1. Setting up Maven

- **Install Maven**: Download and install Apache Maven from the official website.
- **Configure Environment**: Set `M2_HOME` and `MAVEN_HOME` environment variables and update the `PATH` variable to include Maven's `bin` directory.
- **Verify Installation**: Run `mvn -v` in your terminal to ensure Maven is installed correctly.

2. Navigating a Project Structure

- **Standard Directory Layout**:
    - `src/main/java`: Application source code
    - `src/main/resources`: Application resources
    - `src/test/java`: Test source code
    - `src/test/resources`: Test resources
    - `pom.xml`: Project Object Model file

3. The POM File

The `pom.xml` file is the core of any Maven project. It defines the project's configuration, including its dependencies, build settings, plugins, and other configurations. Here's a breakdown of the main components and elements you'll find in a `pom.xml` file:

1. Basic Structure

```
<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/POM/4.0.0/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- Project coordinates -->
    <groupId>com.mphasis</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!-- Dependencies -->
    <dependencies>
        <!-- Example dependency -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.9</version>
        </dependency>
    </dependencies>

    <!-- Build settings -->
```

```
    <build>
        <plugins>
            <!-- Example plugin -->
            <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 properties -->
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
</project>
```

2. Key Elements

- **`<project>`:** Root element of the POM file.
- **`<modelVersion>`:** Specifies the POM model version (usually `4.0.0`).
- **`<groupId>`:** Unique identifier for the group of projects (often your organization's domain name).
- **`<artifactId>`:** Unique identifier for the project.
- **`<version>`:** Version of the project.
- **`<packaging>`:** Type of artifact to create (e.g., `jar`, `war`). Defaults to `jar`.

3. Dependencies

- **`<dependencies>`:** Section where you define project dependencies.
  - **`<dependency>`:**
    - **`<groupId>`:** Identifier of the group that the dependency belongs to.
    - **`<artifactId>`:** Identifier of the dependency artifact.
    - **`<version>`:** Version of the dependency.

4. Build Configuration

- **`<build>`:** Contains settings related to building the project.
  - **`<plugins>`:** Section to define plugins used during the build process.
    - **`<plugin>`:**
      - **`<groupId>`:** Plugin group identifier.
      - **`<artifactId>`:** Plugin artifact identifier.
      - **`<version>`:** Version of the plugin.
      - **`<configuration>`:** Plugin-specific configuration.

5. Properties

- `<properties>`: Section for defining project-wide properties.
  - `<property>`: Key-value pairs used throughout the POM (e.g., Java version).

6. Additional Elements

- `<repositories>`: Define additional repositories for resolving dependencies.
- `<profiles>`: Define build profiles for different environments.
- `<reporting>`: Configuration for generating reports.

Example of Profiles

```
<profiles>
    <profile>
        <id>dev</id>
        <properties>
            <env>development</env>
        </properties>
    </profile>
    <profile>
        <id>prod</id>
        <properties>
            <env>production</env>
        </properties>
    </profile>
</profiles>
```

Example of Repositories

```
<repositories>
    <repository>
        <id>my-repo</id>

<url>https://repository.mphasis.com/maven2</url>
    </repository>
</repositories>
```

4. Building, Testing, and Packaging a Project

Maven, building, testing, and packaging a project are essential steps in the development process. Here's a guide on how to perform these tasks using Maven commands and configurations:

1. Building the Project

Objective: Compile the project's source code.

- **Command**: `mvn compile`

What It Does: Compiles the source code of the project and places the compiled classes in the `target/classes` directory.

2. Testing the Project

Objective: Run tests to ensure the correctness of the code.

- **Command**: `mvn test`

What It Does: Executes the unit tests in the `src/test/java` directory. The results are usually placed in the `target/surefire-reports` directory.

Additional Testing Commands:

- **Run Tests and Generate Reports**: `mvn test` also generates test reports, which you can view to analyze test results.

3. Packaging the Project

Objective: Create a distributable artifact, such as a JAR, WAR, or EAR file.

- **Command**: `mvn package`

What It Does: Compiles the code, runs tests, and packages the compiled code and resources into the specified artifact type. By default, it packages the project into a JAR file, but this can be configured.

Example: For a web application, `mvn package` will generate a WAR file if the `<packaging>` element in `pom.xml` is set to `war`.

