Tuesday, December 30, 2024
By: Devi Prasad Mishra

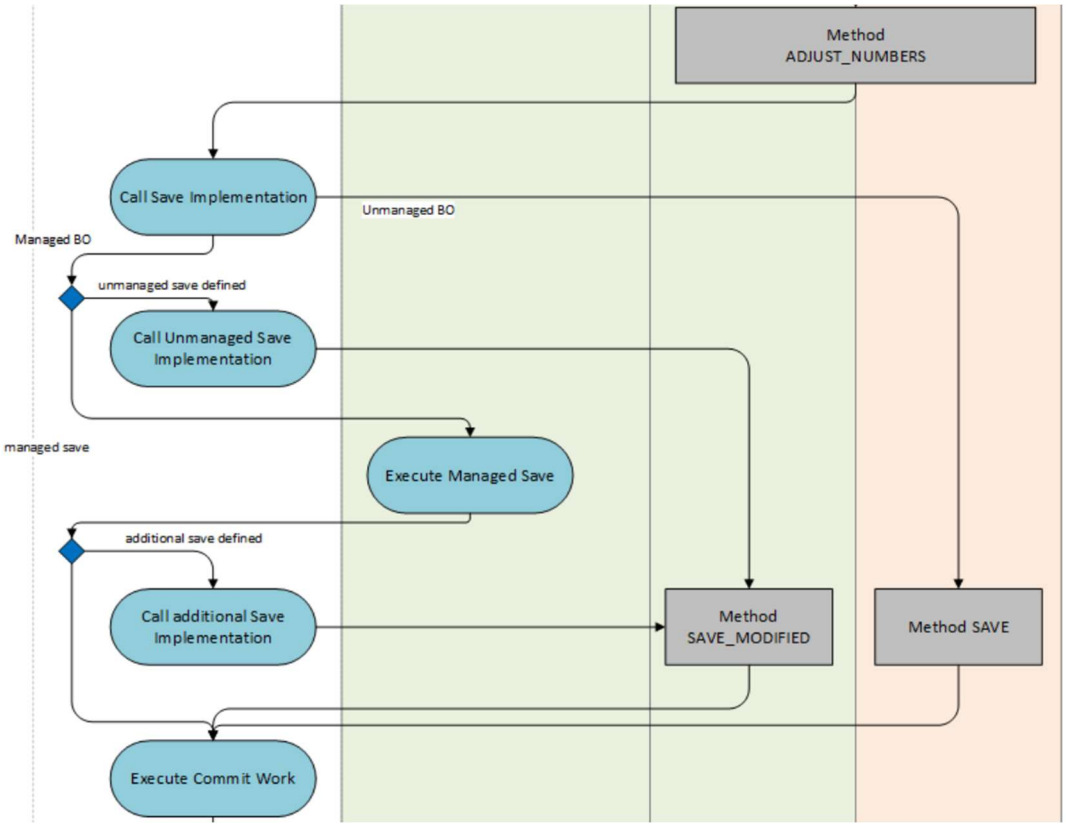
Managed Scenario Save Sequence(Additional Save and Unmanaged Save).

The managed save sequence is a key concept in the RAP (Restful Application Programming) framework used to handle persistence operations for entity instances in ABAP. It automates the process of saving changes made to an entity while allowing customization when needed.

Key Components of the Managed Save Sequence

1. **Standard Behavior**
    - Automatically saves changes made to entity instances in the database.
    - Ensures consistent data persistence without requiring additional coding.
2. **Steps in the Sequence**
    - Data Validation: Validates the modified instance to ensure data integrity.
    - Data Modification: Updates or inserts the instance data into the database.
    - Lifecycle Management: Manages the lifecycle events triggered during the save process.
    - Post-Save Actions: Completes any final actions after the data persisted.
3. **Customizing with Extra Save**
    - The extra save feature allows adding additional logic to the save process, such as logging changes or triggering external services.
    - Developers can implement additional steps using the save_modified method in a saver class.
4. **Change Tracking**
    - Logs changes made to an instance, such as modified fields or newly added data.
    - These changes can be written to a log table for auditing or debugging purposes.

