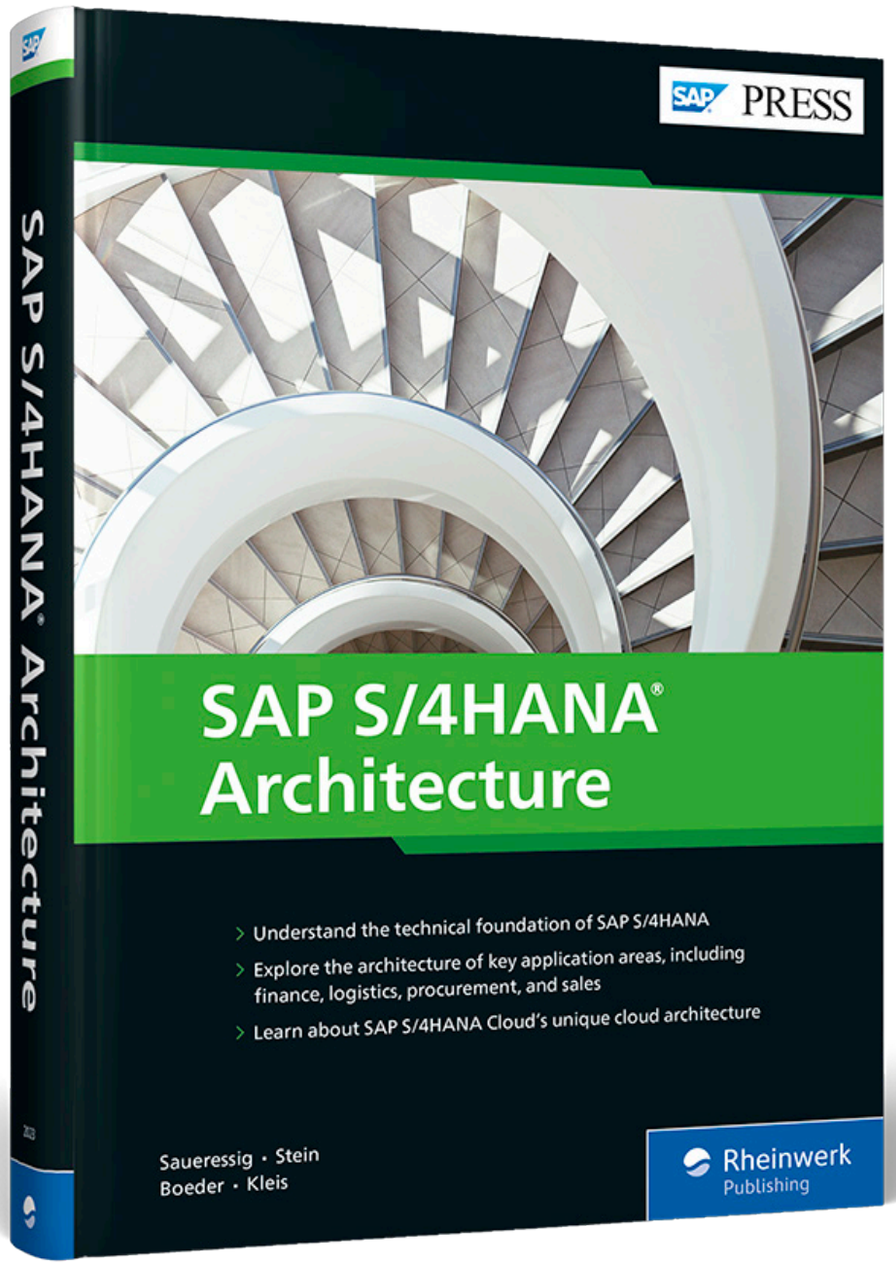


Browse the Book

Technical Architecture Foundation describes the technical foundation of SAP S/4HANA, including a look at the virtual data model and the ABAP RESTful application programming model. This chapter shows the reader the essential parts of a business application and how the semantics function on a technical level.

Thomas Saueressig, Tobias Stein, Jochen Boeder, Wolfram Kleis

SAP S/4HANA Architecture

520 Pages, 2021, $79.95
ISBN 978-1-4932-2023-6

www.sap-press.com/5189

Chapter 2
Technical Architecture Foundation

Source code and data type definitions contain no information about development or architecture decisions. The virtual data model and the ABAP RESTful application programming model break with this paradigm: they focus on the essential parts of a business application and make their semantics transparent.

Now it's time to get technical. We have mentioned the virtual data model (VDM) already a few times. In this chapter, we explain in detail what it is and how it is built. In particular, we explain how the core entity of transactional ERP applications—the business object—is modeled according to the VDM using CDS views. Then we show how the ABAP RESTful application programming model supports developers in implementing ERP applications using these business objects.

2.1 The Virtual Data Model

The *virtual data model* represents the semantic data model of the business applications in SAP S/4HANA. It aims at exposing all the business data in a way that eases its understanding and consumption. It is called a *virtual* model because it abstracts data from the database tables. This way, existing tables can be transformed into an aligned uniform data model where necessary.

Besides describing the business semantics of the applications, the VDM is also an executable model that provides access to the corresponding data at runtime. The VDM is implemented using specifically classified Core Data Services (CDS) entities, which comply with the VDM rules (Section 2.1.1). Among others, these VDM rules include fundamental naming rules (Section 2.1.2) and rules for structuring the VDM entities (Section 2.1.3). On the one hand, the rules foster the consistency of the models; on the other hand, they allow efficient development of applications and APIs that cover analytical consumption (see Chapter 4, Section 4.1), transactional processing (Section 2.2), and search-related consumption scenarios (see Chapter 3, Section 3.2). In addition to providing the data model for applications, VDM views are also used for other tasks, such as CDS-based extraction (see Chapter 6, Section 6.8.1).

The VDM is not intended only to be used for developing applications at SAP. Instead, released SAP VDM models and released SAP services built on them offer a stable interface with a well-defined lifecycle. As an SAP customer or partner, you can use them to build your own applications and for enhancing SAP applications.

2.1.1 Core Data Services

Core Data Services (CDS) support the definition of semantically rich data models. These models are managed by the Data Dictionary of the ABAP platform and can be executed in the database system.

CDS views represent the most important CDS entity type. They capture `select` statements in a syntax that is closely related to that of structured query language (SQL). CDS views can be used, for example, as the data source in ABAP `select` statements. The results of querying a CDS view can be restricted by attaching access control models to the CDS view. This means that the query only returns the data that the current user is authorized to read. These access control models are defined using the CDS *data control language* (DCL). CDS views support the definition of supplementary metadata by means of annotations. Furthermore, CDS views allow modeling associations, which represent directed relations to other CDS entities. Both the annotations and the associations are interpreted by the various consumers of the CDS models. In specific, the ABAP infrastructure uses the corresponding information for deriving additional functionality and services from the CDS models. For example, a CDS view annotated accordingly can be executed by the analytic engine (Section 2.1.4). The analytic engine provides advanced features such as exemption aggregations and hierarchy handling, which are not modeled by the plain `select` statement of the CDS view itself.

2.1.2 Naming Conventions

The VDM is based on a set of common naming rules. Applying the following rules ensures consistent and self-explanatory naming:

- A name is precise and uniquely identifies a subject. Generic names are avoided. For example, the identifier of the sales order document is named `SalesOrder`, not just `ID` or `Order`.
- A name captures the business semantics of a subject.
- Names being unique implies that different subjects must have different names. This is specifically important for identifiers and codes. Using the same name for two fields implies that their underlying value lists match.
- Names are composed from English terms in camel case notation with an uppercase first letter. Underscores are used in predefined cases only. Abbreviations are avoided.

The naming rules are applied to all CDS entities and their parts—for example, names of fields, associations, parameters, or CDS views.

Figure 2.1 illustrates an example of an SAP Fiori app that uses an OData service, which in turn is based on a VDM view stack. As you can see, the first VDM view (at the bottom) maps the technical field name `MATNR` from the database table onto the semantic name `Product`, which is then used on several layers up to the user facing SAP Fiori UI.

