Plugins

Version: 8.2

Revised: June 2021

Copyright and Trademark NoticesCopyright © 2020 SailPoint Technologies, Inc. All Rights Reserved.

Contents

Working with Plugins

The SailPoint Plugin Framework is an extension framework model for IdentityIQ. It enables third parties to develop rich application and service-level enhancements to the core SailPoint platform. It enables plugins to extend the standard user interface, deliver custom REST endpoints, and to deliver custom background services.

A plugin can be a simple REST service or a full page application on top of IdentityIQ. A plugin can include one or all of the following:

- A client side front end
- REST web services
- ServiceDefinition, PolicyDefinitions, and TaskDefinition implementations
- Java classes available for scripting
- Custom plugin configuration
- Database tables

During your initial installation, IdentityIQ is set up to work with plugins. A separate plugin table, `identityiqPlugin`, is created as part of the database schema creation scripts and the `plugins.runSqlScripts`, `plugins.importObjects`, and `plugins.enabled` properties are set to `true` in the `iiq.properties` file.

To disable plugins completely in IdentityIQ, set the plugin property values to `false`.

Plugin Framework

The plugin framework manages the installation and loading of plugins. It provides:

Class path isolation on the server side

Implementers are free to use any third-party libraries or technology they choose, as long as it can be served from a REST end point, a background service, or a Java class called from scripts.

JavaScript isolation on the client side

Implementers are free to use any third-party client side libraries.

Core code protection

The framework insures and certifies no plugin overrides or changes backend product code behavior. Essential for security and upgrading.

Web service extensions

Implementers can define custom REST end points to push and pull data between their plugin and the SailPoint data model.

Plugin installation and removal

Plugins can be dynamically loaded to provide drag and drop installation and removal, or you can choose to require installation prior to application startup.

A plugin's user interface can be as simple as a piece of JavaScript or text injected on an existing page or a full page plugin. The behavior is defined by the `manifest.xml` in the plugin's root directory.

Working with Plugins in IdentityIQ

The plugin feature must be enabled in IdentityIQ and you must have the proper access, such as System Administrator or Plugin Administrator capabilities, before this page can be displayed.

The **Installed Plugins** page displays and enables you to manage your plugins from within IdentityIQ. Open the page by selecting **Plugins** from the list under the **gear** icon.

From the Installed Plugins page you can install, uninstall, enable, disable, and configure your plugins.

- **Install** — click New and either drag and drop a zip file onto the page, or navigate to the directory containing the plugins.
- **Enable/Disable** — click the power button icon to enable or disable plugins. You will be asked to confirm your decision.
- **Uninstall** — click the x icon to uninstall a plugin. See Plugin Installation and Removal for more information on removing a plugin.

See, Configure Plugins Page

Configure Plugins Page

Each plugin can include a Configuration option, which is accessed through a button on the Plugin card on the **gear icon > Plugins** page.

The Configuration page enables you to view detailed information about the plugin, including its version, installation date, and certification level.

This page also enables you to change the values of the Settings objects for the plugin, based on the object type. If no Setting objects were defined when the plugin was created, none will display on the Configuration page.

Working with Plugins from the IdentityIQ Console

The IdentityIQ Console (`iiq console`) provides several commands for managing plugins, as well as for performing scripted installation of multiple plugins. See the **IdentityIQ Console** documentation for more information.

The iiq console includes these plugin commands:

plugin install

Install a single plugin or multiple plugins. Either a path to the zip file of the plugin or a directory containing multiple plugin zip files can be specified. If a directory is specified, any zip file in that directory is installed.

Flags:

- file — path to a plugin file
- dir — the directory containing the plugin zip file to install
- no-cache — the plugin should not be cached after install

plugin upgrade

> For versioning information, see Developing Plugins.

Upgrade to a newer version of a plugin. Upgrading a plugin to the same version or a previous version is not supported. While developing a plugin, this behavior can be disabled for easier testing. To do so, include a "-dev" suffix on the version, for example, 2.0-dev.

The version of a plugin can either be official or development.

Development versions end with the suffix '-dev,' for example, 2.0-dev, and bypass most version checks so that the plugin can be recompiled, upgraded and tested easily.

Official versions drop the '-dev' suffix and can only be installed over a development version or an earlier official version. The minimum upgradeable version must also be valid.

