SSF Tools: Customization Plugin User Guide

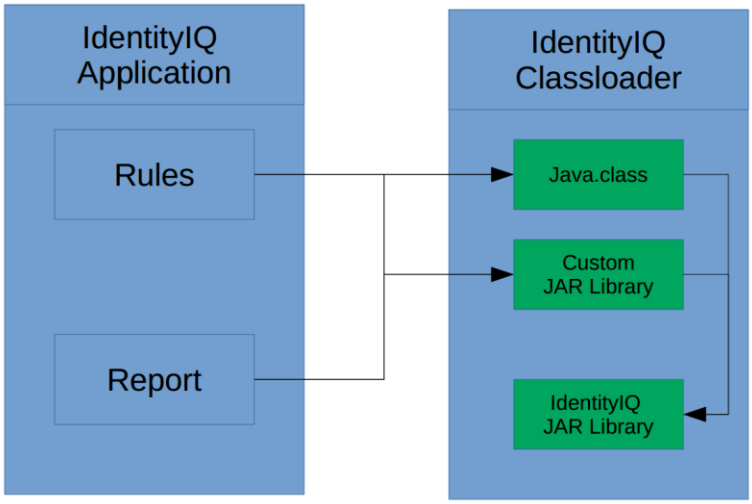
Document Revision History

Revision Date	Written/Edited By	Comments
December 2017	Lukasz Tupaj, Paul Wheeler, Jennifer Mitchell	Initial release with SSD v5.
May 2019	Paul Wheeler	Updated path that plugin is copied to, due to changes in SSB Plugin Deployer in SSD v6.1.

Table of Contents

Introduction

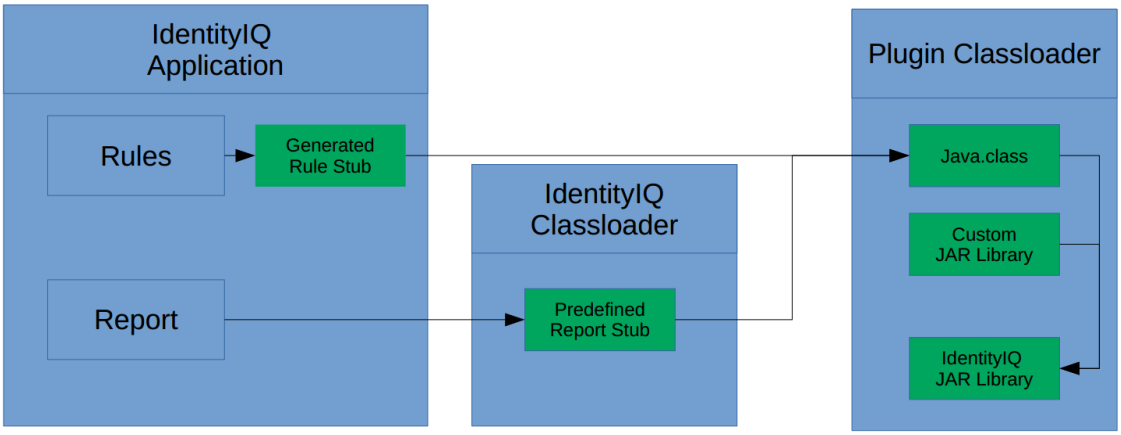
When developing logic for rules, rule libraries, and reports in IdentityIQ, some developers prefer to use pure Java for coding rather than BeanShell. Using this development practice, a BeanShell rule will call out to a method in a compiled Java class instead of executing code from a BeanShell rule or rule library, or a report will reference a compiled Java data source class. This is represented in the diagram below.



The two requirements that make this development practice cumbersome are the need for filesystem access to all the application servers to deploy the updated class, and the need to restart application servers after deploying it, both of which can significantly extend the time required for development and testing. The Customization Plugin is designed to address these issues by allowing “hot deployment” of Java classes through the IdentityIQ UI, without the need to restart application servers.

The IdentityIQ Plugin Framework is an extension framework model for IdentityIQ which enables third parties to develop rich application and service-level enhancements to the core SailPoint platform. Plugins may be installed, updated and removed via the UI, with no requirement for direct filesystem access to the application servers and no need to restart the servers. So adding Java classes to a plugin would seem to be a solution to the above issues for developers who wish to use Java for defining methods for rules, rule libraries, and reports, since the classes can be hot-deployed in the plugin. However, classes that are deployed as part of the Plugin Framework are loaded using a different class loader than regular classes that are added to the filesystem for use by IdentityIQ and methods in these classes are not accessible by default from BeanShell code or report definitions. The Customization Plugin provides a solution that allows access to methods defined in classes deployed in a plugin, simplifying the deployment of Java code.

