**Documentation**

**Getting Started with Maven**

* [Getting Started in 5 Minutes](http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)
* [Getting Started in 30 Minutes](http://maven.apache.org/guides/getting-started/index.html)

**Introductions**

* [The Build Lifecycle](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html)
* [The POM](http://maven.apache.org/guides/introduction/introduction-to-the-pom.html)
* [Profiles](http://maven.apache.org/guides/introduction/introduction-to-profiles.html)
* [Repositories](http://maven.apache.org/guides/introduction/introduction-to-repositories.html)
* [Standard Directory Layout](http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html)
* [The Dependency Mechanism](http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html)
* [Optional Dependencies and Dependency Exclusions](http://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html)

**Plugins**

* [Plugin Development](http://maven.apache.org/guides/introduction/introduction-to-plugins.html)
* [Configuring Plug-ins](http://maven.apache.org/guides/mini/guide-configuring-plugins.html)
* [Plugin Prefix Resolution](http://maven.apache.org/guides/introduction/introduction-to-plugin-prefix-mapping.html)
* [Developing Java Plugins](http://maven.apache.org/guides/plugin/guide-java-plugin-development.html)

**Site**

* [Creating a Site](http://maven.apache.org/guides/mini/guide-site.html)
* [The APT Format](http://maven.apache.org/doxia/references/apt-format.html)
* [Snippet Macro](http://maven.apache.org/guides/mini/guide-snippet-macro.html)

**Archetypes**

* [What is an Archetype](http://maven.apache.org/guides/introduction/introduction-to-archetypes.html)
* [Creating Archetypes](http://maven.apache.org/guides/mini/guide-creating-archetypes.html)

**Upgrading**

* [Relocation of Artifacts](http://maven.apache.org/guides/mini/guide-relocation.html)

**Repositories**

* [Installing 3rd party JARs to Local Repository](http://maven.apache.org/guides/mini/guide-3rd-party-jars-local.html)
* [Deploying 3rd party JARs to Remote Repository](http://maven.apache.org/guides/mini/guide-3rd-party-jars-remote.html)
* [Coping with Sun JARs](http://maven.apache.org/guides/mini/guide-coping-with-sun-jars.html)
* [Remote repository access through authenticated HTTPS](http://maven.apache.org/guides/mini/guide-repository-ssl.html)

**Guides**

* [Creating Assemblies](http://maven.apache.org/guides/mini/guide-assemblies.html)
* [Configuring Archive Plugins](http://maven.apache.org/guides/mini/guide-archive-configuration.html)
* [Configuring Maven](http://maven.apache.org/guides/mini/guide-configuring-maven.html)
* [Mirror Settings](http://maven.apache.org/guides/mini/guide-mirror-settings.html)
* [Deployment and Security Settings](http://maven.apache.org/guides/mini/guide-deployment-security-settings.html)
* [Generating Sources](http://maven.apache.org/guides/mini/guide-generating-sources.html)
* [Working with Manifests](http://maven.apache.org/guides/mini/guide-manifest.html)
* [Maven Classloading](http://maven.apache.org/guides/mini/guide-maven-classloading.html)
* [Using Multiple Modules in a Build](http://maven.apache.org/guides/mini/guide-multiple-modules.html)
* [Using Multiple Repositories](http://maven.apache.org/guides/mini/guide-multiple-repositories.html)
* [Using Proxies](http://maven.apache.org/guides/mini/guide-proxies.html)
* [Using the Release Plugin](http://maven.apache.org/guides/mini/guide-releasing.html)
* [Using Ant with Maven](http://maven.apache.org/guides/mini/guide-using-ant.html)
* [Using Modello](http://maven.apache.org/guides/mini/guide-using-modello.html)
* [Using Extensions](http://maven.apache.org/guides/mini/guide-using-extensions.html)
* [Building For Different Environments with Maven 2](http://maven.apache.org/guides/mini/guide-building-for-different-environments.html)
* [Using Toolchains](http://maven.apache.org/guides/mini/guide-using-toolchains.html)
* [Encrypting passwords in settings.xml](http://maven.apache.org/guides/mini/guide-encryption.html)
* [Guide to HTTP Connection Settings](http://maven.apache.org/guides/mini/guide-http-settings.html)
* [Guide to Selecting Alternative Wagon Providers](http://maven.apache.org/guides/mini/guide-wagon-providers.html)
* [Guide to Building JDK 1.4 Projects Using JDK 1.5](http://maven.apache.org/guides/mini/guide-building-jdk14-on-jdk15.html)
* [Guide to Configuring Default Mojo Executions](http://maven.apache.org/guides/mini/guide-default-execution-ids.html)

