# Services Standard Framework (SSF) User Guide

# Document Revision History

| Revision Date | Written/Edited By | Comments |
|---|---|---|
| 9/30/2016 | Jennifer Mitchell, Cathy Mallet | Restructured, with updates for SSD v2 |
| 11/23/2016 | Paul Wheeler | Minor modifications for revision 2.0.2 |
| 2/1/2017 | Paul Wheeler | Modification to Provision Processor details for SSD v3 |
| 9/27/17 | Paul Wheeler | Changes based on new version of the Field Value Framework in SSD v5. |
| | | |
| | | |

# Table of Contents

# Overview

The Services Standard Framework (SSF) is a set of reusable IdentityIQ configurations and code objects which can be used to speed up and simplify the deployment process on IdentityIQ implementations. It provides this efficiency by abstracting and reusing common architectures and best practices, eliminating the “nuts and bolts” development that is the same across virtually all implementations and allowing implementers to focus their time and efforts on the specific customer requirements.

The SSF is subdivided into packages which are easy to deploy and extend. The components of each package have been clearly segmented into source code (the nuts and bolts) and configurable extensions, making it easy to understand which components should be deployed as-is and which are intended to be customized per installation. For most use cases, workflow development and low-level, API-centric development is eliminated.

Packages in the SSF are grouped into two categories: **Frameworks** and **Features**.

## Frameworks

**Frameworks** are plug-and-play components which are designed to be reusable across multiple use cases. They each implement a component of a use case, rather than being a complete end-to-end use case in themselves. Some, such as the Field Values Framework, are simple and just offer a cleaner, easier way to solve a common problem. Others, such as the Approvals Framework, are more advanced and come with a full suite of configuration options.

The currently available Frameworks are:

- **Field Value** – Standardizes and simplifies the construction of provisioning policies by decoupling the logic to set each field from the actual policy
- **Role Assignment** – A set of methods that are used to dynamically build IdentityIQ account requests to assign roles based on the Identity Selectors of business roles
- **Dynamic Emails** – A single subprocess that can be called from any workflow that simplifies and queues the sending of emails
- **Approvals** – A single subprocess and set of configuration options that dynamically and iteratively processes approval types required for a given use case
- **Provision Processor** – A set of subprocesses that handle the basic steps common to all provisioning workflows, which can be simplified to four simple steps: Build Plan, Get Request Type, Call Provision Processor Subprocess, and Send Emails

## Features

**Features** are complete but configurable end-to-end use cases. Like the Frameworks, Features offer a variety of configuration options. They execute common use cases in their core steps and include “hooks” for additional customizations. The SSF Features leverage the Frameworks to implement their use cases.

The SSF offers several Lifecycle Event (LCE) features, and one UI feature.

**Lifecycle Event Features** are provided as end-to-end solutions comprising the lifecycle event (Identity Trigger), the trigger rule, and the workflow. They each sit on top of a standard rule library and are configured via a handful of properties, a configuration mapping object, and a single customizable rule library. They each include:

- options for determining how to trigger the workflow and how to build the provisioning plan
- hooks for configuring email options and required approval scenarios
- three strategic hooks (before build plan, before provision, after provision) for additional installation-specific workflow requirements

The currently available Lifecycle Event Features are:

- **Joiner** – Used for onboarding of new personnel
- **Leaver** – Used for terminations and/or a leave of absence
- **Attribute Synch** – Used for pushing target attribute changes to other application accounts
- **Mover** – Used for handling transfers within the organization
- **Rehire** – Used to process previous-terminated users who have returned to the company

**UI Features** are workflows available to end users and administrators through the IdentityIQ user interface. They use QuickLinks, and have additional configuration options to control scoping and show configurable forms, as well as configuration options similar to LCE Features. These include options for building the provisioning plan, “hooks” for configuring email options and the required approval scenarios, and the same strategic rule “hooks” as found in the Lifecycle Event Features.

The currently available UI Feature is:

- **Terminate Identity** – Allows authorized users to do immediate terminations from the IdentityIQ user interface

## A Note on IdentityIQ Versions 7.0 and Higher

The first version of the SSF was designed for the 6.2 and 6.3 releases of IdentityIQ. In the version of the SSF packaged with SSD v2, updates were included to make the SSF compatible with IdentityIQ 7.x. All functionality should work as documented here. However, some major updates were introduced to workflows and approvals in IdentityIQ 7.0, namely the workflow step-splitting option, which allows for doing parallel/asynchronous approval/provision processes. This new feature has **not** been incorporated into the latest version of the SSF. The approval framework is a separate entity that can be called from a custom workflow, but it is also the underlying approval subprocess in all SSF Features. These facts have some key implications to consider when using the SSF in 7.x.

1. If approvals are being used in a great number of scenarios and the new split-step feature is needed in all or most of them, simply don’t use the SSF Approval and Provision Processor frameworks, or any of the SSF Features.
2. A more likely scenario is that the split-step feature is needed for request access — whereby a requester can select multiple entitlements and roles and wants them processed ASAP after approval, i.e. doesn’t want to wait on the whole request to be approved before provisioned —

but otherwise, approvals aren't really a factor in other workflows, especially LCE workflows that rarely require approval. If this is the case, use the LCE Features, but configure your own Request Access workflow using the latest split-step feature in IdentityIQ 7.x.

3. If this split-step feature isn't important for your installation, you can use the SSF in its entirety.

## Understanding the Framework and Feature Structures

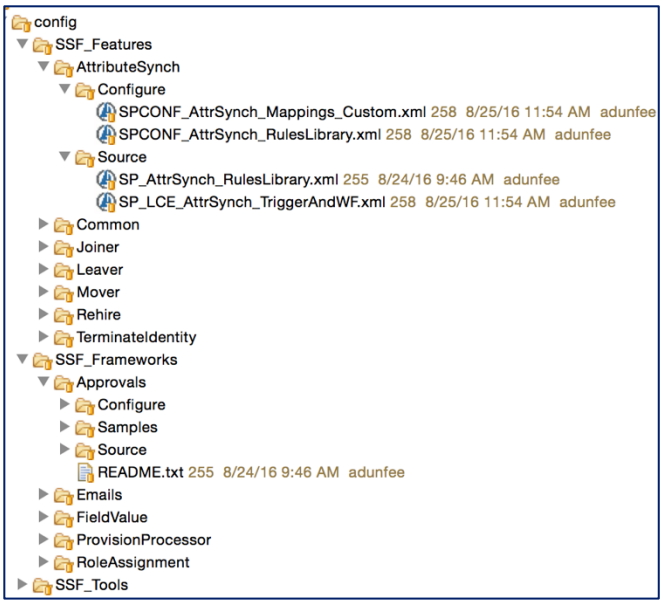
Each Framework or Feature is provided in a standard folder structure, including some or all of these subfolders:

- **Configure** folder – contain objects that should be configured per installation to implement the feature as required by that customer
- **Source** folder – contains source code objects which should be deployed as-is (i.e. should never be edited per installation)
- **Samples** folder – contains examples of how to use or call the feature or framework
- **README.txt** file – contains brief instructions on how to use and configure the feature or framework

The SSF also includes two properties files at the base level of the SSD folder structure, which specify important values to include in the corresponding properties files used by the SSB's ant build process:

- **ssf.target.properties**: a Target Properties file which specifies key variable substitutions and other options specific to the SSF objects; these are used by the build process to enable or disable the Framework or Feature, to set options, and to correctly deploy the artifacts for each environment
- **ssf.ignorefiles.properties**: an Ignore Files Properties file which explicitly tells the build process which file to omit from the build; specifically, this properties file says to ignore all files located under any /Configure folder so that only those framework and feature configuration files which have been copied out to the SSB project's /config folder will be deployed. (Refer to the SSB User Guide for more information on its folder structure.)

The ant scripts and configuration files in the SSB structure look for an [environment].target.properties file and an [environment].ignorefiles.properties file (for example, sandbox.target.properties or prod.ignorefiles.properties) as the build process is executed. These two ssf.*.properties files include SSF-specific target configurations and sets of files to ignore; the contents of these files should be copied into target.properties files and ignorefiles.properties files for each environment to be built. The target.properties values may need to be altered per environment if environment-specific values are required.




**SSF Folder Structure**

```
#*********************************************************************************
#   JOINER FEATURE PROPERTIES
#*********************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_JOINER_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_JOINER_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_JOINER_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_JOINER_SEND_POST_PROVISION_EMAILS%%=false
```

**Target Properties**

```
custom/SSF_Features/AttributeSynch/Configure/SPCONF_AttrSynch_Mappings_Custom.xml
custom/SSF_Features/AttributeSynch/Configure/SPCONF_AttrSynch_RulesLibrary.xml
custom/SSF_Features/Common/Configure/SPCONF_LCE_Common_RulesLibrary.xml
custom/SSF_Features/Joiner/Configure/SPCONF_Joiner_Mappings_Custom.xml
custom/SSF_Features/Joiner/Configure/SPCONF_Joiner_RulesLibrary.xml
custom/SSF_Features/Leaver/Configure/SPCONF_Leaver_Mappings_Custom.xml
custom/SSF_Features/Leaver/Configure/SPCONF_Leaver_RulesLibrary.xml
custom/SSF_Features/Mover/Configure/SPCONF_Mover_Mappings_Custom.xml
custom/SSF_Features/Mover/Configure/SPCONF_Mover_RulesLibrary.xml
custom/SSF_Features/Rehire/Configure/SPCONF_Rehire_Mappings_Custom.xml
custom/SSF_Features/Rehire/Configure/SPCONF_Rehire_RulesLibrary.xml
custom/SSF_Features/TerminateIdentity/Configure/SPCONF_TerminateIdentity_Forms.xml
custom/SSF_Features/TerminateIdentity/Configure/SPCONF_TerminateIdentity_Mappings_Custom.xml
custom/SSF_Features/TerminateIdentity/Configure/SPCONF_TerminateIdentity_RulesLibrary.xml
```

**Ignore Files Properties**

# Quick Start

The Services Standard Framework (SSF) is now part of the Services Standard Deployment (SSD), which combines all things needed for a customer implementation, including the SSF, the Services Standard Build (SSB), project documentation (SSW), performance tools (SSP) and testing framework (SST). The SSD aims to standardize and simplify implementations and can be setup with a few simple steps. The SSF is meant to be deployed within the infrastructure of the SSB, though it is possible to use it without the SSB structure. This document first describes the processes for using it with the SSB and then explains how to use it independently.

## SSB Build Process: New Installation

These are the quick-start steps to use the SSF with a clean build (i.e. in an installation where the SSB environment is not yet established and IdentityIQ has not yet been are installed and initialized). **NOTE:** You can use the SSD Deployer tool to automate steps 5 and 6. See the SSD Deployer User Guide that is included with the SSD.

1. Create your source control repository.
2. Check out the project (usually in Eclipse).
3. Copy the unzipped SSD into the trunk folder of your project.
4. Set up the SSB, as described in the SSB User Guide documentation.
5. Deploy the SSF:
    a. Create the <environment>.target.properties file at the root of the SSD for each environment
    b. Copy the values in ssf.target.properties to your <environment>.target.properties files for each environment
    c. Create the <environment>.ignore.files.properties file at the root of the SSD for each environment
    d. Copy the values in ssf.ignorefiles.properties to your <environment>.ignorefiles.properties for each environment
    e. To turn on the SSF, set the **deploySSF** property to **true in the build.properties file.**
6. Determine which Frameworks and Features of the SSF you will be implementing and copy the files out of those packages' /Configure folders into the appropriate subfolder of the SSB /config folder.
    a. The SSB /config folder is usually divided into subfolders for each object type, so rules go in the /config/rules folder, etc.
7. Configure your business requirements in the appropriate configurable SSF components in the SSB /config folder.
    a. Enable the component(s) via <environment>.target.properties
    b. Set options in the SPCONF <Framework/Feature Name> Custom Mapping object
    c. Write code and other extensions in the SPCONF <Framework/Feature Name> Rules Library object
8. Run the ant build to build a deployable WAR file that includes your customizations (including the configured Frameworks and Features as well as any other configuration objects applicable to your implementation).
9. Deploy your code to the desired IdentityIQ instance.

10. Repeat steps 6-9 as needed to deploy additional components or implement additional features/frameworks.

## SSB Build Process: Existing Infrastructure

In cases where the SSB is already in use and the IdentityIQ environment is already established, the SSF may still be integrated into the SSB structure to leverage its deployment-accelerating capabilities. These are the quick-start steps to use for that option:

1. Copy the SSF_Features and SSF_Frameworks folders from the unzipped SSD into the /config folder of your SSB project.
2. Execute steps 5-10 of the process described above.

## Non-SSB Build Process

Some customers may choose to use the SSF's features and frameworks when using a process other than the SSB for their build process. In this case, the contents of the /Source and /Configure folders of each feature in the SSF_Features folder or each framework in the SSF_Frameworks folders should be deployed into whatever build process has been chosen for the installation. The key points to remember when doing this are:

1. The contents of the /Source folder for each framework or feature should be deployed into the IdentityIQ installation as-is with no modifications.
2. The contents of the /Configure folder for each framework or feature should be customized for the installation before deploying them.

# Frameworks

**Frameworks** are plug-and-play components which implement a component of a use case, and can be re-used across different use cases within an installation. This section describes each framework in detail.

## Field Value

The purpose of the Field Value framework is to consolidate the mappings and logic for Field Values used in Provisioning Policies by using a Custom object for the mapping and a single Rule Library for the custom logic. This consolidates artifact management, promotes code reuse, simplifies logging, and more.

It can be used for all provisioning policy fields that return values from the Identity Cube or derive their value through BeanShell or Velocity logic. It cannot be used for dynamic fields that require user input or fields that reference another field in the provisioning policy. However, a hybrid approach is possible, whereby those dynamic or dependent fields can use embedded scripts while other fields in the same provisioning policy can use the framework.

Note that versions of the SSD before version 5 used a different approach to this framework. This was re-architected in version 5 to improve performance and scalability, and to add new enhancements. However, the changes have been designed with backward compatibility in mind, and if the Custom object introduced in the new version of the framework does not exist or if there is no mapping for an application, the framework will work in "legacy" mode, where methods in the custom rule library based on the name of the application and field will be used. These legacy methods must follow the naming convention "getFV_<ApplicationName>_<FieldName>_Rule".

### Advantages

The Field Value Framework provides these benefits to customers and implementers:

- **Fewer Java imports** – Collecting all logic in a single library cuts down on the need to declare the various import statements for each script or field value rule (i.e. they can be done once in the rule library for all the methods to use instead of repeatedly in each independent script or rule).
- **Improved logging** – All components in this framework use a common log4j logger (rule.SP.FieldValue.RulesLibrary). Using a single log4j declaration makes it more likely that logging will be done correctly with proper trace levels, versus using temporary System.out.println() statements.
- **Fewer rule objects** – Replaces all of the field value rules with one custom library.
- **Decouple policy from logic** – Keeps the application definition cleaner and makes it easier to read and edit the fields.
- **Reuse common logic** – Fields that derive the same value (even across different applications) can draw from common methods.
- **Better collaboration** – Can tokenize versions of the library per environment, allowing, for example, one developer to create a stub version with simple return values and another developer to work on the real logic.
- **Flexibility** – field values can be defined using direct mappings, rules, scripts or Velocity syntax.

## Components

This framework includes four objects, provided in four separate XML files. Two of these should be used as provided (not edited or modified per installation) and the other two are where the installation-specific logic resides.

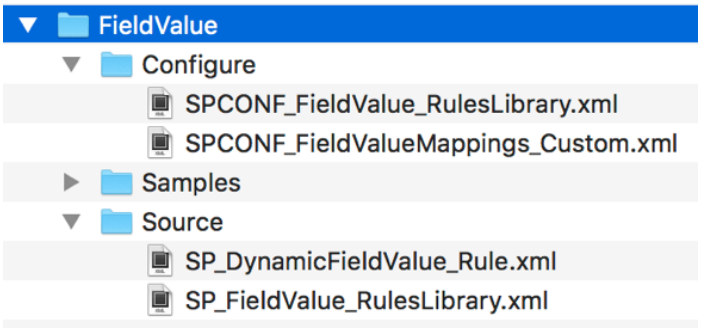
These core framework components *should not* be modified or customized per installation:

- **SP Dynamic Field Value** rule – referenced by all fields in the provisioning policy; provided in SP_DynamicFieldValue_Rule.xml
- **SP Field Value Rules Library** – contains a number of methods used by the framework; provided in SP_FieldValue_RulesLibrary.xml

These framework components *should* be customized per installation:

- **SPCONF Field Value Mappings Custom** – template Custom object containing mappings of attributes as lists of Fields for each application; provided in SPCONF_FieldValueMappings_Custom.xml
- **SPCONF Field Value Rules Library** - template rule library which will be edited per installation to contain field-specific methods for setting the field values; provided in SPCONF_FieldValue_RulesLibrary.xml

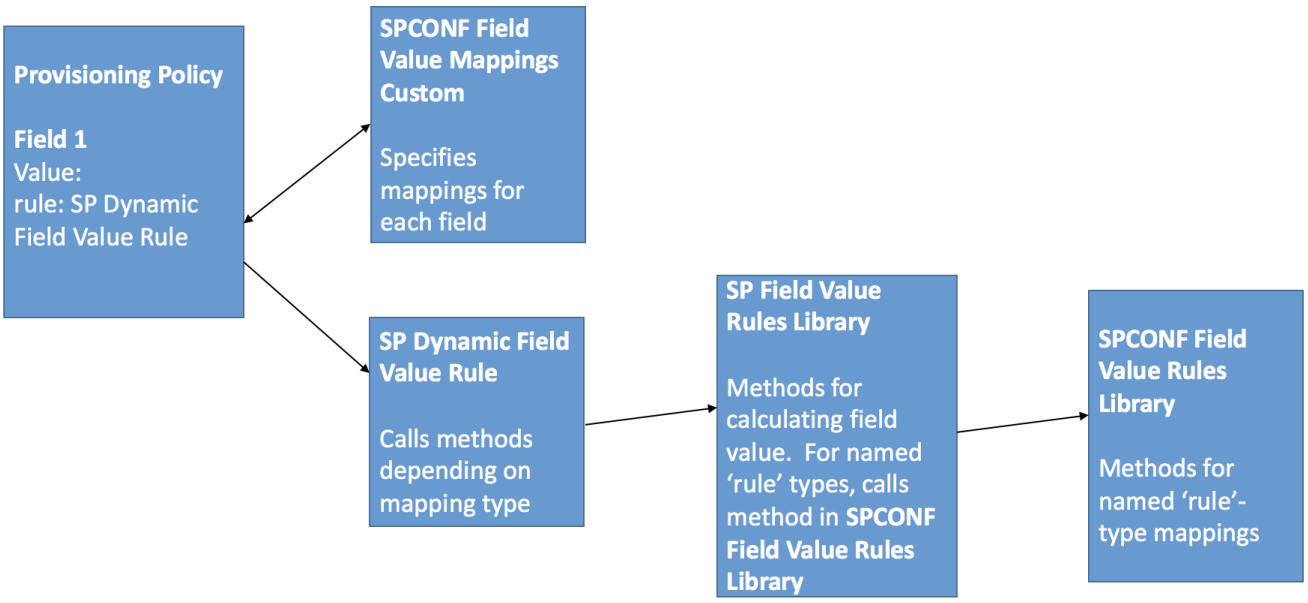
The framework components are provided in the SSF in this structure:




**Field Value Framework Folder Structure**

## How It Works

1. Application provisioning policies set each field value through a call to a single rule: SP Dynamic Field Value Rule.
2. That rule evaluates the Custom object (SPCONF Field Value Mappings Custom) to retrieve the mapping for the field for the specific application, which could be in the form of a direct attribute mapping, rule, script or velocity statement.
3. The rule calls a method in SP Field Value Rules Library depending on the type of mapping:
    - For direct attribute mappings the mapped identity attribute is returned.
    - For velocity statements the rendered value of the statement is returned.
    - For a script, the script source is executed and the result returned.
    - For a rule, a method with the referenced name is called from SPCONF_FieldValue_RulesLibrary.xml and the result returned.
4. The returned value from SP Dynamic Field Value Rule is used as the value of the field in the provisioning policy.




## How to Implement

- Define the application Provisioning Policy fields. For each field:
  - Set the value of the field through a rule reference to SP Dynamic Field Value Rule.
  - Add an **application** attribute to the Field definition. This is used to construct the field-setting method name. Often, this specifies the actual application name (such as "Active Directory") but sometimes a common rule is used to set same attribute on multiple applications. In that case, this may be something like "common" or "general".
  - Add a **template** attribute to the Field definition with the value set to the type of Provisioning Policy (e.g. Create or Update). This value will be passed to the field-setting method in the "op" argument. There may be different logic for setting a field in a create vs. update provisioning policy, and the field-setting method can use this argument to drive that variation.

```
<Field application="Active Directory" template="Create"
displayName="employeeID" name="employeeID" type="string">
      <RuleRef>
          <Reference class="sailpoint.object.Rule" name="SP
Dynamic Field Value Rule Active_Directory"/>
       </RuleRef>
</Field>
```

- Edit the supplied sample **SPCONF Field Value Mappings Custom** object. This should contain entries for each application, as referenced by the 'application' attribute in the Field definition. The value for each entry is a list of Fields. Each Field must have the following properties:
  - **name**: this should be the same as the name of the corresponding Field in the Provisioning Policy.
  - **type**: one of the following values:
    - **attribute**: for a direct mapping to an identity attribute

        - **rule**: for a referenced method in **SPCONF Field Value Mappings Custom**, or a Script contained in the expanded value of this Field
        - **velocity**: for a statement in Velocity syntax that can be in the single-line value of the Field, or using a Script in Velocity syntax in the expanded value
    - **value**: the name of the identity attribute for an **attribute** type, a method name or a self-contained Script for a **rule** type, or a Velocity statement (or expanded to a Velocity Script) for a **velocity** type

  Optionally, for a Field where the type is **rule**, the Field may also have a **defaultValue** property. This will then be available for use in a method in **SPCONF Field Value Rules Library** or in a Script contained in the expanded value of the Field.
  For examples, see the template Custom object that is included in the framework in the file SPCONF_FieldValueMappings_Custom.xml.
- For mappings of type **rule** that do not have self-contained Scripts, write the methods for calculating the value for each field in the **SPCONF Field Value Rules Library**.
    - Method names must be the same as the **value** property of the mapping in **SPCONF Field Value Mappings Custom**.
    - Methods must return an object of the same type as defined in the Provisioning Policy. Multi-valued attributes should be returned as a List.
    - If you are not using the defaultValue option for the Field, the method should accept only SailPointContext, Identity and String as arguments (in that order), with the String being the operation type (e.g. Create or Update).

```
public String getCustomOffice(SailPointContext context,
Identity identity, String op) {
   String office = identity.getStringAttribute("location");
   logger.debug("Returning " + office);
   return office
}
```

    - If you are using the default value option, your method must accept a fourth argument which will be a String representing the default value.

```
public String getCustomOffice(SailPointContext context,
Identity identity, String op, String defaultValue) {
   String office = identity.getStringAttribute("location");
   if (null == office)
      office = defaultValue;
   logger.debug("Returning " + office);
   return office
}
```

- Move the files **SPCONF_FieldValue_RulesLibrary.xml** and **SPCONF_FieldValueMappings_Custom.xml** from the framework's /Configure folder into the /config/rules folder in the SSB folder structure (if you have not already done this with the SSD Deployer). The objects from the framework's /Source folder are automatically deployed with the SSB.
- Import the framework artifacts through the SSB deployment process (or other process).

# Dynamic Emails

The Dynamic Emails Framework is a single subprocess for simplifying the queuing and sending of emails. It can be called from any workflow that needs to send emails to system users. It requires the workflow to provide a list of maps (in the variable emailArgList), with each map containing all of the attributes required to send a single email. Each map represents an email that needs to be sent.

This framework is intended for any workflow that needs to send out multiple emails that do not necessarily need to be sent at a specific step in the workflow process. The contents of each email, as well as the template and recipient, are distinct and independent of each other. The framework is not intended for the approval, reminder or escalation emails sent out on a per-approval step, as those emails need to be sent with the specific step to notify a person of a required action they must take to allow the process to progress or complete.

This framework simplifies workflows by eliminating the need to call a send email step for every required email, which can require an unwieldy number of email workflow steps. It allows various workflows (subprocesses) and rules used throughout the workflow process to add emails and their required attribute values to the email list, eliminating the need to pass each of the variables needed for the email messages back and forth between sections of code. Then, when it is time to send the emails to end users, the framework processes the list and sends the messages all at once.

