WHAT IS: JAVA
By Matthew Tyson, Java Developer, JavaWorld
JAN 30, 2020 9:13 AM PST

About
Everything you need to know about Java programming tools and APIs, with code and examples.

FEATURE

What is Maven? Build and dependency management for Java

Get started with Maven, the most popular build and dependency management tool for Java

Apache Maven is a cornerstone of Java development, and the most used build management tool for Java. Maven's streamlined, XML-based configuration model enables developers to rapidly describe or grasp the outlines of any Java-based project, which makes starting and sharing new projects a snap. Maven also supports test-driven development, long-term project maintenance, and its declarative configuration and wide range of plugins make it a popular option for CI/CD. This article is a quick introduction to Maven, including the Maven POM and directory structure, and commands for building your first Maven project.

Note that the most recent Maven release as of this writing is Maven 3.6.3.

Maven vs Ant and Gradle

Maven isn't the only build tool in the Java ecosystem, although it is the most popular one. Ant, an earlier generation of XML-based configuration tool, lacks Maven's standardized, convention-based practices and dependency management, but does offer flexibility you won't find with Maven. Gradle is a newer tool that runs on top of the Maven ecosystem (using Maven's repositories), but supports using a Groovy- or Kotlin-based DSL for configuration. All three are good build tools in their own right, and each can be integrated into a CI/CD process. What matters is choosing the right one for your needs and knowing how to use it appropriately.

How Maven works

Like many great tools, Maven takes what was once overcomplicated (configuration hell) and simplifies it to digestible parts. Maven consists of three components:

- The POM: The file that describes a Maven project and its dependencies.
- The directory: The standardized format for describing a Maven project in the POM.
- Repositories: Where third-party software is stored and discovered.

The Maven POM: Every Java project that uses Maven has a POM (project object model) file in its root directory. The `pom.xml` describes the project's dependencies and tells you how to build it. (*Dependencies* are third-party software required by the project. Some common examples are JUnit and JDBC. See the Maven Central Repository for a listing of all available tools and popular dependencies.)

The Maven directory: The Maven directory implements what's known as *convention over configuration*, an elegant solution to configuration hell. Rather than requiring developers to define the layout and hand-configure components for each new project (as was the case with `makefile` and Ant), Maven institutes a common project structure and offers a standard file format for describing how it works. You just plug in your requirements, and Maven calls in dependencies and configures the project for you.

Centralized repositories: Finally, Maven uses centralized repositories to both discover and publish project packages as dependencies. When you reference a dependency in your project, Maven will discover it in the centralized repository, download it to a local repository, and install it into your project. Most of the time, all of this is invisible to you as the developer.

Accessing Maven dependencies
By default, Maven resolves dependencies from the Maven Central Repository. A common alternative is JCenter, which has a wider set of available packages. Organizations also publish and host internal repositories, which can be public or private. In order to access a repository, you specify its URL in the Maven POM, or you can instruct Maven to look in other repositories.

Installing Maven

Maven is a Java project, so before you install it you'll need to have the JDK installed in your development environment. (See "What is the JDK? Introduction to the Java Development Kit" for more about downloading and installing the JDK.)

Once you have your Java development environment setup, you can install Maven in just a few steps:

1. Download the latest Maven release (Maven 3.6.3 as of this writing).
2. Extract the `apache.maven` .zip file to a convenient place.
3. Place that file on your path. For example, on a Unix or Linux system: export `PATH=$PATH:/home/maven/`.

You should now have access to the `mvn` command. Type `mvn -v` to make sure you've successfully installed Maven.

The Maven POM

The root of every Maven project is the `pom.xml` file. Despite its reputation for being tedious, XML actually works quite well for this use case. Maven's POM is easy to read and reveals much of what's going on in a project. (If you've worked with JavaScript, the `pom.xml` is similar in purpose to Node NPM's `package.json` file.)

Listing 1 shows a very simple Maven `pom.xml`.

Listing 1. Simple Maven POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javaworld</groupId>
  <artifactId>what-is-maven</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Simple Maven Project</name>
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Understanding the Maven POM

Once you get the hang of it, the POM is not mysterious. To start, you can skim over the XML preamble, which just references the official POM schema. Do notice the XML starting with `modelVersion`, however. That tells Maven what version of the POM to use, in this case Maven POM 4.0.0.

Next, you have `groupId`, `artifactId`, and `version`. Together, these three attributes uniquely identify every Maven-managed resource in the repository. These attributes at the top of the file describe your Maven project.

Now, take a look at the `dependencies` section of the POM, where we describe the project's dependencies. In this case we've pulled in just one dependency so far, JUnit. Notice that JUnit is also described in terms of its `groupId`, `artifactId`, and `version`.

Whether you are describing your own project or a project dependency, these values consistently tell Maven where to find a project in the Maven repository, and which version is available for use.

Hosting your project in a Maven repository

Keep it in mind that the POM defines everything your project needs to run, but it also describes your project as a potential dependency. If you are building a project that will be a dependency--say, creating a library for other projects to use--you will need to make it available in one of four ways:

1. Make it available locally.
2. Publish to a privately managed remote repository.
3. Publish to a cloud-based private repository.
4. Publish to a public repository like Maven Central.