**Maven Tools and IDE Integration**

* [Maven Auto-Completion Using BASH](http://maven.apache.org/guides/mini/guide-bash-m2-completion.html)

**Development Guides**

* [Building Maven from Scratch](http://maven.apache.org/guides/development/guide-building-maven.html)
* [Developing Maven](http://maven.apache.org/guides/development/guide-maven-development.html)
* [The Plugin Documentation Standard](http://maven.apache.org/guides/development/guide-plugin-documentation.html)
* [Maven Documentation Style](http://maven.apache.org/guides/development/guide-documentation-style.html)

**The Maven Community**

* [The Maven Community](http://maven.apache.org/community.html)
* [Helping with Maven](http://maven.apache.org/guides/development/guide-helping.html)
* [Guide for New Committers](http://maven.apache.org/guides/mini/guide-new-committers.html)
* [Testing Development Versions of Plugins](http://maven.apache.org/guides/development/guide-testing-development-plugins.html)
* [3rd Party Resources](http://maven.apache.org/articles.html)

**Conventions**

* [Maven Conventions](http://maven.apache.org/maven-conventions.html)
* [Naming Conventions](http://maven.apache.org/guides/mini/guide-naming-conventions.html)
* [When You Can't Use the Conventions](http://maven.apache.org/guides/mini/guide-using-one-source-directory.html)

**The Central Repository**

* [Uploading Artifacts to the Central Repository](http://maven.apache.org/guides/mini/guide-central-repository-upload.html)
* [Improving the Repository](http://maven.apache.org/guides/mini/guide-maven-evangelism.html)

**References**

* [POM Overview](http://maven.apache.org/pom.html) ([Technical Project Descriptor](http://maven.apache.org/ref/current/maven-model/maven.html))
* [Settings Overview](http://maven.apache.org/settings.html) ([Technical Settings Descriptor](http://maven.apache.org/ref/current/maven-settings/settings.html))
* [Core Plug-ins List](http://maven.apache.org/plugins/index.html)
* [Mojo API](http://maven.apache.org/developers/mojo-api-specification.html)
* [Glossary](http://maven.apache.org/glossary.html)
* [Maven Quick Reference Card - PDF](http://maven.apache.org/guides/MavenQuickReferenceCard.pdf)

**Javadoc API**

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

* [Maven Artifact](http://maven.apache.org/ref/current/maven-artifact/apidocs/)
* [Maven Reporting](http://maven.apache.org/shared/maven-reporting-api/apidocs/)
* [Maven Plugin API](http://maven.apache.org/ref/current/maven-plugin-api/apidocs/)
* [Maven Model](http://maven.apache.org/ref/current/maven-model/apidocs/)
* [Maven Core](http://maven.apache.org/ref/current/maven-core/apidocs/)
* [Maven Settings](http://maven.apache.org/ref/current/maven-settings/apidocs/)

You could also browse the [full technical documentation references](http://maven.apache.org/ref/current/) of the current version of Maven.

**Maven in 5 Minutes**

**Prerequisites**

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

**Installation**

*Maven is a Java tool, so you must have*[*Java*](http://www.oracle.com/technetwork/java/javase/downloads/index.html)*installed in order to proceed.*

First, [download Maven](http://maven.apache.org/download.html) and follow the [installation instructions](http://maven.apache.org/download.html#Installation). After that, type the following in a terminal or in a command prompt:

1. mvn –-version 或mvn –v

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

1. Apache Maven 3.0.5 (r01de14724cdef164cd33c7c8c2fe155faf9602da; 2013-02-19 14:51:28+0100)
2. Maven home: D:\apache-maven-3.0.5\bin\..
3. Java version: 1.6.0\_25, vendor: Sun Microsystems Inc.
4. Java home: C:\Program Files\Java\jdk1.6.0\_25\jre
5. Default locale: nl\_NL, platform encoding: Cp1252
6. OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"

Depending upon your network setup, you may require extra configuration. Check out the [Guide to Configuring Maven](http://maven.apache.org/guides/mini/guide-configuring-maven.html) if necessary.