Using this solution, when a rule or custom report is deployed as part of a plugin, additional stubs are needed to call code from there; those stubs are provided by this package.



The Customization Plugin can be used with IdentityIQ version 7.1 and greater.

Installation and Configuration

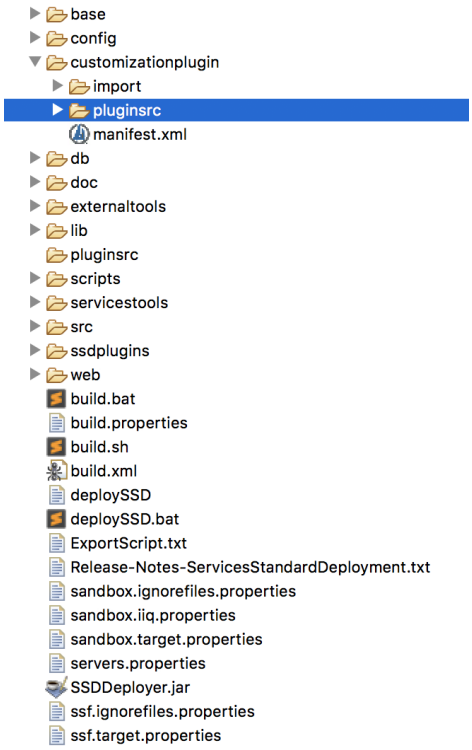
The Customization Plugin is included in the Services Standard Deployment (SSD). To enable it, set the following property in the build.properties file:

```
buildDeploymentPlugin=true
```

This causes the build to pick up a target specified in a second Ant script which builds the plugin for auto-deployment on server startup or for manual deployment through the IdentityIQ Plugin UI page. These deployment details are described below in *Building and Deploying the Customization Plugin*.

Components

The structure of the SSD, including the Customization Plugin, is shown below:



These are the components of the Customization Plugin which are found in the SSB structure:

- customizationplugin folder
- GenericPluginReport Java class
- Ant script for building plugin (in scripts folder)
- 4 lines in the SSB's main build.properties file

The "customizationplugin" folder at the root of the SSD contains the main components of the Customization Plugin:

- **manifest.xml** - the manifest file is used during the plugin build process and is parameterized by tokens substituted with values provided in the build.properties file. The plugin version is calculated and recorded in IdentityIQ during plugin installation as <plugin major version>.<time in format YYYYMMDDHHmmSS>.
- **import folder** – this contains XML definitions of IdentityIQ objects which may be imported into IdentityIQ during plugin deployment. This folder contains two subfolders "install" and "upgrade". For the report use case discussed in this document, the report's TaskDefinition would be included here, though the plugin could be used for deploying a package of things beyond just this customization functionality, so other XML artifacts might also be included here.
- **pluginsrc folder** – this is the most important folder for this whole mechanism: all Java code which needs to be dynamically deployed should be placed here.

The following Java files are included in the pluginsrc folder and are compiled when the plugin is built. These contain source code relating to the annotations used by the plugin to generate stubs (described later). They are required and should not be removed.

- customizationplugin/pluginsrc/sailpoint/services/standard/CustomizationPluginAnnotation/AnnotationBrowser.java
- customizationplugin/pluginsrc/sailpoint/services/standard/CustomizationPluginAnnotation/BshObject.java
- customizationplugin/pluginsrc/sailpoint/services/standard/CustomizationPluginAnnotation/Rule.java
- customizationplugin/pluginsrc/sailpoint/services/standard/CustomizationPluginAnnotation/RuleArgument.java
- customizationplugin/pluginsrc/sailpoint/services/standard/CustomizationPluginAnnotation/RuleLibrary.java
- customizationplugin/pluginsrc/sailpoint/services/standard/CustomizationPluginAnnotation/RuleLibraryMethod.java

The Customization Plugin includes a Java class which acts as a stub data source for reports. After compilation by the build process the class can be found here:

WEB-INF/classes/sailpoint/services/standard/customizationplugin/GenericPluginReport.class

The plugin build process is part of the main SSB build, and an Ant script is included in the “scripts” folder at the root of the SSD:

scripts/build.deployment.plugin.xml

This script contains a target called “sp.services.buildDeploymentPlugin” which is called from the main build.xml file to build the plugin.

The following properties in the build.properties file relate to the Customization Plugin and can be modified as required:

```
#Use plugin mechanism to deploy customizations
buildDeploymentPlugin=true #enables and disables plugin build
deploymentPluginVersion=0.1 #major version of plugin
deploymentPluginName=DeploymentPlugin #customization plugin name
deploymentPluginDisplayName=Deployment Plugin #customization display plugin name
```

Using the Customization Plugin

Unlike most plugins, the Customization Plugin has no UI components and is only used as a way to help developers deploy Java classes as part of a plugin. The classes may be referenced in:

- Java data sources for reports
- IdentityIQ rules and rule libraries

Referencing plugin-deployed Java classes from any other objects is not supported by the plugin.

