Hosting Plugins

Largely superseded by https://jenkins.io/doc/developer/publishing/requesting-hosting/

We encourage plugin developers to host their plugins in the Jenkins community repositories on GitHub.

This makes it easier for the community to help you, as well as to have the community benefit from the plugin. In this way, even when you move on to something else, the community can find someone else to pick up the work.

You can take a quick look at how we work for more information.

Contents

- Contents
- Prerequisites
- Request hosting
- Importing source code
- Hosting the plugin page
  - Using GitHub as a source of documentation
  - Using Jenkins Wiki as source of documentation
  - Referencing the documentation page from the project file
- Changelogs
- Adding Maintainer Information
- Continuous plugin builds
  - ci.jenkins.io
  - jenkins.ci.cloudbees.com (legacy)
- Request upload permissions
- Releasing to jenkins-ci.org
  - Blacklisting plugin in the Update Center
  - Releasing alpha/beta versions of plugins
- Working around common issues
- Frequently asked questions
- Help! My plugin is not showing up in the update center

Prerequisites

In order to host a plugin in the Jenkins Update Centre, meaning that it will be offered to users for install via the Plugin Manager in Jenkins, you must:

- Give your plugin a sensible ID which does not include `jenkins` or `plugin`
  - e.g. IDs like "`jenkins-snapchat-notification-plugin`" are redundant (we know it's a Jenkins plugin!); call it simply "`snapchat-notification`"
  - Similarly, "`notification-plugin-for-snapchat`" is redundant and overly-verbose; again just use "`snapchat-notification`"
    - This is the `<artifactId>` tag in `pom.xml` if you're using Maven
    - This is the `jenkinsPlugin.shortName` value in `build.gradle` if you're using Gradle
    - This is the `name` field in the `.pluginspec` file if you're using Ruby
  - Choose carefully, as you cannot change this ID after your make your first plugin release

- Give your plugin a descriptive but short name; this value will be shown to users in the Jenkins Update Centre
  This should have each main word capitalised, usually ending with "Plugin". For example, "Snapchat Notification Plugin"
  - This is the `<name>` tag in `pom.xml` if you're using Maven
  - This is the `jenkinsPlugin.displayName` value in `build.gradle` if you're using Gradle
  - This is the `display_name` in the `.pluginspec` file if you're using Ruby

- You are also encouraged to give your plugin a short one-line description; this will also be shown in the Update Centre.
  For example, "Allows users to send build notifications via Snapchat". Again, there's no need to state that it's a plugin for Jenkins, or other such redundant text.
  - This is the `<description>` tag in `pom.xml` if you're using Maven
  - This is the top-level `description` value in `build.gradle` if you're using Gradle
  - This is the `description` field in the `.pluginspec` file if you're using Ruby

- For Maven projects, set parent of groupId `org.jenkins-ci.plugins`
  - Artifact `org.jenkins-ci.plugins:plugin` can be used as a parent pom
  - The `groupId` for your project (outside of the parent declaration) should be set to `io.jenkins.plugins`, unless your company or group has their own `groupId` specifier

- Upload your plugin code to a repository on GitHub
  - This will be forked into the Jenkins organisation, which will become the canonical source repo

- Specify an open source license for your code (most plugins use MIT)
  - The Jenkins project does not host closed-source plugins
  - All of the dependencies of your plugin must also be open source-licensed
  - You should specify the license in the plugin metadata (e.g. `pom.xml`), but ideally also in a `LICENSE` file in the root of your repository

- Have a jenkins-ci.org account
    - This gives you access to upload your plugin, document your plugin on this wiki, and track issues in JIRA. It's also required to request hosting (see below).
- Have at least a small README explaining what the plugin does / is about. The idea is to help other developers and users quickly identify what the plugin does when finding the repo.
  Once forked, you will probably want to add a link to the plugin's wiki page too.

Request hosting

Having completed the prerequisites above, log in to JIRA, and file an issue in the HOSTING project. Provide all information requested there.

NOTE

Though we're not absolutely requiring it, we strive to encourage contributions to existing plugins instead of creating new ones whenever possible.
The goal is solely to try and make it clearer and easier for Jenkins users to find the plugin(s) they need (without having to choose between N [very] similar flavours).

