Maven Guide

Maven

Maven is a powerful build tool for Java software projects. Actually, you can build software projects using other languages too, but Maven is developed in Java, and is thus historically used more for Java projects.

The purpose of this tutorial is to make you understand how Maven works. Therefore this tutorial focuses on the core concepts of Maven. Once you understand the core concepts, it is much easier to lookup the fine detail in the Maven documentation, or search for it on the internet.

Actually, the Maven developers claim that Maven is more than just a build tool. You can read what they believe it is, in their document Philosophy of Maven. But for now, just think of it as a build tool. You will find out what Maven really is, once you understand it and start using it.

The Maven website is located here:
http://maven.apache.org
From this website you can download the latest version of Maven and follow the project in general.

What is a Build Tool?

A build tool is a tool that automates everything related to building the software project. Building a software project typically includes one or more of these activities:

- Generating source code (if auto-generated code is used in the project).
- Generating documentation from the source code.
- Compiling source code.
- Packaging compiled code into JAR files or ZIP files.
- Installing the packaged code on a server, in a repository or somewhere else.

Any given software project may have more activities than these needed to build the finished software. Such activities can normally be plugged into a build tool, so these activities can be automated too.

The advantage of automating the build process is that you minimize the risk of humans making errors while building the software manually. Additionally, an automated build tool is typically faster than a human performing the same steps manually.

Maven vs. Ant

Ant is another popular build tool by Apache. If you are used to Ant and you are trying to learn Maven, you will notice a difference in the approach of the two projects.
Ant uses an imperative approach, meaning you specify in the Ant build file what actions Ant should take. You can specify low level actions like copying files, compiling code etc. You specify the actions, and you also specify the sequence in which they are carried out. Ant has no default directory layout.
Maven uses a more declarative approach, meaning that you specify in the Maven POM file *what* to build, but now *how* to build it. The POM file describes your project resources - not how to build it. Contrarily, an Ant file describes how to build your project. In Maven, how to build your project is predefined in the Maven Build Life Cycles, Phases and Goals.

Installing Maven

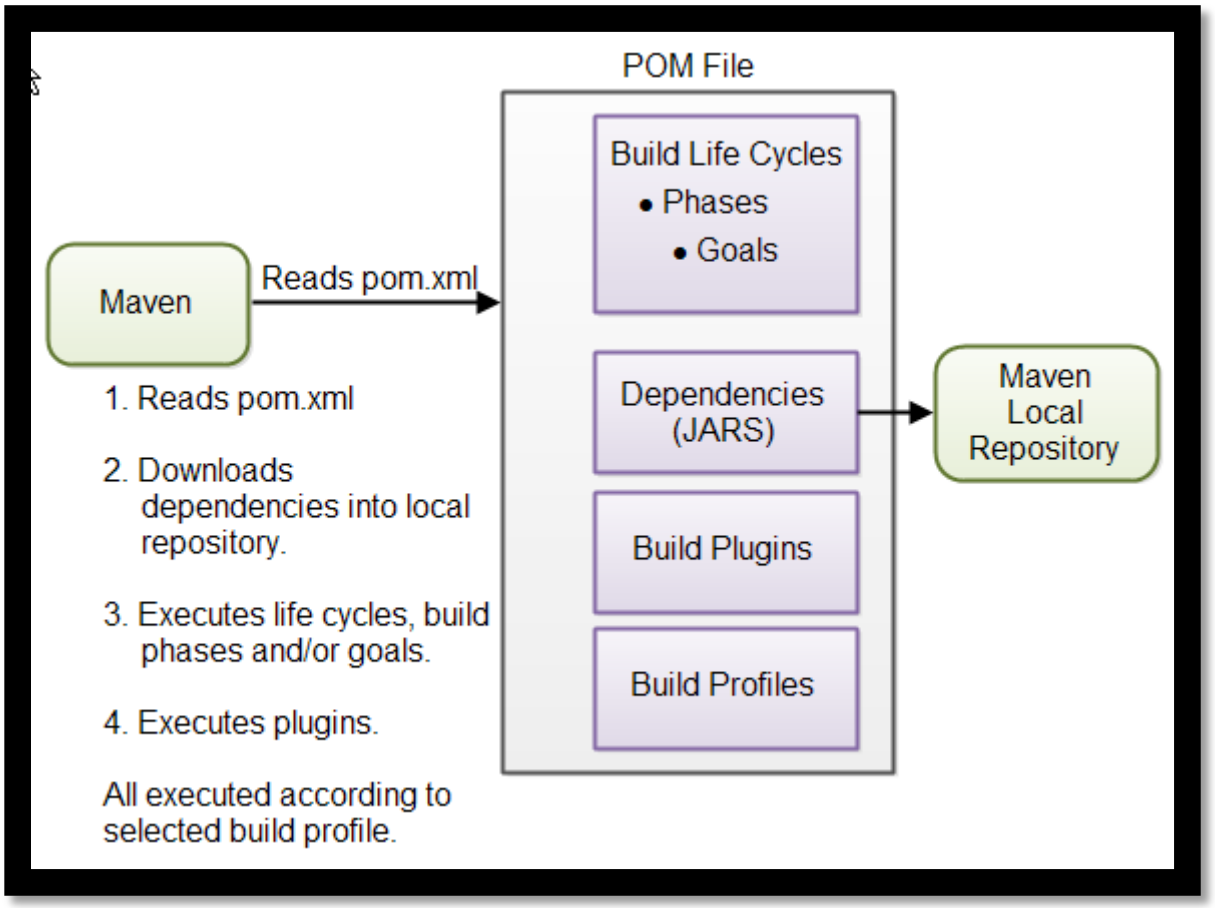
To install Maven on your own system (computer), go to the Maven download page and follow the instructions there. In summary, what you need to do is:

1. Download and unzip Maven.
2. Set the M2_HOME environment variable to point to the directory you unzipped Maven to.
3. Set the M2 environment variable to point to M2_HOME/bin (%M2_HOME%\bin on Windows, $M2_HOME/bin on unix).
4. Add M2 to the PATH environment variable (%M2% on Windows, $M2 on unix).
5. Open a command prompt and type 'mvn' (without quotes) and press enter.

   After typing in the 'mvn' command you should be able to see a Maven error written to the command prompt. Don't worry about the error. It is expected because you haven't yet any POM file to pass to Maven. But the fact that you get a Maven error means that Maven is now installed.
   Note: Maven uses Java when executing, so you need Java installed to. Maven needs a Java version 1.5 or later.

Maven Overview - Core Concepts

Maven is centered around the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.
Here is a diagram illustrating how Maven uses the POM file, and what the POM file primarily contains:



Overview of Maven core concepts.

These concepts are explained briefly below to give you an overview, and then in more detail in their own sections later in this tutorial.

POM Files

When you execute a Maven command you give Maven a POM file to execute the commands on. Maven will then execute the command on the resources described in the POM.

Build Life Cycles, Phases and Goals

The build process in Maven is split up into build life cycles, phases and goals. A build life cycle consists of a sequence of build phases, and each build phase consists of a sequence of goals. When you run Maven you pass a command to Maven. This command is the name of a build life cycle, phase or goal. If a life cycle is requested executed, all build phases in that life cycle are executed. If a build phase is requested executed, all build phases before it in the pre-defined sequence of build phases are executed too.

Dependencies and Repositories

One of the first goals Maven executes is to check the dependencies needed by your project. Dependencies are external JAR files (Java libraries) that your project uses. If the dependencies are not found in the local Maven repository, Maven downloads them from a central Maven repository and puts them in your local repository. The local repository is just a directory on your computer's hard disk. You can specify where the local repository should be located if you want to (I do). You can also specify which remote repository to use for downloading dependencies. All this will be explained in more detail later in this tutorial.

Build Plugins

Build plugins are used to insert extra goals into a build phase. If you need to perform a set of actions for your project which are not covered by the standard Maven build phases and goals, you can add a plugin to the POM file. Maven has some standard plugins you can use, and you can also implement your own in Java if you need to.

Build Profiles

Build profiles are used if you need to build your project in different ways. For instance, you may need to build your project for your local computer, for development and test. And you may need to build it for deployment on your production environment. These two builds may be different. To enable different builds you can add different build profiles to your POM files. When executing Maven you can tell which build profile to use.

Maven vs. Ant

Ant is another popular build tool by Apache. If you are used to Ant and you are trying to learn Maven, you will notice a difference in the approach of the two projects.

Ant uses an imperative approach, meaning you specify in the Ant build file what actions Ant should take. You can specify low level actions like copying files, compiling code etc. You specify the actions, and you also specify the sequence in which they are carried out. Ant has no default directory layout.

Maven uses a more declarative approach, meaning that you specify in the Maven POM file *what* to build, but now *how* to build it. The POM file describes your project resources - not how to build it. Contrarily, an Ant file describes how to build your project. In Maven, how to build your project is predefined in the Maven Build Life Cycles, Phases and Goals.

Maven POM Files

The project object model

The heart of a Maven 2 project is the project object model (or POM for short). It contains a detailed description of your project, including information about versioning and configuration management, dependencies, application and testing resources, team members and structure, and much more. The POM takes the form of an XML file (*pom.xml*), which is placed in your project home directory. A simple pom.xml file is shown here:

```
<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>
```

Let's analyse this a bit. First off, all pom files consist of a "project" xml element. This encompases the entire xml document. The model version element is always the same and is required for maven projects. Here's some simple descriptions of the other elements.

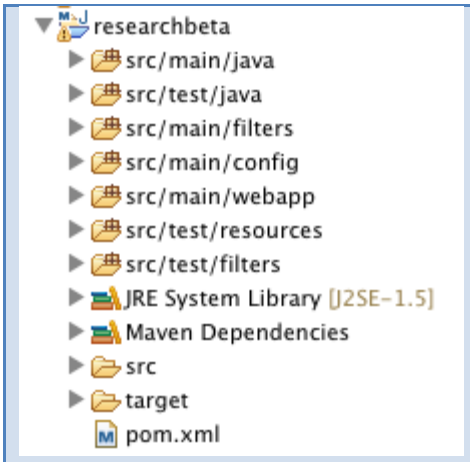
- **groupId** - This corresponds to the inverted domain-name of the project. Although you could use any arbitrary value, it's recommended to use your company domain name here
- **artifactId** - This is a unique id for the project (within the group). This name will be used when creating jar/war/ear files.
- **version** - This represents the current version of the project. More details later
- **name** - This is the human-readable name of the project.
- **url** -This is the url for the project website.
- **packaging** - This defines the "style" of project your building (e.g. ear, jar, war, etc.). For more information on packaging, check out the respective plugins (maven-jar-plugin, maven-war-plugin, etc.)
- **dependencies** - Specifies the dependencies of the project. This will be materialized from any defined maven repositories. More details later
- **repositories** - Specifies alternative locations for maven to look when materializing dependencies.
- **pluginRepositories** - Specifies alternative location for maven to look when materializing build plugins
- **build** - Specifies configuration on *how* to build the project
- **reports** - Specifies configuration on *what* reports to generate for the project.

