Build Process and Maven Explained

# 1. Build Process (In Brief)

The build process is a series of steps that convert source code into a final deliverable (such as a JAR, WAR, or EXE file) that can be run, tested, or deployed. A typical build process includes:  
- Compiling source code into bytecode  
- Packaging compiled code into distributable formats  
- Running tests to ensure quality  
- Copying resources like configuration files  
- Deploying the output to a server or repository  
  
Automation of this process ensures consistency, reduces manual errors, and improves productivity.

# 2. Build Tools to Automate the Build Process

Several tools help automate and manage the build lifecycle:  
- Apache Maven – Popular for Java; uses XML-based configuration (pom.xml)  
- Gradle – Flexible and powerful; supports Groovy/Kotlin DSL  
- Ant – Older tool; uses XML but lacks lifecycle management  
- Make – Common in C/C++ development  
- CMake – Cross-platform build tool, often used with C++  
  
Among these, Maven is widely used in enterprise Java development due to its convention-over-configuration approach.

# Apache Maven

Apache Maven is a project management and build automation tool used primarily for Java projects. It simplifies the build process using a standard project object model (POM).

## 1. Maven Phases

Maven defines a standard build lifecycle, divided into phases. Each phase represents a step in the lifecycle:  
- validate: Check if the project structure is correct  
- compile: Compile the source code  
- test: Run unit tests  
- package: Package the code (e.g., JAR/WAR)  
- verify: Run integration tests (if any)  
- install: Install the built package to the local repository  
- deploy: Copy the final package to a remote repository  
  
Each phase executes the previous ones before it in order.

## 2. Important Maven Commands

- mvn validate: Validates the project structure and configuration  
- mvn test: Runs tests using the testing framework (e.g., JUnit)  
- mvn install: Builds the project and installs the JAR/WAR to the local .m2 repo  
- mvn clean install: Deletes previous builds (clean), then compiles, tests, and installs

## 3. pom.xml

The pom.xml (Project Object Model) is the core configuration file in a Maven project. It defines:  
- Project metadata (groupId, artifactId, version)  
- Dependencies  
- Build plugins  
- Repositories  
- Build settings  
- Lifecycle configurations

Example snippet:  
<project xmlns="http://maven.apache.org/POM/4.0.0">  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.example</groupId>  
 <artifactId>demo-app</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13.2</version>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
</project>

## 4. Maven Repository

Maven uses repositories to store and fetch project artifacts (JARs, plugins, etc.):  
- Local Repository:  
 - Located at ~/.m2/repository  
 - Stores downloaded artifacts and those built with mvn install  
- Global (Remote) Repositories:  
 - Maven Central (https://repo.maven.apache.org/maven2/)  
 - Custom internal/enterprise repositories  
 - Artifacts are downloaded from here if not found locally

## 5. .m2/repository

The .m2/repository directory is the local cache where Maven stores:  
- Downloaded JAR files  
- Plugins  
- Metadata (e.g., .pom files)  
  
Located at ~/.m2/repository by default. If Maven cannot find an artifact locally, it downloads it from the remote repository and caches it here.

## 6. Default Location of mvn Package

By default, after installation, the 'mvn' executable is located at:  
/usr/local/bin