**If you are using Windows, you should look at** [Windows Prerequisites](http://maven.apache.org/guides/getting-started/windows-prerequisites.html) **to ensure that you are prepared to use Maven on Windows.**

**Creating a Project**

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

1. mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

groupId 组织名

aritfactId 项目名

archetypeArtifactId 项目类型

interactiveMode 交互模式

如果直接执行mvn init，则会进入交互模式，然后在交互模式中指定组织名、项目名等

*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（本地仓库，可以自己配置） You may also need to execute the command a couple of times before it succeeds. This is because the remote server may time out before your downloads are complete. Don't worry, there are ways to fix that.*

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

1. cd my-app

Under this directory you will notice the following [standard project structure](http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html).

1. my-app
2. |-- pom.xml
3. `-- src
4. |-- main
5. | `-- java
6. | `-- com
7. | `-- mycompany
8. | `-- app
9. | `-- App.java
10. `-- test
11. `-- java
12. `-- com
13. `-- mycompany
14. `-- app
15. `-- 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 file is the project's Project Object Model, or POM.

**The POM**

The pom.xml file is the core of a project's configuration in Maven. It is a single configuration file（me：单一的配置文件） 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（me：pom配置文件会是很复杂吗，使人望而却步）, but it is not necessary to understand all of the intricacies（me：错综复杂的） just yet to use it effectively. This project's POM is:

1. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3. <modelVersion>4.0.0</modelVersion><!—描述的是这个pom遵循的版本 -->
5. <groupId>com.mycompany.app</groupId><!—组织名 -->
6. <artifactId>my-app</artifactId><!—项目，可以被其他人依赖，类似gradle中工程之间项目依赖时指定的项目名 -->
7. <version>1.0-SNAPSHOT</version><!—项目的版本 -->
8. <packaging>jar</packaging><!—指定项目的打包方式 -->
10. <name>Maven Quick Start Archetype</name>
11. <url>http://maven.apache.org</url>
13. <dependencies><!—定义项目的依赖 -->
14. <dependency>
15. <groupId>junit</groupId>
16. <artifactId>junit</artifactId>
17. <version>4.8.2</version>
18. <scope>test</scope>
19. </dependency>
20. </dependencies>
21. </project>

**What did I just do?**

You executed the Maven goal *archetype:generate*, and passed in various parameters to that goal. The prefix *archetype* is the [plugin](http://maven.apache.org/plugins/index.html) that contains the goal. If you are familiar with [Ant](http://ant.apache.org/), you may conceive 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".

**Build the Project**

1. mvn package

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

1. ...
2. [INFO] ------------------------------------------------------------------------
3. [INFO] BUILD SUCCESSFUL
4. [INFO] ------------------------------------------------------------------------
5. [INFO] Total time: 2 seconds
6. [INFO] Finished at: Thu Jul 07 21:34:52 CEST 2011
7. [INFO] Final Memory: 3M/6M
8. [INFO] ------------------------------------------------------------------------

Unlike the first command executed (*archetype:generate*) 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](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html), which is an ordered sequence of phases（阶段是build声明周期中的一个部分，声明周期是由系列的这些个阶段组成的）. 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（执行compile阶段实际上是按照以下的步骤执行的构建工作：校验-生成源-处理源-生成资源-处理资源-编译）:

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:

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

Which will print the quintessential:

1. Hello World!

**Running Maven Tools**

**Maven Phases**

Although hardly a comprehensive list（复杂的表单）, 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（这些测试不需要也不必要放在部署的源代码中或者打包）
* **package**: take the compiled code and package it in its distributable format, such as a JAR.
* **integration-test（集成测试）**: 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（标准）
* **install**: install the package into the local repository, for use as a dependency in other projects locally
* **deploy（部署）**: 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* if the project type is - you guessed it - a WAR.

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

1. 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).

**Generating the Site**

1. 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.

**Conclusion**

We hope this quick overview has piqued your interest in the versatility of Maven. Note that this is a very truncated quick-start guide. Now you are ready for more comprehensive details concerning the actions you have just performed. Check out the [Maven Getting Started Guide](http://maven.apache.org/guides/getting-started/index.html).

**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（独立的参考和解决方案） for common use cases. For first time users, it is recommended that you step through the material in a sequential fashion(顺序的方式). For users more familiar with Maven, this guide endeavours（尽力，竭力） 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](https://maven.apache.org/download.html) 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（合作的，协作的） 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!

