Oracle Fusion Cloud SCM

Configuring and Extending Product Lifecycle Management

24D

Oracle Fusion Cloud SCM
Configuring and Extending Product Lifecycle Management

24D

G12605-01

Author: Divya Begur, Usha Pereira

Contents

ORACLE

3 Extend Quality Management   59

4 Extend Innovation Management   63

5 FAQs   67

ORACLE

Get Help

There are a number of ways to learn more about your product and interact with Oracle and other users.

Get Help in the Applications

Use help icons ⑦ to access help in the application. If you don't see any help icons on your page, click your user image or name in the global header and select Show Help Icons.

Get Support

You can get support at *My Oracle Support*. For accessible support, visit *Oracle Accessibility Learning and Support*.

Get Training

Increase your knowledge of Oracle Cloud by taking courses at *Oracle University*.

Join Our Community

Use *Cloud Customer Connect* to get information from industry experts at Oracle and in the partner community. You can join forums to connect with other customers, post questions, suggest *ideas* for product enhancements, and watch events.

Learn About Accessibility

For information about Oracle's commitment to accessibility, visit the *Oracle Accessibility Program*. Videos included in this guide are provided as a media alternative for text-based topics also available in this guide.

Share Your Feedback

We welcome your feedback about Oracle Applications user assistance. If you need clarification, find an error, or just want to tell us what you found helpful, we'd like to hear from you.

You can email your feedback to *oracle_fusion_applications_help_ww_grp@oracle.com*.

Thanks for helping us improve our user assistance!

ORACLE

1 Extend Product Lifecycle Management

About This Guide

You can configure and extend the application interfaces and the business objects in Oracle Product Lifecycle Management (PLM) Cloud to support your business needs. The recommended tool for such configuration is Application Composer.

This guide describes the types of configuration that you can perform, and includes frequently asked questions about how to effectively use Application Composer.

Audience

This guide provides information on how administrators and implementors can make application changes using Application Composer.

Additional Information

The table provides a list of the guides and videos that have detailed information on the capabilities of Application Composer.

Related Information

Guide/Video	Description
Configuring Applications Using Application Composer	This guide provides information on how administrators and implementors can make application changes using Application Composer.
Groovy Scripting Reference	This document explains the basics of how to use Groovy scripting language to enhance the functionality of the application.
Watch Videos: Extending PLM Applications	These videos show how to extend and configure PLM objects.

Overview of Application Composer

Application Composer is a browser-based tool that you as an administrator can use to configure applications. You can use this tool to make data model changes.

Administrators can create and configure layouts to meet business requirements. For example, you can create a new object and related fields, and then create new interface pages to expose that object to users. Application Composer is a design-at-runtime tool, which means that you can navigate to Application Composer directly from a Cloud application, make your changes, and see that most changes take immediate effect, without having to sign back into the application.

For Oracle Quality Management, Oracle Innovation Management, and Oracle Product Development, you can use Application Composer to extend the application interface.

> **Note:** To configure change order attributes, use Extensible Flexfields (EFFs).

> **Note:** Application Composer is supported for use only in English. Additionally, Application Composer isn't supported for use with iPad devices.

Overview of Extensible Flexfields

An extensible flexfield provides a configurable expansion space that implementers, such as Oracle Fusion Applications customers, can use to configure additional attributes (segments) without additional programming.

> **Note:** This configuration applies to change orders, change requests, problem reports, corrective actions, items, and manufacturer parts. You can define fields for other objects in Application Composer.

Related Topics
- Extensible Flexfields

What You Can Configure

Use Application Composer to:

- Edit the display label and help text of standard fields;
- Create conditional layouts;
- Assign fields to layouts;
- Create fields of different types (such as text, long text, number, date, choice list, and check box) and add them to standard and administrator-defined objects;
- Define application actions using validation rules, triggers, and functions;
- Set field-level and object-level validation rules;
- Create new objects and child objects;

  > **Note:** You can't create child objects for change orders.

- Create new subtabs and connect to external applications;
- Hide or show objects and tabs;
- Use web services to create and update objects;
- Delete configured attributes that are created in a sandbox.

  > **Note:** Ensure that the configured attributes you're deleting aren't in use or have never been published.

Attributes, or fields, must be assigned to a layout for the application user to see and work with them. A conditional statement assigned to a layout determines when it's displayed and who can see it.

ORACLE

Configuration Scenarios

Let's look at some scenarios where you can tailor the application interface to achieve a specific requirement.

Scenario 1

Your business requires that the description field of all change orders must be automatically updated with the department and location entered by the user.

Solution:

1. In the Setup and Maintenance work area, navigate to:
   - Offering: Product Management
   - Functional Area: Change Orders
   - Task: Manage Change Order and New Item Request Header Descriptive Flexfields
2. Create global descriptive flexfields entitled **Department** and **Location**.
3. In Application Composer, create a script that copies the values of the **Department** and **Location** fields into the **Description** field.

Scenario 2

When users create a quality action, you want to make sure that the triage date is automatically set to 3 days after the creation date.

Solution: Within Application Composer, create a new date field called Triage Date. Define the default value of Triage Date using a Groovy script expression that captures the creation date and adds 3 days. Define an expression in a way that doesn't allow the field to be updated.

Scenario 3

Your business process requires that a requirements specification is created for every idea that has an Approved status. Also, the requirements specification must have the same name and description as the original idea.

Solution A: Within Application Composer, you can add a button to an idea that calls a web service. The web service will create a requirements specification and copy the name and description of the idea to the new requirements specification.

Solution B: Within Application Composer, you can create a trigger that calls a web service whenever an idea is set to Approved status, which will create a requirements specification and copy the name and description of the idea to the new requirements specification.

Scenario 4

On a quality issue, your users must select a service type and then a service center. The selection of the service center is dependent upon the selection of the service type.

Solution A: Within Application Composer, create two fixed choice list fields with appropriate values for service type and service center. Designate the service type field as the parent of the service center field and map the values as wanted so that when a user selects a service type, only the applicable service centers are available for selection.

ORACLE

Solution B: Within Application Composer, create two fixed choice list fields with appropriate values for service type and service center. Use Groovy scripting to create a trigger that will automatically select the service center based on the selection of service type.

> **Note:** Refer to the Triggers section in Create Automation with Scripting.

Related Topics
- Triggers in Create Automation with Scripting

Configuration Nodes

When you open Application Composer, each object type that's available to you is displayed, along with a set of configuration nodes. Here's what you can do in each node:

- Fields: modify standard fields and create new fields.
- Pages: create and modify page layouts.
- Actions and Links: create internal actions or links to external applications with Groovy scripts.
- Server Scripts: create validation rules, triggers, and object functions with Groovy scripts.

  > **Note:** Supported functionality for change orders differs from other objects.

PLM Objects You Can Configure

PLM uses various objects in its processes, and you can configure or extend all of them to some extent. When you plan your configuration requirements, be aware of what's supported for each object type.

Object Types

You can configure the landing page, the Create dialog box, and the Details page of the following objects in Application Composer:

- Idea
- Requirements Specification and Requirements
- Proposal
- Quality Action
- Quality Issue
- Change Order
- Change Request
- Problem Report
- Corrective Action

ORACLE

- Concept

> **Note:** A Create dialog box must include all fields that are tagged as Required. This includes Required fields that you've renamed. If not, an error message appears as you set up the Create dialog box for a business object.

What You Can Configure For Each Object Type

This table summarizes the specific kinds of configuration that you can perform on each PLM object type.

Configuration Types and Objects

Configuration Type	Change Order	Quality Issue and Quality Action	Idea	Proposal	Concept	Concept Component	Requirement	Requirement Specification
Additional Attributes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Page Layouts	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Subtabs	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Show or Hide Tabs	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Configure Buttons or Actions	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Field Groups	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Configure URL Tabs	Yes	No	Yes	Yes	Yes	Yes	No	Yes
Configure first-level Objects	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Configure Child Objects	No	Yes	Yes	Yes	Yes	Yes	No	Yes

Note:

- In Oracle PLM Cloud applications, you're always in an active environment. When you plan to create user interface entities in Application Composer, you must first open a sandbox. After you do that, it's safe to open Application Composer and work without changing the production interface pages until you're completely prepared.

- All page layouts created using Application Composer must be duplicates of the standard template not a copy of another configured page layout.

- Concept structures don't support long text and formulas.

- In the **Pages** node for **Change Orders**, you can make page layouts for the Details page, but you can't modify the fields on the page layout. You can't configure the landing page or the Create dialog box.

- You can't create child objects for change orders. The Create Child Object button is no longer available for change order objects in Application Composer.

- You can define conditions in Application Composer to display the same page layout with different fields for different roles or users.

Access Configured Objects Through a REST Service

You can access the attributes of a configured object through a REST service by using the unique resource identifier.

Here's how:

1. Navigate to the Application Composer work area.

   Note: Ensure that you're in a sandbox.

2. Select **ERP and SCM Cloud** from the Application list.
3. Click the **Custom Objects** node. The Custom Objects page listing the configured objects appears.
4. Select the object for which you want to access the attributes.
5. Click the **Service** link in the REST Resource column.

   A new page containing the REST API host and endpoint opens. The unique resource identifier will be in this format: https://<HOST>/fscmRestApi/resources/latest/<CustomObjectName>_c.
6. Copy this unique resource identifier and paste it in the REST client to access the attributes of the configured objects.

Enable Auditing for Configured Objects

You can configure specific objects and attributes using audit policies, which can then be audited for configuration changes.

Once you enable the objects for audit, navigate to Audit Reports, and view changes such as when the object or its configured attributes were created, updated, or deleted.

1. Navigate to the **Setup and Maintenance > Product Management > Audit Trail** functional area.
2. Click the **Manage Audit Policies** task.
3. Click **Configure Business Object Attributes** to set objects which will be a part of the audit.
4. From the Product list, select **Product Hub** for configured objects.

Set Up Access

You must be assigned configured job roles along with the required privileges to extend applications using Application Composer.

Privileges names and codes required to work with Application Composer

Privilege Name	Privilege Code
Manage Extensible Object privilege	ZCX_MANAGE_EXTENSIBLE_OBJECT_PRIV
Administer Sandbox privilege	FND_ADMINISTER_SANDBOX_PRIV
Manage Product Idea	ACN_MANAGE_PRODUCT_IDEA_PRIV
Review Product Ideas	ACN_REVIEW_PRODUCT_IDEAS_PRIV
Manage Quality Action	ENQ_MANAGE_QUALITY_ACTION_PRIV
Manage Quality Issue	ENQ_MANAGE_QUALITY_ISSUE_PRIV
Review Quality Action	ENQ_REVIEW_QUALITY_ACTION_PRIV
Review Quality Issue	ENQ_REVIEW_QUALITY_ISSUE_PRIV
View Quality Action	ENQ_VIEW_QUALITY_ACTION_PRIV
View Quality Issue	ENQ_VIEW_QUALITY_ISSUE_PRIV
Manage Item Change Order	EGO_MANAGE_ITEM_CHANGE_ORDER_PRIV
View Item Change Order	EGO_VIEW_ITEM_CHANGE_ORDER_PRIV
Manage Product Proposal	ACD_MANAGE_PRODUCT_PROPOSAL_PRIV
Review Product Proposal	ACE_REVIEW_PRODUCT_PROPOSAL_PRIV

Role Required for Child Objects

ORACLE

To create and manage configured child objects: Custom Objects Administration role (ORA_CRM_EXTN_ROLE). This role is created dynamically when you create a new object using Application Composer. You then assign users to the role.

> **Note:** You can't create child objects for change orders. The Create Child Object button is no longer available for change order objects in Application Composer.

Related Topics
- Get Started

Use a Sandbox

You use sandboxes to make application changes and test them without impacting other users in the environment. Make changes to the application whenever you're in a sandbox rather than making direct changes in the mainline environment.

To access the sandboxes, navigate to **Configuration** > **Sandboxes** work area.

Completed application changes created within a test-only sandbox have to be published if they have to be available to other users in the application in the mainline metadata. While creating the sandbox, in the Publishable field, select **Yes** or **No**. If you set the Publishable option to **No**, you can use your sandbox for testing only, but can't publish it. When you're in a sandbox, you can open Application Composer and work without changing the production interface pages.

Also, a best practice is to create a new sandbox for each process change you intend to implement. As an example, say you want to create a set of new basic fields that fit your business process but have no dependency on the other fields, do that in one sandbox and publish it. If you want to create a validation for field or set of fields, do all of that in one sandbox and publish it. Therefore, you can group your modifications together according to the functionality that you want to provide and release or publish them incrementally one sandbox at a time.

Use the Unified Sandboxes user interface. This is the default feature that you get.

> **Note:**
> - Don't create or publish a sandbox for Application Composer or Page Composer configurations while other users or processes are deploying extensible flexfields and descriptive flexfields changes for Product Development objects.
> - Don't deploy flexfields while you've an open sandbox in which you're making Application Composer or Page Composer configurations that you want to publish. Don't deploy the flexfields until the publish operation is complete. Ensure that all the Application Composer or Page Composer configurations in sandbox are tested and published before deploying any extensible flexfields and descriptive flexfields configuration changes for Product Development objects.

ORACLE

Related Topics
- Overview of Sandboxes
- Create and Activate Sandboxes

Get Started

The following procedure is an extremely abbreviated sequence to open Application Composer and have a look around.

Access Application Composer

1. Within an active sandbox, navigate to **Tools** > **Application Composer** work area.
2. When you open Application Composer, the Application list offers **CRM Cloud** and **ERP and SCM Cloud** environments. Select **ERP and SCM Cloud**.
3. For each object node, whether Standard or Custom, expand it further to view and edit object details, such as object fields and pages.
4. For both Standard and Custom Objects, you can view and edit the following details:
   - Fields: Add new fields to an object.
   - Pages: Modify the pages on which an object appears.
   - Actions and links: Add actions or links to desktop pages.
   - Server scripts: Write application logic that controls the behavior of an object's records.

   > **Note:** For change order objects, you can't configure fields in Application Composer.

5. In this brief procedure, you can choose a business object, and we're selecting **Idea**. An idea is a standard object, populated with fields, or attributes. Navigate to **Objects** > **Standard Objects** > **Idea** > **Fields** > **Create** > **Select Field Type** to create fields.
6. After creating the field, go to **Pages**, assign the field to a duplicated **Landing** page layout, and click **Save**.
7. Open **Ideas** > **Manage Ideas** and click on an existing Idea to see the new attribute.

> **Note:** This procedure is applicable for all object types, except change order. For change order objects, you can't configure or create fields in Application Composer.

