SAP ABAP RAP

SAP ABAP RAP Data Modeling Flow
Enhanced Custom Logic Flow

Data Modeling Flow

1. [Define TABLE Structure]
   - DEFINE TABLE (TableName)
   - Structure for data models.
   - Source of raw data.
2. [Define CDS VIEW Entity]
   - Define view ZI_CDS_VIEW_NAME as select from
   - Data model from database tables.
   - Implement data retrieval logic.
3. [CDS Views with Associations]
   - Association to child views.
   - Example: association [..] to ZI_CHILD_VIEW on $projection.JOINFIELDS = _
   - Facilitate joined data representation
4. [Behavior Definitions for CDS Views]
   - Define CRUD operations for CDS entities.
   - Implement validations/ action triggers.
5. [Service Definition for Data Modeling]
   - Expose CDS Views as OData services.
   - Define accessible entities/ operations
6. [Service Binding for Data Modeling]
   - Create binding for data services.
   - Choose OData protocol version.
   - Activate and test service
7. [Fiori UI Application for Data Modeling]

   - Develop Fiori apps for data display
   - Bind data OData services to UI
   - Implement UI logic/configure elements

Enhanced Custom Logic Flow

1. [Define ABAP Class with Interface]
   - Class Definition: ZCL_CLASS_ NAME.
   - Interface: IF_RAP_QUERY_ PROVIDER
2. [Implement Interface Methods]
   - IF_RAP_QUERY_PROVIDER~S ELECT implementation.
   - Manage paging and sorting:
     - DATA(lv_top) = io_request->get_paging()-> get_page_size().
     - DATA(lv_skip) = io_request->get_paging()-> get_offset().
     - DATA(lo_requested_fields) = io_request->get_ requested _elements()
     - DATA(sort_order) = io_request->get_sort _elements().
   - Retrieve and handle business data.
   - Set response details:
     - io_response- Set _total _> number_of_records lines(business_data)).
     - io_response- >set_data(business_data).
3. [Define Custom Entity and Query Annotation]

   - Define custom entity: ZCE_CDS_VIEW_NAME
   - Annotate class with @ObjectModel.query.implem entedBy ABAP:ZCL_CLASS_NAME'
4. [OData Service Consumption & Setup]
   - Create HTTP client and OData service proxy
   - Execute and handle OData requests.
5. [Service Definition and Binding]
   - Expose custom logic as OData
   - Create and activate service binding

Get to Know RAP: Introduction

1 33 8,945

Happy New Year! I hope you all had a wonderful holiday season with friends, family, and colleagues.

As an ABAP developer, we are always looking for new and better ways to create robust and scalable applications. Recently, I have been exploring the ABAP RESTful Application Programming Model (RAP) and I am impressed by its capabilities and ease of use.

If you have been following up on the "Getting Started with Your SAP ABAP journey" blog post and are now looking to familiarize yourself with the ABAP RESTful Application Programming Model (RAP), this series is for you. In this blog series, I will share my experience of getting started with RAP by developing a Fiori App using RAP, among other things.

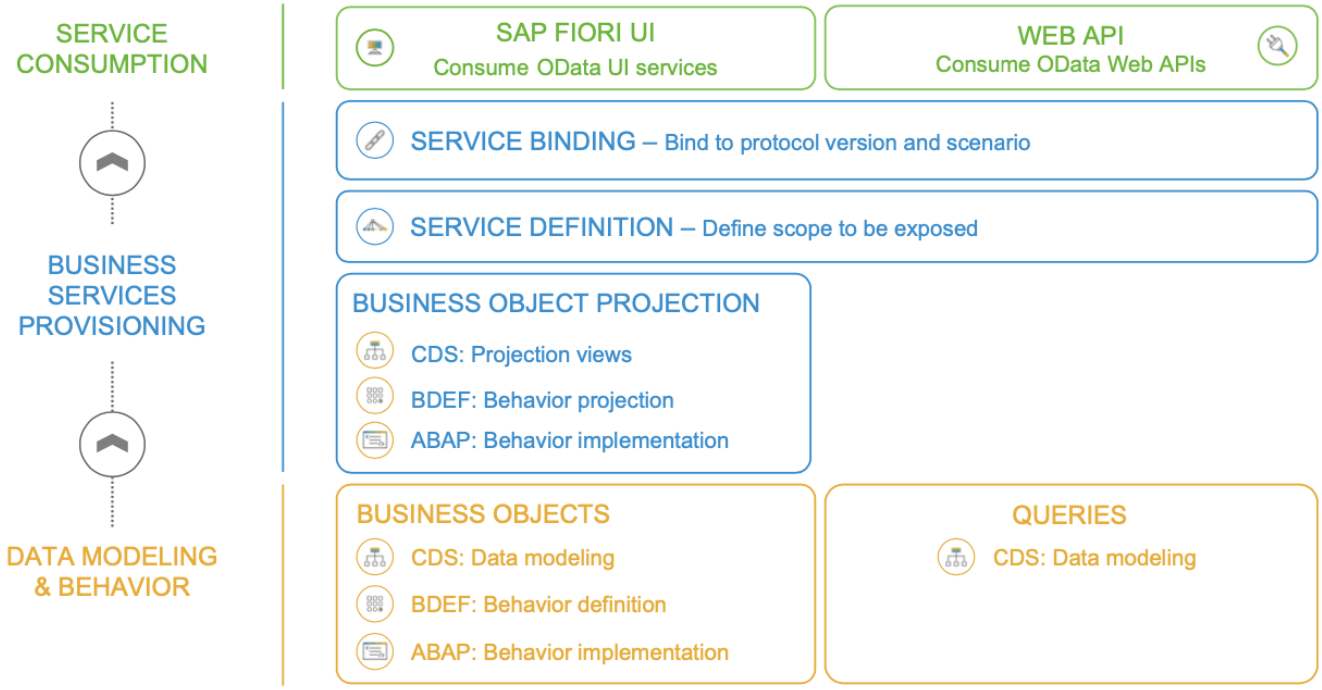
This blog post will be the first in this series of "Get to Know RAP". I will go over a few concepts and provide you with helpful resources I believe will be beneficial in learning the fundamentals before we dive into the exercises.

Introduction:

What is RAP?

Before we begin, let's start by get some background information about REST and the ABAP RESTful Application Programming Model. **REST** (Representational State Transfer) is an architectural style for designing web services.
The **ABAP RESTful Application Programming Model** (RAP) defines the architecture for efficient end-to-end development of intrinsically SAP HANA-optimized OData services, such as Fiori apps. It offers developers an efficient way to build enterprise-ready, SAP HANA-optimized, OData-based Fiori UI services and Web APIs.

Architecture Overview



RAP Big Picture

The diagram above illustrates the major development artifacts involved in creating an OData service using the ABAP RESTful Programming Model and follows a development flow that takes a bottom-up approach.

The main development tasks can be categorized in three layers:

Data Modeling and Behavior

- The data model comprises the description of the different entities involved in a business scenario, for example traveland booking, and their relationships, for example the parent-child relationship between travel and booking.
- CDS is the cornerstone of the ABAP RAP
- The behavior of the data model determines the actions that can be performed on it, including the ability to create, update, or delete data. This can be implemented by using ABAP

Business Services Provisioning

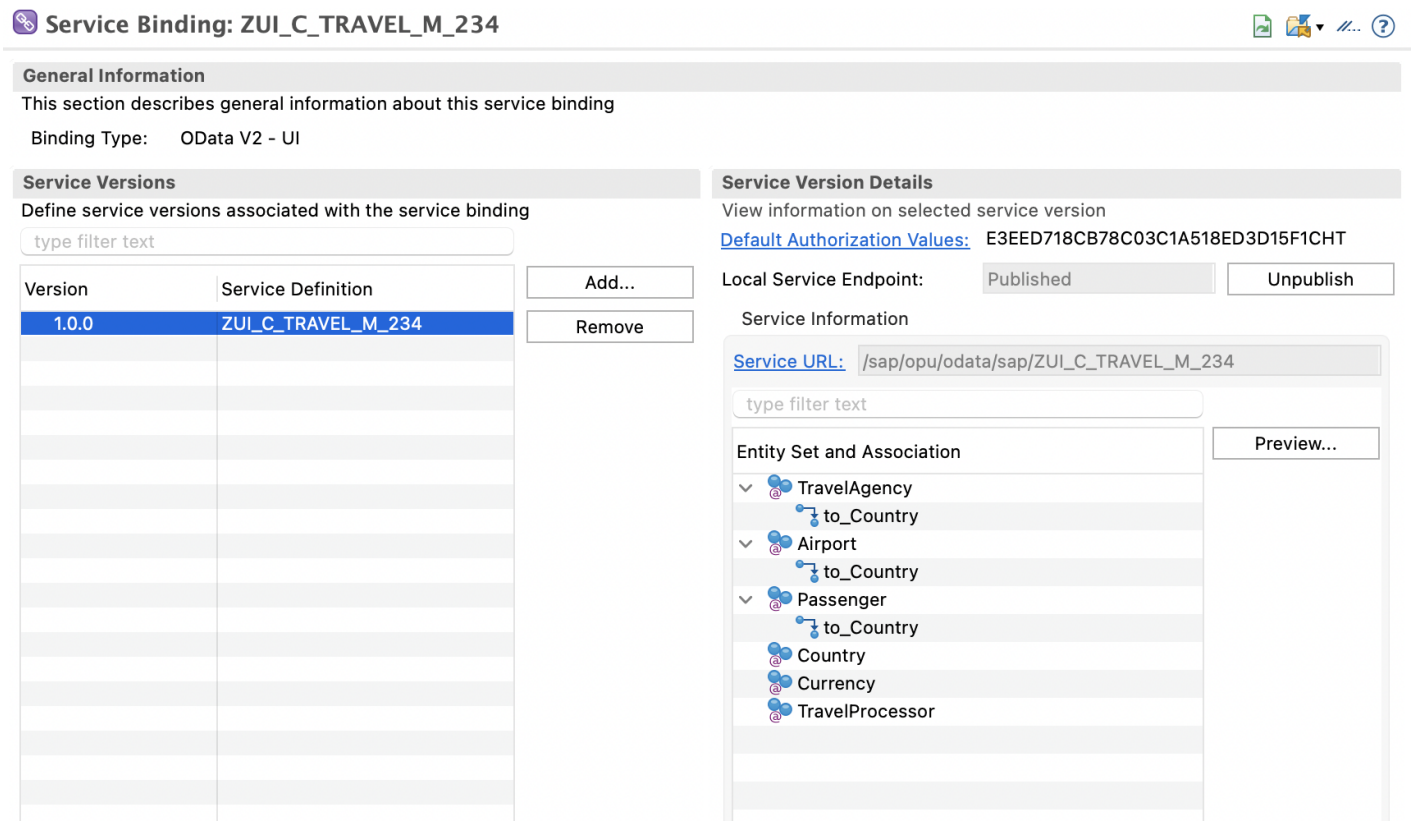
- With the **service definition** you can define which data is exposed as a business service in your travel booking application

```
@EndUserText.label: 'Service definition for travel'
define service ZUI_C_TRAVEL_M_234 {
  expose ZC_TRAVEL_M_000 as TravelProcessor;
  expose /DMO/I_Customer as Passenger;
  expose /DMO/I_Agency as TravelAgency;
  expose /DMO/I_Airport as Airport;
  expose I_Currency as Currency;
  expose I_Country as Country;
}
```

Example

Service Consumption

- Can be made available as a UI service, which is utilized by SAP Fiori Elements app.
- ODATA service that is exposed as Web API does not contain any UI-specific information in its metadata
- It acts as a public interface that can be accessed by any OData client, such as another OData service. For example, you can consume a Web API from another OData service.



