LOGALI

Teoría

EML – Validaciones y Mensajes

ABAP RESTful – Arquitectura Cloud

Contenido

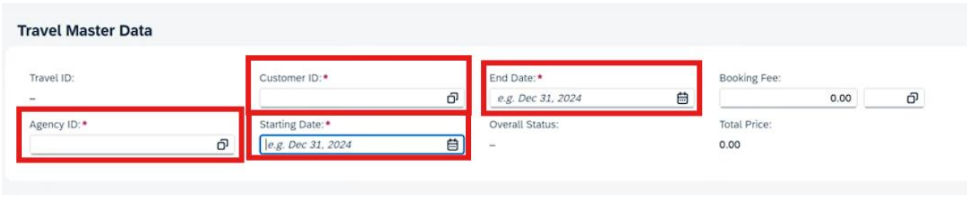
1. EML – Validaciones y Mensajes

Las validaciones de datos son una parte crucial del proceso de aseguramiento de la calidad de la información en una aplicación. Estas validaciones pueden implementarse tanto en el frontend (capa del cliente) como en el backend, garantizando que los datos ingresados cumplan con ciertos criterios antes de ser procesados o almacenados. La correcta implementación de validaciones en ambas capas es esencial para mantener la integridad y confiabilidad de los datos.

1.1. Campos obligatorios

Los campos obligatorios juegan un papel crucial en el proceso de las validaciones de los datos, y deben ser complementados con lógica de programación que asegure que los datos ingresados sean válidos y de la calidad esperada. Esta doble capa de validación asegura que los datos procesados y almacenados cumplan con los criterios establecidos, proporcionando una base sólida para la confiabilidad de la aplicación.

Los campos obligatorios se pueden identificar con un asterisco o estrella roja en la interfaz de usuario, indicando que deben ser llenados para continuar con el proceso y proporcionan una retroalimentación inmediata al usuario sobre la validez de los datos ingresados, mejorando la experiencia del usuario.



Definición de campos obligatorios

Los campos obligatorios se definen en el Behavior Definition con la propiedad Field Mandatory. Esto asegura que se realicen las validaciones necesarias para estos campos. Recordando que se especifican los campos requeridos de la siguiente manera:

field (mandatory) component, component**;**

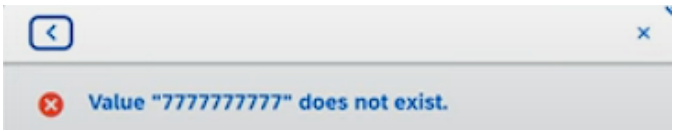
Aunque los campos obligatorios se marcan visualmente, es necesario implementar lógica adicional para validar que estos campos contienen datos antes de crear o actualizar registros. Las validaciones aseguran que los datos cumplan con los estándares de calidad requeridos, evitando errores e inconsistencias. Además, se pueden utilizar Metadata Extensions para vincular elementos de datos con ayudas de búsqueda y otras reglas de validación.

1.2. UI – Use For Validation

Las validaciones en la interfaz de usuario juegan un papel crucial en la calidad y la integridad de los datos ingresados en una aplicación. Utilizando ayudas de búsqueda definidas en los Metadata Extensions y validaciones implícitas por tipo de dato, se puede asegurar que los usuarios ingresen datos válidos y coherentes. Estas validaciones proporcionan una experiencia de usuario mejorada y aseguran que los datos procesados cumplan con los estándares de calidad establecidos.

Ayuda de Búsqueda y Validaciones:

Las ayudas de búsqueda generan automáticamente mensajes de error en la interfaz de usuario de las aplicaciones de Fiori Elements mediante JavaScript cuando el valor ingresado no existe. Esto mejora la calidad de los datos y facilita la asignación correcta de los datos en los campos con estas propiedades.



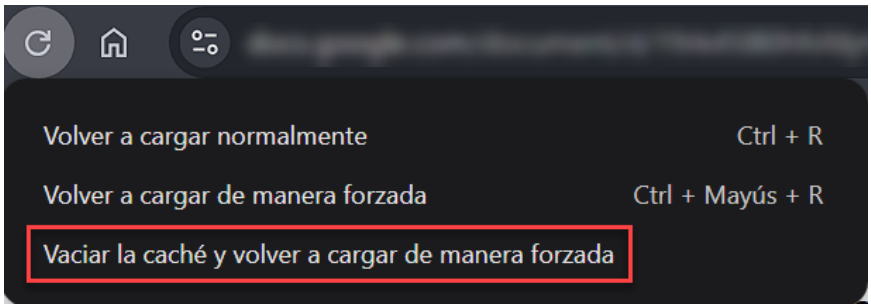
Recordando que las ayudas de búsqueda se configuran utilizando las Metadata Extensions. Esto se hace a través de la anotación **@Consumption.valueHelpDefinition**, especificando la propiedad **use for validation** con el valor **true**, así como la ayuda de búsqueda y el campo donde se habilitará dicha ayuda mediante la propiedad **entity**.

