Contents

[What is Maven? 1](#_Toc123294492)

[Why use Maven? 1](#_Toc123294493)

[Creating a Maven project 1](#_Toc123294494)

[Project Structure 2](#_Toc123294495)

[POM File 3](#_Toc123294496)

[Maven Build Lifecycle 3](#_Toc123294497)

## **What is Maven?**

- Maven is a build tool and package manager

- Created by Apache to reduce the complexity of Java project builds

<https://maven.apache.org/>

- Replaced its predecessor Ant due to its easier to use abilities

- Ant = “Another Neat Tool” for build automation and was very customizable

- Article on Ant vs Maven vs Gradle: <https://www.baeldung.com/ant-maven-gradle>

- Gradle was a project that came after Maven to fix some of the limitations of Maven

- It streamlines the build process

- It helps manage our dependencies

- Utilizes a POM (Project Object Model ) to represent the entire project

- Encourages best practices as you must pass all the tests to build the project

## **Why use Maven?**

- Prevents us from having to manage our dependencies ourselves, we don’t have to go search for third party websites to download their code

- A dependency is a third-party library that our codebase needs to function

- It houses all these third-party libraries in one central repository ([Maven Central Repository](https://mvnrepository.com/))

- It is the most widely used build tool in the Java ecosystem (#2 is Gradle)

- When building the project, it will compile and zip our code into small packages (JAR files, WAR files)

- JAR stands for Java ARchive file and it's essentially a compressed executable file of some Java project

- WAR stands for Web Archive file and EAR stands for Enterprise ARchive file

- JAR vs WAR vs EAR ([StackOverflow Discussion](https://stackoverflow.com/questions/5871053/difference-between-jar-and-war-in-java) – note: always read the whole just 1 answer)

## **Creating a Maven project**

- Archetype Selection

- Useful for picking a template Maven Project so that you hit the ground running and work with

their prebuilt configurations

- Group Id

- Commonly used for denoting the reverse domain name of company or organization

- For subdomains (ex. security, development, etc.) it is advised to append it to your normal group id

- Artifact Id

- Used to reference project

- project name is commonly used, no spaces

- Example: my-example-project

- Version

- Denotes the current version of the project

- Broken down into 3 parts (1.2.3)

- major version (1.x.x)

- minor version (x.1.x)

- patch version (x.x.1)

- Name (optional)

- Represents the “human readable” project name

- `my-example-project` and turn it into My Example Project

- Description (optional)

- Provides a short overview of your project

- Parent Project (optional)

- Allows us to inherit project details from another project

- We just need the group id, artifact id, and version for the parent project and it will

take all the preexisting configuration and add to our project

#### **Maven Project Coordinates**

- Takes the group id, artifact id, and version to uniquely identify your Maven project

- The three together must be unique across Maven central

## **Project Structure**

- src/main/java

- This is the primary use folder in a Java project

- All source code we write, we will write in there

- src/main/resources

- Static assets

- In a web project: HTML/CSS/JS, images, etc.

- Configuration files

- (properties, .env, yaml, json, etc.)

- src/test/java

- Where we'll place all of our test files

- These are used as part of the build process to successfully build

- src/test/resources

- Configuration files for tests

- Any static assets for the tests

- JRE

- Represents the version of Java Runtime Environment we're using

- Maven defaults to using 1.5

- src

- Folder containing the source code

- This one is used at runtime (Don't write code here)

- target

- Any build process needs an output directory/folder to store the end product

- It's a place to hold the JAR/WAR/POM file we created in the packaging phase

- pom.xml

- A giant configuration file that controls the entire project structure

## **POM File**

- Utilizes XML to store project configuration details

- Most similar to the package.json file from NPM

- Allows us to store project configuration detailsfor our project in one spot

- The POM file is commonly split up into two main sections

1. dependencies

- Indicates the third-party library we need for our project to work

- Whenever the project is built, Maven will automatically download all of these for us

- All dependencies have a scope specifying which phase of the project it is needed for

- By default, it is \_\_\_\_\_\_\_\_\_\_

- \_\_\_\_\_\_\_\_ adds it to all classpaths for each phase

- test

- Indicates that this dependency is only required for testing

- It will *not* be included in the final build of the project

- runtime

- Indicates that it's not needed during the compilation, but is during the runtime of the project

2. plugins

- plugins are \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- vs dependency is a library used in your code (dependency vs plugin [StackOverflow discussion](https://stackoverflow.com/questions/11881663/what-is-the-difference-in-maven-between-dependency-and-plugin-tags-in-pom-xml#:~:text=A%20plugin%20is%20an%20extension,test%20and%2For%20runtime%20time))

- Contains \_\_\_\_\_\_\_\_\_\_\_\_ for our project

- Specify things such as the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### **Downloading Dependencies**

- When downloading dependencies for the first time, it will download them into a local folder on your computer .m2

- It downloads them from the internet at \_\_\_\_\_\_\_\_\_\_\_\_\_\_ (kinda npm).

## **Maven Build Lifecycle**

- Maven has specific "phases" in the build of a project

1. Validate

- Ensures that the project is correct and all necessary information is provided

-- this means your project needs a POM and valid xml and the required tags project, groupId, artifactId, and version

- Fails if no POM.xml

2. Compile

- Compiles the source code

- Fails if java compilation errors like missing semicolon or trying to store a string in an int class mismatch etc., dead code

3. Test

- Runs unit tests on the compiled code

- Unit tests test individual methods and the logic of small pieces of code

- Fails if ANY unit test fails (unless specified otherwise)

4. Package

- Packages the compiled code into the specified format: we are using JAR

5. Verify

- Runs integration tests on the compiled code

- integration tests test the code running as a whole

- Fails if ANY test fails

6. Install

- Installs the created package JAR into the local repository so that it can be used in other Maven projects

7. Deploy

- Takes the created package JAR and it publishes it to Maven central (final stage)

- Maven allows the user to specify what phase they'd like to run

- This works by running every phase up to that point

- If I want to run my Maven as test phase I will run validate, compile, and test, but nothing past test