

---

# SAP ABAP RESTful Application Programming Model

Srihariram Chendamarai Kannan

---

---

## Abstract

The world of SAP business applications has experienced significant changes, propelled by evolving demands and technological progress. Two important parts of this change toward the Intelligent Enterprise are SAP HANA and SAP Fiori. SAP HANA has a high-performance in-memory database that allows real-time transactional and analytical processing, and SAP Fiori provides a current and easy-to-use user experience. Furthermore, the increasing demand for cloud support, ranging from hybrid to entirely cloud-based models, has been a substantial impetus. ABAP RESTful Application Programming Model (RAP) is now available on SAP BTP ABAP Environment and SAP S/4HANA, both on-premises and in the cloud. It is designed to meet the wide range of complex needs for the ABAP platform. This article takes a close look at the RAP model, looking at its structure, what it can do, and how it affects the development of SAP applications. It also looks at how well it helps businesses stay ahead in a competitive market while meeting users' changing needs.

---

### Keywords:

SAP ABAP;  
S4HANA;  
RESTful Programming;

---

### Author correspondence:

Srihariram Chendamarai Kannan,  
SAP Lead Analyst  
Edison, New Jersey, USA 08820  
Email: srihariram@hotmail.com

---

## 1. Introduction

The ABAP RESTful Application Programming Model (RAP) is a flexible framework that lets programmers design state-of-the-art, cloud-compatible enterprise applications in a variety of fields, including analytical and transactional ones. The adaptability of this platform makes it easy to improve SAP standard applications on the ABAP framework, whether they are in the cloud or on-premise. It also makes it easy to create SAP HANA-optimized OData services, such as Fiori apps. The RAP paradigm provides comprehensive support for numerous Fiori applications and enables effortless Web API publishing. Modern technologies like Core Data Services (CDS) and a service model framework make it possible to create OData services quickly and easily. Additionally, it incorporates ABAP-based application services and SAPUI5-based user interfaces to enhance custom logic and user experiences. The framework explores the significant influence of the ABAP RAP on enterprise application development, assessing its efficacy in addressing the varied requirements of businesses in the current dynamic digital environment.

RAP provides a consistent development process within the contemporary ABAP Development Tools in Eclipse (ADT), along with a comprehensive set of capabilities for constructing applications across various domains, encompassing transactional and analytical applications, whether starting from the ground up or leveraging pre-existing custom code.

Different types of services and local APIs can be developed with RAP:

- OData-based services for exposure for UI development to build delightful, role-based, responsive, and draft-enabled SAP Fiori apps.
- OData-based services for exposure as Web APIs.
- InA-based, analytical services for building analytical apps in SAP Analytics Cloud.
- SQL-based services for data integration.



## 2.1 Data Modeling & Behavior

The data modeling and behavior layer handles the data and its associated business logic.

### 2.1.1 Data Model

The data model comprises the description of the different entities involved in a business scenario and their relationships. The ABAP RESTful Programming Model employs CDS for defining and structuring the data model, with each real-world entity being represented by a single CDS entity. CDS provides a framework for defining and consuming semantic data models. The ability to build views enables you to specify application-specific attributes within the data model.

Provide the Persistent Tables:

Depending on the development scenario, these can be existing custom tables or tables created specifically for this application.

Create CDS Data Definitions:

The CDS views are the basis for the data model. For each persistent table a corresponding interface CDS view is created.