4. Building, Testing, and Packaging in One Command

- **Command**: `mvn install`

What It Does: Executes the complete build lifecycle, including `compile`, `test`, `package`, and `install` phases.

- **`install`**: Installs the packaged artifact into the local Maven repository (`~/.m2/repository`), making it available for other projects.

Maven Build Lifecycle

The Maven build lifecycle consists of several phases. Here's how they relate to building, testing, and packaging:

1. **validate:** Validate the project's structure and configuration.
2. **compile:** Compile the source code.
3. **test:** Run unit tests.
4. **package:** Package the compiled code into an artifact (e.g., JAR, WAR).
5. **verify**: Perform additional verification, such as integration tests.
6. **install:** Install the artifact into the local repository.
7. **deploy:** Deploy the artifact to a remote repository.

Maven Plugins for Building and Packaging

- **Compiler Plugin**: Handles compiling the code.
    - **Configuration**:

```
<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>
```

- **Surefire Plugin**: Runs unit tests.

    Configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>

        </plugin>
```

- **Jar Plugin**: Packages the project into a JAR file.
    - **Configuration** (if you need specific settings):

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.2.0</version>
    <configuration>
        <archive>
            <manifestEntries>
                <Built-By>${user.name}</Built-By>
                <Built-JDK>${java.version}</Built-JDK>
            </manifestEntries>
        </archive>
    </configuration>
```

```
</plugin>
```

Common Issues and Tips

- **Dependency Problems**: Ensure all dependencies are correctly defined and available in the repositories.
- **Test Failures**: Check the test reports in `target/surefire-reports` for details on failed tests.
- **Packaging Issues**: Verify the `<packaging>` element in `pom.xml` is correctly set for your desired output (e.g., `jar`, `war`).

5. Overview of Dependency Management and Repository

In Maven, dependency management and repositories are crucial for handling external libraries and components that your project relies on. Here's an overview of how they work and how you can configure them:

1. Dependency Management

Objective: Manage external libraries and components that your project depends on. Dependencies are specified in the `pom.xml` file.

Defining Dependencies

Dependencies are added in the `<dependencies>` section of your `pom.xml` file. Each dependency is defined with the following elements:

- **`<dependency>`**: Represents a single dependency.
    - **`<groupId>`**: The identifier of the group that the dependency belongs to.
    - **`<artifactId>`**: The identifier of the artifact (library or component).
    - **`<version>`**: The version of the dependency.
    - **`<scope>`** (optional): Defines the classpath and visibility of the dependency (e.g., `compile`, `test`, `provided`, `runtime`).

Example:

```
<dependencies>
    <!-- Dependency on JUnit for testing -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>

    <!-- Dependency on Apache Commons Lang -->
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
```

```
        <version>3.12.0</version>
    </dependency>
</dependencies>
```

Dependency Scope

- **`compile`:** Default scope. Dependencies are available at compile time and runtime.
- **`provided`**: Dependencies required for compilation but should be provided by the runtime environment (e.g., servlet API).
- **`runtime`**: Dependencies required at runtime but not at compile time.
- **`test`**: Dependencies required only for testing (e.g., JUnit).
- **`system`**: Dependencies provided by the system (requires an explicit path). Not recommended for general use.

Dependency Management Section

For managing dependency versions across multiple modules or to override versions, use the `<dependencyManagement>` section:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.12.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. Repositories

Objective: Repositories are locations where Maven looks for dependencies. They can be local (on your machine) or remote (on the internet or a private network).

Local Repository

- **Location**: **`~/.m2/repository`**
- **Usage**: Maven stores downloaded dependencies and built artifacts here. If a dependency is not found locally, Maven downloads it from remote repositories.

Remote Repositories

- **Maven Central Repository**: The default public repository for most Maven artifacts. You don't need to configure it explicitly as Maven uses it by default.
- **Other Remote Repositories**: You can configure additional remote repositories in your `pom.xml` or `settings.xml` for accessing artifacts not available in Maven Central.

Example of Adding a Remote Repository:

```
<repositories>
    <repository>
        <id>my-repo</id>

<url>https://repository.mphasis.com/maven2</url>
    </repository>
</repositories>
```

Repository Types

- **Proxy Repository**: A local repository that proxies requests to another repository. Useful for caching and offline use.
- **Host Repository**: A repository that hosts your project's artifacts, useful for internal company use.
- **Mirror**: A repository that mirrors other repositories. Configured in the `settings.xml` file to redirect all requests to a different repository.

