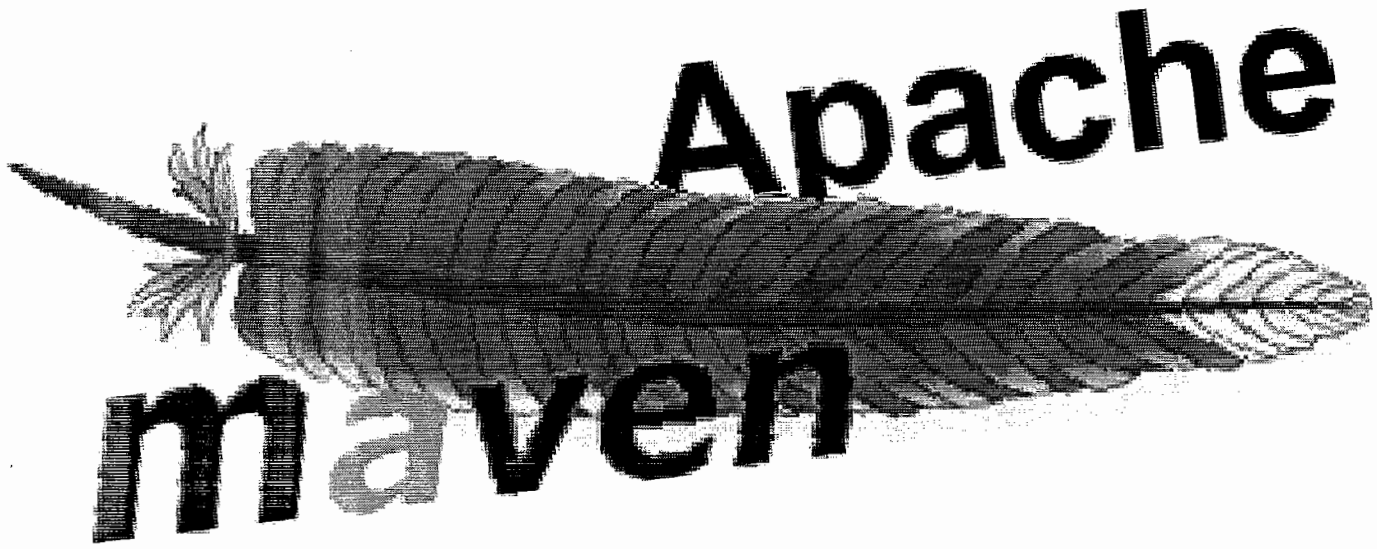
NARESH *i* technologies™



Course Material

By Mr. Nataraj

Naresh *i* Technologies, Opp. Satyam Theatre, Ameerpet, Hyd.
Office: 040- 23746666 Mobile: 9000994007, www.nareshit.com

A Practical Guide To Learn Maven

HighLights

Introduction to Apache Maven

Apache Ant Versus Apache Maven

Key Features of Maven

Downloading and Installing Maven

Maven Repository

Maven Archetype

Standalone Maven Projects

Web Apps through Maven

Maven through Eclipse

Maven Inheritance

Maven Multi-Module Project

Life Cycle of Maven

Maven Interview Questions and Answers

Work hard for what you want because it won't come to you without a fight. You have to be strong and courageous and know that you can do anything you put your mind to. If somebody puts you down or criticizes you, just keep on believing in yourself and turn it into something positive.

#Overview

- In project development, execution and deployment there will be multiple complicated, repetitive operations. These operations are called ***build operations***.
- Keeping our project ready for execution /release is called building project. Doing various activities of build process manually is complex and error prone process.
- To overcome this problem take the support of batch file and automate the process.
- The limitations with .bat file is we cannot write conditional statements to perform build activities.
- To overcome this problem we can go for ***Ant build tool*** where we can automate build activities in conditional environment by using ***build.xml*** file declaratively.
- Ant is just build Tool but it cannot be used for project management activities like downloading jar files, maintaining repositories, providing standard directory structures and etc...
- To solve these problems of ANT we can use ***Maven tool*** which is not only build tool and it is also a ***Project management tool.***

#Introduction to *Apache Maven*

Apache Maven is an advanced build tool to support the developer at the whole process of a software project. Maven acts as both a dependency management tool - it can be used to retrieve jars from a central repository or from a repository you set up - and as a declarative build tool. Maven allows the developer to automate the process of the creation of the initial folder structure for the Java application, performing the compilation, testing, packaging and deployment of the final product. It is implemented in Java which makes it platform-independent.

Maven relies on conventions about how project directories are laid out in order to achieve its "declarativeness." For example, it has a convention for where to put your main code, where to put your web.xml, your unit tests, and so on, but also gives the ability to change them if you need to.

The real out-of-box strength of Maven is its dependency management. You only have to declare the dependencies and Maven will download them, setup the classpath for you, and even deploy the dependencies with your application, if required.

Apache Ant versus *Apache Maven*

vs

- **Ant doesn't have formal conventions.** You have to tell Ant exactly where to find the source, where to put the outputs, etc.
- **Ant is procedural.** You have to tell Ant exactly what to do; tell it to compile, copy, then compress, etc.
- **Ant doesn't have a lifecycle.**
- **Maven uses conventions.** It knows where your source code is automatically, as long as you follow these conventions. You don't need to tell Maven where it is.
- **Maven is declarative;** all you have to do is create a pom.xml file and put your source in the default directory. Maven will take care of the rest.
- **Maven has a lifecycle.** You simply call ***mvn install*** and a series of sequence steps are executed.
- **Maven has intelligence about common project tasks.** To run tests, simple execute ***mvn test***, as long as the files are in the default location. In Ant, you would first have to JUnit JAR file is, then create a classpath that includes the JUnit JAR, then tell Ant where it should look for test source code, write a goal that compiles the test source and then finally execute the unit tests with JUnit.

#Key features of Maven

- Maven tries to avoid as much configuration as possible, by choosing real world default values and supplying project templates (**archetypes**).
- Can download **jar files dynamically**
- Can maintain **multiple repositories** having jar files, plugins and etc..
- Provides **standard project directory structures**
- Gives **Maven inheritance** to share jar files and plugin among the multiple projects
- Allows to develop **multiple module projects**
- Can generate jar, war, ear and etc.. packaging based components or Application
- Can run **unit tests** and can generate **unit test reports**
- Can **generate project documentation**
- Can **clean and install the projects** in the local servers or remote servers

#Downloading and Installing Maven

The Maven project is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project.In case you want to use Maven from the command line, you need to install the Maven command line support. For this you just need to download the Maven's zip file, and extract it to any directory of your choice, and configure the Windows environment variables.

	Link
Binary tar.gz archive	apache-maven-3.3.9-bin.tar.gz
Binary zip archive	apache-maven-3.3.9-bin.zip
Source tar.gz archive	apache-maven-3.3.9-src.tar.gz
Source zip archive	apache-maven-3.3.9-src.zip

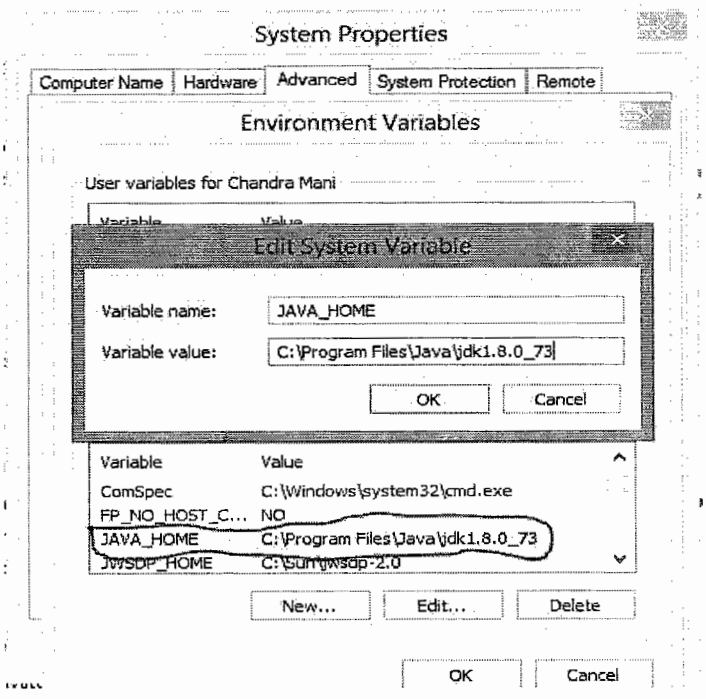
- To download the maven as a zip file, visit official website: http://maven.apache.org/download.cgi and click the above given link**apache-maven-3.3.9-bin.zip** to begin the download and save it to any preferred folder of your choice.

- Before configuring maven make sure that JDK is installed in your system and set **JAVA_HOME** variable to java installation folder.

