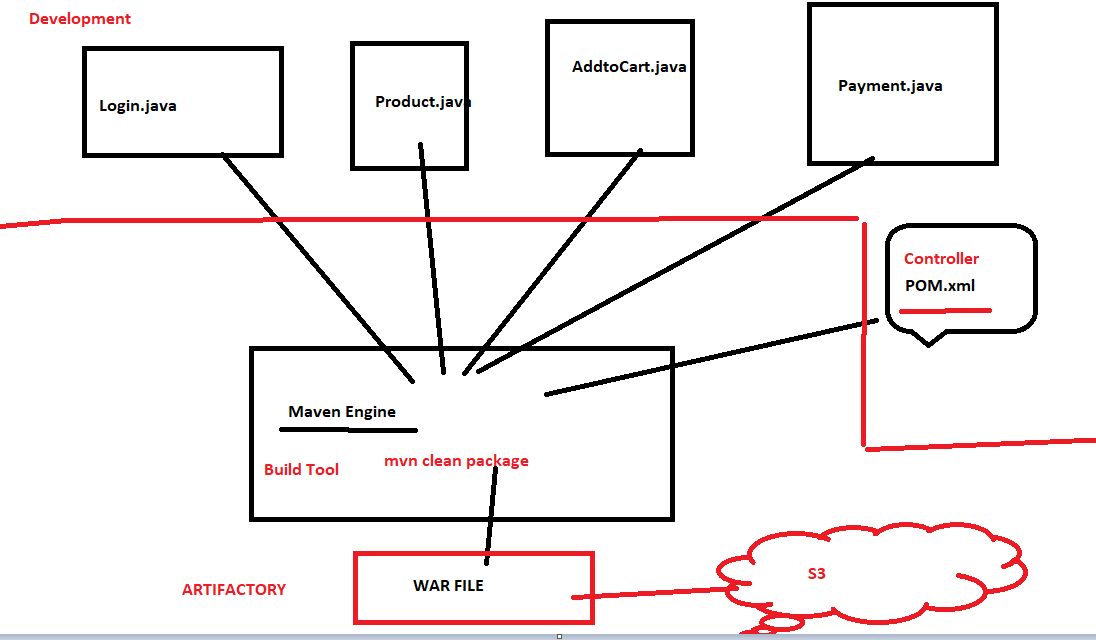
# Maven

is an build tool mainly for Java applications to help the developer at the whole process of a software project.



Check the website - <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>



**What is Maven?**

The Maven project is developed by Apache Software Foundation where it was formerly a part of the Jakarta project. Maven is a **powerful build automation tool** that is primarily used for Java-based projects. Maven helps you tackle two critical aspects of building software –

* It describes how software is built
* It describes the dependencies.

**Maven prefers convention over configuration.** Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven Central Repository and stores them in a local cache. The artifacts of the local projects can also be updated with this local cache. Maven can also help you build and manage projects written in C#, Ruby, Scala, and other languages.

**Project Object Model(POM) file** is an XML file that contains information related to the project and configuration information such as dependencies, source directory, plugin, goals, etc. used by Maven to build the project. When you execute a maven command you give maven a POM file to execute the commands. Maven reads the pom.xml file to accomplish its configuration and operations.

**What Maven does ?**

* **Compilation of Source Code**
* Running Tests (unit tests and functional tests)
* **Packaging the results into JAR’s,WAR’s,RPM’s,etc..**
* Upload the packages to remote repo’s (Nexus,Artifactory)

**Installing Apache Maven**

The installation of Apache Maven is a simple process of extracting the archive and adding the bin folder with the mvn command to the PATH.