Configuring Repositories

- **`pom.xml`**: You can define repositories specific to your project.
- **`settings.xml`**: Located in the Maven `conf` directory or `~/.m2/`, this file can define mirrors and servers for authentication.

Example of Configuring Mirrors in `settings.xml`:

```
<mirrors>
    <mirror>
        <id>central</id>
        <mirrorOf>central</mirrorOf>
        <url>https://mphasis-mirror.com/maven2</url>
    </mirror>
</mirrors>
```

Additional Notes

- **Transitive Dependencies**: Maven automatically includes dependencies of your dependencies. This helps manage complex dependency trees but can sometimes lead to version conflicts.
- **Dependency Exclusions**: You can exclude transitive dependencies if they cause conflicts or are unnecessary.

Example of Excluding a Transitive Dependency:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.3.9</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Understanding these aspects of dependency management and repositories will help you effectively manage your project's libraries and components in Maven.

6. Maven Lifecycles and Phases

In Maven, the build process is organized into lifecycles, each consisting of a series of phases. Understanding these lifecycles and phases is crucial for managing the build, testing, and deployment processes of your project. Here's an overview:

1. Maven Lifecycles

Maven defines three built-in lifecycles that manage different aspects of the build process:

- **Default Lifecycle**: Manages the main build process, including compilation, testing, and packaging.
- **Clean Lifecycle**: Handles the cleaning of the project, removing build artifacts.
- **Site Lifecycle**: Manages the creation of project documentation and reports.

2. Default Lifecycle

The Default Lifecycle is the most commonly used lifecycle. It covers the entire build process from code compilation to packaging and installation.

Phases in the Default Lifecycle

1. `validate`: Validates the project's structure and configuration. Ensures the project is correctly configured and that all required information is present.
2. `compile`: Compiles the source code of the project. The compiled code is placed in the `target/classes` directory.

3. **`test`**: Runs unit tests using a testing framework like JUnit. The test classes are placed in the `src/test/java` directory, and the results are stored in `target/surefire-reports`.
4. **`package`**: Packages the compiled code into a distributable format, such as a JAR, WAR, or EAR file. The artifact is placed in the `target` directory.
5. **`verify`**: Performs additional checks or verifications on the package, such as integration tests, to ensure it meets quality criteria.
6. **`install`**: Installs the package into the local Maven repository (`~/.m2/repository`). This makes it available for other projects on your local machine.
7. **`deploy`**: Deploys the package to a remote repository, such as a company's Maven repository or Maven Central. This step is used for making the artifact available to other developers and projects.

3. Clean Lifecycle

The Clean Lifecycle is responsible for cleaning up the project's working directory. It removes build artifacts to ensure a fresh build.

Phases in the Clean Lifecycle

1. **`pre-clean`**: Executes any actions required before the clean phase. This is often used for custom cleanup tasks.
2. **`clean`**: Deletes the `target` directory, which contains the project's build artifacts and temporary files.
3. **`post-clean`**: Executes any actions required after the clean phase. This is used for any additional cleanup tasks.

4. Site Lifecycle

The Site Lifecycle manages the generation of project documentation and reports.

Phases in the Site Lifecycle

1. **`pre-site`**: Executes any actions required before the site generation phase.
2. **`site`**: Generates the project's site documentation, including reports and site information.
3. **`post-site`**: Executes any actions required after the site generation phase.
4. **`site-deploy`**: Deploys the generated site documentation to a web server or repository.

5. Execution and Binding

- **Binding Phases**: Each phase in a lifecycle is bound to specific goals of Maven plugins. For example, the `compile` phase is bound to the `compile` goal of the `maven-compiler-plugin`, and the `test` phase is bound to the `test` goal of the `maven-surefire-plugin`.
- **Running Phases**: You can run specific phases or the entire lifecycle using Maven commands. For instance:

    - `mvn compile` will run the `validate` and `compile` phases.
    - `mvn package` will run all phases up to and including `package`.
    - `mvn clean install` will first run the `clean` lifecycle and then the `default` lifecycle up to the `install` phase.