In the first case, you do not use a remote repository at all. Instead, other developers will download and install your project locally to their Maven repo, using the `mvn install` command.

In the second case, you use a hosted Maven repository, using privately controlled server to publish and download dependencies. For this you need a repository manager, like Apache Archiva.

A newer alternative is to use a private remote repo, but rely on a cloud-based service to manage it, for instance Cloudsmith. This gives the benefit of remotely hosted dependencies without the work of standing up a repo server. That service is for a fee, of course.

Finally, a small percentage of projects will end up in the Central Maven Repository or JCenter, which are intended for widely-used, public packages. If you are creating an open-source dependency to be used by others, you will need one of these centralized repositories to make your work available to the world.

- Learn more about hosting your project in a Maven repository and get a list of available repositories.
- See the official Maven documentation about the Maven Release Plugin, used to prepare and manage software published to a Maven repository.

Build the Maven package

If you create the `pom.xml` from Listing 1 and put it in a directory, you will be able to run Maven commands against it. Maven has a multitude of commands, and more are available via plugin, but you only need to know a handful to start.

For your first command, try executing `mvn package`. Even though you don't have any source code yet, executing this command tells Maven to download the JUnit dependency. You can check Maven's logging output to see that the dependency has loaded..

Dependency scope

You might have noticed the JUnit dependency in the example POM is marked as `scope test`. *Scope* is an important concept in dependency management, essentially allowing you to define and limit how each dependency will be called and used in your project. The `test` scope ensures the dependency is available when running tests, but not when the app is packaged for deployment.

Another common scope is `provided`, which tells the framework that the dependency is provided by the runtime environment. This is often seen with the Servlet JARS when deploying to a servlet container, as the container will provide those JARS. See the Apache Maven documentation for a complete list of Maven dependency scopes.

Maven's directory structure

When the command is done, notice that Maven has created a `/target` directory. That is the standard location for your project's output. Dependencies you've downloaded will reside in the `/target` directory, along with your compiled application artifacts.

Next you want to add a Java file, which you'll place in the Maven `src/` directory. Create a `/src/main/java/com/javaworld/Hello.java` file, with the contents of Listing 2.

Listing 2. Hello.java

```
com.javaworld

public class Hello {
  public static void main(String[] args){
    System.out.println("Hello, JavaWorld");
  }
}
```

The `/src` path is the standard spot for your project's source files. Most projects put their main files in `/src/main/`, with Java files going into the classpath under `/java`. Additionally, if you want to include assets that are *not* code, like config files or images, you can use `/src/main/resources`. Assets in this path will be added to the main classpath. Test files go into `/src/test/java`.

To review, here are some key parts of a Maven project structure (as defined by the Maven Standard Directory Structure):

Key parts of the Maven Standard Directory Structure

pom.xml	The project descriptor file
/src/main/java	Location of source files
/src/main/resources	Location of non-source assets
/src/test/java	Location of test source files
/target	Location of the build output

Managing your Maven project

The `mvn package` command instructs Maven to bundle up the project. Issue this command when you're ready to collect all your project files in one place. Recall that in the POM file for this project, we set the packaging type to be `jar`, so this command tells Maven to

package the application files into a JAR.

Maven offers a variety of additional options for controlling how the JAR is managed, whether it is a fat or thin JAR, and specifying an executable `mainclass`. See the Maven docs to learn more about file management in Maven.

After you've bundled a project, you'll likely want to issue a `mvn install`. This command pushes the project into the local Maven repository. Once it's in the local repository, it's available to other Maven projects in your local system. This is useful for development scenerios where you and/or your team are creating dependency JARs that are not yet published to a central repository.

Additional Maven commands

Enter `mvn test` when you're ready to run unit tests you've defined in the `/src/java/test` directory.

Enter `mvn compile` when you're ready to compile the project's class files. If you're running a hot-deploy setup, this command triggers the hot deploying class loader. (The hot-deploy tool--like Spring Boot's `mvn spring-boot:run` command--will be watching the classfiles for changes, and compiling will cause your source files to be compiled, and the running application will reflect those changes.)

Starting a new project: Archetypes in Maven and Spring

A *Maven archetype* is a template for starting up new projects based on a variety of pre-defined settings. Each archetype offers pre-packaged dependencies, such as for a Java EE or Java web application project. You can also create a new archetype from an existing project, then use it to rapidly create new projects based on those pre-defined layouts. See the Maven docs to learn more about Apache Maven archetypes.

The Spring framework, which works well with Maven, offers additional, sophisticated capabilities for stubbing out new projects. As an example, Spring Initializr is a tool that lets you very quickly define the elements you want in a new app. Initializr is not a Maven archetype, per se, but it serves the same purpose of generating a project layout based on up-front specifications. From within Initializr, you can type `mvn archetype:generate` and scan through the options to find an archetype that is suitable for what you are building.

Adding dependencies