## Pre-req –

Download java 1.8

yum install java-1.8\* -y

cd /usr/lib/jvm/

ls -ltr

check the latest java openjdk and use it below to set the ENV

**vim $HOME/.bash\_profile**

export JAVA\_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.51.amzn1.x86\_64

PATH=$JAVA\_HOME/bin:$HOME/bin:$PATH

Save this file

source $HOME/.bash\_profile

alternatives --config java

## Install Maven -

cd /usr/local

wget http://apachemirror.wuchna.com/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz

<http://us.mirrors.quenda.co/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz>

<https://www-eu.apache.org/dist/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz>

tar xzvf apache-maven-3.6.3-bin.tar.gz

- creating a soft link

**ln -s apache-maven-3.6.3 maven**

Update .bash\_profile - create ENV file

export M2\_HOME=/usr/local/maven

export PATH=${M2\_HOME}/bin:${PATH}

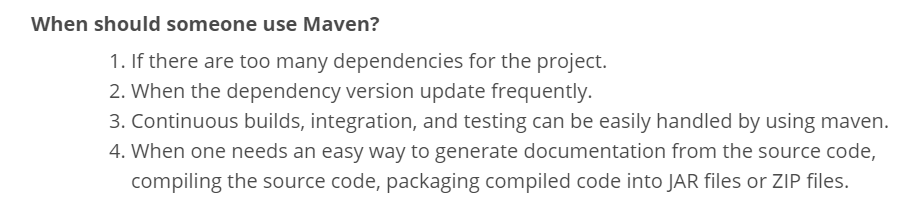
source ~/.bash\_profile

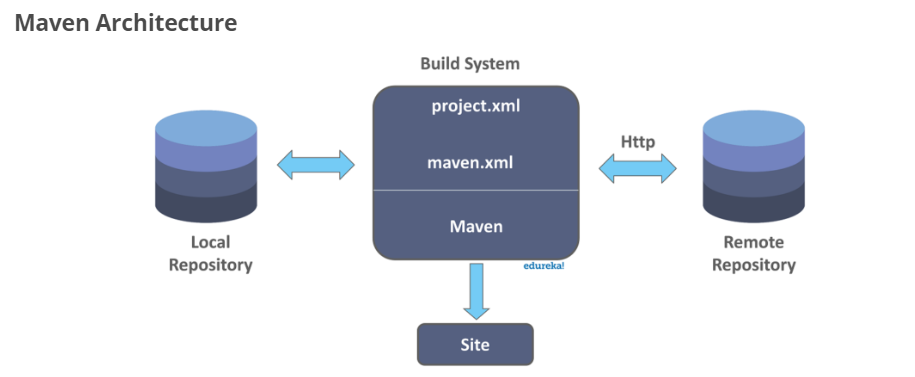
clear

mvn -version

It should print out your installed version of Maven

js$ mvn –version





**Important Concepts**

**First Maven Project**

Let us create our first maven project. Hope you have setup Maven already. In Maven we have ‘Archetype’. It is nothing by a template for projects. Maven provides templates to start a project and using this we can quickly start a Maven project. Execute the following command in cmd prompt,

**agv$**

mvn **archetype:generate** -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false

This will create a sample Maven project skeleton using we can start building the application.We will get a a pom.xml and let us use that to build the newly created Maven project. Go inside the newly created Maven project root and execute the command (this is where the pom.xml is available),

*mvn clean package*

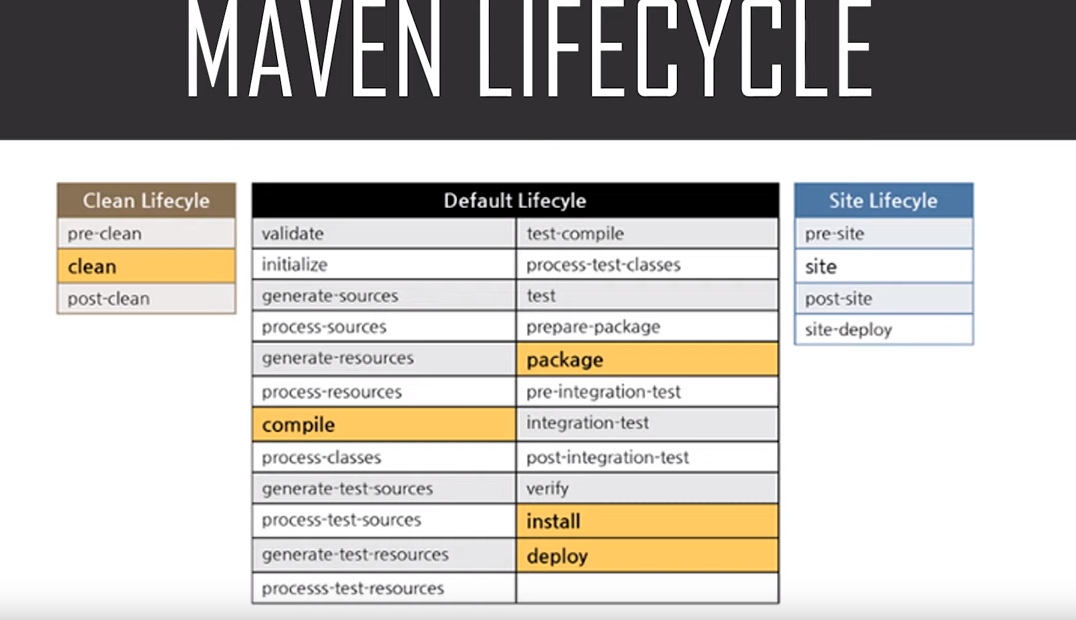
Now this will execute all the Maven phases till the ‘package’ phase. That is, Maven will compile, verify and build the jar file and put it in target folder under the project.

