The release of software packages is an important phase of the Agile Application Lifecycle Management (ALM). The source code should be put into a version control system and extracted to be compiled and packaged. In most cases, your customer requires a complete release package, which also contains additional information. In this article from Agile ALM, you will learn how to create a software release and distribute it using Subversion and Apache Maven.

You may also be interested in...

Releasing with Subversion and Maven

Before you start with the creation of your release/distribution, you need to be clear about what is shipped with the actual software. Typically, a release/distribution needs a lot more than just some compiled Java classes. Some examples of common artifacts in a distribution:

- **Documentation**—Practical user guides, such as installation guides or how-to documents, and API descriptions like JavaDocs or TagDocs, need to be made available as part of a distribution.

- **Examples**—Showcase applications (or reference implementation) and API usage examples. Most open-source releases contain examples, which may be very simple or rather complex, showing many features.

- **Distribution of the source code**—This is not only a question for open-source projects. Releases for in-house projects or business partners may also distribute the source code, which makes debugging easier.

- **Anything else of value for you in your individual situation**—For project specific requirements it may be the case that you need to ship other artifacts than those listed above.

The location of the release deliverables is also important. For instance, with Apache Maven, JAR files (including sources and JUnit tests) are deployed to an in-house repository, which is basically an HTTP/file server (for example, Artifactory or SVN) that follows a defined directory layout. The complete release package, meaning the distribution, is often placed somewhere else. It is very common to place these files on the website of your company or open-source project. Another option is to deliver the distribution on a network file system (NFS), which you use to exchange files with your business partners or other in-house teams.

Once it is clear where to place the distribution and what it contains, you are hitting what is probably the most important question: how do we deploy the release? There are several tools and options available to get a release out. These can be historic make files or some BASH/PERL scripts that run the software build and assemble the final release package. On older Java projects you may still see the usage of homegrown Ant tasks to generate the release and its distribution package. However, today the de-facto standard for Java-based projects is Apache Maven. Maven not only manages the build system of a software project, but it helps to manage the project through its entire lifespan as well. This includes artifacts like:

- **Documentation**—The product website or JavaDoc.

- **Source code management**—The location of the SVN repository.

- **Release management**—Branching and tagging in SVN and creating the distribution of the release.

**COMMON PRACTICES IMPLEMENTED IN MAVEN PLUGINS**

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/huettermann/](http://www.manning.com/huettermann/)
This is one of the benefits that Maven offers: best practices are coded into a lot of available plugins. This means the Agile development process starts with the choice of a powerful tool belt.

Maven is a framework that simply executes plugins that accomplish a variety of important tasks. There are many available plugins for Maven that can be used to perform tasks that are essential for building, packaging, and deploying your software. An example would be to use Maven to post the results of the nightly builds to Twitter (see http://code.google.com/p/maven-twitter-plugin/). In this article, we will describe several major plugins and their required configurations that are ideal for producing a software release.

Before you start to generate a release, you need to understand that there are several ways to do it. You could start to run the release from within the SVN trunk folder, which is probably not a good idea because the main code line is usually developed on the trunk. It is a common practice to create a separate release branch using a standard naming convention such as release-version-x.y.z.

This section uses a common SVN layout:

- myProject/
  - branches/
    - branchA
    - branchB
  - tags/
  - trunk/
  - src
  - pom.xml
...

The first task to enable releasing with Maven is to include the maven-release-plugin in your POM as part of the project's Maven build lifecycle:

```xml
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-release-plugin</artifactId>
<version>2.0</version>
```

Creating the branch and preparing the release

Creating the entire release from a release branch is a good pattern. Doing so ensures that the main development can continue on the trunk, without any undesired side effects. Imagine if some of the changes are just not ready in time for a scheduled release. In this case, you would want to be able to reverse the changes that are not completed. This is much easier if you are using a release branch that contains only the tested and completed features approved for the release. Creating a branch can be done with the tools provided by SVN itself, like svn copy. On Windows, there is a convenient graphical tool (TortoiseSVN) that supports creating branches and tags. Even with Apache Maven, there are several possibilities to create a branch, such as with the maven-scm-plugin or the maven-release-plugin. The maven-release-plugin can be used to help automate the release process including creating a release branch automatically.

