Apache Maven is a powerful build automation tool used primarily for Java projects. It manages project dependencies, builds, and documentation. Below is a comprehensive tutorial for Maven 3.9.

Apache Maven 3.9 Tutorial

Table of Contents

---

1. Prerequisites

- **Java JDK**: Make sure you have JDK 8 or higher installed. You can check your Java version by running:

bash

VerifyOpen In EditorEditCopy code

1java -version

- **Environment Variables**: Set the **JAVA_HOME** environment variable to point to your JDK installation.

2. Installation

1. **Download Maven**: Go to the Apache Maven website and download the latest version of Maven (3.9.x).
2. **Extract the Archive**: Unzip the downloaded file to a directory on your machine.
3. **Set Environment Variables**:
    - Set **M2_HOME** to the directory where Maven is installed.
    - Add **%M2_HOME%\bin** to your **PATH** variable.

4. **Verify Installation**: Open a terminal or command prompt and run:

bash

VerifyOpen In EditorEditCopy code

1mvn -v

You should see Maven version information.

3. Creating a Maven Project

You can create a new Maven project using the following command:

mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=true

- **groupId**: Your project's group ID.
- **artifactId**: Your project's name.
- **archetypeArtifactId**: The template to use for the project.

4. Understanding the POM File

The **pom.xml** file is the core of a Maven project. Here's a simple example:

```
1<project xmlns="http://maven.apache.org/POM/4.0.0"
2         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>com.example</groupId>
6    <artifactId>my-app</artifactId>
7    <version>1.0-SNAPSHOT</version>
8    <properties>
9        <maven.compiler.source>1.8</maven.compiler.source>
10        <maven.compiler.target>1.8</maven.compiler.target>
11    </properties>
12</project>
```

5. Adding Dependencies

To add a dependency, include it in the **<dependencies>** section of your **pom.xml**:

```
1<dependencies>
2    <dependency>
3        <groupId>junit</groupId>
```

```
4       <artifactId>junit</artifactId>
5       <version>4.13.2</version>
6       <scope>test</scope>
7   </dependency>
8</dependencies>
```

6. Building the Project

To build your project, navigate to your project directory and run:

bash

VerifyOpen In EditorEditCopy code

```
1mvn clean install
```

- **clean**: Removes the **target** directory.
- **install**: Compiles the code, runs tests, and packages the application.

7. Maven Lifecycle

Maven has a standard lifecycle with phases:

- **validate**: Check if the project is correct.
- **compile**: Compile the source code.
- **test**: Run tests.
- **package**: Package the compiled code into a JAR/WAR.
- **install**: Install the package into the local repository.
- **deploy**: Copy the final package to a remote repository.

8. Plugins

Maven plugins are used to perform tasks. For example, to use the Maven Compiler Plugin, add this to your **pom.xml**:

xml

VerifyOpen In EditorEditCopy code

```
1<build>
2   <plugins>
3       <plugin>
4           <groupId>org.apache.maven.plugins</groupId>
5           <artifactId>maven-compiler-plugin</artifactId>
6           <version>3.8.1</version>
```

```
7         <configuration>
8             <source>1.8</source>
9
```

please continue with response

Certainly! Let's continue with the tutorial on Maven 3.9.

8. Plugins (continued)

Continuing from where we left off, here's how to configure the Maven Compiler Plugin in your **pom.xml**:

xml

VerifyOpen In EditorEditCopy code

```
1             <target>1.8</target>
2         </configuration>
3     </plugin>
4 </plugins>
5</build>
```

This configuration specifies that the source and target Java versions for compilation are both set to 1.8.

Commonly Used Plugins

- **Maven Surefire Plugin**: Used for running tests.
- **Maven Assembly Plugin**: Used for creating a distributable package.
- **Maven Shade Plugin**: Used for creating an uber JAR with dependencies.