Check the website - <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>

**1. Maven Build Lifecycle**

Maven defines and follows conventions. Right from the project structure to building steps, Maven provides conventions to follow. If we follow those conventions, with minimal configuration we can easily get the build job done.

There are **three built-in build life cycle ‘clean’, ‘default’ and ‘site’**. A life cycle has multiple phases. For example, ‘default’ lifecycle has following phases (listed only the important phases),



**Maven Phases**

Although hardly a comprehensive list, these are the most common *default* lifecycle phases executed.

**validate**: validate the project is correct and all necessary information is available

**compile**: compile the source code of the project

**test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed

**package**: take the compiled code and package it in its distributable format, such as a JAR, WAR. bundles the compiled code (Ex: war / jar)

**integration-test**: process and deploy the package if necessary into an environment where integration tests can be run

**verify**: run any checks to verify the package is valid and meets quality criteria

**install**: install the package into the local repository, for use as a dependency in other projects locally

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

There are two other Maven lifecycles of note beyond the *default* list above. They are

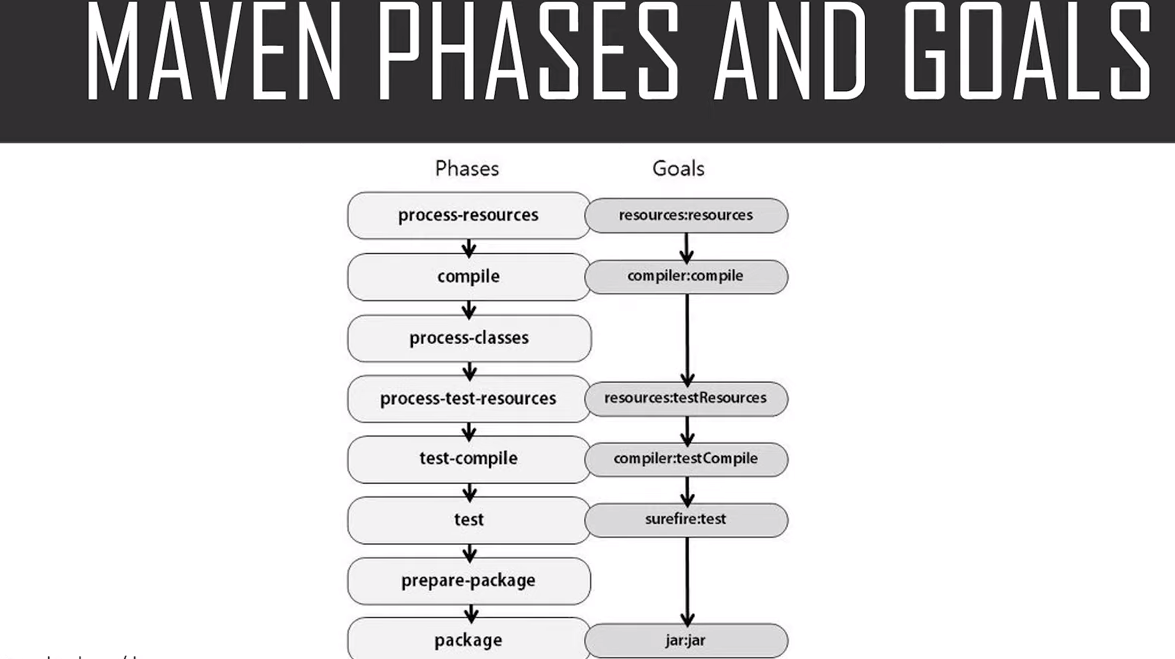
**clean**: cleans up artifacts created by prior builds

