# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>i</td>
</tr>
<tr>
<td>1. What is Maven?</td>
<td>1</td>
</tr>
<tr>
<td>2. Features</td>
<td>3</td>
</tr>
<tr>
<td>3. FAQ</td>
<td>4</td>
</tr>
<tr>
<td>4. Community Overview</td>
<td>11</td>
</tr>
<tr>
<td>4.1. How to Contribute</td>
<td>13</td>
</tr>
<tr>
<td>4.2. Getting Help</td>
<td>15</td>
</tr>
<tr>
<td>4.3. Issue Management</td>
<td>17</td>
</tr>
<tr>
<td>4.4. Source Repository</td>
<td>18</td>
</tr>
<tr>
<td>5. Running Maven</td>
<td>25</td>
</tr>
<tr>
<td>6. Maven Plugins</td>
<td>27</td>
</tr>
<tr>
<td>7. User Centre</td>
<td>35</td>
</tr>
<tr>
<td>7.1. Maven in 5 Minutes</td>
<td>36</td>
</tr>
<tr>
<td>7.2. Getting Started Guide</td>
<td>40</td>
</tr>
<tr>
<td>7.3. POM Reference</td>
<td>60</td>
</tr>
<tr>
<td>7.4. Settings Reference</td>
<td>95</td>
</tr>
<tr>
<td>7.5. Guides</td>
<td></td>
</tr>
<tr>
<td>7.5.1. The Build Lifecycle</td>
<td>106</td>
</tr>
<tr>
<td>7.5.2. The POM</td>
<td>114</td>
</tr>
<tr>
<td>7.5.3. Profiles</td>
<td>123</td>
</tr>
<tr>
<td>7.5.4. Repositories</td>
<td>134</td>
</tr>
<tr>
<td>7.5.5. Standard Directory Layout</td>
<td>137</td>
</tr>
<tr>
<td>7.5.6. The Dependency Mechanism</td>
<td>138</td>
</tr>
<tr>
<td>7.5.7. Plugin Development</td>
<td>152</td>
</tr>
<tr>
<td>7.5.8. Configuring Plug-ins</td>
<td>153</td>
</tr>
<tr>
<td>7.5.9. The Plugin Registry</td>
<td></td>
</tr>
<tr>
<td>7.5.10. Plugin Prefix Resolution</td>
<td>164</td>
</tr>
<tr>
<td>7.5.11. Developing Ant Plugins</td>
<td></td>
</tr>
<tr>
<td>7.5.12. Developing Java Plugins</td>
<td>166</td>
</tr>
<tr>
<td>7.5.13. Creating a Site</td>
<td>176</td>
</tr>
<tr>
<td>7.5.14. Snippet Macro</td>
<td>181</td>
</tr>
<tr>
<td>7.5.15. What is an Archetype</td>
<td>183</td>
</tr>
<tr>
<td>7.5.16. Creating Archetypes</td>
<td>185</td>
</tr>
<tr>
<td>7.5.17. Relocation of Artifacts</td>
<td>188</td>
</tr>
<tr>
<td>7.5.18. Installing 3rd party JARs to Local Repository</td>
<td>190</td>
</tr>
<tr>
<td>7.5.19. Deploying 3rd party JARs to Remote Repository</td>
<td>191</td>
</tr>
<tr>
<td>7.5.20. Coping with Sun JARs</td>
<td></td>
</tr>
<tr>
<td>7.5.21. Remote repository access through authenticated HTTPS</td>
<td>192</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>7.5.22. Creating Assemblies</td>
<td>194</td>
</tr>
<tr>
<td>7.5.23. Configuring Archive Plugins</td>
<td>197</td>
</tr>
<tr>
<td>7.5.24. Configuring Maven</td>
<td>198</td>
</tr>
<tr>
<td>7.5.25. Mirror Settings</td>
<td>201</td>
</tr>
<tr>
<td>7.5.26. Deployment and Security Settings</td>
<td>204</td>
</tr>
<tr>
<td>7.5.26. Generating Sources</td>
<td>205</td>
</tr>
<tr>
<td>7.5.26. Working with Manifests</td>
<td>206</td>
</tr>
<tr>
<td>7.5.26. Maven Classloading</td>
<td>207</td>
</tr>
<tr>
<td>7.5.26. Using Multiple Modules in a Build</td>
<td>209</td>
</tr>
<tr>
<td>7.5.26. Using Multiple Repositories</td>
<td>210</td>
</tr>
<tr>
<td>7.5.26. Using Proxies</td>
<td>212</td>
</tr>
<tr>
<td>7.5.26. Using the Release Plugin</td>
<td>213</td>
</tr>
<tr>
<td>7.5.26. Using Ant with Maven</td>
<td>218</td>
</tr>
<tr>
<td>7.5.26. Using Modello</td>
<td>220</td>
</tr>
<tr>
<td>7.5.26. Using Extensions</td>
<td>226</td>
</tr>
<tr>
<td>7.5.26. Building For Different Environments with Maven</td>
<td>227</td>
</tr>
<tr>
<td>7.5.26. Using Toolchains</td>
<td>230</td>
</tr>
<tr>
<td>7.5.26. Encrypting passwords in settings.xml</td>
<td>233</td>
</tr>
<tr>
<td>7.5.26. Reusable Test JARs</td>
<td>237</td>
</tr>
<tr>
<td>8. Plugin Developer Centre</td>
<td>239</td>
</tr>
<tr>
<td>8.1. Testing your Plugin</td>
<td>240</td>
</tr>
<tr>
<td>8.2. Documenting your Plugin</td>
<td>244</td>
</tr>
<tr>
<td>8.3. Common Bugs and Pitfalls</td>
<td>245</td>
</tr>
<tr>
<td>8.4. Mojo API</td>
<td>252</td>
</tr>
<tr>
<td>9. Maven Repository Centre</td>
<td>266</td>
</tr>
<tr>
<td>9.2. Guide to uploading artifacts</td>
<td></td>
</tr>
<tr>
<td>10. Maven Developer Centre</td>
<td>269</td>
</tr>
<tr>
<td>10.1. Developing Maven</td>
<td>271</td>
</tr>
<tr>
<td>10.2. Building Maven</td>
<td>274</td>
</tr>
<tr>
<td>10.3. Committer Environment</td>
<td>275</td>
</tr>
<tr>
<td>10.4. Committer Settings</td>
<td>276</td>
</tr>
<tr>
<td>10.5. Maven Code Style And Conventions</td>
<td>277</td>
</tr>
<tr>
<td>10.6. Maven JIRA Convention</td>
<td>282</td>
</tr>
<tr>
<td>10.7. Making GPG Keys</td>
<td>284</td>
</tr>
<tr>
<td>10.8. Release Process</td>
<td>287</td>
</tr>
<tr>
<td>10.9. Deploy Maven Current References</td>
<td></td>
</tr>
<tr>
<td>11. External Resources</td>
<td>287</td>
</tr>
<tr>
<td>12. Project Reports</td>
<td>289</td>
</tr>
<tr>
<td>12.1. Team</td>
<td>289</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>12.2. Mailing Lists</td>
<td>298</td>
</tr>
<tr>
<td>12.3. CI Management</td>
<td>299</td>
</tr>
</tbody>
</table>
1 What is Maven?

1.1 Introduction
Maven, a Yiddish word meaning accumulator of knowledge, began as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects, each with their own Ant build files, that were all slightly different. JARs were checked into CVS. We wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information, and a way to share JARs across several projects.

The result is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

1.2 Maven's Objectives
Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal, Maven deals with several areas of concern:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Encouraging better development practices

1.2.1 Making the build process easy
While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does shield developers from many details.

1.2.2 Providing a uniform build system
Maven builds a project using its project object model (POM) and a set of plugins. Once you familiarize yourself with one Maven project, you know how all Maven projects build. This saves time when navigating many projects.

1.2.3 Providing quality project information
Maven provides useful project information that is in part taken from your POM and in part generated from your project's sources. For example, Maven can provide:

- Change log created directly from source control
- Cross referenced sources
- Mailing lists managed by the project
- Dependencies used by the project
- Unit test reports including coverage

Third party code analysis products also provide Maven plugins that add their reports to the standard information given by Maven.

1.2.4 Providing guidelines for best practices development
Maven aims to gather current principles for best practices development and make it easy to guide a project in that direction.
For example, specification, execution, and reporting of unit tests are part of the normal build cycle using Maven. Current unit testing best practices were used as guidelines:

- Keeping test source code in a separate, but parallel source tree
- Using test case naming conventions to locate and execute tests
- Having test cases setup their environment instead of customizing the build for test preparation

Maven also assists in project workflow such as release and issue management. Maven also suggests some guidelines on how to layout your project's directory structure. Once you learn the layout, you can easily navigate other projects that use Maven.

While takes an opinionated approach to project layout, some projects may not fit with this structure for historical reasons. While Maven is designed to be flexible to the needs of different projects, it cannot cater to every situation without compromising its objectives.

If your project has an unusual build structure that cannot be reorganized, you may have to forgo some features or the use of Maven altogether.

### 1.3 What is Maven Not?

You might have heard some of the following things about Maven:

- Maven is a site and documentation tool
- Maven extends Ant to let you download dependencies
- Maven is a set of reusable Ant scriptlets

While Maven does these things, as you can read above in the “What is Maven?” section, these are not the only features Maven has, and its objectives are quite different.
2 Features

2.1 Feature Summary

The following are the key features of Maven in a nutshell:

- Simple project setup that follows best practices - get a new project or module started in seconds
- Consistent usage across all projects - means no ramp up time for new developers coming onto a project
- Superior dependency management including automatic updating, dependency closures (also known as transitive dependencies)
- Able to easily work with multiple projects at the same time
- A large and growing repository of libraries and metadata to use out of the box, and arrangements in place with the largest Open Source projects for real-time availability of their latest releases
- Extensible, with the ability to easily write plugins in Java or scripting languages
- Instant access to new features with little or no extra configuration
- Ant tasks for dependency management and deployment outside of Maven
- Model based builds: Maven is able to build any number of projects into predefined output types such as a JAR, WAR, or distribution based on metadata about the project, without the need to do any scripting in most cases.
- Coherent site of project information: Using the same metadata as for the build process, Maven is able to generate a web site or PDF including any documentation you care to add, and adds to that standard reports about the state of development of the project. Examples of this information can be seen at the bottom of the left-hand navigation of this site under the "Project Information" and "Project Reports" submenus.
- Release management and distribution publication: Without much additional configuration, Maven will integrate with your source control system (such as Subversion or Git) and manage the release of a project based on a certain tag. It can also publish this to a distribution location for use by other projects. Maven is able to publish individual outputs such as a JAR, an archive including other dependencies and documentation, or as a source distribution.
- Dependency management: Maven encourages the use of a central repository of JARs and other dependencies. Maven comes with a mechanism that your project's clients can use to download any JARs required for building your project from a central JAR repository much like Perl's CPAN. This allows users of Maven to reuse JARs across projects and encourages communication between projects to ensure that backward compatibility issues are dealt with.
3 FAQ

3.1 Frequently Asked Technical Questions

1. How do I prevent "[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!"
2. How do I prevent including JARs in WEB-INF/lib? I need a "compile only" scope!
3. How do I list available plugins?
4. How do I determine what version of a plugin I am using?
5. How can I use Ant tasks in a Maven build?
6. How can I use Maven features in an Ant build?
7. How do I set up Maven so it will compile with a target and source JVM of my choice?
8. Is it possible to create my own directory structure?
9. Where is the source code? I couldn't seem to find a link anywhere on the Maven site.
10. Maven can't seem to download the dependencies. Is my installation correct?
11. I have a jar that I want to put into my local repository. How can I copy it in?
12. How do I unsubscribe from Maven mailing lists?
13. How do I skip the tests?
14. How can I run a single unit test?
15. Handle special characters in site
16. Maven compiles my test classes but doesn't run them?
17. Where are Maven SNAPSHOT artifacts?
18. Where are the Maven XSD schemas?
19. Maven doesn't work, how do I get help?
20. How to produce execution debug output or error messages?
21. What is a Mojo?
22. How to find dependencies on public Maven repositories?
23. Why is my Javadoc JAR built twice during release?

How do I prevent "[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!"

This or a similar warning is emitted by a plugin that processes plain text files but has not been configured to use a specific file encoding. So eliminating the warning is simply a matter of finding out which plugin emits it and how to configure the file encoding for it. This is as easy as adding the following property to your POM (or one of its parent POMs):

```
<project>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  ...
</project>
```

How do I prevent including JARs in WEB-INF/lib? I need a "compile only" scope!
The scope you should use for this is provided. This indicates to Maven that the dependency will be provided at run time by its container or the JDK, for example. Dependencies with this scope will not be passed on transitively, nor will they be bundled in a package such as a WAR, or included in the runtime classpath.

How do I list available plugins?
The "Available Plugins" page lists them and provides additional information. See https://maven.apache.org/plugins

How do I determine what version of a plugin I am using?
You can use the Maven Help Plugin's describe goal. For example, to find out the version of the install plugin:

mvn -Dplugin=install help:describe

Note that you must give the plugin prefix as the argument to plugin, not it's artifact ID.

How can I use Ant tasks in a Maven build?
There are currently 2 alternatives:
- For use in a plugin written in Java, Beashell or other Java-like scripting language, you can construct the Ant tasks using the instructions given in the Ant documentation
- If you have very small amounts of Ant script specific to your project, you can use the AntRun plugin.

How can I use Maven features in an Ant build?
The Maven Ant Tasks allow many of the features of Maven, such as dependency management and repository deployment, to be used in an Ant build.

How do I set up Maven so it will compile with a target and source JVM of my choice?
You must configure the source and target parameters in your pom. For example, to set the source and target JVM to 7, you should have in your pom:

```xml
<project>
...
<properties>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```
Or if a parent pom overrides for compiler plugin default values and you can't fix it, you'll have to explicitly force the values in the compiler plugin configuration:

```xml
<project>
...  
<build>
...  
<plugins>
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.3</version>
   <configuration>
      <source>1.7</source>
      <target>1.7</target>
   </configuration>
</plugin>
...  
</plugins>
...  
</build>
...  
</project>
```

Is it possible to create my own directory structure?

Absolutely yes!

By configuring `<sourceDirectory>`, `<resources>` and other elements of the `<build>` section.

In addition, you may need to change the plugin configuration if you are not using plugin defaults for their files/directories.

Where is the source code? I couldn't seem to find a link anywhere on the Maven site.

The source code can be found in our Subversion and Git repositories.

For more information, see Building Maven.

Maven can't seem to download the dependencies. Is my installation correct?

You most probably need to configure Maven to use a proxy. Please see the information on Configuring a proxy for information on how to configure your proxy for Maven.
I have a jar that I want to put into my local repository. How can I copy it in?

If you understand the layout of the Maven repository, you can copy the jar directly into where it is meant to go. Maven will find this file next time it is run.

If you are not confident about the layout of the Maven repository, then you can adapt the following command to load in your jar file, all on one line.

```
mvn install:install-file
-Dfile=<path-to-file>
-DgroupId=<group-id>
-DartifactId=<artifact-id>
-Dversion=<version>
-Dpackaging=<packaging>
-DgeneratePom=true
```

Where:
- `<path-to-file>`  the path to the file to load
- `<group-id>`      the group that the file should be registered under
- `<artifact-id>`   the artifact name for the file
- `<version>`       the version of the file
- `<packaging>`     the packaging of the file e.g. jar

This should load in the file into the Maven repository, renaming it as needed.

How do I unsubscribe from Maven mailing lists?

To unsubscribe from a Maven mailing list you simply send a message to

```
[mailing-list]-unsubscribe@maven.apache.org
```

So, if you have subscribed to users@maven.apache.org then you would send a message to users-unsubscribe@maven.apache.org in order to get off the list. People tend to have problems when they subscribe with one address and attempt to unsubscribe with another. So make sure that you are using the same address when unsubscribing that you used to subscribe before asking for help.

If you find you still cannot get off a list then send a message to [mailing-list]-help@maven.apache.org. These instructions are also appended to every message sent out on a maven mailing list ...

How do I skip the tests?

Add the parameter `-Dmaven.test.skip=true` or `-DskipTests=true` in the command line, depending on whether you want to skip test compilation and execution or only execution. See the example Skipping Tests in the Surefire Plugin's documentation for more details.
How can I run a single unit test?

Use the parameter `-Dtest=MyTest` at the command line. NB: do not specify the entire package (org.apache.x.y.MyTest)

Handle special characters in site

Configure your ide to use the correct encoding. With Eclipse, add –
`-Dfile.encoding=ISO-8859-1` in eclipse.ini file

Configure the reporting output encoding in your pom

```xml
<project>
  ...
  <properties>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
  ...
</project>
```

or if default encoding is overridden in a parent pom that you can't change, configure the site plugin explicitly:

```xml
<project>
  ...
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-site-plugin</artifactId>
    <version>3.6</version>
    <configuration>
      <outputEncoding>UTF-8</outputEncoding>
    </configuration>
  </plugin>
  ...
</project>
```

Configure the file encoding use by mvn. add to MAVEN_OPTS the encoding (same as the ide). This can be made with adding `MAVEN_OPTS="-Dfile.encoding=ISO-8859-1"` in `$HOME/.profile`

Maven compiles my test classes but doesn't run them?

Tests are run by the surefire plugin. The surefire plugin can be configured to run certain test classes and you may have unintentionally done so by specifying a value to `${test}`. Check your settings.xml and pom.xml for a property named "test" which would like this:

```xml
<project>
  ...
  <properties>
    <property>
      <name>test</name>
      ...
</project>
```
Where are Maven SNAPSHOT artifacts?

If you are trying to build a development version of Maven or plugins, you may need to access the Maven snapshot repositories.

You need to update your settings.xml file using the Guide to Plugin Snapshot Repositories

Where are the Maven XSD schemas?

The Maven XSD is located here and the Maven Settings XSD is located here.

Your favorite IDE probably supports XSD schema's for pom.xml and settings.xml editing. You need to specify the following:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   https://maven.apache.org/xsd/maven-4.0.0.xsd">
   ...
</project>
```

```
<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
   https://maven.apache.org/xsd/settings-1.0.0.xsd">
   ...
</settings>
```

Maven doesn't work, how do I get help?

We have compiled a list of available resources on the getting help page.
How to produce execution debug output or error messages?

You could call Maven with -X parameter or -e parameter. For more information, run:

```
mvn --help
```

What is a Mojo?

A mojo is a Maven plain Old Java Object. Each mojo is an executable goal in Maven, and a Maven plugin is a distribution of one or more related mojos.

How to find dependencies on public Maven repositories?

You could use the following search engines:

- https://search.maven.org
- https://repository.apache.org
- https://mvnrepository.com

Why is my Javadoc JAR built twice during release?

With MNG-5940 the release profile goal of the Maven Javadoc Plugin has been changed to jar-no-fork. Revise your configuration to avoid duplicate JAR upload.
4 Community Overview

4.1 The Maven Community

Maven, like any other open source project, relies heavily on the efforts of the entire user community to contribute improvements, report defects, communicate use cases, write documentation, and assist other users in need. This is a quick guide outlining what members of the Maven community can do to make the system work better for everyone.

4.1.1 Helping With Maven

There is already a comprehensive Guide to Helping With Maven. That guide focuses upon beginning as a supporter, with information on how to help the coding effort.

4.1.1.1 Commit Questions or Answers to the Maven User FAQ

If you find things which are not correct or could be explained in a better way or you simply miss things do not hesitate to contact the maven community via the users mailing list and tell us about it.

4.1.1.2 Help Log Defects in JIRA

Just as any other healthy project requires a quick turn-around on defects, and a transparent method of users to have their wishes heard, so too does Maven need your help. Refer to the Issue Management page.

4.1.1.3 Developers

For Maven developers, committers, PMC: there is a Developers Guide.

4.1.2 Being a Good Maven Citizen

The concept of a public repository built into the core architecture of Maven makes it necessarily community-centric. There are a few simple things that Maven users may do to help keep that community thriving.

4.1.2.1 Be a Kind Public Repository User

The best thing that a user can do is to set up their own remote repository mirror containing the projects needed: this is called a repository manager. This reduces strain on the Maven central repository, and allows new users to get acquainted with Maven easier and quicker. This is especially important for power-users and corporations. The incentive behind this is, controlling your own servers can give you desired level of security and more control over uptime, resulting in a better experience for your users. With that said, keep the following sentiment in mind:

DO NOT wget THE ENTIRE REPOSITORY!

Please take only the jars you need. We understand this is may entail more work, but grabbing more than 1.7 TiB of binaries really kills our servers.

4.1.3 User Gathering Spots

These are a few of the watering holes around which Maven users tend to gather.
4.1.3.1 Mailing Lists
Maven has a number of Mailing Lists, and the Maven User List is specifically dedicated to answering questions about all Maven things.

4.1.3.2 Slack
For people actively contributing to Maven, especially committers, there is the ASF Slack workspace available to discuss issues, solve problems and build community in real-time.
5 How to Contribute

5.1 Guide to helping with Maven
As with any open source project, there are several ways you can help:

- Join the mailing lists and answer other user's questions.
- Report bugs, feature requests and other issues in the issue management system.
- Build Maven for yourself, in order to fix bugs.
- Submit patches to reported issues (both those you find, or that others have filed)
  To ease your first contribution, we have a list of "up for grabs" issues, meaning that they should be easy to work on.
- test releases help test releases that are being voted on (see the dev@maven.apache.org mailing list for release votes)
- test snapshot plugins help test the latest development versions of plugins and report issues
- Help with the documentation by pointing out areas that are lacking or unclear, and if you can, submitting Pull Requests to correct it: use the "edit" button in the breadcrumb, just after the page title. You can also create appropriate issues by using the issue management system.

Your participation in the community is much appreciated!

5.2 Why Would I Want to Help?
There are several reasons these are good things.

- By answering other people's questions, you can learn more for yourself
- By submitting your own fixes, they get incorporated faster
- By reporting issues, you ensure that bugs don't get missed, or forgotten
- You are giving back to a community that has given you software for free

5.3 How do I Join the Project?
Projects at Apache operate under a meritocracy, meaning those that the developers notice participating to a high extent will be invited to join the project as a committer.

This is as much based on personality and ability to work with other developers and the community as it is with proven technical ability. Being unhelpful to other users, or obviously looking to become a committer for bragging rights and nothing else is frowned upon, as is asking to be made a committer without having contributed sufficiently to be invited.

5.4 Developers Conventions
There are a number of conventions used in the project, which contributors and developers alike should follow for consistency's sake.

- Maven Code Style And Convention
- Maven Jira Convention
- Maven Git Convention
- Releasing a Maven project

5.5 Resources for committers

- Developer Resources
• About the Apache Software Foundation
• Committer FAQ
• Apache Wiki
6 Getting Help

6.1 Getting Help

So something didn’t work as you expected it to? You think that Maven is broken. What should you do?

Here’s a list of actions that you can take:

6.1.1 You did check the documentation, didn’t you?

Apart from the central Maven site, each of our plugins has a website. Go to the plugins page and follow the link to the plugin you are having problems with.

6.1.2 Try the latest version of Maven or the plugin in question

Before you start intensive investigations on your problem, you should try to update Maven and/or the plugins in question to the latest stable release. After all, the issue you encounter might have been fixed already. To find out what is the latest stable release version, consult Maven’s download section and the plugin index.

6.1.3 Search the user-list archives

Someone else might have experienced the same problem as you before. A list of mail-archives can be found on mailing list index page. Please search one of them before going any further.

6.1.4 Ask on the user list

Our community is very helpful, just ask it the right way. See the references section, at the end of this page, for info on how to do that. Subscribe to the users-list and describe your problem there. Don’t expect to get an answer right away. Sometimes it takes a couple of days.

6.1.5 Submit an issue

If it turns out that there is indeed something wrong with Maven or one of the plugins, you should report it to our issue management system JIRA.

First of all you need to create an account in JIRA. This is so that we can communicate with you while we work together on the issue. Go to the sign up link to create an account if you don’t already have one.

6.1.5.1 Where?

If the problem is in one of the plugins, check the site of that plugin to get the correct link. Each plugin has its own section in JIRA, so using the correct link is important. Click on Project Information and then Issue Management. On that page you will find the correct link.

If the problem is in Maven itself you can find the appropriate link on the issue management page.

6.1.5.2 How?

Just describing the problem is not enough. It takes a developer a lot of time to make a usable POM to even attempt to assess the problem. Issues that states problems without something usable to try out will be closed as incomplete.
Please attach a working POM, or a set of POMs, that we can download and run. We appreciate reports, but if you don’t have something usable for us it’s incredibly hard for us to manage the issues. A POM goes a long way to helping us resolve problems.

Create a POM that can be used to verify that it is a bug. If your pom uses plugins, make sure that you have specified the version for each and every plugin. If you don’t, then we might not be using the same version as you are when we test it.

What we like best are patches that fixes the problem. If you want to create a patch for an issue please read the [Maven Developer Guide](#) first.

### 6.1.6 References

- [How To Ask Questions The Smart Way](#)
- [How to Get Support from Open Source Mailing Lists](#)
7 Issue Management

7.1 Overview
Maven projects use Jira as issue tracking and project management application.

7.2 Issue Management
Maven is composed of nearly 100 parts, each one having its own Jira project or component: see the Maven Jira issues overview to get a summary of open issues.

Issues, bugs, and feature requests should be submitted to the following issue management systems depending on which component is involved:

- Maven “core”: https://issues.apache.org/jira/browse/MNG
- Maven Website: https://issues.apache.org/jira/browse/MNGSITE
- Maven Plugins: Please refer to the Available Plugins page
- Maven Shared Components: Please refer to the Shared Components page
- Doxia: https://issues.apache.org/jira/browse/DOXIA
- JXR: https://issues.apache.org/jira/browse/JXR
- SCM: https://issues.apache.org/jira/browse/SCM
- Wagon: https://issues.apache.org/jira/browse/WAGON

or for Central Repository: https://issues.sonatype.org/browse/MVNCENTRAL
8 Source Repository

8.1 Source Repository

Maven projects use Git or Subversion to manage their source code: decisions to stay with Subversion or move to Git are tracked on Maven's Wiki.

Instructions on Subversion use can be found in the online book Version Control with Subversion. Instructions on Git use can be found in the online book Pro Git. Instructions for using the Apache Software Foundation Git repositories are at https://git-wip-us.apache.org.

8.1.1 Full Maven Sources

As described in the next paragraphs, Maven full source code is dispatched in more than 100 Git repos: Maven core, but also plugins or components, skins, a few svn2git read-only mirrors...

To check out full Maven source code easily, we provide a simple way using additional Google repo tool and an additional Git repository for tool’s manifest:

Apache Maven full source code [https://gitbox.apache.org/repos/asf/maven-sources.git](https://gitbox.apache.org/repos/asf/maven-sources.git) (GitHub mirror)

1. Install a git client if needed and the Google Repo tool (see manual install instructions).
2. Check out a new repo workspace and prepare master branch:
   repo init -u https://gitbox.apache.org/repos/asf/maven-sources.git
   repo sync
   repo start master --all
3. In your IDE, import the projects you’re interested in from the repo workspace. Or directly build with command line the component you want.
Each component has its own Jira project or component for issue tracking: see the Issue Management report to get a summary.

### 8.1.3 Maven Site

The sources for this site are available in a distinct Git repository:

<table>
<thead>
<tr>
<th>Apache Maven Site</th>
<th><a href="https://gitbox.apache.org/repos/asf/maven-site.git">https://gitbox.apache.org/repos/asf/maven-site.git</a> (GitHub mirror)</th>
<th>Jira MNGSITE</th>
</tr>
</thead>
</table>

### 8.1.4 Maven Core

The Git repository for Maven contains a master branch which is the current development version. There is also a branch for maven-2.2.X or maven-3.0.x. In addition, the integration tests for the Maven core have their own repository.

<table>
<thead>
<tr>
<th>Apache Maven</th>
<th><a href="https://gitbox.apache.org/repos/asf/maven.git">https://gitbox.apache.org/repos/asf/maven.git</a> (GitHub mirror)</th>
<th>Jira MNG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Maven Core ITs</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-integration-testing.git">https://gitbox.apache.org/repos/asf/maven-integration-testing.git</a> (GitHub mirror)</td>
<td>Jira MNGITS</td>
</tr>
</tbody>
</table>

### 8.1.5 Other Components

The source repositories for the various plugins are in Git, listed in the documentation of the respective plugin, reachable via the plugin index.

There are also many shared components and subsystems with their own source repositories, mainly in Git, some in Subversion.

#### 8.1.5.1 Components in Git

The components in Git are:

<table>
<thead>
<tr>
<th>Apache Maven Archetype</th>
<th><a href="https://gitbox.apache.org/repos/asf/maven-archetype.git">https://gitbox.apache.org/repos/asf/maven-archetype.git</a> (GitHub mirror)</th>
<th>Jira MARCHETYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Maven Archetypes</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-archetypes.git">https://gitbox.apache.org/repos/asf/maven-archetypes.git</a> (GitHub mirror)</td>
<td>Jira MARCHETYPES</td>
</tr>
<tr>
<td>Apache Maven Artifact Resolver</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-resolver.git">https://gitbox.apache.org/repos/asf/maven-resolver.git</a> (GitHub mirror)</td>
<td>Jira MRESOLVER</td>
</tr>
<tr>
<td>Apache Maven Artifact Resolver Ant Tasks</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-resolver-ant-tasks.git">https://gitbox.apache.org/repos/asf/maven-resolver-ant-tasks.git</a> (GitHub mirror)</td>
<td>Jira MRESOLVER</td>
</tr>
<tr>
<td>Apache Maven Distribution Checking Tool</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-dist-tool.git">https://gitbox.apache.org/repos/asf/maven-dist-tool.git</a> (GitHub mirror)</td>
<td>Jira MINDEXER</td>
</tr>
<tr>
<td>Apache Maven Enforcer</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-enforcer.git">https://gitbox.apache.org/repos/asf/maven-enforcer.git</a> (GitHub mirror)</td>
<td>Jira MENFORCER</td>
</tr>
<tr>
<td>Apache Maven JXR</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jxr.git">https://gitbox.apache.org/repos/asf/maven-jxr.git</a> (GitHub mirror)</td>
<td>Jira JXR</td>
</tr>
<tr>
<td>Apache Maven Indexer</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-indexer.git">https://gitbox.apache.org/repos/asf/maven-indexer.git</a> (GitHub mirror)</td>
<td>Jira MINDEXER</td>
</tr>
<tr>
<td>Apache Maven Plugin Testing</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-plugin-testing.git">https://gitbox.apache.org/repos/asf/maven-plugin-testing.git</a> (GitHub mirror)</td>
<td>Jira MPLUGINTESTING</td>
</tr>
<tr>
<td>Apache Maven Plugin Tools</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-plugin-tools.git">https://gitbox.apache.org/repos/asf/maven-plugin-tools.git</a> (GitHub mirror)</td>
<td>Jira MPLUGINS</td>
</tr>
</tbody>
</table>

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
## Source Repository

<table>
<thead>
<tr>
<th>Source</th>
<th>Repository</th>
<th>Jenkins Job</th>
<th>Jira Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Maven Release</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-release.git">https://gitbox.apache.org/repos/asf/maven-release.git</a></td>
<td>(GitHub mirror)</td>
<td>Jira MRELEASE</td>
</tr>
<tr>
<td>Apache Maven SCM</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-scm.git">https://gitbox.apache.org/repos/asf/maven-scm.git</a></td>
<td>(GitHub mirror)</td>
<td>Jira SCM</td>
</tr>
<tr>
<td>Apache Maven Surefire</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-surefire.git">https://gitbox.apache.org/repos/asf/maven-surefire.git</a></td>
<td>(GitHub mirror)</td>
<td>Jira SUREFIRE</td>
</tr>
<tr>
<td>Apache Maven Wagon</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-wagon.git">https://gitbox.apache.org/repos/asf/maven-wagon.git</a></td>
<td>(GitHub mirror)</td>
<td>Jira WAGON</td>
</tr>
</tbody>
</table>

### Plugins

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Repository</th>
<th>GitHub Mirror</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Maven ACR Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-acr-plugin.git">https://gitbox.apache.org/repos/asf/maven-acr-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Ant Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-ant-plugin.git">https://gitbox.apache.org/repos/asf/maven-ant-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven AntRun Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-antrun-plugin.git">https://gitbox.apache.org/repos/asf/maven-antrun-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Assembly Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-assembly-plugin.git">https://gitbox.apache.org/repos/asf/maven-assembly-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Changelog Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-changelog-plugin.git">https://gitbox.apache.org/repos/asf/maven-changelog-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Changes Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-changes-plugin.git">https://gitbox.apache.org/repos/asf/maven-changes-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Checkstyle Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-checkstyle-plugin.git">https://gitbox.apache.org/repos/asf/maven-checkstyle-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Clean Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-clean-plugin.git">https://gitbox.apache.org/repos/asf/maven-clean-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Compiler Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-compiler-plugin.git">https://gitbox.apache.org/repos/asf/maven-compiler-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Dependency Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-dependency-plugin.git">https://gitbox.apache.org/repos/asf/maven-dependency-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Deploy Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-deploy-plugin.git">https://gitbox.apache.org/repos/asf/maven-deploy-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven DOAP Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-doap-plugin.git">https://gitbox.apache.org/repos/asf/maven-doap-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Documentation Checker Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-docck-plugin.git">https://gitbox.apache.org/repos/asf/maven-docck-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven EAR Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-ear-plugin.git">https://gitbox.apache.org/repos/asf/maven-ear-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven EJB Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-ejb-plugin.git">https://gitbox.apache.org/repos/asf/maven-ejb-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven GPG Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-gpg-plugin.git">https://gitbox.apache.org/repos/asf/maven-gpg-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Help Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-help-plugin.git">https://gitbox.apache.org/repos/asf/maven-help-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Install Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-install-plugin.git">https://gitbox.apache.org/repos/asf/maven-install-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Invoker Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-invoker-plugin.git">https://gitbox.apache.org/repos/asf/maven-invoker-plugin.git</a></td>
<td>(GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven JAR Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jar-plugin.git">https://gitbox.apache.org/repos/asf/maven-jar-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Jarsigner Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jarsigner-plugin.git">https://gitbox.apache.org/repos/asf/maven-jarsigner-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Javadoc Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-javadoc-plugin.git">https://gitbox.apache.org/repos/asf/maven-javadoc-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven JDepRScan Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jdeprscan-plugin.git">https://gitbox.apache.org/repos/asf/maven-jdeprscan-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven JDeps Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jdeps-plugin.git">https://gitbox.apache.org/repos/asf/maven-jdeps-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven JLink Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jlink-plugin.git">https://gitbox.apache.org/repos/asf/maven-jlink-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven JMod Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jmod-plugin.git">https://gitbox.apache.org/repos/asf/maven-jmod-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Linkcheck Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-linkcheck-plugin.git">https://gitbox.apache.org/repos/asf/maven-linkcheck-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Patch Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-patch-plugin.git">https://gitbox.apache.org/repos/asf/maven-patch-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven PDF Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-resources-plugin.git">https://gitbox.apache.org/repos/asf/maven-resources-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven PMD Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-pmd-plugin.git">https://gitbox.apache.org/repos/asf/maven-pmd-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Project Info Reports Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-project-info-reports-plugin.git">https://gitbox.apache.org/repos/asf/maven-project-info-reports-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven RAR Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-rar-plugin.git">https://gitbox.apache.org/repos/asf/maven-rar-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Remote Resources Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-remote-resources-plugin.git">https://gitbox.apache.org/repos/asf/maven-remote-resources-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Repository Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-repository-plugin.git">https://gitbox.apache.org/repos/asf/maven-repository-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Resources Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-resources-plugin.git">https://gitbox.apache.org/repos/asf/maven-resources-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven SCM Publish Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-sccm-publish-plugin.git">https://gitbox.apache.org/repos/asf/maven-sccm-publish-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Scripting Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-scripting-plugin.git">https://gitbox.apache.org/repos/asf/maven-scripting-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Shade Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-shade-plugin.git">https://gitbox.apache.org/repos/asf/maven-shade-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Site Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-site-plugin.git">https://gitbox.apache.org/repos/asf/maven-site-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Source Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-source-plugin.git">https://gitbox.apache.org/repos/asf/maven-source-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Stage Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-stage-plugin.git">https://gitbox.apache.org/repos/asf/maven-stage-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Toolchains Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-toolchains-plugin.git">https://gitbox.apache.org/repos/asf/maven-toolchains-plugin.git</a> (GitHub mirror)</td>
<td></td>
</tr>
</tbody>
</table>

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Source Repository</th>
<th>URL</th>
<th>GitHub mirror</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Maven Verifier Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-verifier-plugin.git">https://gitbox.apache.org/repos/asf/maven-verifier-plugin.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven WAR Plugin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-war-plugin.git">https://gitbox.apache.org/repos/asf/maven-war-plugin.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Parent POMs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Maven Parent POMs</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-parent.git">https://gitbox.apache.org/repos/asf/maven-parent.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Shared Components</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Maven Archiver</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-archiver.git">https://gitbox.apache.org/repos/asf/maven-archiver.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Artifact Resolver</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-artifact-resolver.git">https://gitbox.apache.org/repos/asf/maven-artifact-resolver.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Artifact Transfer</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-artifact-transfer.git">https://gitbox.apache.org/repos/asf/maven-artifact-transfer.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Dependency Tree</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-dependency-tree.git">https://gitbox.apache.org/repos/asf/maven-dependency-tree.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Downloader</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-downloader.git">https://gitbox.apache.org/repos/asf/maven-downloader.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Filtering</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-filtering.git">https://gitbox.apache.org/repos/asf/maven-filtering.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Invoker</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-invoker.git">https://gitbox.apache.org/repos/asf/maven-invoker.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Jarsigner</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-jarsigner.git">https://gitbox.apache.org/repos/asf/maven-jarsigner.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Mapping</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-mapping.git">https://gitbox.apache.org/repos/asf/maven-mapping.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven OSGi</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-osgi.git">https://gitbox.apache.org/repos/asf/maven-osgi.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Project Utils</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-project-utils.git">https://gitbox.apache.org/repos/asf/maven-project-utils.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Reporting API</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-reporting-api.git">https://gitbox.apache.org/repos/asf/maven-reporting-api.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Reporting Executor</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-reporting-exec.git">https://gitbox.apache.org/repos/asf/maven-reporting-exec.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Repository Builder</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-repository-builder.git">https://gitbox.apache.org/repos/asf/maven-repository-builder.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Runtime</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-runtime.git">https://gitbox.apache.org/repos/asf/maven-runtime.git</a></td>
<td>GitHub mirror</td>
</tr>
<tr>
<td>Apache Maven Shared Incremental</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-shared-incremental.git">https://gitbox.apache.org/repos/asf/maven-shared-incremental.git</a></td>
<td>GitHub mirror</td>
</tr>
</tbody>
</table>
### Source Repository

<table>
<thead>
<tr>
<th>Component</th>
<th>Repository URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Maven Shared IO</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-shared-io.git">https://gitbox.apache.org/repos/asf/maven-shared-io.git</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Shared Jar</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-shared-jar.git">https://gitbox.apache.org/repos/asf/maven-shared-jar.git</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Shared Resources</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-shared-resources.git">https://gitbox.apache.org/repos/asf/maven-shared-resources.git</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Shared Utils</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-shared-utils.git">https://gitbox.apache.org/repos/asf/maven-shared-utils.git</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Verifier</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-verifier.git">https://gitbox.apache.org/repos/asf/maven-verifier.git</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Skins</td>
<td></td>
</tr>
<tr>
<td>Apache Maven Default Skin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-default-skin.git">https://gitbox.apache.org/repos/asf/maven-default-skin.git</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Apache Maven Fluido Skin</td>
<td><a href="https://gitbox.apache.org/repos/asf/maven-fluido-skin.git">https://gitbox.apache.org/repos/asf/maven-fluido-skin.git</a> (GitHub mirror)</td>
</tr>
</tbody>
</table>

8.1.5.2 Components in Subversion

Everything in Subversion can be checked-out from a single entry point, referencing each part through `svn:externals`

```
https://svn.apache.org/repos/asf/maven/trunk/
```

You can also check out every component separately. The components in Subversion are:

<table>
<thead>
<tr>
<th>Component</th>
<th>Repository URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maven Project (mainly KEYS)</td>
<td><a href="https://svn.apache.org/repos/asf/maven/project/">https://svn.apache.org/repos/asf/maven/project/</a></td>
</tr>
<tr>
<td>Apache Resource Bundles</td>
<td><a href="https://svn.apache.org/repos/asf/maven/resources/trunk/">https://svn.apache.org/repos/asf/maven/resources/trunk/</a> (GitHub mirror)</td>
</tr>
<tr>
<td>Maven Sandbox</td>
<td><a href="https://svn.apache.org/repos/asf/maven/sandbox/trunk/">https://svn.apache.org/repos/asf/maven/sandbox/trunk/</a> (GitHub mirror)</td>
</tr>
<tr>
<td>A variety of other subsystems (including obsolete trees replaced by git)</td>
<td><a href="https://svn.apache.org/repos/asf/maven/">https://svn.apache.org/repos/asf/maven/</a></td>
</tr>
</tbody>
</table>
9 Running Maven

9.1 Building a Project with Maven

The vast majority of Maven-built projects can be built with the following command:

```
mvn verify
```

This command tells Maven to build all the modules, and to check if all integration tests succeeded (when any was defined).

That's it! If you look in the `target` subdirectory, you should find the build output and the final library or application that was being built.

**Note:** Some projects have multiple modules, so the library or application you are looking for may be in a module subdirectory.

While this will build most projects and Maven encourages this standard convention, builds can be customisable. If this does not suffice, please consult the project's documentation.

9.1.1 More than just the Build

Maven can do more than just build software - it can assist with testing, run web applications and produce reports on projects, as well as any number of other tasks provided by plug-ins.

9.1.2 When Things go Wrong

The following are some common problems when building with Maven, and how to resolve them.

9.1.2.1 Missing Dependencies

A missing dependency presents with an error like the following:

```
INFO] Failed to resolve artifact.
Missing:
---
1) junit:junit:jar:3.8.1
   Try downloading the file manually from the project website.
   Then, install it using the command:
   mvn install:install-file -DgroupId=junit -DartifactId=junit \
   -Dversion=3.8.1 -Dpackaging=jar -Dfile=/path/to/file
   Path to dependency:
   1) org.apache.maven:maven:pom:2.1-SNAPSHOT
   2) junit:junit:jar:3.8.1
---
1 required artifact is missing.
for artifact:
   org.apache.maven:maven:pom:2.1-SNAPSHOT
from the specified remote repositories:
   central (https://repo.maven.apache.org/maven2)
```

To resolve this issue, it depends on what the dependency is and why it is missing. The most common cause is because it can not be redistributed from the repository and must be manually installed using the instructions given in the message. This is most common with some older JARs from Sun (usually `javax.*` group IDs), and is further documented in the [Guide to Coping with Sun JARs](https://maven.apache.org/guide/pom-sun-jar.html).

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
You can check the list of repositories at the end of the error to ensure that the expected ones are listed - it may be that the project requires an alternative repository that has not been declared properly or is not accessible with your Maven configuration.

In other cases, it may be an incorrectly declared dependency (like the typo in the example above) which the project would need to fix, like a compilation error.

9.1.2.2 Inconsistent output

Most plugins are optimized to know if they have to execute their task. In some cases, the output can be polluted from a previous build and the end result is not what you expected. In such rare situations, you can call the clean phase which means: remove the output directory. You can also call it as `mvn clean verify` which means: first clean up the output directory, next build the project and verify the outcome.
10 Maven Plugins

10.1 Available Plugins

Maven is - at its heart - a plugin execution framework; all work is done by plugins. Looking for a specific goal to execute? This page lists the core plugins and others. There are the build and the reporting plugins:

- **Build plugins** will be executed during the build and they should be configured in the `<build/>` element from the POM.
- **Reporting plugins** will be executed during the site generation and they should be configured in the `<reporting/>` element from the POM. Because the result of a Reporting plugin is part of the generated site, Reporting plugins should be both internationalized and localized. You can read more about the localization of our plugins and how you can help.

10.1.1 Supported By The Maven Project

To see the most up-to-date list browse the Maven repository, specifically the [org/apache/maven/plugins](https://search.maven.org/solrsearch/search servlet?command=displayIdentity&identityId=org.apache.maven.plugins) subfolder. *(Plugins are organized according to a directory structure that resembles the standard Java package naming convention)*

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Type*</th>
<th>Version</th>
<th>Release Date</th>
<th>Description</th>
<th>Source Repository</th>
<th>Issue Tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core plugins</td>
<td></td>
<td></td>
<td></td>
<td>Plugins corresponding to default core phases (ie. clean, compile). They may have multiple goals as well.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>clean</td>
<td>B</td>
<td>3.1.0</td>
<td>2018-04-14</td>
<td>Clean up after the build.</td>
<td>Git / GitHub</td>
<td>Jira MCLEAN</td>
</tr>
<tr>
<td>compiler</td>
<td>B</td>
<td>3.8.1</td>
<td>2019-04-28</td>
<td>Compiles Java sources.</td>
<td>Git / GitHub</td>
<td>Jira MCOMPILER</td>
</tr>
<tr>
<td>deploy</td>
<td>B</td>
<td>3.0.0-M1</td>
<td>2018-09-23</td>
<td>Deploy the built artifact to the remote repository.</td>
<td>Git / GitHub</td>
<td>Jira MDEPLOY</td>
</tr>
<tr>
<td>failsafe</td>
<td>B</td>
<td>3.0.0-M5</td>
<td>2020-06-17</td>
<td>Run the JUnit integration tests in an isolated classloader.</td>
<td>Git / GitHub</td>
<td>Jira SUREFIRE</td>
</tr>
<tr>
<td>install</td>
<td>B</td>
<td>3.0.0-M1</td>
<td>2018-09-23</td>
<td>Install the built artifact into the local repository.</td>
<td>Git / GitHub</td>
<td>Jira MINSTALL</td>
</tr>
<tr>
<td>Plugin</td>
<td>Version</td>
<td>Date</td>
<td>Description</td>
<td>Git</td>
<td>Jira</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>resources</td>
<td>B</td>
<td>3.2.0</td>
<td>Copy the resources to the output directory for including in the JAR.</td>
<td>GitHub</td>
<td>MRESOURCES</td>
<td></td>
</tr>
<tr>
<td>site</td>
<td>B</td>
<td>3.9.1</td>
<td>Generate a site for the current project.</td>
<td>GitHub</td>
<td>MSITE</td>
<td></td>
</tr>
<tr>
<td>surefire</td>
<td>B</td>
<td>3.0.0-M5</td>
<td>Run the JUnit unit tests in an isolated classloader.</td>
<td>GitHub</td>
<td>SUREFIRE</td>
<td></td>
</tr>
<tr>
<td>verifier</td>
<td>B</td>
<td>1.1</td>
<td>Useful for integration tests - verifies the existence of certain conditions.</td>
<td>GitHub</td>
<td>MVERIFIER</td>
<td></td>
</tr>
</tbody>
</table>

**Packaging types/tools**

These plugins relate to packaging respective artifact types.

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Git</th>
<th>Jira</th>
</tr>
</thead>
<tbody>
<tr>
<td>ear</td>
<td>B</td>
<td>3.2.0</td>
<td>Generate an EAR from the current project.</td>
<td>GitHub</td>
<td>MEAR</td>
</tr>
<tr>
<td>ejb</td>
<td>B</td>
<td>3.1.0</td>
<td>Build an EJB (and optional client) from the current project.</td>
<td>GitHub</td>
<td>MEJB</td>
</tr>
<tr>
<td>jar</td>
<td>B</td>
<td>3.2.0</td>
<td>Build a JAR from the current project.</td>
<td>GitHub</td>
<td>MJAR</td>
</tr>
<tr>
<td>rar</td>
<td>B</td>
<td>2.4</td>
<td>Build a RAR from the current project.</td>
<td>GitHub</td>
<td>MRAR</td>
</tr>
<tr>
<td>war</td>
<td>B</td>
<td>3.3.2</td>
<td>Build a WAR from the current project.</td>
<td>GitHub</td>
<td>MWAR</td>
</tr>
<tr>
<td>Plugin</td>
<td>Group</td>
<td>Version</td>
<td>Date</td>
<td>Description</td>
<td>Source</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>---------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>app-client/acr</td>
<td>B</td>
<td>3.1.0</td>
<td>2018-06-19</td>
<td>Build a JavaEE application client from the current project.</td>
<td>Git / GitHub</td>
</tr>
<tr>
<td>shade</td>
<td>B</td>
<td>3.2.4</td>
<td>2020-05-31</td>
<td>Build an Uber-JAR from the current project, including dependencies.</td>
<td>Git / GitHub</td>
</tr>
<tr>
<td>source</td>
<td>B</td>
<td>3.2.1</td>
<td>2019-12-21</td>
<td>Build a source-JAR from the current project.</td>
<td>Git / GitHub</td>
</tr>
<tr>
<td>jlink</td>
<td>B</td>
<td>3.1.0</td>
<td>2020-12-28</td>
<td>Build Java Run Time Image.</td>
<td>Git / GitHub</td>
</tr>
<tr>
<td>jmod</td>
<td>B</td>
<td>3.0.0-alpha-1</td>
<td>2017-09-17</td>
<td>Build Java JMod files.</td>
<td>Git / GitHub</td>
</tr>
</tbody>
</table>

**Reporting plugins**

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Group</th>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Source</th>
<th>Jira</th>
</tr>
</thead>
<tbody>
<tr>
<td>changelog</td>
<td>R</td>
<td>2.3</td>
<td>2014-06-24</td>
<td>Generate a list of recent changes from your SCM.</td>
<td>Git / GitHub</td>
<td>MCHANGELOG</td>
</tr>
<tr>
<td>changes</td>
<td>B+R</td>
<td>2.12.1</td>
<td>2016-11-01</td>
<td>Generate a report from an issue tracker or a change document.</td>
<td>Git / GitHub</td>
<td>MCHANGES</td>
</tr>
<tr>
<td>checkstyle</td>
<td>B+R</td>
<td>3.1.2</td>
<td>2021-01-30</td>
<td>Generate a Checkstyle report.</td>
<td>Git / GitHub</td>
<td>MCHECKSTYLE</td>
</tr>
<tr>
<td>doap</td>
<td>B</td>
<td>1.2</td>
<td>2015-03-17</td>
<td>Generate a Description of a Project (DOAP) file from a POM.</td>
<td>Git / GitHub</td>
<td>MDOAP</td>
</tr>
<tr>
<td>Plugin</td>
<td>Status</td>
<td>Version</td>
<td>Date</td>
<td>Description</td>
<td>Git / GitHub</td>
<td>Jira</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
<td>---------</td>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>docck</strong></td>
<td>B</td>
<td>1.1</td>
<td>2015-04-03</td>
<td>Documentation checker plugin.</td>
<td>Git / GitHub</td>
<td>MDOCK</td>
</tr>
<tr>
<td><strong>javadoc</strong></td>
<td>B+R</td>
<td>3.3.1</td>
<td>2021-09-07</td>
<td>Generate Javadoc for the project.</td>
<td>Git / GitHub</td>
<td>MJAVADOC</td>
</tr>
<tr>
<td><strong>jdeps</strong></td>
<td>B</td>
<td>3.1.2</td>
<td>2019-06-12</td>
<td>Run JDK's JDep's tool on the project.</td>
<td>Git / GitHub</td>
<td>MJDEPS</td>
</tr>
<tr>
<td><strong>jxr</strong></td>
<td>R</td>
<td>3.1.1</td>
<td>2021-04-22</td>
<td>Generate a source cross reference.</td>
<td>Git / GitHub</td>
<td>JXR</td>
</tr>
<tr>
<td><strong>linkcheck</strong></td>
<td>R</td>
<td>1.2</td>
<td>2014-10-08</td>
<td>Generate a Linkcheck report of your project's documentation.</td>
<td>Git / GitHub</td>
<td>MLINKCHECK</td>
</tr>
<tr>
<td><strong>pmd</strong></td>
<td>B+R</td>
<td>3.15.0</td>
<td>2021-09-06</td>
<td>Generate a PMD report.</td>
<td>Git / GitHub</td>
<td>MPMD</td>
</tr>
<tr>
<td><strong>project-info-reports</strong></td>
<td>R</td>
<td>3.1.2</td>
<td>2021-04-25</td>
<td>Generate standard project reports.</td>
<td>Git / GitHub</td>
<td>MPIR</td>
</tr>
<tr>
<td><strong>surefire-report</strong></td>
<td>R</td>
<td>3.0.0-M5</td>
<td>2020-06-17</td>
<td>Generate a report based on the results of unit tests.</td>
<td>Git / GitHub</td>
<td>SUREFIRE</td>
</tr>
</tbody>
</table>

**Tools**

These are miscellaneous tools available through Maven by default.

