Simple guide to Maven

Maven builds are controlled by Project Object Model (POM) files. POM files, unlike Ant scripts, describe the project. A POM file doesn’t have code or task steps, it just describes the structure of the project. It’s up to Maven to take that description and use it to build the project.

Maven makes a lot of assumptions. Just about the simplest POM file you could have might look like this:

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany</groupId>

<artifactId>my-project</artifactId>

<version>1.0-SNAPSHOT</version>

</project>

This little file would compile Java code, copy resources, generate a documentation web site and assemble a JAR file. What does it say exactly?

* modelVersion - The Maven POM version number. It’s the version number of the POM specification
* groupId - The project’s organizational group
* artifactId - The name of the artifact the project creates
* version - The version number of the artifact created

**Maven manages dependencies between projects. If your project needs a jar file like Log4J, you’ll add that dependency to your POM file and Maven will automatically fetch that jar file and make it available when the code is compiled**.

How does Maven know how to build your code from just that little project file? It makes even more assumptions. It assumes that your project directory structure looks like this:

/src

/main

/java - All your Java source files, e.g., /com/mycompany/Main.java

/resources - Resource files that need to end up in the project JAR

/webapp - Web files if you’re building a WAR

/test

/java - All your test Java source files

/resources - Resource files needed for testing

/target - Everything generated by the build, include the JAR/WAR file

There are additional directories. All of them together are known as the Standard Directory Layout.

Let’s say that your project stores Java source code in ‘/src/java’ and puts the Java unit test code in ‘/src/test’. Then your POM file would look like this:

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany</groupId>

<artifactId>my-project</artifactId>

<version>1.0-SNAPSHOT</version>

<build>

<sourceDirectory>src/java</sourceDirectory>

<testSourceDirectory>src/test</testSourceDirectory>

</build>

</project>

With just these few elements, Maven supports many “goals”. These goals are akin to Ant tasks and describe what you want Maven to do. Some common goals:

* clean - remove any old build files
* compile - compile the code
* test - compile the code and run the unit tests
* package - compile and test the code and create any build products
* site - generate site documentation

Maven components are stored in repositories. You can define as many repositories as you like, but you can’t install your own JAR files or other build products in the default Maven repositories. You have to create and host your own.

Let’s look at a POM file with a dependency.

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany</groupId>

<artifactId>my-project</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.13</version>

</dependency>

</dependencies>

</project>

This POM file will cause Maven to download the Log4J 1.2.13 component and make it available on the classpath while compiling the code. If you were creating a WAR file, EAR file, etc. it would also copy the jar file into the ‘lib’ directory of the final package. You can add as many dependencies as you like and, optionally, give them a scope. The scope tells Maven whether the dependency is only necessary when testing the code or only necessary to compile but not for packaging.

**Conclusion:**

**What it does:**

Maven is a "build management tool", it is for defining how your .java files get compiled to .class, packaged into .jar (or .war or .ear) files, (pre/post)processed with tools, managing your [CLASSPATH](http://en.wikipedia.org/wiki/Classpath_(Java)), and all others sorts of tasks that are required to build your project. It is similar to [Apache Ant](http://ant.apache.org/) or [Gradle](http://www.gradle.org/) or [Makefiles](http://en.wikipedia.org/wiki/Makefiles) in C/C++, but it attempts to be completely self-contained in it that you shouldn't need any additional tools or scripts by incorporating other common tasks like downloading & installing necessary libraries etc.

It is also designed around the "build portability" theme, so that you don't get issues as having the same code with the same buildscript working on one computer but not on another one (this is a known issue, we have VMs of Windows 98 machines since we couldn't get some of our Delphi applications compiling anywhere else). Because of this, it is also the best way to work on a project between people who use different IDEs since IDE-generated Ant scripts are hard to import into other IDEs, but all IDEs nowadays understand and support Maven ([IntelliJ](http://www.jetbrains.com/idea/features/ant_maven.html), [Eclipse](http://www.eclipse.org/m2e/), and [NetBeans](http://wiki.netbeans.org/Maven)). Even if you don't end up liking Maven, it ends up being the point of reference for all other modern builds tools.

**Why you should use it**

There are three things about Maven that are *very* nice.

1. Maven will (after you declare which ones you are using) download all the libraries that you use *and* the libraries that *they* use for you automatically. This is very nice, and makes dealing with lots of libraries ridiculously easy. This lets you avoid ["dependency hell"](http://en.wikipedia.org/wiki/Dependency_hell). It is similar to Apache Ant's [Ivy](http://ant.apache.org/ivy/).
2. It uses "[Convention over Configuration](http://en.wikipedia.org/wiki/Convention_over_configuration)" so that by default you don't need to define the tasks you want to do. You don't need to write a "compile", "test", "package", or "clean" step like you would have to do in Ant or a Makefile. Just put the files in the places in which Maven expects them and it should work off of the bat.
3. Maven also has lots of nice plug-ins that you can install that will handle many routine tasks from [generating Java classes from an XSD schema using JAXB](https://java.net/projects/maven-jaxb2-plugin/pages/Home) to [measuring test coverage with Cobertura](http://mojo.codehaus.org/cobertura-maven-plugin/). Just add them to your pom.xml and they will integrate with everything else you want to do.

The initial learning curve is steep, but (nearly) every professional Java developer uses Maven or wishes they did. You should use Maven on every project although don't be surprised if it takes you a while to get used to it and that sometimes you wish you could just do things manually, since learning something new sometimes hurts. However, once you truly get used to Maven you will find that build management takes almost no time at all.

**How to Start**

The best place to start is "[Maven in 5 Minutes](http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)". It will get you start with a project ready for you to code in with all the necessary files and folders set-up (yes, I recommend using the quickstart archetype, at least at first).

After you get started you'll want a better understanding over how the tool is intended to be used. For that "[Better Builds with Maven](https://web.archive.org/web/20130903151814/http:/www.maestrodev.com/wp-content/uploads/2012/03/betterbuildswithmaven-2008.pdf)" is the most thorough place to understand the guts of how it works, however, "[Maven: The Complete Reference](http://www.sonatype.com/books/mvnref-book/reference/)" is more up-to-date. Read the first one for understanding, but then use the second one for reference.