate the behavior projection.
5. Test the preview of your OData UI service to see the effect.

Task 2: Add Static Field Control
In the behavior definition on data model level (**`Z##_R_Travel`**), add static field control. Disable any direct editing for the **`Status`** field and the administrative fields, that is, the **`ChangedAt`** field and the **`ChangedBy`** field. Ensure that direct editing of the key fields **`AgencyId`** and **`TravelId`** is not allowed during update but is mandatory during create. Set all other fields as mandatory.

1. Edit the behavior definition on data model level. Add a FIELD statement for the view elements **`Status`**, **`ChangedAt`**, and **`ChangedBy`**. Inside the brackets, add the relevant keyword to disable editing in general.

2. For the view elements **AgencyId** and **TravelId**, disable editing during the update operation and enforce entries during the create operation. Add a corresponding FIELD statement to achieve this behavior or, if a FIELD statement already exists for these fields, adjust it accordingly.
3. Add a FIELD statement, followed by view elements **Description**, **CustomerId**, **BeginDate**, and **EndDate**. Inside the brackets, add the relevant keyword to define the fields as mandatory.
4. Activate the behavior definition.
5. Test the OData UI service to see the effect of static field control.

   Is there an automatic check on the mandatory fields?

   ___

   ___

   ___

Task 3: Provide a Value Help

Add the necessary annotations to view element **CustomerId** to define a value help based on CDS view **/DMO/I_Customer_StdVH**.

1. Edit the definition of your projection view entity **Z##_C_Travel**. Before view element **CustomerId**, add the necessary sub annotation of the `@Consumption`annotation to define a value help for this field.
2. Within the annotation, define a single value help based on the CDS view entity **/DMO/I_Customer_StdVH** and its key field **CustomerID**.
3. Activate the data definition.
4. Run the preview of your OData UI service. Verify that there is now a value help for the customer field.

   Is there an implicit check for existing customer IDs?

   ___

   ___

   ___

Unit 3
Solution 8

Maintain Static Field Properties and Value Help

In this exercise, you enable direct editing of data in the preview of the OData UI service. You extend the behavior definition with static field control and use CDS annotations to provide value help.

Note:
In this exercise, replace ## with your group number.

Table 10: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437C_C_TRAVEL*
CDS Data Definition (Projection)	*/LRN/437C_C_TRAVEL*

Task 1: Enable Direct Editing of Data
Enable direct editing on the object page of your transactional SAP Fiori elements app. To achieve this, make sure that the **`update`** operation is part of your business object and your OData service.

1. Open the behavior definition for your business object **`Z##_R_Travel`** and make sure it defines the **`update`** operation. If the statement is missing, add it. If it is commented out, remove the comment signs.

   a) Make sure that the behavior definition contains the following statement:

```
update;
```

2. Activate the behavior definition.

   a) Press **`Ctrl + F3`** to activate the development object.

3. Open the behavior projection, that is, the behavior definition on projection level (**`Z##_C_Travel`**) and make sure it exposes the **`update`** operation to the OData service. If the statement is missing, add it. If it is commented out, remove the comment signs.

   a) Make sure that the behavior projection contains the following statement:

```
use update;
```

4. Activate the behavior projection.

   a) Press **`Ctrl + F3`** to activate the development object.

5. Test the preview of your OData UI service to see the effect.

   a) Reopen the preview for the OData UI service.

b) Select a travel for which you have change authorization, that is, for which the *Cancel the Travel* button is active.

c) Choose this travel to navigate to the details page.

Result

You see an *Edit* button in the right upper corner.

Task 2: Add Static Field Control

In the behavior definition on data model level (**Z##_R_Travel**), add static field control. Disable any direct editing for the **Status** field and the administrative fields, that is, the **ChangedAt** field and the **ChangedBy** field. Ensure that direct editing of the key fields **AgencyId** and **TravelId** is not allowed during update but is mandatory during create. Set all other fields as mandatory.

1. Edit the behavior definition on data model level. Add a FIELD statement for the view elements **Status**, **ChangedAt**, and **ChangedBy**. Inside the brackets, add the relevant keyword to disable editing in general.

   a) Add the following code:

```
field ( readonly )
Status,
ChangedAt,
ChangedBy;
```

2. For the view elements **AgencyId** and **TravelId**, disable editing during the update operation and enforce entries during the create operation. Add a corresponding FIELD statement to achieve this behavior or, if a FIELD statement already exists for these fields, adjust it accordingly.

   a) Make sure your code is similar to the following:

```
field ( readonly : update, mandatory : create )
AgencyId,
TravelId;
```

3. Add a FIELD statement, followed by view elements **Description**, **CustomerId**, **BeginDate**, and **EndDate**. Inside the brackets, add the relevant keyword to define the fields as mandatory.

   a) Add the following code:

```
field ( mandatory )
Description,
CustomerId,
BeginDate,
EndDate;
```

4. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

5. Test the OData UI service to see the effect of static field control.

   Is there an automatic check on the mandatory fields?

   No, the app displays a red asterisk next to the labels of the mandatory fields, but there is no implicit input check.

   a) Reopen the preview for the OData UI service.

b) Select a travel for which you have change authorization, that is, for which the *Cancel the Travel* button is active.

c) Choose this travel to navigate to the details page.

d) Choose *Edit*.

Result

You see a red asterisk next to the labels *Description*, *Customer ID*, *Starting Date*, and *End Date*. The other input fields are disabled.

e) Clear the input field *Description* and choose *Save*.

Result

Even though the field is shown as mandatory, you can save the data without a description. There is no implicit input check to prevent initial mandatory fields.

Task 3: Provide a Value Help

Add the necessary annotations to view element **`CustomerId`** to define a value help based on CDS view **`/DMO/I_Customer_StdVH`**.

1. Edit the definition of your projection view entity **`Z##_C_Travel`**. Before view element **`CustomerId`**, add the necessary sub annotation of the `@Consumption`annotation to define a value help for this field.

   a) Adjust the code as follows:

```
define root view entity Z##_C_Travel
  provider contract transactional_query
  as projection on Z##_R_Travel
{
  key AgencyId,
  key TravelId,
      @Search.defaultSearchElement: true
      Description,
      @Search.defaultSearchElement: true
      @Consumption.valueHelpDefinition: [
        {                              } ]
      CustomerId,
```

2. Within the annotation, define a single value help based on the CDS view entity **`/DMO/I_Customer_StdVH`** and its key field **`CustomerID`**.

   a) Adjust the code as follows:

```
define root view entity Z##_C_Travel
  provider contract transactional_query
  as projection on Z##_R_Travel
{
  key AgencyId,
  key TravelId,
      @Search.defaultSearchElement: true
      Description,
      @Search.defaultSearchElement: true
      @Consumption.valueHelpDefinition: [
        { entity: { name:    '/DMO/I_Customer_StdVH',
                    element: 'CustomerID' } } ]
      CustomerId,
```

3. Activate the data definition.

   a) Press **`Ctrl + F3`** to activate the development object.

4. Run the preview of your OData UI service. Verify that there is now a value help for the customer field.

Is there an implicit check for existing customer IDs?

No, there is no check yet. The user can enter any customer ID and save the travel.

a) Reopen the preview for the OData UI service.

b) Select a travel for which you have change authorization, that is, for which the *Cancel the Travel* button is active.

c) Choose this travel to navigate to the details page.

d) Choose *Edit*.

e) In the *Customer ID* field, enter a value that is not provided by the value help (for example 999999) and choose *Save*.

   Result

   Even though there is a value help with a list of existing customers, you can save the data referencing a customer that does not exist. There is no implicit input check to prevent inconsistent data.

Provide Input Checks Using Validations

In this exercise, you define and implement validations to provide input checks for input fields.

Note:
In this exercise, replace ## with your group number.

Table 11: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Validate the Description

In the behavior definition on data model level **`Z##_R_TRAVEL`**, define a validation (suggested name: **`validateDescription`**) that is performed during the **`Create`** operation and the **`Update`** operation. Implement the validation to ensure that the *Description* field is not empty. If it is, add the affected travel to the list of failed instances and return a suitable error message.

1. In your behavior definition on data model level **`Z##_R_Travel`**, define a new validation (suggested name: **`validateDescription`**) that is triggered by the **`create`** operation or changes to the **`Description`** field.

2. Activate the behavior definition.

3. Use a quick fix to add the validation method to the local handler class.

4. In the implementation of the **`validateDescription`** method, use EML to derive the value of field **`Description`** for the affected flight travels and implement a loop over the retrieved data using an inline declared field symbol (suggested name: **`<travel>`**).

   Note:
   Use the content of parameter **`keys`** to identify the affected travels and do not forget to use the local variant of the EML statement.

5. If the **`Description`** field contains the initial value, add the key of the current travel instance to the related component of parameter **`FAILED`**.

   Note:
   Use component group **`%tky`** to identify the current travel instance.

6. In the same IF block, add the key of the current travel instance and an instance of the message wrapper class **/LRN/CM_S4D437** to the respective component of parameter **REPORTED**. Make sure that the message is linked to the **Description** input field.

   Note:
   Use a suitable text id constant from the **/LRN/CM_S4D437** class to report that a field is empty.

   Hint:
   Set the component **Description** of component **%element** to a component of the constant structure **if_abap_behv=>mk**.

7. Activate the behavior implementation and test the validation in the preview of your OData UI service.

Task 2: Validate the Customer ID

In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation (suggested name: **validateCustomer**) that is always performed during the **create** operation but during the **update** operation, only if the user changed the value of the **CustomerId** field. Implement the validation to ensure that the *CustomerId* field is not empty and that the entered customer number belongs to an existing customer. Otherwise, issue suitable error messages and add the affected travel to the list of failed instances.

1. In your behavior definition on data model level **Z##_R_Travel**, define a new validation (suggested name: **validateCustomer**) that is triggered by the **create** operation or changes to the **CustomerId** field.

2. Activate the behavior definition and use a quick fix to add the validation method to the local handler class.

3. Implement method **validateCustomer** in the same way as method **validateDescription**. Read the current value of **CustomerId** for the affected travel instances. Add the keys of all instances with initial value in **CustomerId** to the **failed** parameter and a suitable message to the **reported** parameter.

   Hint:
   To save time, copy the code from method **validateDescription** and replace field name **Description** with field name **CustomerId**.

   Caution:
   Make sure that in the READ ENTITIES statement, you actually read the **CustomerId** field!

4. If the customer ID is not initial, implement an existence check for the customer number in the field **CustomerId**. If no customer exists with this ID, add the key of the current flight travel instance to the **failed** parameter and a suitable error message from class **/LRN/CM_S4D437** to the **reported** parameter.

Note:
We recommend that for the existence check you read from CDS view entity **/DMO/I_Customer**. That is the view entity from which the value help view **/DMO/I_Customer_StdVH** retrieves its data.

Note:
The message behind the **customer_not_exist** text id has a place holder that is replaced with the value of the **customerid** constructor parameter of the **/LRN/CM_S4D437** class. When you use this text id, supply the **customerid** constructor parameter with the incorrect user entry.

5. Activate the behavior implementation and test the validation in the preview for your OData UI service.

Task 3: Optional: Validate the Dates

In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation for the starting date (suggested name: **validateBeginDate**) that is triggered by the **create** operation or by changes to the **BeginDate** field. In the implementation of this validation, check that the starting date of the travel is not initial and does not lie in the past.

Define and implement a similar validation for the end date of the travel (suggested name: **validateEndDate**).

Finally, define and implement a validation that ensures that the starting date lies before the end date (suggested name: **validateDateSequence**). Make sure that this validation is triggered by the **create** operation and changes to either of the date fields.

1. In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation (suggested name: **validateBeginDate**) that is triggered by the **create** operation and by changes to the **BeginDate** field.

2. Define a validation (suggested name: **validateEndDate**) that is triggered by the **create** operation and by changes to the **EndDate** field.

3. Define a validation (suggested name: **validateDateSequence**) that is triggered by the **create** operation and by changes to either of the two date fields.

4. Activate the behavior definition and use a quick fix to add the validation methods to the local handler class.

5. Implement the **validateBeginDate** method. Use the implementation of the **validateCustomer** method as a template. Instead of reading data from the database, simply compare the data to the system date.

Note:
Call the **get_system_date** method of class **cl_abap_context_info** to retrieve the system date.

6. Implement the **`validateEndDate`** method. To do so, copy the code from the **`validateBeginDate`** method and replace **`BeginDate`** with **`EndDate`**. Use the text id **`end_date_past`** to report the error that the value of the *EndDate* field is in the past.

7. Implement the **`validateDateSequence`** method. To do so, use the implementation of the other validations as a template. Read both date fields and add the key of the current travel instance to the **`failed`** parameter if the end date lies before the starting date. Add an instance of class **`/LRN/CM_S4D437`** with a suitable error text to the **`reported`** parameter. Link the error message to both date fields.

   Hint:
   Use a VALUE expression to set more than one component of the **`%element`** component.

8. Activate the behavior implementation and test the validations in the preview of your OData UI service.

Unit 3
Solution 9

Provide Input Checks Using Validations

In this exercise, you define and implement validations to provide input checks for input fields.

Note:
In this exercise, replace ## with your group number.

Table 11: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Validate the Description

In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation (suggested name: **validateDescription**) that is performed during the **Create** operation and the **Update** operation. Implement the validation to ensure that the *Description* field is not empty. If it is, add the affected travel to the list of failed instances and return a suitable error message.

1. In your behavior definition on data model level **Z##_R_Travel**, define a new validation (suggested name: **validateDescription**) that is triggered by the **create** operation or changes to the **Description** field.

   a) Add the following code:

```
validation validateDescription on save
{ create;
  field Description;
}
```

2. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

3. Use a quick fix to add the validation method to the local handler class.

   a) Place the cursor on the name of the validation (validateDescription) and press **Ctrl + 1** to invoke the quick assist proposals.

   b) From the list of available quick fixes, choose *Add method for validation validatedescription of entity z##_r_travel in local class lhc_travel.*

4. In the implementation of the **validateDescription** method, use EML to derive the value of field **Description** for the affected flight travels and implement a loop over the retrieved data using an inline declared field symbol (suggested name: **<travel>**).

Note:
Use the content of parameter **keys** to identify the affected travels and do not forget to use the local variant of the EML statement.

a) Between `METHOD validateDescription.` and `ENDMETHOD.`, add the following code:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Travel
    FIELDS ( Description )
    WITH CORRESPONDING #( keys )
    RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  ENDLOOP.
```

5. If the **Description** field contains the initial value, add the key of the current travel instance to the related component of parameter **FAILED**.

Note:
Use component group **%tky** to identify the current travel instance.

a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  IF <travel>-Description IS INITIAL.
    APPEND VALUE #(  %tky = <travel>-%tky )
      TO failed-travel.

  ENDIF.

ENDLOOP.
```

6. In the same IF block, add the key of the current travel instance and an instance of the message wrapper class **/LRN/CM_S4D437** to the respective component of parameter **REPORTED**. Make sure that the message is linked to the **Description** input field.

Note:
Use a suitable text id constant from the **/LRN/CM_S4D437** class to report that a field is empty.

Hint:
Set the component **Description** of component **%element** to a component of the constant structure **if_abap_behv=>mk**.

a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
```

```
  IF <travel>-Description IS INITIAL.
    APPEND VALUE #(  %tky = <travel>-%tky )
      TO failed-travel.

    APPEND VALUE #( %tky                  = <travel>-%tky
                    %msg                  = NEW /lrn/cm_s4d437(
                                 /lrn/cm_s4d437=>field_empty )
                    %element-Description = if_abap_behv=>mk-on )
      TO reported-travel.
  ENDIF.

ENDLOOP.
```

7. Activate the behavior implementation and test the validation in the preview of your OData UI service.

   a) Press **Ctrl + F3** to activate the development object.

   b) Reopen the preview for the OData UI service.

   c) Select a travel for which you have change authorization, that is, for which the *Cancel the Travel* button is active.

   d) Choose this travel to navigate to the details page.

   e) Choose *Edit*.

   f) Remove the text in the *Description* field and choose *Save*.

   Result

   You see an error message *Please make an entry* with a link to the empty input field.

Task 2: Validate the Customer ID

In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation (suggested name: **validateCustomer**) that is always performed during the **create** operation but during the **update** operation, only if the user changed the value of the **CustomerId** field. Implement the validation to ensure that the *CustomerId* field is not empty and that the entered customer number belongs to an existing customer. Otherwise, issue suitable error messages and add the affected travel to the list of failed instances.

1. In your behavior definition on data model level **Z##_R_Travel**, define a new validation (suggested name: **validateCustomer**) that is triggered by the **create** operation or changes to the **CustomerId** field.

   a) Add the following code:

```
validation validateCustomer on save
{ create;
  field CustomerId;
}
```

2. Activate the behavior definition and use a quick fix to add the validation method to the local handler class.

   a) Press **Ctrl + F3** to activate the development object.

   b) Place the cursor on the name of the validation `validateCustomer` and press **Ctrl + 1** to invoke the quick assist proposals.

   c) From the list of available quick fixes, choose *Add method for validation validatecustomer of entity z##_r_travel in local class lhc_travel* .

3. Implement method **validateCustomer** in the same way as method **validateDescription**. Read the current value of **CustomerId** for the affected travel instances. Add the keys of all instances with initial value in **CustomerId** to the **failed** parameter and a suitable message to the **reported** parameter.

   Hint:
   To save time, copy the code from method **validateDescription** and replace field name **Description** with field name **CustomerId**.

   Caution:
   Make sure that in the READ ENTITIES statement, you actually read the **CustomerId** field!

   a) Adjust the code as follows:

```
METHOD validateCustomer.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( CustomerId )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
    IF <travel>-CustomerId IS INITIAL.

      APPEND VALUE #(  %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky                = <travel>-%tky
                      %msg                = NEW /lrn/cm_s4d437(
                                  /lrn/cm_s4d437=>field_empty )
                      %element-CustomerId = if_abap_behv=>mk-on )
        TO reported-travel.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

4. If the customer ID is not initial, implement an existence check for the customer number in the field **CustomerId**. If no customer exists with this ID, add the key of the current flight travel instance to the **failed** parameter and a suitable error message from class **/LRN/CM_S4D437** to the **reported** parameter.

   Note:
   We recommend that for the existence check you read from CDS view entity **/DMO/I_Customer**. That is the view entity from which the value help view **/DMO/I_Customer_StdVH** retrieves its data.

Note:
The message behind the **customer_not_exist** text id has a place holder that is replaced with the value of the **customerid** constructor parameter of the **/LRN/CM_S4D437** class. When you use this text id, supply the **customerid** constructor parameter with the incorrect user entry.

a) Adjust the code as follows:

```
METHOD validateCustomer.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( CustomerId )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
    IF <travel>-CustomerId IS INITIAL.
      APPEND VALUE #(  %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky                = <travel>-%tky
                      %msg                = NEW /lrn/cm_s4d437(
                            /lrn/cm_s4d437=>field_empty )
                      %element-CustomerId = if_abap_behv=>mk-on )
        TO reported-travel.
    ELSE.
      SELECT SINGLE FROM /dmo/i_customer
        FIELDS CustomerID
        WHERE CustomerID = @<travel>-CustomerId
        INTO @DATA(dummy).

      IF sy-subrc <> 0.
        APPEND VALUE #(  %tky = <travel>-%tky )
          TO failed-travel.

        APPEND VALUE #( %tky                = <travel>-%tky
                        %msg                = NEW /lrn/cm_s4d437(
                    textid     = /lrn/cm_s4d437=>customer_not_exist
                    customerid = <travel>-CustomerId )
                        %element-CustomerId = if_abap_behv=>mk-on )
          TO reported-travel.
      ENDIF.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

5. Activate the behavior implementation and test the validation in the preview for your OData UI service.

   a) Press **Ctrl + F3** to activate the development object.

   b) Reopen the preview for the OData UI service.

   c) Select a travel for which you have change authorization, that is, for which the *Cancel the Travel* button is active.

   d) Choose this travel to navigate to the details page.

   e) Choose *Edit*.

f) Remove the entry in the *Customer ID* field and choose *Save*.

Result

You see an error message *Please make an entry* with a link to the empty input field.

g) In the *Customer ID* field, enter **999999** and choose *Save*.

Result

You should see an error message *Customer 999999 does not exist* with a link to the *Customer ID* field.

Task 3: Optional: Validate the Dates

In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation for the starting date (suggested name: **validateBeginDate**) that is triggered by the **create** operation or by changes to the **BeginDate** field. In the implementation of this validation, check that the starting date of the travel is not initial and does not lie in the past.

Define and implement a similar validation for the end date of the travel (suggested name: **validateEndDate**).

Finally, define and implement a validation that ensures that the starting date lies before the end date (suggested name: **validateDateSequence**). Make sure that this validation is triggered by the **create** operation and changes to either of the date fields.

1. In the behavior definition on data model level **Z##_R_TRAVEL**, define a validation (suggested name: **validateBeginDate**) that is triggered by the **create** operation and by changes to the **BeginDate** field.

   a) Add the following code:

```
validation validateBeginDate on save
{ create;
  field BeginDate;
}
```

2. Define a validation (suggested name: **validateEndDate**) that is triggered by the **create** operation and by changes to the **EndDate** field.

   a) Add the following code:

```
validation validateEndDate on save
{ create;
  field EndDate;
}
```

3. Define a validation (suggested name: **validateDateSequence**) that is triggered by the **create** operation and by changes to either of the two date fields.

   a) Add the following code:

```
validation validateDateSequence on save
{ create;
  field BeginDate,
  EndDate;
}
```

4. Activate the behavior definition and use a quick fix to add the validation methods to the local handler class.

   a) Press **Ctrl + F3** to activate the development object.

   b) Place the cursor on the name of one of the new validations and press **Ctrl + 1** to invoke the quick assist proposals.

c) From the list of available quick fixes, choose *Add all 3 missing methods of entity z##_r_travel in local handler class lhc_travel.*

5. Implement the **validateBeginDate** method. Use the implementation of the **validateCustomer** method as a template. Instead of reading data from the database, simply compare the data to the system date.

> Note:
> Call the **get_system_date** method of class **cl_abap_context_info** to retrieve the system date.

a) Adjust the code as follows:

```
METHOD validateBeginDate.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( BeginDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
    IF <travel>-BeginDate IS INITIAL.
      APPEND VALUE #(  %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky                = <travel>-%tky
                      %msg                = NEW /lrn/cm_s4d437(
                                   /lrn/cm_s4d437=>field_empty )
                      %element-BeginDate = if_abap_behv=>mk-on )
        TO reported-travel.
    ELSEIF <travel>-begindate <
                      cl_abap_context_info=>get_system_date(  ).
      APPEND VALUE #(  %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky                = <travel>-%tky
                      %msg                = NEW /lrn/cm_s4d437(
                               /lrn/cm_s4d437=>begin_date_past )
                      %element-Begindate = if_abap_behv=>mk-on )
        TO reported-travel.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

6. Implement the **validateEndDate** method. To do so, copy the code from the **validateBeginDate** method and replace **BeginDate** with **EndDate**. Use the text id **end_date_past** to report the error that the value of the *EndDate* field is in the past.

a) The code now looks as follows:

```
METHOD validateEndDate.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
```

```
    IF <travel>-EndDate IS INITIAL.
      APPEND VALUE #(  %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky             = <travel>-%tky
                      %msg             = NEW /lrn/cm_s4d437(
                               /lrn/cm_s4d437=>field_empty )
                      %element-EndDate = if_abap_behv=>mk-on )
        TO reported-travel.
    ELSEIF <travel>-EndDate <
                    cl_abap_context_info=>get_system_date(  ).
      APPEND VALUE #(  %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky             = <travel>-%tky
                      %msg             = NEW /lrn/cm_s4d437(
                             /lrn/cm_s4d437=>end_date_past )
                      %element-EndDate = if_abap_behv=>mk-on )
        TO reported-travel.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

7. Implement the **validateDateSequence** method. To do so, use the implementation of the other validations as a template. Read both date fields and add the key of the current travel instance to the **failed** parameter if the end date lies before the starting date. Add an instance of class **/LRN/CM_S4D437** with a suitable error text to the **reported** parameter. Link the error message to both date fields.

   Hint:
   Use a VALUE expression to set more than one component of the **%element** component.

   a) Adjust the code as follows:

```
METHOD validateDateSequence.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( BeginDate EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
    IF <travel>-EndDate < <travel>-BeginDate.
      APPEND VALUE #( %tky = <travel>-%tky )
        TO failed-travel.

      APPEND VALUE #( %tky     = <travel>-%tky
                      %msg     = NEW /lrn/cm_s4d437(
                               /lrn/cm_s4d437=>dates_wrong_sequence )
                      %element = VALUE #(
                               BeginDate = if_abap_behv=>mk-on
                               EndDate   = if_abap_behv=>mk-on ) )
        TO reported-travel.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

8. Activate the behavior implementation and test the validations in the preview of your OData UI service.

a) Perform this step as before.

Unit 3
Exercise 10

Implement Unmanaged Early Numbering

In this exercise, you enable the creation of flight travels. You activate unmanaged early numbering and provide the required key values through a dedicated handler method.

Note:
In this exercise, replace ## with your group number.

Table 12: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437C_C_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Enable the Creation of Data
Enable the creation of flight travels in the OData UI service. To achieve this, make sure that the behavior definition and behavior projection contain the necessary statements.

1. Open the behavior definition for your business object **`Z##_R_TRAVEL`** and make sure it defines the **`create`** operation. If the statement is missing, add it. If it is commented out, remove the comment signs.
2. Activate the behavior definition.
3. Open the behavior projection, that is, the behavior definition on projection level **`Z##_C_TRAVEL`** and make sure it exposes the **`create`** operation to the OData service. If the statement is missing, add it. If it is commented out, remove the comment signs.
4. Activate the behavior projection and test the preview of your OData UI service to see the effect.
5. Try to create a new flight travel with initial values in all fields.

   Can you save the travel?

   ____

   ____

   ____

Task 2: Implement Unmanaged Early Numbering
Enable unmanaged early numbering and implement the related handler method. Use suitable methods of class **`/LRN/CL_S4D437_MODEL`** to derive values for the key fields **`AgencyId`** and **`TravelId`**. Finally, disable any direct editing for the key fields.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL**. At the end of the DEFINE BEHAVIOR statement, add the necessary addition to enable unmanaged early numbering.
2. Activate the behavior definition and use a quick fix to add the required numbering method to the local handler class.
3. In the **earlynumbering_create** method, call the **get_agency_by_user** method of the **/LRN/CL_S4D437_MODEL** class and store the result in an inline declared variable (suggested name: **agencyid**).
4. Fill the **travel** component of the **mapped** parameter with one row for each new flight travel instance for which you must set the key values.

> Hint:
> Use a CORRESPONDING expression to transfer the temporary key in the **%cid** column from the **entities** parameter to the **travel** component of the **mapped** parameter.

5. Implement a loop over the **travel** component of **mapped**, assigning an inline-declared field symbol (suggested name: **<mapping>**).
6. Inside the loop, update the current row with values for the key fields **AgencyId** and **TravelId**.

> Note:
> Call the **get_next_travelid** method of the **/LRN/CL_S4D437_MODEL** class to retrieve a new travel number for each entity.

7. Activate the behavior implementation.
8. Open the behavior definition for your business object **Z##_R_TRAVEL** and adjust the static field properties of the key fields **AgencyId** and **TravelId** to disable any direct changes to those fields.
9. Activate the behavior definition and test the preview of your OData UI service to see the effect.

Unit 3
Solution 10

Implement Unmanaged Early Numbering

In this exercise, you enable the creation of flight travels. You activate unmanaged early numbering and provide the required key values through a dedicated handler method.

Note:
In this exercise, replace ## with your group number.

Table 12: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437C_C_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Enable the Creation of Data

Enable the creation of flight travels in the OData UI service. To achieve this, make sure that the behavior definition and behavior projection contain the necessary statements.

1. Open the behavior definition for your business object **`Z##_R_TRAVEL`** and make sure it defines the **`create`** operation. If the statement is missing, add it. If it is commented out, remove the comment signs.

   a) Make sure that the behavior definition contains the following statement:

```
create;
```

2. Activate the behavior definition.

   a) Press **`Ctrl + F3`** to activate the development object.

3. Open the behavior projection, that is, the behavior definition on projection level **`Z##_C_TRAVEL`** and make sure it exposes the **`create`** operation to the OData service. If the statement is missing, add it. If it is commented out, remove the comment signs.

   a) Make sure that the behavior projection contains the following statement:

```
use create;
```

4. Activate the behavior projection and test the preview of your OData UI service to see the effect.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) Reopen the preview for the OData UI service.

      Result

      You see a *Create* button in the toolbar of the table.

5. Try to create a new flight travel with initial values in all fields.

Can you save the travel?

No, the static field control value **mandatory : create** that was specified in the behavior definition for the key fields *AgencyId* and *TravelId* implies an implicit check for noninitial values.

a) In the preview of the OData UI service, choose *Create*.

b) Leave all fields empty and choose *Create*.

Result

You see the error messages *Enter a value for the Agency ID field* and *Enter a value for the Travel ID field*.

