SailPoint

Services Standard Framework User Guide

6034 W. Courtyard Drive, Suite 309
Austin, Texas 78730
Phone 512.346.2000
Fax 512.346.2033
www.sailpoint.com

Table of Contents

Overview

The Services Standard Framework (SSF) is a package of reusable IdentityIQ configurations and code used to speed up and simplify the deployment process on IdentityIQ implementations. It does this by abstracting and reusing common architectures and best practices, eliminating “nuts and bolts” development and allowing implementers to focus on the customer requirements.

The SSF packages are easy to deploy and extend by effectively distinguishing between source code (the nuts and bolts) and configurable extensions. Each SSF package is configured via target properties, a custom mapping object, and a rule library. For most use cases, workflow development and low-level, API-centric development is eliminated.

Frameworks are plug and play and reusable across multiple use cases. Some, such as the Field Values Framework, are simple and simply 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 frameworks currently available 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 IIQ account requests to assign roles based on the Identity Selector of business roles.
- **Dynamic Emails** – A single sub process that can be called from any workflow that simplifying and queuing the sending of emails.
- **Approvals** – A single sub process and set of configuration options that dynamically and iteratively processes approval types required for a given use case.
- **Provision Processor** – A set of subs 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 Sub, Send Emails.

Features, on the other hand, are complete, configurable, end-to-end use cases. These, like the larger frameworks, come with a suite of configuration options. Features take common end-to-end use cases, or workflows, such as the Joiner Workflow, and bake into the underlying framework the steps and actions that are common across all business requirements while carving out “hooks” for specific customization and layering on top a configuration layer to quickly setup many of the most common scenarios. The features currently available are:

LCE Features – All LCE Features are end-to-end solutions comprising the lifecycle event (Identity Trigger), the trigger rule, and the workflow. They sit on top of a standard rule library and are configured via a handful of properties, a configuration mapping object, and a single rule library. They include: options to determine how to trigger and how to build the provisioning plan, hooks to configure email options and the required approval scenarios, and three strategic hooks (before build plan, before provision, after provision) for additional workflow requirements. The LCE Features available are:

- **Joiner** – Used for onboarding.
- **Leaver** – Used for terminations and/or a leave of absence.
- **Attribute Synch** – Used for pushing target attribute changes.
- **Mover** – Used for handling transfers.
- **Rehire** – Used for terminated users who have returned.

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), and performance tools (SSP). The SSD aims to standardize and simplify implementations and can be setup with a few simple steps.

The quick steps to get started are:

1. Create source control repository
2. Checkout the project (usually in Eclipse)
3. Copy the unzipped SSD into the trunk folder of your project
4. Setup the SSB build, servers, <host-mapping>.{iiq/target/ignorefiles}.properties
5. Configure the SSF by moving the various “Configure” files out
6. Configure your business requirements
7. Run the ant build
8. Deploy your code
9. Repeat steps 6-8

Each Framework or Feature contains:

- **Configure folder** – Objects that are required to be configured.
- **Source folder** – That source code that should never be touched.
- **Samples folder** – How to use or call the framework.
- **Target Properties file (ssf.target.properties)** – Specifies object name of the objects in the Configure folder and other options.
- **Ignore Files Properties file (ssf.ignorefiles.properties) –** Explicitly ignores all files located under any Configure folder as a means of forcing you to copy these files out into your project structure.
- **README.txt file** – Quick instructions on how to use and configure.

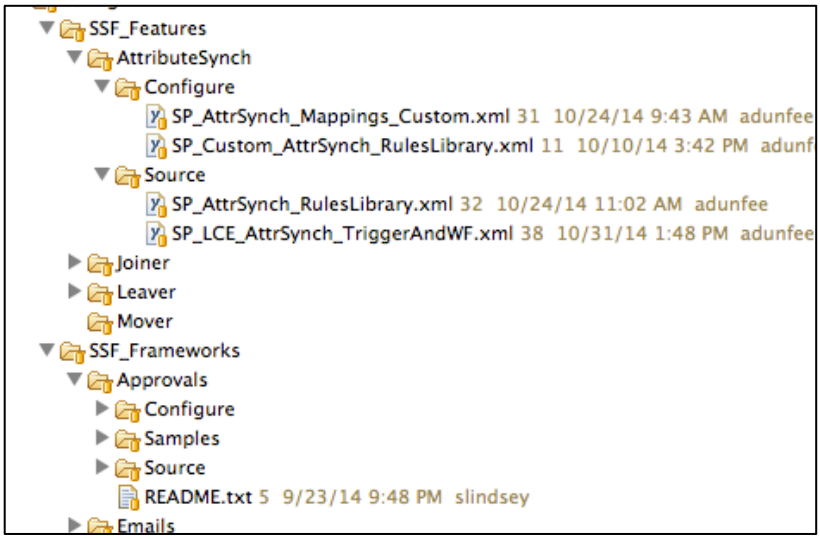

Figure 1 - SSF Folder Structure

```
#*********************************************************************************
#   JOINER FEATURE PROPERTIES
#*********************************************************************************
## SPECIFY WHETHER DISABLED
%%SP_JOINER_IS_DISABLED%%=true
## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
%%SP_JOINER_WF_TRACE_ENABLED%%=false
## ENTER THE NAME OF THE JOINER CUSTOM MAPPING OBJECT NAME
%%SP_JOINER_CUSTOM_OBJECT_NAME%%=SP Joiner Mappings Custom
## ENTER THE NAME OF THE CUSTOM JOINER RULES LIBRARY OBJECT NAME
%%SP_JOINER_RULES_OBJECT_NAME%%=SP Custom Joiner Rules Library
## 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
```

Figure 2 - Target Properties

```
#ADD THESE TO IGNORE THE OOTB CUSTOM OBJECTS
custom/SSF_Frameworks/Approvals/Configure/SP_CST_ApprovalFramework_Custom_RuleLibrary.xml
custom/SSF_Frameworks/Approvals/Configure/SP_CST_ApprovalObjectMappings_Custom.xml
custom/SSF_Frameworks/FieldValue/Configure/SP_CUSTOM_FieldValue_RulesLibrary.xml
custom/SSF_Frameworks/ProvisionProcessor/Configure/SP_EmailTest_Custom.xml
custom/SSF_Features/Joiner/Configure/SP_Custom_Joiner_RulesLibrary.xml
custom/SSF_Features/Joiner/Configure/SP_Joiner_Mappings_Custom.xml
custom/SSF_Features/Leaver/Configure/SP_Custom_Leaver_RulesLibrary.xml
custom/SSF_Features/Leaver/Configure/SP_Leaver_Mappings_Custom.xml
custom/SSF_Features/AttributeSynch/Configure/SP_Custom_AttrSynch_RulesLibrary.xml
custom/SSF_Features/AttributeSynch/Configure/SP_AttrSynch_Mappings_Custom.xml
```

Figure 3 - Ignore Files Properties

The basic configuration instructions for each framework or feature are:

1. The following steps assume that the SSB is configured and setup and that IdentityIQ has been installed and initialized.
2. Copy all files under the Configure folder to a more appropriate location, such as a Rule Library object to the /config/Rule folder or a Custom object to the /config/Custom folder.
3. Rename each file to a customer appropriate name, such as SP_CUSTOM_FieldValue_Rules_Library.xml to cstCustom_FV_RulesLibrary.xml.
4. Update the target properties to give a customer-specific name, to enable (if a feature), or to set other options.
5. Configure the framework or feature per its instructions and your business requirements.
6. Run the standard ant build process.