### Data Model and Business Object Structure

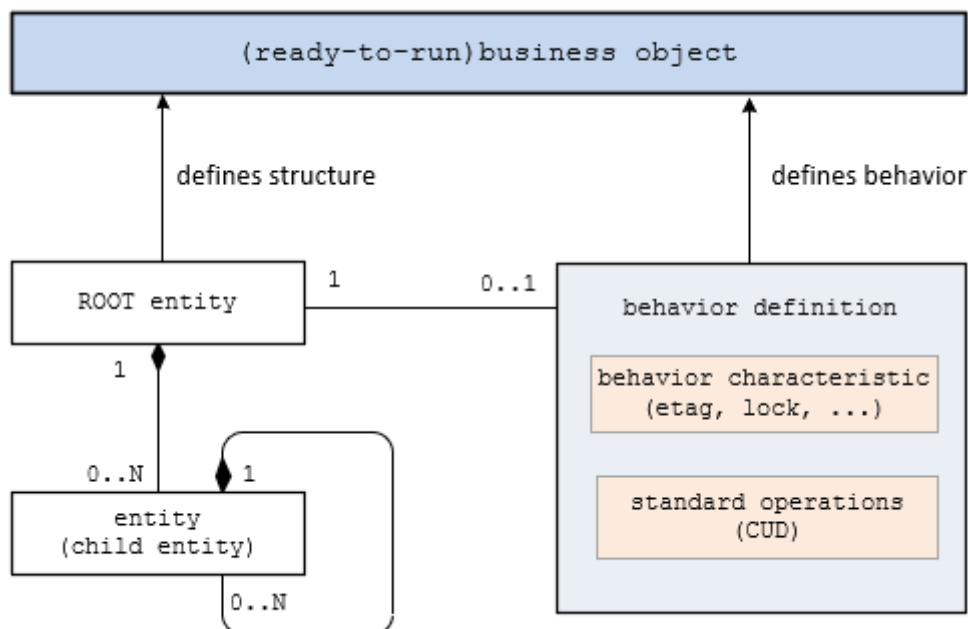


Figure 2. RAP – Data Model and Business Object Structure [2]

As illustrated in the diagram above, a business object's structure is delineated by the sequence of compositions and to-parent associations linking CDS entities, with a root entity situated atop the composition tree. Each tier within a business object's composition tree can offer a set of operations specified in the behavior definition, which references the entities within the CDS data model.

In the context of a managed implementation type, the standard operations (create, update, delete) need only be defined in the behavior definition to attain a fully operational business object. The underlying business object runtime infrastructure inherently handles the interaction phase and save sequence, delivering generic support for transactional processing as a default.

#### Root View (Parent)

This CDS view defines the root entity of the managed business object, serving as the foundation of the compositional hierarchy for parent-child object .

To ensure consistent data processing from the consumer's perspective, all administrative and quantity fields are equipped with appropriate @Semantics annotations. These annotations are crucial for supporting managed ETag handling, enabling automatic updates of pertinent ETag fields following each operation.

#### Sample Code

```

1 @AbapCatalog.viewEnhancementCategory: [#NONE]
2 @AccessControl.authorizationCheck: #NOT_REQUIRED
3 @EndUserText.label: 'OData Root'
4 @Metadata.ignorePropagatedAnnotations: true
5 @ObjectModel.usageType:{
6     serviceQuality: #X,
7     sizeCategory: #S,
8     dataClass: #MIXED
9 }
10 define root view entity ZCDS_I_ROOT
11     as select from zodata_root
12     composition [0..*] of ZCDS_I_LOAD as _Item
13 {
14
15     key load_id,
16         ernam,
17         erdat,
18
19         _Item
20 }

```

#### Child View

In adherence to the composition relationship, associations are established for the booking supplement child entities to link with their corresponding compositional parent entities.

#### Sample code

```

1 @AbapCatalog.viewEnhancementCategory: [#NONE]
2 @AccessControl.authorizationCheck: #NOT_REQUIRED
3 @EndUserText.label: 'OData Load data'
4 @Metadata.ignorePropagatedAnnotations: true
5 @ObjectModel.usageType:{
6     serviceQuality: #X,
7     sizeCategory: #S,
8     dataClass: #MIXED
9 }
10 define view entity ZCDS_I_LOAD
11     as select from zodata_load
12     association to parent ZCDS_I_ROOT as _Header on $projection.load_id = _Header.load_id
13 {
14     key load_id,
15     key unique_id,
16         name_last,
17         name_first,
18         nation,
19         state,
20         birth_date,
21         birth_plac,
22         marital_st,
23         children,
24         ernam,
25         erdat,
26
27         _Header
28 }

```

### 2.1.2 Behavior Definition & Implementation- Managed

In the context of the ABAP RESTful application programming model, a behavior definition (abbreviated as behavior definition) stands as an ABAP Repository entity that elucidates the conduct of a business object. This description is crafted using the Behavior Definition Language (BDL).

