



Defining and Modifying IdentityIQ Workflows

This document describes the process for creating and editing workflows in IdentityIQ. It covers the use of the IdentityIQ UI's business process designer, editing or creating workflows directly through their XML, and a number of advanced topics related to workflows (such as custom forms and running workflows from tasks).

Table of Contents

Introduction	5
Workflow Basics	6
Terminology	6
Important Workflow Objects.....	6
Workflows Operation	6
Provisioning Plans in Workflows.....	7
Triggering Workflows	7
IdentityIQ’s Default Workflows	8
Workflow Types.....	9
Creating Workflows through the IdentityIQ UI	10
Process Editor Interface Overview	10
Process Details Tab.....	11
Process Variables Tab	12
Variable Initialization.....	12
Timing of Variable Definition.....	13
Process Designer Tab	13
Process Steps.....	13
Script	15
Rule	15
Subprocess	16
Call.....	16
Step Arguments	17
Return Variables	18
More on Start and Stop Steps.....	18
Step Icons	18
Approval Steps	19
Approval Details	20
Approval Arguments.....	21
Work Item Configuration.....	22
More on Approval Modes.....	22
Child Approvals.....	23

Step Transitions.....	25
Process Metrics tab	26
Editing Workflow XML.....	28
Accessing the XML.....	28
Debug Pages	28
IIQ Console	29
Re-importing the XML	30
Dollar-Sign Reference Syntax	30
XML Content.....	30
Header Elements	31
Workflow Element.....	31
Variable Definitions	31
Initializer Options	32
Workflow Description.....	33
Rule Libraries	34
Step Elements.....	34
Transition Element	35
Step Actions.....	36
Arguments	37
Return Elements	38
Call.....	39
Wait Attribute	40
“Catches” attribute.....	40
Approval Steps	40
Nested Approvals	44
Workflow Library Methods	45
Standard Workflow Handler.....	45
Identity Library	47
IdentityRequest Library	50
Approval Library	51
Policy Violation Library	51
Role Library	52

LCM Library	52
Monitoring Workflows	53
Viewing the Workflow Case XML.....	53
Advanced Workflow Topics	56
Loops within Workflows	56
Initiating Workflows from a Task or Workflow.....	56
Workflows Run from Custom Tasks.....	56
Workflows Run by Other Workflows	58
Custom Forms	59
Attributes	59
Buttons	59
Sections	60
Fields	60
Example of Custom Form XML	62

Introduction

A workflow is a sequence of operations or steps executed to perform work. The IdentityIQ Workflows – both the standard “out of the box” workflows and custom workflows written for a specific installation -- are triggered by system events. The available triggering events include:

- Role creation
- Identity update
- Identity refresh
- Identity correlation
- Deferred role assignment/deassignment
- 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 and can make use of the product’s workflow library methods and rules as well as custom java scripts and rules. Workflow customization and construction usually involve a combination of XML and Java programming. Some customization activities can be managed through the graphical process editor included in the product, but typically, implementers involved in customizing or creating new workflows will need to be comfortable writing both XML and Java.

This document is divided into four chapters, each covering key subjects related to Workflows:

1. Workflow Basics
2. Creating Workflows through the UI
3. Editing Workflow XML
4. Advanced Workflow Topics

Workflow Basics

This section contains some key concepts that are important to understanding the usage and development of workflows.

Terminology

In IdentityIQ and throughout this document, the terms Business Process and Workflow are used synonymously. The IdentityIQ user interface refers to these sets of connected actions as Business Processes -- the term most often used by business managers. Behind the scenes, in the IdentityIQ object model and XML, they are called Workflows; they control the flow of data through the required processing.

Important Workflow Objects

There are four key objects in the IdentityIQ Object Model that are used in workflows. A basic understanding of these objects is important to fully comprehending workflows. Of these, the one most critical to writing workflows is the WorkflowContext, since it tracks the runtime state of the workflow and is passed to the Workflow Handler and all rules, scripts, and library methods used in workflows; this means data can be extracted from it as needed within any step of the workflow.

Object	Usage
Workflow	Defines the workflow structure and steps involved in the workflow processing
WorkflowCase	Represents a workflow in flight; contains a Workflow element in which the process is outlined and current state data is tracked, as well as identifying information about the workflow's target object
WorkflowContext	Runtime information maintained by Workfower 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
Task Result	Records the completion status of a task, or in this case, the workflow; contained within the WorkflowCase

Workflows Operation

Workflows carry out a sequence of defined actions based on a triggering event, and they can be used to accomplish a wide variety of activities within the system. However, it is important to remember that in its running state, a workflow is tracked through a workflow case, which manages only one target entity at a time: one Identity, one Role, etc. If multiple Identities are modified at one time in a way that requires a workflow to run for all of them, a separate workflow case is created for each to track the processing of the workflow for that single Identity.

Provisioning Plans in Workflows

A Provisioning Plan contains a list of requested changes to an individual Identity. Just as a workflow case can address only one Identity at a time, it can also reference only one provisioning plan at a time. If changes are requested for more than one Identity at a time, even if the same change is requested for all the Identities, a separate provisioning plan will be created for each Identity and each of the provisioning plans will be managed individually by the separate workflow cases created to manage each identity's workflow.

It is a common mistake for developers of custom workflows to attempt to access more than one provisioning plan in a single workflow; this will not work because of the design of the underlying workflow engine.

Triggering Workflows

Workflows are triggered by events that occur in other parts of IdentityIQ. Many are invoked by Lifecycle Manager actions, where changes to an Identity's roles, entitlements, or accounts are requested. Others are triggered by Lifecycle Events, when Identities are created, deactivated, moved from one manager to another, or otherwise modified in a way that is configured to trigger a workflow. Some non-lifecycle events can also activate workflows.

There are three windows in the IdentityIQ application interface where workflows are associated to system activities.

- 1) The workflows invoked by Lifecycle Manager requests are specified in the **Lifecycle Manager Configuration** window's **Business Processes** tab (accessible through the menu options (**System Setup** -> **Lifecycle Manager Configuration**)).

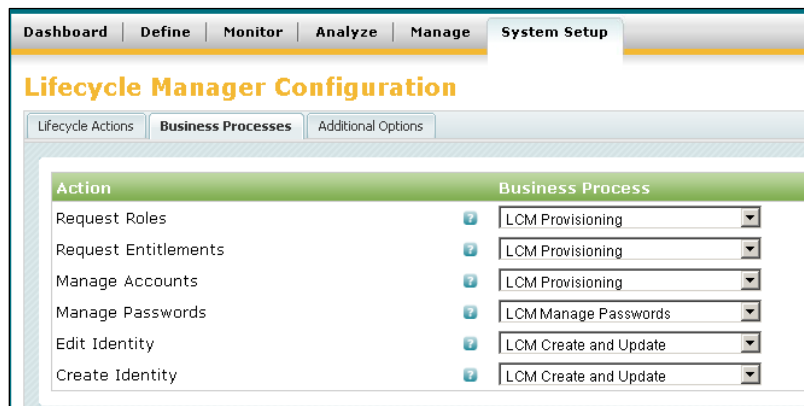


Figure 1: Lifecycle Manager Workflow Configuration

2) Workflows invoked by Lifecycle Events are specified on the **Lifecycle Event** definition itself:

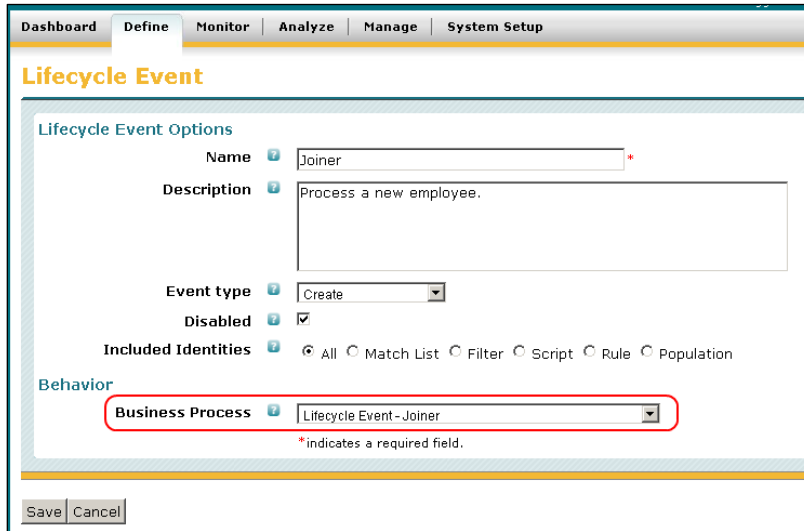


Figure 2: Lifecycle Event Workflow Specification

3) Non-LCM-related workflows are linked to triggering events on the **Configure IdentityIQ Settings** page's **Business Processes** tab (menu option **System Setup** -> **IdentityIQ Configuration**):

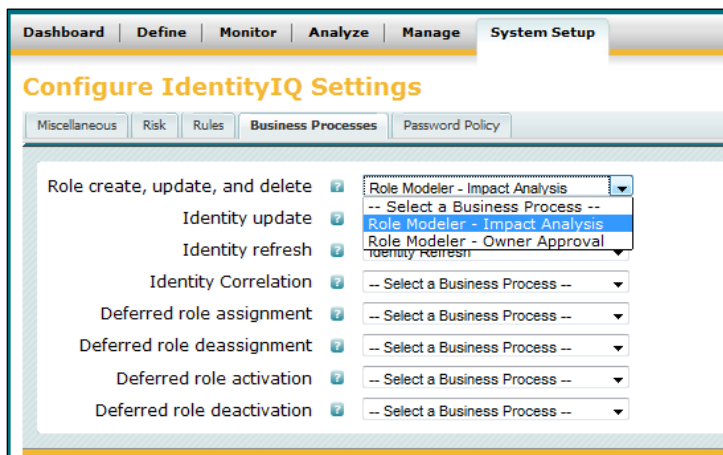


Figure 3: Non-LCM Workflow Selection

It is also possible to configure an IdentityIQ task to kick off a workflow. This is a slightly more complex undertaking that is described in the

Advanced Workflow Topics section at the end of this document.

IdentityIQ's Default Workflows

Out of the box, IdentityIQ is preconfigured with 25-30 workflows that manage activities such as provisioning of roles or entitlements, account management, identity creation, and password management. The default workflows can be configured and customized to address the specific business requirements of each installation. Additionally, new workflows can be written and applied to any of the actions in IdentityIQ that execute workflows.

Workflow Types

The default workflows fall into several pre-defined workflow types; the assigned type is used to determine which workflows to present in the configuration list boxes when workflows are specified for triggering based on a specific system event. For example, Role create, update, and delete actions can trigger workflows of type RoleModeler so only workflows of that type are listed in the drop-down list for that configuration option.



Figure 4: Type filters on Workflow selection lists

NOTE: Workflow can be assigned custom Types but will then only be selectable for triggering through the UI on Lifecycle Events; Lifecycle Events can trigger workflows of any type.

The table below indicates the Workflow Type invoked as a result of each type of action within IdentityIQ.

System Activity	Associated Workflow Type
Role create, update, and delete	RoleModeler
Identity update	IdentityUpdate
Identity refresh	IdentityRefresh
Identity correlation	IdentityCorrelation
Deferred role assignment Deferred role de-assignment	DeferredRoleAssignment
Deferred role activation Deferred role deactivation	DeferredRoleActivation
Request Roles Request Entitlements Manage Accounts Manage Passwords	LCM Provisioning
Edit Identity Create Identity	IdentityUpdate

Lifecycle Event	Any (Lifecycle Event Business Process selection list does not filter on type)
-----------------	---

Some complex workflows are subdivided into multiple “sub-process” workflows that are invoked by a master workflow. This simplifies the structure of the master workflow and makes workflows easier to manage. It also promotes reusability, since more than one master workflow can invoke the same sub-processes. As a standard practice, these smaller workflows are assigned the type “Subprocesses”. Although this type is not associated with any system functionality, this type designation allows a reader to identify the workflow as a sub-process of a larger workflow at a glance.

Creating Workflows through the IdentityIQ UI

IdentityIQ’s user interface provides a graphical tool for defining Workflows, which includes creating the steps that comprise the required actions and outlining the transitions between those steps. This graphical approach can be very helpful in quickly setting up the desired structure. It also provides a user-friendly representation of the process flow that can be useful in creating documentation about the activities included in the workflow.

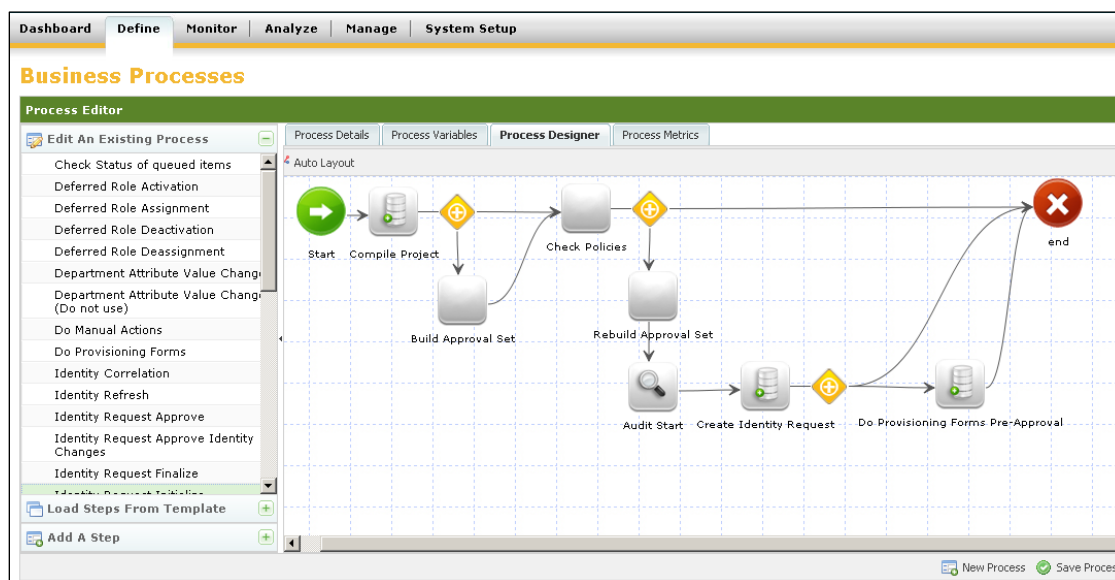


Figure 5: Graphical Process Designer

Commonly, implementers use the graphical editor to outline the process and then move to the XML representation to flesh out or adjust the details of each step. Once the process has been saved, its XML representation can be viewed, edited, and exported from the IdentityIQ Debug pages.

NOTE: Some steps in workflow development may not be possible through the UI. It should be assumed that most workflows will require direct editing in the XML representation, as well as some amount of Java coding. Workflow development generally cannot be completed without an understanding of the syntax of both XML and Java.

Process Editor Interface Overview

The business process editor user interface makes it easy to create a basic workflow. The interface contains four tabs: **Process Details**, **Process Variables**, **Process Designer**, and **Process Metrics**; the functionality of each of these is listed in the table below.

Interface Tab	Purpose
Process Details	Specify Name, Type, and Description of the workflow
Process Variables	Specify any variables that apply to the workflow; this includes any input variables, return values, and working variables for use within the process's steps
Process Designer	Graphically represent the process, specify the actions involved in each step, and provide the evaluation conditions for moving from one step to another (transitions)
Process Metrics	Review statistics gathered for the process as it executes.

To access the **Business Processes** Process Editor, navigate to **Define -> Business Processes**. Select an existing workflow from the **Edit an Existing Process** list to view or modify it, or click **New Process** to create a new Workflow.

