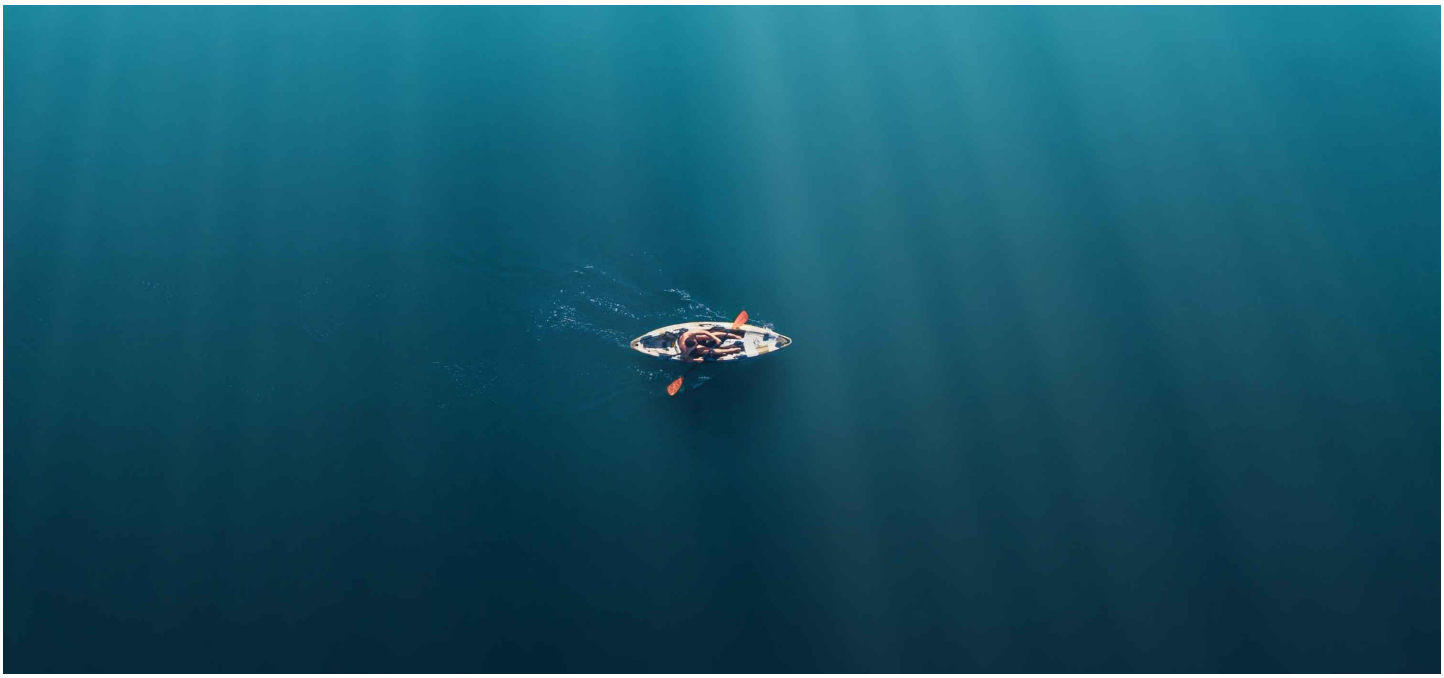


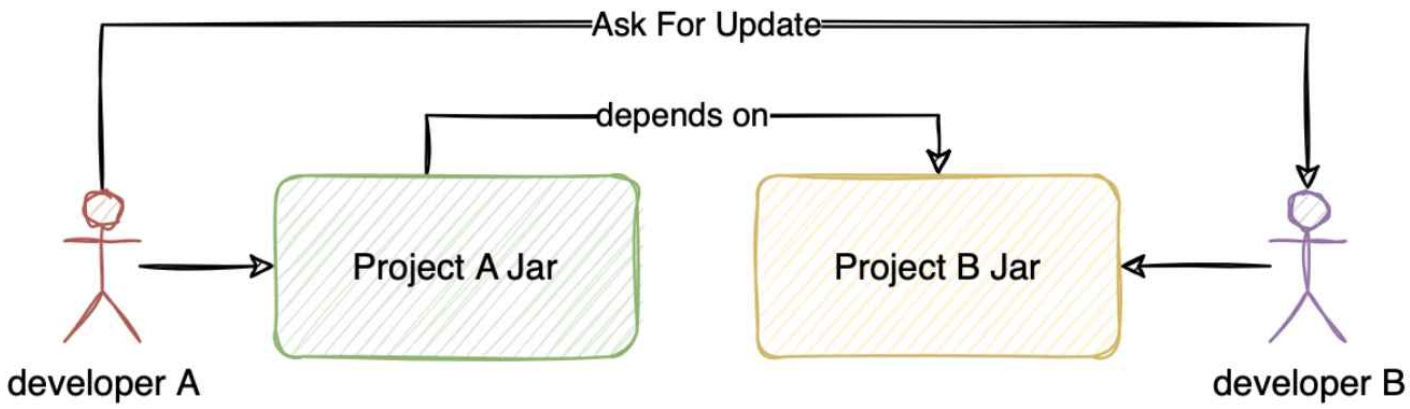
38-Maven

Author: Vincent Lau

Learning Objectives

- Understand what Maven is and why it benefits the development cycle
- Understand the components that a POM file consists of

The World Without Maven



Considering a scenario where we are developing two Java projects, referred to as A and B, where some functionalities in project A depend on certain classes in project B, **raises the question of how to manage this dependency relationship**.

During development

- When someone (developer B) else has written the functionality code, they package project B into a JAR file. Subsequently, this JAR file is imported into the library of project A. By doing so, project A gains the ability to utilize certain classes from project B.
- If any bugs are discovered in project B, it becomes necessary to make the required fixes, repackage project B, and recompile project A.

After development

- Upon completing the development of project A, to ensure its proper functioning, it needs to depend on project B, similar to how one jar file relies on another when used.

Two Approaches (without Maven)

