

# Plug-ins

Augur Systems includes a modular plug-in system in its software products. These plug-ins allow you to customize and expand the product's functionality. Some general plug-ins are included with the product; some may be licensed separately from Augur Systems or third parties, and some you may create yourself. This document describes the plug-in system from the perspective of a developer.

## Interface

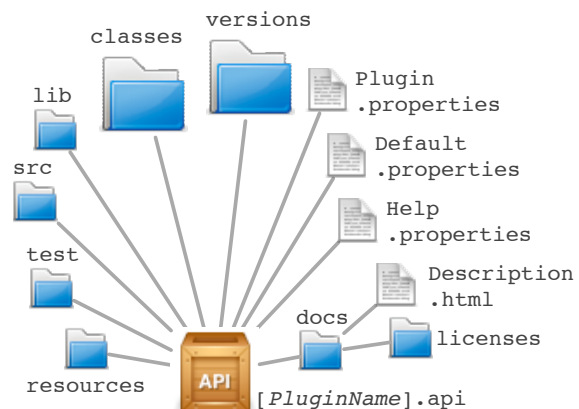
At the heart of a plug-in is the Java code that implements a product's API. That API will be in the form of a Java *interface class* that extends the base interface: `iai.plugins.Plugin`. Each API defined by an Augur Systems product will usually include one or sometimes a few methods to be implemented by a plug-in. Each API is considered a unique *type* of plug-in. Multiple plug-in types may be supported in one product. For example, a product may support one type of plug-in for processing data, and another type that is used to extend a GUI.

In usage, the Augur Systems product considers your plug-in a *class*. The product constructs one or more *instances* of your plug-in class, and then initializes each one with a unique set of `key=value` properties, previously configured by the end-user. Your plug-in code can access these properties. The set of *keys* include both common properties required by the type of plug-in (e.g. the instance name, a log setting, etc.), and special properties that you want to expose to the end-user for configuration. The end-user configures the *values*, with the help of descriptions and defaults that you define. For example, if your plug-in connects to a database, you may want to expose the database host and port as two properties for the user to define.

## Package

A plug-in is packaged as a specially constructed JAR archive file. (A jar is just a ZIP archive that includes a "META-INF" directory for information like meta-data, code signatures, etc.) The package is assigned the special extension ".api" (a convenient acronym for both Augur Plug-In and Application Programmer's Interface). The archive contains the directory structure shown here. The META-INF directory is omitted since it is not explicitly used by plug-ins.

A "Skeleton" plug-in is provided for each plug-in type. It contains this basic folder and file structure, with embedded instructions and a no-op implementation. You can copy this Skeleton plug-in as the starting point for any new plug-ins you develop.



## Folders

The archive contains two mandatory folders, and some optional folders. You can add more folders to augment the current suggested list of optional folders, but try to use these standard suggestions if possible. The plug-in libraries provide tools that enable your plug-in code retrieve any files within the plug-in package.

<b>src</b>	<p>This folder should contain the Java source code files that implement your plug-in. You'll need to use this folder for development, since the build script looks for this directory to compile your plug-in, but it is optional to include this folder in the plug-in package you provide to the end-user. If your implementation is proprietary, you may not wish to distribute the source. The build script (described later) offers both options for you.</p> <p>If your plug-in uses a Java <i>package</i> structure, you can create the necessary subdirectories. The <code>src</code> directory is the code-base (root) for Java package hierarchies.</p>
<b>classes</b>	<p>This folder will contain your compiled Java classes, distributed in subfolders if you used Java packages. This folder is included in the <i>classpath</i> for the plug-in when it is executed in the Augur Systems product. The build script will automatically create this folder when compiling.</p>
<b>lib</b>	<p>This optional folder can contain any required JAR or ZIP libraries. All such files in this folder (or subfolders) are included in the <i>classpath</i> for the plug-in when it is executed in the Augur Systems product. Although your plug-in will also inherit the application's classpath, you should try to keep your plug-in independent of the application, so that future changes to the application's libraries do not cause exceptions (e.g. <code>ClassNotFoundException</code>) or versioning problems. That means that sometimes you might include your own copy of a library that the application already provides.</p>
<b>test</b>	<p>This optional folder can contain any development testing tools. If you opt to ship the <code>src</code> folder, then you should consider including a test environment too, to assist other developers. The build script will include <code>test</code> if you choose the build target that includes <code>src</code>.</p>
<b>versions</b>	<p>This required folder contains text files that document the version history of your plug-in. This history is viewable to the end-user via display options in the Augur Systems product. For each new version you release, you should add a <code>[version].properties</code> summary file, and optionally a <code>[version].html</code> file for a more verbose description. The file names need to follow strict formatting rules, described later in the Versions section.</p>
<b>resources</b>	<p>This optional folder is the recommended place for you to store static information, such as images, serialized data, etc.</p>
<b>docs</b>	<p>This folder contains the <code>Description.html</code> file (described in the next section), the <code>license</code> folder (described below), and any arbitrary files you want to include for developers. End-users will have an interface to read the description and licenses, if any.</p>
<b>licenses</b>	<p>This subfolder of the <code>docs</code> folder should contain any license contracts for your plug-in, including copies of third-party licenses, e.g. open-source LGPL, etc. The files should be formatted as plain text. End-users will have an interface to read the license files, if any.</p>

## Documents

The archive contains four text documents that you should edit:

**Description.html** Use this HTML-formatted file for the general documentation of your plug-in. For example, you might describe architecture, usage, deployment, warnings, and any other general information that may be useful to the end-user. Don't include any version information, since that is defined in the `versions` folder. You should also avoid detailing which API methods your plug-in implements, the plug-in's main class name, or the plug-in type, since that information is automatically calculated and displayed to the user as appropriate.

