Maven Tutorial

Introduction to Maven

What is Maven?

Maven is a build automation tool primarily used for Java projects. It addresses two main aspects of building software: dependency management and project build lifecycle management.

It simplifies the build process by managing dependencies, compiling source code, packaging it into a deliverable (such as a JAR file), and deploying it to a repository.

Maven is based on the concept of a Project Object Model (POM), which is a central piece of information that manages a project's build, reporting, and documentation.

History and Evolution of Maven

Maven was developed by the Apache Software Foundation and released in 2004.

It was created to address the problems with Apache Ant, a popular build tool at the time. Maven was designed to simplify the build process, provide a uniform build system, and offer project information that could be shared among developers.

Over the years, Maven has become one of the most widely used build tools in the Java ecosystem.

Setting Up Maven

This section will guide your trainees through installing and setting up Maven on their machines and integrating it with popular IDEs like IntelliJ IDEA and Eclipse.

Installing Maven

Prerequisites:

Ensure that Java is installed on the machine. Maven requires Java to run. The trainees should have the Java Development Kit (JDK) installed and the JAVA_HOME environment variable set.

Steps to Install Maven:

Download Maven:

Go to the Maven official download page.

Download the latest binary zip archive or tar.gz file.

Extract the Archive:

Extract the downloaded archive to a directory of your choice (e.g., C:\\Program Files\\Apache\\maven on Windows or /usr/local/apache-maven on macOS/Linux).

Set Environment Variables:

Windows:

Right-click on 'This PC' or 'My Computer', then go to Properties → Advanced System Settings → Environment Variables.

Add a new system variable named MAVEN_HOME pointing to the Maven directory.

Append the bin directory of Maven to the Path system variable: C:\\Program Files\\Apache\\maven\\bin.

macOS/Linux:

Open .bash_profile or .zshrc file in your home directory and add:

Copy

```
export MAVEN_HOME=/usr/local/apache-maven
export PATH=$MAVEN_HOME/bin:$PATH
```

Run source ~/.bash_profile or source ~/.zshrc to apply the changes.

Verify Installation:

Open a terminal or command prompt and type mvn -version.

You should see the Maven version information along with Java details if Maven is correctly installed.

Setting up Maven in IDEs

IntelliJ IDEA:

Maven Integration:

IntelliJ IDEA has Maven integration built-in, so no additional setup is required.

When creating a new project, you can select Maven as the project type, and IntelliJ will automatically generate the pom.xml file and manage the project structure.

Importing a Maven Project:

If you already have a Maven project, you can import it by opening IntelliJ and selecting File → Open, then navigating to the pom.xml file of your project.

Maven Tool Window:

IntelliJ provides a Maven tool window where you can run Maven goals, view the lifecycle, and manage dependencies directly from the IDE.

Eclipse:

Installing the Maven Plugin:

Eclipse requires the m2e plugin for Maven integration, though many versions of Eclipse come with this plugin pre-installed.

To install it manually: Help → Eclipse Marketplace → Search for "m2e" → Install.

Creating a Maven Project:

Go to File → New → Project → Maven → Maven Project.

Follow the wizard to set up your project, specifying details like GroupId, ArtifactId, and packaging type.

Importing a Maven Project:

If you have an existing Maven project, import it by selecting File → Import → Maven → Existing Maven Projects, then point to the directory containing the pom.xml.

Maven Integration:

Eclipse provides a "Maven" perspective that allows you to run Maven commands, manage dependencies, and configure the build lifecycle from within the IDE.

Understanding the .m2 Directory and settings.xml

The .m2 Directory:

Located in the user's home directory (e.g., C:\\Users\\YourName\\.m2 on Windows or /home/YourName/.m2 on macOS/Linux).

Contains the repository folder where Maven stores downloaded dependencies. This is known as the local repository.

You can delete this folder if you want to force Maven to re-download dependencies, but usually, Maven manages it automatically.

settings.xml File:

Located in the .m2 directory. This file allows you to customize the behavior of Maven globally across all projects.

Common configurations include:

Mirror configuration to point Maven to a specific repository.

Proxy settings if you're behind a firewall or using a corporate network.

Local repository location customization.

Example settings.xml:

Copy

```
<settings>
  <mirrors>
    <mirror>
      <id>central</id>
      <mirrorOf>central</mirrorOf>
      <url><http://mycompany.com/maven2></url>
    </mirror>
  </mirrors>
  <proxies>
    <proxy>
      <id>example-proxy</id>
      <active>true</active>
      <protocol>http</protocol>
```

```
        <host>proxy.mycompany.com</host>
        <port>8080</port>
      </proxy>
    </proxies>
</settings>
```

Project Object Model (POM)

The POM (Project Object Model) is the core of a Maven project. Understanding the POM file is essential because it contains all the configuration details for a Maven project.

