Teoría

BDL – Behavior Definition Language

ABAP RESTful – Arquitectura Cloud

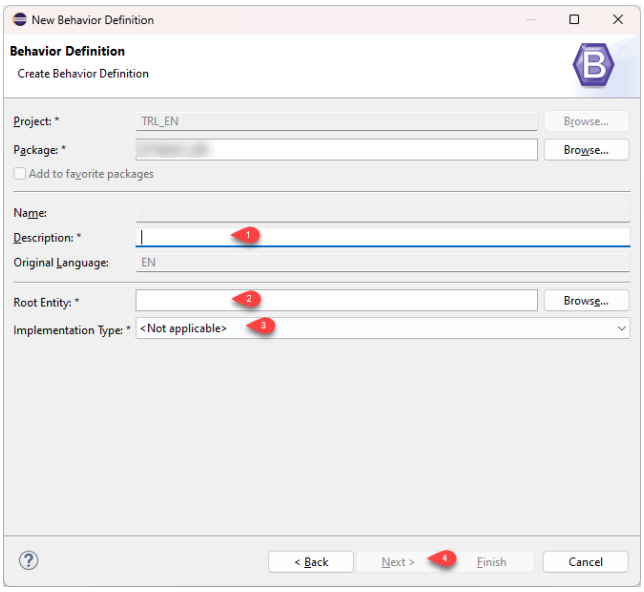
Contenido

1. BDL – Behavior Definition Language

1.1. Managed – Escenario Administrado

Es una metodología utilizada para la gestión de operaciones básicas de creación, actualización y eliminación de datos en una entidad raíz. Este escenario, también conocido como "escenario administrado", permite que el framework de SAP RAP maneje automáticamente las operaciones de persistencia, facilitando el desarrollo y mantenimiento de aplicaciones empresariales. La definición de comportamiento, o Behavior Definition, es fundamental para dar vida a estas operaciones y asegurar que los datos se gestionan de manera eficiente y segura.

Para crear un Behavior Definition, se realiza a través de la carpeta de proyecto con clic derecho en **New** luego en la opción **Other ABAP Repository Object** ubicar la carpeta **Core Data Services** y luego seleccionar **Behavior Definition,** para luego seleccionar la entidad raíz requerida para la definición del comportamiento**.** También es posible seleccionar directamente la entidad raíz y hacer clic derecho y seleccionar la opción **New Behavior Definition**. Este objeto toma el mismo nombre que la entidad raíz y tiene una cardinalidad de uno a uno.



Al seleccionar la entidad raíz para la definición del comportamiento se debe colocar una descripción y un tipo de implementación los cuales pueden ser:

- **Managed**: Este tipo de implementación viene seleccionada por defecto y permite que el framework gestione automáticamente las operaciones de creación, actualización y eliminación. Por lo general este es el tipo de comportamiento que se utiliza.
- **Unmanaged**: El desarrollador debe implementar manualmente estas operaciones en la capa de persistencia.

Sintaxis:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name//alias <alias_name>
persistent table database_name_from_root
lock master
authorization master ( instance )
//etag master <field_name>
{
 create;
 update;
 delete;
 field ( readonly ) ComponentUuID;
}
```

1.2. Mapeo persistencia

El mapeo de persistencia asegura que las propiedades de la entidad de consumo se correspondan correctamente con las columnas de la tabla de persistencia, permitiendo al framework ejecutar estas operaciones de manera correcta y eficiente.

Proceso de Mapeo:

Se define el mapeo utilizando la anotación **mapping for**, indicando el alias del componente de la entidad raíz asignado a través de una igualación con el nombre del campo de la tabla de persistencia.

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name//alias <alias_name>
persistent table database_name_from_root
lock master
authorization master ( instance )
//etag master <field_name>
{
 create;
 update;
 delete;
 field ( readonly ) ComponentUuID;
 mapping for database_name_from_root
  {
    Component_root_entity    = component_database;
  }
}
```

Utilizando herramientas de desarrollo como Eclipse, se puede copiar y ajustar el listado de columnas para asegurar una alineación correcta entre la entidad de consumo y la tabla de persistencia.

1.3. Lock – Bloqueos

El mecanismo de bloqueo en SAP RAP es vital para asegurar la integridad y consistencia de los datos en aplicaciones empresariales. Mediante la correcta configuración de los flags de bloqueo en la definición de comportamiento, se evita que múltiples usuarios puedan editar simultáneamente la misma información, previniendo así la pérdida de datos y las inconsistencias. La implementación adecuada de estos mecanismos, ya sea en escenarios administrados o no administrados, garantiza una gestión eficaz y segura de las operaciones de edición de registros.

