Maven

Table of Contents

[1 What is Maven used for? 2](#_Toc72059766)

[1.1 Creating a Maven Project 3](#_Toc72059767)

[2 Maven Coordinates 5](#_Toc72059768)

[2.1.1 So how can we generate an artifact from our project? 5](#_Toc72059769)

[2.1.2 Running all the tests of our project 5](#_Toc72059770)

[2.1.3 Creating a jar file 5](#_Toc72059771)

[3 Declarative Dependency Management 6](#_Toc72059772)

[3.1 Transitive Dependencies 6](#_Toc72059773)

[3.1.1 What’s the problem with transitive dependencies? 6](#_Toc72059774)

[3.1.2 Dependency Mediation: Strategies for picking transitive dependencies 7](#_Toc72059775)

[3.2 Dependency Scopes 8](#_Toc72059776)

[3.3 Properties 9](#_Toc72059777)

[4 Maven Settings 10](#_Toc72059778)

[4.1 Local Repository 12](#_Toc72059779)

[4.2 Snapshot Update Policy 12](#_Toc72059780)

[5 Maven Architecture in the Enterprise 13](#_Toc72059781)

[6 Running mvn install 13](#_Toc72059782)

[7 The Maven Build Lifecycle 14](#_Toc72059783)

[7.1.1 What does clean do exactly? 16](#_Toc72059784)

[7.1.2 mvn install vs mvn install:install 16](#_Toc72059785)

[7.2 test phase and the surefire Plugin 16](#_Toc72059786)

[7.3 deploy Phase and the deploy Plugin 16](#_Toc72059787)

[7.3.1 The deploy:deploy Mojo 16](#_Toc72059788)

[7.3.2 The deploy:deploy-file Mojo 17](#_Toc72059789)

[8 Maven Plugins 19](#_Toc72059790)

[8.1 Maven Shade Plugin 19](#_Toc72059791)

[8.2 Maven Plugin Development 20](#_Toc72059792)

[8.2.1 What is a Plugin? 20](#_Toc72059793)

[8.2.2 What is a Mojo (And Why the H--- is it Named 'Mojo')? 20](#_Toc72059794)

[8.2.3 Creating your first Plugin 20](#_Toc72059795)

Agenda: (This course designed for understanding and not as reference material)

1. What is Maven used for?
2. The Maven development workflow
3. Dependency management (Declarative)
4. Lifecycles and phases
5. Plugins
6. Multimodule projects

This will be an IDE-free course since IDEs do a lot of magic that normally maven does but behind the scenes, so when you use an IDE, you can’t easily see which part is from maven and which part is from the IDE.

1. What is Maven used for?

The most common answers to this question are:

1. Managing dependencies
2. It’s a build tool

Maven does have a very popular reputation for being a build tool but that’s actually not the primary goal of maven.

* The primary goal of maven is that it’s a project management tool.

This brings the question; how do you create a Java project (IDE independent)?

It turns out that there is no such concept of a project in Java! In Java you basically create a bunch of classes and then say that these all work together to build an application. Well, the application is in your head. It is not something that you declare/manifest as a Java thing. Those are just a group of java classes with some XML, assets and other files. The best you can do is to put everything in one folder and say “Here, this is our project”. But there are more to being a project than just the hierarchy of the files.

**A Java project structure**: What does a Java project need to consist of?

* Source code, all the Java files
* Test code
* Project structure (directories, assets, resources). We need to know where all the files will go.
* Libraries / dependencies. We’ll have a lot of external dependencies and those need to be managed as well. Going to the websites, downloading the dependencies we need, add them to the class path. But it’s not enough just to do it for yourself. You also need to tell your colleague to go to the website and download all the dependencies etc. When you need to update/remove/add another dependency, all those steps must be done by everyone again and again.
* Configuration: There are a lot of configuration options which specify how certain jars work. The build configuration specifies how our project is built and will run.
* Task runner – build, test, run: If you give someone your project, they wouldn’t know what to do, from where to start the app, how to build and run it. You have to explicitly tell them “You need to run this command to start the build/test/start the app”. If 10 people works on a project, you better be sure that all of them use the exact same commands for building, testing and running the app.
* Reporting: Not the most important but still relevant. We need to be able to tell how our project is going/doing. What are the number of classes, dependencies, test cases, test coverage etc.

The language does not have support for such functionalities in an IDE agnostic way. So, this is where maven comes into the picture. Maven will do all these project management steps for us. So, basically, maven manages java projects.

Maven uses the **Project Object Model** to do that.

Maven is an open-source project, actually an Apache project.

* **Provides project structure and manages content**: Maven provides the project structure for us and help us manage the contents of the project. It is opinionated, which means it comes out of the box with an opinion how the project should be structured.
* **Helps manage dependencies**: We don’t have to go to the websites, download the jars and manage them manually.
* **Configurable POM (Project Object Model)**: Maven is also very configurable thanks to the POM.
* **Maven tasks** – **goals and targets**: Provides tasks which allow you to build, compile, deploy and all those things. It comes with a task runner. You can define the order of the tasks to be run and can be configured.

