Business Processes

Version: 8.2

Revised: June 2021

Copyright and Trademark Notices

Copyright © 2021 SailPoint Technologies, Inc. All Rights Reserved.

Contents

Business Process Management

A Business Process is a sequence of operations or steps that are launched to perform work. IdentityIQ Business Processes include standard out-of-the-box processes and custom installation-specific processes. System events trigger both standard and custom IdentityIQ Business Processes. The informal term workflow is also used in this document to refer to a business process.

The following events can trigger a workflow:

- Role creation or modification
- Account Group creation or modification
- Identity update
- Identity refresh
- Identity correlation
- Deferred role assignment, de-assignment
- Deferred role activation, deactivation
- Any Lifecycle Manager event
- Any Lifecycle Event (marked by changes to an Identity's attributes)

Custom workflows can be defined to do a wide variety of processing tasks. You can use:

- IdentityIQ workflow library methods and rules.
- Custom BeanShell scripts and rules.

Customizing or creating workflows generally involves a combination of XML and Java/BeanShell programming. You can manage some customization activities with the IdentityIQ graphical process editor that is included in the product. To customize or create new workflows, typically you need to be comfortable writing XML and Java.

This document has the following topics:

Workflow Basics

This section contains some key concepts for developing and using workflows. Topics include:

Terminology

The terms Business Process and Workflow are used synonymously in IdentityIQ and throughout this document.

The IdentityIQ user interface refers to these sets of connected actions as Business Processes, which is the term that business managers often use. System implementers and users working in the object model typically use the term Workflow.

Important Workflow Objects

The IdentityIQ Object Model uses four key objects in workflows. To work with workflows, you need a basic understanding of these objects.

Object	Usage
Workflow	Defines the workflow structure and steps involved in the workflow processing.
WorkflowCase	Represents a workflow in progress. Contains a workflow element in which the process is outlined and current state data is tracked. Contains identifying information about the workflow target object.
WorkflowContext	Tracks launchtime information the Workflower maintains as it advances through a workflow case. Passed into rules and scripts and to the registered WorkflowHandler. Contains all workflow variables, step arguments, current step or approval, workflow definition, libraries, and WorkflowCase.
TaskResult	Records the completion status of a task, or in this case, the workflow. Contained within the WorkflowCase.

> The most important object for writing workflows is the WorkflowContext object, which tracks the launchtime state of the workflow and performs other critical functions. Because WorkflowContext methods are used in workflows, data can be extracted from it as needed within any step of the workflow.

Workflows Operation

Workflows carry out a sequence of defined actions based on a triggering event and can be used for a variety of activities within the system. In its launching state, a workflow is tracked through a workflow case, which manages only one target entity at a time (one identity, one role, one provisioning plan, etc.).

> If multiple identities are modified at one time in a way that requires a workflow to launch for all of the identities, a separate workflow case is created to track the processing of the workflow for each single identity.

Provisioning Plans in Workflows

A provisioning plan contains a list of requested changes to an identity. Most workflows that change identities contain a single provisioning plan in a workflow variable. When performing Workflow customization you commonly need to inspect and sometimes need to modify the provisioning plan.

> Customization rules might run multiple times, updating the same ResourceObject. For example, once for the provisioning result, once for the result in the provisioning plan, and once for the result in the account request.

> Only one provisioning plan can be referenced in a workflow case at a time.

When you request changes for more than one identity at a time, even if the same change is requested for all the identities:

- A separate provisioning plan is created for each identity.
- A separate workflow case is created to manage the provisioning plan created for each identity.

Triggering Workflows

Events that occur in other parts of IdentityIQ and changes to attributes can trigger Workflows. Common Workflow triggers include:

- **Lifecycle Manager Actions** — Requests to change an identity's roles, entitlements, or accounts can activate workflows.
- **Lifecycle Events** — Creating an identity, deactivating an identity, or moving an identity from one manager to another manager can activate workflows.
- **Non-Lifecycle Events** — Editing a role, editing an account group, and changing a password can active workflows.
- **Identity Attribute Change** — Value changes can activate workflows.
- **Policy Violations** — A policy violation can activate workflows.

This table lists the main areas of IdentityIQ where you can associate Workflows to system activities.

Workflow Trigger	IdentityIQ Setup
Lifecycle Manager Requests	Select **Lifecycle Manager** from the gear icon menu and go to the Business Processes tab.
Lifecycle Events	Select **Lifecycle Events** from the **Setup** menu and specify the business process behavior.
Non-LCM-related Events	Linked to triggering events. Select **Global Settings** from the gear icon menu and go to the **IdentityIQ Configuration** page. Select the Identities, Roles, or Miscellaneous tab and then select a business process.

Workflow Trigger	IdentityIQ Setup
Identity Attribute Change	Configured with a Value Change Workflow. Select **Global Settings** from the gear icon menu and go to the Identity Mapping page. Click an attribute to edit or add a new attribute. On the Edit Identity Attribute page, go to the **Advanced Options -> Value Change Workflow** option to select the business process.
Policy Violation	Select **Policies** from the **Setup** menu, select or create a new policy, and specify the business process behavior

> You can also configure an IdentityIQ task to trigger a workflow. This workflow set up is a more complex process. See Advanced Workflow Topics .

IdentityIQ Default Workflows

IdentityIQ is preconfigured with various standard workflows that manage activities. The following workflows are examples of default workflows that are included with the product:

- Provisioning of roles or entitlements
- Account management
- Identity creation
- Password management

The default workflows can be configured and customized to address the specific business requirements of each installation. Additionally, you can write new workflows and apply them to any of the actions in IdentityIQ that support workflows.

Workflow Types

Default workflows have pre-defined workflow types. IdentityIQ uses these assigned types to determine which workflows to present in the Business Process configuration list boxes. Workflows can be specified to activate based on a specific system event.

For example, role create, update, and delete actions can trigger a **RoleModeler** type of workflow. Only workflows of that type are listed in the drop-down list for that configuration option.

> You can assign custom types to workflows. However, custom type workflows can only be triggered through the user interface on Lifecycle Events, which can trigger workflows of any type.

The table below lists the workflow type associated with each type of action within IdentityIQ.

Process Type	Description
Policy Violation	Workflow activated to launch policy violation actions.

Process Type	Description
Batch Provisioning	Workflow activated to launch batch requests.
Scheduled Assignment	Workflow activated to when a role is ready to be assigned.
Scheduled Role Activation	Workflow activated when a role is ready to be enabled or disabled.
Managed Attribute	Workflow activated when an entitlement is created or edited.
Identity Correlation	Workflow activated when performing identity correlation tasks.
Identity Event	Workflow activated for identity event. For example, sunrise/sunset dates for deferred entitlement, role assignment, or role removal.
Identity Lifecycle	Workflow activated for Lifecycle events. For example, Lifecycle Event - Joiner or Lifecycle Event - Leaver.
Identity Update	Workflow activated when you update an identity through the **Identity** -> **Identity Warehouse** page. Typically requires few or no approvals.
Identity Refresh	Workflow activated for identities that are refreshed using the Identity Refresh task. This type of process can be used for additional customization during refresh, and to present provisioning policy forms if accounts need to be created as a result of automated role assignment.
LCM Identity	Workflow associated with Lifecycle Manager Identity related tasks, for example, LCM Create and Update.
LCM Provisioning	Workflow activated for Lifecycle Manager provisioning tasks.
LCM Registration	Workflow activated for registration tasks.
Policy Violation	Workflow activated to initiate policy violation actions.
Role Modeler	Workflow associated with Role functions. For example, Owner Approval and Role Activation.
Subprocess	Designation of a workflow which is part of a larger workflow.
Password Intercept	Workflow activated when a password change interception event is received.

Sub-process Workflows

Some complex workflows are divided into multiple sub-process workflows that are activated by a master workflow. Using sub-process workflows with a master workflow can:

- Simplify the structure of the master workflow
- Make workflows easier to manage
- Promote re-usability because more than one master workflow can reference the same sub-processes

As a standard practice, these smaller workflows are assigned the **Subprocess** type of workflow. This type is not associated with any system functionality. However, using the Subprocess type designation enables you to easily identify the workflow as a sub-process of a larger workflow.

Transient Workflows

Transient workflows are launched in a special mode that does not persist any information to the database. A workflow remains in the transient state until the workflow reaches an approval step. If the workflow launches to completion without an approval step, nothing is stored in the database unless the browser terminates or the session times out the workflow and any progress made is lost.

> If the browser terminates or the session times out the workflow and any progress made is lost.

Examples of transient workflows include:

- QuickLaunch workflows that can present a series of forms before performing any relevant actions.
- Self-registration workflows that do not require authentication
- Workflows for users trying the registration process, who do not have an inbox where they can see their past attempts

To create a transient workflow, add a variable named **transient** and set the value to **true**.

For transient workflows to work correctly, the user interface code needs to manage the workflow case in a special way, through a WorkflowSession.

The case persists when any of these things happen:

- an approval for someone that is not the submitting user
- a step with a wait='x' in it
- a step with background='true'

Using the Business Process Editor with Workflows

The IdentityIQ user interface provides a graphical tool for defining and editing workflow processes. You can use the IdentityIQ Business Process Editor to:

- Create a new workflow or edit an existing workflow
- Set up the workflow structure.
- Create the steps that define the behavior or the workflow.
- Outline the transitions between the steps.
- Define forms.
- Assign conditions.

This tool also provides a graphical representation of the process flow that can be used to create documentation about the activities included in the workflow.

Typically, administrators use the graphical editor to outline the process and then move to the XML representation to add to or adjust the details of each step. After you save the process, you can view, edit or export the XML representation from the IdentityIQ Debug pages.

> Because some workflow steps cannot be defined with the graphical editor, workflow development can involve direct editing in the XML representation and some amount of Java coding. An understanding of XML and Java syntax is a general requirement for workflow development.

Creating and Editing Workflows

Use the Business Process Editor to create a new workflow or edit an existing workflow. Original workflows can also be created from existing processes.

Basic Workflow How-To Tasks

You can perform the following tasks:

- How To View or Edit a Workflow
- How To Create a New Workflow
- How To Use an Existing Workflow to Create a New Business Process

How To View or Edit a Workflow

1. Navigate to **Setup** -> **Business Processes**.
2. Select an existing workflow from the **Edit an Existing Process** list.
3. Navigate through each of the process tabs to view or modify the workflow data.
4. To save changes to an existing workflow, click **Save**.

How To Create a New Workflow

1. Navigate to **Setup** -> **Business Processes**.
2. Click **New** to create a new workflow and then enter a name for your process.
3. Specify a name and description for the workflow. Use a short descriptive name for the workflow and use a the description that provides an overview of the workflow function.
4. In the **Type** field:
    a. Select from the drop-down list of predefined workflow types. The available types are restricted to the process options related to the workflow.
    b. To enter a custom type, manually enter the type name in the box instead of selecting one from the list. See the Workflow Basics chapters for any limitations to custom types.
5. Navigate through each of the process tabs and specify workflow data.
6. Click **Save**.

How To Use an Existing Workflow to Create a New Business Process

1. Navigate to **Setup** -> **Business Processes**.
2. Select an existing workflow from the **Edit an Existing Process** list.
3. Navigate through each of the process tabs to view or modify the workflow data.
4. Click **Save As** and enter a unique name for the workflow.

Process Editor Tabs

The Process Editor has the following tabs:

Interface Tab	Inputs
Process Details	Specify Name, Type, and Description of the workflow. See Process Details Tab.
Process Variables	Lists the input, output, and processing variables you can use with the workflow. See Process Variables Tab.
Process Designer	To graphically represent the process, specify the actions involved in each step, and provide the evaluation conditions for moving from one step to another. See Process Designer Tab,
Process Metrics	Review statistics gathered for the process as it launches. See Process Metrics Tab.

Process Details Tab

The Process Details tab contains basic information about the workflow, including:

Name: A name for the workflow.