Default Directory Layout

Much of Maven's power comes from the standard practices it encourages. A developer who has previously worked on a Maven project will immediately feel familiar with the structure and organization of a new one. Time need not be wasted reinventing directory structures, conventions, and customized Ant build scripts for each project. Although you can override any particular directory location for your own specific ends, you really should respect the standard Maven directory structure as much as possible, for several reasons

- It makes your POM file smaller and simpler
- It makes the project easier to understand and makes life easier for the poor guy who must maintain the project when you leave
- It makes it easier to integrate plug-ins

The standard Maven directory structure is illustrated below. In the project home directory goes the POM (pom.xml) and two subdirectories: src for all source code and target for generated artifacts. The src directory has a number of subdirectories, each of which has a clearly defined purpose:

	src/main/java: Your Java source code goes here (strangely enough!) src/main/resources: Other resources your application needs src/main/filters: Resource filters, in the form of properties files, which may be used to define variables only known at runtime src/main/config: Configuration file src/main/webapp: The Web application directory for a WAR project src/test/java: Unit tests src/test/resources: Resources to be used for unit tests, but will not be deployed src/test/filters: Resources filters to be used for unit tests, but will not be deployed

A Maven POM file (Project Object Model) is an XML file that describe 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.
Throughout the rest of this section I will describe the most important parts of the POM file. For a full reference of the POM file, see the Maven POM Reference.
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</rtifactId>
   <version>1.0.0</version>
</project>
```

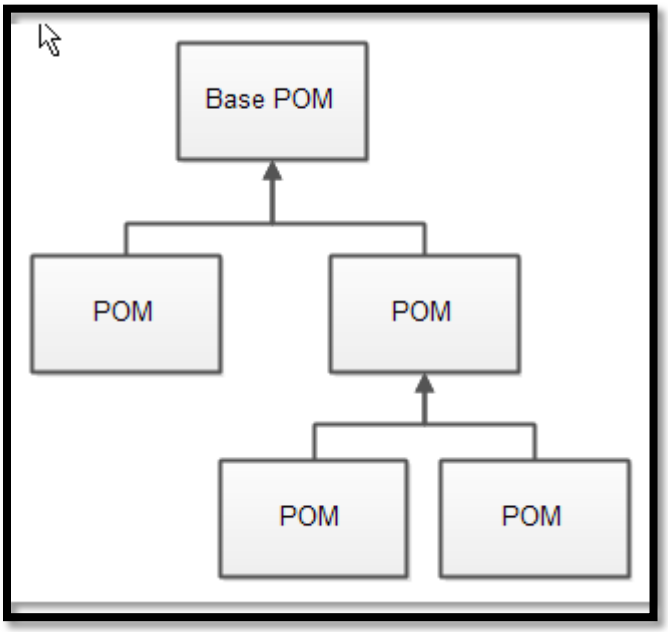
The modelVersion element sets what version of the POM model you are using. Use the one matching the Maven version you are using. Version 4.0.0 matches Maven version 2 and 3.
The groupId element is a unique ID for an organization, or a project (an open source project, for instance). Most often you will use a group ID which 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.
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 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
If your project uses the Maven directory structure, and your project has no external dependencies, then the above minimal POM file is all you need to build your project.
If your project does not follow the standard directory structure, has external dependencies, or need special actions during building, you will need to add more elements to the POM file. These elements are listed in the Maven POM reference (see link above).
In general you can specify a lot of things in the POM which gives Maven more details about how to build your projects. See the Maven POM reference for more information about what can be specified.

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:



Super POM and POM inheritance.

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.
POM inheritance is also covered in more detail in the Maven POM reference.

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 effetive 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 filse 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.
You can read more about the Maven settings files in the Maven Settings Reference.

Running Maven

When you have installed Maven and have created a POM file 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 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.

You can also execute a Maven goal (a subpart of a build phase) by passing the build phase and goal name concatenated with a : in between, as parameter to the Maven command. Here is an example:

```
mvn dependency:copy-dependencies
```

This command executes the copy-dependencies goal of the dependency 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. You can see the full directory layout in the Introduction to the Maven Standard Directory Layout.

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 contains 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. This could be property files used for 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 which 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 (neiterh 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.

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 download and put into your local repository.

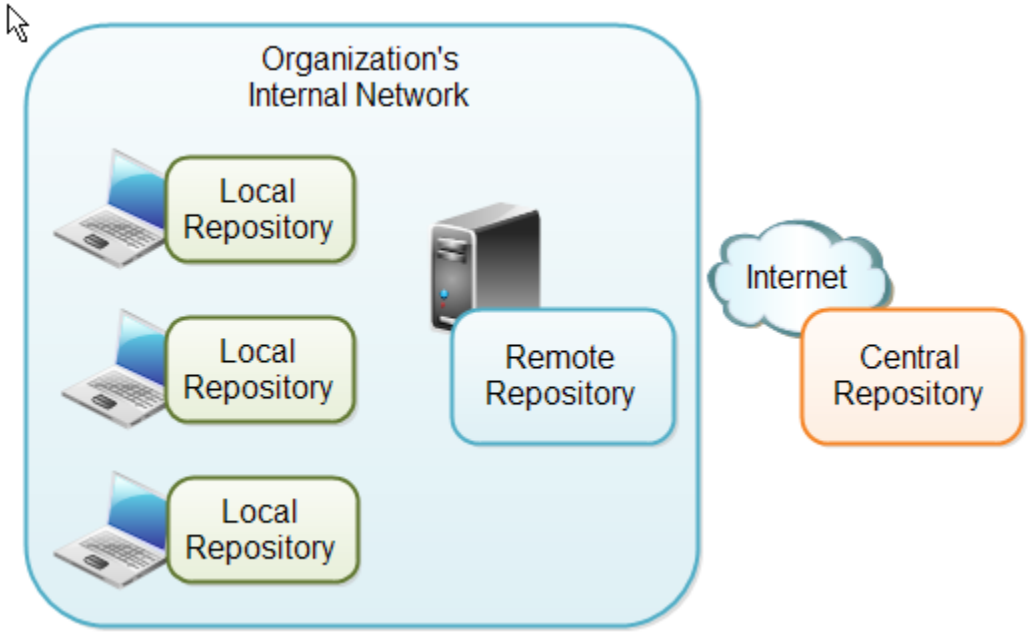
Maven repositories are covered in more detail in the Maven Introduction to Repositories, but here is a quick overview.

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:



Maven Repository Types and 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, but here I will give you a quick overview.

Build Life Cycles

Maven has 3 built-in build life cycles. These are:

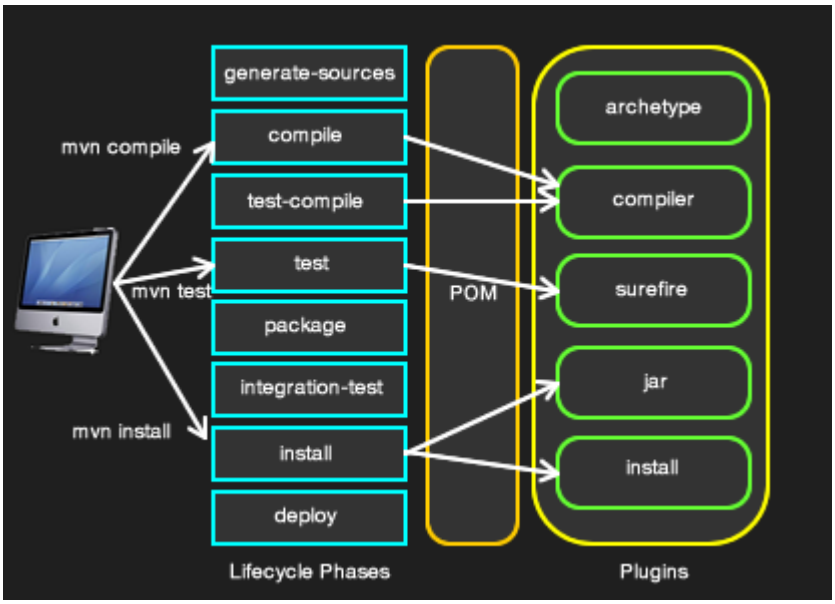
1. **default**
2. **clean**
3. **site**

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.

Maven Lifecycle

Project lifecycles are central to Maven. Most developers are familiar with the notion of build phases such as compile, test, and deploy. Ant has targets with names like those. In Maven, corresponding plug-ins are called directly. To compile Java source code, for instance, the java plug-in is used: Plugin goals can be attached to a lifecycle phase. As Maven moves through the phases in a lifecycle, it will execute the goals attached to each particular phase. Each phase may have zero or more goals bound to it. In the previous section, when you ran mvn install, you might have noticed that more than one goal was executed. Examine the output after running mvn install and take note of the various goals that are executed. When this simple example reached the package phase, it executed the jar goal in the Jar plugin. Since our simple Quickstart project has (by default) a jar packaging type, the jar:jar goal is bound to the package phase. In Maven, this notion is standardized into a set of well-known and well-defined lifecycle phases. Instead of invoking plug-ins, the Maven 2 developer invokes a lifecycle



phase: **$mvn compile**.

Some of the more useful Maven lifecycle phases are the following:

generate-sources: Generates any extra source code needed for the application, which is generally accomplished using the appropriate plug-ins

compile: Compiles the project source code

test-compile: Compiles the project unit tests

test: Runs the unit tests (typically using JUnit) in the src/test directory

package: Packages the compiled code in its distributable format (JAR, WAR, etc.)

integration-test: Processes and deploys the package if necessary into an environment where integration tests can be run

install: Installs the package into the local repository for use as a dependency in other projects on your local machine

- **deploy:** Done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and project.

These phases illustrate the benefits of the recommended practices encouraged by Maven 2: once a developer is familiar with the main Maven lifecycle phases, he should feel at ease with the lifecycle phases of any Maven project. The lifecycle phase invokes the plug-ins it needs to do the job. Invoking a lifecycle phase automatically invokes any previous lifecycle phases as well. Since the lifecycle phases are limited in number, easy to understand, and well organized, becoming familiar with the lifecycle of a new Maven project is easy.

Transitive dependencies

One of the highlights of Maven is transitive dependency management. If you have ever used a tool like urpmi on a Linux box, you'll know what transitive dependencies are. With Maven 1, you have to declare each and every JAR that will be needed, directly or indirectly, by your application. For example, can you list the JARs needed by a Hibernate application? With Maven new versions, you don't have to. You just tell Maven which libraries you need, and Maven will take care of the libraries that your libraries need (and so on). Suppose you want to use Google Guice in your project. You would simply add a new dependency to the dependencies section in pom.xml, as follows:

```
<dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>3.0</version>
</dependency>
```

And that's it! You don't have to hunt around to know in which other JARs (and in which versions) you need to run Guice 3.0.

Dependency scopes

In a real-world enterprise application, you may not need to include all the dependencies in the deployed application. Some JARs are needed only for unit testing, while others will be provided at runtime by the application server. Using a technique called *dependency scoping*, Maven 2 lets you use certain JARs only when you really need them and excludes them from the classpath when you don't. Maven provides four dependency scope:
compile**:** A compile-scope dependency is available in all phases. This is the default value.
provided**:** A provided dependency is used to compile the application, but will not be deployed. You would use this scope when you expect the JDK or application server to provide the JAR. The servlet APIs are a good example.
runtime**:** Runtime-scope dependencies are not needed for compilation, only for execution, such as JDBC (Java Database Connectivity) drivers.
test**:** Test-scope dependencies are needed only to compile and run tests (JUnit, for example).

Site Generation and Reporting

Another important feature of Maven is its ability to generate documentation and reports. In your simple project's directory, execute the following command:

$ mvn site

This will execute the site lifecycle phase. Unlike the default build lifecycle that manages generation of code, manipulation of resources, compilation, packaging, etc., this lifecycle is concerned solely with processing site content under the *src/site* directories and generating reports. After this command executes, you should see a project web site in the *target/site* directory. Load *target/site/index.html* and you should see a basic shell of a project site. This shell contains some reports under "Project Reports" in the lefthand navigation menu, and it also contains information about the project, the dependencies, and developers associated with it under "Project Information." The simple project's web site is mostly empty, since the POM contains very little information about itself beyond its Maven coordinates, a name, a URL, and a single test dependency.

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 to add the extra build functionality you need.

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

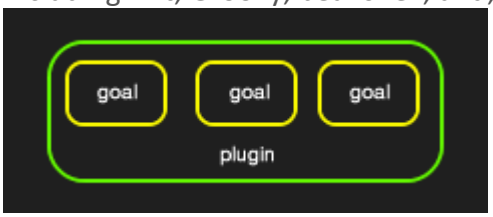
Maven build profiles enable you to build your project using different configurations. Instead of creating two separate POM files, you can just specify a profile with the different build configuration, and build your project with this build profile when needed.
You can read the full story about build profiles in the Maven POM reference under Profiles. 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.

Maven Plugins and Goals

A Maven Plugin is a collection of one or more goals. Examples of Maven plugins can be simple core plugins like the Jar plugin, which contains goals for creating JAR files, Compiler plugin, which contains goals for compiling source code and unit tests, or the Surefire plugin, which contains goals for executing unit tests and generating reports. Other, more specialized Maven plugins include plugins like the Hibernate3 plugin for integration with the popular persistence library Hibernate, the JRuby plugin which allows you to execute ruby as part of a Maven build or to write Maven plugins in Ruby. Maven also provides for the ability to define custom plugins. A custom plugin can be written in Java, or a plugin can be written in any number of languages including Ant, Groovy, beanshell, and, as previously mentioned, Ruby.



A goal is a specific task that may be executed as a standalone goal or along with other goals as part of a larger build. A goal is a “unit of work” in Maven. Examples of goals include the compile goal in the Compiler plugin, which compiles all of the source code for a project, or the test goal of the Surefire plugin, which can execute unit tests. Goals are configured via configuration properties that can be used to customize behavior. For example, the compile goal of the Compiler plugin defines a set of configuration also passed the package parameter to the generate goal asorg.sonatype.mavenbook. If we had omitted the packageName parameter, the package name would have defaulted to org.sonatype.mavenbook.simple.

Settings .xml

settings.xml

IMPORTANT SECTION IN SETTINGS XML

```
<!-- server
    | Specifies the authentication information to use when connecting to a particular server, identified by
    | a unique name within the system (referred to by the 'id' attribute below).
    |
    | NOTE: You should either specify username/password OR privateKey/passphrase, since these pairings are
    |       used together.
    |
   <server>
    <id>deploymentRepo</id>
    <username>repouser</username>
    <password>repopwd</password>
   </server>
   -->