## Advantages

The Dynamic Emails Framework provides these benefits to customers and implementers:

- **Reduces email steps** – By appending to a single list, all emails can be sent out in one step.
- **Easier access to variables** – In standard send email scenarios, each variable that the template needs has to be passed into the step. In this case, the email args can be built out dynamically in underlying rules and referenced with a simple reference, $emailArgs.key.

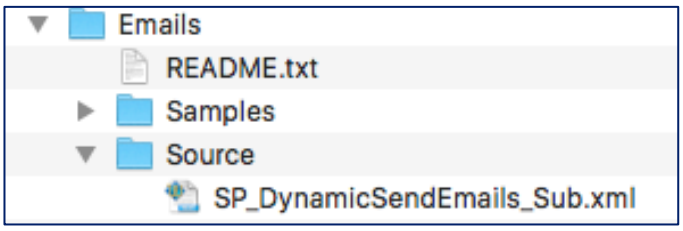
## Components

This framework uses one object which should not be customized per installation:

- **SP Dynamic Send Emails Sub** – workflow which sends the emails according to the map provided in the emailArgList argument; provided in SP_DynamicSendEmails_Sub.xml

There are no custom components required for this framework, other than any custom email template(s) you create for use with this framework, so there is no Configure folder in this Framework.

The framework components are provided in the SSF in this structure:




**Email Framework Folder Structure**

## How It Works

1. Your workflow, or any subprocesses, rules or rule libraries used by it, builds a list ("emailArgList") of maps, with each map representing an email to be fired by the workflow.
    a. The maps include at minimum a "to" value indicating the email recipient, and an "emailTemplate" value designating the email template to use.
    b. It should also include all other variables required to populate variables in the email template.
2. At the end of the workflow, or at any point when you are ready to send all of the queued emails, the workflow calls the **SP Dynamic Send Emails Sub** subprocess, and passes the emailArgList (list of maps) to this subprocess.
3. The **SP Dynamic Send Emails Sub** subprocess iterates through the list of maps, kicking off an email for each map. (Emails are generated using the sendEmail() workflow library method.)

## How to Implement

All steps to implement this framework are done in the workflow which calls the framework subprocess. No customization or configuration is required or expected inside this framework.

1. In your workflow, build and append to the emailArgList. This can be done in any code step, such as a script or a rule. The emailArgList can be appended to by multiple steps throughout your workflow, as needed.

    The emailArgList should be built as an ArrayList like this:

    ```
    List emailArgList = new ArrayList();
    ```

    For each email to be sent, add a map object to the emailArgList. The map must at minimum contain **to** and **emailTemplate**. You can add additional keys as needed for each specific email message.

    For example:

    ```
    Map emailArgs = new HashMap();
    emailArgs.put("to", "admin@sailpoint.com");
    emailArgs.put("emailTemplate", "cst Security Officer Termination Email");
    emailArgs.put("identityName", identityName);
    emailArgs.put("someOtherVal", "this is a test");
    emailArgList.add(emailArgs);
    ```

2. Create any email template(s) that the workflow will use (i.e. any email template specified as the emailTemplate attribute in any of the maps).

    Email templates for emails to be sent using this framework will be provided the emailArgList variable values as a map in the variable "emailArgs", so all variables in the template should be referenced in the email template as $emailArgs.[key].

    For example, the key "`someOtherVal`" from the example above can be added to the template's body or subject as `$emailArgs.someOtherVal` and when the email is sent, the value presented in the email text will be "this is a test".

Here is a basic example of an email template which could be used by the framework:

```
<EmailTemplate name=”Example” >
   <Subject>Access granted to $emailArgs.targetUser</Subject>
   <Body>$emailArgs.requester requested the following access for
$emailArgs.targetUser:
…
</Body>
</EmailTemplate>
```

3. Edit your workflow to add a step which calls the **SP Dynamic Send Emails Sub** subprocess and passes in the variable **emailArgList**.
    - The workflow is also configured to accept **identityName** and **identityDisplayName** as arguments. If identityDisplayName is provided, it is passed to the email template along with the emailArgs map. If identityName is provided and identityDisplayName is not, identityName is used to retrieve the identity's displayName attribute or it is used *as* the identityDisplayName value.

    For example:

```
<Step name="Send Emails">
  <Arg name="emailArgList" value="ref:emailArgList"/>
  <Arg name="identityName" value="ref:identityName"/>
  <Description> Call the Dynamic Email framework subprocess to send collected
emails.
  </Description>
  <WorkflowRef>
      <Reference class="sailpoint.object.Workflow" name="SP Dynamic Send Emails
Sub"/>
  </WorkflowRef>
  <Transition to="Stop"/>
</Step>
```

4. Import the framework artifacts through the SSB deployment process (or other process).

# Role Assignment

The Role Assignment framework lets an implementer set up birthright roles with assignment logic, and then build a role-provisioning account request to provision the appropriate set of roles for a user with a single method call.

The framework is a set of methods that are used to dynamically build IdentityIQ account requests in order to assign roles based on the Identity Selector defined on the business roles. This supports common use cases such as the Joiner or Mover processes, where the implementer needs to do role provisioning in a workflow.

The logic for role assignment lives in the roles, and the logic for building an account request is a single line (method call) that can be added to any workflow or rule.

Using roles with assignment logic specified in them to drive access provisioning is generally more efficient, and flexible, than building provisioning plans for access provisioning directly in the workflow steps.  Having an easy mechanism for evaluating the role logic in a workflow means you can include approvals, notifications (including emailing new credentials to users), and other workflow-facilitated options in your role provisioning process, in contrast to using an identity refresh task-based process for role provisioning, which does not offer those options.

## Advantages

The Role Assignment Framework provides these benefits to customers and implementers:

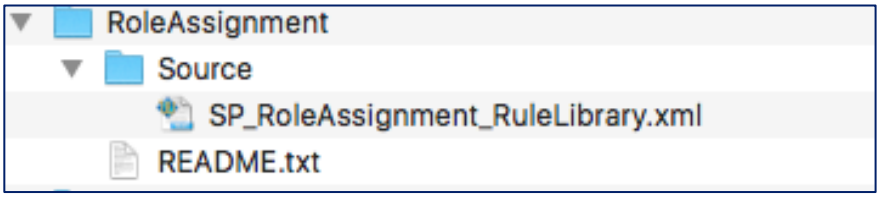
- **Assignment logic in roles –** Removes all of the assignment logic from the workflow and allows for rapidly changing and adding roles.
- **Provisioning in workflows –** Allows for handling approvals, sending credentials emails, or doing other steps based on the accounts provisioned.
- **Greater control of success/failure –** Allows for workflow provisioning retries, which is absent from the identity-refresh based role assignment process.

## Components

This framework contains only one object, which does not need to be modified per installation:

- **SP Role Assignment Rule Library –** contains the logic to evaluate the role assignment logic for all birthright roles and create a provisioning account request for the appropriate roles; provided in SP_RoleAssignment_RuleLibrary.xml

The framework components are provided in the SSF in this structure:




**Role Assignment Folder Structure**

## How It Works

This framework is made up of only one rule library. Its methods must be invoked by a workflow – such as your own custom workflow or one of the SSF Feature workflows described in this document – to do its calculations.

This framework looks at a list of roles defined within a specified Organizational role, evaluates their assignment logic, and recalculates a target identity's access based on that logic. It can either perform:

- only adds (i.e. it will add roles for which the user meets the criteria but will not remove any existing roles) or
- both adds and removes (i.e. it evaluates all roles and removes any which the user should no longer have according to the role's assignment criteria while adding any they should have that they do not already have).

In either mode (add only or add/remove), the method creates and returns an accountRequest to add roles to (or remove roles from) the user's IdentityIQ account. The calling workflow must add the account request to a ProvisioningPlan and provision it.

## How to Implement

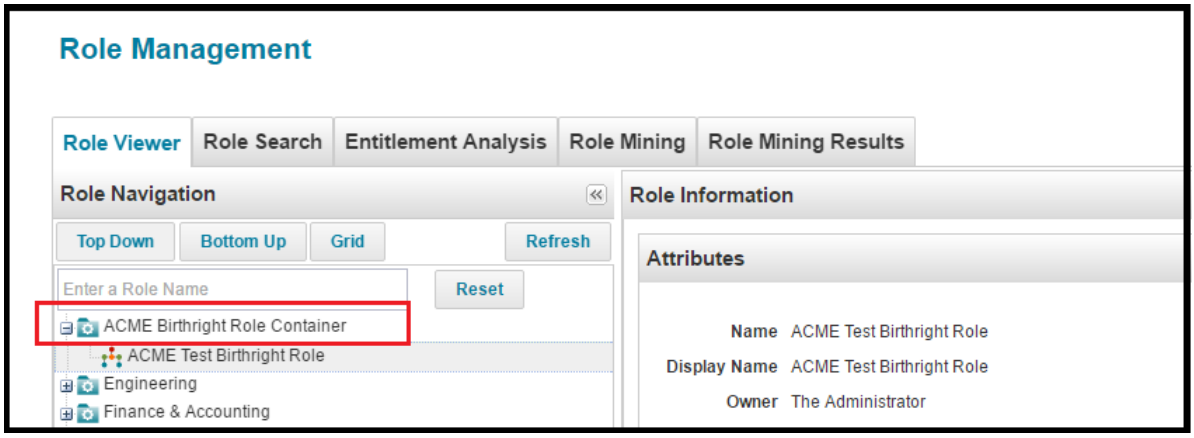
These are the basic steps for implementing this framework. Each step is described in greater detail in the next sections.

1. Create an organizational role to act as a container for all birthright roles
2. Create a new role type for birthright roles
3. Create the birthright roles as needed, including defining the assignment rule logic for them
4. Add the reference to the **SP Role Assignment Rule Library** in the workflow from which you are invoking this framework
5. Call the method in the **SP Role Assignment Rule Library** from your workflow, to get the account request or account requests
6. Set the target properties for this framework for your deployment environment
7. Deploy the framework

### Step 1 - Create a container organizational role

The framework looks only at assignable roles defined within a single organizational role for birthright provisioning. This is done to avoid having to scan every single role in the system to find assignable roles.

Create a birthright roles Organizational role. The name of the organizational role can be any name you choose. You will specify the name of the organizational role in the target properties file for your build.

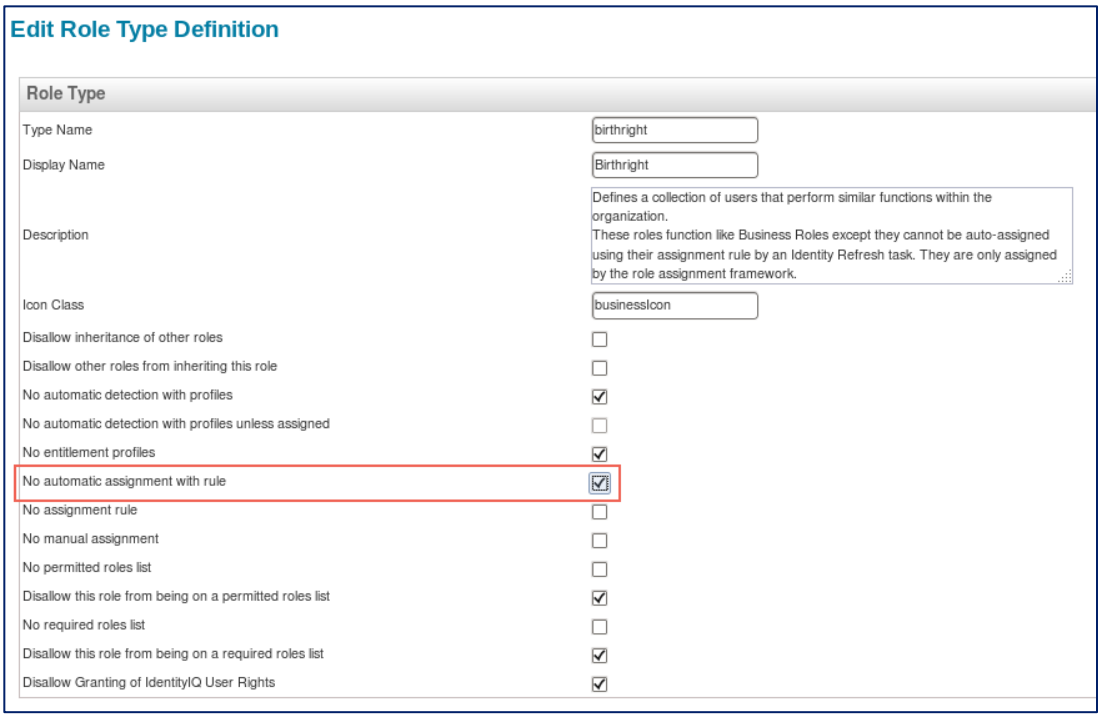



**Role Assignment Organizational Role**

## Step 2 – Create a birthright role type

The roles you want to assign as birthright roles, or process through the automatic add/removal option, should ideally *only* be assigned by this framework, not by a normal identity refresh task with *the Refresh assigned, detected roles and promote additional entitlements* option selected. This is because this framework is used by many of the SSF Features (joiner, mover, etc.), and implementers typically want to enforce the Lifecycle Event's defined process for these role assignments, which the identity refresh task would not do.

To ensure that these roles are only processed by this framework and not the refresh task, configure a **Birthright** role type to match exactly the definition of a Business role *except* select the **No automatic assignment with rule** option. This allows the assignment rules to still be there for this framework to use but prevents identity refresh tasks from assigning the roles using that logic.



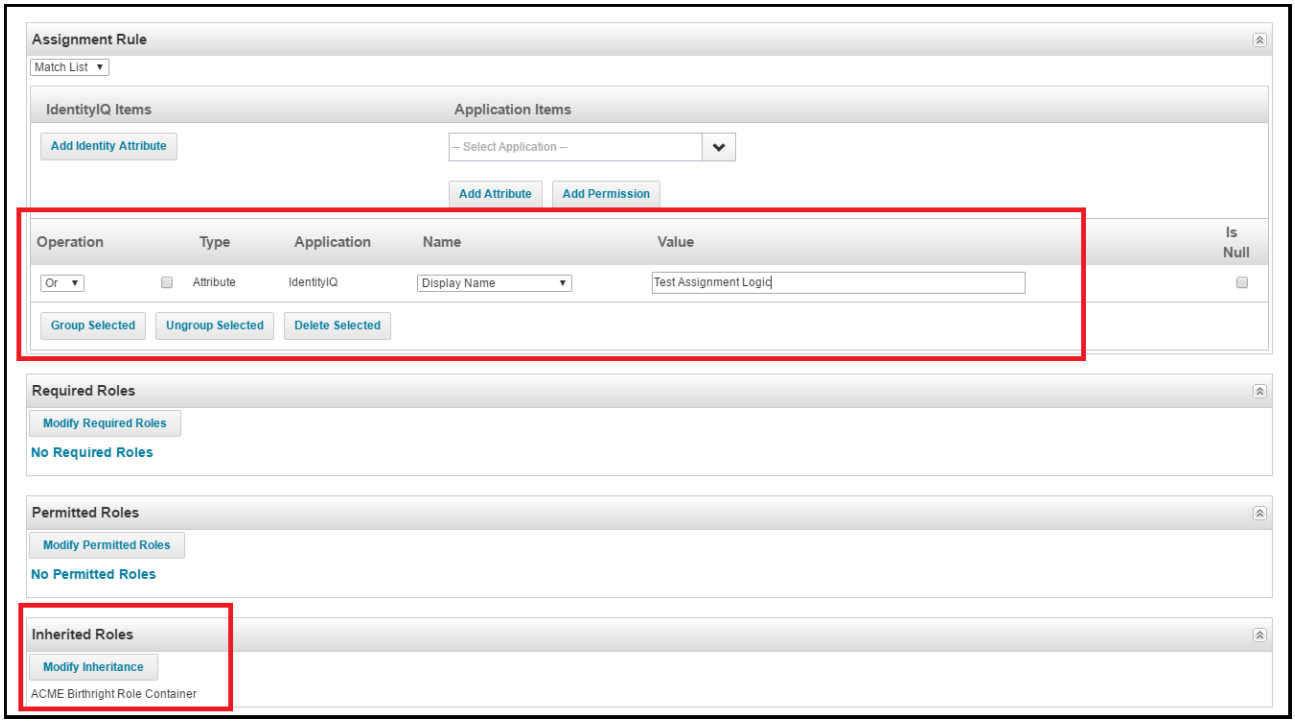

**Birthright Role Type**

**NOTE**: This step is not strictly required – there are other ways this can be managed, such as constraining identity refresh tasks so they do not do role assignment against any identities which have not yet been processed by the appropriate lifecycle events. However, this is the easiest, and the recommended, option for managing that constraint. The role type can be named according to customer preferences.

### Step 3 – Create the birthright roles

For each role that will be assigned by this framework, modify the inheritance and select the organizational role created in Step 1. Only roles which are contained within that organizational role will be examined by this framework.

Add the required assignment logic so this role can be auto-assigned to users by this framework.




**Role Assignment Business Role**

## Step 4 - Add a rule reference to the SP Role Assignment Rule Library

To call the methods in a rule library from a rule, rule library method, or workflow, the rule library must be referenced in that rule or workflow.

If you are calling the framework's methods from a rule or a rule library, include a <ReferencedRules> element which references the **SP Role Assignment Rule Library** rule object:

```
<ReferencedRules>
      <Reference class="sailpoint.object.Rule" name="SP Role Assignment
      Rule Library"/>
</ReferencedRules>
```

If you are call the methods from a workflow step, add a reference to the rule library in the RuleLibraries element:

```
<RuleLibraries>
      <Reference class="sailpoint.object.Rule" name="SP Role Assignment
      Rule Library"/>
</RuleLibraries>
```

## Step 5 - Call the method to get the account request or account requests

Your workflow (or rule or rule library method) must call one of two methods in the SP Role Assignment Rule Library:

1. **getAddOrRemoveRolesAccountRequest** – creates a request that includes adds and removes
2. **getBirthrightRolesIIQAccountRequest** – creates a request that only include adds

Here is an example of building the provisioning plan using the framework to build the role-assignment account request (this logic would appear in your workflow, rule, or rule library method):

```
ProvisioningPlan plan = new ProvisioningPlan();

plan.setIdentity(identity);

logger.trace("Get the roles request");

AccountRequest rolesReq =
getAddOrRemoveRolesAccountRequest(context,identity);

if (rolesReq != null){
      logger.trace("Add the roles req");
      plan.add(rolesReq);
}
```

The plan created from that example would look like this:

```
<ProvisioningPlan>
      <AccountRequest application="IIQ" nativeIdentity="jsmith" op="Modify">
            <AttributeRequest name="assignedRoles" op="Remove">
                  <Value>
                        <List>
                              <String>ACME Employee Birthright Role</String>
                        </List>
                  </Value>
            </AttributeRequest>
            <AttributeRequest name="assignedRoles" op="Add">
                  <Value>
                        <List>
                              <String>ACME Contractor Birthright
                              Role</String>
                        </List>
                  </Value>
            </AttributeRequest>
      </AccountRequest>
</ProvisioningPlan>
```

### Step 6 – Validate the target properties

The ssf.target.properties file includes a template for the target properties which control this Framework. For each build environment, you must set these values to use this framework. Copy these entries to the build environment's target properties file (if you have not already done this with the SSD Deployer) and set them according to your needs.

- **%%SP_BIRTHRIGHT_ROLES_ORGANIZATION_ROLE%%** - specify the name of the organizational role which contains your implementation's birthright roles
- **%%SP_USE_DEFAULT_BIRTHRIGHT_ROLE%%** - set to true if there is a single, default birthright role which should be assigned to anyone who has no other roles
- **%%DEFAULT_SP_BIRTHRIGHT_ROLE%%** - specifies the name of the role if the USE_DEFAULT property is true. This role is only assigned to users if none of the criteria for the roles in the birthright organizational role were found to match the identity.

```
#*************************************************************************************
#   ROLE ASSIGNMENT FRAMEWORK PROPERTIES
#*************************************************************************************
## ENTER THE NAME OF THE ORGANIZATIONAL ROLE CONTAINING BIRTHRIGHT BUSINESS ROLES
%%SP_BIRTHRIGHT_ROLES_ORGANIZATION_ROLE%%=Birthright Roles
## ENTER true/false FOR WHETHER TO ASSIGN A DEFAULT BIRTHRIGHT ROLE
%%SP_USE_DEFAULT_BIRTHRIGHT_ROLE%%=false
## ENTER THE NAME OF THE DEFAULT BIRTHRIGHT ROLE (ONLY USED IF ABOVE IS TRUE)
%%DEFAULT_SP_BIRTHRIGHT_ROLE%%=Default Birthright Role
```

**Role Assignment Framework Property Settings**

### Step 7 – Deploy the framework

Import the framework artifacts through the SSB deployment process (or other process).

# Approval

The purpose of the Approval framework is to provide a flexible approval subprocess which can be used by any workflow, including the out of the box Lifecycle Manager Workflows or the Provision Processor framework workflow (described in this document), to address more complex and/or more customized approval structures than the out of the box approval subprocesses provide. It allows the business to specify a common approval process to be used for all use cases or to differentiate the approval schemes required for different workflow types or even sets of requested items.

It allows for easy configuration of the required approvers, the approval mode, and the reminder and escalation configurations, and provides pre and post approval rule hooks. Just about any approval scenario can be handled with relative ease.

**NOTE**: IdentityIQ 7.0 introduced the ability to process approvals for individual line items in a request independently from each other. This Approval framework can be invoked from within that split-request structure to enable targeted approval structures, processed asynchronously, for individual line items within a single access request.

## Advantages

The Approval Framework provides these benefits to customers and implementers:

- **Greater flexibility and control** – There are no limits to the number of approval scenarios that can be configured, and each scenario can be configured atomically with control over all facets of the given scenario.
- **Standardization** – The options per approval scenario are universally applied.

## Components

This framework includes 5 objects, provided in 5 separate XML files. Two of these should be used as provided (not edited or modified per installation) and the other three are where the installation-specific logic resides.