Implementación del Mecanismo de Bloqueo:

Al crear cualquier definición de comportamiento el framework agrega por defecto un tipo de bloqueo para evitar que múltiples usuarios puedan editar simultáneamente la misma información. Los tipo de bloqueos son los siguientes:

- **lock master:** Asegura que una entidad maestra no pueda ser editada simultáneamente por múltiples usuarios, evitando inconsistencias. Con este tipo de bloqueo el framework gestiona automáticamente el bloqueo necesario para evitar que múltiples usuarios editen la misma información.
- **lock master unmanaged:** El desarrollador debe implementar manualmente los mecanismos de bloqueo utilizando objetos de bloqueo del diccionario de datos para controlar la edición y evitar conflictos.
- **lock dependent by AliasRootEntity:** Se utiliza en escenarios jerárquicos para asegurar que las ediciones en niveles inferiores dependen del estado de la entidad maestra.
- Si se especifica el modo **strict 2**, es obligatorio especificar algún tipo de bloqueo.

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
//lock master unmanaged
//lock dependent by AliasRootEntity
authorization master ( instance )
//etag master <field_name>
{
 create;
 update;
 delete;
```

```
 field ( readonly ) ComponentUuID;
 mapping for database_name_from_root
  {
    Component_root_entity    = component_database;
  }
}
```

1.4. Control de autorizaciones

En las aplicaciones RAP, el control de autorizaciones es una herramienta poderosa para manejar accesos y permisos de manera eficiente y segura, asegurando que cada operación y acceso a los datos se gestione de acuerdo con las políticas y necesidades específicas de la organización. Esta herramienta se maneja a través de dos mecanismos principales:

- **Nivel global:** Se aplica para todo el contenido que se expone en la aplicación, permitiendo situaciones en las que ciertos grupos de usuarios no puedan realizar operaciones sobre todo el conjunto de datos. Este enfoque es útil cuando se desea bloquear operaciones como delete o create de manera global, especialmente cuando los usuarios no tienen los permisos necesarios en los objetos de autorización.
- **Nivel de instancia:** Por otro lado, permite gestionar autorizaciones específicas para ciertos registros. Esto significa que, dependiendo de la información contenida en los registros, como el código de la compañía o el centro de ventas, un usuario podría no tener acceso a ciertos datos. Este control depende directamente de los datos contenidos en cada registro. Por ejemplo, si un registro está asignado a un centro específico al que un usuario no tiene acceso, las operaciones relacionadas con ese registro estarán deshabilitadas.

Estas configuraciones generan métodos en la clase del behavior pool que deben ser implementados para cada situación, permitiendo un control preciso y flexible sobre el acceso a los datos.

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
```

```
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
authorization master ( global, instance ) // global
//authorization master ( instance )          // instance
{
 create;
 update;
 delete;
 field ( readonly ) ComponentUuID;
 mapping for database_name_from_root
  {
   Component_root_entity    = component_database;
  }
}
```

1.5. ETag – Concurrencia

La gestión de concurrencia es crucial para asegurar la calidad y consistencia de los datos cuando múltiples peticiones acceden simultáneamente a los servicios OData expuestos por las aplicaciones. Los mecanismos de concurrencia se definen en la Behavior Definition y utilizan ETags para manejar eficientemente las peticiones concurrentes y prevenir inconsistencias en los datos.

Implementación de ETag:

Para definir estos mecanismos, se deben incorporar las correspondientes configuraciones de ETags junto con campos tipo **timestamp** como semillas o campos de ayuda para los algoritmos de concurrencia dentro del Behavior Definition. Estos campos aseguran que cada petición concurrente se maneje de manera única y consistente. A continuación se describen los tipos de Etag permitidos:

- **Total ETag:** Gestiona la concurrencia a nivel del volumen total de datos expuestos por el endpoint, proporcionando una capa adicional de control sobre las peticiones concurrentes. Se utiliza el campo **LastChangeAt**, para gestionar la concurrencia a nivel del volumen total de datos expuestos.

La implementación del Total ETag requiere la activación del modo Draft, que permite gestionar registros en estado de borrador. Este modo asegura que las modificaciones se apliquen de manera consistente antes de ser finalizadas y expuestas.

- **ETag Master:** Se utiliza para gestionar la concurrencia a nivel de la instancia maestra o entidad raíz. Este tipo de ETag asegura que las peticiones concurrentes a la entidad raíz sean manejadas adecuadamente. Se utiliza el campo **LocalLastChangeAt**, para gestionar la concurrencia a nivel de la instancia maestra.