Using the Customization Plugin with Reports

When a Java data source is used in a report, the IdentityIQ reporting engine calls a report definition located in the default class loader. To allow classes that are loaded by the plugin class loader to be accessed by a report, the GenericPluginReport stub data source must be referenced as the dataSourceClass in the XML for the report. This will redirect execution of the Java data source for the report to the plugin class loader.

A report that uses the Customization Plugin to access a Java data source in another plugin should be defined with the following components:

- The **dataSourceClass** attribute in the DataSource definition must point to "sailpoint.services.standard.customizationplugin.GenericPluginReport"
- An attribute called **pluginName** must specify the name of the plugin where the actual report data source is defined
- An attribute called **pluginClassName** must contain the fully qualified class name of the class where the data source is defined
- The Java data source in the plugin should implement the "JavaDataSource" interface, just like any other report that uses a Java data source

This example illustrates what the report definition would look like with these key elements highlighted.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE TaskDefinition PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<TaskDefinition name="Report Example" executor="sailpoint.reporting.LiveReportExecutor" subType="Identity
Reports"   resultAction="Rename" progressMode="Percentage" template="true" type="LiveReport">
   <Description>Sample report</Description>
   <RequiredRights>
      <Reference class="sailpoint.object.SPRight" name="FullAccessBusinessRoleMembershipReport" />
   </RequiredRights>
   <Attributes>
      <Map>
         <entry key="pluginName" value="DeploymentPlugin"/>
         <entry key="pluginClassName" value="com.sailpoint.services.acme.reports.ExampleReportDataSource"/>
         <entry key="report">
            <value>
               <LiveReport title="Report Example ">
                  <DataSource type="Java"
                     dataSourceClass="sailpoint.services.standard.customizationplugin.GenericPluginReport"
                     defaultSort="name">
                     <QueryParameters>
                        <Parameter argument="applications" property="links.application.id" />
                     </QueryParameters>
                  </DataSource>
                  <Columns>
```

```
                        <ReportColumnConfig field="identityName" header="Identity Name"
                        property="identityName" sortable="true" />
                        <ReportColumnConfig field="roleName" header="Role Name"
                        property="roleName" sortable="true" />
                        <ReportColumnConfig field="applicationName" header="Application Name"
                        property="applicationName" sortable="true" />
                        <ReportColumnConfig field="groupName" header="Group Name"
                        property="groupName" sortable="true" />
                        <ReportColumnConfig field="accountName" header="Account Name"
                        property="accountName" sortable="true" />
                    </Columns>
                </LiveReport>
            </value>
         </entry>
      </Map>
   </Attributes>
   <Signature>
      <Inputs>
         <Argument multi="true" name="applications" type="Application" />
         <Argument multi="false" required="false" displayName="Identity" name="identity" type="Identity" />
         <Argument multi="false" displayName="Manager" name="manager" type="Identity" />
      </Inputs>
   </Signature>
</TaskDefinition>
```

Using the Customization Plugin with Rules and Rule Libraries

As explained in the introduction, calling Java code from within a BeanShell rule is a common practice for some implementers, particularly when the implementer is trying to develop a big piece of business logic involving code that is too complicated to deploy as BeanShell. Plus, some implementers may just prefer to work that way and some implementations may choose to standardize on using Java over BeanShell. Without the Customization Plugin, this requires filesystem access to the application servers and the need to restart them every time code is modified.

The Customization Plugin mechanism offers the ability to define rules and rule libraries that call Java code located in and deployed as part of the plugin.

Without the Customization Plugin, you could, for example, have a method in a custom class, deployed as a regular Java object, called from a certification exclusion rule. The class would look like this:

```
public class AcmeCertificationHelper {
        private static Logger log = Logger.getLogger(AcmeCertificationHelper.class);

        public static String exclusionRule(SailPointContext context, List items, List itemsToExclude, Identity
identity) {
                //TODO: Some logic goes here
        }
}
```

The BeanShell rule which calls the Java code would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Rule language="beanshell" name="Acme_ExclusionRule" type="CertificationExclusion">
  <Source><![CDATA[
com.sailpoint.services.acme.certification.AcmeCertificationHelper.exclusionRule(context,items,itemsToExclude,id
entity);
  ]]></Source>
```

```
</Rule>
```

With the Customization Plugin, the developer does not have to write the BeanShell rule directly. They can instead specify certain annotations in their Java class that cause the plugin to generate the rule stubs to call the class automatically. Generation of the BeanShell stubs is an automated part of the build process.

