



Java: Build Automation with Maven

Welcome to a new post where we'll explore the Apache Maven build tool and how to make the most of it.

 Uday Chauhan

What is Apache Maven?

Apache Maven, is a software project management, and comprehension tool, based on the concept of a project object model, or POM. Maven can manage a project's build, reporting, and documentation from a central piece of information.

A more comprehensive definition of Apache Maven, is that Maven is a project management tool, which encompasses a project object model. It follows a set of standards, it includes a project life cycle, a dependency management system, and logic for executing plugin goals at defined phases in a life cycle. Maven is designed to provide a simple project setup, that uses best practices as a guide.

With Maven, your projects follow a consistent structure. Projects become IDE agnostic, by enforcing a consistent structure, it makes modifications easier in the future, when new developers are introduced to the project. It also ensures that programmers always get the most recent version of compilers, etc.

Most Java projects rely on other projects, and open source frameworks, to function properly. It can be cumbersome to download these dependents manually, and keep track of their versions, as you use them in your project. Maven provides a convenient way to declare these project dependencies, in a separate, external, POM.XML file. It then automatically downloads these dependencies and allows you to use them in your project. This simplifies project dependency management greatly. It is important to note, that in the POM.XML file, you specify the what, and not the how. The POM.XML file, can also serve as documentation tool, conveying your project dependencies and their versions. Software developers refer to Maven, as a build tool. Since it is used to build deployable artifacts from source code. On the other hand, if you asked a project manager they might call it a project management tool, since it follows a development life cycle. In reality, it is both.

Download Maven

To get started using Apache Maven, start by downloading Maven from the maven.apache.org website. From this page we can just use the link in the middle where it says Use Download. Always make sure you download the latest version.

Install Maven on Windows

Now that we have the Maven file downloaded and extracted into our program files directory, we can go ahead and install Maven on our Windows machine. As I stated earlier, the Maven download is not very large. That's because Maven's power is included in its plugins which are located and retrieved from a central repository on an as-needed basis and allowing for greater code reuse. Before the installation we must verify our Java version from the command line using `java -version`. Remember it must be 1.7 or higher.

1. Install JDK and Add 'JAVA_HOME' Environment Variable

To install java, [download JDK installer](#) and add/update the JAVA_HOME variable to JDK install folder.

2. Download Maven and add 'MAVEN_HOME' and 'M2_HOME' Environment Variables

Maven can be downloaded from this [location](#).

Set the M2_HOME and MAVEN_HOME variable to maven installation folder.

3. Include 'maven/bin' directory in 'PATH' variable

To run maven from command prompt, this is necessary. Update the PATH variable with 'Maven-installation/bin' directory.

4. Verify maven in console

Maven installation is complete. Now lets test it from windows command prompt.

- a. Go to start menu and type cmd in application location search box.
- b. Press ENTER. A new command prompt will be opened.
- c. Type `mvn -version` in command prompt and hit ENTER.

```
$ mvn -version
```

Maven with proxy

To set **Maven Proxy** :

Edit the proxies session in your **Home Folder** - `C:/Users/{UserName}/.m2/settings.xml` file. If you cant find the file, create one.

or

Edit the proxies session in your `{M2_HOME}/conf/settings.xml`

You can go through the [Official documentation](#)

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <settings>
3      <proxies>
4          <proxy>
5              <id>httpproxy</id>
6              <active>true</active>
7              <protocol>http</protocol>
8              <host>your-proxy-host</host>
9              <port>your-proxy-port</port>
10             <nonProxyHosts>local.net|some.host.co
11         </proxy>
12         <proxy>
13             <id>httpsproxy</id>
14             <active>true</active>
15             <protocol>https</protocol>
16             <host>your-proxy-host</host>
17             <port>your-proxy-port</port>
18             <nonProxyHosts>local.net|some.host.co

```

If the settings file changes don't work, try this in the command prompt having the POM file.