Before you can use the release-plugin for branch creation, you have to configure your Source Control Management (SCM) settings in the project’s pom.xml file, as seen in listing 1.

**Listing 1 Configuring SCM in project’s model (POM)**

```xml
<scm> #1
  <connection>
    scm:svn:http://server/svn/path/to/project/trunk/
  </connection>
  <developerConnection> #2
    scm:svn:https://server/svn/path/to/project/trunk/
  </developerConnection>
  <url>http://www</url> #3
</scm>

#1 URL to check out code
#2 URL to work on code
#3 URL for web interface
```

Maven site generation

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/huettermann/
The SCM configuration in listing 1 is used when the maven site plugin creates the website of the project. The information about the used Source Code Repository is made available on the generated source-repository.html webpage (see MyFaces or CXF).

The <connection> element represents the URL that is used by everybody who is interested in checking out the source code (#1). Usually, the anonymous checkout is done through an unsecured http connection. The <developerConnection> section contains the URL, which is used by the developers of the project to commit their changes back into the source repository (#2). In almost all cases this is done through a secured http (https) connection. The only exception would be when the repository is already hidden by a corporate firewall. The last parameter (<url>) is a nice-to-have setting. Usually it contains a web interface to the source code (#3), stored in the repository. Most Subversion servers have something like viewvc or websvn installed.

**Using Git as your repository**

If you plan to use Git as your repository, you can use the Gitweb or even host your entire project directly on GitHub. The benefit of a web interface installation is that you can easily view the annotated source code with additional information: you get a colored overview of the actual difference between two revisions and some more advanced information of who added or changed the file in which change. To get similar information, you would need to use a command-line tool for your source code system like git-blame. In SVN, you would use svn blame or TortoiseSVN.

Once you configure the required settings, it is time to create the first branch:

```
mvn release:branch -DBranchName=myFirstRelease
```

Shortly after executing the plugin, it asks for the next version number (that is then recorded in the POM) on your trunk. If your project has the version number 1.0.0-SNAPSHOT (check your pom.xml for its XML <version/> element), you will see the following prompt:

```
What is the new working copy version for “my-tool”? (com.book:my-tool) 1.0.1-SNAPSHOT: 
```

Hitting the ENTER key accepts the suggested value, or you can specify a different version number. After that, the plugin updates the version number that has been automatically incremented in the pom.xml and commits the change to the SVN trunk. The plugin also creates the new branching folder in the source repository. The pattern for the myFirstRelease branch would be something like:

```
https://server/svn/path/to/project/branches/myFirstRelease
```

**MAVEN VERSION SCHEMA**

The maven-release-plugin requires the version scheme to end with a -SNAPSHOT, such as 1.2.4-alpha-4-SNAPSHOT.

The release-plugin checks if you have modified files on your project. If this is the case, you will get a build failure:

```
[ERROR] BUILD FAILURE
[INFO] ---------------------------------------------------------------------------
[INFO] Cannot prepare the release because you have local modifications :
[pom.xml:modified]
```

To continue, you need to evaluate the change and commit it to the SVN trunk or reverse it, if it was not intended.

To continue releasing, you need to check out the newly created branch:

```
svn checkout
https://server/svn/path/to/project/branches/myFirstRelease/
directoryToContainTheBranch
```

If you take a look at the <scm> section of its pom.xml file, you’ll notice that the previous branch goal also updated this section. It contains the right URLs to point to our new SVN branch.

Before you can continue with the release preparation, your project’s pom.xml needs some extra configuration, see listing 2.

