**Maven Common Plugins**

## **Maven LifeCycle Plugins**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

## **Maven Clean Plugin**

The Maven Clean Plugin, as the name implies, attempts to clean the files and directories generated by Maven during its build. While there are plugins that generate additional files, the Clean Plugin assumes that these files are generated inside the target directory.

**Cleaning a Maven project using the command-line**

The Clean Plugin can be called to execute in the command-line without any additional configurations. Like the other plugins, to run the Clean Plugin, you use:

* $ mvn clean:clean

where the first clean refers to the plugin's alias, and the second clean refers to the plugin goal.

However, the Clean Plugin is a special plugin and is bound to its own special lifecycyle phase called clean. Thus, for simplicity, it can also be executed by using:

* $ mvn clean or with other phases/goals like:
* **$ mvn clean package site**

**Running the Clean Plugin automatically during a build**

If for some reason, adding clean to the command-line is not option, the Clean Plugin can be put into a project's pom.xml so that it gets executed everytime the project is built. Below is a sample pom.xml for running the Clean Plugin in the initialize phase everytime the project is built:

<build>  
 <plugins>  
 <plugin>  
 <artifactId>maven-clean-plugin</artifactId>  
 <version>3.1.0</version>  
 <executions>  
 <execution>  
 <id>auto-clean</id>  
 <phase>initialize</phase>  
 <goals>  
 <goal>clean</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

## **Maven Compiler Plugin**

The Compiler Plugin has two goals. Both are already bound to their proper phases within the Maven Lifecycle and are therefore, automatically executed during their respective phases.

* [compiler:compile](https://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html) is bound to the compile phase and is used to compile the main source files.
* [compiler:testCompile](https://maven.apache.org/plugins/maven-compiler-plugin/testCompile-mojo.html) is bound to the test-compile phase and is used to compile the test source files.

The goals for the Compiler Plugin are bound to their respective phases in the build lifecycle. So to compile your sources, you need only to tell maven until which lifecycle to execute. The following will compile your sources:

1. $ mvn compile

**To compile your test sources, you'll do:**

1. mvn test-compile

The above command will execute both compiler:compile and compiler:testCompile since the compile phase happens a few phases before the test-compile phase.

**Configuring Your Compiler Plugin**

Since the Compiler Plugin executes automatically during their phases, you don't have to put executions unlike many other plugins. However, you should specify the version of the Compiler Plugin.

Graphical user interface, text, application

Description automatically generated

**Setting the -source and -target of the Java Compiler**

<properties>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
</properties>

or configure the plugin directly:

<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>

## **Maven Resources Plugin**

The Resources Plugin handles the copying of project resources to the output directory. There are two different kinds of resources: main resources and test resources. The difference is that the main resources are the resources associated to the main source code while the test resources are associated to the test source code.

**Goal Overview**

The Resources Plugin copies files specified by Resource elements, to an output directory. The three variations below only differ in how the resource and output directory elements are specified or defaulted. The Resources Plugin has three goals:

**resources:resources** copies the resources for the main source code to the main output directory.

* This goal usually executes automatically, because it is bound by default to the process-resources life-cycle phase. It always uses the project.build.resources element to specify the resources, and by default uses the project.build.outputDirectory to specify the copy destination.

**resources:testResources** copies the resources for the test source code to the test output directory.

* This goal usually executes automatically, because it is bound by default to the process-test-resources life-cycle phase. It always uses the project.build.testResources element to specify the resources, and by default uses the project.build.testOutputDirectory to specify the copy destination.

**resources:copy-resources** copies resources to an output directory.

* This goal requires that you configure the resources to be copied, and specify the outputDirectory.

Copy resources for the main source code

* $ mvn resources:resources

Copy resources for the unit tests

* $ mvn resources:testResources

Copy resources specified by a <resources> configuration element

* $ mvn resources:copy-resources

**Using Properties**

<properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
</properties>

**Using Resource Plugin**

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-resources-plugin</artifactId>  
 <configuration>  
 <encoding>UTF-8</encoding>  
 </configuration>  
</plugin>

**Configure Command line maven options**

It is required to run maven goals via command line, So We have to use either -D option or MAVEN\_OPTS environment variable.

maven with -D options used to set environment variables, We can use encoding related properties as described below.

* $ mvn -Dproject.build.sourceEncoding=UTF-8
* $ mvn -Dproject.build.sourceEncoding=UTF-8 -Dproject.reporting.outputEncoding=UTF-8 clean deploy

**Global Variable**

Or You can set the Env variable Not Recommended

set "MAVEN\_OPTS=-Duser.language=fr -Dfile.encoding=UTF-8"

set MAVEN\_OPTS= -Dfile.encoding=UTF-8

**Encoding in maven compiler plugin**

Compiler plugin is a plugin used to compile java source files, That means if any characters present in source files, you might get encoding issues.

