MAVEN

COURSE MATERIAL

BY

NAGOOR BABU

Index

MAVEN

Introduction

Maven is a "Yiddish"[German language] word meaning "Accumulator Of Knowledge".

Maven was originally designed to simplify building processes in Jakarta Turbine project. There were several projects and each project contained slightly different ANT build files. JARs were checked into CVS.

Apache group then developed Maven which can build multiple projects together, publish projects information, deploy projects, share JARs across several projects and help in collaboration of teams

MAVEN is a "Project Management Framework", it is much more than a simple Build tool, its declarative and standard approach simplifies many aspects of the Project Lifecycle.

The main Objective of MAVEN is

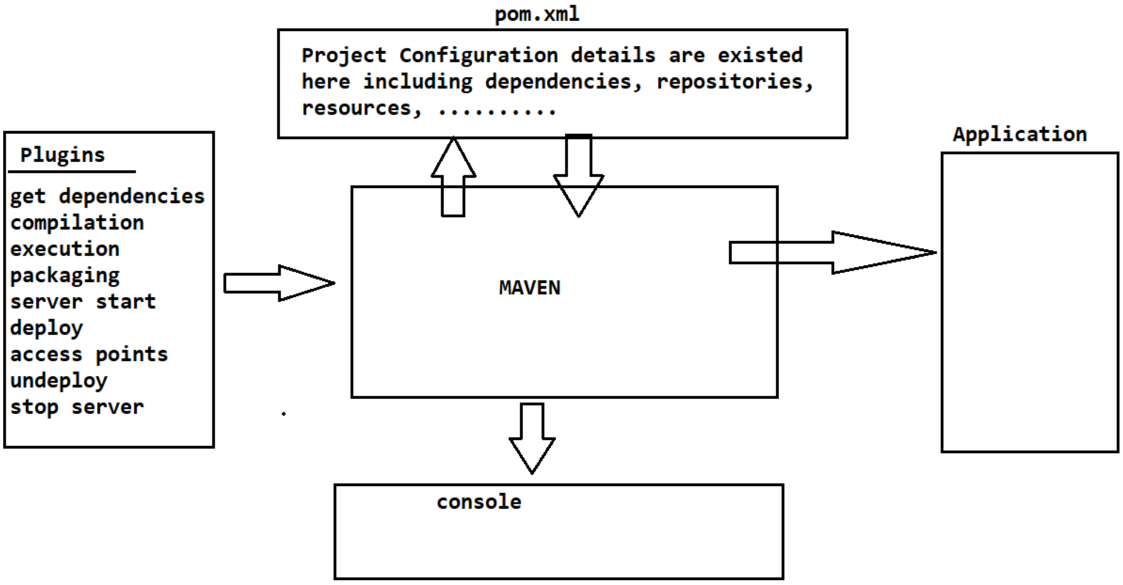
- A comprehensive model for projects, which is reusable, maintainable, and easier to comprehend [Understand].
- Plugins or tools that interact with this declarative model.

MAVEN follows "Convention over Configuration" Principle, which means that developers are not required to create build process themselves, Developers do not have to mention each and every configuration detail. Maven provides sensible default behavior for projects.

MAVEN does the following activities of the project lifecycle automatically.

- Provides default Project Structer
- Download Required Dependencies [Jars files]
- Compiles Source code
- Packaging projects as .jar, .war, .ear,....
- Starts Server
- Deploying Projects into Servers.
- Perform Unit Testing
- Preparing Test Reports.
- Preparing Documentations

- ➢ Undeploy applications from Servers
- ➢ Stops Server.



Project Object Model [pom.xml file]

i. POM Stands for Project Object Model.
ii. POM is the fundamental unit in Maven.
iii. POM is an XML file that contains information about the project and configuration details used by Maven to build the project.
iv. POM contains default values for projects ike build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/test/java; and so on.
v. In MAVEN1 , name of pom file is "project.xml", in MAVEN2 it was renamed to pom.xml.
vi. When we execute MAVEN project then MAVEN will look for the project configurations in pom.xml file and gets the needed things and executes the project.

In Building MAVEN Projects, pom.xml file contains the following configurations.

1. Project Description
2. Repository
3. Dependency Management
4. Project Inheritance
5. Build Configuration
6. Build Profiles

1) Project Description:

In pom file, initial we will identify "Projection Description", it contains Project name, version number, packaged type,.....

To specify the above details, we need to use the following XML tags.

```
<project ..... >
  <!-- Project Description -->
  <modelVersion> --- </modelVersion>
  <groupid> --- </groupid>
  <artifactid> -- </artifactid>
  <version> -- </version>
  -----
  -----
</project>
```

- Where "<project>" tag is root tag in pom.xml file
- Where "<modelVersion>" tag declared which version of the MAVEN we are using.<modelVersion> tag will take 4.0.0 to support for MAVEN2.x/3.x versions.
- Where "<groupid>" tag will take an unique ID for an organization, or a project.Normally we use a group ID similar to the root Java package name of the project.
- Where "<artifactid>" tag will take name of the project.The artifact ID is used as name for a sub directory under the group ID directory in the Maven repository and as part of the name of the JAR file produced when building the project.The build result, a JAR, WAR or EAR file, is called an artifact in Maven.
- Where "<versin>" tag will take Project version number.
- Where "<packaging>" tag will take different packaging formats inorder to delivery the project like jar, war, ear,...