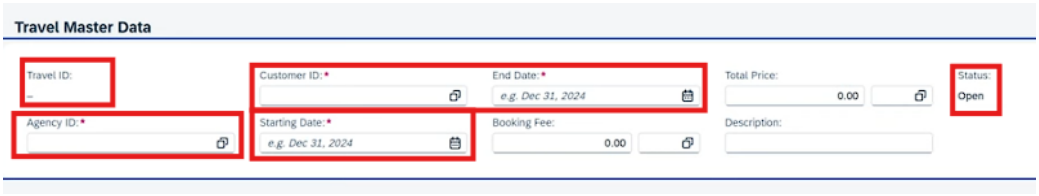
Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( readonly ) ComponentUuID;
 mapping for database_name_from_root
  {
   Component_root_entity   = component_database;
  }
}
```

1.6. Control estático – Solo lectura/Obligatorio

El control estático en aplicaciones RAP se refiere a la configuración de propiedades específicas en los campos de las entidades. Estas propiedades pueden incluir atributos como solo lectura y

obligatoriedad, así como la generación automática de identificadores únicos.



Esto se realiza a través de la definición del comportamiento en Eclipse, lo que facilita la gestión y control de los datos en la aplicación. Dichas características estáticas son las siguientes:

- **Campos de Solo Lectura (Read Only):** Establece un campo como solo lectura, utilizando la característica **field (readonly).** en la definición del comportamiento. Algunos ejemplo de uso son:
  - Para los campos de auditoría, los cuales también se configuran como solo lectura para garantizar la integridad de los datos.
  - Para los campos clave UUID ya que se configuran como campos de solo lectura para evitar que los usuarios los modifiquen.
- **Generación Automática de UUIDs:** Esta configuración permite a los campos clave UUID generar automáticamente un identificador único generado el sistema, eliminando la necesidad de que los usuarios asignen manualmente estos valores. Por medio de la característica **field (numbering manage, readonly).**
- **Campos Obligatorios (Mandatory):** La definición de estos campos garantiza que siempre se proporcionen los valores necesarios para la correcta operación de la aplicación. **field (mandatory).**

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
```

```
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;

 mapping for database_name_from_root
  {
    Component_root_entity    = component_database;
  }
}
```

1.7. Acciones

Las acciones en RAP son mecanismos que permiten la interacción del usuario con la aplicación a través de la interfaz de usuario, así como la ejecución de operaciones internas en la lógica del código fuente. Estas acciones se definen en el documento de definición de comportamiento (Behavior Definition) y pueden impactar directamente en la capa de persistencia de datos.

Tipos de acciones:

- **Acciones Básicas:** Corresponden a las representaciones de las operaciones Odata que van a tener un impacto en la capa de persistencia del estándar RAP. Dichas acciones básicas son eliminar, crear y actualizar registros se representan en la UI a través de los botones Create, Delete y Edit. Estos se definen automáticamente en el Behavior Definition con las palabras reservadas **Create, Update** y **Delete.**
- **Acciones Adicionales:** Además de las acciones estándar, es posible definir otras acciones para los botones personalizados definidos junto con sus características en las Metadata

extensions asociados con la entidad de consumo que está a su vez están vinculadas con la entidad raíz utilizada en el Behavior Definition. Esto se realiza utilizando en la acción el mismo nombre del botón definido en el Metadata Extensión en la característica dataAction.

Sintaxis:

```
action actionName result [1] $self;
```

Donde **result [1] $self,** especifica que el resultado de la acción impactará a la propia instancia del objeto en la que se ejecuta la acción ($self). La cardinalidad [1] indica que esta acción devuelve un único resultado.

Verificación de autorizaciones:

También es posible añadir a la acción del botón una verificación de autorizaciones. Esto se realiza al definir una acción, agregar entre paréntesis **(authorization : option)** para validar si un usuario tiene permiso para ejecutar la acción, mejorando la seguridad y control de acceso. Donde las opciones de la autorización permitidas son las siguientes:

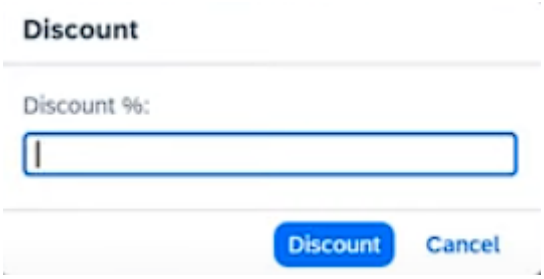
- **authorization : global:** Esta validación se aplica a nivel global, afectando a todos los registros de la entidad. Si un usuario no tiene la autorización global requerida, no podrá ejecutar la acción en ningún registro de la entidad, sin importar las características individuales de cada registro.
- **authorization : instance:** Esta validación se aplica a nivel de instancia, lo que significa que la autorización se verifica individualmente para cada registro o instancia de la entidad. Un usuario puede tener permiso para ejecutar la acción en algunos registros pero no en otros, dependiendo de la información específica de cada registro.
- **authorization : none:** Esta configuración indica que no se requiere ninguna validación de autorización para ejecutar la acción. Cualquier usuario podrá realizar la

acción sin necesidad de cumplir con requisitos específicos de permisos.

- **authorization : update:** Esta validación comprueba que el usuario tiene los permisos necesarios para actualizar el registro antes de permitir la ejecución de la acción. Es útil para acciones que implican cambios en los datos y se requiere asegurar que el usuario tenga los privilegios adecuados para hacer dichas modificaciones.

```
authorization : global (keyword)
authorization : instance (keyword)
authorization : none (keyword)
authorization : update (keyword)
```

- **Parámetros:** Algunas acciones pueden requerir parámetros o valores del usuario, facilitando una interacción más dinámica y segura con la aplicación.



Para estos casos es necesario agregar la instrucción **parameter** en la definición de la acción y especificar una entidad abstracta con los campos y sus respectivos elementos de datos que se utilizarán.