- **The first approach: We publish project B to the public and inform all developers that they must import B's JAR file, when using A's class.**
  - **This approach creates lots of communication overhead and documentation is required to describe the relationship among the Jar Projects**. Don't forget the dependency among projects will be updated after day 1, which is a complicated maintenance process).
- **The second approach: Involves packaging project B into project A.**
  - However, this can result in resource wastage. For example, if developers are concurrently working on another project (Project C) that relies on Project A and Project B (where Project A depends on Project B as mentioned above). **This results in two copies of B.jar, which potentially causes Project C to be packaged in 2 different versions of B.jar.**

The approaches above pertain to dependency issues between projects, which can be cumbersome and **inconvenient to manage manually**. Therefore, Maven is employed to assist with this management process.

What is Maven?

Maven is a build automation tool used primarily for Java projects. Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project.

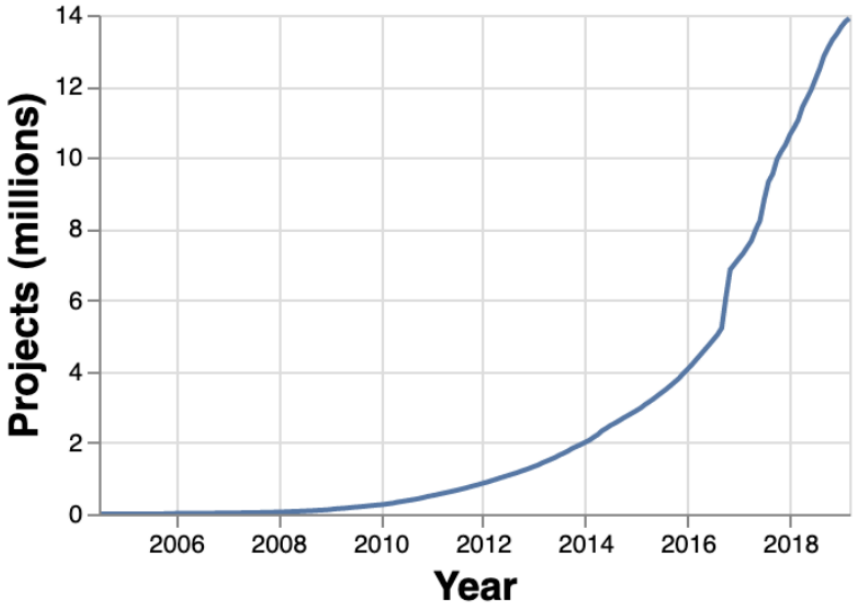
Maven addresses two aspects of building software: how software is built, and its dependencies.