```

mvn install -Dhttp.proxyHost=abcproxy -Dhttp.proxyPort=8080 -Dhttps.proxyHost=abcproxy -
Dhttps.proxyPort=8080

```

IDE integration

If you are accustomed to using Eclipse, then you wanna use the Eclipse version M2Eclipse, which is specifically designed for Maven integration. This version of Eclipse includes the ability to launch Maven builds, handle dependency management based on Maven's pom.xml file, automatic download of required dependencies, and wizards for creating new project and search capabilities for Maven remote repositories.

<https://www.eclipse.org/m2e/> is the homepage for this version of the M2Eclipse website. As you can see, this version of Eclipse is designed specifically for integration of Apache Maven.

Project Object Model (POM)

Maven use of the concept of a Project Object Model, or POM. This model has a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at certain phases in the lifecycle process. One of the things that makes Maven so powerful is that it relies on the concept that projects are set up with default behaviors. For example, the pom.xml file is always located in the base directory. The source code must be in a certain directory. Resources necessary for the project are in a another folder or directory. Test cases are in a specifically named folder. And a target folder is always created that's used for the final JAR file. This folder structure is an important example of how Maven has adopted convention over configuration. By always using a standard folder structure, it allows developers to concentrate on coding. Once the code and resources are placed in the correct directories, and the POM file is updated. Maven handles the rest.

A project model includes:

- A project description
- A unique set of coordinates
- Project attributes
- The project's license information.
- The project version,
- Any authors or contributors to the project
- A list of project dependencies.

Before we go further, let's take a look at a sample POM file.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xs
3      <modelVersion>4.0.0</modelVersion>
4      <groupId>com.ucguy4u</groupId>
5      <artifactId>datastructures</artifactId>
6      <packaging>jar</packaging>
7      <version>1.0-SNAPSHOT</version>
8      <name>datastructures</name>
9      <url>http://maven.apache.org</url>
10     <build>
11         <plugins>
12             <plugin>
13                 <groupId>org.apache.maven.plugins</groupId>
14                 <artifactId>maven-javadoc-plugin</artifactId>
15                 <version>3.2.0</version>
16                 <executions>
17                     <execution>
18                         <id>attach-javadocs</id>
19                         <goals>
20                             <goal>jar</goal>
21                         </goals>
22                     </execution>
23                 </executions>
24                 <configuration>
25                     <additionalOptions>
26                         <additionalOption>-Xdoclint:none</additio
27                     </additionalOptions>
28                     <reportOutputDirectory>./docs</reportOutputD
29                 </configuration>
30             </plugin>
31             <plugin>
32                 <groupId>org.apache.maven.plugins</groupId>

```

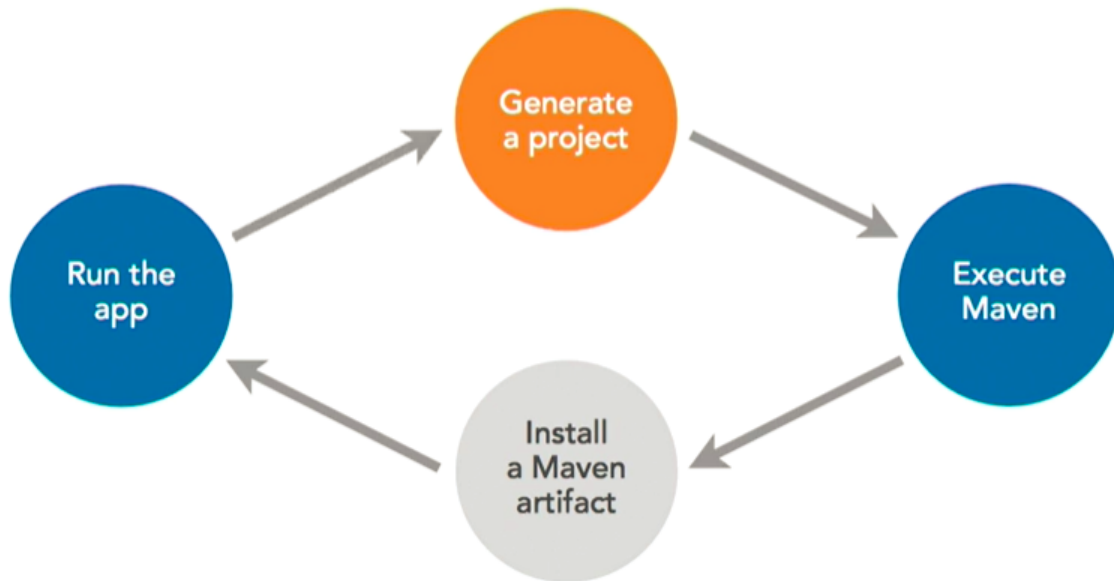
```

33         <artifactId>maven-compiler-plugin</artifactId>
34         <version>2.5.1</version>
35         <inherited>true</inherited>
36         <configuration>
37             <source>1.8</source>
38             <target>1.8</target>
39         </configuration>
40     </plugin>
41 </plugins>
42 </build>
43 <dependencies>
44     <dependency>
45         <groupId>junit</groupId>