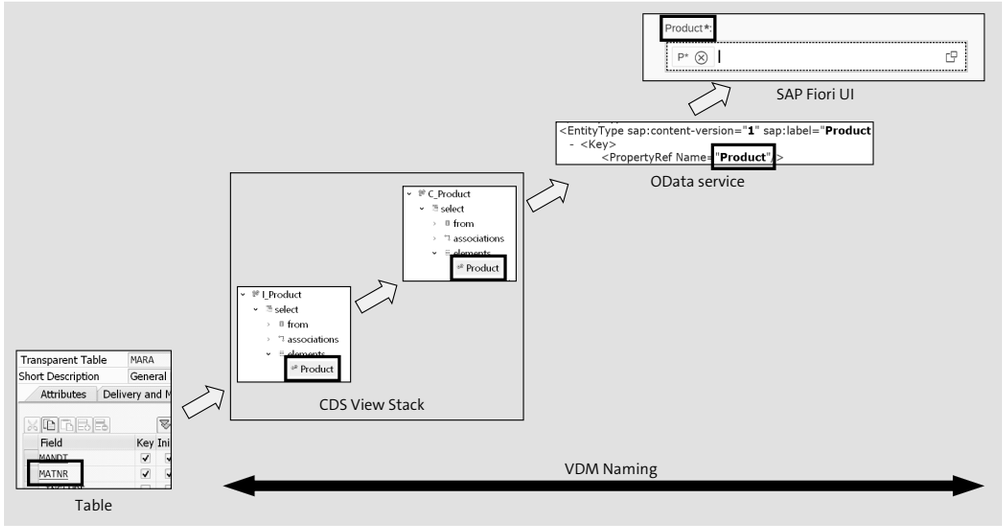


Figure 2.1 Field Names in VDM

2.1.3 Structure of the Virtual Data Model

Within the VDM, CDS entities serve distinct purposes. They are classified accordingly by means of VDM annotations. Note that a CDS entity becomes a VDM entity if it uses VDM annotations and if it adheres to the VDM guidelines.

VDM views are organized in a hierarchical structure following a layered approach. The upper layers select from and define associations to the same or lower layers, but not vice versa. The VDM views are assigned to layers with a special CDS annotation (`@VDM.viewType`).

Figure 2.2 depicts the admissible dependencies between the views as data sources and the different types of views. Along with the dependencies, the typical prefixes of the names of the CDS views are visible here too. Consumption views have name prefix `C_` and remote API views have the prefix `A_`. Basic views and composite views form an interface layer that can be used for building applications. The views of this interface layer have name prefix `I`. As we will explain in the following sections, basic views and composite views can also be restricted reuse views, indicated by name prefix `R_`.

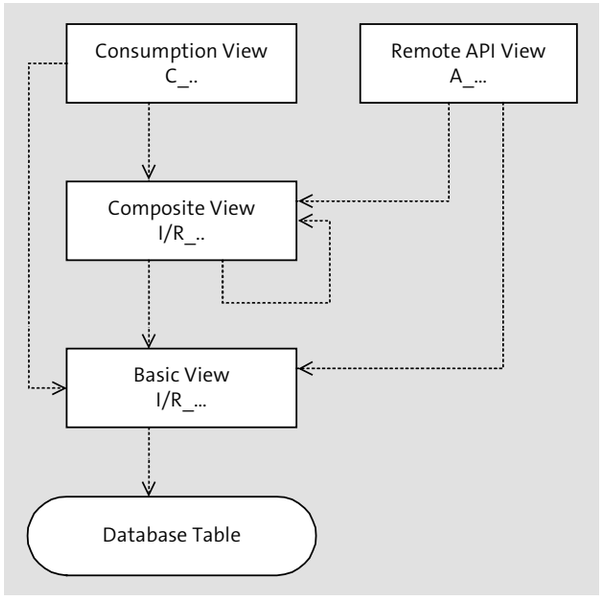


Figure 2.2 View Layering: Select-from Relationships

Basic Views

The lowest level of the VDM view stack is defined by *basic views*. Basic views are the most important constituents of the VDM. They establish what you may call the *entity relationship model* of the applications, from which they get information about data structures, dependencies, and metadata. The main purpose of the basic views is to serve as reusable building blocks for any other nonbasic VDM views. Because the basic views expose all data, there is no need for any other nonbasic view to directly select from the database tables. In fact, accessing database tables from nonbasic VDM views is forbidden in the VDM. This way, basic views provide a complete abstraction from the database tables to higher layers.

Composite Views

Composite views comprise additional functionality on top of the basic views. Like the basic views, they are primarily intended to serve as reusable building blocks for other views. However, they can already be defined in such a way that they support a specific consumption domain. For example, they can define analytical cube views, which consolidate data sources for usage in multiple analytical queries.

Transactional Views

Transactional views are a special flavor of composite views. Transactional views define the data model of a business object and act as an anchor for defining its transactional processing-related aspects (Section 2.2.1). Transactional views may contain elements

that support the transactional processing logic like additional fields that help preserving user input temporarily. Therefore, transactional views are only used in the context of transactional processing—for example, when consumed by other transactional views or by consumption views, which delegate their transactional processing logic to the transactional views.

Consumption Views

Regular basic views support any use case. This means they are defined independent of a specific use case. In contrast, *consumption views* are deliberately tailored for a given purpose. Consumption views are expected to be directly used in a specific consumption scenario. For example, a consumption view can provide exactly the data and metadata (through annotations) that is needed for a specific UI element.

Restricted Reuse Views

By default, basic and composite views define an interface layer (named with the `I_` prefix), which any SAP application may use. Once released, they can also be used by SAP customers and partners. However, development teams sometimes define basic and composite views that are only for local use in their own applications. Such views are not meant to be reused by developers from other application areas. In such a case, a *restricted reuse view* (named with the `R_` prefix) is defined. Examples of restricted reuse views are transactional processing-enabled views, which expose all functions and operations of a business object, including the internal ones.

Enterprise Search Views

Enterprise search uses special CDS views as search models. Enterprise search is described in Chapter 3, Section 3.2.

Remote API Views

Remote API views project the functionality of a single business object for external consumption. They decouple the regular, system-internal VDM model, which can evolve over time, from its external consumers, establishing a stable interface. Based on the remote API views, OData services are defined, which can be consumed by remote applications. The corresponding OData services are published on the SAP API Business Hub (*https://api.sap.com*).

2.1.4 Consumption Scenarios

The most prominent consumption scenarios supported by CDS entities are as follows:

- Analytics
- Enterprise search
- SAP Fiori UI applications
- Remote APIs

Figure 2.3 shows a typical VDM view stack for analytical and transactional processing-enabled applications. Use case-specific views are defined on top of basic and composite reuse views to adapt the data model and functionality to the individual application needs.

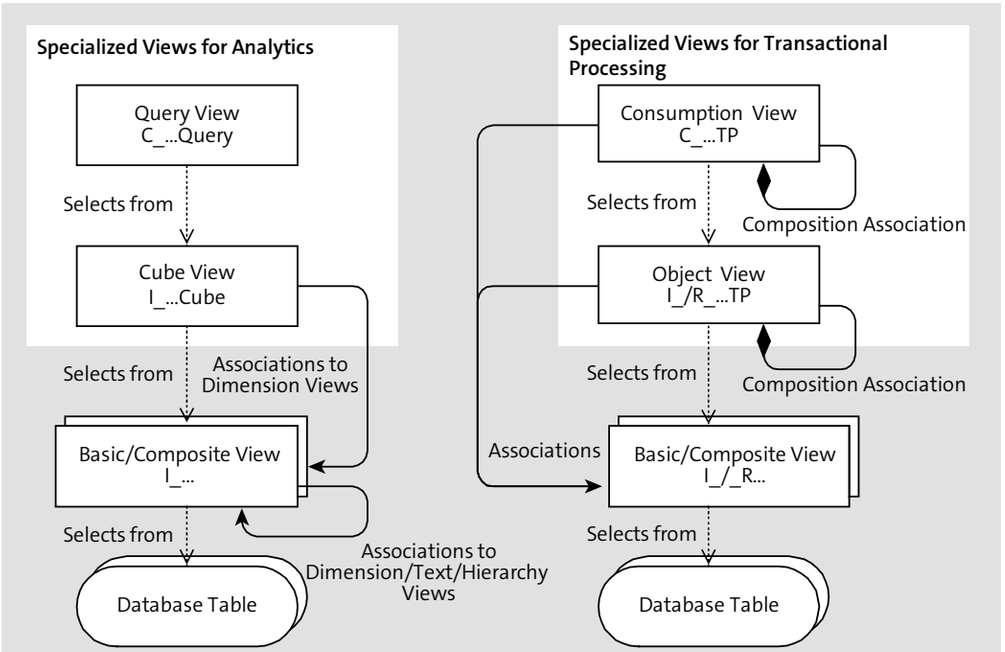


Figure 2.3 View Hierarchies

Analytical Applications

Analytical applications are based on cube views and a network of associated dimension views, which themselves can associate text and hierarchy views. The actual analytical application is defined by an analytical query view, which projects the envisioned functionality from its underlying cube view. Note that unlike other CDS views, the analytical query views themselves are not executed on the database. Instead, the analytic engine interprets their logic and directly executes selections from the cube and dimension views. Analytical applications are covered in detail in Chapter 4, Section 4.1.

