Apache Maven
Current version
User Guide

The Apache Software Foundation **2009-10-16**

Table of Contents

1 What is Maven?

1.1 Introduction

Maven, a Yiddish word meaning *accumulator of knowledge*, was originally started 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 and 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 there are several areas of concern that Maven attempts to deal with:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

1.2.1 Making the build process easy

While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does provide a lot of shielding from the details.

1.2.2 Providing a uniform build system

Maven allows a project to build using its project object model (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform build system. Once you familiarize yourself with how one Maven project builds you automatically know how all Maven projects build saving you immense amounts of time when trying to navigate many projects.

1.2.3 Providing quality project information

Maven provides plenty of 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 document created directly from source control
- Cross referenced sources
- Mailing lists
- Dependency list
- Unit test reports including coverage

As Maven improves the information set provided will improve, all of which will be transparent to users of Maven.

Other products can also provide Maven plugins to allow their set of project information alongside some of the standard information given by Maven, all still based on the POM.

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 your test source code in a separate, but parallel source tree
- Using test case naming conventions to locate and execute tests
- Have test cases setup their environment and don't rely on customizing the build for test preparation.

Maven also aims to assist in project workflow such as release management and issue tracking.

Maven also suggests some guidelines on how to layout your project's directory structure so that once you learn the layout you can easily navigate any other project that uses Maven and the same defaults.

1.2.5 Allowing transparent migration to new features

Maven provides an easy way for Maven clients to update their installations so that they can take advantage of any changes that been made to Maven itself.

Installation of new or updated plugins from third parties or Maven itself has been made trivial for this reason.

1.3 What is Maven Not?

You may 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.

Maven does encourage best practices, but we realise that some projects may not fit with these ideals for historical reasons. While Maven is designed to be flexible, to an extent, in these situations and to the needs of different projects, it can not cater to every situation without making compromises to the integrity of its objectives.

If you decide to use Maven, and have an unusual build structure that you cannot reorganise, you may have to forgo some features or the use of Maven altogether.

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 CVS 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. We are collaborating with the folks at Ibiblio who have graciously allowed the central repository to live on their servers.

3 FAQ

3.1 Frequently Asked Technical Questions

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 what plugin emits it and how to configure the file encoding for it. For plugins that follow our guideline for source file encoding, 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>
```

[top]

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 an package such as a WAR, or included in the runtime classpath.

[top]

How do I list available plugins?

The "Available Plugins" page lists them, and provides additional information to browse the Maven 2 repository. See http://maven.apache.org/plugins

[top]

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.

[top]

How can I use Ant tasks in Maven 2?

There are currently 2 alternatives:

- For use in a plugin written in Java, Beanshell 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.

[top]

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 1.5, you should have in your pom:

```
...
<build>
...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
...
</build>
...
```

[top]

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.

[top]

Where is the source code? I couldn't seem to find a link anywhere on the Maven2 site.

The source code can be found in our subversion repository.

For more information, see Building Maven 2.0.

[top]

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.

[top]

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.

[top]

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 ...

[top]

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.

[top]

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)

[top]

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 output encoding in your pom

```
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>2.0-beta-6</version>
  <configuration>
    <outputEncoding>UTF-8</outputEncoding>
  </configuration>
</plugin>
...
```

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

[top]

How do I include `tools.jar` in my dependencies?

The following code includes `tools.jar` for JDKs on Windows, Linux and Solaris (it is already included in the runtime for Mac OS X and some free JDKs).

```
...
  <profiles>
    <profile>
      <id>default-tools.jar</id>
      <activation>
        <property>
          <name>java.vendor</name>
          <value>Sun Microsystems Inc.</value>
        </property>
      </activation>
      <dependencies>
        <dependency>
          <groupId>com.sun</groupId>
          <artifactId>tools</artifactId>
          <version>1.4.2</version>
          <scope>system</scope>
          <systemPath>${java.home}/../lib/tools.jar</systemPath>
        </dependency>
      </dependencies>
    </profile>
  </profiles>
  ...
```

[top]

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:

```
  ...
  <properties>
    <property>
      <name>test</name>
      <value>some-value</value>
     </property>
  </properties>
  ...
```

or

```
  ...
  <properties>
    <test>some-value</test>
  </properties>
  ...
```

[top]

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

[top]

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

[top]

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

We have compiled a list of available resources on the getting help page

[top]

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
```

[top]

What is a Mojo?

A mojo is a **M**aven plain **O**ld **J**ava **O**bject. Each mojo is an executable *goal* in Maven, and a plugin is a distribution of one or more related mojos.

[top]

How to find dependencies on public Maven repositories?

You could use the following search engines:

- http://repository.apache.org
- http://www.artifact-repository.org
- http://mvnrepository.com
- http://www.mvnbrowser.com
- http://www.jarvana.com
- http://mavensearch.net

[top]

4 Community Overview

4.1 The Maven Community

Maven, like any other opensource project, relies heavily on the efforts of the entire user community to be ever vigilent for improvements, logging of defects, communicating use-cases, generating documentation, and being wary of other users in need. This is a quick guide outlining what members of the Maven community may 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

Documentation is currently a very high priority for the Maven community. Please help out where ever you can, specifically in the work-in-progress FAQ Wiki.

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 Tracking page.

4.1.1.3 Developers

For Maven developers, commiters, 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. There are several tools to make this simpler, such as Nexus or Archiva. 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 all 9+ Gigs of binaries really kills our servers.

4.1.2.2 Host a Mirror

As an extention to the previous statement, if you have access to a large data repository with lots of bandwidth, please consider becomming a mirror for the Maven central repository.

As you can imagine, thousands of users downloading can put quite a strain on one server. If you wish to be a mirror, please file a request in the Maven Project Administration JIRA project.

4.1.2.3 Host a Public Repository

If you have any projects that you wish others to use, host them on your own public repository. That way, your users can simply add your repository to their own project repo list, and viola! Maven can keep you and your users in synch, growing your user-base due simply to its new-found ease of use.

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 IRC (Internet Relay Chat)

Log into the `#maven` IRC channel on `irc.codehaus.org`. If you would like to access this over a web interface, you can do so at http://irc.codehaus.org/ or irc://irc.codehaus.org/maven. IRC logs are browsable at: http://irc.rectang.com/logs/codehaus/%23maven/.

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 list and answer other user's questions
- Report bugs, feature requests and other issues in the issue tracking application.
- Build Maven for yourself, in order to fix bugs.
- Submit patches to reported issues (both those you find, or that others have filed)
- 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 are so inclined, submitting patches to correct it. You can quickly contribute rough thoughts to the wiki, or you can volunteer to help collate and organise information that is already there.

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 SVN Convention

5.5 Resources for committers

- Developer Resources
- About the Apache Software Foundation
- Committer FAQ

- Web Stats
- Mailing List Stats
- 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 form 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 Tracking*. 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 tracking 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 Tracking

7.1 Overview

Maven projects use JIRA a J2EE-based, issue tracking and project management application.

7.2 Issue Tracking

Issues, bugs, and feature requests should be submitted to the following issue tracking systems depending projects.

7.2.1 Maven Project

http://jira.codehaus.org/browse/MNG

7.2.2 Maven Website Project

http://jira.codehaus.org/browse/MNGSITE

7.2.3 Maven Project Administration

http://jira.codehaus.org/browse/MPA

7.2.4 Maven Plugins Projects

Please refer you to the Available Plugins page.

7.2.5 Maven Sub Projects

7.2.5.1 Doxia

http://jira.codehaus.org/browse/DOXIA

7.2.5.2 JXR

http://jira.codehaus.org/browse/JXR

7.2.5.3 SCM

http://jira.codehaus.org/browse/SCM

7.2.5.4 Wagon

http://jira.codehaus.org/browse/WAGON

8 Source Repository

8.1 Source Repository

Maven projects use Subversion to manage their source code. Instructions on Subversion use can be found in the online book Version Control with Subversion.

8.1.1 Web Access

The following list shows the links to the online source repositories for the various development branches of the Maven core:

```
http://svn.apache.org/viewvc/maven/maven-2/branches/maven-2.2.x
http://svn.apache.org/viewvc/maven/maven-3/trunk
```

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

8.1.2 Anonymous Access

The source can be checked out anonymously from SVN with one of these commands depending on the development line you are looking for:

```
$ svn checkout http://svn.apache.org/repos/asf/maven/maven-2/branches/maven-2.2.x m
$ svn checkout http://svn.apache.org/repos/asf/maven/maven-3/trunk maven-3
```

8.1.3 Developer Access

Everyone can access the Subversion repository via HTTP, but committers must checkout the Subversion repository via HTTPS to gain write access:

```
$ svn checkout https://svn.apache.org/repos/asf/maven/maven-2/branches/maven-2.2.x
$ svn checkout https://svn.apache.org/repos/asf/maven/maven-3/trunk maven-3
```

To commit changes to the repository, execute the following command to commit your changes (`svn` will prompt you for your password):

```
$ svn commit --username your-username -m "A message"
```

8.1.4 Access from behind a Firewall

For those users who are stuck behind a corporate firewall which is blocking HTTP access to the Subversion repository, you can try to access it via the developer connection:

```
$ svn checkout https://svn.apache.org/repos/asf/maven/maven-2/branches/maven-2.2.x
$ svn checkout https://svn.apache.org/repos/asf/maven/maven-3/trunk maven-3
```

8.1.5 Access through a Proxy

The Subversion client can go through a proxy, if you configure it to do so. First, edit your `servers` configuration file to indicate which proxy to use. The file's location depends on your operating

system. On Linux or Unix it is located in the directory `~/.subversion`. On Windows it is in `%APPDATA%\Subversion` (try `echo %APPDATA%`, note this is a hidden directory).

There are comments in the file explaining what to do. If you don't have that file, get the latest Subversion client and run any command; this will cause the configuration directory and template files to be created.

Example: Edit the `servers` file and add something like:

```
[global]
http-proxy-host = your.proxy.name
http-proxy-port = 3128
```

9 Continuous Integration

9.1 Continuous Integration

9.1.1 CI Servers

Following is an alphabetical list of some CI servers we've heard mentioned around the Maven community:

- Apache Continuum
- Bamboo (Atlassian)
- Cruise Control
- Hudson
- TeamCity (JetBrains)

10 Running Maven

10.1 Building a Project with Maven

This document centre is for those that have the source code to a project that builds with Maven, and would like to know how to use Maven to build it (or perform other common tasks).

The documents here are also helpful to new Maven users.

- Download Maven - Download the latest version of Maven
- Quick Start - Get started building the project quickly
- Cookbook - Examples of how to perform other common tasks on a Maven-built project
- Use Maven - Learn how to use Maven on your own project

10.1.1 Quick Start

10.1.1.1 Configuring Maven

Maven will run with sensible defaults, so you can get right into it. However, if you are operating under a restricted environment or behind a firewall, you might need to prepare to run Maven, as it requires write access to the home directory (`~/.m2` on Unix/Mac OS X and `C:\Documents and Settings\username\.m2` on Windows) and network access to download binary dependencies.

- Configuring Maven
- Configuring a HTTP Proxy

10.1.1.2 Building a Project

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

```
mvn clean install
```

This command tells Maven to build all the modules, and to install it in the *local repository*. The local repository is created in your home directory (or alternative location that you created it), and is the location that all downloaded binaries and the projects you built are stored.

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.

10.1.1.3 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.

To try some other tasks, see the Cookbook for Running Maven.

10.1.1.4 When Things go Wrong

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

10.Missing Dependencies

A missing dependency presents with an error like the following:

```
[INFO] Failed to resolve artifact.
Missing:
----------
1) jnuit: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=jnuit -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) jnuit: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 (http://repo1.maven.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.

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.

11 Maven Plugins

11.1 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.

11.1.1 Supported By The Maven Project

To see the most up-to-date list browse the Maven repository at http://repo1.maven.org/maven2/, specifically the `org/apache/maven/plugins` subfolder. *(Plugins are organized according to a directory structure that resembles the standard Java package naming convention)*

Plugin	Type*	Version	Release Date	Description	Source Repository	Issue Tracking
Core plugins				**Plugins corresponding to default core phases (ie. clean, compile). They may have muliple goals as well.**		
`clean`	B	2.3	2009-01-10	Clean up after the build.	SVN	JIRA
`compiler`	B	2.0.2	2007-02-13	Compiles Java sources.	SVN	JIRA
`deploy`	B	2.4	2008-08-06	Deploy the built artifact to the remote repository.	SVN	JIRA
`install`	B	2.3	2009-03-25	Install the built artifact into the local repository.	SVN	JIRA
`resources`	B	2.4	2009-08-25	Copy the resources to the output directory for including in the JAR.	SVN	JIRA

site	B	2.0.1	2009-07-14	Generate a site for the current project.	SVN	JIRA
surefire	B	2.4.3	2008-05-14	Run the Junit tests in an isolated classloader.	SVN	JIRA
verifier	B	1.0-beta-1	2006-05-07	Useful for integration tests - verifies the existence of certain conditions.	SVN	JIRA
Packaging types / tools				**These plugins relate to packaging respective artifact types.**		
ear	B	2.3.2	2009-03-07	Generate an EAR from the current project.	SVN	JIRA
ejb	B	2.2	2009-07-14	Build an EJB (and optional client) from the current project.	SVN	JIRA
jar	B	2.2	2008-01-16	Build a JAR from the current project.	SVN	JIRA
rar	B	2.2	2007-02-28	Build a RAR from the current project.	SVN	JIRA
war	B	2.1-beta-1	2009-03-22	Build a WAR from the current project.	SVN	JIRA
shade	B	1.2.1	2009-04-17	Build an Uber-JAR from the current project, including dependencies.	SVN	JIRA

Reporting plugins				**Plugins which generate reports, are configured as reports in the POM and run under the site generation lifecycle.**		
changelog	R	2.1	2007-07-25	Generate a list of recent changes from your SCM.	SVN	JIRA
changes	B+R	2.1	2008-11-24	Generate a report from issue tracking or a change document.	SVN	JIRA
checkstyle	B+R	2.3	2009-07-14	Generate a checkstyle report.	SVN	JIRA
clover	B+R	2.4	2007-04-23	Generate a Clover report. NOTE: Moved to Atlassian.com	SVN	JIRA
doap	B	1.0	2008-08-01	Generate a Description of a Project (DOAP) file from a POM.	SVN	JIRA
docck	B	1.0	2008-11-16	Documentation checker plugin.	SVN	JIRA
javadoc	B+R	2.6	2009-07-29	Generate Javadoc for the project.	SVN	JIRA
jxr	R	2.1	2007-04-05	Generate a source cross reference.	SVN	JIRA
pmd	B+R	2.4	2008-01-08	Generate a PMD report.	SVN	JIRA
project-info-reports	R	2.1.2	2009-07-23	Generate standard project reports.	SVN	JIRA
surefire-report	R	2.4.3	2008-05-14	Generate a report based on the results of unit tests.	SVN	JIRA

Tools				**These are miscellaneous tools available through Maven by default.**		
ant	B	2.2	2009-07-19	Generate an Ant build file for the project.	SVN	JIRA
antrun	B	1.3	2008-10-11	Run a set of ant tasks from a phase of the build.	SVN	JIRA
archetype	B	2.0-alpha-4	2008-09-26	Generate a skeleton project structure from an archetype.	SVN	JIRA
assembly	B	2.2-beta-4	2009-06-05	Build an assembly (distribution) of sources and/or binaries.	SVN	JIRA
dependency	B+R	2.1	2009-01-10	Dependency manipulation (copy, unpack) and analysis.	SVN	JIRA
enforcer	B	1.0-beta-1	2009-02-25	Environmental constraint checking (Maven Version, JDK etc), User Custom Rule Execution.	SVN	JIRA
gpg	B	1.0-alpha-4	2007-09-28	Create signatures for the artifacts and poms.	SVN	JIRA
help	B	2.1	2008-09-04	Get information about the working environment for the project.	SVN	JIRA

invoker	B	1.4	2009-09-24	Run a set of Maven projects and verify the output.	SVN	JIRA
jarsigner	B	1.2	2009-09-30	Signs or verifies project artifacts.	SVN	JIRA
one	B	1.2	2007-09-12	A plugin for interacting with legacy Maven 1.x repositories and builds.	SVN	JIRA
patch	B	1.1	2009-04-13	Use the gnu patch tool to apply patch files to source code.	SVN	JIRA
pdf	B	1.0	2009-06-29	Generate a PDF version of your project's documentation.	SVN	JIRA
plugin	B+R	2.5.1	2009-10-08	Create a Maven plugin descriptor for any Mojo's found in the source tree, to include in the JAR.	SVN	JIRA
release	B	2.0-beta-9	2009-03-28	Release the current project - updating the POM and tagging in the SCM.	SVN	JIRA
reactor	B	1.0	2008-09-27	Build a subset of interdependent projects in a reactor	SVN	JIRA
remote-resources	B	1.1	2009-09-22	Copy remote resources to the output directory for inclusion in the artifact.	SVN	JIRA

repository	B	2.2	2009-08-10	Plugin to help with repository-based tasks.	SVN	JIRA
scm	B	1.1	2008-08-27	Generate a SCM for the current project.	SVN	JIRA
source	B	2.1.1	2009-10-16	Build a JAR of sources for use in IDEs and distribution to the repository.	SVN	JIRA
stage	B	1.0-alpha-2	2009-07-14	Assists with release staging and promotion.	SVN	JIRA
IDEs				**Plugins that simplify integration with integrated developer environments.**		
eclipse	B	2.7	2009-06-13	Generate an Eclipse project file for the current project.	SVN	JIRA
idea	B	2.2	2008-08-08	Create/ update an IDEA workspace for the current project (individual modules are created as IDEA modules)	SVN	JIRA

* **B**uild or **R**eporting plugin

There are also some sandbox plugins into our source reposity.

11.1.2 Outside The Maven Land

11.1.2.1 At codehaus.org

There are also many plug-ins available at the Mojo project at Codehaus.

To see the most up-to-date list, browse the Codehaus repository at http://repository.codehaus.org/, specifically the `org/codehaus/mojo` subfolder. Here are a few common ones:

Plugin (see complete list with version)	Description
build-helper	Attach extra artifacts and source folders to build.
castor	Generate sources from an XSD using Castor.
javacc	Generate sources from a JavaCC grammer.
jdepend	Generate a report on code metrics using JDepend.
native	Compiles C and C++ code with native compilers.
sql	Executes SQL scripts from files or inline.
taglist	Generate a list of tasks based on tags in your code.

11.1.2.2 At code.google.com

There are also many plug-ins available at the Google Code.

11.1.2.3 Misc

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

Plugin	Maintainer	Description
cargo	Cargo Project	Start/stop/configure J2EE containers and deploy to them.
jaxme	Apache Web Services Project	Use the JaxMe JAXB implementation to generate Java sources from XML schema.
jetty	Jetty Project	Jetty Run a Jetty container for rapid webapp development.
jalopy	Triemax	Use Jalopy to format your source code.
rat	Apache Incubator Project	Release Audit Tool (RAT) to verify files.
Genesis Plugins	Apache Geronimo Project	Verify legal files in artifacts.

11.1.3 Resources

1 Guide to Configuring Plugins

12 User Centre

12.1 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
- Build Cookbook - Examples for how to add functionality to your build
- Getting Help - How to get help with Maven

12.1.1 Reference

- POM Reference
- Settings Reference

13 Maven in 5 Minutes

13.1 Maven in 5 Minutes

13.1.1 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:

```
mvn --version
```

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

```
Maven version: 2.0.8
Java version: 1.5.0_12
OS name: "windows 2003" version: "5.2" arch: "x86" Family: "windows"
```

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

13.1.2 Creating a Project

On your command line, execute the following Maven goal:

```
mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app
```

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 *create* goal created a directory with the same name given as the artifactId. Change into that directory.

```
cd my-app
```

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

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

The src/main/java directory contains the project source code, the src/test/java directory contains the test source, and the pom.xml is the project's Project Object Model, or POM.

13.1.2.1 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:

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

13.1.2.2 What did I just do?

You executed the Maven goal *archetype:create*, and passed in various parameters to that goal. The prefix *archetype* is the plugin that contains the goal. If you are familiar with Ant, you may concieve of this as similar to a task. This goal created a simple project based upon an 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".

13.1.2.3 Build the Project

```
mvn package
```

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

```
 ...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Thu Oct 05 21:16:04 CDT 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

Unlike the first command executed (*archetype:create*) you may notice 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 will execute every phase in the sequence up to and including the one defined. For example, if we 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 -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App
```

Which will print the quintessential:

```
Hello World!
```

13.1.3 Running Maven Tools

13.1.3.1 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:jar* if the project type is a JAR, and *war:war* is the project type is - you guessed it - a WAR.

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

```
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).

13.1.3.2 Generating the Site

```
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`.

13.1.4 Conclusion

We hope this quick overview has piqued your interest in the versitility 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.

14 Getting Started Guide

14.1 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!

14.2 Sections

14.2.1 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

If you want more background information on Maven you can check out The Philosophy of Maven and The History of Maven. Now let's move on to how you, the user, can benefit from using Maven.

14.2.2 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. For a more detailed look at how Maven can help you with your development process please refer to The Benefits of Using Maven.

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!

14.2.3 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.

14.2.4 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 archetype:create \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DgroupId=com.mycompany.app \
  -DartifactId=my-app
```

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:

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

`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 plug-ins.
- **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`).
- **packaging** This element indicates the package type to be used by this artifact (e.g. JAR, WAR, EAR, etc.). This not only means if the artifact produced is JAR, WAR, or EAR but can also indicate a specific lifecycle to use as part of the build process. (The lifecycle is a topic we will deal with further on in the guide. For now, just keep in mind that the indicated packaging of a project can play a part in customizing the build lifecycle.) The default value for the `packaging` element is JAR so you do not have to specify this for most projects.
- **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.
- **description** This element provides a basic description of your project. This is often used in Maven's generated documentation.

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
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

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 ...

14.2.5 How do I compile my application sources?

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

```
mvn compile
```

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

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-resources-plugin: \
  checking for updates from central
...
[INFO] artifact org.apache.maven.plugins:maven-compiler-plugin: \
  checking for updates from central
...
[INFO] [resources:resources]
...
[INFO] [compiler:compile]
Compiling 1 source file to <dir>/my-app/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 minutes 54 seconds
[INFO] Finished at: Fri Sep 23 15:48:34 GMT-05:00 2005
[INFO] Final Memory: 2M/6M
[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!

14.2.6 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] ------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-surefire-plugin: \
  checking for updates from central
...
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
Compiling 1 source file to C:\Test\Maven2\test\my-app\target\test-classes
...
[INFO] [surefire:test]
[INFO] Setting reports dir: C:\Test\Maven2\test\my-app\target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.mycompany.app.AppTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0 sec
Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 15 seconds
[INFO] Finished at: Thu Oct 06 08:12:17 MDT 2005
[INFO] Final Memory: 2M/8M
[INFO] ------------------------------------------------------------------------
```

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:

```
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 ...

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

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

```
mvn package
```

If you take a look at the POM for your project you will notice the `packaging` element is set to `jar`. This is how Maven knows to produce a JAR file from the above command (we'll talk more about this later). 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 (`~/.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:

```
mvn install
```

Upon executing this command you should see the following output:

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
Compiling 1 source file to <dir>/my-app/target/classes
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
Compiling 1 source file to <dir>/my-app/target/test-classes
[INFO] [surefire:test]
[INFO] Setting reports dir: <dir>/my-app/target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.mycompany.app.AppTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.001 sec
Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0
[INFO] [jar:jar]
[INFO] Building jar: <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] [install:install]
[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] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Tue Oct 04 13:20:32 GMT-05:00 2005
[INFO] Final Memory: 3M/8M
[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 plug-ins 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.

Perhaps you'd like to generate an IntelliJ IDEA descriptor for the project?

```
mvn idea:idea
```

This can be run over the top of a previous IDEA project - it will update the settings rather than starting fresh.

If you are using Eclipse IDE, just call:

```
mvn eclipse:eclipse
```

Note: some familiar goals from Maven 1.0 are still there - such as `jar:jar`, but they might not behave like you'd expect. Presently, `jar:jar` will not recompile sources - it will simply just create a JAR from the `target/classes` directory, under the assumption everything else had already been done.

14.2.8 How do I use plug-ins?

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

Note for Maven 1.0 Users: In Maven 1.0, you would have added some `preGoal` to `maven.xml` and some entries to `project.properties`. Here, it is a little different.

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:

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

You'll notice that all plugins in Maven 2.0 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 Plug-ins.

14.2.9 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
    |   |               `-- App.java
    |   `-- 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
    `-- mycompany
        `-- app
            `-- App.class
```

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:

```
#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:

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

14.2.10 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:

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</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 ${pom.name} refers to the name of the project, ${pom.version} refers to the version of the project, ${pom.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:

```
# application.properties
application.name=${pom.name}
application.version=${pom.version}
```

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:

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</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>
</project>
```

Then, if we add a reference to this property in the application.properties file:

```
# application.properties
application.name=${pom.name}
application.version=${pom.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
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
  <properties>
    <my.filter.value>hello</my.filter.value>
  </properties>
</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"
```

14.2.11 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):

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

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 (`~/.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:

```
<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">
  <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 http://repo1.maven.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. We can browse ibiblio and look for it, or use Google to help by searching for "site:www.ibiblio.org maven2 log4j". The search shows a directory called /maven2/log4j/log4j (or /pub/packages/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). 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:

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</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.

14.2.12 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 connectiong to the repository in the settings.xml.

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

```
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.codehaus.plexus</groupId>
      <artifactId>plexus-utils</artifactId>
      <version>1.0.4</version>
    </dependency>
  </dependencies>
  <build>
    <filters>
      <filter>src/main/filters/filters.properties</filter>
    </filters>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
  <!--
   |
   |
   |
   -->
  <distributionManagement>
    <repository>
      <id>mycompany-repository</id>
      <name>MyCompany Repository</name>
      <url>scp://repository.mycompany.com/repository/maven2</url>
    </repository>
  </distributionManagement>
</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
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>mycompany-repository</id>
      <username>jvanzyl</username>
      <!-- Default value is ~/.ssh/id_dsa -->
      <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa)
      <passphrase>my_key_passphrase</passphrase>
    </server>
  </servers>
  ...
</settings>
```

Note that if you are connecting to an openssh ssh server which has the parameter "PasswordAuthentication" set to "no" in the sshd_confing, 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.

14.2.13 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:create \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-site \
  -DgroupId=com.mycompany.app \
  -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.

14.2.14 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:create \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp \
    -DgroupId=com.mycompany.app \
    -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:

```
<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-webapp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</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 clean package
```

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

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

The concept of dealing with multiple modules is built in to Maven 2.0. 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:

```
<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>
  <version>1.0-SNAPSHOT</version>
  <artifactId>app</artifactId>
  <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`:

```
  ...
  <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:

```
<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">
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  ...
```

Now, try it... from the top level directory, run:

```
mvn clean install
```

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/
   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/
3239 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-
webapp/pom.xml
   0 Fri Jun 24 10:59:56 EST 2005 WEB-INF/
 215 Fri Jun 24 10:59:56 EST 2005 WEB-INF/web.xml
 123 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-
webapp/pom.properties
  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-3.8.1.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 is different to the `extend` element you may be familiar with from Maven 1.0: 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.

Unlike Maven 1.0, it is not required that you run `install` to successfully perform these steps - you can run `package` on its own and the artifacts in the reactor will be used from the target directories instead of the local repository.

You might like to generate your IDEA workspace again from the top level directory...

```
mvn idea:idea
```

15 POM Reference

15.1 POM Reference

15.2 Introduction

- The POM 4.0.0 XSD

15.2.1 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 need not contain any code at all, merely a `pom.xml`.

15.2.2 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 for Maven 2, and is always required.

```
<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>
  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>
  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>
  <!-- More Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <organization>...</organization>
  <developers>...</developers>
  <contributors>...</contributors>
  <!-- Environment Settings -->
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>
</project>
```

15.3 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, effectively, 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 effectively embed ant tasks inside of the POM. It is ultimately a declaration, however. Where as 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 will 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.

```
<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>org.codehaus.mojo</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>
```

15.3.1 Maven Coordinates

The POM defined above is the minimum that Maven 2 will allow. `groupId:artifactId:version` are all required fields (although, groupId and version need not 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 `$M2_REPO/org/codehaus/mojo`.
- **artifactId**: The artifactId is generally the name that the project is known by. Although the groupId is important, people within the group will rarely mention the groupId in discussion (they are often all be the same ID, such as the Codehaus Mojo project groupId: `org.codehaus.mojo`). It, along with the groupId, create a key that separates this project from every other project in the world (at least, it should :)). Along with the groupId, the artifactId fully defines the artifact's living quarters within the repository. In the case of the above project, `my-project` lives in `$M2_REPO/org/codehaus/mojo/my-project`.
- **version**: This is the last piece of the naming puzzle. `groupId:artifactId` denote a single project but they cannot delineate which incarnation of that project we are talking about. Do we want the `junit:junit` of today (version 4), or of four years ago (version 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 knows *who* we are dealing with, and *when* in its software lifecycle we want them.
- **packaging**: Now that we have our address structure of `groupId:artifactId:version`, there is one more standard label to give us a really complete address. That is the project's artifact type. 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:

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

When no packaging is declared, Maven assumes the artifact 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`, `par`. These define the default list of goals which execute to each corresponding build lifecycle stage for a particular package structure.