Although HTML is the recommended format, there are some command-line tools that will display this file as plain text, after stripping all HTML tags. So try to format the content for this dual purpose. In general, when you intend a line break (e.g. `<p>` or `<br>`), make sure you actually skip a line too, so that the plain text version will look good too.

---

**Default.properties** For most plug-ins, you'll want to define some user-configurable properties. Use this `key=value` properties file to define those configuration keys, and any default values. The Augur Systems product provides a means for a user to configure the final version of these properties; one set for each instance.

You must append an equals character after each `key`, even if it does not have a default value.

If you set the value to the special string `"*secret*"`, then configuration tools will encrypt the user-entered data. To retrieve the secret value during run-time, your plug-in should use the `getSecret()` method (on the instance's `PropertiesParser` configuration object). This method will retrieve the decrypted value of the property. This encryption feature ensures that the data for secret properties are not human-readable where they are stored on the server.

---

**Help.properties** This properties file is a mirror of the keys in the `Default.properties` file, except for each value you enter a short description of the property. This description will be displayed to the end-user when they configure an instance of your plug-in.

---

**Plugin.properties** This properties file defines the plug-in's entry point for the Augur Systems product. It contains the fully-qualified name of the class in your plug-in that implements the interface type.

## Versions

In the `versions` folder, each file's name is a formatted version number, with the extension `".properties"` or `".html"`. The version number's format is:

```
<major>.<minor>.<maintenance>.[a|b].[iteration]
```

The last two fields ("alpha/beta", and the "iteration") are optional.

Example version numbers:

- 1.1.0
- 1.1.2.b
- 1.1.2.b.2

Example properties file names:

- 1.1.0.properties
- 1.1.2.b.properties
- 1.1.2.b.2.properties

The properties file's name format is "`<version>.properties`". It contains two `key=value` pairs. The **date** key's value is the date/time of the version release. The date should be Unix time (the number of milliseconds since 1970). The build script will automatically update the date for the most recent version file. If you need to set the date manually, you can use the Unix command `'date +%s'` to get the current time in seconds; remember to append "000" to convert to milliseconds.

The **summary** key's value is a short (one-line) description of the version. This should be a description of the version, not the plug-in in general.

The name of the HTML file is "`<version>.html`". The HTML file is optional for each version. If you include one, you can describe new features, bugs fixed, etc. You can use any HTML tags to format the text. Note that relative links (e.g. hypertext and images) will not be able to access resources within the package, so we recommend you stick to formatting text and layout. You can use absolute links to web servers, if necessary.

## Info

From the command line you can get information about installed plug-ins. You can execute the following command to see the plug-in's name, version, and description:

```
java -jar [PluginName].api
```

Note that this command is dependent on some application libraries, so the plug-in must be located in the proper deployment directory. Also, the command must be executed from the directory where the plug-in file is located, so you should `'cd'` there first.

## ClassLoader

Augur Systems developed a custom ClassLoader to access plug-in packages. It can resolve classes from both within the plug-in package and from the application's classpath. It also has special methods for accessing data from within the plug-in package.

Note that classes from the application's classpath are actually resolved by the default ClassLoader, as delegated from the plug-in ClassLoader.

# Development

The development area for a plug-in is an unrolling of the plug-in's structure, except all the files and folders are in a development subdirectory of the application with the plug-in's name. For example, to develop a connector named "MyPlugin", you need to create the directory "[home]/plugins/connectors/MyPlugin/". In that directory you would create your "src" subdirectory, and the rest of the files and folders. As noted earlier, you can copy the Skeleton plug-in to create base versions of all the plug-in files and folders.