9. Profiles

Maven profiles allow you to customize builds for different environments (e.g., development, testing, production). You can define profiles in your **pom.xml** like this:

xml

VerifyOpen In EditorEditCopy code

```
1<profiles>
2   <profile>
3       <id>dev</id>
```

```
4       <properties>
5           <env>development</env>
6       </properties>
7   </profile>
8   <profile>
9       <id>prod</id>
10      <properties>
11          <env>production</env>
12      </properties>
13  </profile>
14</profiles>
```

To activate a profile, use the **-P** flag when running Maven:

bash

VerifyOpen In EditorEditCopy code

1mvn clean install -Pdev

10. Best Practices

- **Keep Your POM Clean**: Avoid unnecessary dependencies and plugins.
- **Use Version Ranges**: Specify version ranges for dependencies to allow flexibility.
- **Modularize Your Project**: Break down large projects into multiple modules.
- **Use a Repository Manager**: Consider using a repository manager like Nexus or Artifactory for better dependency management.
- **Document Your POM**: Use comments in your **pom.xml** to explain configurations.

11. Conclusion

Apache Maven is a powerful tool for managing Java projects. By following this tutorial, you should now have a solid understanding of how to install Maven, create a project, manage dependencies, and build your application.

Additional Resources

- Maven Official Documentation
- Maven Repository for finding dependencies.
- Maven Best Practices

Example Project Structure

After creating a Maven project, your directory structure will look something like this:

VerifyOpen In EditorEditCopy code

```
my-app
|
├── pom.xml
└── src
    ├── main
    |   └── java
    |       └── com
    |           └── example
    |               └── App.java
    └── test
        └── java
            └── com
                └── example
                    └── AppTest.java
```

Sample Code

Here's a simple example of what the **App.java** class might look like:

java

VerifyOpen In EditorEditCopy code

```
package com.example;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello, Maven!");
    }
}
```

And a simple test in **AppTest.java**:

java

VerifyOpen In EditorEditCopy code

```
package com.example;
```

```
3import static org.junit.Assert.assertTrue;

4

5import org.junit.Test;

6

7public class AppTest {

8    @Test

9    public void testApp() {

10        assertTrue(true);

11    }

12}
```

Running the Application

After building your project with **mvn clean install**, you can run your application using the Java command:

bash

VerifyOpen In EditorEditCopy code

```
1java -cp target/my-app-1.0-SNAPSHOT.jar com.example.App
```

This command specifies the classpath to the generated JAR file and runs the **App** class.

That concludes the comprehensive tutorial on Maven 3.9! If you have any questions or need further clarification on any topic, feel free to ask!

The POM file, which stands for **Project Object Model**, is the fundamental unit of work in Maven. It is an XML file named **pom.xml** that contains information about the project and configuration details used by Maven to build the project.

Structure of a POM File

A typical **pom.xml** file contains several key elements. Below is a detailed breakdown of these elements:

Basic Structure

xml

VerifyOpen In EditorEditCopy code

```
1<project xmlns="http://maven.apache.org/POM/4.0.0"
2         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5
6    <!-- Project coordinates -->
7    <groupId>com.example</groupId>
8    <artifactId>my-app</artifactId>
9    <version>1.0-SNAPSHOT</version>
10
11    <!-- Properties -->
12    <properties>
13        <maven.compiler.source>1.8</maven.compiler.source>
14        <maven.compiler.target>1.8</maven.compiler.target>
15    </properties>
16
17    <!-- Dependencies -->
18    <dependencies>
19        <!-- Add dependencies here -->
20    </dependencies>
21
22    <!-- Build configuration -->
23    <build>
24        <plugins>
25            <!-- Add plugins here -->
26        </plugins>
27    </build>
28
29    <!-- Profiles -->
30    <profiles>
```

```
31      <!-- Add profiles here -->
32   </profiles>
33</project>
```

Key Elements of the POM File

1. **<project>**: The root element of the POM file, which contains all other elements.
2. **<modelVersion>**: Specifies the version of the POM model being used. For Maven 4.0.0, it should be set to **4.0.0**.
3. **Project Coordinates**:
    - **<groupId>**: Unique identifier for the group or organization that is producing the project (e.g., **com.example**).
    - **<artifactId>**: The unique identifier for the project itself (e.g., **my-app**). This is the name of the JAR/WAR file that Maven produces.
    - **<version>**: The version of the project (e.g., **1.0-SNAPSHOT**). The **SNAPSHOT** suffix indicates that this is a development version.
4. **<properties>**: A section for defining project properties that can be reused throughout the POM file. Common properties include Java version settings:

xml

```
1<properties>
2   <maven.compiler.source>1.8</maven.compiler.source>
3   <maven.compiler.target>1.8</maven.compiler.target>
4</properties>
```

