**Maven POM File**

A Maven POM file (Project Object Model) is an XML file that describes the resources of the project. This includes the directories where the source code, test source etc. is located in, what external dependencies (JAR files) your projects has etc.

The POM file describes *what* to build, but most often not *how* to build it. How to build it is up to the Maven build phases and goals. You can insert custom actions (goals) into the Maven build phase if you need to, though.

Each project has a POM file. The POM file is named pom.xml and should be located in the root directory of your project. A project divided into subprojects will typically have one POM file for the parent project, and one POM file for each subproject. This structure allows both the total project to be built in one step, or any of the subprojects to be built separately.

For a full reference of the POM file, see the [**Maven POM Reference**](http://maven.apache.org/pom.html).

Here is a minimal POM file:

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jenkov</groupId>

<artifactId>java-web-crawler</artifactId>

<version>1.0.0</version>

</project>

The ***modelVersion*** element sets what version of the POM model you are using. Use the one matching the Maven version you are using. Version 4.0.0 matches Maven versions 2 and 3.

The ***groupId*** element is a unique ID for an organization, or a project (an open source project, for instance). Most often you will use a group ID that is similar to the root Java package name of the project. For instance, for my Java Web Crawler project, I may choose the group ID com.jenkov. If the project was an open source project with many independent contributors, perhaps it would make more sense to use a group ID related to the project than an a group ID related to my company. Thus, com.javawebcrawler could be used.

The group ID does not have to be a Java package name, and does not need to use the . notation (dot notation) for separating words in the ID. But, if you do, the project will be located in the Maven repository under a directory structure matching the group ID. Each **.** is replaced with a directory separator, and each word thus represents a directory. The group ID **com.jenkov** would then be located in a directory called **MAVEN\_REPO/com/jenkov**. The **MAVEN\_REPO** part of the directory name will be **replaced with the directory path of the Maven repository(.m2)**.

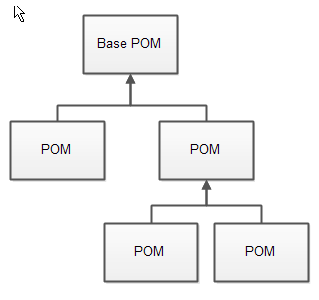
The ***artifactId*** element contains the name of the project you are building. In the case of my Java Web Crawler project, the artifact ID would be java-web-crawler. The artifact ID is used as a name for a subdirectory under the group ID directory in the Maven repository. The artifact ID is also used as part of the name of the JAR file produced when building the project. The output of the build process, the build result that is, is called an artifact in Maven. Most often it is a JAR, WAR or EAR file, but it could also be something else.

The ***versionId*** element contains the version number of the project. If your project has been released in different versions, for instance an open source API, then it is useful to version the builds. That way users of your project can refer to a specific version of your project. The version number is used as a name for a subdirectory under the artifact ID directory. The version number is also used as part of the name of the artifact built.

The above *groupId*, *artifactId* and *version* elements would result in a JAR file being built and put into the local Maven repository at the following path (directory and file name):

**MAVEN\_REPO/com/jenkov/java-web-crawler/1.0.0/java-web-crawler-1.0.0.jar**

**Super POM**

All Maven POM files inherit from a super POM. If no super POM is specified, the POM file inherits from the base POM. Here is a diagram illustrating that:

You can make a POM file explicitly inherit from another POM file. That way you can change the settings across all inheriting POM's via their common super POM. You specify the super POM at the top of a POM file like this:

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.codehaus.mojo</groupId>

<artifactId>my-parent</artifactId>

<version>2.0</version>

<relativePath>../my-parent</relativePath>

</parent>

<artifactId>my-project</artifactId>

...

</project>

An inheriting POM file may override settings from a super POM. Just specify new settings in the inheriting POM file.

**Effective POM**

With all this POM inheritance it may be hard to know what the total POM file looks like when Maven executes. The total POM file (result of all inheritance) is called the *effective POM*. You can get Maven to show you the effective POM using this command:

mvn help:effective-pom

This command will make Maven write out the effective POM to the command line prompt.

**Maven Settings File**

Maven has two settings files. In the settings files you can configure settings for Maven across all Maven POM files. For instance, you can configure:

* Location of local repository
* Active build profile
* Etc.