**site**: generates site documentation for this project

So to go through the above phases, we just have to call one command:

***mvn*** *<phase> {* Ex: ***mvn*** *install }  
js$ mvn install*

For the above command, starting from the first phase, all the phases are executed sequentially till the ‘install’ phase.



A **goal** represents a specific task which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation.

The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below.

The **clean** and **package** arguments are build phases while the **dependency:copy-dependencies** is a goal.

**mvn clean dependency:copy-dependencies package**

Here the *clean* phase will be executed first, followed by the **dependency:copy-dependencies goal**, and finally *package* phase will be executed.

**2. Maven Repository**

Repository is where the build artifacts are stored. Build artifacts means, the dependent files (Ex: dependent jar files) and the build outcome (the package we build out of a project).

There are two types of repositories, local and remote. Local maven repository(.m2) is in the user’s system. It stores the copy of the dependent files that we use in our project as dependencies. Remote maven repository is setup by a third party(nexus) to provide access and distribute dependent files. Ex: repo.maven.apache.org from internet.

**3. POM Example**

POM stands for **Project Object Model**. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml. And has all the configuration settings for the project build.

Generally we define the project dependencies (Ex: dependent jar files for a project), maven plugins to execute and project description /version etc.

Simplest pom.xml should have 4 important information.

1. **modelVersion**-4.0.0 (*POM version for Maven 2 and is always required)*
2. **groupId** —will identify your project uniquely across all projects, ex:ebs.obill.webs, com.companyname.project
3. **artifactId**— is the name of the jar without version(keeping in mind that it should be jar-name friendly)
4. **version** — if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, …)

<project>  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.intuit.jsapp</groupId>  
 <artifactId>js-app</artifactId>  
 <version>1</version>  
</project>

**4. Maven Dependencies**

There is an element available for declaring dependencies in project pom.xml This is used to define the dependencies that will be used by the project. Maven will look for these dependencies when executing in the local maven repository. If not found, then Maven will download those dependencies from the remote repository and store it in the local maven repository.

Example declaring junit and log4j as project dependencies,

<dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>3.8.1</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.12</version>  
 <scope>compile</scope>  
 </dependency>  
 </dependencies>

scope — describes under which context this dependency will be used.

**5. Maven Plugins**

All the execution in Maven is done by plugins. A plugin is mapped to a phase and executed as part of it. A phase is mapped to multiple goals. Those goals are executed by a plugin. We can directly invoke a specific goal while Maven execution. A plugin configuration can be modified using the plugin declaration.

An example for Maven plugin is ‘compiler’, it compiles the java source code. This *compiler* plugin has two goals *compiler:compile* and *compiler:testCompile*.

Using the configuration element, we can supply arguments to the plugin.

<build>  
 <finalName>springexcelexport</finalName>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-war-plugin</artifactId>  
 <version>2.2</version>  
 </plugin>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-plugin</artifactId>  
 <version>2.16</version>  
 <configuration>  
 <skipTests>true</skipTests>  
 </configuration>  
 </plugin>  
</build>

**Maven Project Structure**

Maven uses a convention for project folder structure. If we follow that, we need not describe in our configuration setting, what is located where. Maven knows from where to pick the source files, test cases etc. Following is a snap shot from a Maven project and it shows the project structure.

agv$ tree  
.  
├── pom.xml  
└── src  
 ├── main  
 │ └── java  
 │ └── com  
 │ └── intuit  
 │ └── App.java  
 └── test  
 └── java  
 └── com  
 └── intuit  
 └── AppTest.java9 directories, 3 files

**Second Project – mvn deploy**

Install Apache and Tomcat in the same server and run the commands below -

mvn **archetype:generate** -DgroupId=com.sample -DartifactId=flipkart -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

Next we need to make a few modifications to the project’s **pom.xml** file.