```

This is the file for the datastructure project. The POM file is stored as an XML file. XML files use tags similar to HTML. Except for the names inside the tags are different. In the case of Maven, we have tags such as group ID, artifact ID, packaging, version, etc. The artifact ID is used for the name of the program. In our case, datastructure. Since it's a Java program, the packaging is going to be to create a JAR file. And the version in this case is 1.0. The description, name, and URL are all optional. Below that are the dependencies. When you create a sample program using Maven, it automatically adds a JUnit dependency to allow us to do unit testing for our Java program. So, if I wanted to create a second version of my datastructure project. I'd have to change the version number from 1.0 to 2.0, or 1.1, or something, to make it unique. Features that are enabled by using the POM include dependency management, access to remote repositories, universal reuse of build logic, tool portability and integration, allowing IDEs such as Eclipse, NetBeans, and IntelliJ, to have a common place to find information about a project. And, easy searching and filtering of project artifacts. The Project Object Model, the POM, and the POM file are the heart of Maven projects. And provide key information about the project.

Maven lifecycle



When using Maven it's important to understand the Maven life cycle. Let's take a look at a high level overview of the flow when using Maven. Maven starts by generating a project. A project consists of a POM or Project Object Model and source code that's assembled in the Maven standard directory layout. Next, we execute Maven with a life cycle phase as an argument that prompted Maven to execute a series of plugin goals. After that, we can install a Maven artifact into our local Maven repository. And finally, we can run the app.

Let's take a closer look at the default life cycle phases.

Validate : It is used to validate the project to make sure it is correct and all necessary information is available.

Compile: We compile the source code of the project.

Test: It compiles the source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed just yet.

Package: take the compiled code and package it in its distributable format.

For example, a Java program will be packaged as a Java file or a Java archive file. Integration-test. Process and deploy the package if necessary into an environment where integration tests can be run. Verify runs any checks to verify the package is valid and meets quality criteria. Install the package into the local repository for use as a dependency in other projects locally. And finally, Deploy. This is done in an integration or release environment. It copies the final package to the remote repository for sharing with other developers and projects.

Plugin goals can be attached to each lifecycle phase. As Maven moves through the phases in a lifecycle it will execute the goals attached to each particular phase. Each phase may have zero or more goals bound to it.





For example, when we run `mvn install` we will see that more than one goal is executed. In the package phase it'll automatically execute the JAR goal in the JAR plugin. Let's run an `mvn install` on an existing project and we can see the different goals that are executed.

```

PS C:\DATA\workspace\DataStructures> mvn install
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ucguy4u:datastructures >-----
[INFO] Building datastructures 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ datastructures ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\DATA\workspace\DataStructures\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ datastructures ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ datastructures ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\DATA\workspace\DataStructures\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ datastructures ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ datastructures ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ datastructures ---
[WARNING] JAR will be empty - no content was marked for inclusion!
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ datastructures ---
[INFO] Installing C:\DATA\workspace\DataStructures\target\datastructures-0.0.1-SNAPSHOT.jar to C:\Users\uschauha\.m2\repository\com\ucguy4u\datastructures\0.0.1-SNAPSHOT\datastructures-0.0.1-SNAPSHOT.jar
[INFO] Installing C:\DATA\workspace\DataStructures\pom.xml to C:\Users\uschauha\.m2\repository\com\ucguy4u\datastructures\0.0.1-SNAPSHOT\datastructures-0.0.1-SNAPSHOT.pom
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.536 s
[INFO] Finished at: 2020-08-15T11:40:54+05:30
[INFO] -----
PS C:\DATA\workspace\DataStructures>

```

I've navigated to my directory for data structures. If I run mvn install, which is a lifecycle phase, we can see what goals are executed. At the top you'll see Maven resources-plugin 2.6 resources. So the resources plugin was executed. Next, the compile was executed. There was nothing to compile because all classes were up to date. Next is the test resources, the test compile, and finally using the surefire plugin it ran a test. . Zero failures and zero errors. So everything worked according to our specifications. The last plugin that you'll see here is the JAR plugin. The JAR plugin again is used to create a Java archive file or a JAR file. After the JAR file is created the install then moves the JAR file to local Maven repository.

