### Technolgies(webapp project)

### JDBC

### Servlet

### JSP

### JSTL

### path='jdk software'

### classpath="location of 3rd party api"

### organize the folder=> creating a packages

### create a source code

### set up the platform for compilation => javac

### set up the platform for execution => configuring the server[deployment=> HARD/SMOOTH]

### set up the libary information for server => WEB-INF/lib.

### set up the code information through XML/Annotation for server

### Zip the .class file/.html files/.xml files/.jsp files in the form of WAR(Web Archieve file)

### **..**

### **MAVEN – What is the Build Process?**

The **build process** in a Java project (using Maven or any build tool) typically includes the following steps:

1. **Prepare Resources**  
   → Gather source files, configuration files, images, etc.
2. **Arrange in Folder Structure**  
   → Organize them in standard directories (like src/main/java, src/test/java, resources).
3. **Add Dependencies**  
   → Include required libraries/JAR files via Maven (declared in pom.xml).
4. **Compilation**  
   → Convert .java files to .class files using javac.
5. **Execution**  
   → Run the compiled code to check functionality or to start the app.
6. **Testing**  
   → Execute unit/integration tests (e.g., using JUnit, TestNG).
7. **Packaging**  
   → Bundle the application into a deployable format (like .jar or .war).
8. **Deployment/Release**  
   → Move the packaged app to staging or production environments.

### ✅ Core Java Project Build Process (Steps)

1. **Develop source code and other files**  
   → Write Java classes, configurations, and assets.
2. **Organize files into folders**  
   → Group them under standard folders like src, lib, etc.
3. **Add dependencies (JARs) to classpath**  
   → Include external libraries required for the application.
4. **Compile the code**  
   → Convert .java files to .class files.
5. **Execute & Test the application**  
   → Run the app and perform **unit testing** (testing individual pieces of code).
6. **Package the application for release**  
   → Create a distributable .jar or .war file.

### ❌ Limitations of Manual Build Process

* a. **Hard to remember repetitive tasks**  
  → Developers may forget steps like testing or packaging.
* b. **Risk of incorrect order**  
  → Compiling before adding JARs leads to errors.
* c. **Activities may be skipped**  
  → Testing or packaging may be missed.
* d. **Time-consuming**  
  → Doing each task manually slows down delivery.

..

To automate this process activities we can use .bat file

run.bat

======

cd e:

md xyz

cd xyz

copy

copy

set path =...

set classpath =

javac -d \*. java

java <pkg> .< M

batch file is given to combine all related commands into single command [by using single command we can automate the process]

limitations :

Imagine a .bat script like writing down every single chore you must do, in exact order, with no help if something goes wrong. As soon as the list grows, it falls apart:

* **You must list every step**—there’s no “only redo what changed.”
* **Errors don’t stop the rest**—if one step breaks, the next still runs.
* **Can’t grab outside tools**—it won’t go download the JARs or libraries you need.
* **Windows-only**—it won’t work on Mac or Linux, and even on Windows it’s picky about spaces and quotes.
* **Gets messy fast**—adding new files or folders means editing dozens of lines.

### To avoid this we use

### Maven

Key features of maven:

### 1. Convention over Configuration

* **What it means**: Maven “knows” where you put your code (src/main/java), tests (src/test/java), resources, etc., so you don’t have to spell out every folder path.

### 2. Automatic Dependency Management

* **Pantry analogy**: Declare “I need Spring Boot” in your POM, and Maven goes out to its online pantry (Maven Central) and fetches Spring Boot + all its friends (transitive libraries) for you.

### 3. Central & Local Repositories

* **Local cache**: Once downloaded, jars sit in your local repository so you don’t redownload them each time.
* **Multiple remotes**: You can point Maven at your company’s private repo or public ones, and it’ll pull plugins and jars as needed.

### 4. Standard Project Layout

* **Readability**: Every Maven project “looks the same” on disk, so any teammate can jump in and know where co de, tests, and resources live without asking.

### 5. Inheritance & Reuse

* **Share settings**: A “parent” POM can define common dependencies, plugin versions or corporate policies—child projects inherit those automatically.