5. **<dependencies>**: This section defines the project's dependencies. Each dependency is specified with its own **<dependency>** element, which includes:
    - **<groupId>**: The group ID of the dependency.
    - **<artifactId>**: The artifact ID of the dependency.
    - **<version>**: The version of the dependency.
    - **<scope>** (optional): The scope of the dependency (e.g., **compile**, **test**, **provided**, etc.

```
1<dependencies>
2   <dependency>
3      <groupId>junit</groupId>
4      <artifactId>junit</artifactId>
5      <version>4.13.2</version>
```

```
6      <scope>test</scope>
7    </dependency>
8</dependencies>
```

6. **<build>**: This section contains build-related information such as plugins and configurations:
    - **<plugins>**: A list of plugins that are used during the build process. Each plugin can have its own configuration.

xml

VerifyOpen In EditorEditCopy code

```
1<build>
2    <plugins>
3        <plugin>
4            <groupId>org.apache.maven.plugins</groupId>
5            <artifactId>maven-compiler-plugin</artifactId>
6            <version>3.8.1</version>
7            <configuration>
8                <source>1.8</source>
9                <target>1.8</target>
10           </configuration>
11       </plugin>
12   </plugins>
13</build>
```

7. **<profiles>**: This section allows you to define different build profiles. Each profile can modify the build process for different environments (e.g., development, production).

xml

```
1<profiles>
2    <profile>
3        <id>dev</id>
4        <properties>
5            <env>development</env>
6        </properties>
7    </profile>
```

8</profiles>

Additional Elements

- **<repositories>**

Let's continue with additional elements that can be included in the **pom.xml** file.

Additional Elements of the POM File

8. **<repositories>**: This section allows you to define additional repositories where Maven can look for dependencies. By default, Maven uses the Maven Central repository, but you can specify other repositories if needed.

```
1<repositories>
2   <repository>
3       <id>my-repo</id>
4       <url>http://my.repo.url/repository</url>
5   </repository>
6</repositories>
```

9. **<pluginRepositories>**: Similar to **<repositories>**, this section specifies additional repositories for Maven plugins.

```
1<pluginRepositories>
2   <pluginRepository>
3       <id>my-plugin-repo</id>
4       <url>http://my.repo.url/plugin-repository</url>
5   </pluginRepository>
6</pluginRepositories>
```

10. **<parent>**: If your project is a module of a multi-module project, you can specify the parent POM. This is useful for sharing configurations and dependencies across multiple modules

```
1<parent>
2   <groupId>com.example</groupId>
```

```
3   <artifactId>parent-project</artifactId>
4   <version>1.0</version>
5</parent>
```

11. **<modules>**: If your project is a multi-module project, you can define sub-modules in this section.

xml

VerifyOpen In EditorEditCopy code

```
1<modules>
2   <module>module-a</module>
3   <module>module-b</module>
4</modules>
```

12. **<dependencyManagement>**: This section is used to manage the versions of dependencies in a multi-module project. It allows you to define the versions in one place, which can then be referenced in the **<dependencies>** section of the child modules without specifying the version again.

xml

VerifyOpen In EditorEditCopy code

```
1<dependencyManagement>
2   <dependencies>
3      <dependency>
4         <groupId>com.google.guava</groupId>
5         <artifactId>guava</artifactId>
6         <version>30.1-jre</version>
7      </dependency>
8   </dependencies>
9</dependencyManagement>
```

13. **<properties>**: This section can also include custom properties that can be referenced throughout the POM file. You can use these properties to avoid hardcoding values.

xml

VerifyOpen In EditorEditCopy code

```
1<properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
```