The implementation of a behavior definition occurs either within a single ABAP class (termed a behavior pool) or can be distributed throughout multiple ABAP classes (known as behavior pools). The application developer has the authority to link any number of behavior pools to a single behavior specification, allowing for a cardinality of 1 to N.

In this context, a behavior delineates the actions and characteristics relevant to a certain object under the ABAP RESTful programming model. It includes both a behavioral attribute and a set of operations for each entity that forms the composition tree of the business object.

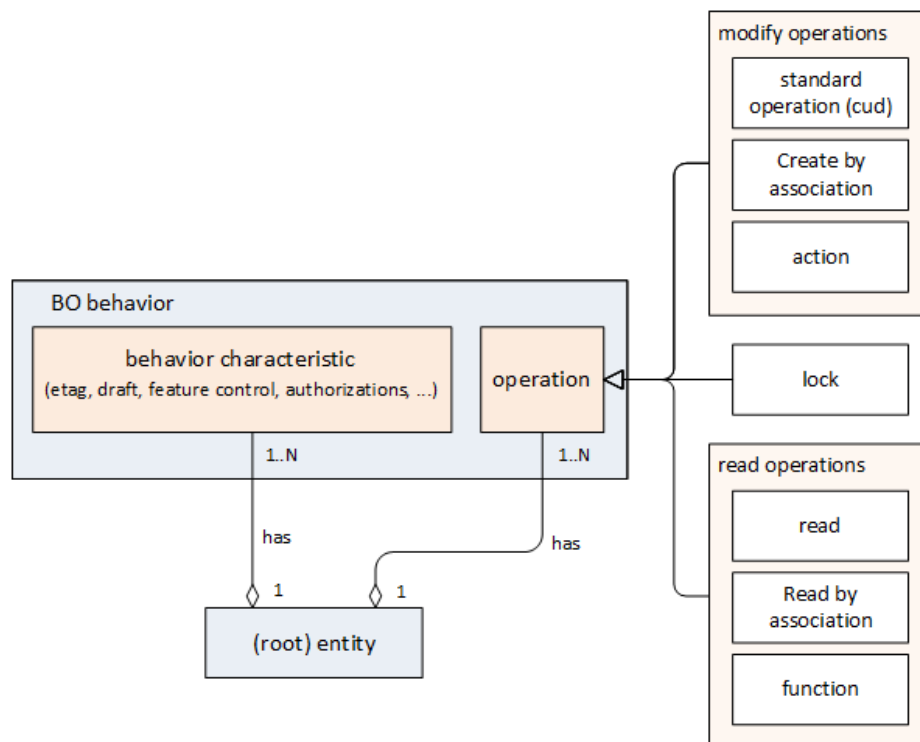


Figure 4. Business Object's Behavior[2]

#### Behavior Characteristic

The behavior characteristic pertains to the aspect of a business object that defines the fundamental properties of an entity. such as:

##### Strict

The strict mode represents the recommended practices for modeling and implementing a RAP business object. When a business object is designated as 'strict,' it undergoes technical enforcement through additional syntax checks on certain design and implementation criteria. This ensures that the business object remains stable throughout its lifecycle, is safe for upgrades, and adheres to best practices.