The settings files are called settings.xml. The two settings files are located at:

* The Maven installation directory: $M2\_HOME/conf/settings.xml
* The user's home directory: ${user.home}/.m2/settings.xml

Both files are optional. If both files are present, the values in the user home settings file overrides the values in the Maven installation settings file.

**Running Maven**

When you have [**installed Maven**](https://jenkov.com/tutorials/maven/maven-tutorial.html#installing-maven) and have created a [**POM file**](https://jenkov.com/tutorials/maven/maven-tutorial.html#maven-pom-files) and put the POM file in the root directory of your project, you can run Maven on your project.

Running Maven is done by executing the **mvn** command from a command prompt. When executing the mvn command you pass the name of a [**build life cycle, phase or goal**](https://jenkov.com/tutorials/maven/maven-tutorial.html#maven-build-life-cycles-phases-and-goals) to it, which Maven then executes. Here is an example:

**mvn install**

This command executes the build phase called install (**part of the default build life cycle**), which builds the project and copies the packaged JAR file into the local Maven repository. Actually, this command executes ***all build phases before install in the build phase sequence***, before executing the install build phase.

You can execute multiple build life cycles or phases by passing more than one argument to the mvn command. Here is an example:

**mvn clean install**

This command first executes the clean build life cycle, which removes compiled classes from the Maven output directory, and then it executes the install build phase.

**Maven Directory Structure**

Maven has a standard directory structure. If you follow that directory structure for your project, you do not need to specify the directories of your source code, test code etc. in your POM file.

I have covered the Maven directory structure in more detail here: [**Maven Directory Structure**](https://jenkov.com/tutorials/maven/directory-structure.html).

Here are the most important directories:

- src

- main

- java

- resources

- webapp

- test

- java

- resources

- target

The ***src*** directory is the root directory of your source code and test code. The **main** directory is the root directory for source code related to the application itself (not test code).

The **test** directory contains the test source code. The java directories under main and test contain the Java code for the application itself (under main) and the Java code for the tests (under test).

The **resources** directory contains other resources needed by your project. These could be property files used for the internationalization of an application, or something else.

The **webapp** directory contains your Java web application, if your project is a web application. The webapp directory will then be the root directory of the web application. Thus the webapp directory contains the WEB-INF directory etc.

The target directory is created by Maven. It contains all the compiled classes, JAR files etc. produced by Maven. When executing the clean build phase, it is the target directory that is cleaned.

**Project Dependencies**

Unless your project is small, your project may need external Java APIs or frameworks which are packaged in their own JAR files. These JAR files are needed on the classpath when you compile your project code.

Keeping your project up-to-date with the correct versions of these external JAR files can be a comprehensive task. Each external JAR may again also need other external JAR files etc. Downloading all these external dependencies (JAR files) recursively and making sure that the right versions are downloaded is cumbersome. Especially when your project grows big, and you get more and more external dependencies.

Luckily, Maven has built-in dependency management. You specify in the POM file what external libraries your project depends on, and which version, and then Maven downloads them for you and puts them in your local Maven repository. If any of these external libraries need other libraries, then these other libraries are also downloaded into your local Maven repository.

You specify your project dependencies inside the dependencies element in the POM file. Here is an example:

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jenkov.crawler</groupId>

<artifactId>java-web-crawler</artifactId>

<version>1.0.0</version>

**<dependencies>**

<**dependency**>

<groupId>org.jsoup</groupId>

<artifactId>jsoup</artifactId>

<version>1.7.1</version>

</**dependency**>

<**dependency**>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.8.1</version>

<scope>test</scope>

</**dependency**>

**</dependencies>**

<build>

</build>

</project>

Notice the dependencies element in bold. Inside it are two dependency elements. Each dependency element describes an external dependency.

Each dependency is described by its groupId, artifactId and version. You may remember that this is also how you identified your own project in the beginning of the POM file. The example above needs the org.jsoup group's jsoup artifact in version 1.7.1, and the junit group's junit artifact in version 4.8.1.

When this POM file is executed by Maven, the two dependencies will be downloaded from a central Maven repository and put into your local Maven repository. If the dependencies are already found in your local repository, Maven will not download them. Only if the dependencies are missing will they be downloaded into your local repository.