Note: The object name for all custom objects will be tokenized, such as %%SP_CUSTOM_FV_RULE_LIBRARY_NAME%%. Do not change the name of the object. The tokenization ensures referential integrity.

Frameworks

The following details each framework.

Field Value

The Field Value framework standardizes and simplifies the construction of provisioning policies by decoupling the logic to set each field from the actual policy. Each field is configured to call one rule, SP Dynamic Field Value Rule. The logic for each field is encapsulated in a specific method (denoted by a standard naming scheme) in a single, configurable rule library. This cleans up tedious, attribute logic; offers better error handling; and reduces the number of field value rules to just the one library.

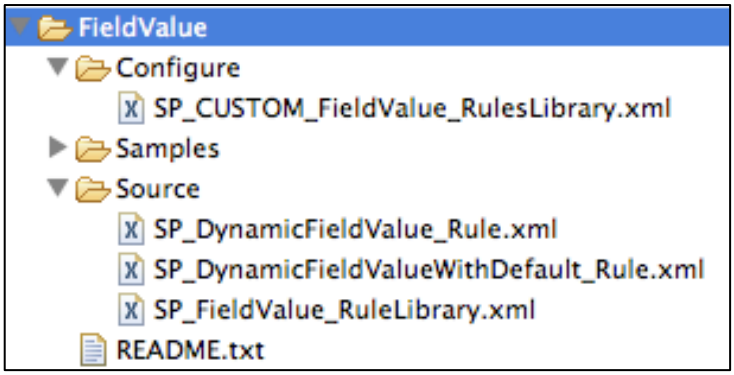

Figure 4 - Field Value Folder Structure

Intended Use

This framework is intended for all provisioning policy fields that return values from the identity cube or derive their value via beanshell logic. It is not intended for dynamic fields that require user input or fields that reference another field in the provisioning policy. However, a hybrid approach is possible whereby some fields can use scripts and other fields can use the single, dynamic field value rule.

Advantages

- **Fewer java imports** – The single library cuts down on the need to declare the various import statements for each script or field value rule.
- **Improved logging** – Single log4j declaration increases possibility that logging will be done correctly with proper trace levels versus temporary System.out.println() statements.
- **Better collaboration** – Can tokenize versions of the library per environment, allowing one developer to create a stub version with simple return values and another developer to work on the real logic, for example.
- **Fewer rule objects** – Replaces all of the field value rules with one library.
- **Decouple policy from logic** – Keeps the application definition cleaner and 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.

How To

The basic steps to use the framework are:

1. Update a field with the proper attributes and rule reference
2. Write the method for the given application/field combination

Step 1 – Update the field's attributes and add the rule reference

For each provisioning policy field, add:

- **An application attribute** – Used to construct the dynamic method call. Doesn't need to be the name of the actual application. For fields that have common logic, the application could be "Common" or "General".
- **A template attribute** – Passed into the method as the String op. Can be used in the method's logic.
- **The rule reference** – Must be SP Dynamic Field Value Rule

For example:

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

Step 2 – Write the method

In the copy of the file, SP_CUSTOM_FieldValue_RulesLibrary.xml, a method must be constructed for each field that calls the rule, SP Dynamic Field Value Rule. The method must use the following standard naming scheme:

getFV_<Application_Name>_<Field_Name>_Rule

The application name is based on the Field attribute *application* and the field name is based on the Field *name* attribute. Both are case sensitive and all dashes, periods, and spaces must be replaced by underscores. For the example above, the method name would be:

getFV_Active_Directory_sAMAccountName_Rule

The method signature requires that it is static, returns an Object, and accepts a SailPointContext, Identity, and String. The following is an example method:

```
public static Object getFV_Active_Directory_sAMAccountName_Rule(
        SailPointContext context,
        Identity identity,
        String op
        ){

        logger.trace("Enter Active Directory sAMAccountName rule");

        String val = identity.getName();

        logger.trace("Exit Active Directory sAMAccountName rule: " + val);
        return val;
}
```

Dynamic Emails

The Dynamic Emails framework is a single sub process that can be called from any workflow that accepts a list (emailArgList) of maps that is used to send out any number of emails. Each map represents an email that needs to be fired. The map has two required entries: to & template. Other attributes can be added and then referenced in the email template as simply $emailArgs.nameOfKey. This simplifies workflows by eliminating the need to call a send email step for every required email, eliminating the need to map each required attribute to the given step, and allowing various workflow and rule logic to add emails (or maps) to the list and then, when ready, to fire them all at once.

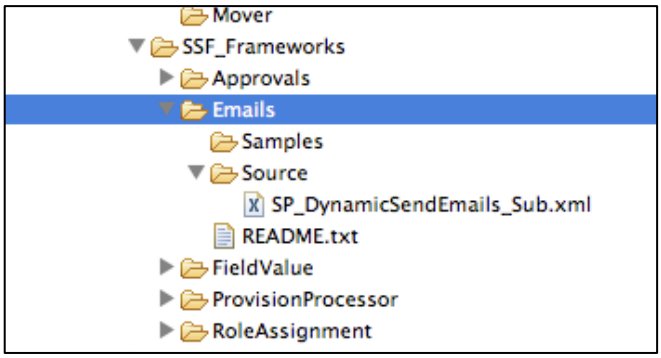

Figure 5 - Dynamic Emails Folder Structure

Intended Use

This framework is intended for any workflow that needs to send out multiple emails, where the contents and required attributes differ from email to email, where the attributes aren't necessarily available to the top workflow that is sending the emails, or any case where it would be unwieldy to add a new step for each required email. The framework is not intended for the approval, reminder or escalation emails sent out on a per-approval step.

Advantages

- **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.

How To

The basic steps to use the framework are:

1. Build and append to the emailArgList
2. Call the sub process
3. Write email templates

There are no properties required for this framework.