These core framework components *should not* be modified or customized per installation:

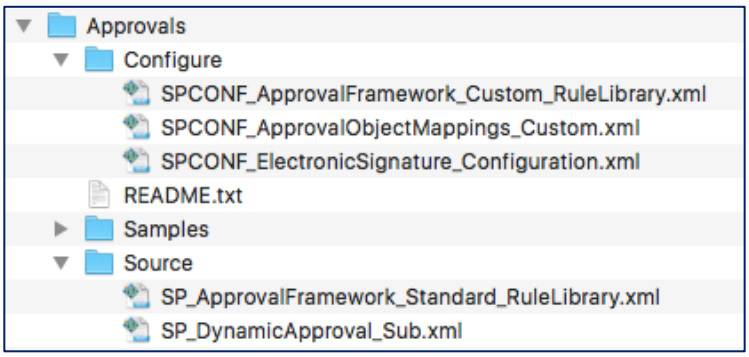
- **SP Dynamic Approval Sub** – subprocess workflow containing the approval logic which incorporates the customizations specified in the installation-specific objects; provided in SP_DynamicApproval_Sub.xml
- **SP Approval Framework Standard Rules Library** – rule library containing methods used throughout the SP Dynamic Approval Sub; provided in SP_ApprovalFramework_Standard_RuleLibrary.xml

These framework components *should* be customized per installation:

- **SPCONF Approval Framework Rules Library** – rule library containing business-specific methods which customize the approval process to meet the organization's business requirements; provided in SPCONF_ApprovalFramework_Custom_RuleLibrary.xml
- **SPCONF Approval Object Mapping** – Custom object which defines and governs the entire approval process; provided in SPCONF_ApprovalObjectMappings_Custom.xml
- **SP Default Electronic Signature** –additional electronic signature entry specified as a merge import to the ElectronicSignature Configuration object; the text (meaning) for this or any other

electronic signatures should be customized and/or added to meet the organization's needs, and the electronic signature should then be referenced in the Custom object as a parameter of one or more of the Approval Types; this is an optional component in the approval process; provided in SPCONF_ElectronicSignature_Configuration.xml

The framework components are provided in the SSF in this structure:




**Approvals Framework Folder Structure**

## How It Works

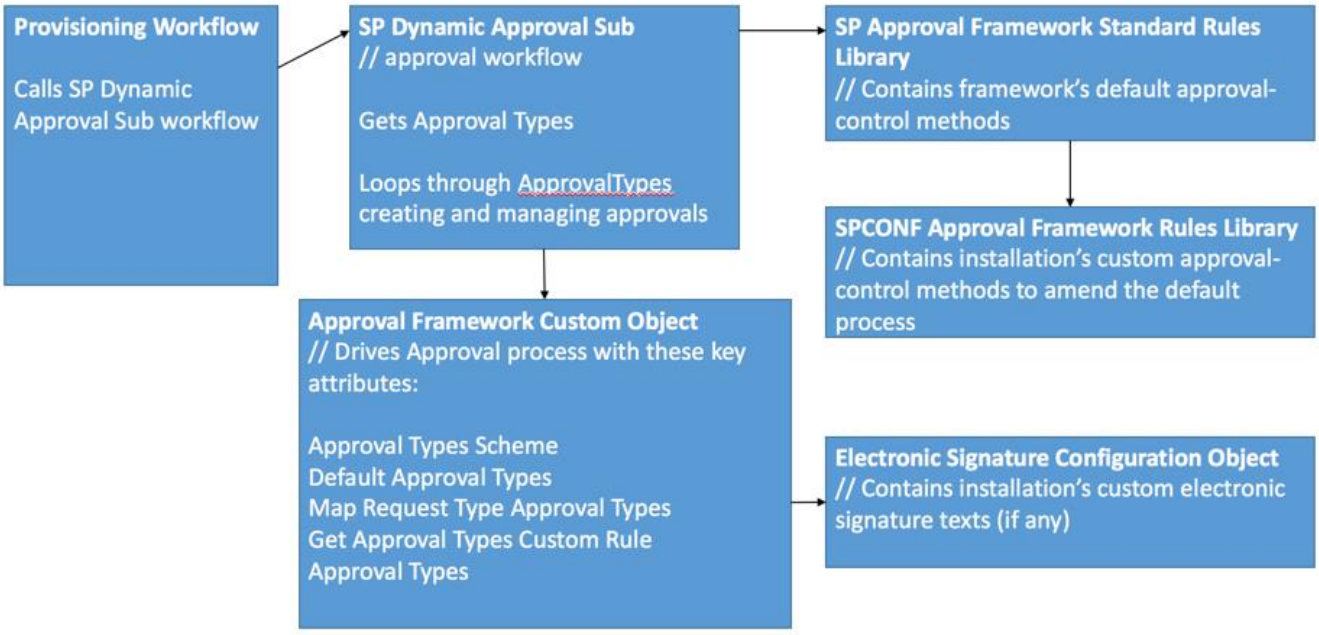
The Approval Framework is built with a great deal of flexibility so it can be configured to meet any customer's needs. In general, the process works like this:

1. The installation's provisioning workflow (or workflows) calls the **SP Dynamic Approval Sub** as its approval subprocess. The "provisioning workflow" could be a customized version of the out-of-the-box workflows like LCM Provisioning, a custom workflow (e.g. joiner workflow or quicklink workflow), or the Provision Processor Framework workflow (described in the next framework section of this document).
2. The SP Dynamic Approval Sub queries the approval framework's **SPCONF Approval Object Mapping** Custom object to determine the appropriate approval structure to use. Several entries in the Custom object's attributes map together define the desired approval structure, as described in the *How to Implement* section below. The approval process may involve seeking the approval of one or more sets of people (e.g. manager, owner, an approval or security team, team leaders, IT staff, etc.).
3. The **SPCONF Approval Object Mapping** Custom object can also specify configuration details for each approval, including:
    - A rule to run prior to the approval
    - The desired approval mode (serial, parallel, etc.)
    - Email templates to use for various notifications
    - Custom names for the approval work items
    - A custom form to use in displaying the approval work item
    - The reminder/escalation process to use
    - Electronic signatures to apply
    - Custom after-scripts or validation scripts

    Any configured pre-approval logic gets run and the approvals get built, assigned, and managed according to these configuration parameters.

4. Each approval is processed in sequence (e.g. manager approval, then owner approval, then approval team, etc.). Between each approval, the workflow can, optionally, filter any rejected items out of the approval so subsequent approvals do not need to review things which have already been rejected.
5. Once all approvals are done, the workflow cleans up some of its staging variables. It can then optionally build email arguments into an emailArgList, which could be returned to the parent workflow to feed into the Dynamic Email Framework to send notifications at the end of this process.

**NOTE**: There are also two Rule Libraries which are part of this framework but which have not specifically been referenced in the steps above. Steps in the SP Dynamic Approval Sub workflow call methods in the SP Approval Framework Standard Rules Library to perform the some of the described functions (such as querying the Custom object for the approval structure or running the pre-approval rule). Those methods, in turn, include rule hooks for custom functionality which allow for running rules specified in the approval configuration details. The custom functionality can be written as stand-along rules or, more commonly, as methods in the SPCONF Approval Framework Rules Library.




The next section details how to configure this framework to take advantage of its many options for customizing the approval process details to meet your business needs.

## How to Implement

These are the basic steps for implementing this framework. Each step is described in greater detail in the next section.

1. Specify the Approval Types Scheme in the framework's **SPCONF Approval Object Mapping** Custom object to define the approval type structure.
2. Also in the **SPCONF Approval Object Mapping** Custom object, specify the configuration for each approval type to be used.

3. Write any custom rules or rule library methods (in the **SPCONF Approval Framework Rule Library**) required for each approval type.
4. Define any electronic signatures used by any of the approval types.
5. Modify your provisioning workflow to call the framework's approval subprocess.
6. Deploy the framework.

### Step 1 – Specify the Approval Types Scheme to define the approval type structure.

The Custom object **SPCONF Approval Object Mappings** drives the behavior of the approval framework, so most of the configuration for implementing this framework is done by adding entries to the attributes map in that object.

The **Approval Types Scheme** entry tells the framework how to determine the approval types for a given use case. The available options are:

- **Default** – this option applies a single, static list of approval types to all approvals for the whole implementation; the list of approval types is specified in the **Default Approval Types** entry of the SPCONF Approval Object Mappings Custom object. This is to be used for simple implementations that have the same approval requirements for all use cases.
- **Map** – use this option to specify a map of request types which each have their own list of required approval types; the map is specified in the **Map Request Type Approvals Types** entry. The SP Dynamic Approval Sub subprocess identifies which map entry to use based on the requestType variable passed from the provisioning workflow. This is typically used in slightly more complex implementations, where the use case can be determined by the top-level workflow and where, for each use case, there is a standard set of required approvals.
- **Custom** – this option uses the rule defined in the **Get Approval Types Custom Rule** entry to return approval types based on any variable(s) passed in from the top-level workflow. This is used in complex implementations with non-linear approval requirements or with approvals determined by the request details. For example, the approval requirements could vary based on the workflow, the requestee, the requested items, etc. This option also requires defining a rule that accepts the workflow variable and returns a list of approval types.

Commonly, the Custom mechanism is used in conjunction with the Map mechanism. This is usually done in one of these two ways:

- The custom rule can use the requestType variable to look up approval types for the most common use cases and then have logic to handle additional exceptions.
- The rule can also be used to dynamically calculate the requestType and then use that to look up the required approval types.

Here is an example that uses the Map option: this says that for all LCM requests (requestType="LCM"), Manager and Owner approval are required, and for all Joiners (requestType="Joiner"), Manager and Department Head approval are required.

```
<entry key="Approval Types Scheme" value="Map" />
<!-- Used if Approval Types Scheme is "Map" -->
<entry key="Map Request Type Approval Types">
 <value>
  <Map>
    <entry key="LCM">
```

```
      <value>
       <List>
        <String>Manager</String>
        <String>Owner</String>
       </List>
      </value>
    </entry>
    <entry key="JOINER">
      <value>
       <List>
        <String>Manager</String>
        <String>Department Head</String>
       </List>
      </value>
    </entry>
  </Map>
 </value>
</entry>
```

### Step 2 – Configure each approval type

The Custom object **SPCONF Approval Object Mappings** contains an entry called **Approval Types** which specifies the configuration details for each approval type in its own map.

These are the available entries in each approval type's map:

- **preApprovalRule** – Specifies a rule that can be called before the approval. This rule could be used to filter out approval items, add comments, do extra auditing, etc. If filtering is to be done, the rule should add the filtered items to the variable tempRemApprovalSet.
- **approvalAfterScriptRule** – Specifies a rule that is called after the approval as part of the approval's After Script processing. This rule is often used to merge any filtered items back into the approval set, but can also be used to add additional comments or do additional logging.
- **approvalValidatorScriptRule** – Specifies a rule that is called during the approval to handle validation requirements, such as requiring comments on rejections.
- **workItemDescriptionRule** – Specifies a rule that is called during the approval to build a custom work item description to provide more meaningful information to the approver about the request.
- **getApprovalOwnersRule** – Specifies a rule that is called during the approval to calculate the approvers' identities. For example, when the approval type is Manager, this would call the getManager() method on the requestee's identity.
- **notifyEmailTemplate** – Specifies the email template that will be sent to the approvers to notify them of their assigned approval work item.
- **approvalMode** – Specifies how the approval will behave. Options are the same as out of the box: serial, serialPoll, parallel, parallelPoll and any.
- **displayName** – an alternative to the workItemDescriptionRule. Specifies the display name of the approval that will appear in the approval title and in the approver's inbox. When the workItemDescriptionRule is not specified or returns no value, the work item name will be "[displayName] Approval – Account Changes for User: [Identity Name]".
- **useDefaultWorkItemConfig** – If true, the default work item config will be used. If false, the workItemConfig entry of the approval type will be used.

- **workItemConfig** – Provides a map of options for the work item config to configure reminders and escalations. See *Configuring the Work Item Config.* **NOTE**: If the attribute "useDefaultWorkItemConfig" (above) is set to true, this workItemConfig attribute is usually omitted because it will be ignored; the workflow would instead apply the configuration in the **Default Work Item Config** entry (also part of the SPCONF Approval Object Mapping Custom object).
- **electronicSignature** – Provide the name of the Electronic Signature object to use if entering credentials is required. Omit if it is not required.
- **useCustomApprovalForm** – Enter true if a custom form will be used and false if one will not. The form will be applied within the out of the box work item approval form to add additional, read-only fields for the approval.
- **getApprovalFormRule** – Enter the name of a rule or method which returns either null or a sailpoint.object.Form object. This can be used to add additional, read-only fields to the approval form. It only applies if "useCustomApprovalForm" is set to true.
    - **NOTE**: Though an identityModel is passed to this form in the approval, prior to version 6.3p4 of IdentityIQ, a product bug prevented full access to the identityModel variable set by the workItemFormBasePath entry, so in those versions, any values that need to be displayed in the form had to be set in the rule/method. In later versions, this form customization becomes less critical because the values could alternatively be added to the identityModel and the form would automatically display them without the form alteration through this rule.
- **template** – Specifies another approval type entry to use as a template. This allows for reuse of configuration – common configurations can be defined in a template and extended or customized per individual approval type map. Entries in the individual approval type maps take precedence over the template, but any attributes specified in the template and not explicitly defined in the given approval type will be used from the template.
- **linkType** – Specifies another approval type entry to copy for this approval type's configuration. This is similar to the template option, except that it does not support customizing or extending the other approval type's configuration; it simply uses that configuration as-is. Its benefit is that it allows you to maintain the configuration in one place to affect the approval for both approval types. This is less commonly used, because you could also specify a template with no overrides to achieve the same objective.

**NOTE**: Any of the attributes whose name includes the word "Rule" can be created as individual Rule objects or as methods in the **SPCONF Approval Framework Rules Library**. If they are written as methods in that rule library, the value of the attribute in the map must start with "**method:**" to tell the framework to look for a rule library method instead of a standalone rule. For example:

```
<entry key="preApprovalRule" value="method:example_PreApproval_Logic" />
```

The following example excerpt from the Approval Types entry in the **SPCONF Approval Object Mappings** Custom object illustrates a template approval type, and an actual approval type (Manager) that uses the template, overrides the approvalMode and displayName, and adds a getApprovalOwnersRule.

```
<entry key="Template Type 1">
 <value>
  <Attributes>
    <Map>
     <entry key="preApprovalRule" value="SP CST Pre Approval Default Splitter Rule" />
     <entry key="approvalAfterScriptRule" value="SP CST Approval After Script Merger
        Rule" />
     <entry key="approvalValidatorScriptRule" value="SP CST Approval Validator Script
        Rule" />
     <entry key="notifyEmailTemplate" value="LCM Identity Update Approval" />
     <entry key="approvalMode" value="parallel" />
     <entry key="displayName" value="Something" />
     <entry key="useDefaultWorkItemConfig" value="true" />
     <entry key="electronicSignature" value="SP Default Electronic Signature" />
     <entry key="useCustomApprovalForm" value="true" />
     <entry key="getApprovalFormRule" value="method:getCSTDefaultApprovalFormRule" />
    </Map>
  </Attributes>
 </value>
</entry>
<entry key="Manager">
  <value>
   <Attributes>
    <Map>
     <entry key="template" value="Template Type 1" />
     <entry key="approvalMode" value="parallelPoll" />
     <entry key="displayName" value="Supervisor" />
     <entry key="getApprovalOwnersRule" value="SP CST Get Approvers Manager Rule" />
    </Map>
   </Attributes>
  </value>
 </entry>
```

### *Configuring the Work Item Config*

A workItemConfig entry can be specified for each approval type in the Approval Types map. Additionally, a Default Work Item Config entry can be configured as a top-level entry in the **SPCONF Approval Object Mappings** Custom object to serve as the default reminder/escalation configuration for any or all approval types.

A work item config is, itself, a map of attributes. These attributes describe how to handle reminders, escalations, and expiration of the work item when the assigned owner does not address it in a timely manner. These are the available attributes to specify in a work item config:

- **reminderEmailTemplate**: name of the email template to send to the owner to remind them of the outstanding work item
- **escalationEmailTemplate**: name of the email template to send to the new owner following escalation of the work item to notify them that the work item has now been assigned to them
- **escalationStyle**: only used in rendering the workItemConfig in the business process editor UI, so not strictly required, but used to drive whether escalation-related or reminder-related fields are displayed for the work item configuration; valid values are “both”, “reminder”, “escalation”
- **hoursBetweenReminders**: length of time (in hours) to wait between sending reminders and/or processing escalations

- **hoursTillEscalation**: number of hours to wait after work item creation before beginning the process of sending reminders or escalating to a new owner
- **maxReminders**: number of reminders to send to the owner before running the escalation process
- **escalationRule**: name of the rule object which determines the new owner when it is time to escalate the work item
- **renderer**: the name of the xhtml file to use to render the work item in the UI

### Step 3 – Write custom rules or rule library methods

Many of the configuration options for each approval type are rules which are run by the approval subprocess workflow at various points (e.g. prior to creating the approval, for identifying the approval owners, as part of the validation or after-script process, etc.). These rules implement custom functionality and therefore must be written for each installation; they are not part of the provided framework. In other words, a rule needs to be written for each entry that specifies a rule name. The Samples folder of this framework provides example rules for each supported rule type.

| Type | Inputs | Return | Details |
|---|---|---|---|
| preApprovalRule | Workflow workflow<br><br>String approvalType | boolean | Accepts the workflow variable, allowing access to all variables. Logic can do any required pre-processing. Must return a boolean indicating whether there are any items left to process for the given approval type. |
| approvalAfterScriptRule | ApprovalSet approvalSet, tempApprovalSetRem<br><br>String approvalType | void | Used to manipulate the approval set and do any necessary merges from the filtered approval set. No returns are required. |
| approvalValidatorScriptRule | ApprovalSet approvalSet<br>WorkItem item<br>String approvalType | String or null | Allows doing any validation. Returns a string if an error should be thrown. Returns null if there are no errors. |
| getApprovalOwnersRule | Workflow workflow | String or List | Calculates the approvers. Can be a string or a list. Recommended to return the name of a work group in cases of |

| | | | |
|---|---|---|---|
| | | | an “any” in a given approval item where a “parallel” is needed for multiple items. |
| getApprovalFormRule | Workflow workflow | Sailpoint.object.Form | Builds and returns a custom form that will be presented to the approvers. |
| workItemDescriptionRule | Workflow workflow | String | Calculates the work item description. |

**NOTE**: As mentioned above, the framework supports specifying methods in the **SPCONF Approval Framework Rules Library** custom rule library, as an alternative to writing individual rules. In fact, the *library method option is preferred* for easier object management and better logging control. To use methods, prefix the method name in the mapping object entry with “method:”, then define the method in the library, **SPCONF Approval Framework Rules Library**. The required parameters for every method are: SailPointContext context, Map params. These can be overloaded with more specific method signatures and called by doing params.get(“desiredParameter”);

For example:

```
public static boolean cstPreApprovalDoNothingRule(SailPointContext context, Map params){
		return cstPreApprovalDoNothingRule( context, params.get("workflow"), params.get("approvalType"));
	}

	public static boolean cstPreApprovalDoNothingRule(SailPointContext context, Workflow workflow, String approvalType){
		logger.trace("Enter cstPreApprovalDoNothingRule");


		ApprovalSet approvalSet = workflow.get("approvalSet");
		boolean hasItems = true;

		logger.trace("Exit cstPreApprovalDoNothingRule: " + hasItems);
		return hasItems;
	}
```

The library, **SPCONF Approval Framework Rules Library**, has a number of examples, such as:

```
public static Object cstGetApproversManagerRule(SailPointContext context, Workflow workflow){
    logger.trace("Enter cstGetApproversManagerRule");

    String identityName = workflow.get("identityName");
    Identity id = context.getObjectByName(Identity.class, identityName);
    String managerName = null;

    if (id != null){
        Identity mgr = id.getManager();
        if (mgr != null){
            managerName = mgr.getName();
        }
    }

    logger.trace("Exit cstGetApproversManagerRule");
    return managerName;
}
```

Sample Approval Method

### Step 4 – Define Electronic Signatures

If the approvals will require an electronic signature, the text for that electronic signature must be defined for the installation. If that text is specific to these approvals and does not already exist for the installation, the electronic signature configuration object must be modified to include the additional required signatures. A basic example, with only US English localized text, is provided in the **SPCONF_ElectronicSignature_Configuration.xml** file. You can modify/copy that example to create as many custom email signature definitions as you need, localized into whatever languages you require.

### Step 5 – Call approval subprocess from provisioning workflow

In many cases, the approval subprocess will be called as part of the provision processor subprocess in the Provision Processor Framework (see next framework section); however, it can also be used independently by any other workflow (including a modified version of the out of the box LCM workflows). This subprocess works off of the approvalSet variable, which is usually built in the initialize step of the main workflow (or provision processor subprocess) at the same time that the provisioning project and identity request object is built.

The approval subprocess must be passed these input arguments:

- approvalSet – contains all items being requested
- requestType – the name of the use case; not required if using the *Default* Approval Types Scheme, but required for *Map* and, in many cases, *Custom*; drives the approval type selections in those schemes

The approval subprocess can also accept the following input arguments:

- **identityName** – the name of the identity the approvals are for
- **identityDisplayName** – the display name of the identity the approvals are for
- **identityModel** - a hash map of variables that can be used on the custom approval form
- **spExtAttrs** – any other extended attributes that might need to be passed up and down the various workflows and made available in the underlying approval rules and methods
- **launcher** – the initiator of the request

- **requestor** – the name of actual requestor. In cases of lifecycle event workflows, the value of the "launcher" argument will be "Scheduler" so this variable can be calculated to determine an actual requestor.
- **plan** – the initial provisioning plan containing the items being requested
- **project** – the compiled provisioning project that will be provisioned if approved
- **updateStandardPostApprovalEmails** – if true, then the emailArgList will be updated with approved/rejected emails; the rest of the arguments below only apply if this is set to true
- **emailArgList** – list of hash maps, each map denoting an email that will be sent out later in the process. This argument is provided to this subprocess if the Dynamic Email Framework is being used (the Provision Processor Framework uses it) so that new emails can be appended to the list of emails to send for approved/rejected notifications
- **approvedTo** – email address of approved recipient
- **approvedTemplate** – email template that will be sent to approved recipient
- **rejectedTo** – email address of rejected recipient
- **rejectedTemplate** – email template that will be sent to rejected recipient

The approval subprocess should return the following variables (by declaring them as Return elements on the parent workflow step which calls this subprocess):

- **approvalSet** – updated with approved/rejected items and comments for each
- **emailArgList** – updated with approved/rejected emails

Here is a sample call to the subprocess which can be copied into any workflow:

```
<Step icon="Task" name="Approve">
  <Arg name="approvalSet" value="ref:approvalSet"/>
  <Arg name="fallbackApprover" value="spadmin"/>
  <Arg name="flow" value="ref:flow"/>
  <Arg name="identityName" value="ref:identityName"/>
  <Arg name="identityModel" value="ref:identityModel"/>
  <Arg name="spExtAttrs" value="ref:spExtAttrs"/>
  <Arg name="identityDisplayName" value="ref:identityDisplayName"/>
  <Arg name="launcher" value="ref:launcher"/>
  <Arg name="plan" value="ref:plan"/>
  <Arg name="policyScheme" value="ref:policyScheme"/>
  <Arg name="policyViolations" value="ref:policyViolations"/>
  <Arg name="trace" value="ref:trace"/>
  <Arg name="workItemComments" value="ref:workItemComments"/>
  <Arg name="requestType" value="ref:requestType" />
  <Arg name="emailArgList" value="ref:emailArgList" />
  <Arg name="approvedTo" value="ref:approvedTo" />
  <Arg name="rejectedTo" value="ref:rejectedTo" />
  <Arg name="requestor" value="ref:requestor" />
  <Arg name="project" value="ref:project" />
  <Arg name="updateStandardPostApproveEmails"
  value="ref:updateStandardPostApproveEmails" />
  <Arg name="approvedTemplate" value="ref:approvedTemplate" />
  <Arg name="rejectedTemplate" value="ref:rejectedTemplate" />
  <Description>
     Call the approval framework subprocess to handle required approvals.
  </Description>
  <Return name="approvalSet" to="approvalSet"/>
  <Return name="emailArgList" to="emailArgList"/>
  <WorkflowRef>
     <Reference class="sailpoint.object.Workflow" name="SP Dynamic Approval Sub"/>
```

```
  </WorkflowRef>
  <Transition to="Provision"/>
</Step >
```

### Step 6 – Deploy the framework

Include the framework's configurable files in your build process by moving the customizable framework XML files from the /Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer). The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects from the framework's /Source folder are automatically deployed with the SSB. Import the framework artifacts through the SSB deployment process (or other process).

### Example Use Case – Simple Joiner

This section details a simple use case to illustrate how to use the Approval framework. The joiner is a lifecycle event launched for every new user. The workflow assigns a single *employee* or *non-employee* birthright role to every user. Non-employees require manager approval. Employees require no approval.

To implement this using this framework you would need to do the following:

1. Create a Joiner workflow with the following steps: Build Plan, Get Request Type, Initialize, Approve, and Provision.
    a. The Build Plan step returns a provisioning plan with an IdentityIQ modify request to add one of the two birthright roles, based on the user type.
    b. The Get Request Type step returns a use case value: either "Non-Employee Joiner" or "Employee Joiner", again based on user type.
    c. The Initialize step calls the out of the box Identity Request Initialize subprocess and returns a provisioning project and approval set.
    d. The Approve step calls this framework's approval subprocess.
    e. The Provision step calls the out of the box Identity Request Provision subprocess to provision the birthright role.
2. Edit the framework's **SPCONF Approval Object Mappings** Custom object to set the ApprovalTypesScheme entry to "Map".
3. Add the following map entries to the Custom object's MapRequestTypeApprovalsTypes entry (this says no approval is required for the EmployeeJoiner request type and Manager approval is required for Non-EmployeeJoiner):
    a. "Employee Joiner"– empty list
    b. "Non-Employee Joiner" – list with the single entry "Manager"
4. Add a "Manager" Approval Types entry to the Custom object with these entries in its map:
    a. getApprovalOwnersRule – "method:getApprovalOwnersManager"
    b. approvalMode – any
    c. displayName – "Manager"
    d. notifyEmailTemplate – Cst Approval Email
    e. electronicSignature – cst Default Electronic Signature

   f. useCustomApprovalForm – true
   g. getCustomApprovalFormRule – cstGetApprovalFormRule
5. Add the method getApprovalOwnersManager to the **SPCONF Approval Framework Rules Library**. Have it return identity.getManager().getName();.
6. Write the rule cstGetApprovalFormRule. Have it return a sailpoint.object.Form object.
7. Create the email template Cst Approval Email.

# Provision Processor

The purpose of the Provision Processor Framework is to provide an alternative to the out of the box LCM Provisioning workflow for handling access requests and account management actions. It utilizes a custom top-level workflow which drives the provisioning process, and calls **SP Identity Request Provision** (a slightly modified version of the out of the box **Identity Request Provision** subprocess workflow that does the provisioning).

The Provision Processor framework invokes the Approval Framework for its approval process, instead of using the out of the box Provisioning Approval Subprocess workflow. The top-level workflow also includes steps which leverage the Dynamic Emails framework.

This framework is intended for all provisioning workflows that follow this basic paradigm: Build Plan, Initialize, Approve and Provision. It is intended to eliminate the most basic workflow development and standardize all Lifecycle Event and LCM workflow processes.