Structure of a pom.xml File

The pom.xml file is an XML file that contains information about the project and configuration details used by Maven to build the project.

Here’s a basic structure of a pom.xml file:

Copy

```
<project xmlns="<http://maven.apache.org/POM/4.0.0>"
        xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
        xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0>
                        <http://maven.apache.org/xsd/maven-4.0.0.xsd>">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>My App</name>
    <url><http://maven.apache.org></url>

    <dependencies>
      <dependency>
        <groupId>junit</groupId>
```

```
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.1</version>
          <configuration>
            <source>1.8</source>
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

Key Elements in POM

modelVersion:

This specifies the version of the POM model being used. Currently, 4.0.0 is the default and most widely used version.

groupId:

The groupId uniquely identifies your project across all projects. Typically, it follows the reverse domain name convention (e.g., com.example).

artifactId:

The artifactId is the name of the jar without the version. It is the name of the project.

version:

The version is the project's current version. Maven uses this version to distinguish between different versions of the same project.

Versions can be specific like 1.0.0, or they can include SNAPSHOT, which indicates a version in development.

packaging:

The type of artifact this project produces. Common packaging types are jar, war, pom, etc.

If omitted, jar is assumed by default.

name and url:

These are optional elements used to provide a human-readable name and project URL.

dependencies:

The dependencies section lists all the external libraries your project depends on. Maven will download these libraries automatically from the central repository or other specified repositories.

build:

The build section is used to specify how the project should be built. This includes specifying plugins, custom build directories, resources, etc.

Dependencies and Dependency Management

Dependencies:

Dependencies are external libraries that your project needs to work. You specify these in the dependencies section of the POM.

Maven automatically downloads the specified versions of dependencies and any transitive dependencies they require.

Example:

Copy

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.5.4</version>
  </dependency>
</dependencies>
```

Transitive Dependencies:

Maven resolves transitive dependencies automatically. For example, if your project depends on A, and A depends on B, Maven will include B as a dependency in your project.

Maven Dependency Scopes and Profiles

1. Dependency Scopes

Maven provides several scopes for dependencies, each determining at what phase of the build process the dependency is available and whether it is included in the final artifact (e.g., a JAR or WAR file). Understanding these scopes is crucial for managing project dependencies effectively.

Here’s a detailed explanation of each scope, with examples:

a. compile Scope

Default Scope: If no scope is specified, Maven uses compile by default.

Availability: The dependency is available in all phases of the build lifecycle, including compilation, testing, and packaging.

Inclusion in Final Artifact: The dependency is included in the final artifact.

Typical Use: Used for dependencies required to build, test, and run the project.

Example:

Copy

```
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-lang3</artifactId>
   <version>3.12.0</version>
   <scope>compile</scope> <!-- This is the default scope -->
</dependency>
```

In this example, commons-lang3 is a utility library used during all phases of the build, so it’s specified with compile scope.

b. provided Scope

Availability: The dependency is available at compile-time but is not included in the final artifact.

Inclusion in Final Artifact: Not included in the final artifact. The assumption is that the runtime environment (e.g., a web server) will provide it.

Typical Use: Used for dependencies that are provided by the runtime environment, such as servlet APIs provided by an application server.

Example:

Copy

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
</dependency>
```

Here, the javax.servlet-api is only needed for compilation and testing. It's expected to be provided by the server (e.g., Tomcat) in production, so it's marked with provided scope.

c. runtime Scope

Availability: The dependency is not required for compilation but is needed during runtime, including execution and tests.

Inclusion in Final Artifact: The dependency is included in the final artifact.

Typical Use: Used for dependencies that are needed only at runtime, like JDBC drivers.

Example:

Copy

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.26</version>
    <scope>runtime</scope>
</dependency>
```

In this example, the MySQL JDBC driver is not required for compiling the code but is needed at runtime when the application connects to a MySQL database.

d. test Scope

Availability: The dependency is available only for the test compilation and execution phases.

Inclusion in Final Artifact: Not included in the final artifact.

Typical Use: Used for libraries that are required only for testing, such as JUnit or Mockito.