Ejemplo:

```
@Consumption.valueHelpDefinition: [{ entity: { name:
'/DMO/I_Agency_StdVH',
                                    element: 'Agency ID' },
                                    useForValidation: true }]
```

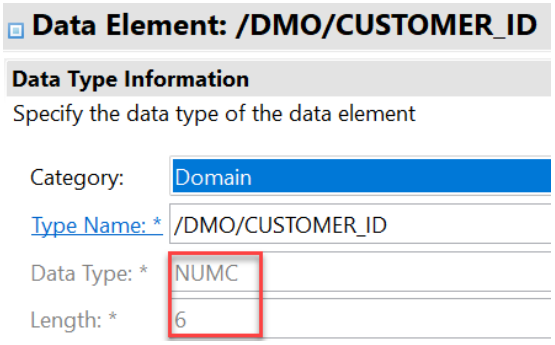
Es posible anular las validaciones de las ayudas de búsquedas comentando la propiedad **useForValidation: true,** de la anotación **@Consumption.valueHelpDefinition.** Para que de esta forma se puedan probar las validaciones que se implementarán en el código fuente los métodos de la clase Behavior Pool.

Para reflejar los cambios en la aplicación, puede ser necesario realizar un **Hard Reload** del navegador, eliminando la caché y recargando todos los recursos. Para realizar esta acción se debe presionar dentro del navegador la tecla **F12,** para luego al hacer clic derecho en el botón para refrescar la página , seleccionar la opción: **Vaciar la caché y volver a cargar de manera forzada.**

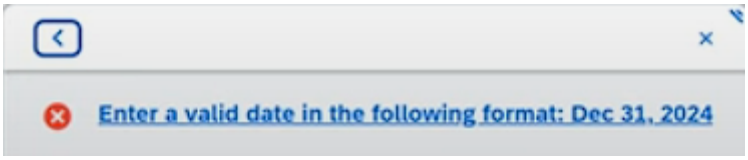


Validaciones Implícitas por Tipo de Dato:

Los campos de entrada tienen validaciones implícitas por el tipo de dato que les corresponde, asegurando que los datos ingresados sean válidos. Por defecto, si un campo ya sea de texto, número o de cualquier tipo excede el número máximo de caracteres permitidos, o si se ingresa un valor diferente al definido en el elemento de datos dentro del diccionario de datos de SAP, se ejecutará una validación automática. RAP se encarga de asegurar la calidad del dato basado en el tipo de dato asignado, incluso sin implementaciones adicionales en el código fuente.

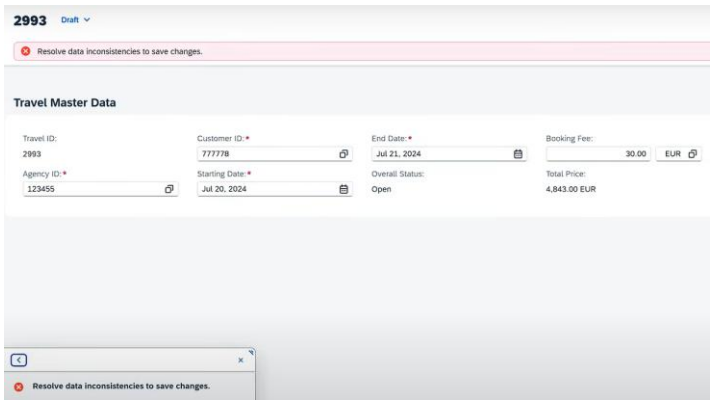


Los campos de tipo fecha también tienen validaciones implícitas que aseguran que se ingresen fechas válidas en el formato configurado en el navegador.



1.3. Validaciones en la Creación/Modificación

Las validaciones en el backend son procesos esenciales que garantizan que las operaciones de creación y modificación de datos en una aplicación cumplan con criterios específicos de calidad y consistencia. Estas validaciones se definen en el Behavior Definition y se implementan en la capa ABAP de SAP, asegurando que los datos ingresados sean correctos y válidos antes de ser procesados o almacenados.