Task 2: Implement Unmanaged Early Numbering

Enable unmanaged early numbering and implement the related handler method. Use suitable methods of class **/LRN/CL_S4D437_MODEL** to derive values for the key fields **AgencyId** and **TravelId**. Finally, disable any direct editing for the key fields.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL**. At the end of the DEFINE BEHAVIOR statement, add the necessary addition to enable unmanaged early numbering.

   a) Adjust the code as follows:

```
define behavior for Z##_R_Travel alias Travel
persistent table z##_travel
lock master
authorization master ( instance )
etag master ChangedAt
early numbering
{
```

2. Activate the behavior definition and use a quick fix to add the required numbering method to the local handler class.

   a) Press **Ctrl + F3** to activate the development object.

   b) Place the cursor on the **create** keyword and press **Ctrl + 1** to invoke the quick assist proposals. Alternatively, choose the warning icon with a light bulb on the left of the code row with this keyword.

   c) From the list of available quick fixes, choose *Add earlynumbering method for create of entity z##_r_travel in local class zbp_##_r_travel->lhc_travel.*

3. In the **earlynumbering_create** method, call the **get_agency_by_user** method of the **/LRN/CL_S4D437_MODEL** class and store the result in an inline declared variable (suggested name: **agencyid**).

   a) Adjust the code as follows:

```
METHOD earlynumbering_create.

  DATA(agencyid) = /lrn/cl_s4d437_model=>get_agency_by_user(  ).

ENDMETHOD.
```

4. Fill the **travel** component of the **mapped** parameter with one row for each new flight travel instance for which you must set the key values.

Hint:
Use a CORRESPONDING expression to transfer the temporary key in the **%cid** column from the **entities** parameter to the **travel** component of the **mapped** parameter.

a) Adjust the code as follows:

```
METHOD earlynumbering_create.

  DATA(agencyid) = /lrn/cl_s4d437_model=>get_agency_by_user(  ).

  mapped-travel = CORRESPONDING #( entities ).

ENDMETHOD.
```

5. Implement a loop over the **travel** component of **mapped**, assigning an inline-declared field symbol (suggested name: **<mapping>**).

   a) Adjust the code as follows:

```
METHOD earlynumbering_create.

  DATA(agencyid) = /lrn/cl_s4d437_model=>get_agency_by_user(  ).

  mapped-travel = CORRESPONDING #( entities ).

  LOOP AT mapped-travel ASSIGNING FIELD-SYMBOL(<mapping>).

  ENDLOOP.

ENDMETHOD.
```

6. Inside the loop, update the current row with values for the key fields **AgencyId** and **TravelId**.

Note:
Call the **get_next_travelid** method of the **/LRN/CL_S4D437_MODEL** class to retrieve a new travel number for each entity.

   a) Adjust the code as follows:

```
METHOD earlynumbering_create.

  DATA(agencyid) = /lrn/cl_s4d437_model=>get_agency_by_user(  ).

  mapped-travel = CORRESPONDING #( entities ).

  LOOP AT mapped-travel ASSIGNING FIELD-SYMBOL(<mapping>).
    <mapping>-AgencyId = agencyid.
    <mapping>-TravelId = /lrn/cl_s4d437_model=>get_next_travelid( ).
  ENDLOOP.

ENDMETHOD.
```

7. Activate the behavior implementation.

   a) Press **Ctrl + F3** to activate the development object.

8. Open the behavior definition for your business object **Z##_R_TRAVEL** and adjust the static field properties of the key fields **AgencyId** and **TravelId** to disable any direct changes to those fields.

   a) Adjust the code as follows:

```
//   field ( readonly : update, mandatory : create )
//   AgencyId,
//   TravelId;

field ( readonly )
AgencyId,
TravelId;
```

9. Activate the behavior definition and test the preview of your OData UI service to see the effect.

   a) Press **Ctrl + F3** to activate the development object.

   b) Reopen the preview for the OData UI service and choose *Create*.

      Result

      The key fields must be displayed as read-only.

   c) Maintain all required fields with valid values, and choose *Create*.

      Result

      Once the travel is created successfully, you see the new values for the key fields.

Define and Implement a Determination

In this exercise, you create a determination that is used to set the content of the *Status* field to the value **N** (= new) when a new flight travel is created.

Note:
In this exercise, replace ## with your group number.

Table 13: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Define a Determination

In the behavior definition on data model level **Z##_R_TRAVEL**, define a determination (suggested name: **determineStatus**) that is to be used to set the initial status for a new flight travel.

1. Open the behavior definition for your business object **Z##_R_TRAVEL**. Define a new determination (suggested name: **determineStatus**) that is only triggered for new instances. The determination should be executed immediately after data changes take place in the transactional buffer.

2. Activate the behavior definition and use a quick fix to add the determination method to the local handler class.

Task 2: Implement the Determination

In the implementation of the **determineStatus** method, use EML to set the content of the *Status* field of newly created flight travels to the value **N** (= new).

1. Implement a READ ENTITIES statement at the beginning of the **determineStatus** method to read the value of the **Status** field for the affected flight travels into an inline declared table named **travels**.

2. After the READ ENTITIES statement, delete all entries from the **travels** table for which the **Status** field is not initial. Make sure that the **determineStatus** method is exited if the cleaned **travels** table does not contain any entries.

3. After checking for existing table entries, implement a MODIFY ENTITIES statement to set the content of the *Status* field to the value **N** (= new) for all flight travels in the **travels** table.

4. Finally, pass the error messages returned by the MODIFY ENTITIES statement to the **reported** parameter of the **determineStatus** method.

Task 3: Test the Determination

Now you will test your work.

1. Activate the behavior implementation class.
2. Test your determination in the preview of the OData UI service.

Unit 3
Solution 11

Define and Implement a Determination

In this exercise, you create a determination that is used to set the content of the *Status* field to the value **N** (= new) when a new flight travel is created.

Note:
In this exercise, replace ## with your group number.

Table 13: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Define a Determination
In the behavior definition on data model level **Z##_R_TRAVEL**, define a determination (suggested name: **determineStatus**) that is to be used to set the initial status for a new flight travel.

1. Open the behavior definition for your business object **Z##_R_TRAVEL**. Define a new determination (suggested name: **determineStatus**) that is only triggered for new instances. The determination should be executed immediately after data changes take place in the transactional buffer.
   a) Add the following code:

```
determination determineStatus on modify
{ create;
}
```

2. Activate the behavior definition and use a quick fix to add the determination method to the local handler class.
   a) Press **Ctrl + F3** to activate the development object.
   b) Place the cursor on the name of the determination *determineStatus* and press **Ctrl + 1** to invoke the quick assist proposals.
   c) From the list of available quick fixes, choose *Add method for determination determineStatus of entity z##_r_travel in local class lhc_travel*.

Task 2: Implement the Determination
In the implementation of the **determineStatus** method, use EML to set the content of the *Status* field of newly created flight travels to the value **N** (= new).

1. Implement a READ ENTITIES statement at the beginning of the **determineStatus** method to read the value of the **Status** field for the affected flight travels into an inline declared table named **travels**.

a) Your code should be similar to the following:

```
METHOD determineStatus.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( Status )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

ENDMETHOD.
```

2. After the READ ENTITIES statement, delete all entries from the **`travels`** table for which the **`Status`** field is not initial. Make sure that the **`determineStatus`** method is exited if the cleaned **`travels`** table does not contain any entries.

   a) Your code should be similar to the following:

```
METHOD determineStatus.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( Status )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  DELETE travels WHERE Status IS NOT INITIAL.
  CHECK travels IS NOT INITIAL.

ENDMETHOD.
```

3. After checking for existing table entries, implement a MODIFY ENTITIES statement to set the content of the *Status* field to the value **`N`** (= new) for all flight travels in the **`travels`** table.

   a) Add the following code:

```
METHOD determineStatus.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( Status )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  DELETE travels WHERE Status IS NOT INITIAL.
  CHECK travels IS NOT INITIAL.

  MODIFY ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      UPDATE FIELDS ( Status )
      WITH VALUE #( FOR key IN travels ( %tky   = key-%tky
                                         Status = 'N' )  ).

ENDMETHOD.
```

4. Finally, pass the error messages returned by the MODIFY ENTITIES statement to the **`reported`** parameter of the **`determineStatus`** method.

   a) Add the following code:

```
METHOD determineStatus.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
```

```
      FIELDS ( Status )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  DELETE travels WHERE Status IS NOT INITIAL.
  CHECK travels IS NOT INITIAL.

  MODIFY ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      UPDATE FIELDS ( Status )
      WITH VALUE #( FOR key IN travels ( %tky   = key-%tky
                                          Status = 'N' )  )
      REPORTED DATA(update_reported).

  reported = CORRESPONDING #( DEEP update_reported ).

ENDMETHOD.
```

Task 3: Test the Determination

Now you will test your work.

1. Activate the behavior implementation class.
   a) Press **`Ctrl + F3`** to activate the development object.

2. Test your determination in the preview of the OData UI service.
   a) Reopen the preview for your OData UI service.

   b) Choose *Create* from the toolbar of the table.

   c) On the object page, enter valid values in the mandatory fields and select the *Create* button.

   Result

   The flight travel is saved and the *Travel Status* field contains the value **`N`**.

Unit 3
Exercise 12

Implement Dynamic Action and Field Control

In this exercise, you disable the **CANCEL_TRAVEL** action for flight travels that have been canceled or which have already ended. You also disallow direct editing of such flight travels.

In addition, you disallow changes to the starting date and the customer ID once the travel has started.

Note:
In this exercise, replace ## with your group number.

Table 14: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Implement Dynamic Operation Control
In the behavior of your business object, enable dynamic feature control for the **CANCEL_TRAVEL** action and standard operation **UPDATE**. Add the **GET_INSTANCE_FEATURES** method to the local handler class and implement it to disable changes to flight travels which have already been canceled or which have ended.

1. In your behavior definition on data model level **Z##_R_Travel**, locate the definition of standard operation **UPDATE** and add the relevant option to enable dynamic feature control based on entity instances.
2. Locate the definition of the **CANCEL_TRAVEL** action and add the relevant option to enable dynamic feature control based on entity instances.
3. Activate the behavior definition.
4. Use a quick fix to add the method for instance-based feature control to the local handler class.
5. In the implementation of method *get_instance_features*, use EML (in local mode) to read the values of fields **Status**, **BeginDate**, and **EndDate** for the affected flight travel instances.
6. Implement a loop over the retrieved data. At the beginning of the loop, append a new line to the **result** parameter with the key of that flight travel instance and assign a new inline declared field symbol to that new line in **result**.
7. After the APPEND statement, implement an IF-ENDIF control structure to check if either the **Status** of the current travel instance is **C**, or the **EndDate** is not initial and lies in the past.

Note:
Call the **get_system_date** method of the **CL_ABAP_CONTEXT_INFO** class to retrieve the current system date.

8. Inside the IF-block, disable the **UPDATE** operation and the action by filling the related components in the new line of the **result** parameter accordingly.

Hint:
You find constants for feature control in attribute **FC-O** of interface **IF_ABAP_BEHV**.

9. Activate the behavior implementation and test the preview of your OData UI service to see the effect.

Task 2: Implement Dynamic Field Control

Enable dynamic field control for the **CustomerId** field and the **BeginDate** field. Set the fields to read-only if the travel has already started, that is, if the starting date lies in the past.

1. In your behavior definition on data model level **Z##_R_Travel**, locate the FIELD statement for the fields **Description**, **CustomerId**, **BeginDate**, and **EndDate**. Comment or remove **CustomerId** and **BeginDate** and define a new FIELD statement for these two fields in which you enable dynamic field control.
2. Activate the behavior definition.
3. Navigate to the method for feature control. After the existing IF-ENDIF control structure, but still inside the loop, implement a second IF-ENDIF structure that checks whether the **BeginDate** component of the current flight travel is not initial and lies in the past.
4. Inside the IF-block, disable editing of **CustomerId** and **BeginDate** fields for the current flight travel instance.

Hint:
In the current line of the **result** parameter, update the value of the respective components. Assign these components a suitable component of the attribute **FC-F** from interface **IF_ABAP_BEHV** as value.

5. Implement an ELSE branch in which you set the fields to **mandatory**.

Note:
This ELSE branch is not optional because the value of **FC-F-MANDATORY** does not equal the initial value!

6. Activate the behavior implementation and test the preview of your OData UI service to see the effect.

Implement Dynamic Action and Field Control

In this exercise, you disable the **CANCEL_TRAVEL** action for flight travels that have been canceled or which have already ended. You also disallow direct editing of such flight travels.

In addition, you disallow changes to the starting date and the customer ID once the travel has started.

Note:
In this exercise, replace ## with your group number.

Table 14: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437C_R_TRAVEL*
ABAP Class	*/LRN/BP_437C_R_TRAVEL*

Task 1: Implement Dynamic Operation Control

In the behavior of your business object, enable dynamic feature control for the **CANCEL_TRAVEL** action and standard operation **UPDATE**. Add the **GET_INSTANCE_FEATURES** method to the local handler class and implement it to disable changes to flight travels which have already been canceled or which have ended.

1. In your behavior definition on data model level **Z##_R_Travel**, locate the definition of standard operation **UPDATE** and add the relevant option to enable dynamic feature control based on entity instances.

   a) Adjust the code as follows:

```
update ( features : instance );
```

2. Locate the definition of the **CANCEL_TRAVEL** action and add the relevant option to enable dynamic feature control based on entity instances.

   a) Adjust the code as follows:

```
action ( features : instance ) cancel_travel;
```

3. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

4. Use a quick fix to add the method for instance-based feature control to the local handler class.

   a) In the code, place the cursor on the name of your root entity *Z##_R_Travel* and press **Ctrl + 1**. Alternatively, you can choose the warning icon with a light bulb next to the code row that begins with `define behavior for`.

   b) From the list of available quick fixes, choose *Add method for operation instance_features on entity z##_r_travel in local class lhc_travel*.

5. In the implementation of method *get_instance_features*, use EML (in local mode) to read the values of fields **Status**, **BeginDate**, and **EndDate** for the affected flight travel instances.

   a) Adjust the code as follows:

```
METHOD get_instance_features.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( Status BeginDate EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

ENDMETHOD.
```

6. Implement a loop over the retrieved data. At the beginning of the loop, append a new line to the **result** parameter with the key of that flight travel instance and assign a new inline declared field symbol to that new line in **result**.

   a) Adjust the code as follows:

```
METHOD get_instance_features.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS ( Status BeginDate EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

    APPEND CORRESPONDING #( <travel> ) TO result
      ASSIGNING FIELD-SYMBOL(<result>).

  ENDLOOP.

ENDMETHOD.
```

7. After the APPEND statement, implement an IF-ENDIF control structure to check if either the **Status** of the current travel instance is **C**, or the **EndDate** is not initial and lies in the past.

Note:
Call the **get_system_date** method of the **CL_ABAP_CONTEXT_INFO** class to retrieve the current system date.

   a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  APPEND CORRESPONDING #( <travel> ) TO result
    ASSIGNING FIELD-SYMBOL(<result>).

  IF <travel>-Status = 'C' OR
    ( <travel>-EndDate IS NOT INITIAL AND
      <travel>-EndDate < cl_abap_context_info=>get_system_date( ) ).

  ENDIF.

ENDLOOP.
```

8. Inside the IF-block, disable the **UPDATE** operation and the action by filling the related components in the new line of the **result** parameter accordingly.

   Hint:
   You find constants for feature control in attribute **FC-O** of interface **IF_ABAP_BEHV**.

   a) Adjust the code as follows:

      Note:
      The ELSE branch is optional because the value of constant **FC-O-ENABLED** equals the initial value.

```
IF <travel>-Status = 'C' OR
  ( <travel>-EndDate IS NOT INITIAL AND
    <travel>-EndDate < cl_abap_context_info=>get_system_date( ) ).

  <result>-%update                 = if_abap_behv=>fc-o-disabled.
  <result>-%action-cancel_travel = if_abap_behv=>fc-o-disabled.

ELSE.

  <result>-%update                 = if_abap_behv=>fc-o-enabled.
  <result>-%action-cancel_travel = if_abap_behv=>fc-o-enabled.

ENDIF.
```

9. Activate the behavior implementation and test the preview of your OData UI service to see the effect.

   a) Press **Ctrl + F3** to activate the development object.

   b) Reopen the preview for the OData UI service and select a flight travel that has already ended. For example, the travel with the description *Travel in the past*.

      Result
      The *Cancel the Travel* button must be inactive for this flight travel.

   c) For this flight travel, navigate to the object page.

      Result
      You see no *Edit* button for this flight travel.

   d) Repeat this with a travel that has not yet ended, for example, the travel with the description *Travel in the future*.

      Result
      The *Cancel the Travel* button must be active for this travel and the *Edit* button must be visible on the object page.

Task 2: Implement Dynamic Field Control

Enable dynamic field control for the **CustomerId** field and the **BeginDate** field. Set the fields to read-only if the travel has already started, that is, if the starting date lies in the past.

1. In your behavior definition on data model level **Z##_R_Travel**, locate the FIELD statement for the fields **Description**, **CustomerId**, **BeginDate**, and **EndDate**.

Comment or remove **CustomerId** and **BeginDate** and define a new FIELD statement for these two fields in which you enable dynamic field control.

a) Adjust the code as follows:

```
field ( mandatory )
Description,
//  CustomerId,
//  BeginDate,
EndDate;

field ( features : instance )
CustomerId,
BeginDate;
```

2. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

3. Navigate to the method for feature control. After the existing IF-ENDIF control structure, but still inside the loop, implement a second IF-ENDIF structure that checks whether the **BeginDate** component of the current flight travel is not initial and lies in the past.

   a) Adjust the code as follows:

```
  ENDIF.

  IF <travel>-BeginDate IS NOT INITIAL AND
     <travel>-BeginDate < cl_abap_context_info=>get_system_date( ).

  ENDIF.

ENDLOOP.
```

4. Inside the IF-block, disable editing of **CustomerId** and **BeginDate** fields for the current flight travel instance.

Hint:
In the current line of the **result** parameter, update the value of the respective components. Assign these components a suitable component of the attribute **FC-F** from interface **IF_ABAP_BEHV** as value.

   a) Adjust the code as follows:

```
IF <travel>-BeginDate IS NOT INITIAL AND
   <travel>-BeginDate < cl_abap_context_info=>get_system_date( ).

  <result>-%field-CustomerId = if_abap_behv=>fc-f-read_only.
  <result>-%field-BeginDate  = if_abap_behv=>fc-f-read_only.

ENDIF.
```

5. Implement an ELSE branch in which you set the fields to **mandatory**.

Note:
This ELSE branch is not optional because the value of **FC-F-MANDATORY** does not equal the initial value!

a) Adjust the code as follows:

```
IF <travel>-BeginDate IS NOT INITIAL AND
   <travel>-BeginDate < cl_abap_context_info=>get_system_date( ).

  <result>-%field-CustomerId = if_abap_behv=>fc-f-read_only.
  <result>-%field-BeginDate  = if_abap_behv=>fc-f-read_only.

ELSE.

  <result>-%field-CustomerId = if_abap_behv=>fc-f-mandatory.
  <result>-%field-BeginDate  = if_abap_behv=>fc-f-mandatory.

ENDIF.
```

6. Activate the behavior implementation and test the preview of your OData UI service to see the effect.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) Reopen the preview for the OData UI service, select a flight travel that has already begun but not yet ended (for example, the travel with the description *Travel ongoing*) and navigate to the object page.

   c) Choose *Edit*.

   Result

   The fields *Customer ID* and *Starting Date* display as **`read-only`**, while the *Description* and *End Date* fields are still editable and marked as **`mandatory`**.

Unit 4
Exercise 13

Enable Draft Handling in the Business Object

In this exercise, you enable draft handling in your business object and in your OData UI service. To properly support optimistic concurrency control in the OData service, you extend the data model with a timestamp field that is updated during every save of a draft.

Note:
In this exercise, replace ## with your group number.

Table 15: Solution

Repository Object Type	Repository Object ID
Database Table	*/LRN/437D_TRAVEL*
CDS Data Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Data Definition (Projection)	*/LRN/437D_C_TRAVEL*
CDS Metadata Extension	*/LRN/437D_C_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437D_R_TRAVEL*

Task 1: Define a Timestamp Field for Draft Changes

Extend your database table **Z##_TRAVEL** with a timestamp field for changes to the draft instance (suggested name: **loc_changed_at**).

Add the field to your data model view entity **Z##_R_Travel** and annotate it with the necessary sub annotation of the *@Semantics* annotation to ensure that the runtime updates the field during every save of a draft.

Add the field to the field mapping in the behavior definition on data model level **Z##_R_TRAVEL**, disallow direct changes and use it as the etag field for optimistic concurrency control.

Add the field to your projection view entity **Z##_C_Travel** and extend the metadata extension, to keep the field hidden in the OData UI service.

1. Edit the definition of your database table **Z##_TRAVEL**. At the end of the field list, add a **loc_changed_at** field based on data element **ABP_LOCINST_LASTCHANGE_TSTMPL**.
2. Activate the database table.
3. Edit the definition of your data model view entity **Z##_R_Travel**. Add the new table field to the element list and specify the alias name **LocChangedAt**.
4. Add annotation *@Semantics.systemDateTime.localInstanceLastChangedAt: true* to the new view element and activate the data definition.

5. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. After the etag master addition of the **`define behavior`** statement, replace the timestamp for changes to the active data **`ChangedAt`** with the new timestamp field **`LocChangedAt`**.

6. Locate the FIELD statement for the **`STATUS`** field and the administrative fields, and add the new field.

7. Scroll down to the **`mapping for`** statement and add the mapping for the new field. Then activate the behavior definition.

8. Edit the definition of your projection view entity **`Z##_C_Travel`**. At the end of the element list, add the new element and activate the data definition.

9. Edit the metadata extension for the projection view **`Z##_C_TRAVEL`**. Add the new element *LocChangedAt* with annotation *@UI.hidden: true*, and activate the metadata extension.

Task 2: Enable Draft-Handling in the Business Object

In the behavior definition on data model level **`Z##_R_TRAVEL`**, add the necessary statement to enable draft handling. Define and generate a draft table for the travel data (suggested name: **`Z##_TRAVEL_D`**). Use the timestamp for changes to the active data **`ChangedAt`** as the total etag field and add the draft actions to the behavior definition.

1. Edit the behavior definition on data model level **`Z##_R_TRAVEL`** and add the **`with draft;`** statement.

2. Scroll down to the DEFINE BEHAVIOR statement. After persistent table z##_travel add draft table, followed by the name of the draft table that you want to create.

3. Use a quick fix to generate the draft table.

4. Analyze the generated database table, then activate it.

   Which fields of the draft table do not correspond to elements of the view entity?

   ____

   ____

   ____

5. Return to the behavior definition. After lock master, add **`total etag`** followed by the name of the timestamp field for changes to the active data (**`ChangedAt`**).

6. Explicitly declare the draft actions **`Edit`**, **`Activate`**, **`Discard`**, **`Resume`**, and the draft determine action **`Prepare`**.

   Note:
   For now, ignore the warnings regarding the missing addition **`optimized`** for the **`Activate`** action and the missing assignment of the validations to the draft action **`Prepare`**.

7. Activate the behavior definition.

Enable Draft Handling in the Business Object

In this exercise, you enable draft handling in your business object and in your OData UI service. To properly support optimistic concurrency control in the OData service, you extend the data model with a timestamp field that is updated during every save of a draft.

Note:
In this exercise, replace ## with your group number.

Table 15: Solution

Repository Object Type	Repository Object ID
Database Table	*/LRN/437D_TRAVEL*
CDS Data Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Data Definition (Projection)	*/LRN/437D_C_TRAVEL*
CDS Metadata Extension	*/LRN/437D_C_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437D_R_TRAVEL*

Task 1: Define a Timestamp Field for Draft Changes

Extend your database table **Z##_TRAVEL** with a timestamp field for changes to the draft instance (suggested name: **loc_changed_at**).

Add the field to your data model view entity **Z##_R_Travel** and annotate it with the necessary sub annotation of the @*Semantics* annotation to ensure that the runtime updates the field during every save of a draft.

Add the field to the field mapping in the behavior definition on data model level **Z##_R_TRAVEL**, disallow direct changes and use it as the etag field for optimistic concurrency control.

Add the field to your projection view entity **Z##_C_Travel** and extend the metadata extension, to keep the field hidden in the OData UI service.

1. Edit the definition of your database table **Z##_TRAVEL**. At the end of the field list, add a **loc_changed_at** field based on data element **ABP_LOCINST_LASTCHANGE_TSTMPL**.

   a) Open the definition of your database table.

   b) At the end of the field list, add the following code:

```
loc_changed_at : abp_locinst_lastchange_tstmpl;
```

2. Activate the database table.

   a) Press **Ctrl + F3** to activate the development object.

3. Edit the definition of your data model view entity **`Z##_R_Travel`**. Add the new table field to the element list and specify the alias name **`LocChangedAt`**.

   a) Adjust the code as follows:

```
@Semantics.systemDateTime.lastChangedAt: true
changed_at     as ChangedAt,
@Semantics.user.lastChangedBy: true
changed_by     as ChangedBy,
loc_changed_at as LocChangedAt
```

4. Add annotation *@Semantics.systemDateTime.localInstanceLastChangedAt: true* to the new view element and activate the data definition.

   a) Adjust the code as follows:

```
@Semantics.systemDateTime.lastChangedAt: true
changed_at     as ChangedAt,
@Semantics.user.lastChangedBy: true
changed_by     as ChangedBy,
@Semantics.systemDateTime.localInstanceLastChangedAt: true
loc_changed_at as LocChangedAt
```

   b) Press **`Ctrl + F3`** to activate the development object.

5. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. After the etag master addition of the **`define behavior`** statement, replace the timestamp for changes to the active data **`ChangedAt`** with the new timestamp field **`LocChangedAt`**.

   a) Adjust the code as follows:

```
etag master LocChangedAt  //ChangedAt
```

6. Locate the FIELD statement for the **`STATUS`** field and the administrative fields, and add the new field.

   a) Adjust the code as follows:

```
field ( readonly )
Status,
ChangedAt,
ChangedBy,
LocChangedAt;
```

7. Scroll down to the **`mapping for`** statement and add the mapping for the new field. Then activate the behavior definition.

   a) Adjust the code as follows:

```
ChangedAt    = changed_at;
ChangedBy    = changed_by;
LocChangedAt = loc_changed_at;
```

   b) Press **`Ctrl + F3`** to activate the development object.

8. Edit the definition of your projection view entity **`Z##_C_Travel`**. At the end of the element list, add the new element and activate the data definition.

   a) Adjust the code as follows:

```
ChangedAt,
ChangedBy,
LocChangedAt
```

   b) Press **`Ctrl + F3`** to activate the development object.

9. Edit the metadata extension for the projection view **`Z##_C_TRAVEL`**. Add the new element *LocChangedAt* with annotation *@UI.hidden: true*, and activate the metadata extension.

a) Adjust the code as follows:

```
@UI.hidden: true
ChangedAt;

@UI.hidden: true
ChangedBy;

@UI.hidden: true
LocChangedAt;
```

b) Press **Ctrl + F3** to activate the development object.

Task 2: Enable Draft-Handling in the Business Object

In the behavior definition on data model level **Z##_R_TRAVEL**, add the necessary statement to enable draft handling. Define and generate a draft table for the travel data (suggested name: **Z##_TRAVEL_D**). Use the timestamp for changes to the active data **ChangedAt** as the total etag field and add the draft actions to the behavior definition.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL** and add the **with draft;** statement.

   a) Adjust the code as follows:

```
managed implementation in class zbp_##_r_travel unique;
strict ( 2 );

with draft;

define behavior for Z##_R_Travel alias Travel
```

   Are there any new syntax errors?

   Yes, there are new syntax errors: one about a missing "total etag" field, one about missing draft persistency and three error messages about the need to explicitly define draft actions.

2. Scroll down to the DEFINE BEHAVIOR statement. After persistent table z##_travel add draft table, followed by the name of the draft table that you want to create.

   a) Adjust the code as follows:

```
define behavior for Z##_R_Travel alias Travel
persistent table z##_travel
draft table z##_travel_d
lock master
```

3. Use a quick fix to generate the draft table.

   a) Place the cursor on the name of the draft table.

   b) Press **Ctrl + 1** to invoke the quick assist proposals.

   c) From the list of available quick fixes, choose *Create draft table z##_travel_d for entity z##_r_travel*.

   d) If desired, adapt the generated description for the draft table and choose *Next*.

   e) Confirm the transport request and choose *Finish*.

4. Analyze the generated database table, then activate it.

Which fields of the draft table do not correspond to elements of the view entity?

The additional fields are the ones in named include `%admin`.

a) The definition of the database table is already opened after the previous step.

b) Press **`Ctrl + F3`** to activate the development object.

5. Return to the behavior definition. After lock master, add **`total etag`** followed by the name of the timestamp field for changes to the active data (**`ChangedAt`**).

   a) Adjust the code as follows:

```
define behavior for Z##_R_Travel alias Travel
persistent table z##_travel
draft table z##_travel_d
lock master
total etag ChangedAt
authorization master ( instance )
etag master LocChangedAt   // ChangedAt
early numbering
{
```

6. Explicitly declare the draft actions **`Edit`**, **`Activate`**, **`Discard`**, **`Resume`**, and the draft determine action **`Prepare`**.

> Note:
> For now, ignore the warnings regarding the missing addition **`optimized`** for the **`Activate`** action and the missing assignment of the validations to the draft action **`Prepare`**.

   a) Add the following code:

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

7. Activate the behavior definition.

   a) Press **`Ctrl + F3`** to activate the development object.

Enable Draft Handling in the OData Service and Adjust the Implementation

In this exercise, you expose the draft capabilities of your business object to your OData UI service and test the draft functionality in the service preview. You then adjust the behavior definition and implementation.

Note:
In this exercise, replace ## with your group number.

Table 16: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437D_C_TRAVEL*
ABAP Class	*/LRN/BP_437D_R_TRAVEL*

Task 1: Analyze Draft Handling in SAP Fiori Elements App

Make the draft capabilities of your business object visible to your OData UI service. Then test the preview of the draft enabled OData UI service.

1. Edit the behavior definiton on projection level **`Z##_C_TRAVEL`** and add the **`use draft;`** statement.
2. Add the five draft actions to the behavior projection to make them part of the OData service.
3. Activate the behavior definition.
4. Reopen the preview for your OData UI service. Create a new flight travel. Leave all input fields empty and return to the list page without choosing *Create* or *Discard Draft*.
5. Display the content of the draft table **`Z##_TRAVEL_D`** in the data preview tool and verify that it contains the draft data of the new travel.

   Which fields have already been filled in the draft?

Why are the fields **`CHANGEDAT`** and **`CHANGEDBY`** not filled yet?

6. Go back to the preview of your OData UI service and resume editing the new flight travel.
7. Leave all input fields empty but this time choose *Create*.

   Is there a problem with the way the error messages from the validations are displayed?

   What must be done to fix the display of the validation messages?

Task 2: Adjust the Validations

Adjust the behavior definition on data model level. Attach all validations to the draft determine action **`Prepare`** to ensure that they are called before draft instances are transferred to active instances. Adjust the implementation of all validations to report the validation messages as state messages rather than transition messages.

1. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. Locate the draft determine action statement, and replace the semicolon after **`Prepare`** with a pair of curly brackets.
2. Inside the curly brackets, list all validations of the root entity of your business object.
3. Make sure that the validations are not executed again during the **`Activation`** if they were already executed during the **`Preparation`**. Then activate the behavior definition.

   Hint:
   You achieve this by defining the Activate action as **`optimized`**.

4. Navigate to the implementation of the **`validateDescription`** validation.
5. Locate the statement where you add a row to the **`travel`** component of the **`reported`** parameter and adjust it so that you fill the component **`%state_area`** with a non-initial value, for example, **`DESC`**.

   Hint:
   To increase the robustness and supportability of your code, we recommend that you define a local constant (suggested name: **`c_area`**) of type string for this value.

6. Scroll up to the beginning of the loop. Before the actual check, add a row to **`reported-travel`** that contains only the transactional key (**`%tky`**) and the same value for the state area.

> Hint:
> You can save time by copying the APPEND statement from inside the IF-bock to the beginning of the loop. Just make sure you delete the value assignment to components **`%msg`** and **`%element`**.

7. Adjust the implementation of the **`validateCustomer`** validation in a similar way.
8. If you implemented validations for the date fields in a previous exercise, adjust the implementation of the **`validateBeginDate`** validation, the **`validateEndDate`** validation, and the **`validateDateSequence`** validation in a similar way. In each method, use a different value for the **`C_AREA`** constant, for example **`BEGINDATE`**, **`ENDDATE`**, and **`SEQUENCE`**.
9. Activate the global class with the behavior implementation and test the display of the validation messages in the preview of the OData UI service.

Task 3: Adjust Dynamic Feature Control

Adjust the feature control implementation to properly handle drafts. For edit draft instances (not new draft instances), the operation and field control must be based on the starting date and end date from the corresponding active instance and not on the values from the draft instance.

1. Edit the implementation of the **`get_instance_features`** method. Locate the loop over the affected flight travels. At the beginning of the loop, after you added a new row to the **`result`** parameter, check whether the current flight travel is a draft instance.

> Hint:
> Compare the **`%is_draft`** component to the corresponding component of the **`mk`** constant in the **`IF_ABAP_BEHV`** interface.

2. If the current flight travel is a draft instance, try reading the starting date and the end date from the corresponding active instance and store the result in an inline-declared data object (suggested name: **`travels_active`**).

> Hint:
> To read the active instance, use the same value for **`%key`** but the initial value for **`%is_draft`**.

3. If the draft is an edit draft, that is, if there is a related active instance, overwrite the starting date and the end date in the current row of **`travels`** with the values from the active instance.
4. If the draft is a new draft, that is, if there is no related active instance, overwrite the starting date and the end date in the current row of **`travels`** with initial values.

5. Activate the behavior implementation and test the feature control in the preview of the OData UI service.

Enable Draft Handling in the OData Service and Adjust the Implementation

In this exercise, you expose the draft capabilities of your business object to your OData UI service and test the draft functionality in the service preview. You then adjust the behavior definition and implementation.

Note:
In this exercise, replace ## with your group number.

Table 16: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437D_C_TRAVEL*
ABAP Class	*/LRN/BP_437D_R_TRAVEL*

Task 1: Analyze Draft Handling in SAP Fiori Elements App

Make the draft capabilities of your business object visible to your OData UI service. Then test the preview of the draft enabled OData UI service.

1. Edit the behavior definiton on projection level **`Z##_C_TRAVEL`** and add the **`use draft;`** statement.

   a) Adjust the code as follows:

```
projection;
strict ( 2 );

use draft;

define behavior for Z##_C_Travel
```

2. Add the five draft actions to the behavior projection to make them part of the OData service.

   a) Add the following code:

```
use action Edit;
use action Activate;
use action Discard;
use action Prepare;
use action Resume;
```

3. Activate the behavior definition.

   a) Press **`Ctrl + F3`** to activate the development object.

4. Reopen the preview for your OData UI service. Create a new flight travel. Leave all input fields empty and return to the list page without choosing *Create* or *Discard Draft*.
   a) Perform this step as before.

5. Display the content of the draft table **`Z##_TRAVEL_D`** in the data preview tool and verify that it contains the draft data of the new travel.
   a) Locate the draft table in the *Project Explorer*.
   b) Open the context menu for the draft table and choose *Open With → Data Preview*.

      Result

      You see one entry in the draft table.

   Which fields have already been filled in the draft?

   The primary key fields (**`MANDT`**, **`AGENCYID`**, **`TRAVELID`**), the **`STATUS`** field, the timestamp field for changes to the local instance **`LOCCHANGEDAT`**, and the administrative fields from the include structure

   Why are the fields **`CHANGEDAT`** and **`CHANGEDBY`** not filled yet?

   These fields are filled when the draft gets activated, that is, when the draft data is copied to the table for active data.

6. Go back to the preview of your OData UI service and resume editing the new flight travel.
   a) From the list of flight travels, select the one labeled *Draft* and navigate to the object page.

7. Leave all input fields empty but this time choose *Create*.
   a) Perform this step as before.

   Is there a problem with the way the error messages from the validations are displayed?

   Yes, the error messages are no longer related to the input fields.

   What must be done to fix the display of the validation messages?

   You must attach the validations to the **`Prepare`** determine action and you have to classify the validation messages as state messages.

Task 2: Adjust the Validations

Adjust the behavior definition on data model level. Attach all validations to the draft determine action **`Prepare`** to ensure that they are called before draft instances are transferred to active instances. Adjust the implementation of all validations to report the validation messages as state messages rather than transition messages.

1. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. Locate the draft determine action statement, and replace the semicolon after **`Prepare`** with a pair of curly brackets.

a) Adjust the code as follows:

```
draft determine action Prepare {  }
```

2. Inside the curly brackets, list all validations of the root entity of your business object.

   a) Adjust the code as follows:

   > Note:
   > If you did not implement validations for the date fields, your list of validations is shorter.

```
draft determine action Prepare
{
  validation validateDescription;
  validation validateCustomer;
  validation validateBeginDate;
  validation validateEndDate;
  validation validateDateSequence;
}
```

3. Make sure that the validations are not executed again during the **Activation** if they were already executed during the **Preparation**. Then activate the behavior definition.

   > Hint:
   > You achieve this by defining the Activate action as **optimized**.

   a) Add the **optimized** addition to the definition of the Activate action as follows:

```
draft action Activate optimized;
```

   b) Press **Ctrl + F3** to activate the development object.

4. Navigate to the implementation of the **validateDescription** validation.

   a) Hold down the **Ctrl** key and choose the name of the validation **validateDescription**.

5. Locate the statement where you add a row to the **travel** component of the **reported** parameter and adjust it so that you fill the component **%state_area** with a non-initial value, for example, **DESC**.

   > Hint:
   > To increase the robustness and supportability of your code, we recommend that you define a local constant (suggested name: **c_area**) of type string for this value.

   a) At the beginning of the method implementation, add the following code:

```
CONSTANTS c_area TYPE string VALUE `DESC`.
```

   b) Adjust the statement where you fill **reported-travel** as follows:

```
APPEND VALUE #( %tky                 = <travel>-%tky
                %msg                 = NEW /lrn/cm_s4d437(
                               /lrn/cm_s4d437=>field_empty )
                %element-Description = if_abap_behv=>mk-on
                %state_area          = c_area )
  TO reported-travel.
```

6. Scroll up to the beginning of the loop. Before the actual check, add a row to **reported-travel** that contains only the transactional key (**%tky**) and the same value for the state area.

   Hint:
   You can save time by copying the APPEND statement from inside the IF-bock to the beginning of the loop. Just make sure you delete the value assignment to components **%msg** and **%element**.

   a) Adjust the ode as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  APPEND VALUE #( %tky        = <travel>-%tky
                  %state_area = c_area )
    TO reported-travel.

  IF <travel>-Description IS INITIAL.
```

7. Adjust the implementation of the **validateCustomer** validation in a similar way.

   a) Add the following code at the beginning of the method:

```
CONSTANTS c_area TYPE string VALUE `CUST`.
```

   b) Add the following code at the beginning of the loop:

```
APPEND VALUE #( %tky        = <travel>-%tky
                %state_area = c_area )
  TO reported-travel.
```

   c) Adjust the APPEND statement in the IF-block as follows:

```
APPEND VALUE #( %tky                = <travel>-%tky
                %msg                = NEW /lrn/cm_s4d437(
                            /lrn/cm_s4d437=>field_empty )
                %element-CustomerId = if_abap_behv=>mk-on
                %state_area         = c_area )
  TO reported-travel.
```

   d) Adjust the APPEND statement in the ELSE-block as follows:

```
APPEND VALUE #( %tky                = <travel>-%tky
                %msg                = NEW /lrn/cm_s4d437(
                  textid     = /lrn/cm_s4d437=>customer_not_exist
                  customerid = <travel>-CustomerId )
                %element-CustomerId = if_abap_behv=>mk-on
                %state_area         = c_area )
  TO reported-travel.
```

8. If you implemented validations for the date fields in a previous exercise, adjust the implementation of the **validateBeginDate** validation, the **validateEndDate** validation, and the **validateDateSequence** validation in a similar way. In each method, use a different value for the **C_AREA** constant, for example **BEGINDATE**, **ENDDATE**, and **SEQUENCE**.

   a) Adjust the code as before.

9. Activate the global class with the behavior implementation and test the display of the validation messages in the preview of the OData UI service.

a) Press **Ctrl + F3** to activate the development object.

Task 3: Adjust Dynamic Feature Control

Adjust the feature control implementation to properly handle drafts. For edit draft instances (not new draft instances), the operation and field control must be based on the starting date and end date from the corresponding active instance and not on the values from the draft instance.

1. Edit the implementation of the **get_instance_features** method. Locate the loop over the affected flight travels. At the beginning of the loop, after you added a new row to the **result** parameter, check whether the current flight travel is a draft instance.

   Hint:
   Compare the **%is_draft** component to the corresponding component of the **mk** constant in the **IF_ABAP_BEHV** interface.

   a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  APPEND CORRESPONDING #( <travel> ) TO result
    ASSIGNING FIELD-SYMBOL(<result>).

  IF <travel>-%is_draft = if_abap_behv=>mk-on.

  ENDIF.
```

2. If the current flight travel is a draft instance, try reading the starting date and the end date from the corresponding active instance and store the result in an inline-declared data object (suggested name: **travels_active**).

   Hint:
   To read the active instance, use the same value for **%key** but the initial value for **%is_draft**.

   a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  APPEND CORRESPONDING #( <travel> ) TO result
    ASSIGNING FIELD-SYMBOL(<result>).

  IF <travel>-%is_draft = if_abap_behv=>mk-on.

    READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        FIELDS ( BeginDate EndDate )
        WITH VALUE #( ( %key = <travel>-%key ) )
        RESULT DATA(travels_active).

  ENDIF.
```

3. If the draft is an edit draft, that is, if there is a related active instance, overwrite the starting date and the end date in the current row of **travels** with the values from the active instance.

a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  APPEND CORRESPONDING #( <travel> ) TO result
    ASSIGNING FIELD-SYMBOL(<result>).

  IF <travel>-%is_draft = if_abap_behv=>mk-on.

    READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        FIELDS ( BeginDate EndDate )
        WITH VALUE #( ( %key = <travel>-%key ) )
        RESULT DATA(travels_active).

    IF travels_active IS NOT INITIAL.
      <travel>-BeginDate = travels_active[ 1 ]-BeginDate.
      <travel>-EndDate   = travels_active[ 1 ]-EndDate.
    ENDIF.

  ENDIF.
```

4. If the draft is a new draft, that is, if there is no related active instance, overwrite the starting date and the end date in the current row of **`travels`** with initial values.

   a) Adjust the code as follows:

```
LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).

  APPEND CORRESPONDING #( <travel> ) TO result
    ASSIGNING FIELD-SYMBOL(<result>).

  IF <travel>-%is_draft = if_abap_behv=>mk-on.

    READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
      ENTITY Travel
        FIELDS ( BeginDate EndDate )
        WITH VALUE #( ( %key = <travel>-%key ) )
        RESULT DATA(travels_active).

    IF travels_active IS NOT INITIAL.
      <travel>-BeginDate = travels_active[ 1 ]-BeginDate.
      <travel>-EndDate   = travels_active[ 1 ]-EndDate.
    ELSE.
      CLEAR <travel>-BeginDate.
      CLEAR <travel>-EndDate.
    ENDIF.
  ENDIF.
```

5. Activate the behavior implementation and test the feature control in the preview of the OData UI service.

   a) Press **`Ctrl + F3`** to activate the development object.

Define Determine Actions and Side Effects

In this exercise, you define determine actions to enable the direct execution of validations and determinations. Then you use side effects to make sure the determine actions are triggered by the UI after certain user actions.

Note:
In this exercise, replace ## with your group number.

Table 17: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Data Definition (Projection)	*/LRN/437D_C_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437D_C_TRAVEL*

Task 1: Define a Determine Action and a Side Effect

In your business object, define a determine action (suggested name: **`checkCustomer`**) to trigger the validation of the customer ID. Expose the determine action to the OData UI service and define a side effect to execute the determine action when the user changes the customer ID.

1. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. After the validations and determination definition, define a determine action (suggested name: **`checkCustomer`**) that triggers the validation of the customer ID.
2. Activate the behavior definition.
3. Edit the behavior definition on projection level **`Z##_C_TRAVEL`** and add the determine action to the projection.
4. After the projection of the determine action, define a side effect that triggers the determine action.
5. Use changes to the **`CustomerId`** field as the side effect trigger.
6. Specify the messages as the side effect target that the UI might have to reload after the execution of the side effect.
7. Activate the behavior definition.
8. Restart the OData UI service preview and verify that the determine action is executed after you edit the Customer ID.

Task 2: Define a Calculated View Element

Extend the root entity of your business object with a view element for the duration of the travel in days (suggested name: **`Duration`**). Use the **`dats_days_between`** SQL function to calculate this information based on the persistent values of **`begin_date`** and **`end_date`**.

Expose the new view element to your OData UI service and add it to the list page and the object page. Then set it to read-only.

1. Edit the CDS view entity on data model level **`Z##_R_Travel`**. Insert a new row before the **`Status`** view element, and use code completion to insert a call of SQL function **`dats_days_between`**.
2. Supply the function with the table fields **`begin_date`** and **`end_date`** and define a name for the new view element (suggested name: **`Duration`**) .
3. Activate the data definition.
4. Now edit the projection view **`Z##_C_Travel`**. Add the new view element and annotate this element with a suitable label, for example, **`Duration (days)`**.
5. Activate the data definition.
6. Now edit the metadata extension **`Z##_C_TRAVEL`** and add *@UI.lineItem* and *@UI.identification* annotations for the new view element.
7. Activate the metadata extension.
8. Now open the behavior definition on data model level **`Z##_R_TRAVEL`** and use a quick fix to add the new field to the draft table.
9. Return to the behavior definition and add the new view element to the same FIELD statement as the **`Status`** view element.
10. Use a pragma to suppress the warning *Field "DURATION" of entity "Z##_R_TRAVEL" does not have a mapping to table "Z##_TRAVEL". A "mapping" definition should be added.*.

> Hint:
> You find the relevant pragma in the *Problem Description* of the syntax warning.

Task 3: Update the Calculated Field with a Side Effect

In your business object, define and implement a determination that calculates the value of the **`Duration`** field based on the starting date and end date (suggested name: **`determineDuration`**). Make sure that the determination is executed during the save phase and is triggered by changes to either of the two date fields.

Then define a determine action (suggested name: **`adjustDuration`**) to trigger the execution of the determination and the validations of the date fields - in case you did the optional part of that exercise. Then define a side effect that is triggered by changes to either of the date fields.

Note:
Because this side effect triggers a determination, it is possible that it is not only relevant for the UI service. Therefore, you define it on data model level and let the behavior projection use the side effect of the underlying business object.

1. In the behavior definition on data model level **`Z##_R_TRAVEL`**, define a determination (suggested name: **`determineDuration`**) that is executed during save and triggered by a change of the **`BeginDate`** element or the **`EndDate`** element.
2. Activate the behavior definition and use a quick fix to add the determination implementation method to the local handler class.
3. In the method implementation, use EML in local mode to read the starting date and the end date of all affected travel instances and store the result in an inline declared internal table (suggested name: **`travels`**).
4. Implement a loop over the travels, assigning an inline declared field symbol (suggested name: **`<travel>`**). Inside the loop, update the **`Duration`** field in the internal table with the difference between **`EndDate`** and **`BeginDate`**.
5. After the loop, implement an EML statement in local mode to update the duration for all affected travel instances.

   Hint:
   You can use your internal table **`travels`** directly after the WITH addition if you place it inside a CORRESPONDING expression.

6. Activate the behavior implementation and return to the behavior definition. Define a determine action (suggested name: **`adjustDuration`**) that triggers the execution of the determination and the validations for the date fields.

   Note:
   If you did not implement validations for the date fields, let the determine action only trigger the determination.

7. Define a side effect that executes the determine action **`adjustDuration`** after the user changed either of the fields **`BeginDate`** or **`EndDate`** and specifies field **`Duration`** and the **`messages`** as targets.
8. Activate the behavior definition on data model level and edit the behavior definition on projection level. Add the determine action **`adjustDuration`** to the projection of the root entity and make the projection reuse the side effects that are defined on data model level.
9. Activate the behavior projection. Then reload the OData UI service preview and verify that the determine action is executed after you edit the date fields.

Unit 4
Solution 15

Define Determine Actions and Side Effects

In this exercise, you define determine actions to enable the direct execution of validations and determinations. Then you use side effects to make sure the determine actions are triggered by the UI after certain user actions.

Note:
In this exercise, replace ## with your group number.

Table 17: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Data Definition (Projection)	*/LRN/437D_C_TRAVEL*
CDS Behavior Definition (Model)	*/LRN/437D_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437D_C_TRAVEL*

Task 1: Define a Determine Action and a Side Effect
In your business object, define a determine action (suggested name: **checkCustomer**) to trigger the validation of the customer ID. Expose the determine action to the OData UI service and define a side effect to execute the determine action when the user changes the customer ID.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL**. After the validations and determination definition, define a determine action (suggested name: **checkCustomer**) that triggers the validation of the customer ID.
   a) After the definition of the validations and the determination, add the following code:

```
determine action checkCustomer
{
  validation validateCustomer;
}
```

2. Activate the behavior definition.
   a) Press **Ctrl + F3** to activate the development object.

3. Edit the behavior definition on projection level **Z##_C_TRAVEL** and add the determine action to the projection.
   a) Add the following code:

```
use action checkCustomer;
```

4. After the projection of the determine action, define a side effect that triggers the determine action.

a) Add the following code:

```
side effects
{
  determine action checkCustomer

}
```

5. Use changes to the **CustomerId** field as the side effect trigger.

   a) Adjust your code as follows:

```
side effects
{
  determine action checkCustomer
  executed on field CustomerId

}
```

6. Specify the messages as the side effect target that the UI might have to reload after the execution of the side effect.

   a) Adjust your code as follows:

```
side effects
{
  determine action checkCustomer
  executed on field CustomerId
  affects messages;
}
```

7. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

8. Restart the OData UI service preview and verify that the determine action is executed after you edit the Customer ID.

   a) Open the service binding, place the cursor on the root entity, and choose *Preview...*.

   b) In the preview, display the details for one of the travels. Change to edit mode and delete the value in the *Customer ID* field.

   c) Press **Tab** to save the draft and jump to the next input field.

   Result

   You immediately see the error message from the validation.

Task 2: Define a Calculated View Element

Extend the root entity of your business object with a view element for the duration of the travel in days (suggested name: **Duration**). Use the **dats_days_between** SQL function to calculate this information based on the persistent values of **begin_date** and **end_date**.

Expose the new view element to your OData UI service and add it to the list page and the object page. Then set it to read-only.

1. Edit the CDS view entity on data model level **Z##_R_Travel**. Insert a new row before the **Status** view element, and use code completion to insert a call of SQL function **dats_days_between**.

   a) In the element list of the view definition, insert a new row above the **Status** view element.

   b) Enter **dats** and press **Ctrl + Space** to invoke code completion.

   c) From the suggestion list, choose *dats_days_between(date1, date2) (function)*.

2. Supply the function with the table fields **begin_date** and **end_date** and define a name for the new view element (suggested name: **Duration**) .
   a) Adjust the code as follows:

```
end_date                                  as EndDate,

dats_days_between( begin_date, end_date ) as Duration,

status                                    as Status,
```

3. Activate the data definition.
   a) Press **Ctrl + F3** to activate the development object.
4. Now edit the projection view **Z##_C_Travel**. Add the new view element and annotate this element with a suitable label, for example, **Duration (days)**.
   a) Adjust the code as follows:

```
EndDate,

@EndUserText.label: 'Duration (days)'
Duration,

Status,
```

5. Activate the data definition.
   a) Press **Ctrl + F3** to activate the development object.
6. Now edit the metadata extension **Z##_C_TRAVEL** and add *@UI.lineItem* and *@UI.identification* annotations for the new view element.
   a) Add the following code:

```
@UI: {
  lineItem:       [ { position: 35, importance: #LOW } ],
  identification: [ { position: 65, importance: #LOW } ]
     }
Duration;
```

7. Activate the metadata extension.
   a) Press **Ctrl + F3** to activate the development object.
8. Now open the behavior definition on data model level **Z##_R_TRAVEL** and use a quick fix to add the new field to the draft table.
   a) Choose the error icon with a light bulb next to the `draft table` statement.
   b) Choose *Recreate draft table z##_travel_d for entity z##_r_travel*.
   c) Press **Ctrl + F3** to activate the draft table.
9. Return to the behavior definition and add the new view element to the same FIELD statement as the **Status** view element.
   a) Adjust the code as follows:

```
field ( readonly )
Status,
Duration,
ChangedAt,
ChangedBy,
LocChangedAt;
```

10. Use a **pragma** to suppress the warning *Field "DURATION" of entity "Z##_R_TRAVEL" does not have a mapping to table "Z##_TRAVEL". A "mapping" definition should be added.*.

Hint:
You find the relevant pragma in the *Problem Description* of the syntax warning.

a) In the *Problems* view, right-click the syntax warning and choose *Problem Description*.

b) Adjust the code as follows:

```
define behavior for Z##_R_Travel alias Travel
  persistent table z##_travel ##UNMAPPED_FIELD
  draft table z##_travel_d
```

Task 3: Update the Calculated Field with a Side Effect

In your business object, define and implement a determination that calculates the value of the **Duration** field based on the starting date and end date (suggested name: **determineDuration**). Make sure that the determination is executed during the save phase and is triggered by changes to either of the two date fields.

Then define a determine action (suggested name: **adjustDuration**) to trigger the execution of the determination and the validations of the date fields - in case you did the optional part of that exercise. Then define a side effect that is triggered by changes to either of the date fields.

Note:
Because this side effect triggers a determination, it is possible that it is not only relevant for the UI service. Therefore, you define it on data model level and let the behavior projection use the side effect of the underlying business object.

1. In the behavior definition on data model level **Z##_R_TRAVEL**, define a determination (suggested name: **determineDuration**) that is executed during save and triggered by a change of the **BeginDate** element or the **EndDate** element.

   a) Add the following code:

```
determination determineDuration on save
{ field BeginDate,
  EndDate;
}
```

2. Activate the behavior definition and use a quick fix to add the determination implementation method to the local handler class.

   a) Press **Ctrl + F3** to activate the development object.

   b) In the **determination** statement, place the cursor on the name of the determination and press **Ctrl + 1** to invoke the quick assist proposals.

   c) From the list of available quick fixes, choose *Add method for determination determineduration of entity z##_r_travel in local class lhc_travel*.

3. In the method implementation, use EML in local mode to read the starting date and the end date of all affected travel instances and store the result in an inline declared internal table (suggested name: **travels**).

a) Adjust the code as follows:

```
METHOD determineDuration.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS (  BeginDate EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

ENDMETHOD.
```

4. Implement a loop over the travels, assigning an inline declared field symbol (suggested name: **`<travel>`**). Inside the loop, update the **`Duration`** field in the internal table with the difference between **`EndDate`** and **`BeginDate`**.

   a) Adjust the code as follows:

```
METHOD determineDuration.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS (  BeginDate EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
    <travel>-Duration = <travel>-EndDate - <travel>-BeginDate.
  ENDLOOP.

ENDMETHOD.
```

5. After the loop, implement an EML statement in local mode to update the duration for all affected travel instances.

> Hint:
> You can use your internal table **`travels`** directly after the WITH addition if you place it inside a CORRESPONDING expression.

   a) Adjust the code as follows:

```
METHOD determineDuration.

  READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      FIELDS (  BeginDate EndDate )
      WITH CORRESPONDING #( keys )
      RESULT DATA(travels).

  LOOP AT travels ASSIGNING FIELD-SYMBOL(<travel>).
    <travel>-Duration = <travel>-EndDate - <travel>-BeginDate.
  ENDLOOP.

  MODIFY ENTITIES OF Z##_R_Travel IN LOCAL MODE
    ENTITY Travel
      UPDATE
      FIELDS ( Duration )
      WITH CORRESPONDING #( travels ).