4</properties>

14. **<reporting>**: This section is used to configure reporting plugins that generate reports for your project. You can specify which reports to generate and their configurations.

xml

VerifyOpen In EditorEditCopy code

```
1<reporting>
2   <plugins>
3      <plugin>
4         <groupId>org.apache.maven.plugins</groupId>
5         <artifactId>maven-site-plugin</artifactId>
6         <version>3.9.1</version>
7      </plugin>
8   </plugins>
9</reporting>
```

Example of a Complete POM File

Here's a complete example of a **pom.xml** file that incorporates many of the elements discussed

```
1<project xmlns="http://maven.apache.org/POM/4.0.0"
2        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
6   <groupId>com.example</groupId>
7   <artifactId>my-app</artifactId>
8   <version>1.0-SNAPSHOT</version>
9   <packaging>jar</packaging>
10
11   <properties>
12      <maven.compiler.source>1.8</maven.compiler.source>
13      <maven.compiler.target>1.8</maven.compiler.target>
14      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
```

```
16
17  <dependencies>
18      <dependency>
19          <groupId>junit</groupId>
20          <artifactId>junit</artifactId>
21          <version>4.13.2</version>
22          <scope>test</scope>
23      </dependency>
24      <dependency>
25          <groupId>com.google.guava</groupId>
26          <artifactId>guava</artifactId>
27          <version>30.1-jre</version>
28      </dependency>
29  </dependencies>
30
31  <build>
32      <plugins>
33          <plugin>
34              <groupId>org.apache.maven.plugins</groupId>
35              <artifactId>maven-compiler-plugin</artifactId
```

complete example of the **pom.xml** file, specifically focusing on the build section and additional configurations

```
1               <version>3.8.1</version>
2               <configuration>
3                   <source>${maven.compiler.source}</source>
4                   <target>${maven.compiler.target}</target>
```

```
5           </configuration>
6       </plugin>
7       <plugin>
8           <groupId>org.apache.maven.plugins</groupId>
9           <artifactId>maven-surefire-plugin</artifactId>
10          <version>2.22.2</version>
11          <configuration>
12              <includes>
13                  <include>**/*Test.java</include>
14              </includes>
15          </configuration>
16      </plugin>
17    </plugins>
18  </build>
19
20  <repositories>
21    <repository>
22      <id>central</id>
23      <url>https://repo.maven.apache.org/maven2</url>
24    </repository>
25  </repositories>
26
27  <pluginRepositories>
28    <pluginRepository>
29      <id>central</id>
30      <url>https://repo.maven.apache.org/maven2</url>
31    </pluginRepository>
32  </pluginRepositories>
33
34  <profiles>
35    <profile>
```

```
36        <id>dev</id>
37        <properties>
38          <env>development</env>
39        </properties>
40      </profile>
41      <profile>
42        <id>prod</id>
43        <properties>
44          <env>production</env>
45        </properties>
46      </profile>
47    </profiles>
48
49    <reporting>
50      <plugins>
51        <plugin>
52          <groupId>org.apache.maven.plugins</groupId>
53          <artifactId>maven-site-plugin</artifactId>
54          <version>3.9.1</version>
55          <configuration>
56            <outputDirectory>${project.build.directory}/site</outputDirectory>
57          </configuration>
58        </plugin>
59      </plugins>
60    </reporting>
61</project>
```

Breakdown of Additional Sections

1. **<build>**:
    - **<plugins>**: This section lists the plugins used during the build process.
        - **Maven Compiler Plugin**: Specifies the Java version for compilation using the properties defined earlier.

        - **Maven Surefire Plugin**: Configures the test framework to include only test classes that match the pattern ***Test.java**.

2. **<repositories>**:
    - Specifies additional repositories where Maven can look for dependencies. In this example, the central Maven repository is explicitly defined, although it's the default.
3. **<pluginRepositories>**:
    - Similar to **<repositories>**, but specifically for plugins. This allows you to specify where Maven should look for plugins.
4. **<profiles>**:
    - Defines different build profiles. Each profile can contain properties and configurations that can be activated during the build process. For example, you can activate the **dev** profile using **mvn clean install -Pdev**.
5. **<reporting>**:
    - Configures reporting plugins. In this example, the Maven Site Plugin is specified to generate a site for the project, with the output directory set to the project's target directory.