**Sections**

* [What is Maven?](https://maven.apache.org/guides/getting-started/index.html#What_is_Maven)
* [How can Maven benefit my development process?](https://maven.apache.org/guides/getting-started/index.html#How_can_Maven_benefit_my_development_process)
* [How do I setup Maven?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_setup_Maven)
* [How do I make my first Maven project?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project)
* [How do I compile my application sources?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_compile_my_application_sources)
* [How do I compile my test sources and run my unit tests?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_compile_my_test_sources_and_run_my_unit_tests)
* [How do I create a JAR and install it in my local repository?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_create_a_JAR_and_install_it_in_my_local_repository)
* [What is a SNAPSHOT version?](https://maven.apache.org/guides/getting-started/index.html#What_is_a_SNAPSHOT_version)
* [How do I use plugins?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_use_plugins)
* [How do I add resources to my JAR?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_add_resources_to_my_JAR)
* [How do I filter resource files?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_filter_resource_files)
* [How do I use external dependencies?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_use_external_dependencies)
* [How do I deploy my jar in my remote repository?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_deploy_my_jar_in_my_remote_repository)
* [How do I create documentation?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_create_documentation)
* [How do I build other types of projects?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_build_other_types_of_projects)
* [How do I build more than one project at once?](https://maven.apache.org/guides/getting-started/index.html#How_do_I_build_more_than_one_project_at_once)

**What is Maven?**

At first glance Maven can appear to be many things, but in a nutshell（一言以蔽之） 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（串行拷贝管理系统）
* Releases
* Distribution

If you want more background information on Maven you can check out [The Philosophy of Maven](https://maven.apache.org/background/philosophy-of-maven.html) and [The History of Maven](https://maven.apache.org/background/history-of-maven.html). Now let's move on to how you, the user, can benefit from using Maven.

**How can Maven benefit my development process?**

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

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

**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](https://maven.apache.org/guides/mini/guide-configuring-maven.html) for more information.

**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（原型机制）. 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（项目的模板） which is combined with some user input to produce a working Maven project that has been tailored（合身的，裁剪讲究的） 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](https://maven.apache.org/guides/introduction/introduction-to-archetypes.html).

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

1. mvn -B archetype:generate \
2. -DarchetypeGroupId=org.apache.maven.archetypes \
3. -DgroupId=com.mycompany.app \
4. -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:

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
6. <groupId>com.mycompany.app</groupId>
7. <artifactId>my-app</artifactId>
8. <packaging>jar</packaging>
9. <version>1.0-SNAPSHOT</version>
10. <name>Maven Quick Start Archetype</name>
11. <url>http://maven.apache.org</url>
12. <dependencies>
13. <dependency>
14. <groupId>junit</groupId>
15. <artifactId>junit</artifactId>
16. <version>4.11</version>
17. <scope>test</scope>
18. </dependency>
19. </dependencies>
20. </project>

pom.xml contains the Project Object Model (POM) for this project（pom.xml包含本项目的项目对象模型）. The POM is the basic unit of work in Maven. This is important to remember because Maven is inherently project-centric（以项目为中心的） 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（一诊室个欧五） 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](https://maven.apache.org/guides/introduction/introduction-to-the-pom.html).

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（指出了本对象模型正在使用的pom的版本）. The version of the model itself changes very infrequently but it is mandatory（该标签是强制的） in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.
* **groupId（项目提供商）** This element indicates the unique identifier of the organization or group that created the project. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example org.apache.maven.plugins is the designated groupId for all Maven plugins.
* **artifactId** This element indicates the unique base name of the primary artifact being generated by this project（该元素指明了项目构建之后的名字，被其他项目依赖的时候可以使用这个名字来引用这个依赖）. 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.本项目的打包类相关，可以是jar/war/ear等). 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（该元素的默认值是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（项目的版本）. 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](https://maven.apache.org/guides/getting-started/index.html#What_is_a_SNAPSHOT_version) and how they work further on in this guide.
* **name** This element indicates the display name used for the project(该元素指明了本项目使用的名字). This is often used in Maven's generated documentation.
* **url** This element indicates where the project's site can be found（表明本项目在哪里可以找到）. 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](https://maven.apache.org/ref/current/maven-model/maven.html). 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:

1. my-app
2. |-- pom.xml
3. `-- src
4. |-- main
5. | `-- java
6. | `-- com
7. | `-- mycompany
8. | `-- app
9. | `-- App.java
10. `-- test
11. `-- java
12. `-- com
13. `-- mycompany
14. `-- app
15. `-- 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](https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html).

