Java means DURGA SOFT..

JAVA TOOLS Material

Maven

MAVEN

CONTENT:

MAVEN (Build Tool)

Maven Overview:

Maven provides a comprehensive approach to managing software projects. From compilation, to distribution, to documentation, to team collaboration, Maven provides the necessary abstractions that encourage reuse and take much of the work out of project builds.

Maven:

Maven is a project management framework. Maven is a build tool or a scripting framework. Maven encompasses a set of build standards, an artifact repository model, and a software engine that manages and describes projects. It defines a standard life cycle for building, testing, and deploying project artifacts. It provides a framework that enables easy reuse of common build logic for all projects following Maven's standards. The Maven project at the Apache Software Foundation is an open source community which produces software tools that understand a common declarative Project Object Model (POM).

Maven 2, a framework that greatly simplifies the process of managing a software project.

Note: Maven is a declarative project management tool that decreases your overall time to market by effectively leveraging cross-project intelligence. It simultaneously reduces your duplication effort and leads to higher code quality.

Maven Origin's:

Maven was borne of the practical desire to make several projects at the Apache Software Foundation (ASF) work in the same, predictable way. Prior to Maven, every project at the ASF had a different approach to compilation, distribution, and Web site generation. The ASF was effectively a series of isolated islands of innovation. While there were some common themes across the separate builds, each community was creating its own build systems and there was no reuse of build logic across projects. The build process for Tomcat was different than the build process for Struts, and the Turbine developers had a different site generation process than the Jakarta Commons developers.

This lack of a common approach to building software meant that every new project tended to copy and paste another project's build system. Ultimately, this copy and paste approach to build reuse reached a critical tipping point at which the amount of work required to maintain the collection of build systems was distracting from the central task of developing high-quality software. In addition, for a project with a difficult build system, the barrier to entry was extremely high; projects such as Jakarta Taglibs had (and continue to have) a tough time attracting developer interest because it could take an hour to configure everything in just the right way. Instead of focusing on creating good component libraries or MVC frameworks, developers were building yet another build system.

Maven entered the scene by way of the Turbine project, and it immediately sparked interest as a sort of Rosetta Stone for software project management. Developers within the Turbine project could freely move between subcomponents, knowing clearly how they all worked just by understanding how one of the components worked. Once developers spent time learning how one project was built, they did not have to go through the process again when they moved on to the next project. Developers at the ASF stopped figuring out creative ways to compile, test, and package software, and instead, started focusing on component development.

The same standards extended to testing, generating documentation, generating metrics and reports, and deploying. If you followed the Maven Build Life Cycle, your project gained a build by default. Soon after the creation of Maven other projects, such as Jakarta Commons, the Codehaus community started to adopt Maven 1 as a foundation for project management.

Many people come to Maven familiar with Ant, so it's a natural association, but Maven is an entirely different creature from Ant. Maven is not just a build tool, and not necessarily a replacement for Ant.

Whereas Ant provides a toolbox for scripting builds, Maven provides standards and a set of patterns in order to facilitate project management through reusable, common build strategies.

Maven Provides:

Maven provides developers to manage the following: Builds, Documentation, Reporting, Dependencies, SCM's, Releases.

Maven provides a useful abstraction for building software in the same way an automobile provides an abstraction for driving. When you purchase a new car, the car provides a known interface; if you've learned how to drive a Jeep, you can easily drive a Camry. Maven takes a similar approach to
software projects: if you can build one Maven project you can build them all, and if you can apply a
testing plugin to one project, you can apply it to all projects. You describe your project using Maven's
model, and you gain access to expertise and best-practices of an entire industry.
Given the highly inter-dependent nature of projects in open source, Maven's ability to standardize locations for source files, documentation, and output, to provide a common layout for project documentation, and to retrieve project dependencies from a shared storage area makes the building
process much less time consuming, and much more transparent.
Maven provides you with:

- A comprehensive model for software projects
- Tools that interact with this declarative model

An individual Maven project's structure and contents are declared in a Project Object Model (POM),
which forms the basis of the entire Maven system.

Maven Principles:

According to Christopher Alexander "patterns help create a shared language for communicating insight and experience about problems and their solutions".
The following Maven principles were inspired by Christopher Alexander's idea of creating a shared language:

- Convention over configuration
- Declarative execution

- Reuse of build logic
- Coherent organization of dependencies.

Maven provides a shared language for software development projects.

Maven provides a structured build life cycle so that problems can be approached in terms of this structure.

Each of the principles above enables developers to describe their projects at a higher level of

abstraction, allowing more effective communication and freeing team members to get on with the important work of creating value at the application level.

The multiple projects of company will use the multiple logical databases that are created in database software of integrated machine on one per project basis.

During development mode of the project, the project will be maintained in the CVS Repository or SVN Repository to make the resources of the project visible and accessible for all developers of the project.

Declarative Execution:

Everything in Maven is driven in a declarative fashion using Maven's Project Object Model (POM) and specifically, the plugin configurations contained in the POM. The execution of Maven's plugins is coordinated by Maven's build life cycle in a declarative fashion with instructions from Maven's POM.

Maven's POM:

Maven is project-centric by design, and the POM is Maven's description of a single project. Without the POM, Maven is useless - the POM is Maven's currency. It is the POM that drives execution in Maven and this approach can be described as model-driven or declarative execution. The POM is an XML document and looks like the following (very) simplified example:

```
<project>
<modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

POM will allow you to compile, test, and generate basic documentation.

The POM contains every important piece of information about your project.

Maven's Super POM carries with it all the default conventions that Maven encourages, and is the analog of the Java language's java.lang.Object class.

POM contains the following key elements:

- **project** - This is the top-level element in all Maven pom.xml files.

- **modelVersion** - This required element indicates the version of the object model that the POM is using. The version of the model itself changes very infrequently, but it is mandatory in order to ensure stability when Maven introduces new features or other model changes.

- **groupId** - This element indicates the unique identifier of the organization or group that created the project. The groupId is one of the key identifiers of a project and is typically

based on the fully qualified domain name of your organization. For example org.apache.maven.plugins is the designated groupId for all Maven plugins.

❖ **artifactId** - This element indicates the unique base name of the primary artifact being generated by this project. A typical artifact produced by Maven would have the form <artifactId>-<version>.<extension> (for example, myapp-1.0.jar). Additional artifacts such as source bundles also use the artifactId as part of their file name.

❖ **packaging** - This element indicates the package type to be used by this artifact (JAR, WAR, EAR, etc.). This not only means that the artifact produced is a JAR, WAR, or EAR, but also indicates a specific life cycle to use as part of the build process. The life cycle is a topic dealt with later in this chapter. For now, just keep in mind that the selected packaging of a project plays a part in customizing the build life cycle. The default value for the packaging element is jar so you do not have to specify this in most cases.

❖ **version** - This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development.

❖ **name** - This element indicates the display name used for the project. This is often used in Maven's generated documentation, and during the build process for your project, or other projects that use it as a dependency.

❖ **url** - This element indicates where the project's site can be found.

❖ **description** - This element provides a basic description of your project.

All POMs inherit from a parent (despite explicitly defined or not).

This base POM is known as the Super POM, and contains values inherited by default.

Maven use the effective pom (configuration from super pom plus project configuration) to execute relevant goal.

It helps developer to specify minimum configuration details in his/her pom.xml.

Although configurations can be overridden easily

Maven Build LifeCycle:

Maven models projects as **"nouns"** which are described by a POM. The POM captures the identity of a project.

Maven the **"verbs"** are goals packaged in Maven plugins which are tied to a phases in a build lifecycle.

A Maven lifecycle consists of a sequence of named phases: prepare-resources, compile, package, and install among other.

There is phase that captures compilation and a phase that captures packaging.

There are pre- and post- phases which can be used to register goals which must run prior to compilation, or tasks which must be run after a particular phase. When you tell Maven to build a project, you are telling Maven to step through a defined sequence of phases and execute any goals which may have been registered with each phase.

A build lifecycle is an organized sequence of phases that exist to give order to a set of goals. Those goals are chosen and bound by the packaging type of the project being acted upon.

There are three standard lifecycles in maven:

- clean
- default
- Site

The clean phase will be executed first, and then the dependency: copy-dependencies goal will be executed, and finally package phase will be executed.

Clean Life Cycle:

When we execute mvn post-clean command, Maven invokes the clean lifecycle consisting of the following phases.

- pre-clean
- clean
- post-clean

Maven clean goal (clean:clean) is bound to the clean phase in the clean lifecycle. Its clean:clean goal deletes the output of a build by deleting the build directory.

Thus when mvn clean command executes, Maven deletes the build directory.

We can customize this behavior by mentioning goals in any of the above phases of clean life cycle.

Default (or Build) Life Cycle:

This is the primary life cycle of Maven and is used to build the application. It has following 23 phases.

There are few important concepts related to Maven Lifecycles which are wroth to mention:

When a phase is called via Maven command,

Ex: **mvn compile**, only phases upto and including that phase will execute. Different maven goals will be bound to different phases of Maven lifecycle depending upon the type of packaging (JAR / WAR / EAR).

Site Lifecycle:

Maven Site plugin is generally used to create fresh documentation to create reports, deploy site etc.

Phases:

- pre-site
- site
- post-site
- site-deploy

command: C:\MVN\project>mvn site

Maven Build Profile:

A Build profile is a set of configuration values which can be used to set or override default values of Maven build.

Using a build profile, you can customize build for different environments such as Production v/s Development environments.

Profiles are specified in pom.xml file using its activeProfiles / profiles elements and are triggered in variety of ways.

Profiles modify the POM at build time, and are used to give parameters different target environments (for example, the path of the database server in the development, testing, and production environments).

Types of Build Profile Build profiles are majorly of three types

Type	Where it is defined
Per Project	Defined in the project POM file, pom.xml
Global	Defined in Maven global settings xml file (%M2_HOME%/conf/settings.xml)
Per User	Defined in Maven settings xml file (%USER_HOME%/.m2/settings.xml)

Profile Activation:

A Maven Build Profile can be activated in various ways.

- Explicitly using command console input.
- Through maven settings.
- Based on environment variables (User/System variables).
- OS Settings (for example, Windows family).
- Present/missing files.

Maven Plugin's:

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to :

- create jar file
- create war file
- compile code files
- unit testing of code
- create project documentation
- create project reports

A plugin generally provides a set of goals and which can be executed using following syntax:

Syntax: mvn [plugin-name]:[goal-name]

For example, a Java project can be compiled with the maven-compiler-plugin's compile-goal by running following command.

Command: mvn compiler:compile

Coherent Organization of Dependencies:

We are now going to delve into how Maven resolves dependencies and discuss the intimately connected concepts of dependencies, artifacts, and repositories. If you recall, our example POM has a single dependency listed for Junit:

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```


This POM states that your project has a dependency on JUnit, which is straightforward, but you may be asking yourself “Where does that dependency come from?” and “Where is the JAR?” The answers to those questions are not readily apparent without some explanation of how Maven's dependencies,
artifacts and repositories work. In “Maven-speak” an artifact is a specific piece of software.
In Java, the most common artifact is a JAR file, but a Java artifact could also be a WAR, SAR, or EAR file. A dependency is a reference to a specific artifact that resides in a repository. In order for Maven to attempt to satisfy a dependency, Maven needs to know what repository to search as well as the dependency's coordinates. A dependency is uniquely identified by the following identifiers: groupId, artifactId and version.
At a basic level, we can describe the process of dependency management as Maven reaching out into the world, grabbing a dependency, and providing this dependency to your software project. There
is more going on behind the scenes, but the key concept is that Maven dependencies are declarative. In the POM you are not specifically telling Maven where the dependencies are physically located, you are simply telling Maven what a specific project expects.
Maven takes the dependency coordinates you provide in the POM, and it supplies these coordinates

to its own internal dependency mechanisms. With Maven, you stop focusing on a collection of JAR files; instead you deal with logical dependencies. Your project doesn't require junit-3.8.1.jar, instead it depends on version 3.8.1 of the junit artifact produced by the junit group. Dependency Management is one of the most powerful features in Maven.
When a dependency is declared within the context of your project, Maven tries to satisfy that dependency by looking in all of the remote repositories to which it has access, in order to find the artifacts that most closely match the dependency request. If a matching artifact is located, Maven transports it from that remote repository to your local repository for project use.

Repositories:

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.

Maven repositories are of three types:

- Local
- Central
- Remote

Local Repository:

- Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time.
- Maven local repository keeps your project's all dependencies (library jars, plugin jars etc).
- When you run a Maven build, then Maven automatically downloads all the dependency jars into the local repository.
- It helps to avoid references to dependencies stored on remote machine every time a project is build.

Maven local repository by default get created by Maven in %USER_HOME% directory. To override the default location, mention another path in **Maven settings.xml file available at %M2_HOME%\conf** directory.