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Status</th>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Git / GitHub</th>
<th>Jira</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>antrun</strong></td>
<td>B</td>
<td>3.0.0</td>
<td>2020-04-15</td>
<td>Run a set of ant tasks from a phase of the build.</td>
<td>Git / GitHub</td>
<td>MANTRUN</td>
</tr>
<tr>
<td><strong>artifact</strong></td>
<td>B</td>
<td>3.2.0</td>
<td>2021-11-30</td>
<td>Manage artifacts tasks like buildinfo.</td>
<td>Git / GitHub</td>
<td>MARTIFACT</td>
</tr>
<tr>
<td><strong>archetype</strong></td>
<td>B</td>
<td>3.2.0</td>
<td>2020-07-21</td>
<td>Generate a skeleton project structure from an archetype.</td>
<td>Git / GitHub</td>
<td>ARCHETYPE</td>
</tr>
<tr>
<td><strong>assembly</strong></td>
<td>B</td>
<td>3.3.0</td>
<td>2020-04-30</td>
<td>Build an assembly (distribution) of sources and/or binaries.</td>
<td>Git / GitHub</td>
<td>MASEMBLY</td>
</tr>
<tr>
<td>dependency</td>
<td>B+R</td>
<td>3.2.0</td>
<td>2021-06-12</td>
<td>Dependency manipulation (copy, unpack) and analysis.</td>
<td>Git / GitHub</td>
<td>Jira MDEPENDENCY</td>
</tr>
<tr>
<td>------------</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
<td>-----------------------------------------------------</td>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>enforcer</td>
<td>B</td>
<td>3.0.0</td>
<td>2021-07-30</td>
<td>Environmental constraint checking (Maven Version, JDK etc), User Custom Rule Execution.</td>
<td>Git / GitHub</td>
<td>Jira MENFORCER</td>
</tr>
<tr>
<td>gpg</td>
<td>B</td>
<td>3.0.1</td>
<td>2021-05-08</td>
<td>Create signatures for the artifacts and poms.</td>
<td>Git / GitHub</td>
<td>Jira MGPG</td>
</tr>
<tr>
<td>help</td>
<td>B</td>
<td>3.2.0</td>
<td>2019-04-16</td>
<td>Get information about the working environment for the project.</td>
<td>Git / GitHub</td>
<td>Jira MPH</td>
</tr>
<tr>
<td>invoker</td>
<td>B+R</td>
<td>3.2.2</td>
<td>2021-02-20</td>
<td>Run a set of Maven projects and verify the output.</td>
<td>Git / GitHub</td>
<td>Jira MINVOKER</td>
</tr>
<tr>
<td>jarsigner</td>
<td>B</td>
<td>3.0.0</td>
<td>2018-11-06</td>
<td>Signs or verifiesproject artifacts.</td>
<td>Git / GitHub</td>
<td>Jira MJARSIGNER</td>
</tr>
<tr>
<td>jdeprscan</td>
<td>B</td>
<td>3.0.0-alpha-1</td>
<td>2017-11-15</td>
<td>Run JDK's JDeprScan tool on the project.</td>
<td>Git / GitHub</td>
<td>Jira MJDEPRSCAN</td>
</tr>
<tr>
<td>patch</td>
<td>B</td>
<td>1.2</td>
<td>2015-03-09</td>
<td>Use the gnu patch tool to apply patch files to source code.</td>
<td>Git / GitHub</td>
<td>Jira MPATCH</td>
</tr>
<tr>
<td>pdf</td>
<td>B</td>
<td>1.5.1</td>
<td>2021-11-28</td>
<td>Generate a PDF version of your project's documentation.</td>
<td>Git / GitHub</td>
<td>Jira MPDF</td>
</tr>
<tr>
<td>Plugin</td>
<td>Version</td>
<td>Date</td>
<td>Description</td>
<td>Git / GitHub</td>
<td>Jira</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>plugin</td>
<td>B+R</td>
<td>3.6.2</td>
<td>2021-11-26 Create a Maven plugin descriptor for any mojos found in the source tree, to include in the JAR.</td>
<td>Git / GitHub</td>
<td>Jira MPLUGIN</td>
<td></td>
</tr>
<tr>
<td>release</td>
<td>B</td>
<td>3.0.0-M4</td>
<td>2021-04-16 Release the current project - updating the POM and tagging in the SCM.</td>
<td>Git / GitHub</td>
<td>Jira MRELEASE</td>
<td></td>
</tr>
<tr>
<td>remote-resources</td>
<td>B</td>
<td>1.7.0</td>
<td>2020-01-21 Copy remote resources to the output directory for inclusion in the artifact.</td>
<td>Git / GitHub</td>
<td>Jira MRRESOURCES</td>
<td></td>
</tr>
<tr>
<td>scm</td>
<td>B</td>
<td>1.12.0</td>
<td>2021-09-13 Execute SCM commands for the current project.</td>
<td>Git / GitHub</td>
<td>Jira SCM</td>
<td></td>
</tr>
<tr>
<td>scm-publish</td>
<td>B</td>
<td>3.1.0</td>
<td>2020-12-26 Publish your Maven website to a scm location.</td>
<td>Git / GitHub</td>
<td>Jira MSCMPUB</td>
<td></td>
</tr>
<tr>
<td>scripting</td>
<td>B</td>
<td>3.0.0</td>
<td>2021-03-01 The Maven Scripting Plugin wraps the Scripting API according to JSR223.</td>
<td>Git / GitHub</td>
<td>Jira MSCRIPTING</td>
<td></td>
</tr>
<tr>
<td>stage</td>
<td>B</td>
<td>1.0</td>
<td>2015-03-03 Assists with release staging and promotion.</td>
<td>Git / GitHub</td>
<td>Jira MSTAGE</td>
<td></td>
</tr>
<tr>
<td>toolchains</td>
<td>B</td>
<td>3.0.0</td>
<td>2019-06-16 Allows to share configuration across plugins.</td>
<td>Git / GitHub</td>
<td>Jira MTOOLCHAINS</td>
<td></td>
</tr>
<tr>
<td>wrapper</td>
<td>B</td>
<td>3.0.2</td>
<td>2021-04-08 Download and unpack the maven wrapper distribution (works only with Maven 4)</td>
<td>Git / GitHub</td>
<td>Jira MWRAPPER</td>
<td></td>
</tr>
</tbody>
</table>

* Build or Reporting plugin
There are also some sandbox plugins into our source repository. Previous archived versions of plugins reference documentations are located here.

### 10.1.2 Retired

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Type*</th>
<th>Version</th>
<th>Retired Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant</td>
<td>B</td>
<td>2.4</td>
<td>2019-06-02</td>
<td>Generate an Ant build file for the project.</td>
</tr>
<tr>
<td>eclipse</td>
<td>B</td>
<td>2.10</td>
<td>2015-10-07</td>
<td>Generate an Eclipse project files for the current project.</td>
</tr>
<tr>
<td>idea</td>
<td>B</td>
<td>2.2.1</td>
<td>2013-07-26</td>
<td>Create/update an IDEA workspace for the current project (individual modules are created as IDEA modules)</td>
</tr>
<tr>
<td>one</td>
<td>B</td>
<td>1.3</td>
<td>2013-07-30</td>
<td>A plugin for interacting with legacy Maven 1.x repositories and builds.</td>
</tr>
<tr>
<td>reactor</td>
<td>B</td>
<td>1.1</td>
<td>2014-03-24</td>
<td>Build a subset of interdependent projects in a reactor (Maven 2 only).</td>
</tr>
<tr>
<td>repository</td>
<td>B</td>
<td>2.4</td>
<td>2019-04-30</td>
<td>Plugin to help with repository-based tasks.</td>
</tr>
</tbody>
</table>

### 10.1.3 Outside The Maven Land

10.1.3.1 At MojoHaus (formerly known as codehaus.org)

There are also many plug-ins available at the MojoHaus project at GitHub.

Here are a few common ones:

<table>
<thead>
<tr>
<th>Plugin (see complete list with version)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animal-sniffer</td>
<td>Build signatures of APIs (JDK for example) and checks your classes against them.</td>
</tr>
<tr>
<td>build-helper</td>
<td>Attach extra artifacts and source folders to build.</td>
</tr>
<tr>
<td>castor</td>
<td>Generate sources from an XSD using Castor.</td>
</tr>
<tr>
<td>clirr</td>
<td>Compare binaries or sources for compatibility using Clirr</td>
</tr>
<tr>
<td>javacc</td>
<td>Generate sources from a JavaCC grammar.</td>
</tr>
</tbody>
</table>
### 10.1.3.2 Misc

A number of other projects provide their own Maven plugins. This includes:

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Maintainer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cargo</td>
<td>Cargo Project</td>
<td>Start/stop/configure J2EE containers and deploy to them.</td>
</tr>
<tr>
<td>clover</td>
<td>Atlassian Clover</td>
<td>Generate a Clover report.</td>
</tr>
<tr>
<td>jetty</td>
<td>Jetty Project</td>
<td>Jetty Run a Jetty container for rapid webapp development.</td>
</tr>
<tr>
<td>jalopy</td>
<td>Triemax</td>
<td>Use Jalopy to format your source code.</td>
</tr>
<tr>
<td>rat</td>
<td>Apache Creadur Project</td>
<td>Release Audit Tool (RAT) to verify files.</td>
</tr>
<tr>
<td>Genesis Plugins</td>
<td>Apache Geronimo Project</td>
<td>Verify legal files in artifacts.</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>Apache Tomcat Project</td>
<td>Run an Apache Tomcat container for rapid webapp development.</td>
</tr>
<tr>
<td>OWASP dependency-check</td>
<td>OWASP Dependency-check Project</td>
<td>Run OWASP Dependency-Check, a utility that identifies project dependencies and checks if there are any known, publicly disclosed, vulnerabilities.</td>
</tr>
<tr>
<td>CycloneDX</td>
<td>CycloneDX Project</td>
<td>Generate Software Bill of Materials (SBOM) in CycloneDX format.</td>
</tr>
<tr>
<td>pgpverify</td>
<td>Simplify4U</td>
<td>Verify PGP signature of all project dependencies.</td>
</tr>
<tr>
<td>buildplan</td>
<td>JC Gay</td>
<td>Inspect the lifecycle of your build.</td>
</tr>
</tbody>
</table>

### 10.1.4 Resources

1. [Guide to Configuring Plugins](#)
11 User Centre

11.1 Maven Users Centre

This documentation centre is for those that have decided to use Maven to build their project, and would like to get started quickly, or are already using Maven and would like to add new functionality or fix a problem in their build.

- Download Maven - Download the latest version of Maven
- The 5 minute test - Learn how to use Maven in 5 minutes
- Getting Started Tutorial - An in depth tutorial once you've learned the basics
- Getting Help - How to get help with Maven

11.1.1 Reference

- POM Reference
- Settings Reference
12 Maven in 5 Minutes

12.1 Maven in 5 Minutes

12.1.1 Prerequisites
You must understand how to install software on your computer. If you do not know how to do this, please ask someone at your office, school, etc. or pay someone to explain this to you. The Maven mailing lists are not the best place to ask for this advice.

12.1.2 Installation
Maven is a Java tool, so you must have Java installed in order to proceed.

First, download Maven and follow the installation instructions. After that, type the following in a terminal or in a command prompt:

```bash
mvn --version
```

It should print out your installed version of Maven, for example:

```
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: D:\apache-maven-3.6.3\apache-maven\bin\..
Java version: 1.8.0_232, vendor: AdoptOpenJDK, runtime: C:\Program Files\AdoptOpenJDK\jdk-8.0.232\jre
Default locale: en_US, platform encoding: Cp1250
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Depending upon your network setup, you may require extra configuration. Check out the Guide to Configuring Maven if necessary.

If you are using Windows, you should look at Windows Prerequisites to ensure that you are prepared to use Maven on Windows.

12.1.3 Creating a Project
You need somewhere for your project to reside. Create a directory somewhere and start a shell in that directory. On your command line, execute the following Maven goal:

```bash
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

If you have just installed Maven, it may take a while on the first run. This is because Maven is downloading the most recent artifacts (plugin jars and other files) into your local repository. You may also need to execute the command a couple of times before it succeeds. This is because the remote server may time out before your downloads are complete. Don't worry, there are ways to fix that.

You will notice that the `generate` goal created a directory with the same name given as the artifactId. Change into that directory.

```bash
cd my-app
```

Under this directory you will notice the following standard project structure.

```
my-app
|-- pom.xml
 `-- src
     |-- main
        | `-- java
        | `-- com
        |     `-- mycompany
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
The `src/main/java` directory contains the project source code, the `src/test/java` directory contains the test source, and the `pom.xml` file is the project's Project Object Model, or POM.

12.1.3.1 The POM

The `pom.xml` file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project in just the way you want. The POM is huge and can be daunting in its complexity, but it is not necessary to understand all of the intricacies just yet to use it effectively. This project's POM is:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

12.1.3.2 What did I just do?

You executed the Maven goal `archetype:generate`, and passed in various parameters to that goal. The prefix `archetype` is the plugin that provides the goal. If you are familiar with Ant, you may conceive of this as similar to a task. This `archetype:generate` goal created a simple project based upon a `maven-archetype-quickstart` archetype. Suffice it to say for now that a plugin is a collection of goals with a general common purpose. For example the jboss-maven-plugin, whose purpose is "deal with various jboss items".

12.1.3.3 Build the Project

`mvn package`

The command line will print out various actions, and end with the following:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
Unlike the first command executed (archetype:generate), the second is simply a single word - package. Rather than a goal, this is a phase. A phase is a step in the build lifecycle, which is an ordered sequence of phases. When a phase is given, Maven executes every phase in the sequence up to and including the one defined. For example, if you execute the compile phase, the phases that actually get executed are:

1. validate
2. generate-sources
3. process-sources
4. generate-resources
5. process-resources
6. compile

You may test the newly compiled and packaged JAR with the following command:

```java
java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App
```

Which will print the quintessential:

Hello World!

### 12.1.4 Java 9 or later

By default your version of Maven might use an old version of the maven-compiler-plugin that is not compatible with Java 9 or later versions. To target Java 9 or later, you should at least use version 3.6.0 of the maven-compiler-plugin and set the maven.compiler.release property to the Java release you are targeting (e.g. 9, 10, 11, 12, etc.).

In the following example, we have configured our Maven project to use version 3.8.1 of maven-compiler-plugin and target Java 11:

```xml
<properties>
  <maven.compiler.release>11</maven.compiler.release>
</properties>
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

To learn more about javac's --release option, see JEP 247.
12.1.5 Running Maven Tools

12.1.5.1 Maven Phases

Although hardly a comprehensive list, these are the most common default lifecycle phases executed.

- **validate**: validate the project is correct and all necessary information is available
- **compile**: compile the source code of the project
- **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package**: take the compiled code and package it in its distributable format, such as a JAR.
- **integration-test**: process and deploy the package if necessary into an environment where integration tests can be run
- **verify**: run any checks to verify the package is valid and meets quality criteria
- **install**: install the package into the local repository, for use as a dependency in other projects locally
- **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

There are two other Maven lifecycles of note beyond the default list above. They are

- **clean**: cleans up artifacts created by prior builds
- **site**: generates site documentation for this project

Phases are actually mapped to underlying goals. The specific goals executed per phase is dependant upon the packaging type of the project. For example, `package` executes `jar:jarm` if the project type is a JAR, and `war:war` if the project type is - you guessed it - a WAR.

An interesting thing to note is that phases and goals may be executed in sequence.

```bash
mvn clean dependency:copy-dependencies package
```

This command will clean the project, copy dependencies, and package the project (executing all phases up to `package`, of course).

12.1.5.2 Generating the Site

```bash
mvn site
```

This phase generates a site based upon information on the project's pom. You can look at the documentation generated under `target/site`.

12.1.6 Conclusion

We hope this quick overview has piqued your interest in the versatility of Maven. Note that this is a very truncated quick-start guide. Now you are ready for more comprehensive details concerning the actions you have just performed. Check out the Maven Getting Started Guide.
13 Getting Started Guide

13.1 Maven Getting Started Guide

This guide is intended as a reference for those working with Maven for the first time, but is also intended to serve as a cookbook with self-contained references and solutions for common use cases. For first time users, it is recommended that you step through the material in a sequential fashion. For users more familiar with Maven, this guide endeavours to provide a quick solution for the need at hand. It is assumed at this point that you have downloaded Maven and installed Maven on your local machine. If you have not done so please refer to the Download and Installation instructions.

Ok, so you now have Maven installed and we're ready to go. Before we jump into our examples we'll very briefly go over what Maven is and how it can help you with your daily work and collaborative efforts with team members. Maven will, of course, work for small projects, but Maven shines in helping teams operate more effectively by allowing team members to focus on what the stakeholders of a project require. You can leave the build infrastructure to Maven!

13.2 Sections

- What is Maven?
- How can Maven benefit my development process?
- How do I setup Maven?
- How do I make my first Maven project?
- How do I compile my application sources?
- How do I compile my test sources and run my unit tests?
- How do I create a JAR and install it in my local repository?
- What is a SNAPSHOT version?
- How do I use plugins?
- How do I add resources to my JAR?
- How do I filter resource files?
- How do I use external dependencies?
- How do I deploy my jar in my remote repository?
- How do I create documentation?
- How do I build other types of projects?
- How do I build more than one project at once?

13.2.1 What is Maven?

At first glance Maven can appear to be many things, but in a nutshell Maven is an attempt to apply patterns to a project's build infrastructure in order to promote comprehension and productivity by providing a clear path in the use of best practices. Maven is essentially a project management and comprehension tool and as such provides a way to help with managing:

- Builds
- Documentation
- Reporting
- Dependencies
- SCMs
- Releases
- Distribution
13.2.2 How can Maven benefit my development process?

Maven can provide benefits for your build process by employing standard conventions and practices to accelerate your development cycle while at the same time helping you achieve a higher rate of success.

Now that we have covered a little bit of the history and purpose of Maven let’s get into some real examples to get you up and running with Maven!

13.2.3 How do I setup Maven?

The defaults for Maven are often sufficient, but if you need to change the cache location or are behind a HTTP proxy, you will need to create configuration. See the Guide to Configuring Maven for more information.

13.2.4 How do I make my first Maven project?

We are going to jump headlong into creating your first Maven project! To create our first Maven project we are going to use Maven's archetype mechanism. An archetype is defined as an original pattern or model from which all other things of the same kind are made. In Maven, an archetype is a template of a project which is combined with some user input to produce a working Maven project that has been tailored to the user's requirements. We are going to show you how the archetype mechanism works now, but if you would like to know more about archetypes please refer to our Introduction to Archetypes.

On to creating your first project! In order to create the simplest of Maven projects, execute the following from the command line:

```
mvn -B archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
```

Once you have executed this command, you will notice a few things have happened. First, you will notice that a directory named `my-app` has been created for the new project, and this directory contains a file named `pom.xml` that should look like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
    </dependency>
  </dependencies>
</project>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
pom.xml contains the Project Object Model (POM) for this project. The POM is the basic unit of work in Maven. This is important to remember because Maven is inherently project-centric in that everything revolves around the notion of a project. In short, the POM contains every important piece of information about your project and is essentially one-stop-shopping for finding anything related to your project. Understanding the POM is important and new users are encouraged to refer to the Introduction to the POM.

This is a very simple POM but still displays the key elements every POM contains, so let's walk through each of them to familiarize you with the POM essentials:

- **project** This is the top-level element in all Maven pom.xml files.
- **modelVersion** This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.
- **groupId** This element indicates the unique identifier of the organization or group that created the project. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example org.apache.maven.plugins is the designated groupId for all Maven plugins.
- **artifactId** This element indicates the unique base name of the primary artifact being generated by this project. The primary artifact for a project is typically a JAR file. Secondary artifacts like source bundles also use the artifactId as part of their final name. A typical artifact produced by Maven would have the form <artifactId>-<version>.<extension> (for example, myapp-1.0.jar).
- **version** This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development. We will discuss the use of snapshots and how they work further on in this guide.
- **name** This element indicates the display name used for the project. This is often used in Maven's generated documentation.
- **url** This element indicates where the project's site can be found. This is often used in Maven's generated documentation.
- **properties** This element contains value placeholders accessible anywhere within a POM.
- **dependencies** This element's children list dependencies. The cornerstone of the POM.
- **build** This element handles things like declaring your project's directory structure and managing plugins.

For a complete reference of what elements are available for use in the POM please refer to our POM Reference. Now let's get back to the project at hand.

After the archetype generation of your first project you will also notice that the following directory structure has been created:

```
my-app
|-- pom.xml
`-- src
```
As you can see, the project created from the archetype has a POM, a source tree for your application's sources and a source tree for your test sources. This is the standard layout for Maven projects (the application sources reside in `${basedir}/src/main/java` and test sources reside in `${basedir}/src/test/java`, where `${basedir}` represents the directory containing `pom.xml`).

If you were to create a Maven project by hand this is the directory structure that we recommend using. This is a Maven convention and to learn more about it you can read our Introduction to the Standard Directory Layout.

Now that we have a POM, some application sources, and some test sources you are probably asking...

**13.2.5 How do I compile my application sources?**

Change to the directory where `pom.xml` is created by archetype:generate and execute the following command to compile your application sources:

```
mvn compile
```

Upon executing this command you should see output like the following:

```
[INFO] Scanning for projects...
[INFO] [INFO] ----------------------< com.mycompany.app:my-app >----------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO] [INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ my-app ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory <dir>/my-app/src/main/resources
[INFO] [INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ my-app ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to <dir>/my-app/target/classes
[INFO] [INFO] --- maven-resources-plugin:3.0.2:testResources (default-testResources) @ my-app ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory <dir>/my-app/src/test/resources
[INFO] [INFO] --- maven-jar-plugin:3.2:jar (default-jar) @ my-app ---
[INFO] Building jar: my-app.jar
[INFO] [INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ my-app ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Building the test suite...
[INFO] [INFO] [INFO] BUILD SUCCESS
[INFO] Total time:  0.899 s
[INFO] Finished at: 2020-07-12T11:31:54+01:00
[INFO] [INFO] ------------------------------------------------------------------------
```

The first time you execute this (or any other) command, Maven will need to download all the plugins and related dependencies it needs to fulfill the command. From a clean installation of Maven, this can take quite a while (in the output above, it took almost 4 minutes). If you execute the command again,
Maven will now have what it needs, so it won't need to download anything new and will be able to execute the command much more quickly.

As you can see from the output, the compiled classes were placed in $\{basedir\}/target/classes, which is another standard convention employed by Maven. So, if you're a keen observer, you'll notice that by using the standard conventions, the POM above is very small and you haven't had to tell Maven explicitly where any of your sources are or where the output should go. By following the standard Maven conventions, you can get a lot done with very little effort! Just as a casual comparison, let's take a look at what you might have had to do in Ant to accomplish the same thing.

Now, this is simply to compile a single tree of application sources and the Ant script shown is pretty much the same size as the POM shown above. But we'll see how much more we can do with just that simple POM!

### 13.2.6 How do I compile my test sources and run my unit tests?

Now you're successfully compiling your application's sources and now you've got some unit tests that you want to compile and execute (because every programmer always writes and executes their unit tests *nudge nudge wink wink*).

Execute the following command:

```
mvn test
```

Upon executing this command you should see output like the following:

```
[INFO] Scanning for projects...
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ my-app ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory <dir>/my-app/src/main/resources
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ my-app ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-resources-plugin:3.0.2:testResources (default-testResources) @ my-app ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory <dir>/my-app/src/test/resources
[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ my-app ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to <dir>/my-app/target/test-classes
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ my-app ---
[INFO] T E S T S
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.mycompany.app.AppTest
[INFO] Results:
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
Some things to notice about the output:

- Maven downloads more dependencies this time. These are the dependencies and plugins necessary for executing the tests (it already has the dependencies it needs for compiling and won't download them again).
- Before compiling and executing the tests Maven compiles the main code (all these classes are up to date because we haven't changed anything since we compiled last).

If you simply want to compile your test sources (but not execute the tests), you can execute the following:

```bash
mvn test-compile
```

Now that you can compile your application sources, compile your tests, and execute the tests, you'll want to move on to the next logical step so you'll be asking ...

### 13.2.7 How do I create a JAR and install it in my local repository?

Making a JAR file is straightforward enough and can be accomplished by executing the following command:

```bash
mvn package
```

You can now take a look in the `${basedir}/target` directory and you will see the generated JAR file.

Now you'll want to install the artifact you've generated (the JAR file) in your local repository (`${user.home}/.m2/repository` is the default location). For more information on repositories you can refer to our [Introduction to Repositories](#) but let's move on to installing our artifact! To do so execute the following command:

```bash
mvn install
```

Upon executing this command you should see the following output:

```bash
[INFO] Scanning for projects...
[INFO] ...
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ...
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ my-app ---
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ my-app ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-resources-plugin:3.0.2:testResources (default-testResources) @ my-app ---
```
...[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ my-app ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ my-app ---
[INFO] T E S T S
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.mycompany.app.AppTest
[INFO] Results:
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ my-app ---
[INFO] Building jar: <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ my-app ---
[INFO] Installing <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar to <local-repository>/com/mycompany/app/my-app/1.0-SNAPSHOT/my-app-1.0-SNAPSHOT.jar
[INFO] Installing <dir>/my-app/pom.xml to <local-repository>/com/mycompany/app/my-app/1.0-SNAPSHOT/my-app-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.678 s
[INFO] Finished at: 2020-07-12T12:04:45+01:00
[INFO] ------------------------------------------------------------------------

Note that the surefire plugin (which executes the test) looks for tests contained in files with a particular naming convention. By default the tests included are:

- **/*Test.java
- /**/Test*.java
- /**/*TestCase.java

And the default excludes are:

- /**/Abstract*Test.java
- /**/Abstract*TestCase.java

You have walked through the process for setting up, building, testing, packaging, and installing a typical Maven project. This is likely the vast majority of what projects will be doing with Maven and if you’ve noticed, everything you’ve been able to do up to this point has been driven by an 18-line file, namely the project's model or POM. If you look at a typical Ant build file that provides the same functionality that we’ve achieved thus far you’ll notice it’s already twice the size of the POM and we're just getting started! There is far more functionality available to you from Maven without requiring any additions to our POM as it currently stands. To get any more functionality out of our example Ant build file you must keep making error-prone additions.

So what else can you get for free? There are a great number of Maven plugins that work out of the box with even a simple POM like we have above. We’ll mention one here specifically as it is one of the highly prized features of Maven: without any work on your part this POM has enough information to generate a web site for your project! You will most likely want to customize your Maven site but if you're pressed for time all you need to do to provide basic information about your project is execute the following command:
mvn site

There are plenty of other standalone goals that can be executed as well, for example:

mvn clean

This will remove the target directory with all the build data before starting so that it is fresh.

13.2.8 What is a SNAPSHOT version?

Notice the value of the version tag in the pom.xml file shown below has the suffix: -SNAPSHOT.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    ...>
    <groupId>...</groupId>
    <artifactId>my-app</artifactId>
    ...
    <version>1.0-SNAPSHOT</version>
    <name>Maven Quick Start Archetype</name>
    ...
</project>
```

The SNAPSHOT value refers to the 'latest' code along a development branch, and provides no guarantee the code is stable or unchanging. Conversely, the code in a 'release' version (any version value without the suffix SNAPSHOT) is unchanging.

In other words, a SNAPSHOT version is the 'development' version before the final 'release' version. The SNAPSHOT is "older" than its release.

During the release process, a version of x.y-SNAPSHOT changes to x.y. The release process also increments the development version to x.(y+1)-SNAPSHOT. For example, version 1.0-SNAPSHOT is released as version 1.0, and the new development version is version 1.1-SNAPSHOT.

13.2.9 How do I use plugins?

Whenever you want to customise the build for a Maven project, this is done by adding or reconfiguring plugins.

For this example, we will configure the Java compiler to allow JDK 5.0 sources. This is as simple as adding this to your POM:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.3</version>
            <configuration>
                <source>1.5</source>
                <target>1.5</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```
You'll notice that all plugins in Maven look much like a dependency - and in some ways they are. This plugin will be automatically downloaded and used - including a specific version if you request it (the default is to use the latest available).

The `configuration` element applies the given parameters to every goal from the compiler plugin. In the above case, the compiler plugin is already used as part of the build process and this just changes the configuration. It is also possible to add new goals to the process, and configure specific goals. For information on this, see the Introduction to the Build Lifecycle.

To find out what configuration is available for a plugin, you can see the Plugins List and navigate to the plugin and goal you are using. For general information about how to configure the available parameters of a plugin, have a look at the Guide to Configuring Plugins.

13.2.10 How do I add resources to my JAR?

Another common use case that can be satisfied which requires no changes to the POM that we have above is packaging resources in the JAR file. For this common task, Maven again relies on the Standard Directory Layout, which means by using standard Maven conventions you can package resources within JARs simply by placing those resources in a standard directory structure.

You see below in our example we have added the directory `${basedir}/src/main/resources` into which we place any resources we wish to package in our JAR. The simple rule employed by Maven is this: any directories or files placed within the `${basedir}/src/main/resources` directory are packaged in your JAR with the exact same structure starting at the base of the JAR.

```
my-app
|-- pom.xml
  `-- src
    |-- main
    |   |-- java
    |   |   |-- com
    |   |   |   `-- mycompany
    |   |   `-- app
    |   `-- resources
    |       `-- META-INF
    |           `-- application.properties
    `-- test
      `-- java
          `-- com
              `-- mycompany
                  `-- app
                      `-- AppTest.java
```

So you can see in our example that we have a META-INF directory with an application.properties file within that directory. If you unpacked the JAR that Maven created for you and took a look at it you would see the following:

```
|-- META-INF
    |-- MANIFEST.MF
    `-- application.properties
      `-- maven
          `-- com.mycompany.app
              `-- my-app
                  |-- pom.properties
                  |-- pom.xml
                  `-- com
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
As you can see, the contents of `${basedir}/src/main/resources` can be found starting at the base of the JAR and our `application.properties` file is there in the `META-INF` directory. You will also notice some other files there like `META-INF/MANIFEST.MF` as well as a `pom.xml` and `pom.properties` file. These come standard with generation of a JAR in Maven. You can create your own manifest if you choose, but Maven will generate one by default if you don't. (You can also modify the entries in the default manifest. We will touch on this later.) The `pom.xml` and `pom.properties` files are packaged up in the JAR so that each artifact produced by Maven is self-describing and also allows you to utilize the metadata in your own application if the need arises. One simple use might be to retrieve the version of your application. Operating on the POM file would require you to use some Maven utilities but the properties can be utilized using the standard Java API and look like the following:

```java
#Generated by Maven
#Tue Oct 04 15:43:21 GMT-05:00 2005
version=1.0-SNAPSHOT
groupId=com.mycompany.app
artifactId=my-app
```

To add resources to the classpath for your unit tests, you follow the same pattern as you do for adding resources to the JAR except the directory you place resources in is `${basedir}/src/test/resources`. At this point you would have a project directory structure that would look like the following:

```
my-app
  |-- pom.xml
  `-- src
    `-- main
      `-- java
        `-- com
          `-- mycompany
            `-- app
              `-- App.java
        `-- resources
          `-- META-INF
            `-- application.properties
    `-- test
      `-- java
        `-- com
          `-- mycompany
            `-- app
              `-- AppTest.java
        `-- resources
          `-- test.properties
```

In a unit test you could use a simple snippet of code like the following to access the resource required for testing:

```java
 InputStream is = getClass().getResourceAsStream( "/test.properties" );
 // Do something with the resource
 ...```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
13.2.11 How do I filter resource files?

Sometimes a resource file will need to contain a value that can only be supplied at build time. To accomplish this in Maven, put a reference to the property that will contain the value into your resource file using the syntax `${<property name>}`. The property can be one of the values defined in your pom.xml, a value defined in the user's settings.xml, a property defined in an external properties file, or a system property.

To have Maven filter resources when copying, simply set filtering to true for the resource directory in your pom.xml:

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org/url</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</project>
```

You'll notice that we had to add the build, resources, and resource elements which weren't there before. In addition, we had to explicitly state that the resources are located in the src/main/resources directory. All of this information was provided as default values previously, but because the default value for filtering is false, we had to add this to our pom.xml in order to override that default value and set filtering to true.

To reference a property defined in your pom.xml, the property name uses the names of the XML elements that define the value, with "pom" being allowed as an alias for the project (root) element. So `${project.name}` refers to the name of the project, `${project.version}` refers to the version of the project, `${project.build.finalName}` refers to the final name of the file created when the built project is packaged, etc. Note that some elements of the POM have default values, so don't need to be explicitly defined in your pom.xml for the values to be available here. Similarly, values in the user's settings.xml can be referenced using property names beginning with "settings" (for example, `${settings.localRepository}` refers to the path of the user's local repository).

To continue our example, let's add a couple of properties to the application.properties file (which we put in the src/main/resources directory) whose values will be supplied when the resource is filtered:
With that in place, you can execute the following command (process-resources is the build lifecycle phase where the resources are copied and filtered):

```
mvn process-resources
```

and the `application.properties` file under `target/classes` (and will eventually go into the jar) looks like this:

```
# application.properties
application.name=Maven Quick Start Archetype
application.version=1.0-SNAPSHOT
```

To reference a property defined in an external file, all you need to do is add a reference to this external file in your `pom.xml`. First, let's create our external properties file and call it `src/main/filters/filter.properties`:

```
# filter.properties
my.filter.value=hello!
```

Next, we'll add a reference to this new file in the `pom.xml`:

```xml
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <filters>
    <filter>src/main/filters/filter.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```
Then, if we add a reference to this property in the `application.properties` file:

```
# application.properties
application.name=${project.name}
application.version=${project.version}
message=${my.filter.value}
```

the next execution of the `mvn process-resources` command will put our new property value into `application.properties`. As an alternative to defining the `my.filter.value` property in an external file, you could also have defined it in the `properties` section of your `pom.xml` and you'd get the same effect (notice I don't need the references to `src/main/filters/filter.properties` either):

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
    <build>
      <dependencies>
        <dependency>
          <groupId>com.mycompany.app</groupId>
          <artifactId>my-app</artifactId>
          <version>1.0-SNAPSHOT</version>
          <scope>test</scope>
        </dependency>
      </dependencies>
      <build>
        <resources>
          <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
          </resource>
        </resources>
        <resources>
          <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
          </resource>
        </resources>
      </build>
      <build>
        <resources>
          <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
          </resource>
        </resources>
      </build>
    </build>
    <properties>
      <my.filter.value>hello</my.filter.value>
    </properties>
  </build>
</project>
```

Filtering resources can also get values from system properties; either the system properties built into Java (like `java.version` or `user.home`) or properties defined on the command line using the standard Java `-D` parameter. To continue the example, let's change our `application.properties` file to look like this:

```
# application.properties
java.version=${java.version}
command.line.prop=${command.line.prop}
```
Now, when you execute the following command (note the definition of the command.line.prop property on the command line), the application.properties file will contain the values from the system properties.

```
mvn process-resources "-Dcommand.line.prop=hello again"
```

### 13.2.12 How do I use external dependencies?

You've probably already noticed a dependencies element in the POM we've been using as an example. You have, in fact, been using an external dependency all this time, but here we'll talk about how this works in a bit more detail. For a more thorough introduction, please refer to our Introduction to Dependency Mechanism.

The dependencies section of the pom.xml lists all of the external dependencies that our project needs in order to build (whether it needs that dependency at compile time, test time, run time, or whatever). Right now, our project is depending on JUnit only (I took out all of the resource filtering stuff for clarity):

```xml
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

For each external dependency, you'll need to define at least 4 things: groupId, artifactId, version, and scope. The groupId, artifactId, and version are the same as those given in the pom.xml for the project that built that dependency. The scope element indicates how your project uses that dependency, and can be values like `compile`, `test`, and `runtime`. For more information on everything you can specify for a dependency, see the Project Descriptor Reference.

For more information about the dependency mechanism as a whole, see Introduction to Dependency Mechanism.

With this information about a dependency, Maven will be able to reference the dependency when it builds the project. Where does Maven reference the dependency from? Maven looks in your local repository (`${user.home}/.m2/repository` is the default location) to find all dependencies. In a previous section, we installed the artifact from our project (my-app-1.0-SNAPSHOT.jar) into the local repository. Once it's installed there, another project can reference that jar as a dependency simply by adding the dependency information to its pom.xml:

```xml
<dependencies>
  <dependency>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
</dependencies>
```
<groupId>com.mycompany.app</groupId>
<artifactId>my-other-app</artifactId>
...
<dependencies>
...
<dependency>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0-SNAPSHOT</version>
<scope>compile</scope>
</dependency>
</dependencies>
</project>

What about dependencies built somewhere else? How do they get into my local repository? Whenever a project references a dependency that isn't available in the local repository, Maven will download the dependency from a remote repository into the local repository. You probably noticed Maven downloading a lot of things when you built your very first project (these downloads were dependencies for the various plugins used to build the project). By default, the remote repository Maven uses can be found (and browsed) at https://repo.maven.apache.org/maven2/. You can also set up your own remote repository (maybe a central repository for your company) to use instead of or in addition to the default remote repository. For more information on repositories you can refer to the Introduction to Repositories.

Let's add another dependency to our project. Let's say we've added some logging to the code and need to add log4j as a dependency. First, we need to know what the groupId, artifactId, and version are for log4j. The appropriate directory on Maven Central is called /maven2/log4j/log4j. In that directory is a file called maven-metadata.xml. Here's what the maven-metadata.xml for log4j looks like:

<metadata>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.1.3</version>
  <versioning>
    <versions>
      <version>1.1.3</version>
      <version>1.2.4</version>
      <version>1.2.5</version>
      <version>1.2.6</version>
      <version>1.2.7</version>
      <version>1.2.8</version>
      <version>1.2.11</version>
      <version>1.2.9</version>
      <version>1.2.12</version>
    </versions>
  </versioning>
</metadata>

From this file, we can see that the groupId we want is "log4j" and the artifactId is "log4j". We see lots of different version values to choose from; for now, we'll just use the latest version, 1.2.12 (some maven-metadata.xml files may also specify which version is the current release version: see repository metadata reference). Alongside the maven-metadata.xml file, we can see a directory corresponding to each version of the log4j library. Inside each of these, we'll find the actual jar file (e.g. log4j-1.2.12.jar) as well as a pom file (this is the pom.xml for the dependency, indicating any
Further dependencies it might have and other information) and another maven-metadata.xml file. There's also an md5 file corresponding to each of these, which contains an MD5 hash for these files. You can use this to authenticate the library or to figure out which version of a particular library you may be using already.

Now that we know the information we need, we can add the dependency to our pom.xml:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

Now, when we compile the project (`mvn compile`), we'll see Maven download the log4j dependency for us.

### 13.2.13 How do I deploy my jar in my remote repository?

For deploying jars to an external repository, you have to configure the repository url in the pom.xml and the authentication information for connecting to the repository in the settings.xml.

Here is an example using scp and username/password authentication:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
Note that if you are connecting to an openssh ssh server which has the parameter
"PasswordAuthentication" set to "no" in the sshd_config, you will have to type your password each
time for username/password authentication (although you can log in using another ssh client by typing
in the username and password). You might want to switch to public key authentication in this case.
Care should be taken if using passwords in `settings.xml`. For more information, see Password Encryption.

### 13.2.14 How do I create documentation?

To get you jump started with Maven’s documentation system you can use the archetype mechanism to generate a site for your existing project using the following command:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-site \n  -DgroupId=com.mycompany.app \n  -DartifactId=my-app-site
```

Now head on over to the Guide to creating a site to learn how to create the documentation for your project.

### 13.2.15 How do I build other types of projects?

Note that the lifecycle applies to any project type. For example, back in the base directory we can create a simple web application:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-webapp \n  -DgroupId=com.mycompany.app \n  -DartifactId=my-webapp
```

Note that these must all be on a single line. This will create a directory called `my-webapp` containing the following project descriptor:

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-webapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>my-webapp</finalName>
  </build>
</project>
```

Note the `<packaging>` element - this tells Maven to build as a WAR. Change into the webapp project’s directory and try:
mvn package

You'll see target/my-webapp.war is built, and that all the normal steps were executed.

13.2.16 How do I build more than one project at once?

The concept of dealing with multiple modules is built in to Maven. In this section, we will show how to build the WAR above, and include the previous JAR as well in one step.

Firstly, we need to add a parent pom.xml file in the directory above the other two, so it should look like this:

```
+- pom.xml
+- my-app
 | +- pom.xml
 | +- src
 |   +- main
 |   +- java
+- my-webapp
 | +- pom.xml
 | +- src
 |   +- main
 |   +- webapp
```

The POM file you'll create should contain the following:

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <modules>
    <module>my-app</module>
    <module>my-webapp</module>
  </modules>
</project>
```

We'll need a dependency on the JAR from the webapp, so add this to my-webapp/pom.xml:

```xml
...<dependencies>
  <dependency>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
...</dependencies>
```

Finally, add the following <parent> element to both of the other pom.xml files in the subdirectories:

```xml
  <parent>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
Now, try it... from the top level directory, run:

mvn verify

The WAR has now been created in my-webapp/target/my-webapp.war, and the JAR is included:

```
$ jar tvf my-webapp/target/my-webapp-1.0-SNAPSHOT.war
  0 Fri Jun 24 10:59:56 EST 2005 META-INF/
  222 Fri Jun 24 10:59:54 EST 2005 META-INF/MANIFEST.MF
  0 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/
  0 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-webapp/
  0 Fri Jun 24 10:59:56 EST 2005 WEB-INF/
  215 Fri Jun 24 10:59:56 EST 2005 WEB-INF/web.xml
  52 Fri Jun 24 10:59:56 EST 2005 index.jsp
  0 Fri Jun 24 10:59:56 EST 2005 WEB-INF/lib/
  2713 Fri Jun 24 10:59:56 EST 2005 WEB-INF/lib/my-app-1.0-SNAPSHOT.jar
```

How does this work? Firstly, the parent POM created (called app), has a packaging of pom and a list of modules defined. This tells Maven to run all operations over the set of projects instead of just the current one (to override this behaviour, you can use the --non-recursive command line option).

Next, we tell the WAR that it requires the my-app JAR. This does a few things: it makes it available on the classpath to any code in the WAR (none in this case), it makes sure the JAR is always built before the WAR, and it indicates to the WAR plugin to include the JAR in its library directory.

You may have noticed that junit-4.11.jar was a dependency, but didn't end up in the WAR. The reason for this is the <scope>test</scope> element - it is only required for testing, and so is not included in the web application as the compile time dependency my-app is.

The final step was to include a parent definition. This ensures that the POM can always be located even if the project is distributed separately from its parent by looking it up in the repository.
14.1 POM Reference

1. Introduction
   1. What is the POM?
   2. Quick Overview
2. The Basics
   1. Maven Coordinates
   2. POM Relationships
      1. Dependencies
         1. Dependency Version Requirement Specification
         2. Version Order Specification
         3. Version Order Testing
         4. Exclusions
      2. Inheritance
         1. The Super POM
         2. Dependency Management
      3. Aggregation (or Multi-Module)
         1. Inheritance v. Aggregation
   3. Properties
3. Build Settings
   1. Build
      1. The BaseBuild Element Set
         1. Resources
         2. Plugins
         1. Default Configuration Inheritance
         2. Advanced Configuration Inheritance
      2. The Build Element Set
      1. Directories
      2. Extensions
   2. Reporting
      1. Report Sets
4. More Project Information
   1. Licenses
   2. Organization
3. Developers
4. Contributors
5. Environment Settings
   1. Issue Management
   2. Continuous Integration Management
   3. Mailing Lists
   4. SCM
   5. Prerequisites
   6. Repositories
   7. Plugin Repositories
   8. Distribution Management
      1. Repository
      2. Site Distribution
      3. Relocation
   9. Profiles
      1. Activation
      2. The BaseBuild Element Set (revisited)
6. Final

14.2 Introduction

- The POM 4.0.0 XSD and descriptor reference documentation

14.2.1 What is the POM?

POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named pom.xml. When in the presence of Maven folks, speaking of a project is speaking in the philosophical sense, beyond a mere collection of files containing code. A project contains configuration files, as well as the developers involved and the roles they play, the defect tracking system, the organization and licenses, the URL of where the project lives, the project's dependencies, and all of the other little pieces that come into play to give code life. It is a one-stop-shop for all things concerning the project. In fact, in the Maven world, a project does not need to contain any code at all, merely a pom.xml.

14.2.2 Quick Overview

This is a listing of the elements directly under the POM's project element. Notice that modelVersion contains 4.0.0. That is currently the only supported POM version, and is always required.

```xml
  <modelVersion>4.0.0</modelVersion>
  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
</project>
```
14.3 The Basics

The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process. It is the declarative manifestation of the "who", "what", and "where", while the build lifecycle is the "when" and "how". That is not to say that the POM cannot affect the flow of the lifecycle - it can. For example, by configuring the maven-antrun-plugin, one can embed Apache Ant tasks inside of the POM. It is ultimately a declaration, however. Whereas a build.xml tells Ant precisely what to do when it is run (procedural), a POM states its configuration (declarative). If some external force causes the lifecycle to skip the Ant plugin execution, it does not stop the plugins that are executed from doing their magic. This is unlike a build.xml file, where tasks are almost always dependant on the lines executed before it.

14.3.1 Maven Coordinates

The POM defined above is the bare minimum that Maven allows. groupId:artifactId:version are all required fields (although, groupId and version do not need to be explicitly defined if they are inherited from a parent - more on inheritance later). The three fields act much like an address and...
timestamp in one. This marks a specific place in a repository, acting like a coordinate system for Maven projects:

- **groupId**: This is generally unique amongst an organization or a project. For example, all core Maven artifacts do (well, should) live under the groupId org.apache.maven. Group ID's do not necessarily use the dot notation, for example, the junit project. Note that the dot-notated groupId does not have to correspond to the package structure that the project contains. It is, however, a good practice to follow. When stored within a repository, the group acts much like the Java packaging structure does in an operating system. The dots are replaced by OS specific directory separators (such as '/' in Unix) which becomes a relative directory structure from the base repository. In the example given, the org.codehaus.mojo group lives within the directory $M2_REPO/org/codehaus/mojo.

- **artifactId**: The artifactId is generally the name that the project is known by. Although the groupId is important, people within the group will rarely mention the groupId in discussion (they are often all be the same ID, such as the MojoHaus project groupId: org.codehaus.mojo). It, along with the groupId, creates a key that separates this project from every other project in the world (at least, it should :)). Along with the groupId, the artifactId fully defines the artifact's living quarters within the repository. In the case of the above project, my-project lives in $M2_REPO/org/codehaus/mojo/my-project.

- **version**: This is the last piece of the naming puzzle. groupId:artifactId denotes a single project but they cannot delineate which incarnation of that project we are talking about. Do we want the junit:junit of 2018 (version 4.12), or of 2007 (version 3.8.2)? In short: code changes, those changes should be versioned, and this element keeps those versions in line. It is also used within an artifact’s repository to separate versions from each other. my-project version 1.0 files live in the directory structure $M2_REPO/org/codehaus/mojo/my-project/1.0.

The three elements given above point to a specific version of a project, letting Maven know who we are dealing with, and when in its software lifecycle we want them.

### 14.3.2 packaging

Now that we have our address structure of groupId:artifactId:version, there is one more standard label to give us a really complete what: that is the project's packaging. In our case, the example POM for org.codehaus.mojo:my-project:1.0 defined above will be packaged as a jar. We could make it into a war by declaring a different packaging:

```xml
  ...
  <packaging>war</packaging>
  ...
</project>
```

When no packaging is declared, Maven assumes the packaging is the default: jar. The valid types are Plexus role-hints (read more on Plexus for a explanation of roles and role-hints) of the component role org.apache.maven.lifecycle.mapping.LifecycleMapping. The current core packaging values are: pom, jar, maven-plugin, ejb, war, ear, rar. These define the default list of goals which execute on each corresponding build lifecycle stage for a particular package structure: see Plugin Bindings for default Lifecycle Reference for details.

### 14.3.3 POM Relationships

One powerful aspect of Maven is its handling of project relationships: this includes dependencies (and transitive dependencies), inheritance, and aggregation (multi-module projects).
Dependency management has a long tradition of being a complicated mess for anything but the most trivial of projects. "Jarmageddon" quickly ensues as the dependency tree becomes large and complicated. "Jar Hell" follows, where versions of dependencies on one system are not equivalent to the versions developed with, either by the wrong version given, or conflicting versions between similarly named jars.

Maven solves both problems through a common local repository from which to link projects correctly, versions and all.

14.3.3.1 Dependencies

The cornerstone of the POM is its dependency list. Most projects depend on others to build and run correctly. If all Maven does for you is manage this list, you have gained a lot. Maven downloads and links the dependencies on compilation, as well as on other goals that require them. As an added bonus, Maven brings in the dependencies of those dependencies (transitive dependencies), allowing your list to focus solely on the dependencies your project requires.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <type>jar</type>
            <scope>test</scope>
            <optional>true</optional>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

- **groupId, artifactId, version:**
  You will see these elements often. This trinity is used to compute the Maven coordinate of a specific project in time, demarcating it as a dependency of this project. The purpose of this computation is to select a version that matches all the dependency declarations (due to transitive dependencies, there can be multiple dependency declarations for the same artifact). The values should be:
  - **groupId, artifactId:** directly the corresponding coordinates of the dependency,
  - **version:** a dependency version requirement specification, that is used to compute the dependency's effective version.

Since the dependency is described by Maven coordinates, you may be thinking: "This means that my project can only depend upon Maven artifacts!" The answer is, "Of course, but that's a good thing." This forces you to depend solely on dependencies that Maven can manage.

There are times, unfortunately, when a project cannot be downloaded from the central Maven repository. For example, a project may depend upon a jar that has a closed-source license which prevents it from being in a central repository. There are three methods for dealing with this scenario.

1. Install the dependency locally using the install plugin. The method is the simplest recommended method. For example:

   ```bash
   mvn install:install-file -Dfile=non-maven-proj.jar -DgroupId=some.group -DartifactId=non-
   ```
Notice that an address is still required, only this time you use the command line and the install plugin will create a POM for you with the given address.

2. Create your own repository and deploy it there. This is a favorite method for companies with an intranet and need to be able to keep everyone in synch. There is a Maven goal called deploy:deploy-file which is similar to the install:install-file goal (read the plugin's goal page for more information).

3. Set the dependency scope to system and define a systemPath. This is not recommended, however, but leads us to explaining the following elements:

- **classifier:**
The classifier distinguishes artifacts that were built from the same POM but differ in content. It is some optional and arbitrary string that - if present - is appended to the artifact name just after the version number.

As a motivation for this element, consider for example a project that offers an artifact targeting Java 11 but at the same time also an artifact that still supports Java 1.8. The first artifact could be equipped with the classifier jdk11 and the second one with jdk8 such that clients can choose which one to use.

Another common use case for classifiers is to attach secondary artifacts to the project's main artifact. If you browse the Maven central repository, you will notice that the classifiers sources and javadoc are used to deploy the project source code and API docs along with the packaged class files.

- **type:**
Corresponds to the chosen dependency type. This defaults to jar. While it usually represents the extension on the filename of the dependency, that is not always the case: a type can be mapped to a different extension and a classifier. The type often corresponds to the packaging used, though this is also not always the case. Some examples are jar, ejb-client and test-jar: see default artifact handlers for a list. New types can be defined by plugins that set extensions to true, so this is not a complete list.

- **scope:**
This element refers to the classpath of the task at hand (compiling and runtime, testing, etc.) as well as how to limit the transitivity of a dependency. There are five scopes available:
  - **compile** - this is the default scope, used if none is specified. Compile dependencies are available in all classpaths. Furthermore, those dependencies are propagated to dependent projects.
  - **provided** - this is much like compile, but indicates you expect the JDK or a container to provide it at runtime. It is only available on the compilation and test classpath, and is not transitive.
  - **runtime** - this scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
  - **test** - this scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. It is not transitive.
  - **system** - this scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

- **systemPath:**
is used only if the dependency scope is system. Otherwise, the build will fail if this element is set. The path must be absolute, so it is recommended to use a property to specify the machine-specific path (more on properties below), such as ${java.home}/lib. Since it is assumed that system scope dependencies are installed a priori, Maven does not check the repositories.
for the project, but instead checks to ensure that the file exists. If not, Maven fails the build and suggests that you download and install it manually.

- **optional:**
  Marks a dependency optional when this project itself is a dependency. For example, imagine a project A that depends upon project B to compile a portion of code that may not be used at runtime, then we may have no need for project B for all project. So if project X adds project A as its own dependency, then Maven does not need to install project B at all. Symbolically, if => represents a required dependency, and --> represents optional, although A=>B may be the case when building A X-->A-->B would be the case when building X.

  In the shortest terms, optional lets other projects know that, when you use this project, you do not require this dependency in order to work correctly.


Dependencies' version elements define version requirements, which are used to compute dependency versions. Soft requirements can be replaced by different versions of the same artifact found elsewhere in the dependency graph. Hard requirements mandate a particular version or versions and override soft requirements. If there are no versions of a dependency that satisfy all the hard requirements for that artifact, the build fails.

Version requirements have the following syntax:

- **1.0**: Soft requirement for 1.0. Use 1.0 if no other version appears earlier in the dependency tree.
- **[1.0]**: Hard requirement for 1.0. Use 1.0 and only 1.0.
- **(1.0)**: Hard requirement for any version <= 1.0.
- **[1.2, 1.3]**: Hard requirement for any version between 1.2 and 1.3 inclusive.
- **[1.0, 2.0)**: 1.0 <= x < 2.0; Hard requirement for any version between 1.0 inclusive and 2.0 exclusive.
- **[1.5, )**: Hard requirement for any version greater than or equal to 1.5.
- **(1.0), [1.2, )**: Hard requirement for any version less than or equal to 1.0 than or greater than or equal to 1.2, but not 1.1. Multiple requirements are separated by commas.
- **(1.1), (1.1, )**: Hard requirement for any version except 1.1; for example because 1.1 has a critical vulnerability.

  Maven picks the highest version of each project that satisfies all the hard requirements of the dependencies on that project. If no version satisfies all the hard requirements, the build fails.

14. **Version Order Specification:**

If version strings are syntactically correct **Semantic Versioning 1.0.0** version numbers, then in almost all cases version comparison follows the precedence rules outlined in that specification. These versions are the commonly encountered alphanumeric ASCII strings such as 2.15.2-alpha. More precisely, this is true if both version numbers to be compared match the "valid semver" production in the BNF grammar in the semantic versioning specification. Maven does not consider any semantics implied by that specification.

**Important:** This is only true for Semantic Versioning 1.0.0. The Maven version order algorithm is not compatible with Semantic Versioning 2.0.0. In particular, Maven does not special case the plus sign or consider build identifiers.

When version strings do not follow semantic versioning, a more complex set of rules is required. The Maven coordinate is split in tokens between dots (’.’), hyphens (’-’) and transitions between digits and characters. The separator is recorded and will have effect on the order. A transition between digits and characters is equivalent to a hyphen. Empty tokens are replaced with " 0". This gives a sequence of version numbers (numeric tokens) and version qualifiers (non-numeric tokens) with " ." or " -" prefixes.
Splitting and Replacing Examples:
- \(1-1.0.0\) -> \(1-1-0.0\)

Then, starting from the end of the version, the trailing "null" values (0, "", "final", "ga") are trimmed. This process is repeated at each remaining hyphen from end to start.

Trimming Examples:
- \(1.0.0\) -> \(1\)
- \(1.ga\) -> \(1\)
- \(1.final\) -> \(1\)
- \(1.\) -> \(1\)
- \(1-\) -> \(1\)
- \(1.0-0.0.0\) -> \(1-foo\)
- \(1.0.0-0.0.0\) -> \(1\)

The version order is the lexicographical order on this sequence of prefixed tokens, the shorter one padded with enough "null" values with matching prefix to have the same length as the longer one. Padded "null" values depend on the prefix of the other version: 0 for ".", "" for ".". The prefixed token order is:
  - if the prefix is the same, then compare the token:
    - Numeric tokens have the natural order.
    - Non-numeric tokens ("qualifiers") have the alphabetical order, except for the following tokens which come first in this order:
      - "alpha" < "beta" < "milestone" < "rc" = "cr" < "snapshot" < "" = "final" = "ga" < "sp"
      - the "alpha", "beta" and "milestone" qualifiers can respectively be shortened to "a", "b" and "m" when directly followed by a number.
    - else ".quality" < "-quality" < "-number" < ".number"

End Result Examples:
- "1" < "1.1" (number padding)
- "1-snapshot" < "1" < "1-sp" (qualifier padding)
- "1-foo2" < "1-foo10" (correctly automatically "switching" to numeric order)
- "1.foo" < "1-foo" < "1-1" < "1.1"
- "1.ga" = "1-ga" = "1-0" = "1.0" = "1" (removing of trailing "null" values)
- "1-sp" > "1-ga"
- "1-sp.1" > "1-ga.1"
- "1-sp-1" < "1-ga-1" = "1-1" (trailing "null" values at each hyphen)
- "1-al" = "1-alpha-1"

Note: Contrary to what was stated in some design documents, for version order, snapshots are not treated differently than releases or any other qualifier.

14. Version Order Testing:
The maven distribution includes a tool to check version order. It was used to produce the examples in the previous paragraphs. Feel free to run it yourself when in doubt. You can run it like this:
```
java -jar ${MAVEN_HOME}/lib/maven-artifact-3.3.9.jar [versions...]
```
Example:
```
$ java -jar ./lib/maven-artifact-3.3.9.jar 1 2 1.1
```
Display parameters as parsed by Maven (in canonical form) and comparison result:
1. 1 == 1
2. 2 == 2
3. 1.1 == 1.1

14. Exclusions

Exclusions tell Maven not to include the specified project that is a dependency of this dependency (in other words, its transitive dependency). For example, the maven-embedder requires maven-core, and we do not wish to use it or its dependencies, then we would add it as an exclusion.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...<dependencies>
  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-embedder</artifactId>
    <version>2.0</version>
    <exclusions>
      <exclusion>
        <groupId>org.apache.maven</groupId>
        <artifactId>maven-core</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  ...
</dependencies>
```

It is also sometimes useful to clip a dependency's transitive dependencies. A dependency may have incorrectly specified scopes, or dependencies that conflict with other dependencies in your project. Using wildcard excludes makes it easy to exclude all a dependency's transitive dependencies. In the case below you may be working with the maven-embedder and you want to manage the dependencies you use yourself, so you clip all the transitive dependencies:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...<dependencies>
  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-embedder</artifactId>
    <version>3.1.0</version>
    <exclusions>
      <exclusion>
        <groupId>*</groupId>
        <artifactId>*</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  ...
</dependencies>
```
• **exclusions**: Exclusions contain one or more exclusion elements, each containing a groupId and artifactId denoting a dependency to exclude. Unlike optional, which may or may not be installed and used, exclusions actively remove themselves from the dependency tree.

### 14.3.3.2 Inheritance

One powerful addition that Maven brings to build management is the concept of project inheritance. Although in build systems such as Ant inheritance can be simulated, Maven makes project inheritance explicit in the project object model.

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <packaging>pom</packaging>
</project>
```

The packaging type required to be pom for **parent** and **aggregation** (multi-module) projects. These types define the goals bound to a set of lifecycle stages. For example, if packaging is jar, then the package phase will execute the `jar:jar` goal. Now we may add values to the parent POM, which will be inherited by its children. Most elements from the parent POM are inherited by its children, including:

- groupId
- version
- description
- url
- inceptionYear
- organization
- licenses
- developers
- contributors
- mailingLists
- scm
- issueManagement
- ciManagement
- properties
- dependencyManagement
- dependencies
- repositories
- pluginRepositories
- build
  - plugin executions with matching ids
  - plugin configuration
Notable elements which are not inherited include:

- artifactId
- name
- prerequisites

Notice the `relativePath` element. It is not required, but may be used as a signifier to Maven to first search the path given for this project's parent, before searching the local and then remote repositories.