ENDMETHOD.
```

6. Activate the behavior implementation and return to the behavior definition. Define a determine action (suggested name: **`adjustDuration`**) that triggers the execution of the determination and the validations for the date fields.

> Note:
> If you did not implement validations for the date fields, let the determine action only trigger the determination.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) In the behavior definition, after the definition of the first determine action, add the following code:

```
determine action adjustDuration
{
  validation validateBeginDate;
  validation validateEnddate;
  validation validateDateSequence;

  determination determineDuration;
}
```

7. Define a side effect that executes the determine action **`adjustDuration`** after the user changed either of the fields **`BeginDate`** or **`EndDate`** and specifies field **`Duration`** and the **`messages`** as targets.

   a) After the definition of the determine action, add the following code:

```
side effects
{
  determine action adjustDuration
  executed on field BeginDate,
  field EndDate
  affects field Duration,
      messages;
}
```

8. Activate the behavior definition on data model level and edit the behavior definition on projection level. Add the determine action **`adjustDuration`** to the projection of the root entity and make the projection reuse the side effects that are defined on data model level.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) In the behavior definition on projection level, adjust the code as follows:

```
projection;
strict ( 2 );

use draft;
use side effects;

define behavior for Z##_C_Travel
use etag
{


  use action adjustDuration;

}
```

9. Activate the behavior projection. Then reload the OData UI service preview and verify that the determine action is executed after you edit the date fields.

a) Press **`Ctrl + F3`** to activate the development object.

b) Open the service binding, place the cursor on the root entity and choose *Preview…*.

c) In the preview, display the details for one of the travels. Change to edit mode and change the value in the *Starting Date* field.

d) Press **`Tab`** to save the draft and jump to the next input field.

   Result

   The determine action is not yet triggered because there is a second trigger field.

e) Change the value of the *End Date* field and press **`Tab`** to save the draft and jump to the next input field.

   Result

   Now, the determine action is triggered. If your input is not valid, you see error messages. If your input is valid, the content of the *Duration* field is updated.

Unit 5
Exercise 16

Define a Composite Business Object

In this exercise, you extend your business object for flight travels. Each flight travel becomes a composition of header data (*root entity*) and several flight travel items (*child entity*). First you copy and adjust the template repository objects for the flight travel items. Then you define the composition. The template repository objects are located in ABAP package **/LRN/S4D437_TEMPLATE**.

Note:
In this exercise, replace ## with your group number.

Table 18: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model, Root)	*/LRN/437E_R_TRAVEL*
CDS Data Definition (Model, Child)	*/LRN/437E_R_TRAVELITEM*
CDS Behavior Definition (Model)	*/LRN/437E_R_TRAVEL*

Task 1: Copy Template for Data Model (Child Entity)
Create copies of the following repository objects and place them in your own ABAP package **ZS4D437_##**. For the repository object names, take the name of the template and replace **/LRN/437T** with **Z##**. Make sure that the copies refer to each other and not to the template objects.

Table 19: Template

Repository Object Type	Repository Object ID
Database Table	*/LRN/437T_TRITEM*
CDS Data Definition	*/LRN/437T_R_TRAVELITEM*

1. Copy transparent table **/LRN/437T_TRITEM** to transparent table **Z##_TRITEM** and activate the new repository object.
2. Create a copy of data definition **/LRN/437T_R_TRAVELITEM** (suggested name: **Z##_R_TRAVELITEM**).
3. Replace the database table in the FROM clause with your own database table **Z##_TRITEM** and activate the new repository object.

Task 2: Define the Composition
In the data definitions on data model level **Z##_R_TRAVEL** and **Z##_R_TRAVELITEM**, add the required associations to establish a parent-child relation between flight travels and flight travel items.

1. Edit the data definition for your child entity **`Z##_R_TRAVELITEM`**. Add the statement **`association to parent`**, followed by the name of your root entity **`Z##_R_Travel`**.
2. Add a meaningful association name (suggested name: **`_Travel`**) and the ON condition for the association.

> Hint:
> In the ON condition, use the key elements of the parent entity and the corresponding elements in the child entity.

Why are you not able to add a cardinality?

____

____

____

3. Perform a syntax check for the data definition and fix the syntax errors.

Why do you have to remove the root keyword?

____

____

____

Why do you have to expose the to parent association?

____

____

____

4. Ignore the *The parent entity Z##_R_Travel does not have a composition definition for Z##_R_TravelItem* warning and activate the data definition.
5. Edit the data definition for the parent entity **`Z##_R_TRAVEL`** and add the statement **`composition of`**, followed by the name of your child entity **`Z##_R_TravelItem`**.
6. Add a suitable cardinality and a meaningful association name (suggested name: **`_TravelItem`**).

Why is it not possible to add an ON condition?

____

____

____

7. Add the composition at the end of the element list of the view entity.
8. Activate the data definition.

Task 3: Add the Composition to the Behavior

Add the child entity to the behavior definition on data model level **`Z##_R_TRAVEL`**. Complete the behavior definition to remove all syntax errors and warnings. In particular, link the child entity to its persistent table **`Z##_TRITEM`** and a (generated) draft table (suggested name: **`Z##_TRITEM_D`**). Define the authorization handling and lock handling for the child entity as being dependent on the root entity, but specify a dedicated etag field for the child entity. Define the field mapping, static field control and managed internal numbering. Finally, add the associations to the behavior definition.

1. Edit your behavior definition on data model level **`Z##_R_TRAVEL`**. At the end of the source code, use a code template to insert a new **`define behavior`** statement. Replace the **`entity`** placeholder with the name of your child entity **`Z##_R_TravelItem`** and the alias placeholder with a suitable alias (suggested name: **`Item`**).
2. Use the persistent table addition to link the behavior definition to your transparent table for travel items **`Z##_TRITEM`** and the draft table addition to link the behavior definition to a (not yet existing) draft table for travel items (suggested name: **`Z##_TRITEM_D`**).
3. Use a quick fix to generate the draft table and activate it.
4. Return to the behavior definition. Use the authorization dependent by and lock dependent by additions to make lock and authorization handling for the child entity dependent from the root entity.

   Hint:
   Use the name of the to parent association to reference the root entity as the authorization master and the lock master.

5. Use the etag master addition to enable optimistic concurrency control based on the timestamp for the last local change to a travel item (field **`LocChangedAt`**).
6. Add the standard operations update and delete to the behavior definition for the child entity.

   Why is it not possible to also add the standard operation create?

   ____

   ____

   ____

7. To enable persistence for all elements of your child entity, define the mapping between table field names and CDS view element names.

   Hint:
   To reduce typing effort, you may copy the mapping statement from the model solution, that is, behavior definition **`/LRN/437E_R_TRAVEL`**.

8. Enable managed internal numbering for the technical key field **`ItemUuid`** and set all fields that are used in the definition of the to parent association to read-only.
9. Add the to parent association **`_Travel`** to the behavior definition of the child entity and the composition **`_TravelItem`** to the behavior definition of the root entity. Create-enable the composition when you add it to the behavior for the root entity.

Note:
Remember that your business object is draft-enabled. Therefore, do not forget to draft-enable the associations.

10. Activate the behavior definition.

Unit 5
Solution 16

Define a Composite Business Object

In this exercise, you extend your business object for flight travels. Each flight travel becomes a composition of header data (*root entity*) and several flight travel items (*child entity*). First you copy and adjust the template repository objects for the flight travel items. Then you define the composition. The template repository objects are located in ABAP package **/LRN/ S4D437_TEMPLATE**.

Note:
In this exercise, replace ## with your group number.

Table 18: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Model, Root)	*/LRN/437E_R_TRAVEL*
CDS Data Definition (Model, Child)	*/LRN/437E_R_TRAVELITEM*
CDS Behavior Definition (Model)	*/LRN/437E_R_TRAVEL*

Task 1: Copy Template for Data Model (Child Entity)
Create copies of the following repository objects and place them in your own ABAP package **ZS4D437_##**. For the repository object names, take the name of the template and replace **/LRN/437T** with **Z##**. Make sure that the copies refer to each other and not to the template objects.

Table 19: Template

Repository Object Type	Repository Object ID
Database Table	*/LRN/437T_TRITEM*
CDS Data Definition	*/LRN/437T_R_TRAVELITEM*

1. Copy transparent table **/LRN/437T_TRITEM** to transparent table **Z##_TRITEM** and activate the new repository object.
   a) In the *Project Explorer*, expand node */LRN/S4D437_EXERCISE* → */LRN/ S4D437_TEMPLATE* → *Dictionary* → *Database Tables*.
   b) Right-click **/LRN/437T_TRITEM** and choose *Duplicate...*.
   c) Enter the name of your own package and the suggested name for the new database table. Then choose *Next*.
   d) Select the same transport request as before and choose *Finish*.

e) Press **Ctrl + F3** to activate the development object.

2. Create a copy of data definition **/LRN/437T_R_TRAVELITEM** (suggested name: **Z##_R_TRAVELITEM**).

   a) In the *Project Explorer* view, locate the data definition **/LRN/437T_R_TRAVELITEM** in package **/LRN/S4D437_TEMPLATE** and right-click it to open the context menu.

   b) From the context menu, select *Duplicate...*.

   c) Enter the name of your package and the name for the copy. Then choose *Next*.

   d) Assign the new object to the same transport request as before and choose *Finish*.

3. Replace the database table in the FROM clause with your own database table **Z##_TRITEM** and activate the new repository object.

   a) After the FROM keyword, replace **/lrn/437t_tritem** with **z##_tritem**.

   b) Press **Ctrl + F3** to activate the development object.

Task 2: Define the Composition

In the data definitions on data model level **Z##_R_TRAVEL** and **Z##_R_TRAVELITEM**, add the required associations to establish a parent-child relation between flight travels and flight travel items.

1. Edit the data definition for your child entity **Z##_R_TRAVELITEM**. Add the statement **association to parent**, followed by the name of your root entity **Z##_R_Travel**.

   a) Adjust the code as follows:

```
define root view entity Z##_R_TravelItem
  as select from z##_tritem
  association to parent Z##_R_Travel
{
```

2. Add a meaningful association name (suggested name: **_Travel**) and the ON condition for the association.

Hint:
In the ON condition, use the key elements of the parent entity and the corresponding elements in the child entity.

Why are you not able to add a cardinality?

The cardinality of to parent associations is always **[1..1]** and cannot be specified manually.

   a) Adjust the code as follows:

```
define root view entity Z##_R_TravelItem
  as select from z##_tritem
  association to parent Z##_R_Travel as _Travel
    on  $projection.AgencyId = _Travel.AgencyId
    and $projection.TravelId = _Travel.TravelId
{
```

3. Perform a syntax check for the data definition and fix the syntax errors.

Why do you have to remove the root keyword?

The root keyword conflicts with the **to parent** association. An entity is either a child or a root entity.

Why do you have to expose the to parent association?

The association is useless if it is not visible to the consumers of the business object.

a) Press **Ctrl + F2** to perform a syntax check and study the errors displayed on the *Problems* view.

b) Fix the *Root entity cannot define a "TO PARENT" association* error by removing the root keyword from the define view entity statement.

c) Fix the *Association _Travel must be included in the selection list* error by adding the association name at the end of the element list.

4. Ignore the *The parent entity Z##_R_Travel does not have a composition definition for Z##_R_TravelItem* warning and activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

5. Edit the data definition for the parent entity **Z##_R_TRAVEL** and add the statement **composition of**, followed by the name of your child entity **Z##_R_TravelItem**.

   a) Adjust the code as follows:

```
define root view entity Z##_R_Travel
  as select from z##_travel
  composition of Z##_R_TravelItem
{
```

6. Add a suitable cardinality and a meaningful association name (suggested name: **_TravelItem**).

   Why is it not possible to add an ON condition?

   For compositions, the ON condition is automatically derived from the mandatory to parent association in the target entity.

   a) Adjust the code as follows:

```
define root view entity Z##_R_Travel
  as select from z##_travel
  composition [0..*] of Z##_R_TravelItem as _TravelItem
{
```

7. Add the composition at the end of the element list of the view entity.

   a) Adjust the code as follows:

```
  @Semantics.systemDateTime.localInstanceLastChangedAt: true
  loc_changed_at  as LocChangedAt,
  _TravelItem
}
```

8. Activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

Task 3: Add the Composition to the Behavior

Add the child entity to the behavior definition on data model level **Z##_R_TRAVEL**. Complete the behavior definition to remove all syntax errors and warnings. In particular, link the child entity to its persistent table **Z##_TRITEM** and a (generated) draft table (suggested name: **Z##_TRITEM_D**). Define the authorization handling and lock handling for the child entity as being dependent on the root entity, but specify a dedicated etag field for the child entity. Define the field mapping, static field control and managed internal numbering. Finally, add the associations to the behavior definition.

1. Edit your behavior definition on data model level **Z##_R_TRAVEL**. At the end of the source code, use a code template to insert a new **define behavior** statement. Replace the **entity** placeholder with the name of your child entity **Z##_R_TravelItem** and the alias placeholder with a suitable alias (suggested name: **Item**).

   a) After the closing bracket at the end, enter a new code row and press **Ctrl + Space** to invoke code completion.

   b) From the suggestion list, choose *defineBehaviorFor - Define Behavior For*.

   c) Ajust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
{

}
```

2. Use the persistent table addition to link the behavior definition to your transparent table for travel items **Z##_TRITEM** and the draft table addition to link the behavior definition to a (not yet existing) draft table for travel items (suggested name: **Z##_TRITEM_D**).

   a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
persistent table z##_tritem
draft table z##_tritem_d
{

}
```

3. Use a quick fix to generate the draft table and activate it.

   a) Place the cursor on the name of the draft table and choose **Ctrl + 1** to invoke the quick assist proposals.

   b) From the list of available quick fixes, choose *Create draft table z##_tritem_d for entity z##_r_travelitem*.

   c) If desired, adapt the generated description for the draft table and choose *Next*.

   d) Select the same transport request as before and choose *Finish*.

   e) Press **Ctrl + F3** to activate the development object.

4. Return to the behavior definition. Use the authorization dependent by and lock dependent by additions to make lock and authorization handling for the child entity dependent from the root entity.

> Hint:
> Use the name of the **to parent** association to reference the root entity as the authorization master and the lock master.

a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
persistent table z##_tritem
draft table z##_tritem_d
authorization dependent by _Travel
lock dependent by _Travel
{

}
```

5. Use the **etag master** addition to enable optimistic concurrency control based on the timestamp for the last local change to a travel item (field **`LocChangedAt`**).

   a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
persistent table z##_tritem
draft table z##_tritem_d
authorization dependent by _Travel
lock dependent by _Travel
etag master LocChangedAt
{

}
```

6. Add the standard operations **update** and **delete** to the behavior definition for the child entity.

   Why is it not possible to also add the standard operation **create**?

   For child entities, the **create** operation is controlled via the behavior for the **`_TravelItem`** composition in the root entity.

   a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
persistent table z##_tritem
draft table z##_tritem_d
authorization dependent by _Travel
lock dependent by _Travel
etag master LocChangedAt
{
  update;
  delete;
}
```

7. To enable persistence for all elements of your child entity, define the mapping between table field names and CDS view element names.

Hint:
To reduce typing effort, you may copy the **mapping** statement from the model solution, that is, behavior definition **/LRN/437E_R_TRAVEL**.

a) Within the curly brackets of the behavior definition, add the following code:

```
mapping for z##_tritem corresponding
  {
    ItemUuid            = item_uuid;
    AgencyId            = agency_id;
    TravelId            = travel_id;
    CarrierId           = carrier_id;
    ConnectionId        = connection_id;
    FlightDate          = flight_date;
    BookingId           = booking_id;
    PassengerFirstName  = passenger_first_name;
    PassengerLastName   = passenger_last_name;
    ChangedAt           = changed_at;
    ChangedBy           = changed_by;
    LocChangedAt        = loc_changed_at;
  }
```

8. Enable managed internal numbering for the technical key field **ItemUuid** and set all fields that are used in the definition of the **to parent** association to read-only.

a) Within the curly brackets, before the **mapping for** statement, add the following code:

```
field ( readonly, numbering : managed )
ItemUuid;

field ( readonly )
AgencyId,
TravelId;
```

9. Add the **to parent** association **_Travel** to the behavior definition of the child entity and the composition **_TravelItem** to the behavior definition of the root entity. Create-enable the composition when you add it to the behavior for the root entity.

Note:
Remember that your business object is draft-enabled. Therefore, do not forget to draft-enable the associations.

a) In the behavior definition for the child entity, add the following code:

```
association _Travel { with draft; }
```

b) In the behavior definition for the root entity, add the following code:

```
association _TravelItem { create; with draft; }
```

10. Activate the behavior definition.

a) Press **Ctrl + F3** to activate the development object.

Unit 5
Exercise 17

Define the Composition in the OData UI Service

In this exercise, you add the child entity to the business object projection and the definition of the OData UI service. Then you add the relevant metadata to display the flight travel items in the service preview.

Note:
In this exercise, replace ## with your group number.

Table 20: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	*/LRN/437E_C_TRAVELITEM*
CDS Behavior Definition (Projection)	*/LRN/437E_C_TRAVEL*
Service Definition	*/LRN/437E_TRAVEL*
CDS Metadata Extension	*/LRN/437E_C_TRAVELITEM*

Task 1: Copy Template for the Projection (Child Entity)
Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace **`/LRN/D437T`** with **`Z##`**. Make sure that the copies refer to your own repository objects and not to the template objects.

Table 21: Template

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	*/LRN/437T_C_TRAVELITEM*
CDS Metadata Extension	*/LRN/437T_C_TRAVELITEM*

1. Create a copy of data definition **`/LRN/437T_C_TRAVELITEM`** (suggested name: **`Z##_C_TravelItem`**).
2. Replace the selection target after the projection on keyword with the name of your own travel items view entity **`Z##_R_TravelItem`** and remove the root keyword.
3. Activate the data definition.
4. Create a copy of metadata extension **`/LRN/437T_C_TRAVELITEM`** (suggested name: **`Z##_C_TRAVELITEM`**).

5. Replace the CDS view after the annotate view keyword with your own projection view for travel items **Z##_C_TravelItem**.

6. Activate the data definition.

Task 2: Add Composition to the Data Model Projection

In the data definitions on projection level **Z##_C_Travel** and **Z##_C_TravelItem**, expose the associations from the underlying data model views. Make sure that the associations do not point at the views on data model level but that they are redirected to point at the respective views on projection level. Make sure that the associations keep their special character as composition and to parent association.

1. Edit the data definition of your projection view for travel items **Z##_C_TravelItem**. At the end of the projection list, add the name of the to parent association, which you defined in the underlying view entity.

2. Redirect the association to the projection view for flight travels.

3. Classify the redirected association as a to parent association.

4. Perform a syntax check.

   Is there a new syntax error?

5. Remove or comment the provider contract addition, then activate the data definition.

   Why do you have to remove the provider contract addition for the projection view definition?

6. Edit the data definition of your projection view for travels **Z##_C_Travel**. At the end of the projection list, add the name of the composition, which you defined in the underlying view entity.

7. Redirect the association to the projection view for flight travel items and classify it as composition.

8. Activate the data definition.

Task 3: Add Composition to the Behavior Projection

Edit the behavior definition on projection level **Z##_C_TRAVEL**. Add a behavior projection for the child entity. Expose its etag definition and add the associations that establish the composition.

1. Edit the behavior definition on projection level **Z##_C_TRAVEL**. At the end of the source code, use a code template to add a new define behavior statement for the child entity **Z##_C_TravelItem**.

2. Add the etag definition to the behavior projection.

3. For the child entity, add the standard operations **update** and **delete** to the projection.
4. Add the **to parent** association **`_Travel`** to the behavior projection of the child entity and the **composition _TravelItem** to the behavior projection of the root entity.

> Note:
> Do not forget to draft-enable both associations and create-enable the composition.

5. Activate the behavior projection.

Task 4: Add Composition to the OData UI Service

Add the child entity of your business object to your service definition **`Z##_TRAVEL`** and extend the UI metadata of the service to display a list of child entities in a second facet on the object page for flight travels.

1. Open your service definition **`Z##_TRAVEL`** and add the projection view for flight travel items. Then activate the service definition.
2. Open your service binding **`Z##_UI_TRAVEL_O2`** and analyze the information under *Entity Set and Association*.

> Note:
> There is no *Refresh* button available in service bindings but you can press **`F5`** to reload the service binding.

What new information do you see?

3. Open the metadata extension for flight travels **`Z##_C_TRAVEL`** and locate the *@UI.facet* annotation. Inside the square brackets, add a comma (,) after the closing curly bracket and insert a copy of the existing **facet** definition.
4. Adjust the second facet definition according to the following table:

Property	Value
id	**`'TravelItem'`**
purpose	**`#STANDARD`**
type	**`#LINEITEM_REFERENCE`**
label	**`'Travel Items'`**
position	**`20`**

5. Inside the second facet definition, add the property **targetElement** with the name of the **composition** as value.

6. Activate the metadata extension and retest the preview for the OData UI service. Make sure that the object page for flight travels displays an (empty) list of flight travel items and that you can create new flight travel items.

Unit 5
Solution 17

Define the Composition in the OData UI Service

In this exercise, you add the child entity to the business object projection and the definition of the OData UI service. Then you add the relevant metadata to display the flight travel items in the service preview.

Note:
In this exercise, replace ## with your group number.

Table 20: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	*/LRN/437E_C_TRAVELITEM*
CDS Behavior Definition (Projection)	*/LRN/437E_C_TRAVEL*
Service Definition	*/LRN/437E_TRAVEL*
CDS Metadata Extension	*/LRN/437E_C_TRAVELITEM*

Task 1: Copy Template for the Projection (Child Entity)
Create copies of the following repository objects and place them in your ABAP package. For the repository object names, take the name of the template and replace **/LRN/D437T** with **Z##**. Make sure that the copies refer to your own repository objects and not to the template objects.

Table 21: Template

Repository Object Type	Repository Object ID
CDS Data Definition (Projection, Child)	*/LRN/437T_C_TRAVELITEM*
CDS Metadata Extension	*/LRN/437T_C_TRAVELITEM*

1. Create a copy of data definition **/LRN/437T_C_TRAVELITEM** (suggested name: **Z##_C_TravelItem**).
   a) In the *Project Explorer* view, locate the **/LRN/437T_C_TRAVELITEM** data definition in the **/LRN/S4D437_TEMPLATE** package and right-click it to open the context menu.
   b) From the context menu, select *Duplicate...*.
   c) Enter the name of your package and the name for the copy. Choose *Next*.
   d) Assign the new object to a transport request and choose *Finish*.

2. Replace the selection target after the projection on keyword with the name of your own travel items view entity **Z##_R_TravelItem** and remove the root keyword.

   a) Adjust the code as follows:

```
define view entity Z##_C_TravelItem
  provider contract transactional_query
  as projection on Z##_R_TravelItem
{
```

3. Activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

4. Create a copy of metadata extension **/LRN/437T_C_TRAVELITEM** (suggested name: **Z##_C_TRAVELITEM**).

   a) In the *Project Explorer* view, locate the **/LRN/437T_C_TRAVELITEM** metadata extension in the **/LRN/S4D437_TEMPLATE** package and right-click it to open the context menu.

   b) From the context menu, select *Duplicate...*.

   c) Enter the name of your package and the name for the copy. Then choose *Next*.

   d) Assign the new object to a transport request and choose *Finish*.

5. Replace the CDS view after the annotate view keyword with your own projection view for travel items **Z##_C_TravelItem**.

   a) Adjust the code as follows:

```
annotate view Z##_C_TravelItem with
{
```

6. Activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

Task 2: Add Composition to the Data Model Projection

In the data definitions on projection level **Z##_C_Travel** and **Z##_C_TravelItem**, expose the associations from the underlying data model views. Make sure that the associations do not point at the views on data model level but that they are redirected to point at the respective views on projection level. Make sure that the associations keep their special character as composition and to parent association.

1. Edit the data definition of your projection view for travel items **Z##_C_TravelItem**. At the end of the projection list, add the name of the to parent association, which you defined in the underlying view entity.

   a) Adjust the code as follows:

```
  LocChangedAt,
  _Travel
}
```

2. Redirect the association to the projection view for flight travels.

   a) Adjust the code as follows:

```
  LocChangedAt,
  _Travel : redirected to Z##_C_Travel
}
```

3. Classify the redirected association as a to parent association.

a) Adjust the code as follows:

```
  LocChangedAt,
  _Travel : redirected to parent Z##_C_Travel
}
```

4. Perform a syntax check.

   Is there a new syntax error?

   Yes, there is a *Provider contract not modifiable if view contains "redirected to parent" associations* error.

   a) Press **Ctrl + F2** and analyze the *Problems* view.

5. Remove or comment the provider contract addition, then activate the data definition.

   Why do you have to remove the provider contract addition for the projection view definition?

   With the redirected to parent association, the view becomes a child entity. For child entities, the provider contract is derived from the root entity.

   a) Adjust the code as follows:

```
define view entity Z##_C_Travelitem
  as projection on Z##_R_TravelItem
{
```

   b) Press **Ctrl + F3** to activate the development object.

6. Edit the data definition of your projection view for travels **Z##_C_Travel**. At the end of the projection list, add the name of the composition, which you defined in the underlying view entity.

   a) Adjust the code as follows:

```
  LocChangedAt,
  _TravelItem
}
```

7. Redirect the association to the projection view for flight travel items and classify it as composition.

   a) Adjust the code as follows:

```
  LocChangedAt,
  _TravelItem : redirected to composition child Z##_C_TravelItem
}
```

8. Activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

Task 3: Add Composition to the Behavior Projection

Edit the behavior definition on projection level **Z##_C_TRAVEL**. Add a behavior projection for the child entity. Expose its etag definition and add the associations that establish the composition.

1. Edit the behavior definition on projection level **Z##_C_TRAVEL**. At the end of the source code, use a code template to add a new define behavior statement for the child entity **Z##_C_TravelItem**.
   a) After the last closing bracket, add a new code row, enter **def**, and press **Ctrl + Space** to invoke code completion.
   b) From the suggestion list, choose *defineBehaviorFor - Define Behavior For*.
   c) Replace **entity** with **Z##_C_TravelItem**.
   d) Remove or comment the alias addition.
2. Add the etag definition to the behavior projection.
   a) Adjust the code as follows:

```
define behavior for Z##_C_TravelItem
use etag
{

}
```

3. For the child entity, add the standard operations update and delete to the projection.
   a) Adjust the code as follows:

```
define behavior for Z##_C_TravelItem
use etag
{
  use update;
  use delete;
}
```

4. Add the to parent association **_Travel** to the behavior projection of the child entity and the composition _TravelItem to the behavior projection of the root entity.

> Note:
> Do not forget to draft-enable both associations and create-enable the composition.

   a) Add the following code to the behavior definition for the root entity:

```
use association _TravelItem { create; with draft; }
```

   a) Adjust the behavior definition for the child entity as follows:

```
define behavior for Z##_C_TravelItem
use etag
{
  use update;
  use delete;

  use association _Travel { with draft; }
}
```

5. Activate the behavior projection.
   a) Press **Ctrl + F3** to activate the development object.

Task 4: Add Composition to the OData UI Service

Add the child entity of your business object to your service definition **Z##_TRAVEL** and extend the UI metadata of the service to display a list of child entities in a second facet on the object page for flight travels.

1. Open your service definition **Z##_TRAVEL** and add the projection view for flight travel items. Then activate the service definition.

   a) Adjust the code as follows:

```
@EndUserText.label: 'Flight Travel Service Definition'
define service Z##_TRAVEL {
  expose Z##_C_Travel;
  expose Z##_C_TravelItem;
}
```

   a) Press **Ctrl + F3** to activate the development object.

2. Open your service binding **Z##_UI_TRAVEL_O2** and analyze the information under *Entity Set and Association*.

   Note:
   There is no *Refresh* button available in service bindings but you can press **F5** to reload the service binding.

   What new information do you see?

   The child entity **Z##_C_TravelItem**, the associations between the root entity and the child entity and the view entities that are used for value helps.

3. Open the metadata extension for flight travels **Z##_C_TRAVEL** and locate the *@UI.facet* annotation. Inside the square brackets, add a comma (,) after the closing curly bracket and insert a copy of the existing facet definition.

   a) Adjust the code as follows:

```
@UI.facet: [ { id:              'Travel',
               purpose:         #STANDARD,
               type:            #IDENTIFICATION_REFERENCE,
               label:           'Travel',
               position:        10 },

             { id:              'Travel',
               purpose:         #STANDARD,
               type:            #IDENTIFICATION_REFERENCE,
               label:           'Travel',
               position:        10 }
           ]
```

4. Adjust the second facet definition according to the following table:

Property	Value
id	**'TravelItem'**
purpose	**#STANDARD**
type	**#LINEITEM_REFERENCE**
label	**'Travel Items'**
position	**20**

a) Adjust the code as follows:

```
@UI.facet: [ { id:             'Travel',
               purpose:        #STANDARD,
               type:           #IDENTIFICATION_REFERENCE,
               label:          'Travel',
               position:       10 },

             { id:             'TravelItem',
               purpose:        #STANDARD,
               type:           #LINEITEM_REFERENCE,
               label:          'Travel Items',
               position:       20 }
           ]
```

5. Inside the second facet definition, add the property **targetElement** with the name of the **composition** as value.

   a) Adjust the code as follows:

```
@UI.facet: [ { id:             'Travel',
               purpose:        #STANDARD,
               type:           #IDENTIFICATION_REFERENCE,
               label:          'Travel',
               position:       10 },

             { id:             'TravelItem',
               purpose:        #STANDARD,
               type:           #LINEITEM_REFERENCE,
               targetElement:  '_TravelItem',
               label:          'Travel Items',
               position:       20 }
           ]
```

6. Activate the metadata extension and retest the preview for the OData UI service. Make sure that the object page for flight travels displays an (empty) list of flight travel items and that you can create new flight travel items.

   a) Press `Ctrl + F3` to activate the development object.

   b) Open the service binding, place the cursor on the root entity and choose *Preview...*.

   c) In the preview, display the details for one of the travels.

   Result

   The object page consists of two facets. The first facet shows the flight travel details and the second facet a list of the related flight travel items. This list is empty.

   d) Choose *Edit* to change to edit mode.

   Result

   On the second facet, you now see a *Create* button above the list of flight travel items.

   e) Choose *Create* and *Apply* to create a new travel item with initial values in all input fields.

   Result

   The list of travel items now contains one entry with initial values.

   f) Choose *Save* to save your changes.

Unit 5
Exercise 18

Implement the Behavior of a Composite Business Object

In this exercise, you implement the behavior for the composite business object. You define and implement a validation for the child entity and a determination.

Note:
In this exercise, replace ## with your group number.

Table 22: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437E_R_TRAVEL*
ABAP Class	*/LRN/BP_437E_R_TRAVEL*

Task 1: Validate the Flight Date

Extend the behavior definition of your business object. For the child entity **`Z##_R_TravelItem`**, define a validation for the flight date (suggested name: **`validateFlightDate`**) that is always performed during the **`create`** operation, but during the **`update`** operation only if the user changed the value of the **`FlightDate`** field. Add the validation to the draft determine action **`Prepare`** and generate the implementation method. If the flight date is initial or if it lies in the past, add the child entity instance to the **`failed`** parameter and the **`reported`** parameter. Use suitable texts from the **`/LRN/CM_S4D437`** class as message texts.

Hint:
If you implemented the **`validateBeginDate`** validation, you can copy that implementation as a starting point. Just make sure you read the data of the child entity and not from the root entity and that you fill the **`item`** component of the response parameters and not the **`travel`** component.

1. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. In the behavior definition for the child entity **`Z##_R_TravelItem`**, add a validation (suggested name: **`validateFlightDate`**) that is triggered by the **`create`** operation or changes to the **`FlightDate`** field.

2. Add the new validation to the list of validations that are to be performed during the draft determine action **`Prepare`** of the root entity.

Hint:
Address the validation in the following form: **<child entity alias>~<validation name>**.

3. Activate the behavior definition and use a quick fix to generate the implementation method for the validation in the local handler class for flight travel items.
4. Edit the validation implementation method. Read the flight date of all affected flight travel items. Store the result in an inline declared internal table (suggested name: **items**). Then loop over the flight travel items and assign an inline declared field symbol (suggested name: **<item>**).
5. At the beginning of the loop, before the actual check, add a new row to the **item** component of the **reported** parameter. In this row, fill **%tky** with the key of the affected instance and **%state_area** with a non-initial value (suggested value: **FLIGHTDATE**) .

Hint:
As before, we recommend that you define a constant for the **%state_area** value (suggested name: **c_area**).

6. After the above implemented APPEND statement, add an IF block to the body of the loop: If the **FlightDate** field contains the initial value, add the key of the current travel item instance to the **item** component of parameter **failed**. In the same IF block, add a new row to the **item** component of the **reported** parameter. In this row, fill **%tky** with the key of the affected instance, **%msg** with an instance of the **/LRN/CM_S4D437** class, using a suitable constant for the **textid** parameter, **%state_area** with the value of the **c_area** constant and **%element** with a flag to link the error message to the **FLightDate** field.
7. Add an ELSEIF branch that is executed, if the flight date lies before the system date. Copy the code from the IF branch to the ELSEIF branch and adjust the error message text.
8. Activate the ABAP class and retest the preview for the OData UI service.