### 6. Multi-Module Projects

* **One command, many services**: Group ten microservices under one parent POM. mvn clean install will build them all in the right order for you.

### 7. Built-In Packaging

* **One-stop build**: Want a JAR, WAR or EAR? Just pick the right packaging in your POM and Maven will compile, test and bundle it.

### 8. Plugin Ecosystem

* **Kitchen appliances**:
  + **Compiler plugin** ⇒ compiles your code
  + **Surefire plugin** ⇒ runs your tests
  + **Shade plugin** ⇒ makes a “fat” JAR
  + **Site plugin** ⇒ generates documentation and reports  
    …and dozens more you can drop in with zero hassle.

### 9. Standard Lifecycle

* **Phases you already know**:

mvn clean # wipe out old builds

mvn compile # turn .java → .class

mvn test # run unit tests

mvn package # bundle into JAR/WAR

mvn install # drop into your local repo

mvn deploy # send to remote repo

You just type mvn install and Maven runs all the steps in order.

### 10. Reporting & Documentation

* **One command for a project website**: mvn site will generate Javadocs, test reports, code coverage, styles, plugin docs—everything in neat HTML

## Maven Lifecycles

Maven groups its build work into **three** main lifecycles—each a sequence of phases you can run:

1. **clean**
   * **What it does:** deletes the target/ folder so you start fresh.
   * **Command:** mvn clean
2. **default (aka “build”)**
   * **What it does:** runs all the core build steps in order:

validate → compile → test → package → verify → install → deploy

* + **Command:** typically mvn install (which drives all earlier phases)

1. **site**
   * **What it does:** generates your project’s website, including Javadoc, test reports, coverage, code metrics, etc.
   * **Command:** mvn site

## Archetypes (Project Templates)

Archetypes are “starter kits” that give you a ready-made folder & POM layout:

* **maven-archetype-quickstart**
  + Good for a simple Java app or library.
  + Sets up src/main/java, src/test/java, a minimal POM, and a sample App.java.
* **maven-archetype-webapp**
  + For web applications (WARs).
  + Creates src/main/webapp/WEB-INF, web.xml, plus the usual Java/src folders.

You just run:

mvn archetype:generate \

-DarchetypeArtifactId=maven-archetype-quickstart \

-DgroupId=com.mycompany.app \

-DartifactId=my-app

…and you get a brand-new project scaffold.

## Running Maven Builds

You can invoke these lifecycles either:

1. **On the command line**
   * cd into your project folder and run mvn clean install, mvn site, etc.
2. **From an IDE**
   * Most Java IDEs (Eclipse, IntelliJ, VS Code) let you **import** the pom.xml and then click “Run” or use their Maven toolbar to invoke phases.

### Maven repository — plain-talk version

1. **What it is**  
   A repository is just a big folder (local or online) that keeps all the libraries and plugins your build might need.
2. **What’s inside**
   * **JARs** – normal Java libraries
   * **Plugins** – mini-tools Maven uses while building
   * (Sometimes) sample or parent projects you can inherit from
3. **Local vs. remote**
   * **Local repo**: a cache on your own PC (~/.m2/repository).
   * **Remote repo**: Maven Central or a company server; Maven downloads from here the first time and then stores a copy locally.
4. **How Maven finds the right file (G + A + V)**  
   Every item is labelled with three fields, often called **GAV**:

| **Field** | **Think of it as** | **Example (spring-aspects)** |
| --- | --- | --- |
| **groupId** | The maker’s name | org.springframework |
| **artifactId** | The product’s name | spring-aspects |
| **version** | Edition number | 5.3.17 (stable) or 5.4.0-SNAPSHOT (work-in-progress) or 5.3.10 – RELEASE(next version ready) |

1. **What happens during a build**
   * Maven reads your <dependency> entry.
   * Checks your local repo for that exact G + A + V.
   * If missing, pulls it once from the remote repo, saves it locally, and continues the build.