EX:

```
<project ..... >
  <!-- Project Description -->
  <modelVersion>4.0.0</modelVersion>
  <groupid>com.durgasoft.banking</groupid>
  <artifactid>icici.accounts</artifactid>
  <version>1.0</version>
  <name>Account Application</name>
  <description> Application for Accounts module in icici Bank project </description>
  -----
</project>
```

2) Repository:

If we use Dependencies in MAVEN Project then MAVEN will search for the dependent JARs in Repositories.
MAVEN will use three types of Repositories in order to get dependencies.

1. Local Repository:

It is a location to manage and supply all dependencies, it will be created by MAVEN when we execute any MAVEN command first time.

In general, MAVEN will create Local Repository at "C:/Users/User_Name/.m2/repository"
EX: C:\Users\LENOVO\.m2\repository

2. Central Repository:

It is a default Repository for MAVEN, it is located at "http://repo1.maven.org/maven2".
IN MAVEN applications, we will use some other repositories are also explicitly like.

1. http://repository.jboss.org/nexus/content/groups/public
2. http://mvnrepository.com

In MAVEN applications, if we want to use the above explict repositories then we have to configure them in pom file by using the following xml tags.

```xml
<repositories>
  <repository>
    <id>jboss</id>
    <name>jboss repo</name>
    <url>http://repository.jboss.org/nexus/content/groups/public/</url>
  </repository>
</repositories>
```

3. Remote Repository:

In some Situations, Maven does not find the dependencies in Local Repository and in central repository, in this context, MAVEN stops the build process and generates some Exceptions. To overcome this problems, Maven has provided a new Features like "Remote Repository".

Remote Repository is a developer's own custom repository containing required libraries or other project jars.

To configure Remote Repository, we have to use the following XML tags in pom.xml file.

```
<repositories>
    <repository>
      <id>durgasoft.lib</id>
      <url>http://library.durgasoft.com/maven2/lib</url>
    </repository>
</repositories>
```

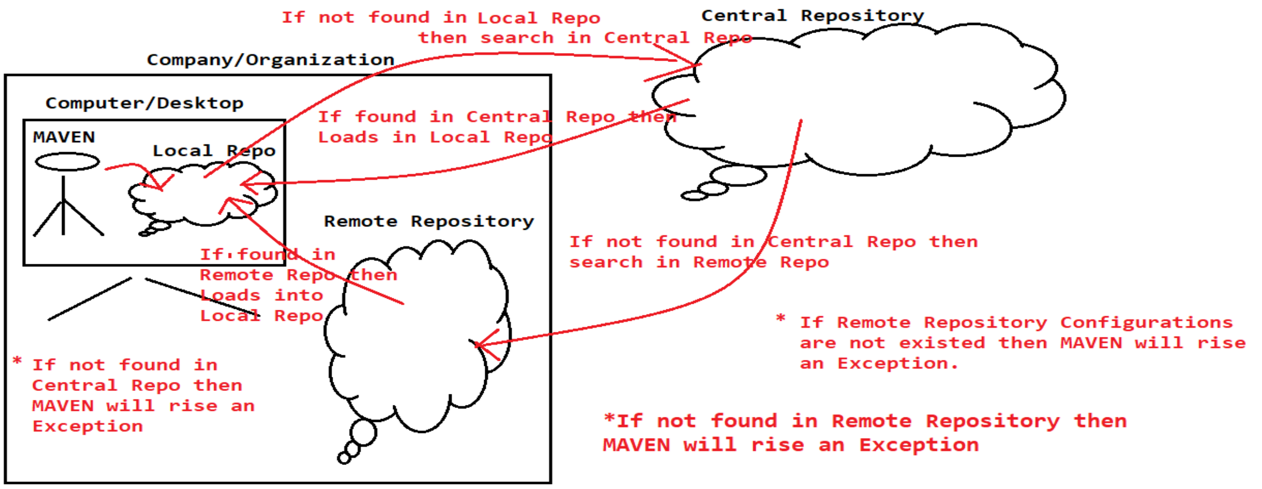
When we run MAVEN project then MAVEN will search for the dependencies in the following order.

1) First, MAVEN will search for the dependencies in local repository, if the required dependencies are available at Local Repository the MAVEN will use them in application. If the dependencies are not available at Local Repository then MAVEN search for them at Central Repository.

2) If the required Dependencies are existed in central repository then MAVEN will load them into Local Repository and MAVEN will use them in the applications. If the required dependencies are not existed in Central Repository then MAVEN will search for them in Remote Repositories as per configuration.

3) If Remote Repository is not configured then MAVEN will stop the application execution and generated some Exceptions.

4) If Remote Repository is configured then MAVEN will search for the required dependencies in Remote Repository, if they are identified then MAVEN will load them into Local Repository for futur reference. If the dependencies are not existed at Remote Repositories then MAVEN will stop the execution and generate some Exceptions.



