**LAB ACTION PLAN FOR WEEK 7**

**Objective:**

To enable students to:

* Understand the structure and lifecycle of a Maven project.
* Build and package Java and Web applications using Maven.
* Add dependencies using **pom.xml**, compile and test using plugins.
* Resolve errors and conflicts arising from dependency mismatches.
* Work with parent and multi-module Maven projects.
* Generate executable JARs and deployable WARs using Maven.

Students must **document observations**, include **screenshots of executions**, and answer **scenario-based questions** after completing the tasks.

### ****Understanding Maven Project Structure****

Maven standardizes the project structure for both Java and web-based applications. src/

├── main/

│ ├── java/ → Application source code

│ └── resources/ → Configuration files like config.properties

└── test/

├── java/ → Unit test source code

└── resources/ → Test resources

The compiled files and reports are generated in the **target/** directory after a successful build.

### ****Steps to Perform Maven Build and Testing****

**----mvn clean install**

* **clean:** Deletes the previous build (**target/** folder)
* **install:** Builds, tests, and installs the package to local **.m2** repository

After execution:

* **target/** folder contains compiled **.class** files and the final **.jar** or **.war**
* Artifact is stored in:  
  **~/.m2/repository/groupId/artifactId/version/**

Add a Dependency (e.g., Gson):

In **pom.xml**:

*<dependency>*

*<groupId>com.google.code.gson</groupId>*

*<artifactId>gson</artifactId>*

*<version>2.10</version>*

*</dependency>*

After running:

*mvn clean install*

Check:

* Gson JAR is downloaded to **.m2/repository/com/google/code/gson/gson/**
* If version is wrong or not found **→ BUILD FAILURE** with dependency resolution error.

**Creation of Maven Java Project**

**Step 1.** Open Eclipse IDE

└── 1.1. Launch Eclipse workspace

**Step 2**. Install Maven Plugin (if not installed)

└── 2.1. Go to "Help" in the top menu

└── 2.1.1. Click "Eclipse Marketplace"

└── 2.1.2. Search for "Maven Integration for Eclipse"

└── 2.1.3. Install the plugin if not already installed

**Step 3**. Create a New Maven Project

└── 3.1. File -> New -> Project...

└── 3.1.1. Expand "Maven"

└── 3.1.2. Select "Maven Project" and click "Next"

**Step 4**. Set Project Configuration

└── 4.1. Select workspace location (default or custom)

└── 4.2. Click "Next"

**Step 5**. Choose Maven Archetype

└── 5.1. Select an archetype(e.g "org.apache.maven.archetypes -> maven-archetype-quickstart 1.4 ")

└── 5.2. Click "Next"

**Step** 6. Define Project Metadata

└── 6.1. Group ID: (e.g., com.example)

└── 6.2. Artifact ID: (e.g., my-maven-project)

└── 6.3. Version: (default is usually fine)

└── 6.4. Click "Finish"

**In Console, artifacts are grouped. When prompted with Y/N, type 'Y'.**

**Step 7**. Maven Project Created

└── 7.1. Project structure is generated with a standard Maven layout

└── 7.2. Includes:

└── src/main/java (for Java source code)

└── src/test/java (for test code)

└── pom.xml (Maven configuration file)

**Step 8**. Update Project Settings (if needed)

└── 8.1. Right-click on the project -> Maven -> Update Project...

└── 8.2. Ensure dependencies are up to date

**Step 9**. Build and Run Maven Project

└── 9.1. Right-click on App.java -> Run As -> Maven Clean

└── 9.1.1. Right-click on App.java -> Run As -> Maven Install

└── 9.1.2. Right-click on App.java -> Run As -> Maven Test

└── 9.1.3. Right-click on App.java -> Run As -> Maven Build

**Step 10**. In the Maven Build dialog:

└── Enter Goals: clean install test

└── Click on Apply -> Click on Run

**Step 11**. Check console for BUILD SUCCESS message.

**Step 12**. Run the application:

└── Right-click on App.java -> Run As -> Java Application

└── Output: "Hello World" displayed.