   <!-- Another sample, using keys to authenticate.
   <server>
    <id>siteServer</id>
    <privateKey>/path/to/private/key</privateKey>
    <passphrase>optional; leave empty if not used.</passphrase>
   </server>
   -->
 </servers>
```

In server section we need to give Nexus or remote repository credential to deploy artifacts when we use mvn Deploy command

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

configure a proxy, have a look at the section about Proxies.

•id: The unique identifier for this proxy. This is used to differentiate between proxy elements.

•active: true if this proxy is active. This is useful for declaring a set of proxies, but only one may be active at a time.

•protocol, host, port: The protocol://host:port of the proxy, seperated into discrete elements.

•username, password: These elements appear as a pair denoting the login and password required to authenticate to this proxy server.

•nonProxyHosts: This is a list of hosts which should not be proxied. The delimiter of the list is the expected type of the proxy server; the example above is pipe delimited - comma delimited is also common

Settings.xml Details

Half of the top-level settings elements are simple values, representing a range of values which configure the core behavior of Maven:

Simple top-level elements in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
   <localRepository>${user.dir}/.m2/repository</localRepository>
   <interactiveMode>true</interactiveMode>
   <usePluginRegistry>false</usePluginRegistry>
   <offline>false</offline>
   <pluginGroups>
      <pluginGroup>org.codehaus.mojo</pluginGroup>
   </pluginGroups>
   ...
</settings>
```

The simple top-level elements are:

localRepository

This value is the path of this build system's local repository. The default value is ${user.dir}/.m2/repository.

interactiveMode

true if Maven should attempt to interact with the user for input, false if not. Defaults to true.

usePluginRegistry

true if Maven should use the ${user.dir}/.m2/plugin-registry.xml file to manage plugin versions, defaults to false.

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 contains org.apache.maven.plugins by default.

Servers

The distributionManagement element of the POM defines the repositories for deployment. However, certain settings such as security credentials should not be distributed along with the pom.xml. This type of information should exist on the build server in the settings.xml.

Server configuration in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                    http://maven.apache.org/xsd/settings-1.0.0.xsd">    ...
   <servers>   <server>
         <id>server001</id>
         <username>my_login</username>
         <password>my_password</password>
         <privateKey>${user.home}/.ssh/id_dsa</privateKey>
         <passphrase>some_passphrase</passphrase>
         <filePermissions>664</filePermissions>
         <directoryPermissions>775</directoryPermissions>
         <configuration></configuration>
      </server>
   </servers>
   ...
</settings>
```

The elements under server are:

id

This is the id of the server (not of the user to login as) that matches the distributionManagement repository element's id.

username, password

These elements appear as a pair denoting the login and password required to authenticate to this server.

privateKey, passphrase

Like the previous two elements, this pair specifies a path to a private key (default is ${user.home}/.ssh/id_dsa) and a passphrase, if required. The passphrase and password elements may be externalized in the future, but for now they must be set plain-text in the settings.xml file.

filePermissions, directoryPermissions

When a repository file or directory is created on deployment, these are the permissions to use. The legal values of each is a three digit number corresponding to *nix file permissions, i.e. 664, or 775.

Mirrors

Mirror configuration in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                              http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <mirrors>
    <mirror>
      <id>planetmirror.com</id>
      <name>PlanetMirror Australia</name>
      <url>http://downloads.planetmirror.com/pub/maven2</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

id, name

The unique identifier of this mirror. The id is used to differentiate between mirror elements.

url

The base URL of this mirror. The build system will use prepend this URL to connect to a repository rather than the default server URL.

mirrorOf

The id of the server that this is a mirror of. For example, to point to a mirror of the Maven central server (http://repo1.maven.org/maven2), set this element to central. This must not match the mirror id.

Proxies

Proxy configuration in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                              http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

id
The unique identifier for this proxy. This is used to differentiate between proxy elements.
active
true if this proxy is active. This is useful for declaring a set of proxies, but only one may be active at a time.
protocol, host, port
The protocol://host:port of the proxy, separated into discrete elements.
username, password
These elements appear as a pair denoting the login and password required to authenticate to this proxy server.
nonProxyHosts
This is a list of hosts which should not be proxied. The delimiter of the list is the expected type of the proxy server; the example above is pipe delimited - comma delimited is also common

.

The profile element in the settings.xml is a truncated version of the pom.xml profile element. It consists of the activation, repositories, pluginRepositories and properties elements. The profile elements only include these four elements because they concern themselves with the build system as a whole (which is the role of the settings.xml file), not about individual project object model settings.

If a profile is active from settings, its values will override any equivalent profiles which matching identifiers in a POM or profiles.xml file.

. Activation

Activations are the key of a profile. Like the POM's profiles, the power of a profile comes from its ability to modify some values only under certain circumstances; those circumstances are specified via an activation element.

Defining Activation Parameters in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                  http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.3</value>
        </property>
        <file>
          <exists>${basedir}/file2.properties</exists>
          <missing>${basedir}/file1.properties</missing>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

Activation occurs when all specified criteria have been met, though not all are required at once.

jdk

activation has a built in, Java-centric check in the jdk element. This will activate if the test is run under a jdk version number that matches the prefix given. In the above example, 1.5.0_06 will match.

os

The os element can define some operating system specific properties shown above.

property

The profile will activate if Maven detects a property (a value which can be dereferenced within the POM by ${name}) of the corresponding name=value pair.

file

Finally, a given filename may activate the profile by the existence of a file, or if it is missing.

The activation element is not the only way that a profile may be activated. The settings.xml file's activeProfile element may contain the profile's id. They may also be activated explicitly through the command line via a comma separated list after the P flag (e.g. -P test).

To see which profile will activate in a certain build, use the maven-help-plugin.

mvn help:active-profiles

Properties

Maven properties are value placeholder, like properties in Ant. Their values are accessible anywhere within a POM by using the notation ${X}, where X is the property. They come in five different styles, all accessible from the settings.xml file:

+env.+X

Prefixing a variable with env. will return the shell's environment variable. For example, ${env.PATH} contains the $path environment variable. (%PATH% in Windows.)

+project.+x

A dot-notated (.) path in the POM will contain the corresponding elements value.

+settings.+x

A dot-notated (.) path in the settings.xml will contain the corresponding elements value.

Java system properties

All properties accessible via java.lang.System.getProperties() are available as POM properties, such as ${java.home}.

x

Set within a properties element or an external file, the value may be used as ${someVar}.

Setting the ${user.install} property in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                  http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <properties>
        <user.install>${user.dir}/our-project</user.install>
      </properties>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

The property ${user.install} is accessible from a POM if this profile is active.

Repositories

Repositories are remote collections of projects from which Maven uses to populate the local repository of the build system. It is from this local repository that Maven calls it plugins and dependencies. Different remote repositories may contain different projects, and under the active profile they may be searched for a matching release or snapshot artifact.

Repository Configuration in settings.xml.

releases, snapshots

These are the policies for each type of artifact, Release or snapshot. With these two sets, a POM has the power to alter the policies for each type independent of the other within a single repository. For example, one may decide to enable only snapshot downloads, possibly for development purposes.

enabled

true or false for whether this repository is enabled for the respective type (releases or snapshots).

updatePolicy

This element specifies how often updates should attempt to occur. Maven will compare the local POMs timestamp to the remote. The choices are: always, daily (default), interval:X (where X is an integer in minutes) or never.

checksumPolicy

When Maven deploys files to the repository, it also deploys corresponding checksum files. Your options are to ignore, fail, or warn on missing or incorrect checksums.

layout

> In the above description of repositories, it was mentioned that they all follow a common layout. This is mostly correct. Maven 2 has a default layout for its repositories; however, Maven 1.x had a different layout. Use this element to specify which if it is default or legacy. If you are upgrading from Maven 1 to Maven 2, and you want to use the same repository which was used in your Maven 1 build, list the layout as legacy.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                              http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>          ...
      <repositories>
        <repository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          <checksumPolicy>fail</checksumPolicy>
        </snapshots>
        <url>http://snapshots.maven.codehaus.org/maven2</url>
        <layout>default</layout>
      </repository>
    </repositories>
    <pluginRepositories>
      ...
    </pluginRepositories>
    ...
  </profile>
 </profiles>
 ...
</settings>
```

