



## Build Tools - Maven

### Recommended Reading

- [Apache Maven Project documentation](#)
- [Baeldung Apache Maven Tutorial](#)
- [Jenkov Maven Tutorial](#)
- [Technology Conversations - Java Build Tools: Ant vs Maven vs Gradle](#)

### Build Tools Overview

Simple desktop applications consisting of a few Java source files can be manually assembled, compiled and executed with nothing more than a text editor and a few commands entered in a console. However the build process quickly becomes very complicated for applications developed for the Enterprise.

Consider, for example, even a simple Java Enterprise Web Application. It will be comprised of compiled Java classes, other externally designed and acquired Java libraries in the form of .jar files, client side files comprised of HTML, CSS, Javascript as well as other resources (e.g., jpeg, gif, png images) and additional resources for both display and for configuration of the application (usually xml or json files). As we will learn later in the class, to deploy a Java web application (or a .war file) it must satisfy the following structure:

```
WebApplication
  client side files (html, css, javascript)
  /WEB-INF
    web.xml
  /classes
    compiled java code (class files)
  /lib
    java libraries (jar files)
```

The process from development through deployment for an application like this will typically involve several steps including:

1. Creating the web application structure (e.g., building directories)
2. Finding and acquiring any external libraries (jar files) that the project is dependent on
3. Copying the dependent libraries to the WEB-INF/lib directory
4. Compiling all source code
5. Copying the compiled Java files into the WEB-INF/classes directory
6. Copying all the client side files into the application
7. Editing and moving any configuration files (web.xml) to appropriate locations
8. Compressing and packaging the final structure into a single .war file
9. Deploying the .war file to a web application server for execution

Repeating this process from build through deployment repeatedly during application development or trying to repeat it across several developers in a project is not only inefficient, but also prone to errors and inconsistencies.

Much of the process involved in building and deploying web applications can be automated through Integrated Development Environments like Eclipse, IntelliJ IDEA, Netbeans, etc. However, the reliance upon IDEs for building and deploying also brings the dependence on the IDEs for development. As a result the IDE becomes a part of the project that must be used by all developers despite personal preferences.

Build tools provide developers with a means of automating most of the steps necessary in the development through deployment process and establishing that automation through templates shared across development teams. Essentially, build tools allow all developers across projects the ability to run a command on the project shared template to construct, deploy, produce documentation and more with regard to the application. All without impacting each developer's choice of IDE (or editor) used for the actual development of code and configuration.

In summary, most build tools accommodate some, if not all of the following tasks involved in building and deploying software:

- Automate the "daily drudge work" with software
  - Handle simple individual tasks like creating directories, compiling code, etc.
- Manage dependencies
  - Facilitate the identification and downloading of external dependencies (aka jar files) including specified versions.
- Provide local and remote repositories for software
  - External libraries can be acquired from established remote repositories on the internet and downloaded to local repositories for faster future access or access in network disconnected environments.
- Support Multi-module projects
  - Provide a framework for projects to be segmented into build components that may or may not be dependent upon each other. Facilitates operations on a subset of modules in the project.
- Create and publish build "artifacts"
  - Ability to generate executable components in many common Java and Enterprise Java forms (.jar, .war, .ear). Can deploy those artifacts for execution or publish them back to build repositories to make them available to other projects and developers.
- Extendable and configurable to provide "out of the box" capability
  - Several build tools provide 3rd party plugins or permit developers to build their own to incorporate specialized build tasks not included in the original software
- A necessary precursor for project integration with other Continuous Integration (CI) Tools
  - Build automation just one step in a CI environment which essentially is a framework that kicks off building and testing in response to specific times or actions (code merges).

### A Brief History of Popular Build Tools and Java Development

The first build tool was released in 1977 on Unix systems. It was essentially a way to set up shell script templates that perform repeated operations like compilation, library linking, etc. It was designed to support repeated build operations for projects using C and assembly language code. Like most things Unix, it was extremely powerful, but not necessarily designed for a "novice" user. Initially, Make was only good for repeating the individual tasks necessary for building software and creating executable artifacts. For many years it was the only choice for build tools and is still in use in many places today.

When Java was introduced in 1996, it was initially used for desktop applications or web based applets that had simple structure and were easily assembled and executed with command line arguments. However with the introduction of Java in the Enterprise a few years later (1999) the build process became more complex and cumbersome. As a result of that evolution, and the need for something more than Make a new tool called Ant was introduced in 2000. Ant uses xml templates to provide automation for basic operations like directory creation, file copying, Java compiling, etc. It does not support dependency management or incorporate overall build lifecycle management like some of the tools that will follow. It is, however, a bit easier to master than other more complex and recent tools and is often a first choice for "newbies" getting acquainted with Java build tools. In 2004 a dependency management tool called Ivy was first introduced to complement Ant and provide a more complete build capability.