To see inheritance in action, just have a look at the ASF or Maven parent POM's.

14. The Super POM

Similar to the inheritance of objects in object oriented programming, POMs that extend a parent POM inherit certain values from that parent. Moreover, just as Java objects ultimately inherit from `java.lang.Object`, all Project Object Models inherit from a base Super POM. The snippet below is the Super POM for Maven 3.5.4.

```xml
  <modelVersion>4.0.0</modelVersion>
  <repositories>
    <repository>
      <id>central</id>
      <name>Central Repository</name>
      <url>https://repo.maven.apache.org/maven2</url>
    </repository>
  </repositories>
</project>
```
<releases>
  <updatePolicy>never</updatePolicy>
</releases>
</pluginRepositories>

<build>
  <directory>${project.basedir}/target</directory>
  <outputDirectory>${project.build.directory}/classes</outputDirectory>
  <finalName>${project.artifactId}-${project.version}</finalName>
  <testOutputDirectory>${project.build.directory}/test-classes</testOutputDirectory>
  <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>
  <scriptSourceDirectory>${project.basedir}/src/main/scripts</scriptSourceDirectory>
  <testSourceDirectory>${project.basedir}/src/test/java</testSourceDirectory>

  <resources>
    <resource>
      <directory>${project.basedir}/src/main/resources</directory>
    </resource>
  </resources>

  <testResources>
    <testResource>
      <directory>${project.basedir}/src/test/resources</directory>
    </testResource>
  </testResources>

  <pluginManagement>
    <!-- NOTE: These plugins will be removed from future versions of the super POM -->
    <!-- They are kept for the moment as they are very unlikely to conflict with lifecycle mappings (MNG-4453) -->
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-5</version>
      </plugin>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>2.8</version>
      </plugin>
      <plugin>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.5.3</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<reporting>
  <outputDirectory>${project.build.directory}/site</outputDirectory>
</reporting>

<profiles>
  <!-- NOTE: The release profile will be removed from future versions of the super POM -->
</profiles>
<activation>
  <property>
    <name>performRelease</name>
    <value>true</value>
  </property>
</activation>

<build>
  <plugins>
    <plugin>
      <inherited>true</inherited>
      <artifactId>maven-source-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-sources</id>
          <goals>
            <goal>jar-no-fork</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <inherited>true</inherited>
      <artifactId>maven-javadoc-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-javadoes</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <inherited>true</inherited>
      <artifactId>maven-deploy-plugin</artifactId>
      <configuration>
        <updateReleaseInfo>true</updateReleaseInfo>
      </configuration>
    </plugin>
  </plugins>
</build>
</profile>
</profiles>
</project>

You can take a look at how the Super POM affects your Project Object Model by creating a minimal pom.xml and executing on the command line: mvn help:effective-pom

14. Dependency Management

Besides inheriting certain top-level elements, parents have elements to configure values for child POMs and transitive dependencies. One of those elements is dependencyManagement.

- **dependencyManagement**: is used by a POM to help manage dependency information across all of its children. If the my-parent project uses dependencyManagement to define a dependency
on junit:junit:4.12, then POMs inheriting from this one can set their dependency giving the groupId= junit and artifactId= junit only and Maven will fill in the version set by the parent. The benefits of this method are obvious. Dependency details can be set in one central location, which propagates to all inheriting POMs. Note that the version and scope of artifacts which are incorporated from transitive dependencies are also controlled by version specifications in a dependency management section. This can lead to unexpected consequences. Consider a case in which your project uses two dependences, dep1 and dep2. dep2 in turn also uses dep1, and requires a particular minimum version to function. If you then use dependencyManagement to specify an older version, dep2 will be forced to use the older version, and fail. So, you must be careful to check the entire dependency tree to avoid this problem; mvn dependency:tree is helpful.

### 14.3.3.3 Aggregation (or Multi-Module)

A project with modules is known as a multi-module, or aggregator project. Modules are projects that this POM lists, and are executed as a group. A pom packaged project may aggregate the build of a set of projects by listing them as modules, which are relative paths to the directories or the POM files of those projects.

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>my-project</module>
    <module>another-project</module>
    <module>third-project/pom-example.xml</module>
  </modules>
</project>
```

You do not need to consider the inter-module dependencies yourself when listing the modules; i.e. the ordering of the modules given by the POM is not important. Maven will topologically sort the modules such that dependencies are always build before dependent modules.

To see aggregation in action, have a look at the Maven or Maven Core Plugins base POM's.

### 14.3.4 Properties

Properties are the last required piece to understand POM basics. Maven properties are value placeholders, like properties in Ant. Their values are accessible anywhere within a POM by using the notation ${X}, where X is the property. Or they can be used by plugins as default values, for example:

```xml
<project>
  ...
</project>
```
They come in five different styles:

1. **env.X**: Prefixing a variable with "env." will return the shell's environment variable. For example, \( \{\text{env.PATH}\} \) contains the PATH environment variable.
   
   Note: While environment variables themselves are case-insensitive on Windows, lookup of properties is case-sensitive. In other words, while the Windows shell returns the same value for \%PATH\% and \%Path\%, Maven distinguishes between \{env.PATH\} and \{env.Path\}. The names of environment variables are normalized to all upper-case for the sake of reliability.

2. **project.x**: A dot (.) notated path in the POM will contain the corresponding element's value. For example: \(<\text{project}><\text{version}>1.0</\text{version}></\text{project}>\) is accessible via \{project.version\}.

3. **settings.x**: A dot (.) notated path in the settings.xml will contain the corresponding element's value. For example: \(<\text{settings}><\text{offline}>false</\text{offline}></\text{settings}>\) is accessible via \{settings.offline\}.

4. **Java System Properties**: All properties accessible via java.lang.System.getProperties() are available as POM properties, such as \{java.home\}.

5. **x**: Set within a \(<\text{properties />}\) element in the POM. The value of \(<\text{properties}><\text{someVar}>value</\text{someVar}></\text{properties}>\) may be used as \{someVar\}.

### 14.4 Build Settings

Beyond the basics of the POM given above, there are two more elements that must be understood before claiming basic competency of the POM. They are the **build** element, that handles things like declaring your project's directory structure and managing plugins; and the **reporting** element, that largely mirrors the build element for reporting purposes.

#### 14.4.1 Build

According to the POM 4.0.0 XSD, the **build** element is conceptually divided into two parts: there is a **BaseBuild** type which contains the set of elements common to both build elements (the top-level build element under **project** and the build element under **profiles**, covered below); and there is the **Build** type, which contains the **BaseBuild** set as well as more elements for the top level definition. Let us begin with an analysis of the common elements between the two.

Note: These different **build** elements may be denoted "project build" and "profile build".
14.4.1.1 The BaseBuild Element Set

**BaseBuild** is exactly as it sounds: the base set of elements between the two **build** elements in the POM.

```xml
<build>
  <defaultGoal>install</defaultGoal>
  <directory>${basedir}/target</directory>
  <finalName>${artifactId}-${version}</finalName>
  <filters>
    <filter>filters/filter1.properties</filter>
  </filters>
  ...
</build>
```

- **defaultGoal**: the default goal or phase to execute if none is given. If a goal is given, it should be defined as it is in the command line (such as `jar:jar`). The same goes for if a phase is defined (such as `install`).
- **directory**: This is the directory where the build will dump its files or, in Maven parlance, the build's target. It aptly defaults to `${basedir}/target`.
- **finalName**: This is the name of the bundled project when it is finally built (sans the file extension, for example: `my-project-1.0.jar`). It defaults to `${artifactId}-${version}`. The term "finalName" is kind of a misnomer, however, as plugins that build the bundled project have every right to ignore/modify this name (but they usually do not). For example, if the `maven-jar-plugin` is configured to give a jar a classifier of `test`, then the actual jar defined above will be built as `my-project-1.0-test.jar`.
- **filter**: Defines `*.properties` files that contain a list of properties that apply to resources which accept their settings (covered below). In other words, the "name=value" pairs defined within the filter files replace `${name}` strings within resources on build. The example above defines the `filter1.properties` file under the `filters/` directory. Maven's default filter directory is `${basedir}/src/main/filters/`.

For a more comprehensive look at what filters are and what they can do, take a look at the quick start guide.

14. Resources

Another feature of **build** elements is specifying where resources exist within your project. Resources are not (usually) code. They are not compiled, but are items meant to be bundled within your project or used for various other reasons, such as code generation.

For example, a Plexus project requires a `configuration.xml` file (which specifies component configurations to the container) to live within the `META-INF/plexus` directory. Although we could just as easily place this file within `src/main/resources/META-INF/plexus`, we want instead to give Plexus its own directory of `src/main/plexus`. In order for the JAR plugin to bundle the resource correctly, you would specify resources similar to the following:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <build>
    ...
  </build>
</project>
```
• **resources**: is a list of resource elements that each describe what and where to include files associated with this project.

• **targetPath**: Specifies the directory structure to place the set of resources from a build. Target path defaults to the base directory. A commonly specified target path for resources that will be packaged in a JAR is META-INF.

• **filtering**: is true or false, denoting if filtering is to be enabled for this resource. Note, that filter 

* .properties files do not have to be defined for filtering to occur - resources can also use properties that are by default defined in the POM (such as ${project.version}), passed into the command line using the "-D" flag (for example, "-Dname=value") or are explicitly defined by the properties element. Filter files were covered above.

• **directory**: This element's value defines where the resources are to be found. The default directory for a build is 

${basedir}/src/main/resources.

• **includes**: A set of files patterns which specify the files to include as resources under that specified directory, using * as a wildcard.

• **excludes**: The same structure as includes, but specifies which files to ignore. In conflicts between include and exclude, exclude wins.

• **testResources**: The testResources element block contains testResource elements. Their definitions are similar to resource elements, but are naturally used during test phases. The one difference is that the default (Super POM defined) test resource directory for a project is 

${basedir}/src/test/resources. Test resources are not deployed.

14. Plugins

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
      </plugin>
    </plugins>
  </build>
</project>
```
Beyond the standard coordinate of `groupId:artifactId:version`, there are elements which configure the plugin or this builds interaction with it.

- **extensions**: `true` or `false`, whether or not to load extensions of this plugin. It is by default false. Extensions are covered later in this document.
- **inherited**: `true` or `false`, whether or not this plugin configuration should apply to POMs which inherit from this one. Default value is `true`.
- **configuration**: This is specific to the individual plugin. Without going too in depth into the mechanics of how plugins work, suffice it to say that whatever properties that the plugin Mojo may expect (these are getters and setters in the Java Mojo bean) can be specified here. In the above example, we are setting the classifier property to test in the `maven-jar-plugin`'s Mojo.

It may be good to note that all configuration elements, wherever they are within the POM, are intended to pass values to another underlying system, such as a plugin. In other words: values within a `configuration` element are never explicitly required by the POM schema, but a plugin goal has every right to require configuration values.

If your POM declares a parent, it inherits plugin configuration from either the `build/plugins` or `pluginManagement` sections of the parent.

- **default configuration inheritance**:

  To illustrate, consider the following fragment from a parent POM:

  ```xml
  <plugin>
  <groupId>my.group</groupId>
  <artifactId>my-plugin</artifactId>
  <configuration>
    <items>
      <item>parent-1</item>
      <item>parent-2</item>
    </items>
  </configuration>
  </plugin>
  ```

  And consider the following plugin configuration from a project that uses that parent as its parent:

  ```xml
  <plugin>
  <groupId>my.group</groupId>
  <artifactId>my-plugin</artifactId>
  <configuration>
  ```
The default behavior is to merge the content of the `configuration` element according to element name. If the child POM has a particular element, that value becomes the effective value. If the child POM does not have an element, but the parent does, the parent value becomes the effective value. Note that this is purely an operation on XML; no code or configuration of the plugin itself is involved. Only the elements, not their values, are involved.

Applying those rules to the example, Maven comes up with:

```xml
<plugin>
  <groupId>my.group</groupId>
  <artifactId>my-plugin</artifactId>
  <configuration>
    <items>
      <item>child-1</item>
    </items>
    <properties>
      <childKey>child</childKey>
    </properties>
  </configuration>
</plugin>
```

- **advanced configuration inheritance**: `combine.children` and `combine.self`
  
  You can control how child POMs inherit configuration from parent POMs by adding attributes to the children of the `configuration` element. The attributes are `combine.children` and `combine.self`. Use these attributes in a child POM to control how Maven combines plugin configuration from the parent with the explicit configuration in the child.

Here is the child configuration with illustrations of the two attributes:

```xml
<configuration>
  <items combine.children="append">
    <!-- combine.children="merge" is the default -->
    <item>child-1</item>
  </items>
  <properties combine.self="override">
    <!-- combine.self="merge" is the default -->
    <childKey>child</childKey>
    <parentKey>parent</parentKey>
  </properties>
</configuration>
```

Now, the effective result is the following:

```xml
<configuration>
  <items>
    <item>child-1</item>
  </items>
  <properties>
    <childKey>child</childKey>
    <parentKey>parent</parentKey>
  </properties>
</configuration>
```
combine.children="append" results in the concatenation of parent and child elements, in that order. combine.self="override", on the other hand, completely suppresses parent configuration. You cannot use both combine.self="override" and combine.children="append" on an element; if you try, override will prevail.

Note that these attributes only apply to the configuration element they are declared on, and are not propagated to nested elements. That is if the content of an item element from the child POM was a complex structure instead of text, its sub-elements would still be subject to the default merge strategy unless they were themselves marked with attributes.

The combine.* attributes are inherited from parent to child POMs. Take care when adding those attributes to a parent POM as this might affect child or grand-child POMs.

- **dependencies**: Dependencies are seen a lot within the POM, and are an element under all plugins element blocks. The dependencies have the same structure and function as under that base build. The major difference in this case is that instead of applying as dependencies of the project, they now apply as dependencies of the plugin that they are under. The power of this is to alter the dependency list of a plugin, perhaps by removing an unused runtime dependency via exclusions, or by altering the version of a required dependency. See above under Dependencies for more information.

- **executions**: It is important to keep in mind that a plugin may have multiple goals. Each goal may have a separate configuration, possibly even binding a plugin’s goal to a different phase altogether. executions configure the execution of a plugin’s goals.

For example, suppose you wanted to bind the antrun:run goal to the verify phase. We want the task to echo the build directory, as well as avoid passing on this configuration to its children (assuming it is a parent) by setting inherited to false. You would get an execution like this:

```xml
    ...
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-antrun-plugin</artifactId>
                <version>1.1</version>
                <executions>
                    <execution>
                        <id>echodir</id>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <phase>verify</phase>
                        <inherited>false</inherited>
                        <configuration>
                            ...</configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
</project>
```
• **id**: Self explanatory. It specifies this execution block between all of the others. When the phase is run, it will be shown in the form: [plugin:goal execution: id]. In the case of this example: [antrun:run execution: echodir]

• **goals**: Like all pluralized POM elements, this contains a list of singular elements. In this case, a list of plugin goals which are being specified by this execution block.

• **phase**: This is the phase that the list of goals will execute in. This is a very powerful option, allowing one to bind any goal to any phase in the build lifecycle, altering the default behavior of Maven.

• **inherited**: Like the inherited element above, setting this to false will suppress Maven from passing this execution onto its children. This element is only meaningful to parent POMs.

• **configuration**: Same as above, but confines the configuration to this specific list of goals, rather than all goals under the plugin.

14. Plugin Management

• **pluginManagement**: is an element that is seen along side plugins. Plugin Management contains plugin elements in much the same way, except that rather than configuring plugin information for this particular project build, it is intended to configure project builds that inherit from this one. However, this only configures plugins that are actually referenced within the plugins element in the children or in the current POM. The children have every right to override pluginManagement definitions.

```xml
...
</build>
...
</pluginManagement>
</plugins>
</plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jar-plugin</artifactId>
<version>2.6</version>
<executions>
<execution>
  <id>pre-process-classes</id>
  <phase>compile</phase>
  <goals>
    <goal>jar</goal>
  </goals>
  <configuration>
    <classifier>pre-process</classifier>
  </configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```
If we added these specifications to the plugins element, they would apply only to a single POM. However, if we apply them under the pluginManagement element, then this POM and all inheriting POMs that add the maven-jar-plugin to the build will get the pre-process-classes execution as well. So rather than the above mess included in every child pom.xml, only the following is required:

```xml
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        ...
      </plugin>
      <plugin>
        ...
      </plugin>
    </plugins>
    ...
  </build>
</project>
```

14.4.1.2 The Build Element Set
The Build type in the XSD denotes those elements that are available only for the "project build". Despite the number of extra elements (six), there are really only two groups of elements that project build contains that are missing from the profile build: directories and extensions.

14. Directories
The set of directory elements live in the parent build element, which set various directory structures for the POM as a whole. Since they do not exist in profile builds, these cannot be altered by profiles.
If the values of a *Directory element above is set as an absolute path (when their properties are expanded) then that directory is used. Otherwise, it is relative to the base build directory: ${basedir}. Please note that the scriptSourceDirectory is nowhere used in Maven and is obsolete.

14. Extensions

Extensions are a list of artifacts that are to be used in this build. They will be included in the running build's classpath. They can enable extensions to the build process (such as add an ftp provider for the Wagon transport mechanism), as well as make plugins active which make changes to the build lifecycle. In short, extensions are artifacts that are activated during build. The extensions do not have to actually do anything nor contain a Mojo. For this reason, extensions are excellent for specifying one out of multiple implementations of a common plugin interface.

```xml
...
<build>
...
<extensions>
  <extension>
    <groupId>org.apache.maven.wagon</groupId>
    <artifactId>wagon-ftp</artifactId>
    <version>1.0-alpha-3</version>
  </extension>
</extensions>
...
</build>
</project>
```

14.4.2 Reporting

Reporting contains the elements that correspond specifically for the site generation phase. Certain Maven plugins can generate reports defined and configured under the reporting element, for example: generating Javadoc reports. Much like the build element's ability to configure plugins, reporting commands the same ability. The glaring difference is that rather than fine-grained control of plug-in goals within the executions block, reporting configures goals within reportSet elements. And the subtler difference is that a plugin configuration under the reporting element works as build plugin configuration, although the opposite is not true (a build plugin configuration does not affect a reporting plugin).

Possibly the only item under the reporting element that would not be familiar to someone who understood the build element is the Boolean excludeDefaults element. This element signifies to the site generator to exclude reports normally generated by default. When a site is generated via the site build cycle, a Project Info section is placed in the left-hand menu, chock full of reports, such as the Project Team report or Dependencies list report. These report goals are generated by maven-project-info-reports-plugin. Being a plugin like any other, it may also be suppressed in the following, more verbose, way, which effectively turns off project-info reports.

```xml
...
<reporting>
  <outputDirectory>${basedir}/target/site</outputDirectory>
  <plugins>
    ...
</plugins>
</reporting>
```
The other difference is the `outputDirectory` element under `plugin`. In the case of reporting, the output directory is `${basedir}/target/site` by default.

### 14.4.2.1 Report Sets

It is important to keep in mind that an individual plugin may have multiple goals. Each goal may have a separate configuration. Report sets configure execution of a report plugin's goals. Does this sound familiar - deja-vu? The same thing was said about build's `execution` element with one difference: you cannot bind a report to another phase. Sorry.

For example, suppose you wanted to configure the `javadoc:javadoc` goal to link to "http://java.sun.com/j2se/1.5.0/docs/api/", but only the `javadoc` goal (not the goal `maven-javadoc-plugin:jar`). We would also like this configuration passed to its children, and set `inherited` to true. The `reportSet` would resemble the following:

```xml
...
<reporting>
  <plugins>
    ...
    <reportSets>
      <reportSet>
        <id>sunlink</id>
        <reports>
          <report>javadoc</report>
        </reports>
        <inherited>true</inherited>
        <configuration>
          <links>
            <link>http://java.sun.com/j2se/1.5.0/docs/api/</link>
          </links>
        </configuration>
      </reportSet>
    </reportSets>
  </plugins>
</reporting>
...
</project>
```
Between build executions and reporting reportSets, it should be clear now as to why they exist. In the simplest sense, they drill down in configuration. The POM must have a way not only to configure plugins, but must also configure the goals of those plugins. That is where these elements come in, giving the POM ultimate granularity in control of its build destiny.

### 14.5 More Project Information

Several elements do not affect the build, but rather document the project for the convenience of developers. Many of these elements are used to fill in project details when generating the project’s web site. However, like all POM declarations, plugins can use them for anything. The following are the simplest elements:

- **name**: Projects tend to have conversational names, beyond the `artifactId`. The Sun engineers did not refer to their project as “java-1.5”, but rather just called it “Tiger”. Here is where to set that value.
- **description**: A short, human readable description of the project. Although this should not replace formal documentation, a quick comment to any readers of the POM is always helpful.
- **url**: The project’s home page.
- **inceptionYear**: The year the project was first created.

#### 14.5.1 Licenses

```xml
<licenses>
  <license>
    <name>Apache License, Version 2.0</name>
    <url>https://www.apache.org/licenses/LICENSE-2.0.txt</url>
    <distribution>repo</distribution>
    <comments>A business-friendly OSS license</comments>
  </license>
</licenses>
```

Licenses are legal documents defining how and when a project (or parts of a project) may be used. A project should list licenses that apply directly to this project, and not list licenses that apply to the project’s dependencies.

- **name, url** and **comments**: are self explanatory, and have been encountered before in other contexts. Using an SPDX identifier as the license name is recommended. The fourth license element is:
- **distribution**: This describes how the project may be legally distributed. The two stated methods are repo (they may be downloaded from a Maven repository) or manual (they must be manually installed).

#### 14.5.2 Organization

Most projects are run by some sort of organization (business, private group, etc.). Here is where the most basic information is set.

```xml
<organization>
  <name>Codehaus Mojo</name>
  <url>http://mojo.codehaus.org</url>
</organization>
```
### 14.5.3 Developers

All projects consist of files that were created, at some time, by a person. Like the other systems that surround a project, so do the people involved with a project have a stake in the project. Developers are presumably members of the project's core development. Note that, although an organization may have many developers (programmers) as members, it is not good form to list them all as developers, but only those who are immediately responsible for the code. A good rule of thumb is, if the person should not be contacted about the project, they do not need to be listed here.

```xml

...<developers>
  <developer>
    <id>jdoe</id>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
    <url>http://www.example.com/jdoe</url>
    <organization>ACME</organization>
    <organizationUrl>http://www.example.com</organizationUrl>
    <roles>
      <role>architect</role>
      <role>developer</role>
    </roles>
    <timezone>America/New_York</timezone>
    <properties>
      <picUrl>http://www.example.com/jdoe/pic</picUrl>
    </properties>
  </developer>
</developers>
...
</project>
```

- **id, name, email**: These correspond to the developer's ID (presumably some unique ID across an organization), the developer's name and email address.
- **organization, organizationUrl**: As you probably guessed, these are the developer's organization name and it's URL, respectively.
- **roles**: A role should specify the standard actions that the person is responsible for. Like a single person can wear many hats, a single person can take on multiple roles.
- **timezone**: A valid time zone ID like America/New_York or Europe/Berlin, or a numerical offset in hours (and fraction) from UTC where the developer lives, e.g., -5 or +1. Time zone IDs are highly preferred because they are not affected by DST and time zone shifts. Refer to the [IANA](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones) for the official time zone database and a listing in [Wikipedia](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones).
- **properties**: This element is where any other properties about the person goes. For example, a link to a personal image or an instant messenger handle. Different plugins may use these properties, or they may simply be for other developers who read the POM.
14.5.4 Contributors
Contributors are like developers yet play an ancillary role in a project's lifecycle. Perhaps the contributor sent in a bug fix, or added some important documentation. A healthy open source project will likely have more contributors than developers.

      ...
      <contributors>
        <contributor>
          <name>Noelle</name>
          <email>some.name@gmail.com</email>
          <url>http://noellemarie.com</url>
          <organization>Noelle Marie</organization>
          <organizationUrl>http://noellemarie.com</organizationUrl>
          <roles>
            <role>tester</role>
          </roles>
          <timezone>America/Vancouver</timezone>
          <properties>
            <gtalk>some.name@gmail.com</gtalk>
          </properties>
        </contributor>
      </contributors>
      ...
    </project>

Contributors contain the same set of elements than developers sans the id element.

14.6 Environment Settings

14.6.1 Issue Management
This defines the defect tracking system (Bugzilla, TestTrack, ClearQuest, etc) used. Although there is nothing stopping a plugin from using this information for something, it's primarily used for generating project documentation.

      ...
      <issueManagement>
        <system>Bugzilla</system>
        <url>http://127.0.0.1/bugzilla</url>
      </issueManagement>
      ...
    </project>

14.6.2 Continuous Integration Management
Continuous integration build systems based upon triggers or timings (such as, hourly or daily) have grown in favor over manual builds in the past few years. As build systems have become more standardized, so have the systems that run the trigger those builds. Although the majority of the configuration is up to the specific program used (Continuum, Cruise Control, etc.), there are a few
configurations which may take place within the POM. Maven has captured a few of the recurring settings within the set of notifier elements. A notifier is the manner in which people are notified of certain build statuses. In the following example, this POM is setting a notifier of type mail (meaning email), and configuring the email address to use on the specified triggers sendOnError, sendOnFailure, and not sendOnSuccess or sendOnWarning.

```xml
  ...
  <ciManagement>
    <system>continuum</system>
    <url>http://127.0.0.1:8080/continuum</url>
    <notifiers>
      <notifier>
        <type>mail</type>
        <sendOnError>true</sendOnError>
        <sendOnFailure>true</sendOnFailure>
        <sendOnSuccess>false</sendOnSuccess>
        <sendOnWarning>false</sendOnWarning>
        <configuration><address>continuum@127.0.0.1</address></configuration>
      </notifier>
    </notifiers>
  </ciManagement>
  ...
</project>
```

14.6.3 Mailing Lists

Mailing lists are a great tool for keeping in touch with people about a project. Most mailing lists are for developers and users.

```xml
  ...
  <mailingLists>
    <mailingList>
      <name>User List</name>
      <subscribe>user-subscribe@127.0.0.1</subscribe>
      <unsubscribe>user-unsubscribe@127.0.0.1</unsubscribe>
      <post>user@127.0.0.1</post>
      <archive>http://127.0.0.1/user</archive>
      <otherArchives>
        <otherArchive>http://base.google.com/base/1/127.0.0.1</otherArchive>
      </otherArchives>
    </mailingList>
  </mailingLists>
  ...
</project>
```

- **subscribe, unsubscribe**: These elements specify the email addresses which are used for performing the relative actions. To subscribe to the user list above, a user would send an email to user-subscribe@127.0.0.1.
- **archive**: This element specifies the url of the archive of old mailing list emails, if one exists. If there are mirrored archives, they can be specified under otherArchives.
• **post**: The email address which one would use in order to post to the mailing list. Note that not all mailing lists have the ability to post to (such as a build failure list).

### 14.6.4 SCM

SCM (Software Configuration Management, also called Source Code/Control Management or, succinctly, version control) is an integral part of any healthy project. If your Maven project uses an SCM system (it does, doesn’t it?) then here is where you would place that information into the POM.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <scm>
    <connection>scm:svn:http://127.0.0.1/svn/my-project</connection>
    <developerConnection>scm:svn:https://127.0.0.1/svn/my-project</developerConnection>
    <tag>HEAD</tag>
    <url>http://127.0.0.1/websvn/my-project</url>
  </scm>
  ...
</project>
```

- **connection**, **developerConnection**: The two connection elements convey to how one is to connect to the version control system through Maven. Where connection requires read access for Maven to be able to find the source code (for example, an update), developerConnection requires a connection that will give write access. The Maven project has spawned another project named Maven SCM, which creates a common API for any SCMs that wish to implement it. The most popular are CVS and Subversion, however, there is a growing list of other supported SCMs. All SCM connections are made through a common URL structure.

  scm:[provider]:[provider_specific]

  Where provider is the type of SCM system. For example, connecting to a CVS repository may look like this:

  scm:cvs:pserver:127.0.0.1:/cvs/root:my-project

- **tag**: Specifies the tag that this project lives under. HEAD (meaning, the SCM root) is the default.
- **url**: A publicly browsable repository. For example, via ViewCVS.

### 14.6.5 Prerequisites

The POM may have certain prerequisites in order to execute correctly. For example, perhaps there was a fix in Maven 2.0.3 that you need in order to deploy using sftp.

In Maven 3, use Maven Enforcer Plugin's requireMavenVersion rule, or other rules to check other aspects.

In Maven 2, here is where you give the prerequisites to building: if these are not met, Maven will fail the build before even starting. The only element that exists as a prerequisite in POM 4.0 is the maven element, which takes a minimum version number. It is checked with Maven 2, it is not any more in Maven 3.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <prerequisites>
    <maven>2.0.6</maven>
  </prerequisites>
</project>
```
14.6.6 Repositories

Repositories are collections of artifacts which adhere to the Maven repository directory layout. In order to be a Maven repository artifact, a POM file must live within the structure $BASE_REPO/groupId/artifactId/version/artifactId-version.pom. $BASE_REPO can be local (file structure) or remote (base URL); the remaining layout will be the same. Repositories exist as a place to collect and store artifacts. Whenever a project has a dependency upon an artifact, Maven will first attempt to use a local copy of the specified artifact. If that artifact does not exist in the local repository, it will then attempt to download from a remote repository. The repository elements within a POM specify those alternate repositories to search.

The repository is one of the most powerful features of the Maven community. By default Maven searches the central repository at https://repo.maven.apache.org/maven2/. Additional repositories can be configured in the pom.xml `repositories` element.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
...
<repositories>
  <repository>
    <releases>
      <enabled>false</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
      <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <name>Nexus Snapshots</name>
    <id>snapshots-repo</id>
    <url>https://oss.sonatype.org/content/repositories/snapshots</url>
    <layout>default</layout>
  </repository>
</repositories>
<pluginRepositories>
  ...
</pluginRepositories>
...
</project>
```

- **releases, snapshots**: These are the policies for each type of artifact, Release or snapshot. With these two sets, a POM has the power to alter the policies for each type independent of the other within a single repository. For example, one may decide to enable only snapshot downloads, possibly for development purposes.

- **enabled**: true or false for whether this repository is enabled for the respective type (releases or snapshots).

- **updatePolicy**: This element specifies how often updates should attempt to occur. Maven will compare the local POM’s timestamp (stored in a repository’s maven-metadata file) to the remote.
The choices are: always, daily (default), interval:X (where X is an integer in minutes) or never.

- **checksumPolicy**: When Maven deploys files to the repository, it also deploys corresponding checksum files. Your options are to ignore, fail, or warn on missing or incorrect checksums.
- **layout**: In the above description of repositories, it was mentioned that they all follow a common layout. This is mostly correct. The layout introduced with Maven 2 is the default layout for repositories used by Maven both 2 & 3; however, Maven 1.x had a different layout. Use this element to specify which if it is default or legacy.

### 14.6.7 Plugin Repositories

Repositories are home to two major types of artifacts. The first are artifacts that are used as dependencies of other artifacts. These are the majority of artifacts that reside within central. The other type of artifact is plugins. Maven plugins are themselves a special type of artifact. Because of this, plugin repositories may be separated from other repositories (although, I have yet to hear a convincing argument for doing so). In any case, the structure of the pluginRepositories element block is similar to the repositories element. The pluginRepository elements each specify a remote location of where Maven can find new plugins.

### 14.6.8 Distribution Management

Distribution management acts precisely as it sounds: it manages the distribution of the artifact and supporting files generated throughout the build process. Starting with the last elements first:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <distributionManagement>
    ...
    <downloadUrl>http://mojo.codehaus.org/my-project</downloadUrl>
    <status>deployed</status>
  </distributionManagement>
  ...
</project>
```

- **downloadUrl**: is the URL of the repository which another POM can point to in order to grab this POM's artifact. In the simplest terms, we told the POM how to upload it (through repository/url), but from where can the public download it? This element answers that question.
- **status**: Warning! Like a baby bird in a nest, the status should never be touched by human hands! The reason for this is that Maven will set the status of the project when it is transported out to the repository. Its valid types are as follows.

  - **none**: No special status. This is the default for a POM.
  - **converted**: The manager of the repository converted this POM from an earlier version to Maven 2.
  - **partner**: This artifact has been synchronized with a partner repository.
  - **deployed**: By far the most common status, meaning that this artifact was deployed from a Maven 2 or 3 instance. This is what you get when you manually deploy using the command-line deploy phase.
  - **verified**: This project has been verified, and should be considered finalized.
14.6.8.1 Repository
Whereas the repositories element specifies in the POM the location and manner in which Maven may download remote artifacts for use by the current project, distributionManagement specifies where (and how) this project will get to a remote repository when it is deployed. The repository elements will be used for snapshot distribution if the snapshotRepository is not defined.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <distributionManagement>
        <repository>
            <uniqueVersion>false</uniqueVersion>
            <id>corp1</id>
            <name>Corporate Repository</name>
            <url>scp://repo/maven2</url>
            <layout>default</layout>
        </repository>
        <snapshotRepository>
            <uniqueVersion>true</uniqueVersion>
            <id>propSnap</id>
            <name>Propellors Snapshots</name>
            <url>sftp://propellers.net/maven</url>
            <layout>legacy</layout>
        </snapshotRepository>
        ...
    </distributionManagement>
    ...
</project>
```

- **id, name**: The id is used to uniquely identify this repository amongst many, and the name is a human readable form.
- **uniqueVersion**: The unique version takes a true or false value to denote whether artifacts deployed to this repository should get a uniquely generated version number, or use the version number defined as part of the address.
- **url**: This is the core of the repository element. It specifies both the location and the transport protocol used to transfer a built artifact (and POM file, and checksum data) to the repository.
- **layout**: These are the same types and purpose as the layout element defined in the repository element. They are default and legacy.

14.6.8.2 Site Distribution
More than distribution to the repositories, distributionManagement is responsible for defining how to deploy the project’s site and documentation.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <distributionManagement>
        ...
    </distributionManagement>
    ...
    <site>
        <id>mojo.website</id>
        <name>Mojo Website</name>
        <url>scp://beaver.codehaus.org/home/projects/mojo/public_html/</url>
    </site>
</project>
```
• **id, name, url**: These elements are similar to their counterparts above in the `distributionManagement` repository element.

### 14.6.8.3 Relocation

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <distributionManagement>
    ...
    <relocation>
      <groupId>org.apache</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0</version>
      <message>We have moved the Project under Apache</message>
    </relocation>
    ...
  </distributionManagement>
  ...
</project>
```

Projects are not static; they are living things (or dying things, as the case may be). A common thing that happens as projects grow, is that they are forced to move to more suitable quarters. For example, when your next wildly successful open source project moves under the Apache umbrella, it would be good to give users a heads-up that the project is being renamed to `org.apache:my-project:1.0`. Besides specifying the new address, it is also good form to provide a message explaining why.

### 14.6.9 Profiles

A new feature of the POM 4.0 is the ability of a project to change settings depending on the environment where it is being built. A `profile` element contains both an optional activation (a profile trigger) and the set of changes to be made to the POM if that profile has been activated. For example, a project built for a test environment may point to a different database than that of the final deployment. Or dependencies may be pulled from different repositories based upon the JDK version used. The elements of profiles are as follows:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>...</activation>
      <build>...</build>
      <modules>...</modules>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <dependencies>...</dependencies>
      <reporting>...</reporting>
    </profile>
    ...
  </profiles>
</project>
```
14.6.9.1 Activation

Activations are the key of a profile. The power of a profile comes from its ability to modify the basic POM only under certain circumstances. Those circumstances are specified via an `activation` element.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
      </activation>
    </profile>
    ...
  </profiles>
</project>
```

Before Maven 3.2.2 Activation occurs when one or more of the specified criteria have been met. When the first positive result is encountered, processing stops and the profile is marked as active. Since Maven 3.2.2 Activation occurs when all of the specified criteria have been met.

- **jdk:** Activation has a built in, Java-centric check in the `jdk` element. This will activate if the test is run under a jdk version number that matches the prefix given. In the above example, 1.5.0_06 will match. Ranges are also supported. See the [maven-enforcer-plugin](https://maven.apache.org/enforcer-plugin/dependency-management.html) for more details about supported ranges.
- **os:** The `os` element can define some operating system specific properties shown above. See the [maven-enforcer-plugins](https://maven.apache.org/enforcer-plugin/dependency-management.html) RequireOS Rule for more details about OS values.
- **property:** The profile will activate if Maven detects a system property and commandline property (a value which can be dereferenced within the POM by `${name}`) of the corresponding name=value pair.
- **file**: Finally, a given filename may activate the profile by the existence of a file, or if it is missing. **NOTE**: interpolation for this element is limited to ${basedir}, System properties and request properties.

The activation element is not the only way that a profile may be activated. The settings.xml file's `activeProfile` element may contain the profile's id. They may also be activated explicitly through the command line via a comma separated list after the `-P` flag (e.g. `-P codecoverage`).

*To see which profile will activate in a certain build, use the maven-help-plugin.*

```mvn help:active-profiles```

14.6.9.2 The BaseBuild Element Set *(revisited)*

As mentioned above, the reason for the two types of build elements reside in the fact that it does not make sense for a profile to configure build directories or extensions as it does in the top level of the POM. Regardless of in which environment the project is built, some values will remain constant, such as the directory structure of the source code. *If you find your project needing to keep two sets of code for different environments, it may be prudent to investigate refactoring the project into two or more separate projects.*

14.7 Final

The Maven POM is big. However, its size is also a testament to its versatility. The ability to abstract all of the aspects of a project into a single artifact is powerful, to say the least. Gone are the days of dozens of disparate build scripts and scattered documentation concerning each individual project. Along with Maven's other stars that make up the Maven galaxy - a well defined build lifecycle, easy to write and maintain plugins, centralized repositories, system-wide and user-based configurations, as well as the increasing number of tools to make developers' jobs easier to maintain complex projects - the POM is the large, but bright, center.

Aspects of this guide were originally published in the Maven 2 Pom Demystified.
15 Settings Reference

15.1 Settings Reference

1. Introduction
   1. Quick Overview
2. Settings Details
   1. Simple Values
   2. Plugin Groups
   3. Servers
      1. Password Encryption
   4. Mirrors
   5. Proxies
   6. Profiles
      1. Activation
      2. Repositories
      3. Plugin Repositories
   7. Active Profiles

15.2 Introduction

15.2.1 Quick Overview

The settings element in the settings.xml file contains elements used to define values which configure Maven execution in various ways, like the pom.xml, but should not be bundled to any specific project, or distributed to an audience. These include values such as the local repository location, alternate remote repository servers, and authentication information.

There are two locations where a settings.xml file may live:

- The Maven install: ${maven.home}/conf/settings.xml
- A user's install: ${user.home}/.m2/settings.xml

The former settings.xml are also called global settings, the latter settings.xml are referred to as user settings. If both files exists, their contents gets merged, with the user-specific settings.xml being dominant.

Tip: If you need to create user-specific settings from scratch, it's easiest to copy the global settings from your Maven installation to your ${user.home}/.m2 directory. Maven's default settings.xml is a template with comments and examples so you can quickly tweak it to match your needs.

Here is an overview of the top elements under 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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository/>
  <interactiveMode/>
  <offline/>
</settings>
```
<pluginGroups/>
<servers/>
<mirrors/>
<proxies/>
<profiles/>
<activeProfiles/>
</settings>

The contents of the settings.xml can be interpolated using the following expressions:

1. `${user.home}` and all other system properties (since Maven 3.0)
2. `${env.HOME}` etc. for environment variables

Note that properties defined in profiles within the settings.xml cannot be used for interpolation.

### 15.3 Settings Details

#### 15.3.1 Simple Values

Half of the top-level settings elements are simple values, representing a range of values which describe elements of the build system that are active full-time.

```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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <offline>false</offline>
  ...
</settings>
```

- **localRepository**: This value is the path of this build system's local repository. The default value is `${user.home}/.m2/repository`. This element is especially useful for a main build server allowing all logged-in users to build from a common local repository.

- **interactiveMode**: `true` if Maven should attempt to interact with the user for input, `false` if not. Defaults to `true`.

- **offline**: `true` if this build system should operate in offline mode, defaults to `false`. This element is useful for build servers which cannot connect to a remote repository, either because of network setup or security reasons.

#### 15.3.2 Plugin Groups

This element contains a list of pluginGroup elements, each contains a groupId. The list is searched when a plugin is used and the groupId is not provided in the command line. This list automatically contains org.apache.maven.plugins and org.codehaus.mojo.

```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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <pluginGroups>
    <pluginGroup>org.eclipse.jetty</pluginGroup>
  </pluginGroups>
  ...
</settings>
```
For example, given the above settings the Maven command line may execute
`org.eclipse.jetty:jetty-maven-plugin:run` with the truncated command:
```
mvn jetty:run
```

### 15.3.3 Servers

The repositories for download and deployment are defined by the `repositories` and `distributionManagement` elements of the POM. However, certain settings such as `username` and `password` should not be distributed along with the `pom.xml`. This type of information should exist on the build server in the `settings.xml`.

```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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
    ...
    <servers>
        <server>
            <id>server001</id>
            <username>my_login</username>
            <password>my_password</password>
            <privateKey>${user.home}/.ssh/id_dsa</privateKey>
            <filePermissions>664</filePermissions>
            <directoryPermissions>775</directoryPermissions>
            <configuration>
                ...
            </configuration>
        </server>
    </servers>
    ...
</settings>
```

- **id**: This is the ID of the server *(not of the user to login as)* that matches the `id` element of the repository/mirror that Maven tries to connect to.
- **username, password**: These elements appear as a pair denoting the login and password required to authenticate to this server.
- **privateKey, passphrase**: Like the previous two elements, this pair specifies a path to a private key (default is `${user.home}/.ssh/id_dsa`) and a passphrase, if required. The passphrase and password elements may be externalized in the future, but for now they must be set plain-text in the `settings.xml` file.
- **filePermissions, directoryPermissions**: When a repository file or directory is created on deployment, these are the permissions to use. The legal values of each is a three digit number corresponding to *nix file permissions, e.g. 664, or 775.

*Note*: If you use a private key to login to the server, make sure you omit the `<password>` element. Otherwise, the key will be ignored.

#### 15.3.3.1 Password Encryption

A new feature - server password and passphrase encryption has been added to 2.1.0+. See details on this page.

### 15.3.4 Mirrors
• **id, name**: The unique identifier and user-friendly name of this mirror. The *id* is used to differentiate between *mirror* elements and to pick the corresponding credentials from the *<servers>* section when connecting to the mirror.

• **url**: The base URL of this mirror. The build system will use this URL to connect to a repository rather than the original repository URL.

• **mirrorOf**: The *id* of the repository that this is a mirror of. For example, to point to a mirror of the Maven central repository (https://repo.maven.apache.org/maven2/), set this element to *central*. More advanced mappings like *repo1,repo2* or *!*inhouse are also possible. This must not match the mirror *id*.

For a more in-depth introduction of mirrors, please read the [Guide to Mirror Settings](https://maven.apache.org/guides/mini/g7-mirrors.html).

### 15.3.5 Proxies

```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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

• **id**: The unique identifier for this proxy. This is used to differentiate between *proxy* elements.

• **active**: *true* if this proxy is active. This is useful for declaring a set of proxies, but only one may be active at a time.

• **protocol, host, port**: The protocol://host:port of the proxy, separated into discrete elements.

• **username, password**: These elements appear as a pair denoting the login and password required to authenticate to this proxy server.
• **nonProxyHosts**: This is a list of hosts which should not be proxied. The delimiter of the list is the expected type of the proxy server; the example above is pipe delimited - comma delimited is also common.

### 15.3.6 Profiles

The profile element in the settings.xml is a truncated version of the pom.xml profile element. It consists of the activation, repositories, pluginRepositories and properties elements. The profile elements only include these four elements because they concerns themselves with the build system as a whole (which is the role of the settings.xml file), not about individual project object model settings.

If a profile is active from settings, its values will override any equivalently ID'd profiles in a POM or profiles.xml file.

#### 15.3.6.1 Activation

Activations are the key of a profile. Like the POM's profiles, the power of a profile comes from its ability to modify some values only under certain circumstances; those circumstances are specified via an activation element.

```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">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.3</value>
        </property>
        <file>
          <exists>${basedir}/file2.properties</exists>
          <missing>${basedir}/file1.properties</missing>
        </file>
      </activation>
    </profile>
    ...
  </profiles>
  ...
</settings>
```

Activation occurs when all specified criteria have been met, though not all are required at once.
• **jdk**: activation has a built-in, Java-centric check in the jdk element. This will activate if the test is run under a jdk version number that matches the prefix given. In the above example, 1.5.0_06 will match. Ranges are also supported. See the [maven-enforcer-plugin](https://maven.apache.org/enforcer-plugin/) for more details about supported ranges.

• **os**: The os element can define some operating system specific properties shown above. See the [maven-enforcer-plugin](https://maven.apache.org/enforcer-plugin/) for more details about OS values.

• **property**: The profile will activate if Maven detects a property (a value which can be dereferenced within the POM by `$(name)`) of the corresponding name=value pair.

• **file**: Finally, a given filename may activate the profile by the existence of a file, or if it is missing.

The activation element is not the only way that a profile may be activated. The settings.xml file's activeProfile element may contain the profile's id. They may also be activated explicitly through the command line via a comma separated list after the `-P` flag (e.g. `-P test`).

*To see which profile will activate in a certain build, use the maven-help-plugin.*

```
mvn help:active-profiles
```

### 15.3.6.2 Properties

Maven properties are value placeholders, like properties in Ant. Their values are accessible anywhere within a POM by using the notation `$\{X\}$`, where `X` is the property. They come in five different styles, all accessible from the settings.xml file:

1. **env.X**: Prefixing a variable with “env.” will return the shell’s environment variable. For example, `$\{env.PATH\}` contains the `$PATH` environment variable ( `%PATH%` in Windows).

2. **project.x**: A dot (.) notated path in the POM will contain the corresponding element’s value. For example: `<project><version>1.0</version></project>` is accessible via `$\{project.version\}`.

3. **settings.x**: A dot (.) notated path in the settings.xml will contain the corresponding element’s value. For example: `<settings><offline>false</offline></settings>` is accessible via `$\{settings.offline\}`.

4. **Java System Properties**: All properties accessible via `java.lang.System.getProperties()` are available as POM properties, such as `$\{java.home\}`.

5. **x**: Set within a `<properties />` element or an external file, the value may be used as `$\{someVar\}`.

```
<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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <properties>
        <user.install>${user.home}/our-project</user.install>
      </properties>
    
  </profiles>
  ...
</settings>
```

The property `$\{user.install\}` is accessible from a POM if this profile is active.
15.3.6.3 Repositories

Repositories are remote collections of projects from which Maven uses to populate the local repository of the build system. It is from this local repository that Maven calls its plugins and dependencies. Different remote repositories may contain different projects, and under the active profile they may be searched for a matching release or snapshot artifact.

```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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <repositories>
        <repository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>http://snapshots.maven.codehaus.org/maven2/</url>
          <layout>default</layout>
        </repository>
      </repositories>
      ...
      <pluginRepositories>
        <pluginRepository>
          <id>myPluginRepo</id>
          <name>My Plugins repo</name>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
          <url>https://maven-central-eu....com/maven2/</url>
        </pluginRepository>
        ...
      </pluginRepositories>
    </profile>
  </profiles>
  ...
</settings>
```

- releases, snapshots: These are the policies for each type of artifact, Release or snapshot. With these two sets, a POM has the power to alter the policies for each type independent of the other
within a single repository. For example, one may decide to enable only snapshot downloads, possibly for development purposes.

- **enabled**: true or false for whether this repository is enabled for the respective type (releases or snapshots).

- **updatePolicy**: This element specifies how often updates should attempt to occur. Maven will compare the local POM’s timestamp (stored in a repository’s maven-metadata file) to the remote. The choices are: always, daily (default), interval:X (where X is an integer in minutes) or never.

