Plugin tutorial

This document, together with the hello-world plugin, shows you how to get started with the plugin development.

Important!

Please first read the "before writing a plugin" page.

There are a **lot** of existing Jenkins plugins, and a lot of people with Jenkins plugin development experience that you can take advantage of!

Check whether there is an existing plugin you can already use, or contribute to, then get in touch via the jenkinsci-dev mailing list, or IRC and explain your plugin idea.
Either we can point you towards an existing plugin, useful APIs, or code snippets that would be helpful.

Table of Contents

What Can Plugins Do?

Jenkins defines extensibility points, which are interfaces or abstract classes that model an aspect of a build system. Those interfaces define contracts of what need to be implemented, and Jenkins allows plugins to contribute those implementations. See this document for more about extension points.

In this document, we'll be implementing a Builder that says hello. (built-in builders include Ant, Maven, and shell script. Builders build a project.)

Setting Up Environment

To develop a plugin, you need Maven 3 (why?) and JDK 6.0 or later. If this is the first time you are using Maven, make sure Maven can download stuff over the internet.

> **Nexus Users**
>
> If you are using the Nexus Maven Repository Manager, you can ignore these instructions, and instead, click here for instructions on how to add Jenkins build prerequisites and the proper `settings.xml` entries.

It may be helpful to add the following to your `~/.m2/settings.xml` (Windows users will find them in `%USERPROFILE%\.m2\settings.xml`):

```
<settings>
  <pluginGroups>
    <pluginGroup>org.jenkins-ci.tools</pluginGroup>
  </pluginGroups>

  <profiles>
    <!-- Give access to Jenkins plugins -->
    <profile>
      <id>jenkins</id>
      <activation>
        <activeByDefault>true</activeByDefault> <!-- change this to false, if you don't like to have it on per
default -->
      </activation>
      <repositories>
        <repository>
          <id>repo.jenkins-ci.org</id>
          <url>https://repo.jenkins-ci.org/public/</url>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>repo.jenkins-ci.org</id>
          <url>https://repo.jenkins-ci.org/public/</url>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <mirrors>
    <mirror>
      <id>repo.jenkins-ci.org</id>
      <url>https://repo.jenkins-ci.org/public/</url>
      <mirrorOf>m.g.o-public</mirrorOf>
    </mirror>
  </mirrors>
</settings>
```

This will let you use short names for Jenkins Maven plugins (i.e. hpi:create instead of org.jenkins-ci.tools:maven-hpi-plugin:1.61:create), though this is unnecessary once you are already working in a plugin project (only useful for initial hpi:create).
Note that adding the Jenkins repositories in a profile like this is not really necessary since most (all?) plugins already define these repositories.
And the mirror declaration is probably unnecessary.

Creating a New Plugin

To start developing a new Jenkins plugin, use an IDE (below), or if you are more comfortable with Maven, run the following command:

```
$ mvn archetype:generate -Dfilter=io.jenkins.archetypes:empty-plugin
```

To create a plugin skeleton for Blue Ocean, please see the generator-blueocean-usain Yeoman generator.

This will ask you a few questions, like the groupId (the Maven jargon for the package name) and the artifactId (the Maven jargon for your project name), then create a skeleton plugin from which you can start with.

Make sure you can build this:

```
$ cd newly-created-directory
$ mvn package
```

Building a Plugin

To build a plugin, run `mvn install`. This will create the file `./target/pluginname.hpi` that you can deploy to Jenkins.

```
$ mvn install
```

Setting up a productive environment with your IDE

NetBeans

NetBeans users can use the IDE's Maven support to open the project directly.

As you navigate through the code, you can tell NetBeans to attach source code JAR files by clicking the "Attach" button that appears in the top of the main content window. This allows you to read the Jenkins core source code as you develop plugins. (Or just select *Download Sources* on the *Dependencies* node.)

You are advised to use the NetBeans plugin for Jenkins/Stapler development. This offers many Jenkins-specific features. Most visibly, create a new plugin using *New Project » Maven » Jenkins Plugin*, and use *Run roject* to test it.

IntelliJ IDEA

IntelliJ 7.0 (or later) users can load pom.xml directly from IDE, and you should see all the source code of libraries and Jenkins core all the way to the bottom. Consider installing IntelliJ IDEA plugin for Stapler to make the development easier.

Alternatively create a new Maven project using *Create from archetype* and *Add an Archetype.* Select the GroupId and ArtifactId as above and choose RELEASE as version. In the next screen select *io.jenkins.plugins* as groupID and choose an artifactId (Project name) and Version to your liking. This will automatically create a maven project based on the specified artifact (e.g. emtpy-plugin).

IntelliJ defaults to downloading sources and JavaDocs on demand. So, to see the source, you may need to click the `Download artifacts` button in the `Maven Projects` tab.