Sometimes a given dependency is not available in the central Maven repository. You can then download the dependency yourself and put it into your local Maven repository. Remember to put it into a subdirectory structure matching the groupId, artifactId and version. Replace all dots (.) with / and separate the groupId, artifactId and version with / too. Then you have your subdirectory structure.

The two dependencies downloaded by the example above will be put into the following subdirectories:

MAVEN\_REPOSITORY\_ROOT/junit/junit/4.8.1

MAVEN\_REPOSITORY\_ROOT/org/jsoup/jsoup/1.7.1

* **External Dependencies:**

An external dependency in Maven is a dependency (JAR file) which is not located in a Maven repository (neither local, central or remote repository). It may be located somewhere on your local hard disk, for instance in the lib directory of a webapp, or somewhere else. The word "external" thus means external to the Maven repository system - not just external to the project. Most dependencies are external to the project, but few are external to the repository system (not located in a repository).

You configure an external dependency like this:

<dependency>

<groupId>mydependency</groupId>

<artifactId>mydependency</artifactId>

**<scope>system</scope>**

**<version>1.0</version>**

**<systemPath>${basedir}\war\WEB-INF\lib\mydependency.jar</systemPath>**

</dependency>

The groupId and artifactId are both set to the name of the dependency. The name of the API used, that is. The scope element value is set to system. The systemPath element is set to point to the location of the JAR file containing the dependency. The ${basedir} points to the directory where the POM is located. The rest of the path is relative from that directory.

* **Snapshot Dependencies**

Snapshot dependencies are dependencies (JAR files) which are under development. Instead of constantly updating the version numbers to get the latest version, you can depend on a snapshot version of the project. Snapshot versions are always downloaded into your local repository for every build, even if a matching snapshot version is already located in your local repository. Always downloading the snapshot dependencies assures that you always have the latest version in your local repository, for every build.

You can tell Maven that your project is a snapshot version simply by appending -SNAPSHOT to the version number in the beginning of the POM (where you also set the groupId and artifactId). Here is a version element example:

**<version>1.0-SNAPSHOT</version>**

Notice the -SNAPSHOT appended to the version number.

Depending on a snapshot version is also done by appending the -SNAPSHOT after the version number when configuring dependencies. Here is an example:

<dependency>

<groupId>com.jenkov</groupId>

<artifactId>java-web-crawler</artifactId>

**<version>1.0-SNAPSHOT</version>**

</dependency>

The -SNAPSHOT appended to the version number tells Maven that this is a snapshot version.

You can configure how often Maven shall download snapshot dependencies in the Maven Settings File.

* **Transitive Dependencies**

If your project depends on a dependency, say Dependency ABC, and Dependency ABC itself depends on another dependency, say Dependency XYZ, then your project has a *transitive dependency* on Dependency XYZ.

* **Exclude Dependency**

Sometimes the direct dependencies of your project may clash with the transitive dependencies of the direct dependencies. For instance, you may be using a JAX-RS implementation which internally uses an older version of the [**Jackson JSON Toolkit**](https://jenkov.com/java-json/jackson-objectmapper.html). However, your application may be using a newer version of the Jackson JSON Toolkit. How do you know which of the two versions will be used?

A solution is to specify for the JAX-RS dependency that its dependency on the older version of the Jackson JSON Toolkit should be excluded. This is also referred to as *dependency exclusion*.

You specify a dependency exclusion inside the declaration of the dependency which transitive dependency you want to exclude. Here is an example of declaring a Maven dependency exclusion:

<dependency>

<groupId>example.jaxrs</groupId>

<artifactId>JAX-RS-TOOLKIT</artifactId>

<version>1.0</version>

<scope>compile</scope>

**<exclusions>**

**<exclusion>**

**<groupId>com.fasterxml.jackson.core</groupId>**

**<artifactId>jackson-core</artifactId>**

**</exclusion>**

**</exclusions>**

</dependency>

With this dependency exclusion declaration in place, whatever version of the excluded dependency that the dependency containing the exclusion is using, will be ignored during Maven's compilation of the project.

**Maven Repositories**

Maven repositories are directories of packaged JAR files with extra meta data. The meta data are POM files describing the projects each packaged JAR file belongs to, including what external dependencies each packaged JAR has. It is this meta data that enables Maven to download dependencies of your dependencies recursively, until the whole tree of dependencies is downloaded and put into your local repository.