Unlike earlier tools like Apache Ant, it uses conventions for the build procedure. Only exceptions need to be specified. **An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins.** It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. **Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache.** This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated.

Maven is built using a plugin-based architecture that allows it to make use of any application controllable through standard input. A C/C++ native plugin is maintained for Maven 2.

Alternative technologies like Gradle and sbt as build tools do not rely on XML, but keep the key concepts Maven introduced. With Apache Ivy, a dedicated dependency manager was developed as well that also supports Maven repositories.

History



Maven, created by Jason van Zyl, began as a sub-project of Apache Turbine in 2002. In 2003, it was voted on and accepted as a top level Apache Software Foundation project.

In July 2004, Maven's release was the critical first milestone, v1.0.

Maven 2 was declared v2.0 in October 2005 after about six months in beta cycles.

Maven 3.0 was released in October 2010 being mostly backwards compatible with Maven 2.

Maven 3.0 information began trickling out in 2008. After eight alpha releases, the first beta version of Maven 3.0 was released in April 2010. Maven 3.0 has reworked the core Project Builder infrastructure resulting in the POM's file-based representation being decoupled from its in-memory object representation. This has expanded the possibility for Maven 3.0 add-ons to leverage non-XML based project definition files. Languages suggested include Ruby (already in private prototype by Jason van Zyl), YAML, and Groovy.

Special attention was given to ensuring backward compatibility of Maven 3 to Maven 2. For most projects, upgrading to Maven 3 will not require any adjustments of their project structure. The first beta of Maven 3 saw the introduction of a parallel build feature which leverages a configurable number of cores on a multi-core machine and is especially suited for large multi-module projects.

From Wikipedia, the free encyclopedia

The POM File (Project Object Model)

A Project Object Model (POM) provides all the configuration for a single project. General configuration covers the project's name, its owner and its dependencies on other projects. One can also configure individual phases of the build process, which are implemented as

plugins. For example, one can configure the compiler-plugin to use Java version 1.5 for compilation, or specify packaging the project even if some unit tests fail.

Project Inheritance
POMs can also inherit configuration from other POMs. All POMs inherit from the Super POM by default. The Super POM provides default configuration, such as default source directories, default plugins, and so on.

Project Aggregation
Larger projects should be divided into several modules, or sub-projects, each with its own POM. One can then write a root POM through which one can compile all the modules with a single command.

- A Maven project is configured via a ***Project Object Model (POM)***, represented by a *pom.xml* file.
- The POM describes the project, manages dependencies, and configures plugins for building the software.
- A typical *pom.xml* looks like:

```
<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.bootcamp.demo</groupId>
    <artifactId>maven-demo-app</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>

    <name>maven-demo</name>
    <url>http://maven.apache.org</url>

    <properties>
        <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>4.12</version>
            <scope>test</scope>
        </dependency>
```

```
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>${maven.compiler.source}</source>
                    <target>${maven.compiler.target}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Project Identifiers

- Maven uses a set of identifiers to uniquely identify a project, and specify how the project artifact should be packaged:
  - `groupId` a unique base name of the company or group that created the project
  - `artifactId` a unique name of the project
  - `version` a version of the project
  - `packaging` a packaging method (e.g. *JAR / WAR / ZIP*)
- The first three identifiers (*groupId:artifactId:version*) combine to form a unique identifier for a project. It is also used to identify external dependencies and specify which versions to use.

Properties

- Properties are accessible anywhere within a *pom.xml.* They make your *pom* file easier to read and maintain.
- We can define properties in the *pom* file, and use them with the notation `*${some.property}*`.

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<!-- other tags -->

```

```
<configuration>
    <source>${maven.compiler.source}</source>
    <target>${maven.compiler.target}</target>
</configuration>
```

Dependencies

- External libraries used by a project are called *dependencies*. Maven automatically downloads these dependencies from a central repository, avoiding the need to store them locally.
- Benefits of the **dependecy management** feature in Maven are:
  - **Consume less storage space** by downloading dependencies only when needed
  - **Make checking out a project quicker**
  - **Provide a medium to exchange binary artifacts everywhere** without the need to build artifact from source every time