   Do you see your error message if you enter no flight date or a flight date in the past?

   ___

   ___

   ___

   What is missing?

   ___

   ___

   ___

9. Extend the READ ENTITIES statement, so that it also reads the **AgencyId** and **TravelId** fields.
10. In the implementation of the flight date validation, adjust the statements where you add messages to the **reported** parameter and fill the **%path** component.

Caution:
Make sure you do not only fill the **AgencyId** and **TravelId** components of **%path-travel**, but also the **%is_draft** component!

Hint:
You can use a CORRESPONDING expression to transfer all three fields from **<item>** to **%path-travel**.

11. Activate the ABAP class and retest the preview for the OData UI service.

Task 2: Read Data by Association

Extend the behavior definition of your business object. For the child entity **Z##_R_TravelItem**, define a determination to adjust the starting date and end date of a flight travel (suggested name: **determineTravelDates**) that is performed if the user changed the value of the **FlightDate** field. If the flight date is before the starting date set the starting date to that flight date. If the flight date is after the end date, set the end date to the flight date.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL** and add a determination for the child entity (suggested name: **determineTravelDates**) that is triggered by changes to the **FlightDate** field.
2. Activate the behavior definition and use a quick fix to generate the implementation method for the determination in the local handler class for flight travel items.
3. In the handler method for the determination, implement a READ ENTITIES statement to read the **FlightDate** field for all affected flight travel items. Store the result in an inline declared internal table (suggested name: **items**).
4. In the same READ ENTITIES statement, add the BY keyword followed by a backslash and the name of the to parent association.
5. Read the **BeginDate** and **EndDate** fields and store the result in an inline declared internal table (suggested name: **travels**) and the link information in another inline declared data object (suggested name: **link**).
6. Implement a loop over the flight travel items, assigning an inline declared field symbol (suggested name: **<item>**). At the beginning of the loop, assign a field symbol to the entry of **travels**, that belongs to the current flight travel item (suggested name for the field symbol: **<travel>**).

Hint:
Use the content of the **link** table to identify the related data. The **link** table contains pairs of **source** and **target** structures, where **source** contains the **%tky** key of a flight travel item and **target** contains the **%tky** key of the related flight travel.

7. Inside the loop, compare the flight date of the travel item to the end date of the travel. If the flight date is after the end date, use the field symbol to adjust the end date of the flight travel.

8. If the flight date is not in the past and lies before the starting date, use the field symbol to adjust the starting date of the flight travel.

9. After the LOOP, implement an EML statement to update the **`BeginDate`** and **`EndDate`** values of the flight travels.

10. Activate the ABAP class and retest the preview for the OData UI service.

Unit 5
Solution 18

Implement the Behavior of a Composite Business Object

In this exercise, you implement the behavior for the composite business object. You define and implement a validation for the child entity and a determination.

Note:
In this exercise, replace ## with your group number.

Table 22: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437E_R_TRAVEL*
ABAP Class	*/LRN/BP_437E_R_TRAVEL*

Task 1: Validate the Flight Date

Extend the behavior definition of your business object. For the child entity **Z##_R_TravelItem**, define a validation for the flight date (suggested name: **validateFlightDate**) that is always performed during the **create** operation, but during the **update** operation only if the user changed the value of the **FlightDate** field. Add the validation to the draft determine action **Prepare** and generate the implementation method. If the flight date is initial or if it lies in the past, add the child entity instance to the **failed** parameter and the **reported** parameter. Use suitable texts from the **/LRN/CM_S4D437** class as message texts.

Hint:
If you implemented the **validateBeginDate** validation, you can copy that implementation as a starting point. Just make sure you read the data of the child entity and not from the root entity and that you fill the **item** component of the response parameters and not the **travel** component.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL**. In the behavior definition for the child entity **Z##_R_TravelItem**, add a validation (suggested name: **validateFlightDate**) that is triggered by the **create** operation or changes to the **FlightDate** field.

   a) Adjust the code as follows:

```
field ( readonly )
AgencyId,
TravelId;

validation validateFlightDate on save
{ create;
  field FlightDate;
}
```

2. Add the new validation to the list of validations that are to be performed during the draft determine action **`Prepare`** of the root entity.

   Hint:
   Address the validation in the following form: **`<child entity alias>~<validation name>`**.

   a) Adjust the code as follows:

```
draft determine action Prepare
{
  validation validateDescription;
  validation validateCustomer;
  validation validateBeginDate;
  validation validateEndDate;
  validation validateDateSequence;
  validation Item~validateFlightDate;
}
```

Note:
If you did not implement the validations for the date fields, the code looks like this:

```
draft determine action Prepare
{
  validation validateDescription;
  validation validateCustomer;
  validation Item~validateFlightDate;
}
```

3. Activate the behavior definition and use a quick fix to generate the implementation method for the validation in the local handler class for flight travel items.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) In the validation definition statement, place the cursor on the name of the validation and press **`Ctrl + 1`**.

   c) From the list of available quick fixes, choose *Add method for validation validateflightdate of entity z##_r_travelitem in new local class*.

4. Edit the validation implementation method. Read the flight date of all affected flight travel items. Store the result in an inline declared internal table (suggested name: **`items`**). Then loop over the flight travel items and assign an inline declared field symbol (suggested name: **`<item>`**).

   a) Between `METHOD validateFlightDate.` and `ENDMETHOD.`, add the following code:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Item
```

```
    FIELDS ( FlightDate )
    WITH CORRESPONDING #(  keys )
    RESULT DATA(items).

LOOP AT items ASSIGNING FIELD-SYMBOL(<item>).

ENDLOOP.
```

5. At the beginning of the loop, before the actual check, add a new row to the **item** component of the **reported** parameter. In this row, fill **%tky** with the key of the affected instance and **%state_area** with a non-initial value (suggested value: **FLIGHTDATE**) .

> Hint:
> As before, we recommend that you define a constant for the **%state_area** value (suggested name: **c_area**).

a) At the beginning of the method, add the following code:

```
CONSTANTS c_area TYPE string VALUE `FLIGHTDATE`.
```

b) Inside the loop, add the following code:

```
APPEND VALUE #( %tky        = <item>-%tky
                %state_area = c_area )
  TO reported-item.
```

6. After the above implemented APPEND statement, add an IF block to the body of the loop: If the **FlightDate** field contains the initial value, add the key of the current travel item instance to the **item** component of parameter **failed**. In the same IF block, add a new row to the **item** component of the **reported** parameter. In this row, fill **%tky** with the key of the affected instance, **%msg** with an instance of the **/LRN/CM_S4D437** class, using a suitable constant for the **textid** parameter, **%state_area** with the value of the **c_area** constant and **%element** with a flag to link the error message to the **FLightDate** field.

a) Inside the loop, add the following code:

```
IF <item>-FlightDate IS INITIAL.

  APPEND VALUE #( %tky = <item>-%tky )
    TO failed-item.

  APPEND VALUE #( %tky                = <item>-%tky
                  %msg                = NEW /lrn/cm_s4d437(
                        /lrn/cm_s4d437=>field_empty )
                  %element-FlightDate = if_abap_behv=>mk-on
                  %state_area         = c_area )
    TO reported-item.

ENDIF.
```

7. Add an ELSEIF branch that is executed, if the flight date lies before the system date. Copy the code from the IF branch to the ELSEIF branch and adjust the error message text.

a) Before `ENDIF.`, add the following code:

```
ELSEIF <item>-FlightDate < cl_abap_context_info=>get_system_date(  ).

  APPEND VALUE #(  %tky = <item>-%tky )
    TO failed-item.

  APPEND VALUE #( %tky                  = <item>-%tky
                    %msg                = NEW /lrn/cm_s4d437(
                           /lrn/cm_s4d437=>flight_date_past )
                    %element-FlightDate = if_abap_behv=>mk-on
                    %state_area         = c_area )
    TO reported-item.
```

8. Activate the ABAP class and retest the preview for the OData UI service.

   Do you see your error message if you enter no flight date or a flight date in the past?

   No, there is only an error message with the generic text *Resolve data inconsistencies to save changes*.

   What is missing?

   You have to map the child entity instance for which you report the error message to its parent entity instance.

   a) Press **Ctrl + F3** to activate the development object.

   b) Open the service binding, place the cursor on the root entity and choose *Preview...*.

   c) In the preview, display the details for one of the travels, change to edit mode, and choose *Create* to create a new flight travel item.

   d) Leave all input fields empty, especially the *Flight Date* field, and choose *Apply*.

   e) Back on the object page of the travel, choose *Save* to trigger the execution of the validations.

      Result

      You see the (transition) message with the generic error text.

9. Extend the READ ENTITIES statement, so that it also reads the **AgencyId** and **TravelId** fields.

   a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Item
    FIELDS ( AgencyId TravelId FlightDate )
    WITH CORRESPONDING #(  keys )
    RESULT DATA(items).
```

10. In the implementation of the flight date validation, adjust the statements where you add messages to the **reported** parameter and fill the **%path** component.

> Caution:
> Make sure you do not only fill the **AgencyId** and **TravelId** components of **%path-travel**, but also the **%is_draft** component!

Hint:
You can use a CORRESPONDING expression to transfer all three fields from **<item>** to **%path-travel**.

a) Adjust the APPEND statement in the IF branch as follows:

```
APPEND VALUE #( %tky                = <item>-%tky
                %msg                = NEW /lrn/cm_s4d437(
                            /lrn/cm_s4d437=>field_empty )
                %element-FlightDate = if_abap_behv=>mk-on
                %state_area         = c_area
                %path-travel = CORRESPONDING #( <item> ) )
  TO reported-item.
```

b) Adjust the APPEND statement in the ELSEIF branch as follows:

```
APPEND VALUE #( %tky                = <item>-%tky
                %msg                = NEW /lrn/cm_s4d437(
                       /lrn/cm_s4d437=>flight_date_past )
                %element-FlightDate = if_abap_behv=>mk-on
                %state_area         = c_area
                %path-travel        = CORRESPONDING #( <item> ) )
  TO reported-item.
```

11. Activate the ABAP class and retest the preview for the OData UI service.

    a) Press **Ctrl + F3** to activate the development object.

    b) Open the service binding, place the cursor on the root entity and choose *Preview...*.

    c) In the preview, display the details for one of the travels, change to edit mode, and choose *Create* to create a new flight travel item.

    d) Leave all input fields empty, especially the *Flight Date* field, and choose *Apply*.

    e) Back on the object page of the travel, choose *Save* to trigger the execution of the validations.

    Result

    You see the (state) message with the link to the *Flight Date* input field.

Task 2: Read Data by Association

Extend the behavior definition of your business object. For the child entity **Z##_R_TravelItem**, define a determination to adjust the starting date and end date of a flight travel (suggested name: **determineTravelDates**) that is performed if the user changed the value of the **FlightDate** field. If the flight date is before the starting date set the starting date to that flight date. If the flight date is after the end date, set the end date to the flight date.

1. Edit the behavior definition on data model level **Z##_R_TRAVEL** and add a determination for the child entity (suggested name: **determineTravelDates**) that is triggered by changes to the **FlightDate** field.

    a) Adjust the code as follows:

```
validation validateFlightDate on save
{ create;
  field FlightDate;
}
```

```
determination determineTravelDates on save
{ field FlightDate;
}
```

2. Activate the behavior definition and use a quick fix to generate the implementation method for the determination in the local handler class for flight travel items.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) Place the cursor on the name of the determination and press **`Ctrl + 1`**.

   c) From the list of available quick fixes, choose *Add method for determination determinetraveldates of entity z##_r_travelitem in local class lhc_item*.

3. In the handler method for the determination, implement a READ ENTITIES statement to read the **`FlightDate`** field for all affected flight travel items. Store the result in an inline declared internal table (suggested name: **`items`**).

   a) Between `METHOD determineTravelDates.` and `ENDMETHOD.`, add the following code:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Item
    FIELDS ( FlightDate )
    WITH CORRESPONDING #( keys )
    RESULT DATA(items).
```

4. In the same READ ENTITIES statement, add the BY keyword followed by a backslash and the name of the to parent association.

   a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Item
    FIELDS ( FlightDate )
    WITH CORRESPONDING #( keys )
    RESULT DATA(items)

    BY \_Travel
    .
```

5. Read the **`BeginDate`** and **`EndDate`** fields and store the result in an inline declared internal table (suggested name: **`travels`**) and the link information in another inline declared data object (suggested name: **`link`**).

   a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Item
    FIELDS ( FlightDate )
    WITH CORRESPONDING #( keys )
    RESULT DATA(items).

    BY \_Travel
    FIELDS ( BeginDate EndDate )
    WITH CORRESPONDING #( keys )
    RESULT DATA(travels)
    LINK DATA(link).
```

6. Implement a loop over the flight travel items, assigning an inline declared field symbol (suggested name: **`<item>`**). At the beginning of the loop, assign a field symbol to the entry of **`travels`**, that belongs to the current flight travel item (suggested name for the field symbol: **`<travel>`**).

Hint:
Use the content of the **link** table to identify the related data. The **link** table contains pairs of **source** and **target** structures, where **source** contains the **%tky** key of a flight travel item and **target** contains the **%tky** key of the related flight travel.

a) After the READ ENTITIES statement, add the following code:

```
LOOP AT items ASSIGNING FIELD-SYMBOL(<item>).

  ASSIGN travels[ %tky =
    link[ source-%tky = <item>-%tky ]-target-%tky ]
      TO FIELD-SYMBOL(<travel>).

ENDLOOP.
```

b) If you prefer a READ TABLE statement, the code inside the loop could look as follows:

```
READ TABLE travels ASSIGNING FIELD-SYMBOL(<travel>)
  WITH KEY %tky = link[ source-%tky = <item>-%tky ]-target-%tky.
```

c) If you are familiar with secondary keys and want to get rid of the syntax warnings, you can reference the secondary keys as follows:

```
ASSIGN travels[ KEY id %tky =
  link[ KEY id source-%tky = <item>-%tky ]-target-%tky ]
    TO FIELD-SYMBOL(<travel>).
```

7. Inside the loop, compare the flight date of the travel item to the end date of the travel. If the flight date is after the end date, use the field symbol to adjust the end date of the flight travel.

   a) Before the ENDLOOP statement, add the following code:

```
IF <travel>-EndDate < <item>-FlightDate.
  <travel>-EndDate = <item>-FlightDate.
ENDIF.
```

8. If the flight date is not in the past and lies before the starting date, use the field symbol to adjust the starting date of the flight travel.

   a) Before the ENDLOOP statement, add the following code:

```
IF <item>-FlightDate > cl_abap_context_info=>get_system_date( )
  AND <item>-FlightDate  < <travel>-BeginDate.
    <travel>-BeginDate = <item>-FlightDate.
ENDIF.
```

9. After the LOOP, implement an EML statement to update the **BeginDate** and **EndDate** values of the flight travels.

   a) Between `ENDLOOP.` and `ENDMETHOD.`, add the following code:

```
MODIFY ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Travel
    UPDATE
    FIELDS ( BeginDate EndDate )
    WITH CORRESPONDING #( travels ).
```

10. Activate the ABAP class and retest the preview for the OData UI service.

    a) Press **Ctrl + F3** to activate the development object.

    b) Open the service binding, place the cursor on the root entity and choose *Preview...*.

c) In the preview, display the details for one of the travels, change to edit mode, and choose *Create* to create a new flight travel item.

d) Enter a value in the *Flight Date* field that lies after the end date of the travel.

e) Back on the object page of the travel, choose *Save* to trigger the execution of the validations and determinations.

Result

You see the adjusted value in the *End Date* field.

Unit 6
Exercise 19

Implement Unmanaged Save

In this exercise, you reuse existing business logic for flight travel items in your business object. The business object as such remains managed. You only enable unmanaged save for the travel items and replace the managed access to your persistent table with calls to the existing code.

Note:
In this exercise, replace ## with your group number.

Table 23: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437F_R_TRAVEL*
ABAP Class	*/LRN/BP_437F_R_TRAVEL*

Task 1: Analyze Existing Code
Analyze existing ABAP class **`/LRN/CL_S4D437_TRITEM`**.

1. Open the source code of ABAP class **`/LRN/CL_S4D437_TRITEM`**.
2. Analyze the public section of the definition part.

   Does the class have an instance constructor? What does it require as input?

   ____
   ____
   ____

   What other public methods does the class have?

   ____
   ____
   ____

   What input do these methods expect?

   ____
   ____
   ____

Task 2: Enable Data Access Implementation
Adjust the behavior definition on data model level **`Z##_R_TRAVEL`**. For the child entity **`Z##_R_TravelItem`**, specify that you want to implement extra coding to be performed

during the save sequence. Then make sure that the framework does no longer take care of write access to the persistent table **`Z##_TRITEM`**.

1. Edit your behavior definition on data model level **`Z##_R_TRAVEL`** and locate the `define behavior` statement for the child entity **`Z##_R_TravelItem`**.

2. Specify that you want to implement additional coding to be performed during the save sequence.

3. Perform a syntax check.

   Are there any new syntax errors or warnings?

4. Do not invoke the related quick fix yet. Instead, comment or remove the previous addition and replace it with the addition that specifies that you want to take care of the persistence of flight travels yourself.

5. Perform another syntax check.

   Are there any new syntax errors or warnings?

6. Remove or comment the `persistent table` addition.

7. Activate the behavior definition. Then, use a quick fix to generate the required method in the save sequence.

   Why is the name of the local class **`lsc_z##_r_travel`** and not **`lsc_travel`** or **`lsc_travelitem`**?

8. Analyze the signature of the generated method.

   What importing parameters does the method have?

   What are the data types of the importing parameters?

Task 3: Implement the Data Access

Implement the **`save_modified`** method of the saver class. For all three importing parameters in turn, implement a loop over the **`item`** component and call the related method of the **`/LRN/CL_S4D437_TRITEM`** class. To facilitate data mapping between element names in the business object and field names in the signature of the **`/LRN/CL_S4D437_TRITEM`** class, define a field mapping for the structure types and use it in CORRESPONDING expressions.

1. In the following, you will create an instance of class **`/LRN/CL_S4D437_TRITEM`** that encapsulates access to your database table **`Z##_TRITEM`**. However, class **`/LRN/CL_S4D437_TRITEM`** and database table **`Z##_TRITEM`** belong to different software components, and an object of one software component cannot be used in another software component by default, as software components provide their functionality for other software components via explicitly released APIs.

   Therefore, you must first C1 release your database table **`Z##_TRITEM`** so that class **`/LRN/CL_S4D437_TRITEM`** can access it.

2. In the **`save_modified`** method of the saver class, now create an instance of the **`/LRN/CL_S4D437_TRITEM`** class and store a reference to the new instance in an inline declared variable (suggested name: **`model`**). Supply the importing parameter of the constructor with the name of your database table for travel items **`Z##_TRITEM`**.

3. Implement a loop over the **`item`** component of the importing parameter **`delete`**, using an inline declared field symbol (suggested name: **`<item_d>`**).

4. For each row, call the **`delete_item`** method of your **`/LRN/CL_S4D437_TRITEM`** instance. Supply its importing parameter with the uuid of the current row of **`delete-item`**.

5. Similarly, implement a loop over the **`item`** component of the importing parameter **`create`**, using an inline declared field symbol (suggested name: **`<item_c>`**).

6. For each row, call the **`create_item`** method of your **`/LRN/CL_S4D437_TRITEM`** instance. Supply its importing parameter with the data from the current row of **`create-item`**. Use a CORRESPONDING expression to convert the row of **`create-item`** to the type of input parameter **`i_item`**, that is, to structure type **`/LRN/437_S_TRITEM`**.

   Note:
   At the moment, the CORRESPONDING expression will only assign the components with identical names in **`/LRN/437_S_TRITEM`** and in the row type of **`create-item`**. That is not the case for most of the components. We use an explicit field mapping in the next step.

7. Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. Define a field mapping for structure type **`/LRN/437_S_TRITEM`**.

   Hint:
   Structure type **`/LRN/437_S_TRITEM`** and persistent table **`Z##_TRITEM`** use the same field names. To save time, you can copy from the field mapping for persistent table **`Z##_TRITEM`**.

8. Return to the implementation of the **save_modified** method. Adjust the CORRESPONDING expression so that it uses the field mapping for structure type **/LRN/437_S_TRITEM** from the behavior definition.

9. Finally, implement a loop over the **item** component of the importing parameter **update**, using an inline declared field symbol (suggested name: **<item_u>**). For each row, call the **update_item** method of your **/LRN/CL_S4D437_TRITEM** instance.

10. Supply the importing parameter **i_item** with the data from the current row of **update-item**. Use the field mapping from the behavior definition as before. Supply importing parameter **i_itemx** in the same way. But this time, use the addition USING CONTROL to make sure the information is extracted from sub component **%control**.

11. Return to the behavior definition and extend the field mapping for structure type **/LRN/437_S_TRITEM**. Specify that the structure type **/LRN/437_S_TRITEMX** is the related control structure, then activate the behavior definition and the behavior implementation.

12. Test the preview for the OData UI service. Make sure that you can successfully create, update and delete flight travel items.

Task 4: Optional: Handle the Messages

Handle the returning parameters of the **delete_item**, **create_item**, and **update_item** methods of class **/LRN/CL_S4D437_TRITEM**. Store the returned messages in the **reported** parameter of the **save_modified** method.

Note:
Because all returning parameters have the same type, it is a good idea to do the required mapping in a private helper method of the saver class.

1. In the saver class, define a new private instance method (suggested name: **map_message**). Define an importing parameter of structure type **SYMSG** (suggested name: **i_msg**) and a returning parameter of type **REF TO if_abap_behv_message** (suggested name: **r_msg**).

2. Use a quick fix to generate the implementation for the method.

3. Implement the method. First, map the message types in component **msgty** of parameter **i_msg** to a local variable of type **if_abap_behv_message=>t_severity** (suggested name: **severity**).

Hint:
As values for the local variable, use the components of the constant structure **if_abap_behv_message=>severity**.

You can either use a SWITCH expression or a more old-fashioned CASE ... ENDCASE structure.

4. Call the inherited functional method **new_message** and assign the result to the returning parameter **r_msg**. Supply the importing parameters with the related components of parameter **i_msg**, except for parameter **severity** which you supply with your local variable **severity**.

5. Return to the implementation of the **`save_modified`** method. Find the call of the **`delete_item`** method and store the result in an inline declared variable (suggested name: **`msg_d`**).

6. If the result is not initial, append a new row to the **`item`** component of the **`reported`** parameter. Fill the **`%tky-itemuuid`** component with the uuid of the current flight travel item and component **`%msg`** via a call of the private method that you just implemented.

7. Adjust the call of **`create_item`** accordingly.

8. Finally, adjust the call of **`update_item`**.

9. Activate the ABAP class.

10. Retest the preview for the OData UI service. Ensure that the messages returned by the **`delete_item`**, **`create_item`**, and **`update_item`** methods of class **`/LRN/CL_S4D437_TRITEM`** are displayed on the UI.

Unit 6
Solution 19

Implement Unmanaged Save

In this exercise, you reuse existing business logic for flight travel items in your business object. The business object as such remains managed. You only enable unmanaged save for the travel items and replace the managed access to your persistent table with calls to the existing code.

Note:
In this exercise, replace ## with your group number.

Table 23: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437F_R_TRAVEL*
ABAP Class	*/LRN/BP_437F_R_TRAVEL*

Task 1: Analyze Existing Code
Analyze existing ABAP class **/LRN/CL_S4D437_TRITEM**.

1. Open the source code of ABAP class **/LRN/CL_S4D437_TRITEM**.
   a) Press **Ctrl + Shift + A**.
   b) Enter **/LRN/CL_S4D437_** as the search string.
   c) Select */LRN/CL_S4D437_TRITEM (Class)* from the list and choose *OK*.
2. Analyze the public section of the definition part.
   a) Analyze the code between `PUBLIC SECTION.` and `PROTECTED SECTION.`

   Does the class have an instance constructor? What does it require as input?

   The class has an instance constructor that requires a database table name as input.

   What other public methods does the class have?

   The class has the instance methods **delete_item**, **create_item**, and **update_item**.

What input do these methods expect?

The **delete_item** method requires the uuid key of a DB table entry to be deleted. The **create_item** method requires a data set to be inserted into the DB table (based on structure type **/LRN/437_S_ITEM**). The **update_item** method requires the same input as **create_item** but in addition, it requires a structure with flags that specifies the fields that are to be updated.

Task 2: Enable Data Access Implementation

Adjust the behavior definition on data model level **Z##_R_TRAVEL**. For the child entity **Z##_R_TravelItem**, specify that you want to implement extra coding to be performed during the save sequence. Then make sure that the framework does no longer take care of write access to the persistent table **Z##_TRITEM**.

1. Edit your behavior definition on data model level **Z##_R_TRAVEL** and locate the `define behavior` statement for the child entity **Z##_R_TravelItem**.
   a) For example, you can press **Ctrl + F** to invoke a search dialog where you search for **R_TravelItem**.
2. Specify that you want to implement additional coding to be performed during the save sequence.
   a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
persistent table z##_tritem
with additional save
draft table z##_tritem_d
```

3. Perform a syntax check.

   Are there any new syntax errors or warnings?

   There are no new syntax errors. But there is a warning *Implementation missing for SAVE_MODIFIED in save sequence*.

   a) Press **Ctrl + F2**.
4. Do not invoke the related quick fix yet. Instead, comment or remove the previous addition and replace it with the addition that specifies that you want to take care of the persistence of flight travels yourself.
   a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
persistent table z##_tritem
with unmanaged save
draft table z##_tritem_d
```

5. Perform another syntax check.

Are there any new syntax errors or warnings?

Yes, there is a new syntax error *It is not possible to specify both "unmanaged save" and "persistent table" at the same time.*

a) Press **Ctrl + F2**.

6. Remove or comment the `persistent table` addition.

   a) Adjust the code as follows:

```
define behavior for Z##_R_TravelItem alias Item
with unmanaged save
draft table z##_tritem_d
```

7. Activate the behavior definition. Then, use a quick fix to generate the required method in the save sequence.

   Why is the name of the local class **lsc_z##_r_travel** and not **lsc_travel** or **lsc_travelitem**?

   The local saver class is always meant for the entire business object, not for an individual entity. Therefore, it uses the name of the business object. It must not use **travel**, because this is the alias name of the root entity, not an alias name for the business object.

   a) Press **Ctrl + F3** to activate the development object.

   b) Place the cursor in the code row that starts with `with unmanaged` and press **Ctrl + 1**. Alternatively, choose the warning icon with a light bulb that is displayed on the left-hand side of this code row.

   c) From the list of available quick fixes, choose *Add required method save_modified in new local saver class*.

      Result

      The quick fix adds a local class **lsc_z##_r_travel** to your behavior pool class **ZBP_##_R_TRAVEL**. The local class inherits from global class **CL_ABAP_BEHAVIOR_SAVER** and redefines the inherited instance method **save_modified**.

8. Analyze the signature of the generated method.

   What importing parameters does the method have?

   The method has three importing parameters **create**, **update**, and **delete**.

What are the data types of the importing parameters?

The importing parameters are typed with the derived types **request for change** or **request for delete**. These types are deep structures, each with a table-like component **item**.

a) For example, place the cursor on `save_modified` and press **F2** to display the *ABAP Element Info* for the method.

Task 3: Implement the Data Access

Implement the **save_modified** method of the saver class. For all three importing parameters in turn, implement a loop over the **item** component and call the related method of the **/LRN/CL_S4D437_TRITEM** class. To facilitate data mapping between element names in the business object and field names in the signature of the **/LRN/CL_S4D437_TRITEM** class, define a field mapping for the structure types and use it in CORRESPONDING expressions.

1. In the following, you will create an instance of class **/LRN/CL_S4D437_TRITEM** that encapsulates access to your database table **Z##_TRITEM**. However, class **/LRN/CL_S4D437_TRITEM** and database table **Z##_TRITEM** belong to different software components, and an object of one software component cannot be used in another software component by default, as software components provide their functionality for other software components via explicitly released APIs.

   Therefore, you must first C1 release your database table **Z##_TRITEM** so that class **/LRN/CL_S4D437_TRITEM** can access it.

   a) In the *Project Explorer*, open the context menu for database table **Z##_TRITEM**.

   b) Choose *API State → Add Use System-Internally (Contract C1)...*.

   c) On the *Add Release Contract* dialog, choose *Next*.

   d) In the *Validate Changes* step, choose *Next*.

   e) Select a transport request and choose *Finish*.

2. In the **save_modified** method of the saver class, now create an instance of the **/LRN/CL_S4D437_TRITEM** class and store a reference to the new instance in an inline declared variable (suggested name: **model**). Supply the importing parameter of the constructor with the name of your database table for travel items **Z##_TRITEM**.

   a) Adjust the code as follows:

```
METHOD save_modified.

  DATA(model) = NEW /lrn/cl_s4d437_tritem(
                   i_table_name = 'Z##_TRITEM' ).

ENDMETHOD.
```

3. Implement a loop over the **item** component of the importing parameter **delete**, using an inline declared field symbol (suggested name: **<item_d>**).

a) Adjust the code as follows:

```
METHOD save_modified.

  DATA(model) = NEW /lrn/cl_s4d437_tritem(
                      i_table_name = 'Z##_TRITEM' ).

  LOOP AT delete-item ASSIGNING FIELD-SYMBOL(<item_d>).

  ENDLOOP.

