Plugin Structure

When you use Maven to develop a Jenkins plugin, Maven does most of this using `hpi`.

A plugin is really just a jar file that follows a certain set of conventions, as described below:

```
foo.hpi
 +- META-INF
 |   +- MANIFEST.MF
 +- WEB-INF
 |   +- classes
 |   +- lib
 +- (static resources)
```

- A plugin needs to have an `.hpi` extension. The file name body ("foo" portion) of the file name is used as the "short name" of a plugin, and it uniquely distingusihes a plugin.
- As you can see, the structure is similar to a WAR file, but there's no `web.xml`.
- `MANIFEST.MF` needs to contain a few additional entries. More on this later.
- `WEB-INF/classes` can have class files that constitute plugins, Jelly view files, and Jelly tag libraries based on tag files. Alternatively, some or all of them can be packaged into a jar and placed into `WEB-INF/lib`.
- `WEB-INF/lib` can have *.jar files, and those are loaded and made available to a plugin `ClassLoader`, along with the contents of `WEB-INF/classes`.
- Static resources, such as images, HTML files, CSS stylesheets, JavaScript files, etc, can be placed at the top of an `.hpi` file (just like a WAR file, again.)

Use the `hpi:create` target to create this structure. Use `hpi:hpi` to create the `.hpi` file.

Manifest

`META-INF/MANIFEST.MF` can have all the normal entries, but it needs to contain two more entries for Jenkins in its main section.

Plugin-Class

This attribute must have the fully qualified class name of the class that derives from Plugin. Jenkins instantiates this instance to activate a plugin, and everything starts from there. Consequently, a plugin must have one Plugin-derived class. This is the Jenkins plugin version of the Main-Class attribute.

Long-Name

This optional attribute can have a human-readable one line description of the plugin. This is used as "the name" for users (whereas the short name is used as the name internally in Jenkins.) When this attribute is not present, the short name is used as the long name.

Plugin-Dependencies

This optional attribute can have a list of comma-separated plugin short names/versions that are required for this plugin to run. The classes and libraries of those plugins are made visible to this plugin's classloader, so that your plugin can rely on them.
This mechanism allows a plugin to define its own extensibility point, and have other plugins provide implemenations.

```
Plugin-Dependencies: module-name:version,module2-name:version
```

Classloader

Per default Jenkins loads every jar from `WEB-INF/lib`, along with the contents of `WEB-INF/classes` after the classes and libraries of the core. If you want to have your own libaries loaded before these (e.g. you want a newer version of velocity or an other library), you can configure your plugin to use a different classloader strategy by telling the hpi plugin in your pom.xml:

pluginFirstClassLoader

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jenkins-ci.tools</groupId>
      <artifactId>maven-hpi-plugin</artifactId>
      <configuration>
        <pluginFirstClassLoader>true</pluginFirstClassLoader>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Static Resources

Static resources inside an `.hpi` file will be made accessible at `${JENKINS_CONTEXT_PATH}/plugin/SHORTNAME/`. For example, if you have `abc/def.png` in `foo.hpi`, and if Jenkins is deployed on `http://localhost/jenkins/`, then the URL `http://localhost/jenkins/plugin/foo/abc/def.png` would display the PNG file.

Index.jelly for Your `Plugin`-derived Class

Your `Plugin` class (that you named in `Plugin-Class` manifest entry) should have `index.jelly` view file, which should render 1-2 paragraph worth of the detailed description of your plugin, perhaps with version numbers, link to the homepage, etc. This jelly script will be used in the plugin configuration page so that the user can learn more about a plugin.

Testing With Jenkins

You can test your plugin with a Jenkins instance to use it as a user would. You can also do this with the debugger using `mvnDebug`.

Debug Plugin Layout: .hpl

The `.hpi` format is primarily meant to be a distribution format. Just like no one debugs the web application by creating a war and deploying it, Jenkins provides another plugin layout called `.hpl` (for "Hudson plugin link"), which is targeted for plugin developers to improve productivity.

You can create and install the `.hpl` file into a Jenkins installation using `mvn hpi:hpl -DhudsonHome=/path/to/jenkinsInstall`

The hpl file can be placed in `$JENKINS_HOME/plugins` just like hpi files. But hpl file just contains a single line of text that points to a manifest file, like this:../path/to/your/plugin/workspace/manifest-debug.mf The file pointed by this is a manifest file. It has the same custom attributes as defined above for `META-INF/MANIFEST.MF`, but it defines a few more custom attributes that allow a plugin developer to specify various pieces of a plugin in different locations in a file system.

```
Plugin-Class: hudson.plugins.jwsdp_sqe.PluginImpl
Class-Path: ./build/classes ./views ./lib/reporter.jar
Long-Name: JWSDP SQE Test Result Plugin
Resource-Path" ./resources
```

For example, the above sample `manifest-debug.mf` states that the static resources of a plugin shall be loaded from the `./resources`, and class files from `./build/classes`, Jelly view files in `./views`, and a library jar file `reporter.jar` shall be made available to the plugin classloader. Absolute path names can be also used.

This mechanism allows a plugin developer to avoid assembly steps. Also, changes to static resources and Jelly views will be reflected instanteneously (provided that you set the system property `stapler.jelly.noCache` to true when you start the web container.

Starting Jenkins with your Plugin

You can test your plugin by starting Jenkins using `mvn hpi:run -DhudsonHome=/path/to/jenkinsInstall`. You can do this with a copy of the Jenkins code that you have checked out if you wish to debug using the source. The `jenkinsInstall` path is the `jenkins` directory that you checked out.