#### Concurrency Control

##### Optimistic Concurrency Control (ETag)

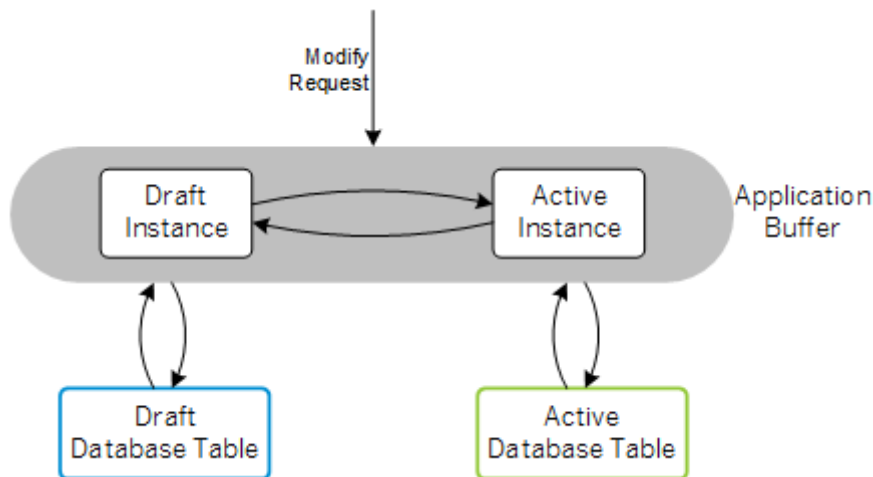
ETags are used for optimistic concurrency control. The optimistic data control approach is founded on the principle that each modification to a dataset is recorded using a designated ETag field. Most often, the ETag field contains a timestamp, a hash value, or any other versioning that precisely identifies the version of the data set.

**Pessimistic Concurrency Control (Locking)**

Pessimistic concurrency control involves the exclusive locking of datasets. Another user cannot change the data set that is being modified by one user at the same time. In managed scenarios, the business object framework handles all locking tasks.

**Draft handling**

Draft-enabled applications enable users to save modified data in the backend and resume their work at a later time or on a different device, even if the application unexpectedly terminates.



Figure

5.

Draft

Draft is an option that you can use for application development with both, the managed and unmanaged implementation type and with mixed scenarios, for example, managed with unmanaged save. In all scenarios, the draft is managed. That means, it is handled by the RAP runtime framework. Furthermore, RAP provides implementation exits for situations where you require draft capabilities tailored to specific business services that influence how drafts are managed.

**Numbering (Early)**

Numbering is about setting values for primary key fields of entity instances during runtime. There are two types of numbering, Early and Late numbering. Managed scenarios support only Early Numbering. The term 'early numbering' in numbering types signifies the assignment of a value to the primary key field at an early stage. This key value can originate from either the consumer (externally) or the framework itself (internally).

**External Numbering**

We refer to external numbering if the consumer hands over the primary key values for the CREATE operation, just like any other values for non-key fields. The runtime framework, whether it's managed or unmanaged, assumes control of the value and manages it until it is ultimately written to the database.

**Internal Numbering****Managed:**

When employing managed numbering, the RAP managed runtime automatically generates a UUID during the CREATE request.

**Unmanaged:**

The concept of unmanaged early numbering is established at the entity level of the business object with early numbering. The implementation is applied to all keys of an entity, so that all key fields must be mapped in the implementation. The signature of this handler method is defined by the keyword FOR NUMBERING, followed by the input parameters entities, the implicit changing parameters reported, failed, and mapped.

**Authorization Control**

In RAP, authorization control safeguards your business object from unauthorized data access.

### Authorization Definition

#### Authorization Master

An entity is defined as authorization master (authorization master ()) if the operations of this entity have their own authorization implementation.

#### Authorization Dependent

An entity is considered authorization-dependent (designated as 'authorization dependent by \_Assoc') when the authorization controls established for the authorization master entity are intended to be enforced for the operations involving this entity as well.

#### Global Authorization

By defining global authorization (authorization master (global)), you implement authority control for the following operations of the entity: Create, Create-by-association, Update, Delete, Static Actions and Instance Actions.

#### Instance Authorization

By defining instance authorization (authorization instance ()), the following operations of the entity can be checked against unauthorized access: Create-by-association, Update, Delete and Instance Actions.

## Operations

Each entity of a business object may offer few set of operations. They can cause business data changes that are performed within a transactional life cycle of the business object.

### Standard Operations

1. Create Operation
2. Update Operation
3. Delete Operation
4. Locking (refer Pessimistic Concurrency Control)

## Actions

An action's typical purpose is to modify specific fields within a business object entity. When employing an action, it doesn't trigger the standard update operation; instead, it invokes the action with its predefined update implementation.

To execute an action, a corresponding trigger is necessary. Actions can be called by

1. A service consumer, for example, by a user interface.
2. Internally, for example, by another action or by a determination via EML.
3. By other business objects via EML

## Functions

Functions are designed to execute calculations or read operations on business objects without generating any unintended side effects. They do not acquire locks on database tables, and any computed data within a function implementation cannot be altered or persisted.