One of today's most popular and powerful Java build tools, Maven, was first introduced in 2002. As will be detailed later in this lecture, it was intended to be a "complete" build tool incorporating every build tool capability discussed earlier including library repositories, dependency management, extensibility, multi-module projects, etc. Like Ant, Maven is based on xml templates called Project Object Modules (POM files). Unlike Ant, Maven is not task oriented. Maven is built on predefined "lifecycles" that perform typical build tasks without a user having to define them (though they may configure them). The operations that Maven executes are based on the selection of lifecycle phases and goals to execute within each phase. There are also many preexisting templates (archetypes) available for developers to generate projects specific to their needs (aka, web application projects, enterprise application projects). Maven is also extensible through available 3rd party plugins that modify and extend the lifecycle steps. Much more detail will be provided about Maven in the remaining sections of the lecture.

In 2012 an alternative to Maven was released for complex Java builds. It is called Gradle. Instead of using XML as its build language, it has its own based on the syntax of the Groovy Programming Language. Like Maven, it utilizes plugins for executing build tasks, but unlike Maven does not provide much basic functionality without the use of plugins. Additionally, while Maven uses lifecycles for phases and goals, Gradle uses acyclic graphs to determine what tasks to run based on the input and output requirements of each. Gradle also provides dependency management, but in a more straightforward way than Maven.

Of the three primary Java build tools, Gradle is currently the most exciting in terms of potential as it has a much smaller learning curve than Maven and offers more opportunity for extended capability through its plugin framework. Despite that potential and the "buzz" that surrounds it, Maven currently has the highest market share of Java developers and projects at roughly 45 % and trending upward. Given that popularity, it makes sense to use that as our entry to educate about Java build tools.

### Maven Overview

As stated in the introduction, Maven was released in 2002. Though written in Java, it can manage projects using a variety of languages and tools including C#, Ruby, Scala, JavaScript. Maven can be used on its own through the command line, however it is nicely coupled with most major IDEs (Eclipse, IntelliJ IDEA, Netbeans and more). Maven utilizes built in life cycles comprised of phases and goals (or tasks) that are individual activities that occur within each phase of a lifecycle.

Common phases in a many Maven build lifecycles include:

1. **validate** the correctness of the projects
2. **check dependencies** needed by project and acquire from local or remote repositories
3. **compile** the project source code into binary artifacts (aka Java class files)
4. **test** the code by executing included unit tests
5. **package** the compiled code, libraries and resources into an archive file (Java .jar, .war, .ear)
6. **verify** that the packaged project is valid
7. **integration-test** end-to-end activities on the packaged application
8. **install** the built project into its deployment environment (web container, etc)
9. **deploy** any project artifact to a remote server or repository for sharing with other projects

As the default Maven capabilities are typically insufficient for more complex projects, 3rd party plugins are publicly available for enhancing or adding goals to the lifecycle phases. Executing a Maven build involves execution with a specified phase of the lifecycle. Maven will then execute all previous phases in the lifecycle up to and including the requested one. So the command:

```
mvn package
```

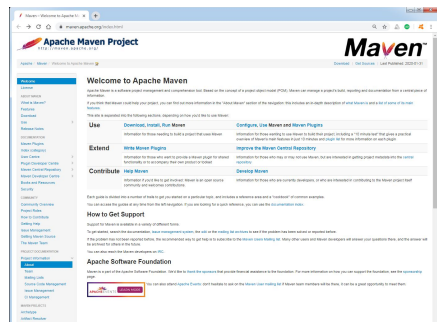
will run the validate, dependency checking, compilation, unit test and package phases (in that order) to generate project artifacts.



The phases mentioned above include tasks that are executed via Maven plugins. Many of the plugins are built into the original Maven install, so there is much you can do "out of the box". Capability is provided for validate, compile, test, package, integration-test, verify, install and deploy. However, Maven capabilities can be extended through a plethora of 3rd party plugins and those living on the edge can even build their own plugins to add or extend Maven capabilities. We will revisit plugins in more detail later in these notes.

## Download and Install Maven

Maven can be downloaded from the Apache website at the following URL:

Once at the home page you can see links to installation, documentation, resources, etc. as illustrated below.



## Downloading Apache Maven 3.6.3

Apache Maven 3.6.3 is a free and open-source software project for project management and build automation across various programming languages, primarily Java. It is a continuation of the Apache Ant project, which is a build automation tool. It is a part of the Apache Software Foundation.

It is a build automation tool that uses a project object model (POM) to manage the build process. It is a part of the Apache Software Foundation.