1. Modify the finalName element to match the following line. The ## characters allows Tomcat to determine both the name of the webapp and the version. This will be used later in this series to run two versions of our webapp at the same time.

<finalName>${project.artifactId}##${project.version}</finalName>

2. Add deploy plugin under the build tag. You will need to change the highlighted sections to match your configuration

<build>

...

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<path>/${project.artifactId}##${project.version}</path>

<url>http://localhost:8080/manager/text</url>

<username>admin</username>

<password>adminadmin</password>

<update>true</update>

</configuration>

</plugin>

</plugins>

...

</build>

3. Create a properties file at **src/main/resources/test.properties** with the following content. We will use resource filtering to insert the version from the pom.xml into this properties file.

version=${project.version}

4. Add filtering of resources. This replaces ${project.version} in the properties file you just created with the version from the pom.xml. This properties file can then be accessed by the jsp pages we are about to modify to display the version of this web app. For more information on resource filtering go to <http://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html>

<build>

...

<resources>

<resource>

<directory>src/main/resources</directory>

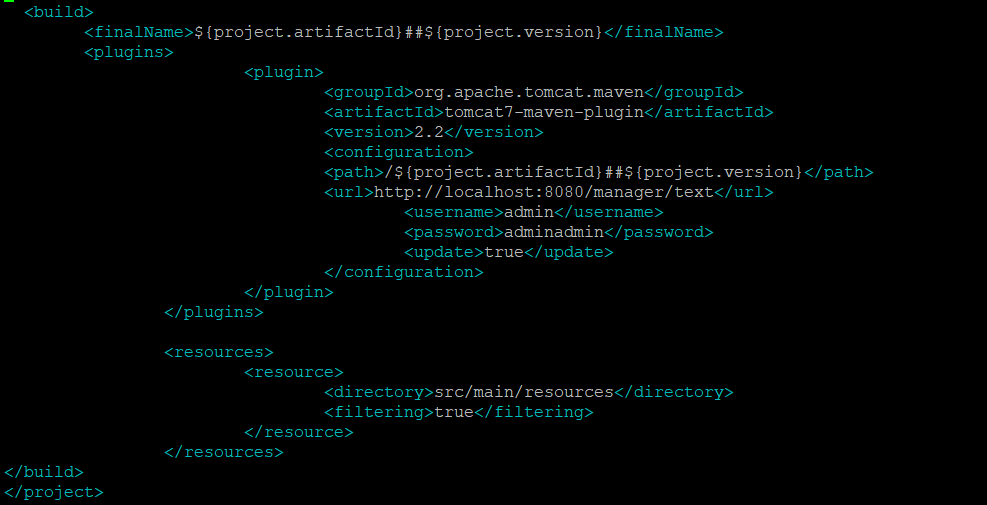
<filtering>true</filtering>

</resource>

</resources>

...

</build>



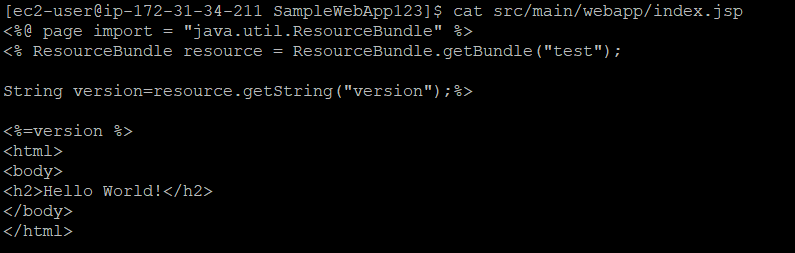
Now we can edit index.jsp to display the version from the properties file. Include the following code in your **src/main/webapp/index.jsp**

<%@ page import = "java.util.ResourceBundle" %>

<% ResourceBundle resource = ResourceBundle.getBundle("test");

String version=resource.getString("version");%>

<%=version %>



Let’s build and deploy our sample web app

cd SampleWebApp123

mvn clean package

mvn tomcat7:deploy - -- This is the main command which we need to see if this is working or not.

Check out the results in your browser (again replacing the highlighted text for your configuration). Go to <http://localhost:8080/SampleWebApp123>

You should see:

Hello World!

1.0-SNAPSHOT