## Determinations

An implicit invocation of a determination occurs within the business object's framework when the determination's trigger condition is met. Trigger conditions have the capability to modify operations and fields. These conditions are assessed at trigger time, a predetermined juncture during the runtime of the business object. Once invoked, a determination can perform computations on data, make modifications to entity instances based on the outcome of the computation, and relay messages to the consumer by conveying them to the corresponding table within the REPORTED structure.

### Key Points:

1. The determination's outcome must remain consistent when the determination is executed multiple times under identical conditions (idempotence).

2. The order of execution for determinations is not predetermined. When multiple determinations are triggered by the same condition, the order of execution cannot be predetermined.
3. Following its triggering, determination must operate independently of other determination.
4. In situations where an instance is created or updated and subsequently deleted within the same request, there is a possibility of encountering an issue with EML read operations in a determination during modification, as instances with the provided key may not be locatable.

### Validations

A validation is automatically triggered by the business object's framework when the validation's trigger condition is satisfied. These trigger conditions have the capacity to alter operations and modified fields. The evaluation of these conditions takes place at the trigger time, a predetermined juncture during the runtime of the business object. Once invoked, a validation can prevent the storage of inconsistent instance data by flagging the keys of unsuccessful instances and placing them within the corresponding table in the FAILED structure. Furthermore, validations can furnish messages to the consumer by relaying them to the appropriate table within the REPORTED structure.

### Save Sequence (Managed)

The save sequence is an integral component of the business object runtime and is invoked after at least one successful modification has been made during the interaction phase.

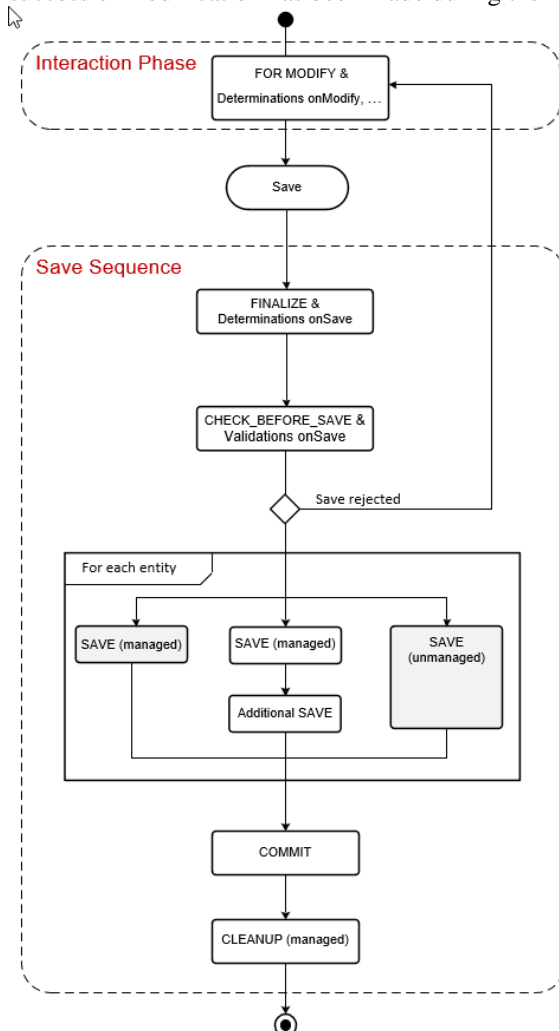


Figure 6. Save Sequence

### FINALIZE

In managed business objects (BOs), the ultimate validation for all related BOs is conducted.



## CHECK\_BEFORE\_SAVE

In managed BOs, the final check for all involved BOs is done via validations.

## ADJUST\_NUMBERS

This is applicable only for Unmanaged scenarios via late numbering.

## SAVE

In managed business objects, the act of saving is executed by the managed BO provider. If the business scenario demands an alternative saving mechanism, the option exists to define an additional or unmanaged save within the managed scenario.