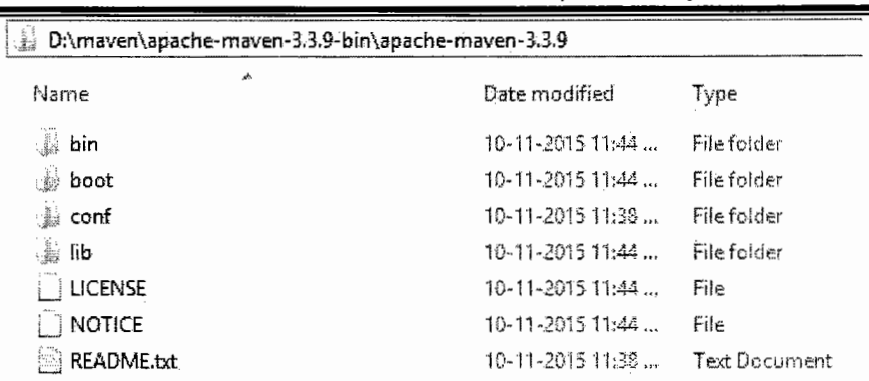
Variable: JAVA_HOME

Value: C:\ProgramFiles\Java\jdk1.8.0_73

(make sure there is no semi-colon at the end)



- Now I suppose you have downloaded and unzipped maven software. I have extracted the maven zip file in the **D:\maven** folder.

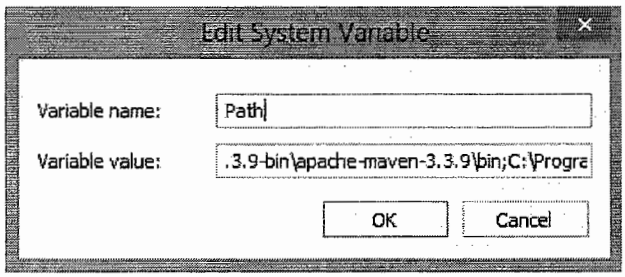


- Add maven bin folder to PATH Variable

  Variable:PATH

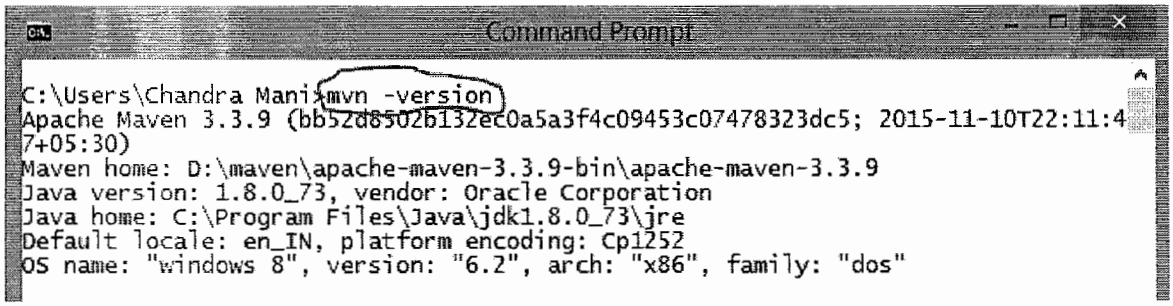
  Value:<Maven_Home>\bin;<existing values>;

  In my case**D:\maven\apache-maven-3.3.9-bin\apache-maven-3.3.9\bin;**



- To verify if maven has been configured successfully, invoke ***mvn –version***(or **mvn –v** or **mvn --version**) in the command prompt

If



you get the message as shown above, it means you have successfully installed and configured Apache Maven in your windows machine.

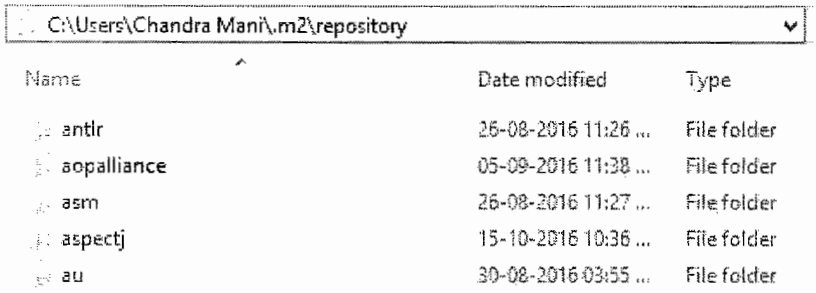
Remember! In maven, all configurations will be done in **pom.xml** file.

#Introduction to Maven Repository

- According to Maven Official Documentation, *"A repository in Maven is used to hold build artifacts and dependencies of varying types."*A maven repository is a place i.e. directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily.There are three types of repository in maven. They are
  1. **Local**
  2. **Central**
  3. **Remote**

1. Local Repository:

- It is user specific repository, generally it will be collected from TL/PL who creates Maven project directory structure.
- Contains jars, plugins, current project related packings and etc..
- Default location: ***c:\users\<usernmae>\.m2***
- Will be created automatically for any maven command apart from (mvn -version)



- Location can be changed through ***<maven_home>\conf\settings.xml*** file using

<localRepository>d:\\maven</localRepository>

```
<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
   | The path to the local repository maven will use to store artifacts.
   |
   | Default: ${user.home}/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->
```

2. Central Repository:

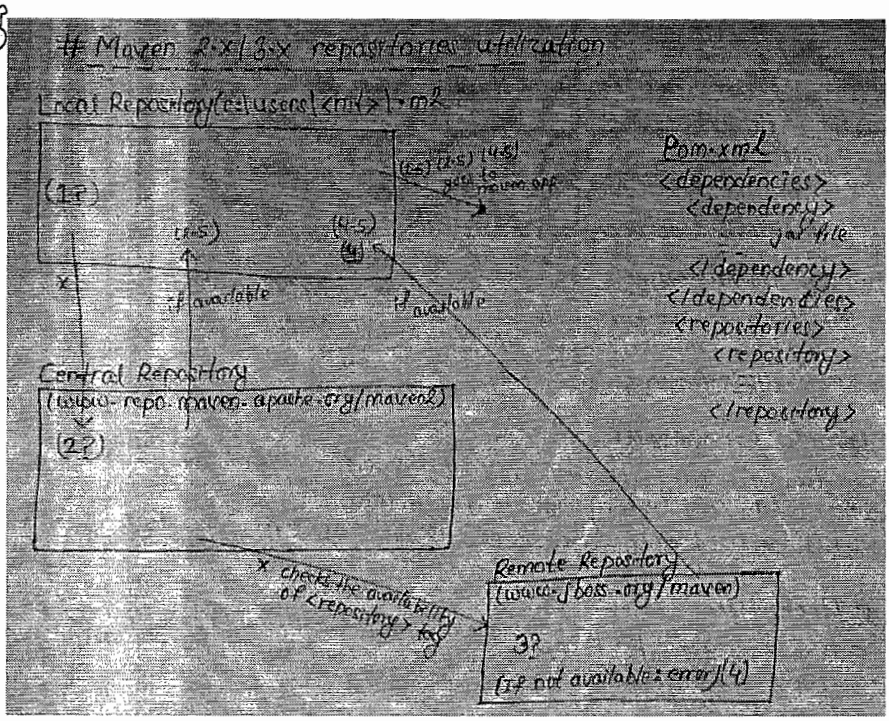
- Avaialble in internet, managed by Apache Maven Community. When Maven does not find any dependency in local repository, it starts searching in central repository.
- URL for central repository: **http://repo.maven.apache.org/maven2**
- Generally maintains free jars and plugins of open source technologies

3. Remote Repository:

- Either company specific or Vendor specific repositories
- Example UHG repository, Jboss repository
- Will be used only when <repository> tag cfg.. in pom.xml

We do all configurations in maven through pom.xml. It is the heart for maven tool. This pom.xml contains project details, packing details (jar/war/ear and etc..), plugin details, remote repository details and dependency(jar) details and etc.

#Repository Utilization in Maven 2.x/3.x



With reference to the above diagram

1?→Maven first searches in Local repository

1.5→If available →gives jar file to app

2? →If not available in Local repository, it will search in central repository

2.5→If available in central repository, it will download in local repository and gives to maven app.

3? If not available in central repository, it will search in remote repository by checking **<repository>**tag in pom.xml

4→If not available→error will be raised

4.5→If available→ the jar will be downloaded to local repository and will be given to maven app.

Every jar that is downloaded to local repository from central repository or vendor's specific remote repository. It is recommended to push to company specific cloud repository or company specific remote repository.

#Maven Archetype

- It is a maven project templating toolkit that helps users to create different types of maven project having maven supplied facilities and user supplied inputs.
- Nearly 1500+ archetypes are supplied by Maven to create different types of project. The popular archetypes are

Archetype ArtifactIds	Description
maven-archetype-quickstart	An archetype to generate a sample Maven project.
maven-archetype-webapp	An archetype to generate a sample Maven Webapp project.
maven-archetype-j2ee-simple	An archetype to generate a simplifed sample J2EE application.