**Creation of Maven web Java Project**

**Step 1**: Open Eclipse

└── 1.1 Launch Eclipse IDE.

└── 1.2 Select or create a workspace.

**Step 2**: Create a New Maven Project

└── 2.1. File -> New -> Project...

└── 2.1.1. Expand "Maven"

└── 2.1.2. Select "Maven Project" and click "Next"

**Step 3**: Choose Maven Archetype

└── 3.1. Select an archetype(e.g "'org.apache.maven.archetypes' -> 'maven-archetype-webapp' 1.4 ")

└── 3.2. Click "Next"

**Step 4**: Configure the Maven Project

└── 4.1 Group Id: Enter a group ID (e.g., com.example).

└── 4.2 Artifact Id: Enter an artifact ID (e.g., my-web-app).

└── 4.3 Click \*\*Finish\*\* to create the project.

**Step 5**: Add Maven Dependencies

└── 5.1 Open the \*\*pom.xml\*\* file in the Maven project.

└── 5.2 Add the necessary dependencies for your web project (e.g., Servlet, JSP):

**Go to browser -> Open mvnrepository.com**

**Search for 'Java Servlet API' -> Select the latest version.**

**Copy the dependency code -> Paste it in MavenWeb’s pom.xml under the target folder**

└── Example:

```xml

<dependency>

<groupId>javax.servlet</groupId>

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

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

```

**Step 6**:-. Configure server:

└── Window -> Show View -> Servers

└── Add server -> Select Tomcat v9.0 server -> Click Next

└── Configure server options (e.g., ports, server location).

**Step 7**:-. Modify 'tomcat-users.xml':

└── Add role and user details under <tomcat-users> tag.

**Step 8**:. Build the project:

└── Right-click on index.jsp -> Run As -> Maven Clean

└── Right-click on index.jsp -> Run As -> Maven Install

└── Right-click on index.jsp -> Run As -> Maven Test

└── Right-click on index.jsp -> Run As -> Maven Build

**Step 9**. In the Maven Build dialog:

└── Enter Goals: clean install test

└── Click on Apply -> Click on Run

**Step 10**. Check console for BUILD SUCCESS message.

**Step 11**. Run the application:

└── Right-click on index.jsp -> Run As -> Run on Server

└── Select the Tomcat server -> Click on Finish

**Step 12**. Output: "Hello World" webpage displayed.

**Note:-Now push yours Maven java project and Maven Web Project into your github**

1. **Configure Java Version via Compiler Plugin**

In **pom.xml**:

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-compiler-plugin</artifactId>*

*<version>3.11.0</version>*

*<configuration>*

*<source>17</source>*

*<target>17</target>*

*</configuration> </plugin>*

Check:

* Run mvn package
* Inspect generated JAR: *jar tf target/myapp.jar*

1. **JUnit Testing and Reports**

Place test files in src/test/java. Example:

*public class AppTest {*

*@Test*

*public void testSum() {*

*assertEquals(5, 2 + 3);*

*}*

*}*

Run:

mvn test

Check:

* target/test-classes/ → compiled test .class files
* target/surefire-reports/ → .txt or .xml test results  
  If test fails → corresponding log and failure trace shown

1. **Handling Errors in pom.xml**

* Typos in version numbers or missing repositories cause BUILD FAILURE
* Maven shows precise error in console
* Fix the dependency tag → re-run mvn clean install

1. **Adding Resource Files**

Put config.properties inside:

src/main/resources/config.properties

To read:

InputStream input = getClass().getClassLoader().getResourceAsStream("config.properties");

After build, check target/classes/ → file should exist there.