It is a build automation tool that uses a project object model (POM) to manage the build process. It is a part of the Apache Software Foundation.

### System Requirements

Java 8 (or later) is required for Maven 3.6.3. It is a build automation tool that uses a project object model (POM) to manage the build process. It is a part of the Apache Software Foundation.

It is a build automation tool that uses a project object model (POM) to manage the build process. It is a part of the Apache Software Foundation.

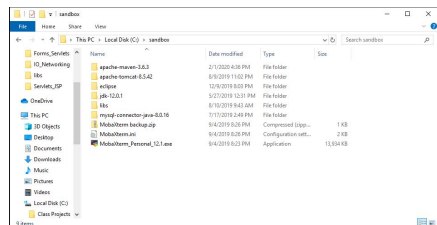
### Files

Apache Maven 3.6.3 is available for download in the following table. It is a build automation tool that uses a project object model (POM) to manage the build process. It is a part of the Apache Software Foundation.

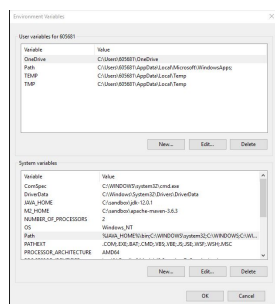
File Name	Size	Checksum	Signature
binaries	1.1 MB	1.1 MB	1.1 MB
Source code	1.1 MB	1.1 MB	1.1 MB
Documentation	1.1 MB	1.1 MB	1.1 MB
Plugins	1.1 MB	1.1 MB	1.1 MB
Repositories	1.1 MB	1.1 MB	1.1 MB
Build tools	1.1 MB	1.1 MB	1.1 MB
Test tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB	1.1 MB	1.1 MB
Reporting tools	1.1 MB	1.1 MB	1.1 MB
Deployment tools	1.1 MB	1.1 MB	1.1 MB
Integration tools	1.1 MB		

[Maven Home Page](#)

The installation instructions are detailed under the first link in the menu, but essentially you have to decompress the downloaded file into a location of your choice and then update your PATH environment variable to include the bin directory under your Maven installation location. For example if installed in the c:\sandbox directory (as illustrated here):



Optionally, you can define a new environment variable for Maven (`M2_HOME`) that represents its install location:



%CD%,  
 %CD%\bin,  
 %CD%\HOME\bin,  
 %CD%\HOME\bin\\*,  
 %CD%\HOME\bin\\*.system32,  
 %CD%\HOME\bin\\*.system32\\*,  
 %CD%\HOME\bin\\*.system32\\*.dlls,  
 %CD%\SYSTEMROOT%\System32\WindowsFontCache\\*,  
 %CD%\SYSTEMROOT%\System32\OpenSSH\

Once done, the installation can be confirmed by logging out, logging back in and typing "mvn -v" at the command prompt.

```
Microsoft Windows [Version 10.0.17763.804]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\l085681>mvn -version

Apache Maven 3.6.3 (cacedd34302096d8abb50b3205418da6a2883f)
Maven home: C:\sandbox\apache-maven-3.6.3\bin..
Java version: 12.0.1, vendor: Oracle Corporation, runtime: C:\sandbox\jdk-12.0.1
Default locale: en_US, platform encoding: cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

C:\Users\l085681>
```

## Maven pom.xml

A Maven Project Object Model (POM) file is an XML file that describes the resources of your project. These are top level elements for project description as well as properties, dependencies and build sections

The POM file describes WHAT to build NOT HOW to build it. How to build it is determined by the Maven phases and goals that are requested for execution. A POM file basically describes a project, and is appropriately named pom.xml. It is associated with your project by being in the root directory of your project, so it defines your project not by its file name, but by its location and content.

All Maven POM files inherit from a Base or Super POM file that contains value and definitions that are passed down to the child POMs. This helps developers minimize the configuration necessary in the project POM files and utilize predefined variables (project paths, etc) that are at their disposal through inheritance.

As will be described later, it is also possible to define sub-projects, each with its own pom.xml file. If you use sub-projects, you can build each sub-project independently, or you can use the top level project to build everything all at once. This at least gives you the flexibility to decide what you want to build.

