LAB ACTION **PLAN FOR WEEK 4**

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.

1. 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.

2. 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

3. 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*

4. 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

5. 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`

6. 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.

7. 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

8. 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
```

9. 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.

10. Scenario-Based Questions:

1. 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?
2. 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?
3. A teammate sends a `.patch` file for a bug fix. How would you apply it and include it in your Maven build?
4. You have multiple **JUnit** test failures and want to rerun only failed tests. How would you approach this?
5. Your Maven build fails due to "Unsupported class version error." What plugin and configuration would you review?
6. You need to change your Java application from a `WAR` to a standalone `JAR`. What `pom.xml` changes are needed?
7. You are required to change the default build output directory from `target/` to `build_output/`. How would you configure it?
8. You want to skip tests during the Maven build. What command would you use?
9. How would you generate a site report (with test coverage, dependency analysis) for a Maven project?
10. How do you build a Java project using Maven, and what files are generated in the **target/** folder after running mvn clean install?
11. 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**?
12. 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**?
13. How can you create an executable **JAR** with a main method using Maven, and which **plugin** helps configure this behavior?
14. 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**?
15. 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?
16. 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**?
17. 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?
18. 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?

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

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.