Valid upgrade paths:

- 1.0 -> 2.0-dev
- 2.0-dev -> 2.0-dev
- 2.0-dev -> 2.0
- 1.0 -> 2.0

Invalid upgrade paths:

- 2.0 -> 2.0
- 2.0 -> 1.0

Flags:

- file — path to a plugin file
- no-cache — the plugin should not be cached after the upgrade

plugin uninstall

Uninstall a plugin.

Flags:

- id — plugin id
- name — plugin name

plugin enable

Enable the plugin.

Flags:

- id — plugin id
- name — plugin name
- no-cache — the plugin should not be cached after being enabled

plugin disable

Disable a plugin.

Flags:

- id — plugin id
- name — plugin name

plugin export

Export a single or all installed plugins to their respective zip files and, optionally, a specified directory.

Flags:

- id — plugin id
- name — plugin name
- * — export all installed plugins
- dir — the directory in which to save the zip files. If the directory does not exist, the command will attempt to create one. If none is specified, the files are save to the current working directory.

plugin status

The enabled status of a single plugin, all installed plugins or the system-wide enabled status of plugins.

Flags:

- id — plugin id
- name — plugin name
- * — view the status of all plugins
- no flag — system-wide status of plugins as defined in iiq.properties

plugin list

A list of all installed plugins.

plugin classes

The list of the classes available (from a plugin or all plugins), and the intended use for each class.

Flags:

- id — plugin id
- name — plugin name
- * — the list of all available classes from all plugins

Developing Plugins

IdentityIQ stores the .zip archive file of the Plugin in the IdentityIQ database in a data LONGBLOB in the `spt_file_bucket` table. The data in the `spt_file_bucket` table is referenced ID to an entry in the `spt_persisted_file` table.

Plugins are loaded from this .zip file after installation or after an application server restart. The .zip file is extracted, and all important files are cached for later use. There are several accessor methods to reference the cached files, but they can also be referenced by the url prefix /identityiq/plugin/*pluginName* followed by the path found in the build structure. Compiled java classes are loaded and cached from the .zip archive using the PluginClassLoader class.

Plugin Versioning Requirements

> The single exception to these requirements are version numbers with -dev appended to the end. This suffix causes version number validation to be bypassed.

To provide better support for upgrading plugins, plugin version numbers must be numeric, contain no alphabetic or other characters, and must separate the elements of the version number with decimal points. Within each segment of the version number, the values between the decimal points, the values are cast as integers, and leading zeroes are trimmed.

For example:

- 04 and 00004 are both interpreted as 4
- A segment containing any non-numeric values is interpreted as 0
- 1.004.alpha is parsed as 1.4.0
- 2.3.4a is parsed as 2.3.0

Plugin Object Model

A plugin is defined in IdentityIQ by the Plugin XML object. This object defines the parameters of the plugin, such as REST Resources, Snippets, Widgets, and Settings. This Plugin object is defined in the manifest.xml file. The Plugin Object is an XML object that defines the features of the plugin. This object tells IdentityIQ what features are in your plugin by defining them as attributes of a Plugin Object. In the Plugin Object you also define items such as the name of the plugin, the rights required for using the plugin, version, snippets, and REST resources.

These attributes are included in the plugin model:

Attribute Name	Description
name	Unique Name of the plugin
installDate	Date that plugin is installed
displayName	Display Name for the plugin
disabled	Status of the plugin
rightRequired	What SPRIGHT is required for this plugin
version	The version of the plugin
minSystemVersion	The minimum version of IdentityIQ that the plugin will run on
maxSystemVersion	The maximum version of IdentityIQ that the plugin will run on
attributes	List of configurable attributes
file	Reference to the persisted file in the database

Plugin Structure

A plugin can include the following components:

- Manifest file
- Build file(s)
- Database Scripts
- UI Elements
- XML Artifacts
- Java Classes
- Java JAR libraries

Not all of these components are required for a plugin - it can be as basic as the manifest, and some javascript/xhtml pages. In order to understand how a plugin operates, and how best to create one, it is important to understand what each of these components does, and how they interact.

Plugin Manifest File

A plugin is defined in IdentityIQ by the Plugin XML object that defines the parameters of the plugin. For example, features such as REST resources, Snippets, Settings. The Plugin object is defined in the `manifest.xml` file. This is a required artifact.