This framework is ready to use as-is and requires no specific customizations. However, some customers may need or want to add custom steps (usually to call custom subprocess workflows) to the top level workflow to support displaying extra forms before or after approvals or to inject any other custom logic between the core provisioning process steps.

## Advantages

The Provision Processor provides these benefits to customers and implementers:

- **Reduces workflow development** – The majority of the workflow steps – those which are common to all use cases - are provided in the framework, ready to use.
- **Reduces testing** – Those workflow steps are proven and tested.
- **Identity Request Integrity** – The subprocesses maintain the identity request object throughout the process.
- **Automatic emails** – The emailArgList variable is constructed automatically based on the success or failure of the provisioning activities, which facilitates the use of the Dynamic Emails framework for email sending. (The Dynamic Email framework would still need to be called by the workflow which uses this framework; the framework workflows do not invoke the Dynamic Email framework directly.)

## Components

This framework includes 4 objects, provided in 4 separate XML files. Three of these should be used as provided (not edited or modified per installation) and the fourth is where any installation-specific logic resides.

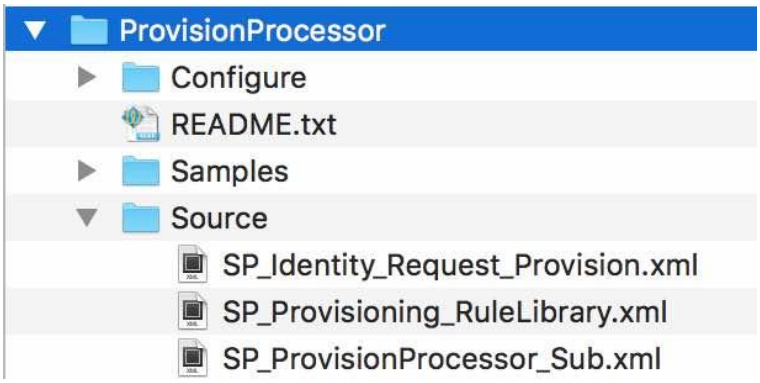
These core framework components *should not* be modified or customized per installation:

- **SP Provision Processor Sub** – main driver workflow for the framework
- **SP Identity Request Provision** – provisioning subprocess workflow
- **SP Provisioning Rules Library** – contains BeanShell methods used by SP Provision Processor Sub

This framework component *can* be customized per installation, though this is not required:

- **SPCONF Email Text Mappings Custom** – specifies email body text to include in notification email messages based on details in the request such as the application involved or the specific operation performed; a template is provided in **SPCONF_EmailText_Custom.xml**

The framework components are provided in the SSF in this structure:




**Provision Processor Framework Folder Structure**

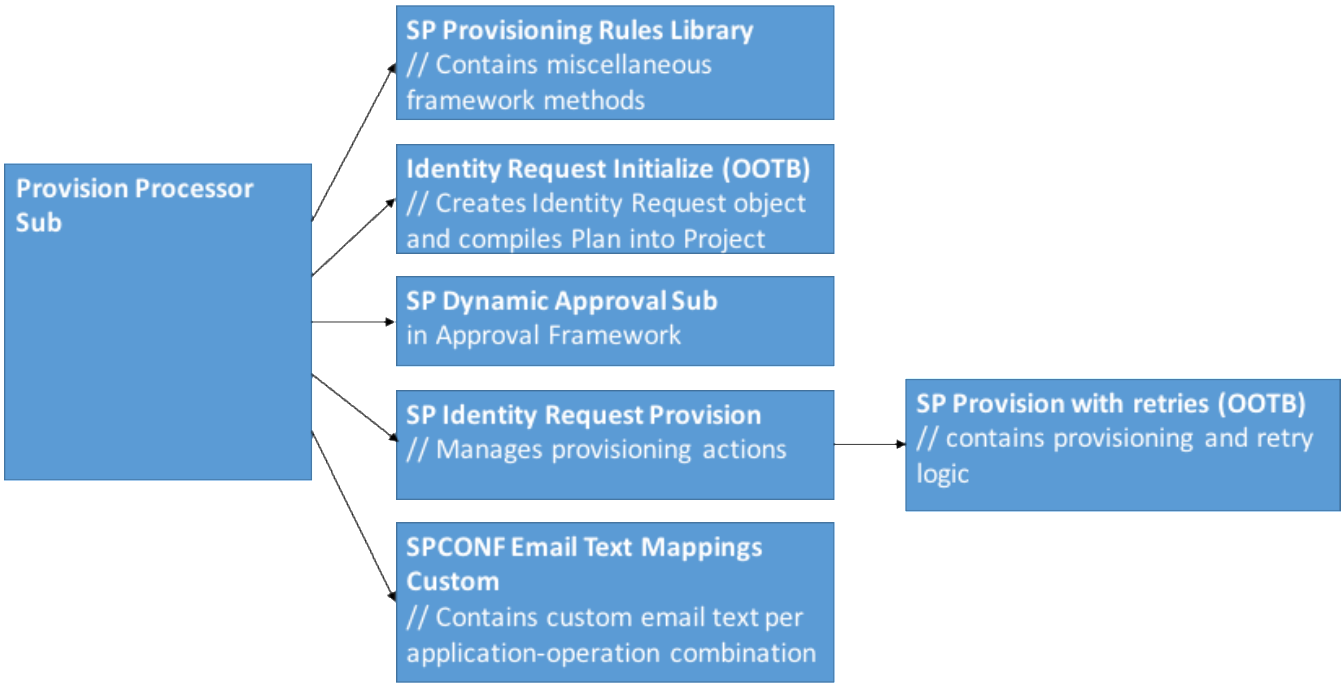
## How It Works

1. The **SP Provision Processor Sub** workflow drives the provisioning process, much like LCM Provisioning would in the out of the box workflows. It calls other subprocesses to manage the steps in the provisioning process, in this order:
    a. **Identity Request Initialize** (an out of the box subprocess)
    b. **SP Dynamic Approval Sub** (from the *Approval* framework previous described)
    c. **SP Identity Request Provision** (a modified version of the out of the box subprocess **Identity Request Provision**)

    It also uses methods in the **SP Provisioning Rules Library** in some steps in its logic.
2. The **Identity Request Initialize** subprocess compiles the provisioning plan into a provisioning project, and creates the identity request for tracking the request progress through the UI.
3. The **SP Dynamic Approval Sub** drives the approval process for the request, as described in the *Approval* Framework section above.
4. The **SP Identity Request Provision** subprocess manages the provisioning process by invoking the out of the box **Provision with retries** subprocess. It also handles monitoring of the status of queued requests and creation of work items for manual provisioning. The only difference between this and the out of the box **Identity Request Provision** subprocess is that this version

passes the “noTriggers” variable to the **Provision with retries** subprocess with a value of “true”. This is designed to prevent a lifecycle event trigger running twice when the attribute values that cause the trigger to fire are not reset until after provisioning occurs.

5. Once the provisioning steps are finished, the SP Provision Processor Sub can then optionally add emails to the emailArgList so the Dynamic Email Framework can be used to send email notifications about the status of the provisioning operation. If desired, the text for the email can be specified per application and operation in the **SPCONF Email Text Mappings** Custom object.




## How to Implement

These are the basic steps for implementing this framework. Each step is described in greater detail in the next sections.

1. Configure any required email text (optional)
2. Call the **SP Provision Processor Sub** workflow from your provisioning workflow, specifying the appropriate input variables.
3. Deploy the framework.

### Step 1 – Configure any required email text

This workflow is set up for you to specify a single email template for all success notifications and a single template for all failure notifications. By adding entries to the SPCONF Email Text Mappings Custom object, you can optionally customize the email content per application and operation. Entries are added per application and operation.

For example, this entry specifies the text to add when a user account is created on the XYZ application.

```
<Custom name="SPCONF Email Text Mappings Custom">
 <Attributes>
  <Map>
   <entry key="XYZ">
     <value>
       <Attributes>
         <Map>
           <entry key="Create">
             <value>
                <String>&lt;p&gt;For help getting started using the XYZ system,
contact John Smith at jsmith@company.com. &lt;/p&gt; </String>
             </value>
           </entry>
         </Map>
       </Attributes>
     </value>
    </entry>
  </Map>
 </Attributes>
</Custom>
```

The Build Emails step, near the end of the SP Provision Processor Sub workflow, adds a hash map of parameters to the emailArgList variable for success/failure emails for each successful or failed account request in the provisioning project. If you have specified custom text in the SPCONF Email Text Mappings Custom, it is added to the map under the name customText. Your email template can then print the text in the message body by referencing the variable $emailArgs.customText.

**NOTES**:

- Many customers do not use this custom text feature. If no custom text exists for a given application-operation combination, the customText attribute will not be set in the map.
- To bypass the Build Emails step entirely and suppress creation of success/failure workflows in this framework, set the workflow variable **updateStandardPostProvEmails** in the SP Provision Processor Sub to “false”.
- The SP Provision Processor Sub does not automatically send these emails. You can modify the getPostProvProjectEmailArgs method call in the **SP Provisioning Rules Library** to force it to send the queued messages from the emailArgList, or your provisioning workflow can call the *Dynamic Emails* Framework after the SP Provision Processor Sub is finished.

### Step 2 – Call the subprocess with the appropriate inputs

The **SP Provision Processor Sub** subprocess expects to be passed an identityName and a plan variable, and it accepts a few other attributes which can drive its functionality. The subprocess can accept the following input arguments:

- **identityName** – The name of the identity being processed
- **identityDisplayName** – The display name of the identity being processed
- **spExtAttrs** – A HashMap containing any extended attributes that might need to be passed up and down various workflows
- **identityModel** – A HashMap containing extended attributes that might need to be

displayed in a given custom form

- **launcher** – The name of the identity that launched the request. For LCE workflows, this is usually Scheduler. For LCM, it is the identity that initiated the request.
- **plan** – The provisioning plan being requested.
- **trace** – Whether to trace the workflow.
- **emailArgList** – Container of all emails that need to be sent. Will be updated throughout the process.
- **requestType** – The use case being processed. Works in conjunction with the approval framework to drive the required approval types.
- **successTo** – The email address that will receive the provisioning success emails.
- **failureTo** – The email address that will receive the provisioning failure emails.
- **successTemplate** – The email template that will be used for the provisioning success emails.
- **failureTemplate** – The email template that will be used for the provisioning failure emails.
- **approvedTo** – The email address that will receive the approved emails.
- **rejectedTo** – The email address that will receive the rejected emails.
- **approvedTemplate** – The email template that will be used for the approved emails.
- **rejectedTemplate** – The email template that will be used for the rejected emails.
- **updateStandardPostProvEmails** – If true, success/failure emails will be sent.
- **updateStandardPostApproveEmails** – If true, approved/rejected emails will be sent.
- **foregroundProvisioning** – If true, provisioning will happen in the foreground. If false, provisioning will happen next time the “Perform Maintenance” task runs. It is strongly recommended that this be set to “false” to avoid identity locking issues when running the SSF features; this is because the workflows normally run as part of an identity refresh, which causes the identity to be locked while the workflow runs, preventing updates to the identity unless the workflow is backgrounded.

The subprocess expects to return the following variables to the calling workflow (though the step in the calling workflow must define these as <Return> elements for it to receive them):

- **emailArgList** - A list of hash maps containing emails that need to be sent out. Will contain any approved/rejected or success/failure emails appended by the sub.
- **approvalSet** – The executed approval set containing the approved/rejected status of each requested item.
- **project** – The executed project, containing all account/attribute requests that were provisioned as well as the result for each.

The following is an example call to the subprocess from a provisioning workflow:

```
<Step icon="Task" name="Process Plan">
<Arg name="fallbackApprover" value="spadmin"/>
<Arg name="flow" value="ref:flow"/>
<Arg name="identityName" value="ref:identityName"/>
<Arg name="identityDisplayName" value="ref:identityDisplayName"/>
```

```
<Arg name="identityModel" value="ref:identityModel"/>
<Arg name="spExtAttrs" value="ref:spExtAttrs"/>
<Arg name="launcher" value="ref:launcher"/>
<Arg name="plan" value="ref:plan"/>
<Arg name="trace" value="ref:trace"/>
<Arg name="emailArgList" value="ref:emailArgList"/>
<Arg name="requestType" value="ref:requestType" />
<Arg name="requestor" value="ref:requestor" />
<Arg name="successTo" value="ref:successTo" />
<Arg name="failureTo" value="ref:failureTo" />
<Arg name="successTemplate" value="ref:successTemplate" />
<Arg name="failureTemplate" value="ref:failureTemplate" />
<Arg name="approvedTo" value="ref:successTo" />
<Arg name="rejectedTo" value="ref:failureTo" />
<Arg name="approvedTemplate" value="ref:approvedTemplate" />
<Arg name="rejectedTemplate" value="ref:rejectedTemplate" />
<Arg name="updateStandardPostProvEmails" value="ref:updateStandardPostProvEmails" />
<Arg name="updateStandardPostApproveEmails"
value="ref:updateStandardPostApproveEmails" />
<Arg name="foregroundProvisioning" value="ref:foregroundProvisioning" />
<Return name="emailArgList" to="emailArgList"/>
<Return name="approvalSet" to="approvalSet"/>
<Return name="project" to="project"/>
<WorkflowRef>
   <Reference class="sailpoint.object.Workflow" name="SP Provision Processor Sub"/>
</WorkflowRef>
<Transition to="Send Emails" />
</Step>
```

### Step 3 – Deploy the framework

You must include the Configure/SPCONF_EmailText_Custom.xml file in your build process by moving the file from the /Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer). The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects from the framework's /Source folder are automatically deployed with the SSB. Import the framework artifacts through the SSB deployment process (or other process).

### Example Use Case – Simple Joiner

The simple use case illustrates how to use the framework. This is the same use case provided in the approvals framework section. The steps specific to the approvals will be omitted and referenced to show what is needed for this framework. This use case will add a step to send emails with the Dynamic Emails framework at the end.

The joiner is a Lifecycle Event launched for every new user. The workflow assigns a single "employee" or "non-employee" birthright role to every user. Non-employees require manager approval. Employees require no approval.

To implement this using this framework you would need to do the following:

1. Create a Joiner workflow with the following steps: Build Plan, Get Request Type, Process Plan, Send Emails.

a. The Build Plan step creates and returns a provisioning plan with an IdentityIQ modify request to add one of the two birthright roles, based on the user type.
b. The Get Request Type step returns a use case value: either “Non-Employee Joiner” or “Employee Joiner”, again based on user type.
c. The Process Plan calls the SP Provision Processor Sub subprocess with the required input variables to provision the birthright role. This step takes the place of the Initialize, Approve and Provision steps in the workflow as discussed in the Approval framework because those steps are part of the Provision Processor framework.
    - The SP Provision Processor Sub uses the Approval framework, so ensure that the Approval framework is configured as described in those example use case details.
d. The Send Emails step calls the SP Send Dynamic Emails Sub (from the Dynamic Emails framework), passing it the emailArgList variable built throughout this workflow. That subprocess sends the required set of email messages specified in the maps in that list.

# Features

The Features are a set of end-to-end solutions for addressing common business requirements in IdentityIQ implementations. Most of them are Lifecycle Events, with their corresponding lifecycle event workflows. All of the Lifecycle Event Features share a common architecture, so once you understand one of the Features, you will quickly know how to use them all. The first section below describes the common architecture. Later sections elaborate on the specific details for each Feature.

**NOTE**: The Terminate Identity is the only Feature which is not a lifecycle event and therefore does not fully follow the pattern described in the Features Commonalities section below.

## Features Commonalities

All Lifecycle Event Features share the same folder structure, parallel sets of target properties, and the same core set of workflow steps. Of course, the actions which occur in each step vary according to the purpose of the lifecycle event and workflow. Both the core-process actions and business-specific actions to be carried out by the workflow are defined through methods in a custom library which are called by the workflow process. These rule "hooks" in the workflow support easier object management by consolidating the functional/business logic into a rule library.

### Advantages

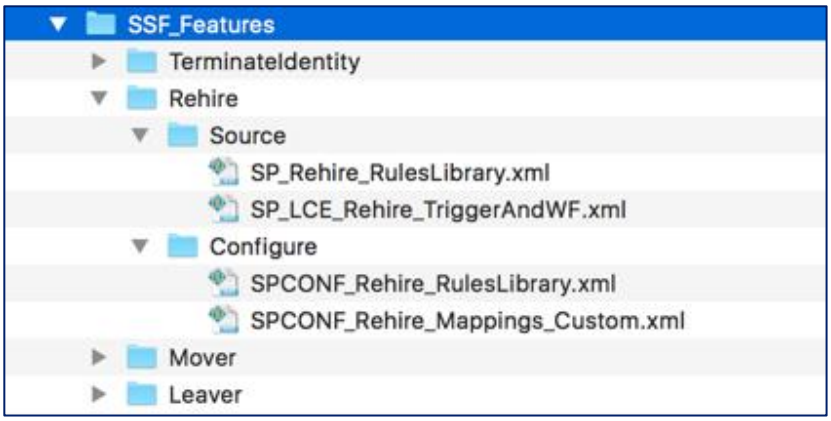
The advantages of using the Features for your Lifecycle Event development are:

- **Proven and tested methodology** – Because they rely heavily on the Frameworks, which are well-tested themselves, the core components of the Features are already proven and tested.
- **Business-centric development** – With the core process provided by the feature, the implementers' focus can be put on the actual business requirements, such as how to assign access, what approvals are required, etc.
- **Rapid deployment** – With the pre-built options provided, each feature can be up and running, with most typical configurations, in hours or even minutes.

### Structure

All Features provide their artifacts in the same folder structure:

- **Configure** – folder containing the files that need to be moved into the SSB /config folder (manually or using the SSD Deployer) and configured per installation
    - A Rule Library – Move to either /config/Rule or /config/RuleLibrary
    - A Custom Mapping Object – Move to /config/Custom
- **Source** – the files that should be left alone to be deployed as-is by the SSB process
    - A Rule Library
    - The Workflow and Trigger




**Example Feature Folder Structure**

**NOTE**: There is a folder in the Features folder tree called **Common** which contains a single rule library called **SPCONF Common Rules Library**. This rule library is included as a referenced rule in all of the Feature workflows, so any methods in it could be used universally by any or all of the Features. At this time, there are no methods provided in this rule library by default. However, customers may wish to implement reusable logic that is needed across all Features here instead of in feature-specific SPCONF rule libraries.


**Common Folder Rules Library**

## Target Properties

There is a set of target properties in the ssf.target.properties file corresponding to each Feature. Comments in the ssf.target.properties file describe how to use each of the properties. Some features may have additional feature-specific properties, but all features include at least these properties:

- **IS_DISABLED** – Determines whether the feature is on or off by setting the disabled flag on the lifecycle event in the IdentityIQ instance
- **WF_TRACE_ENABLED** – Whether or not to turn on workflow tracing for the underlying workflow (used in non-production environments)
- **SEND_APPROVED_EMAILS** – Whether to automatically send approved/rejected emails; when enabled, the workflow, by default, notifies the target user of whether each item of access requested by the lifecycle event workflow was approved or rejected; this notification is done at the end of the workflow
- **SEND_POST_PROVISION_EMAILS** – Whether to automatically send provision success/failure emails; when enabled, the workflow, by default, notifies the target user of whether each approved item of access was successfully provisioned or not; this notification is done at the end of the workflow
- **FOREGROUND_PROVISIONING** – Whether to run provisioning in the foreground; when enabled, provisioning will happen immediately after any approvals; when disabled, provisioning will happen next time the Perform Maintenance task runs. It is strongly recommended that this

be set to “false” to avoid identity locking issues when running the SSF features; this is because the workflows normally run as part of an identity refresh, which causes the identity to be locked while the workflow runs, preventing updates to the identity unless the workflow is backgrounded.

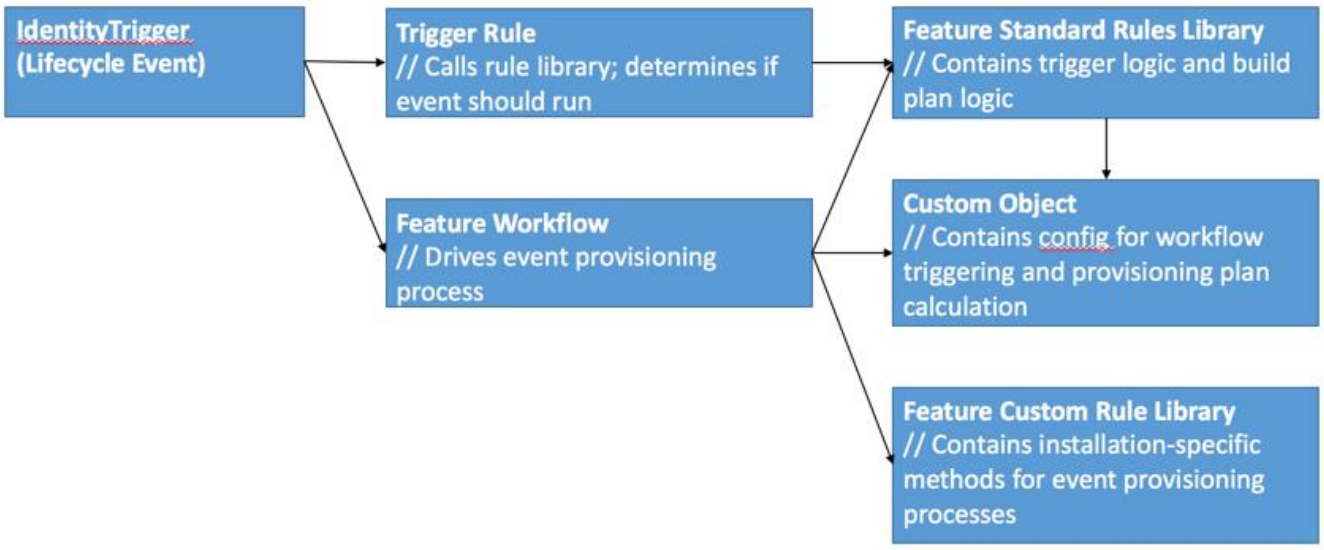
```
#*******************************************************************************
#   JOINER FEATURE PROPERTIES
#*******************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_JOINER_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_JOINER_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_JOINER_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_JOINER_SEND_POST_PROVISION_EMAILS%%=false
## ENTER true/false ON WHETHER TO RUN PROVISIONING IN THE FOREGROUND
%%SP_JOINER_FOREGROUND_PROVISIONING%%=false
```

**Example Target Properties Entries for the Joiner Feature**

## How the Features Work

Each Lifecycle Event Feature follows this same basic process.

1. Lifecycle Events are usually processed either as a part of an identity refresh task, which examines and launches workflows for many identities, or as part of a provisioning workflow, which launches events only for the identity being processed in the provisioning request.
2. When events are processed, the identityTrigger (the Lifecycle Event itself) for each feature’s workflow calls a rule defined as part of the feature to determine if the workflow should be run for each identity being examined.
3. The rule, in turn, calls a rule library method in the feature’s standard rule library.
4. The rule library method queries the feature’s Custom mapping object for the configuration details which specify the conditions for launching the workflow, and if the user matches the criteria, it signals the event to launch the workflow.
5. The feature workflows are designed to support some sort of provisioning action. They all follow the same set of workflow steps, performing event-specific logic in each, with the end goal of provisioning (or deprovisioning) the appropriate access for the user. They all use configurations in their custom maps to direct their processes and methods in their custom and standard rule libraries to provide their logic.



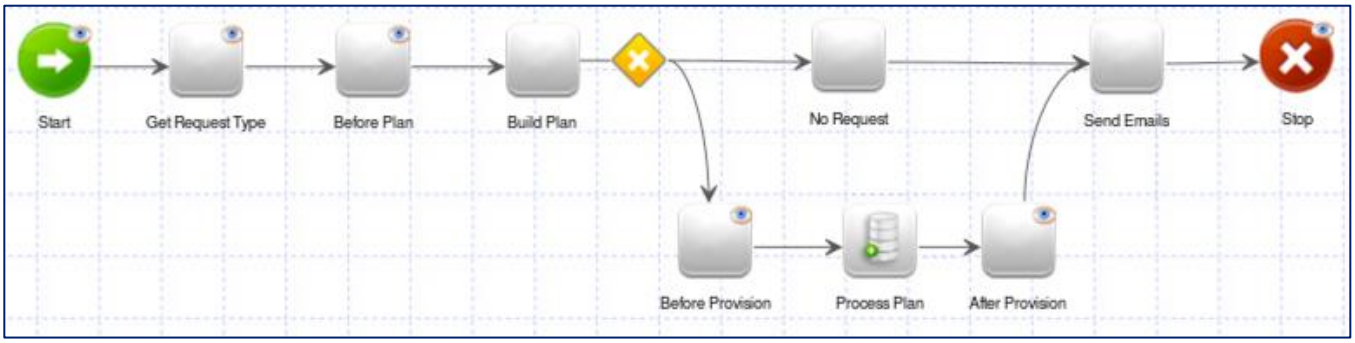

### Feature Workflow Steps

The basic flow of each Lifecycle Event Feature workflow is the same as well. Each Lifecycle Event Feature workflow contains workflow steps which are named the same and perform the same fundamental actions (though the specific operation of that action varies across the workflows). The specific step operations are defined in the Feature's rule libraries, with the workflow step set up to call those rule library methods.