- **checksumPolicy**: When Maven deploys files to the repository, it also deploys corresponding checksum files. Your options are to ignore, fail, or warn on missing or incorrect checksums.

- **layout**: In the above description of repositories, it was mentioned that they all follow a common layout. This is mostly correct. Maven 2 has a default layout for its repositories; however, Maven 1.x had a different layout. Use this element to specify which if it is default or legacy.

### 15.3.6.4 Plugin Repositories

Repositories are home to two major types of artifacts. The first are artifacts that are used as dependencies of other artifacts. These are the majority of artifacts that reside within central. The other type of artifact is plugins. Maven plugins are themselves a special type of artifact. Because of this, plugin repositories may be separated from other repositories (although, I have yet to hear a convincing argument for doing so). In any case, the structure of the pluginRepositories element block is similar to the repositories element. The pluginRepository elements each specify a remote location of where Maven can find new plugins.

### 15.3.7 Active Profiles

```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 https://maven.apache.org/xsd/settings-1.0.0.xsd">
...
<activeProfiles>
  <activeProfile>env-test</activeProfile>
</activeProfiles>
</settings>
```

The final piece of the settings.xml puzzle is the activeProfiles element. This contains a set of activeProfile elements, which each have a value of a profile id. Any profile id defined as an activeProfile will be active, regardless of any environment settings. If no matching profile is found nothing will happen. For example, if env-test is an activeProfile, a profile in a pom.xml (or profile.xml with a corresponding id will be active. If no such profile is found then execution will continue as normal.
16 Guides

16.1 Documentation

16.1.1 Getting Started with Maven

- Getting Started in 5 Minutes
- Getting Started in 30 Minutes

16.1.2 Introductions

- The Build Lifecycle
- The POM
- Profiles
- Repositories
- Standard Directory Layout
- The Dependency Mechanism
  - Optional Dependencies and Dependency Exclusions

16.1.2.1 Plugins

- Plugin Development
- Configuring Plug-ins
- Plugin Prefix Resolution
- Developing Java Plugins

16.1.2.2 Site

- Creating a Site
- The APT Format
- Snippet Macro

16.1.2.3 Archetypes

- What is an Archetype
- Creating Archetypes

16.1.2.4 Upgrading

- Relocation of Artifacts

16.1.2.5 Repositories

- Installing 3rd party JARs to Local Repository
- Deploying 3rd party JARs to Remote Repository
- Repository Management
- Using Multiple Repositories
• Large Scale Centralized Deployments
• Mirror Settings
• Deployment and Security Settings
• Using Proxies
• Authenticated HTTPS
• Remote repository access through authenticated HTTPS
• Relocation

16.1.3 Guides
• Configuring for Reproducible Builds
• Creating Assemblies
• Configuring Archive Plugins
• Configuring Maven
• Generating Sources
• Working with Manifests
• Maven Classloading
• Using Multiple Modules in a Build
• Using the Release Plugin
• Using Ant with Maven
• Using Modello
• Using Extensions
• Building For Different Environments with Maven 2
• Using Toolchains
• Encrypting passwords in settings.xml
• Guide to HTTP Connection Settings
• Guide to Selecting Alternative Wagon Providers
• Guide to Configuring Default Mojo Executions
• Guide to Large Scale Centralized Deployments

16.1.3.1 Maven Tools and IDE Integration
• Maven Auto-Completion Using BASH

16.1.4 Development Guides
• Building Maven from Scratch
• Developing Maven
• The Plugin Documentation Standard
• Maven Documentation Style

16.1.5 The Maven Community
• The Maven Community
• Helping with Maven
• Guide for New Committers
• Testing Development Versions of Plugins
16.1.5.1 Conventions

- Maven Conventions
- Naming Conventions
- When You Can't Use the Conventions

16.1.5.2 The Central Repository

- Introduction to the Central Repository
- Uploading Artifacts to the Central Repository
- Improving the Repository

16.1.6 References

- POM Overview (Technical Project Descriptor)
- Settings Overview (Technical Settings Descriptor)
- Core Plug-ins List
- Mojo API
- Glossary
- Maven Quick Reference Card - PDF

16.1.7 Javadoc API

Here is some useful Javadoc API links to the current version of Maven:

- Maven Artifact
- Maven Reporting
- Maven Plugin API
- Maven Model
- Maven Core
- Maven Settings

You can also browse the full technical documentation references of the current version of Maven.
17 The Build Lifecycle

17.1 Introduction to the Build Lifecycle

17.1.1 Table Of Contents

- Build Lifecycle Basics
- Setting Up Your Project to Use the Build Lifecycle
  - Packaging
  - Plugins
- Lifecycle Reference
- Built-in Lifecycle Bindings

17.1.2 Build Lifecycle Basics

Maven is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

There are three built-in build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's web site.

17.1.2.1 A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

- validate - validate the project is correct and all necessary information is available
- compile - compile the source code of the project
- test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- package - take the compiled code and package it in its distributable format, such as a JAR.
- verify - run any checks on results of integration tests to ensure quality criteria are met
- install - install the package into the local repository, for use as a dependency in other projects locally
- deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

[top]
17.1.2.2 Usual Command Line Calls

You should select the phase that matches your outcome. If you want your jar, run `package`. If you want to run the unit tests, run `test`.

If you are uncertain what you want, the preferred phase to call is `mvn verify`:

This command executes each default lifecycle phase in order (`validate`, `compile`, `package`, etc.), before executing `verify`. You only need to call the last build phase to be executed, in this case, `verify`. In most cases the effect is the same as `package`. However, in case there are integration-tests, these will be executed as well. And during the `verify` phase some additional checks can be done, e.g. if your code written according to the predefined checkstyle rules.

In a build environment, use the following call to cleanly build and deploy artifacts into the shared repository.

`mvn clean deploy`

The same command can be used in a multi-module scenario (i.e. a project with one or more subprojects). Maven traverses into every subproject and executes `clean`, then executes `deploy` (including all of the prior build phase steps).

17.1.2.3 A Build Phase is Made Up of Plugin Goals

However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the plugin goals bound to those build phases.

A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation. The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The `clean` and `package` arguments are build phases, while the `dependency:copy-dependencies` is a goal (of a plugin).

`mvn clean dependency:copy-dependencies package`

If this were to be executed, the `clean` phase will be executed first (meaning it will run all preceding phases of the clean lifecycle, plus the `clean` phase itself), and then the `dependency:copy-dependencies` goal, before finally executing the `package` phase (and all its preceding build phases of the default lifecycle).

Moreover, if a goal is bound to one or more build phases, that goal will be called in all those phases. Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals.

( Note: In Maven 2.0.5 and above, multiple goals bound to a phase are executed in the same order as they are declared in the POM, however multiple instances of the same plugin are not supported. Multiple instances of the same plugin are grouped to execute together and ordered in Maven 2.0.11 and above).

17.1.2.4 Some Phases Are Not Usually Called From the Command Line

The phases named with hyphenated-words (`pre-*`, `post-*`, or `process-*`) are not usually directly called from the command line. These phases sequence the build, producing intermediate results that
are not useful outside the build. In the case of invoking `integration-test`, the environment may be left in a hanging state.

Code coverage tools such as Jacoco and execution container plugins such as Tomcat, Cargo, and Docker bind goals to the `pre-integration-test` phase to prepare the integration test container environment. These plugins also bind goals to the `post-integration-test` phase to collect coverage statistics or decommission the integration test container.

Failsafe and code coverage plugins bind goals to `integration-test` and `verify` phases. The net result is test and coverage reports are available after the `verify` phase. If `integration-test` were to be called from the command line, no reports are generated. Worse is that the integration test container environment is left in a hanging state; the Tomcat webserver or Docker instance is left running, and Maven may not even terminate by itself.

17.1.3 Setting Up Your Project to Use the Build Lifecycle

The build lifecycle is simple enough to use, but when you are constructing a Maven build for a project, how do you go about assigning tasks to each of those build phases?

17.1.3.1 Packaging

The first, and most common way, is to set the packaging for your project via the equally named POM element `<packaging>`. Some of the valid packaging values are `jar`, `war`, `ear` and `pom`. If no packaging value has been specified, it will default to `jar`.

Each packaging contains a list of goals to bind to a particular phase. For example, the `jar` packaging will bind the following goals to build phases of the default lifecycle.

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>process-resources</td>
<td>resources:resources</td>
</tr>
<tr>
<td>compile</td>
<td>compiler:compile</td>
</tr>
<tr>
<td>process-test-resources</td>
<td>resources:testResources</td>
</tr>
<tr>
<td>test-compile</td>
<td>compiler:testCompile</td>
</tr>
<tr>
<td>test</td>
<td>surefire:test</td>
</tr>
<tr>
<td>package</td>
<td>jar:jar</td>
</tr>
<tr>
<td>install</td>
<td>install:install</td>
</tr>
<tr>
<td>deploy</td>
<td>deploy:deploy</td>
</tr>
</tbody>
</table>

This is an almost standard set of bindings; however, some packagings handle them differently. For example, a project that is purely metadata (packaging value is `pom`) only binds goals to the `install` and `deploy` phases (for a complete list of goal-to-build-phase bindings of some of the packaging types, refer to the Lifecycle Reference).

Note that for some packaging types to be available, you may also need to include a particular plugin in the `<build>` section of your POM and specify `<extensions>true</extensions>` for that plugin. One example of a plugin that requires this is the Plexus plugin, which provides a `plexus-application` and `plexus-service` packaging.
17.1.3.2 Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

As you will see in the later sections, plugins can contain information that indicates which lifecycle phase to bind a goal to. Note that adding the plugin on its own is not enough information - you must also specify the goals you want to run as part of your build.

The goals that are configured will be added to the goals already bound to the lifecycle from the packaging selected. If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM. Note that you can use the <executions> element to gain more control over the order of particular goals.

For example, the Modello plugin binds by default its goal modello:java to the generate-sources phase (Note: The modello:java goal generates Java source codes). So to use the Modello plugin and have it generate sources from a model and incorporate that into the build, you would add the following to your POM in the <plugins> section of <build>:

...  
<plugin>  
  <groupId>org.codehaus.modello</groupId>  
  <artifactId>modello-maven-plugin</artifactId>  
  <version>1.8.1</version>  
  <executions>  
    <execution>  
      <configuration>  
        <models>  
          <model>src/main/mdo/maven.mdo</model>  
        </models>  
        <version>4.0.0</version>  
      </configuration>  
      <goals>  
        <goal>java</goal>  
      </goals>  
    </execution>  
  </executions>  
</plugin>  
...

You might be wondering why that <executions> element is there. That is so that you can run the same goal multiple times with different configuration if needed. Separate executions can also be given an ID so that during inheritance or the application of profiles you can control whether goal configuration is merged or turned into an additional execution.

When multiple executions are given that match a particular phase, they are executed in the order specified in the POM, with inherited executions running first.

Now, in the case of modello:java, it only makes sense in the generate-sources phase. But some goals can be used in more than one phase, and there may not be a sensible default. For those, you can specify the phase yourself. For example, let's say you have a goal display:time that echos the current time to the commandline, and you want it to run in the process-test-resources phase to indicate when the tests were started. This would be configured like so:

...
<plugin>
  <groupId>com.mycompany.example</groupId>
  <artifactId>display-maven-plugin</artifactId>
  <version>1.0</version>
  <executions>
    <execution>
      <phase>process-test-resources</phase>
      <goals>
        <goal>time</goal>
      </goals>
    </execution>
  </executions>
</plugin>

17.1.4 Lifecycle Reference

The following lists all build phases of the default, clean and site lifecycles, which are executed in the order given up to the point of the one specified.

17.1.4.1 Clean Lifecycle

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre-clean</td>
<td>execute processes needed prior to the actual project cleaning</td>
</tr>
<tr>
<td>clean</td>
<td>remove all files generated by the previous build</td>
</tr>
<tr>
<td>post-clean</td>
<td>execute processes needed to finalize the project cleaning</td>
</tr>
</tbody>
</table>

17.1.4.2 Default Lifecycle

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validate</td>
<td>validate the project is correct and all necessary information is available.</td>
</tr>
<tr>
<td>initialize</td>
<td>initialize build state, e.g. set properties or create directories.</td>
</tr>
<tr>
<td>generate-sources</td>
<td>generate any source code for inclusion in compilation.</td>
</tr>
<tr>
<td>process-sources</td>
<td>process the source code, for example to filter any values.</td>
</tr>
<tr>
<td>generate-resources</td>
<td>generate resources for inclusion in the package.</td>
</tr>
<tr>
<td>process-resources</td>
<td>copy and process the resources into the destination directory, ready for packaging.</td>
</tr>
<tr>
<td>compile</td>
<td>compile the source code of the project.</td>
</tr>
<tr>
<td>process-classes</td>
<td>post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.</td>
</tr>
<tr>
<td>Task</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>generate-test-sources</td>
<td>generate any test source code for inclusion in compilation.</td>
</tr>
<tr>
<td>process-test-sources</td>
<td>process the test source code, for example to filter any values.</td>
</tr>
<tr>
<td>generate-test-resources</td>
<td>create resources for testing.</td>
</tr>
<tr>
<td>process-test-resources</td>
<td>copy and process the resources into the test destination directory.</td>
</tr>
<tr>
<td>test-compile</td>
<td>compile the test source code into the test destination directory.</td>
</tr>
<tr>
<td>process-test-classes</td>
<td>post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes.</td>
</tr>
<tr>
<td>test</td>
<td>run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.</td>
</tr>
<tr>
<td>prepare-package</td>
<td>perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package.</td>
</tr>
<tr>
<td>package</td>
<td>take the compiled code and package it in its distributable format, such as a JAR.</td>
</tr>
<tr>
<td>pre-integration-test</td>
<td>perform actions required before integration tests are executed. This may involve things such as setting up the required environment.</td>
</tr>
<tr>
<td>integration-test</td>
<td>process and deploy the package if necessary into an environment where integration tests can be run.</td>
</tr>
<tr>
<td>post-integration-test</td>
<td>perform actions required after integration tests have been executed. This may including cleaning up the environment.</td>
</tr>
<tr>
<td>verify</td>
<td>run any checks to verify the package is valid and meets quality criteria.</td>
</tr>
<tr>
<td>install</td>
<td>install the package into the local repository, for use as a dependency in other projects locally.</td>
</tr>
<tr>
<td>deploy</td>
<td>done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.</td>
</tr>
</tbody>
</table>

### 17.1.4.3 Site Lifecycle

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre-site</td>
<td>execute processes needed prior to the actual project site generation</td>
</tr>
<tr>
<td>site</td>
<td>generate the project's site documentation</td>
</tr>
<tr>
<td>post-site</td>
<td>execute processes needed to finalize the site generation, and to prepare for site deployment</td>
</tr>
</tbody>
</table>
17 The Build Lifecycle

17.1.5 Built-in Lifecycle Bindings

Some phases have goals bound to them by default. And for the default lifecycle, these bindings depend on the packaging value. Here are some of the goal-to-build-phase bindings.

17.1.5.1 Clean Lifecycle Bindings

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>clean</td>
<td>clean:clean</td>
</tr>
</tbody>
</table>

17.1.5.2 Default Lifecycle Bindings - Packaging ejb/ejb3/jar/par/rar/war

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>process-resources</td>
<td>resources:resources</td>
</tr>
<tr>
<td>compile</td>
<td>compiler:compile</td>
</tr>
<tr>
<td>process-test-resources</td>
<td>resources:testResources</td>
</tr>
<tr>
<td>test-compile</td>
<td>compiler:testCompile</td>
</tr>
<tr>
<td>test</td>
<td>surefire:test</td>
</tr>
<tr>
<td>package</td>
<td>ejb:ejb or ejb3:ejb3 or jar:jar or par:par or rar:rar or war:war</td>
</tr>
<tr>
<td>install</td>
<td>install:install</td>
</tr>
<tr>
<td>deploy</td>
<td>deploy:deploy</td>
</tr>
</tbody>
</table>

17.1.5.3 Default Lifecycle Bindings - Packaging ear

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate-resources</td>
<td>ear:generate-application-xml</td>
</tr>
<tr>
<td>process-resources</td>
<td>resources:resources</td>
</tr>
<tr>
<td>package</td>
<td>ear:ear</td>
</tr>
<tr>
<td>install</td>
<td>install:install</td>
</tr>
<tr>
<td>deploy</td>
<td>deploy:deploy</td>
</tr>
</tbody>
</table>

17.1.5.4 Default Lifecycle Bindings - Packaging maven-plugin

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>site-deploy</td>
<td>deploy the generated site documentation to the specified web server</td>
</tr>
</tbody>
</table>
### 17.1.5.5 Default Lifecycle Bindings - Packaging `pom`

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td><code>install:install</code></td>
</tr>
<tr>
<td>deploy</td>
<td><code>deploy:deploy</code></td>
</tr>
</tbody>
</table>

### 17.1.5.6 Site Lifecycle Bindings

<table>
<thead>
<tr>
<th>Phase</th>
<th>plugin:goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td><code>site:site</code></td>
</tr>
<tr>
<td>site-deploy</td>
<td><code>site:deploy</code></td>
</tr>
</tbody>
</table>

### 17.1.5.7 References

The full Maven lifecycle is defined by the `components.xml` file in the `maven-core` module, with associated documentation for reference.

Default lifecycle bindings are defined in a separate `default-bindings.xml` descriptor.

See Lifecycles Reference and Plugin Bindings for default Lifecycle Reference for latest documentation taken directly from source code.

[top]
18 The POM

18.1 Introduction to the POM

- What is a POM?
- Super POM
- Minimal POM
- Project Inheritance
  - Example 1
  - Example 2
- Project Aggregation
  - Example 3
  - Example 4
- Project Inheritance vs Project Aggregation
  - Example 5
- Project Interpolation and Variables
  - Available Variables

18.1.1 What is a POM?

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this include the build directory, which is `target`; the source directory, which is `src/main/java`; the test source directory, which is `src/test/java`; and so on. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

[top]

18.1.2 Super POM

The Super POM is Maven's default POM. All POMs extend the Super POM unless explicitly set, meaning the configuration specified in the Super POM is inherited by the POMs you created for your projects.

You can see the Super POM for Maven 3.6.3 in Maven Core reference documentation.

[top]

18.1.3 Minimal POM

The minimum requirement for a POM are the following:

- project root
- modelVersion - should be set to 4.0.0
- groupId - the id of the project's group.
- artifactId - the id of the artifact (project)
- version - the version of the artifact under the specified group

Here's an example:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

A POM requires that its groupId, artifactId, and version be configured. These three values form the project's fully qualified artifact name. This is in the form of `<groupId>:<artifactId>:<version>`. As for the example above, its fully qualified artifact name is "com.mycompany.app:my-app:1".

Also, as mentioned in the first section, if the configuration details are not specified, Maven will use their defaults. One of these default values is the packaging type. Every Maven project has a packaging type. If it is not specified in the POM, then the default value "jar" would be used.

Furthermore, you can see that in the minimal POM the repositories were not specified. If you build your project using the minimal POM, it would inherit the repositories configuration in the Super POM. Therefore when Maven sees the dependencies in the minimal POM, it would know that these dependencies will be downloaded from https://repo.maven.apache.org/maven2 which was specified in the Super POM.

18.1.4 Project Inheritance

Elements in the POM that are merged are the following:

- dependencies
- developers and contributors
- plugin lists (including reports)
- plugin executions with matching ids
- plugin configuration
- resources

The Super POM is one example of project inheritance, however you can also introduce your own parent POMs by specifying the parent element in the POM, as demonstrated in the following examples.

18.1.4.1 Example 1

As an example, let us reuse our previous artifact, com.mycompany.app:my-app:1. And let us introduce another artifact, com.mycompany.app:my-module:1.

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```
And let us specify their directory structure as the following:

```
|-- my-module
|   `-- pom.xml
`-- pom.xml
```

**Note:** `my-module/pom.xml` is the POM of `com.mycompany.app:my-module:1` while `pom.xml` is the POM of `com.mycompany.app:my-app:1`

### 18. The Solution

Now, if we were to turn `com.mycompany.app:my-app:1` into a parent artifact of `com.mycompany.app:my-module:1`, we will have to modify `com.mycompany.app:my-module:1`’s POM to the following configuration:

**com.mycompany.app:my-module:1’s POM**

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
  </parent>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```

Notice that we now have an added section, the parent section. This section allows us to specify which artifact is the parent of our POM. And we do so by specifying the fully qualified artifact name of the parent POM. With this setup, our module can now inherit some of the properties of our parent POM.

Alternatively, if you want the groupId or the version of your modules to be the same as their parents, you can remove the groupId or the version identity of your module in its POM.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
  </parent>
  <artifactId>my-module</artifactId>
</project>
```

This allows the module to inherit the groupId or the version of its parent POM.

### 18.1.4.2 Example 2

#### 18. The Scenario

However, that would work if the parent project was already installed in our local repository or was in that specific directory structure (parent `pom.xml` is one directory higher than that of the module’s `pom.xml`).
But what if the parent is not yet installed and if the directory structure is as in the following example?

```
|-- my-module
|  |-- pom.xml
|-- parent
    |-- pom.xml
```

18. The Solution

To address this directory structure (or any other directory structure), we would have to add the `<relativePath>` element to our parent section.

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.mycompany.app</groupId>
        <artifactId>my-app</artifactId>
        <version>1</version>
        <relativePath>../parent/pom.xml</relativePath>
    </parent>
    <artifactId>my-module</artifactId>
</project>
```

As the name suggests, it's the relative path from the module's pom.xml to the parent's pom.xml.

18.1.5 Project Aggregation

Project Aggregation is similar to Project Inheritance. But instead of specifying the parent POM from the module, it specifies the modules from the parent POM. By doing so, the parent project now knows its modules, and if a Maven command is invoked against the parent project, that Maven command will then be executed to the parent's modules as well. To do Project Aggregation, you must do the following:

- Change the parent POMs packaging to the value "pom".
- Specify in the parent POM the directories of its modules (children POMs).

18.1.5.1 Example 3

18. The Scenario

Given the previous original artifact POMs and directory structure:

**com.mycompany.app:my-app:1's POM**

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
</project>
```

**com.mycompany.app:my-module:1's POM**

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
</project>
```
18. The POM

```xml
<groupId>com.mycompany.app</groupId>
<artifactId>my-module</artifactId>
<version>1</version>
</project>
```

directory structure

```
|-- my-module
 | `-- pom.xml
`-- pom.xml
```

18. The Solution

If we are to aggregate my-module into my-app, we would only have to modify my-app.

```xml
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <version>1</version>
 <packaging>pom</packaging>
 <modules>
   <module>my-module</module>
 </modules>
</project>
```

In the revised com.mycompany.app:my-app:1, the packaging section and the modules sections were added. For the packaging, its value was set to "pom", and for the modules section, we have the element `<module>my-module</module>`. The value of `<module>` is the relative path from the com.mycompany.app:my-app:1 to com.mycompany.app:my-module:1’s POM (by practice, we use the module's artifactId as the module directory's name).

Now, whenever a Maven command processes com.mycompany.app:my-app:1, that same Maven command would be ran against com.mycompany.app:my-module:1 as well. Furthermore, some commands (goals specifically) handle project aggregation differently.

[top]

18.1.5.2 Example 4

18. The Scenario

But what if we change the directory structure to the following:

```
|-- my-module
 | `-- pom.xml
|-- parent
 | `-- pom.xml
```

How would the parent POM specify its modules?

18. The Solution

The answer? - the same way as Example 3, by specifying the path to the module.

```xml
<project>
 <modelVersion>4.0.0</modelVersion>
</project>
```
18.1.6 Project Inheritance vs Project Aggregation

If you have several Maven projects, and they all have similar configurations, you can refactor your projects by pulling out those similar configurations and making a parent project. Thus, all you have to do is to let your Maven projects inherit that parent project, and those configurations would then be applied to all of them.

And if you have a group of projects that are built or processed together, you can create a parent project and have that parent project declare those projects as its modules. By doing so, you’d only have to build the parent and the rest will follow.

But of course, you can have both Project Inheritance and Project Aggregation. Meaning, you can have your modules specify a parent project, and at the same time, have that parent project specify those Maven projects as its modules. You’d just have to apply all three rules:

- Specify in every child POM who their parent POM is.
- Change the parent POMs packaging to the value "pom".
- Specify in the parent POM the directories of its modules (children POMs)

18.1.6.1 Example 5

18. The Scenario

Given the previous original artifact POMs again,

**com.mycompany.app:my-app:1's POM**

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

**com.mycompany.app:my-module:1's POM**

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```

and this **directory structure**

```
.
|-- my-module
```
18. The Solution
To do both project inheritance and aggregation, you only have to apply all three rules.

**com.mycompany.app:my-app:1's POM**

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
  <packaging>pom</packaging>
  <modules>
    <module>../my-module</module>
  </modules>
</project>
```

**com.mycompany.app:my-module:1's POM**

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
    <relativePath>../parent/pom.xml</relativePath>
  </parent>
  <artifactId>my-module</artifactId>
</project>
```

**NOTE:** Profile inheritance the same inheritance strategy as used for the POM itself.

18.1.7 Project Interpolation and Variables

One of the practices that Maven encourages is *don't repeat yourself*. However, there are circumstances where you will need to use the same value in several different locations. To assist in ensuring the value is only specified once, Maven allows you to use both your own and pre-defined variables in the POM.

For example, to access the `project.version` variable, you would reference it like so:

```xml
<version>${project.version}</version>
```

One factor to note is that these variables are processed *after* inheritance as outlined above. This means that if a parent project uses a variable, then its definition in the child, not the parent, will be the one eventually used.
18.1.7.1 Available Variables

18. Project Model Variables

Any field of the model that is a single value element can be referenced as a variable. For example, 
\${project.groupId}, \${project.version}, \${project.build.sourceDirectory} and so on. Refer to the POM reference to see a full list of properties.

These variables are all referenced by the prefix "project.". You may also see references with pom. as the prefix, or the prefix omitted entirely - these forms are now deprecated and should not be used.

18. Special Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>project.basedir</td>
<td>The directory that the current project resides in.</td>
</tr>
<tr>
<td>project.baseUri</td>
<td>The directory that the current project resides in, represented as an URI. Since Maven 2.1.0</td>
</tr>
<tr>
<td>maven.build.timestamp</td>
<td>The timestamp that denotes the start of the build (UTC). Since Maven 2.1.0-M1</td>
</tr>
</tbody>
</table>

The format of the build timestamp can be customized by declaring the property maven.build.timestamp.format as shown in the example below:

```xml
<project>
  ...
  <properties>
    <maven.build.timestamp.format>yyyy-MM-dd'T'HH:mm:ss'Z'</maven.build.timestamp.format>
  </properties>
  ...
</project>
```

The format pattern has to comply with the rules given in the API documentation for SimpleDateFormat. If the property is not present, the format defaults to the value already given in the example.

18. Properties

You are also able to reference any properties defined in the project as a variable. Consider the following example:

```xml
<project>
  ...
  <properties>
    <mavenVersion>3.0</mavenVersion>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-artifact</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-core</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
  </dependencies>
</project>
```
</dependencies>
...
</project>
19 Profiles

19.1 Introduction to Build Profiles

Apache Maven goes to great lengths to ensure that builds are portable. Among other things, this means allowing build configuration inside the POM, avoiding all filesystem references (in inheritance, dependencies, and other places), and leaning much more heavily on the local repository to store the metadata needed to make this possible.

However, sometimes portability is not entirely possible. Under certain conditions, plugins may need to be configured with local filesystem paths. Under other circumstances, a slightly different dependency set will be required, and the project's artifact name may need to be adjusted slightly. And at still other times, you may even need to include a whole plugin in the build lifecycle depending on the detected build environment.

To address these circumstances, Maven supports build profiles. Profiles are specified using a subset of the elements available in the POM itself (plus one extra section), and are triggered in any of a variety of ways. They modify the POM at build time, and are meant to be used in complementary sets to give equivalent-but-different parameters for a set of target environments (providing, for example, the path of the appserver root in the development, testing, and production environments). As such, profiles can easily lead to differing build results from different members of your team. However, used properly, profiles can be used while still preserving project portability. This will also minimize the use of –f option of maven which allows user to create another POM with different parameters or configuration to build which makes it more maintainable since it is running with one POM only.

19.1.1 What are the different types of profile? Where is each defined?

- Per Project
  - Defined in the POM itself (pom.xml).
- Per User
  - Defined in the Maven-settings (%USER_HOME%/.m2/settings.xml).
- Global
  - Defined in the global Maven-settings (${maven.home}/conf/settings.xml).
- Profile descriptor
  - A descriptor located in project basedir (profiles.xml) (no longer supported in Maven 3.0 and above; see Maven 3 compatibility notes)

19.1.2 How can a profile be triggered? How does this vary according to the type of profile being used?

A profile can be activated in several ways:

- From the command line
- Through Maven settings
- Based on environment variables
- OS settings
- Present or missing files

19.1.2.1 Details on profile activation

Profiles can be explicitly specified using the –P command line flag.
This flag is followed by a comma-delimited list of profile IDs to use. The profile(s) specified in the option are activated in addition to any profiles which are activated by their activation configuration or the `<activeProfiles>` section in settings.xml. From Maven 4 onward, Maven will refuse to activate or deactivate a profile that cannot be resolved. To prevent this, prefix the profile identifier with an `?`, marking it as optional:

```
mvn groupId:artifactId:goal -P profile-1,profile-2,?profile-3
```

Profiles can be activated in the Maven settings, via the `<activeProfiles>` section. This section takes a list of `<activeProfile>` elements, each containing a profile-id inside.

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>profile-1</activeProfile>
  </activeProfiles>
  ...
</settings>
```

Profiles listed in the `<activeProfiles>` tag would be activated by default every time a project use it.

Profiles can be automatically triggered based on the detected state of the build environment. These triggers are specified via an `<activation>` section in the profile itself. Currently, this detection is limited to prefix-matching of the JDK version, the presence of a system property or the value of a system property. Here are some examples.

The following configuration will trigger the profile when the JDK's version starts with "1.4" (eg. "1.4.0_08", "1.4.2_07", "1.4"):

```
<profiles>
  <profile>
    <activation>
      <jdk>1.4</jdk>
    </activation>
    ...
  </profile>
</profiles>
```

Ranges can also be used as of Maven 2.1 (refer to the Enforcer Version Range Syntax for more information). The following honours versions 1.3, 1.4 and 1.5.

```
<profiles>
  <profile>
    <activation>
      <jdk>[1.3,1.6)</jdk>
    </activation>
    ...
  </profile>
</profiles>
```

Note: an upper bound such as `1.5]` is likely not to include most releases of 1.5, since they will have an additional "patch" release such as _05 that is not taken into consideration in the above range.

This next one will activate based on OS settings. See the Maven Enforcer Plugin for more details about OS values.

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
The profile below will be activated when the system property "debug" is specified with any value:

```xml
<profiles>
  <profile>
    <activation>
      <property>
        <name>debug</name>
      </property>
    </activation>
  </profile>
</profiles>
```

The following profile will be activated when the system property "debug" is not defined at all:

```xml
<profiles>
  <profile>
    <activation>
      <property>
        <name>!debug</name>
      </property>
    </activation>
  </profile>
</profiles>
```

The following profile will be activated when the system property "debug" is not defined, or is defined with a value which is not "true".

```xml
<profiles>
  <profile>
    <activation>
      <property>
        <name>debug</name>
        <value>!true</value>
      </property>
    </activation>
  </profile>
</profiles>
```
To activate this you would type one of those on the command line:

```bash
mvn groupId:artifactId:goal
mvn groupId:artifactId:goal -Ddebug=false
```

The next example will trigger the profile when the system property "environment" is specified with the value "test":

```xml
<profiles>
  <profile>
    <activation>
      <property>
        <name>environment</name>
        <value>test</value>
      </property>
    </activation>
    ...
  </profile>
</profiles>
```

To activate this you would type this on the command line:

```bash
mvn groupId:artifactId:goal -Denvironment=test
```

As of Maven 3.0, profiles in the POM can also be activated based on properties from active profiles from the settings.xml.

**Note:** Environment variables like `FOO` are available as properties of the form `env.FOO`. Further note that environment variable names are normalized to all upper-case on Windows.

This example will trigger the profile when the generated file `target/generated-sources/axistools/wsdl2java/org/apache/maven` is missing.

```xml
<profiles>
  <profile>
    <activation>
      <file>
        <missing>target/generated-sources/axistools/wsdl2java/org/apache/maven</missing>
      </file>
    </activation>
    ...
  </profile>
</profiles>
```

As of Maven 2.0.9, the tags `<exists>` and `<missing>` could be interpolated. Supported variables are system properties like `${user.home}` and environment variables like `${env.HOME}`. Please note that properties and values defined in the POM itself are not available for interpolation here, e.g. the above example activator cannot use `${project.build.directory}` but needs to hard-code the path `target`.

Profiles can also be active by default using a configuration like the following:

```xml
<profiles>
  <profile>
    <id>profile-1</id>
    <activation>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
19 Profiles

This profile will automatically be active for all builds unless another profile in the same POM is activated using one of the previously described methods. All profiles that are active by default are automatically deactivated when a profile in the POM is activated on the command line or through its activation config.

19.1.2.2 Deactivating a profile
Starting with Maven 2.0.10, one or more profiles can be deactivated using the command line by prefixing their identifier with either the character ‘!’ or ‘-’ as shown below:

```shell
mvn groupId:artifactId:goal -P !profile-1,!profile-2,!?profile-3
```
or
```shell
mvn groupId:artifactId:goal -P -profile-1,-profile-2,-?profile-3
```
This can be used to deactivate profiles marked as activeByDefault or profiles that would otherwise be activated through their activation config.

19.1.3 Which areas of a POM can be customized by each type of profile? Why?
Now that we've talked about where to specify profiles, and how to activate them, it will be useful to talk about what you can specify in a profile. As with the other aspects of profile configuration, this answer is not straightforward.

Depending on where you choose to configure your profile, you will have access to varying POM configuration options.

19.1.3.1 Profiles in external files
Profiles specified in external files (i.e in `settings.xml` or `profiles.xml`) are not portable in the strictest sense. Anything that seems to stand a high chance of changing the result of the build is restricted to the inline profiles in the POM. Things like repository lists could simply be a proprietary repository of approved artifacts, and won't change the outcome of the build. Therefore, you will only be able to modify the `<repositories>` and `<pluginRepositories>` sections, plus an extra `<properties>` section.

The `<properties>` section allows you to specify free-form key-value pairs which will be included in the interpolation process for the POM. This allows you to specify a plugin configuration in the form of `profile.provided.path`.

19.1.3.2 Profiles in POMs
On the other hand, if your profiles can be reasonably specified inside the POM, you have many more options. The trade-off, of course, is that you can only modify that project and it's sub-modules. Since these profiles are specified inline, and therefore have a better chance of preserving portability, it's reasonable to say you can add more information to them without the risk of that information being unavailable to other users.

Profiles specified in the POM can modify the following POM elements:
- `<repositories>`
- `<pluginRepositories>`
- `<dependencies>`
• <plugins>
• <properties> (not actually available in the main POM, but used behind the scenes)
• <modules>
• <reports>
• <reporting>
• <dependencyManagement>
• <distributionManagement>
• a subset of the <build> element, which consists of:
  • <defaultGoal>
  • <resources>
  • <testResources>
  • <directory>
  • <finalName>
  • <filters>
  • <pluginManagement>
  • <plugins>

19.1.3.3 POM elements outside <profiles>
We don't allow modification of some POM elements outside of POM-profiles because these runtime modifications will not be distributed when the POM is deployed to the repository system, making that person's build of that project completely unique from others. While you can do this to some extent with the options given for external profiles, the danger is limited. Another reason is that this POM info is sometimes being reused from the parent POM.

External files such as settings.xml and profiles.xml also does not support elements outside the POM-profiles. Let us take this scenario for elaboration. When the effective POM get deployed to a remote repository, any person can pickup its info out of the repository and use it to build a Maven project directly. Now, imagine that if we can set profiles in dependencies, which is very important to a build, or in any other elements outside POM-profiles in settings.xml then most probably we cannot expect someone else to use that POM from the repository and be able to build it. And we have to also think about how to share the settings.xml with others. Note that too many files to configure is very confusing and very hard to maintain. Bottom line is that since this is build data, it should be in the POM. One of the goals in Maven 2 is to consolidate all the information needed to run a build into a single file, or file hierarchy which is the POM.

19.1.4 Profile Order
All profile elements in a POM from active profiles overwrite the global elements with the same name of the POM or extend those in case of collections. In case multiple profiles are active in the same POM or external file, the ones which are defined later take precedence over the ones defined earlier (independent of their profile id and activation order).

Example:

```
<project>
  ...
  <repositories>
    <repository>
      <id>global-repo</id>
    </repository>
  ...
```
This leads to the repository list: profile-2-repo, profile-1-repo, global-repo.

19.1.5 Profile Pitfalls

We’ve already mentioned the fact that adding profiles to your build has the potential to break portability for your project. We’ve even gone so far as to highlight circumstances where profiles are likely to break project portability. However, it’s worth reiterating those points as part of a more coherent discussion about some pitfalls to avoid when using profiles.

There are two main problem areas to keep in mind when using profiles. First are external properties, usually used in plugin configurations. These pose the risk of breaking portability in your project. The other, more subtle area is the incomplete specification of a natural set of profiles.

19.1.5.1 External Properties

External property definition concerns any property value defined outside the `pom.xml` but not defined in a corresponding profile inside it. The most obvious usage of properties in the POM is in plugin configuration. While it is certainly possible to break project portability without properties, these criterias can have subtle effects that cause builds to fail. For example, specifying appserver paths in a profile that is specified in the `settings.xml` may cause your integration test plugin to fail when
another user on the team attempts to build without a similar settings.xml. Consider the following pom.xml snippet for a web application project:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.myco.plugins</groupId>
        <artifactId>spiffy-integrationTest-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          <appserverHome>${appserver.home}</appserverHome>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

Now, in your local ${user.home}/.m2/settings.xml, you have:

```
<settings>
  ...
  <profiles>
    <profile>
      <id>appserverConfig</id>
      <properties>
        <appserver.home>/path/to/appserver</appserver.home>
      </properties>
    </profile>
  </profiles>
  <activeProfiles>
    <activeProfile>appserverConfig</activeProfile>
  </activeProfiles>
  ...
</settings>
```

When you build the integration-test lifecycle phase, your integration tests pass, since the path you’ve provided allows the test plugin to install and test this web application.

However, when your colleague attempts to build to integration-test, his build fails spectacularly, complaining that it cannot resolve the plugin configuration parameter <appserverHome>, or worse, that the value of that parameter - literally ${appserver.home} - is invalid (if it warns you at all).

Congratulations, your project is now non-portable. Inlining this profile in your pom.xml can help alleviate this, with the obvious drawback that each project hierarchy (allowing for the effects of inheritance) now have to specify this information. Since Maven provides good support for project inheritance, it's possible to stick this sort of configuration in the <pluginManagement> section of a team-level POM or similar, and simply inherit the paths.

Another, less attractive answer might be standardization of development environments. However, this will tend to compromise the productivity gain that Maven is capable of providing.
19.1.5.2 Incomplete Specification of a Natural Profile Set

In addition to the above portability-breaker, it’s easy to fail to cover all cases with your profiles. When you do this, you're usually leaving one of your target environments high and dry. Let's take the example pom.xml snippet from above one more time:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.myco.plugins</groupId>
        <artifactId>spiffy-integrationTest-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          <appserverHome>${appserver.home}</appserverHome>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

Now, consider the following profile, which would be specified inline in the pom.xml:

```xml
<project>
  ...
  <profiles>
    <profile>
      <id>appserverConfig-dev</id>
      <activation>
        <property>
          <name>env</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <appserver.home>/path/to/dev/appserver</appserver.home>
      </properties>
    </profile>
    <profile>
      <id>appserverConfig-dev-2</id>
      <activation>
        <property>
          <name>env</name>
          <value>dev-2</value>
        </property>
      </activation>
      <properties>
        <appserver.home>/path/to/another/dev/appserver2</appserver.home>
      </properties>
    </profile>
  </profiles>
  ...
</project>
```
This profile looks quite similar to the one from the last example, with a few important exceptions: it's plainly geared toward a development environment, a new profile named `appserverConfig-dev-2` is added and it has an activation section that will trigger its inclusion when the system properties contain "env=dev" for a profile named `appserverConfig-dev` and "env=dev-2" for a profile named `appserverConfig-dev-2`. So, executing:

```
mvn -Denv=dev-2 integration-test
```

will result in a successful build, applying the properties given by profile named `appserverConfig-dev-2`. And when we execute

```
mvn -Denv=dev integration-test
```

it will result in a successful build applying the properties given by the profile named `appserverConfig-dev`. However, executing:

```
mvn -Denv=production integration-test
```

will not do a successful build. Why? Because, the resulting non-interpolated literal value of `${appserver.home}` will not be a valid path for deploying and testing your web application. We haven't considered the case for the production environment when writing our profiles. The "production" environment (env=production), along with "test" and possibly even "local" constitute a natural set of target environments for which we may want to build the integration-test lifecycle phase. The incomplete specification of this natural set means we have effectively limited our valid target environments to the development environment. Your teammates - and probably your manager - will not see the humor in this. When you construct profiles to handle cases such as these, be sure to address the entire set of target permutations.

As a quick aside, it's possible for user-specific profiles to act in a similar way. This means that profiles for handling different environments which are keyed to the user can act up when the team adds a new developer. While I suppose this could act as useful training for the newbie, it just wouldn't be nice to throw them to the wolves in this way. Again, be sure to think of the whole set of profiles.

### 19.1.6 How can I tell which profiles are in effect during a build?

Determining active profiles will help the user to know what particular profiles has been executed during a build. We can use the [Maven Help Plugin](https://maven.apache.org/plugins/) to tell what profiles are in effect during a build.

```
mvn help:active-profiles
```

Let us have some small samples that will help us to understand more on the `active-profiles` goal of that plugin.

From the last example of profiles in the `pom.xml`, you'll notice that there are two profiles named `appserverConfig-dev` and `appserverConfig-dev-2` which has been given different values for properties. If we go ahead and execute:

```
mvn help:active-profiles -Denv=dev
```

The result will be a bulleted list of the id of the profile with an activation property of "env=dev" together with the source where it was declared. See sample below.

The following profiles are active:
- `appserverConfig-dev` (source: pom)
Now if we have a profile declared in `settings.xml` (refer to the sample of profile in `settings.xml`) and that have been set to be an active profile and execute:

```
mvn help:active-profiles
```

The result should be something like this

The following profiles are active:
- appserverConfig (source: settings.xml)

Even though we don't have an activation property, a profile has been listed as active. Why? Like we mentioned before, a profile that has been set as an active profile in the `settings.xml` is automatically activated.

Now if we have something like a profile in the `settings.xml` that has been set as an active profile and also triggered a profile in the POM. Which profile do you think will have an effect on the build?

```
mvn help:active-profiles -P appserverConfig-dev
```

This will list the activated profiles:

The following profiles are active:
- appserverConfig-dev (source: pom)
- appserverConfig (source: settings.xml)

Even though it listed the two active profiles, we are not sure which one of them has been applied. To see the effect on the build execute:

```
mvn help:effective-pom -P appserverConfig-dev
```

This will print the effective POM for this build configuration out to the console. Take note that profiles in the `settings.xml` takes higher priority than profiles in the POM. So the profile that has been applied here is `appserverConfig` not `appserverConfig-dev`.

If you want to redirect the output from the plugin to a file called `effective-pom.xml`, use the command-line option `-Doutput=effective-pom.xml`.

### 19.1.7 Naming Conventions

By now you've noticed that profiles are a natural way of addressing the problem of different build configuration requirements for different target environments. Above, we discussed the concept of a "natural set" of profiles to address this situation, and the importance of considering the whole set of profiles that will be required.

However, the question of how to organize and manage the evolution of that set is non-trivial as well. Just as a good developer strives to write self-documenting code, it's important that your profile id's give a hint to their intended use. One good way to do this is to use the common system property trigger as part of the name for the profile. This might result in names like `env-dev`, `env-test`, and `env-prod` for profiles that are triggered by the system property `env`. Such a system leaves a highly intuitive hint on how to activate a build targeted at a particular environment. Thus, to activate a build for the test environment, you need to activate `env-test` by issuing:

```
mvn -Denv=test <phase>
```

The right command-line option can be had by simply substituting "=" for ":" in the profile id.
20 Repositories

20.1 Introduction to Repositories

20.1.1 Artifact Repositories

A repository in Maven holds build artifacts and dependencies of varying types.

There are exactly two types of repositories: local and remote:

1. the **local** repository is a directory on the computer where Maven runs. It caches remote downloads and contains temporary build artifacts that you have not yet released.

2. **remote** repositories refer to any other type of repository, accessed by a variety of protocols such as file:// and https://. These repositories might be a truly remote repository set up by a third party to provide their artifacts for downloading (for example, repo.maven.apache.org). Other "remote" repositories may be internal repositories set up on a file or HTTP server within your company, used to share private artifacts between development teams and for releases.

Local and remote repositories are structured the same way so that scripts can run on either side, or they can be synced for offline use. The layout of the repositories is completely transparent to the Maven user, however.

20.1.2 Using Repositories

In general, you should not need to do anything with the local repository on a regular basis, except clean it out if you are short on disk space (or erase it completely if you are willing to download everything again).

For the remote repositories, they are used for both downloading and uploading (if you have the permission to do so).

20.1.2.1 Downloading from a Remote Repository

Downloading in Maven is triggered by a project declaring a dependency that is not present in the local repository (or for a SNAPSHOT, when the remote repository contains one that is newer). By default, Maven will download from the central repository.

To override this, you need to specify a mirror as shown in Using Mirrors for Repositories.

You can set this in your settings.xml file to globally use a certain mirror. However, it is common for a project to customise the repository in its pom.xml and that your setting will take precedence. If dependencies are not being found, check that you have not overridden the remote repository.

For more information on dependencies, see Dependency Mechanism.

20.1.2.2 Using Mirrors for the Central Repository

There are several official Central repositories geographically distributed. You can make changes to your settings.xml file to use one or more mirrors. Instructions for this can be found in the guide Using Mirrors for Repositories.

20.1.3 Building Offline

If you are temporarily disconnected from the internet and you need to build your projects offline, you can use the offline switch on the CLI:

```
mvn -o package
```
Many plugins honor the offline setting and do not perform any operations that connect to the internet. Some examples are resolving Javadoc links and link checking the site.

20.1.4 Uploading to a Remote Repository

While this is possible for any type of remote repository, you must have the permission to do so. To have someone upload to the Central Maven repository, see Repository Center.

20.1.5 Internal Repositories

When using Maven, particularly in a corporate environment, connecting to the internet to download dependencies is not acceptable for security, speed or bandwidth reasons. For that reason, it is desirable to set up an internal repository to house a copy of artifacts, and to publish private artifacts to.

Such an internal repository can be downloaded using HTTP or the file system (with a file:// URL), and uploaded to using SCP, FTP, or a file copy.

As far as Maven is concerned, there is nothing special about this repository: it is another remote repository that contains artifacts to download to a user's local cache, and is a publish destination for artifact releases.

Additionally, you may want to share the repository server with your generated project sites. For more information on creating and deploying sites, see Creating a Site.

20.1.6 Setting up the Internal Repository

To set up an internal repository just requires that you have a place to put it, and then copy required artifacts there using the same layout as in a remote repository such as repo.maven.apache.org.

It is not recommended that you scrape or rsync:// a full copy of central as there is a large amount of data there and doing so will get you banned. You can use a program such as those described on the Repository Management page to run your internal repository's server, download from the internet as required, and then hold the artifacts in your internal repository for faster downloading later.

The other options available are to manually download and vet releases, then copy them to the internal repository, or to have Maven download them for a user, and manually upload the vetted artifacts to the internal repository which is used for releases. This step is the only one available for artifacts where the license forbids their distribution automatically, such as several J2EE JARs provided by Sun. Refer to the Guide to coping with SUN JARs document for more information.

It should be noted that Maven intends to include enhanced support for such features in the future, including click through licenses on downloading, and verification of signatures.

20.1.7 Using the Internal Repository

Using the internal repository is quite simple. Simply make a change to add a repositories element:

```xml
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>https://myserver/repo</url>
    </repository>
  </repositories>
</project>
```
If your internal repository requires authentication, the `id` element can be used in your `settings` file to specify login information.

### 20.1.8 Deploying to the Internal Repository

One of the most important reasons to have one or more internal repositories is to be able to publish your own private releases.

To publish to the repository, you will need to have access via one of SCP, SFTP, FTP, WebDAV, or the filesystem. Connectivity is accomplished with the various wagons. Some wagons may need to be added as extension to your build.
21 Standard Directory Layout

21.1 Introduction to the Standard Directory Layout

Having a common directory layout allows users familiar with one Maven project to immediately feel at home in another Maven project. The advantages are analogous to adopting a site-wide look-and-feel.

The next section documents the directory layout expected by Maven and the directory layout created by Maven. Try to conform to this structure as much as possible. However, if you can’t, these settings can be overridden via the project descriptor.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/main/java</td>
<td>Application/Library sources</td>
</tr>
<tr>
<td>src/main/resources</td>
<td>Application/Library resources</td>
</tr>
<tr>
<td>src/main/filters</td>
<td>Resource filter files</td>
</tr>
<tr>
<td>src/main/webapp</td>
<td>Web application sources</td>
</tr>
<tr>
<td>src/test/java</td>
<td>Test sources</td>
</tr>
<tr>
<td>src/test/resources</td>
<td>Test resources</td>
</tr>
<tr>
<td>src/test/filters</td>
<td>Test resource filter files</td>
</tr>
<tr>
<td>src/it</td>
<td>Integration Tests (primarily for plugins)</td>
</tr>
<tr>
<td>src/assembly</td>
<td>Assembly descriptors</td>
</tr>
<tr>
<td>src/site</td>
<td>Site</td>
</tr>
<tr>
<td>LICENSE.txt</td>
<td>Project's license</td>
</tr>
<tr>
<td>NOTICE.txt</td>
<td>Notices and attributions required by libraries that the project depends on</td>
</tr>
<tr>
<td>README.txt</td>
<td>Project's readme</td>
</tr>
</tbody>
</table>

At the top level, files descriptive of the project: a pom.xml file. In addition, there are textual documents meant for the user to be able to read immediately on receiving the source: README.txt, LICENSE.txt, etc.

There are just two subdirectories of this structure: src and target. The only other directories that would be expected here are metadata like CVS, .git or .svn, and any subprojects in a multiproject build (each of which would be laid out as above).

The target directory is used to house all output of the build.

The src directory contains all of the source material for building the project, its site and so on. It contains a subdirectory for each type: main for the main build artifact, test for the unit test code and resources, site and so on.

Within artifact producing source directories (ie. main and test), there is one directory for the language java (under which the normal package hierarchy exists), and one for resources (the structure which is copied to the target classpath given the default resource definition).

If there are other contributing sources to the artifact build, they would be under other subdirectories. For example src/main/antlr would contain Antlr grammar definition files.
22 The Dependency Mechanism

22.1 Introduction to the Dependency Mechanism

Dependency management is a core feature of Maven. Managing dependencies for a single project is easy. Managing dependencies for multi-module projects and applications that consist of hundreds of modules is possible. Maven helps a great deal in defining, creating, and maintaining reproducible builds with well-defined classpaths and library versions.

Learn more about:
- Transitive Dependencies
- Excluded/Optional Dependencies
- Dependency Scope
- Dependency Management
- Importing Dependencies
- Bill of Materials (BOM) POMs
- System Dependencies

22.1.1 Transitive Dependencies

Maven avoids the need to discover and specify the libraries that your own dependencies require by including transitive dependencies automatically.

This feature is facilitated by reading the project files of your dependencies from the remote repositories specified. In general, all dependencies of those projects are used in your project, as are any that the project inherits from its parents, or from its dependencies, and so on.

There is no limit to the number of levels that dependencies can be gathered from. A problem arises only if a cyclic dependency is discovered.

With transitive dependencies, the graph of included libraries can quickly grow quite large. For this reason, there are additional features that limit which dependencies are included:

- **Dependency mediation** - this determines what version of an artifact will be chosen when multiple versions are encountered as dependencies. Maven picks the "nearest definition". That is, it uses the version of the closest dependency to your project in the tree of dependencies. You can always guarantee a version by declaring it explicitly in your project’s POM. Note that if two dependency versions are at the same depth in the dependency tree, the first declaration wins.

  - "nearest definition" means that the version used will be the closest one to your project in the tree of dependencies. Consider this tree of dependencies:

```
A
### B
#   ### C
#       ### D 2.0
### E
### E 1.0
```

In text, dependencies for A, B, and C are defined as A -> B -> C -> D 2.0 and A -> E -> D 1.0, then D 1.0 will be used when building A because the path from A to D through E is shorter. You could explicitly add a dependency to D 2.0 in A to force the use of D 2.0, as shown here:
A
### B
#
### C
#
### D 2.0
### E
#
### D 1.0
#
### D 2.0

- **Dependency management** - this allows project authors to directly specify the versions of artifacts to be used when they are encountered in transitive dependencies or in dependencies where no version has been specified. In the example in the preceding section a dependency was directly added to A even though it is not directly used by A. Instead, A can include D as a dependency in its dependencyManagement section and directly control which version of D is used when, or if, it is ever referenced.

- **Dependency scope** - this allows you to only include dependencies appropriate for the current stage of the build. This is described in more detail below.