Here is an example of a minimal POM file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.rfspiegel</groupId>
  <artifactId>my-first-maven</artifactId>
  <version>1.0.0</version>
</project>
```

Every POM file is contained in a *project* tag. The tag contains xml namespace information that references the Super POM and appropriate formatting for the current POM file. The other items in the minimal POM are as follows:

1. a *modelVersion* tag that sets the version of the POM model. That value is mapped to the specific Maven version in user. For Maven 2 and Maven 3, the modelVersion is always 4.0.0
2. a *groupId* element that specifies a unique organizational ID for categorization and organization of individual projects. There is no requirement for the groupId format, but best practice is to follow the same format as used for java packages. Using dot-notation does correlate to a directory structure of the project artifacts within maven repositories. For examples in the present case, the repository will have the following directory path `SMAVEN_REPO/com/rfspiegel/` where this projects artifacts would reside.
3. an *artifactId* element that specifies the specific id for the current project. The artifactId and groupId together should uniquely specify this project from every other project inside and outside of the organization represented by the groupId. The artifactId also extends the repository path of the groupId to the specific project (a sub-directory under the path above) and provides at least a part of the name of the resulting project artifacts.
4. a *version* element contains a defined version number of the project. It actually provides another sub-directory beneath the project for the specific version enabling there to be many project versions stored and maintained in a maven repo. The version number is also used in the actual naming of artifacts along with the artifactId. In this example, if the project generated a jar file, it would be named `my-first-maven-1.0.0.jar` and it would be stored in the `SMAVEN_REPO/com/rfspiegel/my-first-maven/1.0.0` directory.

The groupId, artifactId and version are all required fields for fully defining the project in the Maven universe and navigating its artifacts to a location in a Maven repository.

The default packaging for a Maven artifact is a Java jar file. However, a developer can specify a different kind of artifact using the *packaging* element at the same level of the other elements above. The other types of artifacts that can be generated include pom, jar, maven-plugin, ejb, war, ear and rar. The type of packaging identified with dictate what lifecycle phases are executed on the project.

As mentioned earlier, a project POM will extend a Super POM file that is used by specific versions of Maven. If we were to look at that file, we could get a basic understanding of some of the default properties and behaviors that are present, but not explicitly specified, in project POM files. You can see an overview of the Super POM for maven model version 4.0.0 on the maven pages on the Apache web site to get an idea of the kinds of things that are "prepacked" in pom files that inherit from that [Super POM](#).

## Parent POMs and Inheritance

In addition to these individual POM files, there is the notion of inheritance within a project hierarchy. As just mentioned, Maven POM files inherit from a Super POM file, or if no Super POM is defined, from the base project POM. You can specify which POM a POM inherits from, that way a change in a higher level POM will propagate downwards through the projects.

You define a Super POM by using the "parent" attribute as follows in your POM file

```
<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>
  <parent>
    <groupId>com.rfspiegel</groupId>
    <artifactId>parentDef</artifactId>
    <version>1.0.0</version>
    <relativePath>../parent-path</relativePath>
  </parent>
  <artifactId>my-first-maven</artifactId>
  <version>1.0.0</version>
</project>
```

The sub-project POMs will use this definition, unless they choose to override a specific attribute.

It turns out you can view what the "Effective POM" is, which is the resulting POM after all inheritance is done. You can see that by typing the command

```
mvn help:effective-pom
```

Which will write out the actual POM that the project will use.

## Standard Maven Directory Structure

The standard Maven directory structure is as follows

```
/src
/main
/java
/resources
/webapp
/test
/java
/resources
/target
```

So, "src" above is somewhat self explanatory, this is where the source code for your project lives. The "main" directory contains the functional code, while the "test" directory contains test source code. The "java" directory contains your basic java source code, while the "resources" directory contains other resources your app needs to be built (like external jar files)

The "webapp" directory contains code for a Java Web Application. This turns out to be the root directory of the web application you build, so you would find things like WEB-INF here.

The "target" directory is created by Maven, and contains

- Compiled Java Classes
- JAR or WAR Files produced by Maven

## Dependencies

Typically, you will not be writing stand-alone apps, so you will have dependencies (typically bundled in their own JAR files or libraries). These files need to be in a Java CLASSPATH in order to compile the project. One issue (with advantages and disadvantages) is that once you decide to use an external library/JAR, under most tools it isn't updated when a new release comes out. What this means is that a large project may contain several old, non-updated libraries, which may contain bugs or worse, security issues. The larger the project, the more that this is an issue. Another issue is that the JAR you need may also depend on more JARS, which need to be manually included in the project.

It turns out the people who built Maven decided to come up with a solution for this. Maven has built-in dependency management. This lets you specify in your POM which libraries your project needs, and which version. If any of these libraries need other libraries, Maven will also download them and put them in your local Maven repository. Maven will not automatically update your library version, as that is specified in the POM file.

So, if we want to add a dependency section to the POM file, you include it after the "version" line.