Step 1 – Build and append to the emailArgList

In any workflow, the emailArgList can be built as an ArrayList, such as:

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

For each email, add a map object to the emailArgList. The map must contain to and emailTemplate. 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);
```

Step 2 – Call the sub process
In any workflow, simply call the sub process, SP Dynamic Send Emails Sub, and pass in the variable, emailArgList.

The following is an example:

```
<Step name="Send Emails">
        <Arg name="emailArgList" value="ref:emailArgList"/>
    <Description>
      Call the standard subprocess that will handle the built-in
      owner, manager and security officer approval schemes.
    </Description>
    <WorkflowRef>
      <Reference class="sailpoint.object.Workflow" name="SP Dynamic Send Emails Sub"/>
    </WorkflowRef>
    <Transition to="Stop"/>
</Step>
```

Step 3 – Write the email templates
Write each referenced template. Any value put into the map can be referenced as $emailArgs.key. For example, the key "someOtherVal" from above can be added to the template's body or subject as $emailArgs.someOtherVal and when the email is sent, the value presented with be "this is a test".

Role Assignment

The Role Assignment framework is a set of methods that are used to dynamically build IIQ account requests to assign roles based on the Identity Selector of business roles. This is effective in cases, such as the Joiner or Mover, where the implementer needs to do role provisioning in a workflow. The logic for role assignment lives in the roles. The logic to build an account request is a single line that can be added to a workflow or a rule.

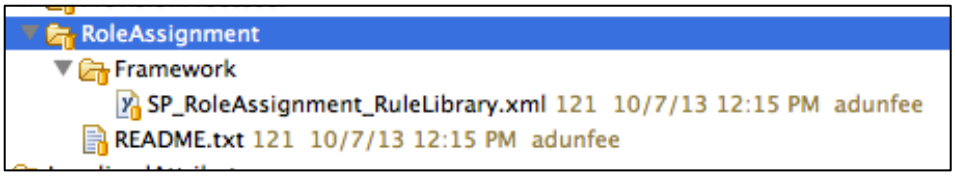

Figure 6 - Role Assignment Folder Structure

Intended Use

There are many cases where birthright provisioning needs to be done in a LCE workflow that can handle approvals; notifications, including emails with new credentials; and other various workflow steps. It can be beneficial to assign birthright access via roles versus building provisioning plans, with individual account requests for each birthright application based on various rules or workflow logic, from scratch. This framework allows an implementer to setup birthright roles with assignment logic and then build account requests and plans with a single method call.

Advantages

- **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.

How To

The basic steps to use the framework are:

1. Create an organizational role to act as a container for the birthright roles
2. Create the business roles
3. Add the rule reference for the SP Role Assignment Rule Library
4. Call the method to get the account request or account requests

After following the general deployment instructions, validate the target properties. For example:

```
#*******************************************************************************
#       ROLE ASSIGNMENT FRAMEWORK PROPERTIES
#*******************************************************************************
## ENTER THE NAME OF THE ORGANIZATIONAL ROLE CONTAINING BIRTHRIGHT BUSINESS ROLES
%%SP_BIRTHRIGHT_ROLES_ORGANIZATION_ROLE%%=ACME Birthright Roles Container
## 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%%=ACME Default Birthright Role
```

Step 1 – Create Organizational Role

The framework uses the organizational role to look for assignable roles. This is done to prevent scanning every single role in the system. There are no rules on what the name needs to be, but the name must be correctly represented in the target properties file.

Figure 7 - Role Assignment Organizational Role

Step 2 – Create Business Roles

For each role, modify the inheritance and select the organizational role created in Step 1. Add proper assignment logic.

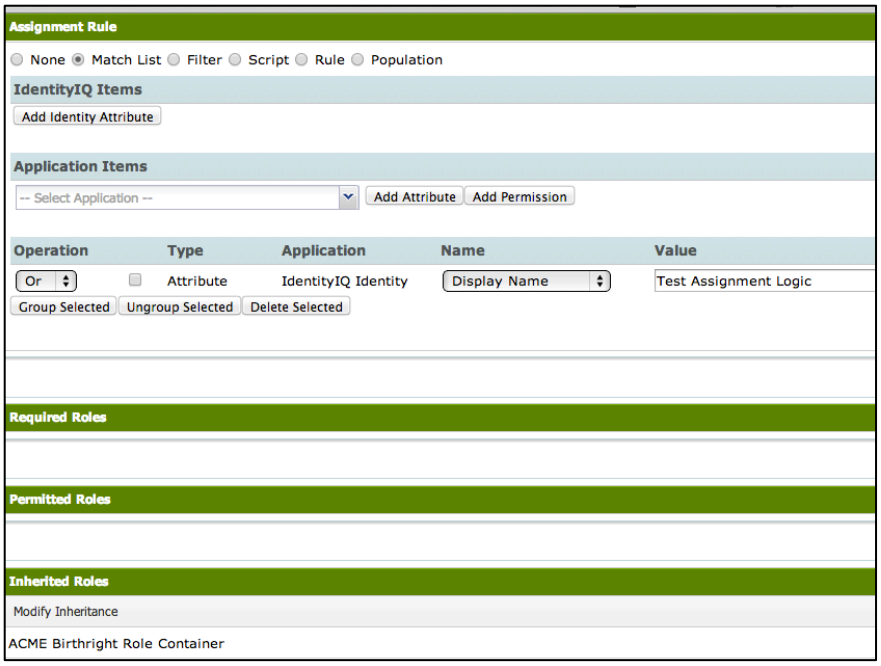

Figure 8 - Role Assignment Business Role

Step 3 – Add Rule Reference

If calling from a rule or rule library, add to the ReferencedRules entry. If calling from a workflow, add to the RuleLibraries entry.

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

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

Step 4 – Call the method(s)

There are two methods that can be called:

1. **getAddOrRemoveRolesAccountRequest** – Will include adds and removes
2. **getBirthrightRolesIIQAccountRequest** – Will only include adds

To build, use the following as an example:

```
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 could then 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>
      </Attributes>
</ProvisioningPlan>
```

Approvals

The Approval framework is a single sub process and set of configuration options that dynamically and iteratively processes all of the approval types required for a given use case, or request type. This allows for each different workflow type, or even set of requested items, to distinguish their required approval scheme and then process their unique approval requirements. For each approval type, it allows for easy configuration of the required approvers, the approval mode, and the work item config (reminders and escalation), and provides pre and post approval hooks. Just about any approval scenario can be handled and with relative ease.

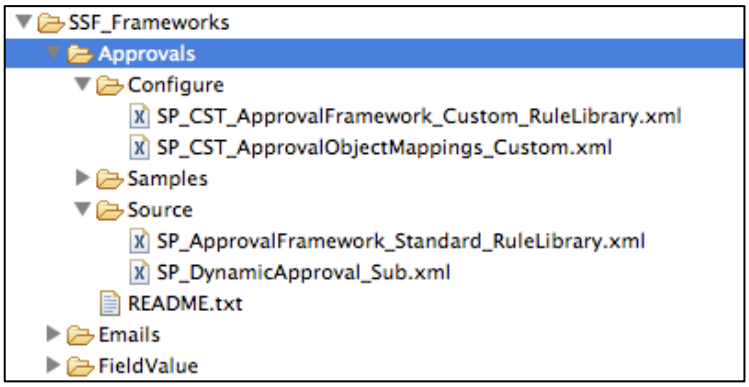

Figure 9 - Approvals Folder Structure

Intended Use

This framework is intended for all workflow approval scenarios that are more complex or require more customization than the OOB sub, which allows for manager, owner, and security officer. It is intended to scale for an array of scenarios and give greater control of each approval type. It will not, by default, handle asynchronous provisioning scenarios, which would require an overall different approach to handling provisioning requests.

Advantages

- **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.

How To

The basic steps to use the framework are:

1. Determine & configure get approval types method
2. Configure each approval type
3. Build custom rules
4. Call sub process with appropriate input variables

Step 1 – Determine & configure get approval type method

The custom approval object mappings object (mapping object), denoted by the property %%APPROVAL_FRAMEWORK_CUSTOM_OBJECT%%, drives the behavior of the approval framework.

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

- **Default** – uses the list of approval types found in the *Default Approval Types* entry. This is to be used for simple implementations that have the same approval requirements for all use cases.
- **Map** – uses the *Map Request Type Approvals Types* entry to lookup the required approval types for each use case, based on the requestType variable. This is to be used for slightly more complex implementations where the use case can be determined by the top-level workflow and for each use case there is a standard set of required approvals.
- **Custom** – uses the rule defined in the *Get Approval Types Custom Rule* entry to return approval types based on any variable passed in from the top-level workflow. This is to be used for complex implementations with non-linear approval requirements or approvals determined via more granular conditional statements based on the workflow, the requestee, the requested items, etc. This option 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. The custom rule will use the requestType variable to lookup 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 lookup the required approval types.

The following is an example that uses the Map option and for all LCM requests, Manager and Owner approval are required; for all Joiners, Manager and Department Head 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

Each approval type is configured in the **Approval Types** entry. For each type, a number of entries are either required or allowed:

- **preApprovalRule** – Specifies a rule that can be called before the approval. This rule can 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.
- **approvalAfterScript** – Specifies a rule that is called after the approval. 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.
- **approvalValidatorScript** – 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.
- **notifyEmailTemplate** – Specifies the email template that will be sent to the approvers.
- **approvalMode** – Specifies how the approval will behave. Options are the same as OOB: serial, serialPoll, parallel, parallelPoll and any.
- **displayName** – Specifies the display name of the approval that will appear in the approval title and in the approver's inbox.
- **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 options for the work item config in regards to reminders and escalations. See Configuring Work Item Config.
- **electronicSignature** – Provide the name of the Electronic Signature object to use if entering credentials is required. Provide no value 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 on top of the OOTB work item approval form and serves to add additional, read only fields for the approval.
- **getApprovalFormRule** – Enter the name of a rule or method. Define a rule or method to match the configuration entry and have the rule either return null or a sailpoint.object.Form object. This can be used to add additional, read only fields to the approval form. *NOTE: There is an open bug preventing full access to the identityModel variable set by the workItemFormBasePath entry. Any values that need to be displayed in the form need to be set in the rule/method supplied. This bug is slated to be resolved in 6.3p4 and 6.4.*
- **template** – Specifies another approval type entry to use as a template. This allows for reuse of configuration. Anything not explicitly defined in the given approval type will be overwritten by the entries in the given template.

The following is an example template approval type and an actual approval type that uses the template and overrides the approvalMode, getApprovalOwnersRule, and displayName:

```
<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="workItemDescriptionRule" value="SP CST Approval Work Item Description… 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>
```

Step 3 – Build custom rules

Note: the framework was updated to allow for using methods in the custom rule library, in place of writing individual rules. Both options are supported. Methods are denoted by prefixing the method name with "method:". In the following instructions "rule" can mean an actual rule or a library method. Library method is preferred.

A rule needs to be written for each entry that specifies a rule name. In the example above, the following rules would need to exist: SP CST Pre Approval Default Splitter Rule, SP CST Get Approvers Manager Rule, etc. There are sample rules provided for each. The following table details each rule type and its inputs and returns.

Type	Inputs	Return	Details
Pre	Workflow workflow String approvalType	boolean	Accepts the workflow var, allowing access to all variables. Logic can do anything. Must return a boolean of whether there are any items left to process for the given approval type.
After	ApprovalSet approvalSet, tempApprovalSetRem	void	Is used to manipulate the approval set and do any necessary merges from the filtered approval set. No returns are

	String approvalType		required.
Validator	ApprovalSet approvalSet WorkItem itm String approvalType	String or null	Allows doing any validation. Return a string if an error should be thrown. Return null if there are no errors.
Owners	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.
Approval Form	Workflow workflow	Sailpoint.object.Form	Build and return a custom form that will be presented to the approvers.
WI Desc	Workflow workflow	String	Calculate the work item description.

The following is an example method from the custom rule library:

```
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;
}
```

Figure 10 - Sample Approval Method

Step 4 – Call sub with appropriate input variables

Generally, the approval sub will be called as part of the provision processor sub (see next framework section); however, it can be used independently. The sub works off of the approvalSet variable, which is usually built in the initialize step of the main workflow (or provision processor sub) at the same time that the provisioning project and identity request object is built.

The approval sub can 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 hashmap 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** – name of actual requestor. In cases of LCE workflows, launcher 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
- **approvalSet** – contains all items being requested
- **requestType** – the name of the use case, often drives what set of approval types will be required
- **emailArgList** – list of hash maps, each map denoting an email that will be sent out. Sent it so that new emails can be appended 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
- **updateStandardPostApprovalEmails** – if true, then the emailArgList will be updated with approved/rejected emails

The approval sub will return the following variables:

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

The following is a sample call to the sub and 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 standard subprocess that will handle the built-in
      owner, manager and security officer approval schemes.
    </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>
```

Quick Hit Use Case – Simple Joiner

The following details a simple use case to illustrate how to use the framework. The joiner is an LCE 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.

Create a workflow with the following steps: Build Plan, Get Request Type, Initialize, Approve, Provision. The build plan step returns a provisioning plan with an IIQ modify request to add one of the two birthright roles. The get request type step returns a use case value. The initialize step calls the OOB Initialize sub and returns a provisioning project and approval set. The approve step calls the approval sub. The provision step calls the OOB provisioning sub to provision the birthright role.

