**Maven for Web Applications – Notes**

🔹 **Purpose**:  
Maven helps manage dependencies and build lifecycle for Java web applications.

**🏗️ Archetype**

* Use maven-archetype-webapp to generate a standard web project structure.
* Command:

bash

mvn archetype:generate -DgroupId=com.example \

-DartifactId=mywebapp \

-DarchetypeArtifactId=maven-archetype-webapp \

-DinteractiveMode=false

**📦 Packaging Type**

* Use <packaging>war</packaging> in pom.xml
* WAR (Web Application Archive) is used to deploy Java web apps to servers like Tomcat, Jetty, etc.

**📁 Standard Folder Structure**

src/

main/

java/ --> Java source code

webapp/ --> Web content (HTML, JSP, etc.)

resources/ --> config files

test/

java/ --> Test cases

**🧩 Dependencies**

* Define servlet/jsp APIs and other web libraries in pom.xml

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

**🧾 Maven WAR Deployment to Tomcat (Hard Deployment)**

🔹 **Step 1: Package the Application**

mvn package

* This command compiles the project and packages it into a .war file.
* The .war file will be located in the target/ directory.

🔹 **Step 2: Deploy WAR File to Tomcat (Hard Deployment)**

* Manually **copy** the WAR file to the webapps/ folder of your **Apache Tomcat** server.

Example:

target/WishApp.war → <TOMCAT\_HOME>/webapps/

🔹 **Step 3: Start Tomcat Server**

* Run the Tomcat server (startup.sh or startup.bat depending on OS)

🔹 **Step 4: Test in Browser**

http://localhost:9999/WishApp

* Here, 9999 is the Tomcat port.
* WishApp is the name of the deployed web application.

**📝 Notes:**

* This is called **hard deployment** because it requires **manual file copying**.
* For **soft deployment**, integration tools like **Tomcat Maven Plugin** or **Spring Boot embedded Tomcat** are preferred.

**🔁 Reuse One Maven Project in Another (Local JAR Usage)**

**📦 Step 1: Package and Install the JAR**

In your **first Maven project** (e.g., MyLibProject), run:

mvn install

✅ What it does:

* Compiles your Java classes
* Creates a .jar file
* Places it inside your **local Maven repository** (.m2 folder)

📁 Path:

php-template

CopyEdit

~/.m2/repository/<groupId>/<artifactId>/<version>/<artifactId>-<version>.jar

**🔗 Step 2: Add Dependency in Second Project**

In your **second Maven project** (MyAppProject), open pom.xml and add:

<dependency>

<groupId>com.example</groupId>

<artifactId>MyLibProject</artifactId>

<version>1.0</version>

</dependency>

🧠 Maven will now **fetch the JAR from your local .m2 repo** and use it just like any external library.

**🧪 Step 3: Use the Code**

Now you can directly use the classes from MyLibProject in your new project:

import com.example.util.MyHelper; // from the JAR

MyHelper.sayHello();

**📌 Summary:**

| **Step** | **Action** |
| --- | --- |
| mvn install | Builds JAR and saves it to .m2 |
| Add dependency | Use groupId/artifactId/version in pom.xml |
| Use it | Java code from Project 1 now available in Project 2 |

**⚙️ What is Logical Deployment?**

Instead of manually copying the .war to Tomcat’s webapps folder, we let **Maven deploy it automatically** using plugins like the **Tomcat Maven Plugin**.

**✅ Step 1: Add Tomcat Plugin to pom.xml**

Add this plugin under the <build> section:

<build>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<url>http://localhost:9999/manager/text</url>

<server>TomcatServer</server>

<path>/WishApp</path>

</configuration>

</plugin>

</plugins>

</build>

Also, define the Tomcat credentials in your settings.xml:

<server>

<id>TomcatServer</id>

<username>admin</username>

<password>admin</password>

</server>

**🚀 Step 2: Deploy Using Maven**

mvn tomcat7:deploy

✅ Maven will:

* Build the project
* Create a .war
* Deploy it directly to running Tomcat (without manual file copy)

**🔍 Access the App:**

Visit:

http://localhost:9999/WishApp

**🔄 Using Another Project’s Code (via .m2)**

You can also **reuse another Maven project’s classes** without copying them manually:

1. First, run mvn install in the other project.
2. Then, add it as a dependency in your current project’s pom.xml.

✅ Maven will **fetch it from .m2 local repository**, no need for source files.

**📝 Summary**

| **Task** | **Traditional (Hard Deployment)** | **Logical (Maven Plugin)** |
| --- | --- | --- |
| WAR Copy | Manual to Tomcat webapps/ | Automatic via Maven |
| Deployment | Physical | Logical |
| Use Other Project | Copy JAR manually | Use via .m2 & mvn install |

## ✅ What is Dependency Scope?

Dependency scope tells Maven **when** (which phase) and **where** (which classpath) a dependency should be available.

Imagine you're building a Java project — not all libraries are needed all the time. Some are just for testing, some only at runtime, and some are already provided by the server (like servlet containers). That’s where **scopes** help.

## 🔍 1. compile (default scope)

### 🔹 What it means:

* This dependency is **needed at all times** — while compiling, testing, and running your application.
* It is also included in the final **WAR/JAR** file.

### 🔧 When to use:

* For core libraries like spring-core, commons-lang3, log4j, etc.