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

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

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

1. mvn compile

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

1. [INFO] ----------------------------------------------------------------------------
2. [INFO] Building Maven Quick Start Archetype
3. [INFO] task-segment: [compile]
4. [INFO] ----------------------------------------------------------------------------
5. [INFO] artifact org.apache.maven.plugins:maven-resources-plugin: \
6. checking for updates from central
7. ...
8. [INFO] artifact org.apache.maven.plugins:maven-compiler-plugin: \
9. checking for updates from central
10. ...
11. [INFO] [resources:resources]
12. ...
13. [INFO] [compiler:compile]
14. Compiling 1 source file to <dir>/my-app/target/classes
15. [INFO] ----------------------------------------------------------------------------
16. [INFO] BUILD SUCCESSFUL
17. [INFO] ----------------------------------------------------------------------------
18. [INFO] Total time: 3 minutes 54 seconds
19. [INFO] Finished at: Fri Sep 23 15:48:34 GMT-05:00 2005
20. [INFO] Final Memory: 2M/6M
21. [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（敏锐的观察者 ）, 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](http://ant.apache.org/) to accomplish the same [thing](https://maven.apache.org/ant/build-a1.xml).

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!

**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:

1. mvn test

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

1. [INFO] ----------------------------------------------------------------------------
2. [INFO] Building Maven Quick Start Archetype
3. [INFO] task-segment: [test]
4. [INFO] ----------------------------------------------------------------------------
5. [INFO] artifact org.apache.maven.plugins:maven-surefire-plugin: \
6. checking for updates from central
7. ...
8. [INFO] [resources:resources]
9. [INFO] [compiler:compile]
10. [INFO] Nothing to compile - all classes are up to date
11. [INFO] [resources:testResources]
12. [INFO] [compiler:testCompile]
13. Compiling 1 source file to C:\Test\Maven2\test\my-app\target\test-classes
14. ...
15. [INFO] [surefire:test]
16. [INFO] Setting reports dir: C:\Test\Maven2\test\my-app\target/surefire-reports
18. -------------------------------------------------------
19. T E S T S
20. -------------------------------------------------------
21. [surefire] Running com.mycompany.app.AppTest
22. [surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0 sec
24. Results :
25. [surefire] Tests run: 1, Failures: 0, Errors: 0
27. [INFO] ----------------------------------------------------------------------------
28. [INFO] BUILD SUCCESSFUL
29. [INFO] ----------------------------------------------------------------------------
30. [INFO] Total time: 15 seconds
31. [INFO] Finished at: Thu Oct 06 08:12:17 MDT 2005
32. [INFO] Final Memory: 2M/8M
33. [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:

1. 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 ...

**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:

1. 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 (${user.home}/.m2/repository is the default location). For more information on repositories you can refer to our [Introduction to Repositories](https://maven.apache.org/guides/introduction/introduction-to-repositories.html) but let's move on to installing our artifact! To do so execute the following command:

1. mvn install

Upon executing this command you should see the following output:

1. [INFO] ----------------------------------------------------------------------------
2. [INFO] Building Maven Quick Start Archetype
3. [INFO] task-segment: [install]
4. [INFO] ----------------------------------------------------------------------------
5. [INFO] [resources:resources]
6. [INFO] [compiler:compile]
7. Compiling 1 source file to <dir>/my-app/target/classes
8. [INFO] [resources:testResources]
9. [INFO] [compiler:testCompile]
10. Compiling 1 source file to <dir>/my-app/target/test-classes
11. [INFO] [surefire:test]
12. [INFO] Setting reports dir: <dir>/my-app/target/surefire-reports
14. -------------------------------------------------------
15. T E S T S
16. -------------------------------------------------------
17. [surefire] Running com.mycompany.app.AppTest
18. [surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.001 sec
20. Results :
21. [surefire] Tests run: 1, Failures: 0, Errors: 0
23. [INFO] [jar:jar]
24. [INFO] Building jar: <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar
25. [INFO] [install:install]
26. [INFO] Installing <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar to \
27. <local-repository>/com/mycompany/app/my-app/1.0-SNAPSHOT/my-app-1.0-SNAPSHOT.jar
28. [INFO] ----------------------------------------------------------------------------
29. [INFO] BUILD SUCCESSFUL
30. [INFO] ----------------------------------------------------------------------------
31. [INFO] Total time: 5 seconds
32. [INFO] Finished at: Tue Oct 04 13:20:32 GMT-05:00 2005
33. [INFO] Final Memory: 3M/8M
34. [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](https://maven.apache.org/ant/build-a1.xml) that provides the same functionality that we've achieved thus far you'll notice it's already twice the size of the POM and we're just getting started! There is far more functionality available to you from Maven without requiring any additions to our POM as it currently stands. To get any more functionality out of our example Ant build file you must keep making error-prone additions.