Transactional Processing-Enabled Applications

The data model is captured by transactional views, which are related through compositional associations to form an entire business object. Therein the actual data selection logic is defined by the CDS view models. The behavior model specifying the supported

Create, Read, Update, Delete (CRUD) operations is defined in an attached *behavior definition model* and implemented in ABAP classes. The consumption view projects relevant functionality and augments the data model with annotations that provide the metadata for rendering the UI application built on the view. Associated interface views can be used to enrich the data model where appropriate. The recommended way to develop transactional applications is using the ABAP RESTful application programming model, which we discuss next in Section 2.2.

2.2 ABAP RESTful Application Programming Model

The *ABAP RESTful application programming model* is the common foundation on which the programming model of SAP S/4HANA is based. The strategy of SAP S/4HANA is to apply this target architecture across all applications. The model fosters SAP's general architecture goal to clearly separate the database model and database layer, the application logic (business logic) and the user interface. The SAP Fiori technology approach technically ensures the separation of the user interface (and related UX) from the business logic, leveraging OData as the protocol of choice. The ABAP RESTful application programming model addresses the modeling and implementation of application logic (often called *business logic*), including aspects that are common for all applications and the service-specific application logic. In addition, the ABAP RESTful application programming model ensures that the complete application implementation is protocol-agnostic and can also be reused when switching to a different protocol or protocol version.

In Section 2.2.1, we describe the common application model and logic, which we also refer to as a *business object* and the *business object implementation*.

The service-specific model and logic we often call a *business service* and the *business service implementation*. We explain its concept for OData, the main protocol used, in Section 2.2.2.

In Section 2.2.3, we describe the overall runtime architecture of the programming model.

2.2.1 Defining and Developing Business Objects

A business object is defined by a data model and its behavior. Figure 2.4 shows the design-time artifacts for defining and implementing business objects with the ABAP RESTful application programming model. The individual concepts and artifacts are explained ahead.

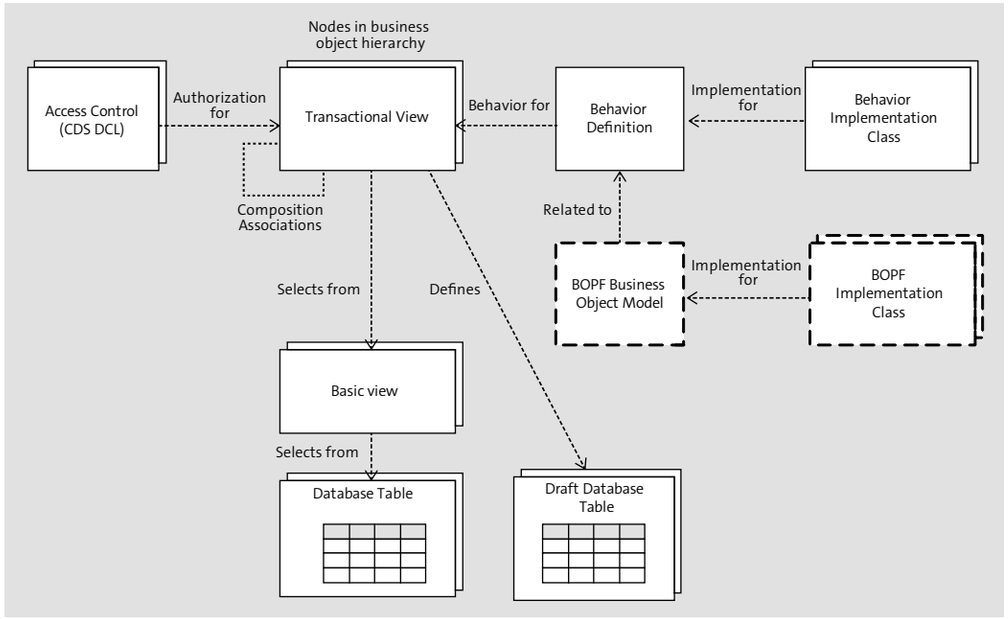


Figure 2.4 Business Object Definition and Implementation

Data Model

The data model of a business object consists of entities, also known as *business object nodes*, which are arranged in a hierarchical compositional tree structure. The data model is defined using CDS entities that are based on basic views of the virtual data model, which we introduced in Section 2.1. In Figure 2.4, the CDS entities for defining business object nodes are called *transactional views*.

In CDS models, *compositions* are a special kind of association that represent a parent-child hierarchical relationship, in which the child is a part of the parent and cannot exist without it. When defining business objects, compositions are especially important because they are needed to construct the business object model from nodes.

An example of a compositional structure is the sales order as a business object that is composed of the sales order header root entity, its sales order item children, and its sales order schedule line grandchildren. Usually we do not distinguish between the business object and its root entity when referring to them, but to avoid misunderstandings, *sales order* refers to all entities within the composition structure, whereas *sales order header* refers to the root entity only.

Other associations to related entities are also relevant and belong to the business object definition but are not part of the business object itself. Such associations can point to other business objects or business object entities—be it for read access, for transactional access, or for the purpose of value helps.

In the example of the sales order, a transactional association within the sales order might be a specialization of the composition, for example `toFree GoodsSalesOrderItems`. Examples of associations to other business objects include associations to the business partner on the sales order header level or to the product on the sales order item level. Usually, the association to the business partner is also a transactional association; during the sales order entry, a new business partner instance may be created as a customer, or the customer data can be updated—for example, with a new address. The association to the product usually is not transactional because during the sales order entry no products are created or changed.

CDS entities can be accessed at runtime with SQL queries. The read authorization for such a query access is defined and modeled in a declarative way with a related access control model defined in CDS data control language (DCL) for each CDS entity.

The data model also defines the semantics of the fields based on the data types used:

- Because the ABAP type system doesn't support certain types, like `Boolean`, `DateTime`, or the Universal Unique Identifier (UUID), we use dedicated annotations on the field level to indicate that, for example, a `character` field with length 1 is a `Boolean`. When exposing these entities via OData, this annotation is evaluated because the OData type system is different and supports such corresponding types (i.e., `Edm.Boolean`, `Edm.DateTimeOffset`, and `Edm.Guid`).
- CDS annotations can add additional semantics on top of the pure data type definition. Such annotations can, for example, define that a field with type `DateTime` contains the time when the data was created or last changed, or that another field of type `Date` contains the "from" date on which some business data becomes valid or the "to" date until when business data is valid.

Developers define value helps or references to texts for IDs or codes as part of the data model using CDS annotations too.

Behavior Model and Implementation

The behavior of a business object is modeled with an artifact called *behavior definition*. The behavior definition is based on and complements the underlying data model. Technically, it's assigned to the root view entity that represents the entire business object, and it inherits its name from the root view entity.

The behavior definition defines the transactional behavior of the business object by declaring the available operations for each entity of the business object. This could include common operations such as create, update, and delete, or application-specific actions and functions. In addition, the behavior definition specifies runtime behavior with regard to locking, entity tag (ETag) handling in the case of HTTP-based consumption (typically through OData), authorization, feature control for the provided operations, support of draft handling, and much more.