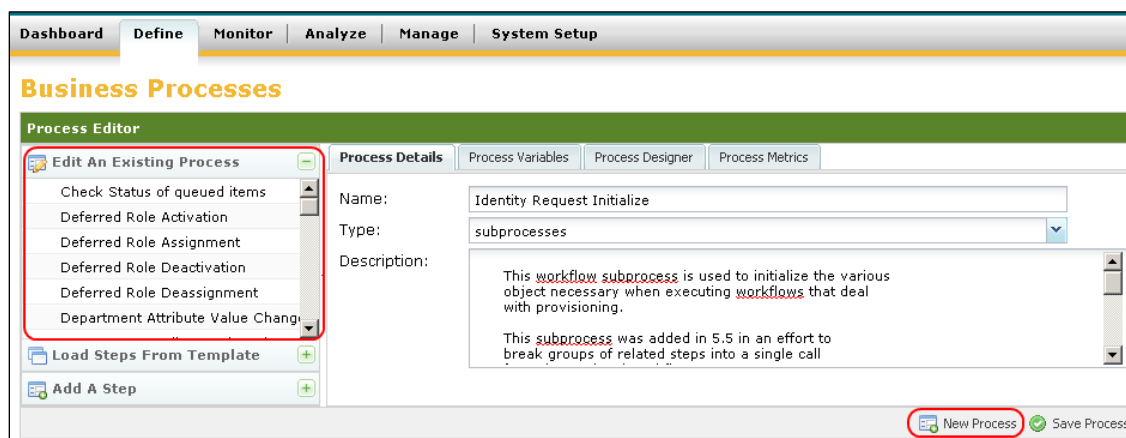


Figure 6: Edit or Create a Business Process

Navigate through each of the process tabs to view or modify the workflow's data, as specified in the next sections.

Process Details Tab

On the **Process Details** tab, specify the **Name**, **Type**, and **Description** for the workflow.

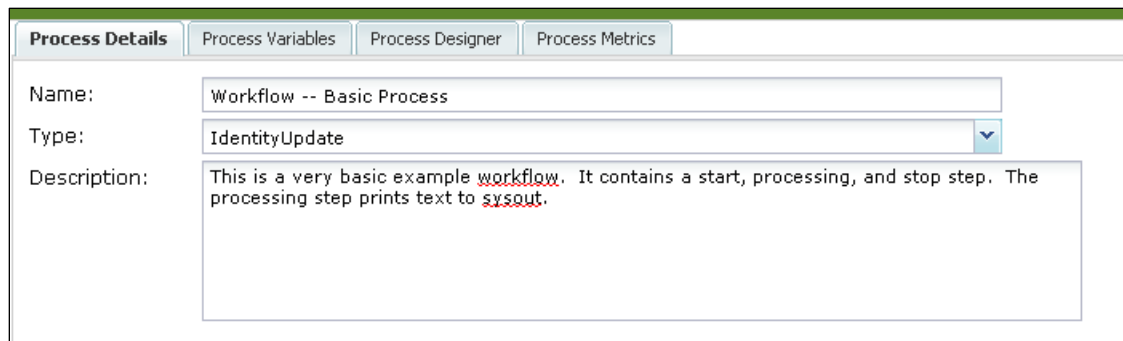


Figure 7: Process Details tab

The **Name** should be a short descriptive name for the workflow, and the **Description** should provide an overview of what the workflow does. The **Type** field should generally be selected from the drop-down list of predefined workflow types but can be a custom type (subject to the limitations described in *Workflow Types* in the Workflow Basics chapter). To enter a custom type, enter the desired type name in the box instead of selecting one from the list.

Process Variables Tab

The Process Variables tab lists variables available to the workflow. These include input and output variable for the workflow, as well as any working variable used in the workflow’s processing.

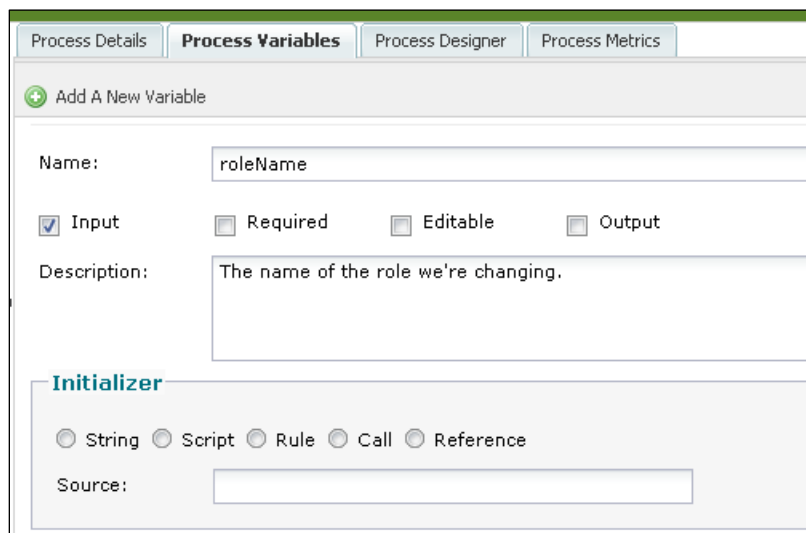


Figure 8: Process Variables tab

Variables can be marked as **Input**, **Required**, **Editable**, and/or **Output**. Those flags’ impacts on the variables are described in this table.

Flag	Purpose
Input	Specifies that the variable is one of the arguments to the workflow, passed in when it is invoked.
Required	Indicates that the variable must contain a value (non-null) when the workflow starts

Editable	Allows the variable to be modified by steps in the workflow
Output	Stores the variable in the workflow's TaskResults Output Variables list; TaskResults is used for recording the progression status and results of the workflow (viewable through the Manage -> Tasks -> Task Results window)

NOTE: As variables are created through the UI, new ones 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 – the *opposite* of the order they are displayed in the UI. If any variables in the XML reference other variables in their initializations, the referenced variable must be declared first, so the order of variable declarations may matter.

Variable Initialization

Variables can be initialized for the workflow by specifying an Initializer for them on this window. The recommended best practice is to use initializers for the workflow variables, rather than creating multiple process steps to initialize each variable.

There are five options for how initialization can occur:

Initialization Type	Description
String	Assigns a literal value to the variable
Script	Sets the variable value through a segment of Java beanshell code
Rule	Assigns the return value of a workflow Rule (Java beanshell code) to the variable
Call	Assigns the return value of a call to a compiled Java method in a workflow library to the variable
Reference	Sets the variable value through a reference to one of the workflow's other process variables (sets to same value as referenced variable)

NOTE: Variable values passed into the workflow through workflow arguments supersede variable initializations, so any value provided in an argument will cause the initializer for that argument variable to be ignored.

Timing of Variable Definition

Some variables will be known at the beginning of workflow development and can be defined even before the graphical process design occurs. Others may come up throughout the development process. The UI does not restrict variable usage during development to only those that have been previously defined on the Process Variables tab, so variable definition can be done before, during, or after the design process.

Process Designer Tab

The bulk of the work in creating a workflow is done on the **Process Designer** tab. The workflow's activities are determined by the Steps and Transitions created for it.

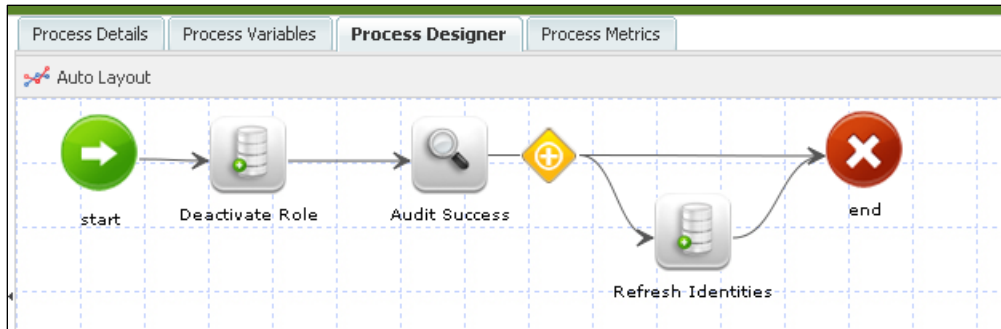


Figure 9: Process Designer

Process Steps

At its very simplest, a workflow involves a minimum of 3 Steps: a Start step, a processing step, and a Stop step (also often called “End”). Though IdentityIQ does not require it, the recommended best practice is for all workflows to contain a Start and Stop step and for these two steps to contain no “action.” Workflows may contain as many or as few processing steps as are necessary to manage the required actions. Steps are added in the Process Designer by clicking the desired step type in the **Add A Step** section of the Process Editor window. The steps can be dragged around the Process Designer grid to line them up visually according to a logical progression.

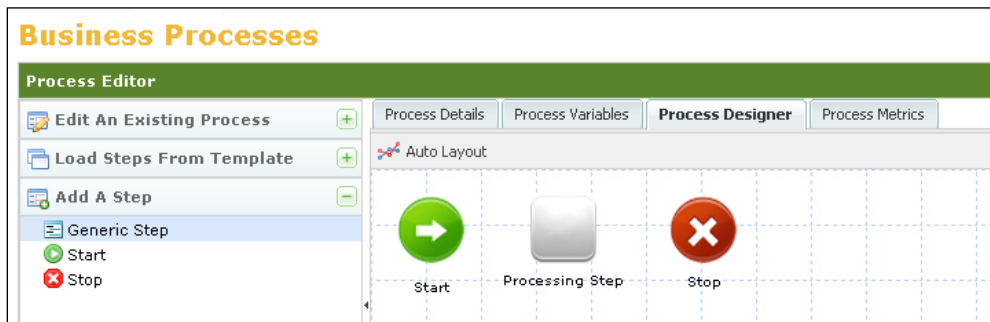


Figure 10: Adding Steps to a Process

The contents of a step can be edited by right-clicking the step and clicking **Edit Step**.

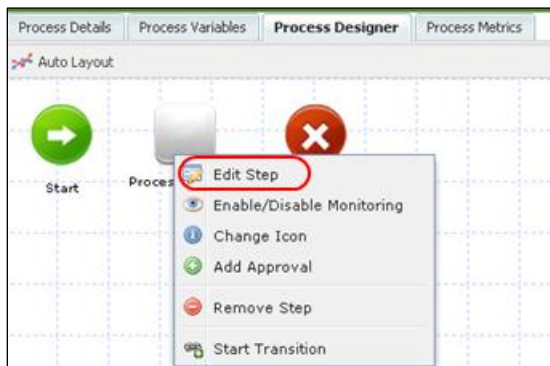


Figure 11: Edit a Process Step

This opens the step details window. Here, the **Name** and **Description** of the step can be recorded; its **Result Variable** (return value), if any, can be named; and its **Action** can be specified.

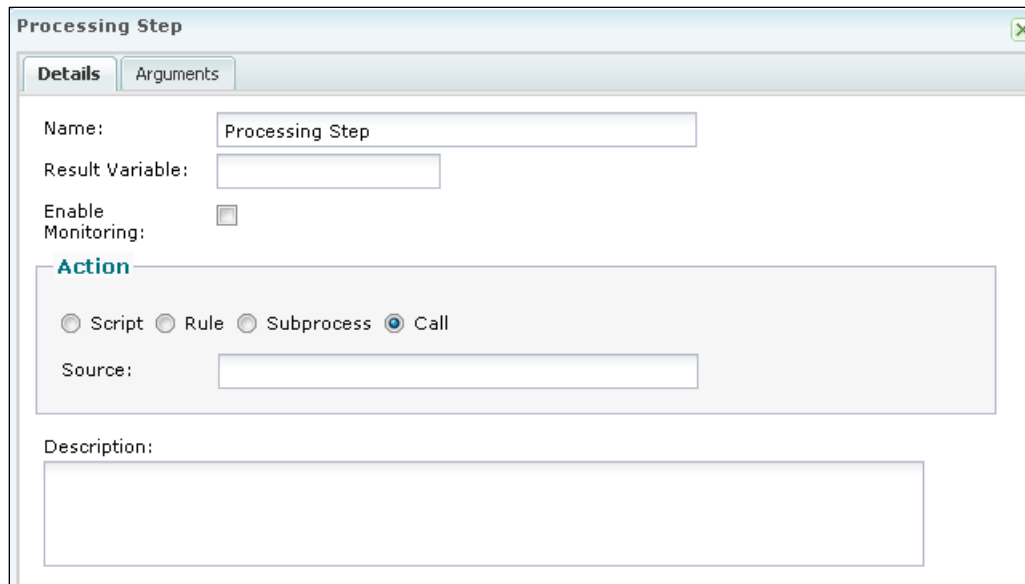


Figure 12: Step Details window

Each step can take one of four types of action:

Action	Description
Script	Execute a segment of Java beanshell code that is included in the step
Rule	Execute a workflow Rule – a block of Java beanshell code encapsulated in a reusable rule
Subprocess	Invoke another defined workflow, passing control to it until it completes
Call	Call a compiled java method in the IdentityIQ workflow library, exposed through the standard workflow handler

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 must be written by the implementer to execute a desired action. They are written directly in the **Source** box on the step’s detail window.

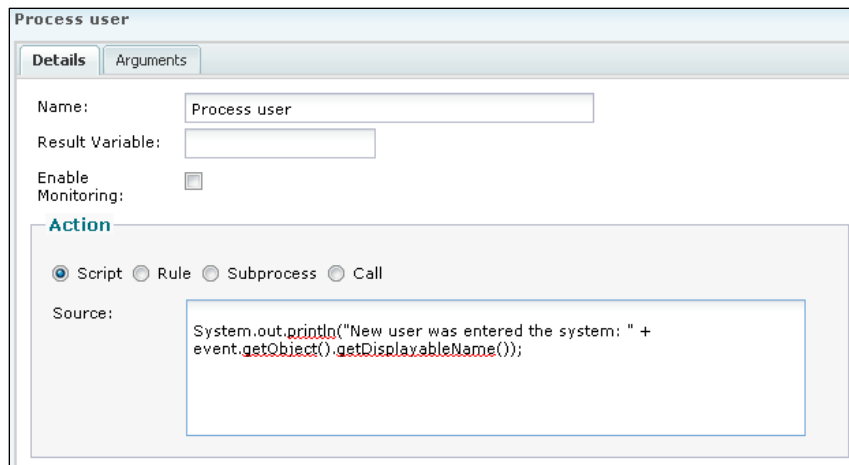



Figure 13: Step containing a Script

NOTE: The script examples in this document all show very short java beanshell code blocks, but there is no set length for a script; a script block within the XML can be as short or long as necessary to accomplish the required processing.

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 invoke a rule of the same type. Rules created through this window are of type "Workflow" and can be used by any workflow. When **Rule** is chosen as the **Action**, an existing workflow rule can be selected from the list or a new rule can be written in the rule editor (opened by clicking the  button).

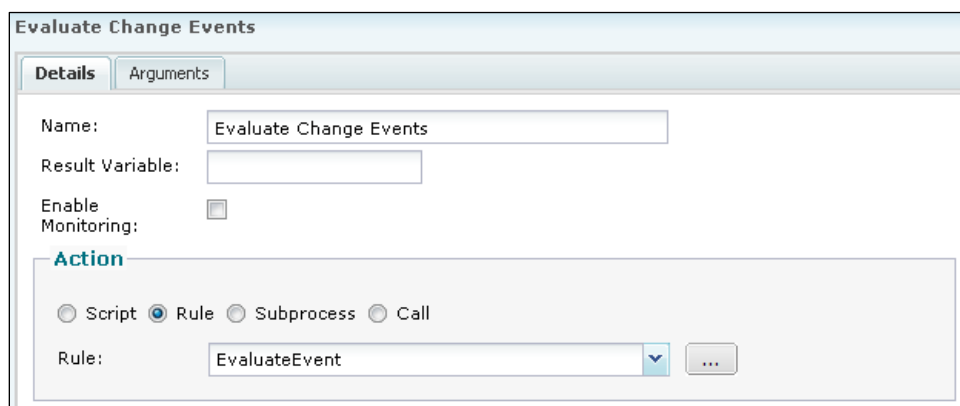


Figure 14: Step executing a Rule

Subprocess

Subprocesses are other workflows. These are used to subdivide complex processes into smaller segments that are more easily managed and can be potentially reused by other workflows. Subprocesses are complete workflows in themselves, so they contain a Start step, a Stop step, and as many processing steps as are needed to complete their activities.