To know more about the different archetypes and its functionalities visit
https://maven.apache.org/guides/introduction/introduction-to-archetypes.html

- Every project directory structure gives pom.xml with minimum details. We can add more tags in that as needed.
- Every Maven Project, dependency jar, plugin and etc are identified with 3 minimum details.

  groupId→Generally company name

  artifactId→Project name/jar name/plugin name

  version→syn:<ver>-SNAPSHOT/RELEASE/FINAL/BETA

#Generating project through Maven

We can create maven projects in any folder of your choice in any drive. Here I am creating it in **d:\maven** folder. To create a new project based on an Archetype, we need to call ***mvn archetype:generate*** goal, like the following:

#Example1 (Generating project in Interactive mode)

Open the Command Prompt and change directory where you want to create your project and call ***mvn archetype:generategoal*** *to begin the process.*

If this is the first time you execute this goal, this may take some time. This is because Maven first loads all required plug-ins and artifacts for the project generation from the Maven central repository.

D:\maven>*mvn archetype:generate*

As soon as you hit enter button, it will start loading all the available artifacts and begins interaction with you like:

Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 902:

Here you have to either enter the archetype number for which you are going to create your project or apply the filter (nothing but filtering archetypes from the list loaded). First of all, we are going to see how to create the standalone project. So, type **maven-archetype-quickstart**and hit enter. Now, simply choose the archetype (hit enter key) and supply the properties configuration and confirm it as shown below.

```
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive co
ntains): 902: maven-archetype-quickstart
Choose archetype:
1: remote -> org.apache.maven.archetypes:maven-archetype-quickstart (An archetyp
e which contains a sample Maven project.)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive co
ntains): 1: 1
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
Choose a number: 6:
Define value for property 'groupId': : naresh
Define value for property 'artifactId': : MathProj1
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  naresh: : com.nt
Confirm properties configuration:
groupId: naresh
artifactId: MathProj1
version: 1.0-SNAPSHOT
package: com.nt
 Y: :
```

If you get BUILD SUCCESS (as shown in the diagram below), that means you have successfully created your first standalone maven project.

```
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype:
 maven-archetype-quickstart:1.1
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: D:\maven
[INFO] Parameter: package, Value: com.nt
[INFO] Parameter: groupId, Value: naresh
[INFO] Parameter: artifactId, Value: MathProj1
[INFO] Parameter: packageName, Value: com.nt
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: D:\maven\MathProj1
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 08:32 min
[INFO] Finished at: 2016-11-30T14:47:02+05:30
[INFO] Final Memory: 9M/38M
[INFO] ------------------------------------------------------------------------
```

After generating the project, the project directory structure looks like the following....

```
d:\maven
    |---->MathProj1
        |--->src
            |---->main
                |-->java
                    |--->com.nt
                        |--->App.java
            |--->test
                |---> ....
        |--->pom.xml
```

In the **App.java** (write the following piece of code)

```
package com.nt;

public class App {

   public int add(int x,int y){

            return x+y;

      }

  public static void main( String[] args ){

    System.out.println( "Hello World!" );

            App app=new App();

            int result= app.add(10,20);

             System.out.println("Result"+ result);

            System.out.println( "End of the App " );

 } }
```

Now start running the following maven goals.....

D:\.....\MathProj1>***mvn compile***

- Compiles the App.java (all source files)

D:\.....\MathProj1>***mvn package***

- Generates jar files in target folder having <projectname>-ver.jar file

D:\.....\MathProj1>***mvn clean***

- Cleans the project .. deletes target folder

D:\.....\MathProj1>***mvn clean package***

- Cleans the project and also creates jar file with latest code

To run jar file App manually

MathProj1>***java -cp target/MathProj1-1.0-SNAPSHOT.jar com.nt.App***

#Maven Unit Testing and Unit Test Report

With the help of we can run the test cases can generate unit test reports in HTML. Maven does so by running the unit tests and recording the results of the unit tests. Maven then generates an HTML report from the unit test results. We can then read the report to see if any of the unit test have failed during the build. Open the **AppTest.java** java file**(D:\maven\MathProj1\src\test\java\com\nt)** of **MathProj1** project and do the following changes in **testAdd()** method ***AppTest.java***

By default Maven's **pom.xml**gives <dependency> tag to download junit jar. So, we can write test cases in **Proj\src\test\java\.....\AppTest.java)**

In AppTest.java

```
public void testAdd() {
        int x=10;
        int y=20;
        int expected=30;
        int actual=new App().add(10,20);
        Assert.assertEquals("Result1",expected,actual);
}
```

Now let us run the above test cases and generate the test report. The Maven unit test reports are generated by the **Maven Surefire plugin**. Therefore such unit test report is also sometimes referred to as ***surefire*** *report*.

D:\.....\MathProj1>***mvn test***

- To run these test cases

D:\.....\MathProj1>***mvn surefire-report:report***

- To run test cases and to get test report

D:\.....\MathProj1>***mvn surefire-report:report-only***

- To get test report directly

(Note: We can get test report in target/site/surefire-report.html form)

D:\.....\MathProj1>***mvn install***

- To keep generated jar file (Application jar) in Local Repository

The above jar file can be placed in local repository in other maven projects by specifying <groupId>, <artifactId>, <version> tags in <dependency> tag.

D:\.....\MathProj1>***mvn site***

- To get documentation about total project. (generates project document in target/site folder as index.html file)

While writing <dependency> tag to download the jar file, we can specify the scope by using <scope> tag. The possible scopes are **compile, runtime, provided, system, test, import** etc

In pom.xml

```
<dependency>
 <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
   <scope>test</scope>
</dependency>
```

Maven is having three life cycle and each life cycle is having certain phases. By specifying these lifecycle phases, we can configure plugins in pom.xml so that plugin will perform the activity at the specified phase.

The Maven Life cycles are

a. ***clean(3 phases)***
b. ***default(23 phases)***
c. ***site (4 phases)***

If you ask maven to perform 17th phase (package) of default life cycle, it also performs the remaining 16 phases in a sequence like compile, test-compile, test and etc....

#Creating standalone maven project in non-interactive mode

Syntax:

mvn archetype:generate

-DgroupId=groupid

-DartifactId=artifactid

-DarchetypeArtifactId=maven-archetype-quickstart

-DinteractiveMode=booleanValue

Example: GenerateSqrRoot

Here we will create a simple standalone project to find the square root of a given number. I am creating all my maven projects in ***D:\maven*** folder.

mvn archetype:generate

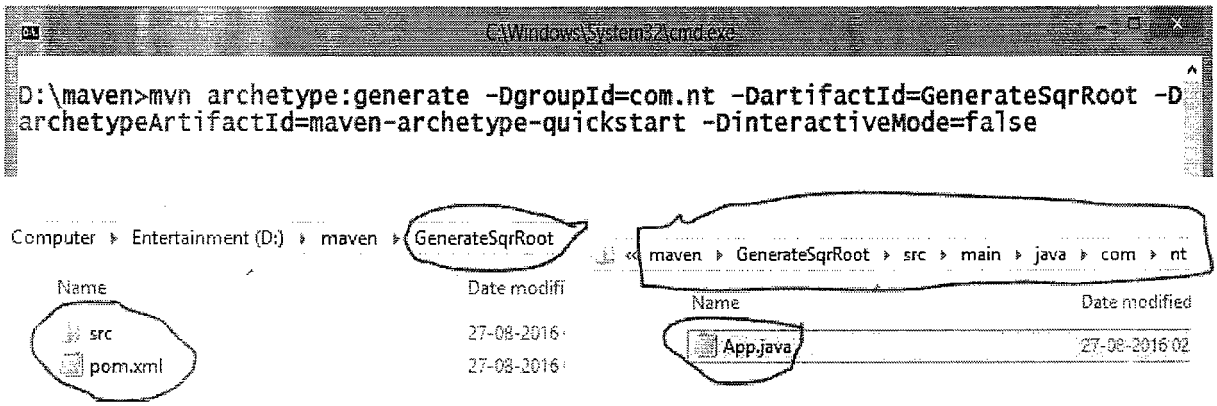
-DgroupId=com.nt

-DartifactId=GenerateSqrRoot

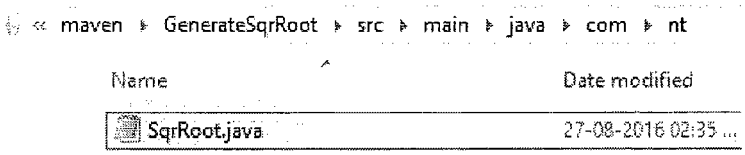
-DarchetypeArtifactId=maven-archetype-quickstart

-DinteractiveMode=false

Here, we are instructing the maven to create the project directly without interactive mode. Run the above command in command prompt specifying the path where you want to generate the project directory structure. (**Caution:***while executing the above command make sure it is in the single line as shown in the picture below.*)



As we can see in the above picture, after the successful execution of the maven command, the directory structure is created with "App.java" java file. Now let's customize and write our own logic to generate square root. Rename that file with "SqrRoot.java" and write our own logic.