So, before creating new plugins and requesting hosting, please carefully read the plugins list or/and your update center from Jenkins for possibly similar plugins.
And if you still decide to create a new one for some reasons, please provide those reasons in the HOSTING request. That will make the process quicker and easier for everybody.

When done with all that, we'll create your plugin repository under https://github.com/jenkinsci, and we'll also create a JIRA component for you. Look for a Github email inviting you to join the Jenkins organization. Once you accept the invitation, you will be able to push to your new repository.

Once you have a Jenkins repository, you can start making the following preparations for your first release.

Importing source code

In some cases you may get an empty repository, rather than a fork, in which case you can import your source code:

```
$ cd path/to/yourplugin
$ git init
$ git add pom.xml src
$ git commit -m "initial import of my plugin"
$ git remote add origin git@github.com:jenkinsci/MYPLUGINNAME.git
$ git push origin master
```

We also recommend you look around some other plugin's POM file (such as this) to trim off the redundant things.

For example, `<repositories>` and `<pluginRepositories>` can be removed. If you need help, contact the dev list.

Declare your repository in your POM

The location of your repository **must** be declared in the POM for plugins hosted in Github, by adding the following fragment:

```
<scm>
  <connection>scm:git:ssh://github.com/jenkinsci/MYPLUGINNAME.git</connection>
  <developerConnection>scm:git:ssh://git@github.com/jenkinsci/MYPLUGINNAME.git</developerConnection>
  <url>https://github.com/jenkinsci/MYPLUGINNAME</url>
</scm>
```

Hosting the plugin page

Plugin pages are hosted on https://plugins.jenkins.io/, these pages are being generated automatically using the metadata from the latest plugin release and an external documentation page. External documentation can be retrieved from GitHub or from Jenkins Wiki. It is possible to create no documentation page at all, but we recommend to create one to improve user experience for your plugin.

Using GitHub as a source of documentation

Since the introduction of GitHub documentation support in September 2019 (announcement), this is the **recommended way** to host documentation. It allows plugin maintainers to provide the same documentation from GitHub pages and the Jenkins plugin site, and at the same time it allows using the Documentation-as-Code techniques when the documentation is a part of the repository and hence all common practices can be applied (versioning, pull requests and reviews, creating documentation in parallel with features, editing docs from GitHub Web UI).

The plugin site can pull documentation from the root README pages and from other locations. For new plugins it is recommended to use README pages as a source of documentation, preferably in Markdown or Asciidoc format (TXT is also supported in such case).

How to publish the plugin documentation from GitHub?

- Create a README page and put the plugin documentation there
  - This page will become a landing page for https://plugins.jenkins.io/
  - More documentation pages can be introduced inside the repository and linked from the README, the plugin site will display both absolute and relative links
  - Images from pages will be displayed by the plugin site as well
- Modify your project URL to point to the GitHub repository, e.g. http://github.com/jenkinsci/your-plugin
  - See the guidelines for Maven and Gradle below
- Release the new plugin version

Documentation examples:

- https://plugins.jenkins.io/configuration-as-code
- https://plugins.jenkins.io/gradle
- https://plugins.jenkins.io/mailer

Using Jenkins Wiki as source of documentation

Even though for new plugins it is recommended to just use GitHub README files, the Jenkins Wiki can still be used if there are strong reasons to do so.

1. Visit the Plugins page, hover over the "Add" menu at the top right, and choose "Page".
2. If you're asked to log in, you should use your jenkins.io account details
3. Enter a page name matching your plugin's name, i.e. with each word capitalised, ending with "Plugin" (e.g. "Snapchat Notification Plugin")
4. Add a plugin "infobox" with your plugin's ID, e.g.:

   ```
   {jenkins-plugin-info:snapchat-notification}
   ```

5. Under the infobox you are encouraged to include the description from your plugin metadata, or some other brief introduction to help people know what your plugin does.
6. Fill out the CAPTCHA and press Save

These are the bare minimum requirements for a plugin page.