Name	Date modified	Type	Size
 _remote.repositories	8/15/2020 11:42 AM	REPOSITORIES File	1 KB
 datastructures-0.0.1-SNAPSHOT.jar	8/15/2020 11:42 AM	Executable Jar File	2 KB
 datastructures-0.0.1-SNAPSHOT.pom	8/15/2020 11:42 AM	POM File	1 KB
 maven-metadata-local.xml	8/15/2020 11:42 AM	XML File	1 KB

You can see here it says that the installing of the datastructures to "C:\Users\
<username>\.m2\repository\com\ucguy4u\datastructures\0.0.1-SNAPSHOT\datastructures-0.0.1-SNAPSHOT.jar". It also installed the POM file into that same repository. That repository is automatically created by Maven and that's where all our local programs are stored.

Maven Repository

One of the big benefits of using Maven is you now have access to the Maven repository. There's actually two repositories that we're gonna talk about.

The first one is the central Maven repository that contains a large collection of Java and other open-source components. Again, the power of Maven is that these open-source components are available to you but they will not be downloaded unless you need them. That enforces that you have the most recent version of each component. The second repository is a local repository that Maven creates on your computer. It's usually located on your home drive in a folder called .m2. This directory contains your Maven repository. When you download a dependency from a remote Maven repository, Maven stores a copy of the dependency in your local repository. In addition, it also places a copy of your jar file and the pom.xml file for each installed project.

Let's start by looking at a website that allows you to browse the central repository. You can find this at search.maven.org. As you can check, you can enter in any component that you want to search for and it does have an advanced search feature and even a browse feature.

Now, let's take a look at the local Maven repository. As you can see, `.m2` folder is located under `c:/Users/<username>`. Inside the local repository, you'll see an `index` folder and a `repository` folder. It's already downloaded a lot of different components that I needed, including `junit`. So all of your projects should be available inside your local repository.

Maven's dependency management

Another feature of Maven is the way it handles Dependency Management. As a programmer, we often take advantage of code reuse, especially in the realm of open-source programming.

For example, in Java, we use APIs that contain libraries of software components that we can use instead of recreating the source code from scratch. Almost all Java programmers have, at some time, used the `Math` functions or the `String` functions in their coding. These functions include `math.pow` to find the value of a number raised to a power, or `math.min` to find the smallest of two numbers. Some of the `String` functions include the `.length` command, the `.substring`, `two lower`, `two upper`. These are just a few examples of what it means to reuse existing code.

Maven also allows us to reuse existing components and plugins using the dependency section of our POM file, our Project Object Model file. For example, a common dependency when working with Java is the `junit` component. We define dependencies inside a project's POM file using its Maven coordinates. Remember, the coordinates include the group ID, artifact ID and version.

Let's refer back to the POM file we saw earlier.

```
<?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-
  instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ucguy4u</groupId>
  <artifactId>datastructures</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>datastructures</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

As you can see, there's a tag called `dependencies`, and inside these, there is one dependency called `junit`. The three tags `groupId`, `artifactId`, and `version` are the coordinates that make this particular dependency unique. The `scope` identifies what part of the life cycle this dependency is going to be used in. In this case, it's the test phase. It is easy to add additional project dependencies by updating this `pom.xml` file. By adding a list of dependencies here in one place, it is also easy for someone to identify what dependencies are required for this particular project. Finally, by

including the dependencies in this external file, it is easy to update the version numbers in one place as dependencies might change.

POM categories and configuration

The Project Object Model or POM Categories and Configuration. The POM file contains all the information about a project. The file is stored with an .XML extension. Here's an example of POM.XML file that has the minimum amount of information required.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>
<groupId>com.ucguy4u</groupId>
<artifactId>datastructures</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>datastructures</name>
<url>http://maven.apache.org</url>

</project>
```