In SqrRoot.java

```
package com.nt;
/**
 * Generate SquareRoot of a given number
```

```
*/
import java.util.Scanner;
import java.lang.Math;
public class SqrRoot{
        public static void main(String args[]){
                Scanner s= new Scanner(System.in);
                System.out.println("Enter number to find its square root: ");
                int n=s.nextInt();
                System.out.println(Math.sqrt(n));
        }//main
}//class
```

Now, we will compile and execute our standalone project with the help of maven. For this we need a plugin "***exec-maven-plugin***". **What actually is this plugin and what it does???**

Official documentation says (http://www.mojohaus.org/exec-maven-plugin/)

Exec Maven Plugin provides 2 goals to help execute system and Java programs.

- ***exec:exec*** execute programs and Java programs in a separate process.
- ***exec:java*** execute Java programs in the same VM.

So, we need to provide this information in pom.xml of the our GenerateSqrRoot project.

Add the following cfgs in the pom.xml of our project after or before <dependencies> tag.

In pom.xml

```
<build>
<plugins>
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.4.0</version>
<configuration>
<mainClass>com.nt.SqrRoot</mainClass>
</configuration>
</plugin>
</plugins>
</build>
```

Execute the project with following command

Invoking "mvn exec:java" on the command line will invoke the plugin which is configured to execute the class "com.nt.SqrRoot". So, to trigger the plugin from the command line, just run:

D:\maven\GenerateSqrRoot>mvn compile

Once code is compiled, the following command runs your class

D:\maven\GenerateSqrRoot>mvn package

D:\maven\GenerateSqrRoot>mvn exec:java

*(**Note**: while maven starts execution, supply your desired integer value to get the square root)*

```
D:\maven\GenerateSqrRoot>mvn exec:java
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building GenerateSqrRoot 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ GenerateSqrRoot ---
Enter number to find its square root:
9
3.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:25 min
[INFO] Finished at: 2016-11-30T17:15:09+05:30
[INFO] Final Memory: 6M/15M
[INFO] ------------------------------------------------------------------------
```

Now let's run the same project with little bit of twist. We can run the same project by specifying the class name in the command itself. For this we have to remove the <configuration> related tag in the pom.xml inside <plugin> tag that we just placed. So that our pom.xml now looks like this....

In pom.xml

```
<build>
<plugins>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
<version>1.4.0</version>
</plugin>
</plugins>
</build>
```

Syntax for execution

mvn exec:java -Dexec.mainClass="com.nt.module.MainClassName"

Execution: *D:\maven\GenerateSqrRoot>**mvn exec:java -Dexec.mainClass="com.nt.SqrRoot"***

```
D:\maven\GenerateSqrRoot>mvn exec:java -Dexec.mainClass="com.nt.SqrRoot"
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building GenerateSqrRoot 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ GenerateSqrRoot
Enter number to find its square root:
81
9.0
```

Let's explore more ways to run the same project.

Now, if you want to execute the **exec:java** goal as part of your standard build, you'll need to bind the goal to a particular phase of the default lifecycle. To do this, declare the phase to which you want to bind the goal in the execution element:

For this we need to specify **<phase>** (phase of your choice like package, test etc.) and **<goal>** tags in pom.xml. So, the changed pom.xml looks like the following.

In pom.xml

```
<build>
<plugins>
        <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.4.0</version>
        <executions>
        <execution>
                <id>my-execution</id>
                <phase>package</phase>
                <goals>
                <goal>java</goal>
                </goals>
        </execution>
        </executions>
        <configuration>
                <mainClass>com.nt.SqrRoot</mainClass>
        </configuration>
        </plugin>
</plugins>
</build>
```

Execution:

*D:\maven\GenerateSqrRoot>**mvn package***

*(**Note**: This executes com.nt.SqrRoot class during the package phase)*

```
D:\maven\GenerateSqrRoot>mvn package
[INFO] Scanning for projects
```

```
[INFO] --- exec-maven-plugin:1.4.0:java (my-execution)
Enter number to find its square root:
9
3.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Example: *SwapDemoCmd*

This is just a proof of concept example. Here we are going to see how to specify command line arguments through command line along with maven commands and how to specify arguments in the pom.xml through **<argument>**tag. Let's do it by creating a fresh maven project.

mvn archetype:generate

-DgroupId=com.nt

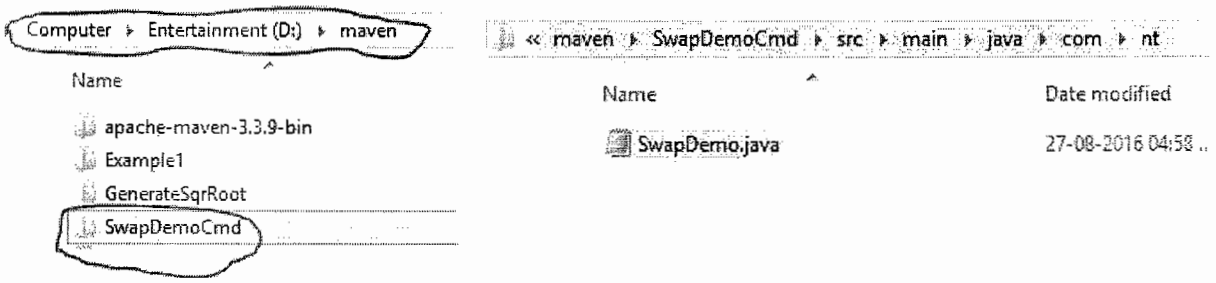
-DartifactId=SwapDemoCmd

-DarchetypeArtifactId=maven-archetype-quickstart

-DinteractiveMode=false

```
D:\maven>mvn archetype:generate -DgroupId=com.nt -DartifactId=SwapDemoCmd -Darch
etypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

This command will generate a directory structure for SwapDemoCmd project with "App.java" as a default java class. Let's change it with "SwapDemo.java".



In SwapDemo.java

```
package com.nt;
/**
 * SwapDemo
 *
 */
class Swap{
	int a,b;
	void accept(int x, int y){
		a=x;
		b=y;
	}
	void swapValues(){
		a=a+b;
		b=a-b;
		a=a-b;
	}
	void disp(){
		System.out.println("Value of a: "+a);
		System.out.println("Value of b: "+b);
	}
}
```

```
class SwapDemo{
	public static void main(String args[]){
		//Check if two values are entered as cmd line args or not
		if(args.length!=2){
			System.out.println("Plz enter two values:");
		}
			else{
				int x1=Integer.parseInt(args[0]);
				int x2=Integer.parseInt(args[1]);
				//create an object of Swap class
				Swap sw= new Swap();
				sw.accept(x1, x2);
				System.out.println("Before swapping");
				System.out.println("================");
				sw.disp();
				sw.swapValues();
				System.out.println("After swapping");
				System.out.println("================");
				sw.disp();
			}
	}
}//class
```

*Now, do some changes in pom.xml of **SwapDemoCmd** project.*

In pom.xml

```
<build>
<plugins>
<plugin>
	<groupId>org.codehaus.mojo</groupId>
	<artifactId>exec-maven-plugin</artifactId>
<version>1.4.0</version>
</plugin>
	<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-archetype-plugin</artifactId>
	<version>2.4</version>
	</plugin>
```

</plugins>

</build>

Execution:

Since we have to pass command line arguments to execute this project, let's see the syntax to do so.

Syntax

mvn exec:java -Dexec.mainClass="com.nt.module.MainClassName" -Dexec.args="arg0 arg1 arg2"

Here, I will pass 8 and 4 as arguments. So the command looks like this.

D:\maven\SwapDemoCmd>mvn compile

D:\maven\SwapDemoCmd>mvn package

D:\maven\SwapDemoCmd>mvn exec:java -Dexec.mainClass="com.nt.SwapDemo" -Dexec.args="8 4"

```
D:\maven\SwapDemoCmd>mvn exec:java -Dexec.mainClass="com.nt.SwapDemo" -Dexec.arg
s="8 4"
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building SwapDemoCmd 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ SwapDemoCmd ---
Before swapping
================
Value of a: 8
Value of b: 4
After swapping
================
Value of a: 4
Value of b: 8
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.683 s
[INFO] Finished at: 2016-11-30T17:45:47+05:30
[INFO] Final Memory: 6M/16M
[INFO] ------------------------------------------------------------------------
```

Wouldn't it be great if we can specify arguments in the pom.xml itself and run the project in a maven phase? For example, we can run the SwapDemo.main() method as part of the package phase. To specify the arguments in pom.xml, we write **<argument>** under **<configuration>** tag. So, mention the following configurations in pom.xml of **SwapDemoCmd** project.