```
<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>C:/maven_repo</localRepository>
</settings>
```

When you run Maven command, Maven will download dependencies to your custom path.

Central Repository:

Maven central repository is repository provided by Maven community. It contains a large number of commonly used libraries.
When Maven does not find any dependency in local repository, it starts searching in central repository using following **URL: http://repo1.maven.org/maven2/**

Key concepts of Central repository

- This repository is managed by Maven community.
- It is not required to be configured.
- It requires internet access to be searched.

To browse the content of central maven repository, maven community has provided a **URL:http://search.maven.org/#browse.** Using this library, a developer can search all the available libraries in central repository.

Remote Repository:

Sometime, Maven does not find a mentioned dependency in central repository as well then it stopped build process and output error message to console. To prevent such situation, Maven provides concept of **Remote Repository** which is developer's own custom repository containing required libraries or other project jars.
For example, using below mentioned POM.xml, Maven will download dependency (not available in central repository) from Remote Repositories mentioned in the same 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.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<dependencies>
<dependency>
<groupId>com.companyname.common-lib</groupId>
<artifactId>common-lib</artifactId>
<version>1.0.0</version>
</dependency>
<dependencies>
<repositories>
<repository>
<id>companyname.lib1</id>
<url>http://download.companyname.org/maven2/lib1</url>
</repository>
<repository>
<id>companyname.lib2</id>
<url>http://download.companyname.org/maven2/lib2</url>
</repository>
</repositories>
</project>
```

Maven Dependency Search Sequence:

When we execute Maven build commands, Maven starts looking for dependency libraries in the following sequence:

- **Step 1** - Search dependency in local repository, if not found, move to step 2 else if found then do the further processing.
- **Step 2** - Search dependency in central repository, if not found and remote repository/repositories is/are mentioned then move to step 4 else if found, then it is downloaded to local repository for future reference.
- **Step 3** - If a remote repository has not been mentioned, Maven simply stops the processing and throws error (Unable to find dependency).
- **Step 4** - Search dependency in remote repository or repositories, if found then it is downloaded to local repository for future reference otherwise Maven as expected stop processing and throws error (Unable to find dependency).

Maven Benefits:

Organizations and projects that adopt Maven benefit from:

- **Coherence:** Maven allows organizations to standardize on a set of best practices. Because Maven projects adhere to a standard model they are less opaque. The definition of this term from the American Heritage dictionary captures the meaning perfectly: "Marked by an orderly, logical, and aesthetically consistent relation of parts."

- **Reusability:** Maven is built upon a foundation of reuse. When you adopt Maven you are effectively reusing the best practices of an entire industry.

- **Agility:** Maven lowers the barrier to reuse not only for build logic, but also for software components. Maven makes it is easier to create a component and then integrate it into a multi-project build. Developers can jump between different projects without the steep

learning curve that accompanies custom, home-grown build systems.

- **Maintainability:** Organizations that adopt Maven can stop "building the build", and focus on

building the application. Maven projects are more maintainable because they follow a common, publicly-defined model.

Maven Environment SetUp:

Maven is Java based tool, so the very first requirement is to have JDK installed in your machine.

System Requirement

JDK	1.5 or Above
Memory	no minimum requirement.
Disk Space	no minimum requirement.
Operating System	no minimum requirement.

verify Java installation in your machine:

Now open console and execute the following **java** command.

Operating System	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version

Let's verify the output for all the operating systems:

Operating System	OutPut
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)

Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)

Create Java Project:

To create your first project, you will use Maven's **Archetype** mechanism. An archetype is defined as an original pattern or model from which all other things of the same kind are made. In Maven, an archetype is a template of a project, which is combined with some user input to produce a fully functional Maven project.

Archetype:

Archetype is a Maven project templating toolkit. An archetype is defined as *an original pattern or model from which all other things of the same kind are made*. The names fits as we are trying to provide a system that provides a consistent means of generating Maven projects. Archetype will help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates.

Archetype ArtifactIds	Description
maven-archetype-archetype	An archetype which contains a sample archetype.
maven-archetype-j2ee-simple	An archetype which contains a simplifed sample J2EE application.
maven-archetype-mojo	An archetype which contains a sample a sample Maven plugin.
maven-archetype-plugin	An archetype which contains a sample Maven plugin.

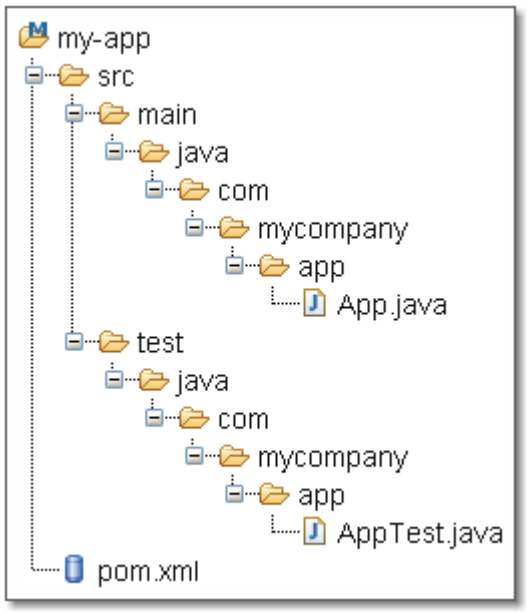
maven-archetype-plugin-site	An archetype which contains a sample Maven plugin site.
maven-archetype-portlet	An archetype which contains a sample JSR-268 Portlet.
maven-archetype-quickstart	An archetype which contains a sample Maven project.
maven-archetype-simple	*An archetype which contains a simple Maven project.*
maven-archetype-site	An archetype which contains a sample Maven site which demonstrates some of the supported document types like APT, XDoc, and FML and demonstrates how to i18n your site.
maven-archetype-site-simple	An archetype which contains a sample Maven site.
maven-archetype-webapp	An archetype which contains a sample Maven Webapp project.

To create the Quick Start Maven project, execute the following:

C:\mvnbook> mvn archetype:create -DgroupId=com.mycompany.app \ -DartifactId=my-app

First, you will notice that a directory named my-app has been created for the new project, and this directory contains your pom.xml, which looks like the following:

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>Maven Quick Start Archetype</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```



Using Project Inheritance:

One of the most powerful features in Maven is *project inheritance*. Using project inheritance allows you to do things like state your organizational information, state your deployment information, or state your common dependencies - all in a single place.

Being the observant user, you have probably taken a peek at all the POMs in each of the projects that make up the Proficio project and noticed the following at the top of each of the POMs:

[...]
<parent>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio</artifactId>
<version>1.0-SNAPSHOT</version>
</parent>
[...]

If you look at the top-level POM for Proficio, you will see that in the dependencies section there is a declaration for JUnit version 3.8.1. In this case the assumption being made is that JUnit will be used for testing in all our child projects. So, by stating the dependency in the top-level POM once, you never have to declare this dependency again, in any of your child POMs. The dependency is stated as following:

<project>
[...]
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
[...]
</project>

What specifically happens for each child POM, is that each one inherits the dependencies section of
the top-level POM. So, if you take a look at the POM for the proficio-core module you will see the following (Note: there is no visible dependency declaration for JUnit):

```
<project>
<parent>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio</artifactId>
<version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<artifactId>proficio-core</artifactId>
<packaging>jar</packaging>
<name>Maven Proficio Core</name>
<dependencies>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-api</artifactId>
</dependency>
<dependency>
<groupId>org.codehaus.plexus</groupId>
<artifactId>plexus-container-default</artifactId>
</dependency>
</dependencies>
</project>
```

In order for you to see what happens during the inheritance process, you will need to use the handy **mvn help:effective-pom command**.
This command will show you the final result for a target POM. After you move into the proficio-core module directory and run the command, take a look at the resulting POM; you will see the JUnit version 3.8.1 dependency:

POM.xml

```
<project>
[...]
<dependencies>
[...]
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
[...]
</dependencies>
[...]
</project>
```

Managing Dependencies:

When you are building applications you typically have a number of dependencies to manage and that number only increases over time, making dependency management difficult to say the least. Maven's strategy for dealing with this problem is to combine the power of project inheritance with specific dependency management elements in the POM.