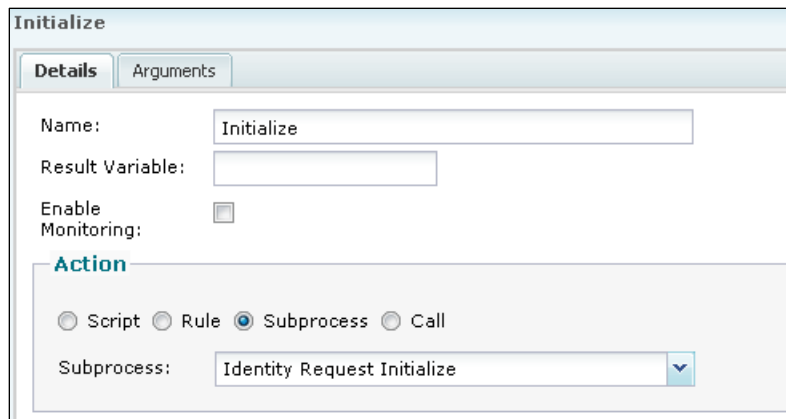


Figure 15: Step invoking a Subprocess

Call

The IdentityIQ workflow library contains a set of methods that can be executed from within a workflow; these are exposed through the standard workflow handler, which is called by the workflow engine every time something happens in a workflow. Every workflow has access to the methods in the standard workflow handler. Additional libraries of methods are also available for use in workflows. When no Library list is specified for the workflow, the default is to include access to the Identity, Role, PolicyViolation, and LCM libraries.

NOTE: Through the XML, other libraries, including custom libraries for an installation, can be specified; the UI does not offer an option for managing the library list. Specifying a library list overrides the default and requires that any of the default libraries whose methods are needed by the workflow also be explicitly included in the library list. See *Workflow Element* in the XML chapter for more details on specifying a library list.

When **Call** is selected for the workflow step's **Action**, the method name is specified in the **Source** box. The methods in these workflow libraries are listed and briefly described in the *Workflow Library Methods* section of the Editing Workflow XML chapter.

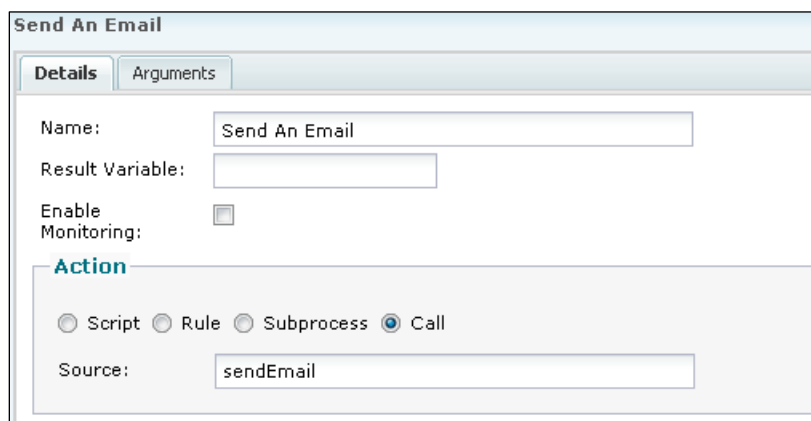


Figure 16: Step calling a workflow library method

Step Arguments

When arguments need to be passed to the script, rule, subprocess, or library method invoked by a step, they must be specified on the step's **Arguments** tab.

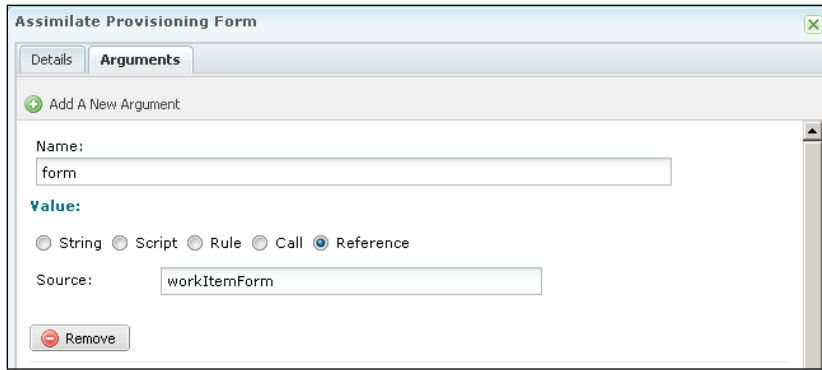


Figure 17: Step Arguments tab

Arguments can be specified in any of the following ways:

Type	Description
String	a literal value (example: explicitly provide the name of an email template to use)
Script	a segment of java beanshell code that returns a value
Rule	a workflow Rule that returns a value (works like Script except beanshell is contained within a re-usable rule)
Call	a call to a workflow library method that returns a value
Reference	a reference to one of the workflow’s process variables

NOTE: Complexity arises when the Script, Rule, or library method used in an Argument requires arguments itself. Scripts, Rules, and Library methods specified within arguments automatically have access to the workflow context, which means they can access workflow variables directly, as needed, through the workflow context “get” methods. However, arguments also automatically get access to the step arguments declared before them in the step. Since retrieving data from the workflow context can be cumbersome, it is often easier to declare the variables needed by the script/rule/call as earlier arguments to the step so they are directly accessible to the later argument’s script, rule, or called method.

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	Call	getManager

Return Variables

Each step can return only one Result Variable, which can be specified through the UI. Steps in which the Action invokes a Subprocess can also use “Return” Variables, by which multiple values can be passed back from the subprocess to the main workflow. These, however, must be specified directly in the XML, as the UI does not provide a vehicle for declaring Return variables. See *Return Elements* for more information on declaring these variables.

More on Start and Stop Steps

Like other steps, Start and Stop steps can contain Actions that execute scripts, rules, subprocesses, or calls to workflow library methods. By convention, these steps are included in every workflow but are there only to designate a clear starting and ending point for the workflow. They are generally left as empty steps (no action), though occasionally, debugging messages may be printed from them 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. To change a step's icon, right-click the step and click **Change Icon**. Select the desired icon style from the pop-up window that appears.

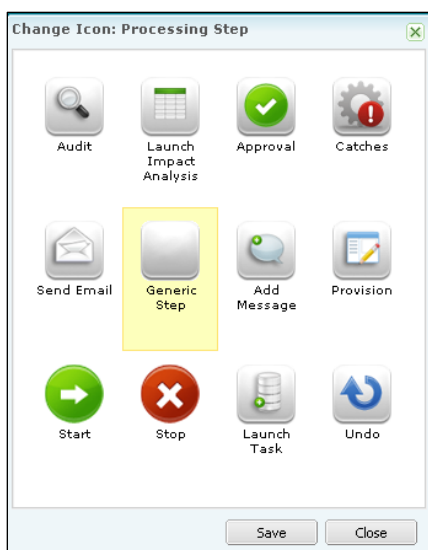


Figure 18: Step Icon Options

Approval Steps

Approval steps are a special case of step in IdentityIQ. Approvals allow data to be gathered from a user through a work item. In a true approval, the user is asked to review a requested action (e.g. granting a Role to an Identity) and give their approval for it to be processed.

Approvals can also be used more broadly to ask a user to provide other data, such as providing a value for a missing attribute, like an Identity's Department name. This second usage of approvals commonly involves the use of custom forms, which must be done through XML rather than the UI. This is documented more in the *Approval Steps* section in the Editing Workflow XML chapter and in the *Custom Forms* section of the Advanced Workflow Topics chapter.

To create a basic approval through the user interface, right-click the desired step and click **Add Approval**.

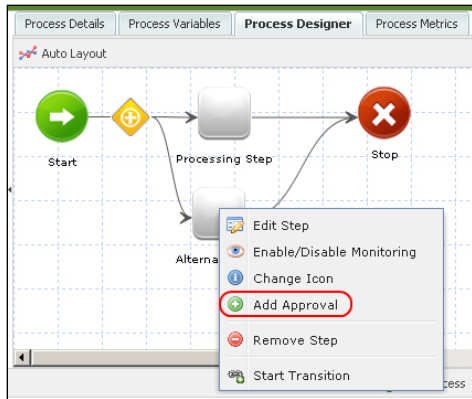


Figure 19: Add Approval to a Step

NOTE: A step can either contain an action or an approval but not both. Approval steps are only used for approval processing, not for performing other actions (scripts, subprocesses, etc.) as well.

Once the approval exists in a step, it can be edited by right-clicking the step and clicking **Edit Approval** or by choosing **Edit Approval** from the Step Details window.

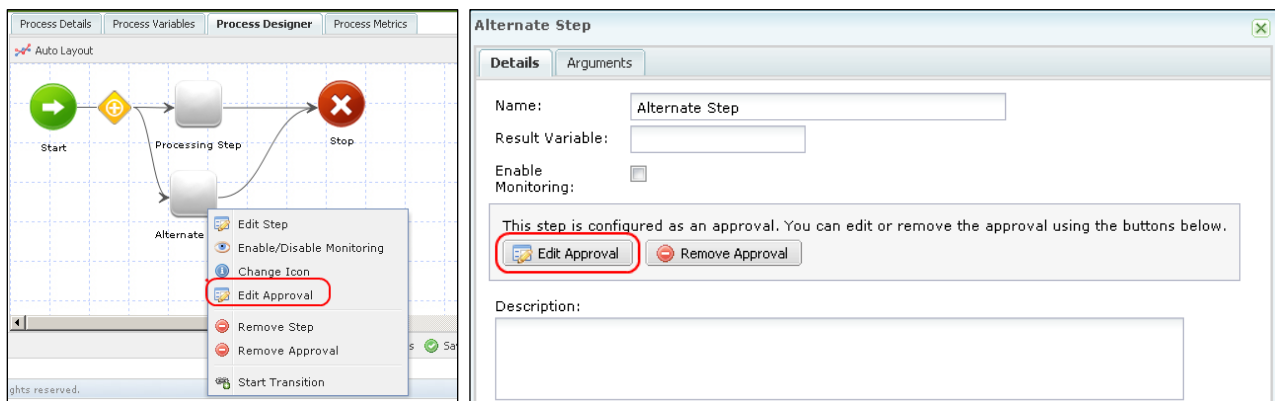


Figure 20: Edit Approval on a Step

The Approval can be constructed in many ways, ranging from a simple one-person approval to a complex approval process involving multiple people with different approval modes and notification schemes. Approvals are highly customizable to meet a wide variety of business needs.

Approval Details

Every approval includes the following fields to be completed on the approval's **Details** tab:

Field	Description
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
Mode	Keyword to specify how approval is processed; Mode can be determined from string, script, rule, call, or reference (default is string)

	<p>Valid values are:</p> <p>serial – approvers are specified in order and item is passed to each approver in that order; if any approver in the chain rejects, item is rejected</p> <p>serialPoll – approvers are specified in order and item is passed to each approver in that order; data is collected on approvals and rejections but rejection by one does not mean rejection of item; action decision is expected to be specified in AfterScript logic</p> <p>parallel – item is sent to all named approvers at one time; item is rejected if any approver rejects it</p> <p>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; action decision is expected to be specified in AfterScript logic</p> <p>any – item is sent to all named approvers at one time; first approver to respond makes the decision for the group</p>
Owner	<p>Approver for the approval; this can be more than one Identity name; it can be specified by string, script, rule, call, or reference</p> <p>When more than one owner is specified, Mode determines how and when the item is submitted to each listed owner (<i>parallel</i>, <i>parallelPoll</i>, and <i>any</i> modes submit the approval work item to all owners at once; <i>serial</i> and <i>serialPoll</i> modes wait until the first owner has completed the approval before submitting to the next in the list)</p>
Description	<p>Defines work item description (shown as the work item Name in the approver’s Inbox); set by string, script, rule, call, or reference</p>

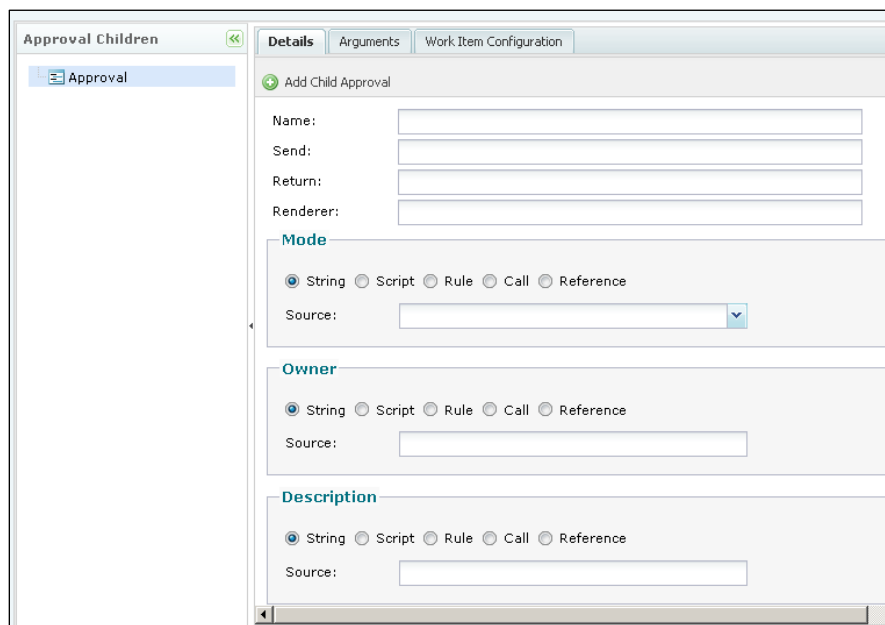


Figure 21: Approval Details window

Approval Arguments

Arguments to the Approval can be set on the **Arguments** tab. In general, most variables are passed to approval through “send” list, but any arguments that require transformation (through script/rule/library method) must be

sent through an Arg element. Additionally, args defined with reserved system names (listed in the XML chapter's *Approval Steps* section) are passed through the Arg element with the reserved name specified.

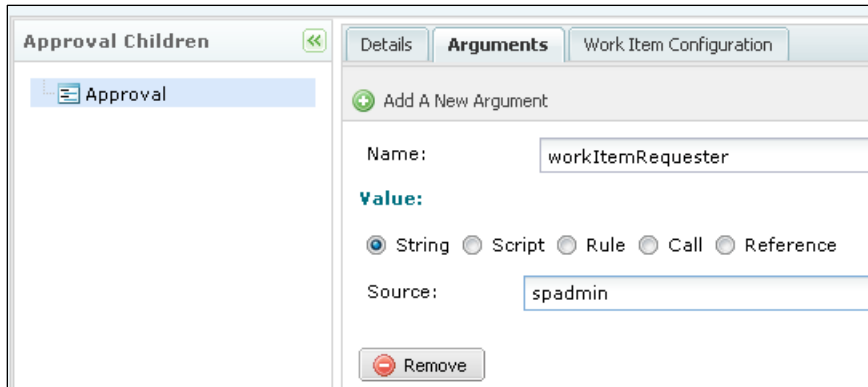


Figure 22: Argument to Approval (reserved name example)

Work Item Configuration

The **Work Item Configuration** tab allows the user to specify some details about the work item's notification and escalation/reminder policy. The work item is what appears in the "owner's" IdentityIQ inbox, requiring their input (i.e. the approval request itself). If no configuration is specified, the default work item configuration is used. To change the configuration for the work item, first select **Override Work Item Configuration**.