Related Topics
- PLM Objects You Can Configure
- Define Objects

Review Published Configuration Changes

After you finish configuration tasks in Application Composer, you can review your changes to make sure that everything is in order. In Metadata Manager, click **Generate Configuration Report** to view a summary of configurations made to layout details against a published sandbox.

ORACLE

You can view a summary of modifications across the following in the Application Composer Configuration Report:

- Standard and Custom Objects
- Global and Object Functions
- Standard Fields or Configured Fields
- Custom Relationships
- Validations
- Triggers
- Object Workflows
- Dynamic Layout

Related Topics
- Tools for Moving and Troubleshooting Configurations

Performance Best Practices for Extending SCM Applications

Let's look at the best practices for extending SCM applications using Application Composer.

Registration
When registering for WSDL in the Application Composer, ensure that the URL is whitelisted in the proxy to be accessed from the Oracle network. This registration is required for both REST and SOAP services.

Using Fields in Web Service Requests

When using fields in web service requests, there are some recommended best practices that you should follow to achieve optimal performance. Review this topic to understand the recommendations and constraints when including fields in a web service payload.

Which Fields Can Be Included in a Web Service Payload?
When creating a field in Application Composer, you can set the **Include in Service Payload** option. This option specifies whether or not the field value can be included in a web service request or response.

Long Text Field (CLOB)
Generally, long text fields are exposed only in detail pages. However, in web service calls, these fields are part of the response payload. Therefore, minimize usage where possible.

If you're going to use long text fields, then follow these recommendations:

- Limit the number of long text fields to two per object, when you use long text fields in web service calls.

- Avoid using a long text field for a text attribute, unless it needs to hold large values.

- Don't use long text fields in web service calls unless absolutely needed. These fields are high memory-consuming candidates.

Dynamic Choice List Fields

Including dynamic choice list (DCL) fields in a payload can impact the web service response time because it requires additional queries to fire. When you work with DCL fields, follow these recommendations:

- Limit the number of DCL fields included in a service payload to four fields.

- Use a good data filter to limit the number of records in the list of values (LOV).

- Only create a DCL field when absolutely needed to meet the business requirement.

- To access the object referenced by a DCL field through Groovy, use the respective DCL field's secondary Related Object Accessor field.
  See *Using the Related Object Accessor Field to Work with a Referenced Object*.

- Don't write complex data security predicates for the target object because that will impact the query execution time, causing performance degradation.

Fixed Choice List Fields

Including fixed choice list (FCL) fields in a payload can impact the web service response time because it requires additional queries to fire. When you work with FCL fields, follow these recommendations:

- Limit the number of FCL fields included in a service payload to four fields.

- Only include an FCL field in the service payload when absolutely necessary.

- Don't create lookup values if they won't be used.

Formula Fields

A formula field gets evaluated whenever it's referenced. Therefore, include only necessary formula fields in the service payload. Follow these recommendations when you work with formula fields:

- Limit the usage of formula fields in the payload.

- Don't include any persistent fields in the formula field calculation logic, unless absolutely needed.

Nonindexed Fields

Only a limited number of columns are indexed in the database table for a custom object. Therefore, you should use this property only for the most frequently searched fields. After the field is created, you can't change this property.

Note that using nonindexed fields for filtering in a web service call can affect performance. When you work with nonindexed fields, follow these recommendations:

- Index only the fields most commonly used as web service filters.

- Avoid indexing fields that will be rarely or minimally used in filtering.

Field Groups

Use field groups to organize your pages and make them look more readable. A field group lets you group fields into collapsible regions, each with its own header that you can modify.

Create field groups as part of either a creation page layout or a details page layout. For detailed instructions on how to create field groups, see *Add Field Groups to a Page Layout*.

Which Fields Can You Group?

The fields that you can select for a field group are attributes of the top-level object that you're creating the page layout for, such as the opportunity object.

Why Use Field Groups?

Field groups organize your page layouts. Here are some reasons why you should use them:

- Group related fields so they always appear together on a page.

  Perhaps you want a group of fields, such as Home Ownership and Purchase Date of Home, to always appear together. Group them.

- Group secondary fields in a region that your end users can optionally expand, if they need to.

  Maybe some fields on a page are useful, but not critical for your end users. Define the field group to be collapsed by default at runtime.

- Manage page layouts with fewer clicks.

  Once you combine fields into a field group, you can easily move that group up or down the page layout, with a single click.

  Multiple field groups always appear together at runtime within a larger field group "container". When designing a page layout, you can move a field group up or down, but only within this larger container.

  In most cases, field groups appear at runtime as regions right below the page's top summary region.

Field Group Validation

Application Composer validates the contents of field groups. You can't add the same field to different groups. This validation applies only across the field groups created for one page type (creation page or details page).

> **Tip:** Although you can't add the same field to multiple field groups, you can easily move a field between groups. This makes it easy to manage fields within groups, if you later change your mind about field placement.

Performance Considerations: Keep Field Groups Collapsed

Field groups are displayed inside a panel group that users can expand or collapse, but are initially in expanded mode by default. However, having 3 or 4 field groups on a page reduces page performance because, as part of drilling down on a record, multiple UI events must be fired to expand the field groups.

ORACLE

For optimal performance, always keep field groups on a page as collapsed by default. Users can then expand field groups as needed.

Performance Considerations: Plan Your Layouts in Advance

You can move fields between field groups, but when customizing page layouts, avoid frequent shuttling of fields between regions and field groups.

As a best practice, plan layouts before making field changes:

1. Decide which fields to include on each page, as well as their relative position (sequence) on the layout. Then add the fields in order.
2. Avoid repeatedly selecting and deselecting the same fields on the same pages.

Each time you select or deselect a field and save the layout, a corresponding customization instruction is saved and remains in metadata. Frequent selecting and deselecting of fields could lead to a large number of instructions that must be resolved at runtime.

Related Topics

- Overview of Dynamic Page Layouts
- Page Layouts for Standard Objects
- Use Field Values to Control a Page Display
- Control a Page Display Based on a User's Role
- Use Advanced Expressions to Control a Page Display

Creating Searchable Fields

When creating custom fields in Application Composer, you can indicate if you want them to be searchable using the Searchable check box. In this context, searchable means available for selection on a list page from the **Add Fields** choice list in Advanced Search mode. For optimal performance, consider the below details when using the Searchable check box.

You can select the Searchable check box for all field types except for long text and formula fields.

When you create a custom field, this check box is selected by default. However, if you keep the Searchable check box selected for all custom fields for an object, then this could potentially degrade list page performance for that object, especially if the object has many custom fields.

Here's how list page performance can be negatively impacted:

- When a user navigates to a list page and expands the Advanced Search, the Add button displays the list of searchable attributes as a dropdown list.

  If a very large number of custom fields have the Searchable property set to true, then the performance of rendering the Advanced Search window can be impacted because it needs to populate all searchable attributes in the list.

- If a user keeps the Advanced Search in its expanded mode on a list page, then when the user searches for or drills into a record and then clicks **Save and Close** to return to the list page, the searchable attributes list will be rendered again.

ORACLE

Therefore, you can reduce the performance impact on a list page by reducing the number of searchable fields. To do this effectively, determine which fields your users will most often use to perform searches for the object. Then, when you create a custom field, deselect the Searchable check box unless it's one of those necessary search fields. This will reduce the performance impact of needing to populate the searchable attributes list whenever Advanced Search is used.

Best Practices for Using Sandboxes

Several users might be working on your implementation in parallel in different sandboxes. So, follow these best practices to avoid conflicts while using sandboxes.

- To manage sandboxes easily and improve the system performance, reduce the number of open publishable sandboxes at any time.

- Reduce the number of active users using a publishable sandbox.

- Make sure a single user doesn't open multiple browsers and work in the same sandbox.

- It's recommended that multiple users don't work in the same sandbox at the same time.

- For best performance, make sure you don't use more than ten sandboxes in parallel.

- Don't make too many configurations in one sandbox. Instead, make small configurations in separate sandboxes and publish them.

- Sign out and sign back in every time you create, activate, publish, or leave a sandbox. Doing this clears any user-level caching to ensure that you're working with the latest configurations.

- Publish your sandboxes before patches are applied, or any release updates or upgrades happen. Otherwise, your unpublished sandboxes that existed before the patch was applied, will be marked as unpublishable to avoid invalid or inconsistent content being published.

- Don't publish your application changes directly in the target environment. To make sure changes aren't published in the target environment, set the **Control Publish Sandbox Action in Production Environment** profile option (FND_ALLOW_PUBLISH_SANDBOX) to **No** in the target environment.

- It's recommended that you don't use sandboxes in the production environment.

- Plan to publish your sandboxes and custom subject areas during periods of low system usage or low user activity.

Related Topics
- Best Practices for Using Page Composer in Sandboxes
- Delete Sandboxes
- Publish Sandboxes
- About Application Composer

ORACLE

Server Scripts

Application Composer supports Groovy as the scripting language that you use to enhance your applications. There are many different contexts in which you can use Groovy scripts.

This topic illustrates the use of validation rules, triggers, and object functions, which you can define using the Server Scripts node for any standard or custom object. For a more detailed explanation of Groovy scripting using Application Composer, see the Groovy Scripting Reference guide.

The server scripts that you can define for any standard or custom object include the following:

- Validation rules

  Write a script to validate either a field or an object.

- Triggers

  Write trigger scripts to automatically execute an action whenever a specific trigger event occurs.

- Object functions

  Write a function that can be reused in multiple contexts. For example, you can reuse an object function inside a trigger or validation rule.

> **Tip:** Server scripts are executed synchronously, which could potentially impact performance if the scripts are long-running or performance-intensive. Before implementing such logic, always consider whether the same logic can be executed in an asynchronous way. For example, you can instead use an object workflow with a Groovy Script action.

Validation Rules

Validation rules are constraints that you can define on either a field or on an object. Write an expression or a longer script to validate a field or object before it can be saved. Define validation rules using the Server Scripts node for any standard or custom object.

- If your requirement involves a single field, use field-level validation.

  A field-level validation rule is a constraint you can define on any standard or custom field. The rule is evaluated at runtime whenever the corresponding field's value is set and the user tabs out of the field. When the rule executes, the field's value hasn't been assigned yet and your rule acts as a gatekeeper until its successful

ORACLE

assignment. Covering the validation logic at the field level reduces overhead of Groovy executions when saving. Use this option whenever possible.

> **Note:** Dynamic choice lists don't support field-level validation rules.

For example, consider a custom TroubleTicket object with a Priority field. You can set a field-level validation rule to validate that the number entered is between 1 and 5.

The expression (or longer script) you write must return a Boolean value that indicates whether the value is valid.

- If the rule returns true, then the field assignment will succeed so long as all other field-level rules on the same field also return true.
- If the rule returns false, then this prevents the field assignment from occurring. Also, the invalid field is visually highlighted in the UI, and the configured error message is displayed to the end user. Because the assignment fails in this situation, the field retains its current value (possibly null, if the value was null before). However, the UI component in the web page allows the users to see and correct their invalid entry to try again.

See "Defining a Field-Level Validation Rule" in the Groovy Scripting Reference guide.

- When validation is needed across multiple related fields in an object, use object-level validation rules. This is a constraint you can define on any standard or custom object.

Use object-level rules to enforce conditions that depend on two or more fields in the object. This ensures that regardless of the order in which the user assigns the values, the rule will be consistently enforced. The rule is evaluated whenever the framework attempts to validate the object. This validation can occur, for example, when submitting changes in a Web form, when navigating from one row to another, as well as when changes to an object are saved. (Rules aren't evaluated if the user saves a record without making changes.)

For example, consider a TroubleTicket object with Priority and AssignedTo fields, where the latter is a dynamic choice list field referencing Contact objects whose Type field is a Staff Member. You can set an object-level validation rule to validate that a trouble ticket of priority 1 or 2 can't be saved without being assigned to a staff member.

The expression (or longer script) you write must return a Boolean value that indicates whether the object is valid:

- If the rule returns true, then the object validation will succeed so long as all other object-level rules on the same object return true.
- If the rule returns false, then this result prevents the object from being saved, and the configured error message is displayed to the end user.

See "Defining an Object-Level Validation Rule" in the Groovy Scripting Reference guide.

> **Note:** When defining a validation rule, don't implement business logic other than:
> - Validations inside an object
> - Field validation rule Groovy script

Triggers

Triggers are scripts that you can write to complement the default processing logic for a standard or custom object. When a specific event occurs, triggers automatically execute an action that you specify in the trigger definition. You can

ORACLE

define triggers both at the object level and at the field level using the Server Scripts node for any standard or custom object. Define object triggers to extend standard processing logic, such as record creation, updates, and deletions.

When you define a trigger, you select the specific event that causes your script to automatically run. This specific event is also referred to as a trigger. Oracle supplies a set number of these trigger events that you can pick from when defining your trigger "scripts."

Choose the correct triggering point when defining your trigger:

- Field-level triggers are scripts that you write to execute an action in response to a change in another field's value. When you define a trigger at the field level, you select the **After Field Changed** trigger and the field that this trigger is watching. You then define the action that you want to happen when the field's value changes.

  The **After Field Changed** trigger calculates other derived field values when the value of the field that you specify changes. Don't use a field-level validation rule to achieve this purpose because while your field-level validation rule may succeed, other field-level validation rules may fail and stop the field's value from actually being changed. Generally, because you want your field-change derivation logic to run only when the field's value changes, the After Field Changed trigger guarantees that you get this behavior.

  See "Defining a Field-Level Trigger to React to Value Changes" in the Groovy Scripting Reference guide.

ORACLE

- Similarly, object-level triggers are scripts that execute an action when a specific event occurs. In the case of object-level triggers, you have many more trigger "events" to pick from, such as:

  - **After Create**

    Fires when a new object record is created. Commonly used to set default values for fields.

  - **Before Invalidate**

    Fires on the parent object when one of its child object records is created, updated, or deleted. For building in relationship logic.

  - **Before Remove**

    Fires when an attempt is made to delete an object record. Can be used to create conditions that prevent deletes.

  - **Before Insert in Database**

    Fires before a new object is inserted into the database. Can be used to ensure a dependent record exists or check for duplicates.

  - **Before Update in Database**

    Fires before an existing object is modified in the database. Could be used to check dependent record values.

  - **Before Delete in Database**

    Fires before an existing object is deleted from the database. Could be used to check dependent record values.