Example:

Copy

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
</dependency>
```

Here, JUnit is only needed for running unit tests and isn't included in the final build output, so it's scoped as test.

e. system Scope

Availability: The dependency is available for compilation and testing but must be explicitly provided by the developer on the system.

Inclusion in Final Artifact: Not included in the final artifact.

Typical Use: Used rarely when the dependency is not available in any remote repository and must be provided locally.

Important: You must explicitly provide the path to the JAR file using the <systemPath> element.

Example:

Copy

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>custom-lib</artifactId>
```

```
    <version>1.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/lib/custom-lib.jar</systemPath>
</dependency>
```

In this example, custom-lib.jar is not available in any Maven repository and is instead provided manually in the lib directory of the project.

Maven Profiles

Maven profiles allow you to customize the build process for different environments or situations. A profile can modify the default behavior by adding, removing, or overriding certain parts of the POM, such as dependencies, plugins, properties, etc.

How Profiles Work

Activation: Profiles can be activated explicitly using the P flag in the Maven command or automatically based on certain conditions (e.g., OS type, JDK version, environment variables).

Scope: Profiles can modify the build process for a particular environment, such as development, testing, or production.

Creating and Using Profiles

a. Simple Profile Example

This profile will activate different configurations based on whether you're in a development or production environment.

POM with Profiles:

Copy

```
<project>
    ...
    <profiles>
        <profile>
            <id>development</id>
            <properties>
                <environment.name>dev</environment.name>
                <db.url>jdbc:h2:mem:devdb</db.url>
            </properties>
```

```
        <dependencies>
          <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>1.4.200</version>
          </dependency>
        </dependencies>
      </profile>

      <profile>
        <id>production</id>
        <properties>
          <environment.name>prod</environment.name>
          <db.url>jdbc:mysql://prod-server/db</db.url>
        </properties>
        <dependencies>
          <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.26</version>
          </dependency>
        </dependencies>
      </profile>
    </profiles>
</project>
```

Explanation:

Development Profile: Activates a H2 in-memory database for development. This is useful for quick setup and teardown without affecting the production database.

Production Profile: Uses a MySQL database for the production environment.

Activating a Profile:

To activate the development profile:

Copy

mvn clean install -Pdevelopment

To activate the production profile:

Copy

mvn clean install -Pproduction

b. Profile Activation Based on Conditions

Profiles can also be activated based on various conditions, such as the operating system, JDK version, or environment variables.

POM with Conditional Profiles:

Copy

```
<project>
  ...
  <profiles>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>Windows</family>
        </os>
      </activation>
      <properties>
        <env.name>windows-env</env.name>
      </properties>
    </profile>

    <profile>
      <id>jdk-11</id>
```

```
        <activation>
          <jdk>11</jdk>
        </activation>
        <properties>
          <jdk.version>11</jdk.version>
        </properties>
      </profile>
    </profiles>
</project>
```

Explanation:

Windows Profile: Activated automatically when Maven is run on a Windows operating system.

JDK 11 Profile: Activated automatically when Maven detects that the project is being built with JDK 11.

Activating Automatically:

If Maven is running on a Windows machine, the windows profile is activated automatically.

If Maven is using JDK 11, the jdk-11 profile is activated automatically.

Maven Build Lifecycle

The Maven build lifecycle is a sequence of phases that define the order in which tasks are executed during the build process of a Maven project. Maven has three built-in build lifecycles:

1. Default Lifecycle (Build)

This is the main lifecycle that handles project deployment. It consists of 23 phases, the most important of which are:

validate: Validates the project is correct and all necessary information is available.

compile: Compiles the source code of the project.

test: Runs the tests using a suitable unit testing framework (like JUnit).

package: Packages the compiled code into a distributable format, such as a JAR or WAR file.

verify: Runs any checks to verify the package is valid and meets quality criteria.

install: Installs the package into the local repository, for use as a dependency in other projects locally.

deploy: Copies the final package to the remote repository for sharing with other developers and projects.

2. Clean Lifecycle

This lifecycle is used to clean up artifacts created by the previous build. It consists of three phases:

pre-clean: Executes processes needed before the actual project cleaning.

clean: Removes files generated during the previous build.

post-clean: Executes processes needed after the project cleaning.

3. Site Lifecycle

This lifecycle is used to create the project's site documentation. It consists of four phases:

pre-site: Executes processes needed before the actual project site generation.

site: Generates the project's site documentation.

post-site: Executes processes needed after the site generation, preparing for site deployment.

site-deploy: Deploys the generated site to a web server.

How to Use the Build Lifecycle

You can execute any of the phases using the Maven command:

Copy

mvn <phase>

For example, running mvn install will run all the phases leading up to and including the install phase.

Understanding these phases allows you to control the build process and ensure that your project is built, tested, packaged, and deployed correctly.