Type: The pre-defined workflow type for this workflow. IdentityIQ uses types to determine which workflows to present in the Business Process configuration list boxes. See Workflow Types for details.

Description: A description of the workflow.

Enable Monitoring: Select this option to turn on metrics tracking for the workflow.

Process Variables Tab

The Process Variables tab lists variables you can use with the workflow. For most of the default processes, the variables are listed in a collapsed, advanced view. You can expand the view to show the details for each variable. Variables include:

- Input variables for workflow
- Output variables for workflow
- Working variables used for processing a workflow

Variables are marked as **Input**, **Required**, **Editable**, or **Output**.

To delete a variable, expand the variable and click **Remove**.

Object	Usage
Input	Specifies that the variable is one of the arguments to the workflow, passed in when it is launched.
Output	Stores the variable in the workflow's task result to allow the user to view the progress and results of the workflow. To view the results, navigate to **Setup** -> **Tasks** -> **Task Results**.
Editable	Enables the variable to be edited in the basic view.
Required	Indicates that the variable must contain a value (non-null) when the workflow starts.
Description	A brief description of the variable and its function.

> The order of variable declarations can make a difference. For variables in the XML that reference other variables in their initializations, the referenced variable must be declared first.

When variables are created through the user interface, the new variables are inserted in the list above the existing variables. When the XML representation of the workflow is generated, the variables are listed in the order they were created, which is the opposite of the display order in the user interface.

Basic View

IdentityIQ has several built-in business processes that are available when you install the product. The commonly used processes are available through the Basic View which is a simplified, form-based view. The information you edit in the Basic View can be also be configured or removed using the Advanced View. The Basic View includes the following business processes:

> Business processes with the LCM label are part of IdentityIQ Lifecycle Manager, which is licensed separately.

- Identity Update
- LCM Create and Update
- LCM Manage Passwords
- LCM Provisioning
- LCM Registration

How to Use the Basic View

1. Navigate to the Debug page and edit the XML of the business process.
2. Manually add and configure the **configForm** attribute to reference the form to be presented in the Basic section of process variables. See also Editing Workflow XML.

> If the reference exists in the business process, but the form does not, an error is displayed and you are returned to the Advanced view.

Variable Initialization

To initialize variables for the workflow, specify an initial value for the variable in this panel. For best results, use initial values for the workflow variables, rather than creating multiple process steps to initialize each variable.

There are five ways that initialization can occur:

Object	Usage
String	Assigns a literal value to the variable.
Reference	Sets the variable value through a reference to one of the other workflow process variables.
Script	Sets the variable with a Beanshell script inside the workflow.
Rule	Sets the variable by calling a Beanshell rule outside the workflow.
Call Method	Assigns the return value of a call to a compiled Java method in a workflow library to the variable.

> Variable values passed into the workflow through workflow arguments supersede variable initial values. Therefore, any value provided in an argument overwrites the initial value for that argument.

Timing of Variable Definition

Variables that are known at the beginning of workflow development can be defined before the graphical process design begins. Throughout the development process you might need to define other variables. Variables are not restricted to only those that were previously defined on the Process Variables tab. Variable definition can be done before, during, or after the design process.

Process Designer Tab

The majority of the work in creating and modifying a workflow is done on the Process Designer tab. The steps and transitions you create for workflow determine the workflow activities and can include the following items.

Process Steps

A workflow involves a minimum of three steps: a start step, a processing step, and a stop step or END. For best results, all workflows should contain a start and stop step and that these two steps contain no actions. Workflows can contain as many or as few processing steps as are necessary to manage the required actions. To add steps using the Process Designer, navigate to the Process Editor and click the desired step type in the **Add A Step** section. You can drag steps around the Process Designer grid to line them up visually in a logical progression.

To add a new step:

1. Click **Add a Step** in the left-hand column to display panel that contains available steps.

   > Only steps associated with the process type and that exist in the Step Library are listed in the Add a Step panel.

2. Click and drag the desired step to a position in the process design grid.

To edit to the contents of a step,

1. Right-click the step icon and select **Edit Step**.
2. The step details window displays. You can:
   - Record the **Name** and **Description** of the step.
   - Name the **Result Variable**, a variable to receive the resulting value of the step action.
   - Specify the **Action** for the step. SeeAction Type for details.

Action Type

Each step can take one of the types of actions listed in the following table.

Object	Usage
Script	Executes a segment of Java BeanShell code that is included in the step.
Rule	Executes a workflow Rule — a block of Java BeanShell code encapsulated in a reusable rule.
Subprocess	Launches another defined workflow, passing control to it until it completes. When you select this option, the list of available subprocesses, workflows of type Sub-process, displays and you are given the option to enable step replication.
Call Method	Calls a compiled java method in the IdentityIQ workflow library, exposed through the standard workflow handler. When you select this option, the list of available methods displays.

> For any of these actions, an appropriate value must be specified or selected before the action can be saved.
>
> For example, if Script is selected, a BeanShell script must be entered in the box. If you choose the Subprocess object, a subprocess must be selected from the list. If the value is not specified, the step is saved with no associated action. Developers who use subprocesses must write the subprocesses before they can complete the step definition of steps in the master process.

The **Enable Monitoring** flag on this window turns on metrics tracking for the step. See Process Metrics Tab for more information on process monitoring and metrics.

Script

Scripts are java BeanShell code that you write in order to execute a desired action. You write scripts directly in the **Source** box in the detail window for the step.

> The script examples in this document all show very short java BeanShell code blocks. There is no set length for a script. A script block within the XML can be any length needed to accomplish the required processing. However, long scripts are frequently encapsulated in rules, as discussed in the next section.

Rule

Rules are also blocks of java BeanShell code. Code encapsulated in a rule is available for reuse by other areas of the application that can launch a rule of the same type. Rules created through this window are of type Workflow and can be used by any workflow. When you choose **Rule** as the **Action**, you can select an existing workflow rule from the list or create a new rule in the rule editor. To open the rule editor, Click the **. . .** icon.

Subprocess

Subprocesses are other workflows. You can use subprocesses to subdivide complex processes into smaller segments that can be easily managed and reused by other workflows. Subprocesses are complete workflows that contain a start step, a stop step, and as many processing steps as are needed to complete their activities.

You can enable step replication to enable multiple subprocesses to run to completion at the same time instead of having them run serially. For example, in an approval step, you can launch multiple approval subprocesses, to multiple approvers, that can take an approval all of the way through provisioning instead of the approval step waiting for all approvals to complete before provisioning can begin.

When you enable replication, you must select an item from the main workflow for replication and an argument that is passed to the action containing the replicated item. Only one item can be replicated per step, and all of the items must be passed the same argument. A new subprocess is generated for each item replicated.

Call

The IdentityIQ workflow library contains a set of methods that you launch within a workflow. Methods are exposed through the standard workflow handler that the workflow engine calls every time an action occurs in a workflow. Every workflow has access to the methods in the standard workflow handler. Additional libraries of methods are also available to use in workflows.

> When no library list is specified for the workflow, the default includes access to the Identity, Role, PolicyViolation, and LCM libraries.

Through the XML, you can specify other libraries, including custom libraries for an installation. The user interface does not provide an option to manage the library list

> Specifying a library list overrides the default. You must explicitly include in the library list any default libraries that contain methods the workflow needs. See Workflow Element for more details on specifying a library list.

When **Call Method** is selected for the workflow step, **Action**, the method name is selected from the **Call Method** list. The methods in these workflow libraries are listed and briefly described in Workflow Library Methods.

Step Arguments

When arguments need to be passed to the script, rule, subprocess, or library method launched by a step, you must specify the argument on **Arguments** tab for the step. Arguments can be specified in the following ways:

Type	Usage
Basic View	Some steps copied from the step library include a configuration form to simplify the specification of arguments. When a step has a configuration form, this is called the Basic View and is shown by default. The Basic View allows you to set arguments using literal values.
Advanced View	The advanced view gives you more control over how the argument values is calculated.
Return Variables	Each step can return only one result variable, which can be specified through the user interface. When a step has an action that launches a subprocess, you can also use return variables. Multiple values can then be passed back from the subprocess to the main workflow. Because the user interface does not provide a vehicle for declaring return variables, You must specify the return variables directly in the XML.

Type	Usage
String	A literal value. For example, the name of an email template to use.
Reference	A reference to one of the workflow's process variables.
Script	A segment of Java BeanShell code that returns a value.
Rule	A workflow rule that returns a value. This functions similar to Script except BeanShell is contained within a re-usable rule.
Call Method	A call to a workflow library method that returns a value.

When a script, rule, or library method is used to calculate an argument value, the configuration can be more complex. If the argument definition needs data to be passed in, you can pass the data by:

- Providing all the current values of workflow variables.
  OR
- Declaring the value of step arguments above an argument.

If desired, you can use ordered step arguments instead of workflow variables if the only use for the value is within this step.

For example, when these two step arguments are declared in this order, the method called to populate Identity_mgr can use the value in Identity_name in its processing if needed.

Argument Name	Value Type	Value Source
Identity_name	Reference	IdentityName
Identity_mgr	Rule	getManagerRule

More on Start and Stop Steps

Similar to other steps, start and stop steps can contain actions that launch scripts, rules, subprocesses, or calls to workflow library methods. By convention, these steps are included in every workflow but are used only to designate a clear starting and ending point for the workflow. These steps are generally empty steps with no action. Occasionally, debugging messages can be printed from these steps to trace workflow progress during development.

Step Icons

When steps are first added through the Process Designer, only three icon types are available: Start, Stop, and Generic Step. A variety of other icons are available. You can use different icons to make it easier to determine the actions each step performs.

To change an icon for a step:

1. Right-click the step icon and click **Change Icon**.
2. Select the desired icon style from the pop-up window.

Approval Steps

Approval steps are a special type of step in IdentityIQ. You can use Approvals to gather data from a user through a work item. In an approval, the user is asked to review a requested action, such as, granting a role to an identity, and then give their approval for the action to be processed.

To create a basic approval through the user interface:

1. Right-click the step.
2. Click **Add Approval**.

> A step can contain an action or an approval, but not both. Approval steps are used for approval processing. Approval steps are not used to perform other actions such as scripts, subprocesses, etc.

To edit an approval that exists in a step

1. Right-click the step and click **Edit Approval**.
2. Alternatively, you can choose **Edit Approval** from the Step Details window.

Approvals are flexible and meet a variety of business needs. An approval can be constructed many ways. approvals range from a simple one-person approval to a complex approval process that involves multiple people with different approval modes and notification schemes.

Approval Details

Every approval includes the following fields to be completed on the Details tab for the approval:

Object	Usage
Name	User-defined name for the approval.
Send	Comma-separated list of process variable names to be sent to the approval.
Return	Comma-separated list of variables names to copy from the completed approval work item back into the workflow.
Renderer	JSF (Java Server Faces) include to render the work item details. Not required if using a default renderer.
Mode	Specifies how approval is processed when multiple owners are specified.
Owner	Approver for the approval. Can be more than one Identity name and is specified by string, reference, script, rule, call method.

Object	Usage
	When more than one owner is specified, mode determines how and when the item is submitted to each listed owner. Parallel, parallelPoll, and any modes submit the approval work item to all owners at the same time. Serial and serialPoll modes wait until the first owner completes the approval before submitting to the next approver in the list.
Description	Defines work item description. Shown as the work item Name in the approver's inbox. Set using string, reference, script, rule, call method.

Approval Arguments.

You can set arguments to the approval on the **Arguments** tab. Generally, variables are passed to approval through the send list. However, any arguments that require transformation, through script, rule, or library method, must be sent through an Arg element. Args defined with reserved system names are passed through the Arg element with the reserved name specified. See Approval Steps for information on reserved system names.

Work Item Configuration

You can specify some details about the notification and escalation/reminder policy for a work item on the **Work Item Configuration** tab. The work item appears in the owner's IdentityIQ inbox and requires their input. If no configuration is specified, the default work item configuration is used.