Locking, ETag definition, and authorization within a business object usually are not specific to every entity but apply to the complete business object or a compositional subtree. To support this modeling for locking, ETags, and authorization, entities can be defined as `MASTER` or `DEPENDENT`. Thus locking, ETags, and authorization are invoked on the `MASTER` entity even if a `DEPENDENT` entity is accessed. For example, the sales order header is the `LOCK MASTER`, and all other entities are `LOCK DEPENDENT`. If a sales order item in a sales order is changed, the `LOCKING` is on the sales order header.

Feature control refers to the availability of operations (create, update, delete, actions, functions) or the changeability of fields. We distinguish between static and dynamic feature control, while the dynamic feature control is again split into global and instance feature control:

- For operations, *static* means that the operation is or is not available for consumers. For example, if creation of an instance is never allowed directly but only through a factory action, the create operation might not be defined at all or marked as *internal*.
- For fields, *static* means, for example, that a field is read-only and thus cannot be changed by a consumer, but only within the business object implementation.
- *Global* means that the operation is controlled independent of the instance. This is the only option for noninstance operations like create on the root entity level. Global can be applied to instance operations or fields, too—for example, when the action is deactivated based on certain configuration settings.
- *Instance* means that the operation or the field is controlled based on the concrete instance. For example, if a sales order is already released, the release action for that sales order is no longer enabled. At the same time, the delivery address cannot be changed further and is set to read-only dynamically for this sales order.

Some of the information specified in the behavior definition is relevant for the consumer of the business object. It defines, for example, the interface of the business object that can be used from ABAP code with the *entity manipulation language* (EML). EML is integrated into the ABAP language and allows reading and modifying business objects based on the operations defined in the behavior definition. Other aspects of the behavior definition are just implementation details. The implementation type in particular is an implementation detail that defines how the business object provider is implemented.

To ensure that the ABAP RESTful application programming model can be used across all applications in SAP S/4HANA, it supports three different implementation types:

- Managed implementation
- BOPF managed implementation
- Unmanaged implementation

Now we'll explain each type.

Managed Implementation Type

The *managed implementation type* is chosen for greenfield development when new applications are implemented or existing applications are refactored. Here, the infrastructure of the ABAP RESTful application programming model provides the main part of the provider implementation. This includes a full CRUD implementation with a transactional buffer, as well as read and write access to the application database tables. The *transactional buffer* is the buffer that holds the transactional changes that are processed during the transaction. When finalizing the transaction and saving the data, these changes are taken and saved to the database. In a managed implementation, the application only deals with its own application logic. The ABAP RESTful application programming model runtime takes care of buffer handling, locking, and so on.

The application logic is defined using determinations and validations that are declared in the behavior definition and can be implemented with application-specific ABAP code:

- A *determination* is application logic that changes data triggered by other changes. For example, the sales order item amount is calculated based on changes to the order quantity or the unit price. A determination might run immediately on modification or only later on saving.
- A *validation* is application logic that checks the consistency of the data in the transactional buffer to ensure only proper data can be finally stored in the database table. A validation runs only on saving. The infrastructure ensures that only relevant determinations and validations run based on so-called trigger definitions.

Available triggers are create, update, delete, and field changes. So, the determination to calculate the sales order item amount only runs if the order quantity or the unit price changes and only for the sales order item that was changed. For a good user interaction, it is sometimes necessary to invoke certain determinations or validations during the interaction phase. To do so, developers define *determine actions* in the behavior definition and assign the determinations and validations to these actions. The consumer can then call the determine action to invoke the determinations and validations.

Developers also use the managed implementation type when they refactor existing code and, for example, existing database tables need to be reused or need to be kept stable for interoperability. In this case, additional exits can be implemented to affect how changes are saved, to add additional save operations—for example, to write change documents—or to implement the lock method to reuse an existing enqueue object.

BOPF-Managed Implementation Type

The BOPF-managed implementation type is quite similar to the managed implementation type but provides the option to reuse and integrate an existing CDS-based BOPF

implementation. The Business Object Processing Framework (BOPF) is a predecessor of the business object framework of the ABAP RESTful application programming model. Before the ABAP RESTful application programming model was introduced, SAP recommended the CDS-based BOPF programming model. With the BOPF-managed option, the corresponding investments are protected and can continue to be used. The functionality of the BOPF-managed implementation type is comparable to the managed implementation type. The transactional buffer handling and further features are provided by BOPF, while applications focus on application logic in determinations and validations. Note that for the BOPF-managed implementation type, the determinations and validations are declared not in the behavior definition, but in the BOPF model as separate design-time artifacts. The link between the behavior definition and the BOPF model is established by naming, as both artifacts have the name of the business object as their name. In Figure 2.4, the artifacts of the BOPF model and the BOPF implementation are shown as dotted boxes because they are specific to the BOPF-managed implementation type. Note that the BOPF implementation replaces the behavior implementation in this case.

Unmanaged Implementation Type

The unmanaged implementation type is chosen for brownfield development when an existing application implementation should be reused and a refactoring or reimplementation to a managed implementation is not an option. In this case, the application is responsible for the complete provider implementation. This means that the application developers must implement all supported operations, such as create, delete, update, and save, on their own. This allows developers to reuse their existing application and delegate the work to the existing application APIs. The unmanaged implementation type is also the current option for traditional BOPF implementations. When developing a new application from scratch, the recommendation is to use the managed implementation type instead.

Comparison of ABAP RESTful Application Programming Model Implementation Types

Developers implement the operations and features of a behavior definition with dedicated ABAP classes. These classes refer to the behavior definition and thus have their method signatures inferred. This applies for the managed and the unmanaged implementation type, but developers must implement different parts in each case. In both use cases, developers must implement actions, functions, authority checks, and feature controls for their business object. In the unmanaged case, the application developer also must program the create, update, and delete operations, the transactional buffer handling, the locking, and the save sequence.

For the managed implementation type, the create, update, and delete operations with buffer handling come for free and cannot be implemented at all by application developers. For locking, the managed scenario offers a generic implementation that can be replaced if needed (with the `UNMANAGED LOCK` option). The ABAP RESTful application programming model framework takes care of saving of data if the database tables correspond to the entity definitions or simple mapping is defined in the behavior definition. If this isn't sufficient, developers can replace the generic implementation (with the `UNMANAGED SAVE` option).

A big benefit is that these ABAP RESTful application programming model features are integrated into the ABAP syntax and thus the signatures are fully typed. With this setup, design-time syntax checks and code completion are supported out of the box.

In the BOPF-managed case, developers implement the operations and features of a behavior definition in the BOPF implementation classes defined in the BOPF model. Because BOPF is a generic framework, the signatures here are only partially typed. The signatures have generic data types only (`TYPE DATA`) because BOPF does not create specific interfaces for each business object.

The Draft Feature

The *draft feature* is a special feature that enables the application to store the user's editing state at any point in time. This feature is also supported from end to end for SAP Fiori UIs implemented with SAP Fiori elements. Thus, for UI-based functionality and services, the draft feature is a core element of the architecture. The draft concept provides the following features:

- Users can interrupt the editing or creation of a business object without losing data.
- Support is provided for switching between devices and browsers during editing.
- In the future, support will probably be available for collaboration of multiple users working on the same business object.

Furthermore, with the draft feature, SAP S/4HANA can follow a stateless paradigm (REST using OData), ensuring cloud qualities such as robustness and scalability, while using an exclusive lock and the invocation of the necessary backend application logic during editing.

The draft implementation usually follows the implementation type of the application. For the managed implementation type, the `with draft` addition enables the draft feature. Then the ABAP RESTful application programming model infrastructure takes care of the draft handling. For the BOPF-managed implementation type, it is similar, with the difference that the BOPF infrastructure is responsible for draft handling.

For the unmanaged implementation type, draft enablement currently only can be defined as managed by the ABAP RESTful application programming model infrastructure (`DRAFT MANAGED`) or by the BOPF infrastructure (`DRAFT BOPF MANAGED`). Thus, the draft handling is always part of the infrastructure. Draft enablement for the unmanaged

implementation type requires that the relevant parts of the application logic are reimplemented so that they can be called by the ABAP RESTful application programming model or BOPF infrastructure for managed draft handling.