6. Customizing Lifecycles

- **Custom Phases**: You can define custom phases and bind them to goals by creating custom plugins or using existing plugins.
- **Plugin Configuration**: Customize the behavior of plugins bound to specific phases by configuring them in your `pom.xml` file.

Example Usage

Running a Full Build:

```
mvn clean install
```

Running Only Tests:

```
mvn test
```

Generating Project Site Documentation:

```
mvn site
```

Deploying Artifacts:

```
mvn deploy
```

Summary

- **Default Lifecycle**: Covers building, testing, and packaging.
- **Clean Lifecycle**: Handles project cleanup.
- **Site Lifecycle**: Manages site documentation.

Understanding these lifecycles and phases helps you manage the Maven build process effectively and integrate it with other development activities.

7.Configuring and Using Plugins

Maven plugins are essential for extending Maven's capabilities and customizing the build process. Plugins perform tasks such as compiling code, running tests, packaging artifacts, and more. Here's how to configure and use Maven plugins effectively:

1. Understanding Maven Plugins

Plugins are used to execute tasks during the build process. They are specified in the `<build>` section of your `pom.xml` file. Each plugin can have one or more goals, which represent specific tasks.

2. Adding Plugins to Your Project

To use a plugin, you need to declare it in the `<plugins>` section within the `<build>` section of your `pom.xml` file.

Example:

```
<build>
    <plugins>
        <!-- Compiler Plugin -->
        <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>

        <!-- Surefire Plugin -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
        </plugin>
    </plugins>
</build>
```

3. Plugin Goals

Each plugin can have multiple goals. A goal is a specific task that the plugin performs. For example, the `maven-compiler-plugin` has goals like `compile` and `testCompile`, while the `maven-surefire-plugin` has goals like `test`.

Example Usage:

- **Compile Code**: `mvn compile` (This uses the `compile` goal of the `maven-compiler-plugin`).
- **Run Tests**: `mvn test` (This uses the `test` goal of the `maven-surefire-plugin`).

4. Plugin Configuration

Plugins can be configured in the `<configuration>` section of the plugin definition in `pom.xml`.

Example: Configuring the `maven-compiler-plugin` to use Java 11.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <source>11</source>
        <target>11</target>
    </configuration>
</plugin>
```

5. Using Plugin Goals

You can run specific goals directly from the command line.

Example: Running the `clean` goal of the `maven-clean-plugin`:

```
mvn clean:clean
```

Example: Running the `package` goal of the `maven-jar-plugin`:

```
mvn jar:jar
```

6. Configuring Plugin Execution Phases

Plugins are bound to specific phases of the Maven lifecycle. You can configure a plugin to execute during a specific phase by setting the `<executions>` section in the plugin configuration.

Example: Configuring the `maven-resources-plugin` to copy resources during the `process-resources` phase.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>3.2.0</version>
    <executions>
        <execution>
            <phase>process-resources</phase>
            <goals>
                <goal>copy-resources</goal>
```

```
            </goals>
            <configuration>

<outputDirectory>${project.build.directory}/resources</outputDirecto
ry>
                <resources>
                    <resource>
                        <directory>src/main/resources</directory>
                    </resource>
                </resources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

7. Common Maven Plugins

Here are some commonly used Maven plugins:

- **`maven-compiler-plugin`:** Compiles Java source files.
- **`maven-surefire-plugin`:** Runs unit tests.
- **`maven-jar-plugin`:** Packages the project into a JAR file.
- **`maven-war-plugin`**: Packages the project into a WAR file for web applications.
- **`maven-clean-plugin`:** Cleans the project by removing the `target` directory.
- **`maven-resources-plugin`:** Handles resource files, such as copying them to the output directory.
- **`maven-install-plugin`:** Installs the artifact into the local Maven repository.

8. Advanced Plugin Configuration

- **Profiles**: You can define profiles in your `pom.xml` to activate different configurations for different environments or stages of development.

Example: Defining a profile that uses a specific plugin configuration.