To change the configuration for the work item

1. Select **Override Work Item Configuration.**
2. To include an electronic signature in the approval step, select **Override Electronic Signature Configuration**.

The following configuration options are available on the **Work Item Configuration** tab:

Option	Description
Initial Notification Email	To change the notification email template, select the template from the list
Escalation	Choose an escalation policy: • **None**: no escalation. • **Send Reminders**: allows configuration of reminder options, such as days before first reminder, frequency, email template. • **Reminders then Escalation**: allows reminder option configuration plus escalation option configuration, such as reminders before escalation, escalation owner rule, escalation email. • **Escalation Only**: allows configuration of escalation options, such as days before expiration, escalation owner rule, escalation email).

Child Approvals

Use Child Approvals to customize approval processing or presentation for the different sets of identities involved in the approval process. For example, a change in a user's assigned region requires someone in HR sign off and also

requires manager approval. Although the approval of the user's own manager is required, any HR individual can completes the sign-off. This type of approval can be created through child approvals.

To create a child approval:

1. Click **Add Child Approval** on the **Details** tab for the parent approval.
2. Click the child approval in the **Approval Children** hierarchy to select it for editing.

To set up the approval described in the example, create two child approvals:

- HR Approval set up — any of the identities who meet the criteria can make the decision for the group
- Manager Approval set up — identity's manager specified as the owner.

> The reference variables HRApprovers and identityManager for the example are process variables defined with initialization scripts that retrieve the appropriate sets of Identities.

If either approval requires a custom work item configuration, you can specify the configuration on the **Work Item Configuration** tab for the approval. Work item configurations are inherited by child approvals if configurations are not specifically overridden for the child. If you want a single custom work item configuration for the entire set of approvals, the configuration should be specified on **Work Item Configuration** tab for the parent approval. In this case, the child approvals inherit the parent configuration.

Form Steps

An approval step can also display a form. Forms are a general way to request information from the user and do not necessarily represent an approval. For example, you can use forms to request a missing attribute such as the department name for an identity or ask the requester for more information about why they are making the request.

You can define a form inside the workflow step or you can reference an external form that is shared with other workflows.

To reference an existing form:

1. Right-click the step and click **Add Form**.
2. In the first screen, click **Reference Form**.
3. In the form reference screen, select a form from the table and select the owner who will be shown the form.

To create a custom form for gathering data from a user:

1. Right-click the step and click **Add Form**.
2. In the first screen, click **Create**.
3. In the form editor, specify the general form properties.

Field	Description
Description	Work item description text displayed on the user’s Home Page.
Send	Comma-separated list of process variables to be passed as initial values for the form fields.

Field	Description
Return	Comma-separated list of form fields to copy back into process variables when the work item is closed.
Owner	The identity to be shown the form. Can be a simple identity name, a name stored in a process variable, or a name calculated by a script, rule, or library method.

A form includes one or more *fields* that define what information you want to show and the information you are asking the user to provide. This form field editor is similar to the field editor for provisioning policies and uses most of the same options.

Field Attribute	Description
Name	System-accessible name for field. Used to reference field programmatically.
Display Name	Label that is displayed on form for the field.
Help Text	Tool tip help text for field.
Type	Field type. Impacts rendering of field on form.
Multi-valued	Flag to determine if the field can contain multiple values (multi-selectable).
Read Only	Field displays a value that cannot be changed.
Hidden	Field is not displayed.
Owner	Field owner. Does not apply to form fields.
Required	Value must be entered.
Refresh Form on Change	Form is refreshed when the value for this field is changed. This field is useful when the value of a field in the form depends on the value in another field.
Display Only	Does not apply for workflow forms.
Authoritative	Does not apply for workflow forms.
Value	Literal, script, or rule to set the initial value of the field.
Allowed Values	Allowed values for the field. Displays as a drop-down list box or combo box based on the multi-valued setting.
Validation	Rule or script that validates the value of a field when the form is saved/submitted. Prevents submission if the value is not valid.
Dynamic	Delays the launch of allowed values, scripts, or rules until the field is selected, instead of launching as soon as the form loads.

The form editor also provides the option to specify buttons to include on the form.

To add a button definition:

1. Click **Add Button**.
2. Select the button **Action** and specify a behavior of the button.
3. Specify addition button options as described in the table below, and click **Save**.

Function	Description
Action	Select the action the button takes when pressed. Choose from the following actions: • **Next** — assimilates form data and advances to the next state, such as OK/Save/Approve/Submit functionality. Sets status of approval to Approved. • **Cancel** — Stops form editing, returns to previous page in the user interface, and leaves work item active. • **Back** — assimilates form data and returns to the previous state. Sets status of approval as **Rejected** and advances workflows. • **Refresh** — Assimilates the posted form data and regenerates the form.Not a state transition. Refresh is a re-display of the form.
Label	Text to display on the button.
Parameter	Name of an optional value to be sent with the form fields when this button is pressed.
Read Only	Non-actionable button.
Skip Validation	Ignores the validation when the form is posted.
Value	Optional value to be sent with the form fields when this button is pressed.

During initial form specification, defined buttons and fields are listed together in the left panel in the order they are added. If some buttons were added before some fields, the button can be intermixed. On the final form, buttons are always grouped together at the end of the form. When the Form Editor is revisited later, the fields are listed together first, in the order they were created, and then the buttons follow in the order they were created.

> Buttons can be reordered in the XML to display in a different order on the form.

Custom forms can also be created or edited through XML. Various advanced form options, such as sections, multi-column layout, are only available through the XML.

See the **Forms** documentation for more information.

Step Conditions

Normally when a transition is made into a step, the step action is executed. In some cases you might want the execution of the step to be optional. You can add a step condition to control whether or not the step action executes. Step conditions can also simplify transition lines in the process because you do not have to create many complex transitions to skip over steps. You can advance from one step to another and let the step conditions determine if the step is executed.

To edit the step conditions:

1. Right-click any process step.
2. Click **Add Step Condition**.
3. Specify addition button options as described in the table below, and click **Save**.

You can express conditions as any of the following:

Type	Description
Reference	Evaluation of a defined process variable. Must be a Boolean variable.
Script	Segment of java code that evaluates process variables.
Rule	Workflow rule that contains a reusable segment of java code to evaluate process variables.
Call Method	Call to launch a Java method in the IdentityIQ workflow library. Exposed through standard workflow handler.

Selecting the **Negate** option changes the evaluation to the opposite condition. For example, if the condition evaluates to False, the negate option changes it to True.

Step Transitions

Steps are connected through Transitions. Transitions can connect one step to the next sequentially. Alternatively, steps can include evaluation statements that enable conditional processing, such as certain data conditions that can cause the workflow to execute Step A versus Step B.

To add a transition do the following:

1. Right-click the process step for starting the transition and select **Start Transition**.
2. Navigate to the process step for ending the transition, right-click and select **End Transition**.
3. Right-click the transition icon and select **Edit Transition** to set the condition.
4. To add additional conditions to this transition, repeat the process.

To edit the transition conditions:

1. Right-click the transition diamond
2. Click **Edit Transitions**.
3. Specify addition button options as described in the table below, and click **Save**.

A step can have as many transitions to next steps as needed. Transition conditions are evaluated in the order they are listed. The first transition that has no condition, or whose condition evaluates to true is taken. Use the up and down arrows in the transitions dialog box to re-order the transitions. As a recommended practice, the final transition should have no condition. That transition is taken when no other transition conditions are met. If a step only has transitions with conditions, and none of the conditions are met, the workflow ends.

Conditions can be expressed as any of the following:

Type	Description
String	Not used. This condition is an artifact of the common structure used for variable setting and does not apply to conditions. A literal value of **True** or **False** can be specified but does not allow any evaluation in the transition. **True** always launches the associated step and **False** always bypasses the associated step.
Reference	Evaluation of a defined process variable. Must be a Boolean variable.
Script	Segment of java code that evaluates process variables.
Rule	Workflow rule containing reusable segment of java code to evaluate process variables.
Call Method	Call to launch a Java method in the IdentityIQ workflow library. Exposed through standard workflow handler.

Transition conditions must evaluate to boolean values. If the value is true, the workflow moves to the step that the transition references. If the value is false, the next transition in the list is evaluated.

Selecting the **Negate** option changes the evaluation to the opposite condition. For example, if the condition evaluates to False, the negate option changes it to True.

> Because all processing options should end with the stop step, every workflow should end with a step that transitions to Stop.

Process Metrics Tab

The Process Metrics tab displays the following statistics that are useful for troubleshooting workflows:

- Number of times the workflow launched.
- Number of times the workflow succeeded or failed.
- Average and maximum duration of the workflow.
- Date the workflow last launched.

You can view additional process metrics, including data tracked at the step level, through the **Intelligence -> Advanced Analytics** -> **Process Metrics Search** tab.

To turn on metrics tracking:

1. For individual workflow steps, select **Enable Monitoring** in the Details window.
2. Alternatively, you can right-click on a step and select **Enable/Disable** from the drop-down menu on the step.

To turn on monitoring for all steps in a workflow, click **Monitor** at the bottom of the business process editor window.

Editing Workflow XML

There are various options for editing workflow XML. You can:

- Create the initial workflow through the user interface and then edit the workflow directly.
- Complete all workflow development in XML.
- Write original XM or use XML from an existing workflow as a template for a new process

All of these methods are valid and can be used as desired.

Accessing the XML

The XML for existing workflows can be viewed and edited through the IdentityIQ Debug pages or can be exported through the IdentityIQ Console.

Debug Pages

To view the XML in the Debug pages, navigate to the Debug pages and Select **Workflow** from the object list to view a list of all defined workflows in the system.

to view the XML representation, click the name of the workflow. From the Debug pages you can edit and save changes. A workflow can also be copied from here and pasted into an external editor of choice.

- View and edit the XML.
- Save changes to the XML.
- Copy and paste the XML to an external editor.

IdentityIQ Console

You can export one or more workflows from IdentityIQ through the console. The console export is the most efficient way to get the XML for all workflows extracted from the system at one time. The IdentityIQ console export command can extract all the Workflow XMLs together into a single file.

See the **IdentityIQ Console** documentation for more details.

After export the XML, you can parse the XML into a separate file for each workflow and save the files in the installation source code control system for later use in system environment migrations or in product upgrade processes.

Object	Usage
Workflow	Defines the workflow structure and steps involved in the workflow processing.
WorkflowCase	Represents a workflow in progress. Contains a Workflow element in which the process is outlined and current state data is tracked, as well as identifying information about the workflow target object.
WorkflowContext	launchtime information that Workflower maintains as it advances through a workflow case. Passed into rules and scripts and to the registered WorkflowHandler. Contains all workflow variables, step arguments, current step or approval, workflow definition, libraries, and workflowCase.

Object	Usage
Task Result	Records the completion status of a task, or in this case, the workflow, contained within the WorkflowCase.

Re-importing the XML

Because the system only launches Workflow XML that is saved within IdentityIQ, XML documents that are edited externally must be re-imported for the changes made to them to take effect.

To re-import an externally saved XML document, use the console import command or from **the Import from File** page accessed from the **gear menu > Global Settings** page.

Dollar-Sign Reference Syntax

You can reference workflow variables inside XML tags and in user interface fields using $() notation. These are resolved into their variable values. For example, if a variable identityName is defined and contains the full name of an Identity, for example, John Smith, an Arg specified as:

```
<Arg name="FullIdentityName" value="$(identityName)">
```

passes “John Smith” as the value for the variable **FullIdentityName**.

When the variable is used alone, it functions the same as specifying value="ref:identityName. However, the more common usage is to include the variable in a longer string such as:

```
<Arg name="Title" value="Role Update for $(identityName)">
```

which passes “Role Update for John Smith” as the value for the variable Title.

XML Content

This section describes the elements present in the workflow XML and explains their usage.

