Maven

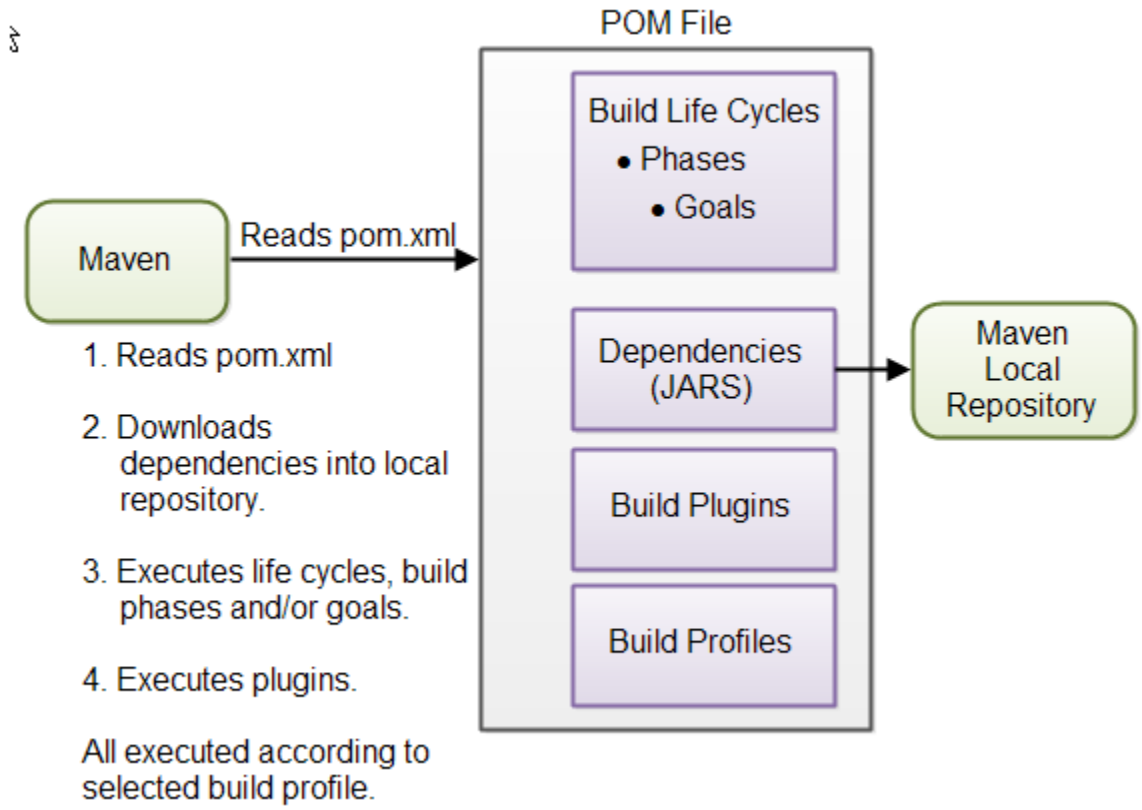
A build tool is a tool that automates everything related to building the software project. Building a software project typically includes one or more of these activities:

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

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:



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.

https://maven.apache.org/plugins/

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.

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>
```

Group ID :

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

Artifact ID :

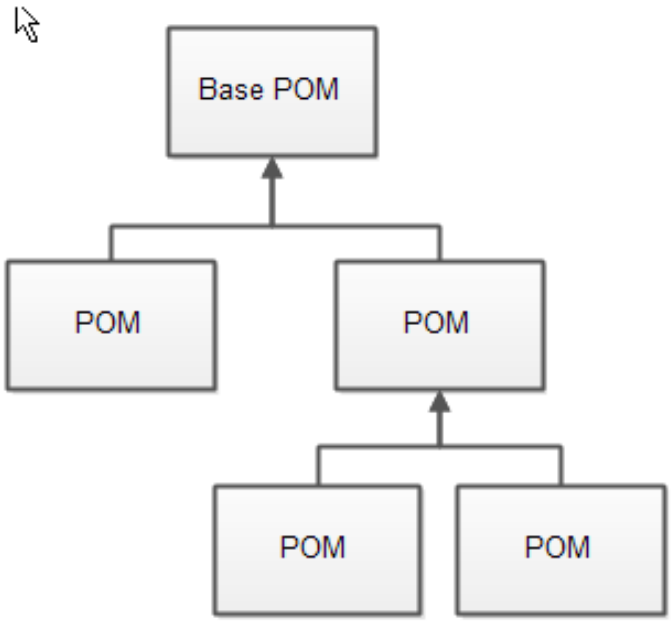
The artifactId element contains the name of the project you are building. In the case of my Java Web Crawler project, the artifact ID would be java-web-crawler. The artifact ID is used as name for a subdirectory under the group ID directory in the Maven repository

Version :

The versionId element contains the version number of the project.

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>
```