ENDMETHOD.
```

4. For each row, call the **delete_item** method of your **/LRN/CL_S4D437_TRITEM** instance. Supply its importing parameter with the uuid of the current row of **delete-item**.

   a) Adjust the code as follows:

```
METHOD save_modified.

  DATA(model) = NEW /lrn/cl_s4d437_tritem(
                      i_table_name = 'Z##_TRITEM' ).

  LOOP AT delete-item ASSIGNING FIELD-SYMBOL(<item_d>).
    model->delete_item( i_uuid = <item_d>-itemuuid ).
  ENDLOOP.

ENDMETHOD.
```

5. Similarly, implement a loop over the **item** component of the importing parameter **create**, using an inline declared field symbol (suggested name: **<item_c>**).

   a) At the end of the method, add the following code:

```
LOOP AT create-item ASSIGNING FIELD-SYMBOL(<item_c>).

ENDLOOP.
```

6. For each row, call the **create_item** method of your **/LRN/CL_S4D437_TRITEM** instance. Supply its importing parameter with the data from the current row of **create-item**. Use a CORRESPONDING expression to convert the row of **create-item** to the type of input parameter **i_item**, that is, to structure type **/LRN/437_S_TRITEM**.

Note:
At the moment, the CORRESPONDING expression will only assign the components with identical names in **/LRN/437_S_TRITEM** and in the row type of **create-item**. That is not the case for most of the components. We use an explicit field mapping in the next step.

   a) Adjust the code as follows:

```
LOOP AT create-item ASSIGNING FIELD-SYMBOL(<item_c>).
  model->create_item(
           i_item = CORRESPONDING #( <item_c> ) ).
ENDLOOP.
```

7. Edit the behavior definition on data model level **Z##_R_TRAVEL**. Define a field mapping for structure type **/LRN/437_S_TRITEM**.

Hint:
Structure type **/LRN/437_S_TRITEM** and persistent table **Z##_TRITEM** use the same field names. To save time, you can copy from the field mapping for persistent table **Z##_TRITEM**.

a) Add the following code to the behavior definition for the child entity **Z##_R_TravelItem**:

```
mapping for /lrn/437_s_tritem corresponding
  {
    ItemUuid           = item_uuid;
    AgencyId           = agency_id;
    TravelId           = travel_id;
    CarrierId          = carrier_id;
    ConnectionId       = connection_id;
    FlightDate         = flight_date;
    BookingId          = booking_id;
    PassengerFirstName = passenger_first_name;
    PassengerLastName  = passenger_last_name;
    ChangedAt          = changed_at;
    ChangedBy          = changed_by;
    LocChangedAt       = loc_changed_at;
  }
```

8. Return to the implementation of the **save_modified** method. Adjust the CORRESPONDING expression so that it uses the field mapping for structure type **/LRN/437_S_TRITEM** from the behavior definition.

   a) Adjust the code as follows:

```
model->create_item(
  i_item = CORRESPONDING #( <item_c> MAPPING FROM ENTITY ) ).
```

9. Finally, implement a loop over the **item** component of the importing parameter **update**, using an inline declared field symbol (suggested name: **<item_u>**). For each row, call the **update_item** method of your **/LRN/CL_S4D437_TRITEM** instance.

   a) At the end of the method, add the following code:

```
LOOP AT update-item ASSIGNING FIELD-SYMBOL(<item_u>).
  model->update_item(
           i_item  =
           i_itemx =  ).
ENDLOOP.
```

10. Supply the importing parameter **i_item** with the data from the current row of **update-item**. Use the field mapping from the behavior definition as before. Supply importing parameter **i_itemx** in the same way. But this time, use the addition USING CONTROL to make sure the information is extracted from sub component **%control**.

    a) Adjust the code as follows:

```
LOOP AT update-item ASSIGNING FIELD-SYMBOL(<item_u>).
  model->update_item(
           i_item  = CORRESPONDING #( <item_u> MAPPING FROM ENTITY )
           i_itemx = CORRESPONDING #( <item_u> MAPPING FROM ENTITY
                                               USING CONTROL ) ).
ENDLOOP.
```

11. Return to the behavior definition and extend the field mapping for structure type **/LRN/437_S_TRITEM**. Specify that the structure type **/LRN/437_S_TRITEMX** is the related control structure, then activate the behavior definition and the behavior implementation.

a) Adjust the code as follows:

```
mapping for /lrn/437_s_tritem
  control /lrn/437_s_tritemx corresponding
  {
```

b) In the behavior definition, press **Ctrl + F3** to activate the development object.

c) In the behavior implementation, press **Ctrl + F3** to activate the development object.

12. Test the preview for the OData UI service. Make sure that you can successfully create, update and delete flight travel items.

a) Open the service binding, place the cursor on the root entity and choose *Preview...*.

b) In the preview, display the details for one of the travels.

c) Choose *Edit* to change to edit mode.

d) Ensure that you can successfully create new travel items via the UI. Also make sure that you can update and delete existing travel items.

e) Use the preview tool in Eclipse to check whether the contents of database table **Z##_TRITEM** have been updated accordingly.

Task 4: Optional: Handle the Messages

Handle the returning parameters of the **delete_item**, **create_item**, and **update_item** methods of class **/LRN/CL_S4D437_TRITEM**. Store the returned messages in the **reported** parameter of the **save_modified** method.

Note:
Because all returning parameters have the same type, it is a good idea to do the required mapping in a private helper method of the saver class.

1. In the saver class, define a new private instance method (suggested name: **map_message**). Define an importing parameter of structure type **SYMSG** (suggested name: **i_msg**) and a returning parameter of type **REF TO if_abap_behv_message** (suggested name: **r_msg**).

a) Adjust the code as follows:

```
CLASS lsc_z##_r_travel DEFINITION INHERITING FROM
cl_abap_behavior_saver.

  PROTECTED SECTION.
    METHODS save_modified REDEFINITION.

  PRIVATE SECTION.
    METHODS
      map_message
        IMPORTING
          i_msg        TYPE symsg
        RETURNING
          VALUE(r_msg) TYPE REF TO if_abap_behv_message.

ENDCLASS.
```

2. Use a quick fix to generate the implementation for the method.

a) Choose the error icon with a light bulb next to the code row with the method name to display the quick assist proposals.

b) From the list of available quick fixes, choose *Add implementation for map_message*.

3. Implement the method. First, map the message types in component **msgty** of parameter **i_msg** to a local variable of type **if_abap_behv_message=>t_severity** (suggested name: **severity**).

Hint:
As values for the local variable, use the components of the constant structure **if_abap_behv_message=>severity**.

You can either use a SWITCH expression or a more old-fashioned CASE ... ENDCASE structure.

a) In the implementation of method **map_message**, add the following code:

```
DATA(severity) = SWITCH #( i_msg-msgty
  WHEN 'S' THEN if_abap_behv_message=>severity-success
  WHEN 'I' THEN if_abap_behv_message=>severity-information
  WHEN 'W' THEN if_abap_behv_message=>severity-warning
  WHEN 'E' THEN if_abap_behv_message=>severity-error
  ELSE          if_abap_behv_message=>severity-none ).
```

Alternative solution:

```
DATA severity TYPE if_abap_behv_message=>t_severity.
CASE i_msg-msgty.
  WHEN 'S'.
    severity = if_abap_behv_message=>severity-success.
  WHEN 'I'.
    severity = if_abap_behv_message=>severity-information.
  WHEN 'W'.
    severity = if_abap_behv_message=>severity-warning.
  WHEN 'E'.
    severity = if_abap_behv_message=>severity-error.
  WHEN OTHERS.
    severity = if_abap_behv_message=>severity-none.
ENDCASE.
```

4. Call the inherited functional method **new_message** and assign the result to the returning parameter **r_msg**. Supply the importing parameters with the related components of parameter **i_msg**, except for parameter **severity** which you supply with your local variable **severity**.

a) At the end of the method **map_message**, add the following code:

```
r_msg = new_message(
  id        = i_msg-msgid
  number    = i_msg-msgno
  severity = severity
  v1        = i_msg-msgv1
  v2        = i_msg-msgv2
  v3        = i_msg-msgv3
  v4        = i_msg-msgv4 ).
```

5. Return to the implementation of the **save_modified** method. Find the call of the **delete_item** method and store the result in an inline declared variable (suggested name: **msg_d**).

a) Adjust the code as follows:

```
DATA(msg_d) = model->delete_item( i_uuid = <item_d>-itemuuid ).
```

6. If the result is not initial, append a new row to the **item** component of the **reported** parameter. Fill the **%tky-itemuuid** component with the uuid of the current flight travel item and component **%msg** via a call of the private method that you just implemented.

   a) Adjust the code as follows:

```
DATA(msg_d) = model->delete_item( i_uuid = <item_d>-itemuuid ).
IF msg_d IS NOT INITIAL.
  APPEND VALUE #( %tky-itemuuid = <item_d>-itemuuid
                  %msg          = map_message( msg_d ) )
    TO reported-item.
ENDIF.
```

7. Adjust the call of **create_item** accordingly.

   a) Adjust the code as follows:

```
DATA(msg_c) = model->create_item(
        i_item = CORRESPONDING #( <item_c> MAPPING FROM ENTITY ) ).
IF msg_c IS NOT INITIAL.
  APPEND VALUE #( %tky-itemuuid = <item_c>-itemuuid
                  %msg          = map_message( msg_c ) )
    TO reported-item.
ENDIF.
```

8. Finally, adjust the call of **update_item**.

   a) Adjust the code as follows:

```
DATA(msg_u) = model->update_item(
           i_item  = CORRESPONDING #( <item_u> MAPPING FROM ENTITY )
           i_itemx = CORRESPONDING #( <item_u> MAPPING FROM ENTITY
                                        USING CONTROL ) ).
IF msg_u IS NOT INITIAL.
  APPEND VALUE #( %tky-itemuuid = <item_u>-itemuuid
                  %msg          = map_message( msg_u ) )
    TO reported-item.
ENDIF.
```

9. Activate the ABAP class.

   a) Press **Ctrl + F3** to activate the development object.

10. Retest the preview for the OData UI service. Ensure that the messages returned by the **delete_item**, **create_item**, and **update_item** methods of class **/LRN/CL_S4D437_TRITEM** are displayed on the UI.

    a) Open the service binding, place the cursor on the root entity and choose *Preview...*.

    b) In the preview, display the details for one of the travels.

    c) Choose *Edit* to change to edit mode.

    d) Make sure that a corresponding success message is displayed when you have successfully created a new travel item.

    e) Also, make sure that appropriate messages are displayed when updating and deleting existing travel items.

Define, Raise, and Handle a Business Event

In this exercise, you define an event in the behavior definition of your business object and raise it in the behavior implementation during the creation of new flight travels. You then handle the event locally in an ABAP class.

Note:
In this exercise, replace ## with your group number.

Table 24: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437G_R_TRAVEL*
ABAP Class (Behavior Pool)	*/LRN/BP_437G_R_TRAVEL*
ABAP Class (Event Handler)	*/LRN/CL_437G_HANDLER*

Task 1: Define and Raise the Event

Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. In the behavior for the root entity, define an event (suggested name: **`TravelCreated`**). Raise the event in an additional save implementation, when one or more new flight travels are created.

1. Edit your behavior definition on data model level **`Z##_R_TRAVEL`** and locate the `define behavior` statement for the root entity **`Z##_R_Travel`**.

2. For the root entity, define an event (suggested name: **`TravelCreated`**).

3. Enable additional save logic for the root entity.

4. Activate the behavior definition.

5. Navigate to the implementation of the **`save_modified`** method.

   Note:
   In our case, this method already exists because the travel item behavior contains the with unmanaged save addition. Otherwise, you have to use a quick fix to generate the method and, if necessary, the saver class.

6. At the end of the **`save_modified`** method implementation, check if there are any new flight travels.

Hint:
Evaluate the **travel** component of the **create** importing parameter.

7. If there are newly created flight travels, raise the **TravelCreated** event, passing the key values from the **create-travel** parameter.

Hint:
A CORRESPONDING expression automatically passes the key fields.

8. Activate the behavior implementation.

Task 2: Handle the Event

Create a new global class (suggested name: **ZCL_##_HANDLER**) to handle the **TravelCreated** event you defined in the previous task. Implement the handler method with a MODIFY ENTITIES statement that uses the business object interface **/LRN/437_I_TRAVELLOG** to create a log entry for the new flight travels.

Note:
Make sure to populate the **Origin** field with the name of your own business object **Z##_R_TRAVEL**. It helps you to distinguish your log entries from entries created by other learners.

1. Create a new global class (suggested name: **ZCL_##_HANDLER**).

2. Enable the global class to handle events of the root entity of your business object **Z##_R_TRAVEL**.

3. Inside the global class, define a local class (suggested name: **lcl_handler**) that inherits from the global class **CL_ABAP_BEHAVIOR_EVENT_HANDLER**. Define a private section in the class.

Hint:
If you use the **lcl** code template to insert the local class, remember to remove the PUBLIC SECTION and PROTECTED SECTION statements. They are not allowed in sub classes of **CL_ABAP_BEHAVIOR_EVENT_HANDLER**.

4. In the local class, define a private method (suggested name: **on_travel_created**) to handle the **TravelCreated** event of the root entity.

Note:
Remember that you need the IMPORTING addition, even though the event does not yet define a parameter.

5. At the beginning of the method implementation, declare an internal table (suggested name: **`log`**) with the derived type **`TABLE FOR CREATE /lrn/437_i_travellog`**.

6. Implement a loop over the importing parameter and fill the internal table with the key values of the new flight travels. In addition, fill the **`Origin`** component with the name of your own business object **`Z##_R_TRAVEL`**.

7. After the loop, implement a MODIFY ENTITIES statement that accesses the **`/LRN/437_I_TRAVELLOG`** business object interface.

8. Create new instances of the root entity **`TravelLog`**, passing the fields **`AgencyID`**, **`TravelID`**, and **`Origin`**. As input, use the internal table **`log`**.

> Note:
> When you use EML to create new instances, always populate the **`%cid`** field with non-initial values. An alternative is the use of the AUTO FILL CID addition.

9. Activate the handler class and test the event handling.

Unit 7
Solution 20

Define, Raise, and Handle a Business Event

In this exercise, you define an event in the behavior definition of your business object and raise it in the behavior implementation during the creation of new flight travels. You then handle the event locally in an ABAP class.

Note:
In this exercise, replace ## with your group number.

Table 24: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437G_R_TRAVEL*
ABAP Class (Behavior Pool)	*/LRN/BP_437G_R_TRAVEL*
ABAP Class (Event Handler)	*/LRN/CL_437G_HANDLER*

Task 1: Define and Raise the Event

Edit the behavior definition on data model level **`Z##_R_TRAVEL`**. In the behavior for the root entity, define an event (suggested name: **`TravelCreated`**). Raise the event in an additional save implementation, when one or more new flight travels are created.

1. Edit your behavior definition on data model level **`Z##_R_TRAVEL`** and locate the `define behavior` statement for the root entity **`Z##_R_Travel`**.
   a) For example, you can press **`Ctrl + F`** to invoke a search dialog where you search for **`R_Travel`**.
2. For the root entity, define an event (suggested name: **`TravelCreated`**).
   a) Somewhere in the behavior definition, for example after the definition of the **`cancel_travel`** action, add the following code:

```
event TravelCreated;
```

3. Enable additional save logic for the root entity.
   a) At the end of the define behavior statement for the root entity, add the `with additional save` addition as follows:

```
define behavior for Z##_R_Travel alias Travel
persistent table z##_travel
draft table z##_travel_d
lock master
total etag ChangedAt
authorization master ( instance )
etag master LocChangedAt
early numbering
```

```
with additional save
{
```

4. Activate the behavior definition.

   a) Press **Ctrl + F3** to activate the development object.

5. Navigate to the implementation of the **save_modified** method.

> Note:
> In our case, this method already exists because the travel item behavior contains the with unmanaged save addition. Otherwise, you have to use a quick fix to generate the method and, if necessary, the saver class.

   a) For example, scroll up to the managed implementation in class statement. Place the cursor on `zbp_##_r_travel` and press **F3**.

   b) Choose the *Local Types* tab.

   c) Scroll down to the `METHOD save_modified.` statement.

6. At the end of the **save_modified** method implementation, check if there are any new flight travels.

> Hint:
> Evaluate the **travel** component of the **create** importing parameter.

   a) Before the next ENDMETHOD statement, add the following code:

```
IF create-travel IS NOT INITIAL.

ENDIF.
```

7. If there are newly created flight travels, raise the **TravelCreated** event, passing the key values from the **create-travel** parameter.

> Hint:
> A CORRESPONDING expression automatically passes the key fields.

   a) Adjust the code as follows:

```
IF create-travel IS NOT INITIAL.
  RAISE ENTITY EVENT Z##_R_Travel~TravelCreated
    FROM CORRESPONDING #( create-travel ) .
ENDIF.
```

8. Activate the behavior implementation.

   a) Press **Ctrl + F3** to activate the development object.

Task 2: Handle the Event

Create a new global class (suggested name: **ZCL_##_HANDLER**) to handle the **TravelCreated** event you defined in the previous task. Implement the handler method with a MODIFY ENTITIES statement that uses the business object interface **/LRN/437_I_TRAVELLOG** to create a log entry for the new flight travels.

Note:
Make sure to populate the **Origin** field with the name of your own business object **Z##_R_TRAVEL**. It helps you to distinguish your log entries from entries created by other learners.

1. Create a new global class (suggested name: **ZCL_##_HANDLER**).

   a) In the *Project Explorer*, right-click the name of your package **ZS4D437_##** and choose *New* → *ABAP Class*.

   b) In the *Name* field, enter **ZCL_##_HANDLER** and in the *Description* field, enter **Handle Business Event TravelCreated**.

   c) Choose *Next*, assign the new development object to a transport request, then choose *Finish*.

2. Enable the global class to handle events of the root entity of your business object **Z##_R_TRAVEL**.

   a) Add the FOR EVENTS OF statement, followed by the name of your root entity as follows:

```
CLASS zcl_##_handler DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC FOR EVENTS OF Z##_R_Travel.
```

3. Inside the global class, define a local class (suggested name: **lcl_handler**) that inherits from the global class **CL_ABAP_BEHAVIOR_EVENT_HANDLER**. Define a private section in the class.

   Hint:
   If you use the **lcl** code template to insert the local class, remember to remove the PUBLIC SECTION and PROTECTED SECTION statements. They are not allowed in sub classes of **CL_ABAP_BEHAVIOR_EVENT_HANDLER**.

   a) Navigate to the *Local Types* tab.

   b) Add the following code:

```
CLASS lcl_handler DEFINITION
  INHERITING FROM cl_abap_behavior_event_handler.

  PRIVATE SECTION.

ENDCLASS.

CLASS lcl_handler IMPLEMENTATION.

ENDCLASS.
```

4. In the local class, define a private method (suggested name: **on_travel_created**) to handle the **TravelCreated** event of the root entity.

Note:
Remember that you need the IMPORTING addition, even though the event does not yet define a parameter.

a) Adjust the code as follows:

```
CLASS lcl_handler DEFINITION
  INHERITING FROM cl_abap_behavior_event_handler.

  PRIVATE SECTION.
    METHODS on_travel_created FOR ENTITY EVENT
      IMPORTING new_travels
        FOR Travel~TravelCreated.
ENDCLASS.

CLASS lcl_handler IMPLEMENTATION.

  METHOD on_travel_created.

  ENDMETHOD.

ENDCLASS.
```

5. At the beginning of the method implementation, declare an internal table (suggested name: **log**) with the derived type **TABLE FOR CREATE /lrn/437_i_travellog**.

a) Adjust the code as follows:

```
METHOD on_travel_created.
  DATA log TYPE TABLE FOR CREATE /lrn/437_i_travellog.

ENDMETHOD.
```

6. Implement a loop over the importing parameter and fill the internal table with the key values of the new flight travels. In addition, fill the **Origin** component with the name of your own business object **Z##_R_TRAVEL**.

a) Adjust the code as follows:

```
METHOD on_travel_created.
  DATA log TYPE TABLE FOR CREATE /lrn/437_i_travellog.

  LOOP AT new_travels ASSIGNING FIELD-SYMBOL(<travel>).
    APPEND VALUE #( AgencyID = <travel>-AgencyId
                    TravelID = <travel>-TravelId
                    Origin   = 'Z##_R_TRAVEL' )
      TO log.
  ENDLOOP.
ENDMETHOD.
```

7. After the loop, implement a MODIFY ENTITIES statement that accesses the **/LRN/437_I_TRAVELLOG** business object interface.

a) At the end of the method, insert the following code:

```
MODIFY ENTITIES OF /LRN/437_I_TravelLog
```

8. Create new instances of the root entity **TravelLog**, passing the fields **AgencyID**, **TravelID**, and **Origin**. As input, use the internal table **log**.

Note:
When you use EML to create new instances, always populate the **%cid** field with non-initial values. An alternative is the use of the AUTO FILL CID addition.

a) Adjust the code as follows:

```
MODIFY ENTITIES OF /LRN/437_I_TravelLog
  ENTITY TravelLog
    CREATE AUTO FILL CID
    FIELDS ( AgencyID TravelID Origin )
    WITH log.
```

9. Activate the handler class and test the event handling.

   a) Press **Ctrl + F3** to activate the development object.

   b) Reopen the preview of your OData UI service and choose *Create* to create a new flight travel.

   c) Enter all mandatory fields for the flight travel and choose *Create*.

   d) Take note of the values displayed in the *Agency ID* field and the *Travel ID* field.

   e) Open the preview for OData UI service **/LRN/UI_437_TRAVELLOG_O2**.

   f) Choose *Go*. (If there are too many entries, enter **Z##_R_TRAVEL** under *Root Entity Name* and choose *Go* again.)

      Result

      You see an entry that corresponds to the flight travel that you have created.

Unit 7
Exercise 21

Add a Parameter to the Business Event

In this exercise, you add a parameter to the business event that you defined in the previous exercise. For that, you define a CDS abstract entity and use it as parameter type.

Note:
In this exercise, replace ## with your group number.

Table 25: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Abstract Entity)	*/LRN/437G_A_EVENT*
CDS Behavior Definition (Model)	*/LRN/437G_R_TRAVEL*
ABAP Class (Behavior Pool)	*/LRN/BP_437G_R_TRAVEL*
ABAP Class (Event Handler)	*/LRN/CL_437G_HANDLER*

Task 1: Define a CDS Abstract Entity

Define a CDS abstract entity (suggested name: **`Z##_A_Event`**) with a single element of type **`abp_root_entity_name`** (suggested name: **`origin`**).

1. Create a new data definition (suggested name: **`Z##_A_Event`**) that defines a CDS abstract entity. Assign it to your own package and use the same transport request as before.

2. In the editor that opens, remove the with parameters addition, because the abstract entity does not need parameters.

3. In the element list, replace placeholder `element_name` with **`origin`** and placeholder `element_type` with **`abp_root_entity_name`**.

4. Activate the abstract entity.

Task 2: Add an Event Parameter

Add a parameter to your event and use the abstract entity **`Z##_A_Event`** as parameter type. Adjust the RAISE ENTITY EVENT statement and populate the parameter with the name of your root entity in upper case (**`Z##_R_TRAVEL`**). Finally, remove the hard-coded value for **`Origin`** from the event handler implementation.

1. Edit your behavior definition on data model level **`Z##_R_TRAVEL`** and locate the definition of the **`TravelCreated`** event.

2. Add an event parameter that is typed with your abstract entity.

3. Activate the behavior definition and use the *Where-Used List* to navigate to the RAISE ENTITY EVENT statement for the event.

4. Adjust the RAISE ENTITY EVENT statement. Make sure the data object after the FROM addition contains not only the key values (columns **`AgencyId`** and **`TravelId`**), but also the name of your root view entity in upper case in the **`origin`** column.

> Note:
> Either define a data object of type `TABLE FOR EVENT Z##_R_Travel~TravelCreated` and fill it in a LOOP-ENDLOOP structure before the RAISE ENTITY EVENT statement, or use a VALUE expression after FROM with a FOR iteration inside.

5. Activate the behavior implementation and use the *Where-Used List* for the event to navigate to the handler method.

6. In the MODIFY ENTITIES statement, remove the data object after WITH and replace it with a CORRESPONDING expression, having the event parameter as input.

> Note:
> This is sufficient because the event parameter now contains the **`origin`** component.

7. Remove or comment the superfluous data declaration and the loop.

8. Activate the handler class and retest the event handling.

Add a Parameter to the Business Event

In this exercise, you add a parameter to the business event that you defined in the previous exercise. For that, you define a CDS abstract entity and use it as parameter type.

Note:
In this exercise, replace ## with your group number.

Table 25: Solution

Repository Object Type	Repository Object ID
CDS Data Definition (Abstract Entity)	*/LRN/437G_A_EVENT*
CDS Behavior Definition (Model)	*/LRN/437G_R_TRAVEL*
ABAP Class (Behavior Pool)	*/LRN/BP_437G_R_TRAVEL*
ABAP Class (Event Handler)	*/LRN/CL_437G_HANDLER*

Task 1: Define a CDS Abstract Entity

Define a CDS abstract entity (suggested name: **`Z##_A_Event`**) with a single element of type **`abp_root_entity_name`** (suggested name: **`origin`**).

1. Create a new data definition (suggested name: **`Z##_A_Event`**) that defines a CDS abstract entity. Assign it to your own package and use the same transport request as before.

   a) In the *Project Explorer* view, open the context menu for the *Data Definitions* folder that is located under the *Core Data Services* folder in your package **`ZS4D437_##`**.

   b) From the context menu, select *New Data Definition*.

   c) Enter the name of the new data definition and a description and choose *Next*.

   d) Select the same transport request as before and choose *Next*.

   e) From the list of templates, choose *Abstract Entity (creation)* → *defineAbstractEntityWithParameters* and choose *Finish*.

2. In the editor that opens, remove the **with parameters** addition, because the abstract entity does not need parameters.

   a) Adjust the code as follows:

```
define abstract entity Z##_A_Event
{
```

3. In the element list, replace placeholder `element_name` with **`origin`** and placeholder `element_type` with **`abp_root_entity_name`**.

a) Adjust the code as follows:

```
define abstract entity Z##_A_Event
{
  origin : abp_root_entity_name;
}
```

4. Activate the abstract entity.

   a) Press **Ctrl + F3** to activate the development object.

Task 2: Add an Event Parameter

Add a parameter to your event and use the abstract entity **Z##_A_Event** as parameter type. Adjust the RAISE ENTITY EVENT statement and populate the parameter with the name of your root entity in upper case (**Z##_R_TRAVEL**). Finally, remove the hard-coded value for **Origin** from the event handler implementation.

1. Edit your behavior definition on data model level **Z##_R_TRAVEL** and locate the definition of the **TravelCreated** event.

   a) For example, you can press **Ctrl + F** to invoke a search dialog where you search for **EVENT**.

2. Add an event parameter that is typed with your abstract entity.

   a) Adjust the code as follows:

```
event TravelCreated parameter Z##_A_Event;
```

3. Activate the behavior definition and use the *Where-Used List* to navigate to the RAISE ENTITY EVENT statement for the event.

   a) Press **Ctrl + F3** to activate the development object.

   b) In the EVENT statement, place the cursor on the event name and press **Ctrl + Shift + G**.

   c) On the *Get Where-Used List* dialog that appears, choose *Finish*.

   d) From the *Where-Used List* in the *Search* view below the editor, choose *ZBP_##_R_TRAVEL (Class)* → *Local Class Implementation* → *RAISE ENTITY EVENT Z##_R_Travel~TravelCreated*.

4. Adjust the RAISE ENTITY EVENT statement. Make sure the data object after the FROM addition contains not only the key values (columns **AgencyId** and **TravelId**), but also the name of your root view entity in upper case in the **origin** column.

> Note:
> Either define a data object of type `TABLE FOR EVENT Z##_R_Travel~TravelCreated` and fill it in a LOOP-ENDLOOP structure before the RAISE ENTITY EVENT statement, or use a VALUE expression after FROM with a FOR iteration inside.

   a) Adjust the code as follows:

```
IF create-travel IS NOT INITIAL.
  DATA event_in TYPE TABLE FOR EVENT Z##_R_Travel~TravelCreated.

  LOOP AT create-travel ASSIGNING FIELD-SYMBOL(<new_travel>).
```

```
    APPEND VALUE #( AgencyId = <new_travel>-AgencyId
                    TravelId = <new_travel>-TravelId
                    origin   = 'Z##_R_TRAVEL' )
      TO event_in.
  ENDLOOP.