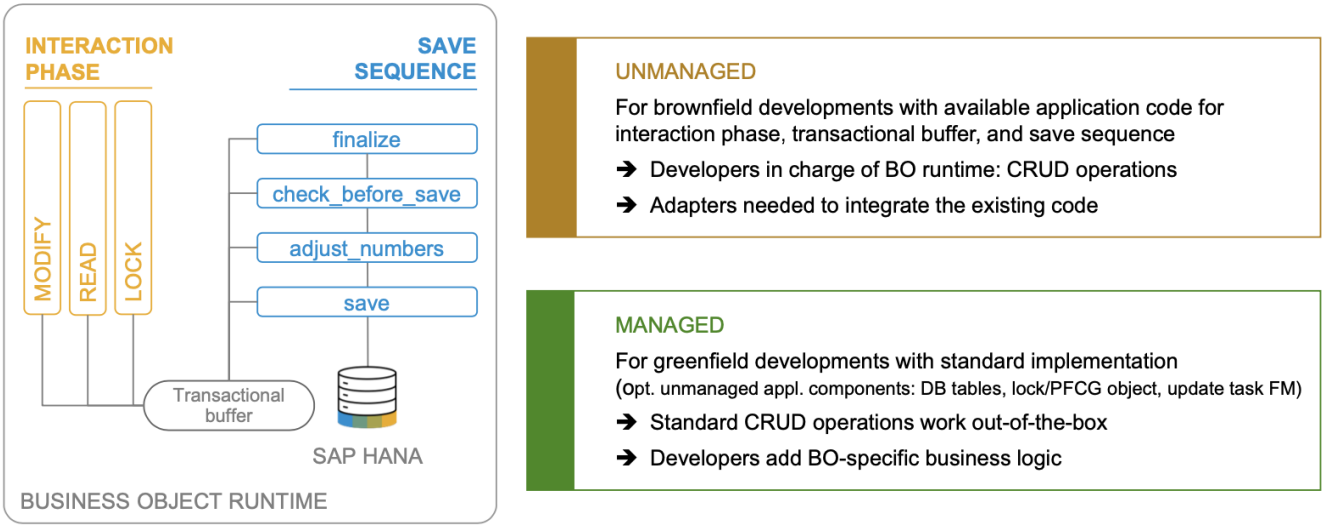
Example

We will go over all of these as we continue the series with detailed examples.

Need To Know:

ABAP RAP provides a programming model for the efficient development of from scratch (greenfield developments) or by integrating legacy code (brownfield developments).

The development of OData-based services starting from scratch, also known as the greenfield development, is supported with the so-called managed implementation type, whereas the brownfield development – i.e., based on existing code – is supported with the so-called unmanaged implementation type.



RAP – Business Object runtime implementation types

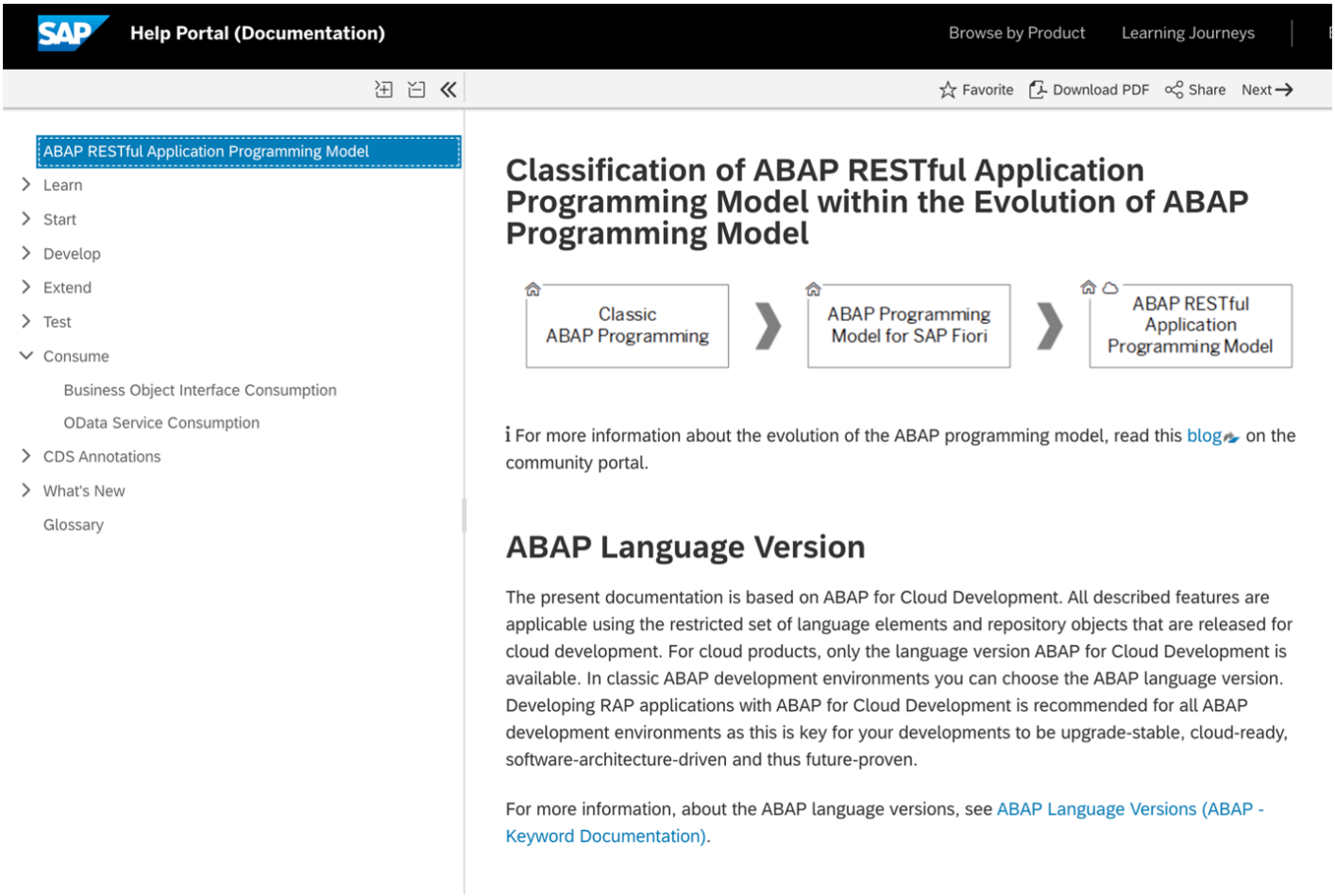
This also provides an efficient development of SAP Fiori apps and Web APIs, which we will be focusing on in this series.

Read this blog post to learn more: https://blogs.sap.com/2021/10/18/modernization-with-rap/

SAP Help Portal

The ABAP RESTful Application Programming Model comes with a set of development guides and best practices to assist the developer onboarding in the SAP Help Portal.

You will find this be helpful as it allows you to understand the concepts behind RAP and also an overview of the prefixing and suffixing guidelines.



SAP Help Portal

What's Next:

The RAP model can be used in a variety of scenarios to build applications. For our first RAP application we will develop a Fiori App using the ABAP RESTful Application Programming Model using steps spanning across all 3 layers mentioned above. During the development process, we will start by defining our data model using CDS.

Prerequisites:

- You have installed ABAP Development Tools (ADT).
- You need an SAP BTP, ABAP environment trial user or a license.
- Have Basic Knowledge of ABAP Core Data Services (CDS)
- Have Basic Knowledge of ABAP Objects

In the next blog post of this series, we will create table persistence and generate data for the application. We will create an ABAP package, database table for storing data and create an ABAP class to generate data.

Define Data Model – Part 1

This is the second post in this series. Here is the previous blog post: [Get to Know RAP: Introduction]

This blog post also follows up with a video that walks you through the tutorial that I will be going over.

Introduction

As mentioned in the previous blog post our goal is to develop a Fiori app using the ABAP RESTful Application Programming Model. We will be using steps spanning across all 3 layers (**Data Modeling and Behavior, Business Service**, **Service Consumption**).

We will start from the **Data Modeling and Behavior** and work our way up. Since our overall goal is to create a completely new transactional app for a Fiori Elements UI following the greenfield approach, we will create everything from scratch. With most new developments, you will always rely on the managed approach since the entire administration of the object is taken over for you by the RAP Framework.

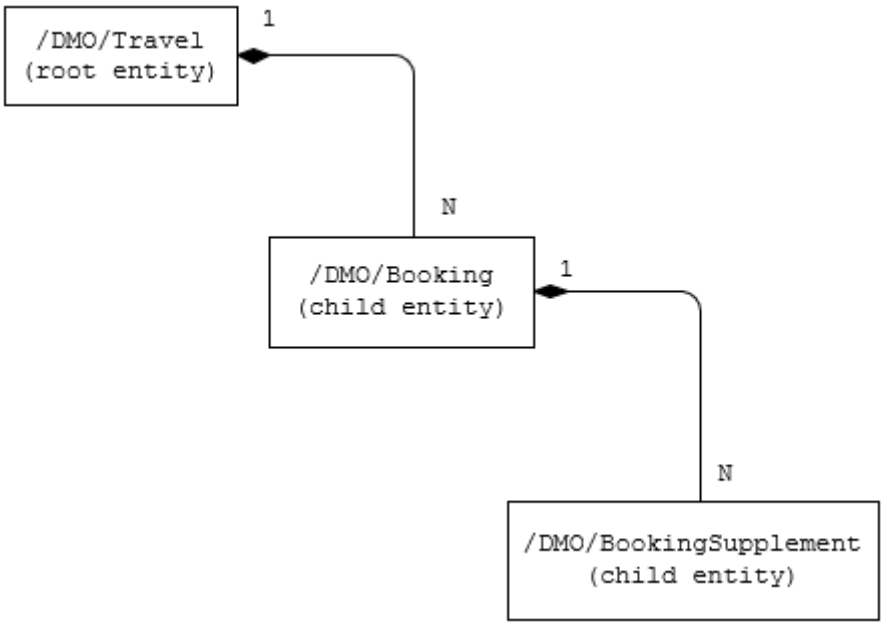
To begin, I will be following along the Develop a Fiori App Using the ABAP RESTful Application Programming Model (Managed Scenario) group tutorials to develop a travel booking SAP Fiori application. For this blog post I will provide you my takeaway on creating table persistence and generating data for it.

Data Model and Business Object

As we previously mentioned before the data model consists of the entities involved in a business scenario, such as TRAVEL and BOOKING, and how they relate to each other. The ABAP RESTful Programming Model uses CDS to organize and define the data model. CDS entities are the fundamental building blocks for your application.

The travel business object consists of three entities that are structured in a hierarchy tree. Every entity in the BO-composition tree is modeled with a CDS view entity. The business entities we are going to work on in our present scenarios are:

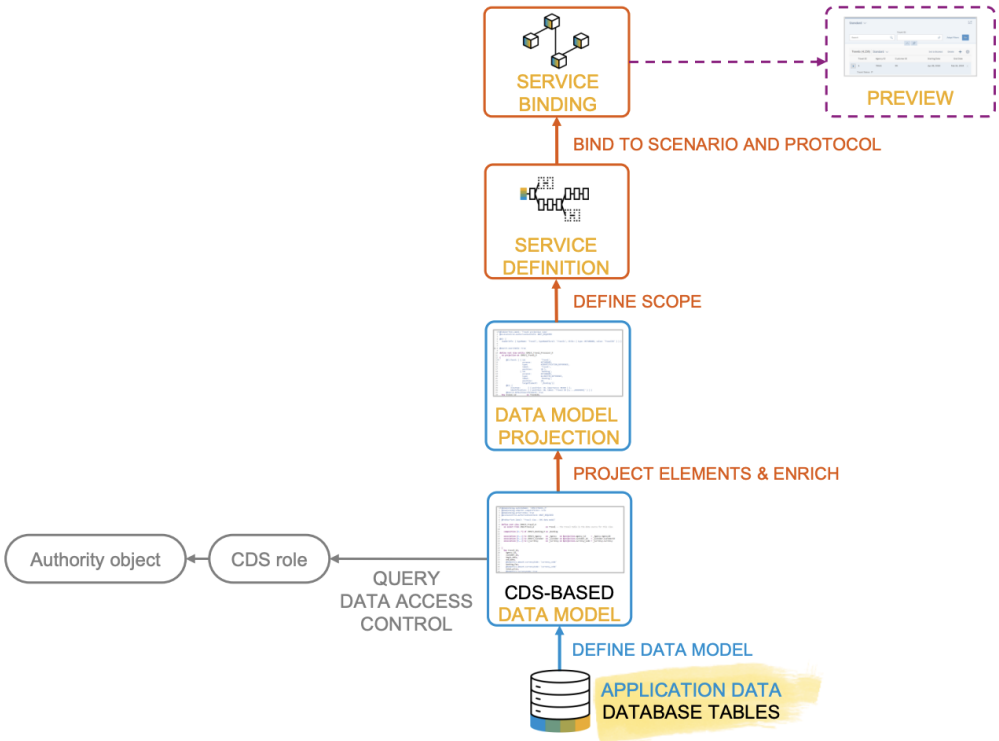
- **Travel**: The root entity defines general travel data, like the agency ID or customer ID, the status of the travel booking and the price of the travel.
- **Booking**: The booking entity defines flight and booking data, the customer data, the customer ID for whom the flight is booked and Travel ID to which the booking belongs.
- **BookingSupplement**: The booking supplement entity presents additional bookable supplement for a certain booking. These are meals and beverages for the flights.