```
<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.rfspiegel</groupId>
  <artifactId>my-first-maven</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>5.6.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

```
<dependency>
<groupId>com.jhuep.someJar</groupId>
<artifactId>someJar</artifactId>
<version>1.2.3</version>
</dependency>
</dependencies>
<build>
</build>
</project>
```

So, each "dependency" element in the "dependencies" section describes an external dependency. Each element has descriptors similar in function to the original POM descriptors.

What is important is where Maven gets these dependencies. It doesn't get them from anywhere, it downloads them from a central Maven repository and then they are cached in your local Maven repository. Once they have been cached locally, future builds will just use the local copies. It turns out that not \*everything\* is available in the central Maven repository, so you always have the option to download the library yourself and put it into your local Maven repository. The key here is to use the same directory structure that was shown with the initial example...use the groupId, artifactId, and version to build the appropriate path. So, for the second entry "someJar", we would load the jar file into

```
<directory path of Maven repository>/com/jhuep/somejar/1.2.3
```

## External Dependencies

An external dependency is something that is not located in your local Maven repository, but somewhere else on your hard disk (like in the lib directory of a webapp). "External" means that it is not in a Maven Repository. You define an external dependency by adding a "systemPath" attribute.

```
<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.rfspiegel</groupId>
  <artifactId>my-first-maven</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>5.6.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.jhuep.someJar</groupId>
      <artifactId>someJar</artifactId>
      <version>1.2.3</version>
    </dependency>
    <dependency>
      <groupId>externalDep</groupId>
      <artifactId>externalDep</artifactId>
      <version>3.1</version>
      <systemPath>${project.basedir}\war\WEB-INF\lib\mydependency.jar</systemPath>
    </dependency>
  </dependencies>
  <build>
  </build>
</project>
```

The `${project.basedir}` expression points to the directory where the POM is located.

## Snapshot Dependencies

Snapshot dependencies are libraries that are under development. Instead of constantly updating version numbers, you can "depend" on a SNAPSHOT version of the project. These are always downloaded (or kept) in your local repository or project space. They are always updated even if a cached copy exists in the Local Repository ensuring you have the latest copy.

You label your project as a SNAPSHOT version by simply appending -SNAPSHOT to the version number when configuring dependencies.

```
<dependency>
<groupId>com.rfspiegel</groupId>
<artifactId>snapshot-example</artifactId>
<version>1.0-SNAPSHOT</version>
</dependency>
```

## Maven Repositories

So, what are Maven Repositories? Basically, they are JAR files in a given directory structure that contain some extra Meta-data, which in turn are just more POM files that describe each project the JAR file belongs to, including any more external dependencies.

There are three types of Maven repositories

**Local Repository** - This is the current set of dependencies that your project uses. They may have been downloaded from the following two repository types, or you may have manually inserted your own entry. This repository is actually used by all of your projects, so there is no duplication of data. Since your own project lives in this repository, it can be used as a dependency for other projects you may create. Maven will put the local repository in your "user" home directory unless you tell it otherwise.

You can specify a different location for your local repository by making an entry in your home directory, by going to the .m2 directory (remember, in \*nix this will be hidden) and editing a settings.xml file. Here is an example of specifying a different location

```
<settings>
  <localRepository>
    c:/Users/Public/mavenRepo
  </localRepository>
</settings>
```

**Central Repository** - This is a repository provided by the Maven community. This is the default location for any dependencies listed in your project. Everyone has access to this repository and no special permissions are needed.

**Remote Repository** - This is a repository hosted by a web server at another location. This is often used by a company to host standard project for company-wide access. Like a Central Repository, any dependencies found in a Remote Repository are downloaded to your Local Repository. You define Remote Repositories in your POM file by adding them after the "dependencies" section: Here is an example of specifying a different location

```
<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.rfspiegel</groupId>
  <artifactId>my-first-maven</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>5.6.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.jhuep.someJar</groupId>
      <artifactId>someJar</artifactId>
      <version>1.2.3</version>
    </dependency>
    <dependency>
      <groupId>externalDep</groupId>
      <artifactId>externalDep</artifactId>
      <version>3.1</version>
      <systemPath>${basedir}\war\WEB-INF\lib\mydependency.jar</systemPath>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <id>com.rbevans</id>
      <url>http://maven.rfspiegel.com/maven2/lib</url>
    </repository>
  </repositories>
  <build>
  </build>
</project>
```

## Properties

Value placeholders that are accessible from anywhere in pom.xml or poms of sub-modules. This allows a single point of change, changing a dependency version in one place vice throughout project poms.