Please check out some good examples of how you should lay out your page:

- Google Play Android Publisher Plugin
- Delivery Pipeline Plugin

Referencing the documentation page from the project file

You should link to your plugin's documentation, whether on the wiki or elsewhere, in your plugin's pom.xml, like this:

```
<project>
  ...
  <url>http://github.com/jenkinsci/your-plugin</url>
  ...
</project>
```

If you're building your plugin with Gradle, you set the URL in your `build.gradle` like so:

```
jenkinsPlugin {
  ...
  url = 'http://github.com/jenkinsci/your-plugin'
  ...
}
```

Or if you have a plugin written in Ruby, you must edit your `.pluginspec` file like so:

```
Jenkins::Plugin::Specification.new do |plugin|
  ...
  plugin.url = 'http://github.com/jenkinsci/your-plugin'
  ...
end
```

This ensures that the update center will list your plugin correctly once the new plugin version is released. If this is missing, or does not point to your Jenkins wiki page, your plugin will not be included in the update center.

Changelogs

Once you have made your first release, you should add release notes to a changelog. You have many options how to do it:

- use GitHub Releases (possibly with the help of Release Drafter), add a link to releases page to your documentation page (recommended)
- create a changelog file (Markdown, Asciidoc) in GitHub and link it from the documentation page
- include the changelog content in the documentation page

Adding Maintainer Information

In your POM, make sure to include developer information, such as:

```
<project>
  ...
  <developers>
    <developer>
      <id>devguy</id>
      <name>Developer Guy</name>
      <email>devguy@developerguy.blah</email>
    </developer>
  </developers>
</project>
```

The ID is your jenkins-ci.org account. The name is a human readable display name. This ensures that the update center and related tools are able to properly display the maintainer for your plugin.
It's highly advisable to include an email address so that people can contact you (this will be shown in the plugin infobox on the wiki), but it's optional. If pull requests go unmerged for a long time, and there's no way of contacting you, the maintainer, others will be encouraged to take over maintainership.

Continuous plugin builds

ci.jenkins.io

If you want changes to your plugin to be automatically built and tested on ci.jenkins.io, just add a `Jenkinsfile` to the root of your Git repository, using the `buildPlugin()` Pipeline step.

Once you've done this, your job will appear on ci.jenkins.io under the Plugins folder, and all subsequent branches and Pull Requests will be built.

This works automatically for all jenkinsci GitHub repositories that end with `-plugin`, so if you have another type of project that you want to build, please open a new issue in the INFRA project, using the "ci" component, and with a title like "Add build for whatever-module". Make sure to add the GitHub URL of your `Jenkinsfile` in the issue description.

jenkins.ci.cloudbees.com (legacy)

Previously, plugins were automatically added to jenkins.ci.cloudbees.com, and built as Maven jobs (clean install -e). All plugin CI broken build results will be mailed to any configured developers in the POM, plus culprits.

Ruby and Gradle plugins are also supported there, but require asking on the developers mailing list to have someone with the proper access rights update the job for you.

Request upload permissions

Permissions to upload plugin releases are independent of GitHub push access and maintained in this repository:

https://github.com/jenkins-infra/repository-permissions-updater

File a PR on that repository for your new plugin as described in the README of that repository. Without this, you'll be unable to release your plugin.

> For those permissions to be really enabled, you MUST have logged in at least in once with your Jenkins account into https://repo.jenkins-ci.org/webapp/#/home.
> If you never did, any modification to the repository above will be ineffective until then.

Releasing to jenkins-ci.org

> - Your plugin must have been forked before you can release. Please also take care of releasing *from* the repository hosted under the jenkinsci organization so that all tags and commits are actually there. This is important for the users so that we actually have the latest sources there in the future if the plugin happen to become unmaintained.
> - Also, please double-check you're allowed to deploy to the path for your plugin. See the previous point about Request upload permissions if you're unsure.
> - **Make sure the plugin would pass CI builds (to avoid tagging unreleasable commits)!** This covers running, but may be not limited only to:
>
>   ```
>   mvn clean package && mvn checkstyle::check && mvn findbugs:check
>   ```
>
>   Note that depending on your plugin's `pom.xml` requiring such checks during a build, you may not have to re-run them explicitly.
>   Consider also cleaning the Javadoc markup, if suggested to by `mvn javadoc:test-javadoc` although this is usually not a blocker for merging PRs and publishing a plugin release.
> - **Make sure you are running the `mvn release:*` steps from your workspace `master` branch**, after having merged all expected PRs and fixed all the bugs you need (and having git-pulled the latest upstream code into your local clone).
>   The `mvn release:prepare` would already use your Github super powers to bump the version and git-tag a release for the world to know!