Editable Entities of the Business Object

Creating Table Persistence

An overview of the typical development workflow in RAP is shown below. The development of our app will also look like this.



In this exercise we will first create a database table to store the **travel** data. We will also re-use from the already existing demo content from the ABAP Flight Reference Scenario (**Agency**, **Customer** and **Flight**), as well as a few more.

> **Note:** In the ABAP Flight Reference Scenario for the persistent database tables, we use the prefix A_ to indicate the active persistence. For detailed information, see Naming Conventions for Development Objects.

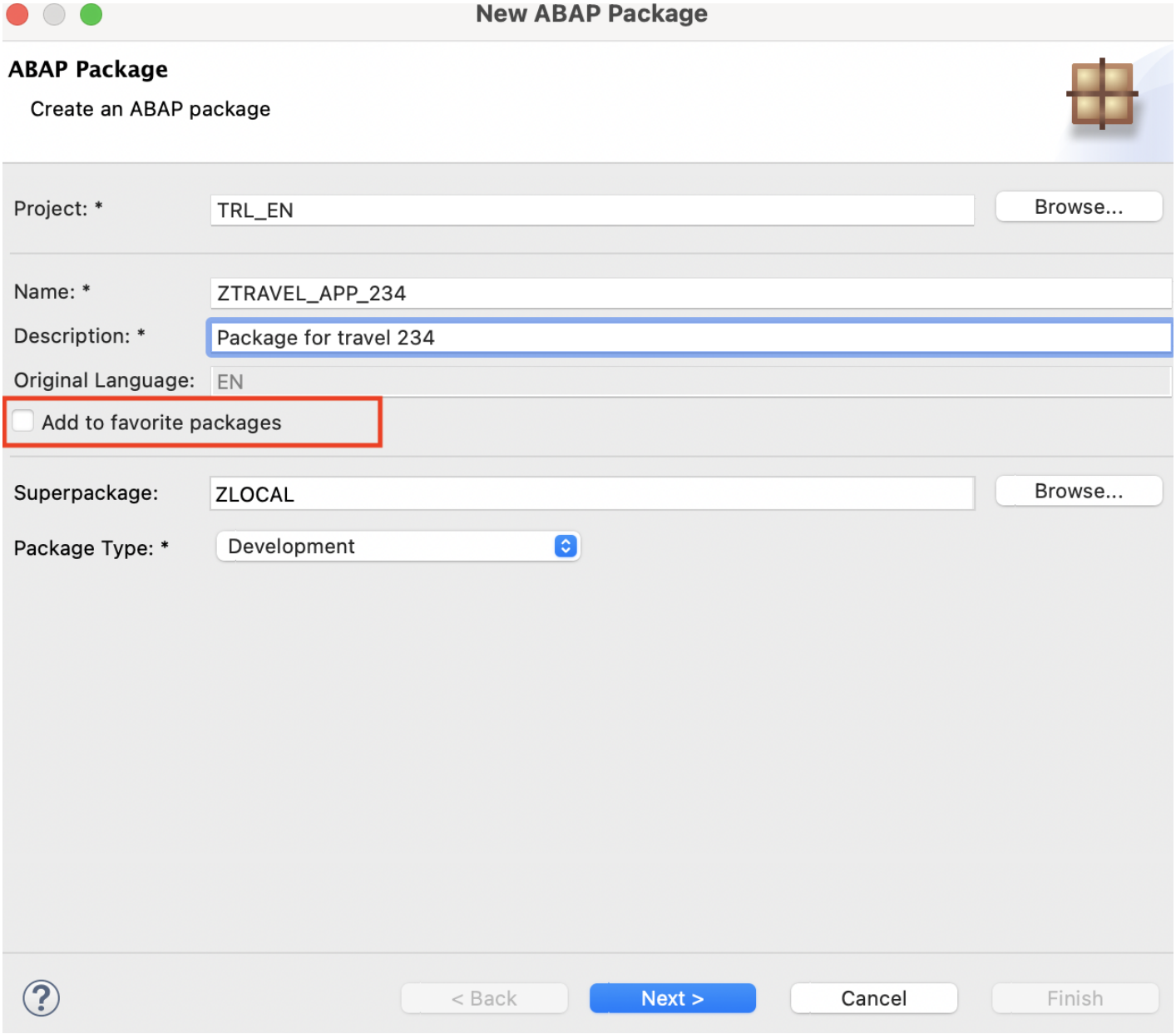
Before we begin, it is important to know the prerequisites:

Prerequisites

- You need an SAP BTP, ABAP environment trial user or a license.
- You have downloaded and installed the latest ABAP Development Tools (ADT).
- **ABAP Flight Reference Scenario** must be available in your ABAP system. You can download the complete reference scenario from **GitHub**: Downloading the ABAP Flight Reference Scenario.

Create ABAP Package

The first thing that needed to be done was to create an ABAP package, this is quite simple. Simply just log into the SAP system. From there, I select the option to create new package. The system prompts me to provide a name and description for the package. Don't forget to check **Add to favorite packages** so that you can access it easily. You don't have to go looking for it in ZLOCAL where you might run into the risk of you accidentally choosing someone else's package.



ABAP Package

It is important to ensure that the package you are creating doesn't already exist in ZLOCAL. This is because the trial version of the ABAP environment is shared and there are many packages available. So, make sure you have a unique name and make sure to add a number that is currently free. I had this issue at first when I was creating my package for the first time, I then found out it was already taken by someone else.

For this tutorial I decided to go with the suffix 234 for the various objects that I'm going to create. After providing the package name and description, I save the package, and it's ready to use. Overall, this step is straightforward and easy to follow.

Database Table

Moving on from creating the package, I then went and created a database table to store the **travel** data that was provided for us in the tutorial. The tutorial will provide you the steps on creating the table. When creating the database table, it important to add in your suffix to the code sample. It took me a while to realize why my code was not activating, only to find out it was because the suffix was not added to the code. I would like to provide you some code explanation, so you are familiar with the steps.

The table that I created will open in the editor and the client field will be added automatically because it is a client-specific table.

```
@EndUserText.label : 'Database table for travel data 000'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table ztravel_000 {
  key client      : abap.clnt not null;
  key mykey       : sysuuid_x16 not null;
  travel_id       : /dmo/travel_id;
  agency_id       : /dmo/agency_id;
  customer_id     : /dmo/customer_id;
  begin_date      : /dmo/begin_date;
  end_date        : /dmo/end_date;
  @Semantics.amount.currencyCode : 'ztravel_000.currency_code'
  booking_fee     : /dmo/booking_fee;
  @Semantics.amount.currencyCode : 'ztravel_000.currency_code'
  total_price     : /dmo/total_price;
  currency_code   : /dmo/currency_code;
  description     : /dmo/description;
  overall_status  : /dmo/overall_status;
  created_by      : syuname;
  created_at      : timestampl;
  last_changed_by : syuname;
  last_changed_at : timestampl;

}
```

Don’t forget to replace all the 000 with your custom number

Code Explanation:

- The table now consists of the key fields, client and mykey. It also includes table fields, such as human-readable TRAVEL_ID, the AGENCY_ID, the CUSTOMER_ID, the TOTAL_PRICE, and the OVERALL_STATUS

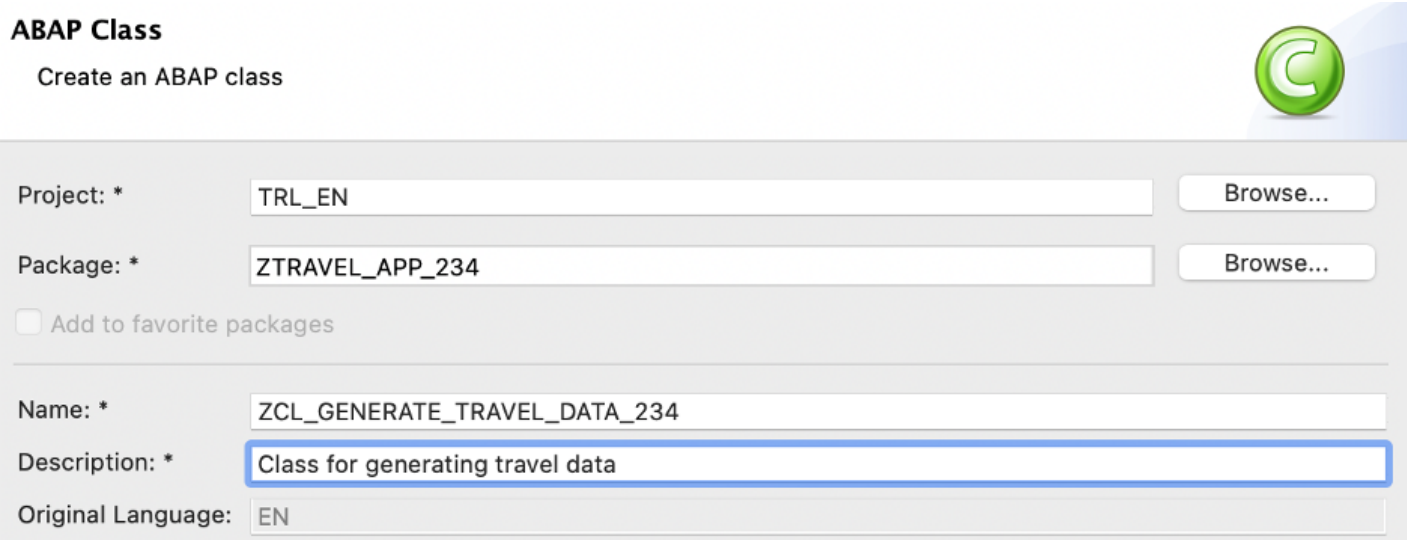
- Some of the objects have /DMO. This is because they belong to the ABAP Flight Reference Scenario examples. If you are using the trial, it is already pre-installed for you.

- We have some standard administration data, such as CREATED_BY, CREATED_AT

- The table field CURRENCY_CODE is specified as the reference field for the amount fields BOOKING_FEE and TOTAL_PRICE using @Semantics.amount.currencyCode

Create ABAP class

The travel list report currently has no data, so we will create an ABAP class to fill the database table with some demo data.