With ABAP RESTful application programming model, transactions are split into two phases, the interaction phase and the save phase (for details, Section 2.2.3). The basic architectural idea is that the interaction application logic is the same for active instances and draft instances. The draft reflects a persistent state of the interaction and the transactional buffer.

The application can have additional application logic that is invoked during the save phase and is relevant for nondraft instances only. The draft instances are saved as well, but they do not run through the related application logic of the save phase. This means that draft instances can be saved with missing or inconsistent data, which would be prevented by validations for nondraft instances.

2.2.2 Defining Business Services

Business services provide external access to certain aspects of one or more business objects, either for consumption by the UI layer or as a remote API to be called by other systems. However, you do not define these services directly on top of the transactional business object models. Instead, application developers create service-specific projections of the underlying business objects with the aspects they want to expose through the business service.

Developers can choose entities from the business object's consumer model and project it for a specific service exposure. This includes entities, their fields and associations, operations, actions, functions, ETags, drafts, or restriction with static feature control. Provider-side features like authorization checks, locking, or dynamic feature control, as well as the application logic, cannot be projected, deactivated, or overwritten because the provider is in charge of its own consistency. Typically, you define different services for different purposes, with different service-specific projections. OData services for the UI typically include information such as value helps, texts, and UI annotations that can be evaluated by metadata-driven SAPUI5 controls. In addition, such services support the draft functionality.

OData services for remote APIs usually expose only the entities of one business object—for example, a transactional business document. Related objects, such as codes and their texts or master data objects and their texts, are not included but are exposed by their own services.

Figure 2.5 extends the diagram from Figure 2.4 with the design-time artifacts for defining and implementing business services (included in the grey box). This includes the service-specific data model and implementation and the service definition and binding. These concepts are explained in the following sections.

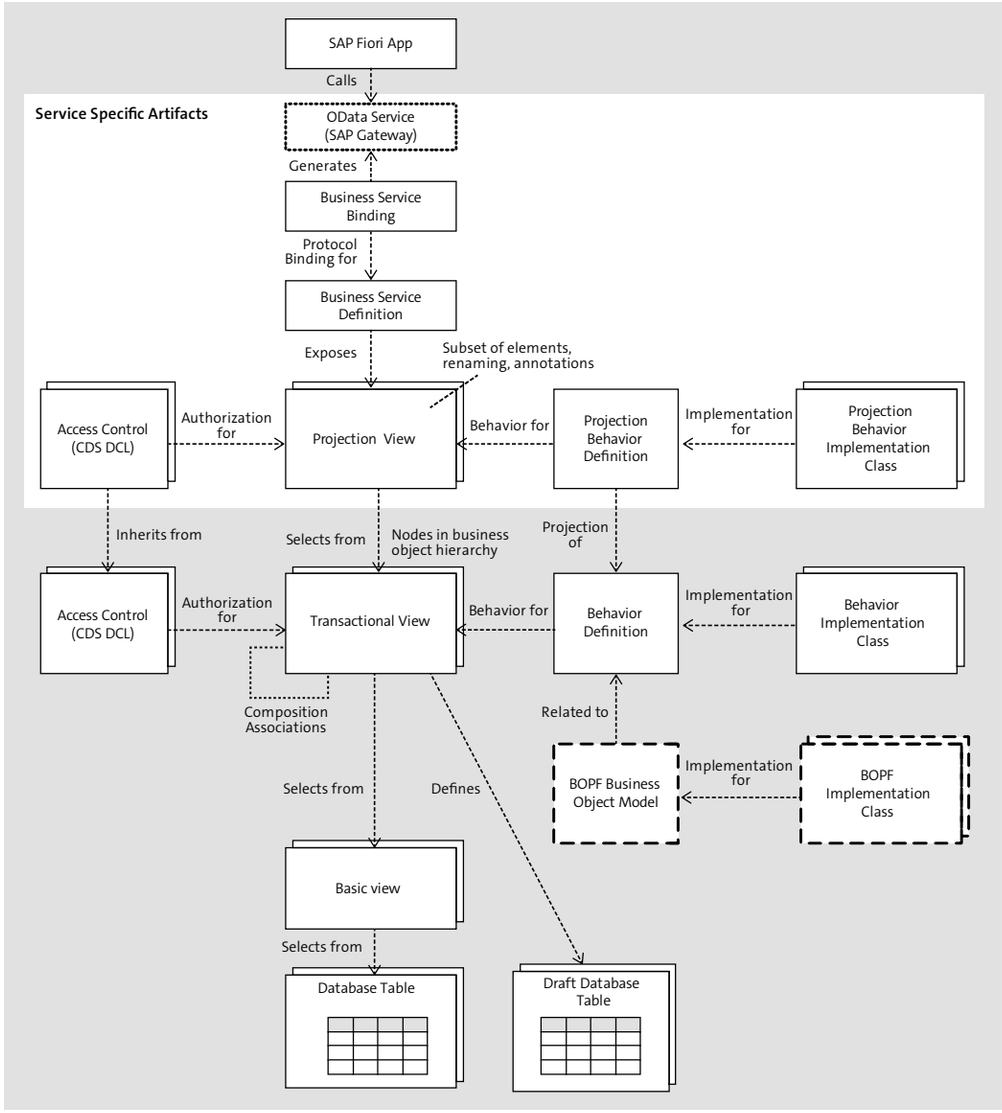


Figure 2.5 Service Definition and Implementation

Service-Specific Data Model

Developers define the business service-specific data model in two steps:

1. Select the entities to be included into the projection.
2. For each entity, decide which fields and associations are included.

In the ABAP RESTful application programming model, the projection is defined with special CDS entity types, called *projection view entities*. This special CDS entity type has

been introduced because not all CDS features are allowed in this kind of view. For example, aggregations, unions, and joins are not allowed. With a special CDS entity type, these restrictions can be enforced with syntax checks.

With projection views, you can not only restrict what entities and entity elements—columns and associations—are exposed. You can also restrict results that are returned by the view by adding a `WHERE` condition.

Sometimes the service layer is not a pure projection, but also needs to add certain additional service-specific elements to the data model. It is possible to add additional fields—currently fields that are calculated in ABAP only, also known as *virtual elements*—or to define new associations to service-specific entities on this service layer.

At runtime, CDS projection view entities can be accessed with SQL. Therefore, developers need to define the read authorization for such a query access and model it for each CDS entity with a related CDS access control definition (specified in data control language). In this case of transactional projection, it's only possible to "inherit" the access control definition of the underlying CDS entity. Otherwise, the authorization of the query access could be different than the authorization checked during the transactional access that is delegated to the underlying layer by definition (see the next section).

Behavior Model

In addition to the behavior definition of the business object, developers can define a projection behavior definition related to the projection root view entity of the projection. The main purpose of the projection behavior definition is to choose which features of the underlying business object behavior definition are exposed as the service-specific behavior. In the projection, only a part of the create/update/delete operations and the functions can be exposed, while others are omitted. In the same way, other features, such as support for draft or ETags, can be included or not. In addition, certain behavior can be restricted, mainly for static feature control. For example, a field that is editable in the business object might be set to read-only in the projection.

Developers can add certain service-specific implementations in the projection layer too. These could include additional checks to prevent certain data from being updated through this projection or additional transformations and augmentations that are not derived automatically by the infrastructure.

All other aspects of the behavior cannot be influenced with the projection behavior definition. It is, for example, not possible to change the implementation type or include or omit aspects like locking or authorization checks or to change or replace the application logic of the underlying business object.

Service Definition and Service Binding

To define an OData service on top of a projection view entity, developers have to add two additional artifacts, the service definition and the service binding:

- The *service definition* defines the scope of the service by listing the entities that are exposed by the service.
- The *service binding* references a service definition and specifies the protocol and protocol version with which the service will be created.

Currently the ABAP platform supports OData V2 and OData V4. The service binding also supports versioning of services. With this, published services can be kept stable by providing a new service version if incompatible changes to the interface are required.