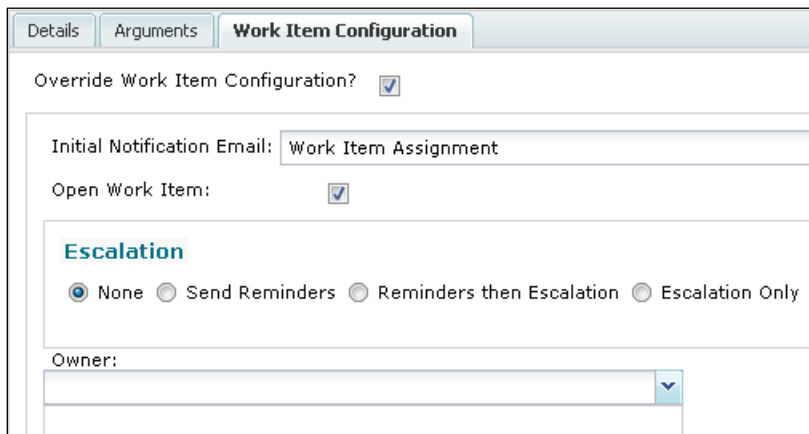


Figure 23: Work Item Configuration specification

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

Option	Description
Initial Notification Email	Change the notification email template by selecting the desired template from the list
Open Work Item	Select to create a work item for the approval; almost always selected Only in rare cases where a notification is desired but no work is required would a work item not be created.
Escalation	Choose an escalation policy: None : no escalation Send Reminders : allows configuration of reminder options (days before first

	<p>reminder, frequency, email template)</p> <p>Reminders then Escalation: allows reminder option configuration plus escalation option configuration (reminders before escalation, escalation owner rule, escalation email)</p> <p>Escalation Only: allows configuration of escalation options (days before expiration, escalation owner rule, escalation email)</p>
Owner	Specify the owner of the work item configuration (not currently used in workflow processing, so can be left blank)

More on Approval Modes

The **Mode** on an approval is a very powerful tool, especially when more than one owner is involved. Through this simple variable specified by the designer, the workflow engine is able to create automatically the order and process in which the item will be reviewed and approved by the specified owner(s).

For example, to have three different Identities review and approve an item, stopping the review process the first time someone rejects the item, all the workflow designer must do is specify the three identities as the approval owners and select “Serial” mode. (The identities can be specified with literals but more commonly are identified programmatically based on their relationship to the approval item – e.g. manager, owner, etc.) Likewise, to get the input of four different people and make a decision based on a majority rule of their responses, specify the four identities as the approval owners and select “SerialPoll” or “ParallelPoll” mode to have the system collect the responses from all the approvers.

NOTE: Since the “Poll” modes are designed to collect responses without making automatic decisions based on any responder’s rejection of an item, these approvals continue forward until they are intercepted and redirected programmatically. They expect the “AfterScript” for the approval to collect the responses and direct the appropriate course of action. The AfterScript cannot be specified through the UI and must be written in XML. See the XML chapter’s *Approval Steps* section for more details.

When multi-person approvals are specified as a single approval with multiple owners, the work item configuration is exactly the same for all owners. To customize the presentation for individual users or to create more complex structures that use different approval modes for different sets of users, child approvals must be defined.

Child Approvals

Child Approvals allow the workflow designer to customize approval processing or presentation for the different sets of Identities involved in the approval process. For example, perhaps a change in a user’s assigned “region” requires someone in HR to sign off on it and also requires manager approval; the approval of the Identity’s own manager is required, but it does not matter which HR individual completes the sign-off, as long as one of them does. This approval can be created through child approvals.

To create a child approval, click **Add Child Approval** on the parent approval’s **Details** tab. Then click the child approval in the **Approval Children** hierarchy to select it for editing.



Figure 24: Add a Child Approval

To set up the approval described in the example, create two child approvals: HR Approval and Manager Approval. HR Approval will be set up as a “Mode: Any” approval so any of the Identities who meet the criteria can make the decision for the group. Manager Approval will be set up as a Serial approval with the Identity’s manager specified as the Owner (since only one person is involved, the mode doesn’t actually matter here, but Serial is the default mode).

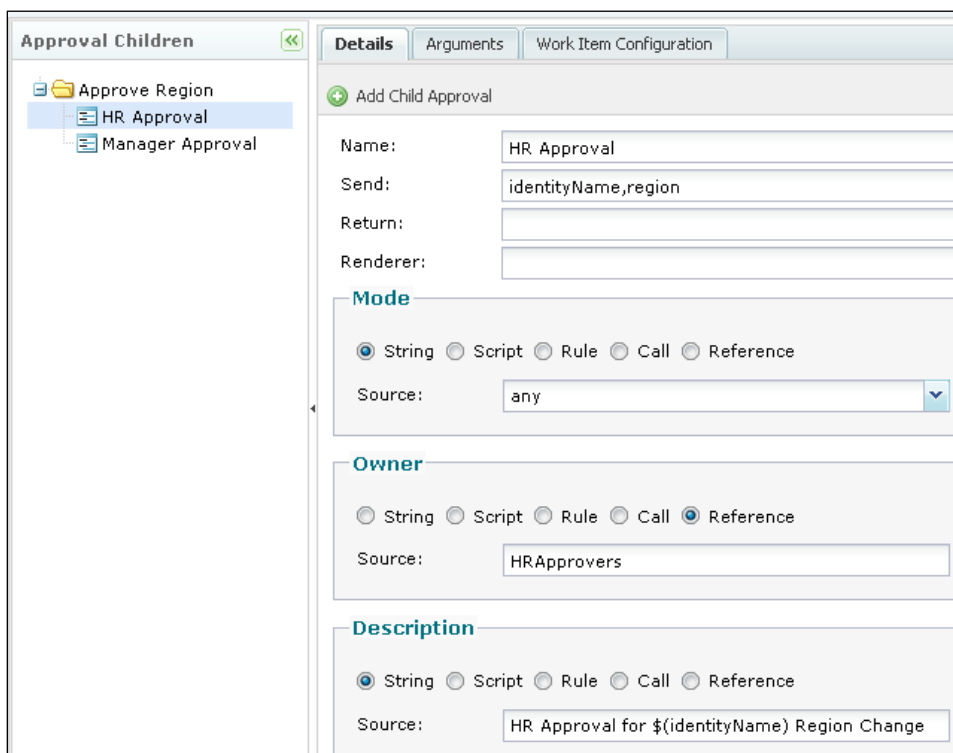


Figure 25: Child Approval 1 (HR Approval)

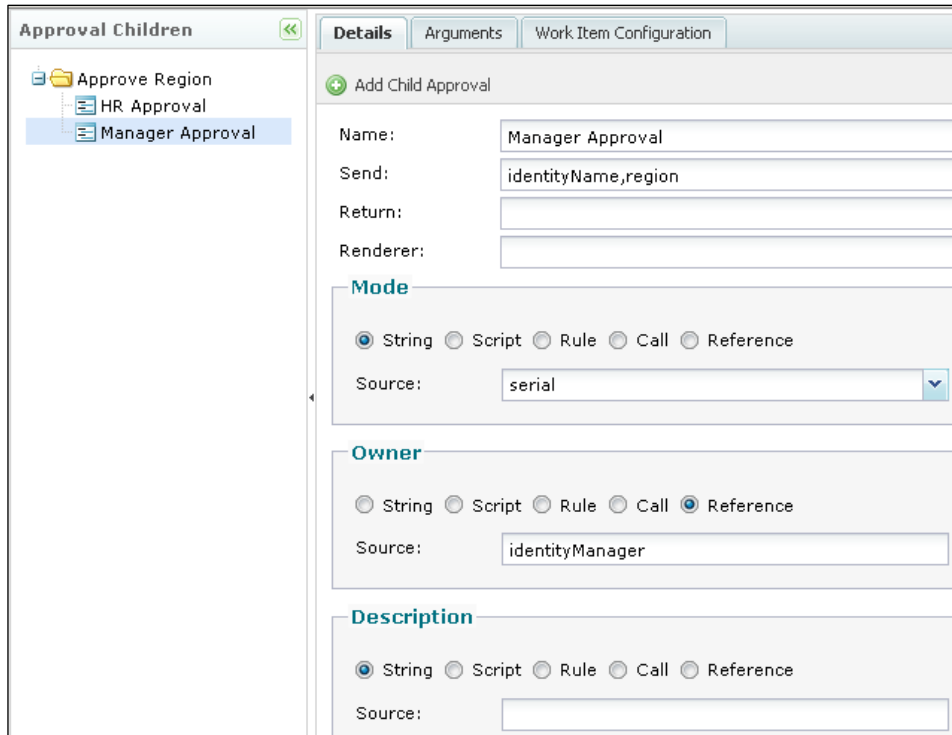


Figure 26: Child Approval 2 (Manager Approval)

NOTE: The reference variables HRApprover and identityManager shown above are process variables defined with initialization scripts that retrieve the appropriate sets of Identities.

If either approval requires a custom work item configuration, it can be specified on that approval’s **Work Item Configuration** tab. Work Item Configurations are inherited by child approvals (unless specifically overridden for the child). Therefore, if a single custom work item configuration is desired for the entire set of approvals, it should be specified on the parent approval’s **Work Item Configuration** tab and will be inherited by the child approvals.

Step Transitions

Steps are connected through “Transitions.” Transitions may simply connect one step to the next in a sequential fashion or may include evaluation statements that allow for conditional processing (i.e. certain data conditions can cause the workflow to execute step A vs. step B).

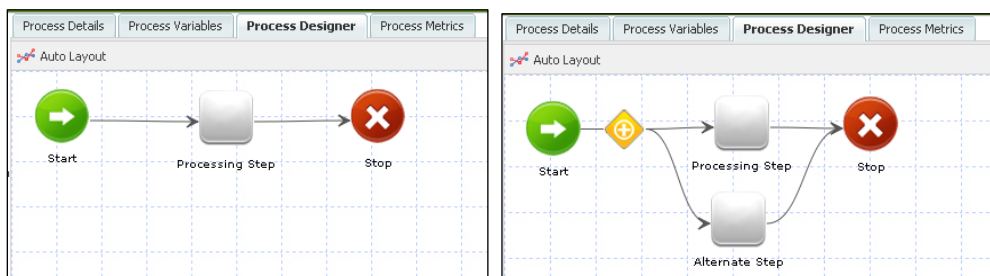


Figure 27: Transitions with and without conditions

To edit the transition conditions, right-click the transition diamond and click **Edit Transitions**.

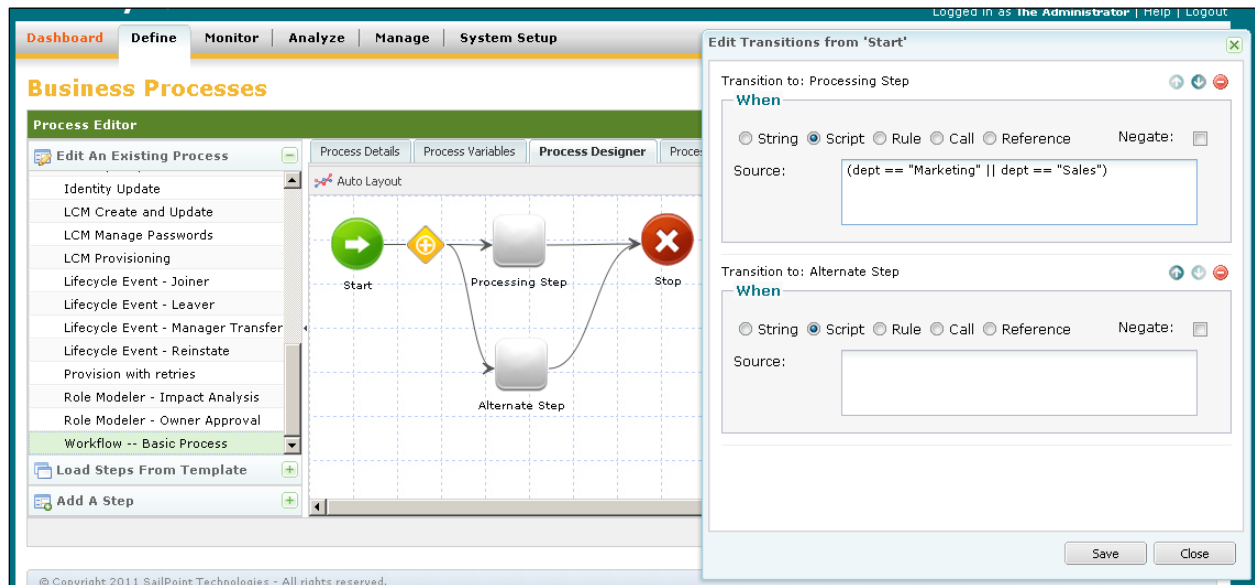


Figure 28: Transition conditions

A transition can have as many conditions to evaluate (and resulting next steps) as desired. Conditions are evaluated in the order they are listed. The up and down arrows in the transitions dialog box can be used to re-order the conditions. The final transition option usually specifies no conditions so its path is followed when none of the other conditions have been met. (This is a recommended best practice.)

Conditions can be expressed as any of the following:

Type	Description
String	Not used; this is an artifact of the common structure used for variable setting and does not really apply to conditions; a literal value of “True” or “False” could be specified but then does not allow any evaluation in the transition – True would always execute the associated step and False would always bypass it
Script	Segment of java code that evaluates process variables
Rule	Workflow rule containing reusable segment of java code to evaluate process variables
Call	Call to invoke a Java method in the IdentityIQ workflow library, exposed through standard workflow handler
Reference	Evaluation of a defined process variable

Conditions must evaluate to Boolean values (True = execute the step specified; False = evaluate the next condition in the list). Selecting the **Negate** option changes the evaluation to its opposite, so if the condition evaluates to False, the negate option changes it to True and results in transition to the specified step.

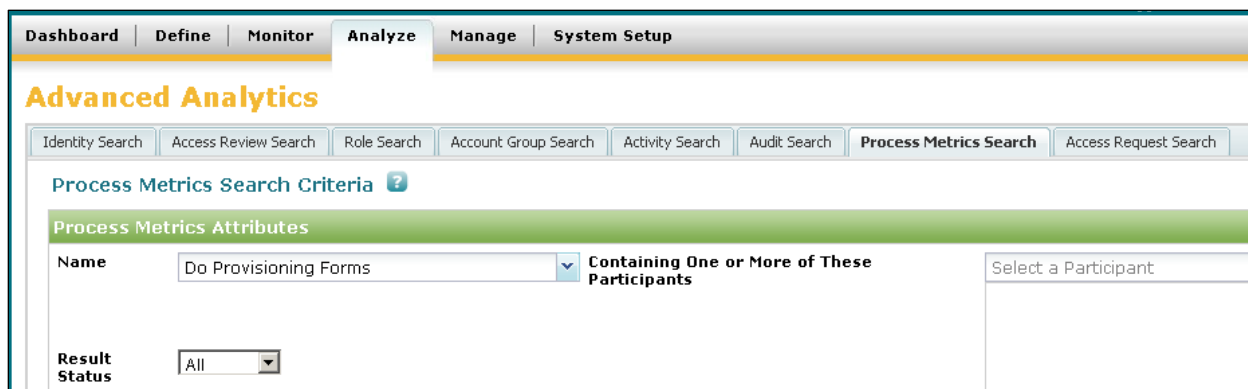
NOTE: Good design dictates that all processing options eventually lead to the “Stop” step, so every route should eventually conclude with a step that transitions to Stop.