If not found in Local Repo then search in Central Repo

If found in Central Repo then Loads in Local Repo

If found in Remote Repo then Loads into Local Repo

If not found in Central Repo then search in Remote Repo

* If not found in Central Repo then MAVEN will rise an Exception

* If Remote Repository Configurations are not existed then MAVEN will rise an Exception.

*If not found in Remote Repository then MAVEN will rise an Exception

3) Dependency Management:

In Applications, Dependencies are the libraries[Collection of JARs] which are required to compile, test and run our applications.

In General, in application development, we will download the required libraries from internet and we will store them in application directory structer.

The main Advantage of MAVEN in applications development is that not to store any Dependent JAR files in Project Directory Structer by downloading them explicitly, MAVEN has given flexibility to the developers like to specify dependent JAR files names in pom file, where MAVEN will search for them in the repositories and MAVEN will load them into the project directory structer automatically.

If we need any Library in MAVEN based applications then we have to declare them in pom file like below.

```
‹dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.5.6-Final</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

If we provide the dependency like above then MAVEN will search for the hibernate library with the name like
http://repo1.maven.org/maven2/org/hibernate/hibernate-core/3.5.6-Final/

MAVEN is following "Transitive Dependencies Mechanism", that is, if our dependencies are required any other libraries then MAVEN will get them automatically without loading them explicitly by the developers.

Dependency Scopes:

In Applications, some dependencies are required to all phases of the project lifecycle like compile, test, run,... and some other required only some of phases of the project lifecycle.

In order to limit the dependencies for the lifecycle phases we will use Dependency Scopes.

There are 6 scopes available in MAVEN

1. Compile
2. Provided
3. Runtime
4. Test
5. System
6. Import

1. Compile:

It is the default scope in MAVEN . This scope will make the dependencies to avail all phases like compile, test, run,....

EX:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.5.6-Final</version>
  <scope>compile</scope>
</dependency>
```

Note: In general, hibernate-core library is required for all phases of the application.

2. Provided:

This scope will make the dependency libraries to avail upto compilation and and upto testinbg, not for runtime, because, at runtime, JDKs or Containers will provide the required dependencies at runtime.

EX:

In web applications, Servlet API is required expliclty to compile and test the project, but, Servlet API is provided by the container at runtime automatically, so that, they are not required to be exported at runtime.

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
```

3. Runtime:

This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test class paths, but not the compile class path.

EX:

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.4</version>
    <scope>runtime</scope>
</dependency>
```

4. Test:

This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive.

EX:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

5. System:

Dependencies with system are similar to ones with scope provided. The only difference is system dependencies are not retrieved from remote repository. They are present under project's subdirectory and are referred from there.

EX:

```
<dependency>
  <groupId>Explicit_Dependency</groupId>
  <artifactId>Explicit_Dependency</artifactId>
  <scope>system</scope>
  <version>1.0</version>
  <systemPath>apps\app.war\WEB-INF\lib\Explicit_Dependency.jar</systemPath>
</dependency>
```

6. Import:

It is available in Maven 2.0.9 or later.

Import scope is only supported on a dependency of type pom in the dependencyManagement section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM's dependencyManagement section.

EX:

```
<dependencyManagement>
    <dependencies>
        <dependency>
```

```
            <groupId>other.pom.group.id</groupId>
            <artifactId>other-pom-artifact-id</artifactId>
            <version>SNAPSHOT</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

4) Project Inheritance:

In MAVEN based applications, it is possible to inherit configurations from one pom file to another pom file inorder to avoid configurations redundency.

To declare parent pom , we have to use "pom" as value to <packaging> tag in parent pom file.

EX:

```
<project ... >
 <modelVersion>4.0.0</modelVersion>
  <groupId>com.durgasdoft</groupId>
  <artifactId>my-parent</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  ---
</project>
```

If we want to inherit parent pom configuration details into a particular chaild pom then we have to configure parent pom in chaild pom.
EX:

```
<project  ..... >
  -----
   <parent>
       <groupId>com.durgasoft</groupId>
       <artifactId>my-parent</artifactId>
       <version>0.0.1-SNAPSHOT</version>
```

```
    </parent>
   ------
</project>
```

Note: In JAVA, java.lang.Object class is common and default super class for every java class inorder to provide 11 common and default methods to each and every java class, similarily , there is a common and default super pom file is existed in maven inorder to provide all common configurations and settings to the chaild pom file.

In general, parent pom contains the following configurations

1. Common data – Developers’ names, SCM address, distribution management etc.
2. Constants – Such as version numbers
3. Common dependencies – Common to all child. It has same effect as writing them several times in individual pom files.
4. Properties – For example plugins, declarations, executions and IDs.
5. Configurations
6. Resources

Note: By default, Maven looks for the parent POM first at project’s root, then the local repository, and lastly in the remote repository. If parent POM file is not located in any other place, then you can use code tag. This relative path shall be relative to project root.

EX:

```
<parent>
      <groupId>com.durgasoft</groupId>
      <artifactId>MavenExamples</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <relativePath>../baseapp/pom.xml</relativePath>
</parent>
```

Note: If we want to get super pom from MAVEN then use the following command on command prompt from the project root location which contains project specific pom file.