For more complex plugins that require support for other field types, and more dynamic behavior, such as drop down lists or password fields, use the advanced plugin settings to define a form or reference a custom plugin configuration file.

Dynamic behavior might include showing or hiding additional fields depending on previous selections. For example, if a user chooses basic authentication, a username and password field would appear, but, if oauth authentication is chosen, it might be more appropriate to show an access token field.

Plugin Settings

> If your plugin requires more than simple input fields, string, boolean or int values, you must use the plugin advanced settings.

Plugin settings are attributes that are available for modification as part of the installation. Click **Configure** to display the configuration settings page. Settings are displayed as a form. If the plugin does not use the plugin advanced settings, the form is created automatically.

Settings from the manifest file are listed, in order, on the plugin settings page.

The Plugin setting object can be used to represent a single setting on the configuration settings page for a Plugin. Each object is used to represent a single configurable setting on the settings page.

Attribute Name	Description
allowedValues	List of allowed values for population of a dropdown
dataType	The type of the setting, for example, "string" or "int" or "boolean"
defaultValue	The default value for the setting
helpText	Associated help text for the setting
label	Label to be displayed for the setting
name	Name of the current setting
value	Value for the setting

Plugin Advanced Settings

Plugin advanced settings are used to define forms or reference a custom plugin configuration file to define more complex plugins.

- settingsForm — define a plugin using a form
- settingsPage — reference a custom HTML/JS file

settingsForm

To use a form for plugin configuration, you can build a form using the form builder then copy and paste that form into the manifest file, or you can build the form directly into the manifest.

Values entered into a form can be accessed using the FormData.values. FormService has functions to assist with validating required fields and displaying errors. Additional validations are built into the HTML and AngularJS code based on the form design, which means Angular will set a field to undefined if it is not valid. These validations can be used to prevent a form from being submitted and show error messages if necessary.

settingsPage

Develop your own configuration settings page by providing the required HTML and javascript. You can use whatever frameworks you prefer for your settings, but they need to fit in with whatever IdentityIQ has loaded. For example, angular is not required, but you can use it.

To use angular frameworks, see SailPoint Angular Components.

Use the settingsPage setting to specify the name of your custom configuration settings page, for example, `config.xhtml`.

Snippets

Snippets are small, configurable snippets of code that can be injected into the rendering of normal IdentityIQ user interface pages. For example, you can insert a menu option, a button, or even a larger set of interface components into an IdentityIQ page.

Snippets must be specified in the plugin's manifest file. They use a regular expression pattern to identify the IdentityIQ pages where the snippet should execute, and therefore appear.

You can have multiple snippet components in the same plugin, some that apply globally, and some that apply to specific targeted pages. You need to define a separate `js` file for each location a snippet applies, and then specify a separate snippet in the manifest file with the right regexPattern to run it on the appropriate pages.

The details for the user interface component's contents, and its placement within the page, are specified in the JavaScript file.

A snippet contains four equally important components:

Component Name	Description
regexPattern	This is a regular expression pattern that is run against the current URL in the browser - if the URL matches the pattern, the Snippet will attempt to displayed
rightRequired	This determines the scope of users allowed to view the Snippet element - should reference an IdentityIQ SPRight object
scripts	This is a list of the scripts to run when a particular URL matches the regexPattern. Normally this will consist of injecting an element into the DOM of the page. The example `hearder.js` file uses JQuery
styleSheets	List of any css files that are required by Snippet Scripts

Widgets

A Widget is a targeted snippet – one that inserts a block of user interface components into a fixed area of the Home page that can be added selectively for different users, as a unit.

Widgets can be configured to appear on the Home page for any or all IdentityIQ users.

The first thing you need, to implement a plugin Widget, is the Widget object itself. When you import that object into IdentityIQ during plugin installation, it defines the existence of the Widget making it available for any user.

Widget objects are simple, the only details about the user interface component that get defined in the object are its name and title.

Widgets require a snippet definition in the manifest file for this plugin. This snippet defines the home page hook for the widget. The regular expression pattern for the widget snippet must specify the IdentityIQ home page.

The rest of the snippet definition has IdentityIQ execute the contents of the specified JavaScript file when it loads pages that meet the regex pattern.