Process Metrics tab

The Process Metrics tab displays statistics about the workflow's execution. It indicates how often it has run, how often it has succeeded or failed, the execution duration (average and maximum) and the date it last ran. This can be useful for troubleshooting workflows by showing at a glance whether the workflow is failing regularly and whether it is a longer-running process than might be expected.

Process Details	Process Variables	Process Designer	Process Metrics
Show times in Minutes			
Executions:	0		
Successful Executions:	0		
Failed Executions:	0		
Active Executions:	0		
Average Execution Time:	0		
Maximum Execution Time:	0		
Date of Last Execution:	This Process has never been executed.		

Additional process metrics, including data tracked at the step level, can be viewed through the **Analyze -> Advanced Analytics -> Process Metrics Search** tab.



Metrics tracking can be turned on for individual workflow steps on the step **Details** window by selecting **Enable Monitoring**.

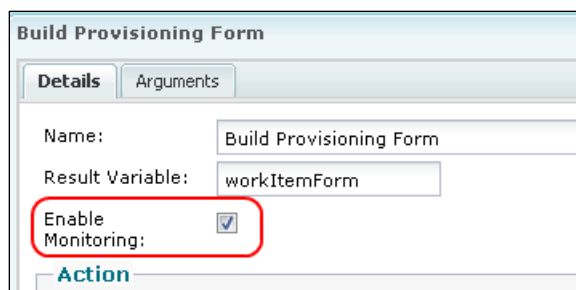


Figure 29: Enable Monitoring Flag

It can be turned on for all steps in a workflow by clicking **Monitor Entire Process** at the bottom of the business process editor window.

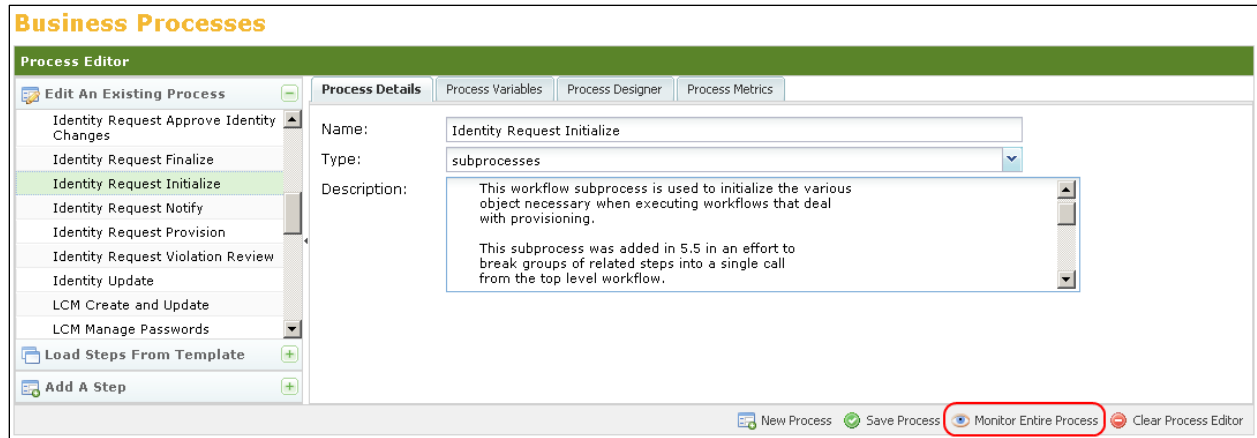


Figure 30: Monitor Entire Process option

Editing Workflow XML

Many implementers choose to do the initial workflow creation through the UI and complete the process by editing the XML directly. Still others may opt to do all workflow development in XML, bypassing the UI altogether. They may choose either to write the XML from scratch or to use an existing workflow's XML as a template for a new process. All of these approaches are valid and may 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 URL: **[IdentityIQ access path]/Debug**. Select **Workflow** from the object list and click **List** to see a list of all defined Workflows in the system.

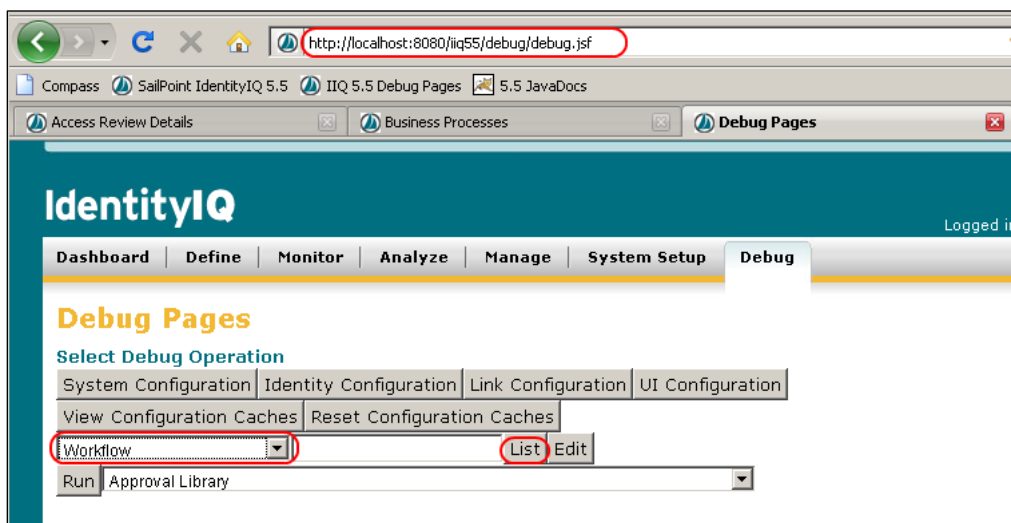


Figure 31: List Workflows from Debug Pages

Debug Pages	
Workflow Objects	
Name	Type
Check Status of queued items	subprocesses
Deferred Role Activation	DeferredRoleActivation
Deferred Role Assignment	DeferredRoleAssignment
Deferred Role Deactivation	DeferredRoleActivation
Deferred Role Deassignment	DeferredRoleAssignment
Department Attribute Value Change	IdentityEvent
Department Attribute Value Change (Do not use)	IdentityEvent
Do Manual Actions	subprocesses
Do Provisioning Forms	subprocesses
Identity Correlation	IdentityCorrelation
Identity Refresh	IdentityRefresh
Identity Request Approve	subprocesses

Figure 32: List of Workflow Objects

Click the desired Workflow's name to view its XML representation. The XML can be edited, and changes saved, through this window. It can also be copied from here and pasted into an external editor of choice – perhaps one that offers syntax highlighting.

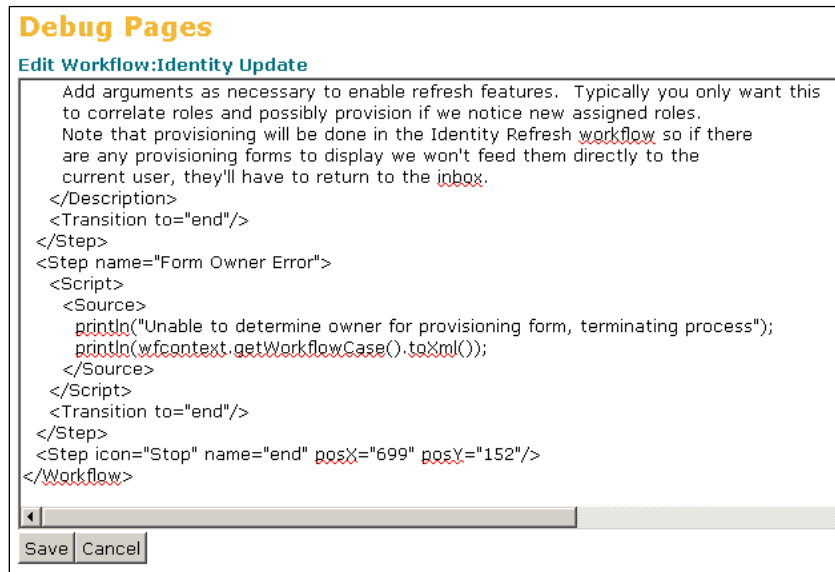


Figure 33: Edit Workflow in Debug Pages

IIQ Console

Alternatively, one or more workflows can be exported 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 once. The iiq console export command can extract all the Workflow XMLs into a single file at once. Many developers then opt to parse the XML into a separate file for each workflow and save the files in the installation's source code control system for later use in system environment migrations or in product upgrade processes.

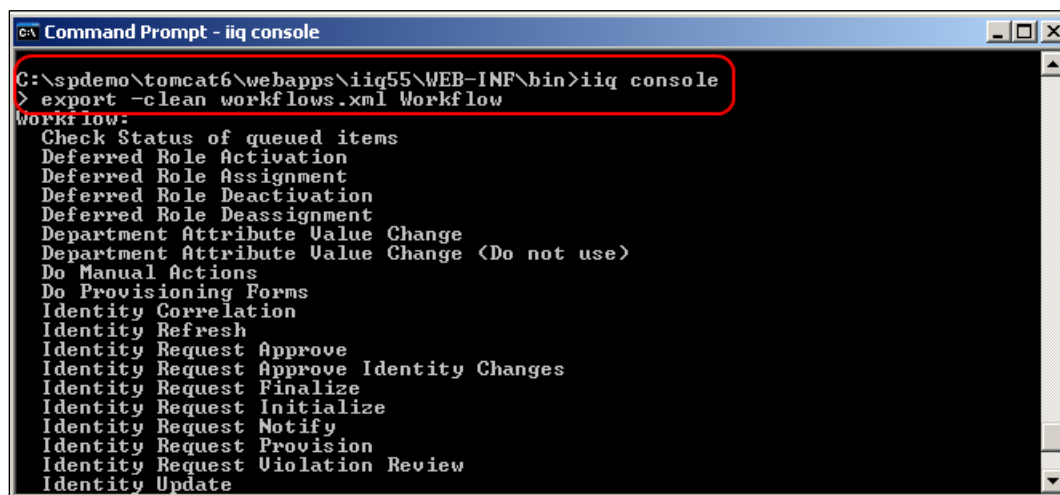


Figure 34: iiq console export command

Re-importing the XML

Only Workflow XML saved within IdentityIQ will be executed by the system, so externally edited XML documents *must* be reimported for the changes made to them to take effect.

This can be done using the console's import command or through the user interface's **System Setup -> Import from File** option.

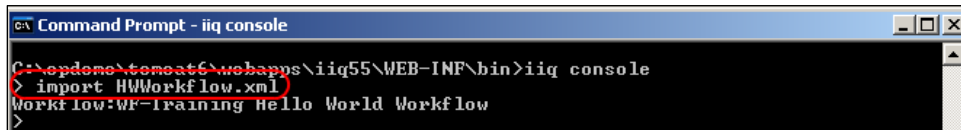


Figure 35: Console Import command



Figure 36: Import from File window

Dollar-Sign Reference Syntax

Workflow variables can be referenced inside XML tags (as well as in user interface fields) through $\$()$ notation and will be resolved into their variable values. For example, if a variable `identityName` is defined and contains the full name of an Identity (e.g. "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 like this, it functions the same as specifying `value="ref:identityName"`, but the more common usage of this 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="IdentityUpdate">
```

The Workflow element's attributes are as follows:

Workflow Attribute	Purpose
name	Short descriptive name for the workflow (displayed in UI selection list-boxes and list of existing business processes on the Process Editor window)
type	Workflow type; used to filter workflow selection lists in configuration windows (where the user can select which workflow to invoke based on certain 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; default is "false", so omitting this argument means that if transition conditions specified all evaluate to false, the workflow will use implicit transitions and execute the next sequential step in the XML; not an issue if developer follows the best practice of making last transition in any set unconditional
libraries	Lists workflow libraries needed by the workflow; when not specified, workflows automatically have access to Identity, Role, PolicyViolation, and LCM libraries (see Workflow Library Methods for library content details)
handler	The default workflow handler is "sailpoint.api.StandardWorkflowHandler" – does not need to be specified when this is used (best practice is to omit in this case); when a custom workflow handler is used, it must EXTEND the default handler (not just replace) and must be specified in the workflow's Handler argument

Variable Definitions

SailPoint's recommended best practice is to identify all variables for the workflow at the top of the XML document (though this is really only for readability). Therefore, the variable definitions come next in the XML.

Minimally, variable elements require a name. Other attributes can indicate the variable's type and usage (input, required, editable, return). A description can, and generally should, be specified for each variable. When needed, an initialization value can also be provided. It is strongly recommended that the initialization option be used rather than creating separate steps to initialize each variable; using initialization values is more efficient

(from a development standpoint), easier to read, and easier to debug (since Trace reports initializations as they occur). For more on variable initialization, see *Initializer Options* below.

```
<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
assignment</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 (often omitted; type declaration is not enforced by the application and generally used primarily for documentation)
initializer	Initialization value for the field; (see below for more details)
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	Purpose
Description	Provide a description of the variable's usage/purpose
Script	Alternative to script in initializer attribute value; should be used for initializer scripts of any length or complexity
Source	Nested within the Script tag; contains the java beanshell source for the action to be executed

Initializer Options

The Initializer attribute requires additional attention. When set through the user interface, options include specifying it as a string, script, rule, call, or reference. The same options are available directly through the XML.

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

script	<p>Assign a value based on the results of a Java beanshell script</p> <p>Examples:</p> <p>(1) Inline Script (use only for very short, simple scripts)</p> <pre><Variable initializer="script:(identityDisplayName != void) ? identityDisplayName : resolveDisplayName(identityName)" input="true" name="identityDisplayName"></pre> <p>(2) Script within nested <script> element (use for most script initializers – scripts of any complexity or length)</p> <pre><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></pre>
rule	<p>Assign a value based on the return value of a workflow Rule</p> <p>Examples:</p> <pre><Variable initializer="rule:wfrule_GetIdentityName" name="IdentityName"></pre>
call	<p>Assign a value based on the return value of a call to a workflow library method</p> <p>Example:</p> <pre><Variable initializer="call:getObjectName" name="roleName"></pre>
ref	<p>Assign a value based on a reference to another workflow variable (rarely used)</p> <p>Example:</p> <pre><Variable initializer="ref:otherVar" name="myVar"/></pre>

Workflow Description

A Description element should be included to describe the purpose of the workflow; this is strictly for a human reader and is not used in the workflow processing but is still strongly recommended. In the UI, the contents of this element are displayed on the Process Details tab of the Business Process definition. For readability, 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 used by workflows have been grouped together into Rule Libraries. These are defined as Rules in IdentityIQ but they contain sets of related but unconnected methods that can be invoked directly by workflow steps (within a “script” action). These are in Rules, rather than in the compiled Java classes, so that their functionality can be easily modified to suit 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 as shown here. (Each Reference element applies to one library, and only the libraries containing the required methods should be included 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>
```

Step Elements

The core of the workflow is contained within the Step elements. At its most basic, a step should contain an icon, name, posX and posY attribute. The action attribute determines what processing the step does. 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>
```

Like Variables, some parts of a Step definition are included as attributes of the step while others are expressed as nested elements within the step.