Header Elements

The following three lines must be included as shown in any workflow document. The <sailpoint> tag must, of course, be matched with a </sailpoint> tag at the end of the workflow document.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE SailPoint PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<sailpoint>
```

Workflow Element

The Workflow tag identifies the name and type of the workflow.

```
<Workflow  explicitTransitions="true" name="WF-Training Hello World Workflow" type-
e="IdentityUpdate">
```

The attributes of a workflow element including the following:

Workflow Attribute	Purpose
configForm	A soft reference to a process variable form presented in the Basic View of Process Variables tab, or a step form presented in the Basic View of the Arguments tab on the Step Editor panel accessed from the Process Designer tab of the Manage Business Process page.
name	Short descriptive name for the workflow this is displayed in user interface selection list-boxes and list of existing business processes on the Process Editor window.
type	Workflow type. Type is used to filter workflow selection lists in configuration windows where you select the workflow based on system activities.
explicitTransitions	Boolean value indicating that transitions between steps are explicitly specified and workflow should not resort to implicit, fall-through, transitions when no transition conditions evaluate to **true**. The default setting is **false.** If you so omit this argument and the specified transition conditions all evaluate to **false**, the workflow uses implicit transitions and launches the next sequential step in the XML. However, if you edit a workflow using the Business Process Editor, the value is changed to **true**. If the developer makes the last transition in any set unconditional, which is considered best practice, the transitions between steps are smoother.
libraries	Lists workflow libraries the workflow needs. If this attribute is not specified, workflows automatically have access to Identity, Role, PolicyViolation, and LCM libraries.
stepLibraries	Lists workflow step libraries the workflow can access. If this attribute is not specified, workflows automatically have access to the Generic Step Library, which provides access to the Start, Stop and Generic steps.
handler	The default workflow handler is sailpoint.api.StandardWorkflowHandler. This attribute does not need to be specified when the default is used. In this case, the best practice is to omit it. If you use a custom workflow handler, the custom handler must EXTEND the default handler and not replace it. The custom handler must be specified in the workflow Handler argument.

Variable Definitions

The recommended best practice is to identify all variables for the workflow at the top of the XML document. The variable definitions come next in the XML.

At a minimum, variable elements require a name. Other attributes can indicate the variable type and use, such as input, required, editable, return. A description can be specified for each variable. When needed, an initialization value can also be provided. Using the initialization option is the recommended practice rather than creating separate steps

to initialize each variable. Using initialization values is more efficient, easier to read, and easier to debug, because Trace reports initializations as they occur. For more information, see Initializer Options .

```
<Variable input="true" name="project" output="true" required="true">
    <Description>
      Project that has account requests in the QUEUED state.
    </Description>
  </Variable>

<Variable editable="true" initializer="true" name="doProvisioning">
    <Description>Set to true to cause immediate provisioning after the assign-
ment</Description>
  </Variable>
```

Some parts of the variable definition are expressed within attributes on the Variables element. Other parts are expressed through nested elements of their own.

Variable Attribute	Purpose
name	Variable name.
type	Variable type. Type declaration is not enforced by the application and is used primarily for documentation.
initializer	Initialization value for the field.
input	Flag indicating that the variable is an argument to the workflow. Omitted if not true.
output	Flag indicating that the variable is a return value for the workflow. Omitted if not true.
required	Flag indicating that the variable is a required field for the workflow. Omitted if not true.
editable	Flag indicating that the variable can be edited by the workflow. Omitted if not true.
Nested Tag within Variable Element	
Description	Provides a description of the purpose for the variable.
Script	Alternative to script in the initializer attribute value. Should be used for initializer scripts of any length or complexity.
Source	Nested within the Script tag and contains the java BeanShell source for the action to be executed.

Initializer Options

The Initializer attribute requires additional attention. When these attributes are set through the user interface, you can specify the attribute as a string, script, rule, call, or reference. The same options are available directly through the XML.

> The initializer for a variable is only used when a value for the variable is not passed in to the workflow.

Initializer Type	Description and Examples
string	Assigns a literal value to the variable. String is the default initializer option so the "string:" prefix can be included or omitted. Examples: <Variable initializer="string:true" name="trace"/> <Variable initializer="spadmin" input="true" name="fallbackApprover">

script	Assigns a value based on the results of a Java BeanShell script.
	
	Examples:
	
	(1) In-line Script. Only use for very short, simple scripts.
	
	`<Variable initializer="script:(identityDisplayName != void) ? identityDisplayName : resolveDisplayName(identityName)" input="true" name="identityDisplayName">`
	
	(2) Script within nested <script> element. Use for most script initializers - scripts of any complexity or length.
	
	`<Variable initializer="script:resolveDisplayName(launcher)" input="true" name="launcherDisplayName">`
	`<Description>`
	`The displayName of the identity being who started this workflow.`
	`Query for this using a projection query and fall back to the name.`
	`</Description>`
	`<Script>`
	`<Source>`
	`// Lookup the launcher's display name for use in email templates.`
	`String returnString = launcher;`
	`Identity launcherId = context.getObject(Identity.class, launcher);`
	`if (null != launcherId) { returnString = launcherId.getDisplayName(); // First+Last }`
	`return returnString;`
	`</Source>`
	`</Script>`
	`</Variable>`

Initializer Type	Description and Examples
rule	Assigns a value based on the return value of a workflow Rule. Examples: `<Variable initializer="rule:wfrule_GetIdentityName" name="IdentityName">`
call	Assigns a value based on the return value of a call to a workflow library method. Example: `<Variable initializer="call:getObjectName" name="roleName">`
ref	Assigns a value based on a reference to another workflow variable. This type is rarely used. Example: `<Variable initializer="ref:otherVar" name="myVar"/>`

Workflow Description

A description element should be included to describe the purpose of the workflow. Although the description element is not used in the workflow process, it is recommended for usability. In the user interface, the contents of this element are displayed on the Process Details tab of the Business Process page. This element should be included near the top of the workflow, either before or after the variable definition section.

```
<Description>
     Workflow called when a role is ready to be enabled.
</Description>
```

Rule Libraries

Some methods the workflows use are grouped together into Rule Libraries. These Rule Libraries are defined as rules in IdentityIQ. However, these libraries contain sets of related but unconnected methods that workflow steps can directly within a script action. Because the rule methods are in rules, rather than in the compiled Java classes, their functionality can be easily modified to meet the needs of each installation. To make the methods within one of these rules available to steps within the workflow, the RuleLibraries element must be declared. See the following example.

> Each Reference element applies to one library. Include only the libraries that contain the required methods in the RuleLibraries declaration for the workflow.

```
<RuleLibraries>
    <Reference class="sailpoint.object.Rule" name="Workflow Library"/>
    <Reference class="sailpoint.object.Rule" name="Approval Library"/>
    <Reference class="sailpoint.object.Rule" name="LCM Workflow Library"/>
</RuleLibraries>
```

> You can create and reference custom libraries using this same syntax.

Step Libraries

Step libraries are designed to offer a group of common functions that can be added to existing workflows from the **Add a Step** panel Business Process Editor. Step libraries are a collection of steps encapsulated by a workflow with the template attribute marked true. The steps do not have any transitions and they are not executable. A Step Library must be defined. See the following example.

> The type does not have to be StepLibrary. However, using the StepLibrary type ensures that these workflows do not appear in other parts of the product.

```
<Workflow name="Provisioning Step Library"
                  type="StepLibrary"
                  template="true">
```

When you edit a new or existing workflow, you can include a list of step libraries by including a comma separated list in the stepLibraries attribute. See the following example.

```
<Workflow name="LCM Provisioning"
                  type="Provisioning"
                  taskType="LCM"
                  libraries="Identity,Role,PolicyViolation,LCM,BatchRequest"
                  stepLibraries="Common,Provisioning"
                  handler='IIQ.api.StandardWorkflowHandler'>
```

In the example above, when you edit a business process with the LCMProvisioning type, the Common and Provisioning steps are available in the **Add a Step** panel of the Business Process Editor.

Steps within a step library workflow can also include a soft reference to a step form that provides a simplified form-based interface that you can use to add arguments to some steps in the workflow. This form-based interface adds a Basic view option to the **Arguments** tab of the Step Editor. The Basic view is built using the information contained in the referenced form. The Advanced view is a list of all possible arguments and is built using the list of arguments that the step library references.

When you add a step form reference to a step library, use the configForm attribute, See the following example.

```
<Workflow name="Provisioning Step Library"
                      template="true"
                  type="StepLibrary">
 <Step configForm="Provisioning Approval Step Form"
                  icon="Task"
                  name="Account Approval">
<Arg name="approvalMode"/>
<Arg name="approvalScheme"/>
<Arg name="approvalSet" value="ref:approvalSet"/>
...
```

In the example above, when you edit an approval step in the Step Editor, the Basic and Advance Views of the Arguments tab are displayed.

Built-in Steps

IdentityIQ includes several built-in steps. The **Start**, **Stop**, and **Generic** steps apply to all workflow types. The following table lists the names, descriptions, and associated workflow types of additional built-in steps.

Step	Description	Process Type
Notify	Allows users to select categories of recipients to notify, the specific recipient, recipients for each category, and the specific email template to use for each category.	Identity Lifecycle LCM Provisioning
Account Approval	Used for provisioning request approvals. The process assumes many of the Provisioning Workflow structures exist.	Identity Lifecycle LCM Provisioning

Step Elements

The core of the workflow is contained within the step elements. At a minimum, a step should contain:. The action attribute determines what processing the step performs. Steps usually contain one or more nested `<Transition>` elements and ideally also contain a nested `<Description>` element that tells the reader what the step is intended to do.

- an icon
- name
- posX attribute
- posY attribute

The action attribute determines what processing the step performs. Steps usually contain one or more nested <Transition> elements and ideally also contain a nested <Description> element that tells the reader what the step is intended to do.

```
<Step icon="Start" name="Start" posX="250" posY="126">
    <Description>
        The workflow's processing starts with this step.
    </Description>
    <Transition to="Initialize"/>
  </Step>
```

> Similar to variables, some parts of a step definition are included as attributes of the step and others are expressed as nested elements within the step.

Step Attribute	Purpose
configForm	A soft reference to the form that is presented to the Basic View of the Arguments tab on the Step Editor panel.
name	Short but descriptive name for step displayed in user interface graphical display below the step icon.
icon	Icon to display for the step in the user interface graphical Process Designer. Valid icon values include: Start, Stop, Default (Generic Step), Analysis (Launch Impact Analysis), Approval, Audit, Catches, Email, Message (Add Message), Provision, Task (Launch Task), and Und
posX, posY	X and Y indicate positions where the step icon should be displayed on the user interface graphical Process Designer grid. If you omit the posX and posY values, the icon is displayed at the top right of the grid. You can drag the icon around to create the desired layout at a later time.
action	The processing action to take for the step, such as a script, rule, subprocess, or call. See Step Actions.
wait	Pauses the action for a specified duration, see Wait Attribute.
catches	Causes the step to be launch when **Complete** status is caught, rather than through a transition from another step. See Catches Attribute.
resultVariable	Variable name that contains the return value from the step.
Nested Tag within Step Element	
Description	Provide a description of the step purpose.
Transition	Identifies the next step the process moves to when the current step is complete. See Transition Element.
Arg	Passes variables to the step. Used for steps that require data to be passed in to them.
Return	Receives return values from subprocess steps. See Return Elements.
Script	Alternative to script in the Action attribute for the step. Use these step attributes for action scripts of any length or complexity.
Source	Nested within the Script tag and contains the Java BeanShell source for the action to execute.

Transition Element

The transition element indicates the name of the next step the process executes following completion of the current step and is always nested within a step in the model. Transitions can contain conditions based on a string, script, rule, call method, or reference (similar to a variable initialization). The return value for conditions must be a Boolean (True/False). When multiple transitions are stipulated, they are evaluated in the order they are listed, and the transition

for the first condition met is followed. The last transition in the list should, as a best practice, not contain any conditions so it can be used as the default action.

Transitions contain two attributes:

- to — next step
- when — condition for progressing to the next step

When a script is evaluated as the condition for a transition, it is often specified through these nested elements instead of as a **when** attribute on the transition element, especially if you use a long script.

Nested Tag Within Transition Element	Purpose
Script	Alternative to script in the transition **when** attribute. The script should be used for scripts of any length or complexity.
Source	Nested within the Script tag. This tag contains the BeanShell source for the condition evaluation.

Example:

```
<Transition to=”end”>
    <Script>
       <Source>
           (“cancel”.equals(violationReviewDecision) || ((size(policyViolations)
            > 0 ) &amp;amp;&amp;amp; (policyScheme.equals(“fail”))))
       </Source>
    </Script>
</Transition>
```

Conditions in the **when** attribute can be specified using the following types of conditions:

Condition Type	Description and Examples
string	Not used. This condition type is an artifact of the common structure used for variable setting and does not apply to conditions. A literal value of **True** or **False** can be specified. However, using one of those literal values does not enable any evaluation in the transition. **True** always executes the associated step and **False** always bypasses the step.
script	Evaluates script result value to determine step transition. Very short scripts are specified inline on the transition element, within the **when** attribute. Longer scripts are expressed within nested <script> and <source> elements. Because script is the default transition when option, the “script:" prefix can be included or omitted. Examples:

Condition Type	Description and Examples
	(1) In-line Script. Use only for very short, simple scripts. <Transition to="Exit On Policy Violation" when="script:((size(policyViolations)> 0) && (policyScheme.equals("fail")))"/> (2) Longer script within nested <script> tag should be use for transition scripts of any complexity or length. <Transition to="end"> <Script> <Source> ("cancel".equals(violationReviewDecision) || ((size (policyViolations) > 0) && (policyScheme.equals("fail")))) </Source> </Script> </Transition>
rule	Evaluates the return value of a workflow rule to determine step transition. Examples: <Transition to="Process Approval" when="rule:RequireApprovalRule">
call	Evaluates return value of a call to a workflow library method to determine step transition. Example: <Transition to:"Check Status" when="call:requiresStatusCheck" />
ref	Evaluates a defined, Boolean, workflow variable to determine step transition. Example: <Transition to="Refresh Identity" when="ref:doRefresh"/>
Unconditional	Specified as last transition option to give a default path for the transition. Example: <Transition to="Approve"/>

Step Actions

Most steps involve much more than a name and a transition. Steps also include an action attribute that executes the workflow processing. The action of a step can be a script or can a rule, subprocess, or a call to a workflow library method.

<table>
<tr><th>Action Type</th><th>Description</th></tr>
<tr><td>Script</td><td>Similar to scripts in other parts of the workflow XML, the script can be contained within the action attribute or can be nested within the Step in a <Script> block.

Examples:

(1) In-line Script, used only for very short, simple scripts.

<pre><Step action="script:approvalSet.setAllProvisioned();"
icon="Task" name="Post Provision">
<Transition to="Stop"/>
</Step></pre>
(2) Longer script within nested <script> tag. Used for action scripts of any complexity or length.

<pre><Step name="Start" icon="Start" posX="20" posY="20">
<Script>
<Source>
String wfName = wfcontext.getWorkflow().getName();
System.out.println("Starting workflow: [" + wfName + "]");
</Source>
</Script>
<Transition to="Compile Provisioning Project"/>
</Step></pre></td></tr>
<tr><td>Rule</td><td>A step can execute a block of Java BeanShell code encapsulated in a reusable workflow Rule.

Example:

<pre><Step action="rule:WFRule_verifyIdentity" icon="Task"
name="Verify Identity" posX="600" posY="202"></pre></td></tr>
<tr><td>Subprocess</td><td>When you include a <WorkflowRef> element within the step and reference the SailPoint.object.Workflow class and the specific workflow by name, a subprocess is defined.

Example:

<pre><Step icon="Task" name="Initialize" posX="320" posY="126">
…
<WorkflowRef>
<Reference class="sailpoint.object.Workflow" name="Identity
Request Initialize"/>
</WorkflowRef></pre></td></tr>
</table>

Action Type	Description
	<Transition to="end"> </Step>
Call	Calls to workflow library methods can be used to do step processing. Call is the default action option. Therefore the "call:" prefix can be included or omitted. Example: <Step action="call:refreshIdentity" icon="Task" name="Refresh Identity" posX="618" posY="242">

Arguments

Any variables to be passed to a script, rule, subprocess, or library method must be declared as step arguments through <Arg> elements. Similar to other variables, the values for arguments can be specified by string, script, rule, call, or reference. The default specification type is string. Therefore, the “string:” qualifier can be omitted. However, arguments are also commonly passed by referencing workflow variables.

```
Step icon="Task" name="Initialize" posX="320" posY="126">
    <Arg name="w" vaflolue="ref:flow"/>
    <Arg name="formTemplate" value="string:Identity Update"/>
    <Arg name="identityName" value="ref:identityName"/>
     …
    <Description>Call the standard subprocess to initialize the request,
       this includes auditing, building the approvalset, compiling the plan into
       project and checking policy violations.</Description>
    …
    <WorkflowRef>
        <Reference class="sailpoint.object.Workflow" name="Identity Request
           Initialize"/>
    </WorkflowRef>
    <Transition to="end">
</Step>
```

When an argument is specified as a script, rule, or call, for example, <Arg name="myVar" value="rule:myWFRule"/>, any needed arguments to the script, rule, or library method cannot be explicitly specified.

Because these scripts, rules, and library methods automatically have access to the workflow context object, the scripts can access workflow variables directly through the workflow context get methods. These scripts/rules/methods can also access any step arguments that were defined before them in the step declaration. For example, the method that identifies the value for the Manager argument can use the value in the identityName argument in its processing, if needed. See the following example.

```
<Step icon="Task" name="Processing Step" posX="320" posY="126">
    <Arg name="identityName" value="ref:identityName"/>
    <Arg name="Manager" value="call:getManager"/>
    …
</Step>
```

The following table lists the available Arg attributes

Arg Attribute	Purpose
name	Variable name in process to which the data is being passed.
value	Value to pass into the variable, such as string, script, rule, call, reference.

Return Elements

To return more than one value from a subprocess, you can declare <Return> elements for the step. At a minimum, a return element contains: a **name** attribute and a **to** attribute. The name attribute is the name of the variable in the sub-process workflow and the **to** attribute is the variable name in the calling (current) workflow. If these names are the same in both workflows, a **to** attribute is not required. However, specifying a to attribute is a best practice for clarity.

Use the merge attribute when the variable is a List and the returned values should be appended to the current work-flow's list instead of replacing it. Similar to Args, value attribute for return elements can be specified as a string, script, rule, call, or reference. String is the default. If the value argument is omitted, the value of the name variable is copied as-is into the to variable, However, a script/rule/method can be used to transform or modify the value as it is passed.

- **name** attribute — name of the variable in the subprocess workflow
- **to** attribute — variable name in the calling (current) workflow

> If these names are the same in both workflows, a **to** attribute is not required. However, specifying the to attribute is best practice.

```
<Step icon="Task" name="Initialize" posX="320" posY="126">
    <Arg name="flow" value="ref:flow"/>
    <Arg name="formTemplate" value="string:Identity Update"/>
    <Arg name="identityName" value="ref:identityName"/>
     …
    <Return name="project" to="project"/>
    <Return merge="true" name="workItemComments" to="workItemComments"/>
    <WorkflowRef>
        <Reference class="sailpoint.object.Workflow" name="Identity Request
           Initialize"/>
    </WorkflowRef>
    <Transition to="end">
</Step>
```

The following table lists the available Return attributes.

Return Attribute	Purpose
name	Variable name in process from which the data is returned.
to	Variable name in the workflow step to which the data is passed.

Return Attribute	Purpose
value	Value to pass into the variable, such as string, script, rule, call, reference.
merge	Flag indicating that the value should be merged into the target variable instead of replacing the variable. This attribute is used for list variables.
local	Only applies to returns on Approvals. (See Approval Steps). A flag that indicate the value is passed to local storage within the parent approval and not passed to a workflow case variable. This attribute is used for complex approvals where a work item state is saved for later analysis in a script.

Call

Use calls to workflow library methods to do step processing. Similar to subprocesses, they sometimes require arguments to be passed to them. You declare method arguments the same way as subprocesses. You use Library methods with a call action. See the following example.

```
<Step action="call:refreshIdentity" icon="Task" name="Refresh Identity" posX="618" posY-
Y="242">
    <Arg name="identityName" value="ref:identityName"/>
    <Arg name="correlateEntitlements" value="string:true"/>
    <Description>Add arguments as necessary to enable refresh features. Typically you
      only want this to correlate roles. Don't ask for provisioning since that
      can result in provisioning policies that need to be presented and it's
      too late for that. This is only to get role detection and exception
      entitlements in the cube.</Description>
    <Transition to="Notify"/>
  </Step>
```

The methods available for the call action are those included in the libraries attribute for the workflow element, if specified. If no libraries attribute is specified, the workflow automatically has access to the methods in the Identity, Role, PolicyViolation, and Lifecycle Manager libraries. If other libraries, including custom libraries, are explicitly listed in the libraries attribute, any of the default libraries whose methods are needed by the workflow must also be explicitly included in the list to be available. See Workflow Library Methods for details about the methods available in each library.

> Installations can create custom libraries for commonly used and required business methods. However, custom library methods must be named with unique names that do not conflict with standard library method names. Conflicts resolve as a reference to the standard library method. It is possible to extend a standard library and overload its method names. Extending a standard library is not consider a best practice. Therefore, the best practice is to create new names for nonstandard methods. Creating new names makes it clear that the method is not a standard method.

Wait Attribute

The step wait attribute causes the workflow to pause in its execution for the duration specified. The wait value can be specified as a string, script, rule, call, or reference. String is the default.

```
<Step name="Wait for next check" wait="ref:provisioningCheckStatusInterval">
    <Description>
       Pause and wait for things to happen on the PE side.
       Use the configurable interval to determine how long
       we wait in between checks.
    </Description>
    <Transition to="CheckStatus"/>
  </Step>
```

This attribute creates a special type of step with the sole purpose of creating a pause in the action. Wait steps are commonly used in re-try logic to enable behind-the-scenes processing to occur before the workflow attempts to repeat an action.

Catches Attribute

These steps are not caused through a transition from a previous step. These steps are caused by a thrown message that the steps intercepts or catches. Currently, only a complete message is thrown and can be caught. This process occurs when one of the following items occurs:

- All sequential steps in a workflow are executed to completion.
  OR
- Failure condition results in the termination of the workflow.

