**The Project Object Model**

**POM stands for Project Object Model**

The pom.xml file contains information of project and configuration information for the maven to build the project such as dependencies, build directory, source directory, test source directory, plugin, goals etc.

While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal. Some of the configuration that can be specified in the POM are following −

* project dependencies
* plugins
* goals
* build profiles
* project version
* developers
* mailing list

**The POM contains four categories of description and configuration:**



* **General project information**
* This includes a project’s name, the URL for a project, the sponsoring organization, and a list of developers and contributors along with the license for a project.
* **Build settings**
* In this section, we customize the behaviour of the default Maven build. We can change the location of source and tests, we can add new plugins, we can attach plugin goals to the lifecycle, and we can customize the site generation parameters.
* **Build environment**
* The build environment consists of profiles that can be activated for use in different environments. For example, during development you may want to deploy to a development server, whereas in production you want to deploy to a production server. The build environment customizes the build settings for specific environments and is often supplemented by a custom settings.xml in ~/.m2.
* **POM relationships**
* A project rarely stands alone; it depends on other projects, inherits POM settings from parent projects, defines its own coordinates, and may include submodules.

**The Super POM**

****

* Basic configuration inherited by any maven project in the world.
* All Maven project POMs extend the Super POM, which defines a set of defaults shared by all projects. This Super POM is a part of the Maven installation.
* The Super POM defines some standard configuration variables that are inherited by all projects.
* The default Super POM defines a single remote Maven repository with an ID of central. This is the Central Repository that all Maven clients are configured to read from by default. This setting can be overridden by a custom settings.xml file. Note that the default Super POM has disabled snapshot artifacts on the Central Repository. If you need to use a snapshot repository, you will need to customize repository settings in your pom.xml or in your settings.xml.
* The Central Repository also contains Maven plugins. The default plugin repository is the central Maven repository. Snapshots are disabled, and the update policy is set to “never,” which means that Maven will never automatically update a plugin if a new version is released.
* The build element sets the default values for directories in the Maven Standard Directory layout

**The Simplest POM**

All Maven POMs inherit defaults from the Super POM. If you are just writing a simple project that produces a JAR from some source in src/main/java, want to run your JUnit tests in src/test/java, and want to build a project site using mvn site, you don’t have to customize anything. All you would need, in this case, is the simplest possible POM shown in The Simplest POM.

This POM defines a groupId, artifactId, and version: the three required coordinates for

every project.

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>org.sonatype.mavenbook.ch08</groupId>

<artifactId>simplest-project</artifactId>

<version>1</version>

</project>

**Effective POM**

$ mvn help:effective-pom

Executing the effective-pom goal should print out an XML document capturing the merge between the Super POM and the POM from The Simplest POM.

POM syntax

**Project Versions**

<major version>.<minor version>.<incremental version>-<qualifier>

A project’s version number is used to group and order releases. Maven versions contain the following

parts: major version, minor version, incremental version, and qualifier.

The version "1.3.5" has a major version of 1, a minor version of 3, and an incremental version of 5. The version "5" has a major version of 5 and no minor or incremental version. The qualifier exists

to capture milestone builds: alpha and beta releases.

For example, the version "1.3-beta-01" has a major version of 1, a minor version of 3, no incremental version and a qualifier of "beta-01".

**Version Build Numbers**

One gotcha for release version numbers is the ordering of the qualifiers. Take the version release numbers “1.2.3-alpha-2” and “1.2.3-alpha-10,” where the “alpha-2” build corresponds to the 2nd alpha build, and the “alpha-10” build corresponds to the 10th alpha build. Even though “alpha-10” should be considered

more recent than “alpha-2,” Maven is going to sort “alpha-10” before “alpha-2” due to a known issue in the way Maven handles version numbers.

If you use "alpha-02" and "alpha-10" this problem will go away

**SNAPSHOT Versions**

Maven versions can contain a string literal to signify that a project is currently under active development

Maven versions can contain a string literal to signify that a project is currently under active development.