We will replace the code and paste in the code snippet we have been provided. I would like to provide you some code explanation, so you are familiar with the steps.

```
CLASS zcl_generate_travel_data_000 DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    INTERFACES if_oo_adt_classrun.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.



CLASS zcl_generate_travel_data_000 IMPLEMENTATION.
  METHOD if_oo_adt_classrun~main.
```

```
    DATA itab TYPE TABLE OF ztravel_000.

*   fill internal travel table (itab)
    itab = VALUE #(
      ( mykey = '02D5290E594C1EDA93815057FD946624' travel_id = '00000022' agency_i
        description = 'mv' overall_status = 'A' created_by = 'MUSTERMANN' created_
      ( mykey = '02D5290E594C1EDA93815C50CD7AE62A' travel_id = '00000106' agency_i
        description = 'Enter your comments here' overall_status = 'A' created_by =
      ( mykey = '02D5290E594C1EDA93858EED2DA2EB0B' travel_id = '00000103' agency_i
        description = 'Enter your comments here' overall_status = 'X' created_by =
    ).

*   delete existing entries in the database table
    DELETE FROM ztravel_000.

*   insert the new table entries
    INSERT ztravel_000 FROM TABLE @itab.

*   output the result as a console message
    out->write( |{ sy-dbcnt } travel entries inserted successfully!| ).

  ENDMETHOD.
ENDCLASS.
```

Code Explanation

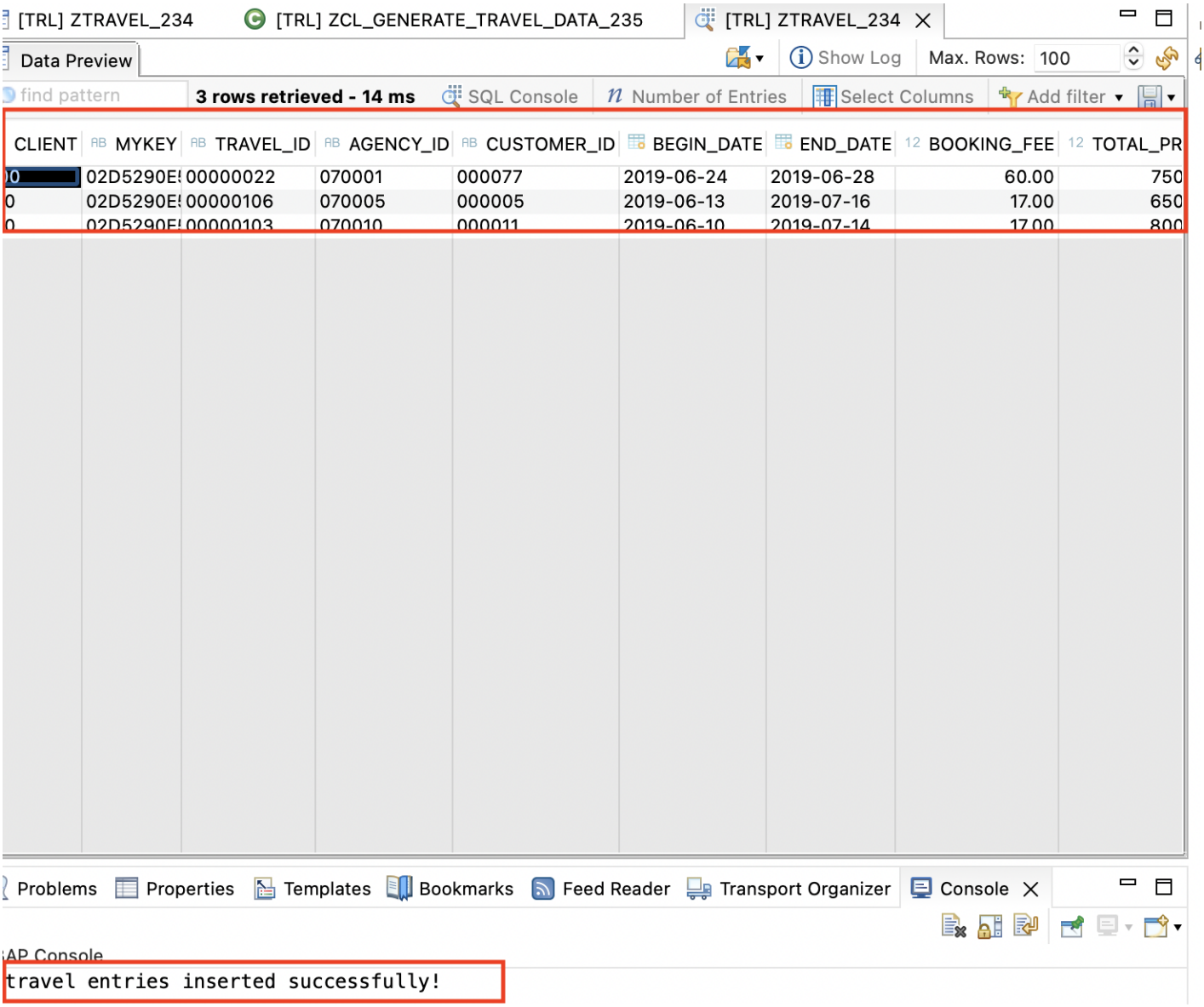
- Adding the interfaces **if_oo_adt_classrun** will allow us to utilize ADT Eclipse. This will allow you to print out any value in the console. This is helpful because it will print out our console message and confirms if everything is correct

- We then clean the table by deleting any existing entries

- We then insert the new table entries we got in our ITAB

- We then output the result as a console message

Now we can review the data. Also, you can see a success message is written to the console.


[TRL] ZTRAVEL_234
[TRL] ZCL_GENERATE_TRAVEL_DATA_235
[TRL] ZTRAVEL_234
Data Preview
Show Log
Max. Rows: 100
find pattern
3 rows retrieved - 14 ms
SQL Console
Number of Entries
Select Columns
Add filter
CLIENT MYKEY TRAVEL_ID AGENCY_ID CUSTOMER_ID BEGIN_DATE END_DATE BOOKING_FEE TOTAL_PR
0 02D5290E! 00000022 070001 000077 2019-06-24 2019-06-28 60.00 750
0 02D5290E! 00000106 070005 000005 2019-06-13 2019-07-16 17.00 650
0 02D5290E! 00000103 070010 000011 2019-06-10 2019-07-14 17.00 800
Problems
Properties
Templates
Bookmarks
Feed Reader
Transport Organizer
Console
AP Console
travel entries inserted successfully!

Define CDS-based data model – Part 2

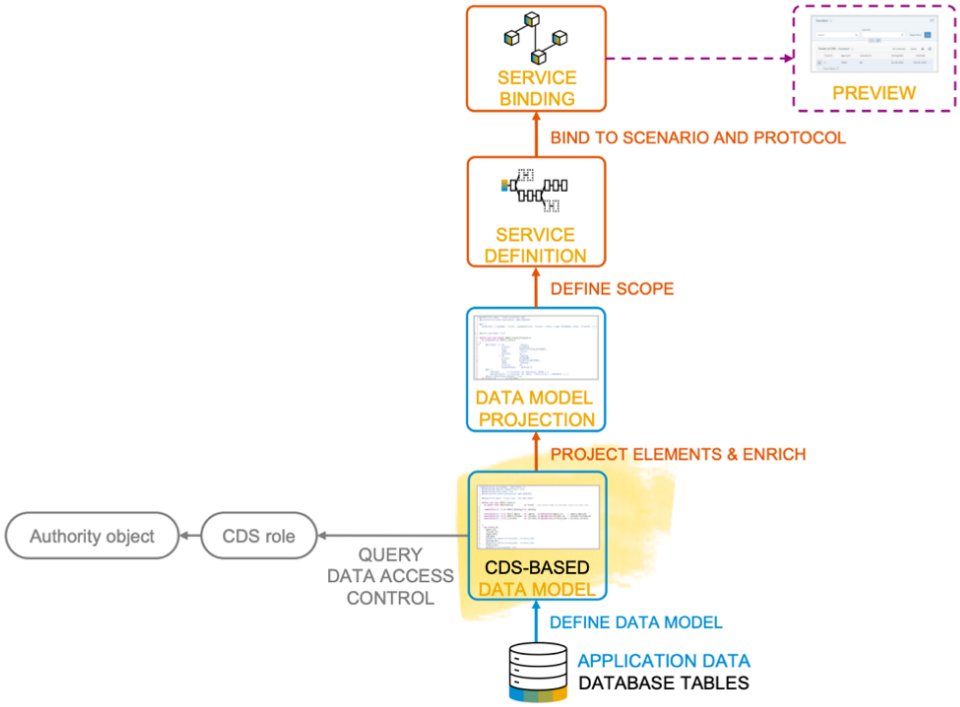
3 13 6,514

Welcome back!

In the previous blog post, I covered my experience with creating a CDS data model and filling in some data. In this post, I'll continue with the tutorials to develop a Fiori app using the ABAP RESTful Application Programming Model. I'll also share key takeaways from my experience defining and exposing a CDS-based travel data model.

CDS- Based Data Model

We went over the development flow in the introduction of this series. In the last post, we created a database table. This time, I'll take you through my experience of exposing a CDS-based travel data model for the Fiori app. We'll use it to define the data model of the travel and booking business object.



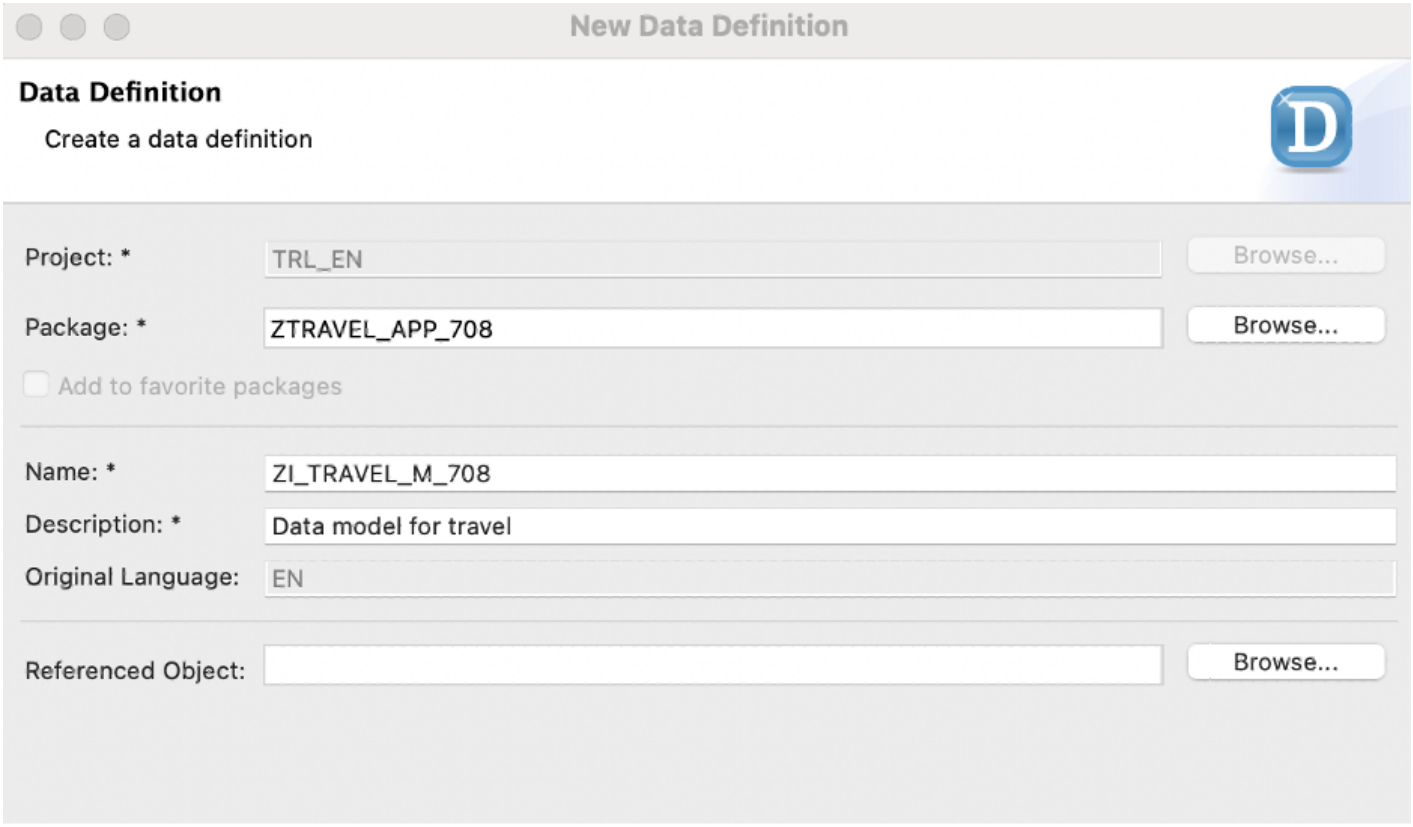
RAP Development Flow

For those unfamiliar, CDS stands for **Core Data Services**. It's the next-gen data modeling infrastructure on the ABAP platform, offering a declarative approach to defining domain-specific, semantically rich data models. CDS Views can be consumed in many ways, including as a data source for SAP Fiori apps, exposed using SAP Gateway in the OData format. You can also build Fiori apps on top of CDS Views using SAP Fiori Elements templates.