- **Acciones Internas:** Las acciones internas no son visibles para los usuarios y se ejecutan dentro de la lógica del código fuente en la clase Behavior Pool. Estas acciones ayudan a modularizar la lógica y realizar operaciones que no requieren interacción directa del usuario, por medio de la instrucción **internal action,** seguida del nombre de la acción.

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
```

```
persistent table database_name_from_root
lock master
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;
action actionName result [1] $self;
action (authorization : update) actionName2 result [1] $self;
//  action ( authorization : update ) actionName2
//    parameter absytact_entity_name result [1] $self;
 mapping for database_name_from_root
  {
    Component_root_entity    = component_database;
  }
}
```

1.8. Control dinámico – Características

El control dinámico en RAP se refiere a la capacidad de modificar las características de los elementos de la UI y los campos de datos en función del estado de los registros y otros factores contextuales. Esto permite que ciertos botones o campos solo estén disponibles o editables bajo condiciones específicas, mejorando la usabilidad y la precisión de la aplicación.

Características Dinámicas para Botones y campos:

Estas características dinámicas se definen en los controles estáticos y las acciones agregando la instrucción **features : option**. Donde las opciones permitidas son **global**, que se aplica a todos los valores

disponibles de la interfaz, o **instance**, que se aplica a un registro específico (llamados instancias en la terminología de RAP). Esto permite que las acciones disponibles en la UI se adapten dinámicamente al estado actual de los datos.

Aunque las características dinámicas no pueden ser agregadas en las acciones internas porque estas acciones no tienen representación en la interfaz de usuario (UI). Las características dinámicas, están diseñadas para afectar elementos visibles y accesibles en la UI, permitiendo que su estado y disponibilidad cambien en función de los datos o el contexto. Las acciones internas, por otro lado, se ejecutan en la lógica del código fuente y no requieren ni tienen una representación visual. Debido a esto, aplicar características dinámicas a acciones internas carece de sentido práctico, ya que no habría elementos visuales para aplicar dichos cambios

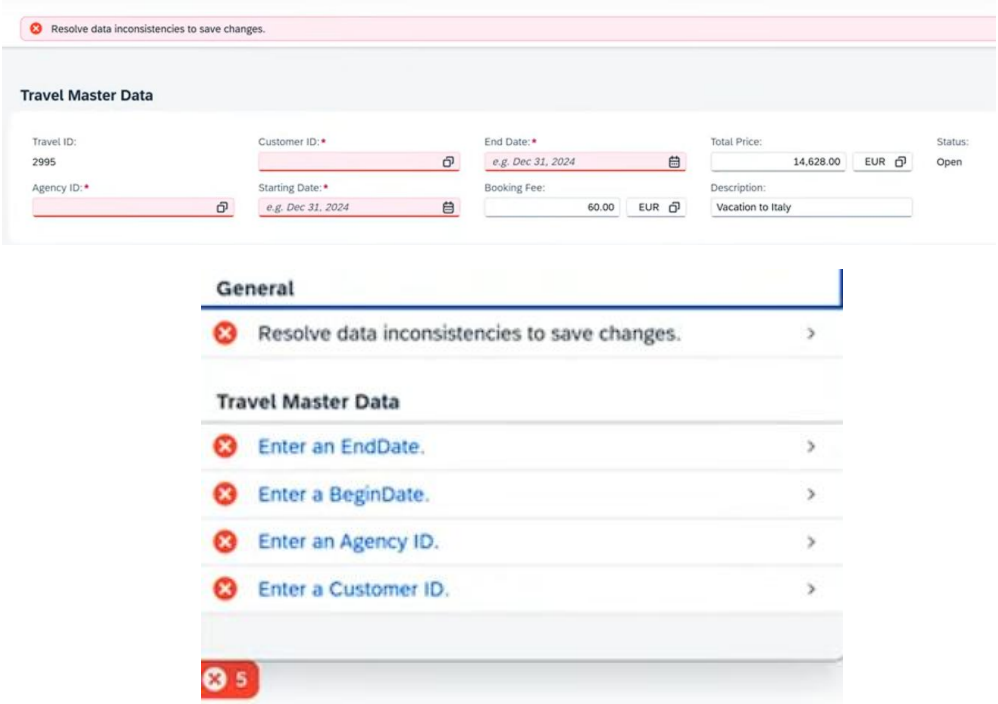
Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;
 field ( features : instance ) Component, Component,...;
action ( features : instance ) actionName result [1] $self;
action ( features : instance, authorization : update) actionName2
result [1] $self;
//  action ( authorization : update ) actionName2
```

```
//    parameter absytact_entity_name result [1] $self;
 mapping for database_name_from_root
   {
     Component_root_entity    = component_database;
   }
}
```

1.9. Validaciones

Son mecanismos implementados en aplicaciones RAP para verificar la integridad y validez de los datos antes de que se guarden o actualicen en la base de datos. Estas validaciones permiten establecer bloqueos en la capa de persistencia para evitar alteraciones indebidas de los datos en las tablas o en los objetos de persistencia. Además, proporcionan mensajes informativos a los usuarios sobre errores en los valores introducidos, facilitando la corrección de los mismos.