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

1. mvn site

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

1. 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?

1. 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:

1. 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.

**What is a SNAPSHOT version?**

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

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. ...
3. <groupId>...</groupId>
4. <artifactId>my-app</artifactId>
5. ...
6. <version>1.0-SNAPSHOT</version>
7. <name>Maven Quick Start Archetype</name>
8. ...

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

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

During the [release](https://maven.apache.org/plugins/maven-release-plugin/) process, a version of **x.y-SNAPSHOT** changes to **x.y**. The release process also increments the development version to **x.(y+1)-SNAPSHOT**. For example, version **1.0-SNAPSHOT** is released as version **1.0**, and the new development version is version **1.1-SNAPSHOT**.

**How do I use plugins?**

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:

1. ...
2. <build>
3. <plugins>
4. <plugin>
5. <groupId>org.apache.maven.plugins</groupId>
6. <artifactId>maven-compiler-plugin</artifactId>
7. <version>3.3</version>
8. <configuration>
9. <source>1.5</source>
10. <target>1.5</target>
11. </configuration>
12. </plugin>
13. </plugins>
14. </build>
15. ...

You'll notice that all plugins in Maven look much like a dependency - and in some ways they are. This plugin will be automatically downloaded and used - including a specific version if you request it (the default is to use the latest available).

The configuration element applies the given parameters to every goal from the compiler plugin. In the above case, the compiler plugin is already used as part of the build process and this just changes the configuration. It is also possible to add new goals to the process, and configure specific goals. For information on this, see the [Introduction to the Build Lifecycle](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html).

To find out what configuration is available for a plugin, you can see the [Plugins List](https://maven.apache.org/plugins/) and navigate to the plugin and goal you are using. For general information about how to configure the available parameters of a plugin, have a look at the [Guide to Configuring Plugins](https://maven.apache.org/guides/mini/guide-configuring-plugins.html).

**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](https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html), 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.

1. my-app
2. |-- pom.xml
3. `-- src
4. |-- main
5. | |-- java
6. | | `-- com
7. | | `-- mycompany
8. | | `-- app
9. | | `-- App.java
10. | `-- resources
11. | `-- META-INF
12. | `-- application.properties
13. `-- test
14. `-- java
15. `-- com
16. `-- mycompany
17. `-- app
18. `-- 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:

1. |-- META-INF
2. | |-- MANIFEST.MF
3. | |-- application.properties
4. | `-- maven
5. | `-- com.mycompany.app
6. | `-- my-app
7. | |-- pom.properties
8. | `-- pom.xml
9. `-- com
10. `-- mycompany
11. `-- app
12. `-- 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:

1. #Generated by Maven
2. #Tue Oct 04 15:43:21 GMT-05:00 2005
3. version=1.0-SNAPSHOT
4. groupId=com.mycompany.app
5. 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:

1. my-app
2. |-- pom.xml
3. `-- src
4. |-- main
5. | |-- java
6. | | `-- com
7. | | `-- mycompany
8. | | `-- app
9. | | `-- App.java
10. | `-- resources
11. | `-- META-INF
12. | |-- application.properties
13. `-- test
14. |-- java
15. | `-- com
16. | `-- mycompany
17. | `-- app
18. | `-- AppTest.java
19. `-- resources
20. `-- test.properties

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

1. ...
3. // Retrieve resource
4. InputStream is = getClass().getResourceAsStream( "/test.properties" );
6. // Do something with the resource
8. ...

**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:

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>jar</packaging>
12. <name>Maven Quick Start Archetype</name>
13. <url>http://maven.apache.org</url>
15. <dependencies>
16. <dependency>
17. <groupId>junit</groupId>
18. <artifactId>junit</artifactId>
19. <version>4.11</version>
20. <scope>test</scope>
21. </dependency>
22. </dependencies>
24. <build>
25. <resources>
26. <resource>
27. <directory>src/main/resources</directory>
28. <filtering>true</filtering>
29. </resource>
30. </resources>
31. </build>
32. </project>

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

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

To continue our example, let's add a couple of properties to the application.properties file (which we put in the src/main/resources directory) whose values will be supplied when the resource is filtered:

1. # application.properties
2. application.name=${project.name}
3. application.version=${project.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):

1. mvn process-resources

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

1. # application.properties
2. application.name=Maven Quick Start Archetype
3. 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:

1. # filter.properties
2. my.filter.value=hello!