When you write applications which consist of multiple, individual projects, it is likely that some of those projects will share common dependencies. When this happens it is critical that the same version of a given dependency is used for all your projects, so that the final application works correctly.

You don't want, for example, to end up with multiple versions of a dependency on the classpath when your application executes, as the results can be far from desirable. You want to make sure that all the versions, of all your dependencies, across all of your projects are in alignment so that your testing accurately reflects what you will deploy as your final result. In order to manage, or align, versions of dependencies across several projects, you use the dependency management section in the top-level POM of an application.

To illustrate how this mechanism works, let's look at the dependency management section of the Proficio top-level POM:

```
<project>
[...]
<dependencyManagement>
<dependencies>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-model</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-api</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-store-memory</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-store-xstream</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-core</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>org.codehaus.plexus</groupId>
<artifactId>plexus-container-default</artifactId>
<version>1.0-alpha-9</version>
</dependency>
```

```
</dependencies>
</dependencyManagement>
[...]
</project>
```

Using Snapshots:

While you are developing an application with multiple modules, it is usually the case that each of the modules are in flux. Your APIs might be undergoing some change or your implementations are undergoing change and are being fleshed out, or you may be doing some refactoring. Your build system needs to be able to deal easily with this real-time flux, and this is where Maven's concept of a *snapshot* comes into play. A snapshot in Maven is an artifact that has been prepared using the most recent sources available. If you look at the top-level POM for Proficio you will see a snapshot version specified:

```
<project>
[...]
<version>1.0-SNAPSHOT</version>
<dependencyManagement>
<dependencies>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-model</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>com.devzuz.mvnbook.proficio</groupId>
<artifactId>proficio-api</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>org.codehaus.plexus</groupId>
<artifactId>plexus-container-default</artifactId>
<version>1.0-alpha-9</version>
</dependency>
</dependencies>
</dependencyManagement>
[...]
</project>
```

Specifying a snapshot version for a dependency means that Maven will look for new versions of that dependency without you having to manually specify a new version. Snapshot dependencies are assumed to be changing, so Maven will attempt to update them. By default Maven will look for snapshots on a daily basis, but you can use the -U command line option to force the search for updates.

Controlling Snapshots:

Snapshots were designed to be used in a team environment as a means for sharing development versions of artifacts that have already been built. Usually, in an environment where a number of modules are undergoing concurrent development, the build involves checking out all of the dependent
projects and building them yourself. Additionally, in some cases, where projects are closely related,
you must build all of the modules simultaneously from a master build.
While building all of the modules from source can work well and is handled by Maven inherently, it can
lead to a number of problems:

- It relies on manual updates from developers, which can be error-prone. This will result in local inconsistencies that can produce non-working builds
- There is no common baseline against which to measure progress
- Building can be slower as multiple dependencies must be rebuilt also
- Changes developed against outdated code can make integration more difficult

As you can see from these issues, building from source doesn't fit well with an environment that promotes continuous integration. Instead, use binary snapshots that have been already built and tested.
In Maven, this is achieved by regularly deploying snapshots to a shared repository, such as the internal repository set up in section 7.3. Considering that example, you'll see that the repository was
defined in proficio/trunk/pom.xml:

```
[...]
<distributionManagement>
<repository>
<id>internal</id>
<url>file://localhost/C:/mvnbook/repository/internal</url>
</repository>
[...]
</distributionManagement>
```

Now, deploy proficio-api to the repository with the following command:

```
C:\mvnbook\proficio\trunk\proficio-api> mvn deploy
```

You'll see that it is treated differently than when it was installed in the local repository. The filename that is used is similar to proficio-api-1.0-20070726.120139-1.jar. In this case, the version used is the time that it was deployed (in the UTC timezone) and the build number. If you were to deploy again, the time stamp would change and the build number would increment to 2. This technique allows you to continue using the latest version by declaring a dependency on 1.0-SNAPSHOT, or to lock down a stable version by declaring the dependency version to be the specific equivalent such as 1.0-20070726.12013-1. While this is not usually the case, locking the version in this way may be important if there are recent changes to the repository that need to be ignored temporarily.

The -U argument in the prior command is required to force Maven to update all of the snapshots in
the build. If it were omitted, by default, no update would be performed. This is because the default
policy is to update snapshots daily – that is, to check for an update the first time that particular dependency is used after midnight local time.

```
C:\mvnbook\proficio\trunk\proficio-core> mvn -U install
```

We can also change the interval by changing the repository configuration. To see this, add the following configuration to the repository configuration you defined above in proficio/trunk/pom.xml:

```
[...]
<repository>
[...]
<snapshots>
<updatePolicy>interval:60</updatePolicy>
</snapshots>
</repository>
[...]
```

In this example, any snapshot dependencies will be checked once an hour to determine if there are
updates in the remote repository. The settings that can be used for the update policy are never, always, daily (the default), and interval:*minutes*.

Code Compile:

Compile the source code using the following command:

C:\mvnbook\my-app> mvn compile

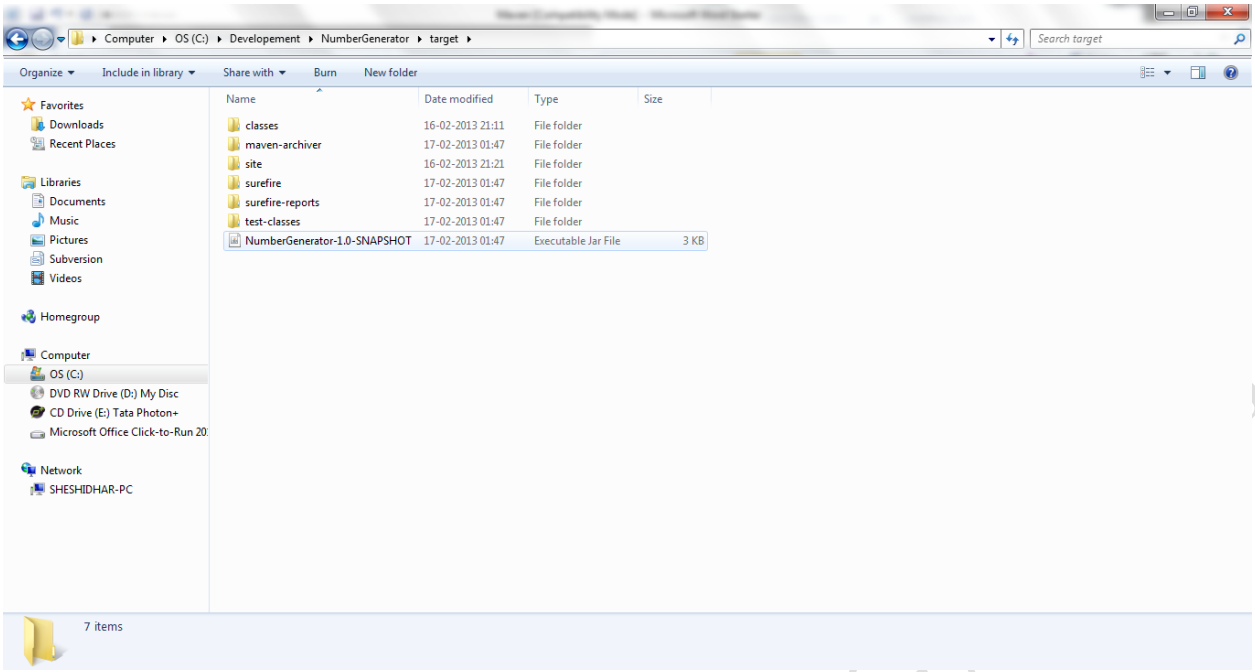
Compiling Test Sources and Running Unit Tests:

mvn test will always run the **compile** and **test-compile** phases first

Packaging and Installation to Your Local Repository:

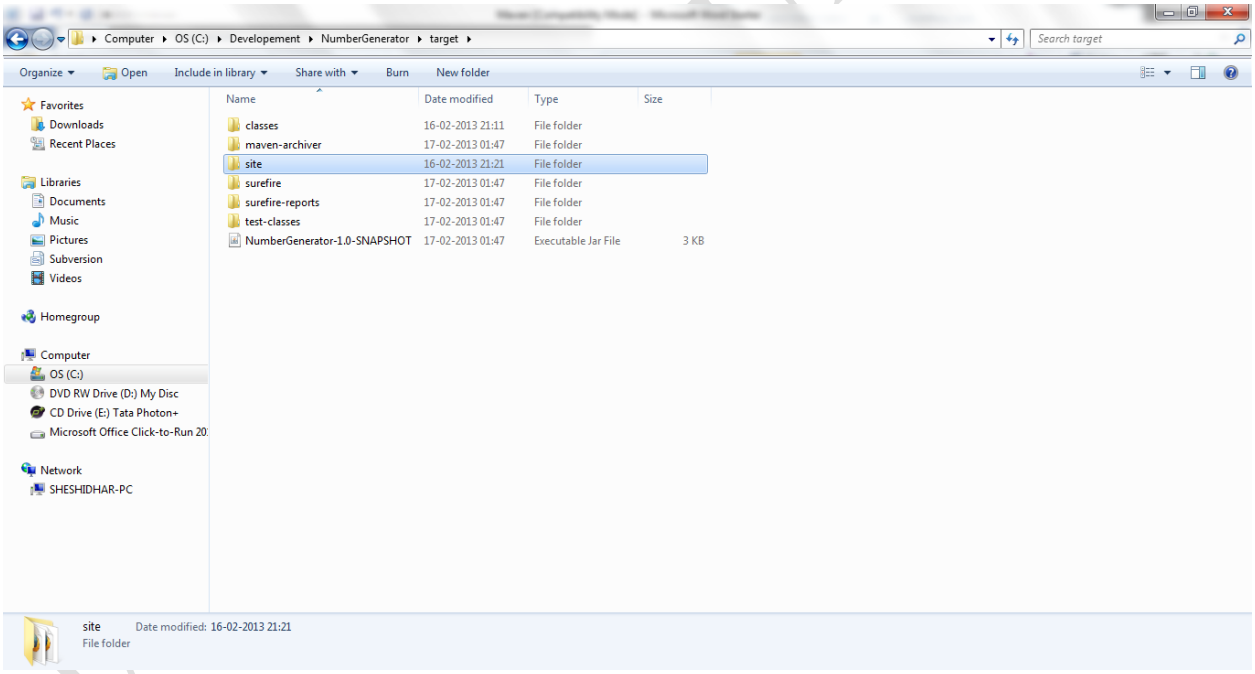
Making a JAR file is straightforward and can be accomplished by executing the following command:

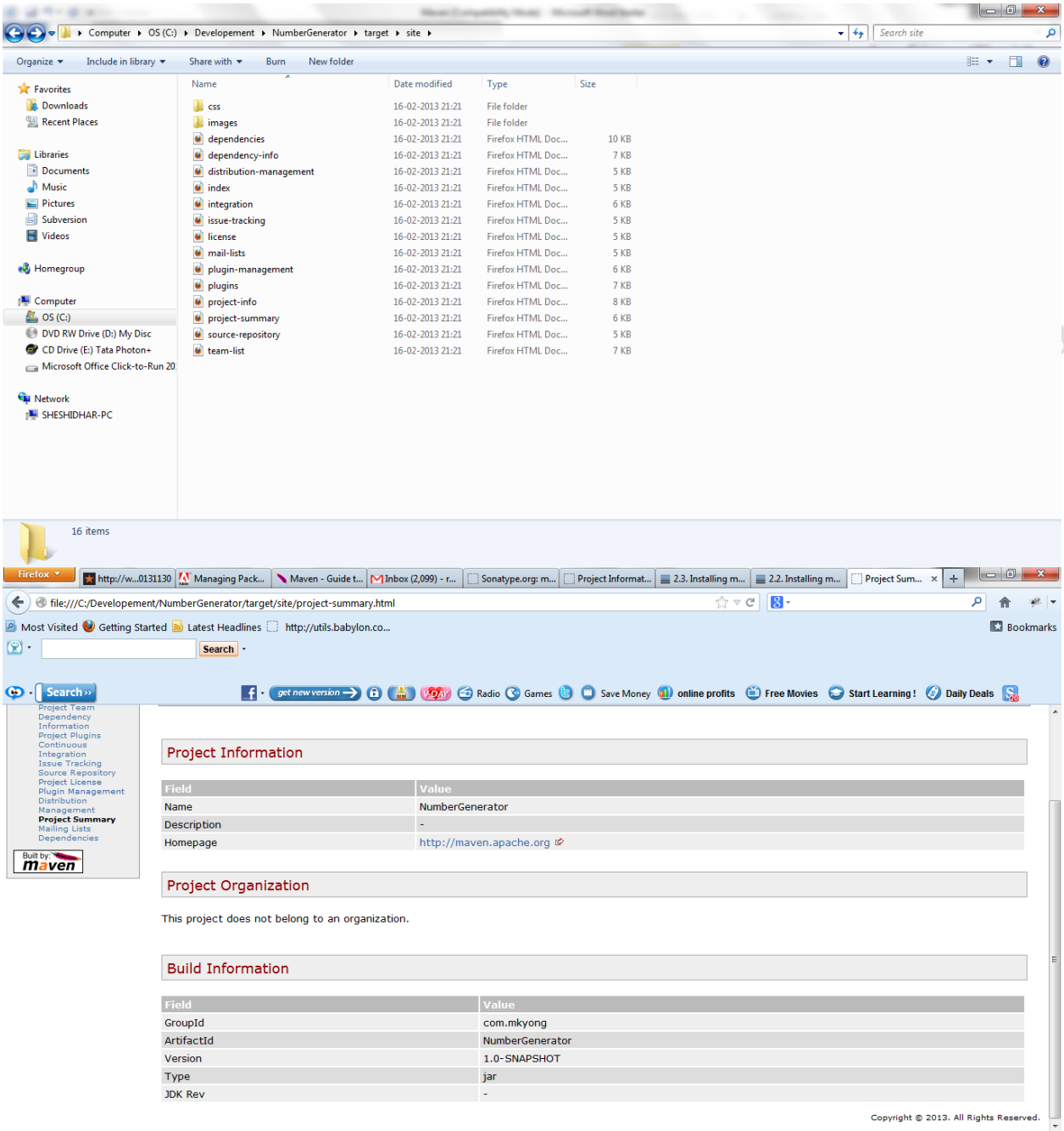
C:\mvnbook\my-app> mvn package

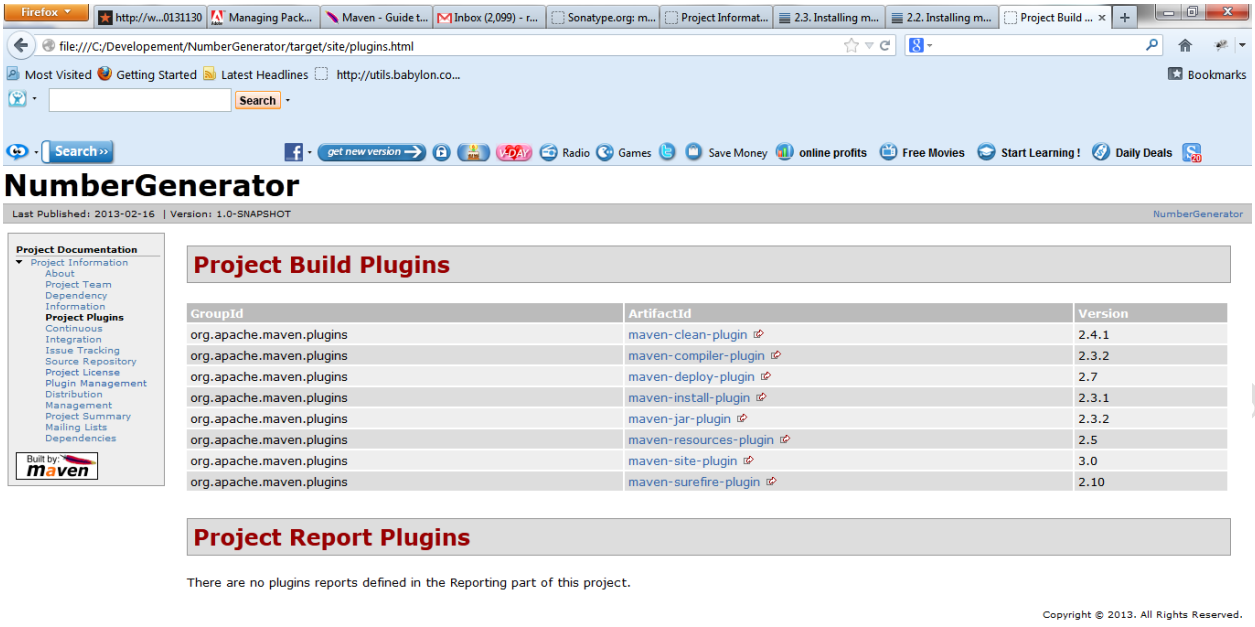


create a basic Web site for your project, simply execute the following command:

C:\mvnbook\my-app> mvn site







Deploying your Application:

Currently Maven supports several methods of deployment, including simple file-based deployment, SSH2 deployment, SFTP deployment, FTP deployment, and external SSH deployment. In order to deploy, you need to correctly configure your **distributionManagement** element in your POM, which would typically be your top-level POM, so that all child POMs can inherit this information. Here are some examples of how to configure your POM via the various deployment mechanisms.

➢ Deploying to the File System:

To deploy to the file system you would use something like the following:

```
<project>
[...]
<distributionManagement>
<repository>
<id>proficio-repository</id>
<name>Proficio Repository</name>
<url>file://${basedir}/target/deploy</url>
</repository>
</distributionManagement>
[...]
</project>
```

➢ Deploying with SSH2:

To deploy to an SSH2 server you would use something like the following:

```
<project>
[...]
<distributionManagement>
<repository>
<id>proficio-repository</id>
<name>Proficio Repository</name>
<url>scp://sshserver.yourcompany.com/deploy</url>
</repository>
</distributionManagement>
[...]
</project>
```

- **Deploying with SFTP:**

To deploy to an SFTP server you would use something like the following:

```
<project>
[...]
<distributionManagement>
<repository>
<id>proficio-repository</id>
<name>Proficio Repository</name>
<url>sftp://ftpserver.yourcompany.com/deploy</url>
</repository>
</distributionManagement>
[...]
</project>
```


Deployment Automation:

In normally a deployment process consists of following steps

- Check-in the code from all project 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.

Problem Statement :

There are normally multiple people involved in above mentioned deployment process. One team may handles check-in of code, other may handle build and so on. It is very likely that any step may get missed out due to manual efforts involved and owing to multi-team environment. For example, older build may not be replaced on network machine and deployment team deployed the older build again.

Solution :

Automate the deployment process by combining

- Maven, to build and release projects,
- SubVersion, source code repository, to manage source code,
- and Remote Repository Manager (Jfrog/Nexus) to manage project binaries.

We'll be using Maven Release plug-in to create an automated release process.

For Example: durga-soft-api project 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>bus-core-api</groupId>
<artifactId>durga-soft-api</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<scm>
<url>http://www.svn.com</url>
<connection>scm:svn:http://localhost:8080/svn/jrepo/trunk/
Framework</connection>
<developerConnection>scm:svn:${username}/${password}@localhost:8080:
common_core_api:1101:code</developerConnection>
</scm>
<distributionManagement>
<repository>
<id>Core-API-Java-Release</id>
<name>Release repository</name>
<url>http://localhost:8081/nexus/content/repositories/
Core-Api-Release</url>
</repository>
</distributionManagement>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-release-plugin</artifactId>
<version>2.0-beta-9</version>
<configuration>
<useReleaseProfile>false</useReleaseProfile>
<goals>deploy</goals>
```

```
<scmCommentPrefix>[durga-soft-api-release-checkin]-<
/scmCommentPrefix>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

In Pom.xml, following are the important elements used:

Element	Description
SCM	Configures the SVN location from where Maven will check out the source code.
Repositories	Location where built WAR/EAR/JAR or any other artifact will be stored after code build is successful.
Plugin	maven-release-plugin is configured to automate the deployment process.

Maven Release Plug-in:

The Maven does following useful tasks using *maven-release-plugin*.

mvn release:clean

It cleans the workspace in case the last release process was not successful.

mvn release:rollback

Rollback the changes done to workspace code and configuration in case the last release process was not successful.

mvn release:prepare

Performs multiple number of operations:

- Checks whether there are any uncommitted local changes or not
- Ensures that there are no SNAPSHOT dependencies

- Changes the version of the application and removes SNAPSHOT from the version to make release
- Update pom files to SVN.
- Run test cases
- Commit the modified POM files
- Tag the code in subversion
- Increment the version number and append SNAPSHOT for future release
- Commit the modified POM files to SVN.

mvn release:perform

Checks out the code using the previously defined tag and run the Maven deploy goal to deploy the war or built artifact to repository

Maven Web Application:

create a simple java web application using **maven-archetype-webapp** plugin.