Characteristics:

* **Convention over configuration**: Means that most things are set in a certain way by default and you don’t need to configure those things. As long as you follow the convention, no configuration is needed. An example: source code goes to src/main/java folder but you can configure it to look somewhere else for the source code.
* **Consistent**
* **Configurable**
* **IDE agnostic:** You don’t need an IDE to create a maven project.
* **IDE supported:** Every major Java IDE supports maven features.
* **Declarative dependency management:** Imperative would mean that you would configure all the steps exactly: website -> download -> put it in this folder … But declarative means that we just tell what dependency and which version we want and maven goes and gets it and manages it.
* **Plugin based architecture:** Most of the features that we will see in this course are powered by plugins. It adds functionality to maven and it also makes maven functionalities extensible.

**Table

Description automatically generated with medium confidenceInstalling Maven**

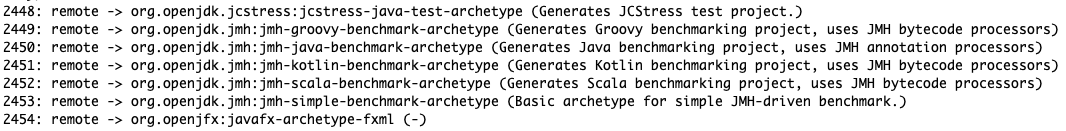
1. Verify that (AdoptOpen) JDK is installed and not just the JRE: java -version
2. Go to the maven website and download maven. The executables are in the bin folder. This folder contains the bare minimum for maven to work. (or download it with your package manager)
3. Add maven executable to the path
   1. Creating a Maven Project

We talked already about the convention of maven. We can use maven to create the project structure which follows all the conventions which makes maven happy. This can be achieved by using the concept of the Archetype. It’s essentially a project template. Maven contains various different archetypes, which are just various different project templates used as a starting point for your java project.

* mvn <action>

mvn archetype:generate

with this command maven will download a lot of information which maven needs from their online repository. Most importantly, information about all the possible archetypes it can create. Here are some archetype examples which were fetched from the online repository:



Maven asks us to pick an archetype (template) from this list:



The default is 1716, which is a bare minimum sample maven project.

A picture containing text

Description automatically generatedChoose a version of this template, the default is again the most recent version:

Now maven will ask us about 2 of the most important values:

* **Group ID**: It is an ID with which you can group all your maven projects. Similar to package names, similar classes are under the same package. This is usually the domain of your organization written backwards.
* **Artifact ID**: The ID of this particular project.
* **Version**: This is associated with the artifact that this project will generate, the starting version of your project. Which will change over the course of developing your Project. SNAPSHOT is kind of a maven concept, which means it is still under development and not the final version.
* **Package**: the package prefix of your project. Picks the same name as the group ID by default.

**What happens behind the scenes?**

Maven comes with a bare minimum and does not have much intelligence. Maven needs to talk to a remote repository in order to get all the necessary information it needs.

* By default it uses the repository: **repo1.maven.apache.org**

Diagram

Description automatically generatedThis can be changed, we can have another repository or multiple repositories. This repository has a bunch of different things but for our purposes atm, we can say it has all the archetypes that we can choose from. Again, we can also have our own list with our own archetypes.

Now when we want to create a new project with maven, maven will fetch the list of all the archetypes from that repo. The list will be returned. When we enter a number, maven then will download that particular template from the repository. Then it will create a project from that archetype. It’s the same concept as Class-Object, a new project is created from the archetpye. So we need to be connected to the internet in order to connect to the repository.

A picture containing chart

Description automatically generated

**Project Structure**

This is our java project maven generated when we picked the default bare minimum maven project archetype, which is ready to be run. But at the time of writing this, the bare maven project came with JUnit 4, and not JUnit Jupiter (JUnit 5), keep that in mind.

1. Maven Coordinates

Imagine that there is this global space where all of the artifacts that maven generates for your projects made available. In order to identify all those artifacts uniquely, maven composes an ID for every artifact with 4 parts:

* Group ID + Artifact ID + Version + Packaging
* com.google guava 5.0.0 jar

These 4 values together form the coordinates for a unique artifact, the 5.0.0 jar guava library. Lets say they release a new version, 5.1.0. Then we would have a different artifact (jar in this case), with a different set of coordinates. So each coordinate set represent the artifact uniquely. These coordinates for an artifact are called **maven coordinates**.

The version coordinate is a little different than the others because it describes the artifact that gets generated out of this project, at the time this project is being built. Simply put, we will be changing the version number to create a new artifact from our project.