Tuesday, December 30, 2024
By: Devi Prasad Mishra



1. Additional Save

The RAP framework supports adding extra functionality to the managed save sequence via the extra save feature, allowing reuse of services like change documents and application logs. By default, managed saves handle this, but additional steps can be included using the saver class and the save modified method. When an entity instance is updated, changes are saved, including updated fields, data, and completed operations. In this scenario, modified data is recorded in a log table.

Action: Create the log table to store the changes.

```
@EndUserText.label : 'log table'
@AbapCatalog.enhancement.category : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #RESTRICTED
define table zlog_tab_upd {

  key changeid      : abap.raw(16) not null;
  salerid           : zsale;
  materialid        : zmatno;
  imageid           : zimgid;
  change_operation  : abap.char(10);
  changed_field     : abap.char(40);
  changed_value     : abap.char(40);
  created_at        : timestampl;

}
```

Tuesday, December 30, 2024
By: Devi Prasad Mishra

To enable additional saving in a managed scenario:

a) **Behavior Definition**: Define the additional save for the entity.
b) **Implementation Class**: Create a local class and implement the save_modified method.

This setup ensures the modified data is handled appropriately.

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

define behavior for zi_saler
implementation in class zbp_i_saler unique
persistent table zsalers_table
with additional save
lock master
authorization master ( instance )
early numbering
{

  create;
  update;
  delete;
  field ( readonly ) SalerId;
  field ( mandatory ) name;
  association _material { create; }
```

When a new instance is created, additional data is stored in the entity's database and log table.

Steps:

1) **Define Saver Class**: Create a saver class in the behavior definition.
2) **Behavior Pool**: Implement required methods in the behavior pool.
3) **Save_Modified Method**: Implement the save_modified method in the saver class to handle the additional save logic.

This ensures proper handling of additional data during entity creation.

```
CLASS lhs_zi_saler DEFINITION  INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS : save_modified REDEFINITION.
ENDCLASS.
```

```
CLASS lhs_zi_saler IMPLEMENTATION.

 METHOD save_modified.
   DATA : lt_log TYPE STANDARD TABLE OF zlog_tab_upd.
   DATA : lt_log_c TYPE STANDARD TABLE OF zlog_tab_upd.
*---------------------------------------------------------------------
* update the log when we create a seller information
*---------------------------------------------------------------------

   IF create-zi_saler IS NOT INITIAL.
     lt_log = CORRESPONDING #( create-zi_saler ).

     LOOP AT lt_log ASSIGNING FIELD-SYMBOL(<lfs_log>).
       GET TIME STAMP FIELD  <lfs_log>-created_at.