For more information on CDS view: https://help.sap.com/docs/btp/sap-abap-cds-development-user-guide/about-abap-cds-development-user-guide

Defining CDS-based travel data model

Okay, let's switch things up a bit! So, in this tutorial we're diving into the world of the CDS travel data model. We start by defying the CDS travel data model. You'll need to make a data definition for the travel root business object, which is pretty simple. Just right-click on your package and choose "Other ABAP Repository Object." And don't forget, as we mentioned in the last post, to change the suffix to something that suits you best.



Data Model for Travel

Note: Since CDS views are (public) interface views, they are prefixed with I_ in accordance with the VDM (virtual data model) naming convention. In addition, we add the suffix _M to the view name in case it is specific for our managed implementation type scenario. For detailed information, see: Naming Conventions for Development Objects
Once the data definition was established, I proceeded to add the necessary components for the association, fields, and UI semantics to the root node.The tutorial provided a code snippet which I replaced in the editor. Essentially, what we're doing is using the travel table as a data source and inserting its fields into the projection list between the curly brackets. As always, I'll provide an explanation of the steps taken in the code for your understanding and reference. It is important to make sure you get a good idea on CDS view to be able to fully understand the code.

```
@EndUserText.label: 'Travel data XXX'
@AccessControl.authorizationCheck: #CHECK
define root view entity ZI_TRAVEL_M_708

  as select from ztravel_708 as Travel

  /* Associations */
  association [0..1] to /DMO/I_Agency   as _Agency   on $projection.agency_id = _Agency.AgencyID
  association [0..1] to /DMO/I_Customer as _Customer on $projection.customer_id = _Customer.CustomerID
  association [0..1] to I_Currency      as _Currency on $projection.currency_code = _Currency.Currency

{

  key mykey,
      travel_id,
      agency_id,
      customer_id,
      begin_date,
      end_date,
      @Semantics.amount.currencyCode: 'currency_code'
      booking_fee,
      @Semantics.amount.currencyCode: 'currency_code'
      total_price,
      currency_code,
      overall_status,
      description,

      /*-- Admin data --*/
      @Semantics.user.createdBy: true
      created_by,
      @Semantics.systemDateTime.createdAt: true
      created_at,
      @Semantics.user.lastChangedBy: true
      last_changed_by,
      @Semantics.systemDateTime.lastChangedAt: true
      last_changed_at,

      /* Public associations */
      _Agency,
      _Customer,
      _Currency
}
```

CDS-based Travel Data Model

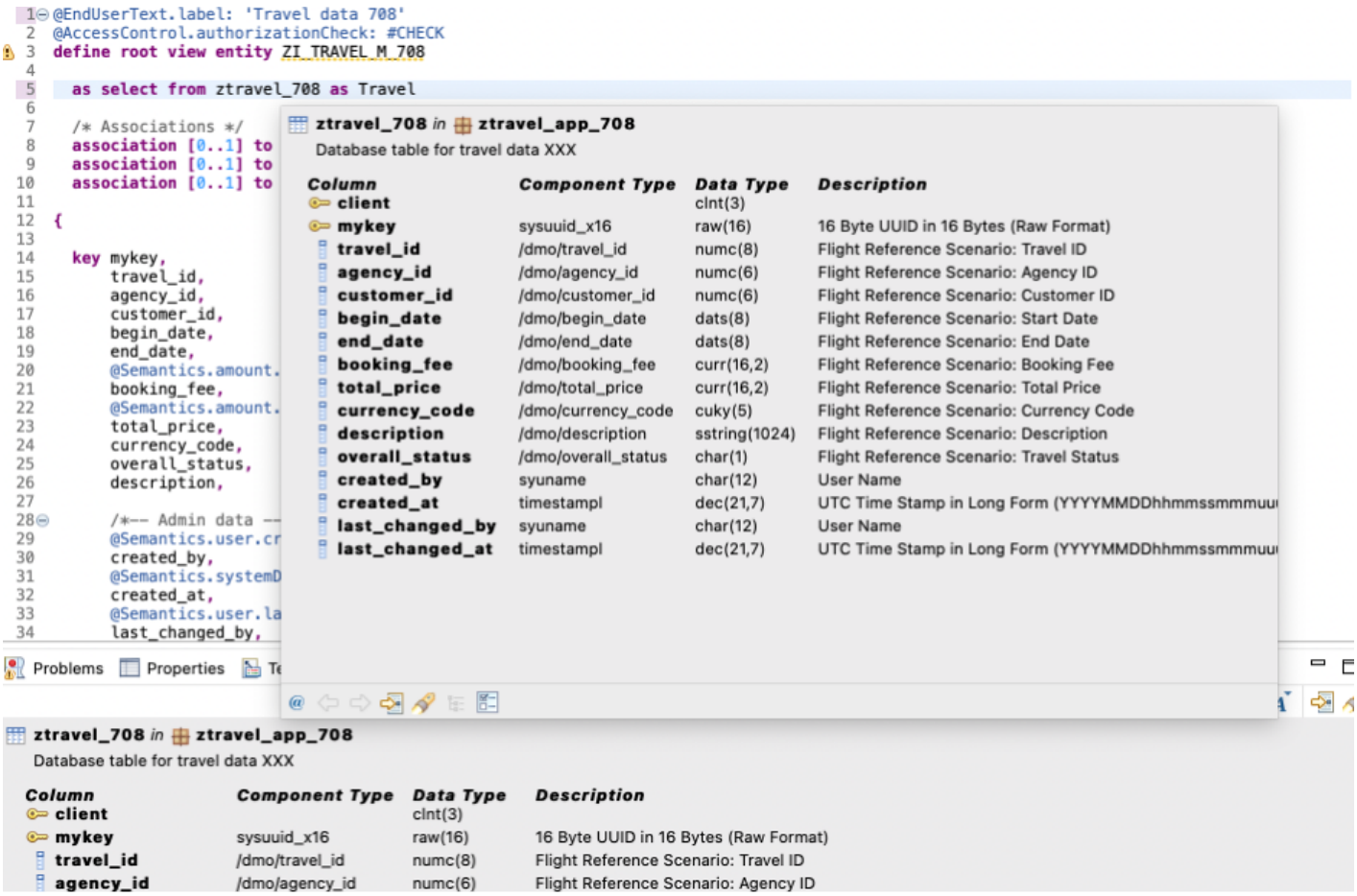
Code Explanation:

1. This code defines a root view entity for the travel root business object, named "**ZI_TRAVEL_M_000**" (with "000" replaced with your desired suffix). It's created by selecting data from the "ztravel_000" entity.
2. Associations are defined in the CDS code to access master data from other entities. These associations refer to CDS entities in the demo app scenario. The entity **"ZI_TRAVEL_M_000"** has a relationship with these entities and the associations are defined with cardinality [0..*]. This means that any number of booking instances can be assigned to each travel instance. The [0..1] indicates that there can be zero or one entity associated with each association.
3. The select list includes standard administration data fields, with semantic annotations for the currency and administrative fields for uniform data processing. The **currencyCode** field is specified as a reference field for the currency fields **Booking_Fee** and **Total_Price**. The **"@Semantics.user.createdBy: true"** annotation indicates that the created_by field stores the user who created the data, while the **"@Semantics.systemDateTime.creationAt: true"** indicates that the **created_at** field stores the creation time and date. The administrative field annotations are necessary to allow automatic updates of the admin fields on every operation.

Code Element Information for Data Definitions:

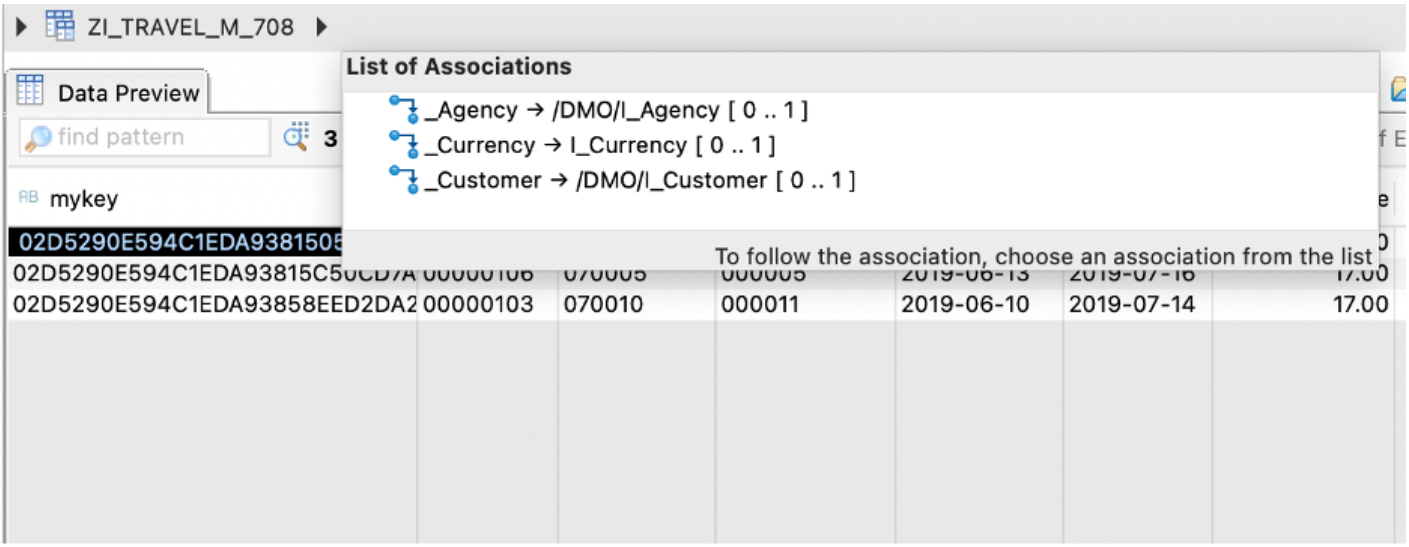
While working on this step, I came across the Element Information popup and ABAP Element Info view. These displays provide information about the ABAP dictionary objects used in CDS. This can be extremely useful when coding, as it provides further details about the elements being used.

You can access the Element Information popup by clicking on the relevant element in the CDS source code and selecting “Show Code Element Information” from the context menu. The ABAP Element Info view can then be opened from the popup by clicking the “Show in ABAP Element Info view” icon. This view allows you to view both your CDS source code and the related element information simultaneously.