You will sometimes see Maven print out a project coordinate as `groupId:artifactId:packaging:version`.

- **classifier**: You may occasionally find a fifth element on the coordinate, and that is the `classifier`. We will visit the classifier later, but for now it suffices to know that those kinds of projects are displayed as `groupId:artifactId:packaging:classifier:version`.

15.3.2 POM Relationships

One powerful aspect of Maven is in its handling of project relationships; that 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 versions as those 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.

15.3.2.1 Dependencies

The cornerstone of the POM is its dependency list. Most every project depends upon others to build and run correctly, and if all Maven does for you is manage this list for you, you have gained a lot. Maven downloads and links the dependencies for you on compilation and 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.

```
<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">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

- **groupId**, **artifactId**, **version**:
  These elements are self-explanatory, and you will see them often. This trinity represents the coordinate of a specific project in time, demarcating it as a dependency of this project. 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:
     `mvn install:install-file -Dfile=non-maven-proj.jar -DgroupId=some.group -DartifactId=non-maven-proj -Dversion=1 -Dpackaging=jar`

     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 allows to distinguish artifacts that were built from the same POM but differ in their 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 JRE 1.5 but at the same time also an artifact that still supports JRE 1.4. The first artifact could be equipped with the classifier `jdk15` and the second one with `jdk14` such that clients can choose which one to use.

Another common use case for classifiers is the need 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 dependant artifact's `packaging` 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 correspongs to the packaging used, though this is also not always the case. Some examples are `jar`, `ejb-client` and `test-jar`. 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 depedency. 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.
  - **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 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 will not check the repositories for the project, but instead checks to ensure that the file exists. If not, Maven will fail the build and suggest that you download and install it manually.
- **optional**:
  Marks optional a dependency when this project itself is a dependency. Confused? 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 will 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.

15. Exclusions

Exclusions explicitly tell Maven that you don't want 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
                      http://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>
  ...
</project>
```

- **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.

15.3.2.2 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 certainly be simulated, Maven has gone the extra step in making project inheritance explicit to the project object model.

```
<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>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. If the packaging is `pom`, the goal executed will be `site:attach-descriptor`. Now we may add values to the parent POM, which will be inherited by its children. The elements in the parent POM that are inherited by its children are:

- dependencies
- developers and contributors
- plugin lists
- reports lists
- plugin executions with matching ids

- plugin configuration

```
<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>org.codehaus.mojo</groupId>
    <artifactId>my-parent</artifactId>
    <version>2.0</version>
    <relativePath>../my-parent</relativePath>
  </parent>
  <artifactId>my-project</artifactId>
</project>
```

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.

15. 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 2.0.x.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>
  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      <layout>default</layout>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
    </pluginRepository>
  </pluginRepositories>
  <build>
    <directory>${project.basedir}/target</directory>
    <outputDirectory>${project.build.directory}/classes</outputDirectory>
    <finalName>${project.artifactId}-${project.version}</finalName>
    <testOutputDirectory>${project.build.directory}/test-classes</testOutputDirecto
    <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>
    <!-- TODO: MNG-3731 maven-plugin-tools-api < 2.4.4 expect this to be relative..
    <scriptSourceDirectory>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>
       <plugins>
         <plugin>
           <artifactId>maven-antrun-plugin</artifactId>
           <version>1.3</version>
         </plugin>
         <plugin>
           <artifactId>maven-assembly-plugin</artifactId>
           <version>2.2-beta-2</version>
         </plugin>
         <plugin>
           <artifactId>maven-clean-plugin</artifactId>
           <version>2.2</version>
         </plugin>
         <plugin>
```

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`

15. 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 POMs to help manage dependency information across all of its children. If the `my-parent` project uses `dependencyManagement` to define a dependency on `junit:junit:4.0`, then POMs inheriting from this one can set their dependency giving the `groupId`= `junit` and `artifactId`= `junit` only, then 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 will propagate to all inheriting POMs. In addition, the version and scope of artifacts which are incorporated from transitive dependencies may also be controlled by specifying them in a dependency management section.

15.3.2.3 Aggregation (or Multi-Module)

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

```
<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>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <modules>
    <module>my-project<module>
    <module>another-project<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, just have a look at the Maven or Maven Core Plugins base POM's.

15.A final note on Inheritance v. Aggregation

Inheritance and aggregation create a nice dynamic to control builds through a single, high-level POM. You will often see projects that are both parents and aggregators. For example, the entire maven core runs through a single base POM `org.apache.maven:maven`, so building the Maven project can be executed by a single command: `mvn compile`. However, although both POM projects, an aggregator project and a parent project are not one in the same and should not be confused. A POM project may be inherited from - but does not necessarily have - any modules that it aggregates. Conversely, a POM project may aggregate projects that do not inherit from it.

15.3.3 Properties

Properties are the last required piece in understanding POM basics. Maven properties are value placeholder, 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:

1. `env.X`: Prefixing a variable with "env." will return the shell's environment variable. For example, `${env.PATH}` contains the PATH environment variable. *Note:* While environment variables themselves are case-insensitive on Windows, lookup of properties is case-sensitive. In other words, while the Windows shell returns the same value for `%PATH%` and `%Path%`, Maven distinguishes between `${env.PATH}` and `${env.Path}`. As of Maven 2.1.0, the names of environment variables are normalized to all upper-case for the sake of reliability.
2. `project.x`: A dot (.) notated path in the POM will contain the corresponding element's value. For example: `<project><version>1.0</version></project>` is accessible via `${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. The value may be used as `${someVar}`.

15.4 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.

15.4.1 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".*

```
<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">
  ...
  <!-- "Project Build" contains more elements than just the BaseBuild set -->
  <build>...</build>
  <profiles>
    <profile>
      <!-- "Profile Build" contains a subset of "Project Build"s elements -->
      <build>...</build>
    </profile>
  </profiles>
</project>
```

15.4.1.1 The BaseBuild Element Set

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

```
<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 `filter/` 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.

15. 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/resource/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:

```
<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">
  <build>
    ...
    <resources>
      <resource>
        <targetPath>META-INF/plexus</targetPath>
        <filtering>false</filtering>
        <directory>${basedir}/src/main/plexus</directory>
        <includes>
          <include>configuration.xml</include>
        </includes>
        <excludes>
          <exclude>**/*.properties</exclude>
        </excludes>
      </resource>
    </resources>
    <testResources>
      ...
    </testResources>
    ...
  </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.

15. Plugins

```
<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">
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.0</version>
        <extensions>false</extensions>
        <inherited>true</inherited>
        <configuration>
          <classifier>test</classifier>
        </configuration>
        <dependencies>...</dependencies>
        <executions>...</executions>
      </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.
- **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.
- **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 dpendency. 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:

```
<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">
  ...
  <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>
              <tasks>
                <echo>Build Dir: ${project.build.directory}</echo>
              </tasks>
            </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 false will supress 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.

15. 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. The children have every right to override `pluginManagement` definitions.

```
<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">
  ...
  <build>
    ...
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <version>2.2</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>
    </pluginManagement>
    ...
  </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:

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

15.4.1.2 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.

15. 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.

```
<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">
  ...
  <build>
    <sourceDirectory>${basedir}/src/main/java</sourceDirectory>
    <scriptSourceDirectory>${basedir}/src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>${basedir}/src/test/java</testSourceDirectory>
    <outputDirectory>${basedir}/target/classes</outputDirectory>
    <testOutputDirectory>${basedir}/target/test-classes</testOutputDirectory>
    ...
  </build>
</project>
```

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}`.

15. 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 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.

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

15.4.2 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.

```
<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">
  ...
  <reporting>
    <outputDirectory>${basedir}/target/site</outputDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.0.1</version>
        <reportSets>
          <reportSet></reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

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

15.4.2.1 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:

```
<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">
  ...
  <reporting>
    <plugins>
      <plugin>
        ...
        <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>
      </plugin>
    </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 they also must configure individual goals of those plugins. That is where these elements come in, giving the POM ultimate granularity in control of its build destiny.

15.5 More Project Information

Although the above information is enough to get a firm grasp on POM authoring, there are far more elements to make developer's live easier. Many of these elements are related to site generation, but like all POM declarations, they may be used for anything, depending upon how certain plugins use it. 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**: Description of a project is always good. Although this should not replace formal documentation, a quick comment to any readers of the POM is always helpful.
- **url**: The URL, like the name, is not required. This is a nice gesture for projects users, however, so that they know where the project lives.
- **inceptionYear**: This is another good documentation point. It will at least help you remember where you have spent the last few years of your life.

15.5.1 Licenses

```
<licenses>
  <license>
    <name>Apache 2</name>
    <url>http://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. Note that a project should list only licenses that may apply directly to this project, and not list licenses that apply to this project's dependencies. Maven currently does little with these documents other than displays them on generated sites. However, there is talk of flexing for different types of licenses, forcing users to accept license agreements for certain types of (non open source) projects.

- **name**, **url** and **comments**: are self explanatory, and have been encountered before in other capacities. 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).

15.5.2 Organization

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

```
<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">
  ...
  <organization>
    <name>Codehaus Mojo</name>
    <url>http://mojo.codehaus.org</url>
  </organization>
</project>
```

15.5.3 Developers

All projects consist of files that were created, at some time, by a person. Like the other systems that surround a project, so to 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 need not be listed here.

```
<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">
  ...
  <developers>
    <developer>
      <id>eric</id>
      <name>Eric</name>
      <email>eredmond@codehaus.org</email>
      <url>http://eric.propellors.net</url>
      <organization>Codehaus</organization>
      <organizationUrl>http://mojo.codehaus.org</organizationUrl>
      <roles>
        <role>architect</role>
        <role>developer</role>
      </roles>
      <timezone>-6</timezone>
      <properties>
        <picUrl>http://tinyurl.com/prv4t</picUrl>
      </properties>
    </developer>
  </developers>
  ...
</project>
```

- **id**, **name**, **email**: These corrospond 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 numerical offset in hours from GMT where the developer lives.
- **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.

15.5.4 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.

```
<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">
  ...
  <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>-5</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.

15.6 Environment Settings

15.6.1 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, its primarily used for generating project 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
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <issueManagement>
    <system>Bugzilla</system>
    <url>http://127.0.0.1/bugzilla/</url>
  </issueManagement>
  ...
</project>
```

15.6.2 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`.

```
<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">
  ...
  <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>
```

15.6.3 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.

```
<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">
  ...
  <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**: There 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).

15.6.4 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.

```
<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">
  ...
  <scm>
    <connection>scm:svn:http://127.0.0.1/svn/my-project</connection>
    <developerConnection>scm:svn:https://127.0.0.1/svn/my-project</developerConnect
    <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) should be the default.
- **url**: A publicly browsable repository. For example, via ViewCVS.

```
<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">
  ...
  <prerequisites>
    <maven>2.0.4</maven>
  </prerequisites>
  ...
</project>
```

- **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. 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.

15.6.5 Repositories

Repositories are collections of artifacts which adhere to the Maven repository directory layout. In order to be a Maven 2 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. The default central Maven repository lives on http://repo1.maven.org/maven2/. Another source for artifacts not yet in iBiblio is the Codehaus snapshots repo.

```
<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">
  ...
  <repositories>
    <repository>
      <releases>
        <enabled>false</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
      </snapshots>
      <id>codehausSnapshots</id>
      <name>Codehaus Snapshots</name>
      <url>http://snapshots.maven.codehaus.org/maven2</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. 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`.

15.6.6 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 plugins 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.

15.6.7 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:

```
<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">
  ...
  <distributionManagement>
    ...
    <downloadUrl>http://mojo.codehaus.org/my-project</downloadUrl>
    <status>deployed</status>
  </distributionManagement>
  ...
</project>
```

- **downloadUrl**: is the url of the repository from whence another POM may 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 could just as easily have been called synched. This means that this artifact has been synched with a partner repository.
    - **deployed**: By far the most common status, meaning that this artifact was deployed from a Maven 2 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.

15.6.7.1 Repository

Where as 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
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <distributionManagement>
    <repository>
      <uniqueVersion>false</uniqueVersion>
      <id>corp1</id>
      <name>Corporate Repository</name>
      <url>scp://repo1/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 to be 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`.

15.6.7.2 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
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <distributionManagement>
    ...
    <site>
      <id>mojo.website</id>
      <name>Mojo Website</name>
      <url>scp://beaver.codehaus.org/home/projects/mojo/public_html/</url>
    </site>
    ...
  </distributionManagement>
  ...
</project>
```

- **id**, **name**, **url**: These elements are similar to their counterparts above in the `distributionManagement` `repository` element.

15.6.7.3 Relocation

```
<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">
  ...
  <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 your users as 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.

15.6.8 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:

```
<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">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>...</activation>
      <build>...</build>
      <modules>...</modules>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <dependencies>...</dependencies>
      <reporting>...</reporting>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
    </profile>
  </profiles>
</project>
```

15.6.8.1 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
                      http://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>
        <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>
</project>
```

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.

- **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 as of Maven 2.1. See the maven-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 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
```

15.6.8.2 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. Reguardless 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 seperate projects.*

15.7 Final

The Maven 2 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.

16 Settings Reference

16.1 Settings Reference

16.2 Introduction

16.2.1 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: `$M2_HOME/conf/settings.xml`
- A user's install: `${user.home}/.m2/settings.xml`

Here is an overview of the top elements under `settings`:

```
<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">
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

16.3 Settings Details

16.3.1 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.

```
<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">
  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <usePluginRegistry>false</usePluginRegistry>
  <offline>false</offline>
  <pluginGroups>
    <pluginGroup>org.mortbay.jetty</pluginGroup>
  </pluginGroups>
  ...
</settings>
```

- **localRepository**: This value is the path of this build system's local repository. The default value is `${user.home}/.m2/repository`. This element is especially useful for a main build server allowing all logged-in users to build from a common local repository.
- **interactiveMode**: `true` if Maven should attempt to interact with the user for input, `false` if not. Defaults to `true`.
- **usePluginRegistry**: `true` if Maven should use the `${user.home}/.m2/plugin-registry.xml` file to manage plugin versions, defaults to `false`. *Note that for the current version of Maven 2.0, the plugin-registry.xml file should not be depended upon. Consider it dormant for now.*
- **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.
- **pluginGroups**: 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`. For example, given the above settings the Maven command line may execute `org.mortbay.jetty:jetty-maven-plugin:run` with the truncated command: `mvn jetty:run`

16.3.2 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`.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>server001</id>
      <username>my_login</username>
      <password>my_password</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>some_passphrase</passphrase>
      <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 corrosponding to *nix file permissions, ie. 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.

16.3.2.1 Password Encryption

A new feature - server password and passphrase encryption has been added to 2.1.x and 3.0 trunks. See details on this page

16.3.3 Mirrors

```
<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">
  ...
  <mirrors>
    <mirror>
      <id>planetmirror.com</id>
      <name>PlanetMirror Australia</name>
      <url>http://downloads.planetmirror.com/pub/maven2</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

- **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 (`http://repo1.maven.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.

16.3.4 Proxies

```
<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">
  ...
  <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, seperated 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.

16.3.5 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.

16.3.5.1 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.

```
<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 as of Maven 2.1. See the maven-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 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
```

16.3.5.2 Properties

Maven properties are value placeholder, 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 files, 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
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <properties>
        <user.install>${user.home}/our-project</user.install>
      </properties>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

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

16.3.5.3 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 it 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.

```
<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>
      ...
      <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>
        ...
      </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`.

16.3.5.4 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 plugins 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.

16.3.6 Active Profiles

```
<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">
  ...
  <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, reguardless 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 corrosponding `id` will be active. If no such profile is found then execution will continue as normal.

17 Guides

17.1 Documentation

17.1.1 Getting Started with Maven

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

17.1.2 Introductions

- The Build Lifecycle
- The POM
- Profiles
- Repositories
- Standard Directory Layout
- The Dependency Mechanism

17.1.2.1 Plugins

- Plugin Development
- Configuring Plug-ins
- The Plugin Registry
- Plugin Prefix Resolution
- Developing Ant Plugins
- Developing Java Plugins

17.1.2.2 Site

- Creating a Site
- The APT Format
- Snippet Macro

17.1.2.3 Archetypes

- What is an Archetype
- Creating Archetypes

17.1.2.4 Upgrading

- From Maven 1.x to Maven 2.x
- Using Maven 1.x repositories with Maven 2.x
- Relocation of Artifacts

17.1.2.5 Repositories

- Installing 3rd party JARs to Local Repository
- Deploying 3rd party JARs to Remote Repository

- Coping with Sun JARs
- Remote repository access through authenticated HTTPS

17.1.3 Guides

- Creating Assemblies
- Configuring Archive Plugins
- Configuring Maven
- Mirror Settings
- Deployment and Security Settings
- Embedding Maven 2.x
- Generating Sources
- Working with Manifests
- Maven Classloading
- Using Multiple Modules in a Build
- Using Multiple Repositories
- Using Proxies
- Using the Release Plugin
- Using Ant with Maven
- Using Modello
- Webapps
- Using Extensions
- Building For Different Environments with Maven 2
- Using Toolchains
- Encrypting passwords in settings.xml

17.1.3.1 Testing

- Reusable Test JARs

17.1.3.2 Maven Tools and IDE Integration

- Eclipse
- IDEA
- Netbeans 4.0 (4.1 and 5.0)
- Maven 2.x Auto-Completion Using BASH

17.1.4 Development Guides

- Building Maven from Scratch
- Developing Maven
- The Plugin Documentation Standard
- Maven Documentation Style

17.1.5 The Maven Community

- The Maven Community

- Helping with Maven
- Guide for New Committers
- Testing Development Versions of Plugins
- 3rd Party Resources

17.1.5.1 Conventions

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

17.1.5.2 The Central Repository

- Uploading Artifacts to the Central Repository
- Improving the Repository

17.1.6 References

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

17.1.7 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 could also browse the full technical documentation references of the current version of Maven.

18 The Build Lifecycle

18.1 Introduction to the Build Lifecycle

18.1.1 Table Of Contents

18.1.2 Build Lifecycle Basics

Maven 2.0 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 site documentation.

18.1.2.1 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 has the following build phases (for a complete list of the build 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.
- `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.

These build phases (plus the other build phases not shown here) are executed sequentially to complete the default lifecycle. Given the build 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 package, install the verifed package to the local repository, then deploy the installed package in a specified environment.

To do all those, you only need to call the last build phase to be executed, in this case, `deploy`:

```
mvn deploy
```

That is because if you call a build phase, it will execute not only that build phase, but also every build phase prior to the called build phase. Thus, doing

```
mvn integration-test
```

will do every build phase before it (`validate`, `compile`, `package`, etc.), before executing `integration-test`.

There are more commands that are part of the lifecycle, which will be discussed in the following sections.

It should also be noted that the same command can be used in a multi-module scenario (i.e. a project with one or more subprojects). For example:

```
mvn clean install
```

This command will traverse into all of the subprojects and run `clean`, then `install` (including all of the prior steps).

[top].

18.1.2.2 A Build Phase is Made Up of 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 goals bound to those build phases.

A 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.

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

If this were to be executed, the `clean` phase will be executed first (meaning it will run all preceeding 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 preceeding 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*).

[top].

18.1.3 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?

18.1.3.1 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.

`process-resources`	`resources:resources`
`compile`	`compiler:compile`
`process-test-resources`	`resources:testResources`
`test-compile`	`compiler:testCompile`
`test`	`surefire:test`
`package`	`jar:jar`
`install`	`install:install`
`deploy`	`deploy:deploy`

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 your `<build>` section of your POM (as described in the next section). One example of a plugin that requires this is the Plexus plugin, which provides a `plexus-application` and `plexus-service` packaging.

[top].

18.1.3.2 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 later 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>1.0-alpha-18</version>
   <executions>
     <execution>
       <configuration>
         <model>maven.mdo</model>
         <modelVersion>4.0.0</modelVersion>
       </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>maven-touch-plugin</artifactId>
   <version>1.0</version>
   <executions>
     <execution>
       <phase>process-test-resources</phase>
       <goals>
         <goal>timestamp</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
...
```

[top].

18.1.4 Lifecycle Reference

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

Clean Lifecycle

pre-clean	executes processes needed prior to the actual project cleaning
clean	remove all files generated by the previous build
post-clean	executes processes needed to finalize the project cleaning

Default Lifecycle

validate	validate the project is correct and all necessary information is available.
initialize	initialize build state, e.g. set properties or create directories.
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-resources	copy and process the resources into the test destination directory.
test-compile	compile the test source code into the test destination directory
process-test-classes	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. For Maven 2.0.5 and above.
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
prepare-package	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above)
package	take the compiled code and package it in its distributable format, such as a JAR.
pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.

`integration-test`	process and deploy the package if necessary into an environment where integration tests can be run.
`post-integration-test`	perform actions required after integration tests have been executed. This may including cleaning up the environment.
`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.

Site Lifecycle

pre-site	executes processes needed prior to the actual project site generation
site	generates the project's site documentation
post-site	executes processes needed to finalize the site generation, and to prepare for site deployment
site-deploy	deploys the generated site documentation to the specified web server

[top].

18.1.5 Built-in Lifecycle Bindings

Some phases have goals binded 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.

18.1.5.1 Clean Lifecycle Bindings

`clean`	`clean:clean`

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

`process-resources`	`resources:resources`
`compile`	`compiler:compile`
`process-test-resources`	`resources:testResources`
`test-compile`	`compiler:testCompile`
`test`	`surefire:test`
`package`	`ejb:ejb` *or* `ejb3:ejb3` *or* `jar:jar` *or* `par:par` *or* `rar:rar` *or* `war:war`
`install`	`install:install`

deploy	deploy:deploy

18.1.5.3 Default Lifecycle Bindings - Packaging `ear`

generate-resources	ear:generateApplicationXml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

18.1.5.4 Default Lifecycle Bindings - Packaging `maven-plugin`

generate-resources	plugin:descriptor
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar *and* plugin:addPluginArtifactMetadata
install	install:install *and* plugin:updateRegistry
deploy	deploy:deploy

18.1.5.5 Default Lifecycle Bindings - Packaging `pom`

package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

18.1.5.6 Site Lifecycle Bindings

site	site:site
site-deploy	site:deploy

18.1.5.7 References

The full Maven lifecycle is defined by the file `components.xml` in the module `maven-core` and viewable from SVN in the branches for Maven 2.2.0 and Maven 3.0.x.

[top].

19 The POM

19.1 Introduction to the POM

19.1.1 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 is the build directory, which is `target`; the source directory, which is `src/main/java`; the test source directory, which is `src/main/test`; and so on.

The POM was renamed from `project.xml` in Maven 1 to `pom.xml` in Maven 2. Instead of having a `maven.xml` file that contains the goals that can be executed, the goals or plugins are now configured in the `pom.xml`. 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]

19.1.2 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. The snippet below is the Super POM for Maven 2.0.x.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>
  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      <layout>default</layout>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
    </pluginRepository>
  </pluginRepositories>
  <build>
    <directory>target</directory>
    <outputDirectory>target/classes</outputDirectory>
    <finalName>${artifactId}-${version}</finalName>
    <testOutputDirectory>target/test-classes</testOutputDirectory>
    <sourceDirectory>src/main/java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>src/test/java</testSourceDirectory>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <directory>src/test/resources</directory>
      </testResource>
    </testResources>
  </build>
  <reporting>
    <outputDirectory>target/site</outputDirectory>
  </reporting>
  <profiles>
    <profile>
      <id>release-profile</id>
      <activation>
        <property>
          <name>performRelease</name>
        </property>
      </activation>
      <build>
        <plugins>
          <plugin>
            <inherited>true</inherited>
```

The snippet below is the Super POM for Maven 2.1.x.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>
  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      <layout>default</layout>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
    </pluginRepository>
  </pluginRepositories>
  <build>
    <directory>${project.basedir}/target</directory>
    <outputDirectory>${project.build.directory}/classes</outputDirectory>
    <finalName>${project.artifactId}-${project.version}</finalName>
    <testOutputDirectory>${project.build.directory}/test-classes</testOutputDirecto
    <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>
    <!-- TODO: MNG-3731 maven-plugin-tools-api < 2.4.4 expect this to be relative..
    <scriptSourceDirectory>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>
       <plugins>
         <plugin>
           <artifactId>maven-antrun-plugin</artifactId>
           <version>1.3</version>
         </plugin>
         <plugin>
           <artifactId>maven-assembly-plugin</artifactId>
           <version>2.2-beta-2</version>
         </plugin>
         <plugin>
           <artifactId>maven-clean-plugin</artifactId>
           <version>2.2</version>
         </plugin>
         <plugin>
```

[top]

19.1.3 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:

```
<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, as 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 `http://repo1.maven.org/maven2` which was specified in the Super POM.

[top]

19.1.4 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.

19.1.4.1 Example 1

19.The Scenario

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.

```
<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

19.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>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <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 we want the groupId and / or the version of your modules to be the same as their parents, you can remove the groupId and / or the version identity of your module in its POM.

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

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

[top]

19.1.4.2 Example 2

19.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

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

19.The Solution

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

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

19.1.5 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)

[top]

19.1.5.1 Example 3

19.The Scenario

Given the previous original artifact POMs, and directory structure,

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

```
<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

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

directory structure

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

19.The Solution

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

```
<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, it's 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]

19.1.5.2 Example 4

19.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?

19.The Solution

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

```
<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>
```

19.1.6 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)

[top]

19.1.6.1 Example 5

19.The Scenario

Given the previous original artifact POMs again,

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

```
<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

```
<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
 |   `-- pom.xml
 `-- parent
     `-- pom.xml
```

19.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

```
<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

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

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

[top]

19.1.7 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:

```
  <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.

19.1.7.1 Available Variables

19.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.

19.Special Variables

`basedir`	The directory that the current project resides in.
`project.baseUri`	The directory that the current project resides in, represented as an URI. *Since Maven 2.1.0*
`maven.build.timestamp`	The timestamp that denotes the start of the build. *Since Maven 2.1.0-M1*

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

```
<project>
  ...
  <properties>
    <maven.build.timestamp.format>yyyyMMdd-HHmm</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.

19.Properties

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

```
<project>
  ...
  <properties>
    <mavenVersion>2.1</mavenVersion>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-artifact</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-project</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
  </dependencies>
  ...
</project>
```

[top]

20 Profiles

20.1 Introduction to Build Profiles

Maven 2.0 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 inhertiance, 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 2.0 introduces the concept of a build profile. 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 runnning with one POM only.

20.1.1 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 (`%M2_HOME%/conf/settings.xml`).
- Profile descriptor
  - a descriptor located in project basedir (`profiles.xml`)

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

A profile can be triggered/activated in several ways:

- Explicitly
- Through Maven settings
- Based on environment variables
- OS settings
- Present or missing files

20.1.2.1 Details on profile activation

Profiles can be explicitly specified using the `-P` CLI option.

This option takes an argument that is a comma-delimited list of profile-ids to use. When this option is specified, no profiles other than those specified in the option argument will be activated.

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

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 everytime 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 follwing 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.

```
<profiles>
  <profile>
    <activation>
      <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
  </activation>
  ...
  </profile>
</profiles>
```

This will activate the profile when the system property "debug" is specified with any value:

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

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

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

Note: Environment variable `FOO` would be set like `env.FOO`.

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

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

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

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

Note: The tags `<exists>` and `<missing>` could be interpolated with some patterns like `${user.home}`.

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

```
<profiles>
  <profile>
    <id>profile-1</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    ...
  </profile>
</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.

20.1.2.2 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:

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

This can be used to deactivate profiles marked as activeByDefault or profiles that would otherwise be activated through their activation config.

20.1.3 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.

20.1.3.1 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}`.

20.1.3.2 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>`
- `<reporting>`
- `<dependencyManagement>`
- `<distributionManagement>`
- a subset of the `<build>` element, which consists of:
  - `<defaultGoal>`
  - `<resources>`
  - `<testResources>`
  - `<finalName>`

20.1.3.3 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.

20.1.4 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.

20.1.4.1 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 critters 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 `~/.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.

20.1.4.2 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:

```
<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`:

```
<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.

20.1.5 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 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`.

20.1.6 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.

21 Repositories

21.1 Introduction to Repositories

21.1.1 Artifact Repositories

A repository in Maven is used to hold build artifacts and dependencies of varying types.

There are strictly only two types of repositories: local and remote. The local repository refers to a copy on your own installation that is a cache of the remote downloads, and also contains the temporary build artifacts that you have not yet released.

Remote repositories refer to any other type of repository, accessed by a variety of protocols such as `file://` and `http://`. These repositories might be a truely remote repository set up by a third party to provide their artifacts for downloading (for example, repo1.maven.org houses Maven's central repository). 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.

The local and remote repositories are structured the same way so that scripts can easily be run on either side, or they can be synced for offline used. In general use, the layout of the repositories is completely transparent to the Maven user, however.

21.1.2 Why not Store JARs in CVS?

It is not recommended that you store your JARs in CVS. Maven tries to promote the notion of a user local repository where JARs, or any project artifacts, can be stored and used for any number of builds. Many projects have dependencies such as XML parsers and standard utilities that are often replicated in typical builds. With Maven these standard utilities can be stored in your local repository and shared by any number of builds.

This has the following advantages:

- **It uses less storage** - while a repository is typically quite large, because each JAR is only kept in the one place it is actually saving space, even though it may not seem that way
- **It makes checking out a project quicker** - initial checkout, and to a small degree updating, a project will be faster if there are no large binary files in CVS. While they may need to be downloaded again afterwards anyway, this only happens once and may not be necessary for some common JARs already in place.
- **No need for versioning** - CVS and other source control systems are designed for versioning files, but external dependencies typically don't change, or if they do their filename changes anyway to indicate the new version. Storing these in CVS doesn't have any added benefit over keeping them in a local artifact cache.

21.1.3 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).

21.1.3.1 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 `repositories` element as follows:

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>http://myserver/repo</url>
    </repository>
  </repositories>
  ...
</project>
```

You can set this in your `settings.xml` file to globally use a certain mirror, however note that it is common for a project to customise the repository in their `pom.xml` and that your setting will take precedence. If you find that dependencies are not being found, check you have not overridden the remote repository.