The contents of the JavaScript file then define both the user interface layout, in the form of a directive, and the controller for the Widget. Because the home page is an Angular page, this JavaScript must specify an Angular controller for the widget.

The name given to this directive must follow a fixed naming convention. It must be specified in relation to the name given to the widget object. Specifically, the widget object name must be prefixed with sp and suffixed with Widget. So the Search widget object requires a directive called sp*WidgetName*Widget.

The directive references the controller for the widget. That controller is also defined within the widget's JavaScript file and defines the variables that serve as the model for the view elements and performs the required operations to set their data values.

Plugin Build File

Apache Ant is a readily available tool that can be used to package plugins prior to deployment and distribution. To provide build specific values, the standard is to also include a `build.properties` file with a simple key-value pair for all build specific tokens.

The following example illustrates how a properties file can be leveraged to enable multiple developers to use the same build process, despite having dissimilar build environments. The actual `build.xml` file is responsible for creating the build directory, compiling any Java classes, packaging those compiled classes into a .jar archive, and archiving in .zip format the complete plugin.

```
jdk.home.1.7=/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk
   iiq.home=/usr/local/apache-tomcat-8.0.30/webapps/identityiq/
   pluginName=TodoPlugin
   version=2.0.0
```

Note that complications can arise when the plugin is built using a different version of Java than the version deployed on the application servers hosting IdentityIQ. Parameterize the `javac` argument in the `build.xml` file with the most compatible Java version available. To do this, add the property target to the `javac` directive, and set equal to whatever version is being targeted.

For example:

```
<javac srcdir="${pluginSrc}" destdir="${pluginClasses}"
 includeantruntime="false" target="1.7">
```

Plugin Database Scripts

Plugins that require persistence of data outside of that allowed by the IdentityIQ object model require at minimum the creation, updating, and deletion of unique tablespace. The plugin framework creates a database named `identityiqPlugin`. The creation of this database is handled by the installation scripts packaged with every download of IdentityIQ, in the `WEB-INF/database` folder. Additionally, a default user identityiqPlugin is created to perform operations, installation and deletion of plugins, on this new database. Similar to the base IdentityIQ username and password, these can be modified and updated in the IdentityIQ `iiq.properties` file located in WEB-INF/classes/iiq.properties.

When creating a plugin, you must create a folder named `db` in your project directory. This folder should be further subdivided into three operation specific folders: `install`, `uninstall`, and upgrade.

The scripts placed in these folders are automatically run when a plugin is installed or deleted. It is recommended that you include scripts for the four major database types supported by IdentityIQ, MySQL, SQLServer, DB2, and Oracle, or note in your documentation which databases are supported. Database specific scripts must include the database type as the file extension, for example `.mysql`. The `upgrade` folder should contain any deltas in table definitions from prior versions of the plugin.

Plugin User Interface Elements

Most plugins have some additional user interface component that appears in IdentityIQ. Images, CSS files, HTML templates, and JavaScript can all be used to provide the interactions and views required by the plugin. Plugins that use a fullPage element look for a file called `page.xhtml` in the build.

To extend the classes loaded with your plugin to the rest of IdentityIQ, you must specifically declare those classes in the manifest file.

Plugin Authorization

To prevent unauthorized access to your new endpoints, each should be guarded with an authorization mechanism. You can constrain which users can see and access the user interface components, and you can secure the REST endpoints you build into your plugin.

When you define snippets, including widget plugin components, in the manifest file for a plugin, you can specify a rightRequired attribute to constrain access. This attribute names a SailPoint SPRight which users must be assigned for the component to appear in their IdentityIQ instance.

You can also specify a rightRequired at the Plugin object level, in the manifest file, which will specify the required SPRight for a user to be able to access the full-page component of the plugin.

If you leave these rightRequired attributes off, all IdentityIQ users will be able to access those plugin components.

SPRights are the most granular permission object in IdentityIQ. In most cases, users are assigned SPRights in IdentityIQ by attaching those rights to one or more Capability objects and then granting the Capability to the appropriate users.

If the plugin contains a full-page component that users can access through a quicklink, the Quicklink access will be governed by Quicklink Populations, like any other IdentityIQ Quicklink.