As you can see, it has a groupId, an artifactId, and a version. Remember those three things make up the Maven coordinates and are required for all projects. Although this is not a comprehensive list, some of the most common categories that are usually included in POM file include the project coordinates, which we just saw, the project's license information, a list of developers and contributors to the project, a list of project dependencies, the name of the project, the URL associated with this project, the packaging type, the scope of the element listed, and even information about inheritance. Each of these categories has its own XML tag. For example, the scope tag would be the less than sign, the word scope, and the greater than sign. Although these are not all the categories that you can use in a POM file, these are the ones that you'll see most often.

POM syntax

The Project Object Model, or POM, is documented in an XML file, where XML stands for Extensible Markup Language, which is always located in the base directory of your project.

Every open XML tag must have a corresponding closing tag. And tags can be nested one inside the other. The file can start with an XML declaration, but that is optional. All values in the file are declared as XML elements. All projects extend the super Project Object Model, or POM, automatically. And the specific project POM contains all pertinent information for that project. As we seen above an example of a pom.xml file that includes the XML declaration, and the required maven coordinates. As you can see, the XML declaration takes up the first four lines. Below that, we have the modelVersion and then the maven coordinates, the groupId, the artifactID, and the version number. The POM file contain many more XML tags depending on the complexity of your project.

Project dependencies

As a programmer, we often rely on other components available to us. Maven provides support for both internal and external dependencies. One of the most common dependencies is the junit dependency. This is used for testing a Java program. Other examples include log4j and jaxen, just to name a few. When adding dependencies, you can add the scope tag to indicate which life cycle phase uses this component. The test scope tag is used for the junit dependency. There are other scopes that are included, such as compile, provided, runtime, and system. Compile is the default scope. Provided is used when the JDK is expected to provide them. Runtime is required for executing and testing, not compiling. Test is not required during the normal operation of an application, and system is similar to provided, but must specify the explicit path to the jar on the locals file system. By placing the dependencies in a separate pom.xml file, it allows us to easily add new dependencies, remove existing dependencies, and even change the version of existing dependencies. This is one of the big benefits of using Maven for your project build.

Project relationships

One of the reasons to use Maven is the ability to easily track down dependencies using the POM file. The relationships between projects can be external or internal. An example of an external relationship might be the Log4j and JUnit, where an internal relationship might be an example where project-a depends on project-b. All project relationships are established using Maven coordinates. Remember, a Maven coordinate is made up of the group ID, the artifact ID, and the version. To indicate a relationship, we describe the dependency as group ID, colon, artifact ID, colon, version. Don't forget that projects also inherit project relationships such as dependencies from parent POM files and from the super POM file.

POM best practices

Grouping dependencies is one of the best practices. This can be done by creating a separate POM file that simply declares a set of common dependencies.

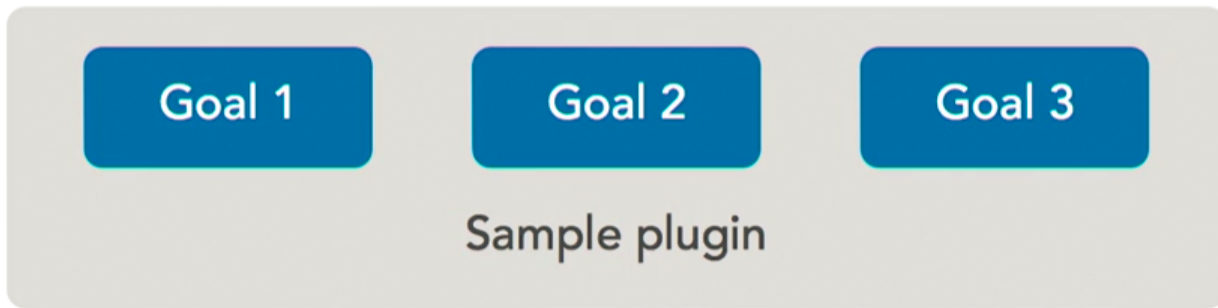
For example, every project that uses Hibernate has a dependency on the Spring framework at mysql.jdbc.Driver. By grouping these dependencies, we can create a separate external POM file. That way, that file can be used for all projects that have the same dependencies as the one that you created using Hibernate. This approach allows other projects to reuse this file. A second consideration regarding best practices is the difference between using inheritance versus a multi-module relationship. There are projects that have components that are totally unrelated to each other. But together, they are required to make a larger project. This is an example when you should use a multi-module approach. A reason to use inheritance occurs when projects have shared dependencies.