For more information on dependencies, see Dependency Mechanism.

21.1.3.2 Using Mirrors for the Central Repository

Like any server, the central repository sometimes goes down. If this happens 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.

21.1.4 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
```

Note that many plugins will honour the offline setting and not perform any operations that would connect to the internet. Some examples are resolving Javadoc links and link checking the site.

21.1.5 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.

21.2 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 from using HTTP or the file system (using a `file://` URL), and uploaded to using SCP, FTP, or a file copy.

Note that 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.

21.2.1 Setting up the Internal Repository

To set up an internal repository just requires that you have a place to put it, and then start copying required artifacts there using the same layout as in a remote repository such as repo1.maven.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, to 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.

21.2.2 Using the Internal Repository

Using the internal repository is quite simple. Simply make a change to add a `repositories` element:

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>http://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.

21.2.3 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 share.

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.

22 Standard Directory Layout

22.1 Introduction to the Standard Directory Layout

Having a common directory layout would allow for 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. Please try to conform to this structure as much as possible; however, if you can't these settings can be overridden via the project descriptor.

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
README.txt	Project's readme

At the top level files descriptive of the project: a `pom.xml` file (and any properties, `maven.xml` or `build.xml` if using Ant). 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` 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.

23 The Dependency Mechanism

23.1 Introduction to the Dependency Mechanism

Dependency management is one of the features of Maven that is best known to users and is one of the areas where Maven excels. There is not much difficulty in managing dependencies for a single a project, but when you start getting into dealing with multi-module projects and applications that consist of tens or hundreds of modules this is where Maven can help you a great deal in maintaining a high degree of control and stability.

Learn more about:

- Transitive Dependencies
    - Excluded/Optional Dependencies
- Dependency Scope
- Dependency Management
    - Importing Dependencies
- System Dependencies

23.1.1 Transitive Dependencies

Transitive dependencies are a new feature in Maven 2.0. This allows you to avoid needing to discover and specify the libraries that your own dependencies require, and including them 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, and will only cause a problem if a cyclic dependency is discovered.

With transitive dependencies, the graph of included libraries can quickly grow quite large. For this reason, there are some additional features that will limit which dependencies are included:

- *Dependency mediation* - this determines what version of a dependency will be used when multiple versions of an artifact are encountered. Currently, Maven 2.0 only supports using the "nearest definition" which means that it will use 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, until Maven 2.0.8 it was not defined which one would win, but since Maven 2.0.9 it's the order in the declaration that counts: the first declaration wins.
    - "nearest definition" means that the version used will be the closest one to your project in the tree of dependencies, eg. if 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
- *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.")

23.1.2 Dependency Scope

Dependency scope is used to limit the transitivity of a depedency, and also to affect the classpath used for various build tasks.

There are 6 scopes available:

- **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. This scope 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.
- **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** *(only available in Maven 2.0.9 or later)*
  This scope is only used on a dependency of type `pom` in the `<dependencyManagement>` section. It indicates that the specified POM should be replaced with the dependencies in that 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, transitive dependencies of that dependency with the scope across the top row will result in a dependency in the main project with the scope listed at the intersection. If no scope is listed, it means the dependency will be omitted.

	compile	provided	runtime	test
compile	compile(*)	-	runtime	-
provided	provided	provided	provided	-
runtime	runtime	-	runtime	_

test	test	-	test	-

(*) Note: it is intended that this should be runtime scope instead, so that all compile dependencies must be explicitly listed - however, there is the case where the library you depend on extends a class from another library, forcing you to have available at compile time. For this reason, compile time dependencies remain as compile scope even when they are transitive.

23.1.3 Dependency Management

The dependency management section is a mechanism for centralizing dependency information. When you have a set of projects that inherits 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:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
      <version>1.0</version>
      <exclusions>
        <exclusion>
          <groupId>group-c</groupId>
          <artifactId>excluded-artifact</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>bar</type>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

Project B:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-c</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>war</type>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>bar</type>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

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:

```
<project>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>group-a</groupId>
        <artifactId>artifact-a</artifactId>
        <version>1.0</version>
        <exclusions>
          <exclusion>
            <groupId>group-c</groupId>
            <artifactId>excluded-artifact</artifactId>
          </exclusion>
        </exclusions>
      </dependency>
      <dependency>
        <groupId>group-c</groupId>
        <artifactId>artifact-b</artifactId>
        <version>1.0</version>
        <type>war</type>
        <scope>runtime</scope>
      </dependency>
      <dependency>
        <groupId>group-a</groupId>
        <artifactId>artifact-b</artifactId>
        <version>1.0</version>
        <type>bar</type>
        <scope>runtime</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

And then the two child poms would become much simpler:

```
<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>
    <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>
```

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:

```
<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>
```

Project B:

```
<project>
  <parent>
    <artifactId>A</artifactId>
    <groupId>maven</groupId>
    <version>1.0</version>
  </parent>
  <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>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>
```

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 will 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.

23.1.3.1 Importing Dependencies

The features defined in this section are only available in Maven 2.0.9 or later. This means that poms declaring the import scope will not be parseable by earlier versions of Maven. Weigh this information carefully before deciding to use it. If you do use it, we suggest you use the enforcer plugin to require a minimum Maven version of 2.0.9. We currently do not recommend using this for projects that get deployed to Central.

The examples in the previous section describe how to specify managed dependencies through inheritence. However, in larger projects it may be impossible to accomplish this since a project can only inherit from a single parent. To accomodate 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:

```
<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>
 <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>
  <groupId>maven</groupId>
  <artifactId>Z</artifactId>
  <packaging>pom</packaging>
  <name>Z</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>X</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>Y</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

In the example above Z imports the managed dependencies from both X and Y. However, both X and Y contain depedency 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.

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.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/x
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test</groupId>
  <artifactId>bom</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <properties>
    <project1Version>1.0.0</project1Version>
    <project2Version>1.0.0</project2Version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>project1</artifactId>
        <version>${project1Version}</version>
      </dependency>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>project2</artifactId>
        <version>${project1Version}</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <modules>
    <module>parent</module>
  </modules>
</project>
```

The parent subproject has the BOM pom as its parent. It is a normal multiproject pom.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
  <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.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
  <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>${project1Version}</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
    </dependency>
  </dependencies>
</project>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>parent</artifactId>
  </parent>
  <groupId>com.test</groupId>
  <artifactId>project2</artifactId>
  <version>${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.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test</groupId>
  <artifactId>use</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>bom</artifactId>
        <version>1.0.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.test</groupId>
      <artifactId>project1</artifactId>
    </dependency>
    <dependency>
      <groupId>com.test</groupId>
      <artifactId>project2</artifactId>
    </dependency>
  </dependencies>
</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 cicularity and an exception will be thrown.
- When referring to artifacts whose poms have transitive dependencies the project will need to specify versions of those artifacts as managed dependencies. Not doing so will result 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).

23.1.4 System Dependencies

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 where available as separate downloads earlier. Typical example are the JDBC standard extensions or the Java Authentication and Authorization Service (JAAS).

A simple example would be:

```
<project>
  ...
  <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>
  ...
</project>
```

If your artifact is provided by the JDK's `tools.jar` the system path would be defined as follows:

```
<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>
```

24 Plugin Development

24.1 Introduction to Maven 2.0 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.

24.1.1 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 2.0 is the Clean Plugin. The Maven Clean plugin (maven-clean-plugin) is responsible for removing the target directory of a Maven 2 project. When you run "mvn clean", Maven 2 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}.

24.1.2 What is a Mojo (*And Why the H--- is it Named 'Mojo'*)?

A Mojo is really just a goal in Maven 2, 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 iteresting word (see definition). From Wikipedia, a "mojo" is defined as: "...a small bag 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."

24.1.3 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.

24.2 Resources

1 Plugin development guide
2 Configuring plugins

24.3 Comparison to Maven 1.x Plugins

24.3.1 Similarities to Maven 1.x

Maven 2.0 is similar to its predecessor in that it has two main functions. First, it organizes project data into a coherent whole, and exposes this data for use within the build process. Second, Maven marshals a set of plugins to do the heavy lifting and execute the actual steps of the build.

Many things in Maven 2 will have at least superficial familiarity to users of Maven 1, and the plugin system is no exception. Maven 2 plugins appear to behave much as their 1.x counterparts do. Like 1.x plugins, they use both project information and custom-defined configurations to perform their work. Also, Maven 2 plugins are organized and executed in a coherent way by the build engine itself - that is to say, the engine is still responsible for organizing and fulfilling a plugin's requirements before executing the plugin itself.

Operationally, Maven 2.0 should feel very much like a more performant big brother of Maven 1.x. While the POM has definitely changed, it has the same basic layout and features (with notable additions). However, this is where the similarity ends. Maven 2.0 is a complete redesign and reimplementation of the Maven build concept. As such, it has a much different and more evolved architecture - at least to our minds. **;-)**

24.3.2 Differences from Maven 1.x

However similar the architectures may seem, Maven 2 offers a much richer environment for its plugins than Maven 1 ever did. The new architecture offers a managed lifecycle, multiple implementation languages, reusability outside of the build system, and many more advantages. Arguably the biggest advantage is the ability to write Maven plugins entirely in Java, which allows developers to tap into a rich landscape of development and testing tools to aid in their efforts.

Prior to Maven 2.0, the build system organized relevant plugins into a loosely defined lifecycle, which was determined based on goal prerequisites and decoration via preGoals and postGoals. That experience was critical for the Maven community. It taught us that even though there may be a million different build scenarios out there, most of the activities in those builds fit into just a few broad categories. Moreover, the category to which a goal fits serves as an accurate predictor for where in the build process the goal should execute. Drawing on this experience, Maven 2.0 defines a lifecycle within which plugins are managed according to their relative position within this lifecycle.

Starting with Maven 2.0, plugins implemented in different programming or scripting languages can coexist within the same build process. This removes the requirement that plugin developers learn a particular scripting language in order to interact with Maven. It also reduced the risk associated with the stability or richness of any particular scripting language.

Also starting with Maven 2.0 is an effort to integrate multiproject builds directly into the core architecture. In Maven 1.x, many large projects were fragmented into smaller builds to sidestep issues such as conditional compilation of a subset of classes; separation of client-server code; or cyclical dependencies between distinct application libraries. This in turn created extra complexity with running builds, since multiple builds had to be run in order to build the application as a whole - one or more per project. While the first version (1.x) did indeed address this new multiple projects issue, it did so as an afterthought. The Reactor was created to act as a sort of *apply-to-all-these* function, and the multiproject plugin was later added to provide Reactor settings for some common build types. However, this solution (it *is* really only one solution, plus some macros) really never integrated the idea of the multi-project build process into the maven core conceptual framework.

24.3.3 Why Change the Plugin Architecture?

See the previous section for the long version, but the short version can be summed up by the following list of benefits.

- A managed lifecycle
- Multiple implementation languages
- Reusability outside of the build system
- The ability to write Maven plugins entirely in Java

In Maven 1.0, a plugin was defined using Jelly, and while it was possibly to write a plugin in Java, you still had to wrap your plugin with some obligatory Jelly script. An XML-based scripting language

which is interpreted at run-time isn't going to be the best choice for performance, and the development team thought it wise to adopt an approach which would allow plugin developers to choose from an array of plugin implementation choices. The first choice in Maven 2 should be Java plugins, but you may also use one of the supported scripting languages like Beanshell.

To summarize, the development team saw some critical gaps in the API and architecture of Maven 1.0 plug-ins, and the team decided that addressing these deficiencies was critical to the future progress of Maven from a useful tool to something more robust.

25 Configuring Plug-ins

25.1 Guide to Configuring Plug-ins

25.1.1 Introduction

In Maven, there are the build and the reporting plugins:

- **Build plugins** will be executed during the build and then, they should be configured in the `<build/>` element.
- **Reporting plugins** will be executed during the site generation and they should be configured in the `<reporting/>` element.

All plugins should have minimal required informations: `groupId`, `artifactId` and `version`.

Important Note: It is recommended to always defined 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, you should specify each version in the `<reporting><plugins/></reporting>` elements (and surely in the `<build><pluginManagement/></build>` elements too).

25.1.2 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, we had a Mojo that performed a query against a particular URL, with a specified timeout and list of options. The Mojo might look like the following:

```
/**
 * @goal query
 */
public class MyQueryMojo
    extends AbstractMojo
{
    /**
     * @parameter expression="${query.url}"
     */
    private String url;
    /**
     * @parameter default-value="60"
     */
    private int timeout;
    /**
     * @parameter
     */
    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>
```

As you can see the elements in the configuration match the names of the fields in the Mojo. The configuration mechanism Maven employs is very similar to the way XStream works where elements in XML are mapped to objects. So from the example above you can see that the mapping is pretty

straight forward 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:

```
mvn myquery:query -Dquery.url=http://maven.apache.org
```

Note that 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.

25.1.2.1 Help Goal

Recent Maven plugins have generally an `help` goal to have in the command line the description of the plugin, with their parameters and types. For instance, to understand the javadoc goal, you need to call:

```
mvn javadoc:help -Ddetail -Dgoal=javadoc
```

And you will see all parameters for the javadoc:javadoc goal, similar to this page.

25.1.2.2 Configuring Parameters

25. 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>
...
```

25. Mapping Complex Objects

Mapping complex types is also fairly straight forward in Maven so 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`. As you can see the mechanism will capitalize the first letter of the element name and use 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 then you must 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>
...
```

25. 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:

25. 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:

```
public class MyAnimalMojo
    extends AbstractMojo
{
    /**
     * @parameter
     */
    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>
  ...
</project>
```

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.

25. Mapping Maps

In the same way, you could define maps like the following:

```
...
    /**
     * My Map.
     *
     * @parameter
     */
    private Map myMap;
...
```

```
...
  <configuration>
    <myMap>
      <key1>value1</key1>
      <key2>value2</key2>
    </myMap>
  </configuration>
...
```

25. Mapping Properties

Properties should be defined like the following:

```
...
    /**
     * My Properties.
     *
     * @parameter
     */
    private Properties myProperties;
...
```

```
...
  <configuration>
    <myProperties>
      <property>
        <name>propertyName1</name>
        <value>propertyValue1</value>
      <property>
      <property>
        <name>propertyName2</name>
        <value>propertyValue2</value>
      <property>
    </myProperties>
  </configuration>
...
```

25.1.3 Configuring Build Plugins

The following is only to configure Build plugins in the `<build/>` element.

25.1.3.1 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:

```
<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>
  ...
</project>
```

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.

```
<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.

```
/**
 * @goal query
 * @phase package
 */
public class MyBindedQueryMojo
    extends AbstractMojo
{
    /**
     * @parameter expression="${query.url}"
     */
    private String url;
    /**
     * @parameter default-value="60"
     */
    private int timeout;
    /**
     * @parameter
     */
    private String[] options;
    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

From the above mojo example, `MyBindedQueryMojo` 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>`.

```
<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>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Now, `MyBindedQueryMojo` default phase which is package has been overrided by install phase.

Note: Configurations inside the `<executions>` tag differ from those that are outside `<executions>` in that they cannot be used from a direct command line invocation. Instead they are only applied when the lifecycle phase they are bound to are invoked. Alternatively, if you move a configuration section outside of the executions section, it will apply globally to all invocations of the plugin.

25.1.3.2 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:

```
<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>
```

25.1.3.3 Using the <inherited/> Tag In Build Plugins

By default, plugin configuration should be propagated to child POMs, so to break the inheritance, you could uses the <inherited/> tag:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.2</version>
        <inherited>false</inherited>
        ...
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

25.1.4 Configuring Reporting Plugins

The following is only to configure Reporting plugins in the <reporting/> element.

25.1.4.1 Using the `<reporting/>` Tag VS `<build/>` Tag

Configuring a reporting plugin in the <reporting/> or <build/> elements in the pom have **NOT** the same behavior!

`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/>.

`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/>.

25.1.4.2 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.

```
<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>
```

Notes:

1 To exclude all reports, you need to use:

```
  <reportSets>
    <reportSet>
      <reports/>
    </reportSet>
  </reportSets>
```

2 Refer to each Plugin Documentation (i.e. plugin-info.html) to know the available report goals.

25.1.4.3 Using the `<inherited/>` Tag In Reporting Plugins

Similar to the build plugins, to break the inheritance, you could uses the `<inherited/>` tag:

```
<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>
```

26 The Plugin Registry

26.1 Introduction to the Plugin Registry

The Maven 2 plugin registry (~/.m2/plugin-registry.xml) is a mechanism to help the user exert some control over their build environment. Rather than simply fetching the latest version of every plugin used in a given build, this registry allows the user to peg a plugin to a particular version, and only update to newer versions under certain restricted circumstances. There are various ways to configure or bypass this feature, and the feature itself can be managed on either a per-user or global level.

26.1.1 Warning!

The plugin registry is currently in a semi-dormant state within Maven 2. This is because it has been shown to have some subtle behavior that is not quite intuitive. While we believe it's important to allow the user to pin down which version of a particular plugin is used across all builds, it's not clear that this type of information should be machine-specific (i.e. tied to something outside the project directory).

Users should be cautious when attempting to use the `plugin-registry.xml`. Redesign of this feature in upcoming 2.1 and/or 2.2 is likely.

For now, Maven *should* keep using the same version of a plugin - assuming a different version is not specified in the POM - until the user chooses to run with the `-U` option explicitly enabled.

26.1.2 A Tour of plugin-registry.xml

The plugin registry file (per-user: ~/.m2/plugin-registry.xml, global: $M2_HOME/conf/plugin-registry.xml) contains a set of plugin-version registrations, along with some configuration parameters for the registry itself.

Currently, the plugin registry supports configuration options for the following:

- **updateInterval** - Determines how often (or whether) the registered plugins are checked for updates.

  Combined with the **lastChecked** plugin attribute, this determines whether a particular plugin will be checked for updates during a given build. Valid settings are: never, always, and interval:TTT (TTT is a short specification for a time interval, which follows the pattern /([0-9]+[wdhm])+/). Intervals are specified down to the minute resolution. An example of an interval specification might be:

  `interval:4w2h30m` (check every 4 weeks, 2 hours, and 30 minutes)
- **autoUpdate** - Specifies whether the user should be prompted before registering plugin-version updates. This is a boolean value, accepting true/false.
- **checkLatest** - Specifies whether the LATEST artifact metadata should be consulted while determining versions for *unregistered* plugins.

  LATEST metadata is always published when a plugin is installed or deployed to a repository, and so will always reference the newest copy of the plugin, regardless of whether this is a snapshot version or not.

  NOTE: Registered plugins will currently only ever be updated with the results of RELEASE metadata resolution.

  Obviously, the plugin registry also contains information about resolved plugin versions. The following information is tracked for each registered plugin:
- **groupId** - The plugin's group id.

- **artifactId** - The plugin's artifact id.
- **lastChecked** - The timestamp from the last time updates were checked for this plugin.
- **useVersion** - The currently registered version for this plugin. This is the version Maven will use when executing this plugin's mojos.
- **rejectedVersions** - A list of versions discovered for this plugin which have been rejected by the user. This keeps Maven from continually prompting the user to update a given plugin to the same new version.

26.1.3 Using (or not) the Plugin Registry

There are many ways you can override the default plugin registry settings. Often, this will be desirable for a single, one-off build of a project that deviates from your normal environment configuration. However, before discussing these options, it's important to understand how the plugin registry resolves versions for unregistered plugins, along with plugins in need of an update check.

26.1.3.1 Resolving Plugin Versions

The plugin registry uses a relatively straightforward algorithm for resolving plugin versions. However, considerations for when to check, when to prompt the user, and when to persist resolved plugin versions complicate this implementation considerably. In general, plugin versions are resolved using a four-step process:

1 Check for a plugin configuration in the MavenProject instance. Any plugin configuration found in the MavenProject is treated as authoritative, and will stop the plugin-version resolution/persistence process when found.

2 If the plugin registry is enabled, check it for an existing registered version. If the plugin has been registered, a combination of the updateInterval configuration and the lastChecked attribute (on the plugin) determine whether the plugin needs to be checked for updates. If the plugin doesn't need an update check, the registered version is used.

   If the plugin is due for an update check, the plugin-artifact's RELEASE metadata is resolved. Resolution of this metadata may trigger a prompt to notify the user of the new version, and/or persistence of the new version in the registry. If the update is performed, the lastChecked attribute is updated to reflect this.

3 *If* the **checkLatest** configuration is set to `true`, or the `'--check-plugin-latest'` CLI option (discussed later) is provided, then the LATEST artifact metadata is resolved for the plugin.

   If this metadata is resolved successfully, that version is used. This may trigger a prompt to ask the user whether to register the plugin, and a successive persistence step for the new plugin version.

4 If the version still has not been resolved, RELEASE artifact metadata is checked for the plugin. If this metadata resolves successfully, it is the version used. This may also trigger a prompt to ask the user whether to register the plugin, and a persistence step registering the new plugin version.

   I've alluded to prompting the user and persisting the plugin version into the registry. Now, let's examine the circumstances under which these steps actually take place.

   There are two cases where the user may be prompted to change the plugin registry; when the plugin is not registered, and when the plugin is registered, but an updated version is discovered. By default, the user is prompted to save the resolved version for each plugin, with the option of specifying that a decision should be remembered and applied to all (either yes to all, or no to all) plugins registry updates. However, it is also possible to bypass this behavior in the following ways:

- Specify **autoUpdate** == `true` in the plugin-registry.xml. This configuration parameter means that the user is not prompted, and all updated/discovered versions are to be persisted.
- Specify `'--batch-mode'` (or `'-B'`) from the command line. This functions in the same way as the **autoUpdate** config parameter above.
- Specify `'--no-plugin-updates'` | `'-npu'` from the command line. This prevents any updates or new registrations from taking place, but existing plugin versions in the registry will be used when available.
- Specify `'--check-plugin-updates'` | `'--update-plugins'` | `'-up'` | `'-cpu'` (synonyms) from the command line.
- Specify `'--no-plugin-registry'` | `'-npr'` from the command line. This prevents resolution of plugin versions using the plugin-registry.xml file. The plugin version will be resolved either from the project or from the repository in this case.
- Specify **usePluginRegistry** == `false` in the settings.xml. This configuration parameter will disable use of the plugin registry for the entire build environment, as opposed to the immediate build execution (as in the case of the corresponding command line switch above).

  These force all registered plugins to be updated. The user will still be prompted to approve plugin versions, unless one of the above switches is also provided.

26.1.3.2 Summary of Command Line Options for the Plugin Registry

The following summary of command line options is provided for quick reference:

- `--no-plugin-registry` - Bypass the plugin registry.

  Synonym: `-npr`
- `--no-plugin-latest` - Don't check the LATEST artifact metadata when resolving plugin versions, regardless of the value of **useLatest** in the plugin-registry.xml file.

  Synonym: `-npl`
- `--check-plugin-latest` - Check the LATEST artifact metadata when resolving plugin versions, regardless of the value of **useLatest** in the plugin-registry.xml file.

  Synonym: `-cpl`
- `--no-plugin-updates` - Do not search for updated versions of registered plugins. Only use the repository to resolve unregistered plugins.

  Synonym: `-npu`
- `--check-plugin-updates` - Force the plugin version manager to check for updated versions of any registered plugins, currently using RELEASE metadata **only**.

  Synonyms: `--update-plugins` `-up` `-cpu`

27 Plugin Prefix Resolution

27.1 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.

27.1.1 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 assuming the plugin's artifactId fits the pattern:

```
maven-${prefix}-plugin
```

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:

```
<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:

```
mvn somePrefix:goal
```

27.1.2 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.

27.1.3 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: `~/.m2/settings.xml`; global: `$M2_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:

```
<pluginGroups>
  <pluginGroup>org.codehaus.modello</pluginGroup>
</pluginGroups>
```

This allows me to execute the following, which will generate Java classes from the model:

```
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].

27.1.4 Resources

1 Guide to Configuring Maven

28 Developing Ant Plugins

28.1 Developing Ant Plugins for Maven 2.x

28.1.1 WARNING

The documentation below assumes that you have updated your locally cached cached copy of the maven-plugin-plugin. To update your copy, you will need to include the -U option when you build your plugin project:

```
mvn -U clean install
```

The maven-plugin-plugin is responsible for reading plugin metadata in its various forms and writing a standard Maven plugin descriptor based on the input. It was designed to accommodate multiple plugin languages side by side, but its initial design was slightly flawed for plugin languages that don't include the metadata inline with the source (within the same file). Since the 2.0.1 release of Maven, the maven-plugin-plugin has contained revisions to handle this scenario. Since the API has changed (in a backward-compatible way), and since the Ant plugin support requires these changes be in place, you will see an `AbstractMethodError` if you try to build an Ant-based plugin using the old maven-plugin-plugin.

28.1.2 Introduction

The intent of this document is to help users learn to develop Maven plugins using Ant.

As of the 2.0.1 release, Maven supports Ant-driven plugins. These plugins allow the invocation of Ant targets (specified in scripts embedded in the plugin jar) at specific points in the build lifecycle. They can also inject parameter values into the Ant project instances when a target is called.

28.1.3 Conventions

In this guide, we'll use the standard Maven directory structure for projects, to keep our POMs as simple as possible. It's important to note that this is only a standard layout, not a requirement. The important locations for our discussion are the following:

```
/<project-root>
|
+- pom.xml
|
+- /src
|  |
|  +- /main
|  |  |
|  |  +- /scripts (source location for script-driven plugins)
|  |  |  |
...
```

28.1.4 Getting Started

We'll start with the simplest of all possible plugins. This plugin takes no parameters, and will simply print a message out to the screen when invoked. This should familiarize the reader with the basics of mapping Ant build scripts to the Maven plugin framework. From there, we will gradually increase the complexity of our plugin, adding parameters, interacting with standard project locations, and binding

to lifecycle phases. Finally, we'll see how a single Ant build script can be mapped to multiple Maven mojos within the same plugin.

28.1.4.1 Hello, World

Our first plugin will simply print "Hello, World" to the console.

28.The Build Script

The elemental Ant-driven mojo consists of a simple Ant build script, a mapping metadata file, and of course the plugin's POM. If our goal is to print "Hello, World", we might use an Ant build script that looks something like this:

```
hello.build.xml:
--------------------------------
<project>
  <target name="hello">
    <echo>Hello, World</echo>
  </target>
</project>
```

28.The Mapping Document

Once we've created this build script, we need to tell Maven how to use it as a plugin. This involves creating a mapping document. Note that where the build script was named **hello.build.xml**, the mapping document is named **hello.mojos.xml**. The naming of these files is very important, as this is how the plugin parser matches mapping documents to build scripts. It has the general form:

- *basename* **.build.xml** - The Ant build script.
- *basename* **.mojos.xml** - The corresponding mapping document.

A simple mapping document used to wire the above build script into Maven's plugin framework might look as follows:

```
hello.mojos.xml:
-----------------------------
<pluginMetadata>
  <mojos>
    <mojo>
      <goal>hello</goal>

      <!-- this element refers to the Ant target we'll invoke -->
      <call>hello</call>
      <description>
        Say Hello, World.
      </description>
    </mojo>
  </mojos>
</pluginMetadata>
```

28.The POM

Now that we have the build script and mapping document, we're ready to build this plugin. However, in order to build, we need to provide a POM for our new plugin. As it turns out, the POM required for an Ant-driven plugin is fairly complex. This is because we have to configure the maven-plugin-plugin to use the Ant plugin parsing tools in addition to the defaults (such as the Java parsing tools). Our POM might look something like this:

```
pom.xml:
-----------------------------
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.myproject.plugins</groupId>
  <artifactId>hello-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>

  <packaging>maven-plugin</packaging>

  <name>Hello Plugin</name>

  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-script-ant</artifactId>
      <version>2.0.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <!-- NOTE: We don't need groupId if the plugin's groupId is
             org.apache.maven.plugins OR org.codehaus.mojo.
        -->
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>

        <!-- Add the Ant plugin tools -->
        <dependencies>
          <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-tools-ant</artifactId>
            <version>2.0.1</version>
          </dependency>
        </dependencies>

        <!-- Tell the plugin-plugin which prefix we will use.
             Later, we'll configure Maven to allow us to invoke this
             plugin using the "prefix:mojo" shorthand.
        -->
        <configuration>
          <goalPrefix>hello</goalPrefix>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

28.Build It and Run It

Once we have a POM for our new plugin, we can install it into the local repository just as we would any other Maven project:

```
mvn install
```

and invoke it like this:

```
mvn org.myproject.plugins:hello-plugin:hello
```

This should output the following:

```
[echo] Hello, World
```

28.1.5 Using prefix:mojo Invocation Syntax

Our new plugin works, but look at that command line... The next thing is to configure Maven so we can use the familiar **prefix:mojo** invocation syntax, and leave that verbose, fully-qualified mess behind.