Plugin Repositories

The structure of the pluginRepositories element block is similar to the repositories element. The pluginRepository elements each specify a remote location of where Maven can find plugins artifacts.

Plugin Repositories in settings.xml.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <repositories>
        ...
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>http://snapshots.maven.codehaus.org/maven2</url>
          <layout>default</layout>
        </pluginRepository>
      </pluginRepositories>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

Active Profiles

Setting active profiles in settings.xml.

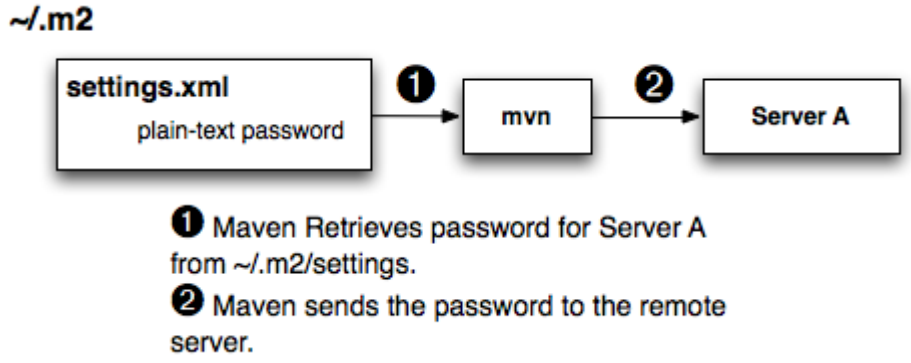
```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                        http://maven.apache.org/xsd/settings-1.0.0.xsd">
   ...
   <activeProfiles>
      <activeProfile>env-test</activeProfile>
   </activeProfiles>
</settings>
```

The final piece of the *settings.xml* puzzle is the activeProfiles element. This contains a set of activeProfile elements, which each have a value of a profile id. Any profile id defined as an activeProfile will be active, regardless of any environment settings. If no matching profile is found nothing will happen. For example, if env-test is an activeProfile, a profile in a *pom.xml* (or *profile.xml* with a corresponding id it will be active. If no such profile is found then execution will continue as normal.