  RAISE ENTITY EVENT Z##_R_Travel~TravelCreated
    FROM event_in.
ENDIF.
```

Alternative implementation using a VALUE expression:

```
IF create-travel IS NOT INITIAL.
  RAISE ENTITY EVENT Z##_R_Travel~TravelCreated
    FROM VALUE #( FOR <new_travel> IN create-travel
                    (
                      AgencyId = <new_travel>-AgencyId
                      TravelId = <new_travel>-TravelId
                      origin   = 'Z##_R_TRAVEL'
                    )
                ).
ENDIF.
```

5. Activate the behavior implementation and use the *Where-Used List* for the event to navigate to the handler method.
   a) Press **`Ctrl + F3`** to activate the development object.
   b) If the *Where-Used List* for the event is no longer visible, place the cursor on the event name in the RAISE ENTITY EVENT statement and press **`Ctrl + Shift + G`**.
   c) On the *Get Where-Used List* dialog that appears, choose *Finish*.
   d) From the *Where-Used List* in the *Search* view below the editor, choose *ZCL_##_HANDLER (Class)* → *Local Class Implementation* → *FOR Travel~TravelCreated*.
6. In the MODIFY ENTITIES statement, remove the data object after WITH and replace it with a CORRESPONDING expression, having the event parameter as input.

   Note:
   This is sufficient because the event parameter now contains the **`origin`** component.

   a) Adjust the code as follows:

```
MODIFY ENTITIES OF /LRN/437_I_TravelLog
  ENTITY TravelLog
    CREATE AUTO FILL CID
    FIELDS ( AgencyID TravelID Origin )
    WITH CORRESPONDING #( new_travels ).
```

7. Remove or comment the superfluous data declaration and the loop.
   a) Select the DATA statement, the ENDLOOP statement, and all code rows between them and press **`Ctrl + <`** to insert a comment sign at the beginning of each selected code row.
8. Activate the handler class and retest the event handling.
   a) Press **`Ctrl + F3`** to activate the development object.

b) Reopen the preview of your OData UI service and choose *Create* to create a new flight travel.

c) Enter all mandatory fields for the flight travel and choose *Create*.

d) Take note of the values displayed in the *Agency ID* field and the *Travel ID* field.

e) Open the preview for OData UI service **`/LRN/UI_437_TRAVELLOG_O2`**.

f) Choose *Go*. (If there are too many entries, enter **`Z##_R_TRAVEL`** under *Root Entity Name* and choose *Go* again.)

Result

You see an entry that corresponds to the flight travel that you created.

Unit 8
Exercise 22

Enable the Data Model Extensibility

In this exercise, you enable the extension of the data model, that is, the extension of the business object and its projection. First, you enable the extension of the database table for flight travel items and the related draft table. Then, you enable the extension of the data model view and the projection view for flight travel items.

Note:
In this exercise, replace ## with your group number.

Table 26: Solution

Repository Object Type	Repository Object ID
Database Table Definition (Active Data)	*/LRN/437H_TRITEM*
Database Table Definition (Draft)	*/LRN/437H_TRIT_D*
Structure Type (Extension Include Structure)	*/LRN/437H_S_EXT_TRITEM*
CDS Data Definition (Model)	*/LRN/437H_R_TRAVELITEM*
CDS Data Definition (Projection)	*/LRN/437H_C_TRAVELITEM*
CDS Data Definition (Extension Include View)	*/LRN/437H_E_TRAVELITEM*

Task 1: Enable Extension of DB Tables

Enable the extension of your database table for flight travel items **`Z##_TRITEM`** and of the corresponding draft table **`Z##_TRITEM_D`**. Follow the best practice to ensure that future extensions are consistent: Define an extension include structure (suggested name: **`Z##_S_EXT_TRITEM`**) with a dummy component and include it in both database table definitions. Then define the extension include structure and both database tables as extensible with char-like and numeric fields and define a mandatory field suffix (suggested suffix: **`ZIT`**).

1. Create a structure type as extension include (suggested name: **`Z##_S_EXT_TRITEM`**).
2. In the structure type, define a single component (suggested name: **`dummy_field`**). As the component type, use the built-in dictionary type CHAR with a length of 1.
3. Enable extensibility for the structure type. Allow extension with char-like and numeric fields but not with deep component types like, for example, table types.
4. Define a mandatory suffix **`Z##`** for extension fields.
5. Activate the structure type.
6. Edit the definition of your database table for flight travel items **`Z##_TRITEM`** and include the structure type into the field list.

Note:
As you C1 released the database table in a previous exercise, you will receive a popup with the following message when you try to edit the table definition: *Object is a stable API. Do not change incompatibly.* Confirm this popup with *OK*.

Why does this cause syntax errors?

7. Enable extensibility for the database table definition with the same enhancement category as in the extension include structure and activate the database table definition.

8. Edit the definition of your draft table for flight travel items **Z##_TRITEM_D**. Include the structure as before and activate the database table definition.

   Why is it not necessary to adjust the value of the *AbapCatalog.enhancement.category* annotation?

Task 2: Enable Extension of CDS Views

Enable the extension of the CDS view entities for flight travel items. Follow the best practice to ensure consistent extensions: Define an extension include view (suggested name: **Z##_E_TravelItem**) with only the key field of flight travel items. Then add an association to the view on data model level **Z##_R_TravelItem** with the extension include view as target (suggested association name: **_Extension**).

Allow the extension of the view on data model level **Z##_R_TravelItem** and the projection view **Z##_C_TravelItem**. In both cases, restrict the extensibility to elements from the target of the association **_Extension**.

1. Create a CDS view entity (suggested name: **Z##_E_TravelItem**) with your database table for active flight travel items **Z##_TRITEM** as data source.

   Note:
   Specify the database table as a *Referenced Object* to generate the element list.

2. In the definition of the view entity, remove all elements except for the key element **ItemUuid**.

3. Enable extensibility for the extension view entity.

4. Restrict the extension to the existing data source.

Note:
To achieve this, you have to define an alias name for the data source (suggested name: **`Item`**). You define this alias in the FROM clause of the SELECT statement.

5. Define a mandatory suffix **`Z##`** for extension elements and activate the data definition.

6. Edit the definition of your CDS view entity for travel items on data model level **`Z##_R_TravelItem`**. Define and expose an association with the extension include view as target (suggested association name: **`_Extension`**).

7. Enable extensibility for the CDS view entity and activate the data definition. Specify the same mandatory element suffix and restrict the extensibility with the **`_Extension`** association as the only allowed data source.

Note:
Adding annotation *AbapCatalog.viewEnhancementCategory* is optional. If it's missing, the default value *[#PROJECTION_LIST]* is used.

8. Edit the definition of your CDS view entity for travel items on projection level **`Z##_C_TravelItem`**. Enable extensibility for the CDS view entity and activate the data definition. Specify the same mandatory element suffix as before and restrict the extensibility with the primary data source **`Z##_R_TravelItem`** as the only allowed data source.

Note:
To achieve this, you must define an alias name for the data source (suggested name: **`Item`**) in the AS PROJECTION ON addition.

Enable the Data Model Extensibility

In this exercise, you enable the extension of the data model, that is, the extension of the business object and its projection. First, you enable the extension of the database table for flight travel items and the related draft table. Then, you enable the extension of the data model view and the projection view for flight travel items.

Note:
In this exercise, replace ## with your group number.

Table 26: Solution

Repository Object Type	Repository Object ID
Database Table Definition (Active Data)	*/LRN/437H_TRITEM*
Database Table Definition (Draft)	*/LRN/437H_TRIT_D*
Structure Type (Extension Include Structure)	*/LRN/437H_S_EXT_TRITEM*
CDS Data Definition (Model)	*/LRN/437H_R_TRAVELITEM*
CDS Data Definition (Projection)	*/LRN/437H_C_TRAVELITEM*
CDS Data Definition (Extension Include View)	*/LRN/437H_E_TRAVELITEM*

Task 1: Enable Extension of DB Tables

Enable the extension of your database table for flight travel items **Z##_TRITEM** and of the corresponding draft table **Z##_TRITEM_D**. Follow the best practice to ensure that future extensions are consistent: Define an extension include structure (suggested name: **Z##_S_EXT_TRITEM**) with a dummy component and include it in both database table definitions. Then define the extension include structure and both database tables as extensible with char-like and numeric fields and define a mandatory field suffix (suggested suffix: **ZIT**).

1. Create a structure type as extension include (suggested name: **Z##_S_EXT_TRITEM**).
   a) In the *Project Explorer*, locate your package **ZS4D437_##** under the *Favorite Packages* node.
   b) Expand your package and right-click the *Dictionary* sub node. Choose *New → Structure*.
   c) Enter **Z##_S_EXT_TRITEM** as *Name* and **Extension Include Structure for Travel Items** as *Description*. Then choose *Next*.
   d) Confirm the transport request and choose *Finish*.

2. In the structure type, define a single component (suggested name: **`dummy_field`**). As the component type, use the built-in dictionary type CHAR with a length of 1.
   a) In the structure type definition, replace the `component_to_be_changed` placeholder with **`dummy_field`**.
   b) Replace `abap.string(0)` with **`abap.char(1)`**.
3. Enable extensibility for the structure type. Allow extension with char-like and numeric fields but not with deep component types like, for example, table types.
   a) In the *AbapCatalog.enhancement.category* annotation, replace the default value `#NOT_EXTENSIBLE` with **`#EXTENSIBLE_CHARACTER_NUMERIC`**.

> Hint:
> Use code completion to choose from the available enhancement categories.

4. Define a mandatory suffix **`Z##`** for extension fields.
   a) Adjust the code as follows:

```
@EndUserText.label : 'Extension Include Structure for Travel Items'
@AbapCatalog.enhancement.category : #EXTENSIBLE_CHARACTER_NUMERIC
@AbapCatalog.enhancement.fieldSuffix : 'Z##'
define structure z##_s_ext_tritem {

  dummy_field : abap.char(1);

}
```

5. Activate the structure type.
   a) Press **`Ctrl + F3`** to activate the development object.
6. Edit the definition of your database table for flight travel items **`Z##_TRITEM`** and include the structure type into the field list.

> Note:
> As you C1 released the database table in a previous exercise, you will receive a popup with the following message when you try to edit the table definition: *Object is a stable API. Do not change incompatibly.* Confirm this popup with *OK*.

   a) At the end of the field list, add the following code:

```
include z##_s_ext_tritem;
```

Why does this cause syntax errors?

The enhancement category of the including object, the database table definition, is more restrictive than the enhancement category of the included object, the extension include structure.

7. Enable extensibility for the database table definition with the same enhancement category as in the extension include structure and activate the database table definition.

a) In the *AbapCatalog.enhancement.category* annotation, replace the default value `#NOT_EXTENSIBLE` with **`#EXTENSIBLE_CHARACTER_NUMERIC`**.

b) Press **`Ctrl + F3`** to activate the development object.

8. Edit the definition of your draft table for flight travel items **`Z##_TRITEM_D`**. Include the structure as before and activate the database table definition.

a) Add the following code at the end of the field list:

```
include z##_s_ext_tritem;
```

b) Press **`Ctrl + F3`** to activate the development object.

Why is it not necessary to adjust the value of the *AbapCatalog.enhancement.category* annotation?

Draft tables are generated with value **`#EXTENSIBLE_ANY`** which is less restrictive than value **`#EXTENSIBLE_CHARACTER_NUMERIC`**.

Task 2: Enable Extension of CDS Views

Enable the extension of the CDS view entities for flight travel items. Follow the best practice to ensure consistent extensions: Define an extension include view (suggested name: **`Z##_E_TravelItem`**) with only the key field of flight travel items. Then add an association to the view on data model level **`Z##_R_TravelItem`** with the extension include view as target (suggested association name: **`_Extension`**).

Allow the extension of the view on data model level **`Z##_R_TravelItem`** and the projection view **`Z##_C_TravelItem`**. In both cases, restrict the extensibility to elements from the target of the association **`_Extension`**.

1. Create a CDS view entity (suggested name: **`Z##_E_TravelItem`**) with your database table for active flight travel items **`Z##_TRITEM`** as data source.

> Note:
> Specify the database table as a *Referenced Object* to generate the element list.

a) In the *Project Explorer*, locate the database table for active travel items **`Z##_TRITEM`**.

b) Right-click the database table and choose *New Data Definition*.

c) Enter **`Z##_E_TravelItem`** as *Name* and **`Extension Include for Travel Items`** as *Description*. Make sure that the *Referenced Object* field contains the name of the database table and choose *Next*.

d) Confirm the transport request and choose *Next*.

e) From the list of templates, select the *defineViewEntity* template and choose *Finish*.

2. In the definition of the view entity, remove all elements except for the key element **`ItemUuid`**.

a) In the element list, select all elements without the keyword KEY and press the **`Delete`** key.

b) Remove the comma at the end of the remaining element.

3. Enable extensibility for the extension view entity.

   a) In the *AbapCatalog.viewEnhancementCategory* annotation, replace the default value `#NONE` with **`#PROJECTION_LIST`**.

   b) Before the DEFINE VIEW ENTITY statement, insert the following code:

```
@AbapCatalog.extensibility: {
  extensible: true
}
```

4. Restrict the extension to the existing data source.

   Note:
   To achieve this, you have to define an alias name for the data source (suggested name: **`Item`**). You define this alias in the FROM clause of the SELECT statement.

   a) Adjust the DEFINE VIEW ENTITY statement as follows:

```
define view entity Z##_E_TravelItem
  as select from z##_tritem as Item
{
    key item_uuid as ItemUuid
}
```

   b) Adjust the *AbapCatalog.extensibility* annotation as follows:

```
@AbapCatalog.extensibility: {
  extensible: true,
  allowNewDatasources: false,
  dataSources: ['Item']
}
```

5. Define a mandatory suffix **`Z##`** for extension elements and activate the data definition.

   a) Adjust the *AbapCatalog.extensibility* annotation as follows:

```
@AbapCatalog.extensibility: {
  extensible: true,
  allowNewDatasources: false,
  dataSources: ['Item'],
  elementSuffix: 'Z##'
}
```

   b) Press **`Ctrl + F3`** to activate the development object.

6. Edit the definition of your CDS view entity for travel items on data model level **`Z##_R_TravelItem`**. Define and expose an association with the extension include view as target (suggested association name: **`_Extension`**).

   a) Adjust the code as follows:

```
define view entity Z##_R_TravelItem
  as select from z##_tritem
  association to parent Z##_R_Travel as _Travel
    on  $projection.AgencyId = _Travel.AgencyId
    and $projection.TravelId = _Travel.TravelId

  association to Z##_E_TravelItem    as _Extension
     on $projection.ItemUuid = _Extension.ItemUuid
{
```

b) At the end of the element list, adjust the code as follows:

```
  _Travel,
  _Extension
}
```

7. Enable extensibility for the CDS view entity and activate the data definition. Specify the same mandatory element suffix and restrict the extensibility with the **_Extension** association as the only allowed data source.

> Note:
> Adding annotation *AbapCatalog.viewEnhancementCategory* is optional. If it's missing, the default value *[#PROJECTION_LIST]* is used.

a) At the beginning of the data definition, add the following code:

```
@AbapCatalog.viewEnhancementCategory: [#PROJECTION_LIST]
@AbapCatalog.extensibility: {
  extensible: true,
  allowNewDatasources: false,
  dataSources: ['_Extension'],
  elementSuffix: 'Z##'
}
```

b) Press **Ctrl + F3** to activate the development object.

8. Edit the definition of your CDS view entity for travel items on projection level **Z##_C_TravelItem**. Enable extensibility for the CDS view entity and activate the data definition. Specify the same mandatory element suffix as before and restrict the extensibility with the primary data source **Z##_R_TravelItem** as the only allowed data source.

> Note:
> To achieve this, you must define an alias name for the data source (suggested name: **Item**) in the AS PROJECTION ON addition.

a) Adjust the DEFINE VIEW ENTITY statement as follows:

```
define view entity Z##_C_TravelItem
  as projection on Z##_R_TravelItem as Item
{
```

b) At the beginning of the data definition, add the following code:

```
@AbapCatalog.viewEnhancementCategory: [#PROJECTION_LIST]
@AbapCatalog.extensibility: {
  extensible: true,
  allowNewDatasources: false,
  dataSources: ['Item'],
  elementSuffix: 'Z##'
}
```

c) Press **Ctrl + F3** to activate the development object.

Unit 8
Exercise 23

Develop Data Model Extensions

In this exercise, you develop an extension of the business object for flight travels and its projection. The goal of this extension is to have an additional booking class field for the flight travel items. To achieve this, you first extend the extension include structure **`Z##_S_EXT_TRITEM`**, which indirectly extends the database table for flight travel items and the related draft table.

After that, you extend the business object and its projection. You begin by extending the extension include view **`Z##_E_TravelItem`** because, in our case, the extension include view is the only allowed data source when extending the view entity on data model level **`Z##_R_TravelItem`**. Then you extend the view entity on data model level **`Z##_R_TravelItem`** and - based on that extension - the projection view entity **`Z##_C_TravelItem`**. Finally, you create a new metadata extension to add the booking class field to the UI metadata of the OData service.

Note:
In this exercise, you extend your own data model objects. This is done for simplicity. To motivate the use of extensions, during this exercise, pretend that the database tables **`Z##_TRITEM`** and **`Z##_TRITEM_D`**, the extension include structure **`Z##_S_EXT_TRITEM`**, the CDS view entities **`Z##_R_TravelItem`** and **`Z##_C_TravelItem`**, and the extension include view **`Z##_E_TravelItem`** lie in a different namespace and that you are not authorized to make direct changes to these objects.

Note:
In this exercise, replace ## with your group number.

Table 27: Solution

Repository Object Type	Repository Object ID
Structure Type (Append Structure)	*/LRN/ZZX_S_TRITEM_CLASS*
CDS Data Definition (Extension Include Extension)	*/LRN/ZZX_E_TRAVELITEM_CLASS*
CDS Data Definition (Model Extension)	*/LRN/ZZX_R_TRAVELITEM_CLASS*
CDS Data Definition (Projection Extension)	*/LRN/ZZX_C_TRAVELITEM_CLASS*
CDS Metadata Extension (Projection)	*/LRN/ZZX_C_TRAVELITEM_CLASS*

Task 1: Extend the Database Tables

Create an append structure (suggested name: **Z##X_S_TRITEM_CLASS**) for your extension include structure **Z##_S_EXT_TRITEM**. Add a field for the booking class (suggested name: **ZZCLASSZ##**) and type the new field with data element **/LRN/CLASS_ID**.

1. Locate your extension include structure in the *Project Explorer*.
2. For the extension include structure, create an append structure (suggested name: **Z##X_S_TRITEM_CLASS**).
3. In the component list of the append structure, define a new component (suggested name: **zzclassz##**) with data element **/LRN/CLASS_ID** as component type.
4. Activate the append structure.
5. Open the definition of the database table for active flight travel items. Open the *Tooltip Description* to confirm that the table contains the new field.

   Result

   You see the new field at the end of the field list. You also see the name of your append structure in the *Extended with* section.
6. Repeat this for the corresponding draft table to confirm that this table also contains the new field.

   Result

   You see the new field at the end of the field list. You also see the name of your append structure in the *Extended with* section.

Task 2: Extend the Data Model Definition

Create a CDS view extension to add the booking class field to the element list of the extension include view (suggested name: **Z##X_E_TravelItem_Class**). Then create a similar CDS view extension for the view entity on data model level (suggested name: **Z##X_R_TravelItem_Class**).

1. In the *Project Explorer*, locate the CDS data definition that contains the definition of your extension include view.
2. For the extension include view, create a CDS view extension (suggested name: **Z##X_E_TravelItem_Class**).
3. Edit the element list of the view extension. Remove the placeholder and use code completion to add the booking class field.
4. To increase readability, define an alias name for the new view element (suggested name: **ZZClassZ##**) and activate the data definition.

> Caution:
> Make sure that the alias name and the field name only differ in uppercase or lowercase!

Why is it crucial that the alias name in the view entity and the field name in the append structure only differ in upper/lower case?

5. For the flight travel item view on data model level **Z##_R_TravelItem**, create a CDS view extension (suggested name: **Z##X_R_TravelItem_Class**).
6. Edit the element list of the view extension. Remove the placeholder and use code completion to add the booking class field.
7. Activate the data definition.

Task 3: Extend the Data Model Projection

Create a CDS view extension to add the booking class field to the CDS view entity on projection level (suggested name: **Z##X_C_TravelItem_Class**). Then create a new metadata extension for the projection view (suggested name: **Z##X_C_TravelItem_Class**). Set the metadata layer to **#CUSTOMER** and add UI metadata for the booking class field.

1. For the travel item view on projection level **Z##_C_TravelItem**, create a CDS view extension (suggested name: **Z##X_C_TravelItem_Class**).
2. Edit the element list of the view extension. Remove the placeholder and use code completion to add the booking class field.
3. Assign a value help to the new field, using **/LRN/437_I_ClassStdVH** as the data source.
4. Activate the data definition.
5. For the flight travel item view on projection level **Z##_C_TravelItem**, create a CDS metadata extension (suggested name: **Z##X_C_TRAVELITEM_CLASS**).
6. In the newly created metadata extension, set the metadata layer to **#CUSTOMER**.
7. Add UI metadata for the booking class field. Make sure, the new field appears on both, the list page and the object page, immediately after the **BookingId** field.

> Hint:
> View element **BookingId** is annotated with *@UI.lineItem.position: 60* and *@UI.identification.position: 60*.

8. Activate the metadata extension.
9. Refresh the preview of the OData UI service and confirm that the booking class extension field is displayed.

   Why are values that you enter not yet persisted on the database?

Develop Data Model Extensions

In this exercise, you develop an extension of the business object for flight travels and its projection. The goal of this extension is to have an additional booking class field for the flight travel items. To achieve this, you first extend the extension include structure **`Z##_S_EXT_TRITEM`**, which indirectly extends the database table for flight travel items and the related draft table.

After that, you extend the business object and its projection. You begin by extending the extension include view **`Z##_E_TravelItem`** because, in our case, the extension include view is the only allowed data source when extending the view entity on data model level **`Z##_R_TravelItem`**. Then you extend the view entity on data model level **`Z##_R_TravelItem`** and - based on that extension - the projection view entity **`Z##_C_TravelItem`**. Finally, you create a new metadata extension to add the booking class field to the UI metadata of the OData service.

Note:
In this exercise, you extend your own data model objects. This is done for simplicity. To motivate the use of extensions, during this exercise, pretend that the database tables **`Z##_TRITEM`** and **`Z##_TRITEM_D`**, the extension include structure **`Z##_S_EXT_TRITEM`**, the CDS view entities **`Z##_R_TravelItem`** and **`Z##_C_TravelItem`**, and the extension include view **`Z##_E_TravelItem`** lie in a different namespace and that you are not authorized to make direct changes to these objects.

Note:
In this exercise, replace ## with your group number.

Table 27: Solution

Repository Object Type	Repository Object ID
Structure Type (Append Structure)	*/LRN/ZZX_S_TRITEM_CLASS*
CDS Data Definition (Extension Include Extension)	*/LRN/ZZX_E_TRAVELITEM_CLASS*
CDS Data Definition (Model Extension)	*/LRN/ZZX_R_TRAVELITEM_CLASS*
CDS Data Definition (Projection Extension)	*/LRN/ZZX_C_TRAVELITEM_CLASS*
CDS Metadata Extension (Projection)	*/LRN/ZZX_C_TRAVELITEM_CLASS*

Task 1: Extend the Database Tables

Create an append structure (suggested name: **Z##X_S_TRITEM_CLASS**) for your extension include structure **Z##_S_EXT_TRITEM**. Add a field for the booking class (suggested name: **ZZCLASSZ##**) and type the new field with data element **/LRN/CLASS_ID**.

1. Locate your extension include structure in the *Project Explorer*.
   a) In the *Project Explorer*, expand the *Favorite Packages* node.
   b) Expand your own package **ZS4D437_##**.
   c) Under your package, expand *Dictionary* → *Structures*.
2. For the extension include structure, create an append structure (suggested name: **Z##X_S_TRITEM_CLASS**).
   a) In the *Project Explorer*, right-click **Z##_S_EXT_TRITEM** and select *New Append Structure*.
   b) On the *New Append Structure* dialog that opens, enter **Z##X_S_TRITEM_CLASS** in the *Name* field and **Extend Travel Item with Booking Class** in the *Description* field. Then choose *Next*.
   c) Assign the append structure to a transport request and choose *Finish*.
3. In the component list of the append structure, define a new component (suggested name: **zzclassz##**) with data element **/LRN/CLASS_ID** as component type.
   a) Adjust the code as follows:

```
extend type z##_s_ext_tritem with z##x_s_tritem_class {
  zzclassz## : /lrn/class_id;
}
```

4. Activate the append structure.
   a) Press **Ctrl + F3** to activate the development object.
5. Open the definition of the database table for active flight travel items. Open the *Tooltip Description* to confirm that the table contains the new field.
   a) In the *Project Explorer*, under your own package, expand *Dictionary* → *Database Tables*.
   b) Double-click *Z##_TRITEM* to open the definition of the database table.
   a) In the code row that begins with `define table`, place the cursor on `z##_tritem` and press **F2**.

   Result

   You see the new field at the end of the field list. You also see the name of your append structure in the *Extended with* section.
6. Repeat this for the corresponding draft table to confirm that this table also contains the new field.
   a) In the *Project Explorer*, double-click *Z##_TRITEM_D* to open the definition of the draft table.
   a) In the code row that begins with `define table`, place the cursor on `z##_tritem_d` and press **F2**.

Result

You see the new field at the end of the field list. You also see the name of your append structure in the *Extended with* section.

Task 2: Extend the Data Model Definition

Create a CDS view extension to add the booking class field to the element list of the extension include view (suggested name: **`Z##X_E_TravelItem_Class`**). Then create a similar CDS view extension for the view entity on data model level (suggested name: **`Z##X_R_TravelItem_Class`**).

1. In the *Project Explorer*, locate the CDS data definition that contains the definition of your extension include view.

   a) In the *Project Explorer*, expand the *Favorite Packages* node.

   b) Expand your own package **`ZS4D437_##`**.

   c) Under your package, expand *Core Data Services* → *Data Definitions*.

2. For the extension include view, create a CDS view extension (suggested name: **`Z##X_E_TravelItem_Class`**).

   a) In the *Project Explorer*, right-click **`Z##_E_TRAVELITEM`** and select *New Data Definition*.

   b) In the *Name* field, enter **`Z##X_E_TravelItem_Class`** and in the *Description* field, enter **`Extend Travel Item with Booking Class`**. Then choose *Next*.

   c) Confirm the transport request and choose *Next*.

   Note:
   Do not choose *Finish* yet or you will not be able to choose the correct template.

   d) From the list of templates, choose *View Extend (creation)* → *extendViewEntity* and choose *Finish*.

3. Edit the element list of the view extension. Remove the placeholder and use code completion to add the booking class field.

   a) In the source code, replace `base_data_source_name.element_name` with **`zz`** and press **`Ctrl + Space`** to invoke code completion.

   b) From the suggestion list, choose *zzclassz## - z##_tritem as Item (column)* to add the append field to the element list of the CDS view extension.

4. To increase readability, define an alias name for the new view element (suggested name: **`ZZClassZ##`**) and activate the data definition.

Caution:
Make sure that the alias name and the field name only differ in uppercase or lowercase!

Why is it crucial that the alias name in the view entity and the field name in the append structure only differ in upper/lower case?

This is important for the draft-handling. Draft-handling only works if the field name in the draft table matches the element name in the view entity.

a) Adjust the code as follows:

```
extend view entity Z##_E_TravelItem with
{
  Item.zzclassz## as ZZClassZ##
}
```

b) Press **Ctrl + F3** to activate the development object.

5. For the flight travel item view on data model level **Z##_R_TravelItem**, create a CDS view extension (suggested name: **Z##X_R_TravelItem_Class**).

   a) In the *Project Explorer*, right-click **Z##_R_TRAVELITEM** and select *New Data Definition*.

   b) Under *Name*, enter **Z##X_R_TravelItem_Class** and under *Description*, enter **Extend Travel Item with Booking Class**. Then choose *Next*.

   c) Confirm the transport request and choose *Next*.

   > Note:
   > Do not choose *Finish* yet or you will not be able to choose the correct template.

   d) From the list of templates, choose *View Extend (creation)* → *extendViewEntity* and choose *Finish*.

6. Edit the element list of the view extension. Remove the placeholder and use code completion to add the booking class field.

   a) In the source code, replace `base_data_source_name.element_name` with **zz** and press **Ctrl + Space** to invoke code completion.

   b) From the suggestion list, choose *ZZClassZ## - Z##_E_TravelItem as _Extension (column)* to add the extension field to the element list of the CDS view extension.

   c) Make sure that the code looks like this:

```
extend view entity Z##_R_TravelItem with
{
  _Extension.ZZClassZ##
}
```

7. Activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

Task 3: Extend the Data Model Projection

Create a CDS view extension to add the booking class field to the CDS view entity on projection level (suggested name: **Z##X_C_TravelItem_Class**). Then create a new metadata extension for the projection view (suggested name: **Z##X_C_TravelItem_Class**). Set the metadata layer to **#CUSTOMER** and add UI metadata for the booking class field.

1. For the travel item view on projection level **Z##_C_TravelItem**, create a CDS view extension (suggested name: **Z##X_C_TravelItem_Class**).

   a) In the *Project Explorer*, right-click **Z##_C_TRAVELITEM** and select *New Data Definition*.

   b) Under *Name*, enter **Z##X_C_TravelItem_Class** and under *Description*, enter **Extend Travel Item with Booking Class**. Then choose *Next*.

   c) Confirm the transport request and choose *Next*.

   Note:
   Do not choose *Finish* yet or you will not be able to choose the correct template.

   d) From the list of templates, choose *View Extend (creation) → extendViewEntity* and choose *Finish*.

2. Edit the element list of the view extension. Remove the placeholder and use code completion to add the booking class field.

   a) In the source code, replace `base_data_source_name.element_name` with **zz** and press **Ctrl + Space** to invoke code completion.

   b) From the suggestion list, choose *ZZClassZ## - Z##_R_TravelItem as Item (column)* to add the extension field to the element list of the CDS view extension.

   c) Make sure that the code looks like this:

```
extend view entity Z##_C_TravelItem with
{
   Item.ZZClassZ##
}
```

3. Assign a value help to the new field, using **/LRN/437_I_ClassStdVH** as the data source.

   a) Adjust the code as follows:

```
extend view entity Z##_C_TravelItem with
{
  @Consumption.valueHelpDefinition: [
    { entity: { name:    '/LRN/437_I_ClassStdVH',
                element: 'ClassID' } } ]
  Item.ZZClassZ##
}
```

4. Activate the data definition.

   a) Press **Ctrl + F3** to activate the development object.

5. For the flight travel item view on projection level **Z##_C_TravelItem**, create a CDS metadata extension (suggested name: **Z##X_C_TRAVELITEM_CLASS**).

   a) In the *Project Explorer*, right-click **Z##_C_TRAVELITEM** and select *New Metadata Extension*.

   b) Under *Name*, enter **Z##X_C_TRAVELITEM_CLASS** and under *Description*, enter **Extend Travel Item with Booking Class**. Then choose *Next*.

   c) Confirm the transport request and choose *Next*.

Note:
Do not choose *Finish* yet or you will not be able to choose the correct template.

d) From the list of templates, choose *Annotate View (creation)* → *annotateView* and choose *Finish*.

6. In the newly created metadata extension, set the metadata layer to **#CUSTOMER**.

    a) In the *@Metadata.layer* annotation, replace the placeholder `layer` with **#CUSTOMER**.

7. Add UI metadata for the booking class field. Make sure, the new field appears on both, the list page and the object page, immediately after the **BookingId** field.

Hint:
View element **BookingId** is annotated with *@UI.lineItem.position: 60* and *@UI.identification.position: 60*.

    a) Adjust the code as follows:

```
@Metadata.layer: #CUSTOMER
annotate view Z##_C_TravelItem with
{
  @UI: {
    lineItem:       [ { position: 65, importance: #HIGH } ],
    identification: [ { position: 65, importance: #HIGH } ] }
  ZZClassZ##;
}
```

8. Activate the metadata extension.

    a) Press **Ctrl + F3** to activate the development object.

9. Refresh the preview of the OData UI service and confirm that the booking class extension field is displayed.

    Why are values that you enter not yet persisted on the database?

    The **unmanaged save** implementation of the business object does not yet know about the extension field.

Unit 8
Exercise 24

Enable and Develop Behavior Extensions

In this exercise, you enable the extension of the standard behavior of your business object for flight travels on both data model and projection level.

Then, you extend the behavior. First, you extend the flight travel items with an *additional save* implementation in which you persist the booking class field in the active table. Then, you add a validation for the booking class field.

Note:
In this exercise, you extend your own business object. It is done for simplicity. To motivate the use of extensions, pretend that once you enabled the extension of the standard behavior, the behavior definitions **Z##_R_TRAVEL** and **Z##_C_TRAVEL** suddenly lie in a different namespace and that you are no longer authorized to make direct changes to these objects.

Note:
In this exercise, replace ## with your group number.

Table 28: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437H_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437H_C_TRAVEL*
CDS Behavior Definition (Model Extension)	*/LRN/ZZX_R_TRAVEL_CLASS*
CDS Behavior Definition (Projection Extension)	*/LRN/ZZX_C_TRAVEL_CLASS*
ABAP Class (Extension Implementation)	*/LRN/BP_ZZX_R_TRAVEL_CLASS*

Task 1: Enable Behavior Extension

Enable the behavior extension for your business object **Z##_R_TRAVEL** and both its entities. Allow the extension with extra validations and determinations (both execution times) and the extension with an additional save implementation. Make sure that it is possible to add the additional validations and determinations to the draft determine action **Prepare**.

1. Open your behavior definition on data model level **Z##_R_TRAVEL** and enable it for extension.
2. Use code completion to add all supported with additions.
3. Perform a syntax check.

Are there extension-related syntax errors?

---

4. Enable the root entity and the child entity of the business object for extension, then repeat the syntax check.

   Is the syntax error gone?

   Are there extension-related syntax warnings?

   ---

5. Enable the draft determine action **`Prepare`** for extension and check that the corresponding syntax warning is gone.
6. Use a pragma to suppress the warnings on missing extensibility of the field mappings.

   Hint:
   You find the relevant pragma in the *Problem Description* for the warning.

   Note:
   Neither the persistent table for flight travels **`Z##_TRAVEL`** nor the structure **`/LRN/437_S_TRITEM`** are extensible. Therefore, extensibility of the mapping is not required. - The mapping for table **`Z##_TRITEM`** could even be deleted, since this table is no longer specified as persistent table and an *unmanaged* save implementation is used instead.

7. Activate the behavior definition.

Task 2: Extend with Additional Save

Make sure the booking class field **`ZZClassZ##`** that you added to the flight travel items in an earlier exercise is written to the corresponding database table field **`ZZCLASSZ##`**. To do so, do not edit the behavior implementation directly. Instead, create a behavior extension for the flight travel business object (suggested name: **`Z##X_R_TRAVEL_CLASS`**) and implement an additional save logic in a behavior extension implementation class.

Note:
In scenarios with managed save, it is sufficient to extend the field mapping to write the extension fields to the append fields in the database table. If the field names only differ in upper/lower case, and if the mapping statement comes with a corresponding addition, even that might not be necessary.

In our scenario, however, the flight travel items are written to the database in an unmanaged save implementation. Here, it would be necessary to add the new fields to the signature of the reused code, that is, the **/LRN/437_S_TRITEM** and **/LRN/437_S_TRITEMX** structures and then extend the field mapping. Because these structures are marked as not extensible, we have to extend the behavior with an additional save.

1. For the behavior definition on data model level **Z##_R_TRAVEL**, create a new behavior extension (suggested name: **Z##X_R_TRAVEL_CLASS**).
2. Extend the child entity **Item** with an additional save definition.
3. Activate the behavior extension.
4. Use a quick fix to generate the extension implementation class (suggested name: **ZBP_##X_R_TRAVEL_CLASS**).
5. Implement the **save_modified** method of the local class **LSC_Z##_R_TRAVEL** in the global class **ZBP_##X_R_TRAVEL_CLASS**. Assigning an inline declared field symbol, loop over the items that require an update of the booking class field.

   Note:
   The items that require an update of the booking class have the value if_abap_behv=>mk-on in the **%control-ZZClassZ##** field.

6. Inside the loop, implement an ABAP SQL UPDATE statement for your database table for active travel items **Z##_TRITEM** to update the booking class field in the affected data set.

   Caution:
   We recommend the UPDATE statement with the SET addition to make sure you do not overwrite any of the other fields.

7. Implement an identical loop over the items that were just created and require an update of the booking class field.

   Caution:
   When the framework executes the additional save logic, it already executed the normal save logic. The normal save logic already created the new entries in the database table. Therefore, do not use the ABAP SQL statement INSERT or you will get duplicate key errors from the database. Instead, use the ABAP SQL statement UPDATE as before.

8. Activate the behavior extension implementation and test the preview of the OData UI service.

Task 3: Extend with Validation

Make sure the user can only enter allowed values in the extension field **ZZClassZ##**. To do so, extend the behavior for the flight travel items with a validation (suggested name: **ZZvalidateClass**) and implement it using the CDS view entity **/LRN/437_I_ClassStdVH** as the data source.

1. Edit the behavior extension on data model level **Z##X_R_TRAVEL_CLASS**. For the flight travel item entity, define a new validation (suggested name: **ZZvalidateClass**) that is triggered during the **create** operation or by changes to the booking class field **ZZClassZ##**.

2. Perform a syntax check.

   Are there extension-related syntax warnings?

   ____

   ____

   ____

3. Extend the draft determine action **Prepare** and add the validation.

   Hint:
   The draft determine action **Prepare** is part of the behavior of the root entity. To address the validation of a child entity, you have to use the alias name of the child entity as a prefix.

4. Activate the behavior extension and use a quick fix to add the validation method to the local handler class.

5. Implement the validation in the newly created local class **LHC_ITEM** inside the global ABAP class **ZBP_##X_R_TRAVEL_CLASS**. As a starting point, copy the implementation of method **/lrn/validateclass** from the local class **LHC_ITEM** in the global ABAP class **/LRN/BP_ZZX_R_TRAVEL_CLASS**.

   Note:
   We copy the model solution to save some time. The copied code is similar to the implementation of the flight date validation. The main difference is the existence check, if the booking class is not initial. But that is something you already implemented in the customer ID validation.

6. In the copied code, find the READ ENTITIES OF statement and replace `/lrn/437h_r_travel` with the business object **Z##_R_TRAVEL** that you extend in this exercise.

7. Everywhere in the implementation, replace the extension field name that is used in the model solution (**/lrn/classzit**) with your own extension field name (**ZZClassZ##**).

Hint:
Use the *Find/Replace* function of the editor.

8. Activate the behavior extension implementation and test the validation in the preview of the OData UI service.

Task 4: Optional: Extend with Side Effect

Extend the behavior definition on data model level **`Z##_R_TRAVEL`** with a determine action (suggested name: **`ZZcheckClass`**) and let it trigger the extension validation **`ZZvalidateClass`**. Then extend the behavior definition with a side effect that executes the determine action after the user entered a value in the *Booking Class* field.

To get the scenario running, the created determine action must finally be exposed to the UI service via an extension of the projection behavior definition.

Note:
Ideally, we would extend the behavior projection with the side effect because it is only relevant for the UI service. However, the extension of behavior projections with side effects is currently not yet supported. Therefore, we add the side effect to the behavior definition on data model level.

1. Edit the behavior extension on data model level **`Z##X_R_TRAVEL_CLASS`**. For the travel item entity, define a new determine action (suggested name: **`ZZcheckClass`**).
2. Specify that no authorizations are required to execute the determine action.
3. Assign the validation of the booking class field **`ZZvalidateClass`** to the determine action.
4. Extend the behavior definition with a side effect that executes the determine action **`ZZcheckClass`** after the user made an entry in the booking class field.
5. Activate the extension of the behavior definition.
6. The last step is to make the created determine action **`ZZcheckClass`** available for the UI service via an extension of the projection behavior definition. To do this, as a first step declare your projection behavior definition **`Z##_C_TRAVEL`** and the **`Z##_C_TravelItem`** entity contained therein as extensible. Don't forget to activate your changes.
7. Now create an extension for the projection behavior definition (suggested name **`Z##X_C_TRAVEL_CLASS`**). In the implementation of the extension, make the determine action **`ZZcheckClass`** available for the Fiori application.
8. Activate the behavior extension on projection level and test the result in a preview of your OData UI service.

Unit 8
Solution 24

Enable and Develop Behavior Extensions

In this exercise, you enable the extension of the standard behavior of your business object for flight travels on both data model and projection level.

Then, you extend the behavior. First, you extend the flight travel items with an *additional save* implementation in which you persist the booking class field in the active table. Then, you add a validation for the booking class field.

Note:
In this exercise, you extend your own business object. It is done for simplicity. To motivate the use of extensions, pretend that once you enabled the extension of the standard behavior, the behavior definitions **`Z##_R_TRAVEL`** and **`Z##_C_TRAVEL`** suddenly lie in a different namespace and that you are no longer authorized to make direct changes to these objects.

Note:
In this exercise, replace ## with your group number.

Table 28: Solution

Repository Object Type	Repository Object ID
CDS Behavior Definition (Model)	*/LRN/437H_R_TRAVEL*
CDS Behavior Definition (Projection)	*/LRN/437H_C_TRAVEL*
CDS Behavior Definition (Model Extension)	*/LRN/ZZX_R_TRAVEL_CLASS*
CDS Behavior Definition (Projection Extension)	*/LRN/ZZX_C_TRAVEL_CLASS*
ABAP Class (Extension Implementation)	*/LRN/BP_ZZX_R_TRAVEL_CLASS*

Task 1: Enable Behavior Extension

Enable the behavior extension for your business object **`Z##_R_TRAVEL`** and both its entities. Allow the extension with extra validations and determinations (both execution times) and the extension with an additional save implementation. Make sure that it is possible to add the additional validations and determinations to the draft determine action **`Prepare`**.

1. Open your behavior definition on data model level **`Z##_R_TRAVEL`** and enable it for extension.

a) Insert the **extensible** addition as follows:

```
managed implementation in class zbp_##_r_travel unique;
strict ( 2 );

with draft;

extensible
{

}
```

2. Use code completion to add all supported **with** additions.
   a) Place the cursor inside the curly brackets after the **extensible** keyword and press **`Ctrl + Space`** to invoke code completion.
   b) From the suggestion list, choose *with additional save; (keyword)*.
   c) Repeat this, choosing all WITH additions until your code looks as follows:

```
managed implementation in class zbp_##_r_travel unique;
strict ( 2 );

with draft;

extensible
{
  with additional save;
  with validations on save;
  with determinations on modify;
  with determinations on save;
}
```

3. Perform a syntax check.

   Are there extension-related syntax errors?

   Yes, there is a new syntax error *The behavior definition allows extensions but does not contain any entities that can be extended*.

   a) Press **`Ctrl + F2`** to perform a syntax check.
   b) Analyze the *Problems* view below the editor.
4. Enable the root entity and the child entity of the business object for extension, then repeat the syntax check.
   a) In the `define behavior for Z##_R_Travel` statement, add the **extensible** addition as follows:

```
define behavior for Z##_R_Travel alias Travel
persistent table z##_travel ##UNMAPPED_FIELD
draft table z##_travel_d
lock master
total etag ChangedAt
authorization master ( instance )
etag master LocChangedAt
early numbering
with additional save
extensible
{
```

b) In the `define behavior for Z##_R_TravelItem` statement, add the **extensible** addition as follows:

```
define behavior for Z##_R_TravelItem alias Item
with unmanaged save
draft table z##_tritem_d
authorization dependent by _Travel
lock dependent by _Travel
etag master LocChangedAt
extensible
{
```

Is the syntax error gone?

Yes, the syntax error is gone.

Are there extension-related syntax warnings?

Yes, there are extension-related syntax warnings: *Determine action "Prepare" should be defined as "extensible" because this behavior definition allows extensions to define determinations or validations "on save". Entity "Z##_R_TRAVEL" is extensible, therefore the "mapping" for "Z##_TRAVEL" should be "corresponding extensible". Entity "Z##_R_TRAVELITEM" is extensible, therefore the "mapping" for "/LRN/437_S_TRITEM" should be "corresponding extensible". Entity "Z##_R_TRAVELITEM" is extensible, therefore the "mapping" for "Z##_TRITEM" should be "corresponding extensible".*

5. Enable the draft determine action **`Prepare`** for extension and check that the corresponding syntax warning is gone.

a) In the `draft determine action Prepare` statement, add the **extensible** addition as follows:

```
draft determine action Prepare
extensible
{
```

6. Use a **pragma** to suppress the warnings on missing extensibility of the field mappings.

Hint:
You find the relevant pragma in the *Problem Description* for the warning.

Note:
Neither the persistent table for flight travels **`Z##_TRAVEL`** nor the structure **`/LRN/437_S_TRITEM`** are extensible. Therefore, extensibility of the mapping is not required. - The mapping for table **`Z##_TRITEM`** could even be deleted, since this table is no longer specified as persistent table and an *unmanaged* save implementation is used instead.

a) In the *Problems* view, right-click the warning message and choose *Problem Description*. The name of the required pragma is `UNEXTENSIBLE_MAPPING`.

b) In the three `mapping for` statements, add the **`##UNEXTENSIBLE_MAPPING`** pragma as follows:

```
mapping for z##_travel corresponding ##unextensible_mapping
  {
```

```
mapping for z##_tritem corresponding ##unextensible_mapping
  {
```

```
mapping for /lrn/437_s_tritem
  control /lrn/437_s_tritemx
  corresponding                      ##unextensible_mapping
  {
```

7. Activate the behavior definition.

   a) Press **`Ctrl + F3`** to activate the development object.

Task 2: Extend with Additional Save

Make sure the booking class field **`ZZClassZ##`** that you added to the flight travel items in an earlier exercise is written to the corresponding database table field **`ZZCLASSZ##`**. To do so, do not edit the behavior implementation directly. Instead, create a behavior extension for the flight travel business object (suggested name: **`Z##X_R_TRAVEL_CLASS`**) and implement an additional save logic in a behavior extension implementation class.

Note:
In scenarios with managed save, it is sufficient to extend the field mapping to write the extension fields to the append fields in the database table. If the field names only differ in upper/lower case, and if the mapping statement comes with a corresponding addition, even that might not be necessary.

In our scenario, however, the flight travel items are written to the database in an unmanaged save implementation. Here, it would be necessary to add the new fields to the signature of the reused code, that is, the **`/LRN/437_S_TRITEM`** and **`/LRN/437_S_TRITEMX`** structures and then extend the field mapping. Because these structures are marked as not extensible, we have to extend the behavior with an additional save.

1. For the behavior definition on data model level **`Z##_R_TRAVEL`**, create a new behavior extension (suggested name: **`Z##X_R_TRAVEL_CLASS`**).

   a) In the *Project Explorer*, expand *Favorite Packages*.

   b) Under your own package **`ZS4D437_##`**, expand *Core Data Services* → *Behavior Definitions*.

   c) Right-click *Z##_R_TRAVEL* and select *New Behavior Extension*.

   d) Under *Name*, enter **`Z##X_R_TRAVEL_CLASS`** and under *Description*, enter **`Extend Travel Item with Booking Class`**. Choose *Next*.

      Note:
      We did not create a BO Interface for our business object. Therefore, leave the *BO Interface* field empty.

   e) Assign the new development object to a transport request and choose *Finish*.

2. Extend the child entity **Item** with an additional save definition.

   a) Adjust the code as follows:

```
extend behavior for Item
with additional save
{
}
```

3. Activate the behavior extension.

   a) Press **Ctrl + F3** to activate the development object.

4. Use a quick fix to generate the extension implementation class (suggested name: **ZBP_##X_R_TRAVEL_CLASS**).

   a) In the `extension implementation in class` statement, place the cursor on `zbp_##x_r_travel_class` and press **Ctrl + 1** to invoke the quick assist proposals. Alternatively, you can choose the warning icon with light bulb on the left of this code row.

   b) From the list of available quick fixes, choose *Create behavior implementation class zbp_##x_r_travel_class*.

   c) Confirm the package and description and choose *Next*.

   d) Assign the object to a transport request and choose *Finish*.

5. Implement the **save_modified** method of the local class **LSC_Z##_R_TRAVEL** in the global class **ZBP_##X_R_TRAVEL_CLASS**. Assigning an inline declared field symbol, loop over the items that require an update of the booking class field.

   Note:
   The items that require an update of the booking class have the value if_abap_behv=>mk-on in the **%control-ZZClassZ##** field.

   a) Adjust the code as follows:

```
METHOD save_modified.

  LOOP AT update-item ASSIGNING FIELD-SYMBOL(<item>)
    WHERE %control-ZZClassZ## = if_abap_behv=>mk-on.

  ENDLOOP.

ENDMETHOD.
```

6. Inside the loop, implement an ABAP SQL UPDATE statement for your database table for active travel items **Z##_TRITEM** to update the booking class field in the affected data set.

   Caution:
   We recommend the UPDATE statement with the SET addition to make sure you do not overwrite any of the other fields.

   a) Adjust the code as follows:

```
METHOD save_modified.
```

```
  LOOP AT update-item ASSIGNING FIELD-SYMBOL(<item>)
    WHERE %control-ZZClassZ## = if_abap_behv=>mk-on.

    UPDATE z##_tritem
      SET zzclassz## = @<item>-ZZClassZ##
      WHERE item_uuid  = @<item>-ItemUuid.

  ENDLOOP.

ENDMETHOD.
```

7. Implement an identical loop over the items that were just created and require an update of the booking class field.

> Caution:
> When the framework executes the additional save logic, it already executed the normal save logic. The normal save logic already created the new entries in the database table. Therefore, do not use the ABAP SQL statement INSERT or you will get duplicate key errors from the database. Instead, use the ABAP SQL statement UPDATE as before.

a) Adjust the code as follows:

```
METHOD save_modified.

  LOOP AT update-item ASSIGNING FIELD-SYMBOL(<item>)
    WHERE %control-ZZClassZ## = if_abap_behv=>mk-on.

    UPDATE z##_tritem
      SET zzclassz## = @<item>-ZZClassZ##
      WHERE item_uuid  = @<item>-ItemUuid.

  ENDLOOP.

  LOOP AT create-item ASSIGNING <item>
    WHERE %control-ZZClassZ## = if_abap_behv=>mk-on.

    UPDATE z##_tritem
      SET zzclassz## = @<item>-ZZClassZ##
      WHERE item_uuid  = @<item>-ItemUuid.

  ENDLOOP.

ENDMETHOD.
```

> Hint:
> Alternatively, you can first merge the lines of **`update-item`** and **`create-item`** in one internal table before you do the loop. This is possible, because **`update`** and **`create`** are typed with the identical derived type. The code could then, for example, look as follows:

```
METHOD save_modified.
  DATA(items) = update-item.
  APPEND LINES OF create-item TO items.

  LOOP AT items ASSIGNING FIELD-SYMBOL(<item>)
```

```
    WHERE %control-ZZClassZ## = if_abap_behv=>mk-on.

    UPDATE z##_tritem
      SET zzclassz## = @<item>-ZZClassZ##
      WHERE item_uuid = @<item>-ItemUuid.

  ENDLOOP.
ENDMETHOD.
```

8. Activate the behavior extension implementation and test the preview of the OData UI service.

   a) Press **`Ctrl + F3`** to activate the development object.

   b) Reopen the preview of the OData UI service and choose *Go* to display the list of flight travels.

   c) Open the object page for a flight travel that lies in the future and choose *Edit*.

   d) Choose *Create* to create a new flight travel item.

   e) Enter valid values for all flight travel item fields and save the data.

   Result

   The value that you entered in the *Booking Class* field is stored in the **`Z##_TRITEM`** database table.

Task 3: Extend with Validation

Make sure the user can only enter allowed values in the extension field **`ZZClassZ##`**. To do so, extend the behavior for the flight travel items with a validation (suggested name: **`ZZvalidateClass`**) and implement it using the CDS view entity **`/LRN/437_I_ClassStdVH`** as the data source.

1. Edit the behavior extension on data model level **`Z##X_R_TRAVEL_CLASS`**. For the flight travel item entity, define a new validation (suggested name: **`ZZvalidateClass`**) that is triggered during the **`create`** operation or by changes to the booking class field **`ZZClassZ##`**.

   a) Adjust the code as follows:

```
extend behavior for Item
with additional save
{
  validation ZZvalidateClass on save
  { create;
    field ZZClassZ##;
  }
}
```

2. Perform a syntax check.

   Are there extension-related syntax warnings?

   Yes, there are extension-related warnings: *Validation "ZZvalidateClass" is not assigned to any determine action (not even "Prepare"). Validation Z##_R_TRAVELITEM ~ ZZVALIDATECLASS is not implemented.*

   a) Press **`Ctrl + F2`** to perform a syntax check.

   b) Analyze the *Problems* view below the editor.

3. Extend the draft determine action **Prepare** and add the validation.

Hint:
The draft determine action **Prepare** is part of the behavior of the root entity. To address the validation of a child entity, you have to use the alias name of the child entity as a prefix.

a) Adjust the code as follows:

```
extend behavior for Travel
{
  extend draft determine action Prepare
  {
    validation Item~ZZvalidateClass;
  }
}
```

4. Activate the behavior extension and use a quick fix to add the validation method to the local handler class.

a) Press **Ctrl + F3** to activate the development object.

b) Place the cursor on the name of the validation `ZZvalidateClass` and press **Ctrl + 1** to invoke the quick assist proposals.

c) From the list of available quick fixes, choose *Add method for validation zzvalidateclass of entity z##_r_travelitem in new local class.*

5. Implement the validation in the newly created local class **LHC_ITEM** inside the global ABAP class **ZBP_##X_R_TRAVEL_CLASS**. As a starting point, copy the implementation of method **/lrn/validateclass** from the local class **LHC_ITEM** in the global ABAP class **/LRN/BP_ZZX_R_TRAVEL_CLASS**.

Note:
We copy the model solution to save some time. The copied code is similar to the implementation of the flight date validation. The main difference is the existence check, if the booking class is not initial. But that is something you already implemented in the customer ID validation.

a) Choose **Ctrl + Shift + A**, enter **/LRN/BP_ZZX_R_TRAVEL_CLASS** , and choose *OK*.

b) Navigate to the *Local Types* tab and scroll down to the `METHOD /lrn/validateclass.` statement.

c) Select the entire code between`METHOD /lrn/validateclass.` and the following `ENDMETHOD.` statement and press **Ctrl + C**.

d) Return to your own class **ZBP_##X_R_TRAVEL_CLASS**. In the local class **LHC_ITEM** contained therein, place the cursor between `METHOD ZZvalidateClass.` and the following `ENDMETHOD.` and choose **Ctrl + V**.

6. In the copied code, find the READ ENTITIES OF statement and replace `/lrn/437h_r_travel` with the business object **Z##_R_TRAVEL** that you extend in this exercise.

a) Adjust the code as follows:

```
READ ENTITIES OF Z##_R_Travel IN LOCAL MODE
  ENTITY Item
    FIELDS ( AgencyId TravelId /lrn/classzit )
    WITH CORRESPONDING #( keys )
    RESULT DATA(items).
```

7. Everywhere in the implementation, replace the extension field name that is used in the model solution (**`/lrn/classzit`**) with your own extension field name (**`ZZClassZ##`**).

> Hint:
> Use the *Find/Replace* function of the editor.

a) Press **`Ctrl + F`** to invoke the search function.

b) Make sure that the *Find* field contains **`/lrn/classzit`**.

c) In the *Replace* field, enter **`ZZClassZ##`**, and choose *Replace all*.

8. Activate the behavior extension implementation and test the validation in the preview of the OData UI service.

a) Press **`Ctrl + F3`** to activate the development object.

b) Reopen the preview of the OData UI service and choose *Go* to display the list of flight travels.

c) Open the object page for a flight travel that lies in the future and choose *Edit*.

d) Choose *Create* to create a new flight travel item.

e) Enter an invalid value in the *Booking Class* field, for example **`X`**, and save the data.

Result

You see the error message *Booking Class X does not exist*.

Task 4: Optional: Extend with Side Effect

Extend the behavior definition on data model level **`Z##_R_TRAVEL`** with a determine action (suggested name: **`ZZcheckClass`**) and let it trigger the extension validation **`ZZvalidateClass`**. Then extend the behavior definition with a side effect that executes the determine action after the user entered a value in the *Booking Class* field.

To get the scenario running, the created determine action must finally be exposed to the UI service via an extension of the projection behavior definition.

> Note:
> Ideally, we would extend the behavior projection with the side effect because it is only relevant for the UI service. However, the extension of behavior projections with side effects is currently not yet supported. Therefore, we add the side effect to the behavior definition on data model level.

1. Edit the behavior extension on data model level **`Z##X_R_TRAVEL_CLASS`**. For the travel item entity, define a new determine action (suggested name: **`ZZcheckClass`**).

a) Adjust the code as follows:

```
extend behavior for Item
with additional save
{
  validation ZZvalidateClass on save
  { create;
    field ZZClassZ##;
  }

  determine action ZZcheckClass
  {

  }
}
```

2. Specify that no authorizations are required to execute the determine action.

   a) Adjust the code as follows:

```
extend behavior for Item
with additional save
{
  validation ZZvalidateClass on save
  { create;
    field ZZClassZ##;
  }

  determine action ( authorization : none ) ZZcheckClass
  {

  }
}
```

3. Assign the validation of the booking class field **`ZZvalidateClass`** to the determine action.

   a) Adjust the code as follows:

```
extend behavior for Item
with additional save
{
  validation ZZvalidateClass on save
  { create;
    field ZZClassZ##;
  }

  determine action ( authorization : none ) ZZcheckClass
  {
    validation ZZvalidateClass;
  }
}
```

4. Extend the behavior definition with a side effect that executes the determine action **`ZZcheckClass`** after the user made an entry in the booking class field.

   a) Adjust the code as follows:

```
  determine action ( authorization : none ) ZZcheckClass
  {
    validation ZZvalidateClass;
  }

  side effects
  {
    determine action ZZcheckClass
    executed on field ZZClassZ##
    affects messages;
  }
}
```

5. Activate the extension of the behavior definition.
   a) Press **`Ctrl + F3`** to activate the development object.

6. The last step is to make the created determine action **`ZZcheckClass`** available for the UI service via an extension of the projection behavior definition. To do this, as a first step declare your projection behavior definition **`Z##_C_TRAVEL`** and the **`Z##_C_TravelItem`** entity contained therein as extensible. Don't forget to activate your changes.
   a) Edit your projection behavior definition **`Z##_C_TRAVEL`**.

   b) Adjust your code as follows:

```
projection;
strict ( 2 );

use draft;
use side effects;

extensible;

[...]

define behavior for Z## _C_TravelItem
use etag
extensible
{
```

   c) Press **`Ctrl + F3`** to activate the development object.

7. Now create an extension for the projection behavior definition (suggested name **`Z##X_C_TRAVEL_CLASS`**). In the implementation of the extension, make the determine action **`ZZcheckClass`** available for the Fiori application.
   a) In the *Project Explorer*, open the context menu for your projection behavior definition **`Z##_C_TRAVEL`**.

   b) Choose *New Behavior Extension*.

   c) In the *Name* field, enter **`Z##X_C_TRAVEL_CLASS`** and in the *Description* field, enter **`Extend Travel Item with Booking Class`**.

   d) Choose *Next*.

   e) Select a transport request and choose *Finish*.

   f) Adjust the implementation of the extension as follows:

```
extension for projection;

extend behavior for Z##_C_TravelItem
{
  use action ZZcheckClass;
}
```

8. Activate the behavior extension on projection level and test the result in a preview of your OData UI service.
   a) Press **`Ctrl + F3`** to activate the development object.

b) Reopen the preview of the OData UI service and choose *Go* to display the list of flight travels.

c) Open the object page for a flight travel that lies in the future and choose *Edit*.

d) Choose *Create* to create a new flight travel item.

e) Enter an invalid value in the *Booking Class* field, for example **`X`**. Press **`Tab`** to leave the input field.

Result

This triggers the determine action and you see the error message *Booking Class X does not exist*.