User must also be authorized to the full page itself, with the rightRequired specified in the plugin manifest, to be able to view the page.

Other authorization points of note. First, whether or not you explicitly authorize system administrators to these components, they will have full visibility and access to them. Second, when you include a widget in your plugin, the widget will appear in the list of available widgets for all users when they are editing their home page and deciding which content to include there. However, if they are not authorized to the widget's snippet, and they add that widget to their Home page, IdentityIQ will add an empty widget and they will neither be able to see nor interact with any of its functional elements.

To secure the endpoints the plugin framework use Annotations. In Java, an annotation is a syntactic metadata that is added, often before a method signature, that describes the parameters used in that method.

An annotation should have at least three parts

- The HTTP method (GET, POST, PUT, DELETE, etc)
- The path or endpoint - this can be parametrized which is useful for pulling back a single record. The above example uses parameterization by adding the variable within {} tags to the end of the URL, and also declaring the @PathParam appName in the input arguments of the method signature
- The authorization of the method - the allowed values are:
    - **@AllowAll** - this allows anyone to interrogate the endpoint
    - **@RequiredRight("<SPRight>")** - allows users who possess the named SPRight to access the endpoint

- **@SystemAdmin** - system administrator access only
- **@Deferred** - Authorization is deferred to the method. When this option is selected, you must also create an Authorizer class that implements the sailpoint.authorization.Authorizer interface. The Authorizer class should overwrite the authorize(UserContext) method of the base Authorizer interface. Inside of the REST resource method, the author would then call authorize().

Plugin XML Artifacts

Any IdentityIQ objects that are required as part of a plugin need to be represented in XML artifacts. This could be something as small as a single new SPRight object or a complex workflow or rule. The mechanism that is used for importing these artifacts during installation is the same as any IdentityIQ object import, so the normal import actions are also available, merge, include, execute, logConfig.

Development of these XML artifacts can be done directly in the build folder, or in the IdentityIQ user interface and either exported using the console or copy and pasted from debug into the build.

When developing in the user interface and then migrating to your build folder using cut and paste, you must remove the id attribute assigned by Hibernate and any other hibernate ID value references. For this reason, it is preferable to export the artifacts using the IdentityIQ console command **./iiq export -clean**.

Everything in the `import` folder is imported The objects can be separated into individual files, or combined into a single file. When a plugin is uninstalled, the XML artifacts that were imported remain in the IdentityIQ database, but the `.zip` archive from which the plugin files was loaded, is removed from the `spt_file_bucket` and `spt_persisted_file` tables.

Plugin Java Classes

Plugins are a powerful productivity-enabler, that give users the ability to extend both the IdentityIQ user interface and server in a well-defined manner.

Plugin Java Classes - REST Classes

The plugin framework relies heavily on REST web services integration for the majority of CRUD (create, read, update, and delete) operations. To create a custom REST Resource:

- Extend the BasePluginResource class
- Secure the new endpoints

Extend BasePluginResource

The BasePluginResource class should be used as the base class for all resources. It provides access to utility methods for accessing plugin settings, getting database connections, and more.

- **getConnection** - gets connection to the datasource specified in the iiq.properties file for the plugins
- **getPluginName** - this method should be overwritten to return the correct name of the plugin
- **getSettingBool** - gets value of boolean plugin setting for plugin name returned by getPluginName()
- **getSettingInt** - gets value of int plugin setting for plugin name returned by getPluginName()
- **getSettingString** - gets value of String plugin setting for plugin name returned by getPluginName()

- **prepareStatement** - convenient security method for getting Java PreparedStatement object for any database queries that are required
    - signature is `prepareStatement(Connection, String, Object...)` where the String would be the SQL statement you want to execute
    - Object... a list of the parameters values, if any, to be used
- **authorize** - should be overwritten by implementers, but by default only ensures that SystemAdministrator can see everything

Introduce additional methods to handle the various endpoints required by the plugin.

Plugin Java Classes - Plugin Executors

The plugin framework enables you to include custom task implementations or services with your plugin. These items rely on executor classes that contain the business logic for these services. The following executors are currently available:

- Service Executors
- Task Executors
- Policy Executors

You must specifically declare the classes to be exported for each of the executors. Only classes specifically declared are accessible from the rest of IdentityIQ. If a class is not declared, it will fail to instantiate when you open the plugin. Classes are declared in the manifest.xml file.