It is also important to note, when you define the coordinates in the POM.xml, they do not define the project, they define the artifact which is generated from the project. And a project will generate many artifacts with different version numbers. Multiple different artifacts can be generated each time a project is built, so those jars will get the coordinates which is defined in the POM.xml.

A picture containing diagram

Description automatically generated

* + 1. So how can we generate an artifact from our project?

We can use the following maven command to compile our project:

* **mvn compile**: All classes under every package in src/main/java will be compiled and the class file will be put in target/src/java.
* **mvn test-compile**: compiles only the test classes under src/test/java

This will create a new folder called target where the all the compiled classes are stored.

Graphical user interface, text, application, email

Description automatically generated

* + 1. Running all the tests of our project

Again, maven knows where all of our test classes are located because of the convention. So with the following command we can run all the Test Classes:

* mvn test
  + 1. Creating a jar file
* **Graphical user interface, text, application

  Description automatically generated**mvn package

This will package the application to the given package format (in our case jar) with a specific naming convention: **artifact ID + version**

If we change the artifact ID or the version, a new jar will be creted in the target folder:



1. Declarative Dependency Management

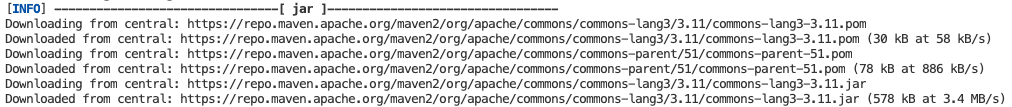
As we also discussed it with the archetypes, the same concept also applies to maven dependencies. Instead of manually downloading dependencies and the dependencies of the downloaded ones and so on, we can just declare which dependencies we want in our project and maven will get them from the central repository and make them available in the classpath.

* Text

  Description automatically generatedAll the dependencies are defined under the **<dependencies>** element
* To search for dependencies, goto <https://mvnrepository.com/>   
  There are also other repositories but this is the most popular one.

After we included the commons-lang jar in our dependency, we can use it’s API.

IDE’s will automatically download/update the dependecies when we change the POM file.

When we do a mvn compile, maven downloads the new dependency:

* 1. Transitive Dependencies

A picture containing text, whiteboard

Description automatically generatedA library which we provide in the POM can have it’s own dependencies. If not, then maven will just download the given jar and everything is fine but if the jar also needs other dependencies to function, maven will find that dependency and will include it also to your classpath.

Let’s say A is a dependency which needs C to function properly and we just include A as our dependency. Maven will also download C.

A is called a direct dependency and C transitive dependency.

* Maven pulls transitive dependencies by default.

This is a significant advantage since the developers don’t have to hunt manually for all transitive dependencies.

* + 1. What’s the problem with transitive dependencies?

But there is a nuance where this transitive dependency might lead to a problem.

Shape, arrow

Description automatically generated with medium confidenceRectangle

Description automatically generatedLet’s say our pom looks like the following, where we defined the dependencies:

* A with the version 1.0 and
* B with version 2.0

But A 1.0 depends on B 3.0.

What if maven gets all 3 of them and add all of them to the classpath? -> This will NOT work

Because in Java when you add a library to the classpath, it is like a flattened list. Every library in the classpath is its own entry, there is no nesting and nothing which can be used to provide those dependency hierarchy (A needs B 3.0 but my project needs B 2.0 and so on). Mening you can’t define “This dependeny B:3.0 should ONLY be used by A:1.0, you cant specify something like that, cannot encapsulate dependencies.

* Maven will only resolve one dependency in your module and will omit the other versions to avoid any conflict. Even if multiple versions of the same dependency are used in the whole dependency hierarchy, Maven will pick one version using the "**nearest in the dependency tree**" strategy.   
  It is possible to specify different dependency versions using different **profiles**.
  + 1. Dependency Mediation: Strategies for picking transitive dependencies

What will maven do here? Maven will do **Dependency Mediation**.

This is where things get tricky. Maven has to pick one of the 2 same artifacts (different versions).

A picture containing text, clock, gauge

Description automatically generated**Nearest definition strategy**:   
In our example, maven has to choose between B 2.0 and B 3.0. Maven picks the “nearest” one to the POM, the one with the lowest depth. B 2.0 has a depth of 1, B 3.0 has a depth of 2. So maven picks and pulls the nearest definition, which is B 2.0. But this might cause problems, since A depends on B 3.0. There is always a trade-off. If maven picked 3.0, then there is always the chance that your project which depend on 2.0 will run into problems.

A picture containing text, clock

Description automatically generated**First declaration strategy**:   
Lets say A -> C 4.0 and B -> C 5.0. In this case maven has to choose between 2 versions of C. Maven will pick the first one that it encounters, meaning that it goes through the POM, A is defined before B, so it picks the dependencies of A first and gets C 4.0.