- **Excluded dependencies** - If project X depends on project Y, and project Y depends on project Z, the owner of project X can explicitly exclude project Z as a dependency, using the "exclusion" element.

- **Optional dependencies** - If project Y depends on project Z, the owner of project Y can mark project Z as an optional dependency, using the "optional" element. When project X depends on project Y, X will depend only on Y and not on Y's optional dependency Z. The owner of project X may then explicitly add a dependency on Z, at her option. (It may be helpful to think of optional dependencies as "excluded by default.")

Although transitive dependencies can implicitly include desired dependencies, it is a good practice to explicitly specify the dependencies your source code uses directly. This best practice proves its value especially when the dependencies of your project change their dependencies.

For example, assume that your project A specifies a dependency on another project B, and project B specifies a dependency on project C. If you are directly using components in project C, and you don't specify project C in your project A, it may cause build failure when project B suddenly updates/ removes its dependency on project C.

Another reason to directly specify dependencies is that it provides better documentation for your project: one can learn more information by just reading the POM file in your project, or by executing `mvn dependency:tree`.

Maven also provides `dependency:analyze` plugin goal for analyzing the dependencies: it helps making this best practice more achievable.

### 22.1.2 Dependency Scope

Dependency scope is used to limit the transitivity of a dependency and to determine when a dependency is included in a classpath.

There are 6 scopes:

- **compile**
  This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.

- **provided**
This is much like `compile`, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope `provided` because the web container provides those classes. A dependency with this scope is added to the classpath used for compilation and test, but not the runtime classpath. It is not transitive.

- **runtime**
  This scope indicates that the dependency is not required for compilation, but is for execution. Maven includes a dependency with this scope in the runtime and test classpaths, but not the compile classpath.

- **test**
  This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive. Typically this scope is used for test libraries such as JUnit and Mockito. It is also used for non-test libraries such as Apache Commons IO if those libraries are used in unit tests (src/test/java) but not in the model code (src/main/java).

- **system**
  This scope is similar to `provided` except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

- **import**
  This scope is only supported on a dependency of type `pom` in the `<dependencyManagement>` section. It indicates the dependency is to be replaced with the effective list of dependencies in the specified POM’s `<dependencyManagement>` section. Since they are replaced, dependencies with a scope of `import` do not actually participate in limiting the transitivity of a dependency.

Each of the scopes (except for `import`) affects transitive dependencies in different ways, as is demonstrated in the table below. If a dependency is set to the scope in the left column, a transitive dependency of that dependency with the scope across the top row results in a dependency in the main project with the scope listed at the intersection. If no scope is listed, it means the dependency is omitted.

<table>
<thead>
<tr>
<th></th>
<th>compile</th>
<th>provided</th>
<th>runtime</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile</td>
<td><code>compile(*)</code></td>
<td>-</td>
<td>runtime</td>
<td>-</td>
</tr>
<tr>
<td>provided</td>
<td><code>provided</code></td>
<td>-</td>
<td><code>provided</code></td>
<td>-</td>
</tr>
<tr>
<td>runtime</td>
<td>runtime</td>
<td>-</td>
<td>runtime</td>
<td>-</td>
</tr>
<tr>
<td>test</td>
<td>test</td>
<td>-</td>
<td>test</td>
<td>-</td>
</tr>
</tbody>
</table>

(*) **Note**: it is intended that this should be runtime scope instead, so that all compile dependencies must be explicitly listed. However, if a library you depend on extends a class from another library, both must be available at compile time. For this reason, compile time dependencies remain as compile scope even when they are transitive.

### 22.1.3 Dependency Management

The dependency management section is a mechanism for centralizing dependency information. When you have a set of projects that inherit from a common parent, it’s possible to put all information about the dependency in the common POM and have simpler references to the artifacts in the child POMs. The mechanism is best illustrated through some examples. Given these two POMs which extend the same parent:

Project A:
These two example POMs share a common dependency and each has one non-trivial dependency. This information can be put in the parent POM like this:

```xml
<project>

    ...

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>group-a</groupId>
                <artifactId>artifact-b</artifactId>
                <version>1.0</version>
                <type>bar</type>
                <scope>runtime</scope>
            </dependency>

            <dependency>
                <groupId>group-c</groupId>
                <artifactId>artifact-b</artifactId>
                <version>1.0</version>
                <type>war</type>
                <scope>runtime</scope>
            </dependency>

        </dependencies>
    </dependencyManagement>

</project>
```
Then the two child POMs become much simpler:

```xml
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
    </dependency>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <!-- This is not a jar dependency, so we must specify type. -->
      <type>bar</type>
    </dependency>
  </dependencies>
</project>

<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-c</groupId>
      <artifactId>artifact-b</artifactId>
      <!-- This is not a jar dependency, so we must specify type. -->
      <type>war</type>
    </dependency>
  </dependencies>
</project>
```
NOTE: In two of these dependency references, we had to specify the <type/> element. This is because the minimal set of information for matching a dependency reference against a dependencyManagement section is actually {groupId, artifactId, type, classifier}. In many cases, these dependencies will refer to jar artifacts with no classifier. This allows us to shorthand the identity set to {groupId, artifactId}, since the default for the type field is jar, and the default classifier is null.

A second, and very important use of the dependency management section is to control the versions of artifacts used in transitive dependencies. As an example consider these projects:

Project A:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>A</artifactId>
  <packaging>pom</packaging>
  <name>A</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>a</artifactId>
        <version>1.2</version>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>b</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>c</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>d</artifactId>
        <version>1.2</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```
When Maven is run on project B, version 1.0 of artifacts a, b, c, and d will be used regardless of the version specified in their POM.

- a and c both are declared as dependencies of the project so version 1.0 is used due to dependency mediation. Both also have runtime scope since it is directly specified.
- b is defined in B's parent's dependency management section and since dependency management takes precedence over dependency mediation for transitive dependencies, version 1.0 will be selected should it be referenced in a or c's POM. b will also have compile scope.
- Finally, since d is specified in B's dependency management section, should d be a dependency (or transitive dependency) of a or c, version 1.0 will be chosen - again because dependency management takes precedence over dependency mediation and also because the current POM's declaration takes precedence over its parent's declaration.
The reference information about the dependency management tags is available from the project descriptor reference.

22.1.3.1 Importing Dependencies

The examples in the previous section describe how to specify managed dependencies through inheritance. However, in larger projects it may be impossible to accomplish this since a project can only inherit from a single parent. To accommodate this, projects can import managed dependencies from other projects. This is accomplished by declaring a POM artifact as a dependency with a scope of "import".

Project B:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>B</artifactId>
  <packaging>pom</packaging>
  <name>B</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>A</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>d</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>test</groupId>
      <artifactId>a</artifactId>
      <version>1.0</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>test</groupId>
      <artifactId>c</artifactId>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

Assuming A is the POM defined in the preceding example, the end result would be the same. All of A's managed dependencies would be incorporated into B except for d since it is defined in this POM.

Project X:
Project X:

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>X</artifactId>
  <packaging>pom</packaging>
  <name>X</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>a</artifactId>
        <version>1.1</version>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>b</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>

Project Y:

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>Y</artifactId>
  <packaging>pom</packaging>
  <name>Y</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>a</artifactId>
        <version>1.2</version>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>c</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>

Project Z:

<project>
  <modelVersion>4.0.0</modelVersion>
</project>
In the example above Z imports the managed dependencies from both X and Y. However, both X and Y contain dependency a. Here, version 1.1 of a would be used since X is declared first and a is not declared in Z's dependencyManagement.

This process is recursive. For example, if X imports another POM, Q, when Z is processed it will simply appear that all of Q's managed dependencies are defined in X.

### 22.1.3.2 Bill of Materials (BOM) POMs

Imports are most effective when used for defining a "library" of related artifacts that are generally part of a multiproject build. It is fairly common for one project to use one or more artifacts from these libraries. However, it has sometimes been difficult to keep the versions in the project using the artifacts in synch with the versions distributed in the library. The pattern below illustrates how a "bill of materials" (BOM) can be created for use by other projects.

The root of the project is the BOM POM. It defines the versions of all the artifacts that will be created in the library. Other projects that wish to use the library should import this POM into the dependencyManagement section of their POM.
The parent subproject has the BOM POM as its parent. It is a normal multiproject pom.

```xml
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>bom</artifactId>
  </parent>
  <groupId>com.test</groupId>
  <artifactId>parent</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.12</version>
      </dependency>
      <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.1.1</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <modules>
    <module>project1</module>
    <module>project2</module>
  </modules>
</project>
```
Next are the actual project POMs.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>parent</artifactId>
  </parent>
  <groupId>com.test</groupId>
  <artifactId>project1</artifactId>
  <version>${project1Version}</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
    </dependency>
  </dependencies>
</project>
```

The project that follows shows how the library can now be used in another project without having to specify the dependent project's versions.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>parent</artifactId>
  </parent>
  <groupId>com.test</groupId>
  <artifactId>project2</artifactId>
  <version>${project2Version}</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </dependency>
  </dependencies>
</project>
```

The project that follows shows how the library can now be used in another project without having to specify the dependent project's versions.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test</groupId>
  <artifactId>use</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
      </dependency>
      <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```
Finally, when creating projects that import dependencies, beware of the following:

- Do not attempt to import a POM that is defined in a submodule of the current POM. Attempting to do that will result in the build failing since it won't be able to locate the POM.
- Never declare the POM importing a POM as the parent (or grandparent, etc) of the target POM. There is no way to resolve the circularity and an exception will be thrown.
- When referring to artifacts whose POMs have transitive dependencies, the project needs to specify versions of those artifacts as managed dependencies. Not doing so results in a build failure since the artifact may not have a version specified. (This should be considered a best practice in any case as it keeps the versions of artifacts from changing from one build to the next).

### 22.1.4 System Dependencies

Important note: This is deprecated.

Dependencies with the scope `system` are always available and are not looked up in repository. They are usually used to tell Maven about dependencies which are provided by the JDK or the VM. Thus, system dependencies are especially useful for resolving dependencies on artifacts which are now provided by the JDK, but were available as separate downloads earlier. Typical examples are the JDBC standard extensions or the Java Authentication and Authorization Service (JAAS).

A simple example would be:

```xml
<dependencies>
  ...
  <dependency>
    <groupId>javax.sql</groupId>
    <artifactId>jdbc-stdext</artifactId>
    <version>2.0</version>
    <scope>system</scope>
    <systemPath>${java.home}/lib/rt.jar</systemPath>
  </dependency>
</dependencies>
```
If your artifact is provided by the JDK’s `tools.jar`, the system path would be defined as follows:

```xml
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>sun.jdk</groupId>
      <artifactId>tools</artifactId>
      <version>1.5.0</version>
      <scope>system</scope>
      <systemPath>${java.home}/../lib/tools.jar</systemPath>
    </dependency>
  </dependencies>
  ...
</project>
```
23 Plugin Development

23.1 Introduction to Maven Plugin Development

Maven consists of a core engine which provides basic project-processing capabilities and build-process management, and a host of plugins which are used to execute the actual build tasks.

23.1.1 What is a Plugin?

"Maven" is really just a core framework for a collection of Maven Plugins. In other words, plugins are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on. Almost any action that you can think of performing on a project is implemented as a Maven plugin.

Plugins are the central feature of Maven that allow for the reuse of common build logic across multiple projects. They do this by executing an "action" (i.e. creating a WAR file or compiling unit tests) in the context of a project’s description - the Project Object Model (POM). Plugin behavior can be customized through a set of unique parameters which are exposed by a description of each plugin goal (or Mojo).

One of the simplest plugins in Maven is the Clean Plugin. The Maven Clean plugin (maven-clean-plugin) is responsible for removing the target directory of a Maven project. When you run "mvn clean", Maven executes the "clean" goal as defined in the Clean plug-in, and the target directory is removed. The Clean plugin defines a parameter which can be used to customize plugin behavior, this parameter is called outputDirectory and it defaults to ${project.build.directory}.

23.1.2 What is a Mojo (And Why the H--- is it Named 'Mojo')?

A Mojo is really just a goal in Maven, and plug-ins consist of any number of goals (Mojos). Mojos can be defined as annotated Java classes or Beanshell script. A Mojo specifies metadata about a goal: a goal name, which phase of the lifecycle it fits into, and the parameters it is expecting.

MOJO is a play on POJO (Plain-old-Java-object), substituting "Maven" for "Plain". Mojo is also an interesting word (see definition). From Wikipedia, a "mojo" is defined as: "...a small bug worn by a person under the clothes (also known as a mojo hand). Such bags were thought to have supernatural powers, such as protecting from evil, bringing good luck, etc."

23.1.3 What is the Build Lifecycle? (Overview)

The build lifecycle is a series of common stages through which all project builds naturally progress. Plugin goals are bound to specific stages in the lifecycle.

23.2 Resources

1. Plugin Development Center
2. Configuring plugins
24 Configuring Plug-ins

24.1 Guide to Configuring Plug-ins

1. Generic Configuration
   1. Help Goal
2. Configuring Parameters
   1. Mapping Simple Objects
   2. Mapping Complex Objects
   3. Mapping Collections
      1. Mapping Lists
      2. Mapping Maps
      3. Mapping Properties
2. Configuring Build Plugins
   1. Using the <executions> Tag
   2. Using the <dependencies> Tag
   3. Using the <inherited> Tag In Build Plugins
3. Configuring Reporting Plugins
   1. Using the <reporting> Tag VS <build> Tag
   2. Using the <reportSets> Tag
   3. Using the <inherited> Tag In Reporting Plugins

24.1.1 Introduction

In Maven, there are two kinds of plugins, build and reporting:

- **Build plugins** are executed during the build and configured in the <build/> element.
- **Reporting plugins** are executed during the site generation and configured in the <reporting/> element.

All plugins should have minimal required information: groupId, artifactId and version.

**Important Note:** Always define each version of the plugins used by the build to guarantee the build reproducibility. A good practice is to specify them in the <build><pluginManagement/></build> elements for each build plugins. (Generally, you will define a <pluginManagement/> element in a parent POM.) For reporting plugins, specify each version in the <reporting><plugins/></reporting> elements (and surely in the <build><pluginManagement/></build> elements too).

24.1.2 Generic Configuration

Maven plugins (build and reporting) are configured by specifying a <configuration> element where the child elements of the <configuration> element are mapped to fields, or setters, inside your Mojo. (Remember that a plug-in consists of one or more Mojos where a Mojo maps to a goal.) Say, for example, you have a Mojo that performs a query against a particular URL, with a specified timeout and list of options. The Mojo might look like the following:

```java
/**
 * @goal query
```
public class MyQueryMojo
        extends AbstractMojo
        {
            @Parameter(property = "query.url", required = true)
            private String url;
            @Parameter(property = "timeout", required = false, defaultValue = "50")
            private int timeout;
            @Parameter(property = "options")
            private String[] options;
        public void execute()
            throws MojoExecutionException
            {
            ...
            }
        }

To configure the Mojo from your POM with the desired URL, timeout and options you might have something like the following:

<project>
...<build>
    <plugins>
        <plugin>
            <artifactId>maven-myquery-plugin</artifactId>
            <version>1.0</version>
            <configuration>
                <url>http://www.foobar.com/query</url>
                <timeout>10</timeout>
                <options>
                    <option>one</option>
                    <option>two</option>
                    <option>three</option>
                </options>
            </configuration>
        </plugin>
    </plugins>
</build>
...
</project>

The elements in the configuration match the names of the fields in the Mojo. The mapping is straightforward, the url element maps to the url field, the timeout element maps to the timeout field and the options element maps to the options field. The mapping mechanism can deal with arrays by inspecting the type of the field and determining if a suitable mapping is possible.

For Mojos that are intended to be executed directly from the CLI, their parameters usually provide a means to be configured via system properties instead of a <configuration> section in the POM. The plugin documentation for those parameters will list an expression that denotes the system properties for the configuration. In the Mojo above, the parameter url is associated with the expression ${query.url}, meaning its value can be specified by the system property query.url as shown below:

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
The name of the system property does not necessarily match the name of the mojo parameter. While this is a rather common practice, you will often notice plugins that employ some prefix for the system properties to avoid name clashes with other system properties. Though rarely, there are also plugin parameters that (e.g. for historical reasons) employ system properties which are completely unrelated to the parameter name. So be sure to have a close look at the plugin documentation.

24.1.2.1 Help Goal
Most Maven plugins have a help goal that prints a description of the plugin and its parameters and types. For instance, to see help for the javadoc goal, type:

```
mvn javadoc:help -Ddetail -Dgoal=javadoc
```

And you will see all parameters for the javadoc:javadoc goal, similar to this page.

24.1.2.2 Configuring Parameters
24. Mapping Simple Objects
Mapping simple types, like Boolean or Integer, is very simple. The `<configuration>` element might look like the following:

```
<configuration>
  <myString>a string</myString>
  <myBoolean>true</myBoolean>
  <myInteger>10</myInteger>
  <myDouble>1.0</myDouble>
  <myFile>c:\temp</myFile>
  <myURL>http://maven.apache.org</myURL>
</configuration>
```

24. Mapping Complex Objects
Mapping complex types is also fairly straightforward. Let's look at a simple example where we are trying to map a configuration for Person object. The `<configuration/>` element might look like the following:

```
...<configuration>
  <person>
    <firstName>Jason</firstName>
    <lastName>van Zyl</lastName>
  </person>
</configuration>
...
```

The rules for mapping complex objects are as follows:

- There must be a private field that corresponds to name of the element being mapped. So in our case the `person` element must map to a `person` field in the mojo.
- The object instantiated must be in the same package as the Mojo itself. So if your mojo is in `com.mycompany.mojo.query` then the mapping mechanism will look in that package for an
object named Person. The mechanism capitalizes the first letter of the element name and uses that to search for the object to instantiate.

- If you wish to have the object to be instantiated live in a different package or have a more complicated name, specify this using an implementation attribute like the following:

```
...<configuration>
  <person implementation="com.mycompany mojo.query.SuperPerson">
    <firstName>Jason</firstName>
    <lastName>van Zyl</lastName>
  </person>
</configuration>
...```

24. Mapping Collections

The configuration mapping mechanism can easily deal with most collections so let’s go through a few examples to show you how it’s done:

24. Mapping Lists

Mapping lists works in much the same way as mapping to arrays where you a list of elements will be mapped to the List. So if you have a mojo like the following:

```java
public class MyAnimalMojo
    extends AbstractMojo
{
    @Parameter(property = "animals")
    private List animals;
    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

Where you have a field named animals then your configuration for the plug-in would look like the following:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-myanimal-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          <animals>
            <animal>cat</animal>
            <animal>dog</animal>
            <animal>aardvark</animal>
          </animals>
        </configuration>
      </plugin>
    </plugins>
  </build>
```
Where each of the animals listed would be entries in the animals field. Unlike arrays, collections have no specific component type. In order to derive the type of a list item, the following strategy is used:

1. If the XML element contains an implementation hint attribute, that is used
2. If the XML tag contains a ., try that as a fully qualified class name
3. Try the XML tag (with capitalized first letter) as a class in the same package as the mojo/object being configured
4. If the element has no children, assume its type is String. Otherwise, the configuration will fail.

### 24. Mapping Maps

In the same way, you could define maps like the following:

```java
@Parameter(property = "myMap")
private Map myMap;
```

```
<configuration>
  <myMap>
    <key1>value1</key1>
    <key2>value2</key2>
  </myMap>
</configuration>
```

### 24. Mapping Properties

Properties should be defined like the following:

```java
@Parameter(property = "myProperties")
private Properties myProperties;
```

```
<configuration>
  <myProperties>
    <property>
      <name>propertyName1</name>
      <value>propertyValue1</value>
    </property>
    <property>
      <name>propertyName2</name>
      <value>propertyValue2</value>
    </property>
  </myProperties>
</configuration>
```
24.1.3 Configuring Build Plugins

The following is only to configure Build plugins in the <build> element.

24.1.3.1 Using the <executions> Tag

You can also configure a mojo using the <executions> tag. This is most commonly used for mojos that are intended to participate in some phases of the build lifecycle. Using MyQueryMojo as an example, you may have something that will look like:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-myquery-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <id>execution1</id>
            <phase>test</phase>
            <configuration>
              <url>http://www.foo.com/query</url>
              <timeout>10</timeout>
              <options>
                <option>one</option>
                <option>two</option>
                <option>three</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
          <execution>
            <id>execution2</id>
            <configuration>
              <url>http://www.bar.com/query</url>
              <timeout>15</timeout>
              <options>
                <option>four</option>
                <option>five</option>
                <option>six</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
```
The first execution with id "execution1" binds this configuration to the test phase. The second execution does not have a `<phase>` tag, how do you think will this execution behave? Well, goals can have a default phase binding as discussed further below. If the goal has a default phase binding then it will execute in that phase. But if the goal is not bound to any lifecycle phase then it simply won't be executed during the build lifecycle.

Note that while execution id's have to be unique among all executions of a single plugin within a POM, they don't have to be unique across an inheritance hierarchy of POMs. Executions of the same id from different POMs are merged. The same applies to executions that are defined by profiles.

How about if we have a multiple executions with different phases bound to it? How do you think will it behave? Let us use the example POM above again, but this time we shall bind `execution2` to a phase.

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        ...
        <executions>
          <execution>
            <id>execution1</id>
            <phase>test</phase>
            ...
          </execution>
          <execution>
            <id>execution2</id>
            <phase>install</phase>
            <configuration>
              <url>http://www.bar.com/query</url>
              <timeout>15</timeout>
              <options>
                <option>four</option>
                <option>five</option>
                <option>six</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

If there are multiple executions bound to different phases, then the mojo is executed once for each phase indicated. Meaning, `execution1` will be executed applying the configuration setup when the phase of the build is test, and `execution2` will be executed applying the configuration setup when the build phase is already in install.
Now, let us have another mojo example which shows a default lifecycle phase binding.

```java
/**
 * @goal query
 * @phase package
 */
public class MyBoundQueryMojo
    extends AbstractMojo
{
    @Parameter(property = "query.url", required = true)
    private String url;
    @Parameter(property = "timeout", required = false, defaultValue = "50")
    private int timeout;
    @Parameter(property = "options")
    private String[] options;
    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

From the above mojo example, `MyBoundQueryMojo` is by default bound to the package phase (see the `@phase` notation). But if we want to execute this mojo during the install phase and not with package we can rebind this mojo into a new lifecycle phase using the `<phase>` tag under `<execution>`.

```xml
<project>
...
<build>
<plugins>
<plugin>
<artifactId>maven-myquery-plugin</artifactId>
<version>1.0</version>
<executions>
<execution>
    <id>execution1</id>
    <phase>install</phase>
    <configuration>
        <url>http://www.bar.com/query</url>
        <timeout>15</timeout>
        <options>
            <option>four</option>
            <option>five</option>
            <option>six</option>
        </options>
    </configuration>
    <goals>
        <goal>query</goal>
    </goals>
</execution>
<executions>
</executions>
</plugin>
</plugins>
```
Now, MyBoundQueryMojo default phase which is package has been overridden by install phase.

**Note:** Configurations inside the `<executions>` element used to differ from those that are outside `<executions>` in that they could not be used from a direct command line invocation because they were only applied when the lifecycle phase they were bound to was invoked. So you had to move a configuration section outside of the `executions` section to apply it globally to all invocations of the plugin. Since Maven 3.3.1 this is not the case anymore as you can specify on the command line the execution id for direct plugin goal invocation. Hence if you want to run the above plugin and it's specific execution1's configuration from the command-line, you can execute:

```bash
mvn myqueryplugin:queryMojo@execution1
```

### 24.1.3.2 Using the `<dependencies>` Tag

You could configure the dependencies of the Build plugins, commonly to use a more recent dependency version.

For instance, the Maven Antrun Plugin version 1.2 uses Ant version 1.6.5, if you want to use the latest Ant version when running this plugin, you need to add `<dependencies>` element like the following:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.2</version>
        ...
        <dependencies>
          <dependency>
            <groupId>org.apache.ant</groupId>
            <artifactId>ant</artifactId>
            <version>1.7.1</version>
          </dependency>
          <dependency>
            <groupId>org.apache.ant</groupId>
            <artifactId>ant-launcher</artifactId>
            <version>1.7.1</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

### 24.1.3.3 Using the `<inherited>` Tag In Build Plugins

By default, plugin configuration should be propagated to child POMs, so to break the inheritance, you could use the `<inherited>` tag:
24 Configuring Plug-ins

24.1.4 Configuring Reporting Plugins

The following is only to configure Reporting plugins in the `<reporting>` element.

24.1.4.1 Using the `<reporting>` Tag VS `<build>` Tag

Configuring a reporting plugin in the `<reporting>` or `<build>` elements in the pom does **NOT** have the same behavior!

```xml
mvn site
```

It uses **only** the parameters defined in the `<configuration>` element of each reporting Plugin specified in the `<reporting>` element, i.e. `site` always **ignores** the parameters defined in the `<configuration>` element of each plugin specified in `<build>`.

```xml
mvn aplugin:areportgoal
```

It uses **firstly** the parameters defined in the `<configuration>` element of each reporting Plugin specified in the `<reporting>` element; if a parameter is not found, it will look up to a parameter defined in the `<configuration>` element of each plugin specified in `<build>`.

24.1.4.2 Using the `<reportSets>` Tag

You can configure a reporting plugin using the `<reportSets>` tag. This is most commonly used to generate reports selectively when running `mvn site`. The following will generate only the project team report.

```xml
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.1.2</version>
        <reportSets>
          <reportSet>
            <reports>
              <report>project-team</report>
            </reports>
          </reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```
24 Configuring Plug-ins

Notes:

1. To exclude all reports, you need to use:

```xml
<reportSets>
  <reportSet>
    <reports/>
  </reportSet>
</reportSets>
```

2. Refer to each Plugin Documentation (i.e. plugin-info.html) to know the available report goals.

24.1.4.3 Using the `<inherited>` Tag In Reporting Plugins

Similar to the build plugins, to break the inheritance, you can use the `<inherited>` tag:

```xml
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.1.2</version>
        <inherited>false</inherited>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```
25 Plugin Prefix Resolution

25.1 Introduction to Plugin Prefix Resolution

When you execute Maven using a standard lifecycle phase, resolving the plugins that participate in that lifecycle is a relatively simple process. However, when you directly invoke a mojo from the command line, as in the case of clean, Maven must have some way of reliably resolving the clean plugin prefix to the maven-clean-plugin. This provides brevity for command-line invocations, while preserving the descriptiveness of the plugin's real artifactId.

To complicate matters even more, not all plugins should be forced to have the same groupId in the repository. Since groupIds are presumed to be controlled by one project, and multiple projects may release plugins for Maven, it follows that plugin-prefix mappings must also accommodate multiple plugin groupIds.

To address these concerns, Maven provides a new piece of repository-level metadata (not associated with any single artifact) for plugin groups, along with a plugin mapping manager to organize multiple plugin groups and provide search functionality.

25.1.1 Specifying a Plugin's Prefix

In order to give users a convenient prefix with which to reference your plugin a prefix must be associated with your plugin when it is built. By default, Maven will make a guess at the plugin-prefix to be used, by removing any instances of "maven" or "plugin" surrounded by hyphens in the plugin's artifact ID. The conventional artifact ID formats to use are:

- maven-$\{prefix\}$-plugin - for official plugins maintained by the Apache Maven team itself (you must not use this naming pattern for your plugin, see this note for more informations)
- $\{prefix\}$-maven-plugin - for plugins from other sources

If your plugin's artifactId fits this pattern, Maven will automatically map your plugin to the correct prefix in the metadata stored within your plugin's groupId path on the repository. However, if you want to customize the prefix used to reference your plugin, you can specify the prefix directly through a configuration parameter on the maven-plugin-plugin in your plugin’s POM:

```xml
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          ...
          <goalPrefix>somePrefix</goalPrefix>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

The above configuration will allow users to refer to your plugin by the prefix somePrefix, as in the following example:
25 Plugin Prefix Resolution

mvn somePrefix:goal

25.1.2 Mapping Prefixes to Plugins

For each groupId configured for searching, Maven will:

1. Download maven-metadata.xml from each remote repository into the local repository, and name it maven-metadata-${repoId}.xml within the path of ${groupId}.
2. Load these metadata files, along with maven-metadata-local.xml (if it exists), within the path of ${groupId}. Merge them.
3. Lookup the plugin prefix in the merged metadata. If it’s mapped, it should refer to a concrete groupId-artifactId pair. Otherwise, go on to #1 for the next groupId in the user's plugin-groups.

These metadata files consist of the groupId it represents (for clarity when the file is opened outside the context of the directory), and a group of plugin elements. Each plugin in this list contains a prefix element denoting the plugin's command-line prefix, and an artifactId element, which provides the other side of the prefix mapping and provides enough information to lookup and use the plugin. When a plugin is installed or deployed, the appropriate metadata file is located - and if the prefix mapping is missing - modified to include the plugin-prefix mapping.

25.1.3 Configuring Maven to Search for Plugins

By default, Maven will search the groupId org.apache.maven.plugins for prefix-to-artifactId mappings for the plugins it needs to perform a given build. However, as previously mentioned, the user may have a need for third-party plugins. Since the Maven project is assumed to have control over the default plugin groupId, this means configuring Maven to search other groupId locations for plugin-prefix mappings.

As it turns out, this is simple. In the Maven settings file (per-user: ${user.home}/.m2/settings.xml; global: ${maven.home}/conf/settings.xml), you can provide a custom pluginGroups section, listing the plugin groupIds you want to search (each groupId goes in its own pluginGroup sub-element). For example, if my project uses a Modello model file, I might have the following in my settings:

```xml
<pluginGroups>
  <pluginGroup>org.codehaus.modello</pluginGroup>
</pluginGroups>
```

This allows me to execute the following, which will generate Java classes from the model:

```sh
mvn -Dversion=4.0.0 -Dmodel=maven.mdo modello:java
```

Maven will always search the following groupId's after searching any plugin groups specified in the user's settings:

- org.apache.maven.plugins
- org.codehaus.mojo

NOTE: When specifying plugin groups to be used in searching for a prefix mapping, order is critical! By specifying a pluginGroup of com.myco.plugins with a prefix of clean, I can override the usage of the maven-clean-plugin when clean:clean is invoked.

NOTE2: For more information on settings.xml, see [1].

25.1.4 Resources

1 Guide to Configuring Maven
26 Developing Java Plugins

26.1 Introduction
This guide is intended to assist users in developing Java plugins for Maven.

26.1.1 Important Notice: Plugin Naming Convention and Apache Maven Trademark
You will typically name your plugin <yourplugin>-maven-plugin.
Calling it maven-<yourplugin>-plugin (note "Maven" is at the beginning of the plugin name) is strongly discouraged since it's a reserved naming pattern for official Apache Maven plugins maintained by the Apache Maven team with groupId org.apache.maven.plugins. Using this naming pattern is an infringement of the Apache Maven Trademark.

26.1.2 Your First Plugin
In this section we will build a simple plugin with one goal which takes no parameters and simply displays a message on the screen when run. Along the way, we will cover the basics of setting up a project to create a plugin, the minimal contents of a Java mojo which will define goal code, and a couple ways to execute the mojo.

26.1.2.1 Your First Mojo
At its simplest, a Java mojo consists simply of a single class representing one plugin’s goal. There is no requirement for multiple classes like EJBs, although a plugin which contains a number of similar mojos is likely to use an abstract superclass for the mojos to consolidate code common to all mojos.

When processing the source tree to find mojos, plugin-tools looks for classes with either @Mojo Java 5 annotation or "goal" javadoc annotation. Any class with this annotation are included in the plugin configuration file.

26.1.2.2 A Simple Mojo
Listed below is a simple mojo class which has no parameters. This is about as simple as a mojo can be. After the listing is a description of the various parts of the source.

```java
class GreetingMojo extends AbstractMojo {
  public void execute() throws MojoExecutionException {
    getLog().info( "Hello, world." );
  }
}
```

- The class org.apache.maven.plugin.AbstractMojo provides most of the infrastructure required to implement a mojo except for the execute method.
• The annotation "@Mojo" is required and control how and when the mojo is executed.
• The `execute` method can throw two exceptions:
  • `org.apache.maven.plugin.MojoExecutionException` if an unexpected problem occurs. Throwing this exception causes a "BUILD ERROR" message to be displayed.
  • `org.apache.maven.plugin.MojoFailureException` if an expected problem (such as a compilation failure) occurs. Throwing this exception causes a "BUILD FAILURE" message to be displayed.
• The `getLog` method (defined in `AbstractMojo`) returns a log4j-like logger object which allows plugins to create messages at levels of "debug", "info", "warn", and "error". This logger is the accepted means to display information to the user. Please have a look at the section Retrieving the Mojo Logger for a hint on its proper usage.

All Mojo annotations are described by the Mojo API Specification.

### 26.1.2.2 Project Definition

Once the mojos have been written for the plugin, it is time to build the plugin. To do this properly, the project's descriptor needs to have a number of settings set properly:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>groupId</code></td>
<td>This is the group ID for the plugin, and should match the common prefix to the packages used by the mojos</td>
</tr>
<tr>
<td><code>artifactId</code></td>
<td>This is the name of the plugin</td>
</tr>
<tr>
<td><code>version</code></td>
<td>This is the version of the plugin</td>
</tr>
<tr>
<td><code>packaging</code></td>
<td>This should be set to &quot;maven-plugin&quot;</td>
</tr>
<tr>
<td><code>dependencies</code></td>
<td>A dependency must be declared to the Maven Plugin Tools API to resolve &quot;AbstractMojo&quot; and related classes</td>
</tr>
</tbody>
</table>

Listed below is an illustration of the sample mojo project's pom with the parameters set as described in the above table:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample.plugin</groupId>
  <artifactId>hello-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>
  <name>Sample Parameter-less Maven Plugin</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>3.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- dependencies to annotations -->
    <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-annotations</artifactId>
      <version>3.4</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```
26.1.2.3 Building a Plugin

There are few plugins goals bound to the standard build lifecycle defined with the maven-plugin packaging:

<table>
<thead>
<tr>
<th>compile</th>
<th>Compiles the Java code for the plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>process-classes</td>
<td>Extracts data to build the plugin descriptor</td>
</tr>
<tr>
<td>test</td>
<td>Runs the plugin's unit tests</td>
</tr>
<tr>
<td>package</td>
<td>Builds the plugin jar</td>
</tr>
<tr>
<td>install</td>
<td>Installs the plugin jar in the local repository</td>
</tr>
<tr>
<td>deploy</td>
<td>Deploys the plugin jar to the remote repository</td>
</tr>
</tbody>
</table>

For more details, you can look at detailed bindings for maven-plugin packaging.

26.1.2.4 Executing Your First Mojo

The most direct means of executing your new plugin is to specify the plugin goal directly on the command line. To do this, you need to configure the hello-maven-plugin plugin in your project:

```xml
...<build>
  <plugins>
    <plugin>
      <groupId>sample.plugin</groupId>
      <artifactId>hello-maven-plugin</artifactId>
      <version>1.0-SNAPSHOT</version>
    </plugin>
  </plugins>
...</build>
```

And, you need to specify a fully-qualified goal in the form of:

```
mvn groupId:artifactId:version:goal
```

For example, to run the simple mojo in the sample plugin, you would enter "mvn sample.plugin:hello-maven-plugin:1.0-SNAPSHOT:sayhi" on the command line.

**Tips:** version is not required to run a standalone goal.

26. Shortening the Command Line

There are several ways to reduce the amount of required typing:

- If you need to run the latest version of a plugin installed in your local repository, you can omit its version number. So just use "mvn sample.plugin:hello-maven-plugin:sayhi" to run your plugin.
- You can assign a shortened prefix to your plugin, such as mvn hello:sayhi. This is done automatically if you follow the convention of using ${prefix}-maven-plugin (or maven-${prefix}-plugin if the plugin is part of the Apache Maven project). You may also assign
one through additional configuration - for more information see Introduction to Plugin Prefix Mapping.

- Finally, you can also add your plugin's groupId to the list of groupIds searched by default. To do this, you need to add the following to your $USER.home/.m2/settings.xml file:

```xml
<pluginGroups>
  <pluginGroup>sample.plugin</pluginGroup>
</pluginGroups>
```

At this point, you can run the mojo with "mvn hello:sayhi".

26. Attaching the Mojo to the Build Lifecycle

You can also configure your plugin to attach specific goals to a particular phase of the build lifecycle. Here is an example:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>sample.plugin</groupId>
            <artifactId>hello-maven-plugin</artifactId>
            <version>1.0-SNAPSHOT</version>
            <executions>
                <execution>
                    <phase>compile</phase>
                    <goals>
                        <goal>sayhi</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This causes the simple mojo to be executed whenever Java code is compiled. For more information on binding a mojo to phases in the lifecycle, please refer to the Build Lifecycle document.

26.1.3 Mojo archetype

To create a new plugin project, you could using the Mojo archetype with the following command line:

```
mvn archetype:generate
-DgroupId=sample.plugin
-DartifactId=hello-maven-plugin
-DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-plugin
```

26.1.4 Parameters

It is unlikely that a mojo will be very useful without parameters. Parameters provide a few very important functions:

- It provides hooks to allow the user to adjust the operation of the plugin to suit their needs.
- It provides a means to easily extract the value of elements from the POM without the need to navigate the objects.
26.1.4.1 Defining Parameters Within a Mojo

Defining a parameter is as simple as creating an instance variable in the mojo and adding the proper annotations. Listed below is an example of a parameter for the simple mojo:

```java
/**
 * The greeting to display.
 */
@Parameter(property = "sayhi.greeting", defaultValue = "Hello World!"
private String greeting;
```

The portion before the annotations is the description of the parameter. The `@Parameter` annotation identifies the variable as a mojo parameter. The `defaultValue` parameter of the annotation defines the default value for the variable. This value can include expressions which reference the project, such as `"${project.version}"` (more can be found in the "Parameter Expressions" document). The `property` parameter can be used to allow configuration of the mojo parameter from the command line by referencing a system property that the user sets via the `-D` option.

26.1.4.2 Configuring Parameters in a Project

Configuring the parameter values for a plugin is done in a Maven project within the `pom.xml` file as part of defining the plugin in the project. An example of configuring a plugin:

```xml
<plugin>
  <groupId>sample.plugin</groupId>
  <artifactId>hello-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <configuration>
    <greeting>Welcome</greeting>
  </configuration>
</plugin>
```

In the configuration section, the element name ("greeting") is the parameter name and the contents of the element ("Welcome") is the value to be assigned to the parameter.

**Note:** More details can be found in the Guide to Configuring Plugins.

26.1.4.3 Parameter Types With One Value

Listed below are the various types of simple variables which can be used as parameters in your mojos, along with any rules on how the values in the POM are interpreted.

26. Boolean

This includes variables typed `boolean` and `Boolean`. When reading the configuration, the text "true" causes the parameter to be set to true and all other text causes the parameter to be set to false. Example:

```java
/**
 * My boolean.
 */
@Parameter
private boolean myBoolean;
```

```xml
<myBoolean>true</myBoolean>
```
26. Integer Numbers
This includes variables typed `byte`, `Byte`, `int`, `Integer`, `long`, `Long`, `short`, and `Short`. When reading the configuration, the text in the XML file is converted to an integer value using either `Integer.parseInt()` or the `valueOf()` method of the appropriate class. This means that the strings must be valid decimal integer values, consisting only of the digits 0 to 9 with an optional – in front for a negative value. Example:

```java
/**
 * My Integer.
 */
@Parameter
private Integer myInteger;

<myInteger>10</myInteger>
```

26. Floating-Point Numbers
This includes variables typed `double`, `Double`, `float`, and `Float`. When reading the configuration, the text in the XML file is converted to binary form using the `valueOf()` method for the appropriate class. This means that the strings can take on any format specified in section 3.10.2 of the Java Language Specification. Some samples of valid values are 1.0 and 6.02E+23.

```java
/**
 * My Double.
 */
@Parameter
private Double myDouble;

<myDouble>1.0</myDouble>
```

26. Dates
This includes variables typed `Date`. When reading the configuration, the text in the XML file is converted using one of the following date formats: "yyyy-MM-dd HH:mm:ss.S a" (a sample date is "2005-10-06 2:22:55.1 PM") or "yyyy-MM-dd HH:mm:ssa" (a sample date is "2005-10-06 2:22:55PM"). Note that parsing is done using `DateFormat.parse()` which allows some leniency in formatting. If the method can parse a date and time out of what is specified it will do so even if it doesn't exactly match the patterns above. Example:

```java
/**
 * My Date.
 */
@Parameter
private Date myDate;

<myDate>2005-10-06 2:22:55.1 PM</myDate>
```

26. Files and Directories
This includes variables typed `File`. When reading the configuration, the text in the XML file is used as the path to the desired file or directory. If the path is relative (does not start with / or a drive letter like C:), the path is relative to the directory containing the POM. Example:

```java
/**
 * My File.
 */
```
26. URLs
This includes variables typed URL. When reading the configuration, the text in the XML file is used as the URL. The format must follow the RFC 2396 guidelines, and looks like any web browser URL (scheme://host:port/path/to/file). No restrictions are placed on the content of any of the parts of the URL while converting the URL.

```java
/**
 * My URL.
 */
@Parameter
private URL myURL;
```

```xml
<myURL>http://maven.apache.org</myURL>
```

26. Plain Text
This includes variables typed char, Character, StringBuffer, and String. When reading the configuration, the text in the XML file is used as the value to be assigned to the parameter. For char and Character parameters, only the first character of the text is used.

26. Enums
Enumeration type parameters can also be used. First you need to define your enumeration type and afterwards you can use the enumeration type in the parameter definition:

```java
public enum Color {
    GREEN,
    RED,
    BLUE
}
/**
 * My Enum
 */
@Parameter
private Color myColor;
```

So lets have a look like you can use such enumeration in your pom configuration:

```xml
<myColor>GREEN</myColor>
```

You can also use elements from the enumeration type as defaultValues like the following:

```java
public enum Color {
    GREEN,
    RED,
    BLUE
}
/**
 * My Enum
 */
```

@Parameter
private Color myColor;
```
26.1.4.4 Parameter Types With Multiple Values
Listed below are the various types of composite objects which can be used as parameters in your mojos, along with any rules on how the values in the POM are interpreted. In general, the class of the object created to hold the parameter value (as well as the class for each element within the parameter value) is determined as follows (the first step which yields a valid class is used):

1. If the XML element contains an implementation hint attribute, that is used
2. If the XML tag contains a . . try that as a fully qualified class name
3. Try the XML tag (with capitalized first letter) as a class in the same package as the mojo/object being configured
4. For arrays, use the component type of the array (for example, use String for a String[] parameter); for collections and maps, use the class specified in the mojo configuration for the collection or map; use String for entries in a collection and values in a map

Once the type for the element is defined, the text in the XML file is converted to the appropriate type of object

26.Arrays
Array type parameters are configured by specifying the parameter multiple times. Example:

```java
/**
 * My Array.
 */
@Parameter
private String[] myArray;

<myArray>
  <param>value1</param>
  <param>value2</param>
</myArray>
```

26.Collections
This category covers any class which implements java.util.Collection such as ArrayList or HashSet. These parameters are configured by specifying the parameter multiple times just like an array. Example:

```java
/**
 * My List.
 */
@Parameter
private List myList;

<myList>
  <param>value1</param>
  <param>value2</param>
</myList>
```

For details on the mapping of the individual collection elements, see Mapping Lists.
26. Maps
This category covers any class which implements `java.util.Map` such as `HashMap` but does not implement `java.util.Properties`. These parameters are configured by including XML tags in the form `<key>value</key>` in the parameter configuration. Example:

```java
/**
 * My Map.
 */
@Parameter
private Map myMap;

<myMap>
  <key1>value1</key1>
  <key2>value2</key2>
</myMap>
```

26. Properties
This category covers any map which implements `java.util.Properties`. These parameters are configured by including XML tags in the form `<property><name>myName</name><value>myValue</value></property>` in the parameter configuration. Example:

```java
/**
 * My Properties.
 */
@Parameter
private Properties myProperties;

<myProperties>
  <property>
    <name>propertyName1</name>
    <value>propertyValue1</value>
  </property>
  <property>
    <name>propertyName2</name>
    <value>propertyValue2</value>
  </property>
</myProperties>
```

26. Other Object Classes
This category covers any class which does not implement `java.util.Map`, `java.util.Collection`, or `java.util.Dictionary`. Example:

```java
/**
 * My Object.
 */
@Parameter
private MyObject myObject;

<myObject>
  <myField>test</myField>
</myObject>
```
Please see Mapping Complex Objects for details on the strategy used to configure those kind of parameters.

26.1.5 Using Setters

You are not restricted to using private field mapping which is good if you are trying to make you Mojos resuable outside the context of Maven. Using the example above we could name our private fields using the underscore convention and provide setters that the configuration mapping mechanism can use. So our Mojo would look like the following:

```java
public class MyQueryMojo
    extends AbstractMojo
{
    @Parameter(property="url")
    private String _url;
    @Parameter(property="timeout")
    private int _timeout;
    @Parameter(property="options")
    private String[] _options;
    public void setUrl( String url )
    {
        _url = url;
    }
    public void setTimeout( int timeout )
    {
        _timeout = timeout;
    }
    public void setOptions( String[] options )
    {
        _options = options;
    }
    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

Note the specification of the property name for each parameter which tells Maven what setter and getter to use when the field's name does not match the intended name of the parameter in the plugin configuration.

26.1.6 Resources

1. **Mojo Documentation**: Mojo API, Mojo annotations  
2. **Maven Plugin Testing Harness**: Testing framework for your Mojos.  
3. **Plexus**: The IoC container used by Maven.  
4. **Plexus Common Utilities**: Set of utilities classes useful for Mojo development.  
5. **Commons IO**: Set of utilities classes useful for file/path handling.  
6. **Common Bugs and Pitfalls**: Overview of problematic coding patterns.
27 Creating a Site

27.1 Creating a site

27.1.1 Creating Content

The first step to creating your site is to create some content. In Maven, the site content is separated by format, as there are several available.

```
+- src/
  +- site/
    +- apt/
        |  +- index.apt
        !
    +- markdown/
        |  +- content.md
    +- fml/
        |  +- general.fml
        |  +- faq.fml
    +- xdoc/
        |  +- other.xml
    +- site.xml
```

You will notice there is now a `${basedir}/src/site` directory within which is contained a `site.xml` site descriptor along with various directories corresponding to the supported document types.

Let's take a look at the examples of the various document types:

- **apt**: the APT format, "Almost Plain Text", is a wiki-like format that allows you to write simple, structured documents (like this one) very quickly. A full reference of the APT Format is available,
- **markdown**: the well known Markdown format,
- **fml**: the FML format is the FAQ format,
- **xdoc**: an XML document conforming to a small and simple set of tags, see the full reference.

Other formats are available, but at this point these 4 are the best tested.

There are also several possible output formats, but as of Maven Site Plugin, only XHTML is available.

Note that all of the above is optional - just one index file is required in one of the input trees. Each of the paths will be merged together to form the root directory of the site.

27.1.2 Customizing the Look & Feel

If you want to tune the way your site looks, you can use a custom skin to provide your own CSS styles. If that is still not enough, you can even tweak the output templates that Maven uses to generate the site documentation.

You can visit the Skins index to have a look at some of the skins that you can use to change the look of your site.
27.1.3 Generating the Site
Generating the site is very simple, and fast!

`mvn site`

By default, the resulting site will be in `target/site/...`

For more information on the Maven Site Plugin, see the [maven-site-plugin reference](https://maven.apache.org/plugins/maven-site-plugin/).

27.1.4 Deploying the Site

27.1.4.1 Classical Website deployment

To be able to deploy the site with a classical network protocol (ftp, scp, webdav), you must first declare a location to distribute to in your `pom.xml`, similar to the repository for deployment:

```xml
<project>
  ...
  <distributionManagement>
    <site>
      <id>website</id>
      <url>scp://www.mycompany.com/www/docs/project/</url>
    </site>
  </distributionManagement>
  ...
</project>
```

- the `<id>` element identifies the repository, so that you can attach credentials to it in your `settings.xml` file using the `<servers>` element as you would for any other repository,
- the `<url>` gives the location to deploy to. Currently, only SSH is supported by default, as above which copies to the host `www.mycompany.com` in the path `/www/docs/project/`, but you can add more protocols as required. If subprojects inherit the site URL from a parent POM, they will automatically get their `<artifactId>` appended to form their effective deployment location.

Once distribution location is configured, deploying the site is done by using the `site-deploy` phase of the site lifecycle.

`mvn site-deploy`

27.1.4.2 GitHub Pages, Apache svnpubsub/gitpubsub Deployment

When site publication is done with a SCM commit, like with GitHub Pages or Apache svnpubsub/gitpubsub, deploying the site will be done with [Maven SCM Publish Plugin](https://maven.apache.orgPlugins/).

For example with a project hosted on GitHub and using GitHub Pages for its site publication:

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-scm-publish-plugin</artifactId>
  <version>3.1.0</version>
  <configuration>
    <pubScmUrl>${project.scm.developerConnection}</pubScmUrl>
    <scmBranch>gh-pages</scmBranch>
  </configuration>
</plugin>
```

Deploying the site is done in 2 steps:
1. staging the content by using the site phase of the site lifecycle followed by site:stage:mvn site site:stage
2. publishing the staged site to the SCM: mvn scm-publish:publish-scm

27.1.5 Creating a Site Descriptor

The site.xml file is used to describe the structure of the site. A sample is given below:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
name="Maven">
  <bannerLeft>
    <name>Maven</name>
    <src>https://maven.apache.org/images/apache-maven-project.png</src>
    <href>https://maven.apache.org/</href>
  </bannerLeft>
  <bannerRight>
    <src>https://maven.apache.org/images/maven-small.gif</src>
  </bannerRight>
  <body>
    <links>
      <item name="Apache" href="http://www.apache.org/" />
      <item name="Maven 1.x" href="https://maven.apache.org/maven-1.x/"/>
      <item name="Maven 2" href="https://maven.apache.org/"/>
    </links>
    <menu name="Maven 2.0">
      <item name="Introduction" href="index.html"/>
      <item name="Download" href="download.html"/>
      <item name="Release Notes" href="release-notes.html"/>
      <item name="General Information" href="about.html"/>
      <item name="For Maven 1.x Users" href="maven1.html"/>
      <item name="Road Map" href="roadmap.html"/>
    </menu>
    <menu ref="reports"/>
    ...
  </body>
</project>
```

**Note:** The `<menu ref="reports"/>` element above. When building the site, this is replaced by a menu with links to all the reports that you have configured.

More information about the site descriptor is available at the dedicated page of Maven Site Plugin.

27.1.6 Adding Extra Resources

You can add any arbitrary resource to your site by including them in a `resources` directory as shown below. Additional CSS files will be picked up when they are placed in the `css` directory within the `resources` directory.

```
+- src/
  +- site/
    +- resources/
      +- css/
        | +- site.css
```
The file `site.css` will be added to the default XHTML output, so it can be used to adjust the default Maven stylesheets if desired.

The file `pic1.jpg` will be available via a relative reference to the `images` directory from any page in your site.

### 27.1.7 Configuring Reports

Maven has several reports that you can add to your web site to display the current state of the project. These reports take the form of plugins, just like those used to build the project.

There are many standard reports that are available by gleaning information from the POM. Currently what is provided by default are:

- Dependencies Report
- Mailing Lists
- Continuous Integration
- Source Repository
- Issue Tracking
- Project Team
- License

To find out more please refer to the [Project Info Reports Plugin](#).