### Additional Save

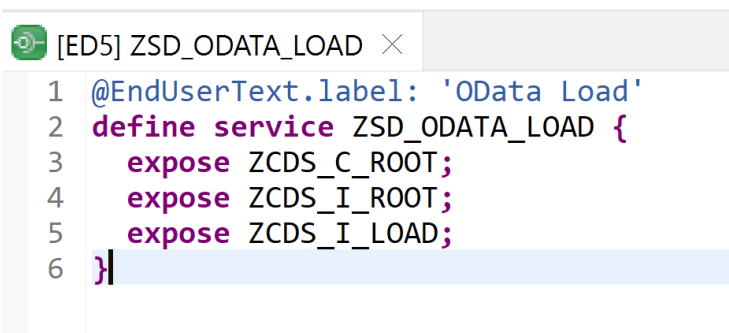
In certain application scenarios, external functionalities must be triggered during the saving process. This occurs after the managed runtime has recorded the modified data of business object instances in the database but prior to the execution of the final commit work. One example is the necessity to activate reuse services such as change documents and the application log within the save sequence, which also involves writing the changes from the current transaction to change requests.

1. The managed runtime will write the changed data of business object's instances to the database.
2. Additional save operations can be made like updating a change document before the final commit work has been executed.
3. The addition of further save actions can be achieved by utilizing the 'with addition save' option within the behavior definition.
4. Implementation can be achieved using the `SAVE_MODIFIED` method from the `CL_ABAP_BEHAVIOR_SAVER` class. This method features distinct parameters for all three operations – Create, Update, and Delete.
5. All errors should be captured in Validations onSave and only database update operations should be performed in the additional save ( update task function module )
6. The commit to database will be triggered by the managed runtime after the managed runtime database update and additional save. Commit or Rollback keywords should not be used.
7. The `SAVE_MODIFIED` method does not have the parameter `ERROR` to report errors and stop the database update.
8. The `MESSAGE` statement is not allowed directly inside the `SAVE_MODIFIED` method, it can be called via a function module or method.

## 2.2 Service Definition

A service definition, as a CDS object, serves as an interface that presents CDS entities via CDS source code, accessible through the business service. The source code for a service definition is encapsulated within a set of curly braces { ... }, acting as a container grouping together the pertinent CDS entities, including their associations with relevant entities, that are intended to be accessible as a unified service.

Each specific CDS entity to be made accessible is identified using the `EXPOSE` keyword, followed by an optional alias name introduced by the `AS` keyword.



```

1 @EndUserText.label: 'OData Load'
2 define service ZSD_ODATA_LOAD {
3   expose ZCDS_C_ROOT;
4   expose ZCDS_I_ROOT;
5   expose ZCDS_I_LOAD;
6 }

```

Figure 7. Service Definition

## 2.4 Service Binding

The business service binding, often referred to as service binding, is an ABAP Repository entity employed to associate a service definition with a client-server communication protocol, such as OData.

As illustrated in the diagram below, a service binding directly depends on a service definition that originates from the underlying CDS-based data model. Multiple service bindings can be generated from an individual service definition. This division between the service definition and the service binding enables a service to effortlessly integrate various service protocols without requiring any re-implementation. Services created in this fashion maintain a distinct separation between the service protocol and the underlying business logic.

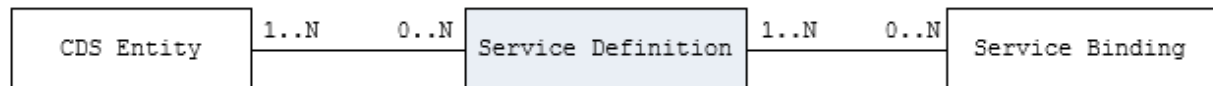


Figure 8. Service binding dependency [2]

### Binding Type

The binding type defines the service category and the particular protocol that the service binding implements.

OData in versions 2 and 4

OData V4 services have a wider range than OData V2 services. Use OData V4 wherever possible for transactional services.

OData for UIs or Web APIs

OData for UIs is designed for access to business services using UI technologies like SAPUI5. The data provided by a business service contains control elements for user interfaces.

OData for Web APIs is limited to the data-only content of the business services and does not contain any control elements for user interfaces.

Service Version

The versioning of services is made by a version number which is assigned to a service binding.