Basically maven will first try to use the “Nearest definition strategy” but if dependencies are on the same level / at the same depth, then it picks the first one it finds.

* We can also change the first declaration strategy behaviour by forcing maven to use nearest definition. Lets say we really want C 5.0. So we can just add it to our POM directly. Now C 5.0 is a direct dependency and because of the nearest definition strategy, maven will pick C 5.0 and not C 4.0.
* **Graphical user interface, text, application, email

  Description automatically generated**Excluding dependencies: We can also exclude dependencies in POM by saying “I don’t want you to get C 4.0”. In most cases we don’t need this because we can just work with the above mentioned 2 strategies to get exactly which dependencies we want.

**Real world example**:

IntelliJ provides a usefull Maven tool which shows the dependency hierarchy:

I included the following okhttp dependency in my project and the tool shows what transitive dependencies it has. Also which dependencies were omited because of duplication or conflict.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Table

Description automatically generated with medium confidenceTable

Description automatically generated with medium confidence

* 1. Dependency Scopes

Every dependency we declare in our POM has a scope associated with it. That scope tells maven where to use that dependency. For instance our commons-lang3 dependency, we want to add it to the classpath when we compile the code, run test cases and also while running the application. So we need it pretty much in every step of the way. But not all dependencies are needed in every step.

Junit for example, we don’t need it when we compile our app (test classes won’t be compiled with mvn compile). We only need it when we want to compile and run our test classes. When we use the test scope, we are telling maven to only use Junit during the test phase.

* **compile** is the default scope, when not provided

So in a high level there are:

1. Compile Classpath
2. Test Classpath
3. Runtime Classpath (all the jars contained in the final distributable)

and depending on the scope, maven will include or exclude dependencies from these classpaths.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, table

Description automatically generated

* 1. Properties

Graphical user interface, text, application, email

Description automatically generated

* It’s recommended to define the version numbers of dependencies as properties.
* Most properties defined in the parent POM are inherited by the sub-modules in a multi-module-project

The Maven settings file, settings.xml, is usually kept in the .m2 directory inside your home directory

1. Maven Settings

The **settings** element in the settings.xml file contains elements used to define values which configure Maven execution in various ways, like the pom.xml, but should not be bundled to any specific project, or distributed to an audience.

These include values such as the

* local repository location,
* Graphical user interface, text, application

  Description automatically generatedalternate remote repository servers and
* authentication information.

There are two locations where a **settings.xml** file may live:

* **The Maven install**: **${maven.home}/conf/settings.xml** (also called **Global Settings**)
* **A user’s install: ${user.home}/.m2/settings.xml** (also called **User Settings**)

If both files exists, their contents gets merged, with the user-specific settings.xml being dominant.

* **localRepository**: This value is the path of this build system’s local repository. The default value is ${user.home}/.m2/repository. This element is especially useful for a main build server allowing all logged-in users to build from a common local repository.
* **interactiveMode**: true if Maven should attempt to interact with the user for input, false if not. Defaults to true.
* **offline**: true if this build system should operate in offline mode, defaults to false. This element is useful for build servers which cannot connect to a remote repository, either because of network setup or security reasons.
* **pluginGroups**: This element contains a list of pluginGroup elements, each contains a groupId. The list is searched when a plugin is used and the groupId is not provided in the command line. This list automatically contains org.apache.maven.plugins and org.codehaus.mojo. Therefore we don’t have to specify the GroupID when using plugins like clean, compile, deploy etc.  
  For example with an entry like **<pluginGroup>org.eclipse.jetty</pluginGroup>**, the command:
  + **mvn org.eclipse.jetty:jetty-maven-plugin:run** turns into
  + **mvn jetty:run**

Graphical user interface, text, application

Description automatically generated**Servers**

The repositories for download and deployment are defined by the **repositories** and **distributionManagement** elements of the POM. However, certain settings such as **username** and **password** should not be distributed along with the pom.xml. This type of information should exist on the build server in the settings.xml.

**Mirrors,**

Graphical user interface, text, application, Teams

Description automatically generated

**TODO:** <https://stackoverflow.com/questions/36757902/what-is-mirror-in-maven-settings-xml-file>

Graphical user interface, text, application, Teams

Description automatically generated**Proxies**

TODO <https://www.baeldung.com/maven-behind-proxy>

**Profiles**

Graphical user interface

Description automatically generated with medium confidence

**ActiveProfiles**

TODO

asd

* 1. Local Repository

Table

Description automatically generatedTable

Description automatically generatedMaven stores the dependencies it downloaded on the machine, so it doesn’t have to download the jar each time it’s needed or when a new project uses the same jar.