### . **Local Repository**

* **Where:** C:\Users\[YourUsername]\.m2\repository
* **What it is:** A personal cache on your machine.
* **Purpose:** Stores the libraries (JARs) downloaded from remote/central, so Maven doesn’t need to re-download them each time.
* **Auto-created:** When you run any Maven command like mvn install, Maven checks here first.

### 🔹 2. **Remote Repository**

* **Where:** On the internet or internal company servers.
* **What it is:** Maintained by third-party teams or organizations (e.g., internal Nexus/Artifactory).
* **Purpose:** Hosts custom or external dependencies not available in the central repo.
* **Example Use Case:** Your company might publish its private libraries to an internal remote repo.

### 🔹 3. **Central Repository**

* **URL:** https://repo.maven.apache.org/maven2
* **What it is:** The **default global repository** maintained by the Maven community.
* **Purpose:** Publicly available library store (e.g., Spring Boot, JUnit, etc.)
* **Access:** Automatically searched by Maven if not found in local or other configured remotes.

### ⚙️ Resolution Order

1. **Local → Remote → Central**
2. Maven looks in the local repo first.
3. If not found, it checks configured remote repos.
4. Then it checks the central repo.

Maven lifecycle: important phases in the lifecycle

## 1. Clean Lifecycle (3 phases)

This lifecycle prepares for a fresh build by removing previous outputs.

| **Phase** | **What it does** |
| --- | --- |
| **pre-clean** | (rarely used) do any setup before cleaning |
| **clean** | remove the target/ directory |
| **post-clean** | (rarely used) finalize after cleaning |

* **mvn clean**  
  Executes **all** phases in the Clean lifecycle, ending in post-clean.  
  ⇒ Deletes target/ so you start with a clean slate.

## 2. Default (a.k.a. Build) Lifecycle (23 phases)

Out of 23 these 7 are important phases

This is where compilation, testing, packaging, and installation happen. When you run a command like mvn package, Maven walks **through** these phases in order, stopping at the one you requested.

| **#** | **Phase** | **What it really does** |  |
| --- | --- | --- | --- |
| 1 | **validate** | Make sure the POM is well-formed and all required data is present. |  |
| 2 | **initialize** | Set up the build state (create dirs, set properties, version checks). |  |
| 3 | **generate-sources** | Produce source files (e.g., JAXB/XJC or ANTLR codegen). |  |
| 4 | **process-sources** | Filter or otherwise transform raw sources before compile. |  |
| 5 | **generate-resources** | Produce resource files (e.g., i18n bundles, Swagger docs). |  |
| 6 | **process-resources** | Copy → target/classes and filter placeholders like ${project.version}. |  |
| 7 | **compile** | Compile main Java/Kotlin/etc. code. |  |
| 8 | **process-classes** | Byte-code enhancement (e.g., Hibernate, Lombok delombok). |  |
| 9 | **generate-test-sources** | Generate test helpers or mocks. |  |
| 10 | **process-test-sources** | Filter test sources. |  |
| 11 | **generate-test-resources** | Produce additional test resources. |  |
| 12 | **process-test-resources** | Copy → target/test-classes. |  |
| 13 | **test-compile** | Compile test code. |  |
| 14 | **process-test-classes** | Post-process compiled tests (byte-code tweak, instrumentation). |  |
| 15 | **test** | Run unit tests with JUnit/TestNG. *Fails build if tests red.* |  |
| 16 | **prepare-package** | Last-minute tweaks before building the artifact (shade, merge manifests). |  |
| 17 | **package** | Assemble the artifact: JAR, WAR, EAR, … |  |
| 18 | **pre-integration-test** | Spin up containers, seed DBs, start servers. |  |
| 19 | **integration-test** | Run end-to-end tests against the packaged app. |  |
| 20 | **post-integration-test** | Tear down the test environment, collect coverage. |  |
| 21 | **verify** | Extra quality gates (Jacoco thresholds, Checkstyle, Owasp-dependency-check). |  |
| 22 | **install** | Copy the final artifact + POM to your local ~/.m2/repository. |  |
| 23 | **deploy** | Upload the artifact to a remote repo (Nexus, Artifactory, GitHub Packages). |  |

out of 23 these are the important phases

validate, compile, test, package, verify, install, deploy

## 3. Site Lifecycle (4 phases)

This lifecycle generates your project’s documentation and reports.

| **Phase** | **What it does** |
| --- | --- |
| **pre-site** | any setup before reporting |
| **site** | generate the HTML reports/docs |
| **post-site** | finalize and copy resources |
| **site-deploy** | publish the generated site to a web server |

just type command mvn phase\_name

if we type the 13 command then the previous phases also will get executed.