In pom.xml

<build>

<plugins>

<plugin>

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

<artifactId>exec-maven-plugin</artifactId>

<version>1.4.0</version>

<executions>

<execution>

<phase>package</phase>

<goals>

```
<goal>java</goal>
</goals>
<configuration>
<mainClass>com.nt.SwapDemo</mainClass>
<arguments>
<argument>3</argument>
<argument>9</argument>
</arguments>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

Execution: *To run the exec plugin with above configuration, simply run the corresponding phase.*

D:\maven\SwapDemoCmd>mvn package

#Converting Maven standalone project into eclipse project

D:\maven\SwapDemoCmd>mvn eclipse:eclipse

Open SwapDemoCmd from Eclipse using File menu→import→General→Existing Project into Workspace→Browse the Root Directory (i.e. ***SwapDemoCmd***) of the project.

#Generating Maven web project by using *maven-archetype-webapp*

mvn archetype:generate

-DgroupId=com.nt.servlet

-DartifactId=WishApp

-DarchetypeArtifactId=maven-archetype-webapp

-Dpackage=com.nt.servlet

-DinteractiveMode=false

```
D:\maven>mvn archetype:generate -DgroupId=com.nt.servlet -DartifactId=WishApp -D
archetypeArtifactId=maven-archetype-webapp -Dpackage=com.nt.servlet -Dinteractiv
eMode=false
```

Hit enter key and you will get the following director structure

```
WishApp
-----------
 src
  |---main
     |---java
        |---com
           |---nt
              |---servlet
     |----webapp
        |-----WEB-INF
             |--web.xml
             |----index.jsp
|---pom.xml
```

(Incase if com.nt.servlet package is not created, create it manually)

```
WishApp
-----------
 src
  |---main
     |---java
        |---com
           |---nt
              |---servlet
                 |---WishServlet.java
     |----webapp
        |-----WEB-INF
             |--web.xml
             |----index.jsp
|---pom.xml
```

In com/nt/servlet folder→WishServlet.java, copy the following code

In WishServlet.java

```
package com.nt.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;
import java.util.Date;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

publicclassWishServletextends HttpServlet {
      publicvoid doGet(HttpServletRequest req, HttpServletResponse res) throws IOException,
ServletException {

            // General setting
            res.setContentType("text/html");
            PrintWriter pw = res.getWriter();
            Calendar calendar = Calendar.getInstance();
            int hr = calendar.get(Calendar.HOUR_OF_DAY);

            Date d = new Date();
            pw.println("<b><i><center> Date and Time is " + d + " </b></i></center>");

            if (hr <= 12)
                  pw.println("<h2>GOOD MORNING</h2>");

            elseif (hr <= 16)
                  pw.println("<h2>GOOD AFTERNOON</h2>");

            elseif (hr <= 20)
                  pw.println("<h2>GOOD EVENING</h2>");

            else {
                  pw.println("<h2>GOOD NIGHT</h2>");
            }
            //close the stream
                  pw.close();
      }// doGet(req,res)

      @Override
```

```
	publicvoid doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
		// TODO Auto-generated method stub
		doGet(req,res);
	}//doPost(-,-)
}// class
```

In index.jsp

```
<html>
<body>
	<h1>HTML-SERVLET COMMUNICATION</h1>
	<a href="wurl">GenerateWishMessage</a>
</body>
</html>
```

In WishApp/pom.xml(Add the following ***<dependency>*** under ***<dependencies>*** tag)

```
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>javax.servlet-api</artifactId>
	<version>3.1.0</version>
	<scope>provided</scope>
</dependency>
```

In web.xml

```
<web-app>
	<servlet>
		<servlet-name>wish</servlet-name>
		<servlet-class>com.nt.servlet.WishServlet</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>wish</servlet-name>
		<url-pattern>/wurl</url-pattern>
	</servlet-mapping>
	<welcome-file-list>
		<welcome-file>index.jsp</welcome-file>
	</welcome-file-list>
</web-app>
```

Execution

D:\maven\WishApp>mvn package

- When you get BUILD SUCCESS while executing the above goal, one war file will be created in target folder of WishApp application. (***D:\maven\WishApp\target\WishApp-1.0-SNAPSHOT.war***)

Now, start the tomcat server and deploy the generated war file (copy and paste ***WishApp-1.0-SNAPSHOT.war***in **<tomcat-version>/webapps** folder)

- http://localhost:4040/WishApp-1.0-SNAPSHOT/

#Working with Tomcat7 Maven Plugin

- We can even work with embedded tomcat server and deploy the generatedwar file. For this we have to use ***Tomcat7 Maven Plugin***and provide its configuration in pom.xml.

Place the following piece of code in ***WishApp/pom.xml*** after or before ***<dependencies>*** tag.

In WishApp/pom.xml

```
<build>
<finalName>WishApp</finalName>
<plugins>
<!-- Config: Maven Tomcat Plugin -->
<plugin>
<groupId>org.apache.tomcat.maven</groupId>
<artifactId>tomcat7-maven-plugin</artifactId>
<version>2.1</version>
<!-- Config: contextPath and Port (Default - /WishApp : 8080) -->
<configuration>
<server>tomcat</server>
<url>http://localhost:2020/manager</url>
                <path>/WishApp</path>
</configuration>
</plugin>
</plugins>
</build>
```

During deployment, this cfg tells Maven to deploy the WAR file to embedded tomcat server via "http://localhost:2020/manager/", on path "/WishApp".

Execution

D:\maven\WishApp>mvn tomcat7:run -Dmaven.tomcat.port=9090

Note: While executing, we have to mention the port number on which it will run.

After executing this command, if we get the following message in the command prompt

INFO: Starting ProtocolHandler ["http-bio-9090"]

then open any browser of your choice and run the web-application (***http://localhost:9090/WishApp***)

Rather than specifying the port number in command prompt, we can do it through pom.xml also with the following cfgs.

In pom.xml

```
<configuration>
```

```
<server>tomcat</server>
<url>http://localhost:2020/manager</url>
 <path>/WishApp</path>
 <port>9090</port>
</configuration>
```

Execution:

D:\maven\WishApp>mvn tomcat7:run

Test the App with this url: ***http://localhost:9090/WishApp***

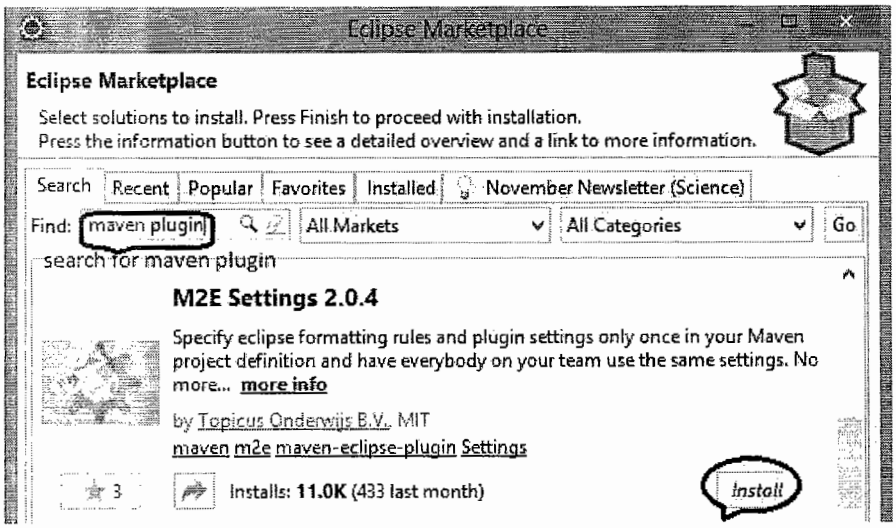
Commands to manipulate WAR file on Tomcat.

- ***mvn tomcat7:deploy*** //deployment
- ***mvn tomcat7:undeploy*** //undeployment
- ***mvn tomcat7:redeploy*** //redeployment

#Maven Eclipse

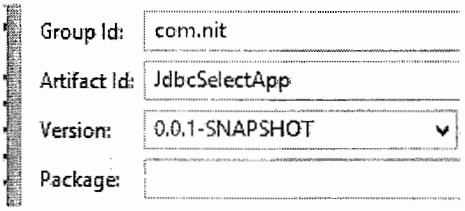
In order to work with maven inside eclipse, we have to install **M2E Settings 2.0.4** plugin from eclipse marketplace.

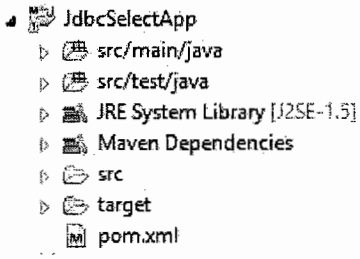
Help menu → Eclipse Marketplace → search maven plugin → **M2E Settings 2.0.4**→install



#Creating standalone maven project in Eclipse

File→New→Maven Project → Next → choose ***maven-archetype-quickstart***and enter GroupId and ArtifactId as shown in the diagram. The default project directory structure as shown in the diagram is created.





As you can see in the newly created project directory structure, the default JRE System Library [J2SE-1.5] (nothing but jdk 1.5) is taken even though you might have installed some other versions of jdk (may be jdk1.7 or jdk 1.8). **According to maven documentation:** ***"at present the default source setting is 1.5 and the default target setting is 1.5, independently of the JDK you run Maven with. If you want to change these defaults, you should set source and target as described in Setting the -source and -target of the Java Compiler."***

You can solve this issue (applicable both in case of standalone-**quickstart** and **webapp** kind of application) by doing certain configuration in pom.xml and updating the maven project. Let's see how to do it.

Step1:Add following configuration after**<dependencies>**tag in **JdbcSelectApp/pom.xml.**

In JdbcSelectApp/pom.xml