To define properties:

```
<properties>
  <spring.version>4.3.5.RELEASE</spring.version>
</properties>
```

To access the property within a POM file, just wrap the property name in `${...}`:

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>${spring.version}</version>
</dependency>
```

Maven already has several defined properties, we've already seen the `${project.basedir}` property. You can look at the Apache documentation to see the other properties available.

## Building a Project

You probably have noticed the "build" elements in some of the above examples. Build is where specific plugins (build functionality) is incorporated along with specific build meta information:

Maven uses a life cycle when building projects. So, the build cycle is actually composed of several build phases, and each phase contains build goals.

Maven already has three build-in build cycles, these are

1. default
2. clean
3. site

Each of these handles a different set of actions when building a project, and each is independent of the others. The "default" build cycle handles the normal routines of compiling and packaging a Maven project. For those of your familiar with "make", the clean build cycle accomplishes the same task, removing temporary files created during the default build cycle. The "site" build cycle is used to create documentation for your project, up to creating a complete website that uses your project.

Every maven build follows a specified lifecycle where steps in the lifecycle are called phases. Important phases include:

- validate - checks the correctness of the projects check dependencies needed by project and acquire them
- compile - ? compiles the provided source code into binary artifacts
- test - executes unit tests
- package - packages compiled code into an archive file (jar, war)
- integration-test - executes additional tests, which require the packaging
- verify - checks if the package is valid
- install - installs the package file into the local Maven repository
- deploy - deploys the package file to a remote server or repository for sharing with other projects

To execute a specific phase of the build, pass its name to the "mvn" command

```
mvn <phase-name> (e.g. maven package)
mvn phase1 phase2 - . . . to run multiple phases
mvn phase:goal to run a specific goal within a phase
```

If a build phase is requested, all build phases prior to it in the pre-defined sequence of phases are executed as well.

```
<build>
<defaultGoal>install</defaultGoal>
<directory>${basedir}/target</directory>
<finalName>${artifactId}-${version}</finalName>
<filters>
<filter>filters/filter1.properties</filter>
</filters>
</build>
```

## Maven Build Plugins

So, we've already seen some plugins (the lifecycle options). According to Apache, Maven is basically a plugin execution framework. So there are a lot of tasks already built in to Maven that you can use.

In this case, I'm not going to copy over the large table of plugins, but you can view it at <https://maven.apache.org/plugins>

Build uses maven plugins which are also available in specified repositories (and can be downloaded to the local repo). The remote repos are distinct from others that contain general libraries.

Every Plugin is a collection of one or more goals (A goal is a unit of work in Maven) goals are attached to a life cycle phase

Plugin functionality is broken into goals which are executed in specific phases of the build. So each phase becomes a sequence of goals executed from designated plugins (plugins add "extra" goals into a build phase beyond standard Maven build phases and goals)

Maven plugins enable you to add your own actions to the build process, just specify the goal and what lifecycle it should run in:

```
<plugin>
<groupId>com.rfspiegel.foo</groupId>
<artifactId>rfspiegel-plugin</artifactId>
<version>1.0</version>
<executions>
<execution>
<phase>verify</phase>
<goals>
<goal>checklinks</goal>
</goals>
</execution>
</executions>
</plugin>
```

## Creating Maven Projects

There are actually two ways to create Maven Projects. The first is to use something called "Maven Archetypes", which are basically templates that help you quickly define the correct directory and POM structure for a project. It turns out there are a LOT of archetypes available (~1300), so it is rare that you need to create a project manually.

For example, some common Archetypes would be:

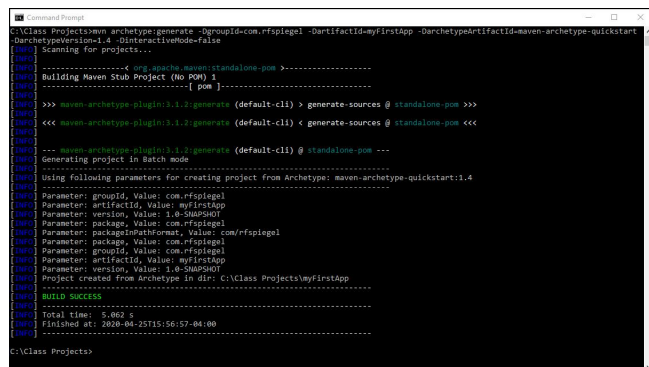
Archetype ArtifactIds	Description
maven-archetype-archetype	Create sample project
maven-archetype-quickstart	Java App
maven-archetype-j2ee-simple	Simplified sample J2EE app
maven-archetype-webapp	Sample Maven Webapp project

So, we will use the "maven-archetype-quickstart" archetype to create a project that will support the classic "Hello World" java application. You can do so by executing the following command (it's a single line command):

```
mvn archetype:generate -DgroupId=com.rfspiegel -DartifactId=myFirstApp -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