  - **Before Rollback in Database**

  - **After Changes Posted to Database**

    Fires after all changes have been posted to the database, but before they're permanently committed. Could be used to make additional changes that will be saved as part of the current transaction.

  For example, consider a Contact object with an OpenTroubleTickets field that needs to be updated any time a trouble ticket is created. You can create a trigger on the TroubleTicket object using the **After Changes Posted to the Database** trigger event. When an event occurs, your trigger can automatically update the OpenTroubleTickets field with a new count.

  For a complete list of the trigger "events" that you can pick from, see "Defining an Object-Level Trigger to Complement Default Processing" in the Groovy Scripting Reference guide.

For optimal performance, follow these guidelines when using triggers:

- Don't exceed 10 triggers per object.

- Combine logically-related actions inside a single trigger. A single trigger produces better performance than multiple triggers.

- When defining a trigger, choose the correct triggering point.

- Avoid using validation logic inside triggers. Instead, use validation rules for any validation logic.

- Before using the newView() API, check to see if related objects or related collection accessors already exist.

  If a relationship already exists, then don't use newView() to query an object. This avoids the firing of additional queries.

- When querying objects programmatically, select an efficient view criteria so that the underlying query is limited.

Object Functions and Global Functions

You can write reusable code as either an object function or global function. Use a function if you anticipate calling the same code from multiple different contexts. Object functions can be called by any script in the same object, or even triggered by a button in the user interface. Global functions can be called from scripts in any object or from other global functions.

- Object functions are useful for code that encapsulates business logic specific to a given object. You can call object functions by name from any other script related to the same object. In addition, you can call them using a button or link in the user interface.

  The supported return types and optional parameter types are the same as for global functions. For a list of the most common types for function return values and parameters, see "Defining Utility Code in a Global Function" in the Groovy Scripting Reference guide.

  See also "Defining Reusable Behavior with an Object Function" in the Groovy Scripting Reference guide.

- Global functions are useful for code that multiple objects want to share. Write user-defined functions using Groovy scripts, which can be referenced in all Groovy script editors throughout Application Composer. For example, you could create two global functions to define standard helper routines to log the start of a block of Groovy script and to log a diagnostic message.

  To call a global function, preface the function name with the adf.util. prefix. When defining a function, you specify a return value and can optionally specify one or more typed parameters that the caller will be required to pass when invoked.

  For a list of the most common types for function return values and parameters, see "Defining Utility Code in a Global Function" in the Groovy Scripting Reference guide.

- For optimal performance, the maximum number of records that can be fetched is restricted to 500 records. If the data size chosen causes the job to fail with ExprTimeoutException, reduce the data size using setMaxFetchSize. Or, use selective filtering to filter the number of records fetched though the view object to less than 500 records.

Privileged Functions

When you define either an object function or global function, the function might run on an object where the runtime user has no privileges to create or update records. Allow users without access to an object's data to run a function with full access, by doing two things:

1. While defining the function, check the Privileged check box to indicate that the function is privileged.
2. Confirm that the Privileged Script Administration role has the right level of access so that the object function can execute successfully. Access to objects isn't given automatically. Instead, you must grant access using the Application Composer security UI.

At runtime, when a user invokes a privileged function from the UI, a temporary login session is activated with the privileged role, Privileged Script Administration (ORA_CRM_EXTN_PRIVILEGE_SCRIPT_ROLE). This privileged role has access to the object in your function, so no permission issues exist. The temporary login session lasts only for the duration of the single function call or anything that function calls internally.

For example, a sales representative has access to opportunity records only, not account records. When a sales representative edits an opportunity, there is a button that updates a related account using a privileged Groovy

ORACLE

script. Even though the sales representative doesn't have update privileges for the account object, when the sales representative clicks the button, the privileged Groovy script executes by switching to the privileged role context to complete the update to the account record.

To make this happen, use the Application Composer security UI to grant account access to the privileged role, Privileged Script Administration.

Related Topics
- Global Functions
- How Object Workflows and Groovy Scripts Work Together

Overview of Groovy Scripts

Groovy is a standard, dynamic scripting language for the Java platform. You write Groovy scripts using Application Composer's expression builder, which appears in many places as you modify existing objects or create new custom ones.

Read this chapter to learn about how and where you can use Groovy scripting in Application Composer.

> **Note:** To fully understand all the scripting features available to you in Application Composer, you should also review the Groovy Scripting Reference guide.

In this chapter, you will learn about:

- Where you can use Groovy in your application, along with examples of one or more lines of Groovy code

- How to access view objects using the newView() function, for programmatic access to object data

- How to create global functions, which is code that multiple objects can share

- How to call Web services from your Groovy scripts. You might call a Web service for access to internal or external data, or, for example, to perform a calculation on your data.

- What kind of scripts will you write?

Write Groovy Scripts

You write Groovy scripts using Application Composer's expression builder, which appears in many places throughout Application Composer as you modify existing objects or create new custom ones.

- You will write shorter scripts to provide an expression to calculate a custom formula field's value or to calculate a custom field's default value, for example.

- You may write somewhat longer scripts to define a field-level validation rule or an object-level validation rule, for example.

Additional examples of where you write Groovy scripts in Application Composer are described in "Groovy Scripting: Explained."

To learn more about how to best use the features available in the expression builder when writing scripts, see "Groovy Tips and Techniques" in the Groovy Scripting Reference guide.

ORACLE

Performance Considerations

For optimal performance, always write your Groovy scripts to query and fetch the fewest possible rows from the database. Minimizing the number of queried rows is one of the most important performance best practices you can follow.

Many of the most significant performance impacts you're likely to encounter will be related to code that queries and fetches a large number of rows. This has several implications for application performance and functionality:

- Response time

  Querying and fetching rows are two of the most expensive operations your application will typically perform and can noticeably impact performance. Large result sets magnify the impact of sub-optimal code executed while iterating over a result set.

- Scalability

  Long-running operations tie up limited shared resources such as database connections and shared memory. This might increase response time of other requests because they have to wait for resources to become available.

- Functionality

  If your application attempts to fetch too many rows, then the application will display a warning to the user and will not fetch any more rows. This limits the user's ability to view data in the UI and causes processing errors in scripts that iterate over rows.

To limit the potential performance impacts described above, most view objects have a 500-row Fetch Limit. Design your customizations with the Fetch Limit in mind as an absolute limit, but always strive to limit your queries to the least number of rows possible to satisfy your business requirements.

For example, use view criteria and bind parameters to tailor a view's default query and limit fetch size for view objects. See *Finding Objects Using a View Criteria*.

Related Topics
- Groovy Scripting
- View Objects in Scripts
- Global Functions
- How to Call SOAP Web Services

Performance Best Practices for Using Loops, Methods and Strings

You must understand and consider the performance impacts of your design decisions when you use Groovy scripting to configure the objects.

Let's look into the following concepts:
- Loops
- Methods

ORACLE

- Strings

Loops

The ability of looping constructs (like while and for) to perform a set of operations repeatedly with minimal code is a very powerful feature. Used incorrectly, loops can trigger significant performance issues.

Looping Over View Result Sets

Take care when looping over the rows in a view's result set. When it's necessary to loop over a view's result set, use view criteria to limit the scope of the view's query as much as possible. It might be tempting to fetch all the rows and then decide which rows you actually need in your script. Querying and fetching rows you don't need can have a significant performance impact on your application.

Execute Loops Conditionally Where Appropriate

If there are conditions that should logically be satisfied before a loop is executed, enforce those prerequisites by enclosing the entire loop in one or more "if" clauses. If you don't, your application will do unnecessary work that slows it down.

Limit Loops to the Fewest Possible Iterations

There's a cost associated with each loop iteration; you'll never go wrong by assuming that each iteration is expensive. Every time you implement a loop, stop and ask yourself these questions:

- If you're iterating over a view's result set, have you limited the result set to smallest possible number of rows that satisfy your business requirement?

- Is it possible to reduce the number of iterations? For example, if you're searching for a particular value, exit the loop when the value is found instead of completing the remaining iterations. But don't use this technique as an excuse to avoid limiting the scope of your query using view criteria.

Methods

Assume Methods Are Expensive to Execute

Any time your code executes a method like newView, getAttribute, or any other method that you didn't implement, assume the worst even if the cost of executing the method appears to be negligible:

- A method's performance characteristics could change due to various factors.

- A method's execution cost is magnified by repeated execution inside a loop, in an attribute-level script, and similar contexts.

- A method might execute rapidly but might consume some limited resource that impacts application scalability such as shared memory or database connections.

Don't be lulled into thinking that any method call is free (or nearly so). For example, there's always a cost associated with calling getAttribute to get an attribute value from a row – the cost might actually be quite high depending on the attribute's implementation. Any time you implement code that calls any method, you need to think about context:

- Inside a loop, would the code work just as well if you executed the method outside of the loop? For example, if the method can return a different value every time the loop iterates, the method must be called inside the loop. If the method will always return the same value, it should be called before the loop is entered.

ORACLE

- Whether inside or outside of a loop, should the method be called every time the script encounters the method? Or is the result of executing the method only useful if certain conditions are met? If so, the method should be placed inside an "if" clause that prevents the method's execution unless the pre-requisite conditions are met.

Consider the following two code examples. The first version always updates the LineEditedCount attribute. The second version only performs the update if the new value is different from the original value. Setting an attribute value might seem like a trivial operation. However, there's always a cost associated with setting an attribute. Some attributes are more costly to set than others – for example an attribute that triggers complex business logic when set. In addition, setting an attribute's value marks the row dirty, triggers row validation and adds the row to the pending transaction for commit – even if the new value is the same as the original value.

Unconditional setAttribute Call

```
setAttribute('Priority',newPriority)
```

Conditional setAttribute Call

// Only assign the value if it's not already what we want it to be

```
if (Priority != newPriority) {
setAttribute('Priority', newPriority)
}
```

Check Current Attribute Value before Setting the Attribute

Don't call a row's setAttribute method if the new value is the same as the current value. Failing to check if the setAttribute call is necessary can lead to the following issues:

- Performance impact – calling setAttribute marks the row as dirty - even if setAttribute doesn't change the attribute's value. That's if the current value of the attribute is five and you call setAttribute(<attribute name>, five), the row is still marked dirty.
  - Dirty rows are added to the transaction and are subject to validation.
  - Dirty rows are posted to the database and committed when saving changes.

Validating and posting rows unnecessarily can negatively impact performance – especially if you fetch and update more rows than are required for your use case.

The following pseudo code example verifies that the proposed new value is different than the existing value before calling the setAttribute method:

```
if (firstName_c != value) { setAttribute('firstName_c',value)}
```

Cache Attribute Values

There's always a cost for retrieving an attribute value from an object. The cost varies by attribute depending on specific features implemented for that attribute. If you need to use the same attribute value more than once, call getAttribute once and cache the result in a variable. Don't call getAttribute for the same value multiple times.

Avoid Performing Expensive Operations in Validation Scripts

You should only use Validation scripts for validation; for example, you should not set attribute values in a validation script - Validation scripts might be called multiple times before an Object is finally committed. When performing any potentially expensive operation in a validation script, consider whether the operation needs to be included in a validation script or can be moved to a less often evaluated script.

Avoid Explicit Calls to Validate the Groovy Script

ORACLE

Validating a row always incurs a cost which varies by object based on the validations implemented for that object. Explicitly calling validate should not be necessary. If you call validate explicitly, you're most likely adding more validation cycles beyond what would otherwise be required.

Variable Initialization

A variable initialization can be as simple as def count = 1.

You can initialize variables with simple scalar variables like this anywhere that satisfies the business requirements without worrying about performance implications. This topic covers variable initializations that require more expensive operations such as a method that performs complex operations and / or consumes shared resources.

The following examples illustrate different ways to initialize a variable including best practices that apply to all variable initializations even when there isn't a specific performance concern. The same best practices that help maximize your application's performance also serve to clarify how and when each variable is used; this tends to result in more reliable code that's easier to maintain.

- Initializes changeVO on every iteration of loop whether or not it's actually used; this is inefficient and will trigger errors if the code creates too many view instances.

- Initializes the changeVO before entering the loop. Because the changeVO query doesn't change from one iteration to the next, this is much more efficient. This does mean that changeVO is created even if the condition.

- Initializes changeVO by calling newView the first time that the specified condition is met. This is the most efficient version of this code. This code creates one changeVO instance at most and none if the condition requiring its use isn'tmet.

Strings

Strings are immutable; they can't be changed after they're created. When you concatenate two strings, a new String object is created to hold the combined Strings – creating new objects takes time and results in more work for the garbage collector. In the following example, notice the use of single quotes to define a String literal.

//Creates new String object to hold String literal plus an expression result

```
def receivedMessages = 20
def readMessages = 10
def confirmation = 'You have ' +
(receivedMessages – readMessges) + ' unread message(s)'
```

Groovy offers a more efficient choice to string concatenation. This option is substituting string using GStrings. A GString is created by using double quotes instead of single quotes.

// Double quoted string with substitution expressions creates a GString

```
def confirmation = "You have ${receivedMessages - readMessages} unread message(s)"
```

```
println(confirmation)
```
//final String created here

GStrings offer two potential performance benefits as compared to standard String concatenation.

- A GString stores the component parts of the String and defers String creation until the String is actually needed.

- A GString doesn't evaluate component expressions - ${…} until the String is created. Expressions aren't limited to simple arithmetic as in the example above; you can include blocks of Groovy code (you must ensure that the final statement returns the value that you want to include in the String).

ORACLE

Looking back at the first example of String concatenation, the expression receivedMessages – readMessages is evaluated and the final String is created when the variable confirmation is set. In the second example using a GString, both expression evaluation and final String creation are deferred until the println statement is executed. If you're defining a literal String with no substitution expressions, it's slightly more efficient to use single-quotes.

Use GStrings and single quotes as appropriate when writing new code – it's very difficult to identify up front how much benefit (if any) a particular implementation will derive from string optimization. But there's no added cost for implementing optimized Strings so the best practice is to hedge your bets and implement optimized Strings from the start.

Move cautiously and temper your expectations if considering a retrofit of existing code. Every code change carries some level of risk. String optimization alone might not produce significant (or even measurable) performance improvement in many cases. If your configurations suffer from poor performance, you should focus first and foremost on the other best practices discussed in this document.

Performance Best Practices for Web Service Implementations

Here are some best practices for optimal performance when working with web services.

Groovy Script

Use the Groovy script to call an external web service to sync data on creating a record in the application.

A "before update" trigger to call an external web service to sync data is a time-consuming process. The entire commit must wait for the web service response, impacting the total runtime performance.