For OData services, the entities of the CDS model are mapped to OData entity sets and entity types. The fields of the CDS entities are mapped to OData properties, and the CDS associations are mapped to OData navigation properties. Whether an OData entity supports `POST`, `PUT`, `PATCH`, or `DELETE` operations, in addition to `GET`, also is derived from the declared operations in the projected behavior definition. Furthermore, all exposed actions and functions of the behavior definition lead to OData actions and OData functions (or OData function imports in OData V2).

2.2.3 Runtime Architecture

Now, what happens at runtime? To illustrate the runtime architecture, we take an application with OData services that is built with the ABAP RESTful application programming model. The architecture is shown in Figure 2.6, with ABAP RESTful application programming model abbreviated as RAP, for better readability. The frontend sends OData requests, which are received by the OData service runtime and forwarded to the ABAP RESTful application programming model runtime through a protocol adapter. The runtime calls application-specific handlers—for example, for global authorization, global feature control, and locking of the lock master.

The ABAP RESTful application programming model runtime executes read requests directly using the corresponding views of the data models. For write operations, the runtime invokes the ABAP RESTful application programming model provider. As explained in Section 2.2.1, there are several implementation types for providers: managed, BOPF managed, and unmanaged.

The ABAP RESTful application programming model runtime accepts requests in entity manipulation language. In the ABAP RESTful programing model, EML provides the runtime access to the modeled entities and business objects. Because EML is part of the standard ABAP language and syntax, it allows a fully typed consumption of any ABAP RESTful application programming model-based implementation. EML allows for access to all external operations, like create, update, delete, any action, any function, the invocation of a lock, or the retrieval of data or feature control and authorization

information. With the `MODIFY ENTITIES` EML statement, several modifying operations of different types (such as create, update, delete, and action) can be bundled within one call—for example, to invoke multiple actions or to create a complete business object in one call and roundtrip.

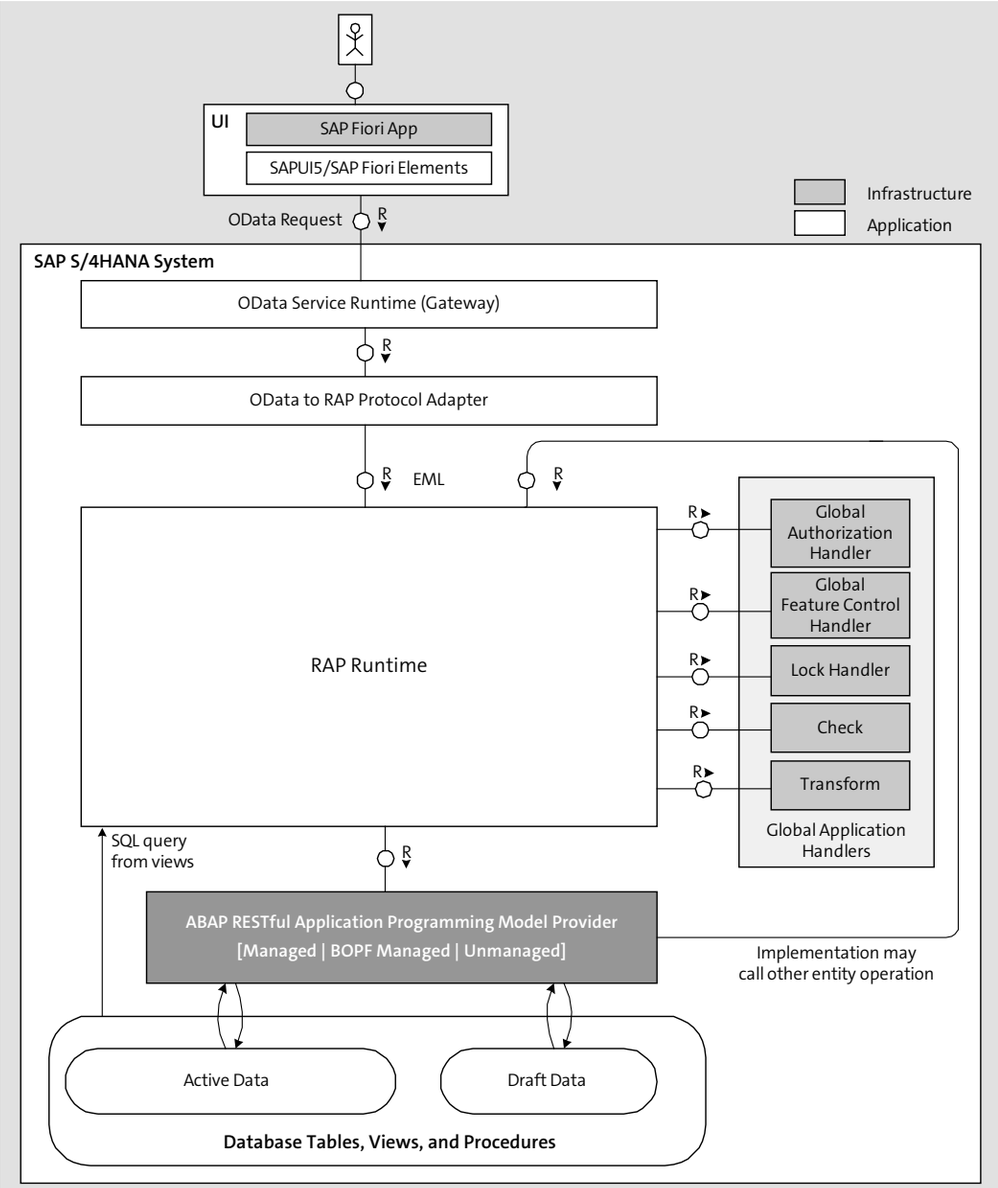


Figure 2.6 ABAP RESTful Application Programming Model Runtime Architecture Overview

Usually other applications or business objects, integration tests, generic consumption tools like data migration, or specific consumption tools like a SOAP API access the business object through EML. For OData, the ABAP platform provides end-to-end support, as mentioned in Section 2.2.2.

Phases of a Transaction in ABAP RESTful Application Programming Model

At runtime, a transaction defines the changes between two consistent saved states. This is also known as a *logical unit of work* (LUW) in the ABAP runtime. ABAP RESTful application programming model splits the transaction into the following two phases:

1. *Interaction*, in which one or multiple entities of one or multiple business objects are changed together
2. *Save*, in which further application logic runs, the data consistency is checked, and the changes are finally committed to the database

If the save phase is rejected due to an inconsistent final state, the transaction returns to the interaction phase to allow adjusting the data to reach a consistent state. Obviously, such an orchestration applies in an end user scenario only; while in machine-to-machine communication, the inconsistent data is not saved and the request is rejected.

Interaction Phase Operation Flow

If an operation is invoked (for example, through EML or OData), the internal processing follows a defined sequence that is partially enforced and provided by the ABAP RESTful application programming model runtime, partially to be implemented by the provider implementation.

Looking at the example of an action call for an instance-bound action, the following steps are performed to invoke this action:

1. Check global authorization. Is the user allowed to invoke this action at all?
2. Check global feature control. Is the action enabled at all?
3. Lock instance. Invoke the enqueue lock of the lock master.
4. Check ETags (only for HTTP requests). Does the provided ETag still match the current ETag?
5. Application-specific (pre)checks.
6. Application-specific transformations and augmentations. Only for projections.
7. Invoke actions of the underlying provider:
   - Check instance authorization. Is the user allowed to invoke this action for this instance?
   - Check instance feature control. Is the action enabled for this instance?
   - Invoke action implementation.
   - Invoke validations and determinations.

In case of a managed provider, the infrastructure executes the provider steps as listed here. Only the implementations of actions, validations, and determinations are application-specific.

In case of an unmanaged provider, the application implementation must perform all the steps. For other operations, the sequence is the same, but not all steps are always relevant, and different application handlers are invoked based on the operation. Static actions, for example, are not related to a specific instance. Consequently, the instance-related steps are not relevant for a static action. Another example is locking, which is not relevant for read access or for functions.

Within the interaction phase, multiple operations can be called that result in changes to the transactional buffer until at some point in time the editing is completed and the data is finally checked and saved to the database.