The easiest way to publish a plugin is to run the maven release plugin with your jenkins-ci.org account:

```
$ mvn release:prepare release:perform -Dusername=... -Dpassword=...
```

If you are using GitHub, make sure that the "origin" remote is pointed to your actual plugin repository.

This will perform all the usual Maven release activity, and it will also post the plugin to the download section. If you run with the `-B` option, Maven will automatically use all the default values without prompting.

> **Do not** split that Maven command into two separate Maven invocations (i. e. `mvn release:prepare` followed by `mvn release:perform`):
> It won't work as you expect and will mess up your release. Always execute the two Maven goals together, in one command.

> If you get back an *401: Unauthorized* or if you encounter any other problem releasing your plugin while providing username and password as environment variable, try to pass username and password for authentication by modifying your settings.xml as described below.

However if your plugin is hosted on GitHub and you have different username and/or password for GitHub and jenkins-ci.org use of the command-line arguments for username/password will result in errors. You will have to start an ssh-agent and import your private key so the release plugin can access the GitHub repository (see Adding a new SSH key to your GitHub account), and configure the password for jenkins-ci.org in your `~/.m2/settings.xml` as described below.

Instead of listing username/password on the command line, you may also specify these in `~/.m2/settings.xml` as follows:

settings.xml

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <servers>
    <server>
      <id>maven.jenkins-ci.org</id> <!-- For parent 1.397 or newer; this ID is used for historical reasons and
independent of the actual host name -->
      <username>...</username>
      <password>...</password>
    </server>
  </servers>

</settings>
```

You may also follow these instructions to encrypt the password stored in `settings.xml`. Once you are done with this run `mvn deploy` to verify that your credential is correctly recognized by Maven.

Additionally, note that your settings.xml file may need extra configuration for publishing the plugin. See here.

Also consider these tips before making a plugin release.

The released plugin should show up in the update center within eight hours.

You will likely need to add your new release info to the Changelog section of your Jenkins plugin wiki page.

If you are using GitHub with SSH you have to omit the username and password parameter to be able to build and push the release. To be able to copy the artifacts to the jenkins-ci.org maven repository you have to set up the settings.xml as described above.

If you are using the Plugin Parent POM version 2.3 or later, the `maven-release-plugin` is configured so that tests are only executed in the `release:prepare` phase. If you want them to be executed in the `release:perform` phase as well, set the `release.skipTests` property to `false`.

Blacklisting plugin in the Update Center

In some cases it might be necessary to blacklist an already release version of a plugin in the Update Center to prevent further installations. Some examples when this might happen are:

- there was a major regression detected in the new plugin version
- there was a binary incompatibility detected causing cross-plugin failures

The Update Center blacklist is handled by the repository here: https://github.com/jenkins-infra/backend-update-center2/blob/master/src/main/resources/artifact-ignores.properties

To blacklist a version, open a PR specifying a plugin version to be banned, example: https://github.com/jenkins-infra/backend-update-center2/pull/183

Releasing alpha/beta versions of plugins

See this nice blog post by Kohsuke about Experimental Plugins Update Center.

Working around common issues

These are some common issues that can occur when releasing a plugin.
If you did release your plugin successfully but are having problems, see also the "Help!" section at the end of this page.

The upload to the Maven repository fails with "401 Unauthorized"

This error means that Maven did not successfully authenticate with Artifactory. This can have multiple reasons:

- You did not specify credentials for the repository in `settings.xml`
- The settings you did specify use an ID not matching the repo ID in your (effective) POM's `distributionManagement` – it's independent of the actual host name and probably needs to be `maven.jenkins-ci.org`.
- The credentials are wrong, for example because you changed your password on https://accounts.jenkins.io

The upload to the Maven repository fails with "403 Forbidden"

Make sure you're allowed to upload this plugin: Permissions to upload are independent of GitHub repository permissions and are defined in YAML files in https://github.com/jenkins-infra/repository-permissions-updater File a PR to that repository as described in its README to request access.

Another possible reason is that the release you're trying to create already exists. We don't grant permission to delete artifacts, which is a prerequisite for being able to replace. Retry with a higher version number. If that still fails, check whether you're trying to upload the same file twice during the same release process.

The upload to maven.jenkins-ci.org fails with "Connection refused", "Connection timed out", or similar

The Maven repository is now located at repo.jenkins-ci.org. (Note that this host name has nothing to do with the repository ID to use in `settings.xml`.)

Either overwrite the repository location in your POM to point to https://repo.jenkins-ci.org, or update to the parent plugins POM 2.5 or newer. With version 2.0, it was made independent of the version of Jenkins your plugin is built against. See the sample POM for new Jenkins plugins as well as INFRA-588, where a lot of commits adapting plugins to no longer reference maven.jenkins-ci.org are referenced in comments.

The release process created a "SNAPSHOT" version / Using git version 1.8.5.2 or around

After cd-ing to your plugin git repo folder, run the maven release commands this way, forcing version 2.5 of maven-release-plugin; alternatively, proceed to the next work-around topic right after this one:

```
mvn org.apache.maven.plugins:maven-release-plugin:2.5:prepare
mvn org.apache.maven.plugins:maven-release-plugin:2.5:perform
git push (should your git status still be "dirty")
```

Please keep this wiki topic at the beginning of hereby work-arounds, for top visibility purposes.

No prepare commits in git

Should the previous (related) work-around not suit you, update maven-release-plugin to 2.5 version. See: jenkinsci-dev maillist

If a release only deploys snapshots consider upgrading the maven-scm-provider-gitexe to 1.9.1 or more (see more details.)

```
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-release-plugin</artifactId>
         <version>2.5.1</version>
         <dependencies>
           <dependency>
             <groupId>org.apache.maven.scm</groupId>
             <artifactId>maven-scm-provider-gitexe</artifactId>
             <version>1.9.2</version>
           </dependency>
         </dependencies>
       </plugin>
```

or running following command may make your maven-release-plugin work correct (this changes output format of `git status`).

```
git config --global --add status.displayCommentPrefix true
```

If a release fails with the following error, that means the version number for your plugin in `pom.xml` does not end with `-SNAPSHOT`. Add this suffix and commit the change, then try again.

```
     [INFO] ------------------------------------------------------------------------
     [ERROR] BUILD FAILURE
     [INFO] ------------------------------------------------------------------------
     [INFO] You don't have a SNAPSHOT project in the reactor projects list.
```

If changes to your pom.xml seem to be having no effect, try one of these to clear out any intermediate files and start over:

```
mvn -Dresume=false release:prepare release:perform
mvn release:rollback
mvn release:clean
```

Releasing failed due to errors relating to Javadoc?

If you see something like this when releasing, including lots of Javadoc errors you can't actually fix, because they're part of the Messages class...

```
[INFO] [ERROR] Failed to execute goal org.apache.maven.plugins:maven-javadoc-plugin:2.8:jar (attach-
javadocs) on project <plugin-artifact-id>:
 MavenReportException: Error while creating archive:
... lots of Javadoc errors here ...
```

..then it's probable that you're building your plugin with Java 8, which has a tool called DocLint enabled by default.

You need to disable DocLint by adding the following to your `pom.xml`. Merge into the existing `build.plugins` section, if there is one.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <configuration>
        <additionalparam>-Xdoclint:none</additionalparam>
      </configuration>
    </plugin>
  </plugins>
</build>
```

My release failed and I want to redo it from scratch

When a release fails in the middle, you unfortunately have to manually cancel out some of the side-effects Maven has caused. See this e-mail for details. The easiest way out is to forget about the botched release and simply move on to the next release number.