Instead of doing this with triggers, you can write an Object Workflow that calls a groovy on the update of a record and syncs to the external system. This asynchronous method of syncing data with the external system will reduce the UI overhead.

REST/SOAP

You can set a field value to NULL using web services. This example will set a Text_c field value to null, even if it had a previously entered value:`<ns2:Text_c> </ns2:Text_c>`

REST – Fetching Fields

Blind fetch fields in REST web services will bring all fields with them. For better performance, you should limit the number of fields that are included in the response by adding filters. Here are some examples of filters you can use.

Examples of Filters

Use Case	Solution
Query to return only specific fields	`?fields=Attribute1,Attribute2` Format for fields in child resource:

Use Case	Solution
	`?fields=Accessor1:Attribute1,Attribute2`
Query to return only data and no links	`?onlyData`
Sort the result returned by Query	`?orderBy=field1:asc,field2:desc`
Filtering the records on a generic expression	`?q=expression1`
Filter records within a specific date range	To extract records filtering on the fields of type Date the parameter q= of the url should be: `<Nameofthefield> > YYYY-MM-DDTHH:MI:SS.sss-HH:MM`
Query to fetch records from a specified offset	`?offset=<number>`
Combining multiple conditions with AND effect	`?q=expression1;q=expression2?q= expression1 and expression2`
Combining multiple conditions with OR effect	`?q= expression1 OR expression2` Example: ?q= Type = ECO OR ChangeAnalyst = John Smith
Filtering on check box fields	`?q=checkbox_c=Y?q=checkbox_c=Y`
Skip AND as a Keyword when present in a value	`?q=RecordName%20LIKE%20%27Fin%20And%20Mgmt%27`

REST – Fetching Number of Records

You can use this limit parameter to limit the number of fields that are fetched for a REST call.?limit=<number>If you don't specify a number in this parameter, the limit is set to 25. But you can manually specify a limit value, so it will fetch that many records. The recommended limit value is 499 or lower, to maintain optimal performance. Don't use a higher limit value for fetching records, even if you've a few records, as memory is allocated based on the limit parameter value you set.

Performance Baseline Measurements
Measure the time it takes to render:

- Object Edit page
- Save and Close flow
- Save and Continue flow

Analysis and Resolution of Performance Issues

This section discusses tools and procedures for resolving performance issues after implementation. It also contains best practices when migrating from one environment to another.

ORACLE

If you identify a performance issue related to your configurations, the following steps offer some guidance on how to isolate and resolve the issue.

Evaluate User Feedback

- - o What were you doing when the issue occurred and was the action related to any aspect of your configurations? The actions will point to specific objects, views and Groovy scripts.
    - o Collect as much specific detail as you can about the issue. Exact navigation paths, exact sequence of operations, items selected, buttons clicked, menus selected, time of day, number of rows affected.

Reproduce the Issue

- Enable logging for the application and try to reproduce the issue yourself – see Logging for more information.
- If you can't reproduce the issue:
    - o Evaluate the possibility of an external issue triggering the issue – a bad Wi-Fi connection for example.
    - o Try a series of tests; for each test, vary a single element of the reported steps or data to see if you can reproduce the issue.
    - o If you can't reproduce the issue, track the incident details and these details - combined with future reports - might help you spot a pattern that will allow you to isolate the issue.
- Is the issue reproducible only some time? Reproducing the issue even once with logging enabled should provide valuable insight. If you do have log entries recorded while reproducing the issue, see Analyze the Application Script Log for suggested next steps. If you're stuck, keep in mind that an inability to consistently reproduce a bug often equates to lack of enough detail regarding the conditions that trigger the bug; always gather as much detail as possible.
- issue is reproducible and not attributable to some external condition or event (network, server load) - see Analyze the Application Script Log for suggested next steps.

Analyze the Application Script Log

The following set of points outline strategies for isolating the source of the problem in some common use cases with the Application Script Log enabled.

- If there are many scripts involved, you can narrow down the list of candidates for elapsed time logging by answering the following questions.
    - o Is the performance problem associated with page loading of a particular page?
        - Have you implemented any Groovy scripts recently that are executed during page load (formula fields, conditionally updatable attribute, conditionally required attribute or object/global scripts executed in these contexts)?
        - Have you added new custom attributes related to the use case? Do these attributes implement scripts as described above?
        - Have you added any new Long Text or other fields that are potentially expensive to fetch and display? See Custom Attributes for best practices related to optimizing custom attribute usage.
- Is the performance problem associated with saving changes?
    - o Have you recently implemented any Groovy scripts that are executed during save operations (Object validation or trigger scripts including - but not limited to - Before Insert in Database, After Insert in Database, Before Update in Database, After Update in Database, Before Delete in Database, After Delete in Database, After Changes Posted to Database, Before Commit in Database, After Commit in Database, Before Rollback in Database, After Rollback in Database or object/global scripts executed in these contexts)?

ORACLE

- Have you added new custom attributes to an object that participates in the use case? If the affected object also suffers from degraded page load performance, review the guidance for page load issues first. If page load performance isn'tdegraded, review the scripts associated with these attributes.
- Does the use case that triggered the issue include a much larger number of rows than a typical use case? Large row counts magnify the impact of poorly performing code.

- Is the performance problem associated with some other action on a page? The diagnostic steps are similar for any use case that suffers performance degradation relative to the baseline.
    - Identify any recently implemented Groovy scripts that are executed in the course of the use case
    - Identify any recently implemented custom attributes along with associated scripts. If the object also suffers from degraded page load or save performance, review the guidance those issues first. If page load and save performance aren't degraded, review the scripts associated with these attributes.
    - Consider the number of database rows affected by the action.

Logging

The Application Script Log (select Runtime Messages in Common Setup in Application Composer) is an important tool for diagnosing various problems including performance related issues. Application Composer places specific limits on the scope of logging to minimize the impact of logging on overall application performance.

- Application - Logging is enabled and disabled at the application level. If a particular logging use case spans multiple applications, you need to enable logging for each application.

- User – Logged in user enables and disables logging for their user id only; a user must have access to Application Composer to enable logging.

- The impact of enabling and disabling logging is limited to the println command:
    - When logging is disabled, the println command does nothing.
    - When logging is enabled, the println command sends output to the Application Script Log.

Any other code that supports logging (for example, code used to generate log content) is executed for all users, all the time; by default, enabling or disabling logging has no impact on any code other than println. This is mainly a concern in cases like the following:

- Your logging support code is performing relatively expensive operations that are performed only to support logging.

- Your logging support code is implemented in a often executed script – for example an attribute formula script. The frequency of execution could elevate otherwise innocuous code to the level of a performance issue.

To limit the potential impact of logging-related code other than println statements, see Guidelines for Implementing Logging Support Code and Example Logging Functions.

To view the Application Script Log, click Runtime Messages in the Common Setup node in Application Composer. In the following screen shots, logging has been enabled for the Sales application and the John Dove user.

Runtime messages in ERP and SCM Cloud

ORACLE
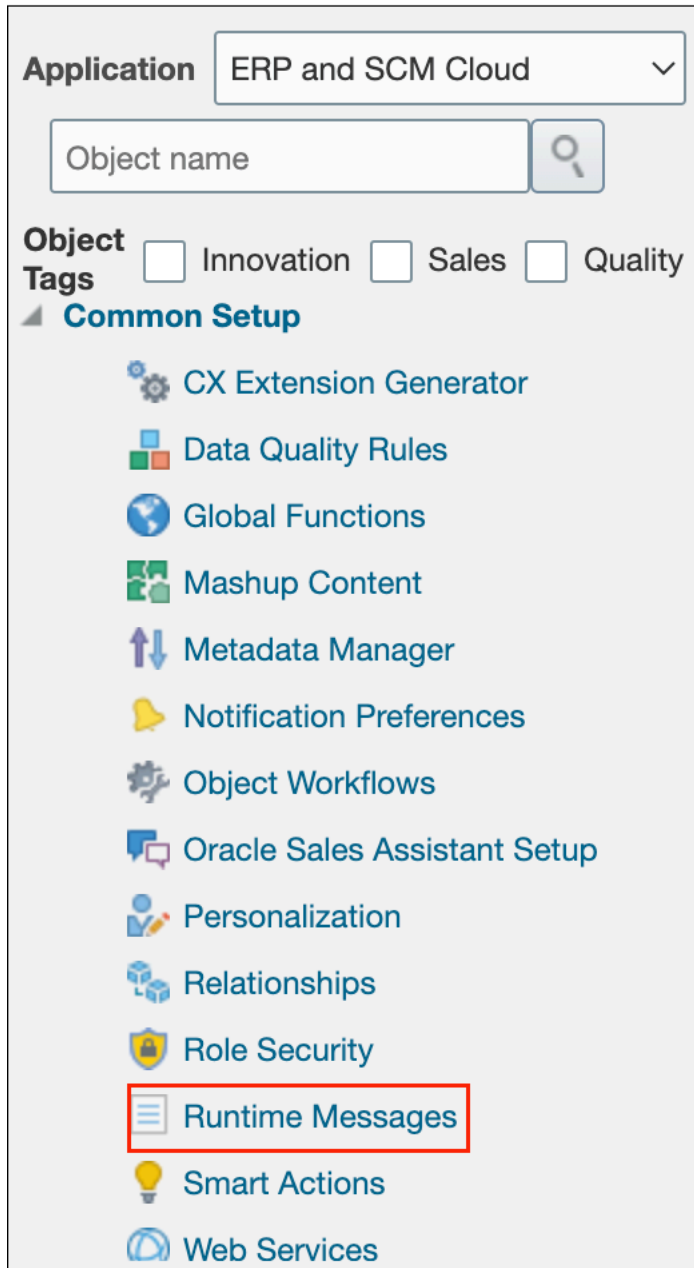
Logging is enabled and disabled for each individual application. In this example, the user has selected the Sales application. Next, select Runtime Messages in Common Setup. Finally the user checks the Enable Application Script Logging. Note that logging is enabled for a current logged in user only – the John.Dove user in this case.

Guidelines for Implementing Logging Support Code

In this context, logging support code refers to any code (other than simple println statements) that's used only for logging. Output from println statements is disabled when you disable Application Script Logging. Any other code used only for logging will execute all the time for all users.

- Avoid implementing logging support code that's expensive in terms of time and/or resources.

ORACLE

- If you plan to implement any logging code that might have performance impact (for example fetching rows that are used only to generate log output) and or logging in scripts that are often executed (for example an attribute formula script), ensure that such code is disabled when not in use.

- Avoid replicating common logging code directly in your individual scripts. Where feasible, concentrate common logging-related work in global logging functions that you call from individual functions. This let's you to enable and disable logging support code in a single place.

Considerations When Using Page Composer and Application Composer Together

Application Composer is the recommended tool for configuring and extending PLM applications. Page Composer can't be used with all PLM objects. If you attempt to add Page Composer-enabled configurations to existing Application Composer-enabled pages, such configurations can break during an update.

Let's look at PLM objects that you can configure using Page Composer.

Page Composer Support for PLM Objects

Supported	Not Supported
You can configure the following PLM objects with Page Composer: • Change Orders • Change Requests • Problem Reports • Corrective Actions	You can't configure the following objects with Page Composer: • Manufacturers • Quality Actions • Quality Issues • Concepts • Proposals • Requirement Specifications • Ideas

You can use Page Composer to do some configurations for change orders, change requests, problem reports, and corrective actions. Use Page Composer to add components only within the tab regions of the predefined tabs like **General Information**, **Affected Items**, and **Attachments**.

However, you can't make changes to the tabs configured by Application Composer or the tab bar component itself. You can't add, rename, hide, delete, or modify the tabs. This is applicable for actions or buttons on the page.

Upgrade - Safe Page Composer Configurations

Page Composer Configurations	Description and Examples
Rename or hide standard fields in the object pages.	Rename standard fields such as: • 'Description' to 'Description of Change' • 'Requested By' to 'Change Originator'

ORACLE

Page Composer Configurations	Description and Examples
	• 'Assigned To' to 'Change Analyst' • Hide the Effective Date field on the Affected Object tab of a Change Order without Revision Control. **Note:** You can use the Show Component property on the Component Properties dialog box to show or hide page components.
Make the fields for standard attributes mandatory.	Make these fields mandatory: • Change Originator • Change Analyst • Reason Code
Reorder the standard and previously configured fields.	Select the page and change the order in which the fields appear in the Component Properties: panelFormLayout page. Add components to the object. Some examples of components: • Box: Use this component to place content on a page. • Image: Use this component to add a picture, a logo, or a linked image to a page. • Hyperlink: Use this component to add a link to a page or a website. • Web Page: Use this component to provide URLs of other web pages within the context of a WebCenter application page. • HTML Markup: Use this simple editor to enter raw text and HTML tags, including JavaScript embedded in HTML <script> tags. • Text: Use this component to add UI text or any other kind of informative content to a page. • Button: Use buttons on Page Composer only if you don't use Application Composer to add or modify configured buttons, actions, or tabs.
Use Expression Language to show or hide fields and their labels.	Add a condition using an expression to control the visibility of a component. If the condition is met, then the component is displayed, otherwise, the component isn't displayed.
Label and font changes, or any CSS changes.	Configure your page layout to define the number, placement, and orientation of content regions. You can set the layout style while creating a page or you can change the layout style even after adding content to the page. **Note:** You can't change the page layout for all pages.
Add Action, Links, or Buttons.	Add an expression to the **Change Status** button to control the display of change orders and change requests.
Validate the user entry in a field.	Display an error message if a value isn't entered for a specific field. You can do this, by selecting the **Required** check box on the Component Properties page, and entering the message that you want to display.
Configure the Change Overview page.	Hide the **Approve** and **Reject** options in the My Worklist or My Changes on the Overview tab in the Product Development work area.

Additional Configurations with Application Composer

Existing Page Composer Configuration	Additional Application Composer Configuration	Supported on Upgrade?
Button created on the General Information page to launch a PaaS application.	Action to launch a configured extension on the same page.	No
Button created on the General Information page to launch a PaaS application.	Groovy scripting that validates the presence of value in the **Priority** field of the General Information page after changes are saved. Groovy script that populates Tasks on the Task tab in the Edit Change Order page.	Yes
Attributes made mandatory using Page Composer on General Information page.	Configured action created using Application Composer on General Information page.	Yes
Add hyperlinks to internal sites like confluence.	Configured button to perform **Save As** of certain aspects like General Information or Affected Object of the current object.	Yes

Note: Apart from the combinatorial scenarios stated in the tables, you must use only one of these tools to configure buttons, actions, or tabs. Otherwise, your changes may not be retained or properly upgraded during release or patch updates.If you've used Page Composer to configure buttons, actions, or tabs, avoid using Application Composer to make further changes to these, including adding expressions. Likewise, if you've modified buttons, actions, and tabs using Application Composer, don't use Page Composer to make additional configurations.