Step Attribute	Purpose
name	Short but descriptive name for step; displayed in UI graphical display below the step icon
icon	Icon to display for step in UI graphical Process Designer; Valid icon values are: “Start”, “Stop”, “Default” (Generic Step), “Analysis” (Launch Impact Analysis), “Approval”, “Audit”, “Catches”, “Email”, “Message” (Add Message), “Provision”, “Task” (Launch Task), “Undo”
posX, posY	X and Y positions for where the step icon should be displayed on the UI’s graphical Process Designer grid. Omitting the posX and posY values causes the icon to be displayed at the top right of the grid, from which it can still be dragged around to create the desired layout at a later time.
action	The processing action to take for the step (a script, rule, subprocess, or call, as described in <i>Most parts</i> of a transition are included as attributes of the transition, but some scripts

	are expressed as nested elements within the transition.												
	<table border="1"> <thead> <tr> <th>Transition Attribute</th> <th>Purpose</th> </tr> </thead> <tbody> <tr> <td>to</td> <td>Name of next step</td> </tr> <tr> <td>when</td> <td>Condition for progressing to the specified next step</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Nested Tag within Transition Element</th> <th>Purpose</th> </tr> </thead> <tbody> <tr> <td>Script</td> <td>Alternative to script in transition's when attribute; should be used for scripts of any length or complexity</td> </tr> <tr> <td>Source</td> <td>Nested within the Script tag; contains the java beanshell source for the condition evaluation</td> </tr> </tbody> </table>	Transition Attribute	Purpose	to	Name of next step	when	Condition for progressing to the specified next step	Nested Tag within Transition Element	Purpose	Script	Alternative to script in transition's when attribute; should be used for scripts of any length or complexity	Source	Nested within the Script tag; contains the java beanshell source for the condition evaluation
Transition Attribute	Purpose												
to	Name of next step												
when	Condition for progressing to the specified next step												
Nested Tag within Transition Element	Purpose												
Script	Alternative to script in transition's when attribute; should be used for scripts of any length or complexity												
Source	Nested within the Script tag; contains the java beanshell source for the condition evaluation												
	Step Actions below)												
wait	Pauses action for a specified duration (see <i>Wait Attribute</i> for details)												
catches	Causes step to be run when "Complete" status is caught, rather than through a transition from another step (see <i>"Catches" attribute</i> for details)												
resultVariable	Variable name containing return value from the step												

Nested Tag within Step Element	Purpose
Description	Provide a description of the step's purpose
Transition	Identifies the next step the process will move to when the current step is complete (see <i>Transition Element</i> below)
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 <i>Return Elements</i> below)
Script	Alternative to script in step's Action attribute; should be used for action scripts of any length or complexity
Source	Nested within the Script tag; contains the java beanshell source for the action to be executed

Transition Element

The Transition element, always nested within a step, indicates the name of the next step the process will execute following completion of the current step. Transitions can contain conditions based on a string, script, rule, call, 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.

Condition Type	Description and Examples
string	Not used; this is an artifact of the common structure used for variable setting and does not really apply to conditions; a literal value of "True" or "False" could be specified but then does not allow any evaluation in the transition – True would always execute the associated step and False would always bypass it
script	Evaluate script result value to determine step transition; very short scripts are specified inline

	<p>on the Transition element, within the “when” attribute. Longer ones are expressed within nested <script> and <source> elements, as shown below.</p> <p>NOTE: Script is the default transition “when” option so the “script:” prefix can be included or omitted.</p> <p>Examples:</p> <p>(1) Inline Script (use only for very short, simple scripts)</p> <pre><Transition to="Exit On Policy Violation" when="script:((size(policyViolations)> 0) && (policyScheme.equals('fail')))" /></pre> <p>(2) Longer script within nested <script> tag (use for transition scripts of any complexity or length)</p> <pre><Transition to="end"> <Script> <Source> ("cancel".equals(violationReviewDecision) ((size(policyViolations) > 0) && (policyScheme.equals("fail")))) </Source> </Script> </Transition></pre>
rule	<p>Evaluate the return value of a workflow Rule to determine step transition</p> <p>Examples:</p> <pre><Transition to="Process Approval" when="rule:RequireApprovalRule"></pre>
call	<p>Evaluate return value of a call to a workflow library method to determine step transition</p> <p>Example:</p> <pre><Transition to:"Check Status" when="call:requiresStatusCheck" /></pre>
ref	<p>Evaluate a defined (Boolean) workflow variable to determine step transition</p> <p>Example:</p> <pre><Transition to="Refresh Identity" when="ref:doRefresh"/></pre>
Unconditional	<p>Specified as last transition option to give a default path for the transition</p> <p>Example:</p> <pre><Transition to="Approve"/></pre>

Most parts of a transition are included as attributes of the transition, but some scripts are expressed as nested elements within the transition.

Transition Attribute	Purpose
to	Name of next step
when	Condition for progressing to the specified next step

Nested Tag within Transition Element	Purpose
Script	Alternative to script in transition's when attribute; should be used for scripts of any length or complexity
Source	Nested within the Script tag; contains the java beanshell source for the condition evaluation

Step Actions

Of course, most steps involve far more than just a name and a transition; they include an “action” attribute that execute the workflow processing. The action of a step may be a script or may invoke a rule, subprocess, or a call to a workflow library method.

Action Type	Description
Script	<p>As with scripts in other parts of the workflow XML, the script itself may be contained within the action attribute or may be nested within the Step in a <Script> block.</p> <p>Examples:</p> <p>(1) Inline Script (use only for very short, simple scripts)</p> <pre><Step action="script:approvalSet.setAllProvisioned();" icon="Task" name="Post Provision"> <Transition to="Stop"/> </Step></pre> <p>(3) Longer script within nested <script> tag (use for action scripts of any complexity or length)</p> <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>
Rule	<p>A step can execute a block of Java beanshell code encapsulated in a reusable workflow Rule.</p> <p>Example:</p> <pre><Step action="rule:WFRule_verifyIdentity" icon="Task" name="Verify Identity" posX="600" posY="202"></pre>
Subprocess	<p>A subprocess is invoked through the inclusion of a <WorkflowRef> element within the step, referencing the sailpoint.object.Workflow class and the specific workflow by name.</p> <p>Example:</p> <pre><Step icon="Task" name="Initialize" posX="320" posY="126"> ... <WorkflowRef> <Reference class="sailpoint.object.Workflow" name="Identity Request Initialize"/> </WorkflowRef> <Transition to="end"> </Step></pre>

Call	<p>Calls to workflow library methods can be used to do step processing.</p> <p>NOTE: Call is the default action option so the “call:” prefix can be included or omitted.</p> <p>Example:</p> <pre><Step action="call:refreshIdentity" icon="Task" name="Refresh Identity" posX="618" posY="242"></pre>
------	---

Arguments

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

```
<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"/>
  ...
  <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 (e.g. <Arg name="myVar" value="rule:myWFRule"/>), any needed arguments to the script, rule, or library method cannot be explicitly specified. However, these scripts, rules, and library methods automatically have access to the workflow context object, which means they can access workflow variables directly through the workflow context’s “get” methods. These scripts/rules/methods can also access any step arguments that were defined before them in the step declaration. So, for example, with the step declaration shown below, the method that identifies the value for the Manager argument can use the value in the identityName argument in its processing, if needed.

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

Available Arg attributes are shown in this table:

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

Return Elements

More than one value can be returned from a subprocess by declaring <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 subprocess 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 actually required, though it is a best practice to specify it anyway for clarity.) The “merge” attribute is used when the variable is a List and the returned values should be appended to the current workflow’s list instead of replacing it. As with Args, Return Elements’ “value” attribute can be specified as a string, script, rule, call, or reference, with string being the default. If the “value” argument is omitted, the value of the “name” variable copied as-is into the “to” variable, but a script/rule/method could be used to transform or modify the value as it is passed.

```
<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>
```

Available Return attributes are shown in this table:

Return Attribute	Purpose
name	Variable name in process from which the data is being returned
to	Variable name in the workflow step to which the data is being passed
value	Value to pass into the variable (string, script, rule, call, reference)
merge	Flag indicating whether the value should be merged into the target variable instead of replacing it (relevant for list variables)
local	Only applies to returns on Approvals (see Approval Steps below); flag indicating that value is being passed to local storage within the parent approval instead of to a workflow case variable; used for complex approvals where a work item state is saved for later analysis in a script

Call

Calls to workflow library methods can be used to do step processing. Like subprocesses, they may require arguments to be passed to them. Declaration of the arguments are done the same way as with subprocesses. They are invoked with a “call” action, as shown below:

```
<Step action="call:refreshIdentity" icon="Task" name="Refresh Identity" posX="618"
posY="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 40rovisioning since that
  can result in provisioing 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 workflow element’s “libraries” attribute, if specified. If no libraries attribute is specified, the workflow automatically has access to the methods in the

Identity, Role, PolicyViolation, and LCM 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.

NOTE: Installations may create custom libraries for commonly used methods required for their business. However, custom library methods must be named with unique names that do not conflict with standard library method names. Conflicts will resolve as a reference to the standard library method. It is possible to extend a standard library and overload its method names, but this is not consider a best practice; it is strongly recommended that new names be created for nonstandard methods so it is clear at first glance that the method used is not a standard one.

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 (default is string).

```
<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 actually creates a special kind of step whose whole purpose is to create a pause in the action. Wait steps are commonly used in re-try logic to allow behind-the-scenes processing to occur before the workflow attempt to repeat an action.

“Catches” attribute

New in version 5.5 of IdentityIQ is the concept of “Catch” steps. These steps are not invoked through a transition from a previous step but are invoked by a “thrown” message that is intercepted (or caught) by the step. At present, only a “complete” message is thrown and can be caught. This occurs when the workflow is otherwise finished – when all sequential steps have been executed to completion or a failure condition has resulted in 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 as currently implemented 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 has been purged.

Each installation may drive custom logic based on catching this “complete” message as desired. Later releases may add a catchable “error” message as well.

Approval Steps

Approval is one of the most common actions performed by a workflow process. 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 to the approver(s).

Some Approval steps are designed to get a user’s approval on a requested change, as is implied by the name “approval”. However, the Approval element can be used any time data needs to be gathered from a user. Typically, when approval steps are used to gather non-approval data, a custom form is used to present the work item to and request the needed information from the user. See *NOTE*: This is distinct from running a workflow as a subprocess. If a workflow is invoked as a subprocess, the calling workflow will wait until the subprocess has finished (and returned control to the caller) before it continues with its processing. This “call” causes a completely separate workflow to begin running, and as soon as the new workflow has been kicked off, the calling workflow will move on to its next step.

Custom Forms in the Advanced Workflow Topics section of this document for more details on usage of custom forms.

As with other Workflow elements, some modifiers are specified as attributes on the Approval element itself while others are specified through nested elements within the Approval.

Approval Attribute	Purpose
mode	<p>Specifies how approval is processed; Mode can be determined from string, script, rule, call, or reference (default = string)</p> <p>Valid values are:</p> <ul style="list-style-type: none"> serial – approvers are specified in order and item is passed to each approver in that order; if any approver in the chain rejects, 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 but rejection by one does not mean rejection of item; action decision is expected to be specified in AfterScript logic parallel – item is sent to all named approvers at one time; 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; action decision is expected to be specified in AfterScript logic any – item is sent to all named approvers at one time; first approver to respond makes the decision for the group
owner	Approver (can be one or more); can be specified by string, script, rule, call, or

	reference (default is string) Mode determines how and when the item is submitted to each listed owner when more than one is specified
renderer	JSP include to render the work item details
return	Comma-separated values (CSV) list of variable names to copy from completed 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; Description can be set by string, script, rule, call, or reference (default=string)
validator	used to validate any information entered by the user during the approval; can be specified as string, script, rule, call, or reference (default=script); usually a nested ValidatorScript element is used instead of a validator argument

Nested Tag within Approval Element	Purpose
AfterScript	<p>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 (as described in <i>Rule Libraries</i>)</p> <p>NOTE: ParallelPoll and serialPoll items always execute this script after all responses have been collected. With either of these modes, the logic in this script should aggregate the results and determine whether the item should be approved or rejected. The criteria for approval or rejection (e.g. majority rule, any approval=approval, etc.) is up to the business.</p> <p>In either “poll” mode, the after script is inherited by child approvals if they do not specify one of their own. In other modes, the after script is not inherited by child approvals.</p>
InterceptorScript	<p>This script is more complex than the AfterScript and is much less frequently used. It 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 in 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 (as with any other script).</p> <p>Method values (for conditional analysis within InterceptorScript logic):</p> <ul style="list-style-type: none"> • startApproval • preAssimilation • postAssimilation • archive • endApproval <p>NOTE: If an InterceptorScript and AfterScript both exist, the InterceptorScript’s “postAssimilation” logic runs before the AfterScript.</p>
ValidatorScript	Script to perform validation on the work item (e.g. any data entered by the user on

	the approval) before it is assimilated; inherited by any child approvals
Source	Nested within the AfterScript, InterceptorScript, and ValidatorScript tags; contains the java beanshell source for the script
Arg	<p>Arguments available to the approval action; specified by string, script, rule, call, or reference; most variables are passed to approval through “send” list, but args that require any transformation (through script/rule/library method) must be sent through an Arg element.</p> <p>Additionally, args defined with reserved system names (listed below) are passed through the Arg element with that name specified:</p> <ul style="list-style-type: none"> • workItemRequester • workItemDescription • workItemType • workItemTargetId • workItemTargetName • workItemTargetClass • workItemDisableNotification • workItemNotificationTemplate • workItemEscalationTemplate • workItemReminderTemplate • workItemEscalationRule • workItemEscalationStyle • workItemHoursTillEscalation • workItemHoursBetweenReminder • workItemMaxReminders • workItemPriority • workItemIdentityRequestId • workItemArchive
Return	<p>Return value defining how things should be assimilated from work item back into the workflow case (alternative to the “return” attribute’s CSV of variables; more complex but more powerful as well);</p> <p>This is rarely used in approvals, but is most often used when returning an approval work item variable to a workflow variable of a different name or when need to transform work item variable’s contents with a script. The usage of Return elements here follows same rules as step returns from steps that invoke subprocesses, with addition of “local” attribute option (see Return Elements above)</p>

This is an example of a basic Approval step. It 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,
    fallbackApprover);" renderer="lcmWorkItemRenderer.xhtml"
    send="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(wfcont
ext, 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, &quot;owner&quot;)" />
<Transition to="Build Security Officer ApprovalSet"
  when="script:isApprovalEnabled(approvalScheme, &quot;securityOfficer&quot;)" />
<Transition to="end"/>

</Step>

```

NOTE: In the AfterScript in this example, the methods not qualified by the library name are found in the LCM Workflow Rule Library (made available to the workflow through the <RuleLibraries> declaration). The `assimilateWorkItemApprovalSetToIdentityRequest` method is part of the `IdentityRequestLibrary`, made 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’s “libraries” attribute list, but when they are executed from within scripts, the library must be specifically imported for the script itself.

