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!

Sections

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.

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

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

```
mvn -B archetype:generate \
  -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...

How do I compile my application sources?

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

```
mvn compile
```

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

```
[INFO] ------------------------------------------------------------------------
[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!

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-repo
rts

-------------------------------------------------------
 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 ...

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 (`~/.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:

```
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
13.
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
19.
20. Results :
21. [surefire] Tests run: 1, Failures: 0, Errors: 0
22.
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-SNAP
    SHOT.jar
28. [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.

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>2.5.1</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.

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

...
```

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>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>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:

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

```
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>
6.
7.   <groupId>com.mycompany.app</groupId>
8.   <artifactId>my-app</artifactId>
9.   <version>1.0-SNAPSHOT</version>
10.  <packaging>jar</packaging>
11.
12.  <name>Maven Quick Start Archetype</name>
13.  <url>http://maven.apache.org</url>
14.
15.  <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>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
```

```
      <artifactId>junit</artifactId>
      <version>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"
```

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>
```

```
6.
7.   <groupId>com.mycompany.app</groupId>
8.   <artifactId>my-app</artifactId>
9.   <version>1.0-SNAPSHOT</version>
10.  <packaging>jar</packaging>
11.
12.  <name>Maven Quick Start Archetype</name>
13.  <url>http://maven.apache.org</url>
14.
15.  <dependencies>
16.    <dependency>
17.      <groupId>junit</groupId>
18.      <artifactId>junit</artifactId>
19.      <version>3.8.1</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.

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:

```
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>
```

```
</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.

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>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>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.

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>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
```

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

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>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.

Care should be taken if using passwords in `settings.xml`. For more information, see Password Encryption.

How do I create documentation?

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

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-site \
  -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.

How do I build other types of projects?

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

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp \
    -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>
```

```
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <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.

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>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>my-app</module>
    <module>my-webapp</module>
  </modules>
</project>
```

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

```
  ...
  <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-webap
p/
3239 Fri Jun 24 10:59:56 EST 2005 META-INF/maven/com.mycompany.app/my-webap
```

```
    p/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-webap
    p/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-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...

```
1. mvn idea:idea
```