These are the standard Feature workflow steps (executed in this order):

1. **Get Request Type** – calls a custom library method to determine the string value that denotes the use case (request type) being processed; this attribute usually drives the approval structure used in the Approval Framework
2. **Before Plan** – calls a custom library method to allow for any business-specific logic customizations prior to creating the provisioning plan
3. **Build Plan** – uses the options in the custom mapping to dynamically build out a provisioning plan
4. **No Request** – calls a custom library method to perform any non-provisioning logic required by the installation, in lieu of provisioning, during the lifecycle event; only executed if no provisioning plan is created in the Build Plan step
5. **Before Provision** - calls a custom library method to allow for any business-specific logic customizations; typically used to adjust the provisioning plan before beginning the provisioning process
6. **Process Plan** – calls the **SP Provision Processor Sub** (from the Provision Processor Framework) to Initialize, Approve, and Provision based on the provisioning plan. (The Approve step of the SP Provision Processor Sub calls the **SP Dynamic Approval Sub** of the Approval Framework.)
7. **After Provision** - calls a custom library method to allow for any business-specific logic customizations to be performed following the provisioning operation

8. **Send Emails** – calls the **SP Dynamic Send Emails Sub** of the Dynamic Emails Framework to send out any required emails (which have been added to the emailArgList throughout any other steps in this workflow)




**Feature Workflow Steps**

## Feature Custom Rule Libraries

All of the Features' custom rule libraries contain, at a minimum, these common methods:

- **getRequestTypeRule** – Return a string to determine the workflow's use case, which can then be used in the underlying approval framework
- **beforePlanRule** – Do any business logic customization prior to provisioning plan creation
- **beforeProvisionRule** – Do any business logic customization before provisioning the plan
- **afterProvisionRule** – Do any business logic customization after provisioning the plan
- a set of methods for calculating to-addresses and email templates for approved/rejected and success/failure email messages (method names vary per event type)

See the ***Appendix – Example Custom Rule Library*** for an example of a custom rule library.

# Joiner

The Joiner feature exists to drive birthright provisioning – automatic provisioning which should occur as identities first join the organization. This feature supports several types of configurations for calculating birthright access for users. The event can provision:

- A static list of roles which should be granted to all joiner users
- A static list of application accounts which should be created for all joiner users
- A variable list of roles which are granted to users based on the role assignment configurations, using the Role Assignment Framework
- A variable set of access defined by a custom rule which creates the provisioning plan dynamically, as defined by the installation's requirements

## Components

These core feature components *should not* be modified or customized per installation:

- **SP LCE Joiner Trigger** – the IdentityTrigger which defines the lifecycle event; provided in SP_LCE_Joiner_TriggerAndWF.xml
- **SP LCE Joiner Trigger Rule** – the rule which determines whether the lifecycle event workflow should be run; provided in SP_LCE_Joiner_TriggerAndWF.xml
- **SP LCE Joiner WF** – the lifecycle event workflow; provided in SP_LCE_Joiner_TriggerAndWF.xml
- **SP Joiner Rules Library** – the Joiner Feature's standard rules library, which contains the trigger logic and the build plan logic; provided in SP_Joiner_RulesLibrary.xml

These feature components should be modified and customized per installation:

- **SPCONF Joiner Mappings Custom** – the Custom object which defines the criteria for running the joiner event and the configuration for the birthright provisioning process; provided in SPCONF_Joiner_Mappings_Custom.xml
- **SPCONF Joiner Rules Library** – the configurable Rules library where all installation-specific logic for the joiner event should be written; provided in SPCONF_Joiner_RulesLibrary.xml

## How It Works

### The Trigger

The standard definition of a "joiner" in IdentityIQ is a new identity being created, usually from aggregation from an authoritative source. But sometimes identity creation occurs for reasons which should not trigger "joiner" events, such as the creation of Service identities or of non-authoritative identities to temporarily support orphaned accounts. Furthermore, what constitutes a "joiner" may vary per organization, so this lifecycle event trigger provides alternative ways to define a "joiner" action.

The trigger options for this workflow are:

- A custom rule which evaluates the identity using whatever logic the organization chooses to define
- An IdentitySelector which defines attribute values that designate a joiner for the organization

Like all of the Feature triggers, this is a rule-based trigger.

1. The **SP LCE Joiner Trigger** calls the **SP LCE Joiner Trigger Rule** which, in turn, calls the **isTriggerJoinerRule** method in the SP Joiner Rules Library.
2. The **isTriggerJoinerRule** method queries the SPCONF Joiner Mappings Custom object for a **Trigger Type** entry which tells how to identify identities which are Joiners. The two valid values are:
    a. **Selector** – the method compares the identity to the match expression in the **Trigger Field Selector** entry in the SPCONF Joiner Mappings Custom object and returns "true" to run the joiner workflow if the identity matches the criteria
    b. **Custom Rule** – the method passes the identity to the rule or rule library method named in **Trigger Custom Rule** in the SPCONF Joiner Mappings Custom object; the rule/method examines the identity and returns "true" if the identity is a Joiner; the isTriggerJoinerRule method then also returns "true" and runs the joiner workflow for the identity (default rule to use is SPCONF Rule Library method isJoinerCustomTriggerRule)

```
<!--  Enter the Trigger Type.
        Options Include:
            - Selector:  Will dynamically re-evalute all target attributes and update as necessary.
            - Custom Rule:  Will call out to the rule defined in the
                Birthright Assignment Custom Rule entry below.  Rule will receive an Identity object.
                Rule must return a ProvisioningPlan.
-->
<entry key="Trigger Type" value="Selector" />

<!--  Create the Selector to determine whether to kick off the Joiner -->
<entry key="Trigger Field Selector">
    <value>
            <IdentitySelector>
              <MatchExpression and="true">
                <MatchTerm name="personStatus" value="A"/>
                <MatchTerm name="joinerDate" value="NULL"/>
              </MatchExpression>
            </IdentitySelector>
    </value>
</entry>

<!-- Used if Trigger Type is Custom Rule.
    Recommended that this value isn't changed
    and the existing method in SP CONF Joiner Rules Library is used-->
<entry key="Trigger Custom Rule" value="method:isJoinerCustomTriggerRule" />
```

**Joiner Trigger Configuration Options in Custom Mapping Object**

### The Workflow

1. The workflow first sets the **requestType**, which can be used by the Approval framework to drive the approval structure. By default, this gets set to "Joiner".
2. A custom rule hook is available to perform installation-specific actions prior to provisioning plan creation. The **joinerBeforePlanRule** rule library method provides a template of actions which could be performed here.
3. Next, the workflow builds the provisioning plan based on the birthright provisioning configuration specified in the SPCONF Joiner Mappings Custom object's **Birthright Assignment Type** attribute.

4. If the plan is empty, the **No Request** step offers a custom rule hook to perform any non-provisioning logic required by the installation, in lieu of provisioning.
5. Otherwise, the **Before Provision** step offers a different custom rule hook to allow for any business-specific logic customizations (usually of the provisioning plan) before beginning the provisioning process.
6. Next, the workflow invokes the *Provision Processor* framework to perform the provisioning operation (including approval through the Approval framework).
7. The **After Provision** step then offers another custom rule hook for any business-specific logic which needs to be performed following the provisioning operation.
8. Lastly, the workflow invokes the *Dynamic Emails* Framework to send out any required emails which were added to the emailArgList throughout any other steps in this workflow or the Provision Processor framework workflow.

## How to Implement

The basic steps to use the framework are:

1. Configure the target properties to activate the event and determine whether emails will be sent out at the end of the provisioning process
2. Configure the trigger for the workflow
3. Select birthright assignment type
4. Configure the assignment details
5. Configure rules to manage additional actions required by the business
6. Deploy the feature

### Step 1 – Configure Target Properties

The ssf.target.properties file provides a template for all the values required to manage the feature. Locate the Joiner Feature Properties section and copy everything in this section to the target.properties file for your environment (if you have not already run the SSD Deployer which does this for you). Set SP_JOINER_IS_DISABLED to false to enable the feature, and set other attributes to true or false as described in the comments associated to each property in that section of the file.

```
#*********************************************************************************
#   JOINER FEATURE PROPERTIES
#*********************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_JOINER_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_JOINER_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_JOINER_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_JOINER_SEND_POST_PROVISION_EMAILS%%=false
## ENTER true/false ON WHETHER TO RUN PROVISIONING IN THE FOREGROUND
%%SP_JOINER_FOREGROUND_PROVISIONING%%=false
```

### Step 2 – Configure Trigger Attributes

Trigger attributes define if and when the lifecycle event's workflow will run for a given identity. All Trigger attributes are set in the SPCONF Joiner Mappings Custom object.

1. Set **Trigger Type** to specify how to decide whether the lifecycle event should be run for a given identity. Trigger Type value must be one of these:

- **Selector**: trigger conditions will be defined by the Trigger Field Selector attribute
- **Custom Rule:** trigger conditions will be the Trigger Custom Rule attribute

2. If Trigger Type is Selector, define the **Trigger Field Selector** attribute. The Identity Trigger looks for the IdentitySelector in this attribute which specifies the condition for when the workflow should be launched. Use the following as an example:

```
<entry key="Trigger Field Selector">
  <value>
    <IdentitySelector>
      <MatchExpression and="true">
        <MatchTerm name="personStatus" value="A"/>
        <MatchTerm name="joinerDate" value="NULL"/>
      </MatchExpression>
    </IdentitySelector>
  </value>
</entry>
```

3. If Trigger Type is Custom Rule, define the **Trigger Custom Rule** attribute. This attribute should specify the name of a rule or a rule library method which will evaluate the identity and return "true" if the joiner event should be run for the identity.
    - The SPCONF Joiner Mappings Custom object predefines this Trigger Custom Rule attribute to point to a rule library method in the SPCONF Joiner Rule Library, called **isJoinerCustomTriggerRule**. The recommendation is to leave that entry as-is and to specify your custom logic for identifying a joiner identity in that method in the SPCONF Joiner Rule Library.

### Step 3 – Specify Birthright Assignment Type

Required "birthright" access can vary across organizations. For some customers, a standard set of roles should be assigned to all users when they join the organization. Others may just want to grant accounts on specific applications, while others have variable lists of roles or entitlements that need to be provisioned for new personnel.

Set the **Birthright Assignment Type** entry in the SPCONF Joiner Mappings Custom object to determine how the provisioning plan is constructed in the Build Plan step of the workflow. This attribute must be one of these four values:

1. **Dynamic Roles**: The workflow will use the Role Assignment Framework to dynamically assign roles to the identity. This, of course, requires that framework to be in place and configured.
2. **Default Roles**: The workflow assigns a static list of roles, found in the **Default Assignments** entry of the SPCONF Joiner Mappings Custom object, to the joiner identity.
3. **Default Applications**: The workflow creates accounts for the identity on a static list of applications, found in the **Default Assignments** entry of the SPCONF Joiner Mappings Custom object.
4. **Custom Rule**: The workflow uses the rule defined in the **Birthright Assignment Custom Rule** entry of the SPCONF Joiner Mappings Custom object to build the provisioning plan to provision the desired birthright access.

### Step 4 – Configure the Assignment Details

Depending on the Birthright Assignment Type specified in step 3, additional attributes must be specified in the SPCONF Joiner Mappings Custom object or elsewhere, as noted here.

1. If the assignment type is **Dynamic Roles**, then configure the Role Assignment Framework as described in that section of this document.
2. If the assignment type is **Default Roles**, enter the list of roles in the **Default Assignments** entry of the SPCONF Joiner Mappings Custom object.

```
<entry key="Birthright Assignment Type" value="Default Roles"/>
<entry key="Default Assignments">
  <value>
    <List>
      <String>LDAP User</String>
      <String>Finance Role 1</String>
    </List>
  </value>
</entry>
```

3. If the assignment type is **Default Applications**, enter the list of application names in the **Default Assignments** entry of the SPCONF Joiner Mappings Custom object.

```
<entry key="Birthright Assignment Type" value="Default Applications"/>
<entry key="Default Assignments">
  <value>
    <List>
      <String>Active Directory</String>
      <String>Enterprise LDAP</String>
    </List>
  </value>
</entry>
```

4. If the assignment type is **Custom Rule**, enter a rule name in the **Birthright Assignment Custom Rule** entry of the SPCONF Joiner Mappings Custom object.
    - This entry is predefined in the SPCONF Joiner Mappings Custom object to point to a rule library method in the SPCONF Joiner Rule Library called buildCustomJoinerPlan. The recommendation is to leave that entry as-is in the Custom object and write your desired business logic in that rule library method. This rule/method must accept an Identity object and return a ProvisioningPlan object.

### Step 5 – Configure the Hooks

Most of the joiner workflow's steps simply call methods in the **SPCONF Joiner Rule Library**, where you can write your own logic to further customize the behavior of the joiner workflow. As provided, the methods may be empty or may provide some template/default logic.

| Method | Calling Workflow Step | Default Logic Provided |
|---|---|---|
| getjoinerRequestTypeRule | Get Request Type | Returns "Joiner" |
| joinerBeforePlanRule | Before Plan | Provides logic to retrieve identityModel and spExtAttr variables from the workflow and return them to the workflow; |

| | | |
|---|---|---|
| | | implementer should insert any logic to modify these variables as desired or perform any other before-plan-generation logic desired |
| joinerNoRequestRule | No Request (only if Build Plan step does not return provisioningPlan) | Sets an extended attribute called joinerAccountDate to today and the standard inactive attribute to false |
| joinerBeforeProvisionRule | Before Provision | None |
| joinerAfterProvisionRule | After Provision | Sets an extended identity attribute called joinerDate to the current date |
| buildCustomJoinerPlan | Build Plan (if Birthright Assignment Type = Custom Rule and Birthright Assignment Custom Rule is left pointing to this method) | Returns empty ProvisioningPlan |
| The remainder of the rule hooks in the SPCONF Joiner Rule Library are for configuring email message “to” addresses and templates to be used by the Provision Processor and Approval frameworks for queuing up information messages to be sent at the end of the workflow process. | | |
| getJoinerProvSuccessEmailTemplateRule | Calculate values passed to Provision Processor framework for use in building the emailArgList when provisioning succeeds or fails | Returns “Joiner Success Email Template” |
| getJoinerProvFailureEmailTemplateRule | | Returns “Joiner Failure Email Template” |
| getJoinerProvSuccessEmailToRule | | Returns the request target identity’s email address |
| getJoinerProvFailureEmailToRule | | Returns the request target identity’s email address |
| getJoinerApprovedEmailTemplateRule | Calculate values passed to Approval framework (through Provision Processor framework) for use in building the emailArgList when provisioning request is approved or rejected | Returns “Joiner Approved Email Template” |
| getJoinerRejectedEmailTemplateRule | | Returns “Joiner Rejected Email Template” |
| getJoinerApprovedEmailToRule | | Returns the request target identity’s email address |
| getJoinerRejectedEmailToRule | | Returns a default (fictitious) email address – method content must be changed from default |

### Step 6 – Deploy the feature

You must include the customizable files in your build process by moving the files from the feature’s Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer) along with the customizable files from dependent frameworks. The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects

from the feature's /Source folder are automatically deployed with the SSB.  Import the artifacts through the SSB deployment process (or other process).

# Mover

The Mover feature supports automatic provisioning which should occur as identities move around in the organization, such as changing job titles, departments, organizations, etc. Typically, access changes must occur as a result of these sorts of changes.

This feature supports several types of configurations for provisioning access changes for these users. These can each be used alone or they can be combined together in the same mover workflow provisioning process. The mover workflow can provision any or all of these:

- A calculated list of roles which are granted to users based on the role assignment configurations, using the Role Assignment Framework
- Attribute synchronization (sharing attribute values from one account across the organization to other application accounts)
- A calculated set of access defined by a custom rule which creates the provisioning plan dynamically, as defined by the installation's requirements

This feature adds an additional step to the key process followed by all features. At the end of the workflow (after the mover provisioning has been completed), it can optionally create a manager certification to allow the user's manager to review the user's full access.

## Components

These core feature components *should not* be modified or customized per installation:

- **SP LCE Mover Trigger** – the IdentityTrigger which defines the lifecycle event; provided in SP_LCE_Mover_TriggerAndWF.xml
- **SP LCE Mover Trigger Rule** – the rule which determines whether the lifecycle event workflow should be run; provided in SP_LCE_Mover_TriggerAndWF.xml
- **SP LCE Mover WF** – the mover lifecycle event workflow; provided in SP_LCE_Mover_TriggerAndWF.xml
- **SP Mover Rules Library** – the Mover Feature's standard rules library, which contains the trigger logic, the build plan logic, and the certification generation logic; provided in SP_Mover_RulesLibrary.xml

These feature components should be modified and customized per installation:

- **SPCONF Mover Mappings Custom** – the Custom object which defines the criteria for running the mover event and the configuration for the mover provisioning process; provided in SPCONF_Mover_Mappings_Custom.xml
- **SPCONF Mover Rules Library** – the configurable Rules library where all installation-specific logic for the mover event should be written; provided in SPCONF_Mover_RulesLibrary.xml

This feature also uses methods in the **SP Attr Synch Rules Library**, which is part of the Attribute Synchronization Feature, so that feature (or at least that rules library) must also be deployed to use this feature.

## How It Works

### The Trigger

This lifecycle event's trigger allows you to define a "mover" event, and thus trigger the mover workflow, through any of these options:

- A custom rule which evaluates the identity using whatever logic the organization chooses to define
- An IdentitySelector which defines attribute values that designate a mover for the organization
- A comparison of the "before" and "after" values of the user's Links (accounts) to determine if the data values for specified account attributes have changed

Like all of the Feature triggers, this is a rule-based trigger.

1. The **SP LCE Mover Trigger** calls the **SP LCE Mover Trigger Rule** which, in turn, calls the **isTriggerMoverRule** method in the SP Mover Rules Library.
2. The **isTriggerMoverRule** method queries the SPCONF Mover Mappings Custom object for a **Trigger Type** entry which tells how to identify identities which are Movers. The three valid values are:
    a. **Selector** – the method compares the identity to the match expression in the **Trigger Field Selector** entry in the SPCONF Mover Mappings Custom object and returns "true" to run the mover workflow if the identity matches the criteria
    b. **Custom Rule** – the method passes the identity to the rule or rule library method named in **Trigger Custom Rule** in the SPCONF Mover Mappings Custom object; the rule/method examines the identity and returns "true" if the identity is a Mover; the isTriggerMoverRule method then also returns "true" and runs the mover workflow for the identity
    c. **Compare Links** – the method passes the old and new versions of the identity to a rule library method (isNewLinkChanged) in the SP Attr Synch Rules Library (in the Attribute Synchronization feature). This method examines the user's accounts on the applications listed in the SPCONF Mover Mappings Custom object in the **Trigger Compare Links** attribute, and specifically the attributes listed in **Trigger Compare Links Schema**, and runs the mover workflow if any of those attribute values have changed.

```
<!--  Enter the Trigger Type.
        Options Include:
            - Selector:  Will dynamically re-evalute all target attributes and update as necessary.
            - Compare Links:  Will review the links defined in the Compare Links entry below.  If any schema attributes
                have changed from the previous to the new identity, the workflow will launch.
            - Custom Rule:  Will call out to the rule defined in the
                Birthright Assignment Custom Rule entry below.  Rule will receive an Identity object.
                Rule must return a ProvisioningPlan.
-->
<entry key="Trigger Type" value="Compare Links" />

<!-- Used if Plan Construction Type is Custom Rule -->
<entry key="Trigger Custom Rule" value="method:isMoverCustomTriggerRule" />

<!--  Create the Selector to determine whether to kick off the Mover -->
<entry key="Trigger Field Selector">
    <value>
            <IdentitySelector>
              <MatchExpression and="true">
                <MatchTerm name="personStatus" value="A"/>
                <MatchTerm name="joinerDate" value="NULL"/>
              </MatchExpression>
            </IdentitySelector>
    </value>
</entry>

<!-- Per link that is being compared, specify the schema attributes that should be compared.
        This mechanism is used in case it is desired not to launch if a user has only been
        terminated or rehired, meaning it will only check to see if other significant attributes have changed.
         -->
<entry key="Trigger Compare Links Schemas">
    <value>
        <Attributes>
            <Map>
                <entry key="HR">
                    <value>
                        <List>
                            <String>DEPARTMENT_ID</String>
                            <String>MGR_ID</String>
                        </List>
                    </value>
                </entry>
            </Map>
        </Attributes>
    </value>
```

**Mover Trigger Configuration Options in Custom Mapping Object**

## The Workflow

1. The workflow first sets the **requestType**, which can be used by the Approval framework to drive the approval structure. By default, this gets set to “Mover”.
2. A custom rule hook is available to perform installation-specific actions prior to provisioning plan creation. The **moverBeforePlanRule** rule library method provides a template of actions which could be performed here.
3. Next, the workflow builds the provisioning plan based on the provisioning configuration specified in the SPCONF Mover Mappings Custom object’s **Plan Construction Types** attribute. (See the How to Implement section below for more details on this configuration.)
4. If the plan is empty, the **No Request** step offers a custom rule hook to perform any non-provisioning logic required by the installation, in lieu of provisioning.
5. Otherwise, the **Before Provision** step offers a different custom rule hook to allow for any business-specific logic customizations (usually of the provisioning plan) before beginning the provisioning process.

6. Next, the workflow invokes the *Provision Processor* framework to perform the provisioning operation (including approval through the Approval framework).
7. The **After Provision** step then offers another custom rule hook for any business-specific logic which needs to be performed following the provisioning operation.
8. Next, the workflow invokes the *Dynamic Emails* Framework to send out any required emails which were added to the emailArgList throughout any other steps in this workflow or the Provision Processor framework workflow.
9. As its final action step, this workflow can optionally create a manager certification for the user's manager to review their complete, updated set of access. This option can be turned on or off by setting the **Launch Manager Cert** attribute in the SPCONF Mover Mappings Custom object to true or false, respectively. This step is a customization of the generic workflow process for features, which is unique to this feature.

## How to Implement

The basic steps to use the framework are:

1. Configure the target properties to activate the event and determine whether emails will be sent out at the end of the provisioning process
2. Configure the trigger for the workflow
3. Specify the mover access assignment process
4. Configure the details for calculating the new roles/entitlements for the mover user
5. Specify whether to launch a manager certification for the user
6. Configure rules to manage additional actions required by the business
7. Deploy the feature

### Step 1 – Configure Target Properties

The ssf.target.properties file provides a template for all the values required to manage the feature. Locate the Mover Feature Properties section and copy everything in this section to the target.properties file for your environment (if you have not already run the SSD Deployer which does this for you). Set **SP_MOVER_IS_DISABLED** to false to enable the feature, and set other attributes to true or false as described in the comments associated to each property in that section of the file.

```
#*******************************************************************************
#   MOVER FEATURE PROPERTIES
#*******************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_MOVER_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_MOVER_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_MOVER_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_MOVER_SEND_POST_PROVISION_EMAILS%%=false
```

### Step 2 – Configure Trigger Attributes

Trigger attributes define if and when the lifecycle event's workflow will run for a given identity. All Trigger attributes are set in the **SPCONF Mover Mappings Custom** object.

1. Set **Trigger Type** to specify how to decide whether the lifecycle event should be run for a given identity. Trigger Type value must be one of these:
    - **Selector**: workflow trigger is specified in the Trigger Field Selector attribute
    - **Compare Links:** workflow triggers are specified in Trigger Compare Links and Trigger Compare Links Schemas
    - **Custom Rule:** workflow trigger is specified in Trigger Custom Rule attribute
2. If Trigger Type is Selector, define the **Trigger Field Selector** attribute. The Identity Trigger looks for the IdentitySelector in this attribute which specifies the condition for when the workflow should be launched. Use the following as an example:

```
<entry key="Trigger Field Selector">
  <value>
    <IdentitySelector>
      <MatchExpression and="true">
        <MatchTerm name="personStatus" value="A"/>
        <MatchTerm name="joinerDate" value="NULL"/>
      </MatchExpression>
    </IdentitySelector>
  </value>
</entry>
```

3. If Trigger Type is Custom Rule, define the **Trigger Custom Rule** attribute. This attribute should specify the name of a rule or a rule library method which will evaluate the identity and return "true" if the mover event should be run for the identity.
    - The SPCONF Mover Mappings Custom object predefines this Trigger Custom Rule attribute to point to a rule library method in the SPCONF Mover Rule Library, called **isMoverCustomTriggerRule**. The recommendation is to leave that entry as-is and to specify your custom logic for identifying a mover identity in that method in the SPCONF Mover Rule Library.
4. If Trigger Type is Compare Links, define the **Trigger Compare Links** and **Trigger Compare Links Schemas** attributes.
    - **Trigger Compare Links** lists the applications for which accounts should be examined.
    - **Trigger Compare Links Schemas** then lists the schema attributes on each of those applications which should be compared.
    - The trigger logic retrieves the identity's application accounts on the listed application(s) and examines each attribute named in the Trigger Compare Links Schema for each application. If the identity's attribute value for any of those attributes has changed, the mover workflow is launched.