The next incremented version is established by introducing an additional service definition into the existing service binding. This additional service definition exposes functional changes or extensions in comparison to the previous version. And, vice versa, the version number can be decreased by removing a service definition from the service binding.

Publishing

To make the local service endpoint of an OData service accessible, it must be published using the 'Publish' button in the service binding editor. This triggers several task lists to enable the service for consumption. By Publishing the service binding enables the service exclusively for the current system. It is not consumable from other systems.

In an on-premises system, the service binding can solely be employed to enable your service locally and conduct testing within the development system. Transporting the service binding with all related artifacts generated locally is impossible. To enable your business service in a qualification or productive system, you have to disable the service in your local system (Unpublish button). Then use Gateway tools to activate and maintain your service.

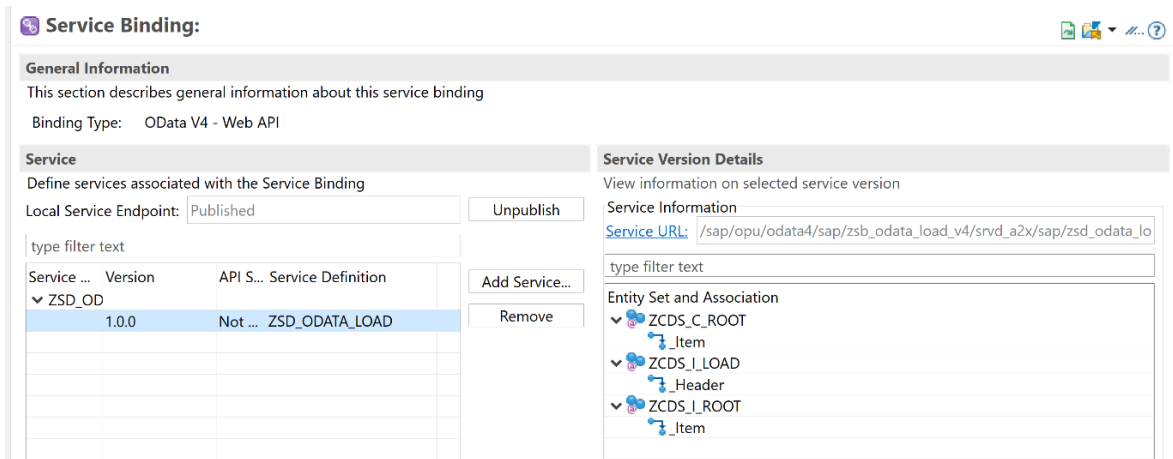


Figure 9. Service binding

### 3. Conclusion

The ABAP RESTful Application Programming Model (RAP) is a dynamic and extensible framework that profoundly influences enterprise application development. This organized architecture enables developers to construct new, cloud-compatible apps across multiple domains, connecting on-premises and cloud-based solutions. RAP's dependence on Core Data Services (CDS) for data modeling and its connection with the OData protocol for service exposure underscore its adherence to industry best practices and standardized development processes.

The comprehensive examination of RAP's data modeling and behavioral layer underscores the essential function of CDS in delineating domain-specific business entities and their interactions. The detailed analysis of elements including root views, child views, behavior definitions, and attributes such as optimistic and pessimistic concurrency control, draft management, and authorization control highlights RAP's strength in managing intricate business situations.

Moreover, the discourse around service definition and service binding highlights RAP's capacity to effortlessly integrate with many communication protocols such as OData, addressing both UI and Web API needs. The notion of versioning and publishing highlights RAP's adaptability in overseeing service evolution and facilitating regulated deployment across various system environments.

### References

- [1] Stefan Haas, Bince Mathew, "ABAP RESTful Programming Model – ABAP Development for SAP S/4HANA".
- [2] [https://help.sap.com/docs/ABAP\\_PLATFORM\\_NEW/fc4c71aa50014fd1b43721701471913d/289477a81ecc4d4e84c0302fb6835035.html](https://help.sap.com/docs/ABAP_PLATFORM_NEW/fc4c71aa50014fd1b43721701471913d/289477a81ecc4d4e84c0302fb6835035.html).