Estas validaciones pueden ser definidas en la capa de backend utilizando el Behavior Definition y requieren una implementación en la clase del Behavior Pool.

Sintaxis:

validation validateComponent on save { create; field AliasComponentName, …; }

Donde la operación **create** puede ser sustituida con algunas de las acciones básicas como: **Create, Update** y **Delete.**

create; (keyword)
delete; (keyword)
field (keyword)
update; (keyword)

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;
 field ( features : instance ) Component, Component,...;
action ( features : instance ) actionName result [1] $self;
action ( features : instance, authorization : update) actionName2
result [1] $self;
//  action ( authorization : update ) actionName2
//    parameter absytact_entity_name result [1] $self;
validation validateComponent on save { create; field AliasComponentName, …; }

 mapping for database_name_from_root
  {
   Component_root_entity   = component_database;
  }
```

```
}
```

Como dato adicional, aunque el orden de las definiciones en la Behavior Definition no es obligatorio, es recomendable agrupar las validaciones de manera ordenada, siguiendo una estructura lógica que primero incluya las operaciones, luego los campos, las acciones y finalmente las validaciones.

1.10. Determinaciones

Las determinaciones en RAP permiten definir cómo y cuándo se deben establecer los valores de ciertos campos en función de reglas de negocio predefinidas. Estas reglas se ejecutan en la capa ABAP y aseguran que los valores de los campos se generen automáticamente de acuerdo con las necesidades específicas del negocio, evitando errores y mejorando la consistencia de los datos.

Los campos que se determinan automáticamente suelen ser de tipo **readonly**, lo que significa que los usuarios no pueden modificarlos manualmente. Esto garantiza la integridad de los datos generados automáticamente.

Implementación de Determinaciones:

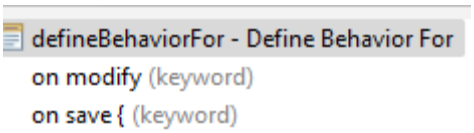
Las determinaciones se definen en el archivo de Behavior Definition utilizando la palabra reservada **determination**. Se especifica el nombre de la determinación y el momento en el que debe ejecutarse, como **on save** on **modify.**

Sintaxis:

```
determination determination_name on save { create; }
determination determination_name2 on modify { create; }
```

Las determinaciones pueden configurarse para ejecutarse en diferentes momentos los cuales son los siguientes:

- **On save:** Durante la creación de un nuevo registro.
- **On modify:** Cuando se modifica un registro existente.



La lógica específica de las determinaciones se implementa en la clase Behavior Pool. Aquí, los métodos correspondientes a cada determinación realizan los cálculos y asignaciones necesarias.

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;
 field ( features : instance ) Component, Component,...;
action ( features : instance ) actionName result [1] $self;
action ( features : instance, authorization : update) actionName2
result [1] $self;
//  action ( authorization : update ) actionName2
//    parameter absytact_entity_name result [1] $self;
validation validateComponent on save { create; field
AliasComponentName, …; }

determination determination_name on save { create; }
 determination determination_name2 on modify { create; }
 mapping for database_name_from_root
```

```
  {
    Component_root_entity    = component_database;
  }
}
```

1.11. Efectos secundarios

Los efectos secundarios en RAP se refieren a la actualización automática y constante de la interfaz de usuario cuando ciertos campos o elementos son modificados. Esta funcionalidad es fundamental para garantizar que los cambios realizados en un campo se reflejan inmediatamente en otros campos relacionados, ofreciendo una experiencia de usuario coherente y responsiva.

La actualización en tiempo real de los elementos en la UI garantiza que los usuarios vean los cambios inmediatamente, lo que mejora la usabilidad y la eficiencia de la aplicación.

Implementación de Efectos Secundarios:

Los efectos secundarios se definen en la Behavior Definition utilizando la palabra clave Side effects. Aquí, se especifican las relaciones entre los campos y cómo los cambios en uno pueden afectar a otros.

Implementación de Efectos Secundarios:

Los efectos secundarios se definen en la Behavior Definition utilizando la palabra clave **Side effects.** Aquí, se especifican las relaciones entre los campos y cómo los cambios en uno pueden afectar a otros.

Sintaxis:

```
side effects
{
  field Component1 affects field Component2;
}
```

Interacción con Validaciones:

Las validaciones en la UI también pueden beneficiarse de los efectos secundarios. Los mensajes de error se actualizan en tiempo real a medida que el usuario corrige los errores en los campos, proporcionando retroalimentación inmediata.

Para gestionar estas actualizaciones, se puede utilizar las sentencias **determine action**, que actúan como envoltorios para las validaciones previamente estudiadas, asegurando que cualquier cambio en los campos se refleje inmediatamente en los mensajes de error y otros elementos dependientes.