1. **Multi-Module and Parent Projects**

Structure:

*parent/*

*├── pom.xml (packaging: pom)*

*├── core/*

**|** *└── pom.xml*

*└── web/*

└── pom.xml

* Parent pom.xml defines <modules> and common dependencies
* Each submodule builds independently into its target/ directory

1. **Executable JARs**

Add to pom.xml:

*<build>*

*<plugins>*

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-jar-plugin</artifactId>*

*<configuration>*

*<archive>*

*<manifest>*

*<mainClass>com.example.Main</mainClass>*

*</manifest>*

*</archive>*

*</configuration>*

*</plugin>*

*</plugins>*

*</build>*

Run:

mvn package

java -jar target/myapp.jar

1. **Building a WAR File**

Create a Maven web project with structure:

src/main/webapp/

└── WEB-INF/web.xml

Add:

*<packaging>war</packaging>*

*Command:*

*mvn package*

Generates target/mywebapp.war → deploy on Tomcat server.

1. **Scenario-Based Questions:**
2. If my error is about:

*Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.8.1:compile*

*[ERROR] -> [Help 1]*

*[ERROR] To see the full stack trace of the errors, re-run Maven with the -X switch.*

How can I resolve it using maven terminal?

**Answer:** In your terminal, run mvn clean install –X

And in pom.xml check if the version of maven is 3.10 or not. If its not change it to <version>3.10.1</version> <!-- Or the latest stable version -> in plugins.

1. A dependency you added is not recognized by the compiler. What steps would you take to confirm it is available in **.m2** and listed in dependency tree?

#### **Answer:** C**heck the Dependency in** pom.xml

* Ensure that the dependency is correctly declared in your pom.xml file.
* Verify that you have provided the correct group ID, artifact ID, and version.

Example:

<dependencies>

<dependency>

<groupId>com.example</groupId>

<artifactId>my-library</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

Make sure:

* The groupId is correct.
* The artifactId matches the library name.
* The version corresponds to an existing version.

#### 2. **Check Dependency in** .m2 **Local Repository**

Maven downloads dependencies into the .m2 repository located in your user directory (usually ~/.m2/repository/ on Linux/macOS or C:\Users\YourUser\.m2\repository\ on Windows). To confirm that the dependency exists in .m2:

* Navigate to your .m2 directory.
* Follow the path of the groupId, artifactId, and version to check if the JAR file exists.

1. A teammate sends a **.patch** file for a bug fix. How would you apply it and include it in your Maven build?

**Answer:**

1. **Review the patch file**: Check what changes it makes to your codebase.
2. **Apply the patch**: Use git apply if the project is in a Git repository.
3. **Verify the changes**: Check if the patch applied correctly.
4. **Commit the changes**: Add, commit, and push the changes to your repository.
5. **Run the Maven build**: Run mvn clean install to build the project and include the patch in the Maven build.
6. **Handle conflicts if needed**: Resolve any merge conflicts if the patch doesn't apply cleanly.
7. You have multiple **JUnit** test failures and want to rerun only failed tests. How would you approach this?

**Answer:** Maven Surefire and Failsafe plugins support running only the tests that previously failed, which is very convenient when you have many tests, and you only want to rerun the ones that failed.

Here’s how you can use Maven’s **Surefire plugin** to automatically rerun the failed tests:

##### **Run Tests with** -Dsurefire.reportsDirectory **and** -DfailIfNoTests=false

The -Dsurefire.reportsDirectory flag allows Maven to store test results in a specific directory, which is useful for tracking the failed tests.

To rerun only the failed tests after an initial run:

1. **Run your tests the first time** (you’ll need the results to identify failed tests):
2. mvn clean test
3. **Rerun only failed tests:**
4. mvn -Dtest=FailedTestClassName test

Alternatively, use the -Dtest option to run the specific test classes that failed.

To automate this process, you can add configuration to the pom.xml to **rerun only failed tests**. You can set up the **Surefire plugin** to **skip already successful tests** and rerun only the failed ones after the initial run.