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.6.3</version>
</dependency>
```

- The above XML tags declare the *spring-boot-starter* dependency which is uniquely identified by the *groupId, artifactId, and version*.
- As Maven processes the dependencies, it will download the *spring-boot-starter* library into your local Maven repository.

Artifact Coordinates

groupId

A grouping classification, typically referring to an organization, a company and may include a basic theme for one or more projects. A groupId typically follows a dot notation similar to a Java package name. Each token in the dot notation corresponds to a directory in a tree structure on the repository.

For instance, a `groupId of org.apache.commons` corresponds to `$REPO/org/apache/commons`.

artifactId

A proper name for the project. Among the many projects that exist in the group, the artifactId can uniquely identify the artifact. An artifactId follows a simple name nomenclature with hyphenation recommended for multi-word names. Names should ideally be small in length. An artifact manifests itself as a sub-directory under directory tree that represents the groupId.

For instance, an `artifactId of commons-lang3` under a `groupId of org.apache.commons` would determine that the artifact can be found under : `$REPO/org/apache/commons/commons-lang3/` .

version

An identifier that tracks unique builds of an artifact. A version is a string that is constructed by the project' s development team to identify a set of changes from a previous creation of an artifact of the same project. It is strongly recommended to follow semantic versioning schemes for versions, although it is not mandated. A version manifests itself as a sub-directory under the directory tree that represents the groupId and artifactId.

For instance, a `version of 3.1.0` for an `artifactId commons-lang3` under the `groupId of org.apache.commons` would determine that the artifact would be located under: `$REPO/org/apache/commons/commons-lang3/3.1.0/` .

Dependency Scopes

- There are two types of dependencies in Maven - **`direct`** and **`transitive`** .
- Direct dependencies require transitive dependencies. Maven automatically includes required transitive dependencies in our project.
- Use the `mvn dependency:tree` command to list all dependencies including transitive ones in a project.
- Maven has various dependency scopes to specify when to include each dependency:
  - ***Compiled*** included during compile time and runtime (**default scope** when no other is provided).
  - ***Provided*** included during compile time but not packaged with the artifact, because it is expected to have been provided at runtime by other sources, e.g. JDK or a container.
  - ***Runtime*** included during runtime, but not compile time.
  - ***Test*** included only during the test phase of the build cycle.

Build

- The default build section looks like this:

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

- **defaultGoal** - The default goal or phase to execute if none is given.
- **directory** - The directory where the build dump its files, which defaults to `${basedir}/target`.
- **finalName** - This is the name of the bundled project when it is finally built (without the file extension, e.g `my-project-1.0.jar`). It defaults to `${artifactId}-${version}`.
- **filter** - Define `*.properties` files that contain a list of properties that apply to resources which accept their settings.

Maven - POM Reference

Default Plugins

There are several **built-in plugins and default configurations that are pre-configured to support typical Java development tasks**. Here's a list of some of the key plugins that are commonly included and pre-configured:

maven-compiler-plugin

As previously discussed, this plugin is responsible for compiling Java source code. It's usually pre-configured to match the Java version you've specified in your project.

maven-resources-plugin

This plugin handles the copying of resources (e.g., property files, XML files) into the target directory.

maven-surefire-plugin

This plugin is responsible for executing tests. It's commonly used for running unit tests and is pre-configured to use the JUnit testing framework.

maven-jar-plugin

This plugin is used to create JAR (Java Archive) files from compiled classes. By default, it creates a JAR file for the project.

maven-install-plugin

This plugin allows you to install your project's artifacts into your local repository. It's used when you run `mvn install` to make your project's artifacts available to other projects.

maven-clean-plugin

This plugin provides a goal to clean the target directory, removing all build output.

maven-site-plugin

This plugin is used to generate documentation and reports for your project. It's often used for generating project websites.

maven-assembly-plugin

This plugin is used for creating binary distributions of your project, such as ZIP or TAR files containing the compiled code and resources.

maven-release-plugin

This plugin assists in releasing versions of your project, including versioning, tagging, and releasing to a remote repository.

Please note that the specific version and behavior of these plugins can vary depending on your project's configuration, the Maven version you're using, and any customizations you may have made in your `pom.xml` .

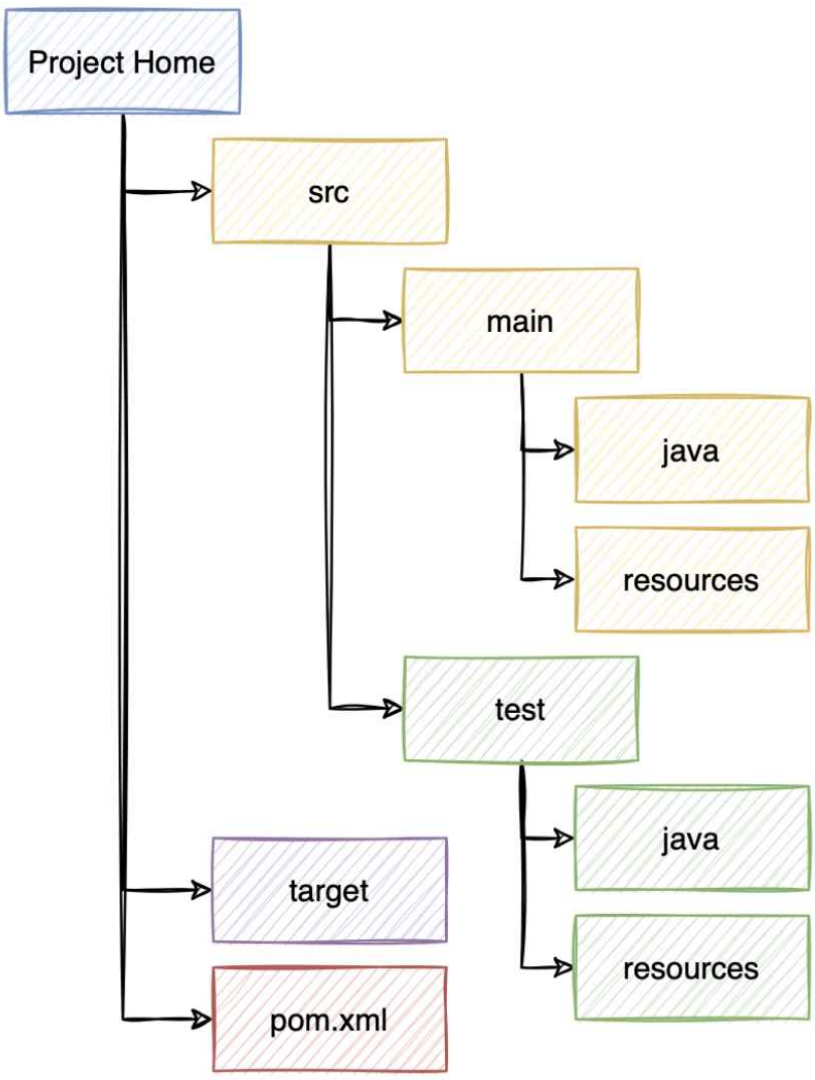
While these plugins are commonly included and pre-configured, you can customize their settings (such as overwritten different versions) or add additional plugins to meet your project's specific requirements.

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.8.0</version>
            </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>
        </plugins>
    </pluginManagement>
</build>
```