       READ TABLE create-zi_saler ASSIGNING FIELD-SYMBOL(<lfs_create>) WITH TABLE KEY entity
                                                                       COMPONENTS   SalerId = <lfs_log>-salerid.
       IF sy-subrc IS INITIAL.
*        If Sales is created
         IF <lfs_create>-%control-SalerId = cl_abap_behv=>flag_changed.
           <lfs_log>-changed_field = 'Saler ID'.
           <lfs_log>-change_operation = 'Created'.
           <lfs_log>-changed_value = <lfs_create>-SalerId.
           TRY.
               <lfs_log>-changeid = cl_system_uuid=>create_uuid_x16_static(  ).
             CATCH cx_uuid_error.
           ENDTRY.
           APPEND <lfs_log> TO lt_log_c.
         ENDIF.
       ENDIF.
       UNASSIGN <lfs_create>.
     ENDLOOP.
     UNASSIGN <lfs_log>.
   ENDIF.
```

Tuesday, December 30, 2024

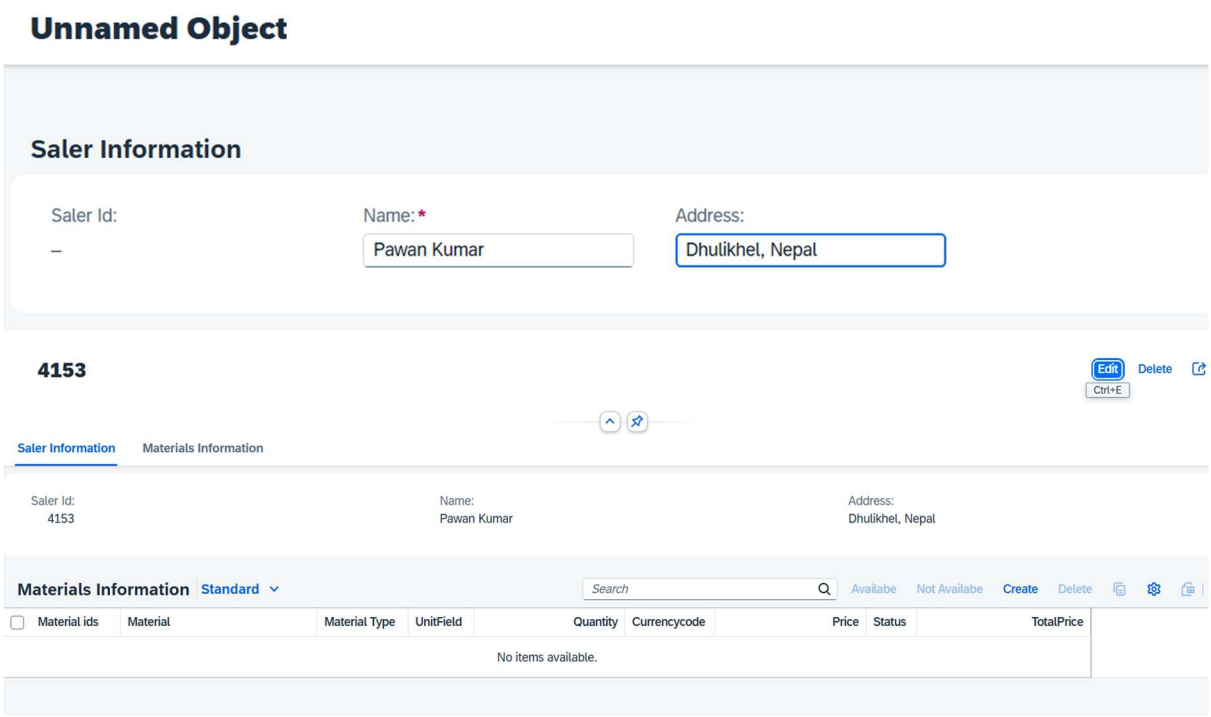
By: Devi Prasad Mishra

```
*---------------------------------------------------------------------
* update the log when we Delete a seller information
*---------------------------------------------------------------------
    IF delete-zi_saler IS NOT INITIAL.
      lt_log = CORRESPONDING #( delete-zi_saler ).

      LOOP AT lt_log ASSIGNING <lfs_log>.
        <lfs_log>-change_operation = 'Deletion'.
        GET TIME STAMP FIELD  <lfs_log>-created_at.

*      If Sales is Deleted
        <lfs_log>-changed_field = 'Saler ID'.
        <lfs_log>-changed_value = <lfs_log>-SalerId.
        TRY.
            <lfs_log>-changeid = cl_system_uuid=>create_uuid_x16_static(  ).
          CATCH cx_uuid_error.
        ENDTRY.
        APPEND <lfs_log> TO lt_log_c.

      ENDLOOP.
      UNASSIGN : <lfs_log>.
    ENDIF.
```

```
*---------------------------------------------------------------------
* update the log when we Update a seller information
*---------------------------------------------------------------------
    IF update-zi_saler IS NOT INITIAL.
      lt_log = CORRESPONDING #( update-zi_saler ).

      LOOP AT update-zi_saler ASSIGNING FIELD-SYMBOL(<lfs_update>).
        ASSIGN lt_log[ salerid = <lfs_update>-SalerId ] TO <lfs_log>.

        <lfs_log>-change_operation = 'Updated' ##NO_TEXT.
        GET TIME STAMP FIELD  <lfs_log>-created_at.

*      If Name of seller is Updated
        IF <lfs_update>-%control-Name = if_abap_behv=>mk-on.
          <lfs_log>-changed_field = 'Name'.
          <lfs_log>-changed_value = <lfs_update>-Name.
          TRY.
              <lfs_log>-changeid = cl_system_uuid=>create_uuid_x16_static(  ).
            CATCH cx_uuid_error.
          ENDTRY.
          APPEND <lfs_log> TO lt_log_c.
        ENDIF.

*      If Address of seller is Updated
        IF <lfs_update>-%control-Address = if_abap_behv=>mk-on.
          <lfs_log>-changed_field = 'Address'.
          <lfs_log>-changed_value = <lfs_update>-Address.
          TRY.
              <lfs_log>-changeid = cl_system_uuid=>create_uuid_x16_static(  ).
            CATCH cx_uuid_error.
          ENDTRY.
          APPEND <lfs_log> TO lt_log_c.
        ENDIF.