**Example configuration for maven-surefire-plugin:**

<build>

<plugins>

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version> <!-- or your current version -->

<configuration>

<!-- Allow rerunning only failed tests -->

<rerunFailingTestsCount>3</rerunFailingTestsCount> <!-- Retry up to 3 times -->

<testFailureIgnore>true</testFailureIgnore> <!-- Allow build to proceed even if some tests fail -->

</configuration>

</plugin>

</plugins>

</build>

* **rerunFailingTestsCount**: This allows you to specify how many times Maven should attempt to rerun the failing tests before considering them as permanent failures.
* **testFailureIgnore**: If you set this to true, Maven will continue building even if tests fail.

With this configuration in place, Maven will attempt to rerun the failed tests based on the configuration you've set.

1. Your Maven build fails due to “Unsupported class version error.” What plugin and configuration would you review?

**Answer:**

In Maven, the **maven-compiler-plugin** controls the version of Java used for compiling the project. If you're compiling with a newer version but running on an older JDK, it will cause this error. To fix it, you need to set the correct **source** and **target** versions in the plugin configuration.

**Example** for compiling with Java 8:

<build>

<plugins>

<plugin>

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

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version> <!-- or latest version -->

<configuration>

<source>1.8</source> <!-- Set the source code version (Java version used to compile) -->

<target>1.8</target> <!-- Set the target bytecode version -->

</configuration>

</plugin>

</plugins>

</build>

This will ensure that the project is compiled with **Java 8**. You can change it to 1.11 or any other version if you're working with Java 11 or higher.

**Important**: Make sure that both the **source** and **target** are set to the same version (unless you have specific reasons for setting them differently).

1. You need to change your Javaapplication from a **WAR** to a standalone **JAR**. What pom.xml changes are needed?

**Answer:**

1. **Change the packaging type**: <packaging>jar</packaging>
2. **Remove Web Application-related dependencies** (e.g., servlet-api, jsp-api, etc.).
3. **Add dependencies for standalone execution** (e.g., Spring Boot, embedded Jetty/Tomcat).
4. **Add Maven Shade Plugin** to create a fat JAR (optional, if you want to bundle dependencies).
5. **Specify a main class** for the entry point of the application.
6. **Use the Spring Boot Maven Plugin** if you're using Spring Boot for packaging the application as an executable JAR.
7. You are required to change the default build output directory from **target/** to **build\_output/**. How would you configure it?

**Answer:** To change the default build output directory from target/ to build\_output/ in Maven, you need to update the pom.xml file to specify a different directory for the build output. This can be done by configuring the **build** section, specifically the **directory** element.

### Steps to Change the Build Output Directory:

#### 1. **Modify the** <build> **section in** pom.xml

Inside your pom.xml, locate the <build> section. Then, add or modify the <directory> tag to specify the new output directory, which in this case is build\_output/.

Here's how you can configure it:

<build>

<directory>build\_output</directory> <!-- Change the output directory -->

<plugins>

<!-- Any other plugins you're using (like maven-compiler, surefire, etc.) -->

</plugins>

</build>

By default, Maven uses target/ for build output. The <directory> tag specifies the location where Maven will place the compiled classes, JAR/WAR file, and other build artifacts. In this case, we are telling Maven to use build\_output/ instead.

1. You want to skip tests during the Maven build. What command would you use?

**Answer:**

**Skip tests execution (but compile them)**: mvn clean install –DskipTests

**Skip both test compilation and execution**: mvn clean install -Dmaven.test.skip=true

These are the commands which help you with above problem.

1. How would you generate a site report (with test coverage, dependency analysis) for a Maven project?

**Answer:** To generate a site report for a Maven project, you use the **mvn site** command. This command processes the **maven-site-plugin** and other reporting plugins defined in your project's pom.xml file. The generated report, including test coverage and dependency analysis, is placed in the **target/site** directory of your project.

1. How do you build a Java project using Maven, and what files are generated in the **target/** folder after running mvn clean install?