## Running Your JAR Manually

Once you’ve got your MathProj1-1.0.jar, launch it with:

java -cp target/MathProj1-1.0.jar in.ineuron.Arithmetic

* -cp (classpath) points to your JAR.
* in.ineuron.Arithmetic is the fully qualified name of the class containing public static void main(...).

To write the test cases code.

1. **Check your POM has JUnit 5.**

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.10.2</version>

<scope>test</scope>

</dependency>

The scope=test flag keeps JUnit out of your runtime classpath.

1. **Create the folder if it’s missing**

src/test/java

1. **Mirror the package of the class you’re testing.**  
   If your production class is com.example.calc.Calculator in  
   src/main/java/com/example/calc/Calculator.java,  
   then place your test in  
   src/test/java/com/example/calc/CalculatorTest.java.
2. **Write the test class.**

package com.example.calc;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

class CalculatorTest {

@Test // marks this as a test method

void add\_shouldReturnSum() {

// Arrange

Calculator calc = new Calculator();

// Act

int result = calc.add(2, 3);

// Assert

assertEquals(5, result);

}

}

You create an object of the source class (*Calculator*) and exercise its logic, then use assertions to verify the outcome.

1. **Run only the test-compilation step (if you like):**

mvn process-test-sources

That copies and filters resources into target/test-classes but **does not** compile or run the tests.  
To compile and execute them you usually run:

mvn test

or the whole flow:

mvn clean verify

### What process-test-sources does:

* It's step **#10** in Maven's default build process.
* It **copies your test files** (like Java files under src/test/java and resources under src/test/resources) into the build folder (usually target).
* It **does NOT compile** those test files. Compilation happens in the **next step**, called test-compile.

### 🧪 Where to write test code:

* Put your test Java files in: src/test/java
* Use the **same package names** as your main code.
* Mark your test methods with @Test (from JUnit).

### 🚀 To run your tests:

Just run:

mvn test

This command will:

1. Compile the test files,
2. Run the tests,
3. Show the results.

### ✅ **Steps to Run Java Class Using** exec-maven-plugin **in Eclipse/STS**

1. **Right-click on your project** in the Project Explorer.
2. Navigate to:  
   Run As ➝ Run Configurations... ➝ Maven Build.
3. In the **Goals** field, type:

exec:java

1. Click **Run**.

### 🔍 What It Does:

This command tells Maven to:

* Use the exec-maven-plugin,
* Run the java goal,
* Execute the class defined in the plugin configuration (mainClass tag in pom.xml).

Eg: run-java-class-using-exec-plugin

Properties:

=========

Instead of writing the version number repeatedly for each Spring dependency, you're doing this:

#### 🔷 Step 1: Define a property

xml

<properties>

<spring.version>5.3.26</spring.version>

</properties>

#### 🔷 Step 2: Use that property in multiple dependencies

xml

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

And similarly for:

* spring-jdbc
* spring-orm
* spring-webmvc

### 🎯 Why This is Useful:

| **✅ Benefit** | **💡 Explanation** |
| --- | --- |
| Centralized versioning | Update version only once in <properties>. |
| Cleaner pom.xml | No duplication of version numbers. |
| Easier upgrades | Change ${spring.version} once → all dependencies update. |

### ✅ Step-by-Step: How to Set ${spring.version}

In your pom.xml, add the following section (if it's not already present):

xml

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<!-- This sets the value of ${spring.version} -->

<spring.version>5.3.26</spring.version>

</properties>

Now whenever you use:

xml

<version>${spring.version}</version>

Maven will **replace it with 5.3.26** during the build.