    For example, this entry tells the trigger rule to evaluate the user's HR application account links.

```
<entry key="Trigger Compare Links">
   <value>
      <List>
         <String>HR</String>
      </List>
   </value>
</entry>
```

    Then this entry tells the trigger rule to compare the before and after values of the HR application's DEPARTMENT_ID and MGR_ID attributes.

```
<entry key="Trigger Compare Links Schemas">
   <value>
     <Attributes>
        <Map>
           <entry key="HR">
             <value>
                <List>
                  <String>DEPARTMENT_ID</String>
                  <String>MGR_ID</String>
                </List>
             </value>
           </entry>
        </Map>
     </Attributes>
   </value>
</entry>
```

**NOTE**: The logic in the rule assumes the user only has one account on the applications being examined. If more than one account can exist per user, the logic must be customized to account for that.

### Step 3 – Specify New Access Assignment Process

Your mover logic will be specific to your business, so the feature allows you to configure the access change logic to suit your needs.

Set the **Plan Construction Types** entry in the SPCONF Mover Mappings Custom object to determine how the provisioning plan is constructed in the Build Plan step of the workflow. This attribute, which is specified as a list, can be any or all of these three values:

1. **Dynamic Roles**: The workflow will use the Role Assignment Framework to dynamically assign roles to the identity. This, of course, requires that framework to be in place and configured.
2. **Attribute Synch**: The workflow will use the Attribute Synch Framework to create or extend the plan with target attribute updates
3. **Custom Rule**: The workflow uses the rule specified in the **Plan Construction Custom Rule** entry of the SPCONF Mover Mappings Custom object to build or append to the provisioning plan to provision the desired access.

### Step 4 – Configure the Assignment Details

Depending on the Plan Construction Type specified in step 3, additional attributes must be specified in the SPCONF Mover Mappings Custom object or elsewhere, as noted here.

1. If the assignment type contains **Dynamic Roles**, then configure the Role Assignment Framework as described in that section of this document.
2. If the assignment type contains **Attribute Synch**, then configure the Attribute Synch Feature as described in that section of this document.
    - **NOTE**: This does not trigger the Attribute Synch Feature workflow separately; instead, this process uses the rule logic and configurations in that feature to build an attribute synchronization *part* of the provisioning plan and append it to the Dynamic Roles plan calculated here (if any). Usually, the Attribute Synch option is specified along with either Dynamic Roles or Custom Rule, rather than by itself; otherwise, the Attribute Synch Feature would just be used directly.

3. If the assignment type contains **Custom Rule**, enter a rule name in the **Plan Construction Custom Rule** entry of the SPCONF Mover Mappings Custom object.
    - This entry is predefined in the SPCONF Mover Mappings Custom object to point to a rule library method in the SPCONF Mover Rule Library called buildCustomMoverPlan. The recommendation is to leave that entry as-is in the Custom object and write your desired business logic in that rule library method.
    - This rule/method is passed a ProvisioningPlan object (in the variable "plan" which is recorded inside the "workflow" argument passed to the rule). It should append to that plan and return the full plan to be provisioned.

### Step 5 – Specify manager certification requirements

If the mover workflow should generate a manager certification at the end of the process, set the **Launch Manager Cert** attribute in the SPCONF Mover Mappings Custom object to "true". This allows the user's manager to review all of the user's access, after the access change provisioned in this workflow, to validate its accuracy.

### Step 6 – Configure the Hooks

Most of the mover workflow's steps simply call methods in the **SPCONF Mover Rules Library**, where you can write your own logic to further customize the behavior of the mover workflow. As provided, the methods may be empty or may provide some template/default logic.

| Method | Calling Workflow Step | Default Logic Provided |
|---|---|---|
| getMoverRequestTypeRule | Get Request Type | Returns "Mover" |
| moverBeforePlanRule | Before Plan | Provides logic to retrieve identityModel and spExtAttr variables from the workflow and return them to the workflow; implementer should insert any logic to modify these variables as desired or perform any other before-plan-generation logic desired |
| moverNoRequestRule | No Request (only if Build Plan step does not return provisioningPlan) | None |
| moverBeforeProvisionRule | Before Provision | None |
| moverAfterProvisionRule | After Provision | None |
| buildCustomMoverPlan | Build Plan (if Plan Construction Type = Custom Rule and Plan Construction Custom Rule is left pointing to this method) | Returns empty ProvisioningPlan, though intention is for implementer to retrieve the provisioning plan from the workflow argument and append to it |
| | | |

| | | |
|---|---|---|
| The remainder of the rule hooks in the SPCONF Mover Rule Library are for configuring email message “to” addresses and templates to be used by the Provision Processor and Approval frameworks for queuing up information messages to be sent at the end of the workflow process. | | |
| getMoverProvSuccessEmailTemplateRule | Calculate values passed to Provision Processor framework for use in building the emailArgList when provisioning succeeds or fails | Returns “Mover Success Email Template” |
| getMoverProvFailureEmailTemplateRule | | Returns “Mover Failure Email Template” |
| getMoverProvSuccessEmailToRule | | Returns the request target identity’s email address |
| getMoverProvFailureEmailToRule | | Returns the request target identity’s email address |
| getMoverApprovedEmailTemplateRule | Calculate values passed to Approval framework (through Provision Processor framework) for use in building the emailArgList when provisioning request is approved or rejected | Returns “Mover Approved Email Template” |
| getMoverRejectedEmailTemplateRule | | Returns “Mover Rejected Email Template” |
| getMoverApprovedEmailToRule | | Returns the request target identity’s email address |
| getMoverRejectedEmailToRule | | Returns a default (fictitious) email address – method content must be changed from default |

### Step 7 – Deploy the feature

You must include the customizable files in your build process by moving the files from the feature’s Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer) along with the customizable files from dependent frameworks. The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects from the feature’s /Source folder are automatically deployed with the SSB. Import the artifacts through the SSB deployment process (or other process).

# Attribute Synch

The Attribute Synch feature drives attribute synchronization through a Lifecycle Event workflow. This feature is different from the out-of-the-box attribute synchronization option. The out-of-the box attribute synchronization functionality pushes all attribute synchronization through top-level identity attributes. This means that if you want to synchronize an attribute from one application to another, you must record it as an identity attribute, not just as an attribute of those two application accounts.

This feature supports provisioning of data values directly from one application account to another at the account level, with no identity attribute involvement required. In most cases, authoritative attribute values are recorded on one central, authoritative application (usually the HR data source) and are synchronized to one or more other applications in the environment. However, this feature also supports sourcing attributes from more than one source application, if needed.

## Components

These core feature components *should not* be modified or customized per installation:

- **SP LCE Attr Synch Trigger** – the IdentityTrigger which defines the lifecycle event; provided in SP_LCE_AttrSynch_TriggerAndWF.xml
- **SP LCE Attr Synch Trigger Rule** – the rule which determines whether the lifecycle event workflow should be run; provided in SP_LCE_ AttrSynch _TriggerAndWF.xml
- **SP LCE Attr Synch WF** – the lifecycle event workflow; provided in SP_LCE_ AttrSynch _TriggerAndWF.xml
- **SP Attr Synch Rules Library** – the Attribute Synch Feature's standard rules library, which contains the trigger logic, the build plan logic, and the certification generation logic; provided in SP_ AttrSynch _RulesLibrary.xml

These feature components should be modified and customized per installation:

- **SPCONF Attr Synch Mappings Custom** – the Custom object which defines the criteria for running the attribute synchronization event and the configuration for the attribute synchronization provisioning process; provided in SPCONF_ AttrSynch _Mappings_Custom.xml
- **SPCONF Attr Synch Rules Library** – the configurable Rules library where all installation-specific logic for the attribute synch event should be written; provided in SPCONF_ AttrSynch _RulesLibrary.xml

## How It Works

### The Trigger

This lifecycle event's trigger allows you to launch the attribute synchronization workflow through any of these options:

- A custom rule which evaluates the identity using whatever logic the organization chooses to define
- An IdentitySelector which defines attribute values that designate a need to run attribute synchronization

- A comparison of the “before” and “after” values of the user’s Links (accounts) on one or more applications to determine if the data values for specified account attributes have changed

**NOTE**: In some cases, this lifecycle event may overlap with other lifecycle events, usually Mover events, and therefore you may not want to run this event if other events have been identified to run at the same time. The trigger configuration allows you to bypass running this event when other events are also being run.

Like all of the Feature triggers, this is a rule-based trigger.

1. The **SP LCE Attr Synch Trigger** calls the **SP LCE Attr Synch Trigger Rule** which, in turn, calls the **isTriggerAttrSynchRule** method in the SP Attr Synch Rules Library.
2. The **isTriggerAttrSynchRule** method queries the SPCONF Attr Synch Mappings Custom object for an entry called **Trigger If No Other LCE Triggered**. If that is set to “true”, this event will only run if no other lifecycle events are being triggered by the identity change currently being evaluated. This allows you to bypass this event if, for example, a Mover event which will include this attribute synchronization process is also configured and being triggered at the same time.
3. If the event is not suppressed (flag in step 2 turned off or no other events found), the **isTriggerAttrSynchRule** method then queries the SPCONF Attr Synch Mappings Custom object for a **Trigger Type** entry which tells how to identify identities which require attribute synchonization. The three valid values are
    a. **Custom Rule** – the method passes the identity to the rule or rule library method named in **Trigger Custom Rule** in the SPCONF Attr Synch Mappings Custom object; that rule/method examines the identity and returns “true” if the attribute synchronization workflow should run (The isTriggerAttrSynchRule method then also returns “true” and runs the attribute synch workflow for the identity.)
    b. **Selector** – the method compares the identity to the match expression in the **Trigger Field Selector** entry in the SPCONF Attr Synch Mappings Custom object and returns “true” to run the workflow if the identity matches the criteria
    c. **Compare Links** – the method passes the old and new versions of the identity to a rule library method **isNewLinkChanged** in the SP Attr Synch Rules Library. This method examines the user’s accounts on the applications listed in the SPCONF Attr Synch Mappings Custom object in the **Trigger Compare Links** attribute, and specifically the attributes listed in **Trigger Compare Links Schema**, and runs the attribute synchronization workflow if any of those attribute values have changed. Only the source applications whose attributes should be watched for changes will be listed in these entries.

### The Workflow

1. The workflow first sets the **requestType**, which can be used by the Approval framework to drive the approval structure. By default, this gets set to “AttrSynch”.
2. A custom rule hook is available to perform installation-specific actions prior to provisioning plan creation. The **attrSynchBeforePlanRule** rule library method provides a template of actions which could be performed here.

3. Next, the workflow builds the provisioning plan based on the provisioning configuration specified in the SPCONF Attr Synch Mappings Custom object's **Plan Construction Type** attribute. (See the How to Implement section below for more details on this configuration.)
4. If the plan is empty, the **No Request** step offers a custom rule hook to perform any non-provisioning logic required by the installation, in lieu of provisioning.
5. Otherwise, the **Before Provision** step offers a different custom rule hook to allow for any business-specific logic customizations (usually of the provisioning plan) before beginning the provisioning process.
6. Next, the workflow invokes the *Provision Processor* framework to perform the provisioning operation (including approval through the Approval framework).
7. The **After Provision** step then offers another custom rule hook for any business-specific logic which needs to be performed following the provisioning operation.
8. Next, the workflow invokes the *Dynamic Emails* Framework to send out any required emails which were added to the emailArgList throughout any other steps in this workflow or the Provision Processor framework workflow.

## How to Implement

The basic steps to use the framework are:

1. Configure the target properties to activate the event and determine whether emails will be sent out at the end of the provisioning process
2. Configure the trigger for the workflow
3. Specify the attribute synchronization calculation process and
4. Configure the details for calculating the attribute synchronization provisioning plan
5. Configure rules to manage additional actions required by the business
6. Deploy the feature

### Step 1 – Configure Target Properties

The ssf.target.properties file provides a template for all the values required to manage the feature. Locate the Attr Synch Feature Properties section and copy everything in this section to the target.properties file for your environment (if you have not already run the SSD Deployer which does this for you). Set **SP_ATTR_SYNCH_IS_DISABLED** to false to enable the feature, and set other attributes to true or false as described in the comments associated to each property in that section of the file.

```
#*********************************************************************************
#   ATTR SYNCH FEATURE PROPERTIES
#*********************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_ATTR_SYNCH_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_ATTR_SYNCH_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_ATTR_SYNCH_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_ATTR_SYNCH_SEND_POST_PROVISION_EMAILS%%=false
## ENTER true/false ON WHETHER TO RUN PROVISIONING IN THE FOREGROUND
%%SP_ATTR_SYNCH_FOREGROUND_PROVISIONING%%=false
```

### Step 2 – Configure Trigger Attributes

Trigger attributes define if and when the lifecycle event's workflow will run for a given identity. All Trigger attributes are set in the **SPCONF Attr Synch Mappings Custom** object.

1. Set **Trigger Type** to specify how to decide whether the lifecycle event should be run for a given identity. Trigger Type value must be one of these:
    - **Selector**: workflow trigger is specified in the Trigger Field Selector attribute
    - **Compare Links:** workflow triggers are specified in Trigger Compare Links and Trigger Compare Links Schemas
    - **Custom Rule:** workflow trigger is specified in Trigger Custom Rule attribute
2. If Trigger Type is Selector, define the **Trigger Field Selector** attribute. The Identity Trigger looks for the IdentitySelector in this attribute which specifies the condition for when the workflow should be launched. Use the following as an example:

```
<entry key="Trigger Field Selector">
  <value>
    <IdentitySelector>
      <MatchExpression and="true">
        <MatchTerm name="personStatus" value="A"/>
        <MatchTerm name="joinerDate" value="NULL"/>
      </MatchExpression>
    </IdentitySelector>
  </value>
</entry>
```

    **NOTE**: For the Attribute Synchronization Feature, this Identity Selector most commonly references Link attributes rather than identity attributes, though either could be used.

3. If Trigger Type is Custom Rule, define the **Trigger Custom Rule** attribute. This attribute should specify the name of a rule or a rule library method which will evaluate the identity and return "true" if the joiner event should be run for the identity.
    - The SPCONF Attr Synch Mappings Custom object predefines this Trigger Custom Rule attribute to point to a rule library method in the SPCONF Attr Synch Rules Library, called **isAttrSynchCustomTriggerRule**. The recommendation is to leave that entry as-is and to specify your custom logic for triggering attribute synchronization in that method in the SPCONF Attr Synch Rules Library.
4. If Trigger Type is Compare Links, define the **Trigger Compare Links** and **Trigger Compare Links Schemas** attributes.
    - **Trigger Compare Links** lists the applications for which accounts should be examined. This should be the list of applications which contain authoritative attributes which require synchronization to other applications.
    - **Trigger Compare Links Schemas** then lists the schema attributes on each of those applications which should be compared. This is the list of authoritative attributes from each application.
    - The trigger logic retrieves the identity's application accounts on the listed application(s) and examines each attribute named in the Trigger Compare Links Schema for each application. If the identity's attribute value for any of those attributes has changed, the attribute synchronization workflow is launched.

For example, this entry tells the trigger rule to evaluate the user's HR application account links.

```
<entry key="Trigger Compare Links">
   <value>
      <List>
         <String>HR</String>
      </List>
   </value>
</entry>
```

Then this entry tells the trigger rule to compare the before and after values of the HR application's DEPARTMENT_ID and MGR_ID attributes.

```
<entry key="Trigger Compare Links Schemas">
   <value>
     <Attributes>
        <Map>
           <entry key="HR">
             <value>
                <List>
                  <String>DEPARTMENT_ID</String>
                  <String>MGR_ID</String>
                </List>
             </value>
           </entry>
        </Map>
     </Attributes>
   </value>
</entry>
```

**NOTE**: The logic in the rule assumes the user only has one account on the applications being examined. If more than one account can exist per user, the logic must be customized to account for that.

### Step 3 – Specify Attribute Synchronization Calculation Process

The feature allows you to configure how the provisioning plan should be constructed for synchronizing attributes to the target systems.

Set the **Plan Construction Type** entry in the SPCONF Attr Synch Mappings Custom object to determine how the provisioning plan is constructed in the Build Plan step of the workflow. This attribute can be any or all of these three values:

1. **Dynamic Targets**: The workflow dynamically re-evaluates all target attributes for the identity's accounts and updates them as necessary.
2. **Custom Rule**: The workflow uses the rule specified in the **Plan Construction Custom Rule** entry of the SPCONF Attr Synch Mappings Custom object to build or append to the provisioning plan to provision the desired access. This is a less common option used in this workflow.

**NOTE**: Unlike the Mover feature, this feature allows specification of only one of these construction types at a time.

### Step 4 – Configure the Plan Creation Details

1. If the Plan Construction Type in step 3 above is **Custom Rule**, enter a rule name in the **Plan Construction Custom Rule** entry of the SPCONF Attr Synch Mappings Custom object.

- This entry is predefined in the SPCONF Attr Synch Mappings Custom object to point to a rule library method in the SPCONF Attr Synch Rule Library called buildCustomAttrSynchPlan. The recommendation is to leave that entry as-is in the Custom object and write your desired business logic in that rule library method.
- This rule/method should build and return a provisioning plan that contains the full set of attributes to provision in the synchronization operation.

2. Otherwise, if the Plan Construction Type is **Dynamic Targets**, set the **Excluded Applications** attribute to include applications whose attribute values should *not* by changed through this synchronization workflow. Set the **Application Skip Fields** attribute to include individual attributes per application to ignore when calculating the attribute synchronization provisioning plan.
    - The **Application Skip Fields** is a map organized by application name; it only needs to include attributes on applications which are *not* listed in Excluded Applications. All attributes on the applications listed in Excluded Applications will be omitted from the provisioning operation.
3. When using the **Dynamic Targets** option, ensure that the attribute values in the Create Provisioning Policy for each application specify the logic for calculating the attributes based on the synchronization source, wherever attribute synchronization is applicable.

### Step 5 – Configure the Hooks

Most of the workflow's steps simply call methods in the **SPCONF Attr Synch Rules Library**, where you can write your own logic to further customize the behavior of the attribute synchronization workflow. As provided, the methods may be empty or may provide some template/default logic.

| Method | Calling Workflow Step | Default Logic Provided |
|---|---|---|
| getAttrSynchRequestTypeRule | Get Request Type | Returns "AttrSynch" |
| attrSynchBeforePlanRule | Before Plan | Provides logic to retrieve identityModel and spExtAttr variables from the workflow and return them to the workflow; implementer should insert any logic to modify these variables as desired or perform any other before-plan-generation logic desired |
| attrSynchNoRequestRule | No Request (only if Build Plan step does not return provisioningPlan) | None |
| attrSynchBeforeProvisionRule | Before Provision | None |
| attrSynchAfterProvisionRule | After Provision | None |
| buildCustomAttrSynchPlan | Build Plan (if Plan Construction Type = Custom Rule and Plan Construction | Returns empty ProvisioningPlan, though the intention is for it to build and return the full provisioning |

| | | |
|---|---|---|
| | Custom Rule is left pointing to this method) | plan for the whole required provisioning operation |
| The remainder of the rule hooks in the SPCONF Attr Synch Rule Library are for configuring email message “to” addresses and templates to be used by the Provision Processor and Approval frameworks for queuing up information messages to be sent at the end of the workflow process. | | |
| getAttrSynchProvSuccessEmailTemplateRule | Calculate values passed to Provision Processor framework for use in building the emailArgList when provisioning succeeds or fails | Returns “AttrSynch Success Email Template” |
| getAttrSynchProvFailureEmailTemplateRule | | Returns “AttrSynch Failure Email Template” |
| getAttrSynchProvSuccessEmailToRule | | Returns the request target identity’s email address |
| getAttrSynchProvFailureEmailToRule | | Returns the request target identity’s email address |
| getAttrSynchApprovedEmailTemplateRule | Calculate values passed to Approval framework (through Provision Processor framework) for use in building the emailArgList when provisioning request is approved or rejected | Returns “AttrSynch Approved Email Template” |
| getAttrSynchRejectedEmailTemplateRule | | Returns “AttrSynch Rejected Email Template” |
| getAttrSynchApprovedEmailToRule | | Returns the request target identity’s email address |
| getAttrSynchRejectedEmailToRule | | Returns a default (fictitious) email address – method content must be changed from default |

### Step 6 – Deploy the feature

You must include the customizable files in your build process by moving the files from the feature’s Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer) along with the customizable files from dependent frameworks. The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects from the feature’s /Source folder are automatically deployed with the SSB. Import the artifacts through the SSB deployment process (or other process).

# Leaver

The Leaver feature is used to remove users' access upon terminations and during leaves of absence. It supports several different options for removing access:

- Disabling all accounts
- Deleting all accounts
- Selectively disabling some accounts while deleting others
- Performing operations specified in a custom rule (which allows even more flexibility in the actions to be taken)

This feature adds an additional step to the key process followed by all features. At the end of the workflow (after the leaver provisioning has been completed), it can optionally schedule a future-dated workflow request to run a second workflow process for the leaver user. This is commonly used to implement a two-stage disable-delete leaver process.

## Components

These core feature components *should not* be modified or customized per installation:

- **SP LCE Leaver Trigger** – the IdentityTrigger which defines the lifecycle event; provided in SP_LCE_Leaver_TriggerAndWF.xml
- **SP LCE Leaver Trigger Rule** – the rule which determines whether the lifecycle event workflow should be run; provided in SP_LCE_Leaver_TriggerAndWF.xml
- **SP LCE Leaver WF** – the lifecycle event workflow; provided in SP_LCE_Leaver_TriggerAndWF.xml
- **SP Leaver Rules Library** – the Leaver Feature's standard rules library, which contains the trigger logic and the build plan logic; provided in SP_Leaver_RulesLibrary.xml

These feature components should be modified and customized per installation:

- **SPCONF Leaver Mappings Custom** – the Custom object which defines the criteria for running the leaver event; provided in SPCONF_Leaver_Mappings_Custom.xml
- **SPCONF Leaver Rules Library** – the configurable Rules library where all installation-specific logic for the leaver event should be written; provided in SPCONF_Leaver_RulesLibrary.xml

## How It Works

### The Trigger

Out of the box, IdentityIQ defines a "leaver" event as marking an identity as inactive, and the default leaver event disables all accounts associated with that identity. This lifecycle event could be triggered based on the identity's inactive attribute but also supports other configurations.

Specifically, the trigger options for this workflow are:

- An IdentitySelector which defines attribute values that designate a leaver for the organization
- A custom rule which evaluates the identity using whatever logic the organization chooses to define

Like all of the Feature triggers, this is a rule-based trigger.