Nested Approvals

Child Approvals created through the UI are expressed as Nested approval elements in the XML representation. When nested approvals exist, the “parent” ceases to be an approval of its own and is there only to organize and contain the child approvals. The Mode on the parent determines how the set of peer Child approvals are processed.

```

<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 will be processed in parallel. Since both of these child approvals are identical (no custom work item config and no children of their own), the same objective could be accomplished with a single approval with multiple owners:

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

The real value in the nested approvals comes when different approval levels are implemented with custom configurations and specifications. For example, the workItemConfig for each of the child approvals can be different, resulting in a notification scheme, escalation policy, etc. for the different approvers. Additionally, nested approvals may be governed by a different approval mode from the one used on the master set and/or may contain their own child approval levels. One child approval might be conducted as an “any” approval (accepting the ruling of the first responder of several listed approvers) while the highest approval level is managed serially. Another child approval may implement custom workItemConfigs for its own child approvals. The example below 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, allows 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 accessible to workflows, if 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.

Example:

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

This example makes methods from the sailpoint.workflow.IdentityLibrary.class accessible to the workflow.

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

The tables below show the workflow libraries and the methods available through them. The methods in the Standard Workflow Handler (while it is not technically a library) are accessible to every workflow and are called through the same syntax as library methods.

Standard Workflow Handler

Method / Usage	Expected Args (*=required)
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(WorkflowContext wfc) Adds message to workflow case that is displayed in UI but not kept in task result (e.g. "Request has been submitted")	message* type (optional severity) arg1-arg4 (up to 4 parameters for the message)
public Object setLaunchMessage(WorkflowContext wfc) Replaces previously added launch message with new message based on new state	message* type (optional severity) arg1-arg4 (up to 4 parameters for the message)
public Object log(WorkflowContext wfc) Send something to log4j	message* level*
public Object print(WorkflowContext wfc) Print something to the console	message*
public Object audit(WorkflowContext wfc) Creates an audit event (allows workflows to put custom	source* action*

entries in audit log (shown in UI)	target string1 – string4
public Object sendEmail(WorkflowContext wfc) Send an email message	to* cc bcc from subject body template* templateVariables sendImmediate exceptionOnFailure
public Object launchTask(WorkflowContext wfc) Launch a defined task	taskDefinition* taskResult sync (true=synchronous execution)
public Object scheduleRequest(WorkflowContext wfc) Launch a generic event request	requestDefinition* requestName (name to assign to request) scheduleDate scheduleDelaySeconds owner
public Object scheduleWorkflowEvent(WorkflowContext wfc) Launch a workflow event request	requestDefinition* requestName (name to assign to request) scheduleDate scheduleDelaySeconds owner workflow* (name of workflow to launch) caseName (optional case name to override default) requestDefinition (not specified – set to standard definition for workflow requests)
public Object commit(WorkflowContext wfc) Commit transaction (not commonly needed in workflows: most commonly used for role approvals)	creator archive
public Object rollback(WorkflowContext wfc) Roll back transaction (not commonly needed in workflows: most commonly used for role approvals)	none

Identity Library

Method / Usage	Expected Args (*=required)
public String getManager(WorkflowContext wfc) Return the name of the manager for the specified identity	identityName
public Object calculateIdentityDifference(WorkflowContext wfc)	oldRoles

Derive a simplified representation of the changes being made to an identity for an approval work item	newRoles plan approvalSet
private void addLinksInformation(WorkflowContext wfc) Modify workflow context's 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(WorkflowContext wfc) Evaluate policy violations that may be incurred by the provisioning plan/project's actions	policies identityName* project plan (either plan or project is required)
public void activateRoleAssignment(WorkflowContext wfc) Assign role(s) to the identity	identity* (ID) role* (ID) detected (Boolean indicating if role was detected vs. assigned)
public void deactivateRoleAssignment(WorkflowContext wfc) Remove role assignments from the identity	identity* (ID) role* (ID) detected (Boolean indicating if role was detected vs. assigned)
public void refreshIdentity(WorkflowContext wfc) Perform an identity refresh on one identity	identity (ID) identityName (either identity or identityName is required)
public void refreshIdentities(WorkflowContext wfc) Perform an identity refresh on a set of identities (can specify one or more identityNames or can specify a filterString or can specify a list of roles – processes the first of those options that is non-null)	identityName identityNames (CSV) filterString identitiesWithRoles (CSV) (any one of these 4 is required)
public Object compileProvisioningProject(WorkflowContext wfc) Compile provisioning plan into provisioning project	plan identityName
public Object buildProvisioningForm(WorkflowContext wfc) Create a form containing provisioning questions (if owner specified, only returns form if there is a form for that owner; if preferredOwner is specified, return that owner's form if exists or any other forms for the identity if none exist for that owner)	project* template (name of form to serve as page template) owner preferredOwner (owner or preferredOwner required but mutually exclusive)
public Object assimilateProvisioningForm(WorkflowContext wfc) Collect data from completed provisioning form and store answers with questions on provisioningProject	project* form*
public Object assimilateAccountIdChanges(WorkflowContext wfc) Update ApprovalSet with any changes to accountIDs	project* approvalSet
public Object buildPlanApprovalForm(WorkflowContext wfc) Build form representing all attributes in a provisioningPlan for approval before provisioning occurs	plan* template
public Object assimilatePlanApprovalForm(WorkflowContext wfc)	form

Collect data from form (assumed to have been created by buildPlanApprovalForm) and put back into the provisioningPlan	plan*
public Object provisionProject(WorkflowContext wfc) Called (by Identity Update and LCM Workflows) after provisioning forms have been completed to provision what remains in the project	project* noTriggers (Boolean)
public Object finishRefresh(WorkflowContext wfc) Called by Identity Refresh workflow, after approvals are done and account completion attributes gathered, to provision what can and complete the refresh process	identitizer refreshOptions (map of args for creating new Identitizer if needed) previousVersion project
public Object buildApprovalSet(WorkflowContext wfc) Called by the LCM workflows to build a simplified ApprovalSet representation of the things in the provisioning plan.	plan*
public Object processApprovalDecisions(WorkflowContext wfc) Process decisions made during approval process audit and react (modifies project's masterPlan and recompiles project if recompile argument is true)	project* dontUpdatePlan disableAudit approvalSet* recompile
public Object processPlanApprovalDecisions(WorkflowContext wfc) Process decisions made during approval process audit and modify the plan (used before plan is compiled into a provisioningProject)	plan* dontUpdatePlan disableAudit approvalSet*
public Object auditLCMStart(WorkflowContext wfc) Create an audit event to mark the start of an LCM flow	approvalSet* flow (name of applicable UI flow)
public Object auditLCMCompletion(WorkflowContext wfc) Create an audit event to mark the completion of an LCM flow	approvalSet* flow
public void disableAllAccounts(WorkflowContext wfc) Used by lifecycle events to disable managed accounts for the identity specified in the workflow	None
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 enable all accounts on the identity specified in the workflow	None
public ProvisioningPlan buildEventPlan(WorkflowContext wfc) Go through all links held by the workflow's specified Identity and creates a plan to enable or disable (specified by "op") all of the Identity's accounts	op* (operation)
public void updatePasswordHistory(WorkflowContext wfc) Add a password to the link password history	plan*
public ProvisioningProject assembleRetryProject(WorkflowContext	project

wfc) Add any account request for an original provisioning project that are retryable, adding them to a new provisioning project Not likely to be used in custom workflow	
public Object retryProvisionProject(WorkflowContext wfc) Execute the retry provisioning project (created in assembleRetryProject) Not likely to be used in custom workflow	project
public Object mergeRetryProjectResults(WorkflowContext wfc) Merge results from retry project onto the main project (called between retries) Not likely to be used in custom workflow	project* retryProject*
public Boolean requiresStatusCheck(WorkflowContext wfc) Identifies whether the project contains any Results that are queued with a requestID stored on the result	project
public Object checkProvisioningStatus(WorkflowContext wfc) Call down to the connector for each Result in the plan that is marked queued with a requestID specified	project
public Integer getProvisioningStatusCheckInterval(WorkflowContext wfc) Compute intervals between status checks for a request (default is 60 minutes)	none
public Integer getProvisioningMaxStatusChecks(WorkflowContext wfc) Compute max number of status checks allowed during a request (default is infinite)	none
public Integer getProvisioningMaxRetries(WorkflowContext wfc) Compute max retries allowed during a request (default is infinite)	none
public Integer getProvisioningRetryThreshold(WorkflowContext wfc) Compute retry threshold (interval between retries) to use for a request (default is 60 minutes)	none

The methods in the libraries below are available for use but will not likely be used in a custom workflow. It is recommended, especially in Release 5.5+ where LCM workflows are subdivided into subprocesses to maximize reusability, that the workflow subprocesses be called by custom workflows instead of the library methods themselves. This information is included here mostly for reference – to document what these methods do and what is passed to them. They are also here to ensure that customization efforts do not *remove* calls to important methods but instead only add other functionality around these method calls.

IdentityRequest Library

Method / Usage	Expected Args (*=required)
public Object createIdentityRequest(WorkflowContext wfc) Create an IdentityRequest object from current workflow	project* flow

context information (tracks status and history of request processing)	source policyViolations
public Object updateIdentityRequestState(WorkflowContext wfc) Modify the IdentityRequest's state	identityRequestId
public Object refreshIdentityRequestAfterApproval (WorkflowContext wfc) Refresh the IdentityRequest to include the provisioningEngine that will handle the item, update the state, and add any expanded attributes that will be provisioned	project
public Object refreshIdentityRequestAfterProvisioning (WorkflowContext wfc) After provisioning, copy interesting task result information back to the IdentityRequest	project
public Object refreshIdentityRequestAfterRetry (WorkflowContext wfc) Go through retry project and update the IdentityRequestItem retry count	project
public Object completeIdentityRequest (WorkflowContext wfc) Mark IdentityRequest status complete, put final plan in request, and refresh the request based on the final project	project policyViolations autoVerify (Boolean)

Approval Library

Method / Usage	Expected Args (*=required)
public SailPointObject getObject(WorkflowContext wfc) Return the object being approved	none
public String getObjectClass(WorkflowContext wfc) Return the simple class name of the object being approved	none
public String getObjectname(WorkflowContext wfc) Return the name of the object being approved	none
public SailPointObject getCurrentObject(WorkflowContext wfc) Return the current persistent version of the object held in the workflowCase (approvalObject)	none
public Identity getObjectOwner(WorkflowContext wfc) Return the current owner of the object being approved (from database lookup)	none
public Identity getNewObjectOwner(WorkflowContext wfc) Return the object owner (in the workflow context – may be different from database-recorded owner)	none
public String getObjectOwnerName(WorkflowContext wfc) Return name of ObjectOwner (from getObjectOwner)	none
public String getNewObjectOwnerName(WorkflowContext wfc) Return name of NewObjectOwner (from getNewObjectOwner)	none
public boolean isOwnerChange(WorkflowContext wfc) Return true if object being approved has had an owner	none

change	
public boolean isSelfApproval(WorkflowContext wfc) Return true if user launching workflow is the same as the owner of the object being approved (used to bypass owner approval – assume will approve if initiating the request themselves)	none

Policy Violation Library

Method / Usage	Expected Args (*=required)
public Object delete(WorkflowContext wfc) Delete current approval object associated with this workflow	none
public Object ignore(WorkflowContext wfc) End the workflow associated with the current approval object without actually doing anything	none
public Object mitigateViolation(WorkflowContext wfc) Mitigate (temporarily allow) a policy violation	expiration* comments
public Object getRemediatables(WorkflowContext wfc)	none
public Object remediateViolation(WorkflowContext wfc) Remediate SOD violations by removing roles named in remediations argument	remediator actor (only used if remediator argument not specified and actor is; use remediator in new method calls) comments remediations*

Role Library

Method / Usage	Expected Args (*=required)
public Object launchImpactAnalysis(WorkflowContext wfc) Start impact analysis task for role in workflow	none
public Object getRoleDifferences(WorkflowContext wfc) Calculate different between role held in workflow and database version of role	none
public Object auditRoleDifferences(WorkflowContext wfc) Create audit events – one for each attribute difference between role states (workflow vs database)	source action target string1
public Approval buildOwnerApproval(WorkflowContext wfc) Set up an approval for the owner of an object (currently used only for Roles)	none
public List<Approval> buildApplicationApprovals(WorkflowContext wfc) For role approval only; build an approval structure for the owners of each application referenced in the role profiles (normally processed as parallelPoll to let application owners submit comments or modify the role but not	

terminate the approval process	
public void enableRole(WorkflowContext wfc) Mark role as enabled	role (name)
public void disableRole(WorkflowContext wfc) Mark role as disabled	role (name)
public void setRoleDisabledStatus(WorkflowContext wfc) Mark role with disabled status indicated in disabled arg (true = disabled, false = enabled)	role (name) disable (Boolean)
public void removeOrphanedRoleRequests(WorkflowContext wfc) Remove incomplete requests to activate/deactivate roles that no longer exist	none
public String getApprovalAuditAction(WorkflowContext wfc) Called by post-approval Audit steps (Audit Failure and Audit Success) of <i>Role Modeler – Owner Approval</i> workflow to determine what type of action should be recorded in audit log; returns “disableRole” if the role is marked as disabled and “updateRole” if it is not	none

LCM Library

At present, the LCM Library contains no public methods. All of its methods have been moved to the Standard Workflow Handler.

Monitoring Workflows

Once a workflow has been initiated, it may run to completion quickly or may take 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 still pending, examine the Task Result for the workflow on the **Monitor -> Tasks -> Task Results** page. The TaskResult for a workflow will exist for a period of time following the successful completion of the workflow, but depending on the retention period set, it may be purged soon after the process has run to completion. While the workflow is still in progress, however, the TaskResult will continue to exist and can be examined for current step and status information.

Dashboard Define Monitor Analyze Manage System Setup			
Tasks			
Tasks Scheduled Tasks Task Results			
Search	Filter by Result Name	Start Date	End Date
Name	Date Complete	Result	Signoff
Update Identity Randy.Knight RolesRequest	Pending...		None
Refresh Groups	9/27/11 1:20 PM	Success	None
Aggregate Enterprise Directory	9/27/11 2:06 PM	Success	None
Refresh Manager Status	9/27/11 2:06 PM	Success	None

Figure 37: Task Results list

Dashboard Define Monitor Analyze Manage System Setup					
Task Result					
Details					
Name	Update Identity Randy.Knight RolesRequest	Started By	Randy.Knight		
Type	LCM	Started	1/16/12 2:52:37 PM		
Description	Workflow Case	Completed			
Status	pending...				
Return to Tasks					
<div style="background-color: #fff9c4; padding: 5px;"> Unable to notify, no email address for: spadmin </div>					
Current Workflow Step: Provision					
Interactions					
Owner	Type	Request	Status	Started	Completed
spadmin	Approval	Owner Approval - Account Changes for User: Randy.Knight	Finished	1/16/12 2:52:42 PM	1/16/12 2:53:22 PM
spadmin	Manual Action	Manual Changes requested for User: Randy.Knight	Open	1/16/12 2:53:28 PM	

Figure 38: Task Result Details (shows status, current step, warning messages, user interactions)

Viewing the Workflow Case XML

The workflow case can also be examined in XML format from the IdentityIQ console or from the debug pages. The status of each step can be determined from the data recorded in the workflow case.

- To get the workflowcase XML from the console, launch the console, list the workflow cases, and get the specific workflow case in question by name (as shown below).

```

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 see the workflowcase XML from the IdentityIQ Debug pages, select **WorkflowCase** from the object list and click **List**. Then click the specific workflow case from the list to display its XML.