For any programming language, including the programming of the pom.xml file, it always helps to use proper indentation to make it easier to read. Another practice is to follow a standard layout where your coordinates are listed first. This is an important example of how Maven has adopted convention over configuration. It also makes the program easier to read and be maintained by other programmers.

Maven Plugins:

Core plugins

A plugin is a collection of one or more goals. And a goal consists of a unit of work in Maven.



In this image, you'll see there are three goals associated with my sample plugin. A plugin may have one or more goals. Maven consists of several core plugins.

These core plugins include:

- JAR plugin: which creates the JAR, or Java Archive files.
- Compiler plugin: which contains goals for compiling source code and unit tests,
- Surefire plugin: which is used for executing unit tests and generating reports.
- Custom plugins: It can be written in Java, or a plugin can be written in any number of languages, including Ant, Groovy, Bean, Ciao, and Ruby.

Let's take a look at all the Maven plugins. The complete list of Maven plugins is available at the [plugins](#). It starts with a list of what they call core plugins. As you can see, some of the core plugins include clean, compiler, deploy, failsafe, install, resources, site, surefire, and verifier. From this site, you can get detailed information about each plugin and the goals that are available for each one.

What do I mean by the goals? Let's take a look at the compiler plugin. The two goals for this plugin are compile and testCompile.

Compile is bound to the compile phase, and is used to compile the main source files.

TestCompile is bound to the test-compile phase, which would probably be in the JUnit. The test-compile phase is used to compile the test source files.

It is important to bookmark this website since names of plugins and goals can change. To execute a single Maven plugin goal, use the command :

```
| mvn pluginname:goal
```

Now I'm ready to execute my mvn command. So I'll do mvn , then I specify the plugin, which in our case was compiler: and then I'll specify the goal, which in our case was compile, and I'll hit enter.

```
PS C:\DATA\Workspace\DataStructures> mvn compiler:compile
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ucguy4u:datastructures >-----
[INFO] Building datastructures 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-cli) @ datastructures ---
[INFO] No sources to compile
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] -----
[INFO] Total time: 0.988 s
[INFO] Finished at: 2020-08-15T12:42:02+05:30
[INFO] -----
```

As you can see, it builds the datastructures 0.0.1-SNAPSHOT. It has the compile, and it says BUILD SUCCESS. It looks like our command worked. So as you can see, this is an example of how to use a plugin in a goal within the Maven environment.

Packaging tools

The next type of plugins that I'd like to review are the packaging tools. So if we check, we see our [core plugins](#), and the next section talks about the packaging types and tools.

As you check, the list starts with EAR, EJB, JAR, RAR, WAR, app-client, shade, and source. As you start to use maven, I think you'll find you use some of the packaging tools more than others. If you're a Java programmer, you'll probably use the JAR plugin, which is used to build a JAR from the current project, where JAR is the Java Archive. The EJB is also used for Java projects, but that's the Enterprise Java Beans package.

RAR stands for Resource Adapter Archive.

WAR is a Web Application Archive.

Each packaging type has different set of goals. These elements affect the steps required to build a project.

For example, if the goal is POM, the project will run the site attach-descriptor goal during the package phase. A jar type runs with JAR:JAR goal instead. JAR is the default packaging type in maven, and the most commonly used.

Let's switch back over to our Command Prompt, and try running the JAR-JAR plugin.

```
PS C:\DATA\Workspace\DataStructures> mvn jar:jar
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ucguy4u:datastructures >-----
[INFO] Building datastructures 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-cli) @ datastructures ---
[WARNING] JAR will be empty - no content was marked for inclusion!
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] -----
[INFO] Total time: 0.735 s
[INFO] Finished at: 2020-08-15T13:43:51+05:30
[INFO] -----
```

where the first word JAR represents the plugin, and the second one represents the goal, I'm trying to create a JAR file. And I'll hit enter. Again, it says build successful. So we just created a Java Archive. Since JAR is the default packaging type in maven, if you omit the packaging type, it will automatically use JAR.