```
<profiles>
    <profile>
        <id>production</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.8.1</version>
                    <configuration>
                        <source>17</source>
                        <target>17</target>
                    </configuration>
                </plugin>
            </plugins>
```

```
        </build>
    </profile>
</profiles>
```

Activating a Profile:

```
mvn package -Pproduction
```

Developing a Custom Plugin

- **Create a Plugin**: Develop your own Maven plugin by implementing the `org.apache.maven.plugin.AbstractMojo` class and defining its behavior.
- **Package and Deploy**: Package the plugin as a JAR file and deploy it to a Maven repository for use in your projects.

Summary

- **Add Plugins**: Declare them in the `<plugins>` section of `pom.xml`.
- **Configure Plugins**: Use the `<configuration>` section to customize their behavior.
- **Run Goals**: Execute specific goals using Maven commands.
- **Bind to Phases**: Use `<executions>` to bind plugins to lifecycle phases.

Understanding and configuring plugins allows you to tailor Maven's build process to fit your project's needs.

8.Developing a Basic Plugin

Developing a basic Maven plugin involves several steps, including setting up a Maven project for the plugin, writing the plugin code, and packaging and deploying it. Here's a step-by-step guide to developing a basic Maven plugin:

1. Setting Up the Plugin Project

1. **Create a New Maven Project**:

   You can use the Maven Archetype plugin to generate a new Maven project for your plugin. Use the `maven-archetype-plugin` to create a basic Maven plugin structure.

   ```
   mvn archetype:generate -DgroupId=com.mphasis -DartifactId=my-maven-plugin -DarchetypeArtifactId=maven-archetype-plugin -DinteractiveMode=false
   ```

   This will create a new project with the necessary structure for a Maven plugin.

2. **Navigate to the Plugin Directory**:

   ```
   cd my-maven-plugin
   ```

2. Configuring the Plugin

Open the `pom.xml` file and update it with the following basic plugin configuration:

```
<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/POM/4.0.0/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>maven-plugin</packaging>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-plugin-plugin</artifactId>
                <version>3.6.0</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>descriptor</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-api</artifactId>
            <version>3.8.6</version>
        </dependency>
    </dependencies>

</project>
```

3. Writing the Plugin Code

1. **Create the Plugin Mojo**:

   The main class of the plugin is called a Mojo. It needs to extend `AbstractMojo` and be annotated with **`@Mojo`.** Create the class in `src/main/java/com/example/`:

```
package com.mphasis;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;

@Mojo(name = "greet")
public class GreetingMojo extends AbstractMojo {

    @Parameter(property = "name", defaultValue = "World")
    private String name;

    @Override
    public void execute() throws MojoExecutionException,
MojoFailureException {
        getLog().info("Hello, " + name + "!");
    }
}
```

- **`@Mojo(name = "greet")`**: Defines the goal of the plugin.
- **`@Parameter`**: Specifies parameters that can be configured when the plugin is executed.

2. **Compile the Plugin**:

   ```
   mvn clean install
   ```

   This command will compile your plugin and package it into a JAR file located in the `target` directory.

4. Testing the Plugin

1. **Use the Plugin in a Sample Project**:

   Create a sample Maven project to test your plugin.

```
<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/POM/4.0.0/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

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

    <build>
        <plugins>
            <plugin>
                <groupId>com.example</groupId>
                <artifactId>my-maven-plugin</artifactId>
                <version>1.0-SNAPSHOT</version>
                <executions>
                    <execution>
```

```
                        <goals>
                            <goal>greet</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

2. **Run the Plugin**:

   Execute the plugin goal in the sample project.

   ```
   mvn com.example:my-maven-plugin:1.0-SNAPSHOT:greet -Dname=John
   ```

   This command should output:

   ```
   [INFO] Hello, John!
   ```

5. Packaging and Deploying the Plugin

To share your plugin with others or use it in different projects, you need to deploy it to a repository.

1. **Deploy to a Remote Repository**:

   Ensure you have the appropriate repository settings in your `pom.xml` or `settings.xml`, then deploy using:

   ```
   mvn deploy
   ```

   This uploads the plugin to the specified remote repository.

6. Summary

- **Create Project**: Use the Maven archetype for plugins.
- **Configure Plugin**: Set up `pom.xml` with necessary dependencies and plugin configurations.
- **Write Mojo**: Implement the plugin's core functionality in Java.
- **Test Plugin**: Use the plugin in a sample project to verify functionality.
- **Deploy Plugin**: Deploy the plugin to a remote repository for distribution.

With these steps, you'll be able to create, configure, and use a basic Maven plugin.