```
mvn help:effective-pom
```

Note: Where effective-pm is super pom configurations and project configurations.

5) Build Configuration:

In MAVEN , Build Configuration is mainly for plugin configurations, resources configurations,.....which are required in MAVEN Project.

MAVEN is simply the collection of plugins, where plugins are used to perform the actions like creating jar files, creating war files, compile Source code, executing unit test code, create project documentation, ...

MAVEN is having "Plugin Execution Framework" at its heart inorder to execute all plugins.

In MAVEN , there are two types of Plugins.

1. Build Plugins
2. Reporting Plugins

1.Build Plugins: These plugins are executed during the build and they should be configured in the <build/> element from the POM.

EX

1.Clean : It is used when you want to remove files generated at build-time in a project's directory.
2.Compiler: Compiles Java source code.
3.Deploy: It can be used to store artifacts in remote repository while deploying the applications in order to share to other projects .
4.Install: It can be used to install artifacts into local repository.
5.Resources: It will include all the project resources in output directory while creating JAR files.
6.Ear: create ear file from the current project.
7.jar: creates jar file from the current project.
8.war: creates war file from the current project.
9.rar: creates rar file from the current project.

2. Reporting plugins: These plugins are executed during the site generation and they should be configured in the <reporting/> element from the POM.

EX:
1.changelog: Generate a list of recent changes from your SCM[Software Configuration Management].
2.changes: Generate a report from an issue tracker or a change document.
3.javadoc: Generate Javadoc for the project.
4.project-info-resports: Generate standard project reports.
5.surfire-report: Generate a report based on the results of unit tests.

IN general, we will use MAVEN compiler plugin inorder to perform Compilation, for this we have to use the following xml tags in pom.xml file.

```
<project ---- >
   <build>
       <plugins>
          <plugin>
             <groupId>org.apache.maven.plugins</groupId>
             <artifactId>maven-compiler-plugin</artifactId>
             <configuration>
               <source>1.8</source>
               <target>1.8</target>
             </configuration>
          </plugin>
       </plugins>
   </build>
</project>
```

By default, all files placed in "src\main\config" are packaged into the generated project artifact and any file which we placed in "src\test\resources" are available in project classpath during unit tests.

If we want to provide our own customized resources location in project then we have to configure them in pom.xml file under <build> tag like below.

```
<build>
  ----
  <resources>
     <resource>
        <directory>src/main/config</directory</directory>
     </resource>
     <resource>
        <directory>src/main/resources</directory</directory>
     </resource>
     ----
  </resources>
  ----
</build>
```

6) Build Profiles:

IN general, profiles are used to customize the build lifecycle for different environments like development, testing, production,.....

Example:

```xml
<profiles>
  <profile>
    <id>development</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
       <jdbc.connection.url>jdbc:oracle:thin:@localhost:1521:xe</jdbc.connection.url>
    </properties>
  </profile>
  <profile>
    <id>test</id>
    <properties>
       <jdbc.connection.url>jdbc:mysql://localhost:3306/durgadb</jdbc.connection.url>
    </properties>
  </profile>
</profiles>
```

Where each and every profile has its own id, it can be used to access the respective environment or profile.

In src/main/resources/db.properties

jdbc.connection.url = ${jdbc.connection.url}

If we provide the above setups like above then at compilation time, the respective jdbc URL will be injected to the "jdbc.connection.url" property depending on the target environment.

Use the following command on command prompt inorder to compile the project.

C:/apps>mvn compile
Here "jdbc.connection.profile" property will take "jdbc:oracle:thin:@localhost:1521:xe" value.

```
C:/apps>mvn compile -Ptest
```

Here "jdbc.connection.profile" property will take "jdbc:mysql://localhost:3306/durgadb" value.

```
jdbc.connection.url = ${jdbc.connection.url}
```

If we provide the above setups like above then at compilation time, the respective jdbc URL will be injected to the "jdbc.connection.url" property depending on the target environment.

Use the following command on command prompt inorder to compile the project.

```
C:/apps>mvn compile
```

Here "jdbc.connection.profile" property will take "jdbc:oracle:thin:@localhost:1521:xe" value.

```
C:/apps>mvn compile -Ptest
```

Here "jdbc.connection.profile" property will take "jdbc:mysql://localhost:3306/durgadb" value.

Archetypes in MAVEN

MAVEN is providing Standard directory structers to prepare projects depending on the ARCHETYPE selection.

To create sample MAVEN standalone application directory structer we will use the following command on command prompt .