To add these reports to your site, you must add the plugins to a special `<reporting>` section in the POM. The following example shows how to configure the standard project information reports that display information from the POM in a friendly format:

```xml
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.8</version>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

If you have included the appropriate `<menu ref="reports"/>` tag in your `site.xml` descriptor, then when you regenerate the site those items will appear in the menu.

**Note:** Many report plugins provide a parameter called `outputDirectory` or similar to specify the destination for their report outputs. This parameter is only relevant if the report plugin is run standalone, i.e. by invocation directly from the command line. In constrast, when reports are generated as part of the site, the configuration of the Maven Site Plugin will determine the effective output directory to ensure that all reports end up in a central location.
27.1.8 Internationalization

Internationalization in Maven is very simple, as long as the reports you are using have that particular locale defined. For an overview of supported languages and instructions on how to add further languages, please see the related article Internationalization from the Maven Site Plugin.

To enable multiple locales, add a configuration similar to the following to your POM:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-site-plugin</artifactId>
        <version>3.4</version>
        <configuration>
          <locales>en,fr</locales>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

This will generate both an English and a French version of the site. If en is your current locale, then it will be generated at the root of the site, with a copy of the French translation of the site in the fr/ subdirectory.

To add your own content for that translation instead of using the default, place a subdirectory with that locale name in your site directory and create a new site descriptor with the locale in the file name. For example:

```bash
+- src/
  +- site/
    +- apt/
      |  +- index.apt (Default version)
      |
    +- fr/
      |  +- apt/
      |     +- index.apt (French version)
      |
    +- site.xml (Default site descriptor)
    +- site_fr.xml (French site descriptor)
```

With one site descriptor per language, the translated site(s) can evolve independently.
28 Snippet Macro

28.1 Guide to the Snippet Macro

When generating your project website with Maven, you have the option of dynamically including snippets of source code in your pages.

A snippet is a section of a source code file that is surrounded by specially formatted comments.

This functionality is inspired by the Confluence snippet macro, and is provided by the Maven Doxia project by way of the Maven Site Plugin.

To include snippets of source code in your documentation, first add comments in the source document surrounding the lines you want to include, and then refer to the snippet by its id in the documentation file. Each snippet must be assigned an id, and the id must be unique within the source document. The id parameter is not required if you want to include the entire file.

Following are examples of snippets in various source documents, as well as the corresponding macros in the APT documentation format.

See the Doxia Macros Guide for more information and examples.

28.1.1 Snippets in Sources

28.1.1.1 Java

```java
// START SNIPPET: snip-id
public static void main( String[] args )
{
    System.out.println( "Hello World!" );
}
// END SNIPPET: snip-id
```

28.1.1.2 XML

```xml
<!-- START SNIPPET: snip-id -->
<navigation-rule>
    <from-view-id>/logon.jsp</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/mainMenu.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
<!-- END SNIPPET: snip-id -->
```

28.1.1.3 JSP

```jsp
<%-- START SNIPPET: snip-id --%>
<ul>
    <li><a href="newPerson!input.action">Create</a> a new person</li>
    <li><a href="listPeople.action">List</a> all people</li>
</ul>
<%-- END SNIPPET: snip-id --%>
```
28.1.2 Snippets in Documentation

28.1.2.1 APT

\%{snippet|id=snip-id|url=http://svn.example.com/path/to/Sample.java}\n\%{snippet|id=snip-id|url=file:///path/to/Sample.java}\n
As of doxia-core version 1.0-alpha-9, a ‘file’ parameter is also available. If a full path is not specified, the location is assumed to be relative to $\{basedir\}$.

~~ Since doxia-core 1.0-alpha-9
\%{snippet|id=abc|file=src/main/webapp/index.jsp}\n
- Macros in apt must not be indented.
- Exactly one of url or file must be specified.
29 What is an Archetype

29.1 Introduction to Archetypes

29.2 What is Archetype?

In short, Archetype is a Maven project templating toolkit. An archetype is defined as an original pattern or model from which all other things of the same kind are made. The name fits as we are trying to provide a system that provides a consistent means of generating Maven projects. Archetype will help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates.

Using archetypes provides a great way to enable developers quickly in a way consistent with best practices employed by your project or organization. Within the Maven project, we use archetypes to try and get our users up and running as quickly as possible by providing a sample project that demonstrates many of the features of Maven, while introducing new users to the best practices employed by Maven. In a matter of seconds, a new user can have a working Maven project to use as a jumping board for investigating more of the features in Maven. We have also tried to make the Archetype mechanism additive, and by that we mean allowing portions of a project to be captured in an archetype so that pieces or aspects of a project can be added to existing projects. A good example of this is the Maven site archetype. If, for example, you have used the quick start archetype to generate a working project, you can then quickly create a site for that project by using the site archetype within that existing project. You can do anything like this with archetypes.

You may want to standardize J2EE development within your organization, so you may want to provide archetypes for EJBs, or WARs, or for your web services. Once these archetypes are created and deployed in your organization's repository, they are available for use by all developers within your organization.

29.2.1 Using an Archetype

To create a new project based on an Archetype, you need to call \texttt{mvn archetype:generate} goal, like the following:

```bash
mvn archetype:generate
```

Please refer to Archetype Plugin page.

29.2.2 Provided Archetypes

Maven provides several Archetype artifacts:

<table>
<thead>
<tr>
<th>Archetype ArtfactIds</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maven-archetype-archetype</td>
<td>An archetype to generate a sample archetype project.</td>
</tr>
<tr>
<td>maven-archetype-j2ee-simple</td>
<td>An archetype to generate a simplified sample J2EE application.</td>
</tr>
<tr>
<td>maven-archetype-mojo</td>
<td>An archetype to generate a sample a sample Maven plugin.</td>
</tr>
<tr>
<td>maven-archetype-plugin</td>
<td>An archetype to generate a sample Maven plugin.</td>
</tr>
<tr>
<td>maven-archetype-plugin-site</td>
<td>An archetype to generate a sample Maven plugin site.</td>
</tr>
<tr>
<td>Archetype Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>maven-archetype-portlet</td>
<td>An archetype to generate a sample JSR-268 Portlet.</td>
</tr>
<tr>
<td>maven-archetype-quickstart</td>
<td>An archetype to generate a sample Maven project.</td>
</tr>
<tr>
<td>maven-archetype-simple</td>
<td>An archetype to generate a simple Maven project.</td>
</tr>
<tr>
<td>maven-archetype-site</td>
<td>An archetype to generate a sample Maven site which demonstrates some of the supported document types like APT, XDoc, and FML and demonstrates how to i18n your site.</td>
</tr>
<tr>
<td>maven-archetype-site-simple</td>
<td>An archetype to generate a sample Maven site.</td>
</tr>
<tr>
<td>maven-archetype-webapp</td>
<td>An archetype to generate a sample Maven Webapp project.</td>
</tr>
</tbody>
</table>

For more information on these archetypes, please refer to the [Maven Archetype Bundles page](https://maven.apache.org/archetype/).

### 29.2.3 What makes up an Archetype?

Archetypes are packaged up in a JAR and they consist of the archetype metadata which describes the contents of archetype, and a set of *Velocity* templates which make up the prototype project. If you would like to know how to make your own archetypes, please refer to our [Guide to creating archetypes](https://maven.apache.org/guides/introduction/introduction-to-archetypes.html).
30 Creating Archetypes

30.1 Guide to Creating Archetypes

Creating an archetype is a pretty straightforward process. An archetype is a very simple artifact, that contains the project prototype you wish to create. An archetype is made up of:

- an archetype descriptor (`archetype-metadata.xml` in directory: `src/main/resources/META-INF/maven/`). It lists all the files that will be contained in the archetype and categorizes them so they can be processed correctly by the archetype generation mechanism.
- the prototype files that are copied by the archetype plugin (directory: `src/main/resources/archetype-resources/`)
- the prototype pom (`pom.xml` in: `src/main/resources/archetype-resources`)
- a pom for the archetype (`pom.xml` in the archetype's root directory).

To create an archetype follow these steps:

30.1.1 1. Create a new project and pom.xml for the archetype artifact

An example pom.xml for an archetype artifact looks as follows:

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.groupId</groupId>
  <artifactId>my-archetype-id</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-archetype</packaging>
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.archetype</groupId>
        <artifactId>archetype-packaging</artifactId>
        <version>3.1.1</version>
      </extension>
    </extensions>
    <filesets>
      <fileSet filtered="true" packaged="true">
        <groupId>org.apache.maven.archetype</groupId>
        <artifactId>archetype-packaging</artifactId>
        <version>3.1.1</version>
      </fileSet>
    </filesets>
  </build>
</project>
```

All you need to specify is a groupId, artifactId and version. These three parameters will be needed later for invoking the archetype via archetype:generate from the commandline.

30.1.2 2. Create the archetype descriptor

The archetype descriptor is a file called `archetype-metadata.xml` which must be located in the `src/main/resources/META-INF/maven/` directory. An example of an archetype descriptor can be found in the quickstart archetype:

```xml
<archetype-descriptor
  xmlns="http://maven.apache.org/plugins/maven-archetype-plugin/archetype-descriptor/1.0"
  <fileSets>
    <fileSet filtered="true" packaged="true">
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
The attribute **name** tag should be the same as the **artifactId** in the archetype **pom.xml**.
The boolean attribute **partial** show if this archetype is representing a full Maven project or only parts.
The **requiredProperties**, **fileSets** and **modules** tags represent the different parts of the project:

- **<requiredProperties>** : List of required properties to generate a project from this archetype
- **<fileSets>** : File sets definition
- **<modules>** : Modules definition

At this point one can only specify individual files to be created but not empty directories.

Thus the quickstart archetype shown above defines the following directory structure:

```
archetype
|-- pom.xml
 `-- src
    `-- main
       `-- resources
          |-- META-INF
          |   `-- maven
          |       `-- archetype-metadata.xml
       `-- archetype-resources
        `-- pom.xml
     `-- src
        |-- main
        |   `-- java
        |       `-- App.java
        `-- test
             `-- JavaTest.java
```

### 30.1.3 3. Create the prototype files and the prototype **pom.xml**

The next component of the archetype to be created is the prototype **pom.xml**. Any **pom.xml** will do, just don't forget to set **artifactId** and **groupId** as variables (`${artifactId}`/`${groupId}`).

An example for a prototype **pom.xml** is:

```xml
    <modelVersion>4.0.0</modelVersion>
    <groupId>${groupId}</groupId>
    <artifactId>${artifactId}</artifactId>
    <version>${version}</version>
    <packaging>jar</packaging>
</project>
```
30.1.4 4. Install the archetype and run the archetype plugin

Now you are ready to install the archetype:

```bash
mvn install
```

Now that you have created an archetype, you can try it on your local system by using the following command. In this command, you need to specify the full information about the archetype you want to use (its groupId, its artifactId, its version) and the information about the new project you want to create (artifactId and groupId). Don't forget to include the version of your archetype (if you don't include the version, you archetype creation may fail with a message that version:RELEASE was not found)

```bash
mvn archetype:generate                                    
  -DarchetypeGroupId=<archetype-groupId>                  
  -DarchetypeArtifactId=<archetype-artifactId>            
  -DarchetypeVersion=<archetype-version>                  
  -DgroupId=<my.groupid>                                  
  -DartifactId=<my-artifactId>
```

Once you are happy with the state of your archetype, you can deploy (or submit it to Maven Central) it as any other artifact and the archetype will then be available to any user of Maven.

30.1.5 Alternative way to start creating your Archetype

Instead of manually creating the directory structure needed for an archetype, simply use

```bash
mvn archetype:generate
  -DgroupId=[your project's group id]
  -DartifactId=[your project's artifact id]
  -DarchetypeGroupId=org.apache.maven.archetypes
  -DarchetypeArtifactId=maven-archetype-archetype
```

Afterwhich, you can now customize the contents of the archetype-resources directory, and archetype-metadata.xml, then, proceed to Step#4 (Install the archetype and run the archetype plugin).
31 Relocation of Artifacts

31.1 Guide to relocation
Sometimes it is necessary to relocate artifacts in the repository. One example of that is when a project moves from one groupId to a different groupId.

Making changes to the repository can have far reaching consequences. So it is best to get it right the first time, hence this guide.

2020 rework in progress, see discussion on dev mailing list, still need analysis of issues, definition of improvements, and of course implementation...

31.1.1 How to relocate a Maven 2 artifact to a different groupId
The goal of the example below is for the foo project to relocate its groupId from bar to org.bar.

31.1.1.1 Working on past versions
1. Copy all foo-related files from /bar/foo/ in your Maven 2 repository to a temporary location.
2. Change the groupId to org.bar in all foo-related POM files in the temporary location.
3. Copy all files from the temporary location to /org/bar/foo/ in your Maven 2 repository.
4. Create a minimal Maven 2 POM file for every old release of foo in your Maven 2 repository. The POM files only need to include groupId, artifactId, version and the relocation section.

**Note:** Before you replace your old POM files in /bar/foo/ with these minimal POM files, make sure you have made backups!

The minimal POM file might look like this for version 1.0 of foo:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>bar</groupId>
  <artifactId>foo</artifactId>
  <version>1.0</version>
  <distributionManagement>
    <relocation>
      <groupId>org.bar</groupId>
    </relocation>
  </distributionManagement>
</project>
```

In this case we are relocating because the groupId has changed. We only need to add the element that has changed to the relocation element. For information on which elements are allowed in the relocation element, see the POM reference.

5. If your project uses MD5 or SHA1 checksums you must now create new checksums for the pom files in /bar/foo/ in your Maven 2 repository. If the POM file needs to be signed, do that as well.

6. If your project syncs with Central, you should now initiate that sync. This might happen automatically depending on your projects sync policy.

Your foo-artifacts are now available to Maven users with both the old and the new groupId. Projects using the old groupId will automatically be redirected to the new groupId and a warning telling the user to update their dependencies will be issued.
31.1.2 Releasing the next version
When the next release of foo is made, you should publish two Maven 2 POM files: first you should publish foo's POM with the new groupId org.bar.

Because data in the repository is not supposed to change, Maven doesn't download POM files that it has already downloaded. Therefore you will also need to publish a relocation POM file with the old groupId bar for the new version: this should be a minimal relocation POM (as described in step 4 above), but for the new version of foo.

For the release after that, you only need to publish a Maven POM with a groupId of org.bar, since users of the previous version have been informed of the changed groupId.

31.1.2 Examples

31.1.2.1 Apache Ant

1. has published its releases until 1.6.5 to Maven 1-compliant ant:ant coordinates (see repository content),
2. starting with 1.7.0, moved to reverse-DNS compliant Maven 2+ org.apache.ant:ant coordinates, (see repository content),
3. published one ant:ant:1.7.0 relocation POM in old groupId to advertise about the move (see repository content).

Notice that past ant:ant versions POMs (until 1.6.5) have not been modified to advertise about the move: Central POM content is not expected to be changed once published (because initial content has been downloaded many times and is not expected to be re-loaded later).

31.1.2.2 Apache POI

1. has published its releases until 3.0-alpha-3 to Maven 1-compliant poi:poi coordinates (see repository content),
2. starting with 3.0-FINAL, moved to reverse-DNS compliant Maven 2+ org.apache.poi:poi coordinates, (see repository content),
3. published poi:poi:3.0-FINAL relocation POM with jars in old groupId to advertise about the move (see repository content).
4. published poi:poi relocation POMs for 3.0.1-FINAL, 3.0.2-beta1/beta2/FINAL and 3.1-beta1/beta2/FINAL in old groupId to advertise about the move (see repository content).
32 Installing 3rd party JARs to Local Repository

32.1 Guide to installing 3rd party JARs

Occasionally, you will have 3rd party JARs that you need to put in your local repository for use in your builds, since they don't exist in any public repository like Maven Central. The JARs must be placed in the local repository in the correct place in order for it to be correctly picked up by Apache Maven.

To make this easier, and less error prone, we have provided an install-file goal in the maven-install-plugin which should make this relatively painless.

To install a JAR in the local repository use the following command:

```
mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id> -DartifactId=<artifact-id> -Dversion=<version> -Dpackaging=<packaging>
```

If there's a pom-file as well, you can install it with the following command:

```
mvn install:install-file -Dfile=<path-to-file> -DpomFile=<path-to-pomfile>
```

With version 2.5 of the maven-install-plugin, it can get even simpler: if the JAR was built by Apache Maven, it'll contain a pom.xml in a subfolder of the META-INF/ directory, which will be read by default. In that case, all you need to do is:

```
mvn org.apache.maven.plugins:maven-install-plugin:2.5.2:install-file -Dfile=<path-to-file>
```
33 Deploying 3rd party JARs to Remote Repository

33.1 Guide to deploying 3rd party JARs to remote repository

Same concept of the install:install-file goal of the maven-install-plugin where the 3rd party JAR is installed in the local repository. But this time instead to local repository the JAR will be install both in the local and remote repository.

To deploy a 3rd party JAR use the deploy:deploy-file goal under maven-deploy-plugin.

First, the wagon-provider(wagon-ftp, wagon-file, etc..) must be placed to your

\$\{maven.home\}/lib

Then execute the command:

```
mvn deploy:deploy-file -DgroupId=<group-id> \\
-DartifactId=<artifact-id> \\
-Dversion=<version> \\
-Dpackaging=<type-of-packaging> \\
-Dfile=<path-to-file> \\
-DrepositoryId=<id-to-map-on-server-section-of-settings.xml> \\
-Durl=<url-of-the-repository-to-deploy>
```

33.1.1 Deploying a 3rd party JAR with a generic POM

By default, deploy:deploy-file generates a generic POM(.pom) to be deploy together with the 3rd party JAR. To disable this feature we should set the generatePOM argument to false.

```
-DgeneratePom=false
```

33.1.2 Deploying a 3rd party JAR with a customized POM

If a POM is already existing for the 3rd Party JAR and you want to deploy it together with the JAR we should use the pomFile argument of the deploy-file goal. See sample below.

```
mvn deploy:deploy-file -DpomFile=<path-to-pom> \\
-Dfile=<path-to-file> \\
-DrepositoryId=<id-to-map-on-server-section-of-settings.xml> \\
-Durl=<url-of-the-repository-to-deploy>
```

Note that groupId, artifactId, version and packaging arguments are not included here because deploy-file goal will get these information from the given POM.

33.1.3 Deploying Source Jars

To deploy a 3rd party source jar, packaging should be set to java-source, and generatePom should be set to false.
34 Remote repository access through authenticated HTTPS

34.1 Guide to Remote repository access through authenticated HTTPS

This document describes how to configure Maven to access a remote repository that sits behind an HTTPS server which requires client authentication with certificates.

34.1.1 The problem

There is a maven repository at https://my.server.com/maven. This server only serves clients authenticated through SSL protocol by a valid certificate signed by an approved certificate authority's certificate which we call the CACert. In the simplest case where the server is used internally by an identified community of users (e.g. corporate intranet), the server's certificate is the certificate authority as the server is used only internally.

So we assume that we have access to the trusted certificate in X.509 format stored in a file named:

/somewhere/in/filesystem/CACert.cert

The client's certificate has been issued by some means not described in this document in PKCS#12 format, which is the format that is accepted by browsers (at least Firefox and Internet Explorer) for import into their keystore. This file is named:

/home/directory/mycertificate.p12

and we assume it is accessible when launching maven. This file contains the client's private key which may be very sensitive information so it is secured by a password:

CertPwD

The remote repository is referenced either through the pom.xml file:


34.1.2 The solution

For maven to use this repository, we should take the following steps:

1. Create a store to hold the server's certificate using Oracle's keytool,
2. Define properties to be used by HttpClient for finding keys and certificate

34.1.2.1 Storing certificate

The following command line imports the certificate authority's certificate into a JKS formatted key store named trust.jks, the trust store.

```
$> keytool -v -alias mavensrv -import \\
    -file /somewhere/in/filesystem/CACert.cert \\
    -keystore trust.jks
```

Enter keystore password:

Owner: ....
Issuer: ....
Serial number: ....
Certificate fingerprints:
  MD5: .......
  SHA1: .......
Trust this certificate? [no]: yes
Certificate was added to keystore

[Storing trust.jks]

Note that it should be possible to import a full chain of certificates with only one root certificate being trusted but the author did not test it.

34.1.2.2 Setting properties
The following properties must be set at start of maven to be accessible when HttpClient starts up.

- **javax.net.ssl.trustStore**
  - the path to the keystore where trusted certificates are stored

- **javax.net.ssl.trustStoreType**
  - the type of storage for this store, maybe either [jks](default) or [pkcs12]

- **javax.net.ssl.trustStorePassword**
  - the password protecting the store

- **javax.net.ssl.keyStore**
  - the path to the keystore where user's private key is stored

- **javax.net.ssl.keyStoreType**
  - the type of storage for this store, maybe either [jks](default) or [pkcs12]

- **javax.net.ssl.keyStorePassword**
  - the password protecting the store

Not all the properties must be set depending of your precise settings: type of store may left to default, password may be empty.

They may be set either on maven's command-line, in `.mavenrc` file or in `MAVEN_OPTS` environment variable. For the setting defined in this document, here is an example `.mavenrc` file:

```
MAVEN_OPTS="-Xmx512m -Djavax.net.ssl.trustStore=trust.jks \
-Djavax.net.ssl.trustStorePassword=  \
-Djavax.net.ssl.keyStore=/home/directory/mycertificate.p12 \
-Djavax.net.ssl.keyStoreType=pkcs12 \
-Djavax.net.ssl.keyStorePassword=XXXXXX"
```

34.1.3 Links
The following links may be useful in understanding SSL infrastructure management in Java:

- [HttpClient's SSL guide](#)
35 Creating Assemblies

35.1 Guide to creating assemblies

The assembly mechanism in Maven provides an easy way to create distributions using an assembly descriptor and dependency information found in your POM. In order to use the assembly plug-in you need to configure the assembly plug-in in your POM and it might look like the following:

```xml
<project>
  <parent>
    <artifactId>maven</artifactId>
    <groupId>org.apache.maven</groupId>
    <version>2.0-beta-3-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven-embedder</artifactId>
  <name>Maven Embedder</name>
  <version>2.0-beta-3-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.5.3</version>
        <configuration>
          <descriptor>${project.basedir}/src/assembly/dep.xml</descriptor>
        </configuration>
        <executions>
          <execution>
            <id>create-archive</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

You'll notice that the assembly descriptor is located in `{project.basedir}/src/assembly` which is the standard location for assembly descriptors.

35.1.1 Creating a binary assembly

This is the most typical usage of the assembly plugin where you are creating a distribution for standard use.

```xml
  <id>bin</id>
</assembly>
```
<!-- TODO: a jarjar format would be better -->
<id>dep</id>
<formats>
  <format>jar</format>
</formats>
<includeBaseDirectory>false</includeBaseDirectory>
<fileSets>
  <fileSet>
    <directory>${project.build.directory}</directory>
    <outputDirectory>/</outputDirectory>
    <includes>
      <include>*.jar</include>
    </includes>
  </fileSet>
  <fileSet>
    <directory>${project.build.directory}/site</directory>
    <outputDirectory>docs</outputDirectory>
  </fileSet>
</fileSets>
<dependencySets>
  <dependencySet>
    <outputDirectory>/</outputDirectory>
    <unpack>true</unpack>
  </dependencySet>
</dependencySets>
</assembly>

You can use a manually defined assembly descriptor as mentioned before but it is simpler to use the pre-defined assembly descriptor `bin` in such cases.

How to use such pre-defined assembly descriptors is described in the documentation of maven-assembly-plugin.
If you like to create a source distribution package the best solution is to use the pre-defined assembly descriptor `src` for such purposes.

```xml
  <id>src</id>
  <formats>
    <format>tar.gz</format>
    <format>tar.bz2</format>
    <format>zip</format>
  </formats>
  <fileSets>
    <fileSet>
      <directory>${project.basedir}</directory>
      <useDefaultExcludes>true</useDefaultExcludes>
      <includes>
        <include>README*</include>
        <include>LICENSE*</include>
        <include>NOTICE*</include>
        <include>pom.xml</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.sourceDirectory}/src</directory>
      <useDefaultExcludes>true</useDefaultExcludes>
    </fileSet>
  </fileSets>
</assembly>
```

You can now create the defined distribution packages via command line like this:

```bash
mvn package
```
36 Configuring Archive Plugins

36.1 Guide to Configuring Archive Plugins

Many Java archive generating plugins accept the `archive` configuration element to customize the generation of the archive. In the standard Maven Plugins, this includes the `jar`, `war`, `ejb`, `ear` and `assembly` plugins.

36.1.1 Disabling Maven Meta Information

By default, Maven generated archives include the `META-INF/maven` directory, which contains the `pom.xml` file used to build the archive, and a `pom.properties` file that includes some basic properties in a small, easier to read format.

To disable the generation of these files, include the following configuration for your plugin (in this example, the WAR plugin is used):

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <archive>
            <addMavenDescriptor>false</addMavenDescriptor>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

36.1.2 Configuring the Manifest

The archive configuration also accepts manifest configuration. See Guide to Working with Manifests for more information.
37 Configuring Maven

37.1 Configuring Maven

Maven configuration occurs at 3 levels:

- **Project** - most static configuration occurs in `pom.xml`
- **Installation** - this is configuration added once for a Maven installation
- **User** - this is configuration specific to a particular user

The separation is quite clear - the project defines information that applies to the project, no matter who is building it, while the others both define settings for the current environment.

**Note:** the installation and user configuration cannot be used to add shared project information - for example, setting `<organization>` or `<distributionManagement>` company-wide.

For this, you should have your projects inherit from a company-wide parent `pom.xml`.

You can specify your user configuration in `${user.home}/.m2/settings.xml`. A full reference to the configuration file is available. This section will show how to make some common configurations. Note that the file is not required - defaults will be used if it is not found.

37.1.1 Configuring your Local Repository

The location of your local repository can be changed in your user configuration. The default value is `${user.home}/.m2/repository/`.

```xml
<settings>
  ...
  <localRepository>/path/to/local/repo/</localRepository>
  ...
</settings>
```

**Note:** The local repository must be an absolute path.

37.1.2 Configuring a Proxy

Proxy configuration can also be specified in the settings file.

For more information, see the Guide to using a Proxy.

37.1.3 Configuring Parallel Artifact Resolution

By default, Maven 2.1.0+ will download up to 5 artifacts (from different groups) at once. To change the size of the thread pool, start Maven using `-Dmaven.artifact.threads`. For example, to only download single artifacts at a time:

```
mvn -Dmaven.artifact.threads=1 verify
```

You may wish to set this option permanently, in which case you can use the `MAVEN_OPTS` environment variable. For example:

```
export MAVEN_OPTS=-Dmaven.artifact.threads=3
```

37.1.4 Security and Deployment Settings

Repositories to deploy to are defined in a project in the `<distributionManagement>` section. However, you cannot put your username, password, or other security settings in that project. For that
reason, you should add a server definition to your own settings with an id that matches that of the deployment repository in the project.

In addition, some repositories may require authorization to download from, so the corresponding settings can be specified in a server element in the same way.

Which settings are required will depend on the type of repository you are deploying to. As of the first release, only SCP deployments and file deployments are supported by default, so only the following SCP configuration is needed:

```xml
<settings>
  ...
  <servers>
    <server>
      <id>repo1</id>
      <username>repouser</username>
      <!-- other optional elements:
      <password>my_login_password</password>
      <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa)
      <passphrase>my_key_passphrase</passphrase>
      -->
    </server>
    ...
  </servers>
  ...
</settings>
```

To encrypt passwords in these sections, refer to Encryption Settings.

### 37.1.5 Using Mirrors for Repositories

Repositories can be declared inside a project, which means that if you have your own custom repositories, those sharing your project easily get the right settings out of the box. However, you may want to use an alternative mirror for a particular repository without changing the project files. Refer to Guide to Mirror Settings for more details.

### 37.1.6 Profiles

Repository configuration can also be put into a profile. You can have multiple profiles, with one set to active so that you can easily switch environments. Read more about profiles in Introduction to Build Profiles.

### 37.1.7 Optional configuration

Maven will work for most tasks with the above configuration, however if you have any environmental specific configuration outside of individual projects then you will need to configure settings. The following sections refer to what is available.

#### 37.1.7.1 Settings

Maven has a settings file located in the Maven installation and/or user home directory that configure environmental specifics such as:

- HTTP proxy server
- repository manager location
- server authentication and passwords
• other configuration properties
For information on this file, see the Settings reference

37.1.7.2 Security
As of Maven 2.1.0+, you can encrypt passwords in your settings file, however you must first configure a master password. For more information on both server passwords and the master password, see the Guide to Password Encryption.

37.1.7.3 Toolchains
As of Maven 2.0.9+, you can build a project using a specific version of JDK independent from the one Maven is running with. For more information, see the Guide to Using Toolchains.
38 Mirror Settings

38.1 Using Mirrors for Repositories

With Repositories you specify from which locations you want to download certain artifacts, such as dependencies and maven-plugins. Repositories can be declared inside a project, which means that if you have your own custom repositories, those sharing your project easily get the right settings out of the box. However, you may want to use an alternative mirror for a particular repository without changing the project files.

Some reasons to use a mirror are:

- There is a synchronized mirror on the internet that is geographically closer and faster
- You want to replace a particular repository with your own internal repository which you have greater control over
- You want to run a repository manager to provide a local cache to a mirror and need to use its URL instead

To configure a mirror of a given repository, you provide it in your settings file (${user.home}/.m2/settings.xml), giving the new repository its own id and url, and specify the mirrorOf setting that is the ID of the repository you are using a mirror of. For example, the ID of the main Maven Central repository included by default is central, so to use the different mirror instance, you would configure the following:

```xml
<settings>
  ...
  <mirrors>
    <mirror>
      <id>other-mirror</id>
      <name>Other Mirror Repository</name>
      <url>https://other-mirror.repo.other-company.com/maven2</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

Note that there can be at most one mirror for a given repository. In other words, you cannot map a single repository to a group of mirrors that all define the same <mirrorOf> value. Maven will not aggregate the mirrors but simply picks the first match. If you want to provide a combined view of several repositories, use a repository manager instead.

The settings descriptor documentation can be found on the Maven Local Settings Model Website.

Note: The official Maven repository is at https://repo.maven.apache.org/maven2 hosted by the Sonatype Company and is distributed worldwide via CDN.

A list of known mirrors is available in the Repository Metadata. These mirrors may not have the same contents and we don't support them in any way.

38.2 Using A Single Repository

You can force Maven to use a single repository by having it mirror all repository requests. The repository must contain all of the desired artifacts, or be able to proxy the requests to other repositories. This setting is most useful when using an internal company repository with a Maven Repository Manager to proxy external requests.
To achieve this, set `mirrorOf` to `*`.

**Note:** This feature is only available in Maven 2.0.5+.

```xml
<settings>
  ...
  <mirrors>
    <mirror>
      <id>internal-repository</id>
      <name>Maven Repository Manager running on repo.mycompany.com</name>
      <url>http://repo.mycompany.com/proxy</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

### 38.3 Advanced Mirror Specification

A single mirror can handle multiple repositories. This is typically used in conjunction with a repository manager, that gives easy centralised configuration of the list of repositories behind.

The syntax:
- `*` matches all repo ids.
- `external:*` matches all repositories except those using localhost or file based repositories. This is used when you want to exclude redirecting repositories that are defined for Integration Testing.
- since Maven 3.8.0, `external:http:*` matches all repositories using HTTP except those using localhost.
- multiple repositories may be specified using a comma as the delimiter
- an exclamation mark may be used in conjunction with one of the above wildcards to exclude a repository id

Be careful not to include extra whitespace around identifiers or wildcards in comma separated lists. For example, a mirror with `<mirrorOf>` set to `!repo1, *` will not mirror anything while `!repo1,*` will mirror everything but repo1.

The position of wildcards within a comma separated list of repository identifiers is not important as the wildcards defer to further processing and explicit includes or excludes stop the processing, overruling any wildcard match.

When you use the advanced syntax and configure multiple mirrors, the declaration order matters. When Maven looks for a mirror of some repository, it first checks for a mirror whose `<mirrorOf>` exactly matches the repository identifier. If no direct match is found, Maven picks the first mirror declaration that matches according to the rules above (if any). Hence, you may influence match order by changing the order of the definitions in the `settings.xml`.

Examples:
- `*` = everything
- `external:*` = everything not on the localhost and not file based.
- `repo,repo1` = repo or repo1
- `*,!repo1` = everything except repo1

```xml
<settings>
  ...
  <mirrors>
```
38 Mirror Settings

<mirror>
  <id>internal-repository</id>
  <name>Maven Repository Manager running on repo.mycompany.com</name>
  <url>http://repo.mycompany.com/proxy</url>
  <mirrorOf>external:*,!foo</mirrorOf>
</mirror>

<mirror>
  <id>foo-repository</id>
  <name>Foo</name>
  <url>http://repo.mycompany.com/foo</url>
  <mirrorOf>foo</mirrorOf>
</mirror>

...</mirrors>

38.4 Creating Your Own Mirror

The size of the central repository is increasing steadily. To save us bandwidth and you time, mirroring the entire central repository is not allowed. (Doing so will get you automatically banned.) Instead, we suggest you setup a repository manager as a proxy.

If you really want to become an official mirror, contact Sonatype at MVNCENTRAL with your location and we'll work to get you setup.
39 Deployment and Security Settings

39.1 Security and Deployment Settings

Repositories to deploy to are defined in a project in the `distributionManagement` section. However, you cannot put your username, password, or other security settings in that project. For that reason, you should add a server definition to your own settings with an id that matches that of the deployment repository in the project.

In addition, some repositories may require authorisation to download from, so the corresponding settings can be specified in a server element in the same way.

Which settings are required will depend on the type of repository you are deploying to. As of the first release, only SCP deployments and file deployments are supported by default, so only the following SCP configuration is needed:

```xml
<settings>
  <servers>
    <server>
      <id>repo1</id>
      <username>repouser</username>
      <!-- other optional elements: -->
      <!-- <password>my_login_password</password> -->
      <!-- <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa) -->
      <!-- <passphrase>my_key_passphrase</passphrase> -->
    </server>
  </servers>
</settings>
```

To encrypt passwords in these sections, refer to Encryption Settings.

**Note:** The settings descriptor documentation can be found on the Maven Local Settings Model Website.
40 Generating Sources

40.1 Guide to generating sources

Let's run though a short example to try and help. To generate sources you must first have a plugin that participates in the `generate-sources` phase like the ANTLR4 Maven Plugin.

So this is all fine and dandy, we have a plugin that wants to generate some sources from a Antlr4 grammar but how do we use it. You need to specify that you want to use it in your POM:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.antlr</groupId>
        <artifactId>antlr4-maven-plugin</artifactId>
        <version>4.5.3</version>
        <executions>
          <execution>
            <id>antlr</id>
            <goals>
              <goal>antlr4</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
  </project>
```

If you then type `mvn compile` Maven will walk through the lifecycle and will eventually hit the `generate-sources` phase and see you have a plugin configured that wants to participate in that phase and the ANTLR4 Maven Plugin is executed with your given configuration. Furthermore during the compile you can observe that all the generated code (from your grammar files) will automatically being compiled without supplemental configuration.
41 Working with Manifests

41.1 Guide to Working with Manifests

In order to modify the manifest of the archive produced by the packaging plug-ins you need to create a configuration for it. The definitive guide for this is the site for the Maven Archiver shared component. This component is used by all our packaging plugins.
42 Maven Classloading

42.1 Guide to Maven Classloading
This is a description of the classloader hierarchy in Maven.

42.1.1 Overview
- System Classloader
- Core Classloader
- Plugin Classloaders
- Custom Classloaders

42.1.2 1. System Classloader
Maven uses the Plexus Classworlds classloading framework with which we create our classloader graph. If you look in your \( \text{${maven.home}/boot} \) directory you will see a single JAR which is the Classworlds JAR we use to boot the classloader graph. The Classworlds JAR is the only element of the Java CLASSPATH and Classworlds then builds the other classloaders or realms in Classworlds terminology.

An Ant script like this will show the contents of the system classloader:

```xml
<target name="info">
    <echo>java.class.path=${java.class.path}</echo>
</target>
```

42.1.3 2. Core Classloader
The second classloader down the graph contains the core requirements of Maven. More precisely, the core classloader has the libraries in \( \text{${maven.home}/lib} \). In general these are just Maven libraries, e.g. instances of MavenProject belong to this classloader. We hope to further separate these in the future to just be Maven APIs and have the implementations selected at runtime as required by the system.

You can add elements to this classloader by extensions. These are loaded into the same place as \( \text{${maven.home}/lib} \) and hence are available to the Maven core and all plugins for the current project and subsequent projects (in future, we plan to remove it from subsequent projects).

42.1.4 3. Plugin Classloaders
After that, each plugin has its own classloader that is a child of Maven's core classloader. The classes in this classloader are taken from the dependencies in the plugin's dependency list.

Users can add dependencies to this classloader by adding dependencies to a plugin in the plugins/plugin section of their project pom.xml. Here is a sample of adding ant-nodeps to the plugin classloader of the Antrun Plugin and hereby enabling the use of additional/optional Ant tasks:

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>1.3</version>
    <dependencies>
        <dependency>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.3</version>
            <dependencies>
                <dependency>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
<groupId>org.apache.ant</groupId>
<artifactId>ant-nodeps</artifactId>
<version>1.7.1</version>
</dependency>
</dependencies>

Plugins can inspect their effective runtime class path via the expressions `${plugin.artifacts}` or `${plugin.artifactMap}` to have a list or map, respectively, of resolved artifacts injected from the PluginDescriptor.

Please note that the plugin classloader does neither contain the dependencies of the current project nor its build output. Instead, plugins can query the project's compile, runtime and test class path from the MavenProject in combination with the mojo annotation requiresDependencyResolution from the Mojo API Specification. For instance, flagging a mojo with @requiresDependencyResolution runtime enables it to query the runtime class path of the current project from which it could create further classloaders.

When a build plugin is executed, the thread's context classloader is set to the plugin classloader.

42.1.5 4. Custom Classloaders

Plugins are free to create further classloaders on their discretion. For example, a plugin might want to create a classloader that combines the plugin class path and the project class path.

It is important to understand that the plugin classloader cannot load classes from any of those custom classloaders. Some factory patterns require that. Here you must add the classes to the plugin classloader as shown before.
43 Using Multiple Modules in a Build

43.1 Guide to Working with Multiple Modules

(If you're working with Maven 4, please refer to the Maven 4 edition of this guide)

As seen in the introduction to the POM, Maven supports project aggregation in addition to project inheritance. This section outlines how Maven processes projects with multiple modules, and how you can work with them more effectively.

43.1.1 The Reactor

The mechanism in Maven that handles multi-module projects is referred to as the *reactor*. This part of the Maven core does the following:

- Collects all the available modules to build
- Sorts the projects into the correct build order
- Builds the selected projects in order

43.1.1.1 Reactor Sorting

Because modules within a multi-module build can depend on each other, it is important that the reactor sorts all the projects in a way that guarantees any project is built before it is required.

The following relationships are honoured when sorting projects:

- a project dependency on another module in the build
- a plugin declaration where the plugin is another module in the build
- a plugin dependency on another module in the build
- a build extension declaration on another module in the build
- the order declared in the `<modules>` element (if no other rule applies)

Note that only "instantiated" references are used - dependencyManagement and pluginManagement elements do not cause a change to the reactor sort order.

43.1.1.2 Command Line Options

No special configuration is required to take advantage of the reactor, however it is possible to customize its behavior.

The following command line switches are available:

- `--resume-from` - resumes a reactor from the specified project (e.g. when it fails in the middle)
- `--also-make` - build the specified projects, and any of their dependencies in the reactor
- `--also-make-dependents` - build the specified projects, and any that depend on them
- `--fail-fast` - the default behavior - whenever a module build fails, stop the overall build immediately
- `--fail-at-end` - if a particular module build fails, continue the rest of the reactor and report all failed modules at the end instead
- `--non-recursive` - do not use a reactor build, even if the current project declares modules and just build the project in the current directory

Refer to the Maven command line interface reference for more information on these switches.

43.1.2 More information

- Chapter 6. A Multi-module Project (Maven by Example)
44 Using Multiple Repositories

44.1 Setting up Multiple Repositories

There are two different ways that you can specify the use of multiple repositories. The first way is to specify in a POM which repositories you want to use. That is supported both inside and outside of build profiles:

```xml
<project>
  ...
  <repositories>
    <repository>
      <id>my-repo1</id>
      <name>your custom repo</name>
      <url>http://jarsm2.dyndns.dk</url>
    </repository>
    <repository>
      <id>my-repo2</id>
      <name>your custom repo</name>
      <url>http://jarsm2.dyndns.dk</url>
    </repository>
  </repositories>
  ...
</project>
```

**NOTE:** You will also get the standard set of repositories as defined in the Super POM.

The other way you can specify multiple repositories is by creating a profile in the $HOME/.m2/settings.xml or $MAVEN_HOME/conf/settings.xml file like the following:

```xml
<settings>
  ...
  <profiles>
    ...
    <profile>
      <id>myprofile</id>
      <repositories>
        <repository>
          <id>my-repo2</id>
          <name>your custom repo</name>
          <url>http://jarsm2.dyndns.dk</url>
        </repository>
      </repositories>
    </profile>
    ...
  </profiles>
  <activeProfiles>
    <activeProfile>myprofile</activeProfile>
  </activeProfiles>
  ...
</settings>
```
If you specify repositories in profiles you must remember to activate that particular profile! As you can see above we do this by registering a profile to be active in the `activeProfiles` element.

You could also activate this profile on the command line by executing the following command:

```
mvn -Pmyprofile ...
```

In fact the `-P` option will take a CSV list of profiles to activate if you wish to activate multiple profiles simultaneously.

**Note:** The settings descriptor documentation can be found on the [Maven Local Settings Model Website](https://maven.apache.org/resources/maven/).

### 44.1.1 Repository Order
Remote repository URLs are queried in the following order for artifacts until one returns a valid result:

1. **effective settings:**
   - A. Global `settings.xml`
   - B. User `settings.xml`

2. **local effective build POM:**
   - A. Local `pom.xml`
   - B. Parent POMs, recursively
   - C. Super POM

3. **effective POMs from dependency path to the artifact.**

For each of these locations, the repositories within the profiles are queried first in the order outlined at [Introduction to build profiles](https://maven.apache.org/resources/maven/).

Before downloading from a repository, _mirrors configuration_ is applied.

Effective settings and local build POM, with profile taken into account, can easily be reviewed to see their repositories order with `mvn help:effective-settings` and `mvn help:effective-pom -Dverbose`.
45 Using Proxies

45.1 Configuring a proxy

You can configure a proxy to use for some or all of your HTTP requests with Maven. The username and password are only required if your proxy requires basic authentication (note that later releases may support storing your passwords in a secured keystore - in the mean time, please ensure your settings.xml file (usually ${user.home}/.m2/settings.xml) is secured with permissions appropriate for your operating system).

The nonProxyHosts setting accepts wild cards, and each host not to proxy is separated by the | character. This matches the JDK configuration equivalent.

```xml
<settings>
   ...
   <proxies>
      <proxy>
         <id>example-proxy</id>
         <active>true</active>
         <protocol>http</protocol>
         <host>proxy.example.com</host>
         <port>8080</port>
         <username>proxyuser</username>
         <password>somepassword</password>
         <nonProxyHosts>www.google.com|*.example.com</nonProxyHosts>
      </proxy>
   </proxies>
   ...
</settings>
```

Please note that currently NTLM proxies are not supported as they have not been tested. You may be able to use the relevant system properties on JDK 1.4+ to make this work.

45.1.1 Resources

1. Settings descriptor documentation
2. Configuring Maven
46 Using the Release Plugin

46.1 Releasing

46.1.1 Introduction

The main aim of the maven-release plugin is to provide a standard mechanism to release project artifacts outside the immediate development team. The plugin provides basic functionality to create a release and to update the project's SCM accordingly.

To create a release the maven-release plugin is executed through maven in 2 stages:
1. Preparing the release.
2. Performing the release.

46.1.2 Preparing the release

The plugin will record release information into a new revision of the project's pom.xml file as well as applying SCM versioning to the project's resources.

The release:prepare goal will:
1. Verify that there are no uncommitted changes in the workspace.
2. Prompt the user for the desired tag, release and development version names.
3. Modify and commit release information into the pom.xml file.
4. Tag the entire project source tree with the new tag name.

The following example shows how to run the release:prepare goal with a Subversion SCM. The commandline example directs the plugin to locate a Subversion SCM on a local file system.

mvn release:prepare
   -Dproject.scm.developerConnection=scm:svn:file:///D:/subversion_data/repos/my_repo/my-app-example/trunk
   -DtagBase=file:///D:/subversion_data/repos/my_repo/my-app-example/tags

When using the release:prepare goal, the user must supply maven with information regarding the current location of the project's SCM. In the previous example maven was supplied with the current location of the development trunk and the new location to record tagged instances of the project.

- **project.scm.developerConnection**
  The current location of the development trunk. A valid SCM URL format appropriate to the SCM provider. The "SCM:Provider:" prefix is used to determine the provider being used.

- **tagbase**
  The new location to record a tagged release. A valid SCM URL format appropriate to the SCM provider without the "SCM:Provider:" prefix.

The previous goal parameters can be supplied while executing maven on the commandline, (as shown in the previous example) or they can be defined and maintained within the project's pom.xml file. The location of the current development trunk is defined within the pom.xml file in the following form:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Application</name>
  <url>http://app.mycompany.com</url>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
To define the `tagBase` parameter within the `pom.xml` file a `tagBase` element must be defined within a `plugins/plugin/configuration` element. The following example shows how this would look within the `pom.xml` file.

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Application</name>
  <url>http://app.mycompany.com</url>
  ...
  <scm>
    <developerConnection>scm:svn:file:///D:/subversion_data/repos/my_repo/my-app-example/trunk</developerConnection>
  </scm>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.5.2</version>
        <configuration>
          ...
          <tagBase>
            file:///D:/subversion_data/repos/my_repo/my-app-example/tags
          </tagBase>
          ...
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```
During the execution of the release:prepare goal maven will interact with the user to gather information about the current release. Maven will prompt the user for the following information:

- **A Desired SCM provider tag name.**
  This is a SCM provider specific value, in the case of the Subversion SCM provider this value is equal to the folder name that will appear under the URL provided by the the tagBase parameter.

- **A Desired project release version.**
  This value is placed in the pom.xml that will define the current release. If a development pom.xml holds a version value of 1.0-SNAPSHOT then the release version would be 1.0. This is not enforced and can be a value appropriate to yourself or a company environment.

- **A New development version.**
  This value is the placed in the next revision of the pom.xml file used for continuing development. If the current release represented version 1.0 then an appropriate value could be 2.0-SNAPSHOT. The SNAPSHOT designator is required to prepare and perform future releases. This value is then committed in the next development revision of the pom.xml file.

After maven has been supplied with the required information the maven-release plugin will interact with the project's SCM and define a release to be extracted and deployed. At the same time the project's development trunk is updated allowing developers to continue with further modifications that will be included within future releases.

### 46.1.3 Performing the release

The plugin will extract file revisions associated with the current release. Maven will compile, test and package the versioned project source code into an artifact. The final deliverable will then be released into an appropriate maven repository.

The release:perform goal will:

1. Extract file revisions versioned under the new tag name.
2. Execute the maven build lifecycle on the extracted instance of the project.
3. Deploy the versioned artifacts to appropriate local and remote repositories.

The following example shows how to run the release:perform goal from the commandline.

```sh
mvn release:perform
```

The release:perform goal requires a file called release.properties to be present within the project root directory. The release.properties file is constructed during the execution of the release:prepare goal and contains all the information needed to locate and extract the correctly tagged version of the project. Shown below is an example of the contents that can appear within an instance of the release.properties file.

**Note:** The location of the release.properties file is under review and could be moved to the target directory.

```none
#Generated by Release Plugin on: Sat Nov 12 11:22:33 GMT 2005
#Sat Nov 12 11:22:33 GMT 2005
maven.username=myusername
checkpoint.transformed-pom-for-release=OK
```
The `release.properties` file is created while preparing the release. After performing the release the file remains within the project root directory until the maven user deletes it. The `release.properties` file can be given to any developer within the team and by simply executing the `release:perform` goal can create and deploy a new instance of the project artifact time and again.

During the execution of the `release:perform` goal the entire maven build lifecycle is executed on the project. The tagged project source code is extracted, compiled, tested, documented and deployed. An instance of the release artifact is deployed to the machine's local repository. Another instance of the release can be deployed to a remote repository by configuring the `distributionManagement` element within the `pom.xml` file.

The following is an example of how a distributionManagement element can be configured within a project `pom.xml` file.

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Application</name>
  <url>http://app.mycompany.com</url>
  ...
  <distributionManagement>
    <repository>
      <id>myRepoId</id>
      <name>myCompanyRepository</name>
      <url>ftp://repository.mycompany.com/repository</url>
    </repository>
  </distributionManagement>
  ...
</project>
```

If the distributionManagement element is not configured within the `pom.xml` file then the deployment of the artifact will fail. Maven will report a failure back to the user for the execution of the maven-deploy plugin. Please refer maven documentation and additional mini guides for the use of the maven-deploy plugin.

The following deliverables are created and deployed to local and remoted repositories after the execution of the `release:perform` goal has finished.

- **artifact id- version.jar**
  The binaries for the current release of the project.
- **artifact id- version-javadoc.jar**
  The javadoc explaining the current functionality of the classes within the current release.
• *artifact id*-version-source.jar
  The source code revisions used to build the current release of the project.
• *artifact id*-version.pom
  The contents of the *pom.xml* file used to create the current release of the project.

### 46.1.4 Troubleshooting

46.1.4.1 I get a “The authenticity of host ‘*host*’ can’t be established.” error and the build hangs
This is because your ~user/.ssh/known_hosts file doesn't have the host listed. You'd normally get a prompt to add the host to the known host list but Maven doesn't propagate that prompt. The solution is to add the host the *known_hosts* file before executing Maven. On Windows, this can be done by installing an OpenSSH client (for example SSHWindows), running `ssh <host>` and accepting to add the host.

46.1.4.2 The site deploy goal hangs
First, this means that you have successfully deployed the artifacts to the remote repo and that it’s only the site deployment that is now an issue. Stop your build, cd to `target/checkout` and run the build again by executing `mvn site:deploy`. You should see a prompt asking you to enter a password. This happens if your key is not in the authorized keys on the server.
47.1 Guide to using Ant with Maven

The example above illustrates how to bind an ant script to a lifecycle phase. You can add a script to each lifecycle phase, by duplicating the execution/ section and specifying a new phase.

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>my-test-app</artifactId>
  <groupId>my-test-group</groupId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.7</version>
        <executions>
          <execution>
            <phase>generate-sources</phase>
            <configuration>
              <tasks>
                <!--
                Place any ant task here. You can add anything you can add between <target> and </target> in a build.xml.
                -->
              </tasks>
            </configuration>
            <goals>
              <goal>run</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

So a concrete example would be something like the following:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>my-test-app</artifactId>
  <groupId>my-test-group</groupId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.7</version>
        <executions>
          <execution>
            <phase>generate-sources</phase>
            <configuration>
              <tasks>
                <!--
                Place any ant task here. You can add anything you can add between <target> and </target> in a build.xml.
                -->
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```
<phase>generate-sources</phase>
<configuration>
	<tasks>
		<exec
dir="${project.basedir}"  
executable="${project.basedir}/src/main/sh/do-something.sh"  
failonerror="true">
			<arg line="arg1 arg2 arg3 arg4" />
		</exec>
	</tasks>
</configuration>
<goals>
	<goal>run</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
48 Using Modello

48.1 Guide to using Modello

Modello is a tool for generating resources from a simple model. From a simple model you can generate things like:

- Java sources
- XML serialization code for the model
- XML deserialization code for model
- Model documentation
- XSD

A typical modello model looks like the following:

```xml
<model xmlns="https://codehaus-plexus.github.io/MODELLO/1.8.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <id>archetype-descriptor</id>
  <name>ArchetypeDescriptor</name>
  <description><![CDATA[
  <p>This is a reference for the Archetype descriptor used to describe archetypes’s metadata.</p>
  <p>The metadata about an archetype is stored in the <code>archetype-metadata.xml</code> file in the <code>META-INF/maven</code> directory of its jar file.</p>]]></description>
  <defaults>
    <default>
      <key>package</key>
      <value>org.apache.maven.archetype.metadata</value>
    </default>
  </defaults>
  <classes>
    <class rootElement="true" xml.tagName="archetype-descriptor">
      <name>ArchetypeDescriptor</name>
      <version>1.0.0+</version>
      <superClass>AbstractArchetypeDescriptor</superClass>
      <fields>
        <field xml.attribute="true">
          <name>name</name>
          <version>1.0.0+</version>
          <type>String</type>
          <required>true</required>
          <description>Name of the Archetype, that will be displayed to the user when choosing it.</description>
        </field>
        <field xml.attribute="true">
          <name>partial</name>
          <version>1.0.0+</version>
          <type>boolean</type>
          <required>false</required>
          <description>Is this archetype representing a full Maven project or only parts?</description>
        </field>
      </fields>
    </class>
  </classes>
</model>
```
<field>
  <name>requiredProperties</name>
  <version>1.0.0+</version>
  <description>List of required properties to generate a project from this archetype.
  <association>
    <type>RequiredProperty</type>
    <multiplicity>*</multiplicity>
  </association>
</field>
</fields>

</class>
<class>
  <name>ModuleDescriptor</name>
  <version>1.0.0+</version>
  <superClass>AbstractArchetypeDescriptor</superClass>
  <fields>
    <field xml.attribute="true">
      <name>id</name>
      <version>1.0.0+</version>
      <type>String</type>
      <required>true</required>
      <description>The module's artifactId.</description>
    </field>
    <field xml.attribute="true">
      <name>dir</name>
      <version>1.0.0+</version>
      <type>String</type>
      <required>true</required>
      <description>The module's directory.</description>
    </field>
    <field xml.attribute="true">
      <name>name</name>
      <version>1.0.0+</version>
      <type>String</type>
      <required>true</required>
      <description>The module's name.</description>
    </field>
  </fields>
</class>
<class>
  <name>AbstractArchetypeDescriptor</name>
  <version>1.0.0+</version>
  <fields>
    <field xdoc.separator="blank">
      <name>fileSets</name>
      <version>1.0.0+</version>
      <association>
        <type>FileSet</type>
        <multiplicity>*</multiplicity>
      </association>
      <required>true</required>
      <description>File sets definition.</description>
    </field>
  </fields>
</class>
<field xdoc.separator="blank">
  <name>modules</name>
  <version>1.0.0+</version>
  <association>
    <type>ModuleDescriptor</type>
    <multiplicity>*</multiplicity>
  </association>
  <required>false</required>
  <description>Modules definition.</description>
</field>
</fields>
</class>
<class name="FileSet">
  <version>1.0.0+</version>
  <description><![CDATA[
A fileset defines the way the project's files located in the jar file are used by the Archetype Plugin to generate a project.
If file or directory name contains <code>__i__property</i> pattern, it is replaced with corresponding property value.
]]></description>
  <fields>
    <field xml.attribute="true">
      <name>filtered</name>
      <version>1.0.0+</version>
      <type>boolean</type>
      <required>false</required>
      <description><![CDATA[
Filesets can be filtered, which means the selected files will be used as Velocity templates.
They can be non-filtered, which means the selected files will be copied without modification.
]]></description>
    </field>
    <field xml.attribute="true">
      <name>packaged</name>
      <version>1.0.0+</version>
      <type>boolean</type>
      <required>false</required>
      <description>Filesets can be packaged, which means the selected files will be generated in a directory structure that is prepended by the package property. They can be non-packaged, which means the selected files will be copied without that prepend.</description>
    </field>
    <field xml.attribute="true">
      <name>encoding</name>
      <version>1.0.0+</version>
      <type>String</type>
      <required>false</required>
      <description>Encoding to use when filtering content.</description>
    </field>
    <field>
      <name>directory</name>
      <version>1.0.0+</version>
      <type>String</type>
      <required>true</required>
      <description>The directory where the files will be searched for, which is also the.
    </field>
  </fields>
</class>
Using Modello

project's files will be generated.<description>
</field>

<field>
  <name>includes</name>
  <version>1.0.0+</version>
  <association>
    <type>String</type>
    <multiplicity>*</multiplicity>
  </association>
  <required>false</required>
  <description>Inclusion definition "à la" Ant.</description>
</field>

<field>
  <name>excludes</name>
  <version>1.0.0+</version>
  <association>
    <type>String</type>
    <multiplicity>*</multiplicity>
  </association>
  <required>false</required>
  <description>Exclusion definition "à la" Ant.</description>
</field>

</fields>
<codeSegments>
<codeSegment>
  <code><![CDATA[
public String toString()
{
    return
getDirectory() + " (" + ( isFiltered() ? "Filtered" : "Copied" ) + "-
+ ( isPackaged() ? "Packaged" : "Flat" ) + ") [" + org.codehaus.plexus.util.StringUtils.join( getIncludes().iterator(), ", ") + " -- " + org.codehaus.plexus.util.StringUtils.join( getExcludes().iterator(), ", ") + "];"
}
]]></code>
</codeSegment>
</codeSegments>
</class>

<field xml.attribute="true">
  <name>key</name>
  <version>1.0.0+</version>
  <type>String</type>
  <required>true</required>
</field>
To utilize Modello, you would configure the modello-maven-plugin something like the following
where you want to generate the Java sources for the model, the xpp3 serialization code and the xpp3
deserialization code (see  modello-plugin-xpp3  for more details):

```xml
<project>...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.modello</groupId>
        <artifactId>modello-maven-plugin</artifactId>
        <version>1.8.3</version>
        <executions>
          <execution>
            <goals>
              <!-- Generate the xpp3 reader code -->
              <goal>xpp3-reader</goal>
              <!-- Generate the xpp3 writer code -->
              <goal>xpp3-writer</goal>
              <!-- Generate the Java sources for the model itself -->
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```
...<\project>
49 Using Extensions

49.1 Using Extensions

Extensions are a way to add libraries to Core Classloader.