Save Phase Operation Flow

The `COMMIT ENTITIES` EML statement finalizes the transaction and the changed data is persisted to the database. For this, the ABAP RESTful application programming model runtime invokes all involved business objects within one transaction in a process called the *save sequence*. This sequence consists of the following steps:

- Finalize (`finalize`)
- Check before save (`check_before_save`)
- Draw numbers (`adjust_numbers`)
- Save (`save`)
- Cleanup (`cleanup`)

Each step is implemented by a corresponding method. The runtime of the ABAP RESTful application programming model calls these methods in sequence for each involved business object.

If one method or business object fails—for example, if it isn't in a consistent state to be saved from an application perspective—the runtime rejects the save operation.

During the *finalize* step, the runtime invokes validations, determinations, and application logic. Then during the *check before save* step, everything is checked for consistency and to determine whether it can be saved at all. If after these two phases an error occurs, the transaction moves back to the interaction phase, allowing the consumer to correct the issues and run the save again.

If everything is consistent, then numbers are drawn from number ranges in the *draw numbers* step. This is required for applications that use late numbering—for example, because there are legal requirements for gapless numbering. Finally, in the *save* step the data is saved to the database, usually by registering an update task function module. After the save is complete, the *cleanup* step ensures that transactional buffers are cleaned up and are reset to the proper state. If in these late phases an issue occurs,

the transaction is rolled back. An explicit rollback can be achieved with the `ROLLBACK ENTITIES` EML statement.

Provider Implementation

The overall architecture and the process for the interaction and save phases are independent of the concrete provider implementation. Every provider needs to fulfill the contract defined by the ABAP RESTful application programming model.

For managed providers, the infrastructure already fulfills most of the contract out of the box and the application developer can focus on the pure application logic. The infrastructure takes care of internal buffer handling of transactional changes, the proper program flow, and even the access to the database (see Figure 2.7, showing the runtime architecture with the managed provider implementation).

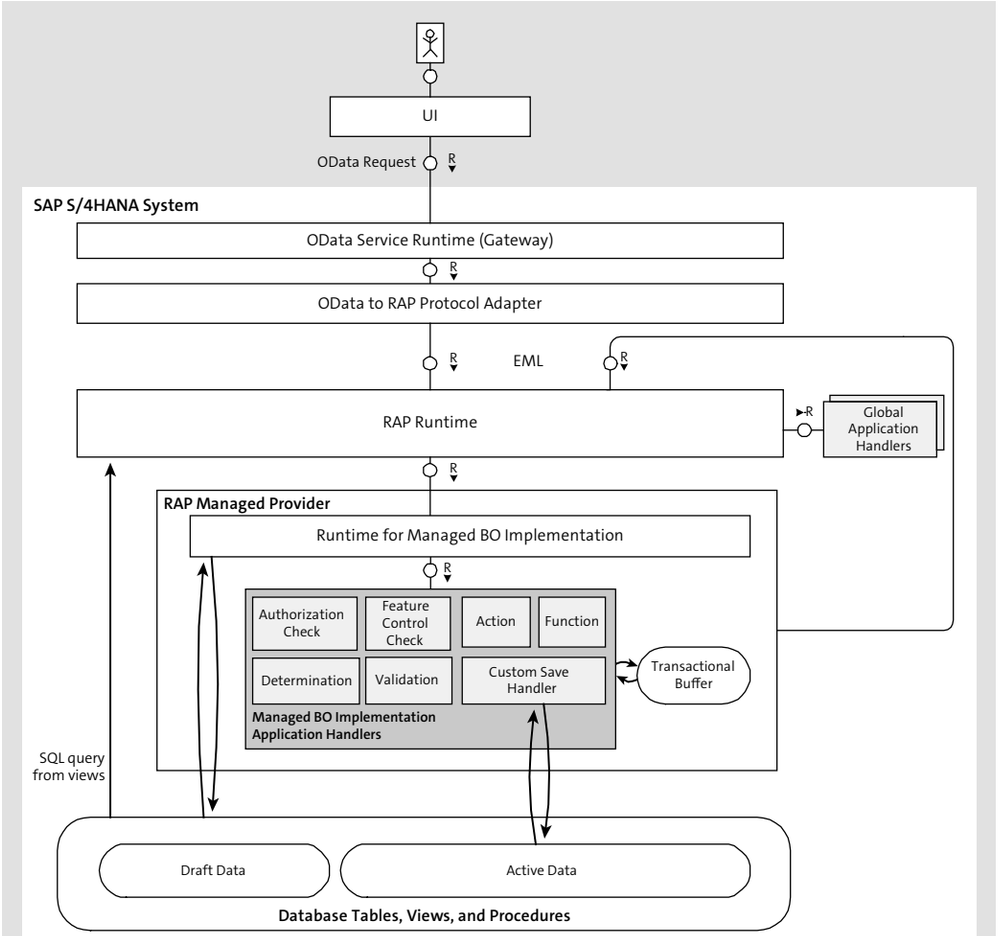


Figure 2.7 Runtime Architecture with Managed Provider

Application developers implement application handlers for different aspects only. Depending on the entity, these can be validations, determinations, entity-specific authorizations, feature controls, actions, and functions. As mentioned, the standard save functionality provided by the infrastructure can be replaced with custom save logic (`UNMANAGED SAVE`; Section 2.2.1). The custom save handler shown in Figure 2.7 represents such a custom save logic. In most cases, the standard logic provided by the infrastructure is sufficient, and a custom save handler is not needed.

For BOPF-managed providers, the situation is similar (see Figure 2.8). The ABAP RESTful application programming model runtime invokes the BOPF runtime through an adapter. The BOPF runtime takes care of the transactional buffer and the overall flow and calls the application-specific business object implementation handlers for determinations, validations, authorization checks, and so on.

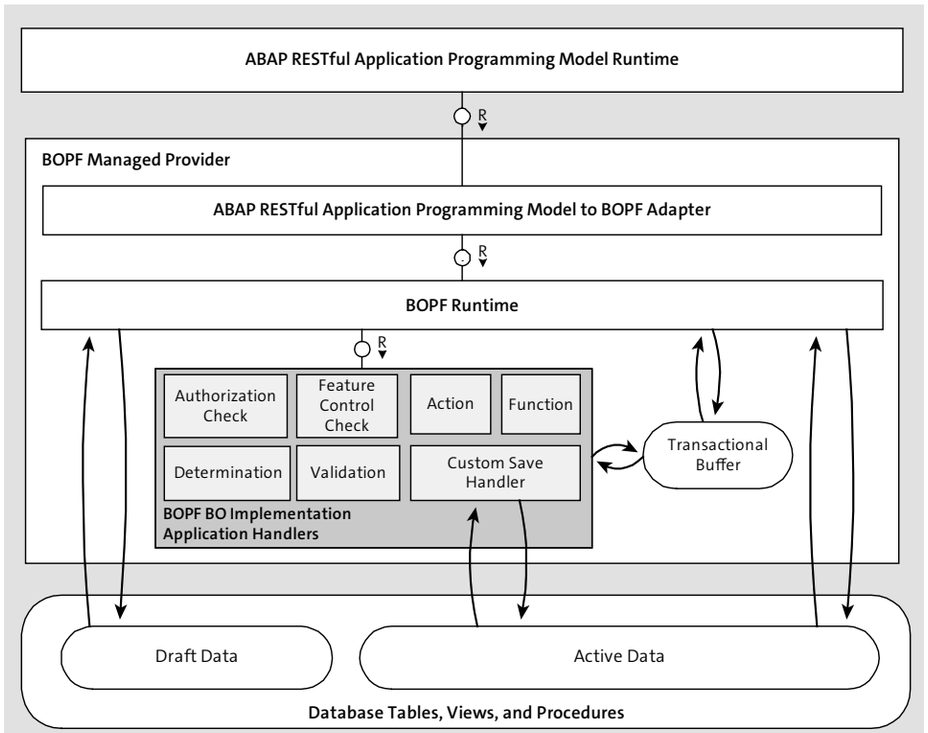


Figure 2.8 BOPF-Managed Provider

In case of an unmanaged provider, developers can reuse the existing application implementation. However, the implementation must fulfill the ABAP RESTful application programming model contract, and the unmanaged provider must implement several aspects that are otherwise provided by the infrastructure. This includes, for example, the create, update, and delete operations, the different transaction phases, the save

sequence, and the transactional buffer. Often the existing application is not structured in a way that fits, for example, with the transaction phases and the save sequence of the ABAP RESTful application programming model, and thus needs to be refactored.