```
C:\mvnapps>mvn archetype:create -DgroupId=com.durgasoft -DartifactId=sampleapp
-Dpackagename=com.durgasoft.bankapp
```

Steps to Create First Project in MAVEN In Standalone:

- Install MAVEN Software.
- Create MAVEN Project
- Write Java Code
- Compile Java Code
- Execute Java Application

1. Install MAVEN Software:

Installation Process:

1. download apache-maven-3.5.4.zip file from internet[https://maven.apache.org/download.cgi]
2. Unzip apache-maven-3.5.4.zip file under C drive and we will get apache-maven-3.5.4 folder.
3. Set the following Environment Variables in System.
   a) JAVA_HOME: C:/Java/jdk1.8.0;
   b) path: C:/Java/jdk1.8.0/bin;C:\apache-maven-3.5.4\bin;
   c) M2_HOME: C:/apache-maven-3.5.4
   d) MAVEN_HOME: C:/apache-maven-3.5.4 [Optional]
4. Test MAVEN Installation:
   Open Command prompt and use the following command.

```
  C:\apache-maven-3.5.4\bin>mvn --version
   Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-
18T00:03:14+05:30)
   Maven home: C:\apache-maven-3.5.4\bin\..
   Java version: 1.8.0, vendor: Oracle Corporation, runtime: C:\Java\jdk1.8.0\jre
   Default locale: en_US, platform encoding: Cp1252
   OS name: "windows 8.1", version: "6.3", arch: "x86", family: "windows"
```

2. Create MAVEN Project:

a) Create a seperate folder for MAVEN applications
   E:/mvn_projects
b) Use the following command on Command prompt.
   E:/mvn_projects>mvn archetype:generate
c) Provide archetype Number: 1202[for maven-archetype-quickstart application]
d) Provide archetype version number : 7
e) Provide "groupId" value: com.durgasoft
f) Provide "archetype" value : sampleapp
g) Provide "version" number : 1.0 or 1.0-SNAPSHOT
h) Provide "package" name : com.durgasoft
i) Provide "Y" to confirm all the above details

If we do the above steps then MAVEN will create quick start project with the following directories and files.

E:/mvn_projects

Sampleapp

```
|--src
|   |---main
|   |    |----java
|   |           |---com
|   |                |---durgasoft
|   |                        |----App.java
|   |---test
|        |----java
|               |---com
|                    |----durgasoft
|                           |-----AppTest.java
|---pom.xml
```

3. Write Java Code:

App.java

```
package com.durgasoft;

/**
 * Hello world!
```

```
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Welcome to Durgasoft" );
    }
}
```

AppTest.java

```
package com.durgasoft;

import static org.junit.Assert.assertTrue;

import org.junit.Test;

/**
 * Unit test for simple App.
 */
public class AppTest
{
    /**
     * Rigorous Test :-)
     */
    @Test
    public void shouldAnswerWithTrue()
    {
        System.out.println("Welcome To Durgasoft");
        assertTrue( true );
    }
}
```

4. Compile Java Code:

To Compile JAVA code, first, goto application folder then use the following command on command prompt.

```
E:\mvn_projects\sampleapp>mvn compile
```

If we use the above command then MAVEN will compile JAVA files and generates the .class files by creating "target" folder under "application folder".

5. Execute MAVEN Project:

IN Two ways we are able to execute MAVEN Project.

1. Execute Test case directly
2. Execute Main App by creating project.
3.

1) Execute Test case directly:

```
E:\mvn_projects\sampleapp>mvn test
```

2)Execute Mian App by creating project.

a) Create jar file for the project:
   E:\mvn_projects\sampleapp>mvn package
b) Set classpath environment variable to the generated jar file:
   E:\mvn_projects\sampleapp>set classpath=E:\mvn_projects\sampleapp\target\sampleapp-1.0.jar;
   Execute Main Class:
   E:\mvn_projects\sampleapp>java com.durgasoft.App
c) Welcome to Durgasoft

Steps to prepare MAVEN Project in Eclipse

1. Create Maven Project in Eclipse:
2. Provide the required configurations in pom.xml
3. Write Application Logic in AppTest.java file
4. Run Maven Project:

1. Create Maven Project in Eclipse:

1) Open Eclipse IDE
2) Right Click on "Project Explorer"
3) Select "New"
4) Select "Others"
5) Search for "Maven".
6) Select for "Maven Project".
7) Click on "Next" button.
8) Click on "Next" button.

9) Provide the following details.
   a) groupId: com.durgasoft
   b) artifactId: sampleapp
   c) version: 0.0.1-SNAPSHOT
   d) package: com.durgasoft

10) Click on "Finish" Button.

2. Provide the required configurations in pom.xml

1. Open pom.xml file File and provide java8 plugin configuration.

```
<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.durgasoft</groupId>
  <artifactId>helloapp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloapp</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
```

```xml
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.7.0</version>
            <configuration>
               <source>1.8</source>
               <target>1.8</target>
            </configuration>
         </plugin>
      </plugins>
   </build>
</project>
```

2. Save pom.xml file and see "Markers" for errors.
3. Right Click on :Project"
4. Select "Maven" .
5. Select "Update Project".
6. Click on "OK" button.

3.Write Application Logic in AppTest.java file

src/test/java/AppTest.java

```java
package com.durgasoft;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
public class AppTest extends TestCase
{
    public AppTest( String testName )
    {
        super( testName );
    }
    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }
    public void testApp()
    {
        System.out.println("Hello Maven!");
        assertTrue( true );
    }
```

```
}
```

Save the above program

4. Run Maven Project:

- ✓ Right Click on Project[Sampleapp].
- ✓ Select "Run As".
- ✓ Select "Maven Test" and see the Result in Console.

To prepare JDBC Application with MAVEN then we have to use the following steps with all the above steps.

1. Install ojdbc6.jar file in Local Repository.
2. Provide ojdbc6.jar file dependency in pom.xml

1. Install ojdbc6.jar file in Local Repository.

a. Keep ojdbc6.jar file in our working directory location.
E:/Mvn_Projects/ ojdbc6.jar

b. use "mvn install:install-file" command on command prompt to install jar file.