Reporting

Once we've created our project and completed the steps for compiling and building our program, we can also get some reports from Maven. As you check, there are several reporting plugins. we have changelog, changes, checkstyle.

Let's try using one of these reporting plugins. Let's use the javadoc.

For now, I'm gonna stick with the first one since my program is fairly simple and I'm gonna use javadoc:javadoc.

Let's go back to our command prompt.

```
| mvn javadoc:javadoc
```

It had to create some files for my javadoc. Now that we created the javadoc, let's go ahead and launch it and take a look.

Maven automatically puts it into my target folder under site/apidocs and go ahead and lunch index.html.

Let's go back to our list of plugins. As you can see, these reporting plugins are pretty powerful. So make sure you give a couple of them a try. Start with javadoc and then maybe try changelog or changes.

Tools

The last set of plugins that I want to review are the tools plugins. One of the tools that we'll use, and that you'll probably find very useful, is called the archetype.

The archetype plugin generates a skeleton project structure from an archetype. An archetype is like a template. Let's take a look at the goals available for archetype.

As you check, Generate creates a Maven project from an archetype. We could also create an archetype from a Project, kind of work our way backwards, and we can use the crawl goal to search a repository for archetypes and updates the catalog.

Let's take a look at how to do that using the archetype.

So to get information on a plugin, we can type the mvn help man which uses describe as the goal.

```
mvn help:describe -Dplugin=archetype
```

```
Name: Maven Archetype Plugin
Description: Maven Archetype is a set of tools to deal with archetypes, i.e.
  an abstract representation of a kind of project that can be instantiated into
  a concrete customized Maven project. An archetype knows which files will be
  part of the instantiated project and which properties to fill to properly
  customize the project.
Group Id: org.apache.maven.plugins
Artifact Id: maven-archetype-plugin
Version: 3.2.0
Goal Prefix: archetype

This plugin has 7 goals:

archetype:crawl
  Description: Crawl a Maven repository (filesystem, not HTTP) and creates a
  catalog file.

archetype:create-from-project
  Description: Creates an archetype project from the current project.

archetype:generate
  Description: Generates a new project from an archetype, or updates the
  actual project if using a partial archetype. If the project is fully
  generated, it is generated in a directory corresponding to its artifactId.
  If the project is updated with a partial archetype, it is done in the
  current directory.

archetype:help
  Description: Display help information on maven-archetype-plugin.
  Call mvn archetype:help -Ddetail=true -Dgoal=<goal-name> to display
  parameter details.

archetype:integration-test
  Description: Execute the archetype integration tests, consisting in
  generating projects from the current archetype and optionally comparing
  generated projects with reference copy.

archetype:jar
  Description: Build a JAR from the current Archetype project.

archetype:update-local-catalog
  Description: Updates the local catalog
```

All right, at the very top it says the name is the Maven archetype plugin. It gives us a description of what the plugin does. It tells us the group ID, the artifact ID, and the version. Which make up our three values that we use for our Maven coordinate.

Create a sample program

Let's create a sample program using Maven. In this sample program we will use the build lifecycle, Maven repositories, dependency management, and POM. You wanna start by opening a command prompt, run the Maven command by typing

```
mvn archetype:generate -DgroupId=com.ucguy4u.app -DartifactId=datastructure -
DarchetypeArtifactId=maven-archetype-quickstart interactiveMode=false
```

You'll see some messages from Maven. And what you're looking for is to make sure that you got "BUILD SUCCESS". At this point it looks like I generated my older structure, my POM file, and even my Java application that will print out "Hello World!" Note, if this is the first time that you're running Maven, it might have taken a little longer. The first time it downloads a number of files which include the most recent version of the Resources plugin, among others.