      ENDLOOP.
      UNASSIGN : <lfs_log>, <lfs_update>.
    ENDIF.
*     insert the log data to the table
    INSERT zlog_tab_upd FROM TABLE @lt_log_C.
```

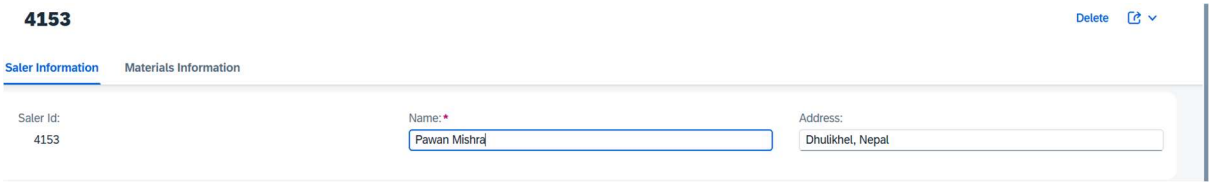
Tuesday, December 30, 2024
By: Devi Prasad Mishra

For creating an instantiation operation.



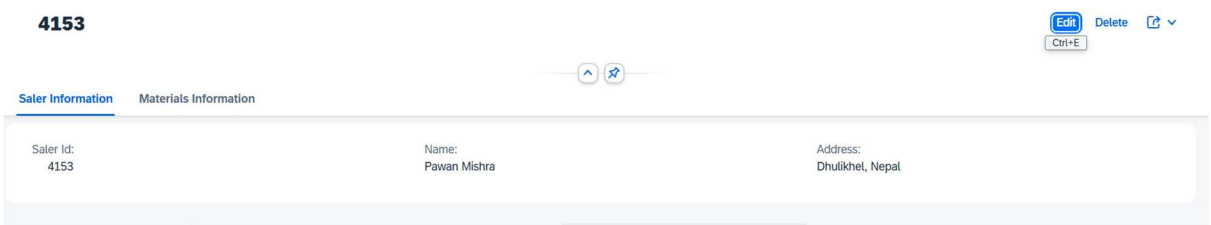
In log table.

For Edit.



In log table

For Delete.



Tuesday, December 30, 2024
By: Devi Prasad Mishra

In log table

CE3B9E4417721...	4149	Created	Saler ID	4149	20241229115538....
CE3B9E4417721...	4149	Updated	Name	Devi Prasad	20241230174056....
EA726B4CB5201...	4153	Created	Saler ID	4153	20241231084619....
EA726B4CB5201...	4153	Updated	Name	Pawan Mishra	20241231085016....
EA726B4CB5201...	4153	Deletion	Saler ID	4153	20241231085212....

2. Unmanaged Save

Unmanaged save is ideal when the default save logic in the RAP framework (managed save) cannot handle specific business requirements. It gives developers full control over how data persisted, enabling customization for complex scenarios like:

- Custom Table Structures: When data is stored in unconventional formats or across multiple tables.
- Advanced Integrations: When saving data involves external APIs or systems.
- Complex Logic: For operations requiring extensive calculations, validations, or dependencies between entities.

Steps to implement unmanage save.
Before that ensure persistence, table is not specified, as unmanaged save implementation cannot have persistence table specified to it.

Step 1: Include the **unmanaged save** keywords for all nodes where unmanaged behavior is required.

```
define behavior for zi_images
implementation in class zcl_b_images unique
//persistent table zimages_table
with unmanaged save
lock dependent by _saler
authorization dependent by _saler
early numbering
{
  update;
  delete;
  field ( readonly ) ImageNumber, Matno, Salerid;
  association _material;
  ancestor association _saler;
```

Step 2: Redefine method **save_modified** in Saver Local class implementation level.

```
CLASS lhs_zi_saler DEFINITION  INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS : save_modified REDEFINITION.
ENDCLASS.

CLASS lhs_zi_saler IMPLEMENTATION.