After the basic workflow is configured, do the following steps:

1. Have Build Plan create a Provisioning Plan with an IIQ Account Request to assign the birthright role based on the user type
2. Have Get Request Type step return “Non-Employee Joiner” or “Employee Joiner” based on the user type
3. Update the *Approval Types Scheme* entry (approval object mappings) to “Map”
4. Update the *Map Request Type Approvals Types* entry with the following mappings:
    a. “Employee Joiner” – empty list
    b. “Non-Employee Joiner” – {“Manager”}
5. Add the following approval types entries:
    a. “Manager”
        i. preApprovalRule – None (can create a default rule that does nothing)
        ii. afterApprovalScriptRule – None
        iii. getApprovalOwnersRule – *cst Get Approval Owners Manager Rule*
        iv. approvalMode – any
        v. displayName – “Manager”
        vi. workItemDescriptionRule – *cst Default WI Desc Rule*
        vii. approvalValidatorScriptRule – None
        viii. notifyEmailTemplate – *cst Approval Email*
        ix. electronicSignature – *cst Default Electronic Signature*
        x. useCustomApprovalForm – *true*
        xi. getCustomApprovalFormRule – *cstGetApprovalFormRule*
6. Write the rule *cst Get Approval Owners Manager Rule*. Have it return identity.getManager().getName();
7. Write the rule *cst Default WI Desc Rule*. Have it return “Approval for “ + workflow.get(“identityDisplayName”);
8. Write the rule *cstGetApprovalFormRule*. Have it return a sailpoint.object.Form object;
9. Write the email template *cst Approval Email*.

Provision Processor

The Provision Processor framework is a set of subs that handle the basic steps common to all provisioning workflows: Initialize, Approve, Provision. The subs utilize the Approvals Framework and automatically append to the emailArgsList for the success or failure of provisioning activities. Most provisioning workflows can be simplified to four simple steps: Build Plan, Get Request Type, Call Provision Processor Sub, Send Emails.

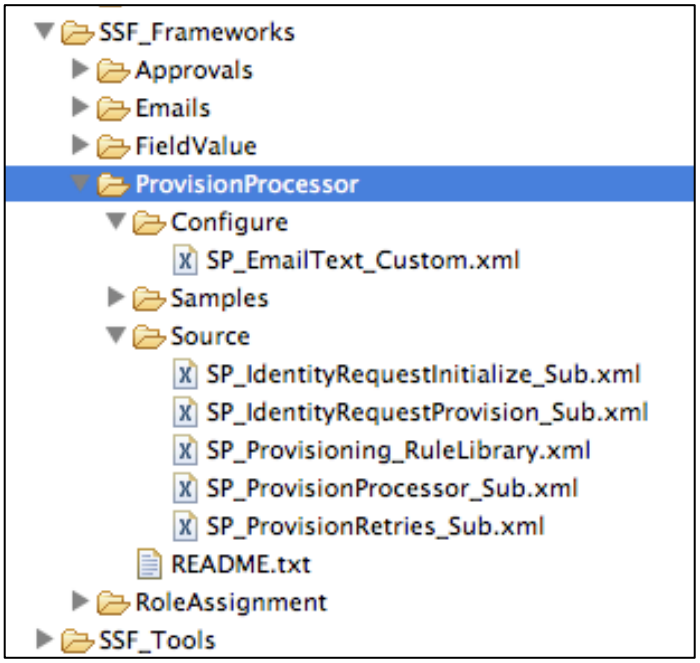

Figure 11 - Provision Processor Folder Structure

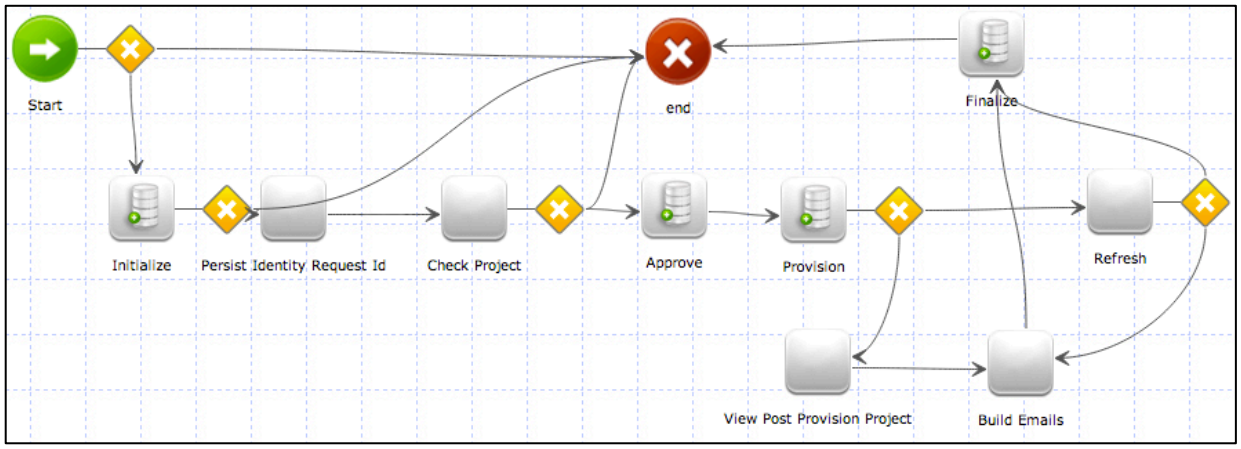

Figure 12 - Provision Processor Sub

Intended Use

This framework is intended for all provisioning workflows that follow the basic paradigm: Build Plan, Initialize, Approve and Provision. It is intended to eliminate the most basic workflow development and standardize all LCE & LCM workflows. Currently, some

restrictions may apply if extra forms are required before or after approvals or if any other steps need to be injected between any of the provision processor steps—Initialize, Approve, Provision. In those cases, the underlying subs can be called directly from the top workflow.

Advantages

- **Reduces workflow development** – The majority of the workflow steps common to all use cases are complete.
- **Reduces testing** – The underlying workflow steps are proven and tested.
- **Identity Request Integrity** – The subs 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.

How To

The basic steps to use the framework are:

1. Configure any required email text (optional)
2. Call sub process with appropriate input variables

Step 1 – Configure any required email text

NOTE: THIS IS OFTEN NOT USED. SEE BELOW FOR HOW TO TURN OFF.

Currently, the emailArgList will be appended with a hash map (or scheduled email) for each successful or failed account request in the provisioning project. (The variable updateStandardPostProvEmails can be set to false to turn this feature off.)

For each successful request, the map object will be updated with a customText entry based on the configuration of the custom object denoted by the property, %%SP_EMAIL_CUSTOM_OBJECT_NAME%%. The customText value can be referenced in the email template with: $emailArgs.customText.