All dependencies are stored in the   
**~/.m2/repository** folder (defined in Settings.xml)

Dependencies are saved hierarchically:

* Group ID -> List of artifact IDs -> Version List -> Artifact
* There is also a settings.xml file under .m2/ where maven can be configured
  1. Graphical user interface, text, application, Teams

     Description automatically generatedSnapshot Update Policy

Graphical user interface, text, application

Description automatically generated

* Can be updated in the <repositories> element in pom.xml for each repository

**Important**: Maven will check the remote repository at the configured time and **ONLY IF** the checksum is different between the local SNAPSHOT-jar and the remote SNAPSHOT-jar, then it will download the newer SNAPSHOT jar and will replace it with the current one (in the .m2 directory). Maven uses time-stamps to determine if a remote SNAPSHOT is new. Because the idea of a snapshot is that it has a fixed version number (ie 1.2-SNAPSHOT) which means that version 1.2 is still being developed. So when a new SNAPSHOT is uploaded, the version will stay the same (1.2-SNAPSHOT) but the timestamp will change. That’s how maven can decide if it needs to be downloaded or not.

A project can have multiple repositories defined. For instance internal **Nexus** repositories can be used when developing an application. Another team might publish a SNAPSHOT there which you use in your project as a dependency, so when you check daily, it might be too slow and you might not get the updates from the remote repository. Keep that in mind when defining your own repository and working with SNAPSHOTS.

1. Maven Architecture in the Enterprise

Usually when working in an enterprise application, typically the company will have their own repository setup, which can only be reached from the company network. The architecture will more or less will look like this:

Diagram

Description automatically generatedA company-intern (local) maven repository will be maintained within the companies network. This local repo will more or less work like a cache for another external library. Employees will configure their central repo as the companies repository. Any time someone wants to install a dependency, maven first will talk to the internal repo, if it can’t find the dependency there then the request will be redirected to an external (global) maven repository and then the newly requested jar will be stored/cached also in the companies repo.

For internal development: now developers can work on an internal project and publish those jars to the internal repo. This way the private project stays in the company. Other employees can then fetch the newly deployed jar.

A lot of companies in certain domains like finance, banking, government etc. Will have hard restrictions set on which public libraries they are allowed to use. All those will be then configured at the companies repository. The jars which are not allowed will simply not show up when requested from maven.

1. Running mvn install

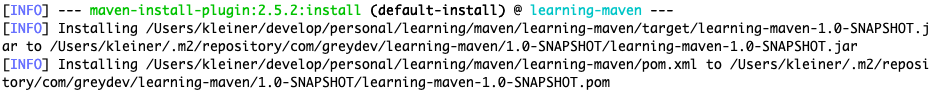
When we do **mvn package** it generates a jar file and puts it to the target folder.

When we do **mvn install**, it does the same but then it also puts that jar into the **.m2** local repository (publishes/installs to the local repository).

Graphical user interface, text, application

Description automatically generatedDiagram

Description automatically generated with medium confidence



This way we can actually use this dependency in other projects. We could just add the group ID and artifact ID to our dependency list and maven will first check the local .m2 directory. It will find the jar there and will make it available.

1. The Maven Build Lifecycle

There are three built-in build lifecycles:

1. **default**: The default lifecycle handles your project deployment
2. **clean**: the clean lifecycle handles project cleaning
3. **site:** the site lifecycle handles the creation of your project's site documentation

**A Build Lifecycle is Made Up of Phases**. Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

Text

Description automatically generatedFor example, the default lifecycle comprises of the following **phases** (there are more! These ones are the main phases):

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

* You only need to call the last build phase to be executed. The ones before the given phase will always be executed automatically. For instance if you execute the package phase, then validate, compile and test will be executed respectively before executing package.
* The same command (mvn clean deploy) can be used in a multi-module scenario (i.e. a project with one or more subprojects). Maven traverses into every subproject and executes clean, then executes deploy (including all of the prior build phase steps).

**A Build Phase is Made Up of Plugin Goals**: However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the plugin goals bound to those build phases.

A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation. The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The clean and package arguments are build phases, while the dependency:copy-dependencies is a goal (of a plugin).

* mvn clean dependency:copy-dependencies package

If this were to be executed, the clean **phase** will be executed first (meaning it will run all preceding phases of the clean lifecycle, plus the clean phase itself), and then the dependency:copy-dependencies **goal**, before finally executing the package **phase** (and all its preceding build phases of the default lifecycle).

Moreover, if a **goal** is bound to one or more **build phases**, that **goal** will be called in all those **phases**.

Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals.

(Note: In Maven 2.0.5 and above, multiple goals bound to a phase are executed in the same order as they are declared in the POM)

Graphical user interface, application

Description automatically generated with medium confidence**Packaging**: The first, and most common way to add goals to phases, is to set the packaging for your project via the equally named POM element <packaging>. Some of the valid packaging values are jar, war, ear, maven-plugin and pom. If no packaging value has been specified, it will default to jar.

Each packaging contains a list of goals to bind to a particular phase. For example, the jar packaging will bind the following goals to build phases of the default lifecycle.