A diferencia de las validaciones en el frontend, que proporcionan retroalimentación inmediata al usuario, las validaciones en el backend controlan la integridad de los datos desde el servidor, ofreciendo una capa adicional de seguridad y confiabilidad. Estas validaciones permiten bloquear transacciones que no cumplan con los requisitos definidos y asegurar la coherencia de la información en toda la aplicación.

Implementación de lógica de las validaciones:

Las validaciones en el backend son más complejas y permiten verificar datos incluso en ausencia de validaciones en el frontend. Esto es especialmente importante para solicitudes que llegan directamente a través del servicio OData, sin pasar por la interfaz de usuario. Los métodos para implementar la lógica de las validaciones se definen en la clase Behavior Pool, siempre que previamente se hayan agregado las sentencias **validation** asociadas en el Behavior Definition asociado. En el caso contrario se debe realizar este proceso utilizando la ayuda del framework a través del botón  para generar la lógica correspondiente y se declara de la siguiente forma:

```
validation validateComponent on save { create; field ComponentID; }
```

Los métodos de validación dentro de la clase del Behavior Pool se pueden identificar porque tienen el mismo nombre que su definición en el Behavior Definition y mantienen el siguiente formato:

```
METHODS validateComponent FOR VALIDATE ON SAVE
    IMPORTING keys FOR root_entity_name~validateComponent .
```

A continuación, se describen los parámetros clave de los métodos de las validaciones y sus funciones:

- **keys**: Este parámetro es una tabla interna que se usa para definir las claves principales que se deben validar. Es esencial para asegurar que los datos ingresados correspondan a registros existentes en la base de datos.

- **Failed**: Este parámetro es una estructura fundamental en los métodos utilizados para agregar la lógica de las validaciones, ya que al especificar un registro de la entidad no valido, permite gestionar errores y mostrar los mensajes de advertencia o error al usuario en la interfaz de usuario.
- **reported**: Este parámetro es una estructura que se utiliza para reportar los resultados de las validaciones. Permitiendo implementar mensajes de error que informen al usuario sobre las razones del bloqueo, aunque en esta lección no se aborda la propagación de mensajes personalizados.

Ejemplo:

Por lo general la lógica de las validaciones comienza con una lectura a la entidad raíz para obtener los datos necesarios y realizar las validaciones en base a ellos.

```
METHOD validateComponent.

  READ ENTITIES OF root_entity_name IN LOCAL MODE
   ENTITY RootEntityAliasName
   FIELDS (component_id)
   WITH CORRESPONDING #( keys )
   RESULT DATA(lt_root_entity).

  DATA lt_database_table TYPE SORTED TABLE OF database_table
                         WITH UNIQUE KEY client component_id.
```

Luego eliminar registros duplicados de la entidad de ser requerido y limpiar los datos que no son necesarios para la validación.

```
lt_database_table = CORRESPONDING #( lt_root_entity DISCARDING
DUPLICATES MAPPING component_id = ComponentID EXCEPT * ).

  DELETE lt_database_table WHERE component_id IS INITIAL.
```

Luego realizar una iteración a los registros de la entidad previamente consultados y filtrados. Para verificar si los valores ingresados en los campos cumplen con los criterios de calidad, un ejemplo sería validar la existencia de un componente clave en una tabla de persistencia.

```
  IF lt_database_table IS NOT INITIAL.
    SELECT FROM database_table  AS ddbb
    INNER JOIN @lt_database_table AS http_req ON
                  ddbb~component_id EQ http_req~component_id
    FIELDS ddbb~component_id
    INTO TABLE @DATA(lt_valid_records).
  ENDIF.

  LOOP AT lt_root_entity INTO DATA(ls_root_entity).
    IF ls_root_entity-ComponentID IS NOT INITIAL AND NOT line_exists(
lt_valid_records[ component_id = ls_root_entity-ComponentID ] ) .
```

Por último y más importante utilizar la estructura **failed,** mediante la especificación de un registro no válido para bloquear el estado transaccional e impedir que se realicen cambios en la capa de persistencia mediante un mensaje de error.

```
      APPEND VALUE #( %tky = ls_root_entity-%tky ) TO failed-
RootEntityAliasName .
    ENDIF.
  ENDLOOP .
 ENDMETHOD.
```

1.4. Validación en Múltiples Campos

Las validaciones en múltiples campos en RAP son esenciales para asegurar que los datos ingresados en una aplicación cumplan con las reglas de negocio definidas. Al agrupar campos relacionados en una sola validación y utilizar métodos específicos dentro del Behavior Pool, se puede garantizar que las relaciones entre los campos sean coherentes y que los datos sean válidos antes de ser procesados o almacenados. Esta metodología no solo mejora la calidad de los datos, sino que también proporciona una experiencia de usuario más robusta y confiable.