  METHOD save_modified.
```

Tuesday, December 30, 2024

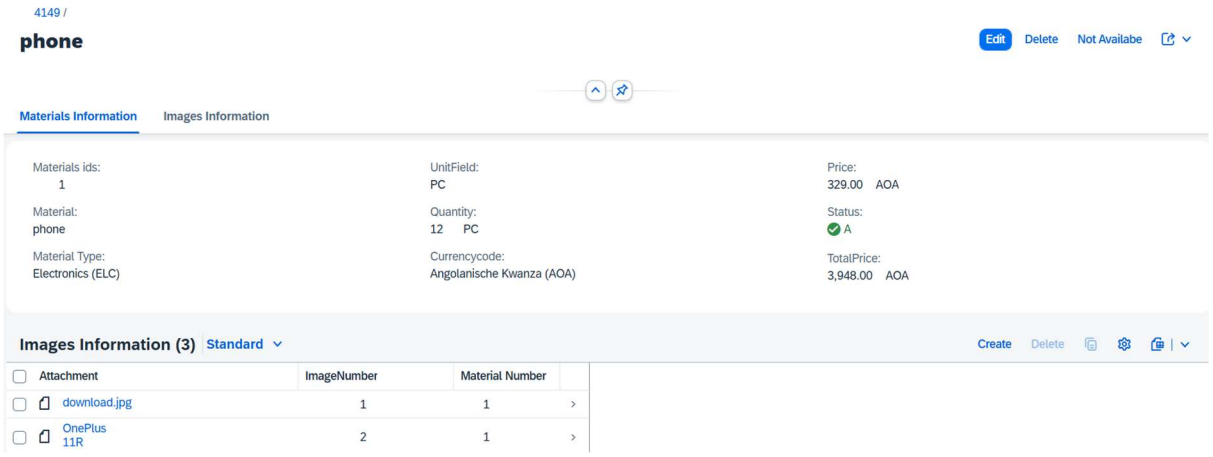
By: Devi Prasad Mishra

```
* ---------------------------------------------------------------------------------------------------
*  Unmanaged Save for Image entity
* ---------------------------------------------------------------------------------------------------
   IF create-zi_images IS NOT INITIAL.
     DATA : lt_images TYPE STANDARD TABLE OF zimages_table,
            ls_return TYPE bapiret2.

     lt_images = CORRESPONDING #( create-zi_images MAPPING image_number = ImageNumber
                                                           attachment = ImageAttachment
                                                           id = Id
                                                           matno = Matno
                                                           salerid = Salerid
                                                           name = ImageName ).

*     CALL FUNCTION 'ZIMAGES_CUD' DESTINATION 'NONE'
*       EXPORTING
*         createx = 'X'
*         values  = lt_images
*       IMPORTING
*         return  = ls_return.
     INSERT zimages_table FROM  TABLE @lt_images.


   ENDIF.

   IF update-zi_images IS NOT INITIAL.
     lt_images = CORRESPONDING #( update-zi_images MAPPING image_number = ImageNumber
                                                           attachment = ImageAttachment
                                                           id = Id
                                                           matno = Matno
                                                           salerid = Salerid
                                                           name = ImageName ).
     UPDATE zimages_table  FROM  TABLE @lt_images.
   ENDIF.

   IF delete-zi_images IS NOT INITIAL.
     lt_images = CORRESPONDING #( delete-zi_images MAPPING image_number = ImageNumber
                                                           matno = Matno
                                                           salerid = Salerid ).

     DELETE zimages_table FROM  TABLE @lt_images.
   ENDIF.

 ENDMETHOD.