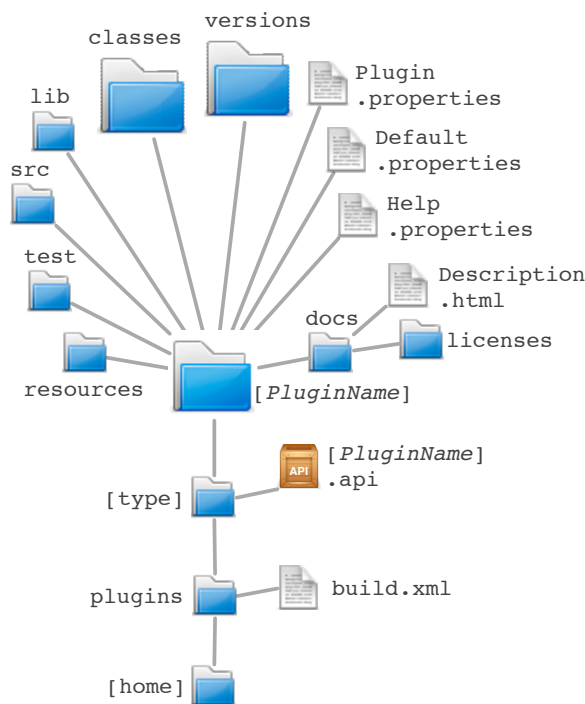
## Build

An Ant build script is included at "[home]/plugins/build.xml". You can edit this script to include your plug-in during the build. Then you can execute 'ant build' from a shell or Ant GUI.

Note that there are two target options regarding the source code. One build target will include the `src` and `test` folders in the package; the other won't. See the example below for instructions.

The 'javac' task is configured to compile only the class defined in "Plugin.properties". Usually, any dependencies will also compile as necessary. If you reference a class by reflection, some dependencies may have to be explicitly compiled in a custom target.

The resulting plug-in package will be written to "[home]/plugins/[type]/[PluginName].api", as shown in the diagram.



## Example

If you are developing a type of Connector plug-in, you can edit the **build:connectors** ant target. Copy/paste one of the lines that set the plug-in name variable. See the example below:

```
<var name="pluginName" value="ServerTcp"><antcall target="-build:plugin+src"/>
```

In this example, you would replace **ServerTcp** with the name of your plug-in. Remember, the plug-in name must be the same as the directory name where it resides.

If you do not want the final package to include the `src` directory, then change the target in the line above to the following. Notice the '+' character was changed to '-', near the end of the line:

```
<var name="pluginName" value="ServerTcp"><antcall target="-build:plugin-src"/>
```

For more information about Ant, and its build script language, see the online documentation here:

<http://ant.apache.org/>

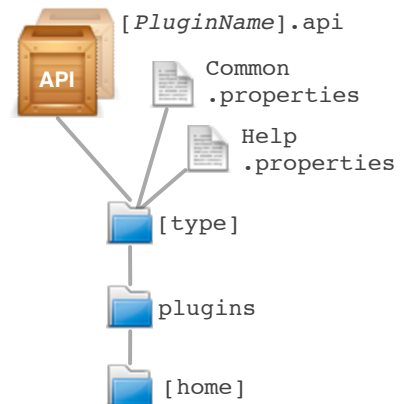
# Socket Station

Socket Station supports one type of plug-in, for channel connections. The job of this plug-in type is to provide a stream of data, usually by connecting to a remote source. This plug-in type is called a *connector*.

## Deployment

Your plug-in is deployed as its jarred-up “[PluginName].api” package file. To install the plug-in, you or the end-user will install the plug-in file in the Socket Station’s plug-in deployment folder. The home directory has a `plugins` folder, and then a subfolder corresponding to the plug-in *type* (e.g. `connectors`). You should place your plug-in that folder. You’ll need to stop/start Socket Station to use the plug-in.

In that same folder you may find two files: `Common.properties` and `Help.properties`. These files define properties common to all plug-ins of that type. Sometimes the end-user administrator may make changes (e.g. edit the defaults) to simplify their future configuration work. Usually, those properties are related to how Socket Station uses plug-ins of that type in general, and are not important to your plug-in implementation. For example, some properties may control how Socket Station logs data collected from your plug-in.



## Configuration

Each connector instance is configured as part of a Socket Station channel properties file. The ‘~/bin/configure’ utility provides an interactive command line for creating or editing channels. The utility will prompt for property definitions.

# Augur

Since version 4, Augur supports two types of plug-ins. One is actually the Socket Station *connector* type, as described above. Augur uses this connector type for connecting to event sources.

The other plug-in type is called a *handler*. This type has several optional methods that integrate with Augur as call-out points. The job of this plug-in is to integrate Augur with external systems (e.g. databases, reporting engines, etc.) and to extend Augur's normal event processing capabilities.

## Deployment

Your plug-in is deployed as its jarred-up “[PluginName].api” package file. To install the plug-in, the end-user administrator needs to load the package file into the Configuration Editor GUI, and then paste the plug-in into the appropriate folder in the configuration tree. After committing the changes, the plug-in will be available for use, without any reset required.

## Configuration

For *connector* types, the end-user administrator can then reference the plug-in when configuring a gateway. The plug-in's properties (as defined in the `Default.properties` and `Help.properties` files) will appear in a GUI interface for configuration.

For *handler* types, the end-user administrator uses the GUI to create instances, and then reference those instances to handler call-out folders in the gateways, rule trees, or rule tree nodes. Each instance is saved as a separate node in the configuration tree, with an editor to configure its properties.

## Info

Augur's Configuration Editor GUI provides extensive information about each installed plug-in, by clicking on the plug-in node in the configuration tree.