Extensions are typically used to enable Wagon providers, used for the transport of artifact between repositories, and plug-ins which provide lifecycle enhancements.

49.1.1 Wagon providers

```xml
<project>
  ...
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
        <artifactId>wagon-ftp</artifactId>
        <version>2.10</version>
      </extension>
    </extensions>
  </build>
  ...
</project>
```

49.1.2 Plug-ins which provide lifecycle enhancements

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          ...
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```
50 Building For Different Environments with Maven

50.1 Building For Different Environments

Building the same artifact for different environments has always been an annoyance. You have multiple environments, for instance test and production servers or, maybe a set of servers that run the same application with different configurations. In this guide I'll explain how you can use profiles to build and package artifacts configured for specific environments. See Introduction to Build Profiles for a more in-depth explanation of the profile concept.

Note:

- This guide assume that you have basic Maven knowledge.
- It will show a way to configure Maven to solve simple configuration set-ups only. By simple configuration set-up I mean cases where you only have a single file or a small set of files that vary for each environment. There are other and better ways to handle two and many-dimensional configuration issues.

This example assume the use of the Standard Directory Layout.

pom.xml

src/
main/
  java/
  resources/
  test/
  java/

Under src/main/resources there are three files:

- environment.properties - This is the default configuration and will be packaged in the artifact by default.
- environment.test.properties - This is the variant for the test environment.
- environment.prod.properties - This is basically the same as the test variant and will be used in the production environment.

In the project descriptor, you need to configure the different profiles. Only the test profile is showed here.

```xml
<profiles>
  <profile>
    <id>test</id>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-antrun-plugin</artifactId>
          <executions>
            <execution>
              <phase>test</phase>
              <goals>
                <goal>run</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
      <configuration>
        <tasks>
          <delete file="${project.build.outputDirectory}/environment.properties"/>
          <copy file="src/main/resources/environment.test.properties"/>
        </tasks>
      </configuration>
    </build>
  </profile>
</profiles>
```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>

<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
      <configuration>
        <classifier>test</classifier>
      </configuration>
    </execution>
  </executions>
</plugin>

Three things are configured in this snippet:

1. It configures the antrun plugin to execute the run goal in the test phase where it will copy the `environment.test.properties` file to `environment.properties`.

2. It will configure the test plugin to skip all tests when building the test and production artifacts. This is useful as you probably don't want to run tests against the production system.

3. It configures the JAR plugin to create an "attached" JAR with the "test" classifier.

To activate this profile execute `mvn -Ptest install` and Maven will execute the steps in the profile in addition to the normal steps. From this build you will get two artifacts, "foo-1.0.jar" and "foo-1.0-test.jar". These two jars will be identical.

50.2 Caveats

- Currently Maven doesn't allow a project build to only produce attached artifacts. (i.e. it has to produce a "main" artifact as well) This results in two equal JARs being packaged and installed. The JAR plugin probably should also get improved support for this case to that two different output directories will be used as the basis for building the JAR.

- The usage of the delete task might seem a bit odd but is required to make sure that the copy task actually will copy the file. The copy task will look at the timestamps of the source and destination files, only when copying the files it won't know that the actually source file might be different than the last time it was executed.
• After the build the test configuration will be in target/classes and won't be overridden because the resources plugin uses the same timestamp checking, so you should always do a clean after executing Maven with a profile.
• For the reasons given above it's imperative that you only build an artifact for a single environment in a single execution at a time and that you execute "mvn clean" whenever you change the profile switches. If not, you might get artifacts with a mixed set of configuration files.

50.3 Resources

1. Introduction to Build Profiles
2. Standard Directory Layout
51 Using Toolchains

51.1 Guide to Using Toolchains

51.1.1 What is Toolchains?

The Maven Toolchains provide a way for plugins to discover what JDK (or other tools) are to be used during the build, without the need to configure them in each plugin nor in every pom.xml, or forcing a precise location among every machine building the project.

With Maven Toolchains applied to JDK toolchain, a project can now be built using a specific version of JDK independent from the one Maven is running with. Think how JDK versions can be set in IDEs like IDEA, NetBeans and Eclipse, or how you can compile with an older JDK from Maven running with a recent one.

51.1.1.1 Prerequisites

Toolchains will only work in Maven 2.0.9 and higher versions. For more details about its design and implementation, please see Toolchains.

Below are some plugins which are toolchain-aware, with the toolchain-type used:

<table>
<thead>
<tr>
<th>Toolchain type</th>
<th>Plugin</th>
<th>Starting with</th>
<th>Hosted at</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdk</td>
<td>maven-compiler-plugin</td>
<td>2.1</td>
<td>Apache Maven</td>
</tr>
<tr>
<td>jdk</td>
<td>maven-jarsigner-plugin</td>
<td>1.3</td>
<td>Apache Maven</td>
</tr>
<tr>
<td>jdk</td>
<td>maven-javadoc-plugin</td>
<td>2.5</td>
<td>Apache Maven</td>
</tr>
<tr>
<td>jdk</td>
<td>maven-pmd-plugin</td>
<td>3.14.0</td>
<td>Apache Maven</td>
</tr>
<tr>
<td>jdk</td>
<td>maven-surefire-plugin</td>
<td>2.5</td>
<td>Apache Maven</td>
</tr>
<tr>
<td>jdk</td>
<td>animal-sniffer-maven-plugin</td>
<td>1.3</td>
<td>MojoHaus</td>
</tr>
<tr>
<td>jdk</td>
<td>cassandra-maven-plugin</td>
<td>0.7.0-1</td>
<td>MojoHaus</td>
</tr>
<tr>
<td>jdk</td>
<td>exec-maven-plugin</td>
<td>1.1.1</td>
<td>MojoHaus</td>
</tr>
<tr>
<td>jdk</td>
<td>jdiff-maven-plugin</td>
<td>1.0-beta-1-SNAPSHOT</td>
<td>MojoHaus</td>
</tr>
<tr>
<td>jdk</td>
<td>keytool-maven-plugin</td>
<td>1.4</td>
<td>MojoHaus</td>
</tr>
<tr>
<td>protobuf</td>
<td>maven-protoc-plugin</td>
<td>0.3.2</td>
<td>github</td>
</tr>
</tbody>
</table>

51.1.2 Using Toolchains in Your Project

There are two essential components that you need to configure in order to use toolchains:

1. the maven-toolchains-plugin in your project POM,
2. the toolchains.xml file on the building machine.

The maven-toolchains-plugin is the one that sets the toolchain to be used by the toolchain-aware plugins in your project.

For example, you want to use a different JDK version to build your project than the version used to run Maven, you can configure the version you want to use via this plugin as shown in the pom.xml below:

```xml
<plugins>
  ...
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-toolchains-plugin</artifactId>
    <version>1.1</version>
    <executions>
      <execution>
        <goals>
          <goal>toolchain</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <toolchains>
        <jdk>
          <version>1.5</version>
          <vendor>sun</vendor>
        </jdk>
      </toolchains>
    </configuration>
  </plugin>
  ...
</plugins>
```

As you can see in the example above, a JDK toolchain with <version> "1.5" and <vendor> "sun" is to be used. Now how does the plugin know where this JDK is installed? This is where the toolchains.xml file comes in.

The toolchains.xml file (see below) is the configuration file where you set the installation paths of your toolchains. This file should be put in your ${user.home}/.m2 directory. When the maven-toolchains-plugin executes, it looks for the toolchains.xml file, reads it and looks for a toolchain matching the toolchains requirements configured in the plugin. In our example, that would be a JDK toolchain with <version> "1.5" and <vendor> "sun". Once a match is found, the plugin then stores the toolchain to be used in the MavenSession. As you can see in our toolchains.xml below, there is indeed a JDK toolchain with <version> "1.5" and <vendor> "sun" configured. So when the maven-compiler-plugin we've configured in our pom.xml above executes, it will see that a JDK toolchain is set in the MavenSession and will thereby use that toolchain (that would be the JDK installed at /path/to/jdk/1.5 for our example) to compile the sources.

Starting with Maven 3.3.1 you can put the toolchains.xml file wherever you like by using the --global-toolchains file option but it is recommended to locate it into ${user.home}/.m2/.
<toolchains>
  <!-- JDK toolchains -->
  <toolchain>
    <type>jdk</type>
    <provides>
      <version>1.5</version>
      <vendor>sun</vendor>
    </provides>
    <configuration>
      <jdkHome>/path/to/jdk/1.5</jdkHome>
    </configuration>
  </toolchain>

  <toolchain>
    <type>jdk</type>
    <provides>
      <version>1.6</version>
      <vendor>sun</vendor>
    </provides>
    <configuration>
      <jdkHome>/path/to/jdk/1.6</jdkHome>
    </configuration>
  </toolchain>

  <!-- other toolchains -->
  <toolchain>
    <type>netbeans</type>
    <provides>
      <version>5.5</version>
    </provides>
    <configuration>
      <installDir>/path/to/netbeans/5.5</installDir>
    </configuration>
  </toolchain>
</toolchains>

Note that you can configure as many toolchains as you want in your toolchains.xml file.
52 Encrypting passwords in settings.xml

52.1 Password Encryption

1. Introduction
2. How to create a master password
3. How to encrypt server passwords
4. How to keep the master password on removable drive
5. Tips

52.1.1 Introduction
Maven supports server password encryption. The main use case, addressed by this solution is:

- multiple users share the same build machine (server, CI box)
- some users have the privilege to deploy Maven artifacts to repositories, some don't.
  - this applies to any server operations, requiring authorization, not only deployment
- settings.xml is shared between users

The implemented solution adds the following capabilities:

- authorized users have an additional settings-security.xml file in their ${user.home}/.m2 folder
  - this file either contains encrypted master password, used to encrypt other passwords
  - or it can contain a relocation - reference to another file, possibly on removable storage
  - this password is created first via CLI for now
- server entries in the settings.xml have passwords and/or keystore passphrases encrypted
  - for now - this is done via CLI after master password has been created and stored in appropriate location

52.1.2 How to create a master password

Use the following command line:

```
mvn --encrypt-master-password <password>
```

Note: Since Maven 3.2.1 the password argument should no longer be used (see Tips below for more information). Maven will prompt for the password. Earlier versions of Maven will not prompt for a password, so it must be typed on the command-line in plaintext.

This command will produce an encrypted version of the password, something like

```
{jSMOWnoPFgsHVPmVz5Vr6t5kRbzGpI8u+9EF1iFQyJo=}
```

Store this password in the ${user.home}/.m2/settings-security.xml; it should look like

```
<settingsSecurity>
  <master>{jSMOWnoPFgsHVPmVz5Vr6t5kRbzGpI8u+9EF1iFQyJo=}</master>
</settingsSecurity>
```

When this is done, you can start encrypting existing server passwords.
52.1.3 How to encrypt server passwords

You have to use the following command line:

```
mvn --encrypt-password <password>
```

*Note:* Just like `--encrypt-master-password` the password argument should no longer be used since Maven 3.2.1 (see Tips below for more information.).

This command produces an encrypted version of it, something like `{COQLCE6DU6GtcS5P=}`

Copy and paste it into the servers section of your `settings.xml` file. This will look like:

```
<settings>
  ...
  <servers>
    ...
    <server>
      <id>my.server</id>
      <username>foo</username>
      <password>{COQLCE6DU6GtcS5P=}</password>
    </server>
    ...
  </servers>
  ...
</settings>
```

Please note that password can contain any information outside of the curly brackets, so that the following will still work:

```
<settings>
  ...
  <servers>
    ...
    <server>
      <id>my.server</id>
      <username>foo</username>
      <password>Oleg reset this password on 2009-03-11, expires on 2009-04-11 {COQLCE6DU6GtcS5P=}</password>
    </server>
    ...
  </servers>
  ...
</settings>
```

Then you can use, say, deploy plugin, to write to this server:

```
mvn deploy:deploy-file -Durl=https://maven.corp.com/repo \\ 
-DrepositoryId=my.server \\ 
-Dfile=your-artifact-1.0.jar \\ 
```

52.1.4 How to keep the master password on removable drive

Create the master password exactly as described above, and store it on a removable drive, for instance on OSX, my USB drive mounts as `/Volumes/mySecureUsb`, so I store
Encrypting passwords in settings.xml

```xml
<settingsSecurity>
  <master>{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>
</settingsSecurity>
```

in the file /Volumes/mySecureUsb/secure/settings-security.xml

And then I create `${user.home}/.m2/settings-security.xml` with the following content:

```xml
<settingsSecurity>
  <relocation>/Volumes/mySecureUsb/secure/settings-security.xml</relocation>
</settingsSecurity>
```

This assures that encryption only works when the USB drive is mounted by the OS. This addresses a use case where only certain people are authorized to deploy and are issued these devices.

52.1.5 Tips

52.1.5.1 Escaping curly-brace literals in your password *(Since: Maven 2.2.0)*

At times, you might find that your password (or the encrypted form of it) contains '{' or '}' as a literal value. If you added such a password as-is to your settings.xml file, you would find that Maven does strange things with it. Specifically, Maven treats all the characters preceding the '{' literal, and all the characters after the '}' literal, as comments. Obviously, this is not the behavior you want. What you really need is a way of escaping the curly-brace literals in your password.

You can do this with the widely used `\` escape character. If your password looks like this:

```
jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+{EF1iFQyJQ=
```

Then, the value you would add to your settings.xml looks like this:

```
{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+\{EF1iFQyJQ=}
```

52.1.5.2 Password Security

Editing `settings.xml` and running the above commands can still leave your password stored locally in plaintext. You may want to check the following locations:

- Shell history (e.g. by running `history`). You may want to clear your history after encrypting the above passwords
- Editor caches (e.g. `~/.viminfo`)

Also note that the encrypted passwords can be decrypted by someone that has the master password and settings security file. Keep this file secure (or stored separately) if you expect the possibility that the `settings.xml` file may be retrieved.

52.1.5.3 Password Escaping on different platforms

On some platforms it might be necessary to quote the password if it contains special characters like `%`, `!`, `®`, etc. For example on Windows you have to be careful about things like the following:

The following example will not work on Windows:

```
mvn --encrypt-master-password a%$^b
```

whereas the following will work on Windows:

```
mvn --encrypt-master-password "a%$^b"
```
If you are on a linux/unix platform you should use single quotes for the above master password. Otherwise the master password will not work (caused by the dollar sign and the exclamation mark).

52.1.5.4 Prompting for Password

In Maven before version 3.2.1 you have to give the password on the command line as an argument which means you might need to escape your password. In addition usually the shell stores the full history of commands you have entered, therefore anyone with access to your computer could restore the password from the shell’s history.

Starting with Maven 3.2.1, the password is an optional argument. If you omit the password, you will be prompted for it which prevents all the issues mentioned above.

We strongly recommend using Maven 3.2.1 and above to prevent problems with escaping special characters and of course security issues related to bash history or environment issues in relationship with the password.
53 Reusable Test JARs

53.1 Guide to using attached tests

Many times you may want to reuse the tests that you have created for a project in another. For example if you have written foo-core and it contains test code in the ${basedir}/src/test/java it would be useful to package up those compiled tests in a JAR and deploy them for general reuse. To do this you would configure the maven-jar-plugin as follows:

```xml
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.0.2</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>test-jar</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

53.1.1 Installing the attached test JAR

In order to install the attached test JAR you simply use the standard install phase by executing the following command:

```
mvn install
```

53.1.2 Deploying the attached test JAR

In order to deploy the attached test JAR you simply use the standard deploy phase by executing the following command:

```
mvn deploy
```

53.1.3 Using the attached test JAR

In order to use the attached test JAR that was created above you simply specify a dependency on the main artifact with a specified type of test-jar and the classifier.

```xml
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>com.myco.app</groupId>
            <artifactId>foo</artifactId>
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
<version>1.0-SNAPSHOT</version>
<classifier>tests</classifier>
<type>test-jar</type>
<scope>test</scope>
</dependency>
</dependencies>
...
</project>
54 Plugin Developer Centre

54.1 Plugin Developers Centre

This documentation centre is for those who are developing Maven plugins. This might be for your own build, or as an accompaniment to your third party tool.

What is a Mojo? A mojo is a Maven plain Old Java Object. Each mojo is an executable goal in Maven, and a plugin is a distribution of one or more related mojos.

- Introduction to Plugin Development - Introduction to concepts
- Your First Mojo - Learn how to write your first plugin
- Your First Report Mojo - Learn how to write your first reporting plugin
- Testing your Plugin - How to write tests for your plugins
- Documenting your Plugin - How to write documentation for your plugins
- TODO: creating and using custom packaging (like maven-archetype packaging)
- Plugins Cookbook - Examples for how to perform common tasks in plugins
- Common Bugs and Pitfalls - Overview of problematic coding patterns

54.1.1 Reference

- Mojo API and Annotation Reference
- Maven API Reference
- Maven Class Loading

54.1.2 Examples

- Injecting POM Properties via settings.xml
- Maven 3 lifecycle extensions
55 Testing your Plugin

55.1 Introduction
Currently, Maven only supports unit testing out of the box. This document is intended to help Maven Developers test plugins with unit tests, integration tests, and functional tests.

55.2 Testing Styles: Unit Testing vs. Functional/Integration Testing
A unit test attempts to verify a mojo as an isolated unit, by mocking out the rest of the Maven environment. A mojo unit test does not attempt to run your plugin in the context of a real Maven build. Unit tests are designed to be fast.

A functional/integration test attempts to use a mojo in a real Maven build, by launching a real instance of Maven in a real project. Normally this requires you to construct special dummy Maven projects with real POM files. Often this requires you to have already installed your plugin into your local repository so it can be used in a real Maven build. Functional tests run much more slowly than unit tests, but they can catch bugs that you may not catch with unit tests.

The general wisdom is that your code should be mostly tested with unit tests, but should also have some functional tests.

55.3 Unit Tests

55.3.1 Using JUnit alone
In principle, you can write a unit test of a plugin Mojo the same way you'd write any other JUnit test case, by writing a class that extends TestCase.

However, most mojos need more information to work properly. For example, you'll probably need to inject a reference to a MavenProject, so your mojo can query project variables.

55.3.2 Using PlexusTestCase
Mojo variables are injected using Plexus, and many Mojos are written to take specific advantage of the Plexus container (by executing a lifecycle or having various injected dependencies).

If all you need are Plexus container services, you can write your class with extends PlexusTestCase instead of TestCase.

With that said, if you need to inject Maven objects into your mojo, you'll probably prefer to use the maven-plugin-testing-harness.

55.3.3 maven-plugin-testing-harness
The maven-plugin-testing-harness is explicitly intended to test the org.apache.maven.reporting.AbstractMavenReport#execute() implementation.

In general, you need to include maven-plugin-testing-harness as a dependency, and create a *MojoTest (by convention) class which extends AbstractMojoTestCase.

```xml
<dependencies>
  ...
  <dependency>
    ...
  ...
</dependencies>
```
<groupId>org.apache.maven.plugin-testing</groupId>
<artifactId>maven-plugin-testing-harness</artifactId>
<version>3.3.0</version>
<scope>test</scope>
</dependency>
...
</dependencies>
...

public class YourMojoTest
extends AbstractMojoTestCase
{
  /**
   * @see junit.framework.TestCase#setUp()
   */
  protected void setUp() throws Exception
  {
    // required for mojo lookups to work
    super.setUp();
  }
  /**
   * @throws Exception
   */
  public void testMojoGoal() throws Exception
  {
    File testPom = new File( getBasedir(),
    "src/test/resources/unit/basic-test/basic-test-plugin-config.xml" );
    YourMojo mojo = (YourMojo) lookupMojo( "yourGoal", testPom );
    assertNotNull( mojo );
  }
}

For more information, refer to Maven Plugin Harness Wiki

55.4 Integration/Functional testing

55.4.1 maven-verifier

maven-verifier tests are run using JUnit or TestNG, and provide a simple class allowing you to launch Maven and assert on its log file and built artifacts. It also provides a ResourceExtractor, which extracts a Maven project from your src/test/resources directory into a temporary working directory where you can do tricky stuff with it.

Maven itself uses maven-verifier to run its core integration tests. For more information, please refer to Creating a Maven Integration Test.

public class TrivialMavenVerifierTest extends TestCase
{
  public void testMyPlugin() throws Exception
  {
    // Check in your dummy Maven project in /src/test/resources/...
    // The testdir is computed from the location of this
// file.
File testDir = ResourceExtractor.simpleExtractResources( getClass(), "/my-dummy-maven-project" );
Verifier verifier;
/*
 * We must first make sure that any artifact created
 * by this test has been removed from the local
 * repository. Failing to do this could cause
 * unstable test results. Fortunately, the verifier
 * makes it easy to do this.
 */
verifier = new Verifier( testDir.getAbsolutePath() );
verifier.deleteArtifact( "org.apache.maven.its.itsample", "parent", "1.0", "pom" );
verifier.deleteArtifact( "org.apache.maven.its.itsample", "checkstyle-test", "1.0", "jar" );
verifier.deleteArtifact( "org.apache.maven.its.itsample", "checkstyle-assembly", "1.0" );
/*
 * The Command Line Options (CLI) are passed to the
 * verifier as a list. This is handy for things like
 * redefining the local repository if needed. In
 * this case, we use the -N flag so that Maven won't
 * recurse. We are only installing the parent pom to
 * the local repo here.
 */
List cliOptions = new ArrayList();
cliOptions.add( "-N" );
verifier.executeGoal( "install" );
/*
 * This is the simplest way to check a build
 * succeeded. It is also the simplest way to create
 * an IT test: make the build pass when the test
 * should pass, and make the build fail when the
 * test should fail. There are other methods
 * supported by the verifier. They can be seen here:
 */
verifier.verifyErrorFreeLog();
/*
 * Reset the streams before executing the verifier
 * again.
 */
verifier.resetStreams();
/*
 * The verifier also supports beanshell scripts for
 * verification of more complex scenarios. There are
 * plenty of examples in the core-it tests here:
 * https://svn.apache.org/repos/asf/maven/core-integration-testing/trunk
 */

Note: maven-verifier and maven-verifier-plugin sound similar, but are totally different unrelated pieces of code. maven-verifier-plugin simply verifies the existence/absence of files on the filesystem. You could use it for functional testing, but you may need more features than maven-verifier-plugin provides.
55.4.2 maven-invoker-plugin

You can use maven-invoker-plugin to invoke Maven and to provide some BeanShell/Groovy tests. Tests written in this way don't run under JUnit/TestNG; instead, they're run by Maven itself.

You can take a look at the maven-install-plugin how there are integration tests are written.

```xml
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-invoker-plugin</artifactId>
        <version>1.10</version>
        <configuration>
          <projectsDirectory>src/it</projectsDirectory>
          <pomIncludes>
            <pomInclude>**/pom.xml</pomInclude>
          </pomIncludes>
          <postBuildHookScript>verify</postBuildHookScript>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>run</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```
56 Documenting your Plugin

56.1 Introduction
A Guide to the Plugin Documentation Standard was created. This document is intended to verify it during the Plugins development.

56.2 Verify Plugin Documentation
The maven-docck-plugin checks that a project complies with the Plugin Documentation Standard. You should verify that all Plugin documentation respects this standard. The maven-docck-plugin can be run:

```
mvn docck:check
```

56.3 References
- Maven Plugin Documentation
57 Common Bugs and Pitfalls

57.1 Common Bugs and Pitfalls

Maven is not the smallest project in terms of source code and has as such already suffered from many bugs. Having a closer look at all the issues revealed some coding problems that had widespread among the various subcomponents. This document lists these commonly occurring anti patterns in order to help the Maven community to prevent rather than fix bugs. Note that the primary focus is on pointing out problems that are subtle in their nature rather than giving a comprehensive guide for Java or Maven development.

- Reading and Writing Text Files
- Converting between URLs and Filesystem Paths
- Handling Strings Case-insensitively
- Creating Resource Bundle Families
- Using System Properties
- Using Shutdown Hooks
- Resolving Relative Paths
- Determining the Output Directory for a Site Report
- Retrieving the Mojo Logger

57.1.1 Reading and Writing Text Files

Textual content is composed of characters while file systems merely store byte streams. A file encoding (aka charset) is used to convert between bytes and characters. The challenge is using the right file encoding.

The JVM has this notion of a default encoding (given by the `file.encoding` property) which it derives from a system's locale. While this might be a convenient feature sometimes, using this default encoding for a project build is in general a bad idea: The build output will depend on the machine/developer who runs the build. As such, usage of the default encoding threatens the dream of a reproducible build.

For example, if developer A has UTF-8 as default encoding while developer B uses ISO-8859-1, text files are very likely to get messed up during resource filtering or similar tasks.

Therefore, developers should avoid any direct or indirect usage of the classes/methods that simply employ the platform's default encoding. For instance, `FileWriter` and `FileReader` should usually be avoided:

```java
/*
 * FIXME: This assumes the source file is using the platform's default encoding.
 */
Reader reader = new FileReader( javaFile );
```

Instead, the classes `OutputStreamWriter` and `OutputStreamReader` can be used in combination with an explicit encoding value. This encoding value can be retrieved from a mojo parameter such that the user can configure the plugin to fit his/her needs.

To save the user from configuring each plugin individually, conventions have been established that allow a user to centrally configure the file encoding per POM. Plugin developers should respect these conventions wherever possible:

- Source File Encoding
- Report Output Encoding

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
Finally note that XML files require special handling because they are equipped with an encoding declaration in the XML prolog. Reading or writing XML files with an encoding that does not match their XML prolog's encoding attribute is a bad idea:

```java
/*
 * FIXME: This assumes the XML encoding declaration matches the platform's default encoding.
 */
Writer writer = new FileWriter( xmlFile );
writer.write( xmlContent );
```

To ease the correct processing of XML files, developers are encouraged to use `ReaderFactory.newXmlReader()` and `WriterFactory.newXmlWriter()` from the Plexus Utilities.

### 57.1.2 Converting between URLs and Filesystem Paths

URLs and filesystem paths are really two different things and converting between them is not trivial. The main source of problems is that different encoding rules apply for the strings that make up a URL or filesystem path. For example, consider the following code snippet and its associated console output:

```java
File file = new File( "foo bar+foo" );
URL url = file.toURI().toURL();
System.out.println( file.toURL() );  // > file:/C:/temp/foo bar+foo
System.out.println( url );          // > file:/C:/temp/foo%20bar+foo
System.out.println( url.getPath() ); // > /C:/temp/foo%20bar+foo
System.out.println( URLDecoder.decode( url.getPath(), "UTF-8" ) ); // > /C:/temp/foo bar foo
```

First of all, please note that `File.toURL()` does not escape the space character (and others). This yields an invalid URL, as per RFC 2396, section 2.4.3 "Excluded US-ASCII Characters". The class `java.net.URL` will silently accept such invalid URLs, in contrast `java.net.URI` will not (see also `URL.toURI()`). For this reason, `File.toURL()` has been deprecated and should be replaced with `File.toURI().toURL()`.

Next, `URL.getPath()` does in general not return a string that can be used as a filesystem path. It returns a substring of the URL and as such can contain escape sequences. The prominent example is the space character which will show up as "%20". People sometimes hack around this by means of `replace("%20", " ")` but that does simply not cover all cases. It's worth to mention that on the other hand the related method `URI.getPath()` does decode escapes but still the result is not a filesystem path (compare the source for the constructor `File(URI)`). To summarize, the following idiom is to be avoided:

```java
URL url = new URL( "file:/C:/Program%20Files/Java/bin/java.exe" );
/*
 * FIXME: This does not decode percent encoded characters.
 */
File path = new File( url.getPath() );
```

To decode a URL, people sometimes also choose `java.net.URLDecoder`. The pitfall with this class is that it actually performs HTML form decoding which is yet another encoding and not the same as the URL encoding (compare the last paragraph in class javadoc about `java.net.URL`).
For instance, a URLDecoder will erroneously convert the character "+" into a space as illustrated by the last sysout in the example above.

In an ideal world, code targeting JRE 1.4+ could easily avoid these problems by using the constructor File(URI) as suggested by the following snippet:

```java
URL url = new URL("file:/C:/Documents and Settings/user/.m2/settings.xml");
/*
 * FIXME: This assumes the URL is fully compliant with RFC 3986.
 */
File path = new File( new URI( url.toExternalForm() ) );
```

The remaining source of frustration is the conversion from URL to URI. As already said, the URL class accepts malformed URLs which will make the constructor of URI throw an exception. And indeed, class loaders from Sun JREs up to Java 1.4 will deliver malformed URLs when queried for a resource. Likewise, the class loaders employed by Maven 2.x deliver malformed resource URLs regardless of the JRE version (see MNG-3607).

For all these reasons, it is recommended to use FileUtils.toFile() from Commons IO or FileUtils.toFile() from a recent Plexus Utilities.

### 57.1.3 Handling Strings Case-insensitively

When developers need to compare strings without regard to case or want to realize a map with case-insensitive string keys, they often employ String.toLowerCase() or String.toUpperCase() to create a "normalized" string before doing a simple String.equals(). Now, the to*Case() methods are overloaded: One takes no arguments and one takes a Locale object.

The gotcha with the arg-less methods is that their output depends on the default locale of the JVM but the default locale is out of control of the developer. That means the string expected by the developer (who runs/tests his code in a JVM using locale `xy`) does not necessarily match the string seen by another user (that runs a JVM with locale `ab`). For example, the comparison shown in the next code snippet is likely to fail for systems with default locale Turkish because Turkish has unusual casing rules for the characters "i" and "I":

```java
/*
 * FIXME: This assumes the casing rules of the current platform
 * match the rules for the English locale.
 */
if ( "info".equals( debugLevel.toLowerCase() ) )
    logger.info( message );
```

For case-insensitive string comparisons which should be locale-insensitive, the method String.equalsIgnoreCase() should be used instead. If only a substring like a prefix/suffix should be compared, the method String.regionMatches() can be used instead.

If the usage of String.to*Case() cannot be avoided, the overloaded version taking a Locale object should be used, passing in Locale.ENGLISH. The resulting code will still run on Non-English systems, the parameter only locks down the casing rules used for the string comparison such that the code delivers the same results on all platforms.

### 57.1.4 Creating Resource Bundle Families

Especially reporting plugins employ resource bundles to support internationalization. One language (usually English) is provided as the fallback/default language in the base resource bundle. Due to
the lookup strategy performed by `ResourceBundle.getBundle()`, one must always provide a dedicated resource bundle for this default language, too. This bundle should be empty because it inherits the strings via the parent chain from the base bundle, but it must exist.

The following example illustrates this requirement. Imagine the broken resource bundle family shown below which is intended to provide localization for English, German and French:

```text
src/
 +- main/
    +- resources/
       +- mymojo-report.properties
       +- mymojo-report_de.properties
       +- mymojo-report_fr.properties
```

Now, if a resource bundle is to be looked up for English on a JVM whose default locale happens to be French, the bundle `mymojo-report_fr.properties` will be loaded instead of the intended bundle `mymojo-report.properties`.

Reporting plugins that suffer from this bug can easily be detected by executing `mvn site -D locales=xy,en` where `xy` denotes any other language code supported by the particular plugin. Specifying `xy` as the first locale will have the Maven Site Plugin change the JVM's default locale to `xy` which in turn causes the lookup for `en` to fail as outlined above unless the plugin has a dedicated resource bundle for English.

### 57.1.5 Using System Properties

Maven's command line supports the definition of system properties via arguments of the form `-D key=value`. While these properties are called system properties, plugins should never use `System.getProperty()` and related methods to query these properties. For example, the following code snippet will not work reliably when Maven is embedded, say into an IDE or a CI server:

```java
public MyMojo extends AbstractMojo
{
    public void execute()
    {
        /*
        * FIXME: This prevents proper embedding into IDEs or CI systems.
        */
        String value = System.getProperty( "maven.test.skip" );
    }
}
```

The problem is that the properties managed by the `System` class are global, i.e. shared among all threads in the current JVM. To prevent conflicts with other code running in the same JVM, Maven plugins should instead query the execution properties. These can be obtained from `MavenSession.getExecutionProperties()`.

### 57.1.6 Using Shutdown Hooks

People occasionally employ shutdown hooks to perform cleanup tasks, e.g. to delete temporary files as shown in the example below:

```java
public MyMojo extends AbstractMojo
{
    public void execute()
```
The problem is that the JVM executing Maven can be running much longer than the actual Maven build. Of course, this does not apply to the standalone invocation of Maven from the command line. However, it affects the embedded usage of Maven in IDEs or CI servers. In those cases, the cleanup tasks will be deferred, too. If the JVM is then executing a bunch of other Maven builds, many such cleanup tasks can sum up, eating up resources of the JVM.

For this reason, plugin developers should avoid usage of shutdown hooks and rather use try/finally blocks to perform cleanup as soon as the resources are no longer needed.

57.1.7 Resolving Relative Paths

It is common practice for users of Maven to specify relative paths in the POM, not to mention that the Super POM does so, too. The intention is to resolve such relative paths against the base directory of the current project. In other words, the paths target/classes and ${{basedir}}/target/classes should resolve to the same directory for a given POM.

Unfortunately, the class java.io.File does not resolve relative paths against the project's base directory. As mentioned in its class javadoc, it resolves relative paths against the current working directory. In plain English: Unless a Maven component has complete control over the current working directory, any usage of java.io.File in combination with a relative path is a bug.

At first glance, one might be tempted to argue that the project base directory is equal to the current working directory. However, this assumption is generally not true. Consider the following scenarios:

a. Reactor Builds

   When a child module is build during a reactor build, the current working directory is usually the base directory of the parent project, not the base directory of the current module. That is the most common scenario where users are faced with the bug.

b. Embedded Maven Invocations

   Other tools, most notably IDEs, that run Maven under the hood may have set the current working directory to their installation folder or whatever they like.

c. Maven Invocations using the -f Switch

   While it is surely an uncommon use-case, the user is free to invoke Maven from an arbitrary working directory by specifying an absolute path like mvn -f /home/me/projects/demo/pom.xml.

Hence this example code is prone to misbehave:

```java
public MyMojo extends AbstractMojo {
    /**
     * @parameter
     */
    private String outputDirectory;
    public void execute() {
    }
}
In order to guarantee reliable builds, Maven and its plugins must manually resolve relative paths against the project's base directory. A simple idiom like the following will do just fine:

```java
File file = new File( path );
if ( !file.isAbsolute() )
{
    file = new File( project.getBasedir(), file );
}
```

Many Maven plugins can get this resolution automatically if they declare their affected mojo parameters of type `java.io.File` instead of `java.lang.String`. This subtle difference in parameter types will trigger a feature known as path translation, i.e., the Maven core will automatically resolve relative paths when it pumps the XML configuration into a mojo.

### 57.1.8 Determining the Output Directory for a Site Report

Most reporting plugins inherit from `AbstractMavenReport`. In doing so, they need to implement the inherited but abstract method `getOutputDirectory()`. To implement this method, plugins usually declare a field named `outputDirectory` which they return in the method. Nothing wrong so far.

Now, some plugins need to create additional files in the report output directory that accompany the report generated via the sink interface. While it is tempting to use either the method `getOutputDirectory()` or the field `outputDirectory` directly in order to setup a path for the output files, this leads most likely to a bug. More precisely, those plugins will not properly output files when run by the Maven Site Plugin as part of the site lifecycle. This is best noticed when the output directory for the site is configured directly in the Maven Site Plugin such that it deviates from the expression `${project.reporting.outputDirectory}` that the plugins use by default. Multi-language site generation is another scenario to exploit this bug which is illustrated below:

```java
public MyReportMojo extends AbstractMavenReport
{
    /**
     * @parameter default-value="${project.reporting.outputDirectory}"*
     */
    private File outputDirectory;
    protected String getOutputDirectory()
    {
        return outputDirectory.getAbsolutePath();
    }
    public void executeReport( Locale locale )
    {
        /*
         * FIXME: This assumes the mojo parameter reflects the effective
         * output directory as used by the Maven Site Plugin.
         */
        
    }
```
There are in principal two situations in which a report mojo could be invoked. The mojo might be run directly from the command line or the default build lifecycle or it might be run indirectly as part of the site generation along with other report mojos. The glaring difference between these two invocations is the way the output directory is controlled. In the first case, the parameter `outputDirectory` from the mojo itself is used. In the second case however, the Maven Site Plugin takes over control and will set the output directory according to its own configuration by calling `MavenReport.setReportOutputDirectory()` on the reports being generated.

Therefore, developers should always use `MavenReport.getReportOutputDirectory()` if they need to query the effective output directory for the report. The implementation of `AbstractMavenReport.getOutputDirectory()` is only intended as a fallback in case the mojo is not run as part of the site generation.

### 57.1.9 Retrieving the Mojo Logger

Maven employs an IoC container named Plexus to setup a plugin's mojos before their execution. In other words, components required by a mojo will be provided by means of dependency injection, more precisely field injection. The important point to keep in mind is that this field injection happens after the mojo's constructor has finished. This means that references to injected components are invalid during the construction time of the mojo.

For example, the next snippet tries to retrieve the mojo logger during construction time but the mojo logger is an injected component and as such has not been properly initialized yet:

```java
public MyMojo extends AbstractMojo
{
    @/****
    * FIXME: This will retrieve a wrong logger instead of the intended mojo logger.
    */
    private Log log = getLog();
    public void execute()
    {
        log.debug( "..." );
    }
}
```

In case of the logger, the above mojo will simply use a default console logger, i.e. the code defect is not immediately noticeable by a `NullPointerException`. This default logger will however use a different message format for its output and also outputs debug messages even if Maven's debug mode was not enabled. For this reason, developers must not try to cache the logger during construction time. The method `getLog()` is fast enough and can simply be called whenever one needs it.
58 Mojo API

58.1 Introduction

Maven plugins can be written in Java or any of a number of scripting languages. Plugins consists of one or more Mojos, each one being the implementation for one of the plugin's goals. Maven tries to stay out of the way of the programmer with its new Mojo API. This opens up the opportunity for many Mojos to be reused outside of Maven, or bridged into Maven from external systems like Ant.

NOTE: For now, we will limit the discussion to Java-based Mojos, since each scripting language will present these same basic requirements with various forms of implementation.

Although the requirements on Mojos are minimal by design, there are still a very few requirements that Mojo developers must keep in mind. Basically, these Mojo requirements are embodied by the org.apache.maven.plugin.Mojo interface, which the Mojo must implement (or else extend its abstract base class counterpart org.apache.maven.plugin.AbstractMojo). This interface guarantees the correct execution contract for the Mojo: no parameters, void return type, and a throws clause that allows only org.apache.maven.plugin.MojoExecutionException and its derivatives. It also guarantees that the Mojo will have access to the standard Maven user-feedback mechanism, org.apache.maven.plugin.logging.Log, so the Mojo can communicate important events to the console or other log sink.

As mentioned before, each Plugin - or packaged set of Mojos - must provide a descriptor called plugin.xml under the path META-INF/maven inside the Plugin jar file. Fortunately, Maven also provides a set of Javadoc annotations (named Mojo Javadoc Tags), Java 5 annotations (named Maven Plugin Tools Java5 Annotations) and tools (named plugin-tools) to generate this descriptor, so developers don't have to worry about directly authoring or maintaining a separate XML metadata file.

To serve as a quick reference for the developer, the rest of this page will document these features (the API, along with the annotations) which are considered the best practice for developing Mojos.

58.2 API Documentation

58.2.1 org.apache.maven.plugin.Mojo

This interface forms the contract required for Mojos to interact with the Maven infrastructure. It features an execute() method, which triggers the Mojo's build-process behavior, and can throw a MojoExecutionException if an error condition occurs. See below for a discussion on proper use of this Exception class. Also included is the setLog(...) method, which simply allows Maven to inject a logging mechanism which will allow the Mojo to communicate to the outside world through standard Maven channels.

58.Method Summary:

- void setLog( org.apache.maven.plugin.logging.Log )
  Inject a standard Maven logging mechanism to allow this Mojo to communicate events and feedback to the user.
- void execute() throws org.apache.maven.plugin.MojoExecutionException
  Perform whatever build-process behavior this Mojo implements. This is the main trigger for the Mojo inside the Maven system, and allows the Mojo to communicate fatal errors by throwing an instance of MojoExecutionException.

The MojoExecutionException (and all error conditions inside the Mojo) should be handled very carefully. The simple wrapping of lower-level exceptions without providing any indication of a user-friendly probable cause is strictly discouraged. In fact, a much better course of action
is to provide error handling code (try/catch stanzas) for each coherent step within the Mojo's execution. Developers are then in a much better position to diagnose the cause of any error, and provide user-friendly feedback in the message of the MojoExecutionException.

58.2.2 org.apache.maven.plugin.AbstractMojo

Currently, this abstract base class simply takes care of managing the Maven log for concrete derivations. In keeping with this, it provides a protected method, getLog():Log, to furnish Log access to these concrete implementations.

58.Method Summary:
- public void setLog( org.apache.maven.plugin.logging.Log )
  [IMPLEMENTED]
  Inject a standard Maven logging mechanism to allow this Mojo to communicate events and feedback to the user.
- protected Log getLog()
  [IMPLEMENTED]
  Furnish access to the standard Maven logging mechanism which is managed in this base class.
- void execute() throws org.apache.maven.plugin.MojoExecutionException
  [ABSTRACT]
  Perform whatever build-process behavior this Mojo implements. See the documentation for Mojo above for more information.

58.2.3 org.apache.maven.plugin.logging.Log

This interface supplies the API for providing feedback to the user from the Mojo, using standard Maven channels. There should be no big surprises here, although you may notice that the methods accept java.lang.CharSequence rather than java.lang.String. This is provided mainly as a convenience, to enable developers to pass things like StringBuffer directly into the logger, rather than formatting first by calling toString().

58.Method Summary:
- void debug( java.lang.CharSequence )
  Send a message to the user in the debug error level.
- void debug( java.lang.CharSequence, java.lang.Throwable )
  Send a message (and accompanying exception) to the user in the debug error level. The error's stacktrace will be output when this error level is enabled.
- void debug( java.lang.Throwable )
  Send an exception to the user in the debug error level. The stack trace for this exception will be output when this error level is enabled.
- void info( java.lang.CharSequence )
  Send a message to the user in the info error level.
- void info( java.lang.CharSequence, java.lang.Throwable )
  Send a message (and accompanying exception) to the user in the info error level. The error's stacktrace will be output when this error level is enabled.
- void warn( java.lang.CharSequence )
  Send an exception to the user in the info error level. The stack trace for this exception will be output when this error level is enabled.
Send a message to the user in the warn error level.

- void warn(java.lang.CharSequence, java.lang.Throwable)
  Send a message (and accompanying exception) to the user in the warn error level. The error's stacktrace will be output when this error level is enabled.

- void warn(java.lang.CharSequence)
  Send an exception to the user in the warn error level. The stack trace for this exception will be output when this error level is enabled.

- void error(java.lang.CharSequence)
  Send a message to the user in the error error level.

- void error(java.lang.CharSequence, java.lang.Throwable)
  Send a message (and accompanying exception) to the user in the error error level. The error's stacktrace will be output when this error level is enabled.

- void error(java.lang.CharSequence)
  Send an exception to the user in the error error level. The stack trace for this exception will be output when this error level is enabled.

58.3 The Descriptor and Annotations

In addition to the normal Java requirements in terms of interfaces and/or abstract base classes which need to be implemented, a plugin descriptor must accompany these classes inside the plugin jar. This descriptor file is used to provide metadata about the parameters and other component requirements for a set of Mojos so that Maven can initialize the Mojo and validate its configuration before executing it. As such, the plugin descriptor has a certain set of information that is required for each Mojo specification to be valid, as well as requirements for the overall plugin descriptor itself.

NOTE: In the following discussion, bolded items are the descriptor's element name along with a Mojo Javadoc tag (if applicable) supporting that piece of the plugin descriptor. A couple of examples are: someElement (@annotation parameterName="parameterValue") or someOtherElement (@annotation <rawAnnotationValue>).

NOTE: since maven-plugin-plugin 3.0, it is now possible to use Maven Plugin Tools Java 5 Annotations equivalent to Mojo Javadoc tags. See Using annotations documentation.

The plugin descriptor (see descriptor reference) must be provided in a jar resource with the path: META-INF/maven/plugin.xml, and it must contain the following:

<table>
<thead>
<tr>
<th>Descriptor Element</th>
<th>Required?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mojos</td>
<td>Yes</td>
<td>Descriptors for each Mojo provided by the plugin, each inside a mojo sub-element. Mojos descriptors are covered in detail below. Obviously, a plugin without any declared Mojos doesn't make sense, so the mojos element is required, along with at least one mojo sub-element.</td>
</tr>
</tbody>
</table>
dependencies | Yes
---|---

A set of dependencies which the plugin requires in order to function. Each dependency is provided inside a `dependency` sub-element. Dependency specifications are covered below. Since all plugins must have a dependency on `maven-plugin-api`, this element is effectively required. Using the plugin toolset, these dependencies can be extracted from the POM’s dependencies.

Each Mojo specified inside a plugin descriptor must provide the following (annotations specified here are at the class level):

<table>
<thead>
<tr>
<th>Descriptor Element</th>
<th>Mojo Javadoc tag</th>
<th>Required?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregator</td>
<td>@aggregator</td>
<td>No</td>
<td>Flags this Mojo to run it in a multi module way, i.e. aggregate the build with the set of projects listed as modules.</td>
</tr>
<tr>
<td>configurator</td>
<td>@configurator&lt;roleHint&gt;</td>
<td>No</td>
<td>The configurator type to use when injecting parameter values into this Mojo. The value is normally deduced from the Mojo’s implementation language, but can be specified to allow a custom ComponentConfigurator implementation to be used. NOTE: This will only be used in very special cases, using a highly controlled vocabulary of possible values. (Elements like this are why it’s a good idea to use the descriptor tools.)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Annotation</td>
<td>Default/Required</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>execute</td>
<td>@execute</td>
<td>No</td>
<td>When this goal is invoked, it will first invoke a parallel lifecycle, ending at the given phase. If a goal is provided instead of a phase, that goal will be executed in isolation. The execution of either will not affect the current project, but instead make available the <code>${executedProject}</code> expression if required. An alternate lifecycle can also be provided: for more information see the <a href="https://maven.apache.org/">documentation on the build lifecycle</a></td>
</tr>
<tr>
<td>executionStrategy</td>
<td>@executionStrategy</td>
<td>No</td>
<td>Specify the execution strategy. <strong>NOTE:</strong> Unsupported since Maven 3.0.</td>
</tr>
<tr>
<td>goal</td>
<td>@goal &lt;goalName&gt;</td>
<td>Yes</td>
<td>The name for the Mojo that users will reference from the command line to execute the Mojo directly, or inside a POM in order to provide Mojo-specific configuration.</td>
</tr>
<tr>
<td>inheritByDefault</td>
<td>@inheritByDefault</td>
<td>No, Default: true</td>
<td>Specify that the Mojo is inherited. <strong>NOTE:</strong> Unsupported since Maven 3.0.</td>
</tr>
<tr>
<td>instantiationStrategy</td>
<td>@instantiationStrategy</td>
<td>No, Default: per-lookup</td>
<td>Specify the instantiation strategy.</td>
</tr>
<tr>
<td>phase</td>
<td>@phase &lt;phaseName&gt;</td>
<td>No</td>
<td>Defines a default phase to bind a mojo execution to if the user does not explicitly set a phase in the POM. <strong>Note:</strong> This annotation will not automagically make a mojo run when the plugin declaration is added to the POM. It merely enables the user to omit the <code>&lt;phase&gt;</code> element from the surrounding <code>&lt;execution&gt;</code> element.</td>
</tr>
</tbody>
</table>
requiresDependency

```java
@requiresDependency
<requiredClassPath>
```

Flags this Mojo as requiring the dependencies in the specified classpath to be resolved before it can execute. The matrix below illustrates which values for `<requiredClassPath>` (first column) are supported and which dependency scopes (first row) they will request to resolve:

```
syst  prov  corr  runt  test
com X   X   X   -   -
org  -    X   X   -   -
cor X   X   X   X   -
+rt  (sin
    Max
    3.0)
te X   X   X   X   X
```

If this annotation is present but no scope is specified, the scope defaults to `runtime`. If the annotation is not present at all, the mojo must not make any assumptions about the artifacts associated with a Maven project.
**requiresDependency**

```java
@requiresDependency
<requiredClassPath>
```

Flags this mojo as requiring information about the dependencies that would make up the specified class path. As the name suggests, this annotation is similar to `@requiresDependencyResolution` and supports the same values for `<requiredClassPath>`. The important difference is that this annotation will not resolve the files for the dependencies, i.e. the artifacts associated with a Maven project can lack a file. As such, this annotation is meant for mojos that only want to analyze the set of transitive dependencies, in particular during early lifecycle phases where full dependency resolution might fail due to projects which haven't been built yet. A mojo may use both this annotation and `@requiresDependencyResolution` at the same time. The resolution state of any dependency that is collected but not requested to be resolved is undefined. Since Maven 3.0.

**requiresDirectInvocation**

```java
@requiresDirectInvocation
<true|false>
```

Flags this Mojo to be invoke directly. **NOTE:** Unsupported since Maven 3.0.

**requiresOnline**

```java
@requiresOnline
<true|false>
```

No. Default: false

Flags this Mojo to require online mode for its operation.

**requiresProject**

```java
@requiresProject
<true|false>
```

No. Default: true

Flags this Mojo to run inside of a project.

**requiresReports**

```java
@requiresReports
<true|false>
```

No. Default: false

Flags this Mojo to require reports. **NOTE:** Unsupported since Maven 3.0.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Annotation</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadSafe</td>
<td>@threadSafe</td>
<td>No. Default: false</td>
<td>Marks this mojo as being thread-safe, i.e. the mojo safely supports concurrent execution during parallel builds. Mojos without this annotation will make Maven output a warning when used during a parallel build session. The short-hand notation @threadSafe without a tag value is equivalent to @threadSafe true. Since Maven 3.0.</td>
</tr>
<tr>
<td>description</td>
<td>none (detected)</td>
<td>No</td>
<td>The description of this Mojo's functionality. Using the toolset, this will be the class-level Javadoc description provided. NOTE: While this is not a required part of the Mojo specification, it SHOULD be provided to enable future tool support for browsing, etc. and for clarity.</td>
</tr>
<tr>
<td>implementation</td>
<td>none (detected)</td>
<td>Yes</td>
<td>The Mojo's fully-qualified class name (or script path in the case of non-Java Mojos).</td>
</tr>
<tr>
<td>language</td>
<td>none (detected)</td>
<td>No. Default: java</td>
<td>The implementation language for this Mojo (Java, beanshell, etc.).</td>
</tr>
<tr>
<td>deprecated</td>
<td>@deprecated</td>
<td>No</td>
<td>Specify the version when the Mojo was deprecated to the API. Similar to Javadoc deprecated. This will trigger a warning when a user tries to configure a parameter marked as deprecated.</td>
</tr>
<tr>
<td>since</td>
<td>@since &lt;since-text&gt;</td>
<td>No</td>
<td>Specify the version when the Mojo was added to the API. Similar to Javadoc since.</td>
</tr>
</tbody>
</table>

Each Mojo specifies the parameters that it expects in order to work. These parameters are the Mojo’s link to the outside world, and will be satisfied through a combination of POM/project values, plugin configurations (from the POM and configuration defaults), and System properties.