### 📒 **Maven Dependency Exclusion – Key Points**

1. **Purpose**:  
   Exclusion is used to **remove specific transitive (indirect) dependencies** brought in by another dependency.
2. **Use Case**:  
   When two libraries bring in different versions of the same JAR, or when you want to **avoid unwanted libraries** (e.g., logging, XML parsers).
3. **Does Not Remove Main Dependency**:  
   Exclusion only removes the unwanted child dependency, not the one you’re directly adding.
4. **Avoid Conflicts**:  
   Helps prevent **duplicate classes**, **version conflicts**, or **security issues** in large projects.
5. **Supports Multiple Exclusions**:  
   You can exclude more than one transitive dependency using multiple <exclusion> tags.

### ✅ **Example: Exclude** jboss-logging **from** hibernate-core

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.15.Final</version>

<exclusions>

<exclusion>

<groupId>org.jboss.logging</groupId>

<artifactId>jboss-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

### 📒 **Custom Repository in Maven – Key Points**

1. **Used when a dependency is not available on Maven Central**  
   (e.g., private, commercial, or internal company artifacts)
2. **Define inside <repositories> tag in pom.xml**
3. **Structure of a custom repository entry:**

<repositories>

<repository>

<id>custom-repo</id>

<url>https://repo.mycompany.com/maven2</url>

</repository>

</repositories>

1. **<id>** is a unique name for identifying the repository.
2. **Maven will try this repository if the dependency is not found locally or in Maven Central.**
3. **Multiple repositories** can be added under the <repositories> tag.
4. **Authentication** (for private repos) is done via Maven’s settings.xml.

### ✅ **Example: Using Custom Repository to Fetch a Library**

<repositories>

<repository>

<id>jitpack.io</id>

<url>https://jitpack.io</url>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>com.github.User</groupId>

<artifactId>SomeLibrary</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

### ✅ What is a Local Repository?

By default, Maven uses the local repository located at:

C:\Users\<your-username>\.m2\repository (on Windows)

~/.m2/repository (on macOS/Linux)

This is where Maven stores dependencies downloaded from remote repositories.

### 🔧 How to Add a Custom JAR to Your Local .m2 Repository

If you have a JAR file (say my-lib.jar) and want to install it to your local .m2 repository, use the following command:

mvn install:install-file -Dfile=path/to/my-lib.jar \

-DgroupId=com.example \

-DartifactId=my-lib \

-Dversion=1.0.0 \

-Dpackaging=jar

This command:

* Installs the JAR into your local .m2/repository
* Organizes it under:  
  .m2/repository/com/example/my-lib/1.0.0/my-lib-1.0.0.jar

### 📄 How to Reference This in Your pom.xml

<dependency>

<groupId>com.example</groupId>

<artifactId>my-lib</artifactId>

<version>1.0.0</version>

</dependency>

No need to add a <repository> section because Maven will check .m2 first.

### 📝 Tips

* To see the local repo path:

mvn help:evaluate -Dexpression=settings.localRepository -q -DforceStdout

* You can configure a **custom local repository location** in settings.xml:

<settings>

<localRepository>D:/custom-maven-repo</localRepository>

</settings>

### 📘 **How to Add a Custom JAR to Local** .m2 **Maven Repository**

#### ✅ 1. Install Maven (if not already)

* Download from: <https://maven.apache.org/download.cgi>
* Extract and add bin folder to system PATH

#### ✅ 2. Use mvn install:install-file Command

mvn install:install-file -Dfile="path/to/your.jar" ^

-DgroupId=your.group.id ^

-DartifactId=your-artifact ^

-Dversion=1.0.0 ^

-Dpackaging=jar

📝 Example:

mvn install:install-file -Dfile="C:\Users\Administrator\Desktop\One Drive\OneDrive\java drivers\ojdbc11.jar" -DgroupId=com.oracle.database.jdbc -DartifactId=ojdbc11 -Dversion=21.9.0.0 -Dpackaging=jar