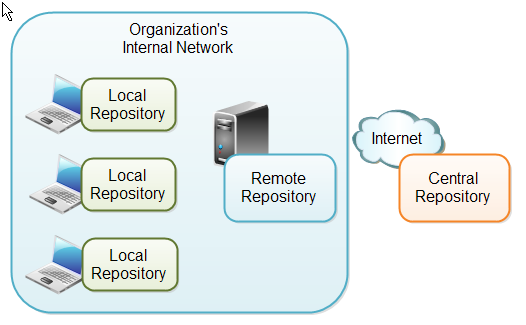
Maven repositories are covered in more detail in the [Maven Introduction to Repositories](http://maven.apache.org/guides/introduction/introduction-to-repositories.html), but here is a quick overview.

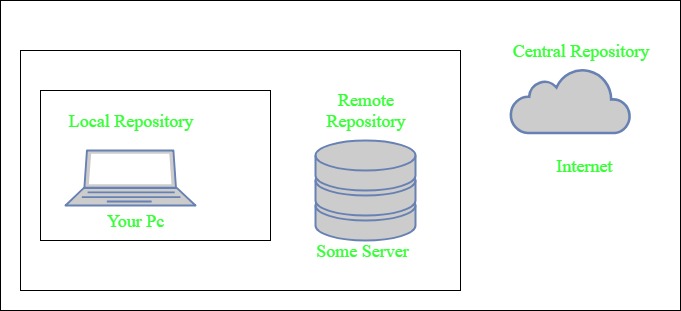
Maven has three types of repository:

* Local repository
* Central repository
* Remote repository

Maven searches these repositories for dependencies in the above sequence. First in the local repository, then in the central repository, and third in remote repositories if specified in the POM.

Here is a diagram illustrating the three repository types and their location:





**Local Repository:**

A local repository is a directory on the developer's computer. This repository will contain all the dependencies Maven downloads. The same Maven repository is typically used for several different projects. Thus Maven only needs to download the dependencies once, even if multiple projects depends on them (e.g. Junit).

Your own projects can also be built and installed in your local repository, using the mvn install command. That way your other projects can use the packaged JAR files of your own projects as external dependencies by specifying them as external dependencies inside their Maven POM files.

By default Maven puts your local repository inside your user home directory on your local computer. However, you can change the location of the local repository by setting the directory inside your Maven settings file. Your Maven settings file is also located in your user-home/.m2 directory and is called settings.xml. Here is how you specify another location for your local repository:

<settings>

<localRepository>

d:\data\java\products\maven\repository

</localRepository>

</settings>

**Central Repository**

The central Maven repository is a repository provided by the Maven community. By default, Maven looks in this central repository for any dependencies needed but not found in your local repository. Maven then downloads these dependencies into your local repository. You need no special configuration to access the central repository.

**Remote Repository**  
A remote repository is a repository on a web server from which Maven can download dependencies, just like the central repository. A remote repository can be located anywhere on the internet, or inside a local network.

A remote repository is often used for hosting projects internal to your organization, which are shared by multiple projects. For instance, a common security project might be used across multiple internal projects. This security project should not be accessible to the outside world, and should thus not be hosted in the public, central Maven repository. Instead, it can be hosted in an internal remote repository.

Dependencies found in a remote repository are also downloaded and put into your local repository by Maven.

You can configure a remote repository in the POM file. Put the following XML elements right after the <dependencies> element:

<repositories>

<repository>

<id>jenkov.code</id>

<url>http://maven.jenkov.com/maven2/lib</url>

</repository>

</repositories>

**Maven Build Life Cycles, Phases and Goals**

When Maven builds a software project it follows a build life cycle. The build life cycle is divided into build phases, and the build phases are divided into build goals. Maven build life cycles, build phases and goals are described in more detail in the [**Maven Introduction to Build Phases**](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html), but here I will give you a quick overview.

**Build Life Cycles**  
Maven has 3 built-in build life cycles. These are:

1. default
2. clean
3. site

Each of these build life cycles takes care of a different aspect of building a software project. Thus, each of these build life cycles are executed independently of each other. You can get Maven to execute more than one build life cycle, but they will be executed in sequence, separately from each other, as if you had executed two separate Maven commands.