<plugins>  
 <plugin>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>2.6.0</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 <encoding>UTF-8</encoding>  
 </configuration>  
 </plugin>  
</plugins>

## **Maven SureFire Plugin**

The Surefire Plugin is used during the test phase of the build lifecycle to execute the unit tests of an application. It generates reports in two different file formats:

* Plain text files (\*.txt)
* XML files (\*.xml)

By default, these files are generated in **${basedir}/target/surefire-reports/TEST-\*.xml.**

**Goals Overview**

The Surefire Plugin has only one goal:

surefire:test runs the unit tests of an application.

The Surefire Plugin can be invoked by calling the test phase of the build lifecycle.

1. $ mvn test

**Usage**

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-plugin</artifactId>  
 <version>3.0.0-M5</version>  
</plugin>

Tests in your test source directory can be any combination of the following:

* TestNG
* JUnit (3.8, 4.x or 5.x)
* POJO

## **Maven Jar Plugin**

**T**his plugin provides the capability to build jars

**Goals Overview**

* jar:jar create a jar file for your project classes inclusive resources.
* jar:test-jar create a jar file for your project test classes .

If you need to define your own MANIFEST.MF file you can simply achieve that via Maven Archiver <http://maven.apache.org/shared/maven-archiver/index.html>

**Usage**

When you want to create a JAR-file with Maven, you first have to create a pom.xml-file with at least the following content:

<project>  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.mycompany.project</groupId>  
 <artifactId>core</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <!-- <packaging>jar</packaging> -->  
</project>

Since 'jar' is the default packaging type it is not required to set it in this case. Apart from the above you will normally want some real java source files which should be located within src/main/java. If you need extra resources on your classpath (for example property files) they should be located in src/main/resources. Now we can create a JAR-file by using the command below:

1. $ mvn package

## **Maven Deploy Plugin**

* The deploy plugin is primarily used during the deploy phase, to add your artifact(s) to a remote repository for sharing with other developers and projects. This is usually done in an integration or release environment. It can also be used to deploy a particular artifact (e.g. a third party jar like Sun's non redistributable reference implementations).

As a repository contains more than the artifacts (POMs, the metadata, MD5 and SHA1 hash files...), deploying means not only copying the artifacts, but making sure all this information is correctly updated. It's the reponsibility of the deploy plugin.

**To work, the deployment will require:**

* Information about the repository: its location, the transport method used to access it (FTP, SCP, SFTP...) and the optional user specific required account information
* Information about the artifact(s): the group, artifact, version, packaging, classifier...
* A deployer: a method to actually perform the deployment. This can be implemented as a wagon transport (making it cross-platform), or use a system specific method.
* The information will be taken from the implied (or specified) pom and from the command line. The settings.xml file may also be parsed to retrieve user credentials.

**Goals Overview**

The deploy plugin has 2 goals:

**deploy:deploy** is used to automatically install the artifact, its pom and the attached artifacts produced by a particular project. Most if not all of the information related to the deployment is stored in the project's pom.

**deploy:deploy-file** is used to install a single artifact along with its pom. In that case the artifact information can be taken from an optionally specified pomFile, but can be completed/overriden using the command line.

**Usage**

The Deploy Plugin has two basic functions. In most project builds, the deploy phase of the build lifecycle is implemented using the deploy:deploy mojo. Also, artifacts which are not built using Maven can be added to any remote repository using the deploy:deploy-file mojo.

**The deploy:deploy Mojo**

* To enable this mojo to function, you must include a valid <distributionManagement/> section POM, which at the minimum provides a <repository/> defining the remote repository location for your artifact. To separate snapshot artifacts from release artifacts, you can also specify a <snapshotRepository/> location. Finally, to deploy a project website, you must specify a <site/> section here as well. It's also important to note that this section can be inherited, allowing you to specify the deployment location one time for a set of related projects.
* If your repository is secured, you may also want to configure your settings.xml file to define corresponding <server/> entries which provides authentication information. Server entries are matched to the different parts of the distributionManagement using their <id/> elements. For example, your project may have a distributionManagement section similar to the following:

<distributionManagement>  
 <repository>  
 <id>internal.repo</id>  
 <name>MyCo Internal Repository</name>  
 <url>Host to Company Repository</url>  
 </repository>  
</distributionManagement>

In this case, you can specify a server definition in your settings.xml to provide authentication information for both of these repositories at once. Your server section might look like this:

<server>  
 <id>internal.repo</id>  
 <username>maven</username>  
 <password>foobar</password>  
</server>

1. $ mvn deploy