ENDCLASS.
```

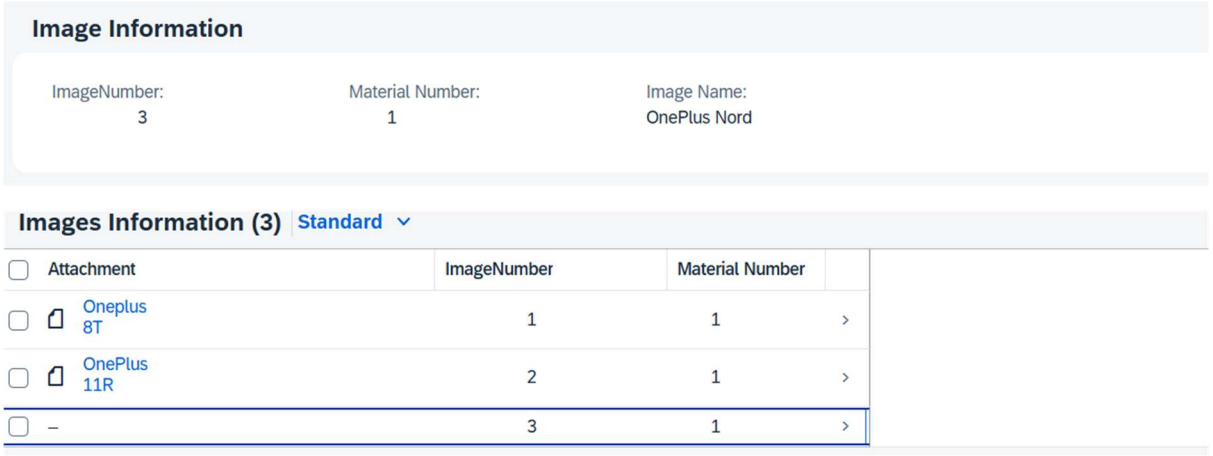
Here, I have used SQL queries, but instead, you can call external APIs or existing function modules to perform Create, Update, or Delete (CUD) operations.

Tuesday, December 30, 2024
By: Devi Prasad Mishra

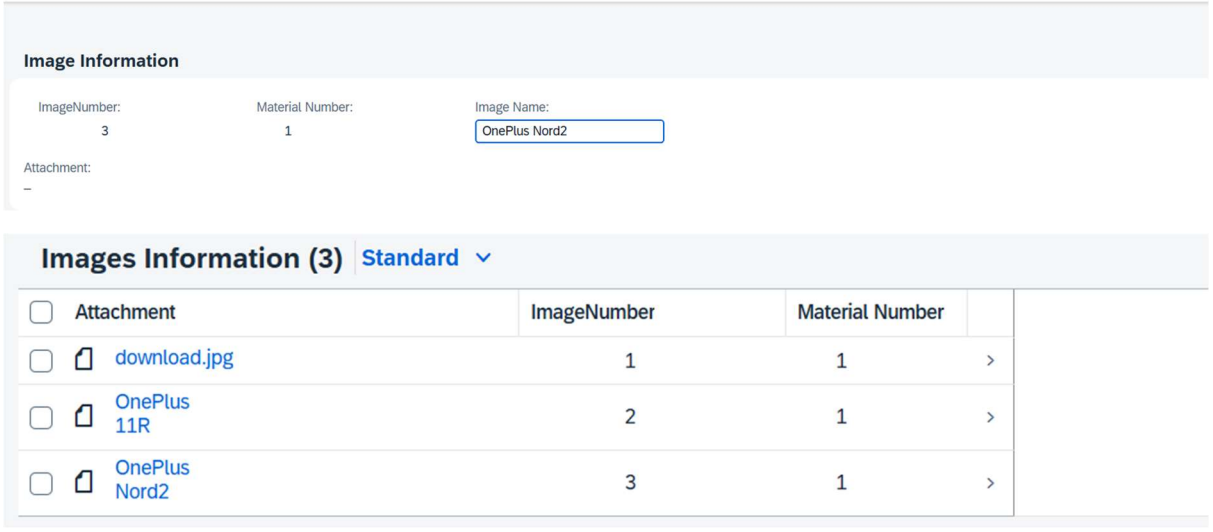
Fiori Application Preview.



Create

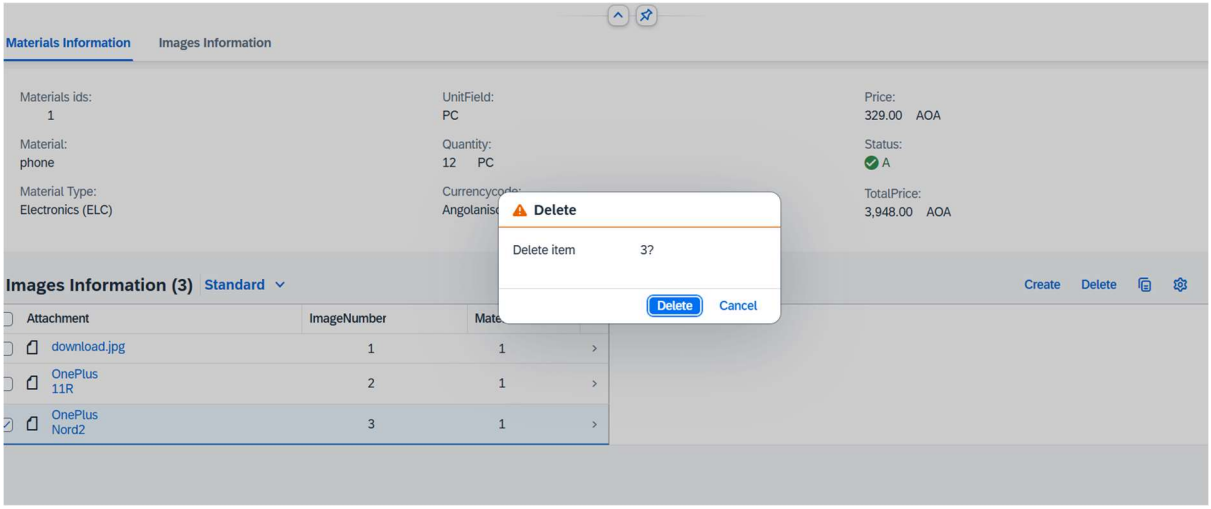


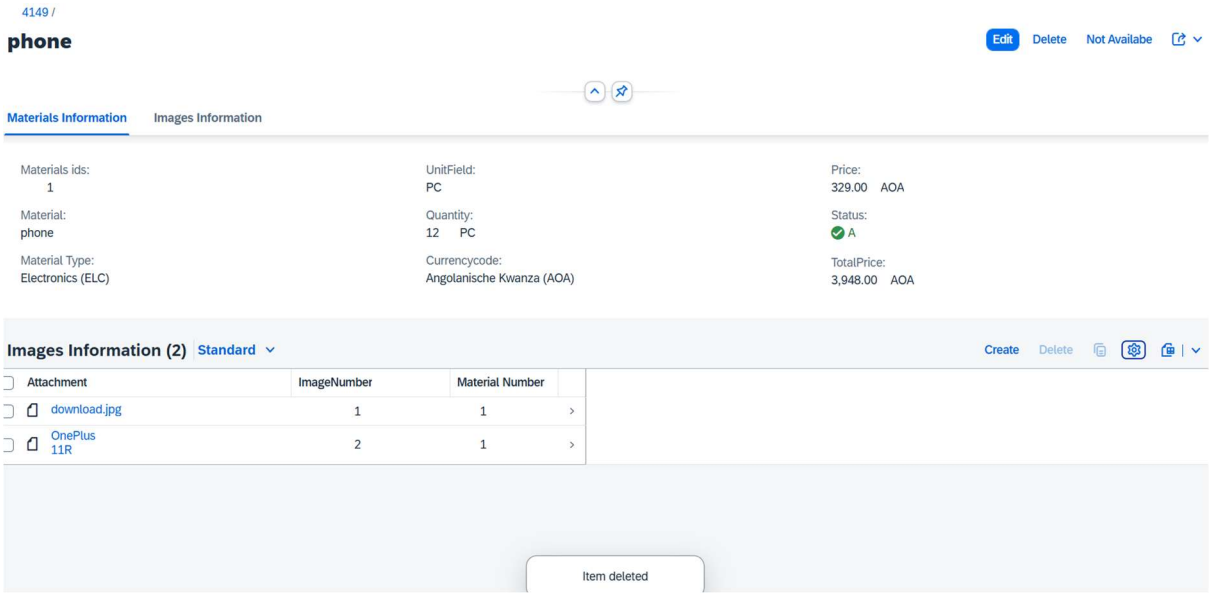
Update



Tuesday, December 30, 2024
By: Devi Prasad Mishra

Delete





Conclusion:

In RAP applications, both **Additional Save** and **Unmanaged Save** offer essential flexibility for customizing data persistence:

- **Additional Save** enhances the standard managed save sequence, allowing developers to introduce custom functionality (like logging or external integrations) while still utilizing the framework's built-in features for validation and lifecycle management.
- **Unmanaged Save** provides full control over the save process, enabling developers to implement custom Create, Update, delete (CUD) operations, such as using SQL queries, external APIs, or function modules, for more complex or non-standard requirements.

Both methods empower developers to tailor data management in the RAP framework, ensuring the application can meet diverse and complex business needs.