1. The **SP LCE Leaver Trigger** calls the **SP LCE Leaver Trigger Rule** which, in turn, calls the **isTriggerLeaverRule** method in the SP Leaver Rules Library.
2. The **isTriggerLeaverRule** method queries the SPCONF Leaver Mappings Custom object for a **Trigger Type** entry which tells how to identify identities which are Leavers. The two valid values are
    a. **Selector** – the method compares the identity to the match expression in the **Trigger Field Selector** entry in the SPCONF Leaver Mappings Custom object and returns “true” to run the leaver workflow if the identity matches the criteria
    b. **Custom Rule** – the method passes the identity to the rule or rule library method named in **Trigger Custom Rule** in the SPCONF Leaver Mappings Custom object; the rule/method examines the identity and returns “true” if the identity is a Leaver; the isTriggerLeaverRule method then also returns “true” and runs the leaver workflow for the identity (default rule to use is SPCONF Rule Library method isLeaverCustomTriggerRule)

```
<!--  Enter the Trigger Type.
        Options Include:
            - Selector:  Will dynamically re-evalute all target attributes and update as necessary.
            - Custom Rule:  Will call out to the rule defined in the
                Birthright Assignment Custom Rule entry below.  Rule will receive an Identity object.
                Rule must return a ProvisioningPlan.
-->
<entry key="Trigger Type" value="Selector" />

<!-- Create the Selector to determine whether to kick off the Leaver -->
<entry key="Trigger Field Selector">
    <value>
        <IdentitySelector>

            <MatchExpression and="true">
                <MatchTerm name="userStatus" value="Terminated" />
                <MatchTerm container="true">
                    <MatchTerm name="inactive" value="false" />
                    <MatchTerm name="inactive" />
                </MatchTerm>
            </MatchExpression>

        </IdentitySelector>
    </value>
</entry>

<!-- Used if Trigger Type is Custom Rule.
    Recommended that this value isn't changed
    and the existing method in SP CONF Joiner Rules Library is used-->
<entry key="Trigger Custom Rule" value="method:isLeaverCustomTriggerRule" />
```

**Leaver Trigger Configuration Options in Custom Mapping Object**

## The Workflow

1. The workflow first sets the **requestType**, which can be used by the Approval framework to drive the approval structure. By default, this gets set to “Leaver”.
2. A custom rule hook is available to perform installation-specific actions prior to provisioning plan creation. The **leaverBeforePlanRule** rule library method provides a template of actions which could be performed here.
3. Next, the workflow builds the provisioning plan based on the birthright provisioning configuration specified in the SPCONF Leaver Mappings Custom object’s **Leaver Build Plan Type** attribute.
4. If the plan is empty, the **No Request** step offers a custom rule hook to perform any non-provisioning logic required by the installation, in lieu of provisioning.

5. Otherwise, the **Before Provision** step offers a different custom rule hook to allow for any business-specific logic customizations (usually of the provisioning plan) before beginning the provisioning process.
6. Next, the workflow invokes the *Provision Processor* framework to perform the provisioning operation (including approval through the Approval framework).
7. The **After Provision** step then offers another custom rule hook for any business-specific logic which needs to be performed following the provisioning operation.
8. The workflow then invokes the *Dynamic Emails* Framework to send out any required emails which were added to the emailArgList throughout any other steps in this workflow or the Provision Processor framework workflow.
9. Lastly, the workflow can schedule a “sunset” workflow request – a scheduled event to run a specified workflow at a specified future date. The most common use case for this is customers who want to terminate users’ access by disabling accounts initially and then, after 60 or 90 days, deleting the accounts. The “sunset” request would be created to process the delete step. **NOTE**: The workflow which is launched by the future-dated request is not included in this framework and requires custom development per installation.

## How to Implement

The basic steps to use this feature are:

1. Configure the target properties to activate the event and determine whether emails will be sent out at the end of the provisioning process
2. Configure the trigger for the workflow
3. Specify the leaver build plan type
4. Configure the details for calculating the leaver provisioning plan
5. Configure rules to manage additional actions required by the business
6. Configure the sunset workflow, if it is to be used
7. Deploy the feature

### Step 1 – Configure Target Properties

The ssf.target.properties file provides a template for all the values required to manage the feature. Locate the Leaver Feature Properties section and copy everything in this section to the target.properties file for your environment (if you have not already run the SSD Deployer which does this for you). Set **SP_LEAVER_IS_DISABLED** to false to enable the feature, and set other attributes to true or false as described in the comments associated to each property in that section of the file, including options for sunsetting the identity.

```
#*********************************************************************************
#   LEAVER FEATURE PROPERTIES
#*********************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_LEAVER_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_LEAVER_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_LEAVER_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_LEAVER_SEND_POST_PROVISION_EMAILS%%=false
## ENTER true/false ON WHETHER TO RUN PROVISIONING IN THE FOREGROUND
%%SP_LEAVER_FOREGROUND_PROVISIONING%%=false
%%SP_LEAVER_SUNSET_REQUEST_NAME%%=SP Leaver Sunset
%%SP_LEAVER_SUNSET_WF%%=SP Cust Sunset WF
%%SUNSET_SCHEDULE_SECONDS%%=7200
```

### Step 2 – Configure Trigger Attributes

Trigger attributes define if and when the lifecycle event's workflow will run for a given identity. All Trigger attributes are set in the **SPCONF Leaver Mappings Custom** object.

1. Set **Trigger Type** to specify how to decide whether the lifecycle event should be run for a given identity. Trigger Type value must be one of these:
    - **Selector**: trigger conditions will be defined by the Trigger Field Selector attribute
    - **Custom Rule:** trigger conditions will be the Trigger Custom Rule attribute
2. If Trigger Type is Selector, define the **Trigger Field Selector** attribute. The Identity Trigger looks for the IdentitySelector in this attribute which specifies the condition for when the workflow should be launched. Use the following as an example:

```
<entry key="Trigger Field Selector">
   <value>
       <IdentitySelector>
         <MatchExpression and="true">
              <MatchTerm name="userStatus" value="Terminated" />
              <MatchTerm container="true">
                 <MatchTerm name="inactive" value="false" />
                 <MatchTerm name="inactive" />
              </MatchTerm>
         </MatchExpression>
       </IdentitySelector>
   </value>
</entry>>
```

3. If Trigger Type is Custom Rule, define the **Trigger Custom Rule** attribute. This attribute should specify the name of a rule or a rule library method which will evaluate the identity and return "true" if the leaver event should be run for the identity.
    - The SPCONF Leaver Mappings Custom object predefines this Trigger Custom Rule attribute to point to a rule library method in the SPCONF Leaver Rule Library, called **isLeaverCustomTriggerRule**. The recommendation is to leave that entry as-is and to write your custom logic for identifying a leaver identity in that method in the **SPCONF Leaver Rule Library**.

### Step 3 – Select the Leaver Build Type Plan

The **Leaver Build Plan Type** entry of the SPCONF Leaver Mappings Custom object determines how the provisioning plan is constructed in the build plan step. Enter one of four options:

1. **Disable All** – This will disable all accounts currently held by the user.

2. **Delete All** – This will delete all current accounts currently held by the user.
3. **Selective Lists** – This will strategically disable and delete accounts based on selective lists of applications, as set in Step 4 below.
4. **Custom Rule** – This will use the rule defined in the **Leaver Build Plan Custom Rule** entry of the mapping object.

### Step 4 – Configure Lists or Custom Rule

1. If the Leaver Build Plan Type is **Disable All** or **Delete All**, no additional configuration is required for the build plan logic. The lifecycle event workflow will automatically add a request to the provisioning plan to disable or delete each account, respectively.
2. If the Leaver Build Plan Type is **Selective Lists**, simply enter the list of applications in either the Default Disables or Default Deletes entries in the SPCONF Leaver Mappings Custom object.
    - **Default Deletes** lists all applications for which delete requests will be created
    - **Default Disables** lists all applications for which disable requests will be created.

    **NOTE**: If an application is not included in either of these lists, its accounts will be skipped and neither deleted nor disabled in the Leaver functionality.

```
<!-- Enter the Birthright Assignment Type. Options Include: -
Disable All: Will dynamically disable all accounts. -
Delete All: Will dynamically delete all accounts. -
Selective Lists: Will disable all in the Default Disables
    list and Delete all in the Default Deletes list
Custom Rule: Will call rule defined in Leaver Build Plan Custom Rule -->
<entry key="Leaver Build Plan Type" value="Disable All" />

<!-- Used if Leaver Build Plan Type is Selective Lists -->
<entry key="Default Disables">
    <value>
        <List>
            <!-- Enter the names of the Applications for automatic disablement -->
            <String>Active Directory</String>
            <String>Enterprise LDAP</String>

        </List>
    </value>
</entry>

<!-- Used if Leaver Build Plan Type is Selective Lists -->
<entry key="Default Deletes">
    <value>
        <List>
            <!-- Enter the names of the Applications for automatic deletion -->
            <String>Active Directory</String>
            <String>Enterprise LDAP</String>

        </List>
    </value>
</entry>
```

3. If the Leaver Build Plan Type is **Custom Rule**, enter a rule name in the **Leaver Build Plan Custom Rule** entry of the SPCONF Leaver Mappings Custom object.
    - This entry is predefined in the SPCONF Leaver Mappings Custom object to point to a rule library method in the SPCONF Leaver Rule Library called buildCustomLeaverPlan. You should leave that entry as-is in the Custom object and write your desired business logic in that rule library method. This rule/method should return a ProvisioningPlan object.

```
<!-- Used if Leaver Build Plan Type is Custom Rule
    Recommended that this value isn't changed
    and the existing method in SP CONF Joiner Rules Library is used -->
<entry key="Leaver Build Plan Custom Rule" value="method:buildCustomLeaverPlan" />
```

The method is found in **SPCONF Leaver Rules Library**. In the method, you can write your custom logic.

```
public static ProvisioningPlan buildCustomLeaverPlan(SailPointContext context, Workflow workflow){
    ProvisioningPlan plan = new ProvisioningPlan();
    return plan;
}
```

### Step 5 – Configure the Hooks

Most of the leaver workflow's steps simply call methods in the **SPCONF Leaver Rule Library**, where you can write your own logic to further customize the behavior of the leaver workflow. As provided, the methods may be empty or may provide some template/default logic.

| Method | Calling Workflow Step | Default Logic Provided |
|---|---|---|
| getLeaverRequestTypeRule | Get Request Type | Returns "Leaver" |
| leaverBeforePlanRule | Before Plan | Provides logic to retrieve identityModel and spExtAttr variables from the workflow and return them to the workflow; implementer should insert any logic to modify these variables as desired or perform any other before-plan-generation logic desired |
| leaverNoRequestRule | No Request (only if Build Plan step does not return provisioningPlan) | Sets an extended attribute called leaverAccountDate to today and the standard inactive attribute to false |
| leaverBeforeProvisionRule | Before Provision | None |
| leaverAfterProvisionRule | After Provision | Sets an extended identity attribute called leaverDate to the current date |
| buildCustomLeaverPlan | Build Plan (if Birthright Assignment Type = Custom Rule and Birthright Assignment Custom Rule is left pointing to this method) | Returns empty ProvisioningPlan |
| getScheduleSunset | Get Schedule Sunset | Returns "false" |
| getSunsetDate | Schedule Sunset | Returns current date and time |
| | | |

| | | |
|---|---|---|
| The remainder of the rule hooks in the SPCONF Leaver Rule Library are for configuring email message “to” addresses and templates to be used by the Provision Processor and Approval frameworks for queuing up information messages to be sent at the end of the workflow process. | | |
| getLeaverProvSuccessEmailTemplateRule | Calculate values passed to Provision Processor framework for use in building the emailArgList when provisioning succeeds or fails | Returns “Leaver Success Email Template” |
| getLeaverProvFailureEmailTemplateRule | | Returns “Leaver Failure Email Template” |
| getLeaverProvSuccessEmailToRule | | Returns the request target identity’s email address |
| getLeaverProvFailureEmailToRule | | Returns the request target identity’s email address |
| getLeaverApprovedEmailTemplateRule | Calculate values passed to Approval framework (through Provision Processor framework) for use in building the emailArgList when provisioning request is approved or rejected | Returns “Leaver Approved Email Template” |
| getLeaverRejectedEmailTemplateRule | | Returns “Leaver Rejected Email Template” |
| getLeaverApprovedEmailToRule | | Returns the request target identity’s email address |
| getLeaverRejectedEmailToRule | | Returns a default (fictitious) email address – method content must be changed from default |

### Step 6 – Configure the Sunset workflow

If you will be using a future-dated workflow to process any part of the leaver functionality, this requires a couple of extra configurations. The most common use case for this step is a two-phase access termination process where the initial action is to disable accounts and the second phase deletes the accounts after a specified period of time.

1. The **getScheduleSunset** method in the SPCONF Leaver Rules Library determines whether a future dated workflow request should be run at all. By default, it returns “false”. When the sunset workflow should be executed, it must be configured to return “true”.
2. The **getSunsetDate** method specifies when to run the sunset workflow. By default, it returns the current date/time. It must instead be configured to return the desired execution date.
3. The ssf.target.properties file includes a **%%SP_LEAVER_SUNSET_WF%%** attribute which specifies which workflow to run. This must specify the name of the desired workflow to run.
4. The workflow which the scheduled event launches needs to be written.

### Step 7 – Deploy the feature

You must include the customizable files in your build process by moving the files from the feature’s Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer) along with the customizable files from dependent frameworks. The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects from the feature’s /Source folder are automatically deployed with the SSB. Import the artifacts through the SSB deployment process (or other process).

# Rehire

The Rehire feature is used for identities that had been terminated or placed on leave and are now returning as an active user. The feature recalculates the user's access, updates target attributes, and/or enables existing access.

The out of the box definition of a "rehire" lifecycle event is triggered when an identity goes from being marked "inactive" to being marked "active" and involves re-enabling their disabled accounts. This workflow allows much more flexibility than that.

It supports several different options for recalculating access:

- Based on the Role Assignment Framework to provision roles appropriate to the identity
- Based on the Attribute Synch Feature to synchronize attribute values from the authoritative application to other applications in the environment
- By examining existing accounts and re-enabling them (if the application is listed as one which should be re-enabled here)
- Through a custom rule which can calculate the required set of access to provision through any custom logic required for the business needs

## Components

These core feature components *should not* be modified or customized per installation:

- **SP LCE Rehire Trigger** – the IdentityTrigger which defines the lifecycle event; provided in SP_LCE_Rehire_TriggerAndWF.xml
- **SP LCE Rehire Trigger Rule** – the rule which determines whether the lifecycle event workflow should be run; provided in SP_LCE_Rehire_TriggerAndWF.xml
- **SP LCE Rehire WF** – the lifecycle event workflow; provided in SP_LCE_Rehire_TriggerAndWF.xml
- **SP Rehire Rules Library** – the Rehire Feature's standard rules library, which contains the trigger logic and the build plan logic; provided in SP_Rehire_RulesLibrary.xml

These feature components should be modified and customized per installation:

- **SPCONF Rehire Mappings Custom** – the Custom object which defines the criteria for running the rehire event; provided in SPCONF_Rehire_Mappings_Custom.xml
- **SPCONF Rehire Rules Library** – the configurable Rules library where all installation-specific logic for the rehire event should be written; provided in SPCONF_Rehire_RulesLibrary.xml

## How it Works

### The Trigger

The trigger options for this workflow are:

- A custom rule which evaluates the identity using whatever logic the organization chooses to define
- An IdentitySelector which defines attribute values that designate a rehire for the organization

- A comparison of the “before” and “after” values of the user’s Links (accounts) to determine if the data values for specified account attributes have changed

Like all of the Feature triggers, this is a rule-based trigger.

1. The **SP LCE Rehire Trigger** calls the **SP LCE Rehire Trigger Rule** which, in turn, calls the **isTriggerRehireRule** method in the SP Rehire Rules Library.
2. The **isTriggerRehireRule** method queries the SPCONF Rehire Mappings Custom object for a **Trigger Type** entry which tells how to identify identities which are Rehires. The three valid values are
    a. **Selector** – the method compares the identity to the match expression in the **Trigger Field Selector** entry in the SPCONF Rehire Mappings Custom object and returns “true” to run the rehire workflow if the identity matches the criteria
    b. **Custom Rule** – the method passes the identity to the rule or rule library method named in **Trigger Custom Rule** in the SPCONF Rehire Mappings Custom object; the rule/method examines the identity and returns “true” if the identity is a Rehire; the isTriggeRehireRule method then also returns “true” and runs the rehire workflow for the identity (default rule to use is SPCONF Rule Library method isRehireCustomTriggerRule)
    c. **Compare Links** – the method passes the old and new versions of the identity to a rule library method (isNewLinkChanged) in the SP Attr Synch Rules Library (in the Attribute Synchronization feature). This method examines the user’s accounts on the applications listed in the SPCONF Rehire Mappings Custom object in the **Trigger Compare Links** attribute, and specifically the attributes listed in **Trigger Compare Links Schema**, and runs the rehire workflow if any of those attribute values have changed.

### The Workflow

1. The workflow first sets the **requestType**, which can be used by the Approval framework to drive the approval structure. By default, this gets set to “Rehire”.
2. A custom rule hook is available to perform installation-specific actions prior to provisioning plan creation. The rehireBeforePlanRule rule library method provides a template of actions which could be performed here.
3. Next, the workflow builds the provisioning plan based on the birthright provisioning configuration specified in the SPCONF Rehire Mappings Custom object’s **Plan Construction Types** attribute.
4. Otherwise, the **Before Provision** step offers a different custom rule hook to allow for any business-specific logic customizations (usually of the provisioning plan) before beginning the provisioning process.
5. Next, the workflow invokes the *Provision Processor* framework to perform the provisioning operation (including approval through the Approval framework).
6. The **After Provision** step then offers another custom rule hook for any business-specific logic which needs to be performed following the provisioning operation.
7. Lastly, the workflow invokes the *Dynamic Emails* Framework to send out any required emails which were added to the emailArgList throughout any other steps in this workflow or the Provision Processor framework workflow.

## How to Implement

The basic steps to use the framework are:

1. Configure the target properties to activate the event and determine whether emails will be sent out at the end of the provisioning process
2. Configure the trigger for the workflow
3. Select plan construction type
4. Configure assignment details
5. Configure rules to manage additional actions required by the business
6. Deploy the feature

### Step 1 – Configure Target Properties

The ssf.target.properties file provides a template for all the values required to manage the feature. Locate the Rehire Feature Properties section and copy everything in this section to the target.properties file for your environment (if you have not already run the SSD Deployer which does this for you). Set **SP_REHIRE_IS_DISABLED** to false to enable the feature, and set other attributes to true or false as described in the comments associated to each property in that section of the file.

```
#*******************************************************************************
#   REHIRE FEATURE PROPERTIES
#*******************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_REHIRE_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_REHIRE_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_REHIRE_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_REHIRE_SEND_POST_PROVISION_EMAILS%%=false
## ENTER true/false ON WHETHER TO RUN PROVISIONING IN THE FOREGROUND
%%SP_REHIRE_FOREGROUND_PROVISIONING%%=false
```

### Step 2 – Configure Trigger Attributes

The Identity Trigger allows for three different models. These are configured in the **SPCONF Rehire Mappings Custom** object, found in SPCONF_Rehire_Mappings_Custom.xml.

1. Set **Trigger Type** to specify how to decide whether the lifecycle event should be run for a given identity. Trigger Type value must be one of these:
    a. **Selector**: workflow trigger is specified in the Trigger Field Selector attribute
    **b. Compare Links:** workflow triggers are specified in Trigger Compare Links and Trigger Compare Links Schemas
    c. **Custom Rule:** workflow trigger is specified in Trigger Custom Rule attribute
2. If Trigger Type is **Selector,** you must configure the Trigger Field Selector entry to specify the identity or link attributes that should cause this workflow to run for the identity. For example:

```
<entry key="Trigger Field Selector">
      <value>
          <IdentitySelector>
            <MatchExpression and="true">
              <MatchTerm name="personStatus" value="A"/>
              <MatchTerm name="inactive" value="true"/>
```

```
            </MatchExpression>
          </IdentitySelector>
        </value>
</entry>
```

3. If Trigger Type is Compare Links, define the **Trigger Compare Links** and **Trigger Compare Links Schemas** attributes.
    a. **Trigger Compare Links** lists the applications for which accounts should be examined.
    b. **Trigger Compare Links Schemas** then lists the schema attributes on each of those applications which should be compared. It can be beneficial to **not** check for attributes that impact other lifecycle events, such as user status.
    c. The trigger logic retrieves the identity's application accounts on the listed application(s) and examines each attribute named in the Trigger Compare Links Schema for each application. If the identity's attribute value for any of those attributes has changed, the rehire workflow is launched.

    For example, this entry tells the trigger rule to evaluate the user's HR application account links.

```
<entry key="Trigger Compare Links">
   <value>
      <List>
         <String>HR</String>
      </List>
   </value>
</entry>
```

    Then this entry tells the trigger rule to compare the before and after values of the HR application's DEPARTMENT_ID and MGR_ID attributes.

```
<entry key="Trigger Compare Links Schemas">
   <value>
     <Attributes>
        <Map>
           <entry key="HR">
             <value>
                <List>
                  <String>DEPARTMENT_ID</String>
                  <String>MGR_ID</String>
                </List>
             </value>
           </entry>
        </Map>
     </Attributes>
   </value>
</entry>
```

4. If Trigger Type is **Custom Rule** define the **Trigger Custom Rule** attribute. This attribute should specify the name of a rule or a rule library method which will evaluate the identity and return "true" if the rehire event should be run for the identity.
    a. The SPCONF Rehire Mappings Custom object predefines this Trigger Custom Rule attribute to point to a rule library method in the SPCONF Rehire Rule Library, called **isRehireCustomTriggerRule**. The recommendation is to leave that entry as-is and to

specify your custom logic for identifying a rehire identity in that method in the SPCONF Rehire Rule Library.

```
public static boolean isRehireCustomTriggerRule(SailPointContext context, HashMap params) {
  rlogger.trace("Enter isRehireCustomTriggerRule");
  boolean flag = false;
  Identity previousIdentity = params.get("previousIdentity");
  Identity newIdentity = params.get("newIdentity");

  // Add logic here to define whether the workflow gets triggered

  rlogger.trace("Exit isRehireCustomTriggerRule");
  return flag;
}
```

### Step 3 – Select Plan Construction Types

The Plan Construction Types entry determines how the provisioning plan is constructed in the build plan step. This entry allows for multiple options to be specified individually or together, and the options are processed in the order below. Specify any or all of these options:

1. **Dynamic Roles** – This will dynamically assign and remove roles by using the Role Assignment Framework. In order for this to work, that framework must be in place and configured.
2. **Attribute Synch** – This will dynamically evaluate all target applications. If set, the logic will use the Application Skip Fields entry in the Attribute Synch Framework configuration to determine what fields to bypass on a per-account basis.
3. **Enable Accounts** – This will dynamically enable any existing accounts held by the identity on applications which are listed in the Enable Accounts Applications list.
4. **Custom Rule** – Enter the name of a rule or method. This is defaulted to method:buildCustomRehirePlan, which is already included in **SPCONF Rehire Rules Library**. This option allows you to write a custom rule or method. If you select this option, you must write a rule or method and enter its name in the Plan Construction Custom Rule entry.

### Step 4 – Configure Assignment Details

1. If Plan Construction Type includes **Dynamic Roles**, configure the Role Assignment Framework as described in that section of this document.
2. If Plan Construction Type includes **Attribute Synch**, configure the Attribute Synch Feature as described in that section of this document.
    - **NOTE**: This does not trigger the Attribute Synch Feature workflow separately; instead, this process uses the rule logic and configurations in that feature to build an attribute synchronization *part* of the provisioning plan and append it to the Dynamic Roles plan calculated already (if any).
3. If Plan Construction Type includes **Enable Accounts**, define the **Enable Accounts Applications** list in the SPCONF Rehire Mappings Custom Object to list all applications for which accounts should be enabled.
4. If the Plan Construction Type contains **Custom Rule**, enter a rule name in the **Plan Construction Custom Rule** entry of the SPCONF Rehire Mappings Custom object.
    - This entry is predefined in the SPCONF Rehire Mappings Custom object to point to a rule library method in the SPCONF Rehire Rule Library called buildCustomRehirePlan. The recommendation is to leave that entry as-is in the Custom object and write your desired business logic in that rule library method.

- This rule/method is passed a ProvisioningPlan object (in the variable “plan” which is recorded inside the “workflow” argument passed to the rule). It should append to that plan and return the full plan to be provisioned.

### Step 5 – Configure the Hooks

Most of the rehire workflow’s steps simply call methods in the **SPCONF Rehire Rule Library**, where you can write your own logic to further customize the behavior of the rehire workflow. As provided, the methods may be empty or may provide some template/default logic.

| Method | Calling Workflow Step | Default Logic Provided |
|---|---|---|
| getRehireRequestTypeRule | Get Request Type | Returns “Rehire” |
| rehireBeforePlanRule | Before Plan | Provides logic to retrieve identityModel and spExtAttr variables from the workflow and return them to the workflow; implementer should insert any logic to modify these variables as desired or perform any other before-plan-generation logic desired |
| rehireNoRequestRule | No Request (only if Build Plan step does not return provisioningPlan) | Sets an extended attribute called rehireAccountDate to today and the standard inactive attribute to false |
| rehireBeforeProvisionRule | Before Provision | None |
| rehireAfterProvisionRule | After Provision | Sets an extended identity attribute called rehireDate to the current date |
| buildCustomRehirePlan | Build Plan (if Plan Construction Type includes Custom Rule and Plan Construction Custom Rule is left pointing to this method) | Returns the existing provisioning plan from the previous steps or an empty ProvisioningPlan (if none created yet), though this is intended for the implementer to append additional request(s) to the existing plan |
| The remainder of the rule hooks in the SPCONF Rehire Rule Library are for configuring email message “to” addresses and templates to be used by the Provision Processor and Approval frameworks for queuing up information messages to be sent at the end of the workflow process. | | |
| getRehireProvSuccessEmailTemplateRule | Calculate values passed to Provision Processor framework for use in building the emailArgList when | Returns “Rehire Success Email Template” |
| getRehireProvFailureEmailTemplateRule | | Returns “Rehire Failure Email Template” |
| getRehireProvSuccessEmailToRule | | Returns the request target identity’s email address |

| | | |
|---|---|---|
| getRehireProvFailureEmailToRule | provisioning succeeds or fails | Returns the request target identity's email address |
| getRehireApprovedEmailTemplateRule | Calculate values passed to Approval framework (through Provision Processor framework) for use in building the emailArgList when provisioning request is approved or rejected | Returns "Rehire Approved Email Template" |
| getRehireRejectedEmailTemplateRule | | Returns "Rehire Rejected Email Template" |
| getRehireApprovedEmailToRule | | Returns the request target identity's email address |
| getRehireRejectedEmailToRule | | Returns a default (fictitious) email address – method content must be changed from default |

### Step 6 – Deploy the feature

You must include the customizable files in your build process by moving the files from the feature's Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer) along with the customizable files from dependent frameworks. The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects from the feature's /Source folder are automatically deployed with the SSB. Import the artifacts through the SSB deployment process (or other process).