De igual forma que el concepto previamente explicado Behavior definition, define la validación agrupando los campos a validar mediante la declaración **validation**, especificando el nombre y los campos implicados.

validation validateComponent **on save { create; field** Component, Component**; }**

La implementación de la lógica de la validación se realiza de manera similar al concepto anterior, pero validando múltiples campos obtenidos de la entidad raíz y devolviendo el registro no válido a la estructura **failed** para mostrar el error en la interfaz de usuario. Esta validación se realiza en el método asociado, que lleva el mismo nombre dentro del Behavior Pool. Este método lee los valores de los campos y aplica las reglas de validación definidas.

1.5. Validación en la Acción

La validación de acciones en aplicaciones empresariales implica asegurarse de que las acciones, como la pulsación de botones en una interfaz de usuario, cumplan con ciertos criterios antes de ser ejecutadas. Este proceso no solo incluye la validación de datos ingresados por el usuario, sino también la verificación de que las acciones solicitadas sean lógicas y permitidas dentro del contexto de la aplicación.

Por lo general, estas validaciones se realizan en las acciones que tienen asociados parámetros a través de una entidad abstracta. Dichos parámetros poseen validaciones implícitas basadas en el tipo de dato definido mediante un elemento de datos. Sin embargo, hay situaciones en las que estas validaciones no son suficientes, por lo que en los métodos de acción con parámetros se añaden validaciones adicionales del tipo de datos antes de continuar con la lógica de la operación de negocio.

Recordando que se definen primeramente en el Behavior Definition por medio de las sentencias action:

```
action ( features : instance, authorization : update ) actionName
  parameter absytact_entity_name result [1] $self;
```

Implementación Técnica:

Recordando que es por medio de la tabla interna **keys,** y el campo **%params,** donde se puede acceder a los parámetros especificados por el usuario. Sin embargo, es necesario utilizar el campo **%tky** para identificar el registro exacto en la tabla mediante la clave técnica.

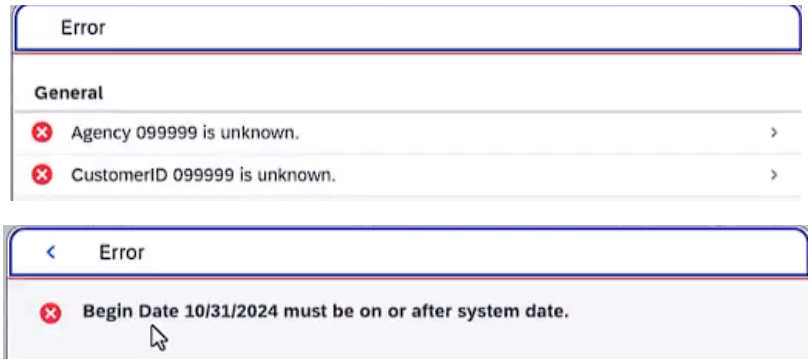
Se realiza una iteración de los registros de la tabla key para extraer la clave que se utilizará para obtener el parámetro especificado por el usuario en los parámetros de entrada. Estas iteraciones se llevan a cabo mediante un bucle, asignando cada registro a un field symbol. Se establecen los condicionales para validar el parámetro. Finalmente, se obtiene el registro con el parámetro especificado no válido junto con la clave técnica y se indica en la estructura **failed** en el campo que corresponde a la entidad raíz, para mostrar en pantalla un mensaje de error que impida continuar con la lógica de la operación de negocio.

Ejemplo:

```
DATA(lt_keys) = keys.
LOOP AT lt_keys ASSIGNING FIELD-SYMBOL(<ls_key>)
WHERE %param-parameter  conditions…
  * Set authorizations to root entity records
  APPEND VALUE #( %tky = <ls_key>-%tky ) TO failed-root_entity_name or    alias.
ENDLOOP .
CHECK sy-subrc NE 0.
```

1.6. Mensajes en Validaciones

La propagación de mensajes en la interfaz de usuario y las validaciones en un framework RAP (RESTful ABAP Programming Model) se centran en informar al usuario sobre errores y validaciones ejecutadas en el backend. Este proceso asegura que los mensajes sean claros, específicos y orienten al usuario a corregir entradas erróneas en formularios. A través de componentes y clases de mensajes específicos, el sistema permite una experiencia de usuario eficiente y precisa al bloquear el estado transaccional en caso de errores y proporcionar mensajes detallados vinculados a los campos de entrada correspondientes.