Use the following attributes to declare classes:

- serviceExecutor
- taskExecutor
- policyExecutor

For example:

```
<entry key="taskExecutors">
  <value>
    <List>
      <String>com.acme.TaskExecutor1</String>
      <String>com.acme.TaskExecutor2</String>
    </List>
  </value>
</entry>
```

Plugin Object Properties

When defining your plugin object you must provide the list of service executors that are included. The list lives inside an attributes map under the key serviceExecutors.

- Plugin Helper methods
- All inherited Service methods

- BasePluginTaskExecutor
- Plugin Helper methods
- All inherited TaskExecutor methods
- BasePluginPolicyExecutor
- Plugin Helper methods
- All inherited PolicyExecutor methods.

Plugin Helper Methods

The list of methods that are included with the BasePlugin classes are as follows:

- getPluginName() - returns a string value of the name of the plugin
- getConnection() - returns a Connection object used to query the database
- getSettingString(String settingName) - returns a String setting value from the Plugin Settings
- getSettingBool(String settingName) - returns a boolean value from the Plugin Settings
- getSettingInt(String settingName) - returns a integer value from the Plugin Settings

You can think of the BasePlugin classes as foundation for creating your specific objects. By using them you gain access to the Plugin Helper Methods, but you are not required to use the BasePlugin classes. You can extend directly from the parent class object.

Implementing a Plugin Service Definition

To implement a Plugin Service there are two parts. The service class, containing the business logic that you want the service to actually do, and the service definition xml, that is loaded into IdentityIQ.

- BasePluginService Class — an abstract class that extends the Service class, as well as implements the PluginContext interface. You can use this class as the foundation for your custom Plugin Service
- Service Definition — specify a pluginName attribute. This tells IdentityIQ to use the plugin class loader for this executor. If this attribute is not specified the executor class will not be found

Implementing a Plugin Task Executor

To implement a Plugin Task Executor there are have two parts. The Task Executor class, which handles the business logic for your task, and the TaskDefinition xml object, which gets loaded into IdentityIQ.

- BasePluginTaskExecutor Class — an abstract class that extends the AbstractTaskExecutor class, as well as implements the PluginContext interface. You can use this class as the foundation for your custom Plugin Executor task
- TaskDefinition — include the pluginName attribute, as this attribute tells IdentityIQ to use the plugin class loader instead of default class loader. If the attribute is not specified the executor class will not be found

Implementing a Policy Executor

To implement a Policy Executor there are have two parts. The Policy Executor class, which handles the business logic for your policy, and the Policy xml object, which gets loaded into IdentityIQ.

- BasePluginPolicyExecutor Class — an abstract class that extends the AbstractPolicyExecutor class, as well as implements the PluginContext interface. You can use this class as the foundation for your custom Plugin Policy Executor
- Policy XML — include the pluginName attribute, as this attribute tells IdentityIQ to use the plugin class loader instead of default class loader. If the attribute is not specified the executor class will not be found.

Plugin Java Classes - Script Classes

> BeanShell executions are referred to as scripting in this document.

The classes installed for plugins can be made available to all BeanShell executions. BeanShell (rules, scriptlets, workflows) scripting invocations are able to use all Java classes, from all plugins, that are declared as exported for scripting. The scripts should fail to load classes which are in plugins, but which are not explicitly exported.

BeanShell executions include:

- rules
- workflow steps (rules and scripts)
- scriptlets

Use the scriptPackages attribute to declare the Java packages exported for use by scripts as follows:

```
<entry key="scriptPackages">
  <value>
    <List>
      <String>com.acme.classy.util</String>
    </List>
  </value>
</entry>
```

Example: Using Plugin Classes From a Rule

This is a simple example of how to call the plugin classes from a rule.