As you know, Maven maps plugins to user-friendly prefixes. However, these prefixes might overlap; that is, multiple plugins may try to use the same prefix inadvertently. To avoid the obvious ambiguity associated with such a collision, Maven will search a predetermined list of plugin groupIds for a given prefix, with the first match winning. So, if we want to add our new plugin to this search, we need to configure the list of plugin groupIds.

28.1.5.1 Configuring Plugin-Prefix Searching

In order to reference our new plugin by prefix, we need to add its groupId to the `<pluginGroups/>` list in the `settings.xml` file. As you probably know, this file is usually found under $HOME/.m2. The added section to make our plugin's groupId searchable should look like this:

```
~/.m2/settings.xml:
------------------------------
<settings>
  .
  .
  .
  <pluginGroups>
    <pluginGroup>org.myproject.plugins</pluginGroup>
  </pluginGroups>
  .
  .
  .
</settings>
```

28.1.5.2 Run It

We can check that this worked by invoking our new mojo once again, this time using the prefix syntax:

```
mvn hello:hello
```

28.1.6 Adding Plugin Parameters

Now, suppose it's not enough that our plugin display static text to the console. Suppose we need it to display a greeting that is a little more personalized. We can easily add support for this by adding a **name** parameter. For good measure, we'll output the current project's name as well.

28.1.6.1 Change the Ant Script

The build script will have to change to output the new information:

```
hello.build.xml:
--------------------------------
<project>
  <target name="hello">
    <echo>Hello, ${name}. You're building project: ${projectName}</echo>
  </target>
</project>
```

28.1.6.2 Change the Mapping Document

Now that we have a build script which requires two new parameters, we have to tell the mapping document about them, so they will be injected into the Ant Project instance.

```
hello.mojos.xml:
-----------------------------
<pluginMetadata>
  <mojos>
    <mojo>
      <goal>hello</goal>

      <!-- this element refers to the Ant target we'll invoke -->
      <call>hello</call>

      <requiresProject>true</requiresProject>

      <description>
        Say Hello, including the user's name, and print the project name to the con
      </description>
      <parameters>
        <parameter>
          <name>name</name>
          <property>name</property>
          <required>true</required>
          <expression>${name}</expression>
          <type>java.lang.String</type>
          <description>The name of the user to greet.</description>
        </parameter>

        <parameter>
          <name>projectName</name>
          <property>projectName</property>
          <required>true</required>
          <readonly>true</readonly>
          <defaultValue>${project.name}</defaultValue>
          <type>java.lang.String</type>
          <description>The name of the project currently being built.</description>
        </parameter>
      </parameters>
    </mojo>
  </mojos>
</pluginMetadata>
```

You'll notice several differences from the old version of the mapping document. First, we've added **requiresProject="true"** to the mojo declaration. This tells Maven that our mojo requires a valid project before it can execute...in our case, we need a project so we can determine the correct **projectName** to use. Next, we've added two parameter declarations to our mojo mapping; one for **name** and another for **projectName**.

The **name** parameter declaration provides an expression attribute...this allows the user to specify `-Dname=somename` on the command line. Otherwise, the only way to configure this parameter would be through a `<configuration/>` element within the plugin specification in the user's POM. Note that this parameter is required to have a value before our mojo can execute.

The **projectName** parameter declaration provides two other interesting items. First, it specifies a defaultValue attribute, which specifies an expression to be evaluated against Maven's current build state in order to extract the parameter's value. Second, it specifies a readonly attribute, which means the user cannot directly configure this parameter - either via command line or configuration within

the POM. It can only be modified by modifying the build state referenced in the defaultValue...in our case, the name element of the POM. Also note that this parameter is declared to be required before our mojo can execute.

28.1.6.3 Rebuild It and Run It

Now that we've modified our plugin, we have to rebuild it before running it again.

```
mvn clean install
```

Next, we should run the plugin again to verify that it's doing what we expect. However, before we can run it, we have some requirements to satisfy. First, we have to be sure we're executing in the context of a valid Maven POM...runnning in the plugin's own project directory should satisfy that requirement. Then, we have to satisfy the name requirement. We can do this directly through the command line. So, the resulting invocation of our plugin will look like this:

```
mvn -Dname=<your-name-here> hello:hello
```

or, in my case:

```
mvn -Dname=John hello:hello
```

This should output the following:

```
[echo] Hello, John. You're building project: Hello Plugin
```

28.1.7 Defining Multiple Mojos from One Build Script

If you're familiar with Ant, you're probably familiar with the common usage pattern of defining multiple build types within a single build script. For instance, you might have a build type for cleaning the project, another for producing the application jar file, and yet another for producing the full distribution including javadocs, etc.

The concept is pretty simple. Discrete chunks of the build process are separated into targets within the script. These targets can reference one another in order to make reuse within the build script possible.

These same concepts map pretty well to Maven, actually. However, instead of targets directly referencing one another, they would be bound to the appropriate phases of the build lifecycle. In this way, multiple Ant targets from the same build script can be reused piecemeal at different points in multiple build lifecycles (clean, site, and the main lifecycle are three examples).

This section will describe how to map multiple logical mojos onto different targets within the same Ant build script. It's also possible to reference targets from multiple build scripts, but we'll cover this later.

28.1.7.1 Two Targets, One Script

To test this, we'll split our echo statement into two targets. Then, we'll reference each as separate mojos in the build. The new script looks like this:

```
one-two.build.xml:
-------------------------------
<project>
  <target name="one">
    <echo>Hello, ${name}.</echo>
  </target>

  <target name="two">
    <echo>You're building project: ${projectName}</echo>
  </target>
</project>
```

28.1.7.2 Map the Mojos

Next, we'll modify our original mapping document to map these two new mojos instead of the old one:

```
one-two.mojos.xml:
------------------------------
<pluginMetadata>
  <mojos>
    <mojo>
      <goal>one</goal>

      <!-- this element refers to the Ant target we'll invoke -->
      <call>one</call>

      <description>
        Say Hello. Include the user's name.
      </description>
      <parameters>
        <parameter>
          <name>name</name>
          <property>name</property>
          <required>true</required>
          <expression>${name}</expression>
          <type>java.lang.String</type>
          <description>The name of the user to greet.</description>
        </parameter>
      </parameters>
    </mojo>

    <mojo>
      <goal>two</goal>

      <!-- this element refers to the Ant target we'll invoke -->
      <call>two</call>
      <requiresProject>true</requiresProject>

      <description>
        Write the project name to the console.
      </description>
      <parameters>
        <parameter>
          <name>projectName</name>
          <property>projectName</property>
          <required>true</required>
          <readonly>true</readonly>
          <defaultValue>${project.name}</defaultValue>
          <type>java.lang.String</type>
          <description>The name of the project currently being built.</description>
        </parameter>
      </parameters>
    </mojo>
  </mojos>
</pluginMetadata>
```

Now that we've split the old functionality into two distinct mojos, there are some interesting consequences. Aside from the obvious, mojo **one** no longer requires a valid project instance in order to execute, since we only require the user's name in order to greet him.

28.1.7.3 Build It, Run It

28.1.7.4 Rebuild It and Run It

Since we've modified our plugin, we have to rebuild it again before re-running it.

```
mvn clean install
```

Now that we have two separate mojos, we can execute them singly, or in any order we choose. We can bind them to phases of the lifecycle using plugin configuration inside the build element of a POM. We can execute them like this:

```
mvn -Dname=John hello:one
RETURNS:
[echo] Hello, John.
mvn hello:two (executed in the plugin's project directory)
RETURNS:
[echo] You're building project: Hello Plugin
```

Alternatively, you could build a POM like this:

```
test-project/pom.xml:
---------------------------
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.myproject.tests</groupId>
  <artifactId>hello-plugin-tests</artifactId>
  <version>1.0</version>

  <name>Test Project</name>

  <build>
    <plugins>
      <plugin>
        <groupId>org.myproject.plugins</groupId>
        <artifactId>hello-plugin</artifactId>
        <version>1.0</version>

        <configuration>
          <name>John</name>
        </configuration>

        <executions>
          <execution>
            <phase>validate</phase>
            <goals>
              <goal>one</goal>
              <goal>two</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Then, simply call Maven on this new POM:

```
cd test-project
mvn validate
```

You should see the following output:

```
[echo] Hello, John.
...
[echo] You're building project: Test Project
```

28.1.7.5 A Note on Multiple Build Scripts

It's worth mentioning that Ant-driven plugins can just as easily contain multiple Ant build scripts. Simply follow the naming rules - naming each A.build.xml, B.build.xml, C.build.xml, etc. for example - and be sure to provide a mapping document to correspond to each build script that contains a mojo (other build scripts may be contained in the plugin, and referenced by one of these; they don't need mapping documents). So, for the above examples (assuming they all contained mojo targets),

you'd need: A.mojos.xml, B.mojos.xml, and C.mojos.xml. If C.build.xml was referenced by A and B, but didn't contain mojo targets, then you don't need a C.mojos.xml for obvious reasons.

28.1.8 Advanced Usage

Below are some tips on some of the more advanced options related to Ant mojos.

28.1.8.1 Component References

If your plugin needs a reference to a Plexus component, it will have to define something similar to the following in the mapping document:

```
<pluginMetadata>
  <mojos>
    <mojo>
      .
      .
      .
      <components>
        <component>
          <role>org.apache.maven.project.MavenProjectBuilder</role>
          <hint>default</hint> <!-- This is optional -->
        </component>
      </components>
      .
      .
      .
    </mojo>
  </mojos>
</pluginMetadata>
```

28.1.8.2 Forking New Lifecycles

In case your plugin needs to fork a new lifecycle, you can include the following in the mapping document:

```
<pluginMetadata>
  <mojos>
    <mojo>
      .
      .
      .
      <execute>
        <lifecycle>my-custom-lifecycle</lifecycle>
        <phase>package</phase>

        <!-- OR -->

        <goal>some:goal</goal>
      </execute>
      .
      .
      .
    </mojo>
  </mojos>
</pluginMetadata>
```

28.1.8.3 Deprecation

As time goes on, you will likely have to deprecate part of your plugin. Whether it's a mojo parameter, or even an entire mojo, Maven can support it, and remind your users that the mojo or configuration they're using is deprecated, and print a message directing them to adjust their usage.

To deprecate a mojo parameter, simply add this:

```
<pluginMetadata>
  <mojos>
    <mojo>
      .
      .
      .
      <parameters>
        <parameter>
          .
          .
          .
          <deprecated>Use this other parameter instead.</deprecated>
          .
          .
          .
        </parameter>
      </parameters>
      .
      .
      .
    </mojo>
  </mojos>
</pluginMetadata>
```

To deprecate an entire mojo, add this:

```
<pluginMetadata>
  <mojos>
    <mojo>
      .
      .
      .
      <deprecated>Use this other mojo instead.</deprecated>
      .
      .
      .
    </mojo>
  </mojos>
</pluginMetadata>
```

29 Developing Java Plugins

29.1 Introduction

This guide is intended to assist users in developing Java plugins for Maven 2.0.

29.1.1 Your First Plugin

In this section we will build a simple plugin 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, and a couple ways to execute the mojo.

29.1.1.1 Your First Mojo

At its simplest, a Java mojo consists simply of a single class. 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, the class `org.apache.maven.tools.plugin.extractor.java.JavaMojoDescriptorExtractor` looks for classes with a "`goal`" annotation on the class. Any class with this annotation are included in the plugin configuration file.

29.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.

```
package sample.plugin;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
/**
 * Says "Hi" to the user.
 * @goal sayhi
 */
public 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 comment line starting with "`@goal`" is an example of an annotation. This annotation is required, but there are a number of annotations which can be used to 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.

29.1.1.2 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:

`groupId`	This is the group ID for the plugin, and should match the common prefix to the packages used by the mojos
`artifactId`	This is the name of the plugin
`version`	This is the version of the plugin
`packaging`	This should be set to "`maven-plugin`"
`dependencies`	A dependency must be declared to the Maven Plugin Tools API to resolve "`AbstractMojo`" and related classes

Listed below is an illustration of the sample mojo project's pom with the parameters set as described in the above table:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample.plugin</groupId>
  <artifactId>maven-hello-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Sample Parameter-less Maven Plugin</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>2.0</version>
    </dependency>
  </dependencies>
</project>
```

29.1.1.3 Build Goals

There are few goals which are defined with the Maven plugin packaging as part of a standard build lifecycle:

`compile`	Compiles the Java code for the plugin and builds the plugin descriptor
`test`	Runs the plugin's unit tests

`package`	Builds the plugin jar
`install`	Installs the plugin jar in the local repository
`deploy`	Deploys the plugin jar to the remote repository

29.1.1.4 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 `maven-hello-plugin` plugin in you project:

```
...
  <build>
    <plugins>
      <plugin>
        <groupId>sample.plugin</groupId>
        <artifactId>maven-hello-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:maven-hello-plugin:1.0-SNAPSHOT:sayhi`" on the command line.

Tips: `version` is not requiered to run a standalone goal.

29.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:maven-hello-plugin:sayhi`" to run your plugin.
- The "`maven-$name-plugin`" and "`$name-maven-plugin`" artifactId patterns are treated in a special way. If no plugin matches the artifactId specified on the command line, Maven will try expanding the artifactId to these patterns in that order. So in the case of our example, you can use "`mvn sample.plugin:hello:sayhi`" to run your plugin. Note: these 2 patterns are used respectively by the official Maven 2 plugins and the Mojo project plugins.
- 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:

```
<pluginGroups>
  <pluginGroup>sample.plugin</pluginGroup>
</pluginGroups>
```

At this point, you can run the mojo with "`mvn hello:sayhi`".

29.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:

```
<build>
  <plugins>
    <plugin>
      <groupId>sample.plugin</groupId>
      <artifactId>maven-hello-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.

29.1.2 Mojo archetype

To create a new plugin project, you could using the Mojo archetype with the following command line:

```
mvn archetype:create \
  -DgroupId=sample.plugin \
  -DartifactId=maven-hello-plugin \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-mojo
```

29.1.3 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.

29.1.3.1 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:

```
    /**
     * The greeting to display.
     *
     * @parameter expression="${sayhi.greeting}" default-value="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 `default-value` 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 `expression` 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.

29.1.3.2 Configuring Parameters in a Project

Configuring the parameter values for a plugin is done in a Maven 2 project within the `pom.xml` file as part of defining the plugin in the project. An example of configuring a plugin:

```
<plugin>
  <groupId>sample.plugin</groupId>
  <artifactId>maven-hello-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.

29.1.3.3 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.

29.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:

```
    /**
     * My boolean.
     *
     * @parameter
     */
    private boolean myBoolean;
```

```
<myBoolean>true</myBoolean>
```

29.Fixed-Point 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:

```
    /**
     * My Integer.
     *
     * @parameter
     */
    private Integer myInteger;
```

```
<myInteger>10</myInteger>
```

29.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`.

```
    /**
     * My Double.
     *
     * @parameter
     */
    private Double myDouble;
```

```
<myDouble>1.0</myDouble>
```

29.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:

```
    /**
     * My Date.
     *
     * @parameter
     */
    private Date myDate;
```

```
<myDate>2005-10-06 2:22:55.1 PM</myDate>
```

29.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:

```
    /**
     * My File.
     *
     * @parameter
     */
    private File myFile;
```

```
<myFile>c:\temp</myFile>
```

29.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.

```
    /**
     * My URL.
     *
     * @parameter
     */
    private URL myURL;
```

```
<myURL>http://maven.apache.org</myURL>
```

29.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.

29.1.3.4 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

29.Arrays

Array type parameters are configured by specifying the parameter multiple times. Example:

```
    /**
     * My Array.
     *
     * @parameter
     */
    private String[] myArray;
```

```
<myArray>
  <param>value1</param>
  <param>value2</param>
</myArray>
```

29.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:

```
    /**
     * 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.

29.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:

```
    /**
     * My Map.
     *
     * @parameter
     */
    private Map myMap;
```

```
<myMap>
  <key1>value1</key1>
  <key2>value2</key2>
</myMap>
```

29.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:

```
    /**
     * My Properties.
     *
     * @parameter
     */
    private Properties myProperties;
```

```
<myProperties>
  <property>
    <name>propertyName1</name>
    <value>propertyValue1</value>
  <property>
  <property>
    <name>propertyName2</name>
    <value>propertyValue2</value>
  <property>
</myProperties>
```

29.Other Object Classes

This category covers any class which does not implement `java.util.Map`, `java.util.Collection`, or `java.util.Dictionary`. Example:

```
    /**
     * 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.

29.1.4 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:

```
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.

29.1.5 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.

30 Creating a Site

30.1 Creating a site

30.1.1 Creating Content

The first step to creating your site is to create some content. In Maven 2.0, the site content is separated by format, as there are several available.

```
+- src/
   +- site/
      +- apt/
      |  +- index.apt
      |
      +- xdoc/
      |  +- other.xml
      |
      +- fml/
      |  +- general.fml
      |  +- faq.fml
      |
      +- site.xml
```

You will notice there is now a `${basedir}/src/site` directory within which is contained a site descriptor along with various directories corresponding to the supported document types. Let's take a look at site descriptor and the examples of the various document types.

The Xdoc format is the same as used in Maven 1.x. However, `navigation.xml` has been replaced by the site descriptor (see below).

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.

The FML format is the FAQ format, also used in Maven 1.x.

Other formats are available, but at this point these 3 are the best tested. There are also several possible output formats, but as of 2.0, 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.

30.1.2 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 site to have a look at some of the skins that you can use to change the look of your site.

30.1.3 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 its plugin reference.

30.1.4 Deploying the Site

To be able to deploy the site, you must first declare a location to distribute to in your `pom.xml`, similar to the repository for deployment.

```
<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, as above which copies to the host `www.mycompany.com` in the path `/www/docs/project/`. If subprojects inherit the site URL from a parent POM, they will automatically append their `<artifactId>` to form their effective deployment location.

Deploying the site is done by using the `site-deploy` phase of the site lifecycle.

```
mvn site-deploy
```

30.1.5 Creating a Site Descriptor

The `site.xml` file is used to describe the layout of the site, and replaces the `navigation.xml` file used in Maven 1.x.

A sample is given below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Maven">
  <bannerLeft>
    <name>Maven</name>
    <src>http://maven.apache.org/images/apache-maven-project.png</src>
    <href>http://maven.apache.org/</href>
  </bannerLeft>
  <bannerRight>
    <src>http://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="http://maven.apache.org/maven-1.x/"/>
      <item name="Maven 2" href="http://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 site for the Maven Site Plugin.

30.1.6 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
         |
         +- images/
            +- pic1.jpg
```

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.

30.1.7 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
- Continous 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:

```
<project>
  ...
  <reporting>
    <plugins>
```

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.0.1</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.

30.1.8 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:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-site-plugin</artifactId>
        <version>2.0-beta-6</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:

```
+- 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.

31 Snippet Macro

31.1 Guide to the Snippet Macro

When generating your project website with Maven, you have the option of dynamically including *snippet*s 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.

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.

31.1.1 Snippets in Sources

31.1.1.1 Java

```
    // START SNIPPET: snip-id
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
    // END SNIPPET: snip-id
```

31.1.1.2 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 -->
```

31.1.1.3 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 --%>
```

31.1.2 Snippets in Documentation

31.1.2.1 APT

```
%{snippet|id=snip-id|url=http://svn.example.com/path/to/Sample.java}
%{snippet|id=snip-id|url=file:///path/to/Sample.java}
```

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}
```

- Macros in apt **must not** be indented.
- Exactly one of `url` or `file` **must** be specified.

32 What is an Archetype

32.1 Introduction to Archetypes

32.2 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 names 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.

32.2.1 Using an Archetype

To create a new project based on an Archetype, you need to call `mvn archetype:generate` goal, like the following:

```
mvn archetype:generate
```

Please refer to Archetype Plugin Page.

32.2.2 Provided Archetypes

Maven provides several Achetype artifacts:

Archetype ArtifactIds	Description
maven-archetype-archetype	An archetype which contains a sample archetype.
maven-archetype-j2ee-simple	An archetype which contains a simplifed sample J2EE application.
maven-archetype-mojo	An archetype which contains a sample a sample Maven plugin.
maven-archetype-plugin	An archetype which contains a sample Maven plugin.

maven-archetype-plugin-site	An archetype which contains a sample Maven plugin site.
maven-archetype-portlet	An archetype which contains a sample JSR-268 Portlet.
maven-archetype-quickstart	An archetype which contains a sample Maven project.
maven-archetype-simple	An archetype which contains a simple Maven project.
maven-archetype-site	An archetype which contains a sample Maven site which demonstrates some of the supported document types like APT, XDoc, and FML and demonstrates how to i18n your site.
maven-archetype-site-simple	An archetype which contains a sample Maven site.
maven-archetype-webapp	An archetype which contains a sample Maven Webapp project.

32.2.3 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.

33 Creating Archetypes

33.1 Guide to Creating Archetypes

Creating an archetype is a pretty straight forward process. An archetype is a very simple plugin, that contains the project prototype you wish to create. An archetype is made up of:

- an archetype descriptor (`archetype.xml` in directory: `src/main/resources/META-INF/`). 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 (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:

33.1.1 1. Create a new project and pom.xml for the archetype plugin

An example `pom.xml` for an archetype plugin looks as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.groupId</groupId>
  <artifactId>my-archetype-id</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
</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:create` from the commandline.

33.1.2 2. Create the archetype descriptor

The archetype descriptor is a file called `archetype.xml` which must be located in the `src/main/resources/META-INF/` directory. An example of an archetype descriptor can be found in the quickstart archetype:

```
<archetype>
  <id>quickstart</id>
  <sources>
    <source>src/main/java/App.java</source>
  </sources>
  <testSources>
    <source>src/test/java/AppTest.java</source>
  </testSources>
</archetype>
```

The `<id>` tag should be the same as the `artifactId` in the archetype `pom.xml`.

An optional `<allowPartial>true</allowPartial>` tag makes it possible to run the `archetype:create` even on existing projects.

The `<sources>`, `<resources>`, `<testSources>`, `<testResources>` and `<siteResources>` tags represent the different sections of the project:

- <sources> = src/main/java
- <resources> = src/main/resources
- <testSources> = src/test/java
- <testResources> = src/test/resources
- <siteResources> = src/site

<sources> and <testSources> can contain <source> elements that specify a source file.

<testResources> and <siteResources> can contain <resource> elements that specify a resource file.

Place other resources such as the ones in the src/main/webapp directory inside the <resources> tag.

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.xml
            `-- archetype-resources
                |-- pom.xml
                `-- src
                    |-- main
                    |   `-- java
                    |       `-- App.java
                    `-- test
                        `-- java
                            `-- AppTest.java
```

33.1.3 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 the set artifactId and groupId as variables (${artifactId} / ${groupId}). Both variables will be initialized from the commandline when calling archetype:create.

An example for a prototype pom.xml is:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>${groupId}</groupId>
  <artifactId>${artifactId}</artifactId>
  <packaging>jar</packaging>
  <version>${version}</version>
  <name>A custom project</name>
  <url>http://www.myorganization.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

33.1.4 4. Install and run the archetype

Now you are ready to install the archetype:

```
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)

```
mvn archetype:create                                    \
  -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 ibiblio) it as any other artifact and the archetype will then be available to any user of Maven.

33.1.5 Alternative way to start creating your Archetype

Instead of manually creating the directory structure needed for an archetype, simply use

```
mvn archetype:create
  -DgroupId=[your project's group id]
  -DartifactId=[your project's artifact id]
  -DarchetypeArtifactId=maven-archetype-archetype
```

Afterwhich, you can now customize the contents of the `archetype-resources` directory, and `archetype.xml`, then, proceed to Step#4 (Install and run the archetype plugin).

34 From Maven 1.x to Maven 2.x

34.1 Guide to Moving from Maven 1.x to Maven 2.x

This document is intended to be continously updated from the mail list archives. For an only slightly out-of-date reference with concrete examples, check out Vincent Massol's JavaZone2005 presentation.

34.1.1 Parallel Builds

It is possible to establish parallel Maven builds, one using the old M1 settings, and a second using M2. The Maven 2 configuration file names and uses have been modified, so the two builds should not conflict.

A Maven 1.x build is configured with the following files:

- [project.xml] Project Object Model (POM) definition
- [maven.xml] Custom build scripts
- [project.properties] general build settings
- [build.properties] local build settings

  A Maven 2 build is configured with a different file set:
- [pom.xml] POM definition
- [settings.xml] local configuration

34.1.2 Migrating the POM

The Project Object Model (POM) has moved from the `project.xml` file to `pom.xml`. The XML schema has also changed, from Version 3 to Version 4.

The new POM is nominally a superset of the old, so the first step in creating a `pom.xml` is to copy over `project.xml`. Then start tweaking. There are several new elements that can be added to a POM, but all are optional so should not cause a problem with an initial build.

If you want some help converting your `project.xml` into a `pom.xml` you can use the maven-one-plugin. If you run the following command, it will convert your `project.xml` into a `pom.xml`:

```
mvn one:convert
```

project.xml:

```
<project>
  <pomVersion>3</pomVersion>
  <id>util</id>
  <name>Generic utility code</name>
  <groupId>project</groupId>
  <currentVersion>1.1</currentVersion>
  <package>org.apache.project.util</package>
  <dependencies>
    ...
  </dependencies>
  <build>
    ...
  </build>
  ...
</project>
```

pom.xml:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>util</artifactId>
  <name>Generic Utility Code</name>
  <groupId>org.apache.project.util</groupId>
  <version>1.1</version>
  <packaging>jar</packaging>
  <dependencies>
  ...
  </dependencies>
  <build>
  ...
  </build>
  ...
</project>
```

For more details, check out the POM Guide.

34.1.3 build.properties and project.properties

These files have been replaced with settings.xml. Like with the POM, you can establish a parallel build environment, so the m1 build never breaks while the m2 build is being debugged.

Additional local build customization options can also be created using profiles.

34.1.4 What to do with maven.xml?

See How do I write custom scripts without a maven.xml file? for an explanation of why maven.xml was discarded, and Introduction to Maven 2.0 Plugin Development for a guide to writing your own plug-ins.

34.1.5 Directory Structure

The POM allows customization of the directory structure in both Maven 1 and Maven 2 using the `<build>` tag. For simplicity, it would be ideal to move source to the Maven 2 default structure, but

it is not required. You can begin by customizing the directories in Maven 2, then when satisfied that both build paths are working, move to the Maven 2 structure and customize the settings in Maven 1.

34.1.6 Migrating Plug-ins

The main conceptual change in plugins and their use has to do with the concept of a build cycle in Maven 2. Instead of using `preGoal` and `postGoal` tags in `maven.xml` to tie plugin goals into the build process, the goals of a plugin are associated with the pre-defined stages of the build cycle. See the Introduction to the Build Lifecycle for more on how plugins relate.

34.1.6.1 Re-use Ant Tasks

See the Ant Script FAQ.

34.1.6.2 Replace scripts with Mojos

The new plugin architecture does not specify a specific language implementation, so Jelly scripts and other such artifacts should be re-usable with wrappers. It is recommended that you look into moving to Mojos.

34.1.6.3 Utilize built-in Maven 2 capabilities

34.Resource filtering to inject POM variables into application

You can turn on resource filtering in your POM. Tokens of the form `${pom.variable}` in resource files will be replaced with the corresponding POM property.

```
<project>
  ...
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</project>
```

34.Multiproject Builds

The old reactor+multiproject plugin combination was established more as an afterthought of the core development. In Maven 2, multiproject support is included in the core, so any scripts required in the past to work around problems with the multiproject plugin should be unnecessary.

34.1.7 Migrating repositories

Every four hours the Maven 1.x repository is converted over to a Maven 2.x repository and we plan to release a plug-in based on our conversion tool but currently.

34.1.8 Related links

- Maven 2 One Plugin
- XSLT from MNG-2337
- using preGoal and postGoal in m2? Thread.

35 Using Maven 1.x repositories with Maven 2.x

35.1 Guide to using Maven 1.x repositories with Maven 2.x

When you are migrating from Maven 1.x to Maven 2.x you will first be trying to convert your build and to make this easier we have provided a way for you to use your existing Maven 1.x repository so that you don't have to convert your repository before trying to migrate your projects. To use a Maven 1.x repository with your Maven 2.x project you need to specify this in your POM as follows:

```
<project>
  ...
  <repositories>
    <repository>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <id>my-m1-repository</id>
      <name>Maven 1.x Repository</name>
      <url>http://repostory.mycompany.com/maven1</url>
      <layout>legacy</layout>
    </repository>
  </repositories>
  ...
</project>
```

Enabling the snapshots is important as Maven 2.x makes a distinction between repositories that contain snapshots and those that don't. In Maven 1.x there is no distinction, so setting snapshots to true will give you the Maven 1.x style repository behaviour while using Maven 2.x.

36 Relocation of Artifacts

36.1 Guide to relocation

Sometimes it is necessary to relocate artifacts in the repository. One example of that is when a project moves from Maven 1 to Maven 2. Maven 1 projects have traditionally used a flat repository structure, while Maven 2 uses a deep repository structure. As an example the Maven 1 project has a groupId of `maven` while the Maven 2 project has a groupId of `org.apache.maven`.