If a version contains the string “-SNAPSHOT,” then Maven will expand this token to a date and time

value converted to UTC (Coordinated Universal Time) when you install or release this component. For

example, if your project has a version of “1.0-SNAPSHOT” and you deploy this project’s artifacts to a

Maven repository, Maven would expand this version to “1.0-20080207-230803-1” if you were to deploy

a release at 11:08 PM on February 7th, 2008 UTC. In other words, when you deploy a snapshot, you are

not making a release of a software component; you are releasing a snapshot of a component at a specific

time.

Why would you use this? SNAPSHOT versions are used for projects under active development.

If your project depends on a software component that is under active development, you can depend on a SNAPSHOT release, and Maven will periodically attempt to download the latest snapshot from a repository

when you run a build. Similarly, if the next release of your system is going to have a version "1.4", your

project would have a version "1.4-SNAPSHOT" until it was formally released.

As a default setting, Maven will not check for SNAPSHOT releases on remote repositories. To depend

on SNAPSHOT releases, users must explicitly enable the ability to download snapshots using a reposi

tory or pluginRepository element in the POM.

When releasing a project, you should resolve all dependencies on SNAPSHOT versions to dependencies

on released versions. If a project depends on a SNAPSHOT, it is not stable as the dependencies may

change over time. Artifacts published to non-snapshot Maven repositories such as http://repo1.maven.org/-

maven2 cannot depend on SNAPSHOT versions, as Maven’s Super POM has snapshot’s disabled from

the Central repository. SNAPSHOT versions are for development only.

**Project Dependencies**

Maven can manage both internal and external dependencies.

An external dependency for a Java project might be a library such as Plexus, the Spring Framework, or Log4J. An internal dependency is illustrated by a web application project depending on another project that contains service classes, model objects, or persistence logic.

</dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.4</version>

<scope>provided</scope>

</dependency>

</dependencies>

**Dependency Scope**

compile

compile is the default scope; all dependencies are compile-scoped if a scope is not supplied.

compile dependencies are available in all classpaths, and they are packaged.

provided

provided dependencies are used when you expect the JDK or a container to provide them. For

example, if you were developing a web application, you would need the Servlet API available on

the compile classpath to compile a servlet, but you wouldn’t want to include the Servlet API in the

packaged WAR; the Servlet API JAR is supplied by your application server or servlet container.

provided dependencies are available on the compilation classpath (not runtime). They are not

transitive, nor are they packaged.

runtime

runtime dependencies are required to execute and test the system, but they are not required for

compilation. For example, you may need a JDBC API JAR at compile time and the JDBC driver

implementation only at runtime.

test

test-scoped dependencies are not required during the normal operation of an application, and

they are available only during test compilation and execution phases.

system

The system scope is similar to provided except that you have to provide an explicit path to the

JAR on the local file system. This is intended to allow compilation against native objects that may

be part of the system libraries. The artifact is assumed to always be available and is not looked up

in a repository. If you declare the scope to be system, you must also provide the systemPath

element. Note that this scope is not recommended (you should always try to reference dependencies

in a public or custom Maven repository).

**Optional Dependencies**

<dependency>

<groupId>swarmcache</groupId>

<artifactId>swarmcache</artifactId>

<version>1.0RC2</version>

<optional>true</optional>

</dependency>

**Dependency Version Ranges**

(, )

Exclusive quantifiers

[, ]

Inclusive quantifiers

Specifying a Dependency Range: JUnit 3.8 - JUnit 4.0

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>[3.8,4.0)</version>

<scope>test</scope>

</dependency>

**Specifying a Dependency Range: JUnit( 3.8.1**

If you want to depend on any version of JUnit no higher than 3.8.1, you would specify only an upper

inclusive boundary.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>[,3.8.1]</version>

<scope>test</scope>

</dependency>

For example, "[4.0,)" means any version greater than or equal to 4.0. "(,2.0)" is any version less than 2.0. "[1.2]" means only version 1.2, and nothing else.

**Transitive Dependencies**