Implementación de mensajes personalizados:

Cuando se ejecuta una validación, se bloquea el estado transaccional. Esto se maneja a través del campo que corresponde a la entidad raíz en la estructura **failed** indicando al framework que dicho registro no es válido y no se deben persistir los cambios si hay errores, y posteriormente se configura el mensaje de validación.

Los mensajes resultantes de una validación se pueden personalizar en la interfaz de usuario utilizando el campo que corresponde a la entidad raíz en la estructura **reported**. El campo se basa en una tabla interna del tipo **reported late** de la entidad raíz, que contiene varios campos. De estos campos, los fundamentales para gestionar los mensajes de validación son los siguientes:

- **%msg:** Este campo es fundamental para la propagación de mensajes de error, advertencia o información. Es del tipo de referencia de la interfaz **if_abap_behv_message**, por lo que, para personalizar los mensajes a través de este campo, se debe

instanciar una clase de excepción que implemente dicha interfaz. Estas clases permiten construir mensajes informativos y orientativos para el usuario. Además, deben estar vinculadas a clases de mensaje, que se utilizarán mediante los parámetros aceptados en el constructor. Las cuentas trial de BTP incluyen la clase de excepción **/dmo/cm/flight_messages**, que puede usarse para definir algunos mensajes de validación y también como modelo para la creación de clases de excepción personalizadas.

```
CLASS /dmo/cm_flight_messages DEFINITION
  PUBLIC
  INHERITING FROM cx_static_check
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.

    INTERFACES if_t100_dyn_msg .
    INTERFACES if_t100_message .
    INTERFACES if_abap_behv_message .

    CONSTANTS:
      gc_msgid TYPE symsgid VALUE '/DMO/CM_FLIGHT',

      BEGIN OF customer_unkown,
        msgid TYPE symsgid VALUE '/DMO/CM_FLIGHT',
        msgno TYPE symsgno VALUE '001',
        attr1 TYPE scx_attrname VALUE 'MV_CUSTOMER_ID',
        attr2 TYPE scx_attrname VALUE '',
        attr3 TYPE scx_attrname VALUE '',
        attr4 TYPE scx_attrname VALUE '',
      END OF customer_unkown,
```

Utilizando los parámetros del constructor de la clase de excepción se puede personalizar el tipo de mensaje para las validaciones tomando como referencia la clase **/dmo/cm/flight_messages**, los campos fundamentales para la personalización son los siguientes:

- **textid:** Se utiliza para seleccionar un tipo de constante que posee aspectos fundamentales como el nombre de la clase (campo msgid), el mensaje utilizado (campo msgno) y los atributos **attr1, attr2, attr3** y **attr4**. Utilizados para mostrar información adicional en el mensaje de la validación.
- **campos:** Son parámetros que están vinculados a variables que pueden aparecer en el mensaje para

mostrar el campo validado, dependiendo del tipo de constante seleccionada. Por lo general, el atributo **attr1** del campo **textid** siempre está vinculado a algunas de las siguientes variables que forman parte de los parámetros del constructor de la clase de excepción **/dmo/cm/flight_messages: travel_id, booking_id, booking_supplement_id, agency_id, customer_id, carrier_id, connection_id, supplement_id, begin_date, end_date, booking_date, flight_date, status** y **currency_code.**

- **severity:** El campo severity se utiliza para indicar el tipo de mensaje. Esto ayuda a categorizar el mensaje según su nivel de importancia, y puede ser utilizado para la gestión de errores y la presentación de alertas en el sistema. Los tipos de mensaje se clasifican en: **none, error, warning, information** y **success** se pueden establecer utilizando la estructura **severity** de la interfaz **if_abap_behv_message.**

- **%element:** Se utiliza para vincular elementos específicos o campos de entrada de la entidad durante la validación. De esta forma permite colocar el cursor o indicar el foco al campo de entrada de la validación y cambiar el color del mismo, orientando al usuario directamente al lugar donde se debe realizar la corrección. Como este campo es del tipo **abp_behv_flag**, se especifica el campo deseado y se puede utilizar la constante **mk** de la interfaz **if_abap_behv**, para habilitar la función.
- **%tky:** Como es usual en los procedimientos de la lógica de las operaciones de la interfaz de usuario, es necesario la clave técnica para especificar el registro correcto dentro de la entidad raíz.