Making changes to the repository can have far reaching consequences. So it is best to get it right the first time, hence this guide. It will go through a couple of different kinds of relocations:

- Maven 1 to Maven 1
- Maven 2 to Maven 2
- Maven 1 to Maven 2

The goal of the examples below is to relocate the groupId from `bar` to `org.bar` for the `foo` project.

36.1.1 How to relocate a Maven 1 artifact to a different groupId

1. Copy all `foo`-related files from `/bar/` in your Maven 1 repository to a temporary location.
2. Change the groupId to `org.bar` in all the `foo`-related pom files in the temporary location.
3. If your project uses MD5 or SHA1 checksums you must now create new checksums for the changed pom files in the temporary location. If the pom file needs to be signed, do that as well.
4. Copy all files from the temporary location to `/org.bar/` in your Maven 1 repository.
5. If your project syncs with ibiblio, you should now initiate that sync. This might happen automatically depending on your projects sync policy.

Your `foo`-artifacts are now available to Maven 1 users with both the old and the new groupId.

36.1.1.1 Releasing the next version

When the next release of `foo` is made, you publish the Maven 1 pom as you have always done. Unfortunately Maven 1 does not have a concept of automatic relocation and notification, so you will have to inform your users of the changed groupId through your regular information channels.

36.1.2 How to relocate a Maven 2 artifact to a different groupId

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`:

```
<project>
  <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 ibiblio, you should now initiate that sync. This might happen automatically depending on your projects sync policy.

Your `foo`-artifacts are now available to Maven 2 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.

36.1.2.1 Releasing the next version

When the next release of `foo` is made, you should publish two Maven 2 pom files. First you should publish a pom with the new groupId `org.bar`.

Because data in the repository is not supposed to change, Maven 2 doesn't download pom files that it has already downloaded. Therefor you will also need to publish a 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 2 pom with a groupId of `org.bar`, since users of the previous version have been informed of the changed groupId.

36.1.3 How to relocate a Maven 1 artifact to a Maven 2 artifact with a different groupId

This is only of interest to organizations (like the Apache Software Foundation) that automatically converts the contents of their Maven 1 repository to their Maven 2 repository.

Follow steps 4 to 6 in the section *How to relocate a Maven 2 artifact to a different groupId* above.

36.1.3.1 Releasing the next version

When the next release of `foo` is made, you should publish the Maven 1 pom as you have always done. In addition to that, you should publish a Maven 2 pom with a groupId of `bar`, a version of <next-version> and include a relocation section. This step can be done once for the first release of a project, after the groupId has been changed, but your users will be happier if you do it more times.

37 Installing 3rd party JARs to Local Repository

37.1 Guide to installing 3rd party JARs

Often times you will have 3rd party JARs that you need to put in your local repository for use in your builds. The JARs must be placed in the local repository in the correct place in order for it to be correctly picked up by Maven. To make this easier, and less error prone, we have provide a goal in the install plug-in 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>
```

38 Deploying 3rd party JARs to Remote Repository

38.1 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 %M2_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-repositor-to-deploy>
```

38.1.1 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
```

38.1.2 Deploying a 3rd party JAR with a customed 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-repositor-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.

38.1.3 Deploying Source Jars

To deploy a 3rd party source jar, packaging should be set to `java-source`, and generatePom should be set to `false`.

39 Coping with Sun JARs

39.1 Coping with Sun JARs

Often users are confronted with the need to build against JARs provide by Sun like the JavaMail JAR, or the Activation JAR and users have found these JARs not present in central repository resulting in a broken build. Unfortunately most of these artifacts fall under Sun's Binary License which disallows us from distributing them from Ibiblio.

Another problem is that Sun's appears not to have any sort of convention for naming their own JARs so we have taken steps in suggesting some common names for Sun's artifacts. You can find a list of our suggestions here:

Product artifact	Group ID	Artifact ID
Java Activation Framework	javax.activation	activation
J2EE	javax.j2ee	j2ee
Java Data Object (JDO)	javax.jdo	jdo
Java Message Service (JMS)	javax.jms	jms
JavaMail	javax.mail	mail
Java Persistence API (JPA) / EJB 3	javax.persistence	persistence-api
J2EE Connector Architecture	javax.resource	connector
J2EE Connector Architecture API	javax.resource	connector-api
Java Authentication and Authorization Service (JAAS)	javax.security	jaas
Java Authorization Contract for Containers	javax.security	jacc
Servlet API	javax.servlet	servlet-api
Servlet JavaServer Pages (JSP)	javax.servlet	jsp-api
Servlet JavaServer Pages Standard Tag Library (JSTL)	javax.servlet	jstl
JDBC 2.0 Optional Package	javax.sql	jdbc-stdext
Java Transaction API (JTA)	javax.transaction	jta
Java XML RPC	javax.xml	jaxrpc
Portlet	javax.portlet	portlet-api
Java Naming and Directory Interface (JNDI)	javax.naming	jndi

If you use our suggestions as noted above when adding a Sun dependency to your POM, Maven 2.x can help you locate the JARs by providing the site where they can be retrieved. It is important that you follow the suggested naming conventions as we cannot store the JARs at the central repository. We can only store metadata about those JARs and it is the metadata that contains location and retrieval information.

Once you have downloaded a particular Sun JAR to your system you can install the JAR in your local repository. Please refer to our Guide to installing 3rd party JARs for instructions on how to accomplish this.

Note: Java.net provides a Maven 2 repository. You could specify it directly in your POM or in your settings.xml between the tags <repositories>:

```
...
      <repositories>
        <repository>
          <id>maven2-repository.dev.java.net</id>
          <name>Java.net Repository for Maven</name>
          <url>http://download.java.net/maven/2/</url>
          <layout>default</layout>
        </repository>
      </repositories>
...
```

40 Remote repository access through authenticated HTTPS

40.1 Guide to Remote repository access through authenticated HTTPS

This document describes how to configure Maven for accessing a remote repository that sits behind an HTTPS server which requires client authentication with certificates. It is expected that this documentation be valid both for Maven 1.x and Maven 2.0.

40.1.1 The problem

You have a server storing a maven repository at addresse `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 (eg. 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 other mean 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 importation in their keystore. This file is named:

```
/home/directory/mycertificate.p12
```

and we assume it is accessible when launching maven. Note that this file contains the client's private key which may be very sensitive information and so is secured by a password:

```
CeRtPwD
```

The remote repository is referenced either through the `pom.xml` file (maven2.0) or one of `build.properties` or `project.properties` (Maven1.X). In Maven 1.X:

```
maven.repo.remote=https://my.server.com/maven,http://www.ibiblio.org/maven
```

40.1.2 The solution

For maven to use this repository, we should take the following steps:

1. Create a store to hold the server's certificate usings Sun's keytool,
2. Defines properties to be used by HttpClient for finding keys and certificate

40.1.2.1 Storing certificate

The following command line imports the certififcate 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: ....
Valid from: Mon Feb 21 22:34:25 CET 2005 until: Thu Feb 19 22:34:25 CET 2015
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.

40.1.2.2 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.

40.Maven 2.0

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"
```

40.For maven 1.X users

Setting these properties in `build.properties` or `project.properties` does **not work**: the properties are needed before any of theses files are opened.

40.1.3 Links

The following links may be useful in understanding SSL infrastructure management in Java:

- Javasecurity infrastructure (1.4.2)
- HttpClient's SSL guide

41 Creating Assemblies

41.1 Guide to creating assemblies

The assembly mechanism in Maven 2.x provides an easy way to create distributions using a assembly descriptor and dependency information found in you 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:

```
<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>2.2-beta-2</version>
        <configuration>
          <descriptor>src/main/assembly/dep.xml</descriptor>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

You'll notice that the assembly descriptor is located in `${basedir}/src/main/assembly` which is the standard location for assembly descriptors.

41.1.1 Creating a binary assembly

This is the most typical usage of the assembly plugin where you are creating a distribution for standard use.

```
<assembly>
  <id>bin</id>
  <formats>
    <format>tar.gz</format>
    <format>tar.bz2</format>
    <format>zip</format>
  </formats>
  <fileSets>
    <fileSet>
      <includes>
        <include>README*</include>
        <include>LICENSE*</include>
        <include>NOTICE*</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>target</directory>
      <outputDirectory></outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

```
<assembly>
  <!-- TODO: a jarjar format would be better -->
  <id>dep</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <outputDirectory>/</outputDirectory>
    </fileSet>
  </fileSets>
  <dependencySets>
    <dependencySet>
      <outputDirectory>/</outputDirectory>
      <unpack>true</unpack>
      <scope>runtime</scope>
      <excludes>
        <exclude>junit:junit</exclude>
        <exclude>commons-lang:commons-lang</exclude>
        <exclude>commons-logging:commons-logging</exclude>
        <exclude>commons-cli:commons-cli</exclude>
        <exclude>jsch:jsch</exclude>
        <exclude>org.apache.maven.wagon:wagon-ssh</exclude>
        <!-- TODO: can probably be removed now -->
        <exclude>plexus:plexus-container-default</exclude>
      </excludes>
    </dependencySet>
  </dependencySets>
</assembly>
```

```
<assembly>
  <id>src</id>
  <formats>
    <format>tar.gz</format>
    <format>tar.bz2</format>
    <format>zip</format>
  </formats>
  <fileSets>
    <fileSet>
      <includes>
        <include>README*</include>
        <include>LICENSE*</include>
        <include>NOTICE*</include>
        <include>pom.xml</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>src</directory>
    </fileSet>
  </fileSets>
</assembly>
```

```
mvn assembly:assembly
```

42 Configuring Archive Plugins

42.1 Guide to Configuring Archive Plugins

Many Java archive generating plugins accept the `archive` configuration element to customise the generation of the archive. In the standard Maven Plugins, this includes the `jar`, `war`, `ejb`, `ear` and `assembly` plugins.

42.1.1 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):

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1-alpha-1</version>
        <configuration>
          <archive>
            <addMavenDescriptor>false</addMavenDescriptor>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

42.1.2 Configuring the Manifest

The archive configuration also accepts manifest configuration. See Guide to Working with Manifests for more information.

43 Configuring Maven

43.1 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.

43.1.1 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/`.

```
<settings>
  ...
  <localRepository>/path/to/local/repo/</localRepository>
  ...
</settings>
```

Note: The local repository must be an absolute path.

43.1.2 Configuring a Proxy

Proxy configuration can also be specified in the settings file.

For more information, see the Guide to using a Proxy.

43.1.3 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 clean install
```

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
```

43.1.4 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:

```
<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.

43.1.5 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.

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 maven-proxy 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, 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 repository included by default is `central`, so to use an Australian mirror, you would configure the following:

```
<settings>
  ...
  <mirrors>
   <mirror>
      <id>planetmirror</id>
      <name>Australian Mirror of http://repo1.maven.org/maven2/</name>
      <url>http://public.planetmirror.com/maven2/</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
```

```
  ...
  </mirrors>
  ...
</settings>
```

More info about mirrors is available in the Guide to Mirror Settings.

43.1.6 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.

44 Mirror Settings

44.1 Using Mirrors for Repositories

Repositories are 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 maven-proxy 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 repository included by default is `central`, so to use the mirror at *ibiblio*, you would configure the following:

```
<settings>
  ...
  <mirrors>
    <mirror>
      <id>ibiblio.org</id>
      <name>ibiblio Mirror of http://repo1.maven.org/maven2/</name>
      <url>http://mirrors.ibiblio.org/pub/mirrors/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 2 repository is at `http://repo1.maven.org/maven2`. A list of known mirrors is available in our wiki article Mirrors Repository. These mirrors may not have the same contents and we don't support them in any way, although we try to keep info in this page accurate.

44.2 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 the Maven Repository Manager to proxy external requests.

To achieve this, set `mirrorOf` to `*`.

Note: This feature is only available in Maven 2.0.5+.

```
<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>
```

44.3 Advanced Mirror Specification

A single mirror can handle multiple repositories when used in conjunction with a repository manager.

The syntax as of Maven 2.0.9:

- `*` matches all repo ids.
- `external:*` matches all repositories except those using localhost or file based repositories. This is used in conjunction with a repository manager when you want to exclude redirecting repositories that are defined for Integration Testing.
- 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

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, keep in mind that their 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

Note: This feature is only available in Maven 2.0.9+.

```
<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>external:*,!foo</mirrorOf>
    </mirror>
    <mirror>
      <id>foo-repository</id>
      <name>Foo</name>
      <url>http://repo.mycompany.com/foo</url>
      <mirrorOf>foo</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

44.4 FTP Access

The repository is available through FTP at `ftp://mirrors.ibiblio.org/pub/mirrors/maven2`

44.5 Creating Your Own Mirror

The central repository requires several dozens GB and growing. Apparently, to save us bandwidth and you time, mirroring the entire central repository is not recommended. Instead, we suggest to setup a repository manager as a proxy.

If you really want to become an official mirror, email us to `dev@maven.apache.org` with your location and we'll add you to the list of mirrors.

45 Deployment and Security Settings

45.1 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:

```
<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.

46 Embedding Maven 2.x

46.1 The Maven Embedder

The Maven Embedder is used by the Maven CLI, by IDE integration projects like Mevenide and potentially any tool that needs to embed Maven's capabilities. You could embed Maven in a Continuous Integration application to run Maven build, an application lifecycle management (ALF) tool, or Ant tasks that utilize Maven's functionality. These are just a few examples of what the Maven Embedder can be used for.

46.2 A Simple Example

```
        File projectDirectory = new File( getBasedir(), "src/examples/simple-projec
        File user = new File( projectDirectory, "settings.xml" );
        Configuration configuration = new DefaultConfiguration()
            .setUserSettingsFile( user )
            .setClassLoader( Thread.currentThread().getContextClassLoader() );
        ConfigurationValidationResult validationResult = MavenEmbedder.validateConf
        if ( validationResult.isValid() )
        {
            MavenEmbedder embedder = new MavenEmbedder( configuration );
            MavenExecutionRequest request = new DefaultMavenExecutionRequest()
                .setBaseDirectory( projectDirectory )
                .setGoals( Arrays.asList( new String[]{"clean", "install"} ) );
            MavenExecutionResult result = embedder.execute( request );
            if ( result.hasExceptions() )
            {
                fail( ((Exception)result.getExceptions().get( 0 )).getMessage() );
            }
            // ----------------------------------------------------------------------
            // You may want to inspect the project after the execution.
            // ----------------------------------------------------------------------
            MavenProject project = result.getProject();
            // Do something with the project
            String groupId = project.getGroupId();
            String artifactId = project.getArtifactId();
            String version = project.getVersion();
            String name = project.getName();
            String environment = project.getProperties().getProperty( "environment"
            assertEquals( "development", environment );
            System.out.println( "You are working in the '" + environment + "' envir
        }
        else
        {
            if ( ! validationResult.isUserSettingsFilePresent() )
            {
                System.out.println( "The specific user settings file '" + user + "'
            }
            else if ( ! validationResult.isUserSettingsFileParses() )
            {
                System.out.println( "Please check your settings file, it is not wel
            }
        }
```

46.3 A Note on Configuring Settings

Currently there is a notion of a user settings, and a global settings where either can specify information about the following:

- Local Repository
- Proxies
- Mirrors
- Server Configurations

- Plugin Groups

If you are using the embedder it is entirely your responsibility to take user and global settings information and specify it in the embedder configuration. The embedder carries with it no defaults about where these are located and how they are used. If you want your embedded use of Maven to mimic the behavior of the Maven CLI insofar as settings use then use the following code:

```
        Configuration configuration = new DefaultConfiguration()
            .setUserSettingsFile( MavenEmbedder.DEFAULT_USER_SETTINGS_FILE )
            .setGlobalSettingsFile( MavenEmbedder.DEFAULT_GLOBAL_SETTINGS_FILE )
            .setClassLoader( Thread.currentThread().getContextClassLoader() );
        ConfigurationValidationResult validationResult = MavenEmbedder.validateConf
        if ( validationResult.isValid() )
        {
            // If the configuration is valid then do your thang ...
        }
```

Also note that the user and global settings are merged, and the user settings are dominant.

46.4 Accessing the Underlying Plexus Container

Though it is not recommended for general use, it is possible to get at the underlying Plexus Container instance if you wish to lookup custom components. The Maven Embedder was specifically designed to be used for Maven and not a general purpose use of Plexus. So if you use this method then you use it at your peril. You can access the Plexus Container using the following:

```
        Configuration configuration = new DefaultConfiguration()
            .setUserSettingsFile( MavenEmbedder.DEFAULT_USER_SETTINGS_FILE )
            .setGlobalSettingsFile( MavenEmbedder.DEFAULT_GLOBAL_SETTINGS_FILE )
            .setClassLoader( Thread.currentThread().getContextClassLoader() );
        ConfigurationValidationResult validationResult = MavenEmbedder.validateConf
        if ( validationResult.isValid() )
        {
            // If the configuration is valid then do your thang ...
        }
        MavenEmbedder embedder = new MavenEmbedder( configuration );
        PlexusContainer container = embedder.getPlexusContainer();
        // Do what you like with the container ...
```

47 Generating Sources

47.1 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 Antlr plugin:

```
/**
 * Generates files based on grammar files with Antlr tool.
 *
 * @goal generate
 * @phase generate-sources
 * @requiresDependencyResolution compile
 * @author <a href="mailto:vincent.siveton@gmail.com">Vincent Siveton</a>
 * @version $Id$
 */
public class AntlrPlugin
    extends AbstractAntlrMojo
{
    /**
     * @see org.apache.maven.plugin.Mojo#execute()
     */
    public void execute()
        throws MojoExecutionException
    {
        executeAntlr();
    }
```

The first two lines say "I want to be fit into the generate-sources phase and my 'handle' is generate".

So this is all fine and dandy, we have a plugin that wants to generate some sources from a Antlr grammar but how do we use it. You need to specify that you want to use it in your POM:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antlr-plugin</artifactId>
        <version>2.0-beta-1</version>
        <configuration>
          <grammars>java.g</grammars>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>generate</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 Antlr plugin is executed with your given configuration.

48 Working with Manifests

48.1 Guide to Working with Manifests

In order to modify the manifest of the resultant JAR produced by the jar plug-in you need to create a configuration for the jar plug-in. In this first example we'll add some entries to the manifest by specifying what we'd like in the `configuration` element of the jar plug-in:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <archive>
            <manifestEntries>
              <mode>development</mode>
              <url>${pom.url}</url>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

As you see above you can use literal values or you can have values from the POM interpolated into literals or simply use straight POM expressions. So this is what your resultant `MANIFEST.MF` will look like inside the generated JAR:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: jvanzyl
Build-Jdk: 1.4.2_09
Extension-Name: my-app
Specification-Vendor: MyCompany Inc
Implementation-Vendor: MyCompany Inc
Implementation-Title: my-app
Implementation-Version: 1.0-SNAPSHOT
mode: development
url: http://maven.apache.org
```

If you need to do more then simply add some manifest entries there are more options like activating indexing of the JAR, setting the main-class, packageName ... Here's an example of what the `configuration` element of the JAR plug-in might look like:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <archive>
            <!--
            <index>true</true>
            -->
            <manifest>
              <mainClass>com.mycompany.app.App</mainClass>
              <packageName>com.mycompany.app</packageName>
              <!-- options
              <addClasspath>true</addClasspath>
              <addExtensions/>
              <classpathPrefix/>
              -->
            </manifest>
            <manifestEntries>
              <mode>development</mode>
              <url>${pom.url}</url>
            </manifestEntries>
            <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

So this is what your resultant `MANIFEST.MF` will look like inside the generated JAR:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: jvanzyl
Package: org.com.foo.app
Build-Jdk: 1.4.2_09
Extension-Name: my-app
Specification-Vendor: MyCompany Inc
Implementation-Vendor: MyCompany Inc
Implementation-Title: my-app
Implementation-Version: 1.0-SNAPSHOT
Main-Class: org.com.foo.App
mode: development
url: http://maven.apache.org
```

49 Maven Classloading

49.1 Guide to Maven Classloading

This is a description of the classloader hierarchy in Maven 2.0.6+.

49.1.1 Overview

- System Classloader
- Core Classloader
- Plugin Classloaders
- Custom Classloaders

49.1.2 1. System Classloader

Maven uses the Classworlds classloading framework with which we create our classloader graph. If you look in your `${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:

```
<target name="info">
  <echo>java.class.path=${java.class.path}</echo>
</target>
```

49.1.3 2. Core Classloader

The second classloader down the graph contains the core requirements of Maven. More precisely, the core classloader has the libraries in `${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 `${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).

49.1.4 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:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.3</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.ant</groupId>
```

```
            <artifactId>ant-nodeps</artifactId>
            <version>1.7.1</version>
          </dependency>
        </dependencies>
        ...
      </plugin>
```

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.

49.1.5 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.

50 Using Multiple Modules in a Build

50.1 Guide to Working with Multiple Modules

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.

50.1.1 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

50.1.1.1 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 modules 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 will not cause a change to the reactor sort order

50.1.1.2 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:

- `-r` - ignore the modules declared in the current project, and instead build the list of projects listed after the `-r` switch (which may include wildcards)
- `--resume-from` - resumes a reactor 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.

50.1.1.3 The Reactor Plugin

For versions of Maven prior to Maven 2.1, or for additional capabilities with the reactor such as building only the modules with SCM changes, the Reactor plugin can be used to further customize the execution of the projects. For information on how to use this, refer to the Reactor Plugin documentation.

50.1.2 More information

- Chapter 6. A Multi-module Project (Maven: The Definitive Guide)

51 Using Multiple Repositories

51.1 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:

```
<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>
```

The `repositories` element is inherited so you would usually specify the repositories to use for a group of projects by defining a `repositories` element at the top of your inheritance chain.

NOTE: You will also get the standard set of repositories as defined in the Super POM.

The other way you can specify the use of multiple repositories by creating a profile in your `~/.m2/settings.xml` file like the following:

```
<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 like 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.

52 Using Proxies

52.1 Configuring a proxy

You can configure a proxy to use for some or all of your HTTP requests in Maven 2.0. 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.

```
<settings>
  .
  .
  <proxies>
   <proxy>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>www.google.com|*.somewhere.com</nonProxyHosts>
    </proxy>
  </proxies>
  .
  .
</settings>
```

Please note that urrently 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.

52.1.1 Resources

1 Settings descriptor documentation
2 Configuring Maven

53 Using the Release Plugin

53.1 Releasing

53.1.1 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.

53.1.2 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>
  ...
  <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>2.0-beta-7</version>
        <configuration>
          ...
          <tagBase>
            file:///D:/subversion_data/repos/my_repo/my-app-example/tags
          </tagBase>
          ...
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

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.

```
<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>2.0-beta-7</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 relese 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.

53.1.3 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.

```
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.

```
#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
scm.tag=1.0
scm.url=scm\:svn\:file\:///D\:/subversion_data/repos/my_repo/my-app-
example/trunk
scm.tag-base=file\:///D\:/subversion_data/repos/my_repo/my-app-example/tags
checkpoint.transform-pom-for-development=OK
checkpoint.local-modifications-checked=OK
checkpoint.initialized=OK
checkpoint.checked-in-release-version=OK
checkpoint.tagged-release=OK
checkpoint.prepared-release=OK
checkpoint.check-in-development-version=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 excuting 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. An 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.

```
<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>myCompanyReporsitory</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 documentationa and additional mini guides for the use of the maven-deploy plugin.

The following delvierables 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.

53.1.4 Troubleshooting

53.1.4.1 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.

53.1.4.2 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.

54 Using Ant with Maven

54.1 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.

```
<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>1.1</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:

```
<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>1.1</version>
        <executions>
          <execution>
            <phase>generate-sources</phase>
            <configuration>
              <tasks>
                <exec
                  dir="${basedir}"
                  executable="${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>
```

55 Using Modello

55.1 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:

```
<model xmlns="http://modello.codehaus.org/MODELLO/1.0.0" xmlns:xsi="http://www.w3.o
  xsi:schemaLocation="http://modello.codehaus.org/MODELLO/1.0.0 http://modello.code
  <id>archetype</id>
  <name>Archetype</name>
  <description><![CDATA[Maven's model for the archetype descriptor.]]></description
  <defaults>
    <default>
      <key>package</key>
      <value>org.apache.maven.archetype.model</value>
    </default>
  </defaults>
  <classes>
    <class rootElement="true" xml.tagName="archetype">
      <name>ArchetypeModel</name>
      <description>Describes the assembly layout and packaging.</description>
      <version>1.0.0</version>
      <fields>
        <field>
          <name>id</name>
          <version>1.0.0</version>
          <required>true</required>
          <type>String</type>
        </field>
        <field>
          <name>allowPartial</name>
          <version>1.0.0</version>
          <required>true</required>
          <type>boolean</type>
        </field>
        <field>
          <name>sources</name>
          <version>1.0.0</version>
          <association>
            <type>String</type>
            <multiplicity>*</multiplicity>
          </association>
        </field>
        <field>
          <name>resources</name>
          <version>1.0.0</version>
          <association>
            <type>String</type>
            <multiplicity>*</multiplicity>
          </association>
        </field>
        <field>
          <name>testSources</name>
          <version>1.0.0</version>
          <association>
            <type>String</type>
            <multiplicity>*</multiplicity>
          </association>
        </field>
        <field>
          <name>testResources</name>
          <version>1.0.0</version>
          <association>
            <type>String</type>
            <multiplicity>*</multiplicity>
          </association>
        </field>
        <field>
```

To utilize Modello you would configure the `maven-modello-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:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.modello</groupId>
        <artifactId>modello-maven-plugin</artifactId>
        <version>1.0</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>
        <configuration>
          <models>
            <model>src/main/mdo/descriptor.mdo</model>
          </models>
          <version>1.0.0</version>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

56 Webapps

```
mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-webapp -Darchetype
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-webapp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</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 clean package
```

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

Now you can modify this webapp project and turn it into anything you need!

57 Using Extensions

57.1 Using Extensions

Extensions are used to enable Wagon providers, used for the transport of artifact between repositories, and plug-ins which provide lifecycle enhancements.

57.1.1 Wagon providers

```
<project>
  ...
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ftp</artifactId>
         <version>1.0-beta-2</version>
      </extension>
    </extensions>
  </build>
  ...
</project>
```

Note: Wagon 1.0-beta-3+ requires Maven 2.1.0 or above. For Maven 2.0.10 and earlier, use Wagon 1.0-beta-2.

Note: Some Wagons require JDK 5.0 to operate correctly.

57.1.2 Plug-ins which provide lifecycle enhancements

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.plexus</groupId>
        <artifactId>plexus-maven-plugin</artifactId>
        <version>1.1-alpha-8-SNAPSHOT</version>
        <extensions>true</extensions>
        <configuration>
          ...
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

58 Building For Different Environments with Maven 2

58.1 Building For Different Environments with Maven 2

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 2 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. Also available for download is a fully-working example project.

```
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, see the accompanying source code for the full pom.xml.

```
 <profiles>
   <profile>
     <id>test</id>
     <build>
       <plugins>
         <plugin>
           <artifactId>maven-antrun-plugin</artifactId>
           <executions>
             <execution>
               <phase>test</phase>
               <goals>
                 <goal>run</goal>
               </goals>
               <configuration>
                 <tasks>
                   <delete file="${project.build.outputDirectory}/
environment.properties"/>
```

```
                  <copy file="src/main/resources/
environment.test.properties"
                        tofile="${project.build.outputDirectory}/
environment.properties"/>
                </tasks>
              </configuration>
            </execution>
          </executions>
        </plugin>
        <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>
      </plugins>
    </build>
  </profile>
  .. Other profiles go here ..
</profiles>
```

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 identical.

58.2 Caveats

- Currently Maven 2 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 beeing packaged and installed. The JAR plugin probably should also get improved support for this use 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 actualy 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.

58.3 Resources

1 Introduction to Build Profiles
2 Standard Directory Layout
3 The accompanying source code

59 Using Toolchains

59.1 Guide to Using Toolchains

59.1.1 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. With toolchains, 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)

Toolchains would only work in Maven 2.0.9 and higher versions. For more details about it's design and implementation, please see Toolchains.

Below are the plugins which are toolchain-aware, meaning they can be used with toolchains. Please note that these are still SNAPSHOT versions and are not yet released.

1. `maven-compiler-plugin-2.1-SNAPSHOT`
2. `maven-javadoc-plugin-2.5-SNAPSHOT`
3. `maven-surefire-plugin-2.5-SNAPSHOT`
4. `exec-maven-plugin-1.1.1-SNAPSHOT` (Codehaus MOJO)

59.1.2 Using Toolchains in Your Project

There are two essential components that you need to configure in order to use toolchains. These are the `maven-toolchains-plugin` and the `toolchains.xml` file.

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. You can configure the version you want to use via this plugin as shown in the `pom.xml` below.