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

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>jar</packaging>
12. <name>Maven Quick Start Archetype</name>
13. <url>http://maven.apache.org</url>
15. <dependencies>
16. <dependency>
17. <groupId>junit</groupId>
18. <artifactId>junit</artifactId>
19. <version>4.11</version>
20. <scope>test</scope>
21. </dependency>
22. </dependencies>
24. <build>
25. <filters>
26. <filter>src/main/filters/filter.properties</filter>
27. </filters>
28. <resources>
29. <resource>
30. <directory>src/main/resources</directory>
31. <filtering>true</filtering>
32. </resource>
33. </resources>
34. </build>
35. </project>

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

1. # application.properties
2. application.name=${project.name}
3. application.version=${project.version}
4. 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):

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>jar</packaging>
12. <name>Maven Quick Start Archetype</name>
13. <url>http://maven.apache.org</url>
15. <dependencies>
16. <dependency>
17. <groupId>junit</groupId>
18. <artifactId>junit</artifactId>
19. <version>4.11</version>
20. <scope>test</scope>
21. </dependency>
22. </dependencies>
24. <build>
25. <resources>
26. <resource>
27. <directory>src/main/resources</directory>
28. <filtering>true</filtering>
29. </resource>
30. </resources>
31. </build>
33. <properties>
34. <my.filter.value>hello</my.filter.value>
35. </properties>
36. </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:

1. # application.properties
2. java.version=${java.version}
3. command.line.prop=${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.

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

**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](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html).

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):

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>jar</packaging>
12. <name>Maven Quick Start Archetype</name>
13. <url>http://maven.apache.org</url>
15. <dependencies>
16. <dependency>
17. <groupId>junit</groupId>
18. <artifactId>junit</artifactId>
19. <version>4.11</version>
20. <scope>test</scope>
21. </dependency>
22. </dependencies>
23. </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](https://maven.apache.org/ref/current/maven-model/maven.html).

For more information about the dependency mechanism as a whole, see [Introduction to Dependency Mechanism](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html).

With this information about a dependency, Maven will be able to reference the dependency when it builds the project. Where does Maven reference the dependency from? Maven looks in your local repository (${user.home}/.m2/repository is the default location) to find all dependencies. In a [previous section](https://maven.apache.org/guides/getting-started/index.html#How_do_I_create_a_JAR_and_install_it_in_my_local_repository), 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:

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <groupId>com.mycompany.app</groupId>
6. <artifactId>my-other-app</artifactId>
7. ...
8. <dependencies>
9. ...
10. <dependency>
11. <groupId>com.mycompany.app</groupId>
12. <artifactId>my-app</artifactId>
13. <version>1.0-SNAPSHOT</version>
14. <scope>compile</scope>
15. </dependency>
16. </dependencies>
17. </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://repo.maven.apache.org/maven2/>. You can also set up your own remote repository (maybe a central repository for your company) to use instead of or in addition to the default remote repository. For more information on repositories you can refer to the [Introduction to Repositories](https://maven.apache.org/guides/introduction/introduction-to-repositories.html).

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:

1. <metadata>
2. <groupId>log4j</groupId>
3. <artifactId>log4j</artifactId>
4. <version>1.1.3</version>
5. <versioning>
6. <versions>
7. <version>1.1.3</version>
8. <version>1.2.4</version>
9. <version>1.2.5</version>
10. <version>1.2.6</version>
11. <version>1.2.7</version>
12. <version>1.2.8</version>
13. <version>1.2.11</version>
14. <version>1.2.9</version>
15. <version>1.2.12</version>
16. </versions>
17. </versioning>
18. </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:

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>jar</packaging>
12. <name>Maven Quick Start Archetype</name>
13. <url>http://maven.apache.org</url>
15. <dependencies>
16. <dependency>
17. <groupId>junit</groupId>
18. <artifactId>junit</artifactId>
19. <version>4.11</version>
20. <scope>test</scope>
21. </dependency>
22. <dependency>
23. <groupId>log4j</groupId>
24. <artifactId>log4j</artifactId>
25. <version>1.2.12</version>
26. <scope>compile</scope>
27. </dependency>
28. </dependencies>
29. </project>