To add custom text, edit this object and add an entry for each application and account-request operation combination. The following is a sample:

```
<Custom name="%%SP_EMAIL_TEXT_CUSTOM_OBJECT_NAME%%">
  <Attributes>
    <Map>
        <entry key="SAMPLE APP">
                <value>
                        <Attributes>
                                <Map>
                                        <entry key="Create">
                                                <value>
                                                        <String>
                                                                &lt;p&gt;ENTER CUSTOM INSTRUCTIONS TO
SEND WHEN AN ACCOUNT IS CREATED
                                                        &lt;/p&gt;
                                                        </String>
                                                </value>
                                        </entry>
                                </Map>
                        </Attributes>
                </value>
```

```
        </entry>
    </Map>
   </Attributes>
</Custom>
```

Step 2 – Call the sub process with the appropriate inputs

The sub works off of the identityName and plan and a few other attributes.

The sub 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.

The sub returns the following variables:

- **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 sub:

```
<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" />
    <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>
```

Quick Hit 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 the step to send emails at the end.

The joiner is an LCE 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.

Create a workflow with the following steps: Build Plan, Get Request Type, Process Plan, Send Emails. The build plan step returns a provisioning plan with an IIQ modify request to add a birthright role. The get request type step returns a use case value. The Process Plan step calls the framework, which will Initialize, Approve, and Provision the plan.

After the basic workflow is configured, do the following steps:

1. Have Build Plan create a Provisioning Plan with an IIQ Account Request to assign the birthright role based on the user type
2. Have Get Request Type step return “Non-Employee Joiner” or “Employee Joiner” based on the user type

3. Configure the approval framework's custom object, approval types, and custom rules. See use case in approval framework section.
4. Have Process Plan call the SP Provision Processor Sub with all of the required input variables.
5. Have Send Emails call the SP Send Dynamic Emails Sub with the emailArgList

Features

The following details each feature. Each feature offers the same advantages to writing custom workflows from scratch. By using a common architecture, each feature will have the same folder structure, set of target properties, and workflow steps. Additionally, each feature will have a custom library with the same methods that are used as “hooks” throughout the workflow process.

The advantages to each feature are:

- **Proven and tested methodology** – Due to the heavy usage of underlying frameworks, most of the components are already proven and tested.
- **Business-centric development** – 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 standard configurations, in hours or even minutes.

The folder structure will be as follows:

- **Root of Feature**
  - **Configure** – the files that need to be moved and configured
    - 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
    - A Rule Library
    - The Workflow and Trigger

Figure 13 - Example Feature Folder Structure

The target properties are:

- **IS_DISABLED** – Determines whether the features is on or off
- **WF_TRACE_ENABLED** – Whether or not to trace out the underlying workflow
- **CUSTOM_OBJECT_NAME** – The name of the custom mapping object in the repository
- **RULES_OBJECT_NAME** – The name of the custom rule library in the repository
- **SEND_APPROVED_EMAILS** – Whether to automatically send approved/rejected emails
- **SEND_POST_PROVISION_EMAILS** – Whether to automatically send provision success/failure emails

```
47 ## SPECIFY WHETHER DISABLED
48 %%SP_JOINER_IS_DISABLED%%=true
49 ## SPECIFY WHETHER WORKFLOW TRACE IS ENABLED
50 %%SP_JOINER_WF_TRACE_ENABLED%%=false
51 ## ENTER THE NAME OF THE JOINER CUSTOM MAPPING OBJECT NAME
52 %%SP_JOINER_CUSTOM_OBJECT_NAME%%=SP Joiner Mappings Custom
53 ## ENTER THE NAME OF THE CUSTOM JOINER RULES LIBRARY OBJECT NAME
54 %%SP_JOINER_RULES_OBJECT_NAME%%=SP Custom Joiner Rules Library
55 ## ENTER true/false ON WHETHER TO SEND DEFAULT APPROVED/REJECTED EMAILS
56 %%SP_JOINER_SEND_APPROVED_EMAILS%%=false
57 ## ENTER true/false ON WHETHER TO SEND DEFAULT POST PROVISION EMAILS
58 %%SP_JOINER_SEND_POST_PROVISION_EMAILS%%=false
```

Figure 14 - Example Target Properties

The workflow steps align with the underlying rule libraries, both source and custom. Generally, the feature workflows have the following steps:

- **Get Request Type** – calls custom library method to determine the string value that denotes the use case (request type) being processed
- **Before Plan** – calls custom library method to allow for any business logic customizations
- **Build Plan** – uses the options in the custom mapping to dynamically build out a provisioning plan
- **Before Provision** - calls custom library method to allow for any business logic customizations
- **Process Plan** – calls the Provision Processor Sub to Initialize, Approve, and Provision based on the plan. The Approve step calls the Approval Framework.
- **After Provision** - calls custom library method to allow for any business logic customizations
- **Send Emails** – calls the Send Emails Framework to dynamically send out any required emails

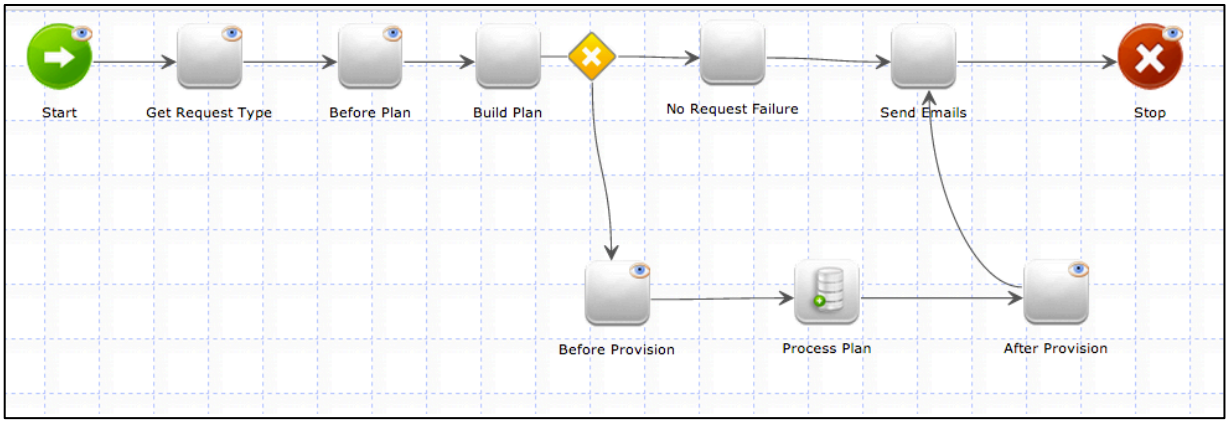

Figure 15 - Feature Workflow Steps

The methods available in the custom library allow for further customization. The main methods available are:

- **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
- **beforeProvisionRule** – Do any business logic customization
- **afterProvisionRule** – Do any business logic customization

There are also a number of methods to return to-addresses and email templates if either of the target properties for sending emails (SEND_APPROVED_EMAILS or SEND_POST_PROVISION_EMAILS) are turned on.