Subversion 1.6 authentication problem

Subversion 1.6 has a known problem with the `--non-interactive` switch that Maven uses, in that the presence of this switch prevents Subversion from accessing your authentication credentials. See this thread for more background discussion.
You can verify whether this is the cause of your problem by comparing the behavior between `svn ls --non-interactive https://svn.jenkins-ci.org/` and `svn ls https://svn.jenkins-ci.org/`. Once verified that this is the root cause, tell Subversion to store the password in plain text

Additionally you need to check that you are not accessing the subversion repository with the "guest" user. So try `svn ls --username YOURNAME https://svn.jenkins-ci.org/` . If it asks for your credentials then simply give it and retry.

OutOfMemoryError: Java heap space

If the release fails because Maven runs out of heap space, you may need to increase its limit by defining the MAVEN_OPTS environment variable. For example, in a bash shell, you could add the following to `~/.profile`:

~/.profile

```
export MAVEN_OPTS=-Xmx300m
```

If you are running Maven inside IntelliJ 9.0.1, its setting for Maven > Runner > VM Parameters applies to only the parent Maven process, but it may be a child Maven process that is running out of memory. Furthermore, if you are running IntelliJ on MacOSX 10.6, normal environment variables, i.e., exported from ~/.profile, are not seen by apps launched from Finder. So, to increase the heap limit of child Maven processes in IntelliJ, you can configure MAVEN_OPTS in /etc/launchd.conf instead. Create or edit the file, e.g. `sudo vi /etc/launchd.conf` and add:

/etc/launchd.conf on MacOSX

```
setenv MAVEN_OPTS -Xmx300m
```

Error deploying artifact: Connection failed: Unable to connect to https://svn.dev.java.net/svn/maven2-repository/trunk/repository/

If the deployment fails it might be trying to write to a legacy repository. The instructions above and in the migration page should help with setting the correct target repository.

HTTP 401 when transferring a file to the Jenkins Maven repository

If, when running `mvn release:perform`, you get an error message similar to the following one, there are a few possibilities:

- Your password is incorrect
- You haven't added your jenkins-ci.org credentials to your `~/.m2/settings.xml` file as described above.
- The ID of the repository as specified in POM (which is normally inherited) doesn't match the ID in `~/.m2/settings.xml`. To verify this, run `mvn help:effective-pom` and look for the <distributionManagement> element. For example, see this. This is because earlier versions of the plugin parent POM had this kind of problems. The easiest solution is bump up the parent POM to 1.409 or later. If you need to keep the parent version as is, then add additional <server> entry in your `settings.xml` with all the variations of the IDs.
- Your plugin with this version was already released, and you are trying to release another one with the same version (check it here).

```
[INFO] [ERROR] Failed to execute goal org.apache.maven.plugins:maven-deploy-plugin:2.5:deploy (default-
deploy) on project sbt:
Failed to deploy artifacts: Could not transfer artifact org.jvnet.hudson.plugins:sbt:hpi:1.0 from/to
maven.jenkins-ci.org (http://maven.jenkins-ci.org:8081/content/repositories/releases/):
Failed to transfer file:
http://maven.jenkins-ci.org:8081/content/repositories/releases/org/jvnet/hudson/plugins/sbt/1.0/sbt-1.0.
hpi.
Return code is: 401 -> [Help 1]
```

Make sure PW encryption is correct:

- Login https://repo.jenkins-ci.org/webapp/#/login with jenkins-ci.org account
- Go to https://repo.jenkins-ci.org/webapp/#/profile
- Unlock "Current Password"
- Put "Encrypted Password" to your settings.xml