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

**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:

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>jar</packaging>
12. <name>Maven Quick Start Archetype</name>
13. <url>http://maven.apache.org</url>
15. <dependencies>
16. <dependency>
17. <groupId>junit</groupId>
18. <artifactId>junit</artifactId>
19. <version>4.11</version>
20. <scope>test</scope>
21. </dependency>
22. <dependency>
23. <groupId>org.apache.codehaus.plexus</groupId>
24. <artifactId>plexus-utils</artifactId>
25. <version>1.0.4</version>
26. </dependency>
27. </dependencies>
29. <build>
30. <filters>
31. <filter>src/main/filters/filters.properties</filter>
32. </filters>
33. <resources>
34. <resource>
35. <directory>src/main/resources</directory>
36. <filtering>true</filtering>
37. </resource>
38. </resources>
39. </build>
40. <!--
41. |
42. |
43. |
44. -->
45. <distributionManagement>
46. <repository>
47. <id>mycompany-repository</id>
48. <name>MyCompany Repository</name>
49. <url>scp://repository.mycompany.com/repository/maven2</url>
50. </repository>
51. </distributionManagement>
52. </project>
53. <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
54. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
55. xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
56. http://maven.apache.org/xsd/settings-1.0.0.xsd">
57. ...
58. <servers>
59. <server>
60. <id>mycompany-repository</id>
61. <username>jvanzyl</username>
62. <!-- Default value is ~/.ssh/id\_dsa -->
63. <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id\_dsa)
64. <passphrase>my\_key\_passphrase</passphrase>
65. </server>
66. </servers>
67. ...
68. </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.

Care should be taken if using passwords in settings.xml. For more information, see [Password Encryption](https://maven.apache.org/guides/mini/guide-encryption.html).

**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:

1. mvn archetype:generate \
2. -DarchetypeGroupId=org.apache.maven.archetypes \
3. -DarchetypeArtifactId=maven-archetype-site \
4. -DgroupId=com.mycompany.app \
5. -DartifactId=my-app-site

Now head on over to the [Guide to creating a site](https://maven.apache.org/guides/mini/guide-site.html) to learn how to create the documentation for your project.

**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:

1. mvn archetype:generate \
2. -DarchetypeGroupId=org.apache.maven.archetypes \
3. -DarchetypeArtifactId=maven-archetype-webapp \
4. -DgroupId=com.mycompany.app \
5. -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:

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>my-webapp</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>war</packaging>
12. <dependencies>
13. <dependency>
14. <groupId>junit</groupId>
15. <artifactId>junit</artifactId>
16. <version>4.11</version>
17. <scope>test</scope>
18. </dependency>
19. </dependencies>
21. <build>
22. <finalName>my-webapp</finalName>
23. </build>
24. </project>

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

1. mvn clean package

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

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

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

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

1. +- pom.xml
2. +- my-app
3. | +- pom.xml
4. | +- src
5. | +- main
6. | +- java
7. +- my-webapp
8. | +- pom.xml
9. | +- src
10. | +- main
11. | +- webapp

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

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>pom</packaging>
12. <modules>
13. <module>my-app</module>
14. <module>my-webapp</module>
15. </modules>
16. </project>

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

1. ...
2. <dependencies>
3. <dependency>
4. <groupId>com.mycompany.app</groupId>
5. <artifactId>my-app</artifactId>
6. <version>1.0-SNAPSHOT</version>
7. </dependency>
8. ...
9. </dependencies>

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

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <parent>
6. <groupId>com.mycompany.app</groupId>
7. <artifactId>app</artifactId>
8. <version>1.0-SNAPSHOT</version>
9. </parent>
10. ...

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

1. mvn clean install

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

1. $ jar tvf my-webapp/target/my-webapp-1.0-SNAPSHOT.war
2. 0 Fri Jun 24 10:59:56 EST 2005 META-INF/
3. 222 Fri Jun 24 10:59:54 EST 2005 META-INF/MANIFEST.MF
4. 0 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/
5. 0 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/
6. 0 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-webapp/
7. 3239 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-webapp/pom.xml
8. 0 Fri Jun 24 10:59:56 EST 2005 WEB-INF/
9. 215 Fri Jun 24 10:59:56 EST 2005 WEB-INF/web.xml
10. 123 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-webapp/pom.properties
11. 52 Fri Jun 24 10:59:56 EST 2005 index.jsp
12. 0 Fri Jun 24 10:59:56 EST 2005 WEB-INF/lib/
13. 2713 Fri Jun 24 10:59:56 EST 2005 WEB-INF/lib/my-app-1.0-SNAPSHOT.jar

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

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

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

The final step was to include a parent definition. This 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...

1. mvn idea:idea