Sintaxis:

```
determine action determineActionName{ validation validationName; }
 side effects
 {
  field BookingFee affects field TotalPrice;
  determine action determineActionNameexecuted on field Component affects messages;
 }
```

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
lock master
// total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;
```

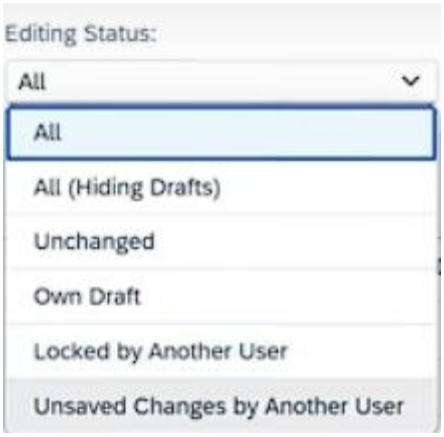
```
 field ( features : instance ) Component, Component,...;
action ( features : instance ) actionName result [1] $self;
action ( features : instance, authorization : update) actionName2
result [1] $self;
//  action ( authorization : update ) actionName2
//    parameter absytact_entity_name result [1] $self;
validation validateComponent on save { create; field
AliasComponentName, …; }

determination determination_name on save { create; }
determination determination_name2 on modify { create; }
determine action determineActionName{ validation
validationName; }
 side effects
 {
  field BookingFee affects field TotalPrice;
  determine action determineActionNameexecuted on field
Component affects messages;
 }
 mapping for database_name_from_root
  {
    Component_root_entity   = component_database;
  }
}
```

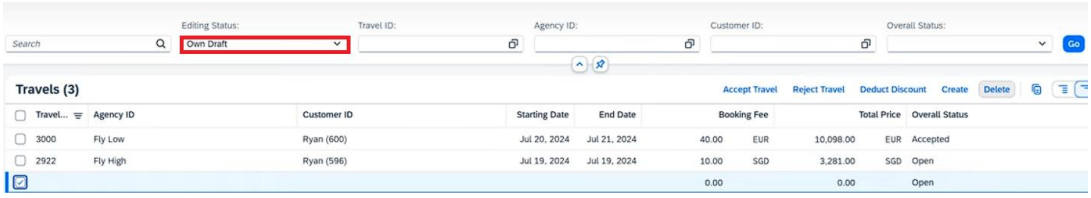
1.12. Draft

El modo Draft permite a los usuarios trabajar en formularios o expedientes sin necesidad de completarlos de una sola vez. Los cambios realizados se guardan automáticamente en un estado temporal, conocido como borrador, permitiendo a los usuarios retomar su trabajo en cualquier momento sin perder el progreso. Esta funcionalidad es esencial para manejar tareas largas o complejas que no pueden completarse en una única sesión.

El modo borrador o Draft habilita en la interfaz de usuario el campo **Editing Status** que permite observar los registros que están en modo borrador, tanto aquellos creados por nosotros como los que corresponden a la aplicación. Además, también muestra los registros que están bloqueados por otros usuarios.



La funcionalidad de Draft permite a los usuarios gestionar eficientemente sus registros en modo borrador. Esta capacidad ofrece la opción de filtrar y visualizar los registros que están en estado de borrador, permitiendo decisiones clave como eliminar el registro, continuar editando en el punto donde se dejó, descartar los cambios realizados o persistir los cambios en la base de datos. La persistencia de los cambios asegura que estos se guarden permanentemente y ya no se consideren borradores.



Implementación del Modo Draft:

Para habilitar el modo Draft, se debe agregar la instrucción **with Draft** en la cabecera del Behavior Definition de la aplicación. Esto requiere la definición de una tabla de borrador (Draft Table) que almacena los datos temporales. Para luego especificarla luego de la tabla de persistencia con la instrucción **draft table.** Al activar el modo draft es necesario agregar la instrucción **total etag.**

Sintaxis:

```
managed implementation in class behavior_definition_name unique;
with draft;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
```

```
draft table draftdatabase_name
lock master
total etag LastChangedAt
etag master LocalLastChangedAt
```

Al implementar el modo draft es necesario definir alguna de las acciones específicas del modo Draft, como **Resume, edit, activate** y **discard.** Estas acciones gestionan las transiciones entre los estados de borrador y persistencia, así como las validaciones necesarias.

```
draft action Resume;
draft action Edit;
draft action Activate optimized;
draft action Discard;
```

Integración con Validaciones y Determinaciones: prepare

Es posible agregar determinaciones en el modo draft para incluir validaciones previamente creadas que se ejecutan continuamente para mantener la coherencia y precisión de los datos. Esto se realiza por medio de la instrucción **draft determine action Prepare.** Las validaciones y efectos secundarios se gestionan para asegurar que cualquier cambio en los datos se refleje de inmediato en la UI, proporcionando una experiencia de usuario fluida y sin interrupciones.

Sintaxis:

```
draft determine action Prepare
{
  validation validationName;
}
```

Ejemplo:

```
managed implementation in class behavior_definition_name unique;
with draft;
strict ( 2 );
define behavior for root_entity_name as AliasRootEntity
persistent table database_name_from_root
```