Eclipse

Use Eclipse Juno (4.2) or later for the best experience.

Eclipse versions between 4.5 and < 4.6.2 contain a bug that causes errors such as **"Only a type can be imported. hudson.model.Job resolves to a package"**.
If you encounter this error please upgrade to Eclipse Neon.2 (4.6.2) or higher (Bug 495598).

As Jenkins plugins are Maven projects, Eclipse users have two ways to load a Jenkins plugin project. One is to use m2e, which tries to make Eclipse understand Maven "natively", and the other is to use Maven Eclipse plugin, which makes Maven generate Eclipse project definitions. At the moment, unless you have some prior experience with m2e, we currently recommend plugin developers to go with the Maven Eclipse plugin.

Eclipse users can run the following Maven command to generate Eclipse project files (the custom outputDirectory parameter is used to work around the lack of JSR-269 annotation processor support in Eclipse:)

```
$ mvn -DdownloadSources=true -DdownloadJavadocs=true -DoutputDirectory=target/eclipse-classes -Declipse.
workspace=/path/to/workspace eclipse:eclipse eclipse:configure-workspace
```

Where /path/to/workspace is the path to your Eclipse workspace.

Once this command completes successfully, use "Import..." (under the File menu in Eclipse) and select "General" > "Existing Projects into Workspace".

Do not select "Existing Maven Projects", which takes you to the m2e route

See Jenkins plugin development with Eclipse for gotchas and other known Eclipse/Maven related issues with Jenkins plugin development.

See Eclipse alternative build setup for an alternative way of setting up the Eclipse build environment, that is a bit more technically involved than using maven, but can give faster build times.

Plugin Workspace Layout

The plugin workspace consists of the following major pieces, described in additional detail at Plugin Structure:

pom.xml

Maven uses the pom.xml file to build your Jenkins plugin. All Jenkins plugins should be based on the Plugin Parent POM:

```
<parent>
    <groupId>org.jenkins-ci.plugins</groupId>
    <artifactId>plugin</artifactId>
    <version>2.2</version>
</parent>
```

If the project is created using the provided archetype, everything is already set up. Up to Jenkins 1.645 the Plugin Parent POM was part of the main Jenkins project and the POM version was the baseline Jenkins version to be used for the plugin.

When using Parent POM version 2.2 or later, the baseline version is selection using the `jenkins.version` property, e.g.:

```
<properties>
    <jenkins.version>1.609.1</jenkins.version>
</properties>
```

src/main/java

Java source files of the plugin.

src/main/resources

Jelly/Groovy views of the plugin. See this document for more about it.

src/main/webapp

Static resources of the plugin, such as images and HTML files.

Source Code

PluginImpl approach

(deprecated in favor of the Extension points approach below)

A plugin's main entry point may be a PluginImpl class that extends from Plugin. Once Jenkins detects this plugin class (via its inheritance relationship from `Plugin`), it will create an instance, and invoke methods.

Extension points approach

A Plugin class is optional; a plugin may simply implement extension points, registering them with the @hudson.Extension annotation for automatic detection by Jenkins.

The bulk work in the plugin consists in implementing those extension points. See the sample source code for more information about how a `Builder` is implemented and what it does.

Here are a few things about extension:

- in general a plugin extension should extend an existing extension point (a class that implements ExtensionPoint), and define an inner static class extending the corresponding descriptor (a class extending hudson.model.Descriptor)
- the @Extension annotation must be placed on the inner descriptor class to let Jenkins know about this extension
- the descriptor handles the global configuration of the extension while the extension class itself handles the individual configuration of the extension. For instance in a plugin defining a class extending LabelAtomProperty, an object of this class is instantiated for each LabelAtom (provided that the plugin is activated by the user in the label's configuration page). If configuration parameters for each individual instance are required, they're handled via a config.jelly file stored in a resource package named after the extension class. When the configuration form is saved, Jenkins calls the extension constructor marked with the @org.kohsuke.stapler.DataBoundConstructor annotation, matching parameters by name

Debugging a Plugin

NetBeans 6.7+ users can just hit **Debug**. For all others, run the following command to launch Jenkins with your plugin:
Convenient:

```
mvnDebug hpi:run
```

Unix:

```
$ export MAVEN_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=8000,suspend=n"
$ mvn hpi:run
```

Windows:

```
> set MAVEN_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=8000,suspend=n
> mvn hpi:run
```

If you open `http://localhost:8080/jenkins` in your browser, you should see the Jenkins page running in Jetty. The `MAVEN_OPTS` portion launches this whole thing with the debugger port 8000, so you should be able to start a debug session to this port from your IDE.

Once this starts running, keep it running. Jetty will pick up all the changes automatically.