Key Commands for the Default Lifecycle

When you run a Maven command, such as mvn package, Maven executes all the phases up to and including the package phase:

mvn validate: Validates the project.

mvn compile: Compiles the source code of the project.

mvn test: Runs the tests using a suitable unit testing framework.

mvn package: Takes the compiled code and packages it into a JAR, WAR, etc.

mvn install: Installs the package into the local repository, making it available for other projects locally.

mvn deploy: Copies the final package to the remote repository.

Phases of the Clean Lifecycle

pre-clean: Executes processes needed prior to cleaning.

clean: Removes files generated during the previous build.

post-clean: Executes processes needed after cleaning.

When you run mvn clean, Maven executes all phases of the Clean lifecycle.

Phases of the Site Lifecycle

pre-site: Executes processes needed prior to generating the site documentation.

site: Generates the project's site documentation.

post-site: Executes processes needed after generating the site documentation.

site-deploy: Deploys the generated site documentation to the server.

When you run mvn site, Maven executes all phases of the Site lifecycle to generate the project documentation.

Maven Repositories

Maven repositories are a central part of the Maven build system. They are where Maven stores and retrieves project dependencies, plugins, and other artifacts. Understanding how repositories work and how to manage them is crucial for effectively using Maven.

Types of Maven Repositories

Maven supports three types of repositories:

Local Repository:

The local repository is a directory on your computer where Maven stores downloaded dependencies, plugins, and other artifacts.

When you build a Maven project, Maven first checks the local repository to see if the required dependencies are already available. If they are, Maven uses them; if not, Maven downloads them from a remote repository (such as the central repository) and stores them locally.

The default location of the local repository is ~/.m2/repository (on Unix-based systems) or

Copy

C:\\Users\\YourUsername\\.m2\\repository (on Windows).

You can configure the location of the local repository in the settings.xml file:

Copy

```
<settings>
  <localRepository>C:/path/to/your/local/repo</localRepository>
```

</settings>

Central Repository:

The central repository is a global, publicly available repository hosted by the Maven project. It contains a vast collection of open-source libraries, plugins, and other project artifacts.

The central repository is the default source from which Maven downloads dependencies if they are not found in the local repository.

Maven automatically connects to the central repository when you request a dependency that is not available locally.

Remote Repository:

A remote repository is any repository that is hosted on a web server other than the central repository. Organizations often maintain their own remote repositories to store proprietary artifacts, plugins, or third-party libraries that are not available in the central repository.

You can configure Maven to use a remote repository by adding it to your pom.xml or settings.xml file.

Example in pom.xml:

Copy

```
<repositories>
  <repository>
    <id>my-company-repo</id>
    <url><http://repository.mycompany.com/maven2></url>
  </repository>
</repositories>
```

How Maven Resolves Dependencies

Maven follows a specific order to resolve dependencies:

Local Repository: Maven checks the local repository first. If the dependency is found, Maven uses it.

Remote Repositories: If the dependency is not found in the local repository, Maven checks any configured remote repositories.

Central Repository: If the dependency is not found in any remote repositories, Maven checks the central repository.

Fail: If Maven cannot find the dependency in any repository, the build fails with an error.

Managing Dependencies with Repositories

Adding Dependencies

You add dependencies to your project by specifying them in the dependencies section of the pom.xml file. Maven then resolves these dependencies by checking the repositories in the order described above.

Example:

Copy

xmlCopy code

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.5.4</version>
  </dependency>
</dependencies>
```

Excluding Transitive Dependencies

Sometimes, a dependency might bring in transitive dependencies that you don't want. You can exclude these unwanted transitive dependencies using the <exclusions> tag.

Example:

Copy

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.5.4</version>
    <exclusions>
      <exclusion>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
```

```
    </exclusions>
  </dependency>
</dependencies>
```

Using Repository Managers

Organizations often use repository managers like Nexus or Artifactory to:

Cache Artifacts: Repository managers cache dependencies from remote repositories, reducing download times and providing redundancy in case a remote repository becomes unavailable.

Store Internal Artifacts: Organizations can store their internal or proprietary libraries and plugins in a secure location.

Enforce Policies: Repository managers can enforce security policies, licensing, and artifact versioning rules.

Example of Configuring a Nexus Repository:

Copy

```
<repositories>
  <repository>
    <id>nexus-repo</id>
    <url><http://nexus.mycompany.com/repository/maven-public/></url>
  </repository>
</repositories>
```

Using Mirrors to Redirect Repositories

Mirrors allow you to redirect Maven to use a different repository for a given URL pattern. This is useful for routing all requests that would go to the central repository to an internal mirror.