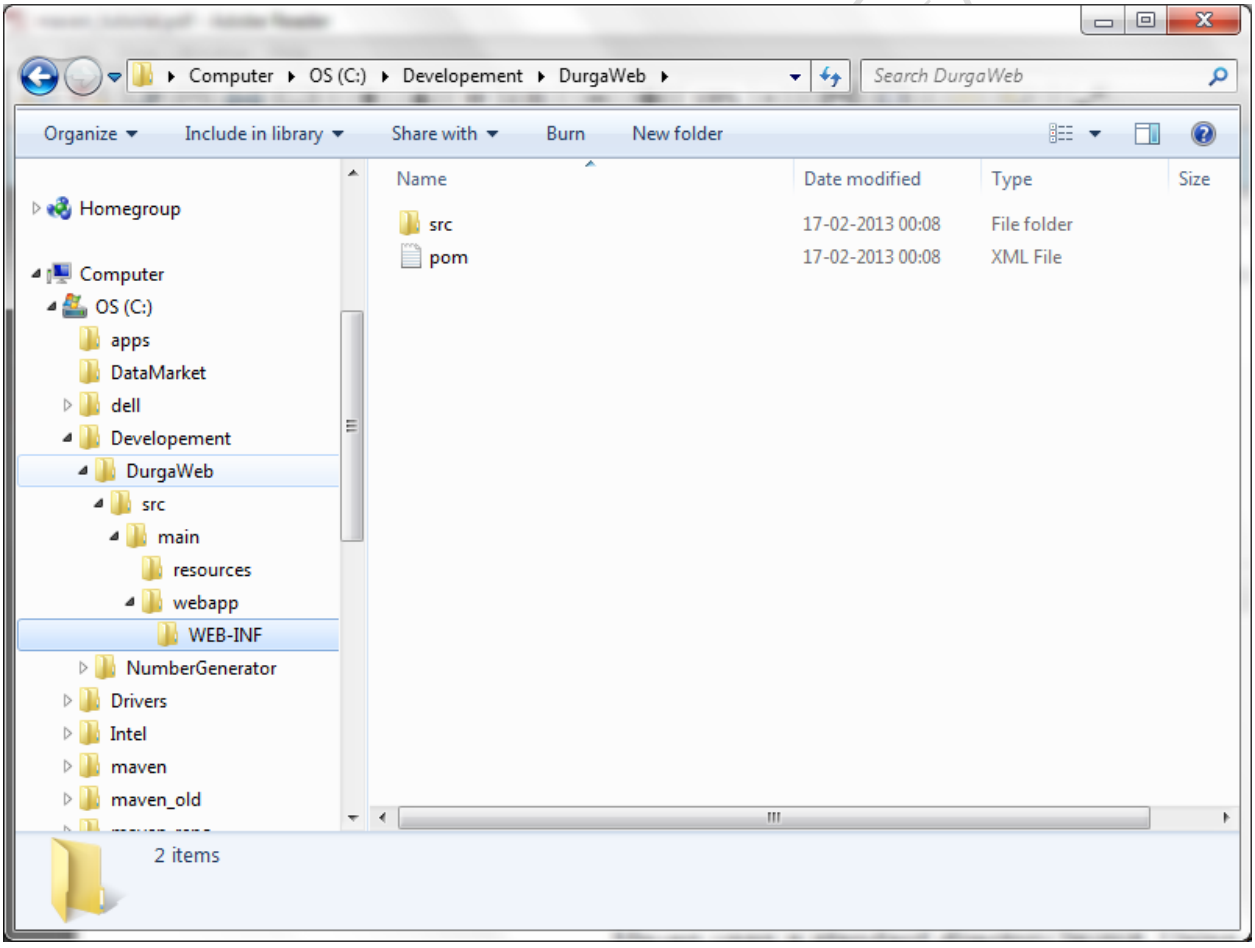
C:\anyfolder>mvn archetype:generate -DgroupId=com.durga.scjp -DartifactId=DurgaWeb -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

Maven will start processing and will create the complete web based java application project structure.

O/P can be seen at console.

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Batch mode
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project
from Old (1.x) Archetype: maven-archetype-webapp:1.0
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.automobile
[INFO] Parameter: packageName, Value: com.companyname.automobile
[INFO] Parameter: package, Value: com.companyname.automobile
[INFO] Parameter: artifactId, Value: trucks
[INFO] Parameter: basedir, Value: C:\MVN
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\trucks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Tue Jul 17 11:00:00 IST 2012
[INFO] Final Memory: 20M/89M
[INFO] ------------------------------------------------------------------------
```



Maven uses a standard directory layout. Using above example, we can understand following key concepts

Folder Structure	Description
DurgaWeb	**contains src folder and pom.xml**
src/main/webapp	**contains index.jsp and WEB-INF folder.**
src/main/webapp/WEB-INF	**contains web.xml**
src/main/resources	**it contains images/properties files .**

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">
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.automobile</groupId>
<artifactId>DurgaWeb</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
<name>trucks 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>
</dependencies>
<build>
<finalName>DurgaWeb</finalName>
</build>
</project>
```

Maven also created a sample JSP Source file

C:\ >Development> DuregaWeb> src > main > webapp > you will see index.jsp.

```
<html>
<body>
<h2>Hello World!</h2>
</body>
</html>
```


Build Web Application:

Let's open command console, go the C:\Devlopment\DurgaWeb\ directory and execute the following **mvn** command.

C:\Development\DurgaWeb>mvn clean package

Maven will start building the project.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building trucks Maven Webapp
[INFO] task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to
copy filtered resources,i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] No sources to compile
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to
copy filtered resources,i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory
C:\Development\DurgaWeb\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] No sources to compile
[INFO] [surefire:test {execution: default-test}]
[INFO] No tests to run.
[INFO] [war:war {execution: default-war}]
[INFO] Packaging webapp
[INFO] Assembling webapp[trucks] in [C:\Development\DurgaWeb\target\trucks]
```