```
<Step catches="complete" icon="Task" name="Finalize">
    <Arg name="project" value="ref:project"/>
    <Arg name="approvalSet" value="ref:approvalSet"/>
    <Arg name="trace" value="ref:trace"/>
    <Description>
       Call the standard subprocess that can audit/finalize the request.
    </Description>
    <WorkflowRef>
       <Reference class="sailpoint.object.Workflow" name="Identity Request Finalize"/>
    </WorkflowRef>
    <Transition to="end"/>
```

The primary purpose of these steps is to update the IdentityRequest object, which tracks and reports the status of a LifecycleManager request, making the history of LCM request processing available even after the TaskResult for the workflow was purged.

Each installation can drive custom logic based on catching this complete message.

Approval Steps

Approval is one of the most common actions that a workflow process performs. The IdentityIQ Approval model is constructed to simplify the process of defining an approval structure. Approvals are a special type of step that contain an <Approval> element, specifying how the approval work item is presented for approval.

Some approval steps are designed to get a user's approval on a requested change, as the name implies. However, the approval element can be used any time data needs to be gathered from a user.

Typically, when you use approval steps to gather non-approval data, you use a custom form to:

- Present the work item to the user
  and
- Request the needed information from the user.

For information on creating approval steps, see the section above. Through the XML, the custom form is manually defined within an approval step. You can also specify custom forms for traditional approvals when you need to present the information differently than the standard approval forms layout. SeeWorkflow Forms for more details on usage of custom forms.

Similar to other Workflow elements, you specify some modifiers as attributes on the approval element and specify other modifiers through nested elements within the approval.

Approval Attribute	Purpose
mode	Specifies how an approval is processed. Mode can be determined from string, script, rule, call, or reference String is the default. The user interface only supports the selection of a string of one of the values listed below.The XML also enables reference to a process variable containing one of those values or the specification of a script, rule, or method call that can determine one of those values programmatically. Valid values are: • **serial** - approvers are specified in order and the item is passed to each approver in that order. If any approver in the chain rejects, the item is rejected. • **serialPoll** - approvers are specified in order and item is passed to each approver in that order. Data is collected on approvals and rejections. However, if one approver rejects, does not necessarily result in the item being rejected. The action decision is expected to be specified in AfterScript logic. • **parallel** - item is sent to all named approvers at one time. The item is rejected if any approver rejects it. • **parallelPoll** - item is sent to all named approvers at one time. Data is collected on approvals and rejections but rejection by one does not mean rejection of item. The action decision is expected to be specified in AfterScript logic. • **any** - item is sent to all named approvers at one time. The first approver to respond makes the decision for the group.
owner	One or more approvers can be specified by string, script, rule, call, or reference. String is the default. The mode determines how and when the item is submitted to each listed owner when more than one is specified.
renderer	JSF include to render the work item details.
return	Comma-separated values (CSV) list of variable names to copy from com-

Approval Attribute	Purpose
	pleted work items back into workflow.
send	CSV list of variable names to include in the work items.
description	Defines work item description. For nested approvals, child approvals use the work item defined by the parent approval unless the child approval defines its own work item. You can set the description by string, script, rule, call, or reference String is the default.
validation	Used to validate any information the user entered during the approval. This attribute can be specified as string, script, rule, call, or reference. Script is the default. You generally use a nested validationScript element instead of a validation argument.
Nested Tag within Approval Element	
AfterScript	Provides instructions for additional processing to be done on the item after the approval is complete, and only if approved. Often uses methods in the Approval Rule Library and LCM Workflow Rule Library. If those methods are to be used, the rule libraries must be explicitly included in the workflow using the <RuleLibraries> element. ParallelPoll and serialPoll items always execute this script after all responses are collected. With either of these modes, the logic in this script should aggregate the results and determine if the item should be approved or rejected. The business determines the criteria for approval or rejection, for example majority rule, any approval=approval, etc. In either poll mode, the AfterScript is inherited by child approvals if one is not specified.In other modes, child approvals do not inherit the after script.
InterceptorScript	This script is more complex than the AfterScript and is used less often. The script is called in several places in the approval processing: at the approval start, pre-Assimilation, post-Assimilation, when the work item is archived, and at the end of the approval. The stage of the processing is passed to the script as an argument called method that can be used to determine what the script should do at that time. The workflow context's args are also passed to the script. Method values for conditional analysis within InterceptorScript logic: • startApproval • preAssimilation • postAssimilation • archive • endApproval If an InterceptorScript and AfterScript exist, the InterceptorScript postAssim-

Approval Attribute	Purpose
	ilation logic launches before the AfterScript.
validationScript	Script to perform validation on the work item. For example, you can use this script to validate any data the user enters on the approval before the data is assimilated. This script is inherited by any child approvals.
Source	Nested within the AfterScript, InterceptorScript, and validationScript tags and contains the java BeanShell source for the script.
Arg	Arguments available to the approval action. Specified by string, script, rule, call, or reference. Most variables are passed to approval through send list. However, args that require any transformation must be sent through an Arg element. Additionally, the following args defined with reserved system names are passed through the Arg element with that name specified: • workItemRequester • workItemDescription • workItemType • workItemTargetId • workItemTargetName • workItemTargetClass • workItemDisableNotification • workItemNotificationTemplate • workItemEscalationTemplate • workItemReminderTemplate • workItemEscalationRule • workItemEscalationStyle • workItemHoursTillEscalation • workItemHoursBetweenReminder • workItemMaxReminders • workItemPriority • workItemIdentityRequestId • workItemArchive
Return	Return value defines how things should be assimilated from a work item back into the workflow case. This attribute is an alternative to the return attribute

Approval Attribute	Purpose
	CSV of variables. It is more complex and also more powerful. This attribute is rarely used in approvals. It is most often used when returning an approval work item variable to a workflow variable of a different name or when you need to transform the variable contents of a work item with a script. The use of these types of return elements follows the same rules as step returns from steps that subprocesses, with addition of local attribute options. See Step Elements.

The following basic approval step example presents an account change to the identity's manager for approval. The AfterScript records the approval decision and creates an audit record.

```
<RuleLibraries>
    <Reference class="sailpoint.object.Rule"  name="Approval Library"/>
    <Reference class="sailpoint.object.Rule"  name="LCM Workflow Library"/>
</RuleLibraries>

<Step icon="Approval" name="Manager Approval">
<Approval mode="serial" owner="script:getManagerName(identityName, launcher, fall-
backApprover);" renderer="lcmWorkItemRenderer.xhtml" send-
d="approvalSet,identityDisplayName,identityName,policyViolations">

<Arg name="workItemDescription" value="Manager Approval - Account Changes for User:
$(identityDisplayName)"/>
<Arg name="workItemNotificationTemplate" value="ref:managerEmailTemplate"/>
<Arg name="workItemRequester" value="$(launcher)"/>

<AfterScript>
        <Source>

          import sailpoint.workflow.IdentityRequestLibrary;
          assimilateWorkItemApprovalSet(wfcontext, item, approvalSet);
IdentityRequestLibrary.assimilateWorkItemApprovalSetToIdentityRequest(wfcontext,
approvalSet);
auditDecisions(item);

</Source>
      </AfterScript>

</Approval>

    <Description>
      If approvalScheme contains manager, send an approval for all
      requested items in the request. This approval will get the entire
      approvalSet as part of the workitem.
    </Description>

<Transition to="Build Owner ApprovalSet"
    when="script:isApprovalEnabled(approvalScheme, &amp;quot;owner&amp;quot;)"/>
<Transition to="Build Security Officer ApprovalSet"
```

```
    when="script:isApprovalEnabled(approvalScheme, &amp;quot;securityOfficer&amp;quot;)"/>
<Transition to="end"/>

  </Step>
```

In the AfterScript in this example, the methods not qualified by the library name are in the LCM Workflow Rule Library that is available to the workflow through the <RuleLibraries> declaration.

The assimilateWorkItemApprovalSetToIdentityRequest method is part of the IdentityRequestLibrary, this is available to the script through the import of that library in the script.

Library methods called through step action attributes are available through the workflow libraries attribute list,. However, when the library methods are executed from within scripts, the library must be specifically imported for the script.

Nested Approvals

Child approvals created through the user interface are expressed as nested approval elements in the XML.When nested approvals exist, the parent ceases to be an approval of its own.In those case, the sole purpose of the parent approval is to organize and contain the child approvals. The mode on the parent determines how to process the set of peer child approvals.

```
<Approval mode="string:parallel" name="Approve Region" owner="ref:regionApprover"
    send="identityName,region">
   <Arg name="workItemDescription" value="string:Approve Region for $(identityName)"/>
   <Approval name="childApproval1" owner="string:Walter.Henderson"
       send="identityName,region"/>
   <Approval name="childApproval2" owner="string:Alan.Bradley"
       send="identityName,region"/>
</Approval>
```

In the example above, childApproval1 and childApproval2 are processed in parallel. Because both of these child approvals are identical (no custom work item config and no children of their own), the same objective can be accomplished with a single approval with multiple owners:

```
<Approval mode="string:parallel" name="Approve Region" owner="ref:regionApprover"
    send="identityName,region">
   <Arg name="workItemDescription" value="string:Approve Region for $(identityName)"/>
   <Approval name="childApproval1" owner="string:Walter.Henderson"
       send="identityName,region"/>
   <Approval name="childApproval2" owner="string:Alan.Bradley"
       send="identityName,region"/>
</Approval>
```

Nested approvals can be used effectively when different approval levels are implemented with custom configurations and specifications. For example, the workItemConfig for each of the child approvals can be different, which can result in a notification scheme, escalation policy, etc. for the different approvers.

Nested approvals can be governed by a different approval mode from the one used on the master set and/or can contain their own child approval levels. One child approval can be done as an any approval, one that accepts the ruling of the first responder of several listed approvers, while the highest approval level is managed serially. Another child approval can implement custom workItemConfigs for its own child approvals. The example below illustrates all of these concepts.

Nested approvals can be used effectively when different approval levels are implemented with custom configurations and specifications. For example, the workItemConfig for each of the child approvals can be. The following example that illustrates all of these concepts.

```
    <!-- Approval submitted to HR and to supervisor and manager in serial manner -->
    <Approval mode="string:serial" name="Approve Region" owner="spadmin"
        send="identityName,region">
       <Arg name="workItemDescription" value="string:Approve Region for $(identityName)"/>

       <!-- HR Personnel approve region (whoever responds first makes decision) -->
       <Approval name="HRApproval" mode="string:any"
            owner="ref:HRApprovers" send="identityName,region"/>

       <!-- Supervisor and Manager approve region serially after HR approves -->
       <!-- Each has a different email template (work item config) for notification -->
       <Approval mode="string:serial" name="SupMgrApproval" send="identityName,region">
           <Approval name="Supervisor" send="identityName,region" owner="Tom.Jones">
             <WorkItemConfig escalationStyle="none">
                <NotificationEmailTemplateRef>
                   <Reference class="sailpoint.object.EmailTemplate"
                     name="SupervisorApprovalEmail"/>
                </NotificationEmailTemplateRef>
             </WorkItemConfig>
           </Approval>
           <Approval name="Manager" send="identityName,region" owner="Mary.Peterson">
             <WorkItemConfig escalationStyle="none">
                <NotificationEmailTemplateRef>
                   <Reference class="sailpoint.object.EmailTemplate"
name="ManagerApprovalEmail"/>
                </NotificationEmailTemplateRef>
             </WorkItemConfig>
           </Approval>
       </Approval>
    </Approval>
```

This ability to nest approvals, with options to assign different approval modes and work item configurations to each, enables implementers to create highly customized approval structures to meet the needs of the installation.

Workflow Library Methods

Workflow Libraries are sets of compiled java methods. To be accessible to workflows, these libraries must be specified as a comma separated list in the libraries attribute of the workflow element. The classes for libraries are named as follows: **SailPoint.workflow.[library]Library.class**. Only the [library] portion is specified in the libraries attribute.

The following example makes methods from the **SailPoint.workflow.IdentityLibrary.class** accessible to the workflow.

Example:

```
<Workflow libraries="Identity" explicitTransitions="true" name="Hello World Workflow"
type="IdentityUpdate">
```

> If no Libraries attribute is specified on the Workflow element, the workflow can access the Identity, Role, PolicyViolation, and LCM libraries by default.

The following table lists the workflow libraries and the methods available. Although the Standard Workflow Handler is not technically a library, the methods in it are accessible to every workflow and are called through the same syntax as library methods.

Standard Workflow Handler

Method / Usage	Description	Expected Args (Required Args are marked with a *)
Object getProperty(WorkflowContext wfc)	Returns value of the named system property.	name*
public Object isProperty (WorkflowContext wfc)	Returns true if the named system property has a value.	name*
public Object getMessage (WorkflowContext wfc)	Returns localized message for use in task results	• message* • type (severity) • arg1-arg4 (up to 4 parameters for the message)
public Object addMessage(WorkflowContext wfc)	Adds message to the workflow case.	• message* • type (optional severity) • arg1-arg4 (up to 4 parameters for the message)
public Object addLaunchMessage	Adds message to workflow case that is displayed in the user inter-	• message* • type (optional severity)

Method / Usage	Description	Expected Args (Required Args are marked with a *)
(WorkflowContext wfc)	face. Not kept in task result. For example, **Request was submitted**.	• arg1-arg4 (up to 4 parameters for the message)
public Object setLaunchMessage(WorkflowContext wfc)	Replaces previously added launch message with a new message based on new state.	• message* • type (optional severity) • arg1-arg4 (up to 4 parameters for the message)
public Object log(WorkflowContext wfc)	Sends something to log4j.	• message* • level*
public Object print(WorkflowContext wfc)	Prints text to the console.	message*
public Object audit(WorkflowContext wfc)	Creates an audit event. Enables workflows to put custom entries in audit log, which displays in the user interface.	• source* • action* • target • string1 - string4
public Object sendEmail (WorkflowContext wfc)	Sends an email message.	• to* • cc • bcc • from • subject • body • template* • templateVariables • sendImmediate • exceptionOnFailure
public Object launchTask (WorkflowContext wfc)	Launches a defined task.	• taskDefinition* • taskResult • sync (true=synchronous execution)
public Object sched-	Launches a generic event request.	• requestDefinition*

Method / Usage	Description	Expected Args (Required Args are marked with a *)
uleRequest(WorkflowContext wfc)		• requestName (name to assign to request) • scheduleDate • scheduleDelaySeconds • owner
public Object scheduleWorkflowEvent(WorkflowContext wfc)	Launches a workflow event request.	• requestName (name to assign to request) • scheduleDate • scheduleDelaySeconds • owner • workflow* (name of workflow to launch) • caseName (optional case name to override default)
public Object commit (WorkflowContext wfc)	Commits a transaction. Not commonly needed in workflows. Most commonly used for role approvals.	• creator • archive
public Object rollback (WorkflowContext wfc)	Rolls back a transaction. Not commonly needed in workflows. Most commonly used for role approvals.	none

Identity Library

Method / Usage	Description	Expected Args (Required Args are marked with a *)
public String getManager(WorkflowContext wfc)	Returns the name of the manager for the specified identity.	identityName
public Object calculateIdentityDifference(WorkflowContext wfc)	Derive a simplified representation of the changes made to an identity for an approval work item.	• oldRoles • newRoles • plan • approvalSet

Method / Usage	Description	Expected Args (Required Args are marked with a *)
private void addLinksInformation (WorkflowContext wfc)	Modifies workflow context lists of links (accounts) to be added, moved, or removed for the identity as a result of the provisioning plan.	• linksToAdd • linksToMove • linksToRemove • plan
public List<Map<String,Object>> checkPolicyViolations(Work-flowContext wfc) Evaluate policy violations that can be incurred by the provisioning plan/-project's actions	Evaluates policy violations that the provisioning plan/-project actions can incur.	• policies • identityName* • project • plan (either plan or project is required)
public void activateRoleAssignment (WorkflowContext wfc)	Assigns a role or roles to the identity.	• identity* (ID) • role* (ID) • detected (Boolean indicating if role was detected vs. assigned)
public void deac-tivateRoleAssignment(Work-flowContext wfc)	Removes role assignments from the identity.	• identity* (ID) • role* (ID) • detected (Boolean indicating if role was detected vs. assigned)
public void refreshIdentity(Work-flowContext wfc)	Performs an identity refresh on one identity.	• identity (ID) • identityName (either identity or iden-tityName is required)
public void refreshIdentities(Work-flowContext wfc)	Performs an identity refresh on a set of identities. Can specify one or more iden-tityNames, a filterString, or a list of roles. Processes the first of the above listed options that is non-null.	• identityName • identityNames (CSV) • filterString • identitiesWithRoles (CSV) • (any one of these 4 is required)

Method / Usage	Description	Expected Args (Required Args are marked with a *)
public Object compileProvisioningProject(WorkflowContext wfc)	Compiles a provisioning plan into provisioning project.	• plan • identityName
public Object buildProvisioningForm (WorkflowContext wfc)	Creates a form to display provisioning policy questions. When requiredOwner is passed as an argument, a form owned by this user is returned. If no more forms for this user exist, null is returned. When preferredOwner is passed as an argument, a form owned by this user is returned. If there are no remaining forms for that owner, a form owned by a different user can be returned.	• project* • template (name of form to serve as page template) • owner • preferredOwner (owner or preferredOwner required but mutually exclusive)
public Object assimilateProvisioningForm(WorkflowContext wfc)	Collects data from completed a provisioning form and stores answers with questions on provisioningProject.	• project* • form*
public Object assimilateAccountIdChanges(WorkflowContext wfc)	Updates ApprovalSet with any changes to accountIDs.	• project* • approvalSet
public Object buildPlanApprovalForm (WorkflowContext wfc)	Builds a form that represents all attributes in a provisioningPlan for an approval before the provisioning occurs.	• plan* • template
public Object assimilatePlanApprovalForm(WorkflowContext wfc)	Collects data from a form and puts the data back into the provisioningPlan. Assumes buildPlanApprovalForm.	• form • plan*
public Object provisionProject(Work-	Called by the Identity	• project*

Method / Usage	Description	Expected Args (Required Args are marked with a *)
flowContext wfc)	Update and LCM Workflows after provisioning forms are completed. Provisions the remaining items in the project.	• noTriggers (Boolean)
public Object finishRefresh(WorkflowContext wfc)	Called by the Identity Refresh workflow, after approvals are done and account completion attributes are gathered. Provisions what it can and completes the refresh process.	• identitizer • refreshOptions (map of args for creating new Identitizer if needed) • previousVersion • project
public Object buildApprovalSet(WorkflowContext wfc)	Called by the Lifecycle Manager workflows. Builds a simplified ApprovalSet representation of the items in the provisioning plan.	plan*
public Object processApprovalDecisions(WorkflowContext wfc)	Processes decisions made during approval process audit and react. Modifies the project masterPlan and recompiles the project if the recompile argument is set to true.	• project* • dontUpdatePlan • disableAudit • approvalSet* • recompile
public Object processPlanApprovalDecisions(WorkflowContext wfc)	Processes decisions made during approval process audit and modifies the Used before the plan is compiled into a provisioningProject.	• plan* • dontUpdatePlan • disableAudit • approvalSet*
public Object auditLCMStart(WorkflowContext wfc)	Creates an audit event to mark the start of an Lifecycle Manager workflow.	• approvalSet* • flow (name of applicable UI flow)
public Object auditLCMCompletion (WorkflowContext wfc)	Creates an audit event to mark the completion of anLifecycle Manager workflow.	• approvalSet* • flow
public void disableAllAccounts(Work-	Used by lifecycle events to	None

Method / Usage	Description	Expected Args (Required Args are marked with a *)
flowContext wfc)	disable managed accounts for the identity specified in the workflow.	
public void enableAllAccounts(WorkflowContext wfc)	Used by Lifecycle events to enable all accounts on the identity specified in the workflow.	None
public void deleteAllAccounts(WorkflowContext wfc)	Used by Lifecycle events to delete all accounts on the identity specified in the workflow.	None
public ProvisioningPlan buildEventPlan(WorkflowContext wfc)	Go through all links that the workflow's specified Identity hold and creates a plan to enable or disable all of the Identity's accounts. Specified by **op**.	op* (operation)
public void updatePasswordHistory (WorkflowContext wfc)	Adds a password to the link password history	plan*
public ProvisioningProject assembleRetryProject(WorkflowContext wfc)	Adds any account request for an original provisioning project that are retryable and then adds them to a new provisioning project. Rarely used in custom workflows.	project
public Object retryProvisionProject (WorkflowContext wfc)	Executes the retry provisioning project, created in assembleRetryProject. Rarely used in custom workflow.	project
public Object mergeRetryProjectResults(WorkflowContext wfc)	Merges results from the retry project onto the main project. Called between retries. Rarely used in custom workflow.	• project* • retryProject*

Method / Usage	Description	Expected Args (Required Args are marked with a *)
public Boolean requiresStatusCheck (WorkflowContext wfc)	Identifies if the project contains any Results that are queued with a requestID stored on the result.	project
public Object checkProvisioningStatus(WorkflowContext wfc)	Calls down to the connector for each Result in the plan that is marked queued with a requestID specified.	project
public Integer getProvisioningStatusCheckInterval(Workflow Context wfc)	Compute intervals between status checks for a request. The default is 60 minutes.	none
public Integer getProvisioningMaxStatusChecks(Workflow Context wfc)	Computes the maximum number of status checks permitted during a request. The default is infinite.	none
public Integer getProvisioningMaxRetries(WorkflowContext wfc)	Computes the maximum number of retries permitted during a request. The default is infinite.	none
public Integer getProvisioningRetryThreshold(WorkflowContext wfc)	Computes the retry threshold, the interval between retries, to use for a request. the Default is 60 minutes.	none

The methods are available for use. However these methods are rarely used in a custom workflow. It is recommended that custom workflows the workflow subprocesses instead of calling the library methods directly.

> This information is included for reference purposes and to document the purpose of the methods and what is passed to them. These explanations are also included to ensure that customizations do not remove calls to important methods from the subprocess workflows and to ensure that customizations only add other functionality around these method calls.

IdentityRequest Library

Method / Usage	Method / Usage	Expected Args (Required Args are marked with a *)
public Object createIdentityRequest(WorkflowContext wfc)	Create s an IdentityRequest object from current workflow context information. Tracks status and history of request processing.	• project* • flow • source • policyViolations
public Object updateIdentityRequestState (WorkflowContext wfc)	Modifies the IdentityRequest's state.	identityRequestId
public Object refreshIdentityRequestAfterApproval (WorkflowContext wfc)	Refreshes the IdentityRequest to include the provisioningEngine that processes the item. Updates the state and adds any expanded attributes that are provisioned.	project
public Object refreshIdentityRequestAfterProvisioning (WorkflowContext wfc)	After provisioning, copies interesting task result information back to the IdentityRequest.	project
public Object refreshIdentityRequestAfterRetry (WorkflowContext wfc)	Scans the retry project and updates the IdentityRequestItem retry count.	project
public Object completeIdentityRequest (WorkflowContext wfc)	Marks IdentityRequest status complete. Puts final plan in request and refreshes the request based on the final project.	• project • policyViolations • autoVerify (Boolean)

Approval Library

Method / Usage	Method / Usage	Expected Args
public SailPointObject getObject(WorkflowContext wfc)	Returns the object being approved.	none
public String getObjectClass (WorkflowContext wfc)	Returns the simple class name of the object being approved.	none
public String getObjectName (WorkflowContext wfc)	Returns the name of the object being approved.	none
public SailPointObject getCurrentObject(Work-	Returns the current persistent version of the object held in the workflowCase (approvalObject).	none

Method / Usage	Method / Usage	Expected Args
flowContext wfc)		
public Identity getObjectOwner(WorkflowContext wfc)	Returns the current owner of the object being approved. Uses database lookup.	none
public Identity getNewObjectOwner(WorkflowContext wfc)	Returns the object owner. In the workflow context, the owner could be different than the database-recorded owner.	none
public String getObjectOwnerName(WorkflowContext wfc)	Returns name of ObjectOwner from getObjectOwner.	none
public String getNewObjectOwnerName(WorkflowContext wfc)	Returns name of NewObjectOwner from getNewObjectOwner.	none
public boolean isOwnerChange(WorkflowContext wfc)	Return true if object being approved has had an owner change.	none
public boolean isSelfApproval(WorkflowContext wfc)	Returns **True** if the user who launches workflow is the same as the owner of the object being approved. Used to bypass an owner approval. Assumes that the user will approve if the user is the one who is initiating the request.	none

Policy Violation Library

Method / Usage	Method / Usage	Expected Args (Required Args are marked with a *)
public Object delete (WorkflowContext wfc)	Deletes the current approval object associated with this workflow.	none
public Object ignore (WorkflowContext wfc)	Ends the workflow associated with the current approval object without performing any actions.	none
public Object mitigateViolation(WorkflowContext wfc)	Mitigates by temporarily allowing a policy violation.	• expiration* • comments
public Object getRemediatables(WorkflowContext wfc)		none

Method / Usage	Method / Usage	Expected Args (Required Args are marked with a *)
public Object getRemediatables(WorkflowContext wfc)		• remediator • actor • comments • remediations* Use if remediator argument is not specified and actor is. Use remediator in new method calls.

Role Library

Method / Usage	Method / Usage	Expected Args (Required Args are marked with a *)
public Object launchImpactAnalysis(WorkflowContext wfc)	Starts an impact analysis task for a role in workflow.	none
public Object getRoleDifferences (WorkflowContext wfc)	Calculates the differences between a role held in workflow and the database version of the role.	none
public Object auditRoleDifferences(WorkflowContext wfc)	Creates one audit event for each attribute difference between role states. Compares workflow vs database.	• source • action • target • string1
public Approval buildOwnerApproval(WorkflowContext wfc)	Sets up an approval for the owner of an object. Currently used only for roles.	none
public List<Approval> buildApplicationApprovals(WorkflowContext wfc)	For role approvals only. Builds an approval structure for the owners of each application referenced in the role profiles. Normally processed as parallelPoll to allow application owners to submit comments or modify the role without terminating the approval process.	none
public void enableRole(WorkflowContext wfc)	Marks role as enabled.	role (name)

Method / Usage	Method / Usage	Expected Args (Required Args are marked with a *)
public void disableRole(WorkflowContext wfc)	Marks role as disabled.	role (name)
public void setRoleDisabledStatus(WorkflowContext wfc)	Marks role with disabled status indicated in the disabled arg. true = disabled false = enabled	• role (name) • disable (Boolean)
public void removeOrphanedRoleRequests (WorkflowContext wfc)	Removes incomplete requests. Used to activate/deactivate roles that no longer exist.	none
public String getApprovalAuditAction(WorkflowContext wfc)	Called by the post-approval audit steps, Audit Failure and Audit Success, of Role Modeler. Owner Approval workflow to determine what type of action should be recorded in audit log. If the role is marked as disabled, returns disableRole. if the role is NOT marked as disabled, returns updateRole .	none

LCM Library

Currently, the Lifecycle Manager Library contains no public methods. All of its methods were moved to the Standard Workflow Handler.

Monitoring Workflows

After a workflow is initiated, the workflow can launch to completion quickly. Sometimes a workflow can take additional time to complete its specified actions. Approval steps often create a delay in the processing while the workflow waits for the approver to review the work item and make a decision on it.

To observe a workflow in flight and understand how much of the process is complete and what actions are pending, you can examine the Task Result for the workflow on the **Setup > Tasks > Task Results** page. The TaskResult for a workflow exists for a period of time following the successful completion of the workflow. Based on the retention period set, the TaskResult can be purged soon after the process launches to completion. While the workflow is still in progress, the TaskResult continues to exist and can be examined for current step and status information.

Viewing the Workflow Case XML

You can examine the workflow case in XML format from the IdentityIQ console or from the Debug pages.The status of each step can then be determined from the data recorded in the workflow case.

To get the **workflowcase** XML from the console:

1. Launch the console.
2. List the workflow cases.
3. Get the specific workflow case in question by name. See the following example.