**Listing 2 maven-release-plugin in project’s POM**

```
<build>
```

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/huettermann/](http://www.manning.com/huettermann/)
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-release-plugin</artifactId>
    <version>2.0</version>
    <configuration>
      <preparationGoals>clean verify install</preparationGoals>
      <tagBase>https://server/svn/path/to/project/tags</tagBase>
    </configuration>
  </plugin>
</plugins>

#1 Includes the maven-release-plugin
#2 Base location of tags

Inside the <build> section of your pom.xml you have to configure the maven-release-plugin (#1). The important part here is the <tagBase> element, which tells the plugin the base location for all of your Subversion tags (#2). By default, the plugin assumes that you create the release out of the trunk folder and, therefore, it would automatically follow the common pattern to create all TAGs as subdirectories of the default SVN tag-location, which is the tags/ folder. Since we want to run the release preparation from a branch, we apply the above configuration to our pom.xml file.

**GENERATING OUT OF THE TRUNK**

Generating releases out of the trunk folder is also commonly used. However, this approach has some risks and you should only take this approach if you have the experience to understand its risks. It is simply safer to have a separate release branch folder for this step. The good news is that the extra setup is not too difficult.

Now you can finally start the preparation of your release:

```
mvn release:prepare
```

What does preparation mean? If your branch has some Maven snapshot dependencies, the plugin asks you if you really want to continue with the release procedure. Generally, it is a very bad idea to branch and release a software package that has a dependency on a snapshot because these are usually generated every day, with a nightly build (basically, snapshots are unreleased software or software under development). The consequence of this is that your released project can become very unstable. In the worst case, this can mean that overnight some deprecated APIs have been removed from your dependency and the result is that your released software is no longer usable. You really want to avoid this scenario and should never release software that has a snapshot dependency; instead, you want to reference stable versions. Once the release-plugin identifies a snapshot dependency on your project, it gives you the following prompt:

```
There are still some remaining snapshot dependencies.: Do you want to resolve them now? (yes/no) no:
```

The default value for the prompt is no because you should avoid snapshot dependencies. If you continue with the release by entering yes, even if there is one or more snapshot dependencies on your project, the plugin will force you to upgrade to the released version of the dependency.

**TIP**

If you want to avoid any snapshot dependencies on your project, you need to use the maven-enforcer-plugin.

For instance, if your project depends on a 1.0.2-SNAPSHOT dependency, the plugin wants to use the 1.0.2 released version of it. If your project really depends on a change that was made in 1.0.2-SNAPSHOT and there is no such release yet, it is recommended to delay releasing the latest changes. Instead, you should use an earlier version of the dependency.

For source code, sample chapters, the Online Author Forum, and other resources, go to

http://www.manning.com/huettermann/
“But I must release with snapshots!”

Sometimes there are no other options and you must release with a snapshot dependency, for example, because a critical bug was fixed on a dependent project. One approach to releasing would be to build the dependency on your system and manually change the version to something that would never be picked by the original project team. An option would be using some version string like 1.0.2-modified-by-MyCompany-for-iteration-1 because it makes clear that your team provided the modified (or hacked) dependency. This special dependency is also a candidate for your distribution, as it is nowhere else available. Note that things like this are only a valid option if the license of the (open source) dependency fits into your own schema. Another option to deal with snapshots is the versions:lock-snapshots plugin. It inspects all POMs and replaces all snapshot versions with the current timestamp version of that snapshot, for example, 20100320.172301-4. You can do your releasing with this modified code base. You could also switch back to regular snapshots via versions:unlock-snapshots plugin.

Maven’s version plugin also provides some other neat goals to work with versions, as seen here.