This is an almost standard set of bindings; however, some packagings handle them differently. For example, a project that is purely metadata (packaging value is pom) only binds goals to the install and deploy phases (for a complete list of goal-to-build-phase bindings of some of the packaging types, refer to the Lifecycle Reference).

**Plugins:** The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

* Plugins can contain information that indicates which lifecycle phase to bind a goal to. Note that adding the plugin on its own is not enough information - you must also specify the goals you want to run as part of your build.

Graphical user interface, text, application

Description automatically generatedThe goals that are configured will be added to the goals already bound to the lifecycle from the packaging selected. If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM. Note that you can use the <executions> element to gain more control over the order of particular goals.

Example: I’ve created a custom plugin (named my-custom-plugin) with a **goal** called sayhi, which is bound to the compile phase by default.

Graphical user interface, text, application

Description automatically generatedThis means, whenever this plugin is included in a project **AND** this goal is added to the <executions> element (remember, adding just the plugin is not enough if you want its goals to be executed with a phase. Ofc, you can always just add the plugin and manually/separately invoke its goals, but we want to execute those goals during phases) then, when the compile phase runs, it will also execute the goal sayhi.

Timeline

Description automatically generatedThe default phase configured inside the plugin (in our case compile) can also be **overridden**. This means now when we execute mvn compile, the sayhi goal WON’T be executed. But when we run mvn install, now the sayhi goal will be executed.

* + 1. What does clean do exactly?

Maven clean goal (clean:clean) is bound to the clean **phase** in the clean **lifecycle**. Its clean:clean goal deletes the output of a build by deleting the build directory. Thus, when mvn clean command executes, Maven deletes the build directory (./target).

* It **does NOT** delete the locally installed articfact, in the .m2/repositories dir.
  + 1. mvn install vs mvn install:install
* **install** is a **Maven phase**. It first invokes all phases prior to it in the lifecycle (e.g., compile, test, and package to name a few) and than invokes whichever mojo "goals" are attached to it.
* **install::install** is a **mojo "goal"**, which performs a specific task (copies the target artifacts from the current project into the local repo).
  1. test phase and the 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. These reports can then be used by the different testing frameworks (TestNG, JUnit, etc.) to display the test results in a more structured way. For instance, Jenkins also uses these reports to display information about the tests, about how many tests passed and so on.

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

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

Which providers are available is controlled simply by the inclusion of the appropriate dependencies (i.e., junit:junit or junit:junit-dep for **JUnit4**, junit-jupiter-engine or junit-vintage-engine for **JUnit5** and org.testng:testng 4.7+ for **TestNG**). Since this is required to compile the test classes anyway, no additional configuration is required.

Note that any normal Surefire integration works identically no matter which providers are in use - so you can still produce a **Cobertura report** and a **Surefire results report** on your project web site for your TestNG tests, for example.

The POJO provider above allows you to write tests that do not depend on either of JUnit and TestNG. It behaves in the same way, running all test\* methods that are public in the class, but the API dependency is not required. To perform assertions, the JDK 1.4 assert keyword can be used. See Using POJO Tests for more information.

* 1. deploy Phase and the deploy Plugin

TODO deploy plugin, how do we use it in jenkins etc.

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).

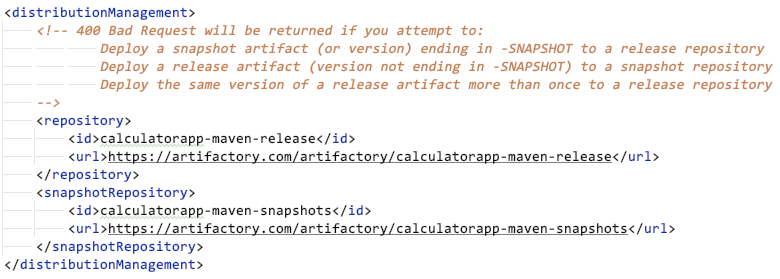
The Deploy Plugin has two basic functions. In most project builds, the deploy **phase** of the build lifecycle is implemented using the deploy:deploy **mojo**. Also, artifacts which are not built using Maven can be added to any remote repository using the deploy:deploy-file **mojo**.

* + 1. The deploy:deploy Mojo

In most cases, this mojo is invoked when you call the deploy phase of the default build lifecycle.

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.

* **distributionManagement**: Specifies where the artifacts will be deployed (The **deploy:deploy** Mojo invoked during the **deploy** phase of the default build lifecycle)
  + **repository**: All release-artifacts (without a -SNAPSHOT at the end) will be deployed here.  
    When no snapshotRepository is specified, all artifacts will be deployed here
  + **snapshotRepository**: when an artifact has a -SNAPSHOT suffix, it will be deployed here



* **repositories**: By default Maven searches the central repository at <https://repo.maven.apache.org/maven2/>   
  Additional repositories can be configured in the pom.xml `**repositories**` element.   
  Here we can also update the **updatePolicy**, **checksumPolicy, enabled** and **layout** of each repo

Graphical user interface, text, application, email

Description automatically generatedWe also need to specify the credentials for the servers in settings.xml ->

Please see the article about **Password Encryption** for instructions on how to avoid clear text passwords in the settings.xml.