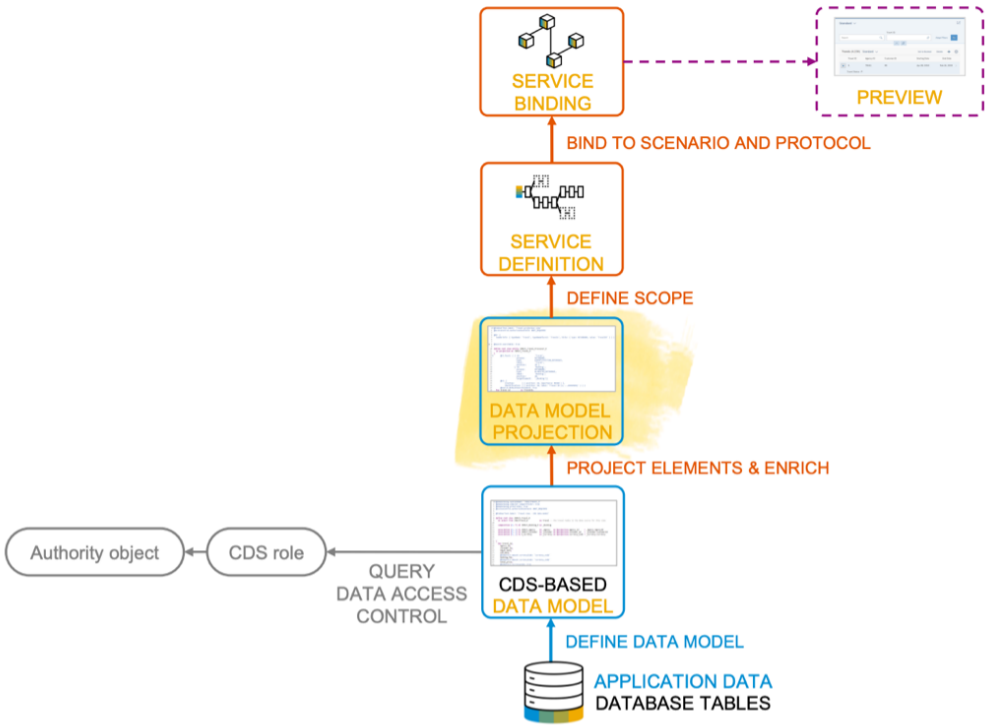
Element Information popup and ABAP Element Info view

The next step was to activate and save the code. After activation, you will be able to preview the data in ADT. During my preview of the data, I discovered various features that can be toggled. For instance, it is possible to navigate to the associated data when previewing the CDS views. I found this feature helpful and believe it could also be of benefit to others.



Projection Views for travel

Once we defined the data model for our travel data, we proceeded to the next step of creating a projection view. This is essentially a subset of the fields from the travel data model that are deemed relevant for our travel booking application. We'll be diving into creating the projection view, following the development flow below.



RAP Development Flow

In this step, we created a projection view by defining a data definition. The steps are similar to those we followed when we created the data model for travel. The process involved right-clicking the package and selecting **"Other ABAP Repository Object."** The naming convention for the projection view follows the SAP S/4HANA data model, starting with the namespace and a capital letter "C" followed by the suffix "_M" to indicate it is specific for the managed implementation type scenario. For example, we named our projection view **"ZC_TRAVEL_M_000"** and provided the description "projection

view for travel.” It is important to note that the projection view is a subset of the fields of the travel data model that are relevant for the travel booking application.

The new data definition appeared in the editor, and we replaced the code with the code snippet from the tutorial. Understanding the code and its purpose is important for being able to effectively use the projection view in your application. That’s why I will provide an explanation of the code so you can have a clear understanding of what we have accomplished in this step.

```
@EndUserText.label: 'Travel projection view - Processor'
@AccessControl.authorizationCheck: #NOT_REQUIRED

@UI: {
 headerInfo: { typeName: 'Travel', typeNamePlural: 'Travels', title: { type: #STAN

@Search.searchable: true

define root view entity ZC_TRAVEL_M_000
  as projection on ZI_TRAVEL_M_000
{
      @UI.facet: [ { id:              'Travel',
                     purpose:         #STANDARD,
                     type:            #IDENTIFICATION_REFERENCE,
                     label:           'Travel',
                     position:        10 } ]

      @UI.hidden: true
  key mykey               as TravelUUID,


      @UI: {
          lineItem:       [ { position: 10, importance: #HIGH } ],
          identification: [ { position: 10, label: 'Travel ID [1,...,99999999]' }
      @Search.defaultSearchElement: true
      travel_id           as TravelID,

      @UI: {
          lineItem:       [ { position: 20, importance: #HIGH } ],
          identification: [ { position: 20 } ],
          selectionField: [ { position: 20 } ] }
      @Consumption.valueHelpDefinition: [{ entity : {name: '/DMO/I_Agency', elemen

      @ObjectModel.text.element: ['AgencyName']
      @Search.defaultSearchElement: true
      agency_id           as AgencyID,
      _Agency.Name        as AgencyName,

      @UI: {
          lineItem:       [ { position: 30, importance: #HIGH } ],
          identification: [ { position: 30 } ],
```

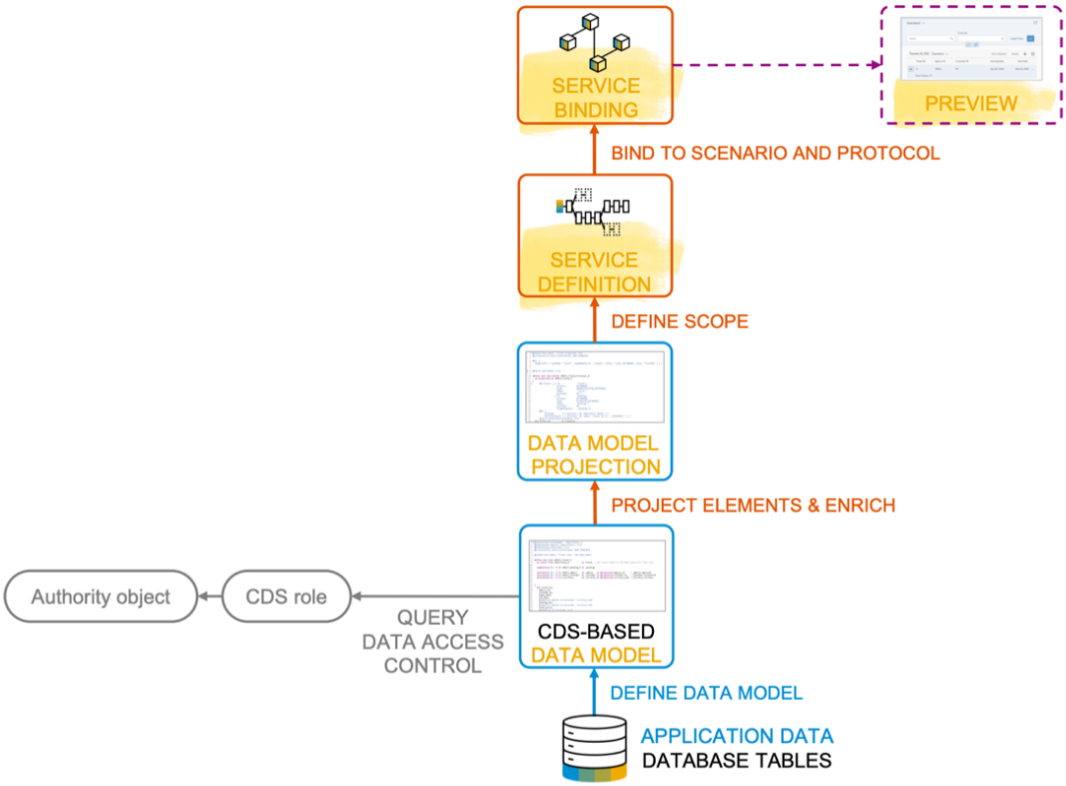
```
        selectionField: [ { position: 30 } ] }
    @Consumption.valueHelpDefinition: [{ entity : {name: '/DMO/I_Customer', elem

    @ObjectModel.text.element: ['CustomerName']
    @Search.defaultSearchElement: true
    customer_id        as CustomerID,

    @UI.hidden: true
    _Customer.LastName as CustomerName,

    @UI: {
        lineItem:       [ { position: 40, importance: #MEDIUM } ],
        identification: [ { position: 40 } ] }
    begin_date         as BeginDate,

    @UI: {
        lineItem:       [ { position: 41, importance: #MEDIUM } ],
        identification: [ { position: 41 } ] }
    end_date           as EndDate,

    @UI: {
        lineItem:       [ { position: 50, importance: #MEDIUM } ],
        identification: [ { position: 50, label: 'Total Price' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    total_price        as TotalPrice,

    @Consumption.valueHelpDefinition: [{entity: {name: 'I_Currency', element: 'C
    currency_code      as CurrencyCode,

    @UI: {
    lineItem:       [ { position: 60, importance: #HIGH },
                      { type: #FOR_ACTION, dataAction: 'acceptTravel', label: 'A
    identification: [ { position: 60, label: 'Status [O(Open)|A(Accepted)|X(Canc
    overall_status     as TravelStatus,

    @UI.identification: [ { position: 70, label: 'Remarks' } ]
    description        as Description,

    @UI.hidden: true
    last_changed_at    as LastChangedAt

}
```

Code Explanation:

- **@UI**: Specifies the user interface (UI) attributes of the view entity, including the header information, line item display, identification, selection field, and facet display

- **@Search.searchable: true** enables the full-text search, making the entity searchable
- **@Search.DefaultSearchElement** enables the freestyle search for the view columns
- **agency_id** as **AgencyID:** is used to store the agency ID and is defined as **AgencyID**.
  The **@Consumption.valueHelpDefination** annotation specifies that a value help should be provided for this attribute and the entity to be used for value help is **/DMO/I_Agency** and its local element is **'AgencyID'.** The **@ObjectModel.text.element**t annotation specifies that the text to displayed for this attribute is '**AgencyName'**
- **begin_date** as **BeginDate** and **end_date** as **EndDate**: The **begin_date** and **end_date** attribute is used to store the start and end date of the travel.
- **Total_price** as **TotalPrice**: This attribute is used to store the total price of the travel.
  The **@Semenatics.amount.currencyCode** annotation specifies that the currency code should be used for this attribute.
- **overall_status** as **TravelStatus:** This attribute is used to store the status of the travel (Open, Accepted, Canceled). The UI annotations specify its position, importance, and label in the UI.

Service Definition and Service Binding



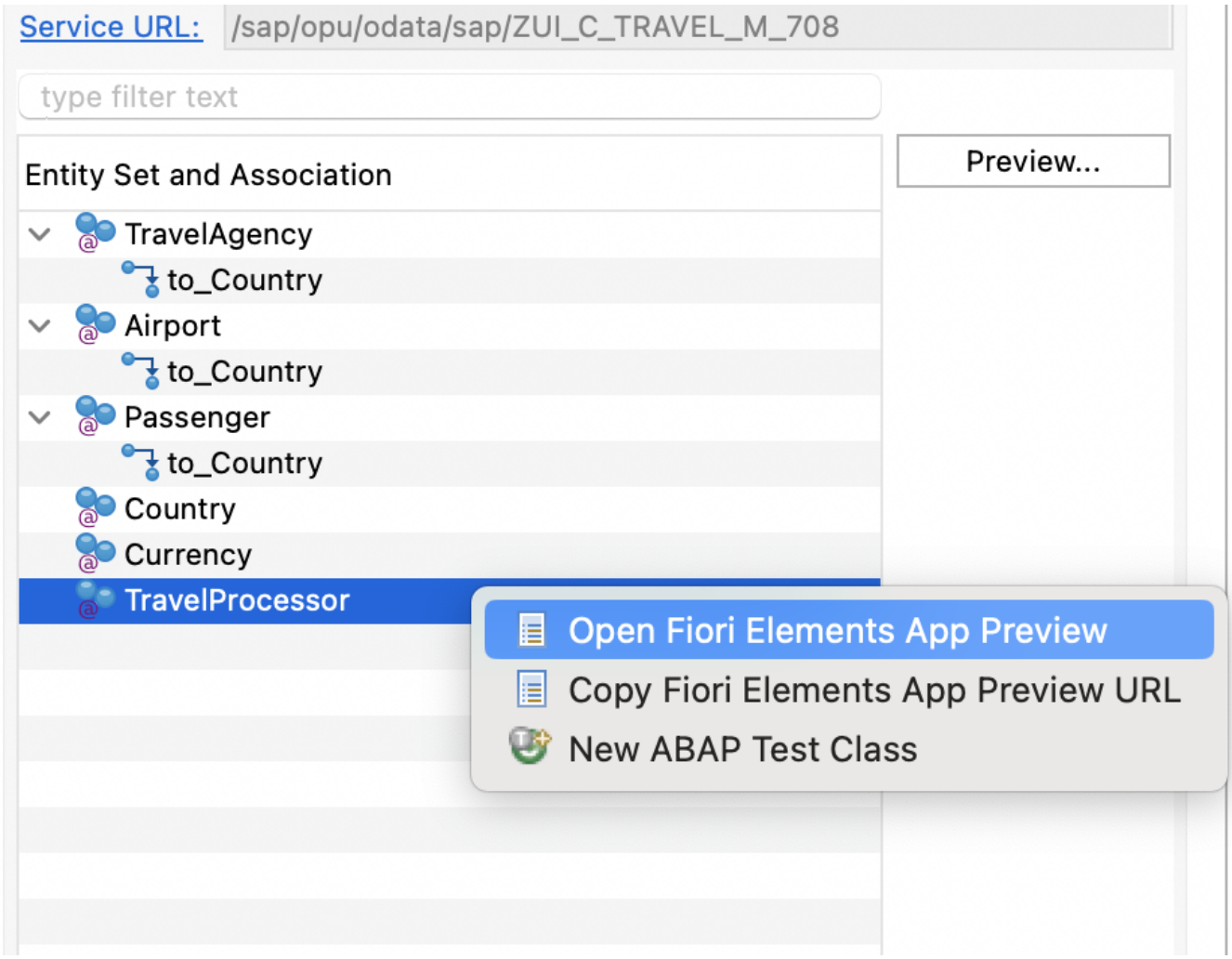
The next step in the tutorial was to create the OData service and preview the app. This is a crucial step as it brings our travel data model and projection view to life by enabling it to be accessed and consumed by external applications. By creating the OData service and previewing the app, we will be able to validate the correctness of our implementation and ensure that everything is working as expected.