```
IIQ console
> List workflowcase
[system will list all in-flight workflowcases by ID and name]
> get workflowcase "[workflowcase name]"[system will display the XML for the workflow
case]
```

To view the workflowcase XML from the IdentityIQ Debug pages:

1. Select WorkflowCase from the object list.
2. Click the specific workflow case from the list to display its XML.

Advanced Workflow Topics

This section includes these advanced Workflow topics:

- Loops within Workflows
- Launching Workflows from a Task or Workflow
- Workflow Forms

Loops within Workflows

A loop occurs when a step transitions back to a step that executed previously. The state of that step is re-initialized and the step is executed again. A loop can transition back any number of steps. You define a loop transition the same way you would any other transition. However, you must just select a target step that appears before the loop transition in the process designer.

In most case, when you create a loop transition, you want to give it a conditional *When* expression. If a loop transition is unconditional, the workflow can enter an infinite loop and not be able to finish.

Launching Workflows from a Task or Workflow

You can launch workflows from tasks or other workflows without using a system event to trigger the workflow.

Workflows Launched from Custom Tasks

You can launch workflows from a custom task in IdentityIQ. Because tasks are compiled java classes, the custom task must be written as a Java method.

To create a workflow from a custom task:

1. Create a WorkflowLaunch object in the Java method.
2. Populate the object with the data the workflow requires.
3. Use the *Workflower* class to launch the workflow.

It is often necessary for one workflow to launch another workflow. This can be performed in Beanshell using code similar to the previous example. However, using the workflow library method scheduleWorkflowEvent is easier. Not only does this method launch a workflow, it also allows you to delay the launch until a time in the future.

To have one workflow to launch another workflow, create a step and select scheduleWorkflowEvent as the action. This method requires the following arguments:

```
import java.util.HashMap;
import sailpoint.api.sailpointContext;
import sailpoint.api.Workflower;
import sailpoint.integration.ProvisioningPlan;
import sailpoint.integration.ProvisioningPlan.AccountRequest;
import sailpoint.integration.ProvisioningPlan.AttributeRequest;
import sailpoint.object.Identity;
import sailpoint.object.Workflow;
import sailpoint.object.WorkflowLaunch;
import sailpoint.tools.GeneralException;
```

```
                import sailpoint.tools.xml.XMLObjectFactory;


                HashMap launchArgsMap = new HashMap();

                String myIdentityName = "T339222";
               Identity myIdentity = context.getObjectByName(Identity.class, myIden-
tityName);

                //Create Provisioning Plan and add needed attribute values
                ProvisioningPlan plan = new ProvisioningPlan();
                plan.setIdentity(myIdentity);
                AccountRequest accountRequest = new AccountRequest();
                AttributeRequest attributeRequest = new AttributeRequest();

                accountRequest.setApplication("IIQ");
                accountRequest.setNativeIdentity(wbIdentity);
                accountRequest.setOperation("Modify");

                attributeRequest.setOperation("Add");
                attributeRequest.setName("assignedRoles");
                attributeRequest.setValue("Benefits Clerk");

                accountRequest.add(attributeRequest);
                plan.add(accountRequest);

                //Add needed Workflow Launch Variables to map of name/value pairs
                launchArgsMap.put("allowRequestsWithViolations","true");
                launchArgsMap.put("approvalMode","parallelPoll");
                launchArgsMap.put("approvalScheme","worldbank");
                launchArgsMap.put("approvalSet","");
                launchArgsMap.put("doRefresh","");
                launchArgsMap.put("enableRetryRequest","false");
                launchArgsMap.put("fallbackApprover","admin");
                launchArgsMap.put("flow","RolesRequest");
                launchArgsMap.put("foregroundProvisioning","true");
                launchArgsMap.put("identityDisplayName","John.Smith");
                launchArgsMap.put("identityName","John.Smith");
                launchArgsMap.put("identityRequestId","");
                launchArgsMap.put("launcher","admin");
                launchArgsMap.put("notificationScheme","user,requester");
                launchArgsMap.put("optimisticProvisioning","false");
                launchArgsMap.put("plan",plan);
                launchArgsMap.put("policiesToCheck","");
                launchArgsMap.put("policyScheme","continue");
                launchArgsMap.put("policyViolations","");
                launchArgsMap.put("project","");
                launchArgsMap.put("requireViolationReviewComments","true");
                launchArgsMap.put("securityOfficerName","");
                launchArgsMap.put("sessionOwner","admin");
                launchArgsMap.put("source","LCM");
                launchArgsMap.put("trace","true");
                launchArgsMap.put("violationReviewDecision","");
                launchArgsMap.put("workItemComments","");
```

```
                sailpoint.object.ProvisioningPlan spPlan = new sail-
point.object.ProvisioningPlan();
                spPlan.fromMap(plan.toMap());
                launchArgsMap.put("plan", spPlan);

                //Create WorkflowLaunch and set values
                WorkflowLaunch wflaunch = new WorkflowLaunch();
                Workflow wf = (Workflow) context.getObjectByName(Work-
flow.class,"myWorkflowName");
                wflaunch.setWorkflowName(wf.getName());
                wflaunch.setWorkflowRef(wf.getName());
                wflaunch.setCaseName("LCM Provisioning");
                wflaunch.setVariables(launchArgsMap);


                //Create Workflower and launch workflow from WorkflowLaunch
                Workflower workflower = new Workflower(context);
                WorkflowLaunch launch = workflower.launch(wflaunch);

                // print workflowcase ID (example only; might not want to do this in the
task)
                String workFlowId = launch.getWorkflowCase().getId();
                System.out.println("workFlowId: "+workFlowId);
```

Workflows Launched by Other Workflows

Installations often have one workflow start another workflow using the scheduleWorkflowEvent method in the Standard Workflow Handler. One of the initiating workflow steps launches the method through a call action.

Arguments to the step include the following:

Name	Value
workflow	Name of the workflow to launch.
requestName	Name to be assigned to the request. If not specified, the name of workflow is the default.
scheduleDate	Date and time you want the workflow to launch. Must be specified with a java.util.Date value. If this argument is not set, the workflow launches immediately.
scheduleDelaySeconds	An alternative to using scheduleDate. When set, the value is the number of seconds to delay before launching the workflow.
caseName	Specify a user friendly name for workflowCase to be displayed in the user interface. If no name is specified, the default is the name of workflow.
launcher	Name of the identity to be displayed as the *launcher* of the new workflow case. If this argument is not specified, the launcher of the initiating workflow is used.

A workflow that is launched by another workflow is different from a workflow that is launched as a subprocess. If a workflow is launched as a subprocess, the calling workflow waits until the subprocess is completed. After the workflow returns control to the caller, the processing continues.

A workflow that is launched by another workflow causes a completely separate workflow to begin launching. After the new workflow is started, the original or calling workflow moves on to its next step.

Workflow Forms

Standard work item forms are available for presenting approval or other data requests to approvers. However, some installations prefer to use custom forms for these activities. Based on the type of the data collection effort, a custom form might be required. You can build a custom form using a <Form> element in the XML that is embedded within the <Approval> element.

> The <Approval> element can be used to collect data from a user, even if the workflow is not an approval. You generally use custom forms for these activities because the normal approval forms do not apply. However, you can also use custom forms for traditional approval activities when you need a different presentation format.

The basic elements in a Form definition are:

```
<Form>
   <Attributes>    (map of name/value pairs that influence the form renderer)
   <Button>     (determine form processing actions)
   <Section>    (Subdivision of form; may contain nested Sections and Fields)
     <Field>    (may contain Attributes map, Script to set value, Allowed Values Defin-
ition script, and Validation Script)
```

For detailed information about working with forms, see the **Forms** documentation.

Process Variable and Step Forms

You use forms added to steps on the **Process Designer** tab in the Business Process Editor to request data that a process needs from a user. For example, you can add a form to a step to request a value for a missing attribute.

However, to present information on the Basic Views of the **Process Variables** tab and the **Arguments** tab of the Step Editor, you use process variable and step forms.

To simplify the information displayed on the Process Variables tab:

- Variables are displayed in more logical groups.
- Variables that are rarely, if ever, modified are hidden.

Changes made in the Basic View are persisted to the Advanced View and more complex configuration can be performed there if needed.

The step forms are referenced from the workflows or stepLibraries. These forms define the form that is presented on the **Arguments** tab of the Step Editor panel and works similar to the process variable forms.

Both of these forms are referenced from workflows using the configForm variable. The forms can be defined, viewed and edited on the IdentityIQ debug page.