Ejemplo:

```
LOOP AT lt_root_entity INTO DATA(ls_root_entity).
```

```
    IF ls_root_entity-ComponentID IS NOT INITIAL AND NOT line_exists(
lt_valid_records[ component_id = ls_root_entity-ComponentID ] ) .
  * Set authorizations to root entity records
     APPEND VALUE #( %tky = ls_root_entity-%tky ) TO failed-RootEntityAliasName .

* Customize error messages
     APPEND VALUE #( %tky = ls_root_entity-%tky
                          %msg = NEW /dmo/cm_flight_messages(
                                 textid =
/dmo/cm_flight_messages=>customer_unkown
                                 customer_id = ls_root_entity-ComponentID
                                 severity = if_abap_behv_message=>severity-error )
                          %element-CustomerID = if_abap_behv=>mk-on )
                            TO       reported-RootEntityAliasName.
    ENDIF.
  ENDLOOP .
ENDMETHOD.
```

1.7. Mensajes en Acciones

La propagación de mensajes personalizados en ABAP es un proceso que permite la validación y comunicación de errores específicos durante la ejecución de acciones en la interfaz de usuario. En este contexto, se implementa una lógica que no solo bloquea estados transaccionales inválidos, sino que también genera mensajes de error detallados y personalizados que se muestran al usuario. Estos mensajes incluyen información específica sobre la causa del error, la severidad del mismo y la acción que lo provocó, mejorando la claridad y la experiencia del usuario.

Implementación de mensajes personalizados:

De manera similar a los mensajes en las validaciones la estructura **failed** se utiliza para registrar y manejar errores que ocurren durante el proceso de validación. Y la estructura **reported** se utiliza para gestionar y propagar mensajes de validación que deben ser comunicados al usuario. Sin embargo, para este caso, se utilizará un campo adicional en el parámetro de la entidad raíz de la estructura **reported** fundamental para gestionar los mensajes de las acciones durante la validación.

- **%op:** Se utiliza para vincular mensajes de error o información específica a una operación en particular que se ha realizado,

para estos casos una acción. Esto es especialmente útil para proporcionar retroalimentación detallada y contexto al usuario sobre la acción que provocó un error o un mensaje informativo.

Ejemplo:

```
DATA(lt_keys) = keys.
LOOP AT lt_keys ASSIGNING FIELD-SYMBOL(<ls_key>)
WHERE %param-parameter  conditions…
 * Set authorizations to root entity records
 APPEND VALUE #( %tky = <ls_key>-%tky ) TO failed-root_entity_name or    alias.

* Customize error messages
APPEND VALUE #( %tky = <ls_key>-%tky
                %msg = NEW /dmo/cm_flight_messages(
                           textid = /dmo/cm_flight_messages=>discount_invalid
                           severity = if_abap_behv_message=>severity-error )
                %op-%action-CustomAction = if_abap_behv=>mk-on
            ) TO       reported-RootEntityAliasName.
ENDLOOP .
CHECK sy-subrc NE 0.
```

1.8. Mensajes en Autorizaciones de Instancia

La propagación de mensajes personalizados en autorizaciones de instancia dentro de ABAP es una técnica que permite enviar mensajes claros y específicos al usuario cuando se realizan chequeos de autorizaciones, tanto globales como de instancia. Esta técnica es esencial para informar al usuario sobre la razón por la cual no se permiten ciertas operaciones, como editar, eliminar o crear registros con IDs específicos. A través de la implementación en el Behavior Pool en el método **get_instance_authorizations**, se configura la lógica que valida y propagar estos mensajes personalizados, mejorando así la transparencia y comprensión del sistema por parte del usuario.

Implementación de mensajes personalizados:

El proceso es similar a los mensajes en las validaciones anteriores con la diferencia que los resultados de las autorizaciones se realiza por medio de la estructura **result.** Por lo que luego de verificar que un registro no cumple con las condiciones de la autorización por dicha

tabla se personaliza el mensaje de error utilizando el campo que corresponde a la entidad raíz en la estructura **reported**.