```
<plugins>
 ...
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.1-SNAPSHOT</version>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-toolchains-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <executions>
      <execution>
        <phase>validate</phase>
        <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, the `maven-toolchain` component used by the plugin would look for the `toolchains.xml` file, read it and look for the matching toolchain 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 sets 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 would see that a JDK toolchain is set in the MavenSession and would thereby use that toolchain (that would be the JDK installed at `/path/to/jdk/1.5` for our example) to compile the sources.

```
<?xml version="1.0" encoding="UTF8"?>
<toolchains>
  <toolchain>
     <type>jdk</type>
     <provides>
         <version>1.5</version>
         <vendor>sun</vendor>
         <id>default</id>
     </provides>
     <configuration>
        <jdkHome>/path/to/jdk/1.5</jdkHome>
     </configuration>
  </toolchain>
  <toolchain>
     <type>jdk</type>
     <provides>
         <version>1.6</version>
         <vendor>sun</vendor>
         <id>ide</id>
     </provides>
     <configuration>
        <jdkHome>/path/to/jdk/1.6</jdkHome>
     </configuration>
  </toolchain>
  <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.

60 Encrypting passwords in settings.xml

60.1 Password Encryption

60.1.1 Introduction

Maven 2.1.0+ now 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 `~/.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

60.1.2 How to create a master password

Use the following command line:

```
mvn --encrypt-master-password <password>
```

This command will produce an encrypted version of the password, something like

```
{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}
```

Store this password in the `~/.m2/settings-security.xml`; it should look like

```
<settingsSecurity>
  <master>{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>
</settingsSecurity>
```

When this is done, you can start encrypting existing server passwords.

60.1.3 How to encrypt server passwords

You will have to use the following command line:

```
mvn --encrypt-password <password>
```

This command will produce an encrypted version of it, something like

```
{COQLCE6DU6GtcS5P=}
```

Cut-n-paste it into your `settings.xml` file in the server section. 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 {COQL
    </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 \
```

60.1.4 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

```
<settingsSecurity>
  <master>{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>
</settingsSecurity>
```

in the file `/Volumes/mySecureUsb/secure/settings-security.xml`

And then I create `~/.m2/settings-security.xml` with the following content:

```
<settingsSecurity>
  <relocation>/Volumes/mySecureUsb/secure/settings-security.xml</relocation>
</settingsSecurity>
```

This assures that encryption will only work when the usb drive is mounted by OS. This addresses a use case where only certain people are authorized to deploy and are issued these devices.

60.1.5 Tips

60.1.5.1 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) may actually contain '{' 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 will treat all the characters preceding the '{' literal, and all the characters after the '}' literal, as comments. Obviously, this is not the behavior you want in such a situation. What you really need is a way of **escaping** the curly-brace literals in your password.

Starting in Maven 2.2.0, you can do just 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 would look like this:

```
{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+\{EF1iFQyJQ=}
```

61 Reusable Test JARs

61.1 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:

```
<project>
  <build>
    <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-jar-plugin</artifactId>
       <version>2.2</version>
       <executions>
         <execution>
           <goals>
             <goal>test-jar</goal>
           </goals>
         </execution>
       </executions>
     </plugin>
    </plugins>
  </build>
</project>
```

61.1.1 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
```

61.1.2 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
```

61.1.3 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`:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>com.myco.app</groupId>
      <artifactId>foo</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>test-jar</type>
      <scope>test</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

Note that previous editions of this guide suggested to use `<classifier>tests</classifier>` instead of `<type>test-jar</type>`. While this currently works for some cases, it does not properly work during a reactor build of the test JAR module and any consumer if a lifecycle phase prior to `install` is invoked. In such a scenario, Maven will not resolve the test JAR from the output of the reactor build but from the local/remote repository. Apparently, the JAR from the repositories could be outdated or completely missing, causing a build failure (cf. MNG-2045).

62 Eclipse Integration

62.1 Eclipse plugins for Maven

The following plugins allow using Maven from the Eclipse IDE, avoiding the use of the Maven command line interface. They integrate Maven in the IDE in different ways, please check their sites for more information. Both plugins allow running Maven goals from Eclipse, see the output in a view inside the IDE and synchronize Maven POM information with Eclipse project information to some extent.

62.2 The Maven Integration for Eclipse (m2eclipse, Eclipse m2e)

The Maven Integration for Eclipse is the first and most mature of the projects aimed at integrating Maven within the Eclipse IDE. It is released under the EPL 1.0 license.

Features include:

- Launching Maven builds from within Eclipse
- Dependency management for Eclipse build path based on Maven's pom.xml
- Resolving Maven dependencies from the Eclipse workspace without installing to local Maven repository
- Automatic downloading of the required dependencies and sources from the remote Maven repositories
- Wizards for creating new Maven projects, pom.xml and to enable Maven support on existing projects
- Quick search for dependencies in remote Maven repositories
- Quick fixes in the Java editor for looking up required dependencies/jars by the class or package name
- Integration with other Eclipse tools, such as WTP, AJDT, Mylyn, Subclipse and others.

For installation instructions, see the m2eclipse website. For the most recent list of features, see the New and Noteworthy page on the m2eclipse wiki.

Currently, this project is being incubated at Eclipse. Subscribe to the Eclipse eclipse.technology.m2e newsgroup or use the web interface to stay up-to-date with the latest progress.

62.3 Eclipse Integration for Apache Maven (Eclipse IAM), formerly Q for Eclipse

Eclipse IAM is a newer Apache Maven plugin for Eclipse with a fresh approach for Maven integration with the Eclipse IDE and other Eclipse plugins (JDT, WTP, Candy for Appfuse,...), also opening the doors for other Eclipse plugin developers to access Maven features as easy as possible.

With five releases already, and continuing to make one every other month it's quickly maturing.

Check the FAQ and Installation instructions

You can join the newsgroup at

- server: news.eclipse.org
- group: eclipse.technology.iam
- or at the web interface

63 Netbeans Integration

63.1 Maven 2.x Module for Netbeans

The Netbeans integration was for a long time developed at http://mevenide.codehaus.org, it was moved to netbeans.org and will be part of standard NetBeans distribution since 7.0. It allows to load any Maven 2 project into Netbeans and start coding immediately. To get a current feature list along with screenshots, description and hints please refer to the NetBeans.org wiki page.

63.1.1 Binaries and Installation

The Maven integration is easily accessible in NetBeans 6.0, 6.1 and 6.5 via the Tools/Plugins dialog. In 7.0 and later it's part of the standard installation. More detailed instructions on installation available (for older versions of NetBeans) at the Mevenide site.

63.1.2 Bugs reports and enhancement requests

Bug, enhancements and feature requests are to be filed in the NetBeans.org issue tracking system.

64 Plugin Developer Centre

64.1 Plugin Developers Centre

This documentation centre is for those that 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 **M**aven plain **O**ld **J**ava **O**bject. Each mojo is an executable *goal* in Maven, and a plugin is a distribution of one or more related mojos.

- Your First Mojo - Learn how to write your first plugin
- Testing your Plugin - How to write tests for your plugins
- Documenting your Plugin - How to write documentation for your plugins
- Plugins Cookbook - Examples for how to perform common tasks in plugins
- Common Bugs and Pitfalls - Overview of problematic coding patterns

64.1.1 Reference

- Mojo API and Annotation Reference
- Maven API Reference
- Maven Class Loading

64.1.2 Examples

- Injecting POM Properties via settings.xml

65 Testing your Plugin

65.1 Introduction

Currently, Maven only supports unit testing out of the box. This document is intended to help Maven Developers to test Plugins with Unit Tests, Integration Tests or Functional tests.

Note: There are a lot of different ways to test a Maven plugin. For a review of different strategies and tools, please refer to Review of Plugin Testing Strategies

65.2 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.

65.3 Unit Tests

65.3.1 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.

65.3.2 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 you all you need is 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.

65.3.3 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 dependency, and create a *MojoTest (by convention) class which `extends AbstractMojoTestCase`.

```
...
  <dependencies>
    ...
    <dependency>
      <groupId>org.apache.maven.shared</groupId>
      <artifactId>maven-plugin-testing-harness</artifactId>
      <version>1.0-beta-1</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, please refer to Maven Plugin Harness Wiki

65.4 Integration/Functional testing

65.4.1 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-d
        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",
        verifier.deleteArtifact( "org.apache.maven.its.itsample", "checkstyle-test"
        verifier.deleteArtifact( "org.apache.maven.its.itsample", "checkstyle-assem
        /*
         * 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:
         * http://maven.apache.org/shared/maven-verifier/apidocs/index.html
         */
        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:
         * http://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.

65.4.2 maven-invoker-plugin

You can use maven-invoker-plugin to invoke Maven and to provide some BeanShell tests. Tests written in this way don't run under JUnit/TestNG; instead, they're run by Maven itself.

```
<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">
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-invoker-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
        <configuration>
          <debug>true</debug>
          <projectsDirectory>src/it</projectsDirectory>
          <pomIncludes>
            <pomInclude>**/pom.xml</pomInclude>
          </pomIncludes>
          <postBuildHookScript>verify.bsh</postBuildHookScript>
        </configuration>
        <executions>
          <execution>
            <phase>integration-test</phase>
            <goals>
              <goal>run</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

65.4.3 shitty-maven-plugin

The shitty-maven-plugin (Super Helpful Integration Testing ThingY) provides a simple way to run integration tests for a project (single module or multi-module).

shitty-maven-plugin does many of the same things as the maven-invoker-plugin (it supports Groovy tests instead of BeanShell tests), though it has some features that aren't available in maven-invoker-plugin. Notably, it provides some advanced setup steps to automatically install your plugin using a special version name ("TESTING"), so your dummy projects can depend on that version explicitly.

65.4.4 maven-it-plugin

Note: maven-it-plugin is not at 1.0 yet. Use it at your own risk.

The maven-it-plugin is used directly in the `integration-test` phase.

Note: this it plugin can not be used to test a plugin that is being built for the first time (i.e. with no release). In this case, you could, for instance, defined an it-snapshot of the plugin with `maven-install-plugin`. during the `pre-integration-test` phase.

```
<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">
  ...
  <artifactId>maven-XXX-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <description>Test Report plugin in the site phase</description>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-it-plugin</artifactId>
        <version>1.0-alpha-1-SNAPSHOT</version>
        <configuration>
          <integrationTestsDirectory>${basedir}/src/it</integrationTestsDirectory>
          <includes>
            <include>**/pom.xml</include>
          </includes>
          <goals>site</goals>
        </configuration>
        <executions>
          <execution>
            <phase>integration-test</phase>
            <goals>
              <goal>test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <!-- Need to install IT snapshot of maven-XXX-plugin -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-install-plugin</artifactId>
        <version>2.2-SNAPSHOT</version>
        <executions>
          <execution>
            <id>it-test</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>install-file</goal>
            </goals>
            <configuration>
              <file>${basedir}/target/maven-XXX-plugin-1.0-SNAPSHOT.jar</file>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-XXX-plugin</artifactId>
              <version>1.0-it-SNAPSHOT</version> <!-- IT SNAPSHOT -->
              <packaging>maven-plugin</packaging>
              <pomFile>${basedir}/pom.xml</pomFile>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <!-- Testing the result of the it pom.xml -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.4.2</version>
```

The it pom should use the it snapshot:

```
<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">
...
  <reporting>
    <outputDirectory>
      ${basedir}/../../../../../target/it/it1/target/site
    </outputDirectory>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-XXX-plugin</artifactId>
        <version>1.0-it-SNAPSHOT</version>
      </plugin>
    </plugins>
  </reporting>
...
</project>
```

65.4.5 maven-plugin-management-plugin

The maven-plugin-management-plugin is to stage/unstage a plugin into the local repository for pre/post-integration-test.

You need to configure the `maven-plugin-test-plugin` and the `maven-surefire-plugin`:

```
<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">
  ...
  <artifactId>maven-XXX-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <description>Test Report plugin in the site phase</description>
  <prerequisites>
    <maven>2.0.4</maven>
  </prerequisites>
  ...
  <dependencies>
    ...
    <!-- Due to the Maven 2.0.4 prerequisites.
    By default, maven-plugin-test-plugin uses 2.0.1 -->
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-embedder</artifactId>
      <version>2.0.4</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-core</artifactId>
      <version>2.0.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-plugin-management-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
        <executions>
          <execution>
            <id>pre-it-test</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>stage</goal>
            </goals>
          </execution>
          <execution>
            <id>post-it-test</id>
            <phase>post-integration-test</phase>
            <goals>
              <goal>unstage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <!-- Testing the result of the it pom.xml -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.4.2</version>
        <executions>
          <execution>
```

The *TestIt classes could use the Maven Embedder to provide tests:

```
public class MyMojoTestIt
    extends PlexusTestCase
{
    /**
     * @throws Exception
     */
    public void testDefaultProject()
        throws Exception
    {
        MavenEmbedder maven = new MavenEmbedder();
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        maven.setClassLoader( classLoader );
        maven.setLogger( new MavenEmbedderConsoleLogger() );
        maven.setOffline( true );
        maven.setLocalRepositoryDirectory( getTestFile( "target/local-repo" ) );
        maven.start();
        File itbasedir = new File( getBasedir(), "src/it/it1" );
        MavenProject pom =
          maven.readProjectWithDependencies( new File( itbasedir, "pom.xml" ) );
        EventMonitor eventMonitor =
          new DefaultEventMonitor(
            new PlexusLoggerAdapter(
              new MavenEmbedderConsoleLogger() ) );
        maven.execute( pom,
          Collections.singletonList(
            "org.apache.maven.plugins:maven-XXX-plugin:1.0-SNAPSHOT:yourGoal" ),
          eventMonitor,
          new ConsoleDownloadMonitor(),
          null,
          itbasedir );
        maven.stop();
    }
}
```

Note: The maven-plugin-management-plugin is similar to `maven-plugin-test-plugin`.

66 Documenting your Plugin

66.1 Introduction

A Guide to the Plugin Documentation Standard was created. This document is intended to verify it during the Plugins development.

66.2 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
```

66.3 References

- Maven Plugin Documentation

67 Common Bugs and Pitfalls

67.1 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
- Depending on Plexus Utilities 1.1+

67.1.1 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:

```
/*
 * 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 whereever possible:

- Source File Encoding
- Report Output Encoding

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:

```
/*
 * FIXME: This assumes the XML encoding declaration matches the platform's default
 */
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.

67.1.2 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:

```
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:

```
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 is 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 targetting JRE 1.4+ could easily avoid these problems by using the constructor `File(URI)` as suggested by the following snippet:

```
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.

67.1.3 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":

```
/*
 * 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 comparisions 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.

67.1.4 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:

```
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.

67.1.5 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:

```
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()`.

67.1.6 Using Shutdown Hooks

People occasionally employ shutdown hooks to perform cleanup tasks, e.g. to delete temporary files as shown in the example below:

```
public MyMojo extends AbstractMojo
{
    public void execute()
    {
        File tempFile = File.createTempFile( "temp", null );
        /*
         * FIXME: This assumes the JVM terminates soon after
         * the Maven build has finished.
         */
        tempFile.deleteOnExit();
    }
}
```

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.

67.1.7 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:

```
public MyMojo extends AbstractMojo
{
    /**
     * @parameter
     */
    private String outputDirectory;
    public void execute()
    {
        /*
         * FIXME: This will resolve relative paths like "target/classes" against
         * the user's working directory instead of the project's base directory.
         */
        File outputDir = new File( outputDirectory ).getAbsoluteFile();
    }
}
```

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:

```
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.

67.1.8 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:

```
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.
         */
        outputDirectory.mkdirs();
    }
}
```

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.

67.1.9 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:

```
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.

67.1.10 Depending on Plexus Utilities 1.1+

Up to Maven 2.0.5, version 1.1 of the artifact `plexus-utils` was included in the Maven core class loader which is shared with the plugin class realm. This effectively prevented plugins from using another/newer version of `plexus-utils`. This has been solved starting with Maven 2.0.6 by shading (most of) the classes from `plexus-utils` (see MNG-2892).

In short, plugins that strictly require a newer version of `plexus-utils` also require Maven 2.0.6 as a minimum. Hence, a POM snippet for a Maven plugin like shown below is misleading:

```
<project>
  <packaging>maven-plugin</packaging>
  ...
  <prerequisites>
    <!-- FIXME: This assumes the plugin works with plexus-utils:1.1, too -->
    <maven>2.0</maven>
  </prerequisites>
  ...
  <dependencies>
    <dependency>
      <groupId>org.codehaus.plexus</groupId>
      <artifactId>plexus-utils</artifactId>
      <version>1.5.6</version>
    </dependency>
  </dependencies>
  ...
</project>
```

68 Mojo API

68.1 Introduction

Starting with 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.monitor.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 and 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.

68.2 API Documentation

68.2.1 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.

68.Method Summary:

- `void setLog( org.apache.maven.monitor.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`.

68.2.2 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.

68.Method Summary:

- `public void setLog( org.apache.maven.monitor.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.

68.2.3 org.apache.maven.monitor.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()` .

68.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 info( 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.
- `void warn( java.lang.CharSequence )`

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.

68.3 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 Javadoc annotation (if applicable) supporting that piece of the plugin descriptor. A couple of examples are: **someElement (@annotation parameterName="parameterValue")** or **someOtherElement (@annotation <rawAnnotationValue>)** .

The plugin descriptor must be provided in a jar resource with the path: `META-INF/maven/plugin.xml` , and it must contain the following:

Descriptor Element	Required?	Notes
mojos	Yes	Descriptors for each Mojo provided by the plugin, each inside a **mojo** sub-element. Mojo 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.

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):

Descriptor Element	Annotation	Required?	Notes
aggregator	@aggregator	No	Flags this Mojo to run it in a multi module way, i.e. aggregate the build with the set of projects listed as modules.
configurator	@configurator <roleHint>	No	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.)*

execute	• @execute phase="<phaseName> lifecycle="<lifecycleId>" • @execute phase="<phaseName> • @execute goal="<goalName>"	No	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 `${executedProject}` expression if required. An alternate lifecycle can also be provided: for more information see the documentation on the build lifecycle.
executionStrategy	@executionStrategy <strategy>	No	Specify the execution strategy. *NOTE: Currently supports **once-per-session**, **always**.*
goal	@goal <goalName>	Yes	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.
inheritByDefault	@inheritByDefault <true|false>	No. Default: `true`	Specify that the Mojo is inherited.
instantiationStrategy	@instantiationStrategy <per-lookup>	No. Default: `per-lookup`	Specify the instantiation strategy.
phase	@phase <phaseName>	No	Binds this Mojo to a particular phase of the standard build lifecycle, if specified. *NOTE: This is only required if this Mojo is to participate in the standard build process.*
requiresDependencyResolu	@requiresDependencyResc <requiredScope>	No	Flags this Mojo as requiring the dependencies in the specified scope (or an implied scope) to be resolved before it can execute. Currently supports `compile`, `runtime`, and `test` scopes. If this annotation is present but no scope is specified, the scope defaults to `runtime`.
requiresDirectInvocation	@requiresDirectInvocation <true|false>	No. Default: `false`	Flags this Mojo to be invoke directly.

requiresOnline	@requiresOnline <true|false>	No. Default: `false`	Flags this Mojo to require online mode for its operation.
requiresProject	@requiresProject <true|false>	No. Default: `true`	Flags this Mojo to run inside of a project.
requiresReports	@requiresReports <true|false>	No. Default: `false`	Flags this Mojo to require reports.
description	none (detected)	No	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.*
implementation	none (detected)	Yes	The Mojo's fully-qualified class name (or script path in the case of non-Java Mojos).
language	none (detected)	No. Default: `java`	The implementation language for this Mojo (Java, beanshell, etc.).
deprecated	@deprecated <deprecated-text>	No	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.
since	@since <since-text>	No	Specify the version when the Mojo was added to the API. Similar to Javadoc since.

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:

Descriptor Element	Annotation	Required?	Notes
alias	@parameter alias="myAlias"	No	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.
configuration	@component role="..." roleHint="..."	No	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 "default". You can customise either of these by providing a `role` and/or `roleHint` parameter. *e.g.* `@component role="org.apache.maven.artifac roleHint="ear"`. **Note:** This is identical to the deprecated form of parameter: `@parameter expression="${component.yourpa`

configuration	@parameter expression="${aSystemPro default-value="${anExpression}"	No	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}-${project.version}-special`. The system property given by `expression` enables users to override the default value from the command line via `-DaSystemProperty=value`. *NOTE: If neither `default-value` nor `expression` are specified, the parameter can only be configured from the POM. The use of '${' and '}' is required to delimit actual expressions which may be evaluated.*

editable	@readonly	No	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 <build><finalName/ ></build> 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. *NOTE: Specification of this annotation flags the parameter as non-editable; there is no true/ false value.*
required	@required	No	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. *NOTE: Specification of this annotation flags the parameter as required; there is no true/false value.*
description	none (detected)	No	The description of this parameter's use inside the Mojo. *Using the toolset, this is detected as the Javadoc description for the field. NOTE: 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.*

name	none (detected)	Yes	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.*
type	none (detected)	Yes	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.*
deprecated	@deprecated <deprecated-text>	No	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.
since	@since <since-text>	No	Specify the version when the Mojo was added to the API. Similar to Javadoc since.

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).

68.4 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.

68.5 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.

POM Element	Required for Java Mojos?	Sample Declaration	Notes
packaging	Yes	`<packaging> maven-plugin </ packaging>`	The POM must declare a packaging element which describes this project as a Maven plugin project.
scriptSourceDirectory	No	`<scriptSourceDirec src/main/ scripts </ scriptSourceDirecto`	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.)

68.6 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 $USER_HOME/.IntelliJIdea/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

4 aggregator, execute, goal, phase, requiresDirectInvocation, requiresProject, requiresReports, requiresOnline, parameter, component, required, readonly

68.7 Resources

This section simply gives a listing of pointers for more information.

- QDox Project (Javadoc annotations) [link]
- Plexus Project (Plexus container) [link]
- Maven Plugin Descriptor API [link]
- MojoDescriptor API [link]

69 Maven Repository Centre

69.1 Maven Repository Centre

This documentation centre is for those that need to use or contribute to the Maven repository. This includes those that need dependencies for their own build, notice errors in the repository metadata, or projects that wish to have their releases added to the Maven repository.

- Maintaining your Metadata - Information for third-party projects
- Guide to Maven Evangelism - Helping to improve the metadata of the dependencies you use
- Guide to uploading artifacts - How to get things uploaded to the repository

70 Guide to Maven Evangelism

70.1 Guide to add, improve or fix metadata in the Central Maven 2 repository

There are artifacts in the repository that don't have POMs. They come from the Maven 1 repositories of our partners (Apache Software Foundation, Codehaus,...). We know it but can't do anything unless you provide a POM for it or you ask the project in question to add the POM when they add the artifacts.

We don't change dependencies in POMs already in the repository anymore as builds need to be reproducible. Same applies to POMs that don't exist. We can add a POM with no dependencies, because doing any other way would break previous builds that were using that project.

An alternative is to create a new version with the fixes. If the broken project is org.foo/bar/1.0 you can provide a fixed POM,JAR,... under org.foo/bar/1.0-1 (add a comment to the POM explaining what is being fixed and why). See Maven Respository Upload for the instructions to get this new version in the repository.

You need to contact the original publisher of the metadata to make sure in next versions it will be fixed or improved before getting it into the repository.

For any other types of issues related to metadata in the repository, open an issue at JIRA MEV with the relevant information and explain the reasons why it is an issue.

- Important:* by default assume that we won't trust your info, so you must provide all links to the project documentation you can to convince us that your solution is right.

71 Guide to uploading artifacts

71.1 Guide to uploading artifacts to the Central Repository

In order for users of Maven to utilize artifacts produced by your project you must deploy them to a remote repository. Many open source projects want to allow users of their projects who build with Maven to have transparent access to their project's artifacts. In order to allow for this a project must have their artifacts deployed to the Central Repository.

71.2 Requirements

Only releases can be uploaded to the central repository, that means files that won't change and that only depend on other files already released and available in the repository.

There are some requirements for the minimal information in the POMs that are in the central repository. At least these must be present:

- modelVersion
- groupId
- artifactId
- packaging
- name
- version
- description
- url
- licenses
- scm url
- dependencies

71.2.1 groupId

The `groupId` will identify your project uniquely across all projects, so we need to enforce a naming schema. For projects with artifacts already uploaded to the Central Repository it can be equal to the one used previously, but for new projects it has to follow the package name rules, what means that has to be at least as a domain name you control, and you can create as many subgroups as you want. There are a lot of poorly defined package names so you **must provide proof that you control the domain** that matches the groupId. Provide proof means that the project is hosted at that domain or it's owned by a member, in that case you must give the link to the registrar database (whois) where the owner is listed and the page in the project web where the owner is associated with the project. eg. If you use a com.sun.xyz package name we expect that the project is hosted at http://xyz.sun.com.

Look at More information about package names. Check also the guide about Maven naming conventions.

Examples:

- www.springframework.org -> org.springframework
- oness.sf.net -> net.sf.oness

71.2.2 Explanation

Some folks have asked why do we require all this information in the POM for deployed artifacts so here's a small explanation. The POM being deployed with the artifact is part of the process to make

transitive dependencies a reality in Maven. The logic for getting transitive dependencies working is really not that hard, the problem is getting the data. The other applications that are made possible by having all the POMs available for artifacts are vast, so by placing them into the repository as part of the process we open up the doors to new ideas that involve unified access to project POMs.

We also ask for license now because it is possible that your project's license may change in the course of its life time and we are trying create tools to help normal people sort out licensing issues. For example, knowing all the licenses for a particular graph of artifacts we could have some strategies that would identify potential licensing problems.

71.2.3 A basic sample:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven</artifactId>
  <packaging>jar</packaging>
  <name>Maven core</name>
  <version>2.0</version>
  <description>The maven main core project description</description>
  <url>http://maven.apache.org</url>
  <licenses>
    <license>
      <name>The Apache Software License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
  <scm>
    <url>http://svn.apache.org/viewcvs.cgi/maven</url>
  </scm>
  <dependencies>
    <dependency>
      <groupId>...</groupId>
      <artifactId>...</artifactId>
      <version>...</version>
    </dependency>
    ...
  </dependencies>
  <!--
  NOT ALLOWED: (see FAQ)
  <repositories></repositories>
  <pluginRepositories></pluginRepositories>
  -->
</project>
```

71.2.4 FAQ and common mistakes

- I have other `repositories` or `pluginRepositories` listed in my POM, is that a problem?

  Yes, the central repository must be self contained, which means that all your dependencies must already be in the central repository. You need to remove the `repositories` and `pluginRepositories` entries and make sure your project still builds when your local repository cache is empty.

The only exception allowed is when a dependency can not be distributed from the central repository due to the license. In that case only the POM for that dependency is required, listing where the dependency can be downloaded from. See an example.

- I have a patched version of the foo project developed at foo.com, what `groupId` should I use?

  When you patch / modify a third party project, that patched version becomes your project and therefore should be distributed under a `groupId` you control as any project you would have developed, never under `com.foo`. See above considerations about `groupId`.
- My project is hosted at a project hosting service like SourceForge or dev.java.net, what should I use as groupId?

  If your project name is `foo` at SourceForge, then `net.sf.foo`. If it's `foo` at dev.java.net, then `net.java.dev.foo`

71.3 Sync'ing your own repository to the central repository automatically

This is the preferred process. You first have to setup your project to deploy to a remote repository.

You must make sure the remote repository server has rsync or rsync over ssh, there are free services like Sourceforge that provide you ssh server access. Or you can setup your own server.

After you are able to deploy to your remote repository **make sure you only deploy releases**. Then you need to provide us the following information in a comma delimited format

- groupId, eg. `org.apache.maven`
- location, eg. `mavensync@web.sourceforge.net:/home/groups/m/ma/maven-js-plugin/htdocs/m2repo` for rsync over ssh or `rsync://maven2.hortis.ch/releases` for non secure rsync
- protocol, `rsync` or `rsync_ssh`
- contactName, your name
- contactMail, your email (not a mailing list)

The comma delimited version would be something like this. Note that the two last columns are for internal purposes, so you can omit them.

```
"org.apache.maven","mavensync@web.sourceforge.net:/home/groups/m/ma/maven-js-plugin
"org.apache.maven","rsync://maven2.hortis.ch/releases","rsync","John Doe","doe@doe.
```

You can see the list of automatically synchronized repositories.

Open an issue in JIRA with the information (in the comma delimited format), and we'll add it to the list of automatically synced repositories.

Make sure you **provide proof of owning the domain** that matches the `groupId` (see `groupId` considerations above). Proof means either the server to sync from has a name under that domain, your name shows up in a prominent place in the domain or you provide a link to a whois database where your name shows up as the domain owner.

If you are using ssh in your own server you need to add the maven public key to the authorized ones to allow us to log in to the machine.

Remember to subscribe to the repo-maintainers mailing list. (Or watch the Atom feed.)

Important: nothing is deleted or changed in the Central Repository after it is synced (except maven-metadata.xml files).

71.3.1 Sync FAQ

- Can I provide a sync for a `groupId` of a third party project?

If the third party project is not willing to provide a repository to sync from, and you are a regular user of Maven and the third party project, the answer is yes. You can set up a repository as if the project were yours (see instructions above). Please create a JIRA issue first describing what you are trying to do and why, and you may be designated the "unofficial" mantainer of the repository section associated with that project, and you will be responsible of publishing the new releases for that project in your repository if other users request it. You can opt out at any time.

71.4 Manual upload of artifacts

Note that this manual process is time consuming and **will only be accepted for a limited number of requests**. If you want to upload frequently read the section above about automatic sync.

Estimated process time is FOUR WEEKS. If you want to use the manual process, that is the estimated time to process **if no problems are detected**. It means that for each version you release and want to upload to the central repository you will have to wait that time. If a problem is detected, it will be noted in the JIRA issue and **you will wait again** until the next time the issues are processed.