The **pom.xml** file is a powerful configuration file that defines how your project is built, its dependencies, and various settings related to the build process. Understanding how to structure and configure your **pom.xml** file is crucial for effectively using Maven in your projects.

Additional Resources

- **Maven Documentation**: The official Maven documentation provides detailed information on all aspects of the POM file: Apache Maven POM Reference.
- **Maven Plugin Documentation**: For more information on specific Maven plugins, you can visit the Maven Plugins Documentation.

If you have any further questions or need clarification on specific aspects of the **pom.xml**, feel free to ask!

Sure! In a Maven **pom.xml** file, the **<dependency>** element includes a **<scope>** element that defines the scope of the dependency. The scope determines when and how the dependency is used during the build process.

The following scopes are available:

1. **compile**: This is the default scope. The dependency is available in all classpaths: compile, test, and runtime. This means that the dependency is required during the compilation of your project, testing, and runtime.
2. **provided**: The dependency is required during the compilation of your project and testing, but it is assumed to be provided by the container or runtime environment during runtime. For example, when deploying a web application to a Java EE server, you might specify a dependency with the **provided** scope for the Java EE API, as the server already provides these libraries.
3. **runtime**: The dependency is not required during the compilation of your project, but it is required during the test and runtime phases. For example, you might use a third-party library for data access during runtime but not require it during the compilation of your project.
4. **test**: The dependency is required only during the test phase. It is not required during the compilation of your project or runtime. This is typically used for testing libraries like JUnit or Mockito.
5. **system**: This scope is similar to **provided**, but you must explicitly provide the JAR file using the **<systemPath>** element. This is not recommended, as it makes your build less portable.
6. **import**: This scope is used to import the dependencies from another POM file. It is only allowed in the **<dependencyManagement>** section and is used to manage the versions of the dependencies.

Here's an example of using different scopes in the **pom.xml** file:

```
1<dependencies>
2   <dependency>
3      <groupId>junit</groupId>
4      <artifactId>junit</artifactId>
5      <version>4.13.2</version>
6      <scope>test</scope>
7   </dependency>
8   <dependency>
9      <groupId>org.apache.logging.log4j</groupId>
10       <artifactId>log4j-api</artifactId>
11       <version>2.14.1</version>
12    </dependency>
13    <dependency>
14       <groupId>org.apache.logging.log4j</groupId>
15       <artifactId>log4j-core</artifactId>
```

```
16      <version>2.14.1</version>
17      <scope>runtime</scope>
18    </dependency>
19    <dependency>
20      <groupId>javax.servlet</groupId>
21      <artifactId>javax.servlet-api</artifactId>
22      <version>3.1.0</version>
23      <scope>provided</scope>
24    </dependency>
25</dependencies>
```

In this example, JUnit is used for testing and has a **test** scope. The **log4j-api** dependency is required during the compilation of your project, but the **log4j-core** dependency, which provides the implementation, is required only during runtime. The Servlet API is provided by the container, so it has a **provided** scope.

Using the correct scope for your dependencies ensures that your project is built and deployed correctly, without including unnecessary libraries during runtime or missing required libraries during the build process.

The **import** scope in Maven is used to import the dependency management section from another POM file. This is typically useful when you want to manage the versions of a set of dependencies in a parent POM and reference them in child POMs without having to redefine the versions.

Example of Using import Scope

Let's consider a scenario where you have a parent project that manages a set of dependencies, and you want to import those dependencies into a child project.