So let's go ahead and run `mvn install` to install our new program. Install is not a goal, so I don't attach a plugin or a goal; it's actually a phase in the lifecycle, and it will actually invoke several plugins.

```
mvn install
```

```
PS C:\DATA\workspace\DataStructures> mvn install
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ucguy4u:datastructures >-----
[INFO] Building datastructures 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ datastructures ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\DATA\workspace\DataStructures\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ datastructures ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ datastructures ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\DATA\workspace\DataStructures\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ datastructures ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ datastructures ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ datastructures ---
[WARNING] JAR will be empty - no content was marked for inclusion!
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ datastructures ---
[INFO] Installing C:\DATA\workspace\DataStructures\target\datastructures-0.0.1-SNAPSHOT.jar to C:\Users\uschauha\.m2\repository\com\ucguy4u\datastructures\0.0.1-SNAPSHOT\datastructures-0.0.1-SNAPSHOT.jar
[INFO] Installing C:\DATA\workspace\DataStructures\pom.xml to C:\Users\uschauha\.m2\repository\com\ucguy4u\datastructures\0.0.1-SNAPSHOT\datastructures-0.0.1-SNAPSHOT.pom
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.612 s
[INFO] Finished at: 2020-08-15T16:10:21+05:30
[INFO] -----
```

successfully created the jar file; it actually ran one test, and that was successful as well. The install command also creates an artifact that's added to our local Maven repository.

OK, the last thing I wanna show you is how to run the project now that we created the jar file.

Let's go back to our command prompt, and I'm gonna go ahead and type in

```
java -cp .\target\datastructures-0.0.1-SNAPSHOT.jar
com.ucguy4u.datastructures.linkedlist.singlylinklist.SinglyLinkedList
```

-CP is used for a class search path. It will search directories and look for zip/jar files.

Write unit tests

Let's talk a little bit about unit testing. As I'm sure you'll agree, unit testing is a critical step in any programming project. What's really nice about Maven is that it provides built-in support for unit testing. JUnit plug-in is used to easily test our application. When we first created our project using the archetype quick start to get our project created, it automatically created a test directory with a test application. Notice it automatically has the dependency of JUnit. That was done by Maven for us. Here is our class called AppTest that was created automatically for us. So this is what it would look like if one of your unit tests failed. So remember, when you create your project with Maven, it will create a shell AppTest file, but you're gonna need to go in there and add your own test cases.

Add dependencies

There are times when you need to add dependencies. This is one of the benefits of using Maven. It makes adding dependencies easy. Remember, Maven supports both internal and external dependencies. Whenever a project references a dependency that isn't available in a local repository, Maven will download the dependency from a remote repository into the local repository. So far, all of our projects have included the JUnit dependency. It is sometimes going to be necessary to add other dependencies required by your project. Let's say we've added some logging to our code for debugging purposes, and we need to add the Log4j as a dependency. Let's add this dependency to our calculator project. In order to add the dependency, we need to edit the pom.xml file.

```
<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>1.2.17</version>
  </dependency>
</dependencies>
```

The logging is used in the compile phase of the life cycle. And we don't want to end our dependency tag. In order to use a dependency, just update the pom file.

Packaging your app

The last part of the process is packaging your application. The packaging information is stored in your pom.xml file. Some sample packaging types include jar, for Java Archive Files, war, for Web Archive File, EAR. Remember, the default is a jar file. If the type is omitted, Maven will automatically create a jar file. Let's take a look at the pom.xml for our calculator app. That packaging tag says jar, because our application was a Java application. So it will automatically create a Java archive. Go to the Command Prompt, and from within the base project folder, we can type MVN Package, and it will create the jar file. This is also done when you run the MVN Install as well as even the MVN Test. But now we have our jar file, we have a copy of it in our local repository, and we're ready to go. So remember, when you're ready to package your application, check your pom.xml file to see what packaging type you have declared.

Next steps

The Apache Maven framework is one of many projects available through the open source Apache license. Apache Maven is a software project management and comprehension tool. Based on the concept of the project-object model, or POM, Maven can manage a project's build, reporting, and documentation from a central piece of information. After going through this article, I hope you have a better understanding of the process of using Maven to create new projects, update existing projects, and even integrate with popular IDEs. There's a lot to learn when using Maven, so make sure you bookmark maven.apache.org/ website for future reference. Most importantly, remember to always be a lifelong learner, keep programming, and have fun while doing it. Thanks for your time. I hope to see you again.