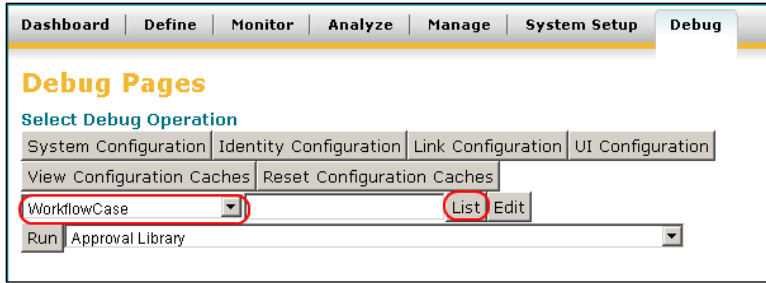


Figure 39: List WorkflowCases

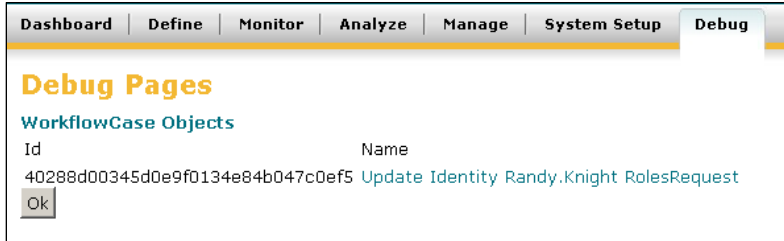


Figure 40: Select desired WorkflowCase

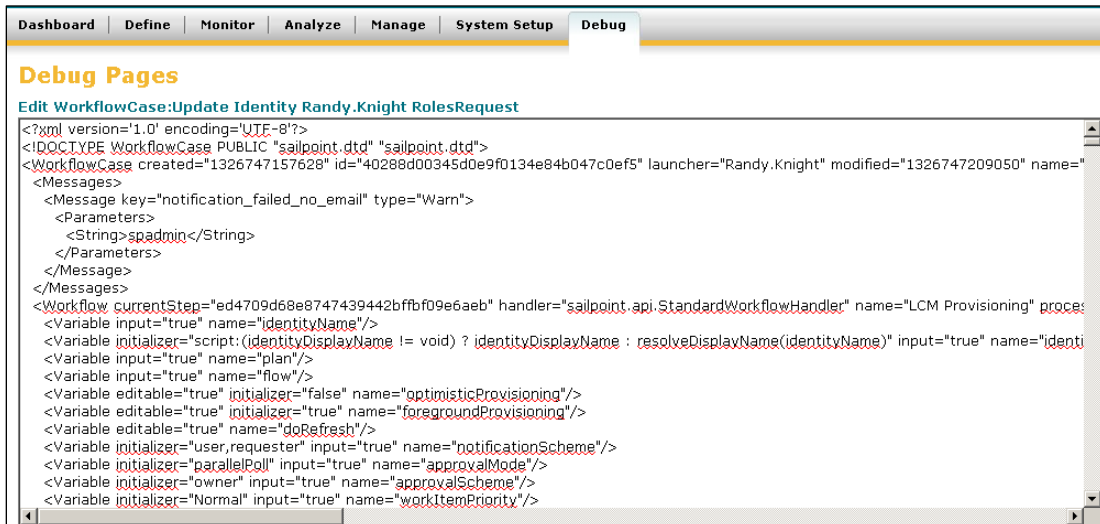
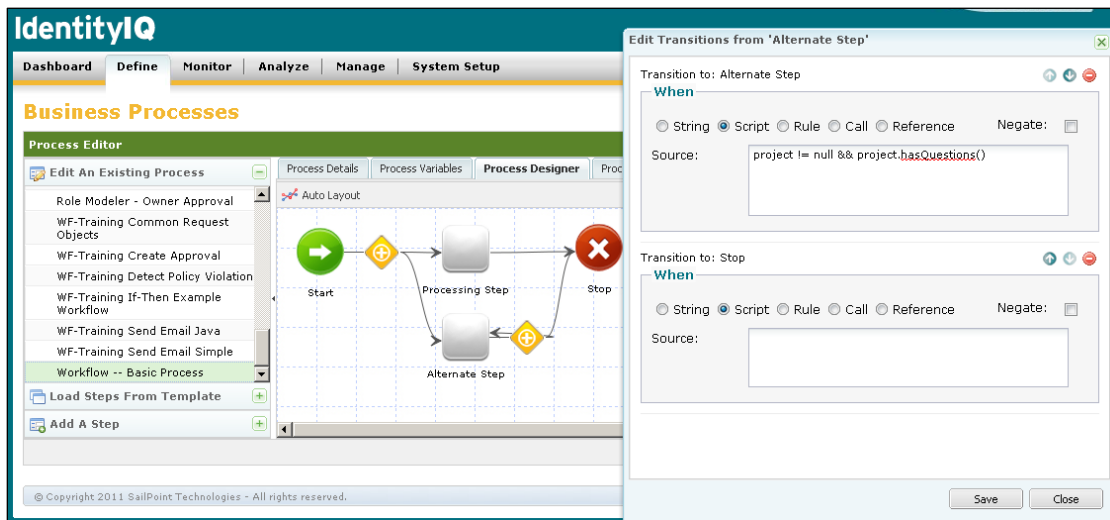


Figure 41: View WorkflowCase XML

Advanced Workflow Topics

Loops within Workflows

Creating a loop within an IdentityIQ workflow is actually fairly simple to accomplish. A transition can be specified from a step back to itself, with the “when” clause on that transition option specifying the looping condition. If the loop needs to encompass a series of steps instead of just one, the transition from the final step must return the workflow to the first step in the set (instead of to itself) when the looping condition is met.



It is also possible to define a subprocess that contains the series of steps that are within the loop. In the main process, the loop is created through a step transition from the step that invokes the subprocess back to itself. In the UI, it will appear as shown above; the step details within the step would specify a “subprocess” action that invokes the loop subprocess.

Initiating Workflows from a Task or Workflow

This section describes how workflows can be run without needing a triggering system event to initiate it.

Workflows Run from Custom Tasks

Workflows can be initiated by a custom Task in IdentityIQ. Tasks are compiled java classes, so the custom task must be written as a java method. Within that method, a workflowLaunch object should be created and populated with the data required by the workflow. A Workflower object must be created, passing in a sailpoint context object, and then the workflower’s launch method is called, passing in the populated workflowLaunch.

The code segment below illustrates how to run a workflow from a java method. This example hard codes most of the required workflow variable values, but they could also be set dynamically by passing in task variables instead of literal values.

```
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, myIdentityName);

//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","spadmin");
launchArgsMap.put("flow","RolesRequest");
launchArgsMap.put("foregroundProvisioning","true");
launchArgsMap.put("identityDisplayName","John.Smith");
launchArgsMap.put("identityName","John.Smith");
launchArgsMap.put("identityRequestId","");
launchArgsMap.put("launcher","spadmin");
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","spadmin");
launchArgsMap.put("source","LCM");
launchArgsMap.put("trace","true");
launchArgsMap.put("violationReviewDecision","");
launchArgsMap.put("workItemComments","");

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

```
//Create WorkflowLaunch and set values
WorkflowLaunch wflaunch = new WorkflowLaunch();
Workflow wf = (Workflow) context.getObjectByName(Workflow.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 Run by Other Workflows

Installations commonly want to have one workflow kick off another workflow. This is done through the use of the `ScheduleWorkflowEvent` method in the Standard Workflow Handler. That method is executed by one of the initiating workflow’s steps through a “call” action.

Arguments to the step should be as follows:

Name	Value
workflow	the name of the workflow to run
requestDefinition	name of the RequestDefinition to launch (required) (RequestDefinitions are XML objects viewable/editable through the Debug pages; specifies the appropriate executor) For requests invoking workflows, the RequestDefinition is “Workflow Request”, which invokes the WorkflowRequestExecutor.
requestName	Name to be assigned to the request (defaults to name of workflow if not specified)
scheduleDate	Only specify if workflow should be initiated on a later date; omit if kicking off immediately
scheduleDelaySeconds	Specifying 1 (with no scheduleDate) runs the workflow after a one second delay – basically immediately
Owner	Workflow owner (usually same as owner of initiating workflow)
caseName	Name for workflowCase (user friendly name -- will be shown in UI; defaults to name of workflow if not specified)
Launcher	Identity requesting the workflow to launch (usually launcher of the initiating workflow)

NOTE: This is distinct from running a workflow as a subprocess. If a workflow is invoked as a subprocess, the calling workflow will wait until the subprocess has finished (and returned control to the caller) before it continues with its processing. This “call” causes a completely separate workflow to begin running, and as soon as the new workflow has been kicked off, the calling workflow will move on to its next step.

Custom Forms

Although there are standard work item forms available for presenting approval (or other data) requests to approvers, some installations may prefer to use custom forms for these activities. In some cases, the request may require a custom form due to the nature of the data collection effort. Custom forms can be built through a <Form> element in the XML, embedded within the <Approval> element.

NOTE: As mentioned in the *Approval Steps* section, the <Approval> element can be used to collect data from a user, even if it is not actually an “approval”, per se. Custom forms are commonly used for these activities, since the normal approval forms do not really apply. However, custom forms can also be used for traditional approval activities when a different presentation format is desired.

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
                    Definition script, and Validation Script)

```

Attributes

Forms can include a map of attributes that are used by the renderer. These are specified by any of the following keys:

Key	Description
pageTitle	Title to render at top of page (typically larger and a different color than the form title)
title	Form Title (shown at top of form body)
subtitle	Form subtitle (shown below title)
readOnly	If “true”, makes form read-only so the fields are rendered as uneditable text or as disabled HTML components
hideIncompleteFields	If “true”, causes any fields that do not have all of their dependencies met to be hidden

NOTE: The Boolean attributes are only specified if they are true; they default to false if not included.

Attributes maps are specified as shown here:

```

<Attributes>
  <Map>
    <entry key="pageTitle" value="Review Non-Employee Request"/>
    <entry key="readOnly" value="true"/>
  </Map>
</Attributes>

```

Buttons

Buttons on the form must include two attributes: a label and an action.

Button Attributes	Description
label	Text displayed on button

action	<p>Indicates how to process the form based on the button clicked.</p> <p>Possible actions are:</p> <p>next: assimilate form data and advance to the next state (OK/Save/Approve/Submit functionality); sets status of approval to “Approved”</p> <p>cancel: stop form editing, return to previous page in UI, and leave work item active (to be dealt with later)</p> <p>back: assimilate form data and return to previous state; sets status of approval as “Rejected” and advances workflows</p> <p>refresh: assimilate the posted form data and regenerate the form; not a state transition – just a redisplay of the form (rarely used)</p>
--------	---

This is an example of a Button element.

```
<Button label='Submit' action='next' />
```

Sections

Sections are marked by the <Section> tag, can be modified by the attributes shown in the table below.

Section Attributes	Description
name	Internal name for section (may be referenced by field objects in some templates)
label	If non-null, label is displayed above section fields; can be text, message catalog keys, or \$() variables
type	<p>Optional rendering type; default (when no type is specified) is a two-column form containing editable fields</p> <p>Other type options are:</p> <p>datatable: two-column table with non-editable fields (for informational tables to give form user context for the form’s requested data)</p> <p>text: block of informational text (each field within form rendered with breaks between them)</p>
priority	Used in form assembly to influence order of the fields

These are examples of Section elements.

```
<Section name="authorizations" label="Authorizations" type="datatable">
<Section name="requestSelector">
```

Sections contain nested Field elements, as described in the next section.

Fields

Fields are, of course, the core element of forms, since they are how data gets communicated to and from the user to which the form is presented. Fields have many attribute options available to them, some of which are commonly used and others of which are used very infrequently.

Commonly used field attributes in custom forms are:

Field Attributes	Description
displayName	Label for the field; may be text or a message key

name	Variable name in which the field's value will be stored
required	Boolean indicating whether the field must be given a value or not (marks field with * on form to indicate required, if true)
type	Field datatype; Valid values are: string, int, long, boolean, date; also Permission, Rule, Identity SailPoint objects (Permission, Rule, Identity) are displayed as drop-down lists. The <i>name</i> of the object, rather than an actual object, should be used as the Field "name" attribute to cause the desired object to be pre-selected in the drop-down list. Boolean fields are rendered as a checkbox.
helpKey	Pop up help text; may be text or a message key
multi	Boolean indicating whether the field is multi-selectable (used along with AllowedValuesDefinition that populates field selection list)

These are the less-commonly used attributes that can be specified for a field:

Field Attributes	Description
filter	for fields where "type" is a SailPointObject subclass, a filter may be specified to restrict the set of selectable objects presented in the drop-down list
allowedValues	List of allowed values for the field; more commonly specified in an AllowedValuesDefinition element nested within the Field element
readOnly	the field is used for information display only
dependencies	list of other fields that must be evaluated first
value	Default/initial value for field (can be specified by string, rule, script, call, or reference; string is default); used within "text" sections to specify the text to display; in data entry fields, this can be overwritten by user
validation	script to validate the value entered by the user
attributes	extensible rendering attributes specific to the type or inputTemplate
priority	a number that can be used to influence the ordering of fields in a section
inputTemplate	Path to an xhtml template used to generate the UI input when a form is being created from a signature

Fields often contain nested elements that help control the display or usage of the field. The field's display can be modified by an Attribute map that specifies the height, width, or field type (most notably "textarea"). Fields can also be created as selectable lists by specifying an AllowedValuesDefinition. Entered Field values can be validated based on a ValidationScript, and a default value for the Field can be specified through an embedded Script element.

Nested Elements within Field Elements	Description
Attributes	Attribute map to control field rendering (primarily used for textareas) Example: <pre> <Attributes> <Map> <entry key="height" value="200"/> <entry key="width" value="450"/> <entry key="xtype" value="textarea"/> </Map> </pre>

	</Attributes> NOTE: Height and width are in pixels. Xtype is used to specify a textarea field type (e.g. for “comments” fields)
Script	Script used to initialize the value of the field (must contain nested <Source> block in which java beanshell code is housed)
AllowedValuesDefinition	Populates a list of values from which the user can select a value for the field; usually contains a <Script> block that specifies the list programmatically, but it’s also possible to specify the list using <String> elements that contain the actual text to display in the field’s drop-down list.
ValidationScript	Script used to examine and validate the field value entered by the user; value entered is given to the Validation Script in the variable named “value” (must contain nested <Source> block in which java beanshell code is housed)

Example of Custom Form XML

This example XML creates a custom form that displays the Identity’s name and asks the user to select a Region to which the Identity should be assigned.

```

<Step name="Need Region" posX="359" posY="182">
  <Approval name="Need Region" owner="ref:launcher" return="region"
    send="identityName">
    <Arg name="workItemDescription"
      value="string:Fill in Region for $(identityName)"/>
    <Form>
      <Attributes>
        <Map>
          <entry key="pageTitle" value="Get Region"/>
          <entry key="title" value="Need Region for Identity"/>
        </Map>
      </Attributes>
      <Button action="back" label="Abort"/>
      <Button action="next" label="Submit"/>
      <Button action="cancel" label="Return Item to Inbox"/>

      <Section name='userInstructions' type='text'>
        <Field value="Employees must be assigned to a region. Please provide the
        correct region for this employee." />
      </Section>

      <Section type="datatable">
        <Field displayName="Employee Name" name="identityName" readOnly="true"/>
      </Section>

      <Section name="Edit These Fields">
        <Field displayName="Region Value" name="region" required="true"
          type="String">
          <AllowedValuesDefinition>
            <Script>
              <Source>
                import java.util.ArrayList;
                import sailpoint.api.*;
                import sailpoint.object.*;

                List regions = new ArrayList();

                QueryOptions qo = new QueryOptions();

                qo.setDistinct(true);
              </Source>
            </Script>
          </AllowedValuesDefinition>
        </Field>
      </Section>
    </Form>
  </Approval>
</Step>

```

```
go.addOrdering("region", true);

List props = new ArrayList();
props.add("region");

Iterator result = context.search(Identity.class, go, props);
while (result.hasNext()) {
    Object [] record = result.next();
    String region= (String) record[0];
    System.out.println("region: " + region);
    regions.add(region);
}
return regions;
</Source>
</Script>
</AllowedValuesDefinition>
<ValidationScript>
    <Source>

        // validation variable comes in as value
import sailpoint.tools.Message;
List messages = new ArrayList();
if(value.length() < 7) {
    Message msg = new Message();
    msg.setKey("New region must be more than 7 characters.");
    messages.add(msg);
}
return messages;

    </Source>
</ValidationScript>
</Field>
</Section>
</Form>
</Approval>
</Step>
```


Document Revision History

Revision Date	Written/Edited By	Comments
March 2012	Jennifer Mitchell	Initial Creation (Current IdentityIQ Version: 5.5)