Parent POM (parent-pom.xml)

Here's an example of a parent POM that defines some dependencies in the **<dependencyManagement>** section:

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

```
5
6   <groupId>com.example</groupId>
7   <artifactId>parent-project</artifactId>
8   <version>1.0-SNAPSHOT</version>
9   <packaging>pom</packaging>
10
11   <dependencyManagement>
12      <dependencies>
13         <dependency>
14            <groupId>org.springframework</groupId>
15            <artifactId>spring-core</artifactId>
16            <version>5.3.10</version>
17         </dependency>
18         <dependency>
19            <groupId>org.springframework</groupId>
20            <artifactId>spring-context</artifactId>
21            <version>5.3.10</version>
22         </dependency>
23      </dependencies>
24   </dependencyManagement>
25</project>
```

Child POM (child-pom.xml)

Now, in your child project, you can import the dependency management from the parent POM using the **import** scope. Here's how you would do it:

xml

VerifyOpen In EditorEditCopy code

```
1<project xmlns="http://maven.apache.org/POM/4.0.0"
2         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
```

```
<parent>
  <groupId>com.example</groupId>
  <artifactId>parent-project</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>

<artifactId>child-project</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>parent-project</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <!-- No need to specify the version here -->
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <!-- No need to specify the version here -->
```

```
37      </dependency>
38   </dependencies>
39</project>
```

Explanation

1. **Parent POM**:
    - The parent POM (**parent-pom.xml**) defines the dependencies in the **<dependencyManagement>** section, specifying the versions for **spring-core** and **spring-context**.
2. **Child POM**:
    - The child POM (**child-pom.xml**) specifies the parent using the **<parent>** element.
    - In the **<dependencyManagement>** section of the child POM, it imports the parent POM using the **import** scope. This allows the child project to inherit the dependency versions defined in the parent POM.
    - When declaring the actual dependencies in the **<dependencies>** section, the child project does not need to specify the versions for **spring-core** and **spring-context**, as they are inherited from the parent.

Important Notes

- The **import** scope can only be used in the **<dependencyManagement>** section.
- The imported POM must be of type **pom**.
- The **import** scope is particularly useful in multi-module projects where you want to centralize dependency version management in a parent POM.

This approach helps maintain consistency across multiple modules and simplifies version management.

Maven's lifecycle is a set of predefined phases that dictate the order in which tasks are executed during the build process of a project. It provides a systematic way to manage the entire lifecycle of a project, from initial setup to deployment. Maven has three primary lifecycles:

1. **Default Lifecycle**: This is the main lifecycle used for building and deploying a project. It consists of several phases that handle the compilation, testing, packaging, and installation of the project.
2. **Clean Lifecycle**: This lifecycle is responsible for cleaning up the project. It ensures that any previously generated artifacts are removed before a new build is initiated.

3. **Site Lifecycle**: This lifecycle is used to generate project documentation and reports. It manages the creation of the project's website and associated documentation.

Maven's build lifecycle is a structured process that defines the sequence of phases for building and managing projects. There are three main lifecycles:

1. **Default Lifecycle**: Handles project deployment, including phases like compile, test, package, and install.
2. **Clean Lifecycle**: Handles project cleaning, ensuring that previous build artifacts are removed before a new build starts.
3. **Site Lifecycle**: Manages the creation of project documentation and reports.

Each lifecycle consists of a series of phases, which are executed in a specific order. Here's a breakdown of the key phases in each lifecycle:

Default Lifecycle Phases:

- **validate**: Checks if the project is correct and all necessary information is available.
- **compile**: Compiles the source code of the project.
- **test**: Runs tests using a suitable testing framework.
- **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.
- **deploy**: Copies the final package to a remote repository for sharing with other developers and projects.

Clean Lifecycle Phases:

- **pre-clean**: Executes any tasks before the clean phase.
- **clean**: Removes all files generated by the previous build.
- **post-clean**: Executes any tasks after the clean phase.

Site Lifecycle Phases:

- **pre-site**: Executes any tasks before the site generation.
- **site**: Generates the project's site documentation.
- **post-site**: Executes any tasks after the site generation.
- **site-deploy**: Deploys the generated site to a web server.

Execution of Phases:

- When a specific phase is invoked (e.g., **mvn compile**), Maven executes that phase and all preceding phases in the lifecycle.
- Goals can be bound to specific phases, allowing for customization of the build process. For example, the **maven-compiler-plugin** is typically bound to the compile phase to handle source code compilation.