Ejemplo:

```
* Iterate through the root entity records
  DATA(lv_technical_name) = cl_abap_context_info=>get_user_technical_name( ).
  LOOP AT lt_root_entity INTO DATA(ls_root_entity).
    IF lv_technical_name EQ 'User Name' AND ls_root_entity-component EQ 'Value'.
      lv_update_granted = abap_true.
    ELSE.
      lv_update_granted = abap_false.

* Customize error messages
    APPEND VALUE #( %tky = ls_root_entity-%tky
                    %msg = NEW /dmo/cm_flight_messages(
                               textid = /dmo/cm_flight_messages=>not_authorized
                               severity = if_abap_behv_message=>severity-error )
                    %element-component = if_abap_behv=>mk-on )
                    TO       reported-RootEntityAliasName.
    ENDIF.

* Set authorizations to root entity records
    APPEND VALUE #( LET upd_auth = COND #(
                          WHEN lv_update_granted EQ abap_true
                           THEN if_abap_behv=>auth-allowed
                           ELSE if_abap_behv=>auth-unauthorized  )
                           IN
                           %tky = ls_root_entity-%tky
                           %update = upd_auth
                           %action-Edit = upd_auth ) TO result.
  ENDLOOP.
```

1.9. Mensajes en Autorizaciones Globales

La propagación de mensajes personalizados en autorizaciones globales dentro de ABAP es un proceso que permite enviar mensajes claros y específicos al usuario cuando se realizan chequeos de autorizaciones a nivel global. Este mecanismo asegura que los usuarios reciban retroalimentación precisa sobre las operaciones que no están autorizados a realizar, como la creación, actualización o eliminación de registros. Los mensajes se gestionan a través de la estructura **reported**, utilizando clases de mensajes y constantes predefinidas. Esto mejora la transparencia y comprensión del

sistema, proporcionando una experiencia de usuario más coherente y efectiva.

Implementación de mensajes personalizados:

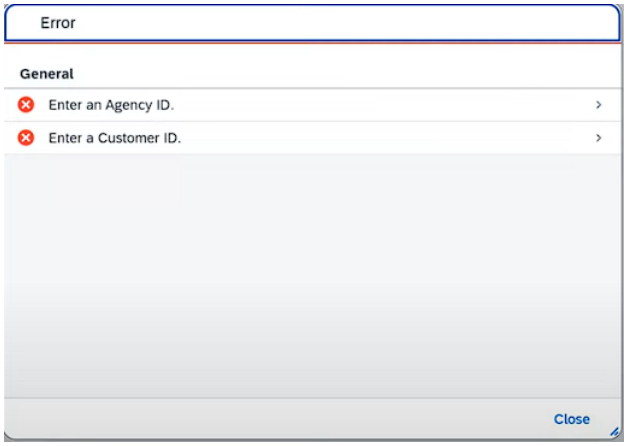
El proceso es similar a los mensajes en las validaciones de las autorizaciones de instancia con la diferencia que se realiza en el método **get_global_authorizations** donde los resultados de las autorizaciones se realiza por medio de estructura **result.** Por lo que luego de verificar que un registro no cumple con las condiciones de la autorización y modificar la estructura **result,** se personaliza el mensaje de error utilizando el campo que corresponde a la entidad raíz en la estructura **reported**. Al estar en un contexto global no es necesario identificar la clave técnica y tampoco colocar el cursor en alguno de los elementos de la interfaz con los campos **%tky** y **%element** respectivamente.

Ejemplo:

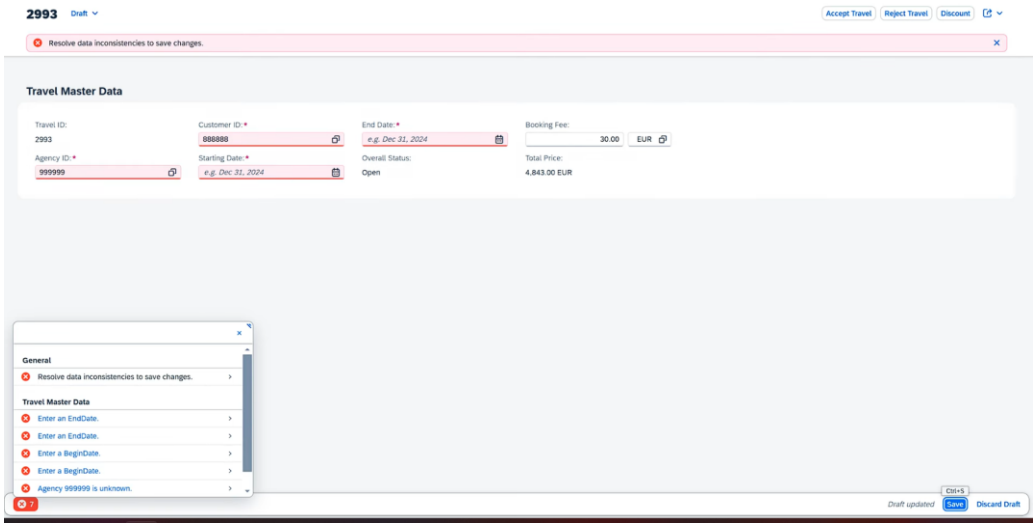
```
METHOD get_global_authorizations.
  DATA(lv_technical_name) = cl_abap_context_info=>get_user_technical_name( ).
 * Set authorizations to root entity records
  IF requested_authorizations-%create EQ if_abap_behv=>mk-on.
    IF lv_technical_name EQ 'User Name' .
      result-%create = if_abap_behv=>auth-allowed.
    ELSE.
      result-%create = if_abap_behv=>auth-unauthorized.

    * Customize error messages
      APPEND VALUE #( %msg = NEW /dmo/cm_flight_messages(
                                  textid = /dmo/cm_flight_messages=>not_authorized
                                  severity = if_abap_behv_message=>severity-error )
                    ) TO        reported-RootEntityAliasName.
    ENDIF.
  ENDIF.
ENDMETHOD.
```