.Encrypting Passwords in Maven Settings

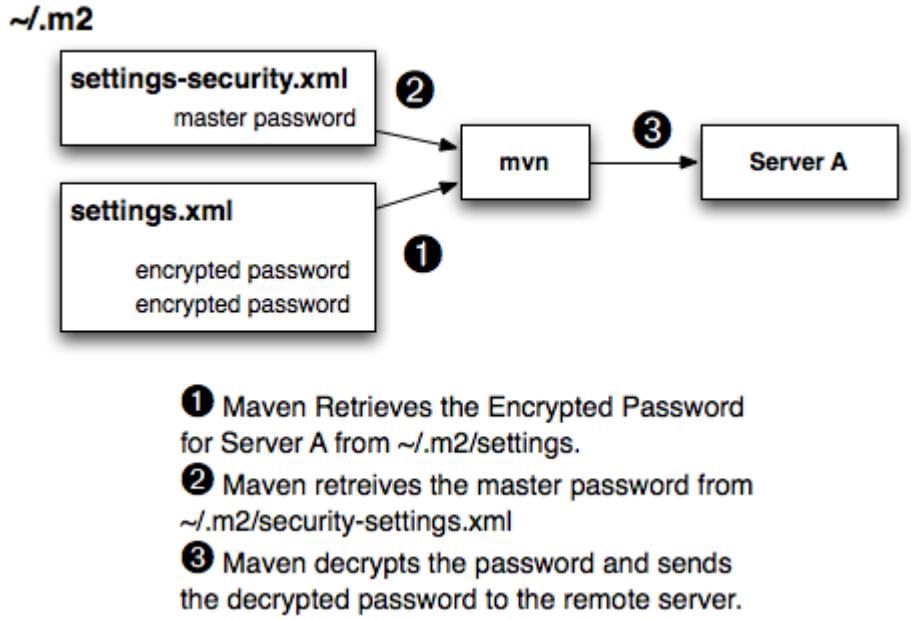
Once you start to use Maven to deploy software to remote repositories and to interact with source control systems directly, you will start to collect a number of passwords in your Maven Settings and without a mechanism for encrypting these passwords, a developer's *~/.m2/settings.xml* will quickly become a security risk as it will contain plain-text passwords to source control and repository managers. Maven 2.1 introduced a facility to encrypt passwords in a user's Maven Settings (*~/.m2/settings.xml*). To do this, you must first create a master password and store this master password in a *security-settings.xml* file in *~/.m2/settings-security.xml*. You can then use this master password to encrypt passwords stored in Maven Settings (*~/.m2/settings.xml*).

To illustrate this feature, consider the process Maven uses to retrieve an unencrypted server password for a user's Maven Settings as shown in Figure 15.1, "Storing Unencrypted Passwords in Maven Settings". A user will reference a named server using an identifier in a project's POM, Maven looks for a matching server in Maven Settings. When it finds a matching server element in Maven Settings, Maven will then use the password associated with that server element and send this password along to the server. The password is stored as plain-text in *~/.m2/settings.xml* and it is readily available to anyone who has read access to this file.



Storing Unencrypted Passwords in Maven Settings

Next, consider the process Maven uses to support encrypted passwords as shown in "Storing Encrypted Passwords in Maven Settings".



. Storing Encrypted Passwords in Maven Settings

To configure encrypted passwords, create a master password by running mvn -emp or mvn --encrypt-master-password followed by your master password.

$ mvn -emp mypassword

{rsB56BJcqoEHZqEZ0R1VR4TIspmODx1Ln8/PVvsgaGw=}

Maven prints out an encrypted copy of the password to standard out. Copy this encrypted password and paste it into a *~/.m2/settings-security.xml* file as shown in

settings-security.xml with Master Password.

```
<settingsSecurity>
   <master>{rsB56BJcqoEHZqEZ0R1VR4TIspmODx1Ln8/PVvsgaGw=}</master>
</settingsSecurity>
```

After you have created a master password, you can then encrypt passwords for use in your Maven Settings. To encrypt a password with the master password, run mvn -ep or mvn --encrypt-password. Assume that you have a repository manager and you need to send a username of "deployment" and a password of "qualityFIRST". To encrypt this particular password, you would run the following command:

```
$ mvn -ep qualityFIRST
{uMrbEOEf/VQHnc0W2X49Qab75j9LSTwiM3mg2LCrOzI=}
```

At this point, copy the encrypted password printed from the output of mvn -ep and paste it into your Maven Settings.

Storing an Encrypted Password in Maven Settings (~/.m2/settings.xml).

```
<settings>
  <servers>
    <server>
      <id>nexus</id>
      <username>deployment</username>
      <password>{uMrbEOEf/VQHnc0W2X49Qab75j9LSTwiM3mg2LCrOzI=}</password>
    </server>
  </servers>
  ...
</settings>
```

When you run a Maven build that needs to interact with the repository manager, Maven will retrieve the Master password from the *~/.m2/settings-security.xml* file and use this master password to decrypt the password stored in your *~/.m2/settings.xml* file. Maven will then send the decrypted password to the server.

What does this buy you? It allows you to avoid storing your passwords in *~/.m2/settings.xml* as plain-text passwords providing you with the peace of mind that your critical passwords are not being stored, unprotected in a Maven Settings file. Note that this feature does not provide for encryption of the password while it is being sent to the remote server. An enterprising attacker could still capture the password using a network analysis tool.

For an extra level of security, you can encourage your developers to store the encrypted master password on a removable storage device like a USB hard drive. Using this method, a developer would plug a removable drive into a workstation when she wanted to perform a deployment or interact with a remote server. To support this, your *~/.m2/settings-security.xml* file would contain a reference to the location of the *settings-security.xml* file using the relocation element.

Configuring Relocation of the Master Password.

```
<settingsSecurity>
  <relocation>/Volumes/usb-key/settings-security.xml</relocation>
</settingsSecurity>
```

The developer would then store the settings-security.xml file at /Volumes/usb-key/settings-security.xml which would only be available if the developer were sitting at the workstation.

Thanking you……