71.4.1 Step 1: Create an upload bundle

Use the repository plugin provided with the standard Maven distribution to create an upload bundle:

```
mvn repository:bundle-create
```

The bundle will be created in your `target` directory with the name: `${pom.artifactId}-${pom.currentVersion}-bundle.jar`

If you want to include a jar with java sources in your upload (recommended, unless your license doesn't allow sources to be redistributed) the command to run is:

Note: due to a bug in `repository:bundle-create` you will need to **manually add the javadoc jar** to the bundle jar, using zip or any other compression program.

```
mvn source:jar javadoc:jar repository:bundle-create
```

If you are not using Maven as your build system, and want something uploaded to the Central Repository then you just need to make a bundle jar manually. Please use the `jar` executable, not `zip`, `pkzip` or equivalent. The bundle should have the following content:

```
pom.xml
foo-1.0.jar (or whatever artifact is referred to in the pom.xml)
foo-1.0-sources.jar (optional, jar containing java sources)
foo-1.0-javadoc.jar (optional, jar containing javadocs)
```

The names of the jar files inside the bundle must be built from the <artifactId> and <version> in the `pom.xml` file, like this:

```
${artifactId}-${version}.jar
${artifactId}-${version}-sources.jar  (optional)
${artifactId}-${version}-javadoc.jar  (optional)
```

Note: the bundle will be read by a script, so it must follow the above format.

Be sure to always check previous versions of the POMs in the repository to use the information already there as a base.

71.4.2 Step 2: Posting the request

Post your request to JIRA. Make sure that the project is "Maven Upload Requests" and the issue type is "Wish". In the Description field, you must write the URL to the upload bundle. If you're uploading more than one bundle please add all the URLs under the same issue. Then leave a blank line and provide the following information:

- A URL where the project can be found.
- If you are one of its developers, a URL where your name or email can be found inside the project site.

This will speed up the uploading process.

You can place any additional comments in the following paragraph. So the Description field might look like:

```
http://wiggle.sourceforge.net/downloads/wiggle-1.0-bundle.jar
http://wiggle.sourceforge.net
http://wiggle.sourceforge.net/team-list.html
I'm a developer in wiggle, please upload!
```

or

```
http://wiggle.sourceforge.net/downloads/wiggle-1.0-bundle.jar
http://wiggle.sourceforge.net
http://wiggle.sourceforge.net/team-list.html
I'm a developer in wiggle, and want to use the org.wiggle groupId
I own wiggle.org domain, you can see my name in
http://reports.internic.net/cgi/whois?whois_nic=wiggle.org&type=domain
or you can see the project web page in www.wiggle.org
```

71.4.3 Manual process FAQ and common mistakes

- I use parent POMs. How do I include them in the bundle?

  You can't. You need to use the automated synchronization process noted above.
- I want to upload several bundles. Do I need to create a JIRA issue for each one?

  No, please ignore the Bundle URL field in this case. Just put the URLs of all the bundles in the Description or Comments fields of one single JIRA issue.

71.5 Maven partners

There is a list of sites that automatically sync their project repository with the central one. If you want a project from any of these sites to be uploaded to the Central Repository you'll have to contact their project maintainers. Check the updated list

Some of the most known groups automatically synced are:

- The Apache Software Foundation
- Codehaus
- MortBay Jetty
- OpenSymphony

71.6 For Maven developers

The scripts to make the upload to the repository are at `https://svn.apache.org/repos/asf/maven/repository-tools/trunk` in the `src/bin` directory.

Those scripts are checked out to repo1.maven.org in the directory `/home/maven/bin`, so after logging in as the user `maven` you can go to the directory `bin/bundle-upload` and run

```
./deploy-bundle bundle_URL [groupId] [version] [classifier]
```

That script will download the bundle, decompress it and show the POM. You have to make sure that the POM is correct. The script uses the command `less` to show the POM. Use the `space` key to step through it to the end. The press the `q` key to proceed. After that a summary with `groupId`, `artifactId` and `version` will be shown, and whether or not the group already exists. This is useful as we have to be careful creating new groups, making sure they follow the conventions and that they don't already exist with another name. If the POM is not correct or there's any doubt the upload must be aborted with `Ctrl-C`, and a comment posted in the upload request in JIRA. If there's no response from the reporter within one month the request will be closed as "Incomplete".

If `groupId` and `version` are not specified in the command line, the script will try to get the values from the POM. It won't work if the POM extends another POM and those elements are not present in the POM included in the bundle.

Things to remember:

- All the dependencies have to already be present in the central repository.
- If there are no dependencies it's suspicious, and the reporter must be asked if that's correct. Do that in an issue comment in JIRA.
- Parent POMs have to already be present in the central repository.
- All the minimal information previously mentioned has to be in the POM.
- POMs must include at least as much information as previous versions, and the dependencies shouldn't change too much.
- While checking a previous version, also check if it was relocated. If so ask the reporter to update the bundle with the new information.
- GroupIds have to follow the previously stated naming conventions.
- Upload requests for popular projects require you to be extremely careful (javax.* groups, spring, hibernate, ...).

72 Maven Developer Centre

72.1 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.

72.1.1 Contributors Resources

- Guide to helping with Maven
- Developing Maven 2
- Common Bugs and Pitfalls
- Building Maven 2
- Continuous Integration
- Source Repository

72.1.2 Committers Resources

72.1.2.1 General Resources

- Guide for new Maven committers
- Committer Environment
- Committer Settings

72.1.3 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 SVN Convention

Note: If you cannot find your answers here, feel free to ask the Maven Developer List.

72.1.4 Making Releases

- Making GPG Keys
- Release Process

72.1.5 Deploy Maven references

- Deploy Maven Current References

72.1.6 Others Resources

- Maven Web Stats
- Maven Mailing List Stats
- ASF Development Infrastructure Information

- About the Apache Software Foundation

73 Developing Maven 2

73.1 Developing Maven 2

This document describes how to get started into developing Maven 2 itself. There is a separate page describing how to building Maven 2.

73.1.1 Finding some work to do

First of all you need something to work on! Unless you have found a particular issue you would like to work on the Maven team has categorized a few issues that we could use *your* help to solve them.

JIRA has RSS feeds available if you'd like to include those in your favorite feed aggregator.

We categorize the issues in three different categories:

- **Novice**: No previous exposure to the code needed. *(rss feed)*
- **Intermediate**: Exposure to Maven pluins and/or internals required. *(rss feed)*
- **Expert**: Good knowledge of Maven internals and it's dependencies required. *(rss feed)*

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.

73.1.2 Creating and submitting a patch

When you have either completed an issue or just want some feedback on the work you have done, create a patch and attach the patch to the issue in question. We have a couple of guidelines when creating patches:

- Patch the trunk, not a tag. Otherwise, your patch is outdated the moment you create it and might not be applicable to the development head.
- Always create the patch from the root of the Maven project, i.e. where the `pom.xml` file is.
- If this was a new piece of work without a JIRA issue, create a JIRA issue for it now.
- Name the file `MNG-<issue number>-<artifact id>.patch`.
- Attach the patch to the JIRA issue you were working on (do not paste its content in as a comment though). When adding the patch add a comment to the issue explaining what it does. Shortly after, someone will apply the patch and close the issue.

An example on how to create a patch from the command line:

```
$ svn diff > MNG-123-maven-core.patch
```

If you are picking up an issue with a existing patch attached to the issue you can apply the patch to your working directory directly from JIRA like this. The `wget` and `patch` commands will only be available if you are on a UNIX platform or using Cygwin on windows.

```
$ wget -O - -q <URL to the patch from JIRA> | patch -p0
```

If the patch is in a local file `MNG-123.patch` and you want to apply that use this command:

```
$ patch -p0 < MNG-123.patch
```

A couple of notes:

- If you are using another tool for creating patches, make sure that the patch doesn't include absolute paths. Including absolute paths in the patch will make the useless for us as we most likely don't have the same directory structure as you.
- Make sure that you follow our code style, see Further Links.

73.1.3 Patch acceptance criteria

73.2 There are a number of criteria that a patch 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 patches may be rejected as they take the project in a different direction to that which the current development community has chosen. This is usually discussed on an issue well before a patch 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 patches 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.

73.3 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!

73.3.1 Related Projects

Maven 2 has a few dependencies on other projects.

- **Plexus**

  Plexus is a full-fledged container supporting different kinds of component lifecycles. It's native lifecycle is like any other modern IoC container, using field injection of both requirements and configuration. All core Maven 2 functionality are Plexus components.

  You can read more about Plexus.
- **Modello**

  Modello is a simple tool for representing an object model and generate code and resources from the model. Maven is using Modello to generate all Java objects, XML readers and writers, XML Schema and HTML documentation.

  You can read more about Modello.
- **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 support 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.
- **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.
- **Mojo**

"Mojo" is really two things when it comes to Maven. It is both Maven's plug-in API but also a separate Codehaus project hosting these plugins.

The Mojo Project is a plugin forge for all non-core Maven plugins. As we try to keep the Mojos as independent of Maven as possible to increase their reuse we try to keep them a bit away from Maven itself. There is also a lower bar for becoming a part of the project.

73.3.2 Sub Projects

73.3.2.1 Maven SCM

Maven SCM (Source Control Management) is an reusable API which is independent of Maven itself and it is used by the SCM related Maven Plugins. The core part of Maven itself doesn't depend on Maven SCM.

73.3.2.2 Maven Wagon

Maven Wagon is also a standalone API that deals with transporting files and directories. Maven Core uses the Wagon API to download and upload artifacts and artifact metadata and the site plug-in uses it to publish the site.

73.3.3 Further Links

- Maven Code Style And Code Convention
- Maven JIRA Convention
- Maven SVN Convention

74 Building Maven 2

74.1 Building Maven

74.1.1 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.

Note, that you don't need to bootstrap Maven for day to day use, or to develop plugins. While we encourage getting involved and fixing bugs that you find, for day to day use we recommend using the latest release.

74.1.2 Checking out the sources

All of the source code for Maven and its related libraries is in Subversion. You can browse the repository, or checkout specific modules directly.

To build Maven 2.2 (the current stable branch), you need the `maven-2.2.x` branch of the `maven-2` module. To check that out, run the command:

```
svn co https://svn.apache.org/repos/asf/maven/maven-2/branches/
maven-2.2.x maven-2.2.x
```

To build Maven 3.0 (unstable development branch), you need the `trunk` of the `components` module. To check that out, run the command:

```
svn co https://svn.apache.org/repos/asf/maven/components/trunk maven-3
```

Alternatively, you can check out all Maven projects in one directory using:

```
svn co https://svn.apache.org/repos/asf/maven/trunks maven
```

If you have checked out `trunks`, the `maven-2.2.x` directory will contain the Maven 2.2 source code, and the `components` directory will contain the 3.0 source code. Note that neither directory contains any of the plugins.

Note: For Windows users, the checkout could be not complete with the following message:

```
svn: Can't open file 'XXX': The system cannot find the path specified.
```

The problem is that while Windows allows filenames up to 256 characters the maximum path length it allows is 260 characters. You will be able to check it out to the root directory without problem.

74.1.2.1 Other Modules

Other modules you might be interested in related to Maven development are:

- `plugins/trunk` - The sources of the Maven plugins. These can be individually installed, or built together.
- `plugin-tools/trunk` - Set of tools for Maven plugins like test harness.
- `release/trunk` - Release manager and plugin.
- `site/trunk` - The Maven website.
- `skins/trunk` - Skins for generated site used by site plugin.
- Some Maven sub projects
  - `wagon/trunk` - Maven Wagon, used by the artifact code and others for providing the transport layer to get and put artifacts in a repository.

    - `scm/trunk` - Maven SCM, a generic API to communicate with various different SCM providers, used by Continuum and the release and SCM plugins.
    - `doxia/trunk` - The Doxia site generation library used by several report plugins and site plugin.
    - `surefire/trunk` - The Surefire test runner.
- `shared/trunk` - Collection of shared libraries like file/path handling.
- `sandbox/trunk` - Sandbox codes.
- Plexus - the IoC container used by Maven.

If you're looking at the `trunks` directory with ViewVC, there is seemingly nothing there. We use externals definitions to link together all the trunks into one logical location for convenience. If you want to see what is being linked into one logical location you can use the following command:

```
svn propget svn:externals
```

74.1.3 Building Maven

74.1.3.1 Building Maven With Maven Installed

If you already have Maven installed, it can be faster to build a new version with Maven, rather than a clean bootstrap.

To do this, run from the `components` or `maven-2.2.x` directory:

```
mvn install
```

Optionally, you can use the following to run the full (long) suite of integration tests:

```
mvn install -Prun-its
```

The assemblies will be created in `apache-maven/target` for Maven 2.0.x or `maven-distribution` for Maven 2.1, and can be unzipped to the location where you'd like Maven installed.

74.1.3.2 Building Maven Without Maven Installed

If you do not have Maven installed, you can use Apache Ant to build Maven.

Once you have checked out the code, change into the `components` or `maven-2.2.x` directory that was created.

Set the M2_HOME environment variable to the location that should contain Maven. This directory **must** be named after the Maven version you want to build and install, for example `/usr/local/maven-2.2-SNAPSHOT`.

```
export M2_HOME=/usr/local/maven-2.2-SNAPSHOT
PATH=$M2_HOME/bin:$PATH
```

or

```
set M2_HOME=c:\maven-2.2-SNAPSHOT
set PATH=%M2_HOME%\bin;%PATH%
```

From this, run the `ant` command:

```
ant
```

This will download dependencies, build Maven, and install it into the directory you specified as `M2_HOME` above.

If you have any problems or get any failures during the run, please report them to the Maven Developers List.

For more information, consult the project help in the Ant build file.

```
ant -projecthelp
```

The result is included here for convenience:

```
Buildfile: build.xml
    The first time you build Maven from source, you have to build Maven without
    Maven. This Ant script builds a minimal Maven, just enough to re-launch
    Maven again in this directory and generate an installation assembly. Then we
    extract the assembly and re-run the Maven build one more time, this time
    with the full generated Maven.
    To run this script, you must set the M2_HOME environment variable or the
    maven.home property to the location that should contain Maven. This
    directory *must* be named after the maven version you want to install, e.g.
    /usr/local/maven-2.1-SNAPSHOT.
    You can set the maven.repo.local property to specify a custom location for
    your local repository for the bootstrap process.
Main targets:
 classpath-pre     constructs a classpath reference containing our dependencies,
                   and verifies that all files are present
 clean-bootstrap   cleans up generated bootstrap classes
 compile-boot      compiles the bootstrap sources
 extract-assembly  extracts the maven assembly into maven.home
 generate-sources  generates Java sources from Modello mdo model files
 maven-assembly    generates the Maven installation assembly using the bootstrap
                   Maven
 maven-compile     compiles Maven using the bootstrap Maven, skipping automated
                   tests
 pull              copies all required dependencies from the Maven remote
                   repository into your local repository.  Set the 'skip.pull'
                   property to skip this step, but only if you're sure you
                   already have all of the dependencies downloaded to
                   your local repository
 run-full-maven    runs the full extracted Maven, now with tests
Default target: all
```

75 Committer Environment

75.1 Introduction

This document is intended to set up the Maven committer environment.

75.2 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.

75.3 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

75.4 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.

75.4.1 IntelliJ IDEA 4.5+

Download `maven-idea-codestyle.xml` and copy it to `~/.IntelliJIDEA/config/codestyles` then restart IDEA. 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.

75.4.2 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.

75.5 Setting up SSH public/private keys

By default, SSH (Secure Shell) asks you to enter your password each time, i.e.:

```
>ssh vsiveton@apache.org
Password:
```

SSH can be set up with public/private key pairs so that you don't have to type the password each time. You need to execute the following on your development machine:

```
> ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (~/.ssh/id_dsa): (just type return)
Enter passphrase (empty for no passphrase): (just type return)
Enter same passphrase again: (just type return)
Your identification has been saved in ~/.ssh/id_dsa
Your public key has been saved in ~/.ssh/id_dsa.pub
The key fingerprint is:
ec:06:c7:44:9e:a6:2c:c0:8a:87:04:07:a0:5b:94:d2 YOUR_MACHINE_USERNAME @ YOUR_MACHIN
```

Then, paste the content of the local `~/.ssh/id_dsa.pub` file into the file `/home/YOUR_APACHE_USERNAME/.ssh/authorized_keys` on the Apache remote host.

Note: under Cygwin, it is located at `{cygwin.install.path}\home\YOUR_MACHINE_USERNAME\.ssh`. You need to copy the content of `{cygwin.install.path}\home\YOUR_MACHINE_USERNAME\.ssh` into `C:\Documents and Settings\YOUR_MACHINE_USERNAME\.ssh` for Maven.

To test the installation, try to log in again on Apache. You should not be asked for your password any more.

```
>ssh vsiveton@apache.org
Last login: Tue Oct 10 03:50:10 2006 from ipXXX-XXX-XXX-XXX
Copyright (c) 1980, 1983, 1986, 1988, 1990, 1991, 1993, 1994
        The Regents of the University of California.  All rights reserved.
FreeBSD 6.1-RELEASE (SMP-turbo) #0: Thu May 11 11:50:25 PDT 2006
This is an Apache Software Foundation server.
For more information, see http://www.apache.org/dev/
Time to change your password? Type "passwd" and follow the prompts.
                -- Dru <genesis@istar.ca>
-bash-2.05b$
```

75.6 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
- WinSCP: SFTP client for Windows.
- TortoiseSVN: Subversion client, implemented as a Windows shell extension.
- GnuPG: GNU Privacy Guard.

76 Committer Settings

76.1 Introduction

This document is intended to set up the Maven committer settings, i.e. the `${user.home}/.m2/settings.xml`.

76.2 Enable Apache Servers

Maven uses several servers configuration to deploy snapshots, releases and documentation on the Apache servers. You need to tell to Maven what your Apache username is. Please note that some servers use your SVN credentials while others use your SSH credentials.

```
<settings>
...
  <servers>
    <!-- To publish a snapshot of some part of Maven -->
    <server>
      <id>apache.snapshots.https</id>
      <username> <!-- YOUR APACHE SVN USERNAME --> </username>
      <password> <!-- YOUR APACHE SVN PASSWORD --> </password>
    </server>
    <!-- To publish a website of some part of Maven -->
    <server>
      <id>apache.website</id>
      <username> <!-- YOUR APACHE SSH USERNAME --> </username>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
    </server>
    <!-- To stage a release of some part of Maven -->
    <server>
      <id>apache.releases.https</id>
      <username> <!-- YOUR APACHE SVN USERNAME --> </username>
      <password> <!-- YOUR APACHE SVN PASSWORD --> </password>
    </server>
    <!-- To stage a website of some part of Maven -->
    <server>
      <id>stagingSite</id> <!-- must match hard-coded repository identifier in site
      <username> <!-- YOUR APACHE SSH USERNAME --> </username>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
    </server>
    ...
  </servers>
</settings>
```

You also need to be a member of the group `apcvs` and `maven` on `people.apache.org`.

77 Maven Code Style And Conventions

77.1 Maven Code Style And Code Conventions

This document describes how developers and contributors should write code. The reasoning of these styles and conventions is mainly for consistency, readability and maintainability reasons.

77.1.1 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 in all versionned files.
- **Trailing Whitespaces**: Remove all trailing whitespaces. If your are an Eclipse user, you could use the Anyedit Eclipse 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, could override these generic rules.

77.1.2 Java

77.1.2.1 Java Code Style

The Maven style for Java is mainly:

- **White space**: One space after control statements and between arguments (i.e. `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.
- **Readingness**: Specify code grouping members, if needed. For instance in a Mojo class, you could have:

```
public class MyMojo
{
    // ----------------------------------------------------------------------
    // Mojo components
    // ----------------------------------------------------------------------
    /**
     * Artifact factory.
     *
     * @component
     */
    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.

77.IntelliJ IDEA 4.5+

Download `maven-idea-codestyle.xml` and copy it to `~/.IntelliJIDEA/config/codestyles` then restart IDEA. 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.

77.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.

77.1.2.2 Java Code Convention

For consistency reasons, our Java code convention is mainly:

- **Naming**: Constants (i.e. static final members) values should always be in upper case. Using short, descriptive names for classes and methods.
- **Organization**: Avoid using a lot of public inner classes. Prefer interfaces instead of default implementation.
- **Modifier**: Avoid using final modifier on all member variables and arguments. Prefer using private or protected member instead of public member.
- **Exceptions**: Throw meaningful exceptions to makes debugging and testing more easy.
- **Documentation**: Document public interfaces well, i.e. all non-trivial public and protected functions should include Javadoc that indicates what it does. **Note**: it is an ongoing convention for the Maven Team.
- **Testing**: All non-trivial public classes should include corresponding unit or IT tests.

77.1.2.3 JavaDoc Convention

TO BE DISCUSSED

77.1.3 XML

77.1.3.1 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
    <!-- ============================================================================
```

77.1.3.2 Generic XML Code Convention

No generic code convention exists yet for XML files.

77.1.3.3 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:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
  <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: The team plans to create a Maven plugin with reorder and reformat goals (See MOJO-928).

77.1.3.4 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.

77.1.3.5 FML Code Convention

For readability reasons, FML files should respect:

- **FAQ**: Always use a new line with indentation for <faq/> tags.

78 Maven JIRA Convention

78.1 Maven JIRA Convention

This document describes how Maven developers should use JIRA, our issue tracking.

78.1.1 When To Create a JIRA Issue?

This section discusses when to create a JIRA issue versus just committing a change in SVN.

- **Minor changes**, like code reformatting, documentation fixes, etc. that aren't going to impact other users can be committed without much issue.
- **Larger changes**, like bug fixes, API changes, significant refactoring, new classes, and pretty much any change of more than 100 lines, should have a JIRA ticket associated with it, or at least an email discussion.

78.1.2 How To Use Issue Details?

This section presents some conventions about the issue fields.

78.1.2.1 Priority

Committers has the responsibility to realign priority by editing the issue.

Reasoning: having a correct release note.

78.1.2.2 Assignee

Committers could assign an issue to a specific committer if he thinks it is the right committer.

78.1.2.3 Component/s

Committers has the responsibility to specify the correct the component by editing the issue.

Reasoning: having a correct release note.

78.1.2.4 Affects Version/s

By default, the Maven team considers that an issue, which affects a given version, affects also precedent versions, i.e. issue which affects Maven 2.0.9 will affect also 2.0, 2.0.1 ... 2.0.9. If it is a regression, the committers should specify the affected versions.

Reasoning: having a correct release note.

78.1.2.5 Fix Version/s

TO BE DISCUSSED

78.1.2.6 Time Tracking

The Maven team never uses it. Committers could do it, but like said, it will never be used.

78.1.3 Further Links

- JIRA Documentation
- What is an Issue?
- What is a project?

- how we handle JIRA versions Thread

79 Maven SVN Convention

79.1 Maven SVN Convention

This document describes how developers should use SVN, our SCM.

79.1.1 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

79.1.2 Commit Message Template

Commits should have a message that follows this template:

```
[issue1, issue2] <<comment>>
Submitted by: (when it was a patch, put that persons name there)
o some comments
```

Where:

- **issue** can be omitted if there was no relevant JIRA issue, though it is strongly encouraged to create one for significant changes.
- **Submitted by** only needs to be specified when a patch is being applied for a non-committer.
- **comments** some words about the commits.

79.2 eg:

```
[MNG-1456] Added the foo to the bar
Submitted by: Baz Bazman
o applied without change
```

79.2.1 Apply User Patch

By default, the committer should apply the patch without any **major** modifications. In a second step, the committer could apply any changes as usual.

79.2.2 Edit Commit Message

If you want to edit a commit message, you could call:

```
svn pe svn:log --revprop -r XXX
```

where **XXX** is the wanted version

79.2.3 Other useful Subversion commands while developing

If you've done a chunk of work and you would like ditch your changes and start from scratch use this command to revert to the original checkout:

`$ svn revert -R .`

The `-R` argument means that the command will recurse down all directories and revert all changes.

Before committing code to the Subversion repository we always set the `svn:ignore` property on the directory to prevent some files and directories to be checked in. We always exclude the IDE project files and the `target/` directory. Instead of keeping all of the excludes in mind all the time it's useful to put them all in a file and reference the file with the `-F` option:

```
$ svn propset svn:ignore -F ~/bin/svnignore .
```

An example svnignore file:

```
target
*~
*.log
.classpath
.project
*.ipr
*.iws
*.iml
```

80 Making GPG Keys

80.1 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.

80.1.1 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\secr
created
gpg: keyring `C:/Documents and Settings/Siveton Vincent/Application Data/gnupg\pubr
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 keysize do you want? (2048) 2048
Requested keysize 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
generator a better chance to gain enough entropy.
++++++++++++++++++++.++++++++++..++++++++++++++++++++++++++++++++++++++++++++
+++.+++++++++++++++.+++++++++++++++++++..+++++++++++++++>++++++++++.............
.........................+++++
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:
created
gpg: key 07DDB702 marked as ultimately trusted
public and secret key created and signed.
gpg: checking the trustdb
```

80.1.2 gpg --list-sigs

```
>gpg --list-sigs "Vincent Siveton" && gpg --armor --export "Vincent Siveton"
pub   1024D/07DDB702 2006-10-10
uid                  Vincent Siveton <vsiveton@apache.org>
sig 3        07DDB702 2006-10-10  Vincent Siveton <vsiveton@apache.org>
sub   2048g/D2814A59 2006-10-10
sig          07DDB702 2006-10-10  Vincent Siveton <vsiveton@apache.org>
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.5 (MingW32)
mQGiBEUrnAsRBACQDiYGc1IQmkENLO9iznBg8otGPEbzqQozT5tsip5mB30f6Mc/
uuLxJkLdna7Ul3goIXDtCeLJq38gHvruNtVNR6S+juJFkd5sLEH8UJ18PbKuo/9I
KGlzjtCYUUDC48czRr0efhqd54NH8ydNdpaZ76NGPPYfpXtk7kKgH/nPiwCgxozK
IG2frMuWIvdFafbqdAl7y/sD/1Csf0r9jtHEeXOuyhm8jCGrSwzLbHUGKPUQP37P
ajECHoWp6HnvHEEEpgVl+UjfZvrcVhzUoP+3r5HAugqERfkzK8AWc7qjIRjf64kU
sjvto31G2KYz17Y8K9y4LkRkUspu8uw903pKnW/QELg4vvPVaEYpVVIdS6+cUreu
V0hOA/4tW7T/GpzSbQmjvnIRQ7GVHbQeXsANwrS6NmGYIxafN9P9dfHV+eUieTu6
rvMP9coOhTYyEKZksrXw2MmXx5SzgxsXT0g4wDXbwxPYFfIdGUzFMobnVXiZ3G8l
JEl9cML0cg3ZL1SoDmVf2iG3e3Yxxsne4AE1SU+0bbq0t7rqALQlVmluY2VudCBT
aXZldG9uIDx2c2l2ZXRvbkBhcGFjaGUub3JnPohgBBMRAgAgBQJFK5wLAhsDBgsJ
CAcDAgQVAggDBBYCAwECHgECF4AACgkQhPTUcAfdtwLP3gCbB/V1afp8hzxgirCS
d2r6zCkJQ2IAoLKD/RIkkerNintYzrubJliJKBsRuQINBEUrnBgQCAD1+Sx+sBDL
1XCDtxQGsrZmMnJJVK/w4TPa/8weJkuZ1GSpINOjInmqESuehvCLoOoyfcuDVXlR
PUZhKZLPEKfJlFptGNK19oTO/CoQN+SJLwR41FoumsBaf1YSSRpAukyx2J6cUxqf
uWrK/T8PmgDw4YzmY96tev//41eZ5tSOxpoUM8ypnTaShtS9pjgHijEG0b7wBqeU
e1OGOiLHgKyjEJUmlTaLm1SxJ84eq0uAvYb+rb/QoWWLpjvr2/mo1kzUvCPgo3fh
kgOxCxsC9QD836Mi5HFK6CRYU3yAFu5+/jM+LJzELy3u7uMuOSP6yuiK8WXopdbN
WHOiJQfdc2gTAAMFCADdljjAG7L+8de6JzsEduKErKqWlTEWa99n1knaGKzdUUOP
WrKxwqgI6PAJbxOfG4vBdDa6M6+nySJDMybQsOCFyNx91/7jYkgkmv8Jkt8CTW4z
P4HKlFYMAFpU95ftpTAAMAlr+t+nZRTHi94/VHMv4yLGzB/xapbzToHKuUt1Yqom
Ncio5px7RVoicn13/i/GeY72fIdC2LRGo6PXlmmDQemoAnUw0RJoEtzapb0j/tWd
BdAtQQX/Ks7qhk4aDDHGgO+CdHAB8PLHDpMpUX5Zc9JX1xhyJcS8d/LPUpXtt9WN
eekqDpx+jNmySJr6os7rPAkjx6jDUvHPiuKdT4aviEkEGBECAAkFAkUrnBgCGwwA
CgkQhPTUcAfdtwJL9ACgmLuDxE+oZaMhyFSmXWN0fM36Bd8AoLYrvwydB9+nNnhJ
85TjkMPTgjp9
=Hg4C
-----END PGP PUBLIC KEY BLOCK-----
```

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.

You can read more about Checksums And Signatures.

81 Release Process

81.1 Releasing A Maven Project

What follows is a description of releasing a Maven project to a staging repository, whereupon it is scrutinized by the community, approved, and transferred to a production repository.

81.1.1 Prerequisites

Be sure that:

- you have all Maven servers defined in your settings.xml. For more information, please refer to Committer settings.
- you have created your GPG keys. For more information, please refer to Making GPG Keys.
- you have a GPG client installed and on your shell's path. See http://www.gnupg.org/.
- you have a Subversion 1.5+ client installed and on your shell's path. See http://subversion.tigris.org/.
- you have a Java 1.4.2 JDK installed and on your shell's path. See http://java.sun.com/j2se/1.4.2/download.html.
- you have set the environment variable MAVEN_OPTS=-Xmx512m
- you are using Maven 2.0.x (or 2.2.x), as Maven 2.1.x is known to produce wrong gpg pom signatures (see MGPG-14).

Formerly, a `release` profile was required in the `${user.home}/.m2/settings.xml` to define the staging repository. As of inheritance from the Apache parent POM version 5, a repository manager will automatically handle staging (see below for details). Hence, configuration of `deploy.altRepository` is no longer necessary and should be removed from your existing `release` profile.

Here's what your `release` profile might look like in your `${user.home}/.m2/settings.xml`:

```
<settings>
  ...
  <profiles>
    <profile>
      <id>apache-release</id>
      <properties>
        <gpg.passphrase> <!-- YOUR KEY PASSPHRASE --> </gpg.passphrase>
      </properties>
    </profile>
  </profiles>
  ...
</settings>
```

Everything that you need to release has been configured in the POM all Maven projects inherit from. The release plugin configuration being used is the following:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.0-beta-9</version>
  <configuration>
    <useReleaseProfile>false</useReleaseProfile>
    <goals>deploy</goals>
    <arguments>-Papache-release</arguments>
  </configuration>
</plugin>
```

And the profile being used for releases is the following:

```
<profile>
  <id>apache-release</id>
  <build>
    <plugins>
      <!-- We want to sign the artifact, the POM, and all attached artifacts -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-gpg-plugin</artifactId>
        <configuration>
          <passphrase>${gpg.passphrase}</passphrase>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>sign</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <!-- We want to deploy the artifact to a staging location for perusal -->
      <plugin>
        <inherited>true</inherited>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-deploy-plugin</artifactId>
        <configuration>
          <updateReleaseInfo>true</updateReleaseInfo>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <executions>
          <execution>
            <id>attach-sources</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <configuration>
          <encoding>${project.build.sourceEncoding}</encoding>
        </configuration>
        <executions>
          <execution>
            <id>attach-javadocs</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

81.1.2 Verify you are using JDK 1.4.2

Maven 2.0.X and its plugins should be built with JDK 1.4.2.