```
draft table draftdatabase_name
lock master
total etag LastChangedAt
etag master LocalLastChangedAt
authorization master ( instance )
{
 create;
 update;
 delete;
 field ( numbering : managed, readonly ) ComponentUuID;
 field ( mandatory ) Component, Component,...;
 field ( readonly ) Component, Component,...;
 field ( features : instance ) Component, Component,...;
action ( features : instance ) actionName result [1] $self;
action ( features : instance, authorization : update) actionName2
result [1] $self;
//  action ( authorization : update ) actionName2
//    parameter absytact_entity_name result [1] $self;
validation validateComponent on save { create; field
AliasComponentName, …; }

determination determination_name on save { create; }
determination determination_name2 on modify { create; }
determine action determineActionName{ validation validationName;
}
 side effects
 {
  field BookingFee affects field TotalPrice;
  determine action determineActionNameexecuted on field
Component affects messages;
 }
 draft action Resume;
 draft action Edit;
 draft action Activate optimized;
 draft action Discard;
 draft determine action Prepare
 {
  validation validateComponent;
```

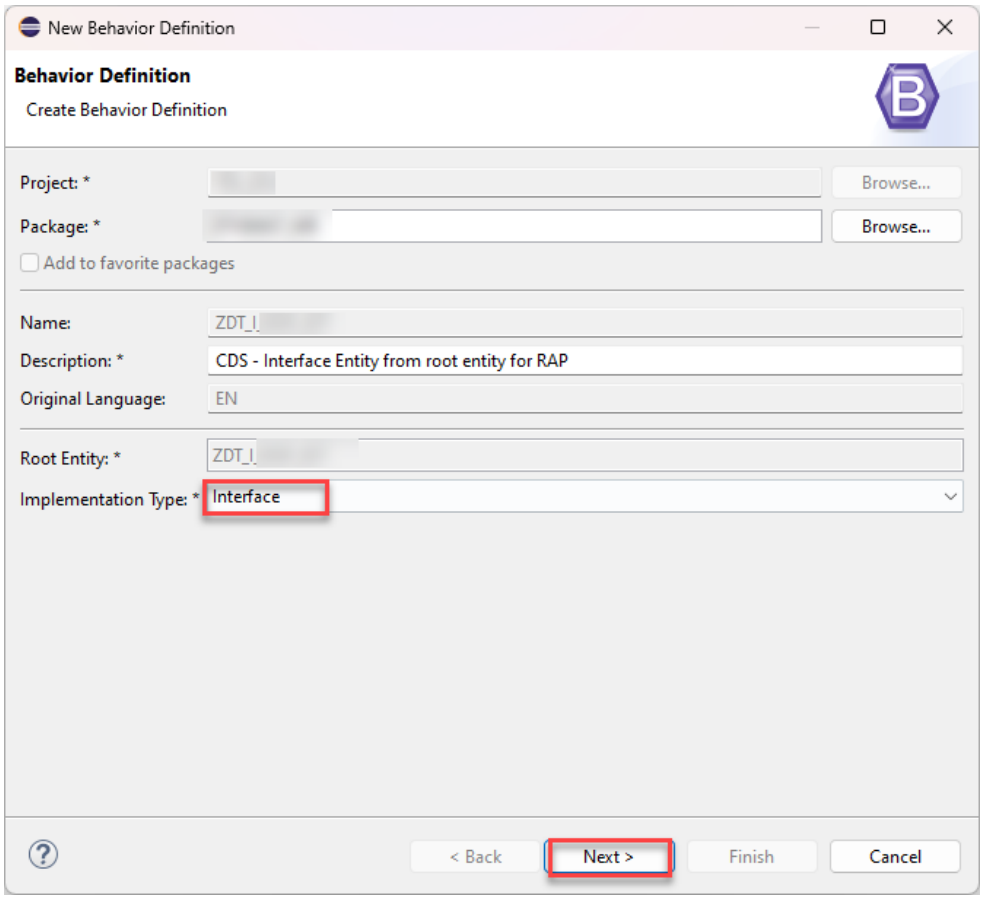
```
 }
 mapping for database_name_from_root
  {
    Component_root_entity    = component_database;
  }
}
```

1.13. Comportamiento de Interfaz – Definición

El comportamiento de interfaz en RAP permite definir y controlar qué operaciones y datos de una aplicación pueden ser accedidos y utilizados desde la capa del consumidor o por otras aplicaciones. A través de una definición estructurada y clara, se asegura que solo los elementos deseados sean visibles y accesibles, protegiendo así la integridad y seguridad de la aplicación.

Creación del Behavior Definition de la Entidad Interfaz:

Para exponer las operaciones y datos a otras aplicaciones o capas del sistema, se crea un Behavior Definition a partir de una entidad de interfaz con una proyección a la entidad raíz y que tenga un tipo de contrato **transactional_interface**. Esta interfaz actúa como un contrato que especifica qué operaciones pueden ser utilizadas externamente. Para hacer esto se puede seleccionar directamente la entidad de interfaz y hacer clic derecho y seleccionar la opción **New Behavior Definition**. Este objeto toma el mismo nombre que la entidad interfaz y establecer el tipo de implementación como **Interface.**



Dentro de la definición, se puede controlar qué operaciones están permitidas. Por ejemplo, se puede permitir o restringir el acceso a operaciones como **create, update, delete, y el uso del modo Draft.**

Sintaxis:

```
interface;
use draft;
define behavior for interface_entity_name alias root_entity_name
{
 use create;
 use update;
 use delete;
 use action actionName;
 use action actionName2;
 use action determineActionValidation;

 use action Resume;
 use action Edit;
 use action Activate;
```

```
 use action Discard;
 use action Prepare;
}
```

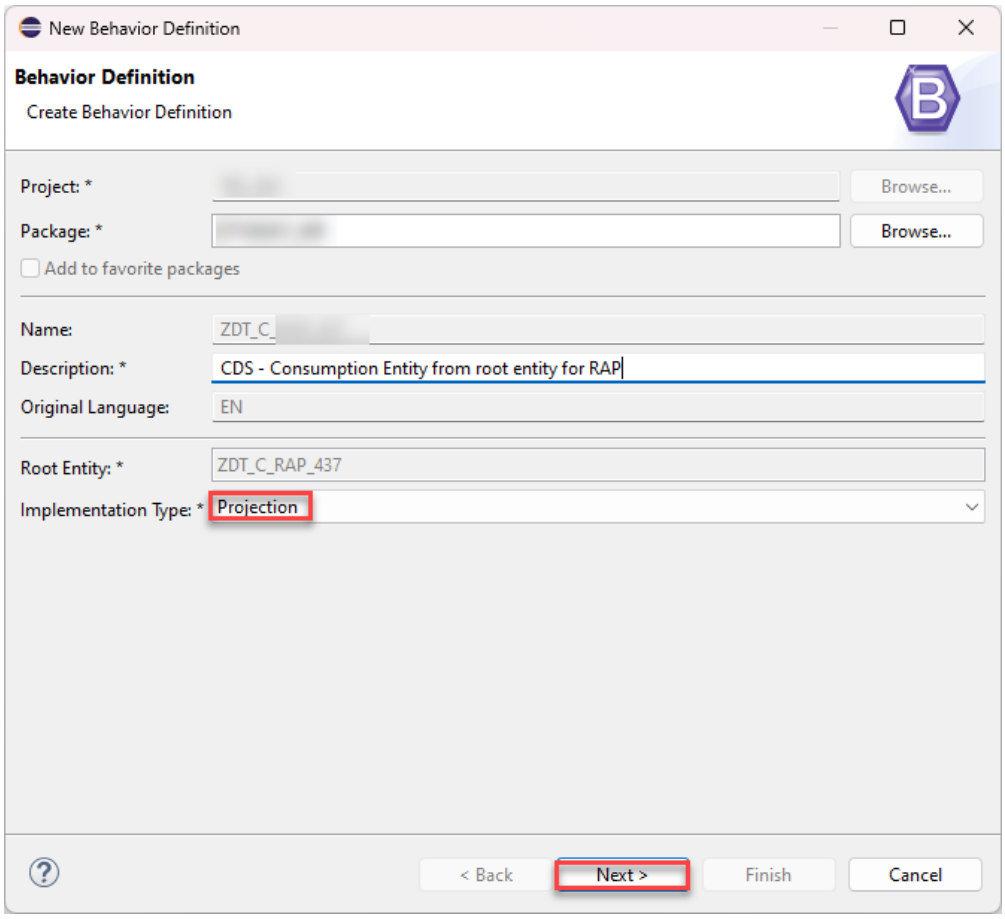
Es posible crear múltiples interfaces o proyecciones para diferentes usuarios o aplicaciones. Por ejemplo, una interfaz puede permitir operaciones de creación y actualización pero restringir la eliminación, mientras que otra interfaz puede estar diseñada para gestores que solo aprueben o rechacen registros.

1.14. Comportamiento de Proyección – Definición

La proyección del comportamiento en RAP se refiere a la definición y exposición de manera y controlada las operaciones y datos que una aplicación puede proporcionar de la entidad de consumo. Esto se logra a través de la creación de un Behavior Definition el cual define claramente qué elementos base serán accesibles externamente.

Creación del Behavior Definition de la Entidad de Consumo:

Para exponer las operaciones y datos a otras aplicaciones o capas del sistema, se crea un Behavior Definition a partir de una entidad de consumo con una proyección a la entidad raíz y que tenga un tipo de contrato **transactional_query**. Esta interfaz actúa como un contrato que especifica qué operaciones pueden ser utilizadas externamente. Para hacer esto se puede seleccionar directamente la entidad de consumo y hacer clic derecho y seleccionar la opción **New Behavior Definition**. Este objeto toma el mismo nombre que la entidad de consumo, para luego establecer el tipo de implementación como **Projection.**



De igual forma este tipo de comportamiento, puede controlar qué operaciones están permitidas. Por ejemplo, se puede permitir o restringir el acceso a operaciones como **create, update, delete, y el uso del modo Draft.**

Aunque no siempre es obligatorio, la creación de proyecciones es esencial para comprender y gestionar cómo las aplicaciones estándar y custom pueden interactuar con los objetos de negocio en RAP.