### 📦 Example:

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.12.0</version>

</dependency>

(No need to specify scope; compile is default)

## 🔍 2. provided

### 🔹 What it means:

* You **need** the dependency for compilation, but it will be **provided at runtime** by the **server** or **container** (like Tomcat, JDK).

### ⚠️ Not included in final WAR/JAR.

### 🔧 When to use:

* For javax.servlet-api, JSP, or anything the container already has.

### 📦 Example:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

💡 You don’t want to package servlet classes if Tomcat already has them.

## 🔍 3. runtime

### 🔹 What it means:

* You **don’t need it for compilation**, but it is **needed when the app runs**.

### 🔧 When to use:

* For libraries like JDBC drivers, database clients, etc.

### 📦 Example:

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.32</version>

<scope>runtime</scope>

</dependency>

💡 You write your code against interfaces (like javax.sql.DataSource), but the actual driver is needed only at runtime.

## 🔍 4. test

### 🔹 What it means:

* This dependency is **only available when running tests**.
* Not included in the final build.

### 🔧 When to use:

* For JUnit, Mockito, TestNG, etc.

### 📦 Example:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

💡 Perfect for unit tests — you don't want JUnit in your production WAR.

## 🔍 5. system (⚠️ Legacy/rarely used)

### 🔹 What it means:

* Similar to provided, but you must specify the full path to the JAR.
* Maven won’t try to download it — it assumes it’s already on your system.

### 🔧 When to use:

* Only when using **local JARs** not present in any repository.

### 📦 Example:

<dependency>

<groupId>com.custom</groupId>

<artifactId>my-lib</artifactId>

<version>1.0</version>

<scope>system</scope>

<systemPath>${project.basedir}/lib/my-lib.jar</systemPath>

</dependency>

💡 Not recommended. Better to use mvn install and pull from .m2.

## 🔍 6. import (used in *dependencyManagement* only)

### 🔹 What it means:

* You’re **importing dependencies** from another POM (usually a BOM file — Bill Of Materials).
* Helps **centralize versions** in one place.

### 🔧 When to use:

* When using Spring Boot, Micronaut, or any BOM-based library management.

### 📦 Example:

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>3.1.1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

💡 Helps you avoid version conflicts.

## 🧠 Summary with Use Cases

| **Scope** | **Available In Phases** | **Included in Final Build?** | **Use For** |
| --- | --- | --- | --- |
| compile | Compile, Test, Runtime | ✅ Yes | Core application libraries |
| provided | Compile only | ❌ No | Servlet API, Java EE, etc. |
| runtime | Runtime, Test | ✅ Yes | JDBC drivers, logging backends |
| test | Test only | ❌ No | JUnit, Mockito |
| system | Compile (via system path) | ❌ No | External JARs (not in repo) |
| import | In dependencyManagement | N/A | Managing versions via BOM |

Note : normally compile, runtime, provided

### 📦 Maven Multi-Module Project

* **Definition**:  
  A **multi-module project** in Maven is a way to organize related modules within a single project. Each module represents a separate component, but all are managed together under a parent project.
* **Purpose**:  
  When multiple modules are part of a project and are closely related, it is better to structure them as a **multi-module project**.
* **How it works**:
  + Maven ensures all **sub-modules are built in the correct order** before the main module is built.
  + This structure is helpful in large-scale applications where components like DAO, service, and web layers are separated.
* **Agile Practice Alignment**:
  + In Agile methodology, multiple layers (like DAO, Service, Web) are often treated as **separate modules**.
    - DAO module → Data Access
    - Service module → Business Logic
    - Web module → Front-end / API exposure
* **Conclusion**:  
  For modular development and maintainability, **Maven multi-module** setup is a preferred structure.

**🧩 Why Use Multi-Module in Agile?**

In **Agile development**, we build projects using a **layered structure**. This means we separate the code into different parts (called layers), each with its own responsibility:

* **Controller** – handles user input or API calls (Web layer)
* **Service** – contains the business logic (what the application should do)
* **DAO** – connects to the database and performs data operations

To make the project more organized and easier to manage:

✅ We keep each layer in a **separate module**  
✅ All modules are placed under one **main project** (called a **multi-module project**)  
✅ This setup is supported by **Maven**

**🔧 Benefits of Multi-Module Approach in Agile**

* Each module (layer) can be **developed and tested separately**
* **Teams can work independently** on different modules
* **Builds become faster and more efficient**
* Code is better **organized and reusable**

#### 🏗 Example Structure of Maven Multi-Module Project

ParentProject/

├── pom.xml # Parent POM

├── proj01-controller/ # Module: Controller (Proj-01)

│ └── pom.xml

├── proj01-service/ # Module: Service (Proj-01)

│ └── pom.xml

├── proj01-dao/ # Module: DAO (Proj-01)

│ └── pom.xml

├── proj02-controller/ # Module: Controller (Proj-02)

│ └── pom.xml

├── proj02-service/ # Module: Service (Proj-02)

│ └── pom.xml

├── proj02-dao/ # Module: DAO (Proj-02)

│ └── pom.xml

#### 🔄 Dependency Flow (Layered Approach)

Each project (e.g., PROJ-01 or PROJ-02) follows this module dependency:

Controller Module (depends on) → Service Module (depends on) → DAO Module

* proj01-controller → depends on proj01-service
* proj01-service → depends on proj01-dao

Likewise for PROJ-02.

Eg: MavenMultiModule