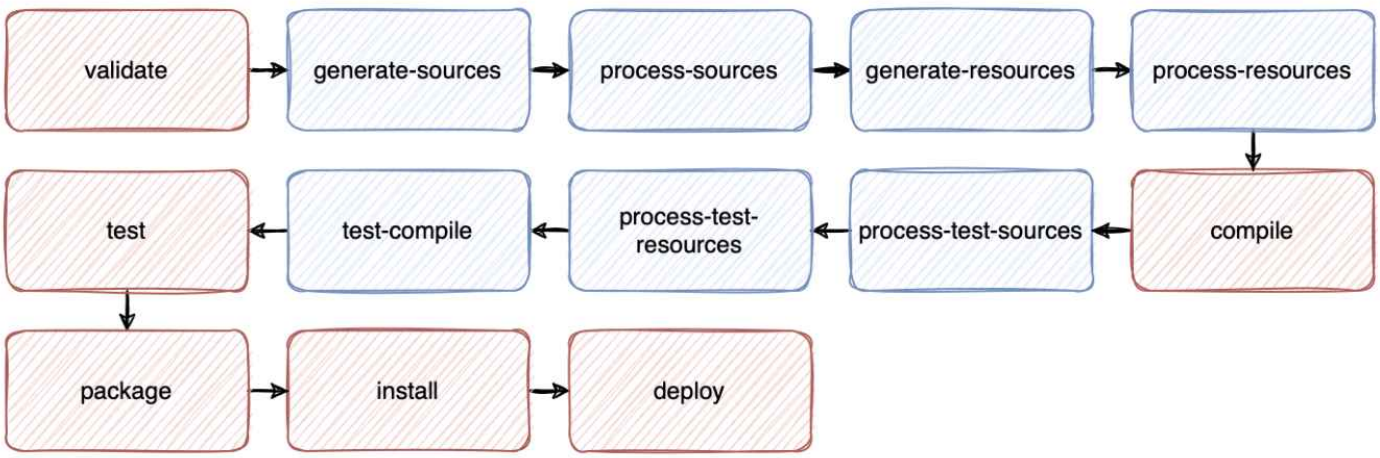
Project Structure (Normal, module based will be different)