```
>mvn --version
...
Maven version: 2.0.X
Java version: 1.4.2_18
```

81.1.3 Release Process for Part Of Maven

1 Prepare your POMs for release:

   a Make sure there are **no** snapshots in the POMs to be released.

   b Check that your POMs will not lose content when they are rewritten during the release process.

   - Verify that all `pom.xml` files have an SCM definition.
   - `mvn release:prepare -DdryRun=true`
   - Diff the original file `pom.xml` with the one called `pom.xml.tag` to see if the license or any other info has been removed. This has been known to happen if the starting `<project>` tag is **not** on a single line. The only things that should be different between these files are the `<version>` and `<scm>` elements. Any other changes, you must backport yourself to the original `pom.xml` file and commit before proceeding with the release.

2 Publish a snapshot:

```
>mvn deploy
...
[INFO] [deploy:deploy]
[INFO] Retrieving previous build number from apache.snapshots.https
...
```

If you experience an error during deployment like a HTTP 401 check your settings for the required server entries as outlined in the Prerequisites.

Note: Be sure that the generated artifacts respect the Apache release rules: NOTICE and LICENSE files should be present in the META-INF directory within the jar. For `-sources` artifacts, be sure that your POM does **not** use the maven-source-plugin:2.0.3 which is broken. The recommended version at this time is 2.0.4.

Note: You should verify the deployment under Maven Snapshot repository on Apache.

```
https://repository.apache.org/content/repositories/snapshots/org/apache/maven/pl
```

3 Prepare the release

```
mvn release:clean
mvn release:prepare
```

Note: Preparing the release will create the new tag in SVN, automatically checking in on your behalf.

4 Stage the release for a vote

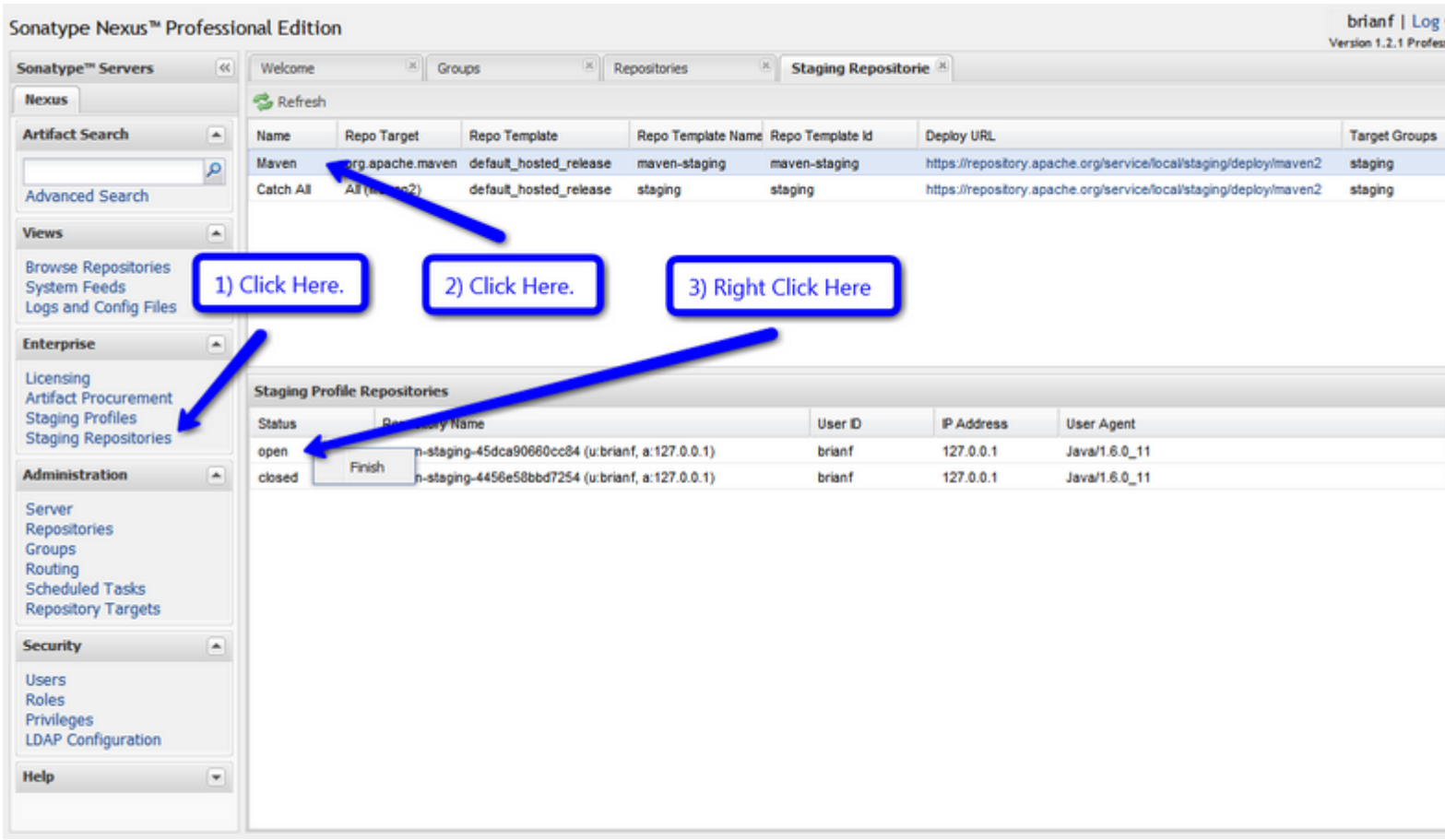
```
mvn release:perform
```

The release will automatically be inserted into a temporary staging repository for you. See the Nexus staging documentation for full details.

5 Close the staging repository

Login to `https://repository.apache.org` using your Apache SVN credentials. Click on "Staging". Then click on "org.apache.maven" in the list of repositories. In the panel below you should see an open repository that is linked to your username and ip. Right click on this repository and select "Close". This will close the repository from future deployments and make it available for others to view. If you are staging multiple releases together, skip this step until you have staged everything.

See the image below for details.

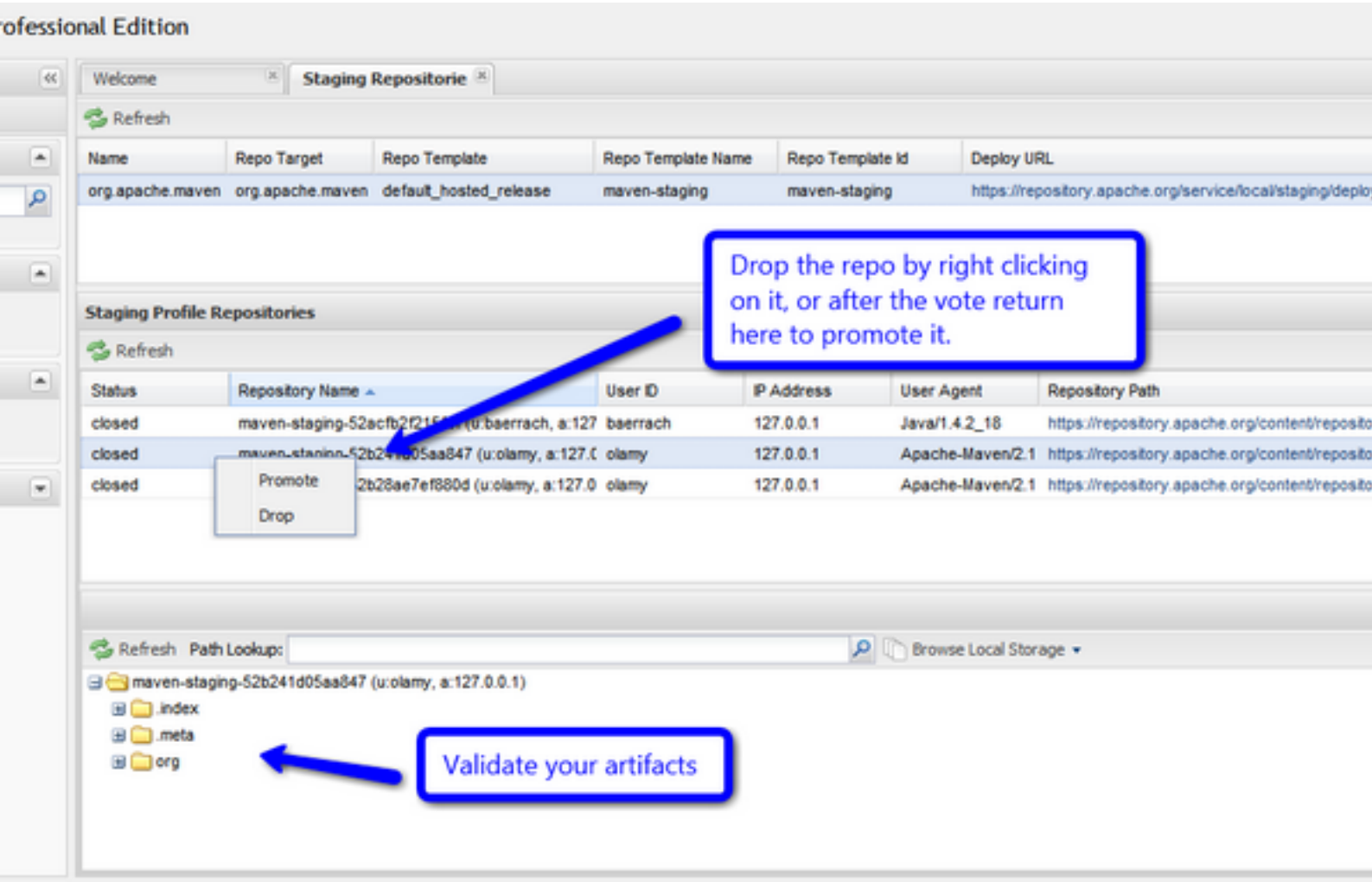


Closing the repository

6 Verify the Staged artifacts

If you click on your repository, a tree view will appear below. You can then browse the contents to ensure the artifacts are as you expect them. Pay particular attention to the existence of `*.asc` (signature) files. If the you don't like the content of the repository, right click your repository and choose "Drop". You can then rollback your release and repeat the process.

Note the repository URL, you will need this in your vote email.



Validating the artifacts

7 Stage the latest documentation

The plugin parent POM is configured to stage the documentation in a "versioned" directory such as `/plugins/maven-XXX-plugin-Y.Z`.

a Stage the documentation for the current release version (not the new snapshot).

```
cd target/checkout
mvn site:stage-deploy -Preporting
```

Note: It requires Maven 2.1.0 or higher to successfully deploy to `people.apache.org` via SSH. Older Maven versions will fail due to `com.jcraft.jsch.JSchException: Algorithm negotiation fail`.

Note: You should verify the deployment of the site on the Maven website (you need to wait the sync).

```
http://maven.apache.org/plugins/maven-XXX-plugin-Y.Z/
```

Some developers have reported problems with the `site:stage-deploy` goal. In that case, you can stage the site locally and upload it manually:

```
mvn site:stage -Preporting
scp -r target/staging/people.apache.org/www/maven.apache.org/plugins/maven-x
```

b Verify/change folder permissions to 0775 and files permissions to 0664. Log on to `people.apache.org` and change to the directory above the staging directory. That would be `/www/maven.apache.org/plugins` for a plugin. Then run these commands:

```
find . -type d -exec chmod a+rx,g+w {} \;
find . -type f -exec chmod 664 {} \;
```

8 Propose a vote on the dev list with the closed issues, the issues left, the staging repository and the staging site. For instance:

```
To: "Maven Developers List" <dev@maven.apache.org>
Subject: [VOTE] Release Maven XXX plugin version Y.Z
Hi,
We solved N issues:
http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=XXX&styleName=Html&ve
There are still a couple of issues left in JIRA:
http://jira.codehaus.org/secure/IssueNavigator.jspa?reset=true&pid=XXX&status=1
Staging repo:
https://repository.apache.org/content/repositories/maven-staging-[YOUR REPOSITOR
Staging site:
http://maven.apache.org/plugins/maven-XXX-plugin-Y.Z/
Guide to testing staged releases:
http://maven.apache.org/guides/development/guide-testing-releases.html
Vote open for 72 hours.
[ ] +1
[ ] +0
[ ] -1
```

To get the JIRA release notes link, browse to the plugin's JIRA page, select the *Road Map* link, and use the link to *Release Notes* that is next to the version being released.

To get the list of issues left in JIRA, browse to the plugin's JIRA page, and from the *Preset Filters* on the right, use the link for *Outstanding* issues.

9 Check the vote results

Copied from Votes on Package Releases.

```
Votes on whether a package is ready to be released follow a format similar to ma
-- except that the decision is officially determined solely by whether at least
+1 votes were registered. Releases may not be vetoed. Generally the community wi
vote to release if anyone identifies serious problems, but in most cases the ult
once three or more positive votes have been garnered, lies with the individual s
release manager. The specifics of the process may vary from project to project,
'minimum of three +1 votes' rule is universal.
```

The list of binding voters is available at http://people.apache.org/~jim/projects.html#maven-pmc

Once a vote is successful, post the result to the dev list and cc the PMC. For instance:

```
To: "Maven Developers List" <dev@maven.apache.org>
CC: "Maven Project Management Committee List" <private@maven.apache.org>
Subject: [RESULT] [VOTE] Release Maven XXX plugin version Y.Z
Hi,
The vote has passed with the following result :
+1 (binding): <<list of names>>
+1 (non binding): <<list of names>>
I will promote the artifacts to the central repo.
```

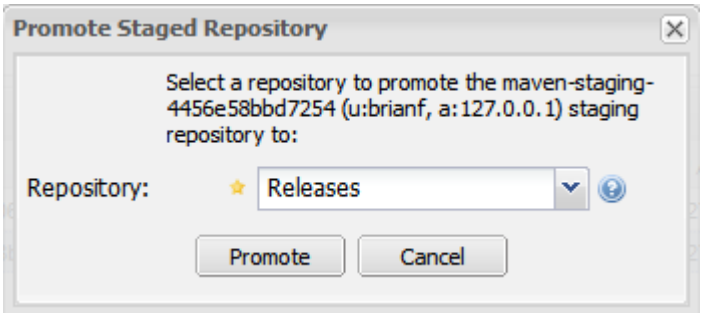
If the vote is unsuccessful, the process will need to be restarted. Be sure to drop your staging repository as described above.

10 Promote the release

Once the release is deemed fit for public consumption it can be transfered to a production repository where it will be available to all users.

Login to `https://repository.apache.org` with your Apache SVN credentials. Click on "Staging" and then on the repository with id "maven-staging". Find your closed staging repository, right click on it and choose "Promote". Select the "Releases" repository and click "Promote".

See image below.



Promote the artifacts

Next click on "Repositories", select the "Releases" repository and validate that your artifacts exist as you expect them.

11 Deploy the current website

Note: Be sure to generate and deploy the site using the same version of the release. Typically, you need to check out the tag (or go to `target/checkout`)

```
cd target\checkout
mvn site-deploy -Preporting
```

Note: You can not just copy the documentation from Step 7 into the released documentation as the links are not identical. See the email thread http://www.nabble.com/forum/ViewPost.jtp?post=24018250&framed=y

12 Review the website

Wait for the files to arrive at

```
http://maven.apache.org/plugins/maven-XXX-plugin
```

or for a shared component at

```
http://maven.apache.org/shared/maven-XXX
```

The wait is necessary to allow the site to be rsync'ed into production.

13 Update the maven site

Check out the maven site project: `https://svn.apache.org/repos/asf/maven/site/trunk`

If this is a plugin release, update the version number for the plugin on the *src/site/apt/plugins/index.apt* page.

If this is a shared component release, update the version number for the component on the *src/site/apt/shared/index.apt* page.

Commit your changes.

14 Update JIRA

For a plugin, go to Admin section in JIRA for the `maven-XXX-plugin` project and mark the `Y.Z` version as released. Create version `Y.Z+1`, if that hasn't already been done.

If this is a shared component, go to Admin section in JIRA for the MSHARED project and mark the `maven-XXX-Y.Z` version as released. Create version `maven-XXX-Y.Z+1`, if that hasn't already been done.

15 Create an announcement. For instance:

Note: You must send this email from your apache email account, e.g. YOUR_APACHE_USERNAME@apache.org otherwise the email to announce@maven.apache.org will bounce.

```
From: YOUR_APACHE_USERNAME@apache.org
To: announce@maven.apache.org, users@maven.apache.org
Cc: dev@maven.apache.org
Subject: [ANN] Maven XXX Plugin Y.Z Released
The Maven team is pleased to announce the release of the Maven XXX Plugin, versi
This plugin (insert short description of the plugin's purpose).
http://maven.apache.org/plugins/maven-XXX-plugin/
You should specify the version in your project's plugin configuration:
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-XXX-plugin</artifactId>
 <version>Y.Z</version>
</plugin>
Release Notes - Maven 2.x XXX Plugin - Version Y.Z
(Copy Here Release Notes in Text Format from JIRA)
Enjoy,
-The Maven team
```

16 Add the release to the next board report, in the private subversion area.

This is a PMC activity. If you are not a PMC member then email the `"Maven Developers List" <dev@maven.apache.org>` with a request to update this file for your release.

```
From: YOUR_APACHE_USERNAME@apache.org
To: dev@maven.apache.org
Subject: [PMC] Maven XXX Plugin Y.Z needs adding to board report
The Maven XXX Plugin has been released.
Can this get added to the next board report please.
```

17 Add the release to the wiki, under the *Recent Releases* section of the front page and on the Releases page.

Note: If you don't have access to edit this page email "Jason van Zyl" <jason@maven.org>

18 Celebrate :o)

81.2 Trouble Shooting

81.2.1 mvn release:prepare "commit failed" during `Prepare the release`

If you get an error message similar to:

```
    [INFO] Unable to tag SCM
    Provider message:
    The svn tag command failed.
    Command output:
    svn: Commit failed (details follow):
    svn: File '/repos/asf/maven/plugins/tags/maven-eclipse-plugin-2.7/src/main/java
```

Then the resolution is to use a Subversion client 1.6+ and to run `svn update`.

81.2.2 mvn release:prepare "commit failed" during `Prepare the release`

If you get an error message similar to:

```
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Unable to tag SCM
Provider message:
The svn tag command failed.
Command output:
svn: Path 'https://svn.apache.org/repos/asf/maven/plugins/tags/maven-eclipse-plugin
```

Then the resolution is to delete the tag using `svn del -m "re-releasing build" <svn path>`

82 Deploy Maven Current References

82.1 Introduction

This document gives step-by-step instructions for deploying the Maven current references. The primary audience is Maven PMC.

82.2 Prerequisite

Be sure that:

- you have all Maven servers defined in your settings.xml. For more information, please refer to Committer settings.
- you have created your GPG keys. For more information, please refer to Making GPG Keys.

82.3 Deploy Maven Current References

1 Do a fresh check out of a release tag of Maven, for example:

```
$ svn checkout https://svn.apache.org/repos/asf/maven/maven-2/tags/maven-2.2.0 m
```

2 Execute the site goal for the *maven-2.2.0* project

```
maven-2.2.0$ mvn site -Preporting
```

3 Verify the documentation before deploying

You could also use the *stage* goal of the Maven Site Plugin to verify the site output. For instance:

```
maven-2.2.0$ mvn site:stage -Preporting -DstagingDirectory=/tmp/maven-2.2.0
```

4 Deploy to people.apache.org

```
maven-2.2.0$ mvn site-deploy -Preporting
```

It will create a new folder *2.2.0* in */www/maven.apache.org/ref/* on the Apache server.

Note: It will take an hour or so to sync.

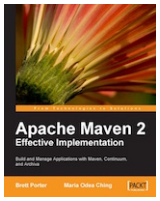
5 Update the current ref symlink to point the last Maven release

- Connect you to *people.apache.org* and go to */www/maven.apache.org/ref/* folder.
- Create a symlink from *2.2.0* folder to *current* if it is the newest stable release of Maven:

```
ref$ ln -s 2.2.0/ ./current
```

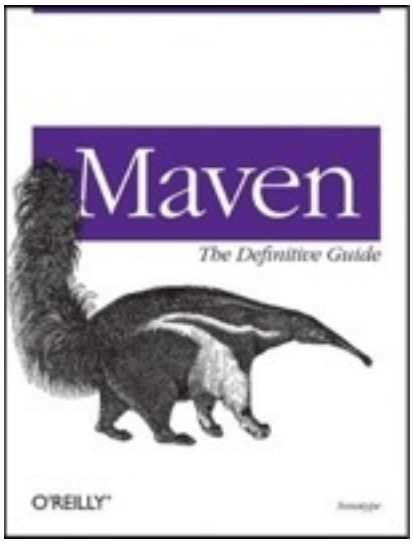
83 External Resources

83.1 Books on Maven



Apache Maven 2: Effective Implementation

- **Covers:** Maven 2.0.9, 2.2.1, and above
- **Published:** Packt Publishing (September 15, 2009)
- **Authors:** Brett Porter, Maria Odea Ching
- **Buy the Book:** Packt; Amazon

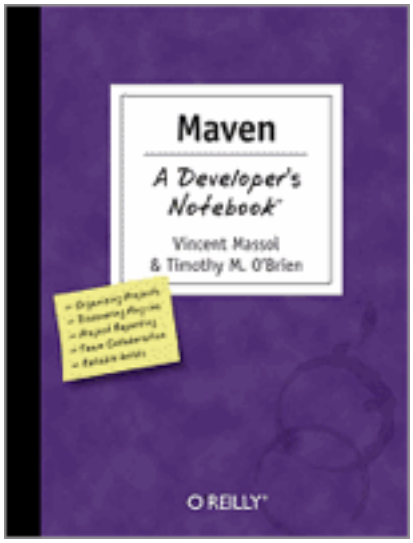


Maven: The Definitive Guide (Readable HTML and Free PDF Download)

- **Covers:** Maven 2.0.9+
- **Published:** O'Reilly (Edition 1: October 1, 2008)
- **Authors:** Sonatype (Jason van Zyl, Brian Fox, John Casey, Bruce Snyder, Tim O'Brien, Eric Redmond)
- **Read Online:** http://www.sonatype.com/books/maven-book/
- **Buy the Book:** Amazon

Better Builds with Maven (Free PDF Download)

- **Covers:**Maven 2.0.4
- **Published:** MaestroDev (March 2006)
- **Authors:** John Casey, Vincent Massol, Brett Porter, Carlos Sanchez
- **Read Online:** http://www.maestrodev.com/better-build-maven



Maven: A Developer's Notebook

- **Covers:**Maven 1.0.2
- **Published:** O'Reilly (July 2005)
- **Authors:** Vincent Massol, Tim O'Brien

83.2 Miscellaneous on Maven

If you're interested in testing your Maven skills, check out JavaBlackBelt's Maven exam. This exam is being written collaboratively by the community. Feel free to add new questions, suggest improvements, etc.

83.3 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.

Title	Publisher	Author	Published
Create a Customized Build Process in Maven		John Casey	August 2009
Maven: mas que una herramienta de construccion (in Spanish)		Manuel Recena	June 2009
Introduction to m2eclipse	TheServerSide	Tim O'Brien, Bruce Snyder, Eugene Kuleshov	July 2008
Maven 2.x (in Turkish)	Anadolu Üniversitesi	Mustafa Sait Özen	August 2007
Setting up the Internal Repository	The Server Side	Avneet Mangat	June 2007
Building Web Applications with Maven 2	java.net	Will Iverson	1 March 2007
Introduction to Apache Maven 2	developerWorks	Sing Li	19 December 2006
Maven - Menos mal que has venido (in Spanish)	Universidad de Sevilla	Manuel J. Recena Soto	6 November 2006
FAQ for Maven 2 and Continuum (in French)	Developpez.com	Eric Reboisson	11 October 2006
Keep Your Maven Projects Portable Throughout the Build Cycle	DevX	Eric Redmond	8 September 2006
Automation for the people: Choosing a Continuous Integration server	deverloperWorks	Paul Duvall	5 September 2006
Java Posse #070 - Interview with Brett Porter of Maven	Java Posse	Tor Norbye, Carl Quinn, Dick Wall, Joe Nuxoll, Brett Porter	18 July 2006
Continuous Integration with Continuum	Java.net	John Ferguson Smart	30 May 2006
The Maven 2 POM demystified	JavaWorld	Eric Redmond	29 May 2006
Maven: Building Complex Systems	Dr.Dobb's	Gigi Sayfan	21 April 2006
Working with maven 2	PeopleWare	Jan Dockx	13 April 2006
Maven 2.0: Compile, Test, Run, Deploy, and More	onjava	Chris Hardin	29 March 2006
Descripcion tecnica de Maven (in Spanish)	Metaware Inc	Juan Pablo Santos Rodríguez	13 March 2006

Get the most out of Maven 2 site generation	JavaWorld	John Ferguson Smart	27 February 2006
An introduction to Maven 2 (in french)	Developpez.com	Denis Cabasson	27 January 2006
Maven 2.0 - Javapolis 2005		Vincent Massol	15 December 2005
An introduction to Maven 2	JavaWorld	John Ferguson Smart	5 December 2005
Taking the Maven 2 Plunge		David DeWolf	1 October 2005
Building J2EE Projects with Maven	OnJava	Vincent Massol	7 September 2005
Maven 2.0 and Continuum SJUG Presentation		Brett Porter	1 June 2005
Exploiting Maven in Eclipse	developerWorks	Gilles Dodinet	24 May 2005
Managing WebSphere Portal V5.1 projects with Apache Maven and Rational Application Developer 6.0	developerWorks	Hinrich Boog	30 March 2005
Maven 1.0 Javapolis Presentation		Vincent Massol	16 December 2004
Master and Commander by Julien Dubois	Oracle	Julien Dubois	November 2004
installing and working with Maven (in German)		Manfred Wolff	August 2004
Apache's Maven Comes of Age (Coverage of the release of Maven 1.0)	internetnews.com	Sean Michael Kerner	15 July 2004
Extending Maven Through Plugins by Eric Pugh	OnJava	Eric Pugh	17 March 2004
Maven Magic - a tutorial on Maven and J2EE projects.	TheServerSide	Srikanth Shenoy	November 2003
Developing with Maven by Rob Herbst	OnJava	Rob Herbst	22 October 2003
Apache Maven Simplifies the Java Build Process Even More Than Ant	DevX	Dave Ford	2 September 2003
Building J2EE applications with Maven (Slides from TheServerSide Symposium)	TheServerSide	Vincent Massol	27 June 2003

Maven ties together tools for better code management	JavaWorld	Jeff Linwood	11 October 2002
How to get Maven to build your web service into a WAR on AstroGrid	Astrogrid		
Some Maven FAQs on AstroGrid	Astrogrid		
Some Useful Maven Notes on AstroGrid	Astrogrid		
A tutorial for Maven, J2EE projects, and MevenIDE (in Portuguese).			