ORACLE

2 Extend Change Management

Change Orders

Change Orders let you process changes to user-defined item attributes, structures, packs, associations, and item revisions.

Product data stewards and product managers can manage product change orders. They can create change orders within predefined change types, author product changes, view product changes, submit changes for review and approval, track change statuses, and implement changes on a scheduled date.

You can configure certain aspects of change orders within Application Composer in accordance with your business requirement.

> **Note:** In Application Composer, you can configure problem reports and corrective actions like you configure change orders.

What You Can Configure for Change Orders

Here are the configurations that are applicable for change orders.

- Subtabs that include context links and relationships
- Show or Hide Tabs
- Buttons or Actions
- URL Tabs
- First-level Objects

> **Note:** The Create Child Object button is no longer available for the change order object. However, you can create a one-to-many relationship for your configured object from the Relationships node in Application Composer. Child objects that you've created prior to update 23D are available, and you can edit the attributes of such objects, although usage isn't recommended.

Configure Change Orders

You can configure the change order objects by adding new fields in the Setup and Maintenance work area.

Define new subtabs within the Application Composer to create object relationships on page layouts. Write application logic, such as triggers, validation rules, and object functions that you can use in change order objects.

Let's see how we can configure change orders.

- Set up fields in the Setup and Maintenance work area

ORACLE

- Create new page layouts

- Create new tabs like related objects, context links, and mashup content

- Create actions and links

- Create automations with scripting

- Work with global functions

Related Topics
- Watch Video: Global Functions
- Assign Conditions to Page Layouts
- Use Field Values to Control a Page Display
- Use Advanced Expressions to Control a Page Display
- Control a Page Display Based on a User's Role
- Define Objects
- Related Object Subtabs

Set Up Fields for Change Orders

You can configure new fields for change orders in the Setup and Maintenance work area.

Set Up Fields

You can configure change orders (and other objects that are derived from a change type, such as change requests, problem reports, and corrective actions) to a certain extent using Application Composer. For example, you can define new subtabs to create object relationships on page layouts, or write application logic such as triggers, validation rules and object functions to be used in the change objects. But if you want to configure change attributes or fields, you must set up extensible flexfields from the Setup and Maintenance work area.

Here are the basic configuration steps:

1. To create change order attributes, in the Setup and Maintenance work area, do the following:
   - Offering: Product Management
   - Functional Area: Change Orders
   - Task: Manage Change Order and New Item Request Header Descriptive Flexfields

   Note:
   - Deploy descriptive flexfields before you create a sandbox for working in Application Composer.
   - Attributes can be either Global Segments - applicable to all change types, or Context Segments - applicable to a specific change type.

2. Assign attributes to a change type.
   - To assign attributes to a specific change type, enter the internal name of the change type as the context code of the segment/attribute context group and then add new attributes to the group.

     Attributes can be of the following type:
     - Character (Text)

ORACLE

- Number

- Date

- Date Time

- List of Values (Character or Number)

Set these attribute segments as required. You can also set a date range.

Note:

- You must deploy the attribute segments and groups if you want to make them available for use.

- Attributes are assigned to the change order General Information page when they're deployed.

- Use the Sequence property in the setup task to define the display order. All the global attributes are displayed first, followed by the display of contextual attributes.

- In the setup task, you can disable an attribute so that it will no longer display.

This image displays a change object that has admin-created fields.
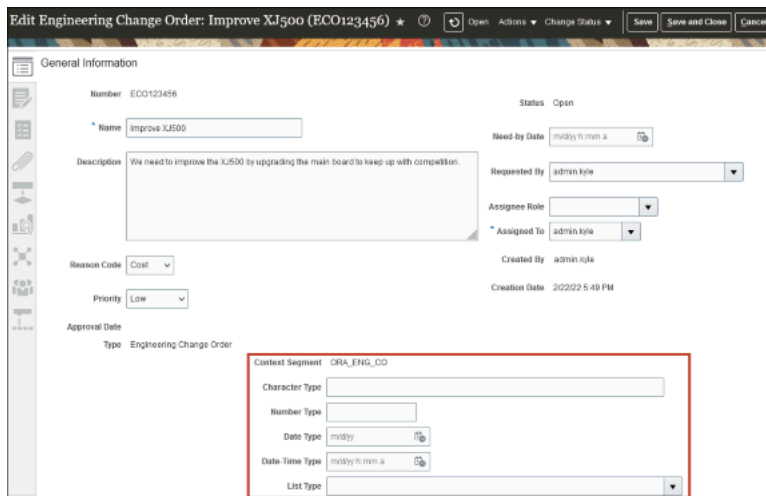
Create New Page Layouts

Use Application Composer to modify or create new page layouts. You can make changes that some, or all, of your end users can see, depending on the conditions you set.

Create Page Layouts

- In the Page Layouts node, you can create multiple page layouts for change orders. The multiple page layouts can be assigned based on conditions such as when a user selects a specific value or when a change order has reached a specific status or if the user has a specific role assigned.

ORACLE

For example, if you created a page layout and entered the following Advanced Expression to that layout: StatusName == "Open", then that layout would only appear when a change order entered the Open status.

- In addition, you can add action buttons and/or Action menu selections to a page layout that allows the user to run Groovy scripts defined by an administrator in Application Composer.

  This image displays a change order with a configured action button.
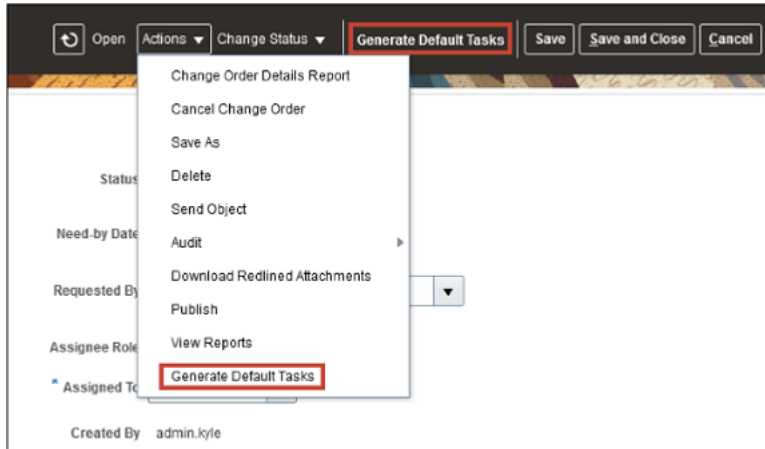
- Within a page layout, you can add new tabs with admin-defined content to fit your business process.
- You can also hide tabs and change the tab order within a page layout, but first you must set the Default Subtab, or you will receive an error. This feature enables you to configure what tabs users see, based on the conditions defined in the page layouts.

  This image displays the Subtabs Region of Application Composer
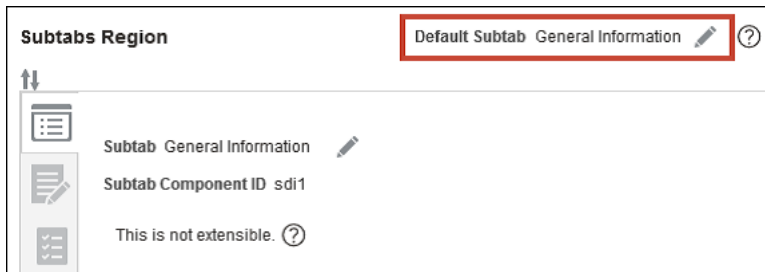
Create Tabs

Use Application Composer to create extra tabs within a page layout.

Admin-Defined Tabs

Use the tabs to display either a related object, a context link, or mashup content.

ORACLE

> **Note:** You can't create child objects for change orders. The Create Child Object button is no longer available for change order objects in Application Composer.

Create Related Object

To assign a related object, first create a new object in the Custom Objects node. After you create a new object, relate it to the change order object.

> **Note:** You can't create child objects for change orders. The **Create Child Object** button is no longer available for the change order object. However, you can create a one-to-many relationship for your configured object from the Relationships node in Application Composer. Child objects that you've created before Update 23D are available, and you can edit the attributes of such objects, although usage isn't recommended.

Here's how:

1. Select the **Relationships** node in the Common Setup pane to create a new relationship.
2. In the Create Relationship page, set **Change Order** as the Source Object.
3. Set the **Custom Object** as the Target Object.
4. Set the Cardinality to **1: M**.
5. Click **Save and Close**.

After defining the relationship between the new object and the change order object, add the object as a new tab on change orders. Here's how:

1. Navigate to **Standard Objects** > **Change Orders** > **Pages**. Select the page layout that you've created.
2. In Details Layout <page_name>, in the Subtabs Region, click **Add** icon to create a new tab.
3. Select **Related Object** and click **Next**.
4. Select the related object as the Data Object.
5. Click **Save and Close**.

Create Context Link

Here's how you create a context link:

1. Navigate to **Standard Objects** > **Change Orders** > **Pages**. Select the page layout that you've created.
2. In Details Layout <page_name>, in the Subtabs Region, click the **Add** tab icon to create a new tab.
3. Select **Context link** and click **Next**.
4. Select the Data Object.
5. Define the search criteria.
6. Select fields for the summary table.
7. Click **Save and Close**.

Assign Mashup Content

Here's how you assign mashup content:

1. Select the **Mashup Content** node in the Common Setup pane to register a web application.
2. On the Register Web Application page, enter the URL of the page you want to display on your new tab.
3. Navigate to **Standard Objects** > **Change Orders** > **Pages**. Select the page layout that you've created.
4. In Details Layout <page_name>, in the Subtabs Region, click the **Add** tab icon to create a new tab.
5. Select **Mashup Content** and click **Next** to add the new tab to the change order page.
6. Click the button next to your web application.

The web application page is rendered within your new tab.

7. Click **Insert**.
8. Click **Save and Close**.

Create Actions and Links

You can define an action or link for an object, and use Application Composer's work area configuration pages to add that action or link to a user interface page, such as a list page or details page.

Create Actions and Links

Here's how you add actions and links to a change order:

1. In the Change Order node, click **Actions and Links**.
2. In the Change Order: Actions and Links page, click the **Create** button.
3. In the Change Order: Create Action or Link page, enter the Display Label, and Name.
4. Select **Action** as the type.

   > **Note:** Links aren't supported for change orders.

5. Select Script as the Source.
6. Enter the script and choose when the script has to be run.

   - Actions must be assigned a script from the Object Functions tab found in the Server Scripts node.

   - After creating an action, you must add it to a page layout.

     Here's how you add the action to a page layout:
     i. In the Pages node, open a page layout and click the edit icon next to the Actions menu.
     ii. Add the action either to the Actions menu or as an independent button.

Create Automation with Scripting

You write Groovy scripts using Application Composer's expression builder.

Server Scripts

In the Server Scripts node, you can use Groovy scripting and web service calls to automate actions that fit your business process. Within this node you can create Validations, Triggers, or Object Functions.

Validations

- Use validations to define a condition. If the condition isn't met, ensure that you create an error message that guides the user to the correct action.

- Validations can be either Object Rules or Field Rules.

- An object rule is invoked when the user clicks **Save**. It then checks the condition to determine if the error message should be displayed.

- A field rule is invoked when the value of the selected field changes. It then checks the condition to determine if the error message should be displayed.

Triggers

Triggers are scripts that you can write to complement the default processing logic for a standard or configured object. You create triggers from the Server Scripts node in Application Composer.

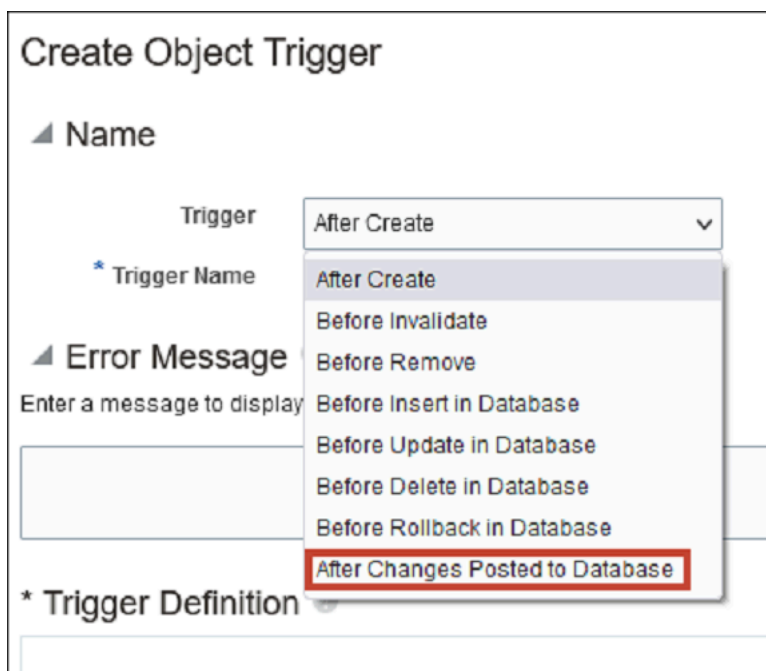Triggers update a change order based on the defined condition.

Triggers are of two types:

- Object Triggers
- Field Trigger

Object Triggers

An Object Trigger is invoked when you click **Save** or **Save and Close**. The only supported object trigger for change orders is **After Changes Posted to Database**.

This image displays an object trigger

Field Triggers

A field trigger is invoked when the value of the selected field changes.

Example

Let's say you want to automatically set the Priority of the change order to Urgent when the Reason Code is set to Error. Create a trigger that runs the following Groovy script:

```
Copy if (ReasonCode == "ERROR"){ setAttribute("PriorityCode", 'URGENT') }
```

> **Note:** The internal names of Reason Code and Priority Code are used in the code. The values associated with ReasonCode and PriorityCode are the Lookup Code for each value. You can find these values in the Manage Change Reasons and Manage Change Priorities setup tasks in Setup and Maintenance task.

> **Note:**
> - Avoid parallel updates to change header attributes by triggers and global functions. For example, if your object-level or field-level triggers attempt to update header attributes while web service calls (invoked through global functions as part of entry-exit criteria) also attempt to update these attributes, modifications made by the triggers may not get saved because the web service calls would have bumped up the object version.
>
> - The content in this topic for Triggers is also applicable for these objects: quality issues, and actions, ideas, concepts, requirements, and proposals.
>
> .