```
E:/Mvn_Projects> mvn install:install-file -DgroupId=oracle -DartifactId=oracle-jdbc
-Dpackaging=jar -Dversion=11.2.0-XE -DgeneratePom=true -Dfile=ojdbc6.jar
```

2. Provide ojdbc6.jar file dependency in pom.xml

```
<dependencies>
  <dependency>
      <groupId>oracle</groupId>
      <artifactId>oracle-jdbc</artifactId>
      <version>11.2.0-XE</version>
      <scope>runtime</scope>
   </dependency>
</dependencies>
```

EX:

```
├───src
│   ├───main
```

```
│   │   └───java
│   │       └───com
│   │           └───durgasoft
|   |                |----App.java
|   |                |----JdbcApp.java
│   └───test
│       └───java
│           └───com
│               └───durgasoft
|                    |---AppTest.java
|---pom.xml
```

pom.xml

```
<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.durgasoft</groupId>
  <artifactId>helloapp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloapp</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
       <groupId>oracle</groupId>
       <artifactId>oracle-jdbc</artifactId>
```

```
        <version>11.2.0-XE</version>
        <scope>runtime</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.7.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
   </build>
</project>
```

JdbcApp.java

```
package com.durgasoft;

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

public class JdbcApp {
    Connection con;
    Statement st;
    ResultSet rs;
    public JdbcApp() {
        try {
            Class.forName("oracle.jdbc.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "durga");
            st = con.createStatement();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
    public void displayEmpDetails() {
        try {
            rs = st.executeQuery("select * from emp1");
            System.out.println("ENO\tENAME\tESAL\tEADDR");
            System.out.println("------------------------------");
            while(rs.next()) {
                System.out.print(rs.getInt(1)+"\t");
                System.out.print(rs.getString(2)+"\t");
                System.out.print(rs.getFloat(3)+"\t");
                System.out.println(rs.getString(4));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

AppTest.java

```
package com.durgasoft;

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

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * Unit test for simple App.
 */
public class AppTest
    extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
```

```
     */
    public AppTest( String testName )
    {
        super( testName );
    }

    /**
     * @return the suite of tests being tested
     */
    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }

    /**
     * Rigourous Test :-)
     */
    public void testApp()
    {
        JdbcApp jdbcApp = new JdbcApp();
        jdbcApp.displayEmpDetails();
        assertTrue( true );
    }
}
```

If we want to use MySQL Database in Our project then we have to use the following dependency in pom file.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
```

In Java Application
Driver Class: com.mysql.cj.jdbc.Driver
Driver UIRL: jdbc:mysql://localhost:3306/durgadb
DB User Name: root
DB Password : root

Example:

pom.xml

```
<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.durgasoft</groupId>
  <artifactId>helloapp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloapp</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <!--
    <dependency>
        <groupId>oracle</groupId>
        <artifactId>oracle-jdbc</artifactId>
        <version>11.2.0-XE</version>
        <scope>runtime</scope>
    </dependency>
    -->

    <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
```

```
  </dependencies>
  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.7.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

JdbcApp.java

```
package com.durgasoft;

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

public class JdbcApp {
    Connection con;
    Statement st;
    ResultSet rs;
    public JdbcApp() {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection("jdbc:mysql://localhost:3300/durgadb", "root", "root");
            st = con.createStatement();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void displayEmpDetails() {
        try {
            rs = st.executeQuery("select * from emp1");
            System.out.println("ENO\tENAME\tESAL\tEADDR");
            System.out.println("------------------------------");
            while(rs.next()) {
                System.out.print(rs.getInt(1)+"\t");
                System.out.print(rs.getString(2)+"\t");
                System.out.print(rs.getFloat(3)+"\t");
                System.out.println(rs.getString(4));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

AppTest.java

```
package com.durgasoft;

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

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * Unit test for simple App.
 */
public class AppTest
    extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
    public AppTest( String testName )
    {
```

```
        super( testName );
    }

    /**
     * @return the suite of tests being tested
     */
    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }

    /**
     * Rigourous Test :-)
     */
    public void testApp()
    {
        System.out.println("Welcome To Maven");
        JdbcApp jdbcApp = new JdbcApp();
        jdbcApp.displayEmpDetails();
        /*
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("oracle.jdbc.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "durga");
            st = con.createStatement();
            rs = st.executeQuery("select * from emp1");
            System.out.println("ENO\tENAME\tESAL\tEADDR");
            System.out.println("------------------------------");
            while(rs.next()) {
                System.out.print(rs.getInt(1)+"\t");
                System.out.print(rs.getString(2)+"\t");
                System.out.print(rs.getFloat(3)+"\t");
                System.out.println(rs.getString(4));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }*/
        assertTrue( true );
    }
}
```

Web Application in Maven

Steps to Prepare Web Application in Maven:

1. Create Web Project in Maven
2. Provide Configurations in pom.xml file
3. Prepare Web Resources.
4. Update Project
5. Run Web Application

1. Create Web Project in Maven:

i. Click on File in "Eclipse".
ii. Click on "New".
iii. Click on "Others".
iv. Click on "Maven".
v. Select "Maven Project".
vi. Click on "Next".
vii. Click on "Next".
viii. Select "org.apache.maven.archetypes maven-archetype-webapp 1.0".
ix. Click on "Next" button.
x. Provide the following details
   a. Group Id: com.durgasoft
   b. Artifact Id: loginapp
   c. Version: 0.0.1-SNAPSHOT
   d. packag: com.durgasoft
xi. Click on "Finish" button.

2. Provide Configurations in pom.xml file:

1. Provide Compiler Plugin for JAVA8 version:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.7.0</version>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
```

```
     </configuration>
</plugin>
```

2. Provide Tomcat Plugin:

```
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
</plugin>
```

3 .Provide Servlet API Dependency:

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

4. Save pom.xml file.
5. Update Maven Project.

a. Right Click on Project.
b. Select "Maven".
c. Select "Update Project".
   or
a. Select "Markers" in Tools bar.
b. Select "Maven Problems".
c. Right Click on "Project configuration is not up-to-date with pom.xml".
d. Select "Quick Fix" .
e. Click on "Finish" button.

EX: pom.xml

```
<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/maven-v4_0_0.xsd">
```

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.durgasoft</groupId>
  <artifactId>loginapp</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>loginapp Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>loginapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```

3. Prepare Web Resources:

a) Prepare html file or jsp file:

1. select "src".
2. select "main".
3. Right Click on "webapps".
4. Select "New".
5. Select "Html File".
6. Provide File Name "loginform.html".
7. Click on "Next".
8. Click on "Finish".
9. Provide Html Code in loginform.html.
10. Save Html File.

EX: loginform.html

```html
<html>
<body>
<form action="./hello" method="post">
    User Name<input type="text" name="uname"/><br>
    Password<input type="password" name="upwd"/><br>
    <input type="submit" value="Login"/>
</form>
</body>
</html>

success.html
```

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h1>Durga Software Solutions</h1>
<h2>User Login Status</h2>
<font color="red" size="7">User Login Success</font>
<h3>
```

```
<a href="./loginform.html">|User Login Form|</a>
</h3>
</body>
</html>

failure.html
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h1>Durga Software Solutions</h1>
<h2>User Login Status</h2>
<font color="red" size="7">User Login Failure</font>
<h3>
<a href="./loginform.html">|User Login Form|</a>
</h3>
</body>
</html>
```

b) Prepare Servlets:

1) Right Click on Java Resources.
2) Select "New".
3) Select "Source Folder".
4) Provide the following details.
   a. Project Name : loginapp
   b. Folder Name: src/main/servlets
   c. Click on "Finish" Button.
5) Right Click on "src/main/servlets" folder.
6) Select "New".
7) Select "Servlet".
8) Provide the following details.
   a) Package Name: com.durgasoft.servlets
   b) Class Name : LoginServlet
9) Click on "Next" button.

10)Edit URL Pattern "/login".
11)Click on "Next" button.
12)Select "doPost" [bydefault Selected].
13)Click on "Finish" button.
14)Write Application Logic in Servlets.

EX: LoginServlet.java

```java
package com.durgasoft.servlets;

import java.io.IOException;

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

import com.durgasoft.service.UserService;
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        String uname = request.getParameter("uname");
        String upwd = request.getParameter("upwd");
        RequestDispatcher reqDispatcher = null;
        UserService userService = new UserService();
        String status = userService.checkLogin(uname, upwd);
        if(status.equals("success")) {
            reqDispatcher = request.getRequestDispatcher("success.html");
            reqDispatcher.forward(request, response);
        }else {
            reqDispatcher = request.getRequestDispatcher("failure.html");
            reqDispatcher.forward(request, response);
        }

    }

}
```

web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>LoginServlet</servlet-name>
    <display-name>LoginServlet</display-name>
    <description></description>
    <servlet-class>com.durgasoft.servlets.LoginServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>loginform.html</welcome-file>
  </welcome-file-list>
</web-app>
```

c)Prepare Service Class As per the requirement:

1) Right Click on "src/main/servlets"
2) Select "New".
3) Select "Class"
4) Provide the following details.
   i) Package Name : com.durgasoft.service
   ii) Class Name   : UserService
5) Select "Next" button
6) Provide Application Logic in UserService class

EX:UserService.java

```
package com.durgasoft.service;