To continue with the preparation, let’s assume there already was a release and we just haven’t picked up the released version yet. You could manually update the version and commit the change to SVN or let the release-plugin do it for you. During the execution of the release:prepare goal, the plugin asks you about the version number for the release:

What is the release version for "my-tool"? (com.book:my-tool) 1.0.0: : 1.0.0-alpha

What is SCM release tag or label for "my-tool"? (com.book:my-tool) my-tool-1.0.0-alpha: :

What is the new development version for "my-tool"? (com.book:my-tool) 1.0.1-alpha-SNAPSHOT: :

For the version number, you could accept the suggested 1.0.0 value or you could specify a different version ID. It is always good to have some alpha, beta, or release-candidate releases before shipping the final release. Let’s imagine there is a demand for an alpha release. On the prompt, simply type in 1.0.0-alpha and accept the suggested defaults for the Subversion tag and the next version number increment. After you specify the version number, the plugin triggers the regular project build and also executes some SVN tasks. Under the hood, it creates the tag (URL would be http://server/svn/path/to/project/tags/my-tool-1.0.0-alpha) and it also updates and commits the new version number (1.0.1-alpha-SNAPSHOT) to your previously created branch.

SVN TAG

A tag inside of the Subversion repository is by convention a folder that is never updated. Tag folders can be understood as a single source of truth because they identify the exact version of the source code that was used to create a baseline release! Later, if you have to patch an existing release, for instance due to a critical security bug, you usually create a branch of the corresponding tag folder to make sure there is nothing else to patch.

By just committing changes to the tag (what is possible by default), the tag becomes a branch automatically (by convention). If you want to be sure that this does not happen, restrict the access to tag folders, e.g. by using SVN hooks.

If problems occur during the release:prepare goal, execution could lead to a build failure. In such cases, you need to solve the failures, of course. For instance, if your SVN server is down and you need your admin to get it back up, once the problems are fixed, you can simply continue where the maven-release-plugin exited the job by entering the command:

mvn release:prepare -Dresume

Another option would be running mvn release:clean release:prepare, which first deletes all temporary files created by the plugin and afterwards reruns the entire release preparation again. If the release-plugin had already changed some of your files, like the <version> in the pom.xml files, you could roll back these changes like so:

mvn release:rollback

The release-plugin has some built-in safety, which ensures you are not spending extra hours on reassembling the release again and again. This is part of the Agile strategy behind Maven and similar tools.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/huettermann/
Perform the release

Before you can perform the last steps on your release, the creation of the release, the release-plugin requires a little bit more configuration in your pom.xml. What you need to configure is the distribution management section. Here is an example of what this can look like:

```xml
<distributionManagement>
  <repository>
    <id>local-repository</id>
    <name>My staging component repo</name>
    <url>file:///m2_repo</url>
  </repository>
</distributionManagement>
```

Inside the `<distributionManagement>` section, you specify the Maven repository where the release should be deployed to. Usually the Maven repository is a remote (HTTP) server and the upload to it is done via SSH. However, for now we will use a folder on the local file system because it works as well. To finish the release procedure you simply need to invoke the release:perform goal:

```
mvn release:perform
```

The plugin now creates a target/checkout directory and checks out the previously created Subversion tagged version of your project. Next, it executes the normal build process. However, with the release:perform goal, we not only generate the normal JAR file (or whatever deployment unit your project delivers), the plugin also generates the matching javadoc-jar and source-jar files:

```text
my-tool-1.0.0-alpha-javadoc.jar
my-tool-1.0.0-alpha-sources.jar
my-tool-1.0.0-alpha.jar
```

During the execution of the release:perform goal, the maven-release-plugin uses the deploy-plugin and handles the upload to the Maven repository, which has been configured in the `<distributionManagement>` section of your project's pom.xml file. If no error occurs, the release-plugin cleans up its temporary files by internally calling `mvn release:clean`. This erases the backup POM files (pom.xml.releaseBackup) and the release.properties file, which stores information about the ongoing release process. Congratulations! Your release is complete. Over the course of this guide, you saw that Maven has many useful features that are implemented through its plugins. However, there is still some room to improve this release process. By configuring a remote repository (the component repository), the invocation of mvn release:perform will deploy the artifacts to the server. However, you don't always want to directly deploy the newly generated Maven artifacts to a production Maven repository. In a case like that, the release:stage goal is your friend:

```
mvn release:stage -DstagingRepository=remote-repo::default::scpexe://URL_TO_DIR
```

The release:stage goal does not require a repository URL to be present in the pom.xml, but the same settings.xml configuration is still needed! Note that the `stagingRepository` parameter starts with the ID of the specified server.