```
[INFO] Processing war project
[INFO] Copying webapp resources[C:\Development\DurgaWeb\src\main\webapp]
[INFO] Webapp assembled in[77 msecs]
[INFO] Building war: C:\Development\DurgaWeb\target\trucks.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Tue Jul 17 11:22:45 IST 2012
[INFO] Final Memory: 11M/85M
[INFO] ------------------------------------------------------------------------
```

Deploy Web Application:

Now copy the **DurgaWeb.war** created in **C:\ > Development >DurgaWeb > target >** folder to your webserver webapp directory and restart the webserver.

Test Web Application:

Run the web-application using URL : **http://<server-name>:<port-number>/trucks/index.jsp**
Verify the output.

Maven Eclipse Integration:

Eclipse provides an excellent plugin m2eclipse which seamlessly integrates Maven and Eclipse together.

Some of features of m2eclipse are listed below:

- You can run Maven goals from Eclipse.

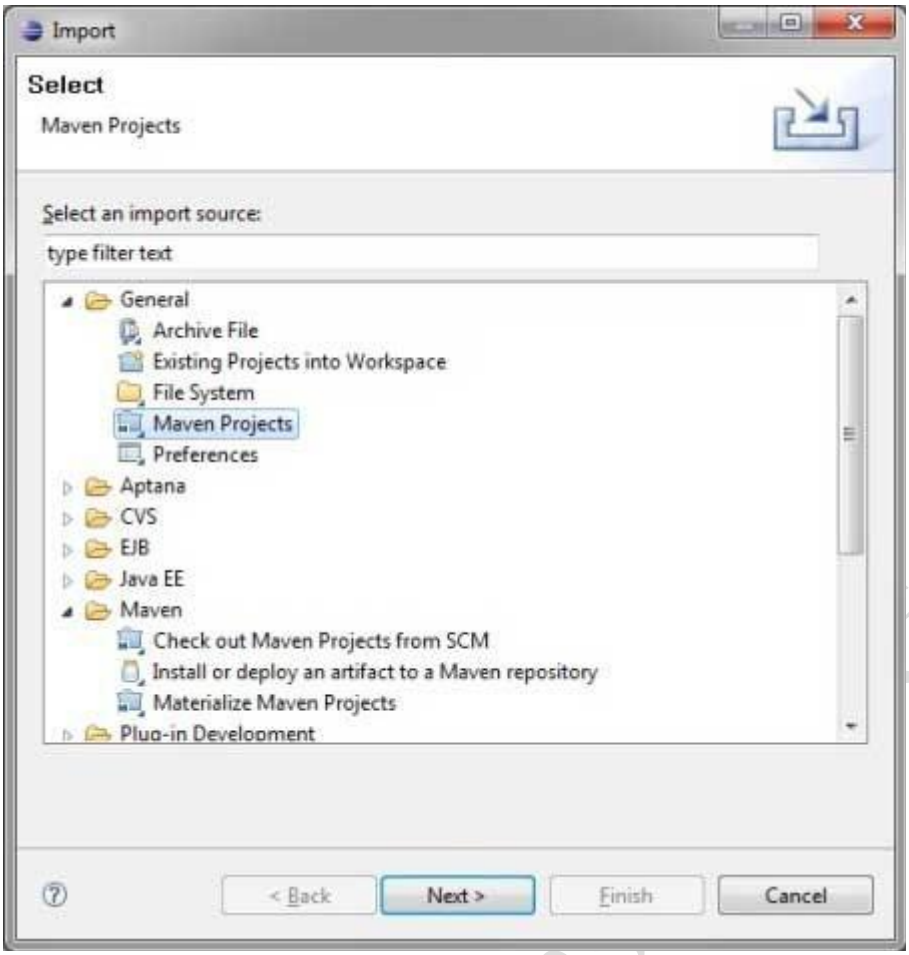
- You can view the output of Maven commands inside the Eclipse using its own console.
- You can update maven dependencies with IDE.
- You can Launch Maven builds from within Eclipse.
- It does the dependency management for Eclipse build path based on Maven's pom.xml.
- It resolves Maven dependencies from the Eclipse workspace without installing to local Maven repository (requires dependency project be in same workspace).
- It automatic downloads required dependencies and sources from the remote Maven repositories.

Use any of the following links to install m2eclipse:

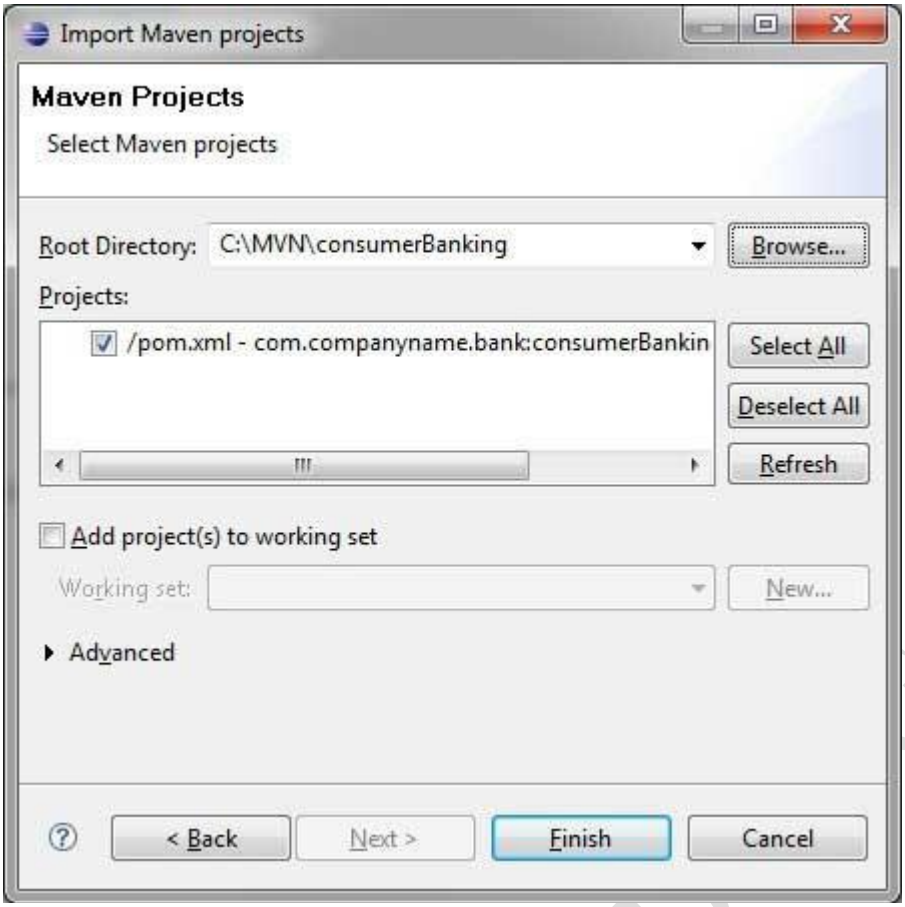
Eclipse	URL
Eclipse 3.5 (Gallileo)	http://www.sonatype.com/books/m2eclipse-book/reference/ch02s03.html
Eclipse 3.6 (Helios)	http://www.sonatype.com/books/m2eclipse-book/reference/install-sect-marketplace.html

Import a maven project in Eclipse:

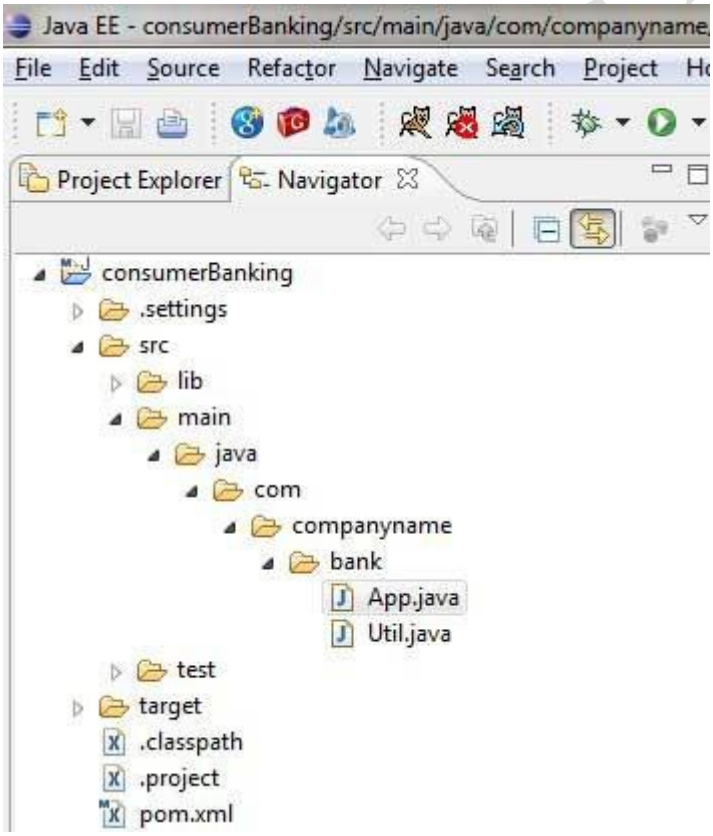
- Open Eclipse.
- Select File > Import > option.
- Select Maven Projects Option. Click on Next Button.



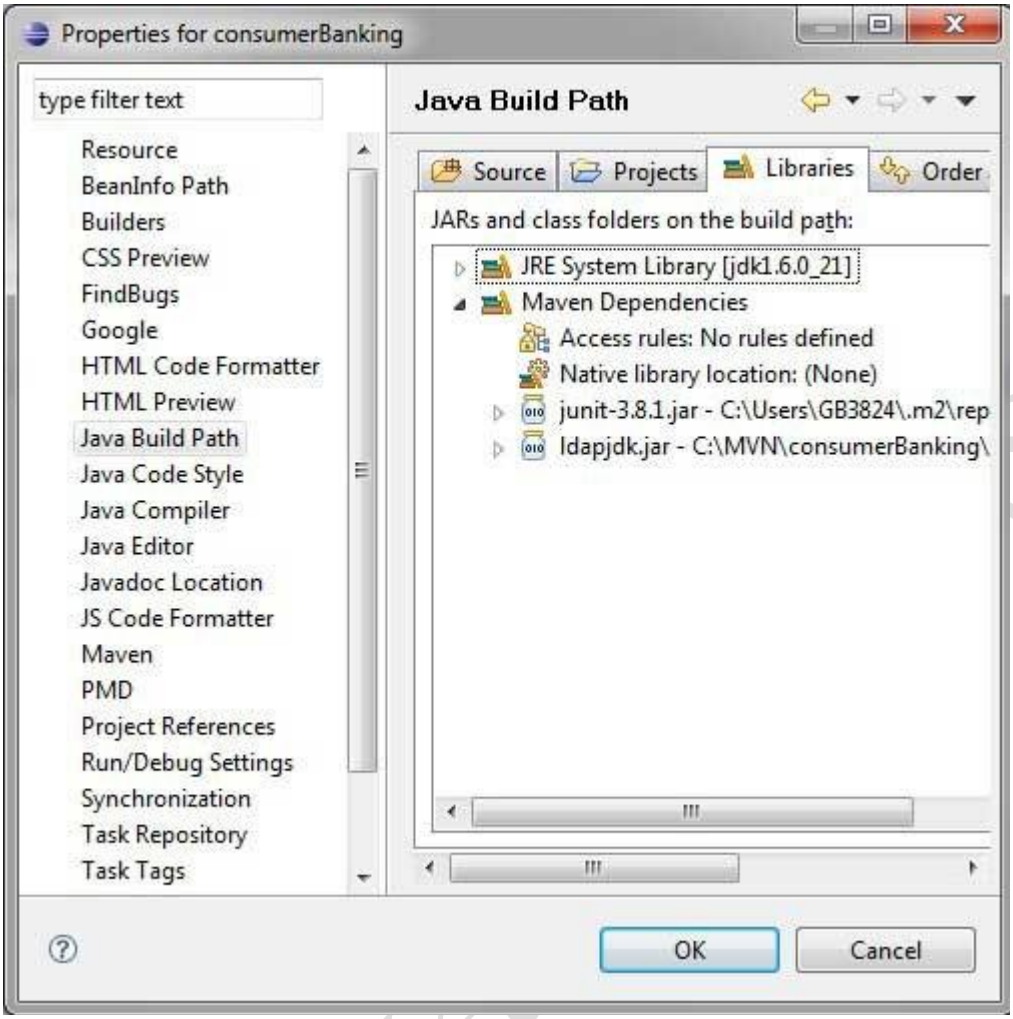
- Select Project location, where a project was created using Maven. We've create a Java Project consumerBanking.See Maven Creating Project to see how to create a project using Maven.
- Click Finish Button.



Now, you can see the maven project in eclipse.



Now, have a look at consumerBanking project properties. You can see that Eclipse has added Maven dependencies to java build path.

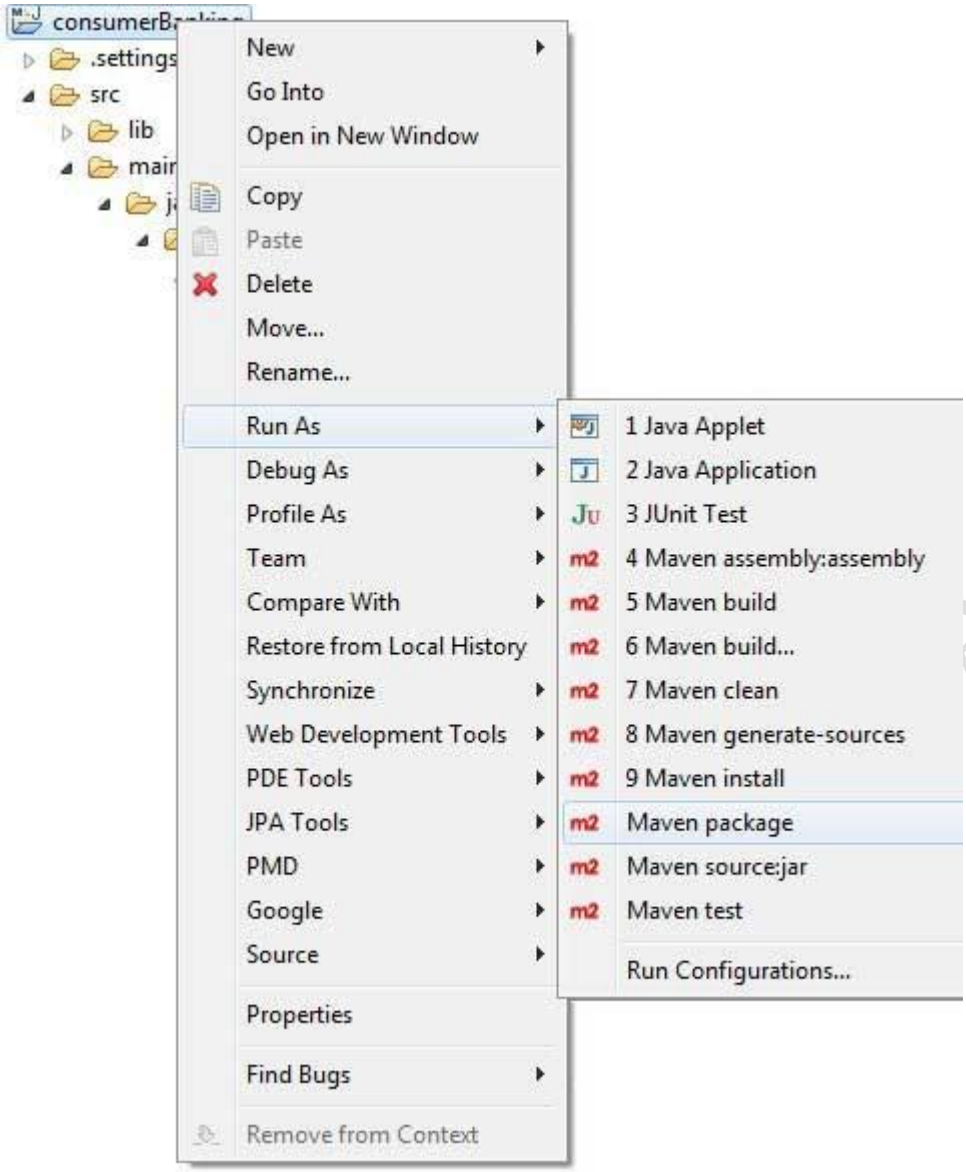


Now, Its time to build this project using maven capability of eclipse.

- Right Click on consumerBanking project to open context menu.
- Select Run as option
- Then maven package option
- Maven will start building the project. You can see the output in Eclipse Console.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]
[INFO] Id: com.companyname.bank:consumerBanking:jar:1.0-SNAPSHOT
[INFO] task-segment: [package]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
```

```
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
[INFO] Surefire report directory:
C:\MVN\consumerBanking\target\surefire-reports
-------------------------------------------------------
T E S T S
-------------------------------------------------------
Running com.companyname.bank.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] [jar:jar]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Thu Jul 12 18:18:24 IST 2012
[INFO] Final Memory: 2M/15M
[INFO] ------------------------------------------------------------------------
```