Object Functions

Object functions are used by Actions as discussed in the Actions and Links section.

> **Note:** Only change order actions and Application Composer triggers that are described in these sections are supported.

Work with Descriptive Flexfields and Scripting

Groovy scripting provides access to global descriptive flexfields and contextual descriptive flexfields. Global descriptive flexfields display on all change types while contextual descriptive flexfields only display on the change type to which they're assigned.

Work with Global Descriptive Flexfields

To work with global descriptive flexfields, use the ChangeObjectDFF accessor to access the collection of descriptive flexfields. In the descriptive flexfield collection, use the internal name of the attribute to get it and set it.

In the following example, the value of the global descriptive flexfield Implementation Reason is copied to the global descriptive flexfield PCN.

Let's look at the format for Object Functions and Field Triggers:

```
def changeDFF = ChangeObjectDFF
def reason = changeDFF.implementationReason
changeDFF.setAttribute('pcn', reason)
```

Let's look at the format for Object Triggers:

```
def changeDFF = ChangeObjectDFF
def reason = changeDFF.implementationReason
if (reason != changeDFF.getAttribute('pcn'))
 {
  changeDFF.setAttribute('pcn', reason)
```

ORACLE

```
    }
```

Work with Contextual Descriptive Flexfields

Use the ChangeObjectDFF accessor to access the descriptive flexfield collection like global descriptive flexfields. However, with contextual descriptive flexfields, assign the API name (all lower case) to a variable, and use the variable to get or set the attribute.

In the following example, the value of the global descriptive flexfield Implementation Reason is copied to the contextual descriptive flexfield Justification.

Let's look at the format for Object Functions and Field Triggers:

```
def apiName = 'justification'
def changeDFF = ChangeObjectDFF
def reason = changeDFF.implementationReason
changeDFF.setAttribute(apiName, reason)
```

Let's look at the format for Object Triggers:

```
def apiName = 'justification'
def changeDFF = ChangeObjectDFF
def reason = changeDFF.implementationReason
if(reason != changeDFF.getAttribute(apiName))
  {
  changeDFF.setAttribute(apiName, reason)
  }
```

Work with Global Functions

Global functions enable you to:

- Create reusable code that you can reference in other scripts.

- Invoke a script from a product rule using InvokeGlobalFunction.

Since change orders are available in Application Composer, you can create triggers and validations for conditionally invoking scripts in most cases. However, invoking a script based on status change can only be accomplished through a global function being invoked by a product rule triggered as an entry or exit criteria.

Items aren't available in Application Composer. Therefore, you can only apply scripts to items as global functions that are invoked through a product rule.

Since global functions have no current object context, you must consider the following points while working with global functions:

- You won't be able to use Groovy functions to perform operations in your script.

- Global functions call REST web services to accomplish what's needed.

- To write global functions that work on a particular object, and call them from an object function, refer to Passing the Current Object to a Global Function.

ORACLE

- To pass the current object context when invoking a global function from a product rule, create a parameter in the global function that receives the object ID from the rule.

This image displays the Global Functions in Application Composer.
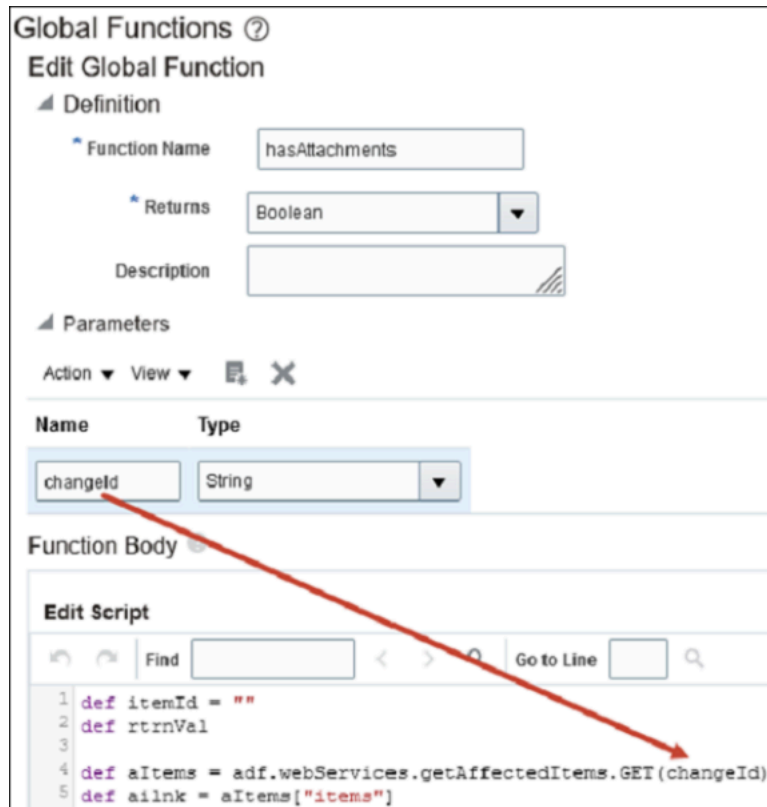
This image displays the validation condition that shows the global function
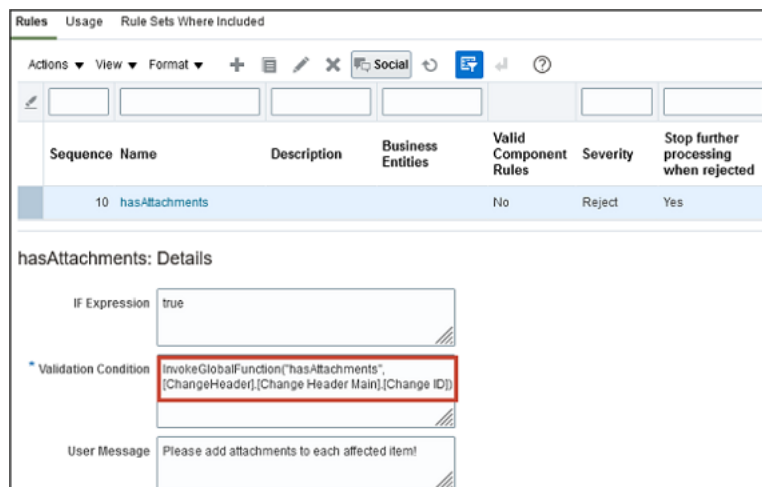
Groovy Script Examples

Application Composer supports Groovy scripting, which is a standard, dynamic scripting language for the Java platform. This section provides an example of how to use the setAttribute() function and the getAttribute() function with one or more lines of Groovy code.

Set an Attribute Value

To set the value of an attribute, use the setAttribute() function. The following example sets the Priority field to "High" on a change order using the PriorityCode attribute.

```
setAttribute("PriorityCode", "HIGH")
```

To find the code names for the priority values, in the Setup and Maintenance work area, go to the following:

- Offering: Product Management
- Functional Area: Change Orders
- Task: Manage Change Priorities

Get the Lookup Code for the value that you want to assign.

Get an Attribute Value

To get the value of an attribute, either use the getAttribute() function or just use the API name of the attribute.

Each of these scripts does the same thing, they return the value in the Priority field on a change order to Runtime Messages:

```
def getPriority = getAttribute("Priority") println(getPriority)

def priority = Priority println(priority)

println(Priority)
```

The command println() prints the value in the parenthesis to the **Runtime Messages** log in Application Composer in the Common Setup pane. To troubleshoot a script, it's very important to enable **Runtime Messages** in Application Composer.

Let's see how to enable **Runtime Messages**:

1. In the Common Setup pane, select **Runtime Messages**.
2. In the Application Script Log page, select the check box **Enable My Application Script Logging**.

Related Topics

- Global Functions
- Defining Utility Code in a Global Function
- Item Rule Object Functions
- Passing the Current Object to a Global Function

ORACLE

Work with Collections

Subobjects such as Tasks and Attachments are organized into collections. To work with attributes in a collection, use the accessor and iterate through the records or rows in the collection.

Let's look at how to work with collections.

- Get task attributes
- Set task attributes
- Add a task to change order

Get Task Attributes

Tasks are stored as a collection and therefore to access them you must iterate through them. In this case, the accessor is Change Task.

```
def changeTasks = ChangeTask
changeTasks.reset()
while(changeTasks.hasNext()){
def eachTask = changeTasks.next()
println("ActionCode: "+eachTask.ActionCode)
println("Assigned By: "+eachTask.AssignedByText)
println("Assigned On: "+eachTask.AssignedDate)
}
```

Set Task Attributes

You can update some task attributes by iterating through them, using an "if" statement to access the desired task, and then using set Attribute to set a value.

Let's look at the code to set task attributes:

```
def changeTasks = ChangeTask
changeTasks.reset()
while(changeTasks.hasNext()){
def eachTask = changeTasks.next()
if(eachTask.Name == "Update tasks"){
eachTask.setAttribute("Description",'This is another new description')
println("Description: "+eachTask.Description)
}
}
```

Add a Task to Change Order

Let's look at the code to add a task to a change order:

```
def changeTasks = ChangeTask
def newTask = changeTasks.createRow()
newTask.setAttribute("Name", "Update the design doc")
newTask.setAttribute("RequiredFlag", "N")
newTask.setAttribute("CompleteBeforeStatusCode", 6)
newTask.setAttribute("Description", "Update the design doc to fit the proposed changes.")
newTask.setAttribute("AssignedTo", "<username>")
changeTasks.insertRow(newTask)
```

ORACLE

Use Global Functions to Define Change Order Entry and Exit Criteria

You can use global functions when you write Groovy scripts using the expression builder in Application Composer.

You can use global functions to define the entry and exit criteria for change orders. Here are a few best practices, recommendations, and examples, including:

- Best practices for working with attributes in global functions.
- How to use the entry and exit criteria to populate approvers.
- Examples that show how global functions can be used with item rules.

Best Practices for Working with Attributes in Global Functions

The best practices while working with attributes in global functions are applicable to:

- Validating attributes
- Validating whether the change tabs and change affected item tabs have content (Has Content)

> **Note:** The entry criteria are applicable only for approval and open statuses, while the entry and exit criteria are applicable for interim approval status. Hence, use the open or interim approval status while validating use cases.

Here are the best practices applicable for validating attributes:

- Ensure that the validation is successful and doesn't time out. Test the validation with increments of 10 attributes at a time. If the validation takes longer than 10 seconds it will time out.
- Use the Limit parameter to limit the number of fields that are fetched for a REST call.

  Syntax: `?limit=<number>`

  The value of the Limit parameter is set to 25, in case you don't specify a value for this parameter. If you manually specify a limit value, it will fetch that many records. If you've only a few records, don't specify a high value for the Limit parameter because memory is allocated based on the Limit parameter.

- Check the attribute values of the change extensible flexfields, item operational attributes, and item extensible flexfields.
  - Limit the number of attributes to 10 while validating the values.
  - Use the configured objects to perform validations ahead of time. The configured object stores the result, and you use the global function to access the results. This global function is invoked as part of entry and exit criteria.
  - Use the query parameters to filter the data rather than iterating through all the attachments. For example, if you're trying to check if an affected item has a certain attachment category, attachment category=Secured Attachment, then you can build your query URL in the following way: `/productChangeOrdersV2/<changeId>/child/changeOrderAffectedObject/<itemUniqId>/child/changeOrderAffectedItemAttachment?q=CategoryCode="Secured Attachment"`.

- Validating whether the tabs on the change and it's affected item have content (Has Content).

○ When working on a "has content" use case, check the array size. If the array size is greater than zero, don't iterate through the result set. For example, you can check the array size for an affected item that has AMLs.

Use Entry and Exit Criteria to Add Approvers

You can define the entry and exit criteria in the Groovy script to add approvers to a workflow. Let's say you want to add approvers based on certain change header attributes. Set the change extensible flexfield attribute Team to Packaging, then add the Packaging_Role attribute as approver to interim approval or approval status.

Here's how you define the entry and exit criteria to add approvers:

1. Create a configured object that will evaluate Team to change order extensible flexfield attribute and returns the role to be added as an approver.
2. Execute the script on a scheduled process (Scheduled Custom Groovy Object Function) to populate the result in a configured attribute.
3. Configure entry or exit criteria from Interim Approval 1 to Interim Approval 2 or Interim Approval to Approval that invokes a global function and populates the approvers.

Examples of Using Global Functions with Item Rules

You can use global functions with item rules. The global functions are defined in Application Composer.

Here are the configuration steps:

1. Create a web service to get the affected items.
2. Create a global function in Application Composer.
3. Set up the item rule in the **Rules** tab of the Edit Rule Set page of the change order.
4. Configure the exit criteria in the **Workflow** tab of the Edit Change Order Type page.

Example 1 Restrict Status Change if Affected Items Have No AML

You can write a Groovy script that restricts the status change of a change order from Open to Approval if the affected items don't have AML.

Use Case: In the change exit criteria for the Open to Approval phase, the script should check if the affected objects have at least one AML (approved manufacturers list). If there's no AML, it should restrict the promotion to approval.