The ***default*** life cycle handles everything related to compiling and packaging your project.

The ***clean*** life cycle handles everything related to removing temporary files from the output directory, including generated source files, compiled classes, previous JAR files etc.

The ***site*** life cycle handles everything related to generating documentation for your project. In fact, site can generate a complete website with documentation for your project.

**Build Phases**  
Each build life cycle is divided into a sequence of build phases, and the build phases are again subdivided into goals. Thus, the total build process is a sequence of build life cycle(s), build phases, and goals.

You can execute either a whole build life cycle like clean or site, a **build phase like install which is part of the default build life cycle**, or a build goal like dependency:copy-dependencies. Note: You cannot execute the default life cycle directly. You have to specify a build phase or goal inside the default life cycle.

When you execute a build phase, all build phases before that build phase in this standard phase sequence are executed. Thus, executing the install build phase really means executing all build phases before the install phase, and then execute the install phase after that.

The default life cycle is of most interest since that is what builds the code. Since you cannot execute the default life cycle directly, you need to execute a build phase or goal from the default life cycle. The default life cycle has an extensive sequence of build phases and goals, ,so I will not describe them all here. The most commonly used build phases are:

|  |  |
| --- | --- |
| Build Phase | Description |
| validate | Validates that the project is correct and all necessary information is available. This also makes sure the dependencies are downloaded. |
| compile | Compiles the source code of the project. |
| test | Runs the tests against the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed. |
| package | Packs the compiled code in its distributable format, such as a JAR. |
| install | Install the package into the local repository, for use as a dependency in other projects locally. |
| deploy | Copies the final package to the remote repository for sharing with other developers and projects. |

You execute one of these build phases by passing its name to the mvn command. Here is an example:

mvn package

This example executes the package build phase, and thus also all build phases before it in Maven's predefined build phase sequence.

If the standard Maven build phases and goals are not enough to build your project, you can create [**Maven plugins**](https://jenkov.com/tutorials/maven/maven-tutorial.html#maven-plugins) to add the extra build functionality you need.

Generally, when we run any of the above commands, we add the **mvn clean** step so that the target folder generated from the previous build is removed before running a newer build. This is how the command would look on integrating the *clean* step with *install* phase: **mvn clean install.**

Consider a scenario where we do not want to run the tests while packaging or installing the Java project. In this case, we use **-DskipTests** along with the actual command. If we need to run the *install* step by skipping the tests associated with the project, the command would be: **mvn install -DskipTests**

**Build Goals**  
Build goals are the finest steps in the Maven build process. ***A goal can be bound to one or more build phases, or to none at all. If a goal is not bound to any build phase, you can only execute it by passing the goals name to the mvn command***. If a goal is bound to multiple build phases, that goal will get executed during each of the build phases it is bound to.

**Maven Build Profiles**

Maven build profiles enable you to build your project using different configurations. Instead of creating two separate POM files, you can just specify a profile with the different build configuration, and build your project with this build profile when needed.

You can read the full story about build profiles in the Maven POM reference under [**Profiles**](http://maven.apache.org/pom.html#Profiles). Here I will give you a quick overview though.

Maven build profiles are specified inside the POM file, inside the profiles element. Each build profile is nested inside a profile element. Here is an example:

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jenkov.crawler</groupId>

<artifactId>java-web-crawler</artifactId>

<version>1.0.0</version>

<profiles>

<profile>

<id>test</id>

<activation>...</activation>

<build>...</build>

<modules>...</modules>

<repositories>...</repositories>

<pluginRepositories>...</pluginRepositories>

<dependencies>...</dependencies>

<reporting>...</reporting>

<dependencyManagement>...</dependencyManagement>

<distributionManagement>...</distributionManagement>

</profile>

</profiles>

</project>

A build profile describes what changes should be made to the POM file when executing under that build profile. This could be changing the applications configuration file to use etc. The elements inside the profile element will override the values of the elements with the same name further up in the POM.

Inside the profile element you can see a activation element. This element describes the condition that triggers this build profile to be used. One way to choose what profile is being executed is in the settings.xml file. There you can set the active profile. Another way is to add -P profile-name to the Maven command line. See the profile documentation for more information.