Maven Directory Structure

```
- src
  - main
    - java
    - resources
    - webapp
  - test
    - java
    - resources

- target
```

Properties

Custom properties can help to make your pom.xml file easier to read and maintain. In the classic use case, you would use custom properties to define versions for your project's dependencies.

Maven properties are value-placeholders and are accessible anywhere within a pom.xml by using the notation ${name}, where name is the property.

```
<properties>
   <spring.version>4.3.5.RELEASE</spring.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
   </dependency>
</dependencies>
```

Using *Profiles*

Another important feature of Maven is its support for profiles. A profile is basically a set of configuration values. By using profiles, you can customize the build for different environments such as Production/Test/Development:

```
<profiles>
   <profile>
      <id>production</id>
      <build>
         <plugins>
            <plugin>
            //...
            </plugin>
         </plugins>
      </build>
   </profile>
   <profile>
```

```
 <id>development</id>
      <activation>
         <activeByDefault>true</activeByDefault>
      </activation>
      <build>
         <plugins>
            <plugin>
            //...
            </plugin>
         </plugins>
      </build>
   </profile>
 </profiles>
```

As you can see in the example above, the default profile is set to development. If you want to run the production profile, you can use the following Maven command:

```
mvn clean install -Pproduction
```

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.

Maven has built-in dependency management, You specify your project dependencies inside the dependencies element in the 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.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>
```

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,

```
<dependency>
  <groupId>mydependency</groupId>
  <artifactId>mydependency</artifactId>
  <scope>system</scope>
  <version>1.0</version>
  <systemPath>${basedir}\war\WEB-INF\lib\mydependency.jar</systemPath>
</dependency>
```

Snapshot Dependencies

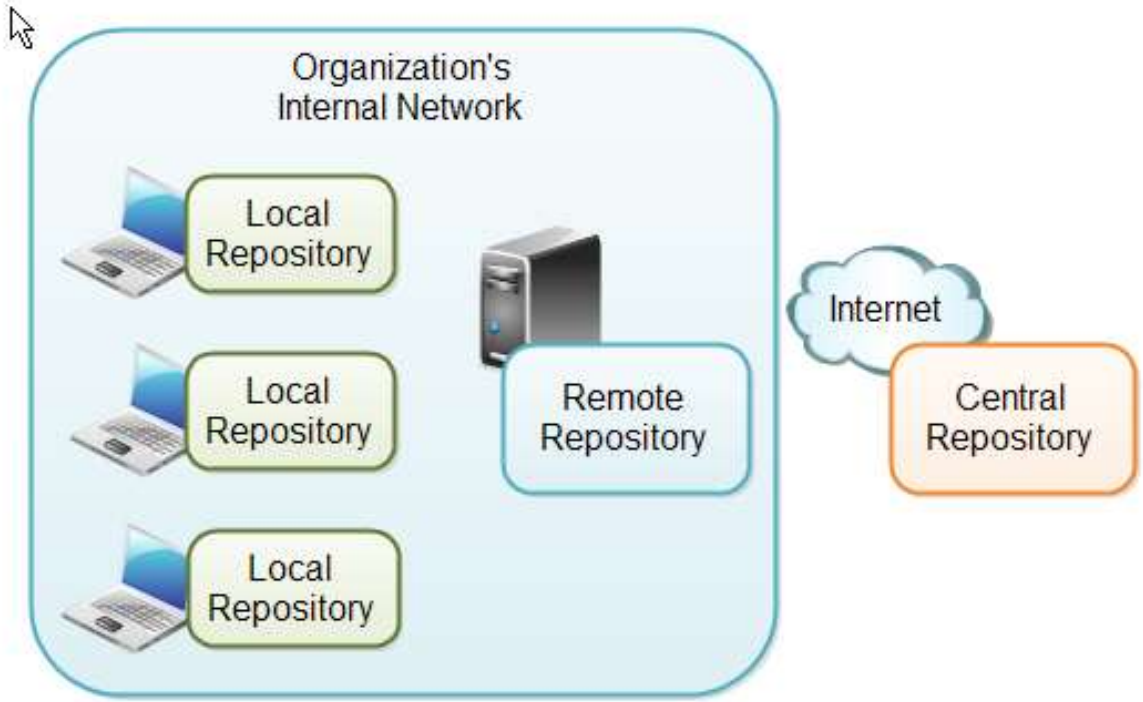
Snapshot dependencies are dependencies (JAR files) which are under development. Instead of constantly updating the version numbers to get the latest version, you can depend on a snapshot version of the project. Snapshot versions are always downloaded into your local repository for every build, even if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

