MAVEN

- Maven is an automation project management tool.
- It is used to build the code
- once we build the code we will get JAR/WAR/EAR
- Maven is used to add the dependencies to our application. Maven is based on POM.xml

POM: Project Object Model

XML: Extensible Markup Language.

Dependencies: In the context of Maven, dependencies refer to external libraries or pieces of code that your application needs in order to function properly. Instead of writing everything from scratch, developers can use these pre-built libraries for common tasks like logging, database connections, testing, and more.

- POM.xml contains project related data (metadata, kind of project, kind of output, description, dependencies).
- Maven was developed by Apache software foundations.
- Maven was released in 2004.
- Maven can build any number of projects into desired Output such as .jar, .war and .ear

.jar = java archive file

.war = web archive file

.ear = enterprise archive

- It is mostly used for java-based projects. It was initially released on 13 July 2004. Maven is written in java.

MAVEN BUILD LIFE CYCLE:

1.Generate Resource

2.Compile Code

3.Unit Test

4.Package (build)

5.install (into Local repo or Artifactory)

6.Deploy (to servers)

7.Clean (to delete all the runtime files)

WE HAVE 7 PHASES IN BUILD LIFE CYCLE(COMMANDS)

STEP-1: LAUNCH EC2 INSTANCE

INSTALL JAVA : yum install java-1.8.0-openjdk -y

TO CHECK THE VERSION: java -version

INSTALL MAVEN : yum install maven -y

TO CHECK VERSION: mvn -v

CLONE THE REPO: git clone https://github.com/devops0014/one.git

CHECKOUT TO BRANCH : git checkout master

STEP-2: COMPILE THE CODE

GOAL : mvn compile

STEP-3: TEST THE CODE

GOAL: mvn test

STEP-4: BUILDN THE CODE

GOAL: mvn package

STEP-5: INSTALL THE CODE

GOAL: mvn install

STEP-6: CLEAN THE CODE (opt)

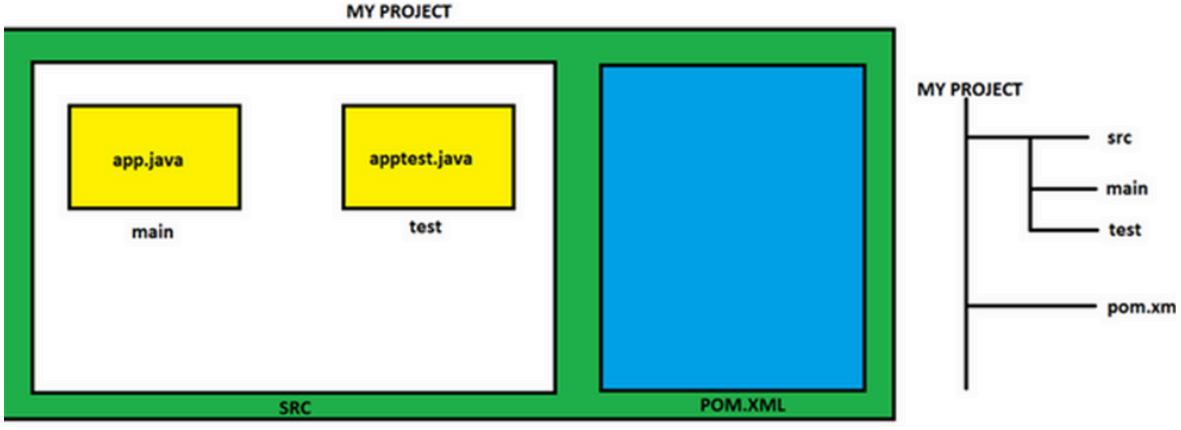
GOAL: mvn clean

NOTE: TO DO ALL THESE STEPS IN SINGLE GOAL : mvn clean package

BUILD TOOL:

- It is used to set up everything which is required to run your java code This can be applied to your entire java project.
- It generates source code, compiling code, packaging code to a jar etc.
- POM refers the XML file that have all information regarding project and configuration details
- Main configuration file is in pom.xml.
- It has description of the Project details regarding version and configuration management.
- The XML file is in the Project home directory.