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

Joiner

The Joiner feature is to be used for onboarding to assign birthright access and offers:

- A role-type selector to determine when to trigger
- Four, easy-to-configure options for assignment of birthright access
- Hooks to configure email options and to return the request type for approvals
- Three strategic hooks (Before Plan, Before Provision, After Provision) to call out or process any additional workflow requirements

How To

The basic steps to use the framework are:

Step 1 – Configure Target Properties

Validate and set the target properties. Rename the objects, set disabled to false, and determine whether to automatically send post provision emails.

Step 2 – Configure Trigger Field Selector

The Identity Trigger uses an IdentitySelector found in the custom mapping object. The selector determines 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>
```

Step 3 – Select Birthright Assignment Type

The Birthright Assignment Type entry of the mapping object determines how the provisioning plan is constructed in the build plan step. Enter one of four options:

1. **Dynamic Roles** – This will dynamically assign roles by using the role assignment framework. In order for this to work, that framework must be in place and configured.
2. **Default Roles** – This will assign a static list of roles found in the Default Assignments entry of the mapping object.
3. **Default Applications** – This will assign a static list of applications found in the Default Assignments entry of the mapping object.
4. **Custom Rule** – This will use the rule defined in the Birthright Assignment Custom Rule entry of the mapping object. The rule will receive an Identity object and must return a ProvisioningPlan object.

Step 4 – Configure the Assignments

If the assignment type is Dynamic Roles, then configure the role assignment framework.

If the assignment type is Default Roles or Default Applications, simply enter the list of roles or applications in the Default Assignments entry. For example:

```
<!--  Used if Birthright Assignment Types is either Default Roles or Default Apps -->
<entry key="Default Assignments">
        <value>
                <List>
                <!-- Enter the names of the Bundles or Applications for automatic assignment -->
                        <!--  Examples if roles: -->
                        <String>Finance Role 1</String>

                        <!-- Examples if apps: -->
                        <String>Active Directory</String>
                        <String>Enterprise LDAP</String>

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

If the assignment type is Custom Rule, enter a rule name in the entry Birthright Assignment Custom Rule. This rule must then be written to accept an Identity and return a ProvisioningPlan.

Step 5 – Configure the Hooks

There are a number of methods in the custom rule library. Each method is called at a specific time in the workflow and allows the implementer to further customize the behavior of the workflow. See Appendix – Example Custom Rule Library for an example of a custom rule library.

Leaver

The Leaver feature is to be used for terminations and leaves of absence and offers:

- A role-type selector to determine when to trigger
- Four, easy-to-configure options for turning off access
- Hooks to configure email options and to return the request type for approvals
- Three strategic hooks (Before Plan, Before Provision, After Provision) to call out or process any additional workflow requirements

How To

The basic steps to use the framework are:

Step 1 – Configure Target Properties

Validate and set the target properties. Rename the objects, set disabled to false, and determine whether to automatically send post provision emails.

Step 2 – Configure Trigger Field Selector

The Identity Trigger uses an IdentitySelector found in the custom mapping object. The selector determines 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="T"/>
                                <MatchTerm name="inactive" value="false"/>
                        </MatchExpression>
                </IdentitySelector>
        </value>
</entry>
```

Step 3 – Select Leaver Build Plan Type

The Leaver Build Plan Type entry of the mapping 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 current access.
2. **Delete All** – This will delete all current access.
3. **Selective Lists** – This will strategically disable and delete based on selective lists of applications.
4. **Custom Rule** – This will use the rule defined in the Leaver Build Plan Custom Rule entry of the mapping object. The rule will receive an Identity object and must return a ProvisioningPlan object.

Step 4 – Configure the Selective Lists

If the assignment type is Selective Lists, simply enter the list of applications in either the Default Disables or Default Deletes entries. For example:

```
<!-- 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>
```

Figure 16 - Leaver Selective Lists

If the assignment type is Custom Rule, enter a rule name in the entry Leaver Build Plan Custom Rule. This rule must then be written to accept an Identity and return a ProvisioningPlan.

Step 5 – Configure the Hooks

There are a number of methods in the custom rule library. Each method is called at a specific time in the workflow and allows the implementer to further customize the behavior of the workflow. See Appendix – Example Custom Rule Library for an example of a custom rule library.

Attribute Synch

The Attribute Synch feature is to be used for detecting source attribute changes and using these to recalculate and provision target attribute updates. It offers:

- Multiple trigger options:
  - Role type selector
  - Source Link Comparison
  - Custom Rule
- The ability to skip fields, usually fields that are only provisioned on create
- Hooks to configure email options and to return the request type for approvals
- Three strategic hooks (Before Plan, Before Provision, After Provision) to call out or process any additional workflow requirements

How To

The basic steps to use the framework are:

Step 1 – Configure Target Properties

Validate and set the target properties. Rename the objects, set disabled to false, and determine whether to automatically send post provision emails.

Step 2 – Configure Trigger Options

The Identity Trigger allows for three different models:

1. **Selector** – This option uses a role model selector based on identity and link attributes. If this option is selected, you must configure the **Trigger Field Selector** entry.
2. **Compare Links** – This option will dynamically compare the attribute values in selected links from the previous and new identities. If it finds any differences, it will launch the workflow. If this option is selected, you must configure the **Trigger Compare Links** and **Trigger Compare Links Schemas** entries.
   a. **Trigger Compare Links** – Enter the names of accounts to review for changes.
   b. **Trigger Compare Links Schemas** – Per application name, enter the attributes to review for changes. It can be beneficial to not check for attributes that impact other lifecycle events, such as user status.
3. **Custom Rule** – This option allows you to write a custom rule or method. If this option is selected, you must write a rule or method and enter its name in the **Trigger Custom Rule** entry.
   a. **Trigger Custom Rule** – Enter the name of a rule or method. This is defaulted to method:isAttrSynchCustomTriggerRule, which is already included in the Custom Rule Library.

```
<!--  Enter the Trigger Type.
        Options Include:
            - Selector:  Will dynamically re-evalute all target attributes and update as neces
            - Compare Links:  Will review the links defined in the Compare Links entry below.
                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
                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:isAttrSynchCustomTriggerRule" />

<!--  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>

<!-- 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 attribut
         -->
<entry key="Trigger Compare Links Schemas">
    <value>
        <Attributes>
            <Map>
                <entry key="HR">
                    <value>
                        <List>
                            <String>EMPL_ID</String>
                            <String>FIRST_NAME</String>
```

Figure 17 - Attribute Synch Trigger Options

Step 3 – Select Plan Construction Type

The Plan Construction Type entry determines how the provisioning plan is constructed in the build plan step. Enter one of the options:

1. **Dynamic Targets** – This will dynamically evaluate all target applications. If set, the logic will use the **Application Skip Fields** entry to determine what fields to bypass on a per-account basis.
2. **Custom Rule** – This option allows you to write a custom rule or method. If this option is selected, you must write a rule or method and enter its name in the **Plan Construction Custom Rule** entry.

Step 4 – Configure the Application Skip Fields

Enter the application names or application types and for each, their respective lists of fields to ignore. For example:

```
        Fields resulting in a rename might be handled specially
         -->
    <entry key="Application Skip Fields">
        <value>
            <Attributes>
                <Map>
                    <entry key="Application Name">
                        <value>
                            <List>
                                <String></String>
                            </List>
                        </value>
                    </entry>
                    <entry key="Active Directory - Direct">
                        <value>
                            <List>
                                <String>userAccountControl</String>
                                <String>extendedAttribute5</String>
                            </List>
                        </value>
                    </entry>
                    <entry key="LDAP">
                        <value>
                            <List>
                                <String>carLicense</String>
                            </List>
                        </value>
                    </entry>
                </Map>
            </Attributes>

        </value>

    </entry>
```

Figure 18 - Application Skip Fields

Step 5 – Configure the Hooks

There are a number of methods in the custom rule library. Each method is called at a specific time in the workflow and allows the implementer to further customize the behavior of the workflow. See Appendix – Example Custom Rule Library for an example of a custom rule library.

Mover

The Mover feature is to be used for detecting significant source attribute changes, such as department or organization, and using these to recalculate the user's access, update target attributes, and/or launch an access review. It offers:

- Multiple trigger options:
  - Role type selector
  - Source Link Comparison
  - Custom Rule
- Three built-in, additive plan construction types
  - Dynamic roles – uses role assignment to recalculate the user's access
  - Attribute synch – integrates with the attribute synch feature to recalculate downstream, target attribute values
  - Custom rule – allows develop to append their own logic
- Ability to automatically launch a manager certification
- Hooks to configure email options and to return the request type for approvals
- Three strategic hooks (Before Plan, Before Provision, After Provision) to call out or process any additional workflow requirements

How To

The basic steps to use the framework are:

Step 1 – Configure Target Properties

Validate and set the target properties. Rename the objects, set disabled to false, and determine whether to automatically send post provision emails.

Step 2 – Configure Trigger Options

The Identity Trigger allows for three different models:

1. **Selector** – This option uses a role model selector based on identity and link attributes. If this option is selected, you must configure the **Trigger Field Selector** entry.
2. **Compare Links** – This option will dynamically compare the attribute values in selected links from the previous and new identities. If it finds any differences, it will launch the workflow. If this option is selected, you must configure the **Trigger Compare Links** and **Trigger Compare Links Schemas** entries.
   a. **Trigger Compare Links** – Enter the names of accounts to review for changes.
   b. **Trigger Compare Links Schemas** – Per application name, enter the attributes to review for changes. It can be beneficial to not check for attributes that impact other lifecycle events, such as user status.
3. **Custom Rule** – This option allows you to write a custom rule or method. If this option is selected, you must write a rule or method and enter its name in the **Trigger Custom Rule** entry.
   a. **Trigger Custom Rule** – Enter the name of a rule or method. This is defaulted to method:isAttrSynchCustomTriggerRule, which is already included in the Custom Rule Library.

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 and the options are processed in the order below. Select the 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. **Custom Rule** – This option allows you to write a custom rule or method. If this option is selected, you must write a rule or method and enter its name in the **Plan Construction Custom Rule** entry.

Step 4 – Determine Whether to Launch the Manager Certification

Set the **Launch Manager Cert** entry to either true or false. If true, a manager cert will be launched at the end of the workflow.

Step 5 – Configure the Hooks

There are a number of methods in the custom rule library. Each method is called at a specific time in the workflow and allows the implementer to further customize the behavior of the workflow. See Appendix – Example Custom Rule Library for an example of a custom rule library.

Rehire

The Rehire feature is to be used for identities that had been terminated or placed on leave and are now returning as an active user, in order to recalculate the user's access, update target attributes, and/or enable existing access. It offers:

- Multiple trigger options:
  - Role type selector
  - Source Link Comparison
  - Custom Rule
- Four built-in, additive plan construction types
  - Dynamic roles – uses role assignment to recalculate the user's access
  - Attribute synch – integrates with the attribute synch feature to recalculate downstream, target attribute values
  - Enable Accounts – enables any existing access the user still has
  - Custom rule – allows develop to append their own logic
- Hooks to configure email options and to return the request type for approvals
- Three strategic hooks (Before Plan, Before Provision, After Provision) to call out or process any additional workflow requirements

How To

The basic steps to use the framework are:

Step 1 – Configure Target Properties

Validate and set the target properties. Rename the objects, set disabled to false, and determine whether to automatically send post provision emails.

Step 2 – Configure Trigger Options

The Identity Trigger allows for three different models:

1. **Selector** – This option uses a role model selector based on identity and link attributes. If this option is selected, you must configure the **Trigger Field Selector** entry.
2. **Compare Links** – This option will dynamically compare the attribute values in selected links from the previous and new identities. If it finds any differences, it will launch the workflow. If this option is selected, you must configure the **Trigger Compare Links** and **Trigger Compare Links Schemas** entries.
   a. **Trigger Compare Links** – Enter the names of accounts to review for changes.
   b. **Trigger Compare Links Schemas** – Per application name, enter the attributes to review for changes. It can be beneficial to not check for attributes that impact other lifecycle events, such as user status.
3. **Custom Rule** – This option allows you to write a custom rule or method. If this option is selected, you must write a rule or method and enter its name in the **Trigger Custom Rule** entry.
   a. **Trigger Custom Rule** – Enter the name of a rule or method. This is defaulted to method:isRehireCustomTriggerRule, which is already included in the Custom Rule Library.

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 and the options are processed in the order below. Select the 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. If set, the logic will use the **Enable Accounts Applications** entry to selectively determine which accounts to enable.
4. **Custom Rule** – This option allows you to write a custom rule or method. If this option is selected, you must write a rule or method and enter its name in the **Plan Construction Custom Rule** entry.

Step 4 – Configure the Hooks

There are a number of methods in the custom rule library. Each method is called at a specific time in the workflow and allows the implementer to further customize the behavior of the workflow. See Appendix – Example Custom Rule Library for an example of a custom rule library.

Troubleshooting

The following details steps that can be taken to troubleshoot the SSF.

Logging

All SSF rules and workflows are configured for log4j. The following is a quick list of log4j statements that can be added to /WEB-INF/classes/log4j.properties:

```
#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
```

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

```
%%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%%
```

Appendix – Example Custom Rule Library

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

- The getRequestTypeRule method calculates and returns request type based on an identity attribute, “userType”.
- The beforePlanRule method updates some workflow attributes, namely spExtAttrs, which will be passed up and down the chain of workflows and can be access 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 just 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 getRequestTypeRule(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 beforePlanRule(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 beforeProvisionRule(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 afterProvisionRule(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");
	}
```