```
<build>
	<finalName>JdbcSelectApp</finalName>
	<plugins>
		<plugin>
			<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<version>3.1</version>
				<configuration>
					<source>1.8</source>
					<target>1.8</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
```

Important Note: Since I have installed jdk 1.8 version so, I have cfg <source> and <target> to 1.8. You do the changes as per your Jdk installation.

Step2: Update the Maven project

Right click on the **JdbcSelectApp→Maven→Update Project..→Update Maven Project** dialog box will appear. **Make sure JdbcSelectApp is checked→** Hit **OK button.** You are done...!!

Once you are done with all these modifications now let's focus on project development.

- Add package ***com.nt.jdbc*** to ***src/main/java*** folder.
- Create on java class ***SelectTest.java*** under ***com.nt.jdbc*** package of ***JdbcSelectApp.***

Now, we are going to work with oracle database whose jar file is commercial. Commercial jar files are not available in central repository. At the same time we cannot collect them from remote repository because they are commercial. In this situation, we can directly add them to local repository by using ***mvn install:install*** goal in command prompt.

mvn install:install-file

–Dfile=<path of ojdbc jar from your local system>

-DgroupId=oracle

-DartifactId=ojdbc6

-Dversion=11.2

The**<dependency>**equivalent tag is

<dependency>
<groupId>oracle</groupId>
<artifactId>ojdbc6</artifactId>
<version>11.2</version>
</dependency>

But there is a Good News for lazy programmers like us is, if you don't want to install this ojdbc jar manually, there is one third-party supplied simulator jar file for ojdbc6.jar in maven centralrepository. Just add its dependency and enjoy all database related application development through maven. So, you can add the following ***<dependency>***if you don't want to go with the above step.

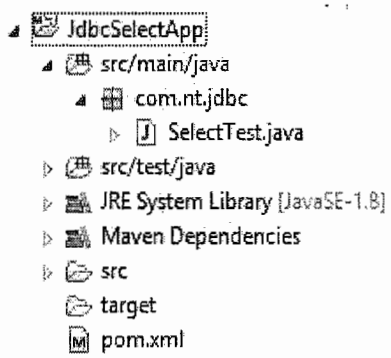
<dependency>

<groupId>com.hynnet</groupId>

<artifactId>oracle-driver-ojdbc6</artifactId>

<version>12.1.0.1</version>

</dependency>



In SelectTest.java

```
package com.nt.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

publicclass SelectTest {
      privatestaticfinal String SELECT_MAX_SAL="SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE
SAL=(SELECT max(SAL) FROM EMP)";
      publicstaticvoid main(String[] args) {
            Connection con = null;
            Statement st = null;
            ResultSet rs = null;
            try {
                  //Register the type4 oracle driver
                  Class.forName("oracle.jdbc.driver.OracleDriver");

                  //Get the Connection object
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","scott","tiger");

                  // Create the Statement Object
                  if (con != null)
                        st = con.createStatement();

                  // Send and execute the SQL query
                  if (st != null)
                        rs = st.executeQuery(SELECT_MAX_SAL);
```

```
                    // Process and execute the ResultSet object
                    if (rs != null) {
                          while (rs.next()) {
      System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getString(3) + " " +
rs.getInt(4));
                          } // while
                    }

              } // try
              catch (SQLException se) {
                    se.printStackTrace();
              } catch (ClassNotFoundException cfe) {
                    cfe.printStackTrace();
              } catch (Exception e) {
                    e.printStackTrace();
              } finally {
                    try {
                          if (rs != null)
                                rs.close();
                    } catch (SQLException se1) {
                          se1.printStackTrace();
                    }

                    try {
                          if (st != null)
                                st.close();
                    } catch (SQLException se2) {
                          se2.printStackTrace();
                    }

                    try {
                          if (con != null)
                                con.close();
                    } catch (SQLException se) {
                          se.printStackTrace();
                    }

              } // finally
      }// main
}//class
```

- Finally add the following **<plugin> under <build>** tag in ***pom.xml*** *specifying the* ***<mainClass>***

```
<plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.4.0</version>
      <configuration>
            <mainClass>com.nt.jdbc.SelectTest</mainClass>
      </configuration>
</plugin>
```

Execution

Right click on **JdbcSelectApp→Run As→Maven Build..→** Specify the Goals: (***exec:java***) **→Run**

****

Another useful feature of maven is that if we add **<dependency> tag** for main jar file in pom.xml file, it downloads mian and dependents jar files.

✓ **Example:** If we add **<dependency>tag for <hibernate-core>**tag, it not only downloads the main jar file, it also downloads lot of dependent jar files.

By using **<repositories>/<repository>** ta, we can make maven even downloading jar files from remote repositories to local repositories if they are not available in central repositories.

✓ **Example:** RestEasy jar files are available in repository maintained by JBOSS.

```
<repositories>
<repository>
      <id>JBOSS Repository</id>
      <url>https://repository.jboss.org/nexus/content/group/public-jboss</url>
</repository>
</repositories>
```

#Maven Inheritance

In one project pom.xml, we can add ***<parent>*** tag having other project info like ***<groupId>***, ***<artifactId>*** and ***<version>*** to get other project dependencies and plugins.

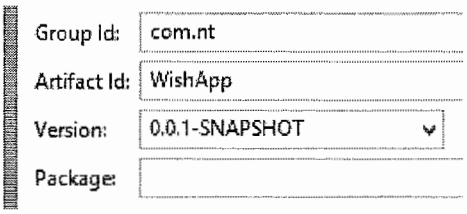
```
<parent>
<groupId>Naresh</groupId>
<artifactId>HBproj</artifactId>
<version>0.0.1-SNAPSHOT</version>
</parent>
```

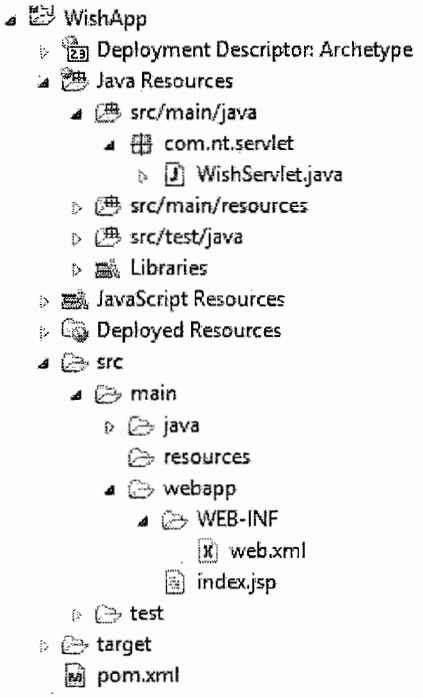
Example: In ***pom.xml*** of ***HBProj2***, add this code to get plugins and dependencies of ***HBProj1.***

Create any Sample Project (say ***HBProj1*** with certain dependencies) and add ***<parent>*** having dependency info of ***HBProj1*** in ***pom.xml*** of other sample project (***say HBproj1***).

#Creating Maven web project using Eclipse IDE

File → New → Maven Project → Next → choose ***maven-archetype-webapp*** and enter GroupId and ArtifactId as shown in the diagram. The default project directory structure as shown in the diagram is created.





➢ Add ***servlet-api*** related dependency tag in ***pom.xml*** and also change java version to 1.8 in java compiler, project facets through project properties.

➢ Create package ***com.nt.servlet*** under ***src/main/java*** and add ***WishServlet.java*** class. Copy and paste the same logic in WishServlet.java as we did previously, configure the servlet in ***web.xml*** and if you want you can even work with ***Tomcat7 Maven Plugin*** by adding its configuration in ***pom.xml***.

We can run the above application in the following ways.

➢ Run the application as normal ***Run on Server*** application by configuring tomcat with eclipse

- ***Run As maven build..*** goals=***clean package***
- Configure ***maven tomcat plugin*** in ***pom.xml*** as discussed earlier and use ***Run as maven build..*** goals=***tomcat7:run***

#Maven Multi-Module Project

Whatever project we created so far are single module projects. The concept of Multi-Module Project is very simple. When various modules are part of a project and closely related to each other, then the project should be structured as a multi module project. In a multi module project, Maven ensures that all sub modules are built in proper order before the main module build.

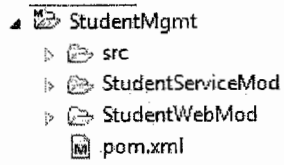
According to agile methodology, we take multiple layers as multiple modules in a project i.e. all DAO classes together in one module, all service classes together in one module and all web components together in one module. To develop such projects in maven, we can take maven multi-module project.

Example: ***StudentMgmt*** (Proof-Of-Concept example)

- We will split ***StudentMgmt***App in two modules, ***StudentServiceMod*** and ***StudentWebMod.***

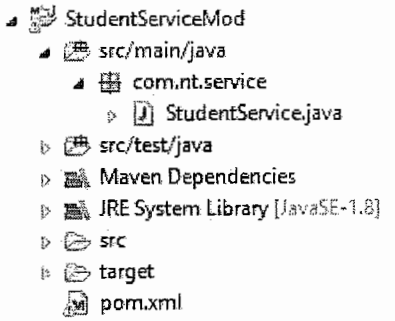
1. **Creating Maven Project by taking packing as pom**

Click on File → New →Maven Project → check Create simple project → Next Enter **GroupId:*Naresh*, ArtifactId: *StudentMgmt*** and **Packaging: *pom*** → Finish.



2. **Add maven module to *StudentMgmt* project having name *StudentServiceMod* (use quickstart archetype)**

Right click on ***StudentMgmt*** app→ New → Others→ Expand Maven dir. → Select Maven Module → Next→Module Name: ***StudentServiceMod***→ Next →select quickstart archetype →Next → Finish



3. **Add one service class in StudentMgmt project in src/main/java package**

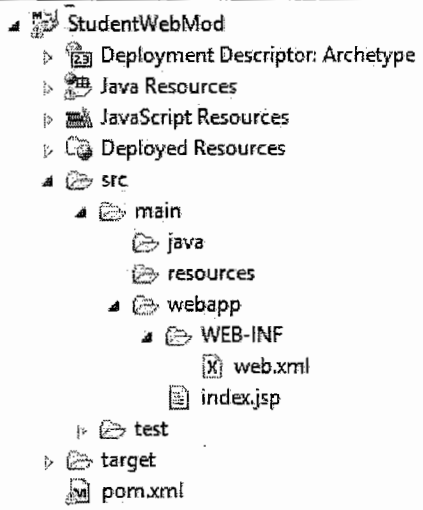
```
In StudentService
package com.nt.service;