**Answer:** You build a Java project with Maven by running the command mvn clean install from the project's root directory. This command triggers a series of lifecycle phases to compile, test, and package your code.

## Files Generated in the target/ Folder

After successfully running mvn clean install, the target/ folder contains all the build outputs. The exact files can vary slightly depending on your project, but the common ones are:

1. **classes/**: Contains the compiled .class files from your main source code.
2. **test-classes/**: Contains the compiled .class files from your test source code.
3. **[your-project-name]-[version].jar** (or .war): This is your project's main artifact, containing the compiled code and resources.
4. **maven-archiver/**: A directory used by Maven to store build metadata.
5. **surefire-reports/** (or **failsafe-reports/**): Contains the XML and text reports for your unit and integration tests. These reports detail which tests passed or failed.
6. How does Maven resolve dependency conflicts when two libraries use different versions of the same dependency, and how can you view and manage the **dependency tree**?

**Answer:** Once you've identified a conflict, you can manage it using two primary methods in your pom.xml:

**<dependencyManagement>**: This section allows you to centralize dependency versions for a project, especially in multi-module projects. You can declare a specific version of a transitive dependency here, which forces all modules to use that version, overriding the "nearest-wins" rule.

**<exclusions>**: This is a more direct approach to manually resolve conflicts by explicitly excluding a transitive dependency from a specific library. This is useful when you want to prevent a particular library from bringing in an older or incompatible version of another dependency.

1. How do you write and run a JUnit test in a Maven project, and where are the compiled test classes and reports stored after running **mvn test**?

**Answer:** Maven follows a standard directory layout. All your test classes must be placed under the **src/test/java** directory to be recognized and run by Maven. Your test class should mirror the package structure of the code it's testing. For example, if you have a class at src/main/java/com/example/Calculator.java, its corresponding test class should be at src/test/java/com/example/CalculatorTest.java.

A basic JUnit 5 test class looks like this:

package com.example;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class CalculatorTest {

@Test

void testAdd() {

Calculator calculator = new Calculator();

assertEquals(5, calculator.add(2, 3), "2 + 3 should equal 5");

}

}

You must also include the appropriate JUnit dependencies in your pom.xml file, typically with the <scope>test</scope> to ensure they are only used for testing and not included in your final application.

To run all unit tests in your project, use the **mvn test** command. This command compiles both your main source code and your test source code, then executes the tests using the **Maven Surefire Plugin**. This plugin automatically discovers any class with a name ending in Test or Tests (among other conventions) within the src/test/java directory.

After you run mvn test, Maven organizes all the build artifacts in the **target/** directory.

* **Compiled Test Classes**: The compiled .class files for your tests are stored in **target/test-classes/**.
* **Test Reports**: The test results are stored in **target/surefire-reports/**. These reports are typically generated as XML and plain text files, providing a detailed summary of the test run, including which tests passed or failed.

1. How can you create an executable **JAR** with a main method using Maven, and which **plugin** helps configure this behavior?

**Answer:** To create an executable JAR in Maven with a main method using Eclipse:

1. **Create a Maven Project** in Eclipse using a simple quickstart template (e.g., maven-archetype-quickstart).
2. **Add the Maven Shade Plugin** to your pom.xml to bundle the main class and dependencies into an executable JAR.

Example configuration:

<plugin>

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

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.1</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass>com.yourpackage.Main</mainClass>

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

1. **Create a Main Class** with a main method.
2. **Update Dependencies** in Eclipse (Maven > Update Project).
3. **Build the Project** using Run As > Maven Build, and type clean package to generate the executable JAR in the target directory.
4. **Run the JAR** with:
5. java -jar target/your-artifact-id-1.0-SNAPSHOT.jar

Alternatively, use the **Maven Assembly Plugin** for more control, especially if you need to include dependencies in the JAR.

1. How do you install and use a custom third-party JAR file in your Maven project, and how can you confirm it’s included in the build and **classpath**?