Example in settings.xml:

Copy

```
<mirrors>
  <mirror>
    <id>central-mirror</id>
    <mirrorOf>central</mirrorOf>
    <url><http://mycompany.com/maven/central></url>
```

```
    </mirror>
</mirrors>
```

In this example, all requests to the central repository are redirected to http://mycompany.com/maven/central.

Maven Dependency Management

Maven's dependency management system is one of its most powerful features. It automatically handles the inclusion of libraries that your project needs, as well as their transitive dependencies (dependencies of your dependencies). Understanding how to manage these dependencies effectively is key to maintaining a clean and efficient build.

Dependency Scopes (Recap)

As we covered earlier, Maven supports different dependency scopes (compile, provided, runtime, test, system). These scopes determine when a dependency is available during the build process and whether it is included in the final artifact.

Here's a quick recap:

compile: Available in all phases and included in the final artifact.

provided: Available at compile-time but not included in the final artifact (e.g., servlet APIs provided by the server).

runtime: Not needed for compilation but required during execution (e.g., JDBC drivers).

test: Available only for testing, not included in the final artifact.

system: Similar to provided, but the JAR must be available on the local system.

Transitive Dependencies

Transitive dependencies are dependencies that are required by the libraries your project depends on. Maven resolves these automatically, so you don’t have to manually include every single dependency in your pom.xml.

Example of Transitive Dependencies:

Suppose your project depends on spring-core, which in turn depends on commons-logging.

Maven will automatically download commons-logging as a transitive dependency when you add spring-core to your project.

Copy

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
```

```
    <version>5.3.9</version>
</dependency>
```

Explanation: When Maven processes this dependency, it will also download and include any libraries that spring-core depends on, such as commons-logging.

Dependency Management in Multi-Module Projects

In multi-module projects, you often have a parent POM that defines dependencies for all modules. Maven provides a <dependencyManagement> section that allows you to centrally manage dependency versions across multiple modules.

Using <dependencyManagement> in the Parent POM:

Copy

```
<project>
    <groupId>com.example</groupId>
    <artifactId>parent-project</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>5.3.9</version>
            </dependency>
            <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
                <version>1.7.30</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

```
  <modules>
    <module>module-a</module>
    <module>module-b</module>
  </modules>
</project>
```

Explanation: Dependencies defined in <dependencyManagement> are not automatically included in the child modules. Instead, they provide a version and scope template that child modules can use.

Child Module POM:

Copy

```
<project>
  <parent>
    <groupId>com.example</groupId>
    <artifactId>parent-project</artifactId>
    <version>1.0.0</version>
  </parent>

  <artifactId>module-a</artifactId>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
    </dependency>
  </dependencies>
</project>
```

Explanation: The child module inherits the version and scope of spring-core from the parent POM, thanks to the <dependencyManagement> section.

Excluding Unwanted Transitive Dependencies

Sometimes, you might want to exclude a transitive dependency that is brought in by one of your dependencies but is not needed by your project.

Example of Excluding Transitive Dependencies:

Copy

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.5.4</version>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Explanation: This configuration excludes the spring-boot-starter-tomcat dependency, which might be useful if you're using a different server (e.g., Jetty) and don't want Tomcat libraries included.

Managing Conflicts with Dependency Mediation

When multiple versions of the same dependency are included via transitive dependencies, Maven uses a process called "dependency mediation" to determine which version to use. Maven usually selects the nearest definition (i.e., the version specified in the pom.xml or the version closest to the root of the dependency tree).

Example of Dependency Conflict:

If A depends on B:1.0 and C depends on B:2.0, and both A and C are included in your project, Maven will choose one version of B.

You can control which version is selected by explicitly declaring the desired version in your pom.xml.

Copy

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>B</artifactId>
  <version>2.0</version>
</dependency>
```

Explanation: By explicitly declaring the version of B, you override the transitive versions and ensure that B:2.0 is used.

Best Practices for Dependency Management

Minimize the Number of Dependencies: Only include the libraries that are essential to your project. More dependencies can lead to larger builds and more complex dependency graphs, increasing the risk of conflicts.

Use <dependencyManagement>: In multi-module projects, use the <dependencyManagement> section to centralize the management of dependency versions.

Exclude Unnecessary Transitive Dependencies: Use the <exclusions> tag to remove dependencies that are not needed by your project.

Resolve Conflicts Explicitly: When facing version conflicts, explicitly declare the version you want to use in your pom.xml.

Regularly Update Dependencies: Keep your dependencies up to date to benefit from the latest features and security patches.