1. Create a web service.
   a. Web service: GetAffectedItem

      URL: https://hostname/fscmRestApi/resources/11.13.18.05/productChangeOrders/##changeId##/child/AffectedObject
   b. Web service: GetAIAML

ORACLE

URL: https://hostname/productChangeOrders/##changeId##/child/AffectedObject/##itemId##/child/AffectedItemAML

2. Create a global function - hasAML:

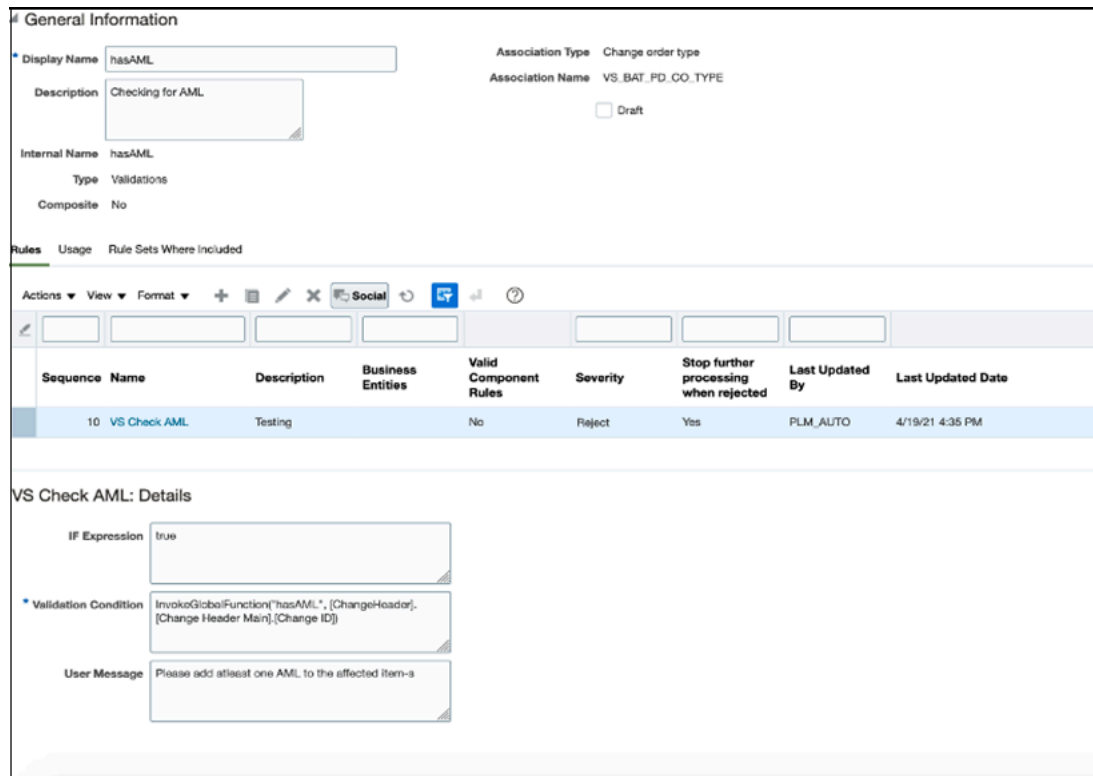**Note:** This global function is provided only as a reference.

- Trigger: Item Rules

- Object Change Order

- Tab: Affected Item -> AML

- Application Composer Navigation: Change Order -> Global Function

- Function:

```
def itemId = ""
def rtrnVal
 changeId = String.valueOf(changeId)
 println(changeId)
 def aitems = adf.webServices.GetAffectedItem.GET(changeId)
 def ailnk = aitems["items"]
 for(affectedItem in ailnk)
 {
 println("In the loop")
 def links = affectedItem["links"]
 for(lnk in links)
 {
 if(lnk["rel"].toString() =="self" && lnk["name"].toString()=="AffectedObject")
 {
 println(lnk["href"])
 itemId = String.valueOf(lnk["href"].toString().replace("https://hostname/fscmRestApi/
resources/11.13.18.05/productChangeOrders/"+changeId+"/child/AffectedObject/",""))
 println(itemId)
 }
 println(itemId)
 def attResult = adf.webServices.GetAIAttachments.GET(changeId,itemId)
 def attachments = attResult["items"]
 println(attachments["count"])
 if(attachments!="[]")
 {
 rtrnVal = false
 println("No attachments")
 println(rtrnVal)
 }
 }
 }
return rtrnVal
```

ORACLE

3. Set up the item rule in the Rules tab of the General Information page of the Edit Rule Set: <Validate Attachment> for the change order.

```
InvokeGlobalFunction("hasAML", [ChangeHeader].[Change Header Main].[Change ID])
```

Here's how you define the rule in the Edit Rule Set page.

4. Configuring the exit criteria.

  Add the hasAML rule as the exit criteria for open status.

Example 2 Restrict Status Change if Affected Items Have No Attachment

You can write a Groovy script that restricts the status change of a change order from Open to Approval if the affected items don't have attachment.

In the change exit criteria for the Open to Approval phase, the script should check if the affected objects if the affected items has atleast one attachment. If there are no attachments, it should restrict the promotion to approval.

Here are the configuration steps:

1. Create a web service.
   a. Web service: GetAffectedItem

      URL: https://hostname/fscmRestApi/resources/11.13.18.05/productChangeOrders/##changeId##/child/AffectedObject
   b. Web service: GetAIAttachments

      URL: https://hostname/fscmRestApi/resources/11.13.18.05/productChangeOrders/##changeId##/child/AffectedObject/##itemId##/child/AffectedItemAttachment

2. Create a global function in Application Composer.

Note: This global function is provided only as reference.

- o Trigger: Item Rules
- o Object: Change Order
- o Tab: Affected Item -> Attachment
- o Application Composer Navigation: Change Order ->Global Function Criteria at Open status
- o Function:

```
def itemId = ""
def rtrnVal
 changeId = String.valueOf(changeId)
 println(changeId)
 def aitems = adf.webServices.GetAffectedItem.GET(changeId)
 def ailnk = aitems["items"]
 for(affectedItem in ailnk)
 {
 println("In the loop")
 def links = affectedItem["links"]
 for(lnk in links)
 {
 if(lnk["rel"].toString() =="self" && lnk["name"].toString()=="AffectedObject")
 {
 println(lnk["href"])
 itemId = String.valueOf(lnk["href"].toString().replace("https://hostname/fscmRestApi/
resources/11.13.18.05/productChangeOrders/"+changeId+"/child/AffectedObject/",""))
 println(itemId)
 }
 println(itemId)
 def attResult = adf.webServices.GetAIAttachments.GET(changeId,itemId)
 def attachments = attResult["items"]
 println(attachments["count"])
 if(attachments!="[]")
 {
 rtrnVal = false
 println("No attachments")
 println(rtrnVal)
 }
 }
 }
 return rtrnVal
```

3. Set up Item Rule: `InvokeGlobalFunction("hasAttachment", [ChangeHeader].[Change Header Main].[Change ID])`

Here's how you define the rule in Edit Rule page
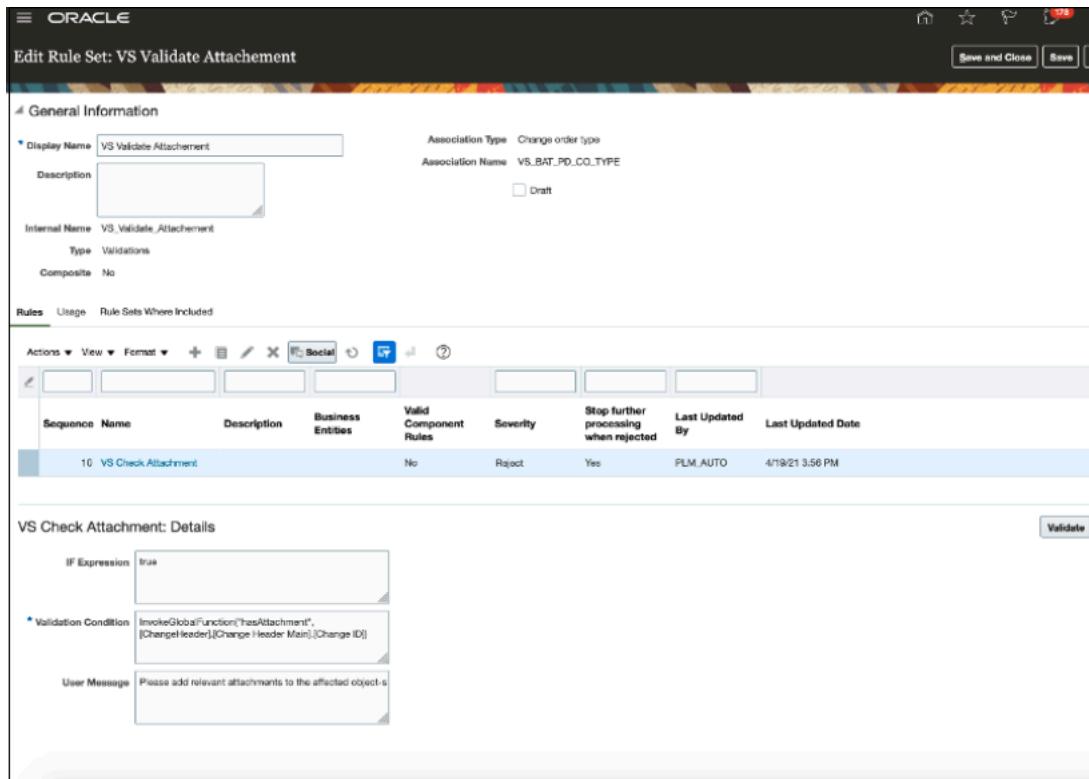
4. Define the change order entry criteria in the Workflow tab of the Edit Change Order Type page.

Add the VS Validate Attachment as the entry criteria for the approval status.

ORACLE
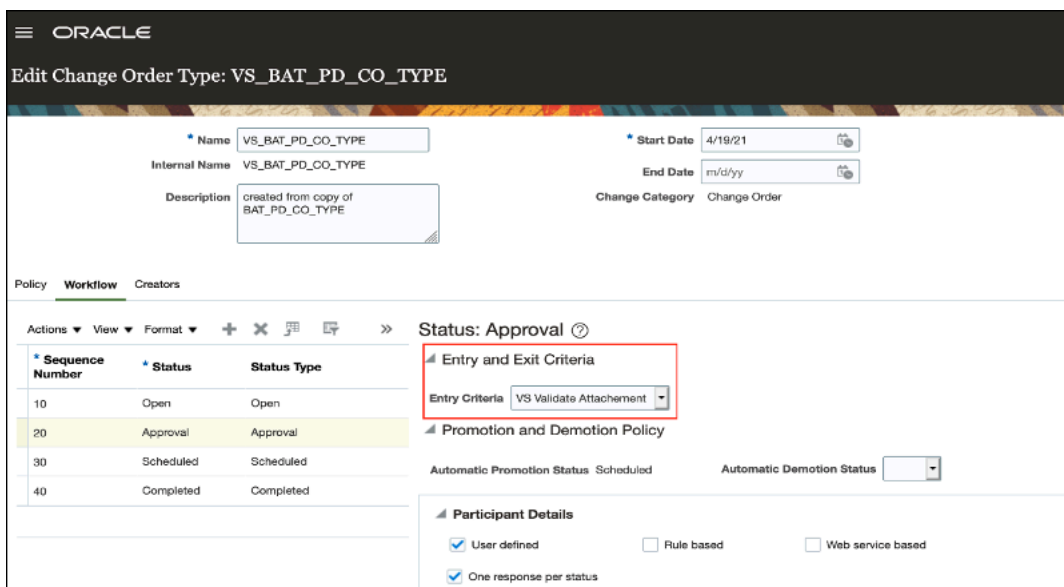
Example 3 Restricts Status Change if the Operational Attributes of Affected Items Have No Values

You can write a Groovy script that restricts status change of a change order from interim approval1 to interim approval2 if the operational attribute of affected items don't have values.

In the change exit criteria for the Interim Approval 1 to Interim Approval2 phase, the script should check if the operational attributes have been filled in for the affected items. If the operational attributes don't have values, it should restrict the promotion to interim approval2.

Here are the configuration steps:

1. Create a web service.
   a. Web service: GetAffectedItem

      URL:

      https://hostname/fscmRestApi/resoucres/11.13.18.05/productChangeOrders/##changedId##/child/AffectedObject
   b. Web service: GetAIAttributes

      URL:

      https://hostname/fscmRestApi/resources/11.13.8.05/productChangeOrdersV2/##changeId##//child/changeOrderAffectedObject/##itemId##/child/changeOrderAffectedItem

2. Create a global function in Application Composer.

> **Note:**
> - This global function is provided only as a reference. Performance may be impacted depending on the number of items and the number of attributes which you are validating.
> - REST API doesn't support all the operational attributes. See the SCM REST API documentation for details.
> - Trigger: Item Rules
> - Object: Commercialization Change Order
> - Tab: Item
> - Application Composer Navigation: Change Order ->Global Function Criteria at Interim Approval
> - Function:
>
> ```
> def itemId = ""
> def rtrnVal = true
>  println(changeId)
>  def aitems = adf.webServices.GetAffectedItem.GET(changeId)
>  def ailnk = aitems["items"]
>  for(affectedItem in ailnk)
>  {
>  println("In the loop")
>  def links = affectedItem["links"]
>  for(lnk in links)
>  {
>  if(lnk["rel"].toString() =="self" && lnk["name"].toString()=="AffectedObject")
>  {
>  println(lnk["href"])
>  itemId = String.valueOf(lnk["href"].toString().replace("https://hostname/fscmRestApi/
> resources/11.13.18.05/productChangeOrders/"+changeId+"/child/AffectedObject/",""))
>  println(itemId)
>  //Invoke affected item REST service
>  def affectedItems = adf.webServices.GetAIAttributes.GET(changeId, itemId)
>  def aiAttr = affectedItems["items"]
>  for(attr in aiAttr)
>  {
>  println(attr["EnforceShipToLocationValue"])
>  if(attr["EnforceShipToLocationValue"]== null)
>  {
>  rtrnVal=false
>  }
>  }
>  }
>  }
>  }
> return rtrnVal
> ```

3. Set up Item Rule in the Rules tab of the Edit Rule Set page of the change order.

```
InvokeGlobalFunction("validateAttrVal", [ChangeHeader].[Change Header Main].[Change ID])
```

Here's how you define a rule in the Edit Rule Set page

* Display Name	M1ORGCCO	Association Type Change order type
Description		Association Name M1ORGCCO
		☐ Draft
Internal Name	M1ORGCCO	
Type	Validations	
Composite	No	

Rules Usage Rule Sets Where Included

Actions ▼ View ▼ Format ▼ ➕ 📋 ✏️ ✖️ » 📑 ↵ ❓

Sequence	Name	Description	Business Entities	Valid Component Rules	Severity	Stop further processing when rejected
10	Validate			No	Reject	Yes

Validate: Details

IF Expression	true
* Validation Condition	InvokeGlobalFunction("checkOperationalAttr", [ChangeHeader].[Change Header Main].[Change ID])
User Message	Please set value for UnitWidthQuantity

4. Change order criteria: Define the criteria in the Workflow tab of the Edit Change Order Type page.

Add the M1ORGCCO rule as the exit criteria for interim approval status.

Example 4 SOAP Use Case: Restricts the Status Change if the Operational Attributes Have No Value

You can write a Groovy script that restricts status change of a change order from interim approval1 to interim approval2 if the operational attribute of the affected items don't have a value.

In the change exit criteria for the Interim Approval 1 to Interim Approval2 phase, the script checks if the affected items' operational attributes have value. If the operational attributes don't have a value, it should restrict the promotion to interim approval2.