publicclass StudentService {
      publicint getRank(int no){
            return 100;
      }//getRank(-)
}//class
```

4. **Add maven module to StudentMgmt App having name StudentWebMod** (use webapp archetype)

Right click on ***StudentMgmt*** app → New → Others → Expand Maven dir. → Select Maven Module → Next→Module Name: ***StudentWebMod***→ Next →select webapp archetype →Next → Finish

5. **Add <dependency> tag for servlet-api jar and also <dependency> tag for StudentServiceMod in pom.xml file of web module**




```
<dependency>
	<groupId>com.nit</groupId>
	<artifactId>StudentServiceMod</artifactId>
	<version>0.0.1-SNAPSHOT</version>
</dependency>
```

6. **Add following code in index.jsp to use StudentServiceMod**

```
<%@page import="com.nt.service.*" %>

<%
  StudentService service=new StudentService();
  out.println("Student Rank::"+service.getRank(101));
%>
```

➢ Right click on **StudentMgmt** App→Run As →Maven Build..→Enter goals: ***clean package***

General FAQs on Maven

How to exclude the conflicting jars while we are working with multiple technologies related jar in *pom.xml* in a simple way??

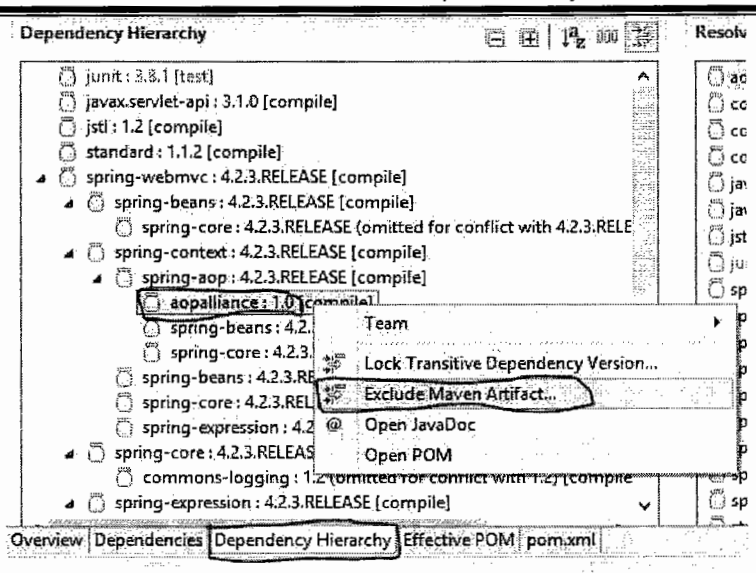
(Here I assume, we have already added jars in pom.xml)

1. **One way is to manually write the <exclusion> tag. Following is the example**

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-context</artifactId>
	<version>4.2.3</version>
<exclusions>
	<!-- Exclude Commons Logging in favor of SLF4j --
>
	<exclusion>
		<groupId>commons-logging</groupId>
		<artifactId>commons-logging</artifactId>
	</exclusion>
</exclusions>
```

2. Second way is to exclude it with the help of Dependency Hierarchy

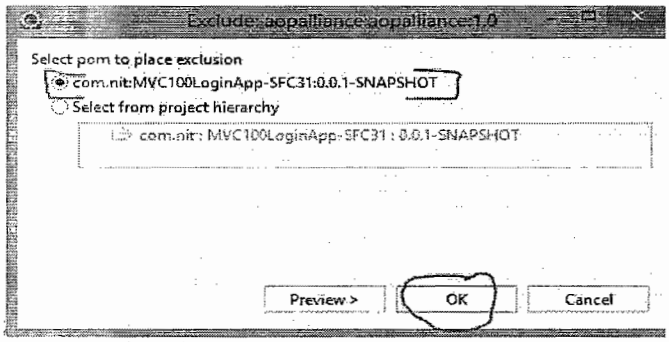
- Open the ***Project Object Model***(pom.xml) of your application
- Click the Dependency Hierarchy tab



It will show the list of jars which are dependent on each other. Now we are having a situation to exclude the conflicting jar from the main jar. (For this, we have to use <exclusions> and <exclusion> tags in pom.xml. But it can be done in an easy way through Dependency Hierarchy).

- Right Click the conflicting jar and select ***Exclude Maven Artifact***and click ok.

For eg: In the given dependency hierarchy suppose I don't need aopalliance jar, I am excluding it. See the image below.




```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.2.3.RELEASE</version>
    <exclusions>
        <exclusion>
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

What is the use of <properties> tag in pom.xml?

When we are working with multiple jars of same technologies, for each corresponding jar we write the <dependency> tag specifying the version. Sometimes what happens, when have to change the jar's version, for which we do the changes throughout the <dependency> tag of that technology. So, instead of doing the changes at multiple places, we can do the change at one place and it will be reflected for all corresponding jars. For the sake of this we take the <properties> tag where we take separate tag identifying the version of specific technology. See the following sample pom.

Add this configuration before <dependencies> tag

```
<properties>
        <org.springframework-version>3.1.1.RELEASE</org.springframework-version>
        <org.aspectj-version>1.6.10</org.aspectj-version>
        <org.slf4j-version>1.6.6</org.slf4j-version>
</properties>
```

We will use the above mentioned tags in the following ways.

```
<dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>${org.aspectj-version}</version>
</dependency>
```

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

What are the steps involved in project deployment?

Normally a deployment process consists of following steps –

- Check-in the code from all projects in progress into the SVN or source code repository and tag it.
- Download the complete source code from SVN.
- Build the application.
- Store the build output either WAR or EAR file to a common network location.
- Get the file from network and deploy the file to the production site.
- Updated the documentation with date and updated version number of the application.

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.

How to configure remote repository in maven pom.xml?

- By using ***<repositories>*** tag in pom.xml before or after <dependencies> tag. For example: ***inject jar*** related dependency in not available in central repository but it is available in repository maintained by Redhat.