Now, right click on App.java. Select Run As option. Select As Java Application. You will see the result

O/P: Hello World!

Maven Programs

1) Struts Application
2) Hibernate Application
3) Spring using JDBC
4) Sample Web Application.

Struts Application

Steps to Struts Application using Maven

Generate a simple webapp with archetype:generate:

C:\Development>mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp -DgroupId=maven-tutorial -DartifactId=struts1app

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] Setting property: classpath.resource.loader.class =>
'org.codehaus.plexus.velocity.ContextClassLoaderResourceLoader'.
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Interactive mode
Define value for version:  1.0-SNAPSHOT: :
Confirm properties configuration:
groupId: maven-tutorial
artifactId: struts1app
version: 1.0-SNAPSHOT
package: maven-tutorial
 Y: :
[INFO] ------------------------------------------------------------------------
-
[INFO] Using following parameters for creating OldArchetype: maven-archetype-
webapp:1.0
[INFO] ------------------------------------------------------------------------
-
[INFO] Parameter: groupId, Value: maven-tutorial
[INFO] Parameter: packageName, Value: maven-tutorial
```

```
[INFO] Parameter: package, Value: maven-tutorial
[INFO] Parameter: artifactId, Value: struts1app
[INFO] Parameter: basedir, Value: C:\maven
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] ********************* End of debug info from resources from generated POM
**********************
[INFO] OldArchetype created in dir: C:\Development\struts1app
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 20 seconds
[INFO] Finished at: Mon Sep 07 10:36:45 CDT 2009
[INFO] Final Memory: 8M/14M
[INFO] ------------------------------------------------------------------------
```


C:\Development>

Maven generates a directory structure like this:

struts1app
struts1app/pom.xml
struts1app/src
struts1app/src/main
struts1app/src/main/resources
struts1app/src/main/webapp
struts1app/src/main/webapp/index.jsp
struts1app/src/main/webapp/WEB-INF
struts1app/src/main/webapp/WEB-INF/web.xml

Simple POM:

<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">
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven-tutorial</groupId>
  <artifactId>struts1app</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>struts1app 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>
  </dependencies>
  <build>
    <finalName>struts1app</finalName>
  </build>
</project>
```


Create **settings.xml** in **C:\Users\{Username}\.m2**, add **Java.net** repository for Java EE dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
   <profiles>
      <profile>
         <id>DefaultProfile</id>
```

```
        <activation>
           <activeByDefault>true</activeByDefault>
        </activation>

          <repositories>
               <repository>
                  <id>maven2-repository.dev.java.net</id>
                  <name>Java.net Repository for Maven</name>
                  <url>http://download.java.net/maven/2/</url>
                  <layout>default</layout>
               </repository>
          </repositories>
       </profile>
   </profiles>
</settings>
```

Add Java EE and Struts dependencies in pom.xml. Note that the Java EE dependency has scope provided, meaning that the web app container provides the jars, therefore we don't need to bundle them with our war fie.

```
<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">
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven-tutorial</groupId>
  <artifactId>struts1app</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>struts1app 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</groupId>
        <artifactId>javaee-api</artifactId>
        <version>6.0-SNAPSHOT</version>
        <scope>provided</scope>
    </dependency>
```

```
    <dependency>
        <groupId>struts</groupId>
        <artifactId>struts</artifactId>
        <version>1.2.9</version>
    </dependency>
  </dependencies>

  <build>
    <finalName>struts1app</finalName>
  </build>
</project>
```

Create a directory named java under main, create the Struts form and action classes:

src/main/java
src/main/java/com
src/main/java/com/dss
src/main/java/com/dss/HelloAction.java
src/main/java/com/dss/HelloForm.java

HelloForm.java

```
package com.dss;

import org.apache.struts.action.ActionForm;

public class HelloForm extends ActionForm {
    private String name;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

HelloAction.java

```
package com.dss;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.*;

public class HelloAction extends Action {

   public ActionForward execute(ActionMapping map,
                                ActionForm form,
                                HttpServletRequest req,
                                HttpServletResponse resp)
                         throws IOException, ServletException {
        HelloForm f = (HelloForm) form;
        req.setAttribute("name", f.getName());
        return map.findForward("hello");
   }

}
```

Create a directory named jsp under webapp, and create 2 jsp files:

index.jsp:

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>

<html>
<body>
   <html:form action="/hello" method="post">
        Enter Name: <html:text property="name"/>
        <br/>
        <input type="submit" name="submit" value="Go"/>
   </html:form>
</body>
```

```
</html>
```

hello.jsp:

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"
%>

<html>
<body>
<h2>Hello, <bean:write name="name"/>!</h2>
</body>
</html>
```

Change the contents of the existing index.jsp to:

```
<html>
<head>
<meta http-equiv="refresh" content="0;URL=index.do">
</head>
<body>
</body>
</html>
```

Create struts-config.xml under WEB-INF:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
"http://struts.apache.org/dtds/struts-config_1_3.dtd">

<struts-config>
   <global-forwards>
      <forward name="index" path="/jsp/index.jsp"/>
   </global-forwards>

   <form-beans>
      <form-bean name="helloForm" type="com.dss.HelloForm" />
   </form-beans>

   <action-mappings>
```

```
    <action
      path="/index"
        name="helloForm"
      type="org.apache.struts.actions.ForwardAction"
      parameter="/jsp/index.jsp"
      validate="false">
    </action>

    <action path="/hello"
            name="helloForm"
            scope="request"
            type="com.dss.HelloAction"
            validate="false">
      <forward name="hello" path="/jsp/hello.jsp" />
    </action>
  </action-mappings>
</struts-config>
```

Change the contents of WEB-INF/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>action</servlet-name>
   <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
   <init-param>
     <param-name>config</param-name>
     <param-value>/WEB-INF/struts-config.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
   <servlet-name>action</servlet-name>
   <url-pattern>*.do</url-pattern>
 </servlet-mapping>
</web-app>
```

The final structure of the project should look like:

```
struts1app
struts1app/pom.xml
struts1app/src
struts1app/src/main
struts1app/src/main/java
```

struts1app/src/main/java/com
struts1app/src/main/java/com/dss
struts1app/src/main/java/com/dss/HelloAction.java
struts1app/src/main/java/com/dss/HelloForm.java
struts1app/src/main/resources
struts1app/src/main/webapp
struts1app/src/main/webapp/index.jsp
struts1app/src/main/webapp/jsp
struts1app/src/main/webapp/jsp/hello.jsp
struts1app/src/main/webapp/jsp/index.jsp
struts1app/src/main/webapp/WEB-INF
struts1app/src/main/webapp/WEB-INF/struts-config.xml
struts1app/src/main/webapp/WEB-INF/web.xml

Finally, build with "mvn package".

Maven will generates struts1app.war and place it in target folder. Take the war file and deploy it in server.

==========

Hibernate with Maven Application

```
CREATE TABLE DBUSER (
  USER_ID       NUMBER (5)    NOT NULL,
  USERNAME      VARCHAR2 (20)  NOT NULL,
  CREATED_BY    VARCHAR2 (20)  NOT NULL,
  CREATED_DATE  DATE          NOT NULL,
  PRIMARY KEY ( USER_ID )
);
```

Use Maven archetype:generate to create a standard project structure.

C:\Development>mvn archetype:generate -DgroupId=com.mkyong -DartifactId=HibernateExample -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=**false**

Maven to Eclipse IDE

To convert the generated Maven based project to Eclipse project, and import it into your Eclipse IDE.

mvn eclipse:eclipse

Add Hibernate and Oracle Dependency

Update **pom.xml** file, and add all related dependencies.

```
<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">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.dss.common</groupId>
	<artifactId>HibernateExample</artifactId>
	<packaging>jar</packaging>
	<version>1.0</version>
	<name>HibernateExample</name>
	<url>http://maven.apache.org</url>

	<repositories>
		<repository>
			<id>JBoss repository</id>

	<url>http://repository.jboss.org/nexus/content/groups/public/</url>
		</repository>
	</repositories>

	<dependencies>

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

		<!-- ORACLE database driver -->
		<dependency>
		<groupId>com.oracle</groupId>
		<artifactId>ojdbc14</artifactId>
```

```
		<version>10.2.0</version>
		</dependency>

		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-core</artifactId>
			<version>3.6.3.Final</version>
		</dependency>

		<dependency>
			<groupId>javassist</groupId>
			<artifactId>javassist</artifactId>
			<version>3.12.1.GA</version>
		</dependency>

	</dependencies>
</project>
```

Hibernate Mapping file (hbm) + Model

Create a Hibernate XML mapping file and Model class for table "**DBUSER**".