Make sure you use Maven 3

If `release:prepare` fails with the following error, you are most likely using "Cygwin" on Windows. There is a failure of path translation somewhere between Maven and the Cygwin Subversion client. The easiest way around this is to install a native Windows Subversion client (eg. Win32Svn) and re-run from a DOS/CMD window.

```
[INFO] Checking in modified POMs...
[INFO] Executing: cmd.exe /X /C "svn --non-interactive commit
       --file C:\DOCUME~1\username\LOCALS~1\Temp\maven-scm-1876439793.commit
       --targets C:\DOCUME~1\username\LOCALS~1\Temp\maven-scm-7190199490958000758-targets"
[INFO] Working directory: D:\home\ed\projects\hudson\plugins\lavalamp
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Unable to commit files
Provider message:
The svn command failed.
svn: '/cygdrive/d/home/username/projects/hudson/plugins/lavalamp/D:/home/username' does not exist
```

If a release fails with either of the following errors, that means you don't have `hudson/plugins/pom.xml` installed in your local repository. Run `mvn -N install` in the `plugins` directory and retry a release.

```
[INFO] Executing: mvn deploy
--no-plugin-updates -DperformRelease=true
    [INFO] Scanning for projects...
    Downloading:
http://repo1.maven.org/maven2/org/jvnet/hudson/plugins/plugin/1.NNN/plugin-1.NNN.pom
    [INFO] ------------------------------------------------------------------------
    [ERROR] FATAL ERROR
    [INFO] ------------------------------------------------------------------------
    [INFO] Error building POM (may not be this project's POM).


    Project ID: null:xxxxxx:hpi:N.N
```

```
[INFO] Scanning for projects...
       Downloading: http://repo1.maven.org/maven2/org/jvnet/hudson/plugins/plugin/1.212/plugin-1.212.pom
       [INFO] ------------------------------------------------------------------------
       [ERROR] FATAL ERROR
       [INFO] ------------------------------------------------------------------------
       [INFO] Failed to resolve artifact.

    GroupId: org.jvnet.hudson.plugins
       ArtifactId: plugin
       Version: 1.212

    Reason: Unable to download the artifact from any repository

      org.jvnet.hudson.plugins:plugin:pom:1.212

    from the specified remote repositories:
          central (http://repo1.maven.org/maven2)
```

Could not find maven-hpi-plugin jar

If release:prepare fails with an error like the following:

```
[ERROR] Unresolveable build extension: Plugin org.jenkins-ci.tools:maven-hpi-plugin:1.95 or one of its
dependencies could not be resolved:
Could not find artifact org.jenkins-ci.tools:maven-hpi-plugin:jar:1.95 in central (http://repo.maven.
apache.org/maven2)
```

You may need to update your local settings.xml. (A build might work, but not a release:prepare.) See here.

Subversion 409 Conflict

If a `release:perform` fails with an error like the following, retry `mvn release:perform` later. We don't know the root cause of this problem, but the problem does seem to disappear after a while.

```
[INFO] [INFO] Error deploying artifact: Connection failed: Unable to connect to
  https://svn.dev.java.net/svn/maven2-repository/trunk/repository/
[INFO]
[INFO] svn: The specified baseline is not the latest baseline, so it may not be checked out.
[INFO] svn: CHECKOUT of '/svn/maven2-repository/!svn/bln/1348162': 409 Conflict (https://svn.dev.java.
net)
[INFO] [INFO] ------------------------------------------------------------------------
[INFO] [INFO] For more information, run Maven with the -e switch
[INFO] [INFO] ------------------------------------------------------------------------
```

git push hangs

Stackoverflow offers suggestions on how to correct this at http://stackoverflow.com/questions/3243755/maven-error-releasing-code-to-github-hangs-after-push.

Unable to parse configuration of mojo ... pluginArtifacts

While updating a plugin that was not quite touched for years (so some older-style config is a likely suspect), got this error message:

```
Unable to parse configuration of mojo org.codehaus.mojo:findbugs-maven-plugin:3.0.3:findbugs for
parameter pluginArtifacts:
Cannot assign configuration entry 'pluginArtifacts' with value '${plugin.artifacts}'
of type java.util.Collections.UnmodifiableRandomAccessList to property of type java.util.ArrayList
```

After some discussion on IRC and googling for similar issues on github, the likely verdict was that instead Maven version was a bit too new for one of the components used to make the release (e.g. https://github.com/fhoeben/hsac-fitnesse-fixtures/issues/238), so downgrading from 3.6.0 to 3.5.x (downloaded straight from https://archive.apache.org/dist/maven/maven-3/3.5.4/binaries/) seems to have worked. Note that the libraries mentioned in that issue were supposedly fixed in a later version to work well with more Maven releases, so this troubleshooting point should be obsolete as soon as your plugin's dependency tree includes that fix (or newer).