```
<repositories>
<repository>
<id>redhat</id>
<url>https://maven.repository.redhat.com/ga/</url>
</repository>
</repositories>
```

```
<dependency>
<groupId>javax.inject</groupId>
<artifactId>javax.inject</artifactId>
<version>1.0.0.redhat-6</version>
</dependency>
```

Life Cycle of Maven

Maven has 3 Life Cycle.Each Life Cycle contains number of Phases

- Clean
- default
- site

Clean:-This Life Cycle contains 3 phases

Pre-clean	Executes processes needed prior to the actual project cleaning
Clean	Remove all the file generated in previous build
Post-clean	Executes processes needed to finalize the project cleaning

Default: This Life Cycle contains 23 phases

1	validate	validate the project is correct and all necessary information is available.
2	initialize	initialize build state, e.g. set properties or create directories.
3	generate-sources	generate any source code for inclusion in compilation.
4	process-sources	process the source code, for example to filter any values.
5	generate-resources	generate resources for inclusion in the package.
6	process-resources	copy and process the resources into the destination directory, ready for packaging.
7	compile	compile the source code of the project.

8	process-classes	post-process the generated files from compilation,for example to do bytecode enhancement on Java classes.
9	generate-test-sources	generate any test source code for inclusion in compile
10	process-test-sources	process the test source code, for example to filter any values.
11	generate-test-resources	create resources for testing.
12	process-test-resources	copy and process the resources into the test destination directory.
13	test-compile	compile the test source code into the test destination directory
14	process-test-classes	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. For Maven 2.0.5 and above.
15	test	run tests using a suitable unit testing framework.These tests should not require the code be packaged or deployed.
16	prepare-package	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above)
17	package	take the compiled code and package it in its distributable format, such as a JAR.
18	pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
19	integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
20	post-integration-test	perform actions required after integration tests have been executed. This may including cleaning up the environment.
21	verify	run any checks to verify the package is valid and meets quality criteria.
22	install	install the package into the local repository, for use as a dependency in other projects locally.
23	deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Site: It contains 4 phases

Pre-site	executes processes needed prior to the actual project site generation
Site	generates the project's site documentation
Post-site	executes processes needed to finalize the site generation, and to prepare for site deployment
Site-deploy	deploys the generated site documentation to the specified web server

Below are some of the List of Important Plugins and goals

Plugin Name	Goals Available	Purpose

maven-compiler-plugin	compile,testCompile	Compile java file and testCase file
exec-maven-plugin	exec,java	Compiles and executes java classes
maven-antrun-plugin	run	Execute ant tasks
tomcat-maven-plugin	Start,stop,deploy, redeploy,undeploy	Deploys the project into tomcat server
jboss-maven-plugin	--do--	Deploys the project into jboss

Working with webapp-jee5 archetype

In order to work with webapp-jee5 archetype, first of all we need to download this archetype in our local system from maven central repository. Following are the steps given for this.

File → New → Maven Project→ Next→ Configure→Add Remote Catalog→ In Catalog File: https://repo1.maven.org/maven2/archetype-catalog.xml, In Description: RemoteCatalog→Verify(wait till all archetypes are downloaded)→OK→OK→(make sure in Catalog, RemoteCatalog is choosen) Now filter for webapp-jee5 archetype and choose it→Next→Enter groupId,artifactId and create your project.

#Maven Interview Questions and Answers

1. **Explain what is Maven? How does it work?**

➢ Maven is a project management tool. It provides the developer a complete build lifecycle framework. On executing Maven commands, it will look for POM file in Maven; it will run the command on the resources described in the POM.

2. **List out what are the aspects does Maven Manages?**

➢ Maven handles following activities of a developer

- Build
- Documentation
- Reporting
- Dependencies
- SCMs
- Releases
- Distribution
- Mailing list

3. **Mention the three build lifecycle of Maven?**

- Clean: Cleans up artifacts that are created by prior builds
- Default (build): Used to create the application
- Site: For the project generates site documentation

4. **Explain what is POM?**

➢ In Maven, POM (Project Object Model) is the fundamental unit of work. It is an XML file which holds the information about the project and configuration details used to build a project by Maven.

5. **Explain what is Maven artifact?**

➢ Usually an artifact is a JAR file which gets arrayed to a Maven repository. One or more artifacts a maven build produces such as compiled JAR and a sources JAR. Each artifact includes a group ID, an artifact ID and a version string.

6. Explain what is Maven Repository? What are their types?

➢ A Maven repository is a location where all the project jars, library jars, plugins or any other particular project related artifacts are stored and can be easily used by Maven. Their types are local, central and remote.

7. Why Maven Plugins are used?

➢ Maven plugins are used to

- Create a jar file
- Create war file
- Compile code files
- Unit testing of code
- Documenting projects
- Reporting

8. List out the dependency scope in Maven?

➢ The various dependency scope used in Maven are:

- ***Compile:*** It is the default scope, and it indicates what dependency is available in the classpath of the project
- ***Provided:*** It indicates that the dependency is provided by JDK or web server or container at runtime
- ***Runtime:*** This tells that the dependency is not needed for compilation but is required during execution
- ***Test:*** It says dependency is available only for the test compilation and execution phases
- ***System:*** It indicates you have to provide the system path
- ***Import:*** This indicates that the identified or specified POM should be replaced with the dependencies in that POM's section

9. Mention how profiles are specified in Maven?

➢ Profiles are specified in Maven by using a subset of the elements existing in the POM itself.

10. Explain how you can exclude dependency?

➢ By using the exclusion element, dependency can be excluded

11. Mention the difference between Apache Ant and Maven?

➢ Apache Ant Maven

- Ant is a toolbox – Maven is a framework
- Ant does not have formal conventions like project directory structure – Maven has conventions
- Ant is procedural; you have to tell to compile, copy and compress – Maven is declarative (information on what to make & how to build)
- Ant does not have lifecycle; you have to add sequence of tasks manually – Maven has a lifecycle
- Ant scripts are not reusable – Maven plugins are reusable

12. In Maven what are the two setting files called and what are their location?

➢ In Maven, the setting files are called settings.xml, and the two setting files are located at

- Maven installation directory: $M2_Home/conf/settings.xml
- User's home directory: ${ user.home }/ .m2 / settings.xml

13. List out what are the build phases in Maven?

- Build phases in Maven are

  - Validate
  - Compile
  - Test
  - Package
  - Install
  - Deploy

14. List out the build, source and test source directory for POM in Maven?

- Build = Target
- Source = src/main/java
- Test = src/main/test

15. Where do you find the class files when you compile a Maven project?

- You will find the class files ${basedir}/target/classes/.

16. Explain what would the "jar: jar" goal do?

- jar: jar will not recompile sources; it will imply just create a JAR from the target/classes directory considering that everything else has been done.

17. List out what are the Maven's order of inheritance?

- The maven's order of inheritance is

  - Parent Pom
  - Project Pom
  - Settings
  - CLI parameters

18. For POM what are the minimum required elements?

- The minimum required elements for POM are project root, modelVersion, groupID, artifactID and version.

19. Explain how you can produce execution debug output or error messages?

- To produce execution debug output you could call Maven with X parameter or e parameter.

20. Explain how to run test classes in Maven?

- To run test classes in Maven, you need surefire plugin, check and configure your settings in setting.xml and pom.xml for a property named "test."

21. How can I change the default location of the generated jar when I command "mvn package"?

- By default, the location of the generated jar is in ${project.build.directory} or in your target directory. We can change this by configuring the outputDirectory of maven-jar-plugin.

22. **What is a Mojo?**

- A mojo is a Maven plain Old Java Object. Each mojo is an executable goal in Maven, and a plugin is a distribution of one or more related mojos.

23. **How does Maven looks for a dependency or resource?**

- It refers to the settings.xml to look for the repositories to look for the resource. First It looks into the configured local repository, then it looks into the configured Remote repositories. If the

resource is still not found, it looks it within maven repository central i.e repo1.maven.org. If its still not found, it throws the exception saying "Unable to find resource in repository central".

24. **What would you do if you have to add a jar to the project using Maven ?**

If its already there in Maven local repository, We can add that as a dependency in the project pom file with its Group Id, Artifact Id and version.
We can provide additional attribute SystemPath if its unable to locate the jar in the local repository.
If its not there in the local repository, we can install it first in the local repository and then can add it as dependency.

25. **Have you ever had problem getting your projects in eclipse refreshed after you made changes in the Pom files ?**

- Yes, It happens many times but I would usually perform mvn eclipse:eclipse and this would resolve the project refresh problems.

26. **What is the difference between compile and install ?**

- **Compile** compiles the source code of the project ***whereas*** **Install** installs the package into the local repository, for use as a dependency in other projects locally.

27. **How can we see Dependencies for the project and where exactly they are defined ?**
Using ***mvn dependency:tree***

28. **What is a transitive dependency? Can we override Transitive Dependency version and If Yes, how?**

- Transitive dependency is the dependencies not defined directly in the current POM but the POM of the dependent projects. Transitive dependencies allows to avoid specifying the libraries that are required by the project which are specified in other dependent projects - Remote or Local.

- Yes we can override transitive dependency version by specifying the dependency in the current POM.

29. **What is a cyclic dependency ?**

- A has dependency of B, B has dependency of C and C has dependency of A.

- With Maven 2 , came transitive dependency wherein in above scenario, C will acts as a dependency of A as if this dependency has been defined directly in A but the negative side is that if it leads to cyclic dependency , it creates problems.

30. **What is a project's fully qualified artifact name?**

a. <groupId>:<artifactId>:<version>
b. <groupId>:<artifactId>
c. <artifactId>:<groupId>:<version>
d. <artifactId>:<version>

Ans. ***<groupId>:<artifactId>:<version>***

31. What does the "You cannot have two plugin executions with the same (or missing) elements" message mean?

- It means that you have executed a plugin multiple times with the same <id>. Provide each <execution> with a unique <id> then it would be ok.

32. What means SNAPSHOT in Maven?

- SNAPSHOT is a type of version that indicates a current deployment copy. Maven checks during each build for a new SNAPSHOT version in the remote repository.

33. What is the difference between version and SNAPSHOT ?

- Maven will download always the specified version. In case of SNAPSHOT Maven wil download the latest SNAPSHOT.