**Answer:**

1. Install the custom JAR using mvn install:install-file.
2. Add it as a dependency in pom.xml.
3. Verify inclusion in the build via mvn dependency:tree and check the target folder.
4. Ensure it’s in the classpath (e.g., in IDE's External Libraries section).
5. Optionally, deploy to a remote repository if needed.
6. How do you create a Maven web project that packages into a WAR file, and what is the standard folder structure for such a project?

**Answer:**

1. **Create a Maven web project** using the maven-archetype-webapp archetype.
2. **Ensure packaging is war** in the pom.xml.
3. Follow the **standard folder structure** for web applications:
4. src/main/java/ for Java classes.
5. src/main/webapp/ for web resources (JSP, HTML, CSS, JS).
6. WEB-INF/ for web.xml.
7. **Build the WAR** using mvn clean package.
8. **Deploy the WAR** to a servlet container like Tomcat or Jetty.
9. What command do you use to build a WAR file in Maven, where is it generated, and how can you deploy it to a server like **Apache Tomcat**?

**Answer:**

1. **Build the WAR** using mvn clean package.
2. **Find the WAR file** in the target/ folder.
3. **Deploy manually** by copying the WAR to Tomcat's webapps/ folder or use the **Maven Tomcat plugin** for automatic deployment.
4. How do you add JSTL and **servlet-api dependencies** in a Maven web project, and why should the servlet API use provided scope instead of compile?

**Answer:**

1. **Add JSTL and Servlet API Dependencies:**  
   In your pom.xml, add the following dependencies for JSTL and Servlet API:

<dependencies>

<!-- JSTL Dependency -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<!-- Servlet API Dependency (provided scope) -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

</dependencies>

1. **Why Use Provided Scope for Servlet API:**  
   The **provided scope** means the servlet container (like Tomcat) provides the servlet API at runtime, so it is not included in the WAR file. This avoids duplication.
2. **Compile Scope vs Provided Scope:**  
   The **compile scope** includes the dependency in the build, meaning it would be packaged in the WAR, which is unnecessary since Tomcat already includes the servlet API.
3. **JSTL:**  
   JSTL provides standard tags for JSP, like loops and conditionals. It's added as a regular dependency with no special scope.
4. **Servlet API Version:**  
   Always use the version of the servlet API that matches your web server's version (e.g., Tomcat 9 uses servlet 4.0).
5. How do you set up a multi-module Maven web project with separate modules for core logic and web interface, and how are these modules built and connected?

**Answer:** I have shared the file separately on multi module on this theme. Go through it.

1. How do you configure a Maven web project, and how does its packaging and execution differ from a traditional WAR-based application?

**Answer:**

1. **Maven Web Project Configuration:**  
   In Maven, configure a web project by setting the packaging type to war in the pom.xml and adding necessary dependencies like servlet-api and JSTL.

<packaging>war</packaging>

1. **Difference from Traditional WAR:**  
   Unlike traditional WAR projects, Maven handles dependency management, builds the project into a WAR file, and ensures consistent versioning. You use mvn clean package to build, and Maven automatically includes required dependencies.
2. **Execution:**  
   Maven projects can be deployed easily to servers (like Tomcat) using plugins, whereas traditional WAR projects may require manual configuration and dependency management.

### ****Conclusion****

Mastering Maven empowers students to structure projects efficiently, manage complex dependencies, and automate testing and packaging. By practicing real-world scenarios—such as resolving build errors, handling resource files, applying patches, and deploying to servers—students gain valuable hands-on experience aligned with professional software development workflows. Maven not only streamlines builds but also enforces standardization and reusability across projects. Through Maven, students learn efficient project management, dependency control, multi-module integration, and reproducible builds. Understanding Maven’s lifecycle, plugin system, and error handling prepares students for professional DevOps and CI/CD workflows. This lab equips students with hands-on experience in packaging, testing, resolving conflicts, and deploying real-world Java and Web applications.