#### ✅ 3. Verify the Installed Location

* Go to:

makefile

CopyEdit

C:\Users\<YourUsername>\.m2\repository\com\oracle\database\jdbc\ojdbc11\21.9.0.0\

* The .jar file should appear here

#### ✅ 4. Add Dependency in pom.xml

<dependency>

<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc11</artifactId>

<version>21.9.0.0</version>

</dependency>

#### ✅ 5. Refresh Your Maven Project

* In STS/IntelliJ: Right-click → Maven → **Update Project**
* Or run:

mvn clean install

### note : ✅ **Yes, the GAV (GroupId, ArtifactId, Version) used in the Maven install command must exactly match the GAV you use in** pom.xml.

### 🔁 Why?

When you run this:

mvn install:install-file \

-Dfile=ojdbc11.jar \

-DgroupId=com.oracle.database.jdbc \

-DartifactId=ojdbc11 \

-Dversion=21.9.0.0 \

-Dpackaging=jar

👉 Maven stores the JAR in this path:

~/.m2/repository/com/oracle/database/jdbc/ojdbc11/21.9.0.0/ojdbc11-21.9.0.0.jar

Then when you add this in pom.xml:

<dependency>

<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc11</artifactId>

<version>21.9.0.0</version>

</dependency>

👉 Maven looks for that exact file path in .m2. If there's **any mismatch** in the groupId, artifactId, or version — Maven won't find the JAR, and your project will fail to build with a missing dependency.

Eg: AddCustomJarM2LocalRepo

### ✅ Maven Inheritance – Key Points (Notes):

1. **Maven supports inheritance** of configuration through the concept of a parent POM.
2. A **child POM inherits** the configurations from the parent POM such as dependencies, plugins, properties, etc.
3. The **parent POM** is specified using the <parent> tag in the child POM.
4. It **promotes reusability and consistency** across multiple modules/projects.
5. Common configurations like plugin versions or repositories can be maintained in one place (parent).
6. You can **override or add additional configurations** in the child as needed.
7. Inheritance is useful in **multi-module projects** or organizations with many related Maven projects.

| **Feature** | **Maven Inheritance (<parent>)** | **Maven Dependency (<dependency>)** |
| --- | --- | --- |
| Inherit plugin configs | ✅ Yes | ❌ No |
| Inherit properties | ✅ Yes | ❌ No |
| Inherit dependencies (via <dependencyManagement>) | ✅ Yes | ❌ No |
| Use Java code from parent | ❌ No | ✅ Yes (with <dependency>) |

### ❓ Your Question:

In your earlier example, ChildApp is calling GreetingUtil from parent-project. But you said Maven inheritance (<parent>) does not share code. So how did it work?

### ✅ Answer:

The reason it **worked in my example** is because I **added the parent project as a dependency** in the child’s pom.xml.

Here is that important part again from the child POM:

<dependencies>

<dependency>

<groupId>in.orcas</groupId>

<artifactId>parent-project</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

### 🔍 This line is what makes GreetingUtil (in the parent) available to the child!

### 🧠 Now the Key Rule:

| **Usage** | **Requires <parent>?** | **Requires <dependency>?** | **Result** |
| --- | --- | --- | --- |
| Inherit plugin/configs | ✅ Yes | ❌ No | Works |
| Use Java classes from parent | ❌ No | ✅ Yes | Works |
| Do both | ✅ Yes | ✅ Yes | Works ✅ |

So in my example:

* ✅ I used <parent> to inherit config (like compiler settings).
* ✅ I used <dependency> to access parent’s Java code like GreetingUtil.

### 🔴 If You Remove <dependency> from Child:

Then the code will NOT compile because the Java class (GreetingUtil) will NOT be on the classpath of the child.

### ✅ Final Summary (very short):

* Maven **inheritance (<parent>) is for configuration only**.
* To **use Java code**, you must **add the parent as a <dependency>**.
* In my example, I used **both**, which is why it worked.

Eg: parent-project

Eg: child-project