Create following "DBUser.hbm.xml" file and put it under "**src/main/resources/com/dss/user**".

Note: Create the folder if it does not exists.

File : DBUser.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
   <class name="com.dss.user.DBUser" table="DBUSER">
      <id name="userId" type="int">
         <column name="USER_ID" precision="5" scale="0" />
         <generator class="assigned" />
      </id>
      <property name="username" type="string">
         <column name="USERNAME" length="20" not-null="true" />
      </property>
      <property name="createdBy" type="string">
         <column name="CREATED_BY" length="20" not-null="true" />
      </property>
      <property name="createdDate" type="date">
         <column name="CREATED_DATE" length="7" not-null="true" />
      </property>
   </class>
</hibernate-mapping>
```

Create a "DBUser.java" file and put it under "src/main/java/com/dss/user/"

File : DBUser.java

```
package com.dss.user;

import java.util.Date;

public class DBUser implements java.io.Serializable {

        private int userId;
        private String username;
        private String createdBy;
        private Date createdDate;

        public DBUser() {
        }

        public DBUser(int userId, String username, String createdBy,
                        Date createdDate) {
                this.userId = userId;
                this.username = username;
                this.createdBy = createdBy;
                this.createdDate = createdDate;
        }

        public int getUserId() {
                return this.userId;
        }
```

```
	public void setUserId(int userId) {
		this.userId = userId;
	}

	public String getUsername() {
		return this.username;
	}

	public void setUsername(String username) {
		this.username = username;
	}

	public String getCreatedBy() {
		return this.createdBy;
	}

	public void setCreatedBy(String createdBy) {
		this.createdBy = createdBy;
	}

	public Date getCreatedDate() {
		return this.createdDate;
	}

	public void setCreatedDate(Date createdDate) {
		this.createdDate = createdDate;
	}

}
```


Hibernate Configuration File

Create a Hibernate configuration file "**hibernate.cfg.xml**" and put it under the root of resources folder, "**src/main/resources/hibernate.cfg.xml**", and fill in your Oracle database details. And map to above Hibernate mapping file – "**DBUser.hbm.xml**".

File : hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>
  <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property
>
  <property
name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
  <property name="hibernate.connection.username">system</property>
  <property name="hibernate.connection.password">admin</property>
  <property
name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
  <property name="show_sql">true</property>
  <mapping resource="com/dss/user/DBUser.hbm.xml"></mapping>
</session-factory>
</hibernate-configuration>
```

7. Hibernate Utility

Create a classic "`HibernateUtil.java`" class to take care of Hibernate session management. And put under "**src/main/java/com/dss/util/HibernateUtil.java**"

File : HibernateUtil.java

```
package com.dss.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

	private static final SessionFactory sessionFactory =
buildSessionFactory();

	private static SessionFactory buildSessionFactory() {
		try {
			// Create the SessionFactory from hibernate.cfg.xml
			return new
Configuration().configure().buildSessionFactory();
		} catch (Throwable ex) {
			// Make sure you log the exception, as it might be
swallowed
			System.err.println("Initial SessionFactory creation
failed." + ex);
			throw new ExceptionInInitializerError(ex);
		}
	}

	public static SessionFactory getSessionFactory() {
		return sessionFactory;
	}

	public static void shutdown() {
		// Close caches and connection pools
		getSessionFactory().close();
	}
```

```
}
```

Hibernate Coding

Update "App.java", to code Hibernate to save a dummy user record into a table "**DBUSER**".

File : App.java

```
package com.dss;

import java.util.Date;
import org.hibernate.Session;
import com.dss.util.HibernateUtil;
import com.dss.user.DBUser;

public class App {
	public static void main(String[] args) {
		System.out.println("Maven + Hibernate + Oracle");
		Session session = HibernateUtil.getSessionFactory().openSession();

		session.beginTransaction();
		DBUser user = new DBUser();

		user.setUserId(100);
		user.setUsername("superman");
		user.setCreatedBy("system");
		user.setCreatedDate(new Date());

		session.save(user);
		session.getTransaction().commit();
	}
}
```


Spring Using JDBC using Maven Example:

1)Crate a table CUSTOMER in oracle DB.

```
CREATE TABLE CUSTOMER
    ( CUST_ID NUMBER(5) NOT NULL ,
      NAME VARCHAR2(20) NULL ,
      AGE NUMBER(2) NULL ,
      CONSTRAINT PK_PERSON PRIMARY KEY (CUST_ID)
    );
```

Use Maven archetype:generate to create a standard project structure.

C:\Development>mvn archetype:generate -DgroupId=com.dss -DartifactId=SpringJDBCExample -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=**false**

Add Hibernate and Oracle Dependency

Update **pom.xml** file, and add all related dependencies.

```
<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">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.dss.common</groupId>
 <artifactId>SpringJDBCExample</artifactId>
 <packaging>jar</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>SpringJDBCExample</name>
 <url>http://maven.apache.org</url>
  <dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
```

```
  </dependency>
 <!-- Spring framework -->
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.5.6</version>
</dependency>
  <dependency>
        <groupId>com.oracle</groupId>
            <artifactId>ojdbc14</artifactId>
            <version>10.2.0</version>
        </dependency>
 </dependencies>
</project>
```

Customer model

Add a customer model to store customer's data.

```
package com.dss.customer.model;

import java.sql.Timestamp;

public class Customer
{
        int custId;
        String name;
        int age;
        //getter and setter methods

}
```

Data Access Object (DAO) pattern

Craete **CustomerDao.java** under **\src\main\java\com\mkyong\customer\dao**

Customer Dao interface.

```
package com.mkyong.customer.dao;

import com.mkyong.customer.model.Customer;
```

```
public interface CustomerDAO
{
	public void insert(Customer customer);
	public Customer findByCustomerId(int custId);
}
```

Create **JdbcCustomerDAO.java** under **src\main\java\com\mkyong\customer\dao\impl**

Customer Dao implementation, JdbcCustomerDAO.java use JDBC to issue a simple insert and select statement.

```
package com.dss.customer.dao.impl;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.dss.customer.dao.CustomerDAO;
import com.dss.customer.model.Customer;

public class JdbcCustomerDAO implements CustomerDAO
{
	private DataSource dataSource;

	public void setDataSource(DataSource dataSource) {
		this.dataSource = dataSource;
	}

	public void insert(Customer customer){

		String sql = "INSERT INTO CUSTOMER " +
				"(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";
		Connection conn = null;

		try {
			conn = dataSource.getConnection();
			PreparedStatement ps = conn.prepareStatement(sql);
			ps.setInt(1, customer.getCustId());
			ps.setString(2, customer.getName());
			ps.setInt(3, customer.getAge());
			ps.executeUpdate();
			ps.close();

		} catch (SQLException e) {
			throw new RuntimeException(e);

		} finally {
			if (conn != null) {
				try {
					conn.close();
				} catch (SQLException e) {}
			}
		}
	}

	public Customer findByCustomerId(int custId){

		String sql = "SELECT * FROM CUSTOMER WHERE CUST_ID = ?";
```

```
                Connection conn = null;

                try {
                        conn = dataSource.getConnection();
                        PreparedStatement ps = conn.prepareStatement(sql);
                        ps.setInt(1, custId);
                        Customer customer = null;
                        ResultSet rs = ps.executeQuery();
                        if (rs.next()) {
                                customer = new Customer(
                                        rs.getInt("CUST_ID"),
                                        rs.getString("NAME"),
                                        rs.getInt("Age")
                                );
                        }
                        rs.close();
                        ps.close();
                        return customer;
                } catch (SQLException e) {
                        throw new RuntimeException(e);
                } finally {
                        if (conn != null) {
                                try {
                                conn.close();
                                } catch (SQLException e) {}
                        }
                }
        }
}
```


Spring bean configuration

Create the Spring bean configuration file for customerDAO and datasource.
Crate File **Spring-Customer.xml** under ***src/main/resources/customer/***

```
<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-2.5.xsd">

        <bean id="customerDAO" class="com.dss.customer.dao.impl.JdbcCustomerDAO">
                <property name="dataSource" ref="dataSource" />
```

```
        </bean>

</beans>
```

Create File **Spring-Datasource.xml** under ***src/main/resources/database***

```
<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-2.5.xsd">

        <bean id="dataSource"

        class="org.springframework.jdbc.datasource.DriverManagerDataSource">

                <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver" />
                <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
                <property name="username" value="root" />
                <property name="password" value="password" />
        </bean>

</beans>
```

Craete File Spring-Module.xml under ***src/main/resources/***

```
<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-2.5.xsd">

        <import resource="database/Spring-Datasource.xml" />
        <import resource="customer/Spring-Customer.xml" />

</beans>
```

Run App.java

```
package com.dss.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.dss.customer.dao.CustomerDAO;
import com.dss.customer.model.Customer;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("Spring-Module.xml");

        CustomerDAO customerDAO = (CustomerDAO) context.getBean("customerDAO");
        Customer customer = new Customer(1, "durga",28);
        customerDAO.insert(customer);
         Customer customer1 = customerDAO.findByCustomerId(1);
        System.out.println(customer1);
```

```
    }
}
```

Steps to Generate Sample Web Application using Maven

Using webapp archetype generate the generic Web Application Structure.

```
C:\Development>mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes
 -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.0
 -DgroupId=com.dss -DartifactId=ServletMavenDSS -Dversion=1.0-SNAPSHOT
```

C:\Development>

Maven generates a directory structure like this:

ServletMavenDSS
ServletMavenDSS /pom.xml
ServletMavenDSS/src
ServletMavenDSS/src/main
ServletMavenDSS/src/main/resources
ServletMavenDSS/src/main/webapp
ServletMavenDSS/src/main/webapp/index.jsp
ServletMavenDSS/src/main/webapp/WEB-INF
ServletMavenDSS/src/main/webapp/WEB-INF/web.xml

The project's pom.xml file is shown below

```
<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">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.dsstest</groupId>
  <artifactId>ServletMavenDSS</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>mywebtest 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>
  </dependencies>
  <build>
    <finalName>ServletMavenDSS</finalName>
  </build>
</project>
```

Type Command:

```
C:\Development> mvn install
```

Maven will compile all the source files place all the class files in target folder, Unittesting will be done, and generate war file place in target folder.

Take a war file and deploy it in server. It will display.
O/P: Hello World!