For instance, to develop the same certification exclusion logic shown in the previous example we can create the Java class like this. The developer needs to provide the rule naming and argument details using annotations in the Java class:

```
public class AcmeCertificationHelper {
        private static Logger log = Logger.getLogger(AcmeCertificationHelper.class);

        @Rule(name = "Acme_ExclusionRule", type = "CertificationExclusion")
        public static String exclusionRule(@RuleArgument(name = "context") SailPointContext context,
@RuleArgument(name = "items") List items, @RuleArgument(name = "itemsToExclude") List itemsToExclude,
                        @RuleArgument(name = "identity") Identity identity) {

                //TODO: Some logic goes here

        }
}
```

To generate a rule stub in IdentityIQ, the following annotations are required:

- **@Rule** – this annotation is specified on the method in the class. It tells the build process that for this "public static" method, a BeanShell stub needs to be created. This annotation takes two arguments:
    - **name** – this value will be used as BeanShell rule name in IIQ
    - **type** – the value provided here will be used to define the "type" property of the rule
- **@RuleArgument** – this annotation is used to remap an argument passed to a rule to a Java method argument. This annotation takes one argument:
    - **name** – the value provided in this field defines the name of the variable passed to the IdentityIQ rule and will be mapped to a method argument. For example:

      ```
      exclusionRule(@RuleArgument(name = "context") SailPointContext context,
      ```

      This means that the variable "context" will be passed to the "context" argument of this method.

BeanShell stubs are generated automatically during the Customization Plugin generation process in the standard build. When a plugin is deployed, all stubs are automatically imported into the target environment.

This Rule stub would be dynamically-generated stub for the certification exclusion Java class example above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Rule language="beanshell" name="Acme_ExclusionRule" type="CertificationExclusion">
```

```
  <Source><![CDATA[

        ClassLoader clsLoader =
sailpoint.server.Environment.getEnvironment().getPluginsCache().getClassLoader("CustomizationPlugin");

        try
        {
                Class clObj =
clsLoader.loadClass("com.sailpoint.services.acme.certification.AcmeCertificationHelper");
                java.lang.reflect.Method mToCall =
clObj.getMethod("exclusionRule",sailpoint.api.SailPointContext.class,java.util.List.class,java.util.List.class,
sailpoint.object.Identity.class);
                return mToCall.invoke(null,context,items,itemsToExclude,identity);


        }
        catch(Exception e)
        {
                log.error("exclusionRule call error",e);
                throw e;
        }

        ]]></Source>
</Rule>
```

The same process applies to rule libraries. Rule libraries allow reuse of BeanShell code in other rules or scripts. The build process for the Customization Plugin can also generate BeanShell rule library stubs for Java classes deployed as part of the plugin.

Consider this example Java class:

```
public class TestLibrary {
        public static String someMethod1(SailPointContext context)
        {
                return "output";
        }

        public static String someMethod2(SailPointContext context, String identityName)
        {
                return "output";
        }
}
```

By adding the @RuleLibrary and @RuleLibraryMethod annotations, this class can be adapted to auto-generate a BeanShell rule library available to other BeanShell objects:

```
@RuleLibrary(name="TestLibrary")
public class TestLibrary {

        @RuleLibraryMethod
        public static String someMethod1(SailPointContext context)
        {
                return "output";
        }
        @RuleLibraryMethod
        public static String someMethod2(SailPointContext context, String identityName)
        {
                return "output";
        }
}
```

These annotations would result in the creation of a BeanShell rule library called “TestLibrary”, through which two public methods, “someMethod1” and “someMethod2”, would be available.

For Rule Libraries, these annotations are required:

- **RuleLibrary** – this annotation is specified at the class level; it tells the build script that for this class, a rule library stub needs to be created. The name argument specifies the name of the BeanShell rule object to be generated.
- **RuleLibraryMethod** – this annotation needs to be used for each “public static” method which we want to have available in the generated rule library. The output rule library stub contains the definition of all methods annotated by “RuleLibraryMethod”. Method names and arguments will be set exactly as specified in the source Java object.

Building and Deploying the Customization Plugin

When you have developed your Java classes that you want to deploy as part of the plugin, you can build and deploy the plugin as part of the main SSB build process. Ensure you have your Java code in a package folder structure under the pluginsrc folder and that you have the “buildDeploymentPlugin” property set to “true” in the SSB’s build.properties file, then run the normal build process.

When the SSB project is built by the main ant script, the generated plugin is placed in two output folders:

- build/buildCustomizationPlugin/DeploymentPlugin/dist/<file>.zip
- build/extract/WEB-INF/plugins/ssb/install/<file>.zip

The second of these allows the plugin to be deployed automatically on server startup (See the SSB User Guide for more information). Alternatively, you can deploy the plugin zip file through the IdentityIQ UI (Gear icon -> Plugins -> New).