MAVEN DIRECTORY STRUCTURE:



The standard Maven directory structure follows a convention that makes it easy to manage source code, resources, and other project files in a logical and organized manner. Here's what the typical directory structure looks like for a Maven project:

```
my-app/                  # Root directory of your project
 |
 ├── pom.xml                    # Maven Project Object Model (POM) file (defines dependencies, plugins, etc.)
 ├── src/                # Source directory for the project
 |   ├── main/            # Application's main code and resources
 |   |   ├── java/          # Java source code
 |   |   |   └── com/        # Java packages (e.g., com/example/app)
 |   |   └── resources/      # Application resources (config files, etc.)
 |   |       └── application.properties (Example: properties file for configuration)
 |   └── test/            # Test source code and resources
 |       ├── java/          # Unit and integration test Java code
 |       └── resources/      # Test resources (mock data, config, etc.)
 |
 └── target/                # Compiled output directory (generated by Maven during build)
     ├── classes/          # Compiled application classes
     ├── test-classes/       # Compiled test classes
     ├── my-app-1.0-SNAPSHOT.jar  # Packaged application artifact
     └── ...              # Other generated files (reports, temporary files, etc.)
```

Key Components:

pom.xml: The most important file in a Maven project. It contains information about the project, including dependencies, plugins, and other build instructions.

src/main/java: This is where the Java source code for your application resides.

src/main/resources: Non-code resources like configuration files (.properties, XML files, etc.) are stored here. Maven will include them in the final packaged artifact (like a JAR or WAR).

src/test/java: This folder holds the unit and integration test Java code. Typically, this includes frameworks like JUnit or TestNG for testing.

src/test/resources: Stores resources needed during testing, such as test-specific configuration files, mock data, or other test assets.

target/: After building the project, Maven places compiled class files and packaged artifacts (like JARs) here.

Benefits of Following this Structure:

Consistency: The structure is the same across all Maven projects, making it easy for new developers to navigate and understand your project.

Automation: Maven knows where to find source code, resources, and tests based on this structure, making builds more predictable.

Separation of Concerns: Keeping test code, application code, and resources in separate directories ensures clarity and reduces the chance of mixing things up.

ANT VS MAVEN VS GRADLE

Feature	Ant	Maven	Gradle
Convention	No formal conventions. Project structure must be specified in build.xml.	Has conventions for project structure (e.g., src/main/java, etc.).	Follows conventions but allows flexibility to define custom structures.
Build Type	Procedural. You must define what steps to perform and in which order.	Declarative. Define the project's dependencies and Maven does the rest.	Declarative by default but allows procedural tasks if needed (combines both).
Lifecycle	No lifecycle phases. Each task must be explicitly defined.	Has a lifecycle with predefined build phases (e.g., compile, test, package).	Flexible lifecycle with customizable tasks and phases.
Type	A tool (a toolbox of individual tasks).	A framework for project management.	A build automation tool and a project management framework.
Main Usage	Mainly a build tool for compiling and running tasks.	Primarily a project management tool, handling dependencies, builds, etc.	A versatile build tool with performance improvements and flexibility.
Reusability	Ant scripts are not easily reusable.	Maven plugins and configurations are reusable.	Highly reusable via plugins, tasks, and configurations.
Preference	Less preferred due to lack of conventions and lifecycle management.	More preferred because of its standardization and ease of use.	Increasingly preferred due to performance and flexibility, especially for large projects.
Dependency Management	No built-in dependency management. Dependencies must be managed manually.	Built-in dependency management via pom.xml and repositories like Maven Central.	Excellent dependency management with faster resolution, using a transitive model similar to Maven.