An example from Jenkins:

Text, letter

Description automatically generated

* + 1. The deploy:deploy-file Mojo

A picture containing application

Description automatically generatedThe deploy:deploy-file **mojo** is used primarily for deploying artifacts which were not built by Maven. The project's development team may or may not provide a POM for the artifact, and in some cases you may want to deploy the artifact to an internal remote repository. The deploy-file mojo provides functionality covering all of these use cases, and offers a wide range of configurability for generating a POM on-the-fly. Additionally, you can specify what layout your repository uses. The full usage statement of the deploy-file mojo can be described as:

* + 1. Building an executable JAR (“no main manifest attribute” when running java -jar …)

Graphical user interface, text, application, email

Description automatically generatedTo make a jar executable you need a file called   
**META-INF/MANIFEST.MF**, the file itself should have (at least) this one liner:

* Main-Class: com.mypackage.MyClass

Where **com.mypackage.MyClass** is the class holding the public static void main(String[] args) entry point.

For Maven, something like the following snippet should do the trick. Note that this is only the plugin definition, not the full pom.xml

**Building a Jar with Maven (a fat-JAR)**

By default, Maven **doesn't** bundle dependencies in the JAR file it builds, and if you're not providing them on the classpath when you're trying to execute your JAR file at the command-line then you’ll get a NoClassDefFoundError Exception when you run **java -jar ./target/my-jar-1.0-SNAPSHOT**

* You could manually specify the libraries on the classpath with the **-cp** parameter, but that quickly becomes tiresome.

Graphical user interface, application

Description automatically generatedA better solution is to "**shade**" the library code into your output JAR file. There is a Maven plugin called the **maven-shade-plugin** to do this. You need to register it in your POM, and it will automatically build an "**uber-JAR / fat-JAR / executable-JAR**" containing your classes and the classes for your library code too when you run mvn package.

* It was also done with **maven-assembly-plugin** but it is now   
  DEPRECATED, so better use the **maven-shade-plugin**

**TODO:**

**With Spring Boot we use spring-boot-maven-plugin, does this “override” the original JAR and therefore rename the first created jar as .original? How does that work?**

1. Maven Plugins
   1. Maven Shade Plugin

Graphical user interface, text, application, email

Description automatically generatedRunning the mvn package again will produce the following output by the maven shade plugin:

This is the contents of the generated jar after **mvn package** is called WITOUT the shade Plugin:

A picture containing timeline

Description automatically generated

Graphical user interface, application

Description automatically generatedTable

Description automatically generated

With the shade Plugin, we can see all the dependencies are included:

Two jar files will be created in the target folder.

1. **notion-backup-1.0-SNAPSHOT.jar** - Project and dependency classes in a single jar, this is what you want.
2. A picture containing timeline

   Description automatically generated**original-notion-backup-1.0-SNAPSHOT.jar** - Only your project classes

**Spring Boot**

For a Spring Boot application, the **spring-boot-maven-plugin** does the creation of the fat jar!

* 1. Maven Plugin Development
     1. What is a Plugin?

"Maven" is really just a core framework for a collection of Maven Plugins. In other words, plugins are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on. Almost any action that you can think of performing on a project is implemented as a Maven plugin.

Plugins are the central feature of Maven that allow for the reuse of common build logic across multiple projects. They do this by executing an "action" (i.e. creating a WAR file or compiling unit tests) in the context of a project's description - the Project Object Model (POM). Plugin behavior can be customized through a set of unique parameters which are exposed by a description of each plugin goal (or Mojo).

One of the simplest plugins in Maven is the Clean Plugin. The [Maven Clean plugin](https://maven.apache.org/plugins/maven-clean-plugin/) (maven-clean-plugin) is responsible for removing the target directory of a Maven project. When you run "mvn clean", Maven executes the "clean" goal as defined in the Clean plug-in, and the target directory is removed. The Clean plugin [defines a parameter](https://maven.apache.org/plugins/maven-clean-plugin/clean-mojo.html) which can be used to customize plugin behavior, this parameter is called outputDirectory and it defaults to ${project.build.directory}.

* + 1. What is a Mojo (And Why the H--- is it Named 'Mojo')?

A Mojo is really just a **goal** in Maven, and plug-ins consist of any number of **goals** (**Mojos**). Mojos can be defined as annotated Java classes or Beanshell script. A Mojo specifies metadata about a goal: a goal name, which phase of the lifecycle it fits into, and the parameters it is expecting.