A directory structure for a Java project auto-generated by Maven:



Directory	Purpose
project home	Contains the pom.xml and all subdirectories
src/main/java	Contains the deliverable Java sourcecode for the project
src/main/resources	Contains the deliverable resources for the project, such as property files
src/test/java	Contains the testing Java sourcecode (JUnit or TestNG test cases, for example) for the project
src/test/resources	Contains resources necessary for testing
target	Complied java class files
pom.xml	Define project build/ dependency/ release process, refer to pom section

Build lifecycles



Phase	Description
clean	mvn clean: 清空编译结果，为下一次编译做准备。
compile	mvn compile: 编译主程序源文件，Maven将源文件分为主程序源文件和测试程序源文件，他们放在不同的目录。
test	mvn test: 编译测试程序源文件。

package	mvn package: 将主程序编译后的class文件、资源文件以及配置文件，打包为对应的类型(jar包或war包，需要在pom.xml中设置)。
install	mvn install: 将打包后的文件按照一定规则(groupId+artifactId的方式)安装到本地仓库中。
deploy	mvn deploy: 部署程序。

Installation

- Install the latest version of Maven from Apache Maven.
- Verify the installation has been successful by typing this in the terminal:

```
mvn --version
```

- You should see something like this if Maven is installed correctly:

```
Apache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
Maven home: D:\apache-maven-3.8.4
Java version: 11.0.3, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-11.0.3
Default locale: en_US, platform encoding: MS950
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Creating a Maven Project

- There are two ways to create a Maven project - via an *Integrated Development Environment (IDE)*, such as *Intellij*, or the *Command Line*.

VSCode

- Start a new project > Select "Maven" > Next
- [Optional] Create an archetype > Next
- Fill in the *Name, Location, GroupId, ArtifactId,* and *Version* > Finish

Basic Project Structure

- The generated project structure will look something like this:

```
maven-demo
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- bootcamp
    |               `-- demo
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- bootcamp
                    `-- demo
                        `-- AppTest.java
```

The POM File

- The *pom.xml* file should look similar to this:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.bootcamp.demo</groupId>
    <artifactId>maven-demo</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>maven-demo</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

- As you can see, the *junit* dependency is provided by default.

Compiling and Packaging a Project

- The next step is to compile the project:

```
mvn compile
```

- Maven will run through all *lifecycle* phases required by the *compile* phase to build the project's sources. If you want to run only the *test* phase, you can use:

```
mvn test
```

- To invoke the *package* phase and produce the compiled archive *jar* file, you can use:

```
mvn package
```

Building a Fat JAR with Maven

- A fat JAR means a single JAR file that contains all the compiled Java classes from your project, as well as the JAR files of all the external dependencies your project depends on.
- Fat JARs are handy when you need to build a standalone executable JAR file. This is especially useful when we need to package an application inside a Docker container.

Build a Fat JAR With Maven

Executing a JAR File

Command Line with Maven

- Configure the *exec-maven-plugin* in the *pom.xml:*

```xml
<plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>3.0.0</version>
          <configuration>
              <mainClass>com.bootcamp.demo.App</mainClass>
          </configuration>
</plugin>
```

- Then, execute this plugin goal:

```
mvn exec:java
```

Command Line without Maven

```
java -jar maven-demo.jar
```