NOTE[1]: For this discussion on Mojo parameters, a single annotation may span multiple elements in the descriptor’s specification for that parameter. Duplicate annotation declarations in this section will be used to detail each parameter of an annotation separately.
NOTE[2]: In many cases, simply annotating a Mojo field with `@parameter` will be enough to allow injection of a value for that parameter using POM configuration elements. The discussion below shows advanced usage for this annotation, along with others.

Each parameter for a Mojo must be specified in the plugin descriptor as follows:

<table>
<thead>
<tr>
<th>Descriptor Element</th>
<th>Mojo Javadoc tag</th>
<th>Required?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>@parameter</td>
<td>No</td>
<td>Specifies an alias which can be used to configure this parameter from the POM. This is primarily useful to improve user-friendliness, where Mojo field names are not intuitive to the user or are otherwise not conducive to configuration via the POM.</td>
</tr>
<tr>
<td></td>
<td>alias=&quot;myAlias&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>configuration</td>
<td>@component</td>
<td>No</td>
<td>Populates the field with an instance of a Plexus component. This is like declaring a requirement in a Plexus component. The default requirement will have a role equal to the declared type of the field, and will use the role hint &quot;default&quot;. You can customise either of these by providing a role and/or roleHint parameter. e.g. @component role=&quot;org.apache.maven.artifact.ArtifactHandler&quot; roleHint=&quot;ear&quot;. <strong>Note:</strong> This is identical to the deprecated form of parameter: @parameter expression=&quot;${component.yourpackage.YourComponentClass#roleHint}&quot;.</td>
</tr>
<tr>
<td></td>
<td>role=&quot;...&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>roleHint=&quot;...&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
configuration

maven-plugin-plugin 2.x:  
@parameter
expression="${aSystemProperty}
default-value="${anExpression}"

maven-plugin-plugin 3.x:
@parameter
property="aSystemProperty"
default-value="${anExpression}"

Specifies the expressions used to calculate the value to be injected into this parameter of the Mojo at buildtime.

The expression given by default-value is commonly used to refer to specific elements in the POM, such as 
${project.resources},
which refers to the list of resources meant to accompany the classes in the resulting JAR file. Of course, the default value need not be an expression but can also be a simple constant like true or 1.5. And for parameters of type String one can mix expressions with literal values, e.g.

```
${project.artifactId}-${project.version}-special.
```

The system property given by property in maven-plugin-plugin 3.x or expression in maven-plugin-plugin 2.x enables users to override the default value from the command line via -DaSystemProperty=value.

NOTE: If neither default-value nor property or expression are specified, the parameter can only be configured from the POM. The use of ‘$‘ and ‘}’ in default value is required to delimit actual expressions which may be evaluated.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Annotation</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editable</td>
<td>@readonly</td>
<td>No</td>
<td>Specifies that this parameter cannot be configured directly by the user (as in the case of POM-specified configuration). This is useful when you want to force the user to use common POM elements rather than plugin configurations, as in the case where you want to use the artifact's final name as a parameter. In this case, you want the user to modify <code>&lt;build&gt;&lt;finalName/&gt;&lt;/build&gt;</code> rather than specifying a value for finalName directly in the plugin configuration section. It is also useful to ensure that - for example - a List-typed parameter which expects items of type Artifact doesn't get a List full of Strings. <strong>NOTE:</strong> Specification of this annotation flags the parameter as non-editable; there is no true/false value.</td>
</tr>
<tr>
<td>required</td>
<td>@required</td>
<td>No</td>
<td>Whether this parameter is required for the Mojo to function. This is used to validate the configuration for a Mojo before it is injected, and before the Mojo is executed from some half-state. <strong>NOTE:</strong> Specification of this annotation flags the parameter as required; there is no true/false value.</td>
</tr>
<tr>
<td>description</td>
<td>none (detected)</td>
<td>No</td>
<td>The description of this parameter's use inside the Mojo. <em>Using the toolset, this is detected as the Javadoc description for the field.</em> <strong>NOTE:</strong> While this is not a required part of the parameter specification, it SHOULD be provided to enable future tool support for browsing, etc. and for clarity.</td>
</tr>
<tr>
<td>name</td>
<td>none (detected)</td>
<td>Yes</td>
<td>The name of the parameter, to be used in configuring this parameter from the Mojo's declared defaults (discussed below) or from the POM. Using the toolset, this is detected as the Java field name.</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>-----</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>type</td>
<td>none (detected)</td>
<td>Yes</td>
<td>The Java type for this parameter. This is used to validate the result of any expressions used to calculate the value which should be injected into the Mojo for this parameter. Using the toolset, this is detected as the class of the Java field corresponding to this parameter.</td>
</tr>
<tr>
<td>deprecated</td>
<td>@deprecated</td>
<td>No</td>
<td>Specify the version when the Mojo was deprecated to the API. Similar to Javadoc deprecated. This will trigger a warning when a user tries to configure a parameter marked as deprecated.</td>
</tr>
<tr>
<td>since</td>
<td>@since &lt;since-text&gt;</td>
<td>No</td>
<td>Specify the version when the Mojo was added to the API. Similar to Javadoc since.</td>
</tr>
</tbody>
</table>

The final component of a plugin descriptor is the dependencies. This enables the plugin to function independently of its POM (or at least to declare the libraries it needs to run). Dependencies are taken from the runtime scope of the plugin's calculated dependencies (from the POM). Dependencies are specified in exactly the same manner as in the POM, except for the <scope> element (all dependencies in the plugin descriptor are assumed to be runtime, because this is a runtime profile for the plugin).

### 58.4 Plugin Tools

By now, we've mentioned the plugin tools several times without telling you what they are or how to use them. Instead of manually writing (and maintaining) the metadata detailed above, Maven ships with some tools to aid in this task. In fact, the only thing a plugin developer needs to do is declare his project to be a plugin from within the POM. Once this is done, Maven will call the appropriate descriptor generators, etc. to produce an artifact that is ready for use within Maven builds. Optional metadata can be injected via Javadoc annotation (and possibly JDK5 annotations in the future) as described above, enabling richer interactions between the Mojo and the user. The section below describes the changes to the POM which are necessary to create plugin artifacts.
58.5 Project Descriptor (POM) Requirements

From the POM, Maven plugin projects look quite similar to any other project. For pure Java plugins, the differences are even smaller than for script-based plugins. The following details the POM elements which are necessary to build a Maven plugin artifact.

<table>
<thead>
<tr>
<th>POM Element</th>
<th>Required for Java Mojos?</th>
<th>Sample Declaration</th>
<th>Notes</th>
</tr>
</thead>
</table>
| packaging         | Yes                      | `<packaging>
  maven-plugin </packaging>` | The POM must declare a packaging element which describes this project as a Maven plugin project. |
| scriptSourceDirectory | No                        | `<scriptSourceDirectory>
  src/main/
  scripts </scriptSourceDirectory>` | In the case of script-based Mojos (which are not covered in detail within this document), the POM must include an additional element to distinguish script sources from (optional) Java supporting classes. This element is scriptSourceDirectory, inside the build section. This directory is included in the list of resources which accompany any compiled code in the resulting artifact. It is specified separately from the resources in the build section to denote its special status as an alternate source directory for scripts. |

After making the changes above, the developer can simply call

```
mvn install
```

to install the plugin to the local repository. (Any of the other standard lifecycle targets like package, deploy, etc. are also available in like fashion.)

58.6 IDE integration

If you're using JetBrains IntelliJ IDEA to develop your plugin, you can use the following to configure the javadoc annotations as live templates.

1. Download this file, and place it in `~/IntelliJ IDEA/config/templates`
2. (re)startup IntelliJ IDEA (templates are loaded on startup)
3. add the following list to Settings -> IDE -> Errors -> General -> Unknown javadoc tags -> Additional javadoc tags

```
  aggregator, execute, goal, phase, requiresDirectInvocation, requiresProject, requiresReports, requiresOnline, parameter, component, required, readonly
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
58.7 Resources

This section simply gives a listing of pointers for more information.

- QDox Project (Javadoc annotations) [link]
- Plexus Project (Plexus container) [link]
- Maven Plugin API [link]
- MojoDescriptor API [link]
59 Maven Repository Centre

59.1 Maven Central Repository

This documentation is for those that need to use or contribute to the Maven central repository. This includes those that need dependencies for their own build or projects that wish to have their releases added to the Maven central repository, even if they don't use Maven as their build tool.

⚠️ Discontinuing support for TLSv1.1 and below as of June 15th 2018 and Discontinuing support for HTTP as of January 15th 2020

- Maintaining your Metadata - Information for third-party projects
- Guide to uploading artifacts - How to get things uploaded to the central repository
- Fixing Central Metadata - How to fix issues in content already uploaded
See also the Introduction to Repositories.
60 Maven Developer Centre

60.1 Maven Developer Centre
This documentation centre is for people who are Maven developers, or would like to contribute.
If you cannot find your answers here, feel free to ask the Maven Developer List.

60.1.1 Contributors Resources
- Guide to helping with Maven
- Developing Maven
- Building Maven
- Source Code
- Continuous Integration
- Common Bugs and Pitfalls
- Apache Maven Project Roles

60.1.2 Committers Resources

60.1.2.1 General Resources
- Guide for new Maven committers
- Committer Environment
- Committer Settings
- Retirement Plan for Plugins
- Maven Dependency Policies
- Maven Plugins and Components Compatibility Plan

60.1.3 Developers Conventions
There are a number of conventions used in the Maven projects, which contributors and developers alike should follow for consistency's sake.
- Maven Code Style And Conventions
- Maven JIRA Convention
- Maven Git Convention

Note: If you cannot find your answers here, feel free to ask the Maven Developer List.

60.1.4 Making Releases
- Making GPG Keys
- Release Process

60.1.5 Maven Website
- Deploy Maven Website

60.1.6 Other Resources
- ASF Development Infrastructure Information
• About the Apache Software Foundation
61 Developing Maven

61.1 Developing Maven

This document describes how to get started developing Maven itself. There is a separate page describing how to build Maven.

61.1.1 Finding some work to do

First of all you need something to work on! Issues can be found in several JIRA projects. Another good place to look for work is the Up for grabs list. This list contains relatively simple issues that can be worked on without a lot of prerequisite knowledge.

When you find a issue you would like to work on, add a comment in the issue log so the core developers and other people looking for work know that someone is already working on it.

61.1.2 Where's the source?

See https://maven.apache.org/scm.html for information. The Maven project uses the Apache GitBox Repositories, and all of them are dual-mirrored to GitHub.

61.1.3 Don't forget tests!

You will find many unit tests. If at all possible, create or modify a unit test to demonstrate the problem, and then validate your fix.

If you need to mock a class to write a test, use the Mockito framework. Parts of the Maven codebase predate Mockito so you will encounter existing tests that use EasyMock, PowerMock, and JMock. However, all newly written mocks should use Mockito, even if this means a module or a single class uses multiple mocking frameworks. If an existing test class has complicated legacy mock setup, you can add new Mockito based tests in a new test class. There is no requirement that all tests for a single model class must be in the same test class. It is OK to have multiple test classes per model class.

If the problem case can't be set up in the unit tests, add an integration test. Before submitting a patch, in any case, you should run all of the integration tests. The tests require an empty local repository. See Core IT Suite documentation for more details.

61.1.4 Creating and submitting a patch

The most convenient way is to create a GitHub fork from the Git repository you are working with. When you have either completed an issue or just want some feedback on the work you have done, create a pull request. We have a couple of guidelines when submitting contributions:

- Create your branch from master, not from a tag. Otherwise, your patch is outdated the moment you create it and might not be applicable to the development head.

- If this was a new piece of work without a JIRA issue, create a JIRA issue for it now.

- Name the branch after the issue number; the branch name would start with <jira-project-id>-<ticket-id>.

- Push your branch with the commit(s) to your fork.

- Create a pull request to submit your contribution. Shortly after, someone will review the pull request and give you feedback on it.

A short note:

- Make sure that you follow our code style, see Further Links.
61.1.5 Pull request acceptance criteria

There are a number of criteria that a pull request will be judged on:

- Whether it works and does what is intended. This one is probably obvious!
- Whether it fits the spirit of the project. Some pull requests may be rejected as they take the project in a different direction than the current development community has chosen. This is usually discussed on an issue well before a pull request is contributed, so if you are unsure, discuss it there or on the mailing lists first. Feel free to continue discussing it (with new justification) if you disagree, or appeal to a wider audience on the mailing lists.
- Whether it contains tests. It is expected that any pull request relating to functionality will be accompanied by unit tests and/or integration tests. It is strongly desired (and will be requested) for bug fixes too, but will not be the basis for not applying it. At a bare minimum, the change should not decrease the amount of automated test coverage. As a community, we are focusing on increasing the current coverage, as there are several areas that do not receive automated testing.
- Whether it contains documentation. All new functionality needs to be documented for users, even if it is very rough for someone to expand on later. While rough is acceptable, incomplete is not. As with automated testing, as a community we are striving to increase the current coverage of documentation.

Above all, don't be discouraged. These are the same requirements the current committers should hold each other to as well. And remember, your contributions are always welcome!

61.1.6 Related Projects

Maven has a few dependencies on other projects:

- **Plexus**
  
Plexus is a full-fledged container supporting different kinds of component lifecycles. Its native lifecycle is like any other modern IoC container, using field injection of both requirements and configuration. All core Maven functionality are Plexus components.

  You can read more about Plexus.

- **Modello**
  
Modello is a simple tool for representing an object model and generating code and resources from the model. Maven uses Modello to generate all Java objects, XML readers and writers, XML Schema, and HTML documentation.

  You can read more about Modello.

- **Mojo**
  
"Mojo" is really two things when it comes to Maven: it is both Maven's plug-in API and also a separate MojoHaus project hosting a lot of plugins.

  The MojoHaus Project is a plugin forge for non-core Maven plugins. There is also a lower bar for becoming a part of the project.

61.1.7 Sub Projects

- **Maven Surefire**
  
Surefire is a testing framework. It can run regular JUnit tests so you won't have to change anything in your code to use it. It supports scripting tests in BeanShell and Jython and has special "batteries" for writing acceptance and functional tests for the web and for testing XML-RPC code.

  You can read more about Surefire.

- **Maven Doxia**
Doxia is Maven's documentation engine. It has a sink and parser API that can be used to plug in support for input and output documents.

You can read more about Doxia and the currently supported document formats.

- **Maven SCM**
  Maven SCM (Source Control Management) is a reusable API which is independent of Maven itself. It is used by the SCM related Maven Plugins. The core part of Maven doesn't depend on Maven SCM.
  You can read more about Scm.

- **Maven Wagon**
  Maven Wagon is a standalone API that deals with transporting files and directories. Maven Core uses the Wagon API to download and upload artifacts and artifact metadata. The site plug-in uses it to publish the site.
  You can read more about Wagon.

### 61.1.8 Further Links

- [Maven Code Style And Code Convention](#)
- [Maven JIRA Convention](#)
62 Building Maven

62.1 Building Maven

62.1.1 Why would I want to build Maven?
Building Maven yourself is for one of two reasons:

- to try out a bleeding edge feature or bugfix (issues can be found in JIRA),
- to fix a problem you are having and submit a patch to the developers team.

62.1.2 Checking out the sources
All of the source code for Maven and its related libraries is in managed in the ASF source code repositories: for details, see https://maven.apache.org/scm.html.

62.1.3 Building Maven

62.1.3.1 Building a Maven Plugin or Component
Building a Maven plugin or component is like any Maven build:

```
 mvn install
```

62. Running Integration Tests
Before submitting a patch, it is advised to run the integration tests, which are available in the run-its profile:

```
 mvn -Prun-its install
```

62.1.3.2 Building Maven core
Until Maven 3.3, Maven core build could be boostrapped with an Ant build. This bootstrap has been removed in Maven 3.5: you need a pre-built Maven to build Maven from source.

To do this, run from the source directory:

```
 mvn install
```

The assemblies will be created in apache-maven, and can be manually unzipped to the location where you'd like the resulting Maven installed.

If you want to have the resulting Maven directly copied to a directory, you can use the distributionTargetDir property:

```
mvn -DdistributionTargetDir="$HOME/app/maven/apache-maven-SNAPSHOT" install
```

62. Running the full Maven core integration tests
Before checking in a change or submitting a patch to Maven core, it is required to run the core integration tests. Using your local build of Maven, run:

```
 mvn test -Prun-its
```

Consult Core IT Suite documentation for more options.
63 Committer Environment

63.1 Introduction
This document is intended to set up the Maven committer environment.

63.2 Source File Encoding
When editing source files, make sure you use the right file encoding. For the Maven project, UTF-8 has been chosen as the default file encoding. UTF-8 is an encoding scheme for the Unicode character set and as such allows to encode all characters that Java can handle. The source files should not contain the byte order mark (BOM). There can be exceptions to this general rule, e.g. properties files are usually encoded using ISO-8859-1 as per the JRE API, so please keep this in mind, too.

63.3 Subversion Configuration
Before committing files in subversion repository, you need to read the Committer Subversion Access document and you must set your svn client with this properties file: svn-eol-style.txt

63.4 Maven Code Style
The following sections show how to set up the code style for Maven in IDEA and Eclipse. It is strongly preferred that patches use this style before they are supplied.

63.4.1 IntelliJ IDEA 4.5+
Download maven-idea-codestyle.xml and copy it to ~/Library/Preferences/IntelliJIdeaxx/codestyles then restart IDEA (with xx your idea version). On Windows, try C:\Documents and Settings\<username>\.IntelliJIDEA\config\codestyles
After this, restart IDEA and open the settings to select the new code style.

63.4.2 Eclipse 3.2+
Download maven-eclipse-codestyle.xml.
After this, select Window > Preferences, and open up the configuration for Java > Code Style > Code Formatter. Click on the button labeled Import... and select the file you downloaded. Give the style a name, and click OK.

63.5 Useful software
The Maven Team uses several software. Here is a partial list:

- Cygwin: collection of free software tools to allow various versions of Microsoft Windows to act somewhat like a Unix system
- GnuPG: GNU Privacy Guard.
64 Committer Settings

64.1 Introduction
This document is intended to set up the Maven committer settings, i.e. the ${user.home}/.m2/settings.xml.

64.1.1 Enable Apache Servers
Maven uses several servers configuration to deploy snapshots and releases on the Apache servers. You need to tell to Maven what your Apache username is.

It is highly recommended to use Maven’s password encryption capabilities for your passwords.

```xml
<settings>
  ...
  <servers>
    <!-- To publish a snapshot of some part of Maven -->
    <server>
      <id>apache.snapshots.https</id>
      <username> <!-- YOUR APACHE LDAP USERNAME --> </username>
      <password> <!-- YOUR APACHE LDAP PASSWORD --> </password>
    </server>
    <!-- To stage a release of some part of Maven -->
    <server>
      <id>apache.releases.https</id>
      <username> <!-- YOUR APACHE LDAP USERNAME --> </username>
      <password> <!-- YOUR APACHE LDAP PASSWORD --> </password>
    </server>
    ...
  </servers>
  ...
</settings>
```

64.1.2 Enable sending announcement e-mails
To be able to send out announcements of Maven releases you need to add a couple of properties to the apache-release profile.

```xml
<settings>
  ...
  <profiles>
    <profile>
      <id>apache-release</id>
      <properties>
        <apache.availid> <!-- YOUR APACHE LDAP USERNAME --> </apache.availid>
        <smtp.host> <!-- YOUR SMTP SERVER --> </smtp.host>
      </properties>
    </profile>
    ...
  </profiles>
</settings>
```
65 Maven Code Style And Conventions

65.1 Maven Code Style And Code Conventions
This document describes how developers and contributors should format code in order to improve consistency, readability and maintainability.

65.1.1 Generic Code Style And Convention
All working files (java, xml, others) should respect the following conventions:

- **License Header**: Always add the current [ASF license header](https://www.apache.org/licenses/LICENSE-2.0.html) in all files checked into the source code repository.
- **Trailing Whitespace**: Remove all trailing whitespace. If you use Eclipse, you can use the [Anyedit Eclipse Plugin](https://marketplace.eclipse.org/content/anyedit-plugin)

and the following style:

- **Indentation**: *Never* use tabs!
- **Line wrapping**: Always use a 120-column line width.

**Note**: The specific styles and conventions, listed in the next sections, can override these generic rules.

65.1.2 Java

65.1.2.1 Java Code Style
The Maven style for Java is mainly:

- **White space**: One space after control statements and between arguments (e.g. `if (foo)` instead of `if(foo)`), `myFunc(foo, bar, baz)` instead of `myFunc(foo, bar, baz)`). No spaces after methods names (i.e. `void myMethod(), myMethod( "foo" )`)
- **Indentation**: Always use 4 space indents and *never* use tabs!
- **Blocks**: Always enclose with a new line brace.
- **Line wrapping**: Always use a 120-column line width for Java code and Javadoc.
- **Readiness**: Specify code grouping members, if needed. For instance in a Mojo class, you could have:

```java
public class MyMojo {
    // ----------------------------------------------------------------------
    // Mojo components
    // ----------------------------------------------------------------------
    /**
     * Artifact factory.
     */
    private ArtifactFactory artifactFactory;
    ...
    // ----------------------------------------------------------------------
    // Mojo parameters
    // ----------------------------------------------------------------------
    /**
     * The POM.
     */
```
* @parameter expression="${project}" *
* @required */
private MavenProject project;
...
// Mojo options
...
// Public methods
/**
 * @inheritDoc
 */
public void execute()
    throws MojoExecutionException {
    ...
}
// Protected methods
...
// Private methods
...
// Static methods
...
}

The following sections show how to set up the code style for Maven in IDEA and Eclipse. It is strongly preferred that patches use this style before they are applied.

65.1 IntelliJ IDEA

Download maven-idea-codestyle.xml and import it into IDEA using File > Settings > Editor > Code Style > Gear icon > Import Scheme > IntelliJ IDEA Code Style XML.

65.2 Eclipse

Download maven-eclipse-codestyle.xml.

After this, select Window > Preferences, and open up the configuration for Java > Code Style > Code Formatter. Click on the button labeled Import... and select the file you downloaded. Give the style a name, and click OK.

65.1.2.2 Java Code Convention

For consistency reasons, our Java code convention is mainly:

- **Naming**: Constants (i.e. static final members) should always be in upper case. Use short, descriptive names for classes and methods.

- **Organization**: Avoid using public inner classes. Prefer interfaces instead of default implementation.
• **Modifier**: Avoid using final modifier on all fields and arguments. Prefer using private or protected fields instead of public fields.

• **Exceptions**: Throw meaningful exceptions to make debugging and testing easier.

• **Documentation**: Document public interfaces well, i.e. all non-trivial public and protected functions should include Javadoc that indicates what they do.

• **Testing**: All non-trivial public classes should have corresponding unit or integration tests.

65.1.2.3 Java Code Convention - import layouts

For consistency reasons, Java imports should be organized as:

• import javax.*
• blank line
• import java.*
• blank line
• import all other imports
• blank line
• import static all other imports

all imports in each group should be sorted alphabetically.

65.1.2.4 JavaDoc Convention

TO BE DISCUSSED

65.1.3 XML

65.1.3.1 XML Code Style

The Maven style for XML files is mainly:

• **Indentation**: Always use 2 space indents, unless you're wrapping a new XML tags line in which case you should indent 4 spaces.

• **Line Breaks**: Always use a new line with indentation for complex XML types and no line break for simple XML types. Always use a new line to separate XML sections or blocks, for instance:

```
<aTag>
  <simpleType>This is a simple type</simpleType>
  <complexType>
    <simpleType>This is a complex type</simpleType>
  </complexType>
</aTag>
```

In some cases, adding comments could improve the readability of blocks, for instance:

```
<!-- Simple XML documentation -->
```

or

```
<!-- Block documentation -->
```

65.1.3.2 Generic XML Code Convention

No generic code convention exists yet for XML files.
65.1.3.3 POM Code Convention

The team has voted during the end of June 2008 to follow a specific POM convention to ordering POM elements. The consequence of this vote is that the Maven project descriptor is no more considered as the reference for the ordering.

The following is the recommended ordering for all Maven POM files:

```xml
  <modelVersion/>
  <parent/>
  <groupId/>
  <artifactId/>
  <version/>
  <packaging/>
  <name/>
  <description/>
  <url/>
  <inceptionYear/>
  <organization/>
  <licenses/>
  <developers/>
  <contributors/>
  <mailingLists/>
  <prerequisites/>
  <modules/>
  <scm/>
  <issueManagement/>
  <ciManagement/>
  <distributionManagement/>
  <properties/>
  <dependencyManagement/>
  <dependencies/>
  <repositories/>
  <pluginRepositories/>
  <build/>
  <reporting/>
  <profiles/>
</project>
```

Comments:

1. The `<project/>` element is always on one line.
2. The blocks are voluntary separated by a new line to improve the readingness.
3. The dependencies in `<dependencies/>` and `<dependencyManagement/>` tags have no specific ordering. Developers are free to choose the ordering, but grouping dependencies by topics (like groupId i.e. org.apache.maven) is a good practice.

Note: There exist two alternatives to change order of a pom file Tidy Maven Plugin or the Sortpom Maven Plugin.

65.1.3.4 XDOC Code Convention

For consistency and readability reasons, XDOC files should respect:

- **Metadata**: Always specify metadata in the `<properties/>` tag.
- **Sections**: Always use a new line with indentation for `<section/>` tags.
65.1.3.5 FML Code Convention

For readability reasons, FML files should respect:

• **FAQ**: Always use a new line with indentation for `<faq/>` tags.
66 Maven JIRA Convention

66.1 Maven JIRA Conventions
This document describes how Maven developers should use JIRA, our issue management system.

66.1.1 When To Create a JIRA Issue?
This section discusses when to create a JIRA issue versus just committing a change in Git (eventually through a PR).

- **Minor changes** such as code reformatting, documentation fixes, etc. that aren't going to impact other users can be committed without a JIRA issue.
- **Larger changes** such as bug fixes, API changes, significant refactoring, new classes, and pretty much any change of more than 100 lines, should have JIRA tickets.

Creating a JIRA issue and referring it in the commit comments simplifies tracking the changes that happen in a release, using JIRA automatic release notes creation.

66.1.2 How To Use Issue Details?
This section presents some conventions about the issue fields.

66.1.2.1 Priority
Committers have the responsibility to realign priority by editing the issue.

*Reasoning*: having a correct release note

66.1.2.2 Assignee
Committers can assign an issue to a specific committer that person seems to be well placed to address it.

66.1.2.3 Component/s
Committers have the responsibility to specify the correct component by editing the issue.

*Reasoning*: having a correct release note.

66.1.2.4 Affects Version/s
By default, the Maven team considers that an issue which affects a given version also affects preceding versions. For example, an issue that affects Maven 3.6.3 also affects 3.6.0, 3.6.1, 3.6.2. If it is a regression, the committers should specify the affected versions.

*Reasoning*: having a correct release note.

66.1.2.5 Fix Version/s
Update to correct version after merging to master.

66.1.2.6 Time Tracking
The Maven team doesn't use this. Committers can if it helps them.

66.1.3 Further Links
- JIRA Documentation
• What is an Issue?
• What is a project?
67 Making GPG Keys

67.1 Introduction

You need to add your GPG keys in https://svn.apache.org/repos/asf/maven/project/KEYS before a release. Here are some useful GnuPG commands to generate your Keys.

67.1.1 gpg --gen-key

```
> gpg --gen-key

gpg (GnuPG) 1.4.5; Copyright (C) 2006 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.
gpg: keyring `C:/Documents and Settings/Siveton Vincent/Application Data/gnupg/secring.gpg'
created

gpg: keyring `C:/Documents and Settings/Siveton Vincent/Application Data/gnupg/pubring.gpg'
created

Please select what kind of key you want:
  (1) DSA and Elgamal (default)
  (2) DSA (sign only)
  (5) RSA (sign only)
Your selection? 1
DSA keypair will have 1024 bits.
ELG-E keys may be between 1024 and 4096 bits long.
What keys size do you want? (2048) 2048
Requested key size is 2048 bits
Please specify how long the key should be valid.
  0 = key does not expire
  <n> = key expires in n days
  <n>w = key expires in n weeks
  <n>m = key expires in n months
  <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y

You need a user ID to identify your key; the software constructs the user ID from the Real Name, Comment and Email Address in this form:
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: Vincent Siveton
Email address: vsiveton@apache.org

Comment:
You selected this USER-ID:
"Vincent Siveton <vsiveton@apache.org>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
You don't want a passphrase - this is probably a *bad* idea!
I will do it anyway. You can change your passphrase at any time,
using this program with the option "--edit-key".

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
```
We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

```
+............................+............................+............................+
++..+..........................++..+..........................++..+..........................+
+............................+............................+............................+
```

gpg: C:/Documents and Settings/Siveton Vincent/Application Data/gnupg\trustdb.gpg: trustdb created

gpg: key 07DDB702 marked as ultimately trusted

gpg: checking the trustdb

gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model

gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u

```
pub   1024D/07DDB702 2006-10-10
Key fingerprint = 71F6 F555 8A61 71C4 330D B868 84F4 D470 07DD B702
```

```
sudo gpg --list-sigs
```

```
---BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.5 (MingW32)
mQGiBEUrnAsRBACQDiyGc1QmEKLQo9IznB8otGPEBzqOozT5tsip5MB30f6/Mc/uuLxkJLdn7u1L3o1XOtcCeLJq38gHvrufNVRN6s/jufJKd5sLH8UJ18PbKuo/91Kg1lejtcYUDDU48czRceOehqds5ANH8ydN6Pda276NGPYPyfxt7kKgH/nPwCGqoZK1g2r1oMMymVfAfbqda17y/sd/1csf09rjtHExOuyhmjCGjSrzwLbHUgKPhQ37PajECH0wc6HnVHHEEcrpV1+Uj2zVrcVhzUoP+3r5HAAqgErfzKX8AweCqj1jRf64KU
sjvto31lg2kZy17y8k9y4lKrkUspu8uw903pKnW/QEL4g4vVPaEypVlVsd6c+uReu
V09h/ATW7/GpzSbQmvjnIRQ7GVHbQcXaANwrS6NmgXYxafN999dfHV+eUieU6
rvMPM9c0o/bYdYEk2skrXw2MmXs5SgxsXTQg4DxwxpYPFf/dGUsFMobnVX12ZG8l
JE19cmC9lc3L1s0mDv21lg3e3YXxsn4AEISU+0bbq017rqALQVlVmlV2Y2uCB
axZlDg9uIDz2c2122ZXRvbkhBhCGFjaGUBu3JnPcqbBMRaAgBQFJ5WLaHsBgsJ
CAcDaGqXaggADBhYCAwECVgEC4AACgkQhPTUcAfxtwLg/3gCB/B/Ylaf8hXzgCICS
d2r6zCj02jIAoLKD/RtkkerNIntYzrUdUJi1JKBsRuQINEBunBqCQAD1+SxBdL
1X2vtgXQGrzMmnJ1V/k/w4TPA/8weJku2lPSPINOjInmqESeuhvClOooyfcBhV1R
PU2hKZLPEKfJlFtpGNK19oTO/CoQn+S9JLW41FoumsBafiYSSRpaUxyx2J6cUqf
uWrk/T9FmpDw4Yzmt96e/ven/4ie1S5coxpu0MB9ynTaShts9ypjHj1jEg0b7bWQeU
elOOLiHhgyKyjJUmItLMxlSx384eq0uAvYb/rb/QOWWLPsVr/2o1kzvcUCpgo3h
kgOxGxsc5qD836Hf5HFKCERYS3FayF5+/jM+LjZL3y3u7uOOP6yu1KXWxp0dBn
WHO1JQfcd2TAAFMFCADdiJlg7L+8de6JzsaEDuKerKqWITEWA99N1knakzKdUUOp
WrKxqgig16APAbx0fG4VbDadaM6+m+n5JDMybQcOFdNg9x17/j7ykzkmv8Jkt8CtW4z
```

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
You need to append this result to https://svn.apache.org/repos/asf/maven/project/KEYS.
You also need to upload your key to the public server: http://pgp.mit.edu/ by copying the same you appended in the text field and submit. You can ensure by searching your email in key search engine.

67.1.3 gpg --fingerprint

```bash
>gpg --fingerprint vsiveton
pub  1024D/07DBB702 2006-10-10
    Key fingerprint = 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000
uid                  Vincent Siveton <vsiveton@apache.org>
sub   2048g/02814A59 2006-10-10
```

Go to https://id.apache.org, log in and fill OpenPGP Public Key Primary Fingerprint: with the value of Key fingerprint.
You can read more about Checksums And Signatures and How to Generate PGP Signatures With Maven.
68 External Resources

68.1 Books on Maven

Apache Maven Cookbook
- **Covers:** Maven 3
- **Published:** Packt Publishing (August 2015)
- **Authors:** Raghuram Bharathan
- **Buy the Book:** Packt, Amazon

Apache Maven 3.0 Cookbook
- **Covers:** Maven 3
- **Published:** Packt Publishing (August 2011)
- **Authors:** Srirangan
- **Buy the Book:** Packt, Amazon

Getting Started with Apache Maven [Video]
- **Covers:** Maven 2.2.1, and above
- **Published:** Packt Publishing (September 30, 2013)
- **Authors:** Russell Gold
- **Buy the Video:** Packt
68.2 Articles on Maven

If you are writing an article on Maven we suggest contacting the developers on the mailing list as we would be happy to provide feedback to help ensure accuracy in your article. Just ping us on the dev mailing list to get in touch.

<table>
<thead>
<tr>
<th>Title</th>
<th>Publisher</th>
<th>Author</th>
<th>Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Maven Toolchains</td>
<td></td>
<td>Maarten Mulders</td>
<td>March 2021</td>
</tr>
<tr>
<td>Customise the Maven Release process</td>
<td></td>
<td>Maarten Mulders</td>
<td>January 2020</td>
</tr>
<tr>
<td>Create a Customized Build Process in Maven</td>
<td></td>
<td>John Casey</td>
<td>August 2009</td>
</tr>
<tr>
<td>Maven: mas que una herramienta de construccion (in Spanish)</td>
<td></td>
<td>Manuel Recena Soto</td>
<td>June 2009</td>
</tr>
<tr>
<td>Introduction to m2eclipse</td>
<td>TheServerSide</td>
<td>Tim O’Brien, Bruce Snyder, Eugene Kuleshov</td>
<td>July 2008</td>
</tr>
<tr>
<td>Maven 2.x (in Turkish)</td>
<td>Anadolu Universitési</td>
<td>Mustafa Sait Özен</td>
<td>August 2007</td>
</tr>
<tr>
<td>Setting up the Internal Repository</td>
<td>The Server Side</td>
<td>Avneet Mangat</td>
<td>June 2007</td>
</tr>
<tr>
<td>Maven - Menos mal que has venido (in Spanish)</td>
<td>Universidad de Sevilla</td>
<td>Manuel J. Recena Soto</td>
<td>6 November 2006</td>
</tr>
<tr>
<td>FAQ for Maven and Continuum (in French)</td>
<td>Developpez.com</td>
<td>Eric Reboisson</td>
<td>11 October 2006</td>
</tr>
<tr>
<td>Java Posse #070 - Interview with Brett Porter of Maven</td>
<td>Java Posse</td>
<td>Tor Norbye, Carl Quinn, Dick Wall, Joe Nuxoll, Brett Porter</td>
<td>18 July 2006</td>
</tr>
<tr>
<td>The Maven 2 POM demystified</td>
<td>JavaWorld</td>
<td>Eric Redmond</td>
<td>29 May 2006</td>
</tr>
<tr>
<td>Maven: Building Complex Systems</td>
<td>Dr.Dobb’s</td>
<td>Gigi Sayfan</td>
<td>21 April 2006</td>
</tr>
<tr>
<td>Get the most out of Maven site generation</td>
<td>JavaWorld</td>
<td>John Ferguson Smart</td>
<td>27 February 2006</td>
</tr>
<tr>
<td>An introduction to Maven 2</td>
<td>JavaWorld</td>
<td>John Ferguson Smart</td>
<td>5 December 2005</td>
</tr>
<tr>
<td>Building J2EE Projects with Maven</td>
<td>OnJava</td>
<td>Vincent Massol</td>
<td>7 September 2005</td>
</tr>
<tr>
<td>Master and Commander by Julien Dubois</td>
<td>Oracle</td>
<td>Julien Dubois</td>
<td>November 2004</td>
</tr>
<tr>
<td>Apache's Maven Comes of Age (Coverage of the release of Maven 1.0)</td>
<td>internetnews.com</td>
<td>Sean Michael Kerner</td>
<td>15 July 2004</td>
</tr>
</tbody>
</table>

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.
69 Team

69.1 Project Team

A successful project requires many people to play many roles. Some members write code or documentation, while others are valuable as testers, submitting patches and suggestions.

The project team is comprised of Members and Contributors. Members have direct access to the source of a project and actively evolve the code-base. Contributors improve the project through submission of patches and suggestions to the Members. The number of Contributors to the project is unbounded. Get involved today. All contributions to the project are greatly appreciated.

69.1.1 Members

The following is a list of developers with commit privileges that have directly contributed to the project in one way or another.

<table>
<thead>
<tr>
<th>Image</th>
<th>Id</th>
<th>Name</th>
<th>Email</th>
<th>Organization</th>
<th>Roles</th>
<th>Time Zone</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rfscholte</td>
<td>Robert Scholte</td>
<td><a href="mailto:rfscholte@apache.org">rfscholte@apache.org</a></td>
<td>-</td>
<td>PMC Chair</td>
<td>Europe/Amsterdam</td>
<td>twitter=@rfscholte</td>
</tr>
<tr>
<td></td>
<td>aheritier</td>
<td>Arnaud Héritier</td>
<td><a href="mailto:aheritier@apache.org">aheritier@apache.org</a></td>
<td>-</td>
<td>PMC Member</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>andham</td>
<td>Anders Hammar</td>
<td><a href="mailto:andham@apache.org">andham@apache.org</a></td>
<td>-</td>
<td>PMC Member</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>baerrach</td>
<td>Barrie Treloar</td>
<td><a href="mailto:baerrach@apache.org">baerrach@apache.org</a></td>
<td>-</td>
<td>PMC Member</td>
<td>Australia/Adelaide</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>bimargulies</td>
<td>Benson Margulies</td>
<td><a href="mailto:bimargulies@apache.org">bimargulies@apache.org</a></td>
<td>-</td>
<td>PMC Member</td>
<td>America/New_York</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>brianf</td>
<td>Brian Fox</td>
<td><a href="mailto:brianf@apache.org">brianf@apache.org</a></td>
<td>Sonatype</td>
<td>PMC Member</td>
<td>-5</td>
<td>-</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
<td>Email</td>
<td>Role</td>
<td>Score</td>
<td>Location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>------------------------</td>
<td>---------------</td>
<td>-------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cstamas</td>
<td>Tamas Cservenak</td>
<td><a href="mailto:cstamas@apache.org">cstamas@apache.org</a></td>
<td>PMC Member</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dennisl</td>
<td>Dennis Lundberg</td>
<td><a href="mailto:dennisl@apache.org">dennisl@apache.org</a></td>
<td>PMC Member</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dkulp</td>
<td>Daniel Kulp</td>
<td><a href="mailto:dkulp@apache.org">dkulp@apache.org</a></td>
<td>PMC Member</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>evenisse</td>
<td>Emmanuel Venisse</td>
<td><a href="mailto:evenisse@apache.org">evenisse@apache.org</a></td>
<td>PMC Member</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gboue</td>
<td>Guillaume Boué</td>
<td><a href="mailto:gboue@apache.org">gboue@apache.org</a></td>
<td>PMC Member</td>
<td></td>
<td>Europe/Paris</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hboutemy</td>
<td>Hervé Boutemy</td>
<td><a href="mailto:hboutemy@apache.org">hboutemy@apache.org</a></td>
<td>PMC Member</td>
<td></td>
<td>Europe/Paris</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifedorenko</td>
<td>Igor Fedorenko</td>
<td>igor@ifedorenko@Sonatype</td>
<td>PMC Member</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvanzyl</td>
<td>Jason van Zyl</td>
<td><a href="mailto:jason@maven.org">jason@maven.org</a></td>
<td>PMC Member</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>khmarbaise</td>
<td>Karl Heinz Marbaise</td>
<td><a href="mailto:khmarbaise@apache.org">khmarbaise@apache.org</a></td>
<td>PMC Member</td>
<td>+1</td>
<td>twitter=@khmarbaise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>krosenvold</td>
<td>Kristian Rosenvold</td>
<td><a href="mailto:krosenvold@apache.org">krosenvold@apache.org</a></td>
<td>PMC Member</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkleint</td>
<td>Milos Kleint</td>
<td></td>
<td>PMC Member</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>olamy</td>
<td>Olivier Lamy</td>
<td><a href="mailto:olamy@apache.org">olamy@apache.org</a></td>
<td>PMC Member</td>
<td>Australia/Brisbane</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
<td>------------------</td>
<td>------------</td>
<td>-------------------</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>michaelo</td>
<td>Michael Osipov</td>
<td><a href="mailto:michaelo@apache.org">michaelo@apache.org</a></td>
<td>PMC Member</td>
<td>Europe/Berlin</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rgoers</td>
<td>Ralph Goers</td>
<td><a href="mailto:rgoers@apache.org">rgoers@apache.org</a></td>
<td>PMC Member</td>
<td>-8</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stephenc</td>
<td>Stephen Connolly</td>
<td><a href="mailto:stephenc@apache.org">stephenc@apache.org</a></td>
<td>PMC Member</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>struberg</td>
<td>Mark Struberg</td>
<td><a href="mailto:struberg@apache.org">struberg@apache.org</a></td>
<td>PMC Member</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tibordigana</td>
<td>Tibor Digače</td>
<td><a href="mailto:tibordigana@apache.org">tibordigana@apache.org</a></td>
<td>PMC Member</td>
<td>Europe/Bratislava</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsiveton</td>
<td>Vincent Siveton</td>
<td><a href="mailto:vsiveton@apache.org">vsiveton@apache.org</a></td>
<td>PMC Member</td>
<td>-5</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wfay</td>
<td>Wayne Fay</td>
<td><a href="mailto:wfay@apache.org">wfay@apache.org</a></td>
<td>PMC Member</td>
<td>-6</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>adangel</td>
<td>Andreas Dangel</td>
<td><a href="mailto:adangel@apache.org">adangel@apache.org</a></td>
<td>Committer</td>
<td>Europe/Berlin</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bdemers</td>
<td>Brian Demers</td>
<td><a href="mailto:bdemers@apache.org">bdemers@apache.org</a></td>
<td>Committer</td>
<td>-5</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bellingard</td>
<td>Fabrice Bellingard</td>
<td>-</td>
<td>Committer</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Full Name</td>
<td>Email</td>
<td>Role</td>
<td>Change</td>
<td>Location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------</td>
<td>---------------------------</td>
<td>--------------------</td>
<td>--------</td>
<td>----------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bentmann</td>
<td>Benjamin Bentmann</td>
<td>bentmann@sonatype</td>
<td>Committer</td>
<td>+1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chrisgwarp</td>
<td>Chris Graham</td>
<td>chrisgwarp@a</td>
<td>Committer</td>
<td>-</td>
<td>Australia/Melbourne</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dantran</td>
<td>Dan Tran</td>
<td>dantran@apa</td>
<td>Committer</td>
<td>-8</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbradicich</td>
<td>Damian Bradicich</td>
<td>dbradicich@sonatype</td>
<td>Committer</td>
<td>-5</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brett</td>
<td>Brett Porter</td>
<td>brett@apache</td>
<td>Committer</td>
<td>+10</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfabulich</td>
<td>Daniel Fabulich</td>
<td>dfabulich@ap</td>
<td>Committer</td>
<td>-8</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eolivelli</td>
<td>Enrico Olivelli</td>
<td>eolivelli@apa Diennea</td>
<td>Committer</td>
<td>-</td>
<td>Europe/Rome</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fgiust</td>
<td>Fabrizio Giustina</td>
<td>fgiust@apache openmind</td>
<td>Committer</td>
<td>+1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>godin</td>
<td>Evgeny Mandrikov</td>
<td>godin@apache SonarSource</td>
<td>Committer</td>
<td>+3</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handyande</td>
<td>Andrew Williams</td>
<td>handyande@</td>
<td>Committer</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imod</td>
<td>Dominik Bartholdi</td>
<td>imod@apache</td>
<td>Committer</td>
<td>-</td>
<td>Europe/Zurich</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Email</td>
<td>Company, City</td>
<td>Committer</td>
<td>Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------</td>
<td>---------------</td>
<td>-----------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jjensen</td>
<td>Jeff Jensen</td>
<td>-</td>
<td>Committer</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itheussl</td>
<td>Lukas Theussl</td>
<td>itheussl@apa</td>
<td>Committer</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>markh</td>
<td>Mark Hobson</td>
<td>markh@apac</td>
<td>Committer</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>martinkanters</td>
<td>Martin Kanters</td>
<td>martinkanters</td>
<td>Committer</td>
<td>Europe/Amsterdam</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mthmulders</td>
<td>Maarten Mulders</td>
<td>mthmulders@Info Support</td>
<td>Committer</td>
<td>Europe/Amsterdam</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mauro</td>
<td>Mauro Talevi</td>
<td>-</td>
<td>Committer</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfriedenhage</td>
<td>Mirko Friedenhagen</td>
<td>mfriedenhage</td>
<td>Committer</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmoser</td>
<td>Manfred Moser</td>
<td>mmoser@apac</td>
<td>Committer</td>
<td>-8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nicolas</td>
<td>Nicolas de Loof</td>
<td>-</td>
<td>Committer</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oching</td>
<td>Maria Odea B. Ching</td>
<td>-</td>
<td>Committer</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgier</td>
<td>Paul Gier</td>
<td>pgier@apache</td>
<td>Red Hat</td>
<td>Committer</td>
<td>-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>First Name</td>
<td>Email</td>
<td>Role</td>
<td>Location</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>----------------------</td>
<td>-----------</td>
<td>-------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ptahchiev</td>
<td>Petar Tahchiev</td>
<td><a href="mailto:ptahchiev@apache.org">ptahchiev@apache.org</a></td>
<td>Committer</td>
<td>+2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rafale</td>
<td>Raphaël Piéroni</td>
<td><a href="mailto:rafale@apache.org">rafale@apache.org</a></td>
<td>Committer</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>schulte</td>
<td>Christian Schulte</td>
<td><a href="mailto:schulte@apache.org">schulte@apache.org</a></td>
<td>Committer</td>
<td>Europe/Berlin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>snicoll</td>
<td>Stephane Nicoll</td>
<td><a href="mailto:snicoll@apache.org">snicoll@apache.org</a></td>
<td>Committer</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simonetripodi</td>
<td>Simone Tripodi</td>
<td><a href="mailto:simonetripodi@apache.org">simonetripodi@apache.org</a></td>
<td>Committer</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sor</td>
<td>Christian Stein</td>
<td><a href="mailto:sor@apache.org">sor@apache.org</a></td>
<td>Committer</td>
<td>Europe/Berlin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tchemit</td>
<td>Tony Chemit</td>
<td><a href="mailto:tchemit@apache.org">tchemit@apache.org</a></td>
<td>Committer</td>
<td>Europe/Paris</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmassol</td>
<td>Vincent Massol</td>
<td><a href="mailto:vmassol@apache.org">vmassol@apache.org</a></td>
<td>Committer</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slachiewicz</td>
<td>Sylwester Lachiewicz</td>
<td><a href="mailto:slachiewicz@apache.org">slachiewicz@apache.org</a></td>
<td>Committer</td>
<td>Europe/Warsaw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elharo</td>
<td>Elliotte Rusty Harold</td>
<td><a href="mailto:elharo@apache.org">elharo@apache.org</a></td>
<td>Committer</td>
<td>America/New_York</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>agudian</td>
<td>Andreas Gudian</td>
<td><a href="mailto:agudian@apache.org">agudian@apache.org</a></td>
<td>Emeritus</td>
<td>Europe/Berlin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>First Name</td>
<td>Last Name</td>
<td>Email/Username</td>
<td>Title</td>
<td>Status</td>
<td>Additional Info</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>-----------</td>
<td>----------------</td>
<td>---------</td>
<td>--------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>aramirez</td>
<td>Allan Q.</td>
<td>Ramirez</td>
<td></td>
<td></td>
<td>Emeritus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bayard</td>
<td>Henri</td>
<td>Yandell</td>
<td></td>
<td></td>
<td>Emeritus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>carlos</td>
<td>Carlos</td>
<td>Sanchez</td>
<td>carlos@apache</td>
<td>Emeritus</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chrisjs</td>
<td>Chris</td>
<td>Stevenson</td>
<td></td>
<td>Emeritus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dblevins</td>
<td>David</td>
<td>Blevins</td>
<td></td>
<td>Emeritus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dir</td>
<td>Daniel</td>
<td>Rall</td>
<td></td>
<td>Emeritus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>epunzalan</td>
<td>Edwin</td>
<td>Punzalan</td>
<td>epunzalan@e</td>
<td>Emeritus</td>
<td>-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>felipeal</td>
<td>Felipe</td>
<td>Leme</td>
<td></td>
<td>Emeritus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jdcassey</td>
<td>John</td>
<td>Casey</td>
<td>jdcassey@apa</td>
<td>Emeritus</td>
<td>-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmconnell</td>
<td>Jesse</td>
<td>McConnell</td>
<td>jmconnell@j</td>
<td>Emeritus</td>
<td>-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>joakime</td>
<td>Joakim</td>
<td>Erdfelt</td>
<td>joakime@apache</td>
<td>Emeritus</td>
<td>-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Full Name</td>
<td>Email</td>
<td>Affiliation</td>
<td>Status</td>
<td>Phone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------</td>
<td>----------------------------</td>
<td>------------------</td>
<td>---------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jruiz</td>
<td>Johnny Ruiz III</td>
<td><a href="mailto:jruiz@apache.org">jruiz@apache.org</a></td>
<td>-</td>
<td>Emeritus</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jstrachan</td>
<td>James Strachan</td>
<td>-</td>
<td>-</td>
<td>Emeritus</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jtolentino</td>
<td>Ernesto Tolentino Jr.</td>
<td>jtolentino@ap ASF</td>
<td>ASF</td>
<td>+8</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kenney</td>
<td>Kenney Westerhof</td>
<td>kenney@apa Neonics</td>
<td>-</td>
<td>Emeritus</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mperham</td>
<td>Mike Perham</td>
<td>mperham@gr IBM</td>
<td>IBM</td>
<td>-6</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ogusakov</td>
<td>Oleg Gusakov</td>
<td>-</td>
<td>-</td>
<td>Emeritus</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pschneider</td>
<td>Patrick Schneider</td>
<td>pschneider@i</td>
<td>-</td>
<td>Emeritus</td>
<td>-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rinku</td>
<td>Rahul Thakur</td>
<td>-</td>
<td>-</td>
<td>Emeritus</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shinobu</td>
<td>Shinobu Kuwai</td>
<td>-</td>
<td>-</td>
<td>Emeritus</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>smorgrav</td>
<td>Torbjorn Eikli Smorgrav</td>
<td>-</td>
<td>-</td>
<td>Emeritus</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trygvis</td>
<td>Trygve Laugstol</td>
<td>trygvis@apac ASF</td>
<td>-</td>
<td>Emeritus</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
69.1.2 Contributors

There are no contributors listed for this project. Please check back again later.
70 Mailing Lists

70.1 Project Mailing Lists

These are the mailing lists that have been established for this project. For each list, there is a subscribe, unsubscribe, and an archive link.

<table>
<thead>
<tr>
<th>Name</th>
<th>Subscribe</th>
<th>Unsubscribe</th>
<th>Post</th>
<th>Archive</th>
<th>Other Archives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maven User List</td>
<td>Subscribe</td>
<td></td>
<td></td>
<td></td>
<td>lists.apache.org</td>
</tr>
<tr>
<td>Maven Developer List</td>
<td>Subscribe</td>
<td></td>
<td></td>
<td></td>
<td>mail-archives.apache.org</td>
</tr>
<tr>
<td>Maven Issues List</td>
<td>Subscribe</td>
<td></td>
<td>-</td>
<td></td>
<td>mail-archives.apache.org</td>
</tr>
<tr>
<td>Maven Commits List</td>
<td>Subscribe</td>
<td></td>
<td>-</td>
<td></td>
<td>mail-archives.apache.org</td>
</tr>
<tr>
<td>Maven Announcements List</td>
<td>Subscribe</td>
<td></td>
<td></td>
<td></td>
<td>mail-archives.apache.org</td>
</tr>
<tr>
<td>Maven Notifications List</td>
<td>Subscribe</td>
<td></td>
<td>-</td>
<td></td>
<td>mail-archives.apache.org</td>
</tr>
</tbody>
</table>

©2021, The Apache Software Foundation • ALL RIGHTS RESERVED.


71 CI Management

71.1 Overview

This project uses Jenkins.

71.2 Access

The following is a link to the continuous integration system used by the project:

https://ci-maven.apache.org/job/Maven/job/maven-box/

71.3 Notifiers

Configuration for notifying developers/users when a build is unsuccessful, including user information
and notification mode.

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>mail</td>
<td>-</td>
<td>address=<a href="mailto:notifications@maven.apache.org">notifications@maven.apache.org</a></td>
</tr>
</tbody>
</table>