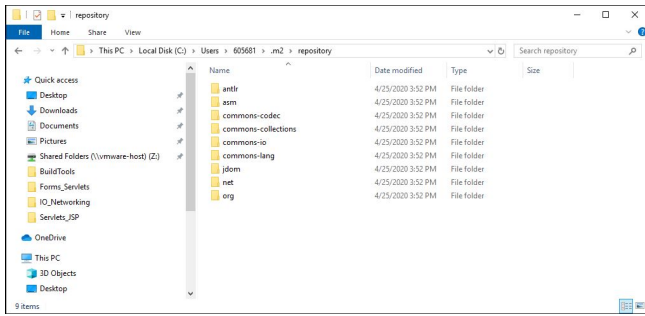
Apache notes that if this is the first time you have run Maven, it may take a while for this to complete, that is because Maven is creating your local repository and downloading the most recent artifacts.

So, if you run the command in a command window (Windows) or shell (Mac/\*nix), you should see something like this:

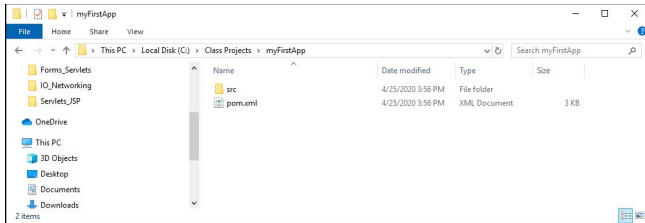


```
C:\Class Projects>mvn archetype:generate -DgroupId=com.rfspiegel -DartifactId=myFirstApp -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
[INFO] Scanning for projects...
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----[ pom ]-----
[INFO] >>> maven-archetype-plugin:1.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO] <<< maven-archetype-plugin:1.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO] --- maven-archetype-plugin:1.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in batch mode
[INFO] -----
[INFO] Using following parameters for creating project from archetype: maven-archetype-quickstart:1.4
[INFO] Parameter: groupId, Value: com.rfspiegel
[INFO] Parameter: artifactId, Value: myFirstApp
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.rfspiegel
[INFO] Parameter: packageInPathFormat, Value: com/rfspiegel
[INFO] Parameter: package, Value: com.rfspiegel
[INFO] Parameter: groupId, Value: com.rfspiegel
[INFO] Parameter: artifactId, Value: myFirstApp
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from archetype in dir: C:\Class Projects\myFirstApp
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 5.062 s
[INFO] Finished at: 2020-04-25T15:50:57-04:00
[INFO] -----
C:\Class Projects>
```

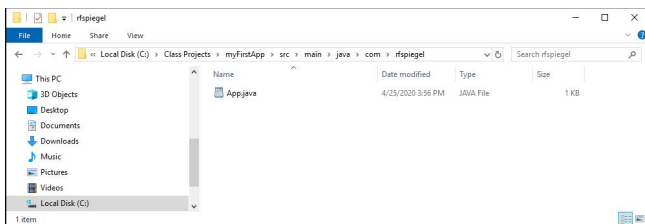
Let's look at what running this command has created. First, we now have our local repository in the expected .m2 directory in our home directory!



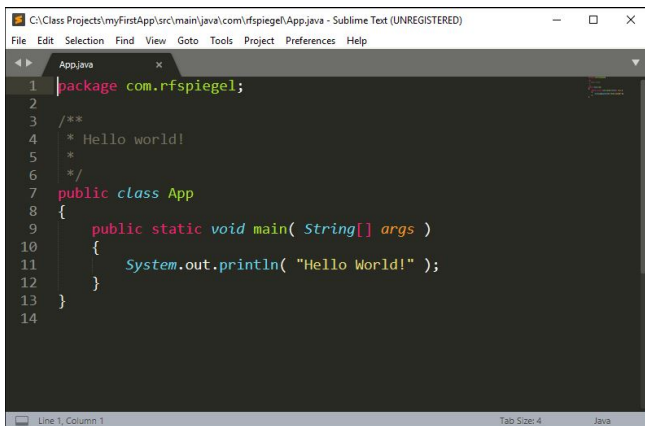
We also have our application in our home directory as well



This archetype creates a standard App.java file to serve as a starting point for your app.



And, luck is with us!!! If we open App.java in an editor, we see that it has by default created that sought after Hello World application!



Now, you might think that running the "mvn package" command, which states that it produces a distributable format, such as a JAR would give you a JAR you could run.

Unfortunately, it doesn't include a manifest in the resulting JAR, so it isn't runnable.

To get it into runnable format, you have several options. One of the easiest is to use is the **Apache Maven Assembly Plugin**. The allows you to aggregate the project output along with all of its modules, dependencies, documentation into a single runnable JAR file.