Here are the configuration steps:

1. Create a web service.
   o Web service: PDCS

     URL: https://hostname/fscmservice/ProductDesignChangeOrderService?WSDL

2. Create a global function in Application Composer.

> **Note:** This global function is provided only as a reference. Performance might be impacted depending on the number of items and the number of attributes which you're validating.

- ○ Object: Commercialization Change Order

- ○ Tab: Item

- ○ Application Composer Navigation: Change Order -> Global Function Criteria at Interim Approval

- ○ Function:

```
def result=adf.webServices.PDCS.getChangeOrder(ChangeId)
def rtrnVal = true
println((String.valueOf(result.RevisedItemLine.Item.UnitWidthQuantity).replace("[","")).replace("]
def UnitWidthQuantity =
  (String.valueOf(result.RevisedItemLine.Item.UnitWidthQuantity).replace("[","")).replace("]","")
if(UnitWidthQuantity == "null")
 {
 rtrnVal=false
 }
return rtrnVal
```

ORACLE

3. Set up the item rule in the Rules tab.

```
InvokeGlobalFunction("checkOperationalAttr", [ChangeHeader].[Change Header Main].[Change ID])
```

Here's how you define the rule in the Edit Rule Set page.

4. Configure the change order criteria for the change order in the Workflow tab of the Edit Change Order Type page.

Add the M1ORGCCO as the exit criteria for interim approval status.

ORACLE

3 Extend Quality Management

Quality Management

Oracle Quality Management is performed using four different object types. These include problem reports, corrective actions, quality issues, and quality actions.

> **Note:** To configure problem reports and corrective actions, use the information provided in change orders.

What You Can Configure for Quality Management

Here are the configurations that are applicable for quality management objects.

- Additional Attribute Types, which include:
  - Dynamic Choice List
  - Fixed Choice List
- Check box
- Text
- Number
- Currency
- Percentage
- Datetime
- Time
- Long Text
- Formula
- Record Type
- Page Layout
- Subtabs that include context links and relationships
- Show or Hide Tabs
- Buttons or Actions
- URL Tabs
- First-level Objects
- Child Objects

Configure Quality Issues and Quality Actions

You can configure fields and page layouts for quality management objects in Application Composer.

ORACLE

Fields

You can create and modify all extra fields for quality issues and actions in Application Composer. The fields aren't visible to users until they're added to a page layout. Fields can be used through Groovy scripting and REST web services without adding it to a page layout. You can create the following type of fields for both issues and actions:

- Text: allows users to enter 254 characters of simple text.

  > **Note:** The 1500-character limit applies if the characters are single byte. If the characters are multibyte, such as Japanese or Chinese, then the maximum character limit is 1500 characters divided by the number of bytes per multibyte character. For example, if characters are 2 bytes, then the name is limited to a maximum of 750 characters. If a mix of characters is used, then the maximum sum of character bytes that's supported is 1500.

- Number: allows users to enter a number, including decimals.

- Date: allows users to select a date.

- Long Text: allows users to enter 32,000 characters.

  > **Note:** It's recommended that you don't exceed five long text fields on an object for performance reasons. Long text isn't supported by Audit History.

- Check box: provides a check box to select or deselect (true/false).

- Percentage: a numeric field that displays as a percentage on the page. The percentage field will take the entered value and store it as a percentage of 100. For example, if you enter 95, the value is stored as .95 although it displays 95%.

- Datetime: allows users to select a date and time.

- Record Type: invokes page layout to change based on value selection.

- Choice List (Fixed): allows the administrator to create a list of values from which users can select.

- Choice List (Dynamic): allows users to select from a list of other objects such as ideas or requirements.

- Formula: allows the administrator to prepare a calculation based on a formula and data from other fields.

Page Layouts

Here are some of the points that you must consider for page layouts:

- Fields for quality issues and quality actions are made available on the user interface through page layouts. Use the Pages node to create multiple page layouts that can apply to specific scenarios based on criteria that you define.

- The Standard page layout provides a basic display but can't be modified.

- Page layouts are displayed based on two factors: order and criteria.
  - The order of display of page layouts is based on how they're organized in the application.

  - You can change the order of the layouts to suit your purposes.

  - If you've defined the criteria for display of the page layout, based on the current user session, the page layouts are scanned from the beginning and the page layout is displayed based on the criteria.

  - Sometimes, multiple criteria may apply for the display of page layouts. In such cases, the page layout display is based on the defined criteria, and the order in which the page layout is organized. Usually, the first page layout that matches at least one criterion is displayed.

  - Page layout criteria come in the form of: Type, Role, and Advanced Expression.

- Type is determined by a Record Type field added to the page layout. You can set the page layout display based on a particular value in a Record Type field.
- Role is the assignment of one or more roles to the display of a page layout. If the user with a particular role is viewing the object, the page layout corresponding to that role is displayed.
- Advanced Expression provides you with the ability to create a Groovy script that will determine the page layout display. For example, in the Advanced Expression, if the Severity is set to High then that corresponding page layout is displayed.

  Example: Severity == "ORA_HIGH"

> **Note:** It's a best practice to create a copy of the standard page and leave it as a default layout. All page layouts created using Application Composer must be duplicates of the standard template not a copy of another configured page layout.

Custom Objects

New objects you create and configure using Application Composer are located in the Custom Objects node. You can create top-level objects (objects without a parent) or child objects (objects created in the context of a parent).

> **Note:** If you create a subtab using Application Composer, and expose the configured objects:
> - The Save As functionality won't save these objects.
> - The configured child objects aren't supported.

Important Considerations

Let's look at some of the considerations applicable to quality management objects:

- You can't use triggers on status attributes. But you can use entry and exit criteria to trigger an action based on change in the status, using global functions.
- You can use entry or exit criteria for validation use cases only. Assignment use cases aren't supported. Assignment use cases include any operation that results in adding or modifying the attribute values, or the content in any of the tabs.
- You can't use the old value, new value functions for tracking the status.
- In case you are using a workaround for storing the status code of an attribute, and using triggers based on changes to this attribute, you can do so between two open statuses only.

  It's not supported for the following status changes:

  - Open to Interim
  - Interim to Interim
  - Interim to Approval
  - Approval to Complete

Related Topics
- Watch Video: Use a Trigger to Preset Questions in the Description

ORACLE

4 Extend Innovation Management

Innovation Management

Oracle Innovation Management helps you capture ideas from any source for new products, services, markets, or customer experiences. Each proposal is evaluated for value, cost, and constraints.

Innovation Management documents, prioritizes, and agrees on requirements that can be leveraged in developing innovation concepts. Reuse existing items, trace requirements through design, and validate that each has been met to reduce new product introduction risks.

Use Application Composer to extend ideas, concepts, proposals, and requirements. Use Application Composer to develop attributes for each requested idea.

You can use Application Composer to extend Innovation Management objects in the following ways:

- Add standard and configured attributes to the pages.

- Leverage logical expressions to control the visibility of the configured page layouts.

- Use Groovy scripting to validate either a field or an object, automatically execute an action when a specific event is triggered and write object functions that can be reused in multiple contexts.

- Further, use Groovy scripting to update attributes between standard and configured objects; between two configured objects; and between two standard objects.

- Use Groovy scripting to validate through either a field or an object rule, and automatically execute an action when a specific event is triggered. These Server Scripts can be defined for both standard and configured objects.

- Provide separate Create, View, and Manage access with granular privileges.

What You Can Configure for Innovation Management Objects

Here are the configurations that are applicable for Innovation Management objects.

- Additional Attribute Types, which include:
  - Dynamic Choice List: A dynamic choice list is a field that contains a list of values, which are populated from the actual data of another object.

    Note: From a cross-application perspective, you can create a dynamic choice list field for an object in one web application that's populated by the records from a supported object in another web application.

  - Fixed Choice List: A fixed choice list is a field that contains a list of static values that are populated from the lookup types. At run time, you can select one or more values from this field, depending on the field's definition.

    You can create cascading lists of values that are narrowed down based on user selection. For example, you can create a list of countries, states, or counties, with each list displaying values based on the value selected in the parent field. If the state selected is Oregon, the list of counties displayed can be limited to the counties within that state.

ORACLE

- Check box: A check box field is a field where, in the runtime application you can select a check box, indicating a true or false attribute of a record. The available operators are Equal to, Greater than, Greater than or equal to, Is blank, Is not blank, Less than, Less than or equal to, and Not equal to.

- Text: A text field is a field where, in the runtime application, you can enter a combination of letters, numbers, or symbols.

- Number: A number field is a field where you can enter a number during runtime.

- Currency: A currency field is a field where you can enter a currency amount.

- Percentage: A percentage field is a field where you can enter a percentage. The Application Composer automatically adds the percent sign to the number.

- Datetime: A datetime field is a field where you can enter a date, or select a date from a calendar, and enter a time of day. You can show the date or time, or both.

- Date: A date field is a field where you can enter a date, or select a date from a calendar. This type of field has no time component.

- Long Text: A long text field is a field where, in the runtime application, you can enter a combination of letters, numbers, or symbols. This field type supports 32,000 characters. Long text isn't supported by Audit History.

  Note: Don't use more than five long text fields in an object for performance reasons.

- Formula: A formula field is a field that's calculated in the runtime application using the Groovy-based expression included in the formula field's definition.

- Record Type: A record type field contains a list of static values that are populated from lookup types. Associating a choice list value with a role means that only users with that role can see the choice list value. Associating a choice list value with a page layout means that after that value is selected at runtime, the associated page layout is displayed.

- Page Layout: Modify the pages on which an object appears.

- Subtabs that include context links and relationships: Add subtabs to the standard or configured object's details page to display details that are related to the current object but derived from another object, or from another source. You can specify the source of subtab data.

- Buttons or Actions: Add buttons or links to desktop pages.

- Related Objects: Related objects are connected in a foreign key-based relationship between two objects.

- Custom Objects: Custom Objects are objects that are created using Application Composer. You can create either top-level objects (objects without a parent) or child objects (objects created in the context of a parent).

  Note: If you create a subtab using Application Composer, and expose the configured objects:
  - The Save As functionality won't save these objects.
  - The configured child objects aren't supported.

Related Topics
- Triggers in Create Automation with Scripting
- Best Practices for Long Text Fields
- Watch Video: Create Cascading Select Lists

ORACLE

Configure Ideas

You can extend ideas by adding new fields or attributes, and creating page layouts.

Add actions and links to the already existing page layouts, or add new tabs to the existing page layouts to extend the Idea object. Another configuration is adding new fields to an existing object (standard objects) or create entirely new objects (custom objects).

Standard Objects

Standard objects come with tabs, and each of the standard objects lets you create and add multiple instances of the child object to one instance of the parent. You can add the child object as a tab to the parent object.

Custom Objects

Custom Objects are objects that you create using the Application Composer. You can create either top-level objects (objects without a parent) or child objects (objects created in the context of a parent).

Let's look at the attributes of a child object:

- Display Label - node name in Application Composer.

- Plural Label - page title for the object's work area and the icon name in the Navigator.

- Record Name Label - name of the field where you enter the record name.

- Record Name Data Type - This can either be user-entered text or an autogenerated sequence to identify a record.
    - Use "Text" for users to enter their own names for each record.
    - Use "Automatically Generated Sequence" to automatically create the name for each record.

- Object Name - internal name of the object.

- Child Collection Name - This is automatically created and used to assign a generic function across the collection or list.

- Security - define who can use the child object. In the Security node, set up which roles have the privilege to use the object.

    Note: If you create a subtab using Application Composer, and expose Custom Objects or configured child objects:
    - The Save As functionality won't save the Custom Objects.
    - The configured child objects aren't supported.

ORACLE

5 FAQs

What's the difference between fixed choice lists and dynamic choice lists?

A fixed choice list and a dynamic choice list are similar in that the ultimate goal of both types of choice lists is to generate a field with a list of values.

For a fixed choice list, the field's specific list values are populated from a lookup type that you select when you define the field. The list displays in a single column and doesn't change.

A dynamic choice list, meanwhile, is populated from an existing object's actual data, which you can add filters to. Based on how you define the field, the list is dynamically populated at runtime and its values can change depending on the user's context. In addition, you can add more columns to the dynamic choice list field to assist your users in making a selection at runtime.

What Application Composer tasks are available only within a sandbox?

Most Application Composer tasks require you to be in a sandbox. For example, these menu items are available to you only if you're in an active sandbox session.

- Objects

  - Custom Objects
  - Standard Objects

- Common Setup

  - Relationships
  - Role Security
  - Object Workflows
  - Global Functions
  - Run Time Messages
  - Mobile Application Setup
  - Outlook Setup
  - Personalization
  - Web Services
  - Metadata Manager

These menu items are the exceptions. They're available only in a sandbox-free session.

- Custom Subject Areas
- E-Mail Templates
- Import and Export
- Business Processes

Can two objects have the same record number?

Yes, two objects can have the same record number. The record number is unique only within a configured object.

How frequently can I publish a sandbox?

Integration sandboxes are typically published once a week. Publishing integration sandboxes less frequently than once a week isn't recommended.

When you publish an integration sandbox, all private sandboxes are invalid because the label in the mainline metadata application has changed. If you made changes to private sandboxes that you want to retain, then document those changes and then delete all the private sandboxes.

When do I publish a sandbox?

You can publish a sandbox after you've tested and verified that the application changes done in that sandbox are ready to be moved to the mainline metadata.

You must test the following configurations outside a sandbox:

- Import/Export
- Web services
- Custom subject area creation
- Object workflow
- E-mail templates

Can I delete a sandbox?

Yes. You can delete sandboxes. However, you can delete only those that aren't published.

ORACLE

Before you delete a sandbox, you must first confirm that the sandbox isn't active.

> **CAUTION:** Deletion of partial content of a sandbox is risky. It's recommended that you don't use this option.

After you've tested your application changes, you must move those changes to the integration sandbox. Publish your integration sandbox and then delete all the test-only sandboxes. You can then create and work in new sandboxes, including a new integration sandbox.

Can I delete unused configured attributes?

Yes, you can delete any unwanted or unused configured attributes in a sandbox. You can only delete the configured attributes that have never been published.

Here's how you delete the configured attribute:

1. Navigate to Application Composer work area.

   > **Note:** Ensure that you're in a sandbox.

2. Select the **ERP and SCM Cloud** option from the Application list.
3. Select the **Quality** check box in the Object tags.
4. Expand **Standard Objects** > **Quality Issue** > **Fields**.
5. Select the configured attribute and click the **Delete** button.

   In case the attribute is still in use, a message appears restricting you from deleting it. If the attribute is in use, review and remove all usages before you try again.

   The unused configured attribute is deleted.

ORACLE