(Should not be) Skipping Findbugs

Skipping findbugs is something very much frowned upon, better address cleanly the issues found by it. That said, when making a new release of old plugin codebase that had those reported issues before you came on board, or where the issues can not be fixed by yourself (e.g. are in third party code), turning a blind eye to them may be an interim acceptable option.

As exemplified in https://github.com/jenkinsci/SCTMExecutor-plugin/blob/f801eecfbdf2216f8daf9dfe518e55a62527f26e/pom.xml#L20 one can include a toggle in their `pom.xml` section `properties` :

```
        <properties>
                ...
                <findbugs.failOnError>false</findbugs.failOnError>
                <findbugs.excludeFilterFile>findbugs-exclude.xml</findbugs.excludeFilterFile>
        </properties>
```

with the exclude file containing rules for the tool, like:

```
<FindBugsFilter>
        <Match>
          <Package name="~com\.borland\.sctm.*" />
        </Match>
</FindBugsFilter>
```

PS: Unfortunately, the usual local development cycle with `mvn clean test package install` does not trigger the findbugs-related targets, so you only discover problems when trying to make a release. You can also execute `mvn findbugs:check` explicitly to generate the report.

PPS: The X11 interface spawned by `mvn findbugs:gui` (after the analysis tool has executed and left some notes in your workspace) is quite useful in both pointing to the lines of code with issues, and in detailing what the problem is and some reasonable ways to fix it.

For TortoiseGit users

If you use TortoiseGit as a Git client, do followings to have maven to run with TortoiseGit settings.

1. Write the username/password of your jenkins-ci.org account in settings.xml (See above for details).
    - Passing username/password in the command line seems result that SSH client recognizes the username as the hostname, and fails to connect.
2. Add the path of msysgit to PATH environment variable. You can see the path to msysgit in TortoiseGit>Settings>General, "Git.exe Path"

    ```
    set PATH=C:\Program Files\Git\bin
    ```

3. Set the path of TortoisePlink.exe (in the bin directory of TortoiseGit) to GIT_SSH environment variable.

    ```
    set GIT_SSH=C:\Program Files\TortoiseGit\bin\TortoisePlink.exe
    ```

4. Start pageant, and load putty key (ppk).

    ```
    pageant path\to\id_rsa.ppk
    ```

5. Run release command without username/password

    ```
    mvn release:prepare release:perform
    ```

Frequently asked questions

What should the Java package name be?

Most old plugins use `org.jenkinsci.plugins.* but recent ones use io.jenkins.plugins.*` , but if you'd like to use your own package name, feel free to do so.

Help! My plugin is not showing up in the update center

1. First, check the release Maven repo and see if your plugin is listed there. If not, your release process failed, and it never left your PC. If the other items on this list don't help to resolve this issue, capture the output from `mvn release:prepare release:perform` and send it to the dev list.
2. If your new version appears in the SNAPSHOT repo (rather than the "release" repo), see the "Working around common issues" section above
3. Check if your new version is in https://updates.jenkins-ci.org/update-center.json. This file is updated every 4 hours (plus mirror propagation time), so there's some delay before your new version appears here. So, as a rule of thumbs, **please wait at least 8 hours before starting to worry about any issue**.
4. If your plugin appears there but not in your Jenkins update center, visit Manage Plugins / Advanced and click "Check Now" to make Jenkins retrieve the latest `update-center.json` data.
5. (Currently unavailable due to INFRA-810) Search for your plugin's artifact ID in the Update Centre build log to see whether your plugin has been included, or why it has been excluded.
    a. Check that your your plugin's wiki page is a child of the Plugins page
    b. Check that you have entered the correct plugin wiki URL in your POM
6. Check the INFRA project for existing issues, or ask on IRC or the developers' mailing list; otherwise you can open an INFRA issue