1. When you make changes to view files in `src/main/resources` or resource files in `src/main/webapp`, just hit F5 in your browser to see the changes.
2. When you change Java source files, compile them in your IDE (NetBeans 6.7+: **Debug > Apply Code Changes**) and Jetty should automatically redeploy Jenkins to pick up those changes. There is no need to run `mvn` at all.

MAVEN_OPTS can be used to specify all sorts of other JVM parameters, like `-Xmx`

Changing port

If you need to launch the Jenkins on a different port than 8080, set the port through the system property `jetty.port`.

```
$ mvn hpi:run -Djetty.port=8090
```

Setting context path

maven-hpi-plugin 1.65 or later (used by parent POM 1.401 or later) can set the context path by using a system property. On more recent versions of Jenkins the "/jenkins" prefix is added automatically.

```
$ mvn hpi:run -Dhpi.prefix=/jenkins
```

Changing code while debugging

Depending on what you change, you can see it in the running instance without restarting the whole Maven process:

- Views: Groovy/Jelly views are re-compiled every time a browser requests a page, so just reload a page in the browser and you'll see the changes. This is also true for help files you write.
- Java code: If you are debugging, JVM's hot swap feature can be used to reload code so long as you do not touch the method signature. (For example, from NetBeans use **Debug » Apply Code Changes**.) Beyond that, you can hit ENTER in the Maven process and it'll reload the Jenkins webapp, though generally it is better to stop the process and start again. See Developing with JRebel for how to get a JRebel license for OSS projects to improve this experience.
- POM: If you change POM, you'll have to stop and restart Maven to see the changes.

If you are using version 1.120 or later of `maven-hpi-plugin`, which would be the default when using version 2.16 or later of the plugin parent POM, you can use the same features to debug changes spanning multiple plugins, or even a custom build of Jenkins core. Just make sure you have `SNAPSHOT` dependencies set up between the associated Maven modules (`<scope>test</scope>` suffices), and that the upstream module(s) have been built (e.g., `mvn -DskipTests clean install`). Now `hpi:run` on the downstream plugin and you should be able to reload views defined in any of the linked modules.

Distributing a Plugin

To create a distribution image of your plugin, run the following Maven command:

```
$ mvn package
```

This should create `target/*.hpi` file. Other users can use Jenkins' web UI to upload this plugin to Jenkins (or place it in `$JENKINS_HOME/plugins`.)

Releasing a Plugin and Hosting a Plugin on jenkins-ci.org

If you got to this point, you should definitely consider hosting your plugin on jenkins-ci.org. Move on to this document for how to do that. This includes the instructions for releasing the plugin.

Using custom builds of plugins included in the Jenkins WAR

If you are building a patched version of one of the plugins in the Jenkins core, the deployment process is a bit different. This is because Jenkins will itself manage these plugins *unless you tell it not to*.

Deploying a custom build of a core plugin

1. Stop Jenkins
2. Copy the custom HPI file to $JENKINS_HOME/plugins
3. Remove the previously expanded plugin directory
4. Create an empty file called <plugin>.hpi.pinned - e.g. maven-plugin.hpi.pinned
5. Start Jenkins

Other tips

1. Consider running Maven like this `mvn -o ...` to avoid hitting repositories every time. This will make various operations considerably faster.
2. Subscribe to the users' alias from here so that we can get in touch with you.
3. When you bump up the version of Jenkins you depend on, make sure to run `mvn clean` once, in particular to delete `target/work` that Jetty uses. *Newer versions may just use work, not target/work.* Otherwise your Jetty may continue to pick up old left-over JAR files.
4. If you have a high latency network connection to the Maven repository, you might find it faster to first run the build once on a server near to the maven repository, then rsync the .m2/repository folder across to your local computer
5. The 'package' and 'install' targets will by default run many tests. You can add '-DskipTests' on the command-line to skip these (of course you should run the tests before committing any changes)
6. The 'compiler:compile' goal is faster than 'compile', because it is a goal, rather than a lifecycle event, and therefore avoids certain earlier goals, such as setting up resources

Other resources

Besides this tutorial, there are other tutorials and examples available on line:

- Making your plugin compatible with Pipeline
- Stephen Connolly's 7 part tutorial (Writing a Hudson plugin)
    - Part 1 - Preparation
    - Part 2 - Understanding m2 and freestyle projects
    - Part 3 - Subcontracting for Publisher and MavenReporter
    - Part 4 - Abstract Publishers and MavenReporters
    - Part 5 - Reporting
    - Part 5½ - Typos corrected
    - Part 6 - Parsing the results
    - Part 7 - Putting it all together
    - yet not finished: health reports
- Quick start guide in Japanese
- ~martino:/2011/10/27/The JenkinsPluginTotallySimpleGuide by martinO
- Implementing My First Jenkins Plugin: AnsiColor by Daniel Doubrovkine
- Tutorial: Create a Jenkins Plugin to integrate Jenkins and Nexus Repository by Marcel Birkner