```

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

public class UserService {
    Connection con;
    Statement st;
    ResultSet rs;
    String status = "";
    public UserService() {
        try {
            Class.forName("oracle.jdbc.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "durga");
            st = con.createStatement();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public String checkLogin(String uname, String upwd) {
        try {
            rs = st.executeQuery("select * from reg_Users where uname = '"+uname+"' and upwd = '"+upwd+"'");
            boolean b = rs.next();
            if(b == true) {
                status = "success";
            }else {
                status = "failure";
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return status;
    }
}
```

4. Update Project:

1. Right Click on Project.
2. Select "Maven".
3. Select "Update Project".

4. Select Project and Click on "OK" button.

5. Run Application:

1. Right Click on Project.
2. Select "Run As".
3. Select "Maven Build...".
4. Provide Value to "Goals" : tomcat:run -Dmaven.tomcat.port=9944[Any port number]
5. Click on "Run" button.
6. Copy the URL from Console.
7.Past the URL in Browser Addressbar.
   http://localhost:9944/loginapp/loginform.html
8. Provide data and click on "Login" button and get response from LoginServlet.

Spring Core Module Application in MAVEN

Steps to Prepare Spring Core Module Application in MAVEN:

1. Create Maven Quick Start Project.
2. Provide Configurations in pom.xml file.
3. Prepare Spring Resources like Beans, Configuration Files, Test and Cases...
4. Run Spring Application

1. Create Maven Quick Start Project.

1. Open Eclipse IDE
2. Right Click on "Project Explorer"
3. Select "New"
4. Select "Others"
5. Search for "Maven".
6. Select for "Maven Project".
7. Click on "Next" button.
8. Click on "Next" button.
9. Provide the following details.
    a. groupId: com.durgasoft
    b. artifactId: springapp1
    c. version: 0.0.1-SNAPSHOT
    d. package: com.durgasoft
10. Click on "Finish" Button.

2. Provide Configurations in pom.xml file.

1. Open pom.xml file File and provide java8 plugin configuration:

EX:

```
<build>
   <plugins>
       <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.7.0</version>
          <configuration>
```

```
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

2. Provide Spring dependencies in pom.xml file

EX:

```
<dependencies>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.3.18.RELEASE</version>
  </dependency>

  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>4.3.18.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>4.3.18.RELEASE</version>
  </dependency>
</dependencies>
```

Note: In the above dependencies, just add spring-core and spring-context dependencies in pom.xml file then automatically all other dependencies are also be loaded.

EX: pom.xml

```
<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.durgasoft</groupId>
  <artifactId>springapp1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>springapp</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

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

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.3.18.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>4.3.18.RELEASE</version>
    </dependency>

```

```
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>4.3.18.RELEASE</version>
    </dependency>
  </dependencies>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.7.0</version>
          <configuration>
            <source>1.8</source>
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

2. Save pom.xml file and see "Markers" for errors.
3. Right Click on :Project"
4. Select "Maven" .
5. Select "Update Project".
6. Click on "OK" button.

3. Prepare Spring Resources like Beans, Configuration Files, Test Cases...

a) Prepare Bean class:

1. Right Click on "src/java/main".
2. Select "New"
3. Select "Class".
4. Provide the following details
   Package: com.durgasoft.beans
   Class   : HelloBean
5 .Click on "Finish" button.
6. Provide properties and methods in Bean class.

EX: HelloBean.java

```java
package com.durgasoft.beans;

public class HelloBean {
    public String sayHello() {
        return "Hello User!";
    }
}
```

b) Prepare Spring Configuration File:

1. Right Click on "src\java\main".
2. Select "Package".
3. Provide package name "com.durgasoft.resources".
4. Click on "Finish" button.
5. Right Click on "com.durgasoft.resources" package.
6. Select "New".
7. Select "Other".
8. Search "XML" and select "XML File".
9. Click on "Next" button.
10. Provide file Name: applicationContext.xml
11. Click on "Next" button.
12. Click on "Finish" Button.

EX: applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="helloBean" class="com.durgasoft.beans.HelloBean"/>
</beans>
```

c) Prepare Test Application:

1. Right Click on "src/java/main".
2. Select "New"
3. Select "Class".
4. Provide the following details
   Package: com.durgasoft.test
   Class   : TestClient
5. Click on "Finish" button.
6. Provide application Logic in TestClient class.

EX:TestClient.java

```java
package com.durgasoft.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.durgasoft.beans.HelloBean;

public class TestClient {
    public void test() {
        ApplicationContext context = new ClassPathXmlApplicationContext("com/durgasoft/resources/applicationContext.xml");
        HelloBean bean = (HelloBean) context.getBean("helloBean");
        String message = bean.sayHello();
        System.out.println(message);
    }
}
```

d)Access TestClient Application from JUnit test[src\test\main\TestApp.java]:

EX:TestApp.java

```java
package com.durgasoft;

import com.durgasoft.test.TestClient;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
public class AppTest
```

```
    extends TestCase
{
    public AppTest( String testName )
    {
        super( testName );
    }

    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }

    public void testApp()
    {
        TestClient client = new TestClient();
        client.test();

        assertTrue( true );
    }
}
```

4. Run Spring Application:

1. Right Click on Project[springapp1].
2. Select "Run As".
3. Select "Maven Test" and see the Result in Console.