# Terminate Identity

The Terminate Identity Feature is a QuickLink driven workflow, rather than a lifecycle event workflow, which allows authorized users to initiate a personnel termination process. It supports immediate termination or scheduled termination at a future date.

Though this feature is UI Feature, not a Lifecycle Event Feature, it resembles the Lifecycle Event Features in some important ways:

- It includes all of the components necessary to implement the full end-to-end solution for this requirement.
- It is composed of a workflow, a Custom object which contains a map of configuration options, a standard rule library which contains the basic workflow functionality, and a configurable rule library where implementers are expected to write the installation-specific logic.
- Its workflow includes rule hooks to customize the logic in the same places in the logic flow (before creating the plan, before provisioning, after provisioning, and when no provisioning plan is calculated).

## Components

These core feature components *should not* be modified or customized per installation:

- **SP LCM Terminate Identity Right** – the SPRight object representing the permission which must be granted to the user to make this feature available to them; provided in SP_LCM_TerminateIdentity_LinkAndWF.xml
- **SP LCM Terminate Identity Scope** – the Dynamic Scope (Quicklink Population) which grants access to the Terminate QuickLink to anyone who has the SP LCM Terminate Identity Right; provided in SP_LCM_TerminateIdentity_LinkAndWF.xml
- **SP Terminate Identity** – Quicklink which launches the termination workflow; provided in SP_LCM_TerminateIdentity_LinkAndWF.xml
- **SP LCM Terminate Identity WF** – the termination workflow; provided in SP_LCM_TerminateIdentity_LinkAndWF.xml
- **SP Terminate Identity Rules Library** – the Terminate Identity Feature's standard rules library, which contains the logic for presenting the user form in the workflow as well as the build plan logic; provided in SP_TerminateIdentity_RulesLibrary.xml

**NOTE**: The object structure for the QuickLink and DynamicScope comply with the pre 7.0 required structure, but IdentityIQ's import tool automatically converts this to the required structure in 7.x installations, adding the QuickLinkOptions object to manage the connection between the QuickLink and DynamicScope.

These feature components should be modified and customized per installation:

- **SPCONF Terminate Identity Mappings Custom** – the Custom object which specifies the forms to use in the workflow and the configuration for the termination provisioning process; provided in SPCONF_TerminateIdentity_Mappings_Custom.xml

- **SPCONF Terminate Rules Library** – the configurable Rules library where all installation-specific logic for the termination workflow should be written; provided in SPCONF_TerminateIdentity_RulesLibrary.xml
- **SP Terminate Identity Entry Form** – default form for termination request which allows the requester to specify a termination date when the action should be processed; can be customized, replaced, or not used at all, as needed; provided in SPCONF_TerminateIdentity_Forms.xml
- **SP Terminate Identity Confirm Form** – default read-only confirmation form which shows the user's name and email address, and the termination date; can be customized or replaced; provided in SPCONF_TerminateIdentity_Forms.xml

## How It Works

### The QuickLink

Instead of being driven by a lifecycle event trigger, this process is started by a user clicking a QuickLink in the IdentityIQ user interface. Access to the QuickLink is controlled by the **SP LCM Terminate Identity Scope** DynamicScope (called a Quicklink Population in the 7.0+ UI), which allows anyone who has been assigned the **SP LCM Terminate Identity Right** SPRight to see and use the QuickLink.

The QuickLink is a "request for others" quicklink, so the requester is first prompted to choose a target user for the request. Then it launches the **SP LCM Terminate Identity WF** workflow to process the termination request.

### The Workflow

1. The workflow first creates and populates an **IdentityModel** map for easier management of the form interactions in the early steps of the workflow process. By default, only name, displayName, email, and termination date are set into that model, but any attributes which you want displayed on the form(s) should be added to the model as well.
2. Next, it checks to see if a data entry form should be presented to allow the requester to specify a termination date or provide any other input data for the request. This step also calculates the forms to use for the data entry form and the confirmation form.
3. If data entry is configured, the entry form gets displayed and the requester provides any needed information.
4. The confirmation form, which is a read-only form, displays information about the target user (and may re-display data entered on the data entry form if applicable) so the requester can verify and confirm the termination request.
5. Next, the workflow sets the **requestType**, which can be used by the Approval framework to drive the approval structure. By default, this gets set to "Terminate Identity".
6. A custom rule hook is available to perform installation-specific actions prior to provisioning plan creation. The termBeforePlanRule rule library method provides a template of actions which could be performed here.
7. Next, the workflow builds the provisioning plan based on the terminate provisioning configuration specified in the SPCONF Terminate Identity Mappings Custom object's **Plan Construction Types** attribute.

8. If the plan is empty, the **No Request** step offers a custom rule hook to perform any non-provisioning logic required by the installation, in lieu of provisioning.
9. Otherwise, the **Before Provision** step offers a different custom rule hook to allow for any business-specific logic customizations (usually of the provisioning plan) before beginning the provisioning process.
10. Next, the workflow invokes the Provision Processor framework to perform the provisioning operation (including approval through the Approval framework).
11. The **After Provision** step then offers another custom rule hook for any business-specific logic which needs to be performed following the provisioning operation.
12. Lastly, the workflow invokes the Dynamic Emails Framework to send out any required emails which were added to the emailArgList throughout any other steps in this workflow or the Provision Processor framework workflow.

## How to Implement

The basic steps to use the framework are:

1. Configure the target properties to activate the event and determine whether emails will be sent out at the end of the provisioning process
2. Configure the QuickLink permissions for the workflow
3. Configure the Form details
4. Select the provisioning plan construction type
5. Configure the assignment details
6. Configure rules to manage additional actions required by the business
7. Deploy the feature

### Step 1 – Configure Target Properties

The ssf.target.properties file provides a template for all the values required to manage the feature. Locate the Terminate Identity Feature Properties section and copy everything in this section to the target.properties file for your environment (if you have not already run the SSD Deployer which does this for you). Set SP_TERMINATE_IDENTITY_IS_DISABLED to false to enable the feature, and set other attributes to true or false as described in the comments associated to each property in that section of the file.

```
#*******************************************************************************
#   TERMINATE IDENTITY FEATURE PROPERTIES
#*******************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_TERMINATE_IDENTITY_IS_DISABLED%%=true
## SPECIFY THE CATEGORY
%%SP_TERMINATE_IDENTITY_LINK_CATEGORY%%=Admin
## SPECIFY HOW THE KEY FOR HOW THE LINK WILL BE DISPLAYED
%%SP_TERMINATE_IDENTITY_LINK_MESSAGE_KEY%%=Terminate Identity
## SPECIFY WHAT ORDER THE LINK WILL SHOW UP
%%SP_TERMINATE_IDENTITY_LINK_ORDERING%%=3
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_TERMINATE_IDENTITY_WF_TRACE_ENABLED%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
%%SP_TERMINATE_IDENTITY_SEND_APPROVED_EMAILS%%=false
## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
%%SP_TERMINATE_IDENTITY_SEND_POST_PROVISION_EMAILS%%=false
## ENTER true/false ON WHETHER TO RUN PROVISIONING IN THE FOREGROUND
%%SP_TERMINATE_IDENTITY_FOREGROUND_PROVISIONING%%=false
```

**Terminate Identity Properties in ssf.target.properties**

**NOTE**: The Category, Message Key, and Ordering properties all affect how the QuickLink appears in the user interface.

### Step 2 – Configure Quicklink Permissions

This QuickLink is connected to a custom SPRight, and access to it requires the user to have that right in IdentityIQ. Custom rights can be added to one or more Capabilities to grant the right to any user who has been assigned that Capability.

To add the SPRight to an existing Capability, add this Reference under the <RightRefs> tag in the XML of the Capability object:

```
<Reference class="sailpoint.object.SPRight" name="SP_LCM_TerminateIdentity_SPRight"/>
```

Alternatively you can create a new Capability specifically to grant access to the QuickLink and assign that Capability to the appropriate users. An example custom capability definition is shown below:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Capability PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Capability displayName="Identity Terminator" name="IdentityTerminator">
  <Description>Allows use of the Terminate Identity QuickLink</Description>
  <RightRefs>
    <Reference class="sailpoint.object.SPRight" name="SP_LCM_TerminateIdentity_SPRight"/>
  </RightRefs>
</Capability>
```

### Step 3 – Configure Form Details

Three entries in the **SPCONF Terminate Identity Mappings Custom** object determine what forms are shown to the requester when they launch the termination workflow.

- To allow the requester to provide input information into the process, set the **Use Custom Entry Form** attribute to “true” and specify the desired form name in the **Entry Form Name** attribute. The default form allows the user to specify a termination date to make the termination action happen at a future date instead of immediately.

- Specify the name of the desired read-only confirmation form in the **Confirm Form Name** entry. This form displays identifying information about the target user so the requester can confirm they have chosen the right person to terminate.

**NOTE**: The default forms, which are provided with the framework, are already named in the Entry Form Name and Confirm Form Name attributes. Typically, those forms are customized per installation and the form names are left as-is in the Mappings Custom object. The forms can be customized to add other identifying information about the target user or to add other attributes which the requester can provide in the termination process.

### Step 4 – Specify Provisioning Plan Construction Type

Set the **Plan Construction Types** entry in the SPCONF Terminate Identity Mappings Custom object to determine how the provisioning plan is constructed in the Build Plan step of the workflow. This attribute, which is specified as a list, can be any or all of these three values:

1. **Disable Accounts**: The workflow will disable all of the user's accounts on applications listed in the Disable Accounts Applications list.
2. **Delete Accounts**: The workflow will delete all of the user's accounts on applications listed in the Delete Accounts Applications list**.**
3. **Custom Rule**: The workflow uses the rule specified in the **Plan Construction Custom Rule** entry of the SPCONF Terminate Identity Mappings Custom object to build or append to the provisioning plan to provision the desired access.

**NOTE**: Applications which appear on both the Disable Accounts and Delete Accounts lists will be deleted if both operations are requested.

### Step 4 – Configure the Assignment Details

1. If the plan construction type includes **Disable Accounts**, then populate the **Disable Accounts Applications** list in the SPCONF Terminate Identity Mappings Custom object with the list of applications whose accounts should be disabled.
2. If the plan construction type includes **Delete Accounts**, then populate the **Delete Accounts Applications** list in the SPCONF Terminate Identity Mappings Custom object with the list of applications whose accounts should be deleted.
3. If the assignment type is **Custom Rule**, enter a rule name in the **Plan Construction Custom Rule** entry of the SPCONF Terminate Identity Mappings Custom object.
    - This entry is predefined in the SPCONF Terminate Identity Mappings Custom object to point to a rule library method in the SPCONF Terminate Identity Rule Library called **buildCustomTerminateIdentityPlan**. The recommendation is to leave that entry as-is in the Custom object and write your desired business logic in that rule library method.
    - This option runs last, so the rule should append to any plan created for disabling or deleting accounts through the Disable Accounts and Delete Accounts configuration.

```
<!--  Enter the Plan Construction Types.  More than one can be selected.
    The order processed is always:  Dynamic Roles, Attribute Synch, Custom Rule.
        Options Include:
            - Disable Accounts:  will dynamically disable accounts tied to the cube, based on the list of
                applications in the Disable Accounts Applications entry
            - Delete Accounts:  will dynamically delete accounts tied to the cube, based on the list of
                applications in the Delete Accounts Applications entry
            - Custom Rule:  Will call out to the rule defined in the
                Plan Construction Custom Rule entry below.  Rule will receive an Identity object.
                Rule must return a ProvisioningPlan.
-->
<entry key="Plan Construction Types" >
    <value>
        <List>
            <String>Disable Accounts</String>
            <String>Delete Accounts</String>
            <String>Custom Rule</String>
        </List>
    </value>
</entry>

<!-- Used if Plan Construction Type is Custom Rule -->
<entry key="Plan Construction Custom Rule" value="method:buildCustomTerminateIdentityPlan" />

<!-- Used if Plan Construction Types contains Disable Accounts.  List the
    application names that should be disabled -->
<entry key="Disable Accounts Applications">
    <value>
        <List>
            <String>Active Directory - Main</String>
        </List>
    </value>
</entry>

<!-- Used if Plan Construction Types contains Delete Accounts.  List the
    application names that should be deleted -->
<entry key="Delete Accounts Applications">
    <value>
        <List>
            <String>Active Directory - Main</String>
        </List>
    </value>
</entry>

<!-- NOTE:  IF AN APP IS ADDED TO BOTH THE DISABLE AND DELETE LISTS, DELETES WILL
    TAKE PRIORITY AS THEY ARE MORE STRINGENT -->
```

**Assignment Configuration in Custom Mapping Object**

## Step 5 – Configure the Hooks

Most of the workflow's steps simply call methods in the SPCONF Terminate Identity Rule Library, where you can write your own logic to further customize the behavior of the joiner workflow. As provided, the methods may be empty or may provide some template/default logic.

| Method | Calling Workflow Step | Default Logic Provided |
|---|---|---|
| termInitIdentityModel | Init Identity Model | Populates an identityModel Attributes map with name, display name, email, and terminationDate |
| getTermRequestTypeRule | Get Request Type | Returns "Terminate Identity" |
| termBeforePlanRule | Before Plan | Provides logic to retrieve identityModel and spExtAttr variables from the workflow and return them to the workflow; implementer should insert any |

| | | |
|---|---|---|
| | | logic to modify these variables as desired or perform any other before-plan-generation logic desired |
| termNoRequestRule | No Request (only if Build Plan step does not return provisioningPlan) | None |
| termBeforeProvisionRule | Before Provision | None |
| termAfterProvisionRule | After Provision | Marks the identity as inactive |
| buildCustomTerminateIdentityPlan | Build Plan (if Plan Construction Type = Custom Rule and Plan Construction Custom Rule is left pointing to this method) | Either retrieves the plan from the workflow and returns that plan or returns an empty ProvisioningPlan, but the intent is for the implementer to customize this to insert other plan-creation logic |
| The remainder of the rule hooks in the SPCONF Terminate Identity Rule Library are for configuring email message “to” addresses and templates to be used by the Provision Processor and Approval frameworks for queuing up information messages to be sent at the end of the workflow process. | | |
| getTermProvSuccessEmailTemplateRule | Calculate values passed to Provision Processor framework for use in building the emailArgList when provisioning succeeds or fails | Returns “Terminate Identity Success Email Template” |
| getTermProvFailureEmailTemplateRule | | Returns “Terminate Identity Failure Email Template” |
| getTermProvSuccessEmailToRule | | Returns the request target identity’s email address |
| getTermProvFailureEmailToRule | | Returns the request target identity’s email address |
| getTermApprovedEmailTemplateRule | Calculate values passed to Approval framework (through Provision Processor framework) for use in building the emailArgList when provisioning request is approved or rejected | Returns “Terminate Identity Approved Email Template” |
| getTermRejectedEmailTemplateRule | | Returns “Terminate Identity Rejected Email Template” |
| getTermApprovedEmailToRule | | Returns the request target identity’s email address |
| getTermRejectedEmailToRule | | Returns a default (fictitious) email address – method content must be changed from default |

### Step 6 – Deploy the feature

You must include the customizable files in your build process by moving the files from the feature’s Configure folder into the /config folder in the SSB folder structure (if you have not already done this with the SSD Deployer) along with the customizable files from dependent frameworks. The /config folder is usually organized into subfolders by object type, so put files in the appropriate subfolders. The objects

from the feature's /Source folder are automatically deployed with the SSB.  Import the artifacts through the SSB deployment process (or other process).

# Troubleshooting

Troubleshooting the SSF Features can be done by turning on logging through Log4j entries and by turning on workflow tracing for the various workflows.

## Logging

All SSF rules and workflows are configured for log4j. Specifically, the rules libraries all have log4j loggers in them, and turning on logging for each of those loggers prints log statements which are included in methods throughout the libraries.

This is a list of log4j statements that can be added to /WEB-INF/classes/log4j.properties to turn on debug level logging for each of the features and frameworks:

```
#Custom SSF Loggers
log4j.logger.rule.SP.FieldValue.RulesLibrary=debug
log4j.logger.rule.SP.ApprovalFramework.RulesLibrary=debug
log4j.logger.rule.SP.RoleAssignment.RulesLibrary=debug
log4j.logger.rule.SP_Provisioning.RulesLibrary=debug
log4j.logger.rule.SP.Joiner.RulesLibrary=debug
log4j.logger.rule.SP.Leaver.RulesLibrary=debug
log4j.logger.rule.SP.AttrSynch.RulesLibrary=debug
log4j.logger.rule.SP.Mover.RulesLibrary=debug
log4j.logger.rule.SP.Rehire.RulesLibrary=debug
log4j.logger.rule.SP.TerminateIdentity.RulesLibrary=debug
```

There are debug statements and trace statements in the libraries, so turning them on at the trace level can also be helpful.

## Workflow Tracing

In addition, any of the following target properties can be set to true to turn on workflow trace for the corresponding workflow:

```
%%SP_JOINER_WF_TRACE_ENABLED%%
%%SP_LEAVER_WF_TRACE_ENABLED%%
%%SP_ATTR_SYNCH_WF_TRACE_ENABLED%%
%%SP_MOVER_WF_TRACE_ENABLED%%
%%SP_REHIRE_WF_TRACE_ENABLED%%
%%SP_TERMINATE_IDENTITY_WF_TRACE_ENABLED%%
```

# Appendix – Example Custom Rule Library

The following is an example of the methods in a custom rule library. Things to note in this example:

- The **getRequestTypeRule** method calculates and returns request type based on an identity attribute, “userType” so the workflow can implement different details (including approval requirements) for joiners who are employees vs. contractors vs. vendors.
- The **beforePlanRule** method updates some workflow attributes, namely identityModel, which will be passed up and down the chain of workflows and can be accessed in the approval framework, either to make decisions about the approval or to display on a custom approval form
- The **afterProvisionRule** method has logic to get any provisioning errors. It can make decisions based on this. Currently, it illustrates how to retrieve the errors from the workflow, but it does not act upon them. It instead calls logic to set the identity’s joinerDate to today.

```
private static Log jlogger = LogFactory.getLog("rule.SP.Joiner.RulesLibrary");

/* Return the request Type */
public static String getJoinerRequestTypeRule(SailPointContext context, Workflow workflow){
        String requestType = "Joiner";
        String identityName = workflow.get("identityName");
        Identity identity = context.getObjectByName(Identity.class, identityName);

        if (identity == null){
                jlogger.warn("No identity found for: " + identityName);
                return null;
        }

        String userType = identity.getAttribute("userType");

        if (userType == null){
                jlogger.warn("No user type found for: " + identityName);
                return null;
        }

        if (userType.equalsIgnoreCase("Contractor")){
                requestType = "Contractor Joiner";
        } else if (userType.equalsIgnoreCase("Vendor")){
                requestType = "Vendor Joiner";
        } else {
                requestType = "Employee Joiner";
        }

        context.decache(identity);
        identity = null;

        jlogger.trace("Exit get request type: " + requestType);
        return requestType;
}

/* Do any updates to workflow variables before ProvisioningPlan is compiled */
public static void joinerBeforePlanRule(SailPointContext context, Workflow workflow){
        jlogger.trace("Enter Joiner before plan rule");

        jlogger.trace("Getting extension attributes that can be used for customizations");
```

```
		Attributes identityModel = initWorkflowAttributesVar(workflow, "identityModel");
		Attributes spExtAttrs = initWorkflowAttributesVar(workflow, "spExtAttrs");

		//TODO:  CAN PUT LOGIC HERE TO INITIALIZE VALUES
		identityModel.put("testField", "");
		identityModel.put("testField2", "This should show up on approval form");

		jlogger.debug("Have identityModel: " + identityModel);

		workflow.put("identityModel", identityModel);
		workflow.put("spExtAttrs", spExtAttrs);

		context.decache(identity);
		identity = null;

		jlogger.trace("Exit Joiner before plan rule");
}

/* Do any updates to workflow variables before ProvisioningProject is provisioned */
public static void joinerBeforeProvisionRule(SailPointContext context, Workflow workflow){
		jlogger.trace("Enter Joiner beforeProvisionRule");
		jlogger.debug("In Joiner beforeProvisionRule, get identityModel.");
		Attributes identityModel = initWorkflowAttributesVar(workflow, "identityModel");
		jlogger.debug("Have identityModel: " + identityModel);


		jlogger.trace("Exit Joiner beforeProvisionRule");
}

/* Do any updates to workflow variables after ProvisioningProject is provisioned */
public static void joinerAfterProvisionRule(SailPointContext context, Workflow workflow){
		//TODO: ANALYZE PROJECT RESULTS, RESET JOINER DATE IF NECESSARY, I.E. IF FAILED.
		jlogger.trace("Enter Joiner afterProvisionRule");
		List errors = getErrors(context, workflow);

		String identityName = workflow.get("identityName");
		SimpleDateFormat format = new SimpleDateFormat("yyyyMMdd");
		Date now = new Date();
		String joinerDate = format.format(now);

		if (identityName != null){
			setIdentityAttribute(context, identityName, "joinerDate", joinerDate);
		}

		jlogger.trace("Exit Joiner afterProvisionRule");
}

/* Do any updates when a joiner occurs but a request is not required */
public void joinerNoRequestRule(SailPointContext context, Workflow workflow){
		jlogger.trace("Enter Joiner joinerNoRequest");

		/*jlogger.trace("Update joiner account date");
		SimpleDateFormat format = new SimpleDateFormat("yyyyMMdd");
		Date now = new Date();
		String currentDate = format.format(now);
		Identity li = ObjectUtil.lockIdentityByName(context, identityName);
```

```
            li.setAttribute("joinerAccountDate", currentDate);
            li.setInactive(false);
            ObjectUtil.unlockIdentity(context, li);
            context.commitTransaction();
            context.decache(li);
            li = null;*/

            jlogger.trace("Exit Joiner joinerNoRequest");
}

/* Return prov success email */
public static String getJoinerProvSuccessEmailTemplateRule(SailPointContext context, Workflow workflow){
            String val = "Joiner Success Email Template";

            return val;
}

/* Return prov failure email */
public static String getJoinerProvFailureEmailTemplateRule(SailPointContext context, Workflow workflow){
            String val = "Joiner Failure Email Template";

            return val;
}

/* Return approved email */
public static String getJoinerApprovedEmailTemplateRule(SailPointContext context, Workflow workflow){
            String val = "Joiner Approved Email Template";

            return val;
}

/* Return rejected email */
public static String getJoinerRejectedEmailTemplateRule(SailPointContext context, Workflow workflow){
            String val = "Joiner Rejected Email Template";

            return val;
}

/* Return prov success email to */
public static String getJoinerProvSuccessEmailToRule(SailPointContext context, Workflow workflow){
            Identity identity = context.getObject(Identity.class, workflow.get("identityName"));
            String val = identity.getEmail();

            context.decache(identity);
            identity = null;

            return val;
}

/* Return prov failure email to */
public static String getJoinerProvFailureEmailToRule(SailPointContext context, Workflow workflow){
            Identity identity = context.getObject(Identity.class, workflow.get("identityName"));
            String val = identity.getEmail();

            context.decache(identity);
            identity = null;
```

```
        return val;
}

/* Return approved email to */
public static String getJoinerApprovedEmailToRule(SailPointContext context, Workflow workflow){
        Identity identity = context.getObject(Identity.class, workflow.get("identityName"));
        String val = identity.getEmail();

        context.decache(identity);
        identity = null;

        return val;
}

/* Return rejected email to */
public static String getJoinerRejectedEmailToRule(SailPointContext context, Workflow workflow){
        String val = "somedefaultaddress@sailpoint.com";

        return val;
}
```