The service definition outlines the scope of the OData service, while the service binding connects it to the OData protocol as a UI service.

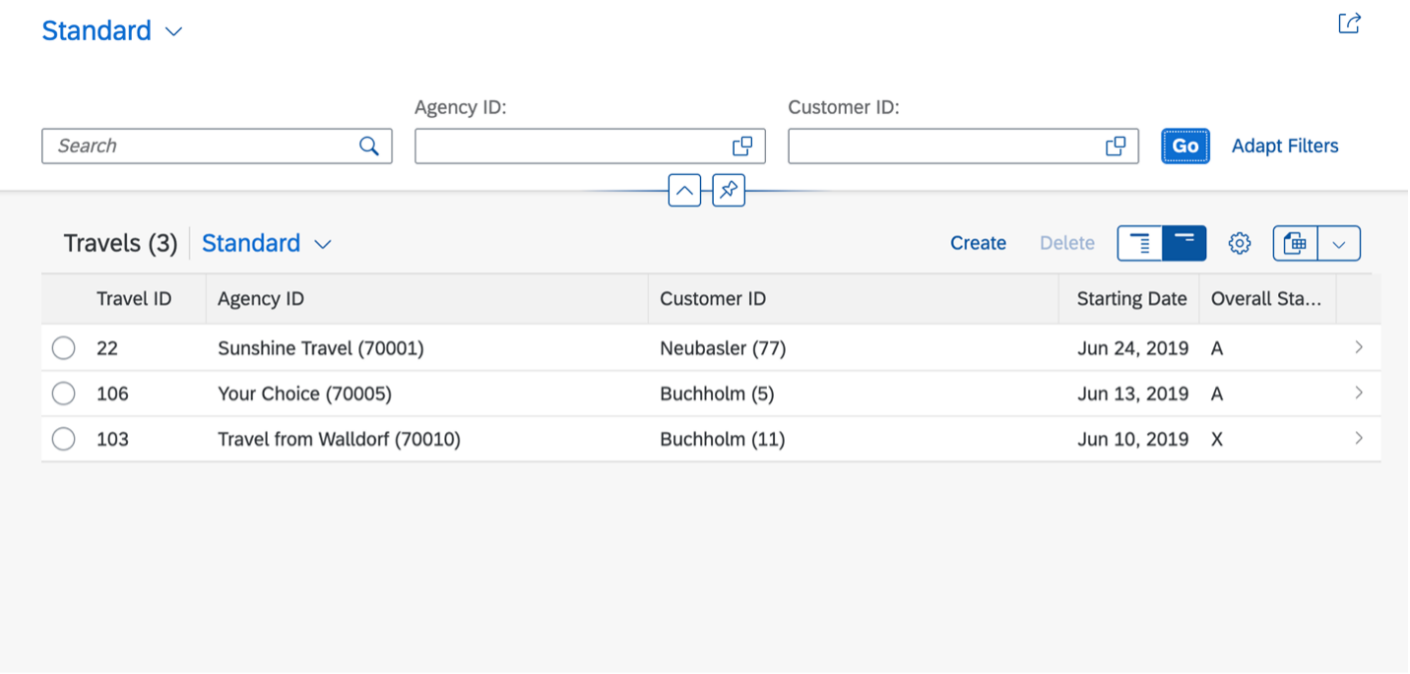
```
1  @EndUserText.label: 'Service Definition for ZC_Travel_M_708'
2  define service ZUI_C_TRAVEL_M_708 {
3    expose ZC_TRAVEL_M_XXX as TravelProcessor;
4    expose /DMO/I_Customer as Passenger;
5    expose /DMO/I_Agency as TravelAgency;
6    expose /DMO/I_Airport as Airport;
7    expose I_Currency as Currency;
8    expose I_Country as Country;
9  }
```

The tutorial guides you through creating the service definition and service binding. Remember to activate the service binding after creation and publish it. The activation and publishing process may take a few minutes, so feel free to grab a coffee or check your emails while you wait. Once the process is complete, you will be able to view your service along with its entities and associations.

To preview the app, open the Fiori Elements App Preview by right-clicking on it in the project explorer and selecting the option “Open Fiori Elements App Preview” from the context menu. This allows you to see how your app looks and functions before actually deploying it.



You will then see the travel list report app. Just a heads up, once you open the travel list report app, don't forget to press **"GO"** to load the back-end data. I got stuck there for a bit, so no worries, I got you covered!



CDS Behavior Definition

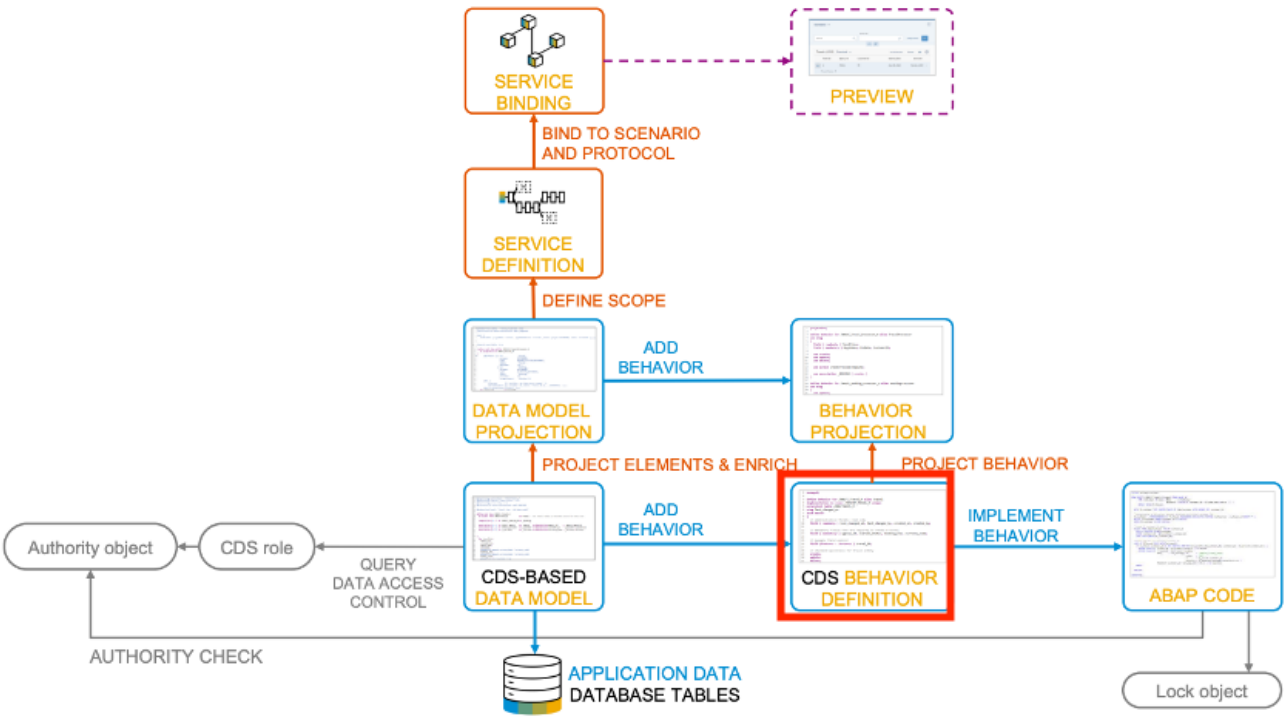
3 22 14,671

Welcome back to our ongoing series on getting to know RAP!

In the previous blog post, we went over the steps on how to create a CDS data model and how to project this data into an OData service. Additionally, we also showed how to consume this data model using the SAP Fiori elements app preview.

In this blog post, we will dive into creating a behavior definition and implementation for a managed scenario. The behavior definition is a crucial aspect of any CDS data model as it defines the actions that can be performed on the data, such as creating, updating, and deleting records.

To create the behavior definition, we will utilize the context menu in the root CDS interface view that was created in the previous blog post. This allows us to quickly and easily define the desired behaviors for our travel booking data. We will also go over the steps to implement these behaviors and how they can be used in the SAP Fiori elements app preview.



RAP Development Flow

Behavior Definition

Behavior definitions are a crucial part of our CDS data models because they determine what we can do with our data. The tutorial walks you through the process of creating a behavior definition for the root CDS view, which will apply to all the entities within it.

To create a behavior definition, simply right-click on the data definition **ZI_TRAVEL_M_000** and choose **"New Behavior Definition."** It's worth mentioning that in this case, we're working with a managed implementation, so make sure the

implementation type is set to **“Managed”** and that the name of the behavior definition is exactly the same as the root CDS view.