MOJO is a play on POJO (Plain-old-Java-object), substituting "Maven" for "Plain". Mojo is also an interesting word (see definition). From Wikipedia, a "mojo" is defined as: "...a small bag worn by a person under the clothes (also known as a mojo hand). Such bags were thought to have supernatural powers, such as protecting from evil, bringing good luck, etc."

* + 1. Creating your first Plugin

**Naming conventions**: You will typically name your plugin <yourplugin>-maven-plugin.

Calling it maven-<yourplugin>-plugin (note "Maven" is at the beginning of the plugin name) is **strongly discouraged** since it's a reserved naming pattern for official Apache Maven plugins maintained by the Apache Maven team with groupId org.apache.maven.plugins. Using this naming pattern is an infringement of the Apache Maven Trademark.

At its simplest, a Java mojo consists simply of a single class representing one plugin's goal. There is no requirement for multiple classes like EJBs, although a plugin which contains a number of similar mojos is likely to use an abstract superclass for the mojos to consolidate code common to all mojos. When processing the source tree to find mojos, **plugin-tools** looks for classes with either **@Mojo** Java 5 annotation or "**goal**" javadoc annotation. Any class with this annotation are included in the plugin configuration file.

* Mojo is a Java class that represents a goal that our plugin will execute. A plugin contains one or more mojos.

**Create a new maven project with the following pom.xml**:

|  |
| --- |
| <artifactId>my-custom-plugin</artifactId> <name>my-custom-plugin (Maven Plugin)</name> *<!-- Packaging -->* <packaging>maven-plugin</packaging>  <dependencies>  *<!-- The maven-plugin-api dependency is required and   contains the necessary classes and interfaces to create our mojo.-->* <dependency>  <groupId>org.apache.maven</groupId>  <artifactId>maven-plugin-api</artifactId>  <version>3.8.1</version>  </dependency>  *<!-- dependencies to annotations -->* <dependency>  <groupId>org.apache.maven.plugin-tools</groupId>  <artifactId>maven-plugin-annotations</artifactId>  <version>3.6.1</version>  <scope>provided</scope>  </dependency> </dependencies>  *<!-- For JDK11 support there is a bug in version 3.0,3.1,3.2. You must use 3.3 or later: -->* <build>  <pluginManagement>  <plugins>  <plugin>  <artifactId>maven-plugin-plugin</artifactId>  <version>3.6.1</version>  </plugin>  </plugins>  </pluginManagement> </build> |

Apperantly there is a bug with the older versions of maven-plugin-plugin and Java 11, so I had to specify the plugin version.

Text

Description automatically generatedA picture containing table

Description automatically generatedI was getting the following error: (when plugin version is 3.2, see right)

**Implement Mojo/Goal**

|  |
| --- |
| import org.apache.maven.plugin.AbstractMojo; import org.apache.maven.plugin.MojoExecutionException; import org.apache.maven.plugins.annotations.Mojo;  */\*\*  \* Says "Hi" to the user.  \*/* @Mojo(name = **"sayhi"**) public class MyCustomMojo extends AbstractMojo {  public void **execute**() throws MojoExecutionException {  getLog().info(**"Hello, world."**);  } } |

Text

Description automatically generated with low confidence

**Graphical user interface, text, application, email

Description automatically generatedBuilding / Running the plugin**

Built the plugin with **mvn install.** Then I’ve added to following Plugin definition to a different project. Since I’ve installed **my-custom-plugin**, this project was able to find the dependency in the **.m2** repository, else it wouldnt be able to compile the project since it wouldnt be able to find the dependency.

Text

Description automatically generatedAfter adding this plugin into my project, I was able to see the new plugin **my-custom** and the new Mojo/Goal **sayhi**:

You need to specify a fully-qualified goal in the form of (version not required in a standalone goal):

* **mvn groupId:artifactId:version:goal**
* **mvn com.greydev:my-custom-plugin:0.0.1-SNAPSHOT:sayhi**
* **mvn com.greydev:my-custom-plugin:sayhi**

Table

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated**Shortening the Command Line**

Timeline

Description automatically generated with medium confidence**Attaching the Mojo (Plugin goal) to the Build Lifecycle**

You can also configure your plugin to attach specific goals to a particular phase of the build lifecycle. Here is an example:

* This causes the simple mojo to be executed whenever Java code is **compiled**.

asd

Asdasd

**TODO**: Bill of Materials



**TODO:**

When there are multiple jars (original, exec etc.) all are deployed to NEXUS

Whats a jar-specifier, we use **exec** in our projects as the specifier, why?

Test-reports, multiple projects???

**Build only certain modules:**

Only build module1 and module2:

mvn clean install -DskipTests=true **-pl** **\!module1**,**\!module2**

Exclude module1 and module2

mvn clean install -DskipTests=true **-pl module1,module2**

We need to escape ! for the shell, that’s why the back slashes.