As explained earlier (Section 2.2.1), the draft functionality must be handled either by the ABAP RESTful application programming model infrastructure or by BOPF, even for unmanaged providers. Figure 2.9 illustrates this option for an unmanaged provider with drafts managed by the infrastructure of the ABAP RESTful application programming model.

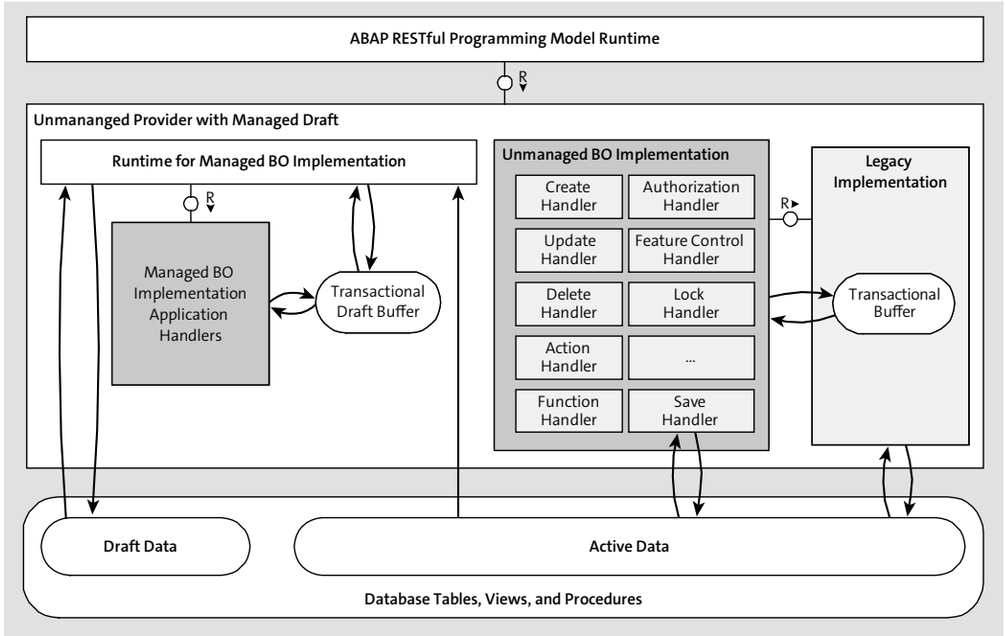


Figure 2.9 Unmanaged Provider with Managed Draft

Consumption through OData

As stated before, the ABAP RESTful application programming model provides full end-to-end support for consuming business objects through an OData service (OData V2 and OData V4). Thus, if an OData service is defined using a service binding (Section 2.2.2), the ABAP platform provides the complete protocol layer and translates any kind of request to either SQL (for query read access) or EML (for transactional access). With this architecture approach, the application implementation does not have to take the protocol and protocol version used into consideration. Thus, supporting additional channels or switching the protocol or protocol version can be achieved with very low efforts. In addition, the application can focus on real application logic instead of dealing with technical protocol topics.

2.3 Summary

To stay compatible with SAP ERP, the data model of SAP S/4HANA uses the same database tables for primary data – while derived data such as aggregates is computed on the fly However, it shields these database tables using an understandable, comprehensive VDM that uses well-known business terminology, documents the relationships between entities, and enriches the entities with additional business semantics. The VDM is composed from a CDS hierarchy. The data model of a business object consists of CDS views too.

Based on this business object definition, application developers can use the ABAP RESTful application programming model to implement the transactional behavior, especially in CRUD operations.

With the managed, BOPF-managed, and unmanaged implementation types, the programming model supports three flavors. Each has a different level of framework support and solves a different use case, from developing a new application to refactoring an existing application.

By using the ABAP RESTful application programming model, application developers can easily allow business users to store a draft version of their business object.

Now you've seen that the transactional programming model relies on the VDM. In the next chapter, you'll learn how SAP Fiori apps consume the data exposed by the VDM.

Contents

Index

C

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

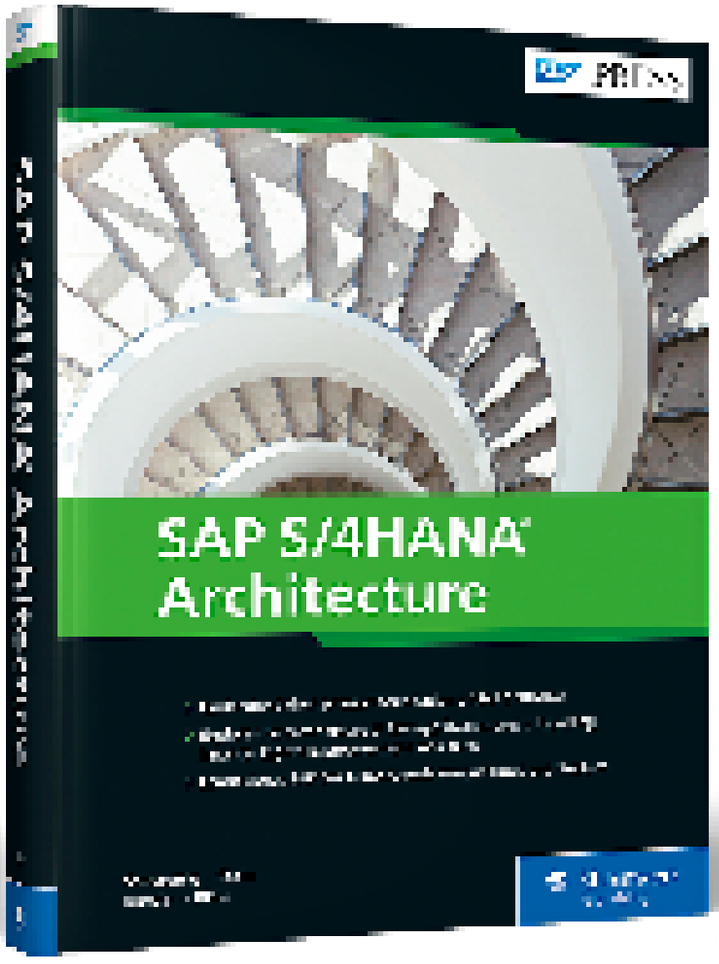
S

T

U

V

W



Thomas Saueressig, Tobias Stein, Jochen Boeder, Wolfram Kleis

SAP S/4HANA Architecture

520 Pages, 2021, $79.95
ISBN 978-1-4932-2023-6

www.sap-press.com/5189

Thomas Saueressig is a member of the Executive Board of SAP SE. He leads the SAP Product Engineering area and has global responsibility for all business software applications. This includes all functional areas from product strategy and management to product development and innovation as well as product delivery and support.

Tobias Stein is a development senior executive and leads the SAP S/4HANA architecture area. Tobias is a software architecture expert and has more than 22 years of experience with SAP's ERP (SAP Business Suite and SAP S/4HANA) application and solution architectures. As a member of SAP's global leadership team, he is heading the central architecture unit in the SAP S/4HANA development organization. He is a TOGAF-certified Enterprise Architect.

Jochen Boeder is vice president of the SAP S/4HANA architecture area. He has more than 15 years of experience in the architecture of SAP business applications. At SAP, he leads a team of senior architects in the SAP S/4HANA engineering organization. He has co-authored several IT books, most prominently, The Architecture of SAP ERP.

Dr. Wolfram Kleis is a software architect in the SAP S/4HANA architecture area. He joined SAP more than 20 years ago and held several positions as developer and architect. Most recently he engineered cloud applications for Internet of Things before joining the central SAP S/4HANA architecture unit. He has a strong passion for describing software architecture and is the author of the SAP HANA chapter in the book The Architecture of SAP ERP.