Once you’ve created the behavior definition, you’ll see that it is automatically generated based on the implementation type you selected. Now, let’s take a look at the code that we’re asked to replace.

```
managed implementation in class zbp_i_travel_m_000 unique;

define behavior for ZI_TRAVEL_M_000 alias Travel
persistent table ztravel_000
etag master last_changed_at
lock master
{

  // semantic key is calculated in a determination
  field ( readonly ) travel_id;

  // administrative fields (read only)
  field ( readonly ) last_changed_at, last_changed_by, created_at, created_by;

  // mandatory fields that are required to create a travel
  field ( mandatory ) agency_id, overall_status, booking_fee, currency_code;

  // mandatory fields that are required to create a travel
  field ( mandatory ) Begin_Date, End_Date, Customer_ID;

  // standard operations for travel entity
  create;
  update;
  delete;
}
```

Code Explanation

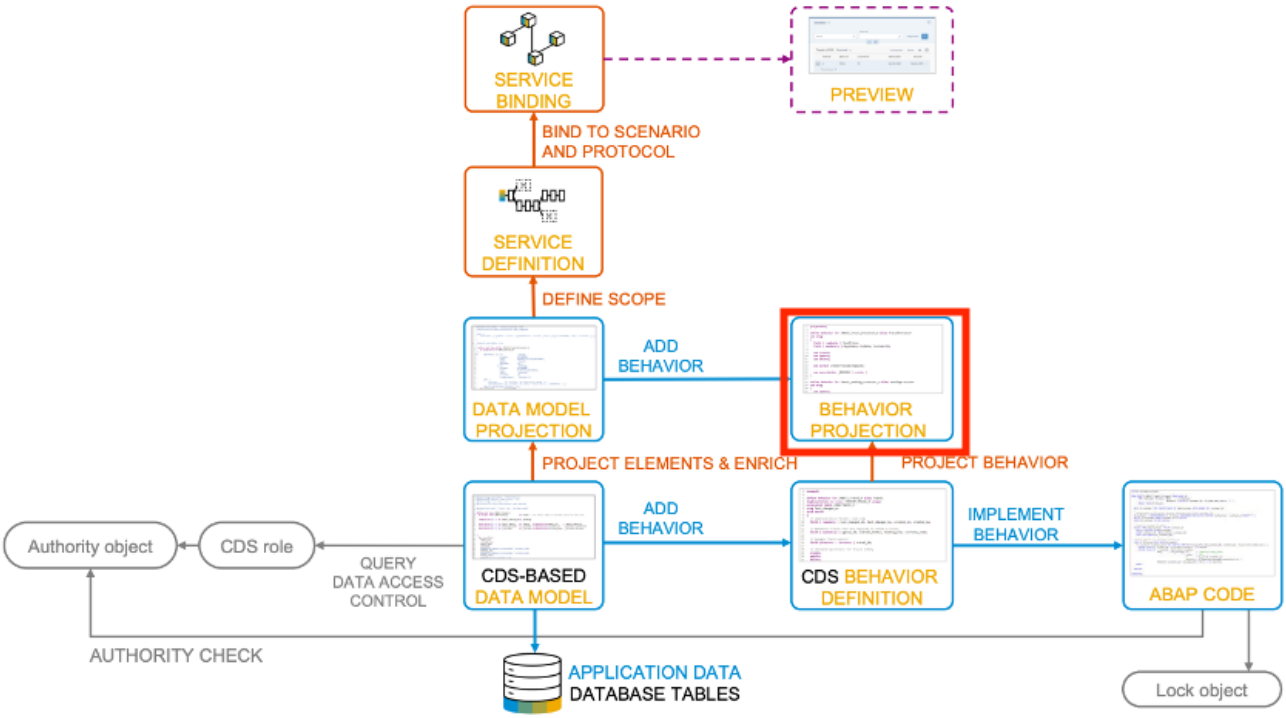
1. First we assign an alias to the travel entity and specify the persistency by indicating the corresponding database table. This allows the managed runtime to perform create, update, and delete operations directly on the database table.
2. The **etag master last_changed_at** is used to manage versioning of the data stored in the table. It records the time and date when a change was made to the data, enabling us to track changes over time. We then specified the **lock master** for the root entity. The **lock master** is used to ensure data consistency by locking the data while it is being updated. This prevents other processes from accessing or modifying the data until the update is complete.
3. The semantic key **TRAVEL_ID** is calculated in a determination and is read-only, meaning that its value cannot be changed by the user. You can think of it like a serial number – you can’t change it once it’s assigned.
4. The administrative fields **last_changed_at**, **last_changed_by**, **created_at**, and **created_by** are also read-only and provide information about when the data was last changed and by whom.
5. The fields **agency_id**, **overall_status**, **booking_fee**, and **currency_code** are mandatory fields that are required to create a travel booking. Additionally, **Begin_Date**, **End_Date**, and **Customer_ID** are also mandatory fields that must

be provided in order to create a travel booking. These fields help ensure that the necessary information is provided for each travel booking, making it easier to manage and keep track of the data.

After making the necessary additions to the code, it is important to save and activate the changes to take effect.You may encounter a warning message during this process, but it should be resolved once the behavior implementation has been successfully created. This step is key to making sure the changes we made take effect and are ready for use.

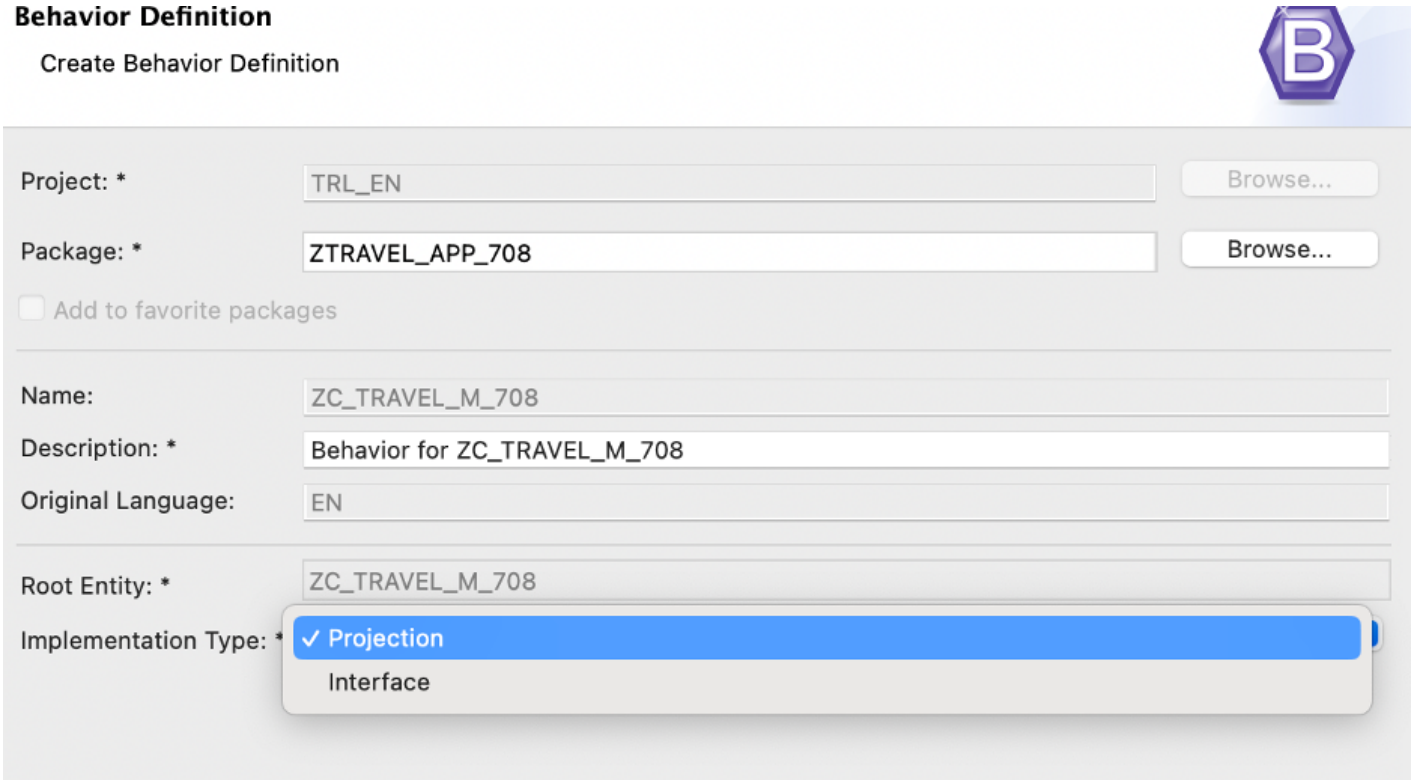
Behavior Definition For Projection View

So, we've set up the behavior definition for our managed travel business object and given it the behaviors we want for the travel entity. Now, it's time to move on to the next step: the behavior projection. This is just a fancy way of mapping the transactional abilities from the base behavior definition. All we gotta do is create another behavior definition based on the root CDS projection view we made earlier. This view is a representation of the base behavior definition, and it's where we can get even more specific with the transactional abilities we want for our travel entity.



RAP Development Flow

Projections are a way to simplify data access and manipulation in RAP. They allow you to present data in a specific format that makes it easier for users to understand and work with. Just like we did before, we'll right-click on the data definition **ZC_TRAVEL_M_000** and select **"New Behavior Definition".**



Once you have assigned a transport request and clicked finish, you'll notice that the behavior definition is set based on the implementation type you selected earlier. Now, it's time to replace the code with the following:

```
projection;

define behavior for ZC_TRAVEL_M_000 alias TravelProcessor
use etag
{
  use create;
  use update;
  use delete;
}
```

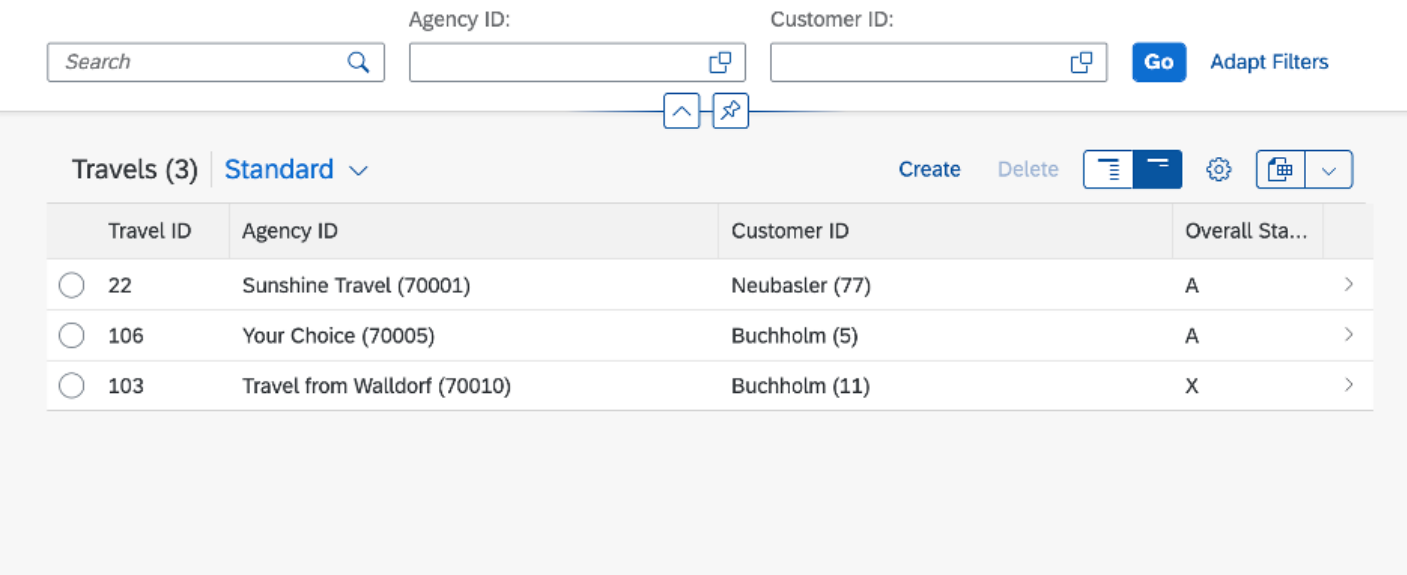
Code Explanation:

1. We start by defining an alias for the travel entity.
2. The next step is to enable **ETag** handling. The use of **ETag**s helps ensure that the user is always working with the latest version of the data. This is because whenever the user performs **create**, **update**, or **delete** operations on the projection, the data in the underlying entity will also be updated. The **ETags** also help prevent conflicts in case multiple users are working with the same data simultaneously.

Projections in ABAP RAP are a convenient way to access and manipulate data. They guarantee that the data you're working with is always accurate and up-to-date.

After you've added the code, saved and activated it, you should switch to the service binding, activate the service for **ZUIC_C_TRAVEL_M_000**, and double-click on **TravelProcessor** to start the preview. Be sure to refresh the application and click **"GO"**. Note that in the trial environment, it may take a while for the changes to be reflected.

Once the changes are reflected, you will see that the create, update, and delete operations are now enabled, allowing you to play around with the application.



What's Next:

In this blog post, I have guided you through the process of creating a behavior definition for a managed travel business object. We went over the steps to define the behavior for the travel entity, followed by creating the behavior definition projection. In the next blog post, we will dive deeper into enhancing the behavior definition and implementation by adding actions and validations.