```
<dependency>
    <groupId>com.jenkov</groupId>
    <artifactId>java-web-crawler</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

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

Maven has three types of repository:
Local repository
Central repository
Remote repository



Local Repository

A local repository is a directory on the developer's computer. This repository will contain all the dependencies Maven downloads

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

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.

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

Build Phases

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.

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.

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.

```
<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>
```

Maven Plugins

Maven plugins enable you to add your own actions to the build process. You do so by creating a simple Java class that extends a special Maven class, and then create a POM for the project. The plugin should be located in its own project.

Install Java

Step 1: Update your CentOS 7 system

```
sudo yum install epel-release
sudo yum update
```

Step 2: Install Java

```
sudo yum install java-1.8.0-openjdk
sudo yum install java-1.8.0-openjdk-devel
Java -version
```

set two environment variables: "JAVA_HOME" and "JRE_HOME"

```
sudo cp /etc/profile /etc/profile_backup
echo 'export JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk' | sudo tee -
a /etc/profile
echo 'export JRE_HOME=/usr/lib/jvm/jre' | sudo tee -a /etc/profile
source /etc/profile
```

Verify environment variables

```
echo $JAVA_HOME
echo $JRE_HOME
```

Maven Setup

Download Maven

1. You can download Maven distribution from the Apache Maven website. Currently, the latest release of Maven is version 3.3.9.

wget http://mirrors.viethosting.vn/apache/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz

2. Unpack Maven

sudo tar -xf apache-maven-3.3.9-bin.tar.gz -C /usr/local

The above command will unpack the Maven distribution to /usr/local/apache-maven-3.3.9

3. Next, we will create a symbolic link to the Maven distribution:

cd /usr/local

sudo ln -s apache-maven-3.3.9 maven

4. Create MAVEN_HOME Environment Variables

Create a maven.sh file at /etc/profile.d folder (you can use vi with below command)

sudo vi /etc/profile.d/maven.sh

Enter the follow content to the file:

export M2_HOME=/usr/local/maven

export PATH=${M2_HOME}/bin:${PATH}

Save the file.

5. We will need to activate the above environment variables. We can do that by log out and log in again or simply run below command:

source /etc/profile

6. We can verify whether Maven is installed successfully or not by type command:

mvn -v

Build Project

Build template for an appplication:
mvn archetype:generate
select template number
choose version
give groupid
give artifact id
give version

it will create pom.xm and a folder with artifactid name(it contains app and test sub folders)

To compile application
go to pom.xml directory
mvn compile

To create JAR
mvn package

To run the program
java -cp target/samplejava-1.0-SNAPSHOT.jar org.sankar.javaapp.App (org.sankar.javaapp is groupid and App is class name)

Creating Web Application

mvn archetype:generate
choose web application template j2ee14(269) --> 1156(template for webapp 3.39)
enter 269 and version : latest
groupid : org.sankar.javaweb
artifact id : javaweb
version : fine with default

we can see three dependencies(jsp , servelet and junit)
we can see build phase in the pom.xml apart from dependencies
in the build phase : it will tell how compiler plugin configured

Build
mvn compile
mvn package