There is only one goal in the assembly plugin, and that is the "single" goal. All former goals have been deprecated. Lets look at the pom.xml file that was generated by our archetype command, with two small additions (look for the "added to take us from the quickstart" comments). Configuring the maven-assembly-plugin defines the Manifest data required to make the JAR runnable.

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.rfspiegel</groupId>
  <artifactId>myFirstApp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <!-- added to take us from the quickstart configuration to a runnable jar -->
  <packaging>jar</packaging>
  <!-- End of mod -->
  <name>myFirstApp</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to parent pom) -->
      <plugins>
        <!-- clean lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#clean_Lifecycle -->
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>3.1.0</version>
        </plugin>
        <!-- default lifecycle, jar packaging: see https://maven.apache.org/ref/current/maven-core/default-bindings.html#Plugin_bindings_for_jar_packaging -->
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

```

</plugin>
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.1</version>
</plugin>
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
</plugin>
<plugin>
  <artifactId>maven-install-plugin</artifactId>
  <version>2.5.2</version>
</plugin>
<plugin>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.8.2</version>
</plugin>
<!-- site lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#site_Lifecycle -->
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.7.1</version>
</plugin>
<plugin>
  <artifactId>maven-project-info-reports-plugin</artifactId>
  <version>3.0.0</version>
</plugin>
<!-- added to take us from the quickstart configuration to a runnable jar -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <archive>
      <manifest>
        <mainClass>com.rfspiegel.App</mainClass>
      </manifest>
    </archive>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<!-- End of mod -->
</plugins>
</pluginManagement>
</build>
</project>

```

At this point, if we run this POM we first tell Maven to compile the code, then to run the assembly phase with the "single" goal as follows:

```
mvn compile assembly:single
```

The first time you do this, you will see a lot more output as more dependencies are loaded from the Central Repository, but this is what you should get

```

C:\Class Projects>mvn compile assembly:single
[INFO] Scanning for projects...
[INFO]
[INFO] --- maven-assembly-plugin:3.0.0-jar:jar (default-jar) @ myFirstApp ---
[INFO] Building myFirstApp 1.0-SNAPSHOT
[INFO]
[INFO] --- maven-assembly-plugin:3.0.0-jar:jar (default-jar) @ myFirstApp ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Class Projects\myFirstApp\src\main\resources
[INFO]
[INFO] --- maven-assembly-plugin:3.0.0-jar:jar (default-jar) @ myFirstApp ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-assembly-plugin:3.0.0-jar:jar (default-jar) @ myFirstApp ---
[INFO] Building jar: C:\Class Projects\myFirstApp\target\myFirstApp-1.0-SNAPSHOT-jar-with-dependencies.jar
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 4.894 s
[INFO] Finished at: 2020-04-25T16:14:35-04:00
[INFO]
C:\Class Projects\myFirstApp>

```

Note that once built, I can run the jar file from the command line!

## Multi-module projects

In a multi-module project, there is a parent project/pom and sub-projects/modules that have poms that inherit from parent. The idea of a multi-module project is to have one POM that controls all life cycles, and then modules (sub-projects) that actually build a JAR/WAR.

You can easily create a "parent" POM by

```
mvn archetype:generate -DgroupId=com.rfspiegel -DartifactId=parent-project
```

Next update pom to contain <packaging>pom</packaging> to indicate its a parent module

```

<?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/xsd/maven-4.0.0.xsd">
  <groupId>com.rfspiegel</groupId>
  <artifactId>parent-project</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Multi Chapter Simple Parent Project</name>
  <modules>
    <module>core</module>
    <module>service</module>
    <module>webapp</module>
  </modules>
</project>

```

You will note that there is also a "modules" attribute, that defines the multiple modules you wish to control. For this example, they are named core, service and webapp. You then create the other module projects:

```

cd parent-project
mvn archetype:generate -DgroupId=com.rfspiegel -DartifactId=core
mvn archetype:generate -DgroupId=com.rfspiegel -DartifactId=service
mvn archetype:generate -DgroupId=com.rfspiegel -DartifactId=webapp

```

This will result in the following projects

1. core
2. service
3. webapp

Inside each submodule the POM for each will have a parent section:

```

<parent>
  <groupId>com.rfspiegel</groupId>
  <artifactId>parent-project</artifactId>
  <version>1.0</version>
</parent>

```

For dependency Management in Multi-module project, you put dependency info in parent pom

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.3.5.RELEASE</version>
    </dependency>
    //...
  </dependencies>
</dependencyManagement>

```

Reference dependency in submodule poms using only groupId and artifactId

```
<dependencies>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
</dependency>
//...
</dependencies>
```

You can override version (if a submodule needs different version) just by adding version info in submodule dependency