The example Java class named com.acme.classy.util.ClassyUtil can be used from a rule if declared properly in the manifest.xml using the scriptPackages entry shown above.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Rule language="beanshell" name="ClassyRule" >
  <Description>Returns the current timestamp by calling class from
                      ClassyPlugin</Description>
  <Signature returnType="string"/>
  <Source>
import com.acme.classy.util.ClassyUtil;
long now = ClassyUtil.now();
return "The current timestamp is: " + now;
 </Source>
</Rule>
```

SailPoint Angular Components

To implement the SailPoint-styled angular components, your project needs to include the SailPointBundleLibrary JavaScript file. There are specific directive dependencies on this library when you use the SailPoint-styled components.

Widgets and snippets donot require this library in the plugin project, because for those, the plugin architecture automatically references this library from your IdentityIQ instance. But if you are going to use these in a full page plugin implementation, your plugin needs to include a copy of this JavaScript library, copied from your IdentityIQ version. Plus, your `page.xhtml` component needs to explicitly reference the JavaScript file in a script element so IdentityIQ can resolve the SailPoint Angular directives you are be using. This library needs to match the version of IdentityIQ where the plugin will be deployed.

Your angular module definition needs to specify any other modules that your module has dependencies on. This list will vary, depending on the elements you include. For example, if you are using modal boxes, you need to include `sailpoint.modal`, tree components rely on `sailpoint.tree`, and so on. If your user interface only contains some of the more basic of the field types, you need fewer modules in your dependency list.

An example plugin that demonstrates the different components that are part of the SailPoint Angular bundle, called the Kitchen Sink plugin is installed with IdentityIQ. The plugin is available for download from Compass. Once the plugin is downloaded, you can install and view the plugin through IdentityIQ to see how the different field types behave. Then you can examine the HTML and JavaScript files to see the directives involved and how they are populated.

The library contains a wide array of field types, just like you might see throughout core product pages. For example, selection lists, plain text entry fields or fields where you can enter multiple separate text values as a list, dropdown list boxes where you can select from a set of choices, and a special directive for when the choices should be Boolean true-false values, date pickers, checkboxes, and radio buttons.

The library offers SailPoint styled buttons, and the demo plugin shows how you can use them to attach logic to do things like open modal boxes of various types, post an alert notification to the page or navigate the user to another page in IdentityIQ while preserving navigation history.

Beyond the simpler field types and buttons, there are a few components that create more complex elements. For example, a tree directive for showing nested relationships in data, a directive for displaying a list of cards that is configured for paging when result sets are large, and a directive for a data table that matches the tables in the Angular pages of IdentityIQ.

Internationalization

Message catalog files are specified per language with the two character abbreviation for the corresponding language, or the four-character options when you have locale-specific message catalogs, for example, `TrainingPlugin_fr_ca.properties` for Canadian French. These files should be recorded in the messages folder of your plugin project.

To guard against collisions with IdentityIQ base product message key names, or message keys from other plugins, the best practice is to name your plugin's message keys with a prefix that makes them unique to your plugin. For example, consider using your plugin's name as a prefix.

Both the user interface and server side code can access the provided catalogs.

In the user interface, full page plugins and widget plugins use two different mechanisms, because of differences in library support that are included in the pages.

In a Full page plugin component, the HTML uses the `msgs` function to translate the text.

In Widgets or other snippets that display text, the `spTranslate` function in the SailPoint Bundle Library does the translation, by piping the message key through it.

It is possible to use this syntax in a full page plugin, but only if you have included the SailPoint Angular Bundle Library and have declared a dependency for your angular module on the `sailpoint.i18n module`. The `spTranslate` function is part of that module.

In server-side code, localization is done with the Message object's localize method, passing it the message catalog key to look up and translate.

Plugin Installation and Removal

To install a plugin, click the gear icon and select **Plugins** to navigate to the Installed Plugins page. Click **New** and drag and drop or upload the plugin `.zip` file.

If you downloaded a plugin, the `.zip` file should be included with the download. If you developed the plugin yourself, the `.zip` file is in your project directory under `build/`*`your plugin name`*`/dist`.

When a plugin is installed, the database scripts from the `db/install` folder run, which creates any tables necessary for the plugin, the XML configuration files are imported into the IdentityIQ database from the `import/install` folder, any compiled classes are loaded into the unique plugin classloader, and the manifest file is imported creating the Plugin object.

Remove a plugin by clicking **X** on the appropriate Plugin card on the Installed Plugins page. Database scripts in charge of cleaning up data run from the `db/uninstall` folder and the manifest file (the Plugin object) is removed.

Additional steps might need to be taken to edit the System Configuration file to remove objects created by the plugin, such as quicklinks, or to remove tables created in the plugin database.