A software release requires more than just compiling sources from a Subversion tag. The generated artifacts (such as documentation and binary JAR file) are reviewed and tested by the QA team. Of course, every project has a bunch of JUnit tests and the entire development even starts with a test case. The final testing is still done by the QA team, who eventually approves the release. The QA gets the released artifacts from a staging Maven Repository. The best solution would be that the QA and the customer get exactly those files that have been tested and verified by the QA team.

Sometimes, it is a requirement that you need to make the exact JAR files available that you used for testing, for example, when a project is implementing a Java standard and executing a test compatibility test kit (TCK) as part of the required test plan. This could be the above specified `file:///m2_repo` directory, which could be mounted as a network device. The QA test plan is executed against the previously created release artifacts. Once the QA team gives their OK, the Maven artifacts should be deployed to a production/release Maven repository. Here are a few options for automating the deployment:

1. Rebuild the bits from the Subversion tag. Once you build the code, you will need to use the maven-deploy-plugin to get the artifacts deployed. This plugin reruns the important parts of the previous release procedure, without having QA to test the generated bits. Testing is important because it is always possible that some mistakes happen while rebuilding the release from the TAG folder.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/huettermann/](http://www.manning.com/huettermann/)
2. Copy the artifacts to the final repository using secure copy, `scp`, or copy, `cp`, to manually copy the artifacts. This ensures the exact JAR files are made available through the Maven repository. However, doing so would destroy the maven metadata (see below).

3. Using the maven-stage-plugin. Use the stage:copy goal: `mvn stage:copy`

   -Dsource="file:///m2_local_staging_repo/"
   -Dtarget="scp://maven_repository_server/path/to/repo"
   -Dversion=1.2.3

   The plugin is basically a smarter version of the copy process, which also honors the maven metadata. The plugin creates a ZIP file of all in the local staging repository and uploads it to the given remote server by using scp. Once the upload is done, the ZIP is extracted and later removed. This means that the artifacts are now correctly deployed to your Maven repository. The plugin is useful, especially when you care about having intact maven metadata files. However, the plugin itself has some small issues. You have to specify a (meaningless) version parameter and it is not possible to download from a remote server to your local folder. Additionally, your current user account needs to exist on the server and you are required to have the "write" access rights for the remote directory. Finally, the password entered for the remote account is displayed in plaintext. These are significant limitations, but the overall benefit outweighs the extra effort and results in having correct metadata files.

4. Copy the artifact via the repository manager feature set. Artifactory provides a context menu for all artifacts in your repositories, as seen in figure 2. You can mark artifacts and copy (or move) them to another repository (for example, a special staging repository). Artifactory provides a dry run feature. This way, you can first try to execute the command without any commit to keep a consistent state even if something goes wrong. If your check is successful, you can then execute the command. You can also use advanced staging strategies by applying matrix parameters and smart searches.

![Artifactory Image](image_url)

**Summary**

This article introduced release management with Maven. We learned that it is possible to host Maven artifacts in Subversion, although using a dedicated repository manager like Artifactory adds much more value and results in a better solution.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/huettermann/](http://www.manning.com/huettermann/)
Here are some other Manning titles you might be interested in:

**Becoming Agile**
...in an imperfect world
Greg Smith and Ahmed Sidky

**Specification by Example**
Gojko Adzic

**Enterprise Search in Action**
Marc Teutelink

Last updated: July 20, 2011

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/huettermann/](http://www.manning.com/huettermann/)