1.10. Agrupación Mensajes con State Area

Las validaciones de campos en la interfaz de usuario se propagan en un contexto general por defecto en una ventana en el centro de la interfaz de usuario.



Para mejorar esto, se puede agrupar mensajes específicos utilizando el campo **%state_area**, del campo de la entidad raíz en la estructura **reported.** Lo que permite mejorar la claridad y organización de los mensajes de error o advertencia. Los mensajes agrupados facilitan la identificación de los errores, ya que se muestran claramente en el popup de la parte inferior izquierda de la pantalla.



Implementación del State Area:

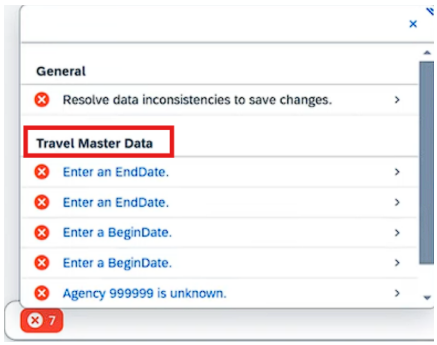
Para implementar esta propiedad en la interfaz de usuario se debe agregar al campo **%state_area** a todos los mensajes personalizados de la validación que se requieran agrupar un identificador el cual es una cualquier cadena de caracteres permitiendo a agrupar los mensajes. La agrupación de mensajes utilizando **State Area** en validaciones de la interfaz de usuario es una práctica esencial en ABAP para mejorar la claridad y organización de los mensajes de error o advertencia.

Ejemplo:

```
  LOOP AT lt_root_entity INTO DATA(ls_root_entity).
   IF ls_root_entity-ComponentID IS NOT INITIAL AND NOT line_exists(
lt_valid_records[ component_id = ls_root_entity-ComponentID ] ) .
    APPEND VALUE #( %tky = ls_root_entity-%tky ) TO failed-RootEntityAliasName .
    APPEND VALUE #( %tky = ls_root_entity-%tky
                          %msg = NEW /dmo/cm_flight_messages(
                                 textid =   /dmo/cm_flight_messages=>customer_unkown
                                 customer_id = ls_root_entity-ComponentID
                                 severity = if_abap_behv_message=>severity-error )
                          %state_area = 'VALIDATE_COMPONENT'

                          %element-CustomerID = if_abap_behv=>mk-on )
                                TO        reported-RootEntityAliasName.
   ENDIF.
  ENDLOOP .
ENDMETHOD.
```

Puntos importantes a considerar:

- El título que aparece en la ventana de la agrupación en el **State Area**, corresponde al especificado en el Metadata Extensions asociado a la entidad de consumo con la anotación **@UI.facet** con la propiedad **label**.



Ejemplo:

```
@UI.facet: [{ id: 'InternalIdentifier',
purpose: #STANDARD,
type: #IDENTIFICATION_REFERENCE, label: 'UserLabel',
position: 10 }, { id, purpose, type, label, position }]
ComponentID;
```

- Por lo general se anulan las validaciones de las ayudas de búsquedas en el Metadata Extensions asociado a la entidad de consumo para realizar las validaciones implementadas en el código fuente los métodos de la clase Behavior Pool para luego realizar un Hard Reload para para reflejar los cambios en la interfaz de usuario. Concepto estudiado en el punto **UI – Use For Validation.**