Apache's Maven Tutorial

APACHE MAVEN TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Maven Tutorial

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

This tutorial will teach you how to use Maven in your day-2-day life of any project development using Java, or any other programming language.

Audience

This tutorial has been prepared for the beginners to help them understand basic functionality of Maven tool. After completing this tutorial you will find yourself at a moderate level of expertise in using Apache Maven from where you can take yourself to next levels.

Prerequisites

We assume you are going to use Maven to handle enterprise level Java projects development. So it will be good if you have knowledge of software development, Java SE, overview of Java EE development and deployment process.

Copyright & Disclaimer Notice

Table of Contents

CHAPTER

1

Maven Overview

This chapter describes the basic detail about Maven, how it emerged, what are the strengths of Maven and why we should use Maven.

Maven is a project management and comprehension tool. Maven provides developers a complete build lifecycle framework. Development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle.

In case of multiple development teams environment, Maven can set-up the way to work as per standards in a very short time. As most of the project setups are simple and reusable, Maven makes life of developer easy while creating reports, checks, build and testing automation setups.

Maven provides developers ways to manage following:

- Builds
- Documentation
- Reporting
- Dependencies
- SCMs
- Releases
- Distribution
- mailing list

To summarize, Maven simplifies and standardizes the project build process. It handles compilation, distribution, documentation, team collaboration and other tasks seamlessly. Maven increases reusability and takes care of most of build related tasks.

Maven History

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 Objective

Maven primary goal is to provide developer

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

Maven project structure and contents are declared in an xml file, pom.xml referred as Project Object Model (POM), which is the fundamental unit of the entire Maven system. Refer to Maven POM section for more details.

Convention over Configuration

Maven uses *Convention* over *Configuration* which means developers are not required to create build process themselves.

Developers do not have to mention each and every configuration details. Maven provides sensible default behavior for projects. When a Maven project is created, Maven creates default project structure. Developer is only required to place files accordingly and he/she need not to define any configuration in pom.xml.

As an example, following table shows the default values for project source code files, resource files and other configurations. Assuming, **${basedir}** denotes the project location:

Item	Default
source code	${basedir}/src/main/java
resources	${basedir}/src/main/resources
Tests	${basedir}/target/classes
Complied byte code	${basedir}/src/test
distributable JAR	${basedir}/target

In order to build the project, Maven provides developers options to mention life-cycle goals and project dependencies (that rely on Maven pluging capabilities and on its default conventions). Much of the project management and build related tasks are maintained by Maven plugins.

Developers can build any given Maven project without need to understand how the individual plugins work. Refer to Maven Plug-ins section for more details.

CHAPTER

2

Maven Environment Setup

This section describes how to setup your system environment before you start doing using Maven framework.

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.

Step 1 - verify Java installation in your machine

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

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:~ joseph$ java -version

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

OS	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)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

If you do not have Java installed, install the Java Software Development Kit (SDK) fromhttp://www.oracle.com/technetwork/java/javase/downloads/index.html. We are assuming Java 1.6.0_21 as installed version for this tutorial.

Step 2: Set JAVA environment

Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine. For example

OS	Output
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME=/usr/local/java-current
Mac	export JAVA_HOME=/Library/Java/Home

Append Java compiler location to System Path.

OS	Output
Windows	Append the string ;C:\Program Files\Java\jdk1.6.0_21\bin to the end of the system variable, Path.
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

Verify Java Installation using **java -version** command explained above.

Step 3: Download Maven archive

Download Maven 2.2.1 from http://maven.apache.org/download.html

OS	Archive name
Windows	apache-maven-2.0.11-bin.zip
Linux	apache-maven-2.0.11-bin.tar.gz
Mac	apache-maven-2.0.11-bin.tar.gz

Step 4: Extract the Maven archive

Extract the archive, to the directory you wish to install Maven 2.2.1. The subdirectory apache-maven-2.2.1 will be created from the archive.

OS	Location (can be different based on your installation)
Windows	C:\Program Files\Apache Software Foundation\apache-maven-2.2.1

Linux	/usr/local/apache-maven
Mac	/usr/local/apache-maven

Step 5: Set Maven environment variables

Add M2_HOME, M2, MAVEN_OPTS to environment variables.

OS	Output
Windows	Set the environment variables using system properties. *M2_HOME=C:\Program Files\Apache Software Foundation\apache-maven-2.2.1* *M2=%M2_HOME%\bin* *MAVEN_OPTS=-Xms256m -Xmx512m*
Linux	Open command terminal and set environment variables. *export M2_HOME=/usr/local/apache-maven/apache-maven-2.2.1* *export M2=%M2_HOME%\bin* *export MAVEN_OPTS=-Xms256m -Xmx512m*
Mac	Open command terminal and set environment variables. *export M2_HOME=/usr/local/apache-maven/apache-maven-2.2.1* *export M2=%M2_HOME%\bin* *export MAVEN_OPTS=-Xms256m -Xmx512m*

Step 6: Add Maven bin directory location to system path

Now append M2 variable to System Path

OS	Output
Windows	Append the string ;%M2% to the end of the system variable, Path.
Linux	export PATH=$M2:$PATH
Mac	export PATH=$M2:$PATH

Step 8: Verify Maven installation

Now open console, execute the following **mvn** command.

OS	Task	Command
Windows	Open Command Console	c:\> mvn --version
Linux	Open Command Terminal	$ mvn --version
Mac	Open Terminal	machine:~ joseph$ mvn --version

Finally, verify the output of the above commands, which should be something as follows:

OS	Output
Windows	Apache Maven 2.2.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre

Linux	Apache Maven 2.2.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre
Mac	Apache Maven 2.2.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre

Congratulations! you are now all set to use Apache Maven for your projects.

CHAPTER

3

Maven POM

Maven POM stands for *Project Object Model*. It is fundamental Unit of Work in Maven.

It is an XML file. It always resides in the base directory of the project as pom.xml.

The POM contains information about the project and various configuration details used by Maven to build the project(s).

POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal. Some of the configuration that can be specified in the POM are following:

- project dependencies
- plugins
- goals
- build profiles
- project version
- developers
- mailing list

Before creating a POM, we should first decide the project **group** (groupId), its **name**(artifactId) and its version as these attributes help in uniquely identifying the project in repository.

Example 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/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.companyname.project-group</groupId>
```

```
    <artifactId&gtproject</artifactId>
    <version>1.0</version>

</project>
```

It should be noted that there should be a single POM file for each project.

- All POM files require the project element and three mandatory fields: groupId, artifactId,version.
- Projects notation in repository is groupId:artifactId:version.
- Root element of POM.xml is project and it has three major sub-nodes :

Node	**Description**
groupId	This is an Id of project's group. This is generally unique amongst an organization or a project. For example, a banking group com.company.bank has all bank related projects.
artifactId	This is an Id of the project. This is generally name of the project. For example, consumer-banking. Along with the groupId, the artifactId defines the artifact's location within the repository.
version	This is the version of the project.Along with the groupId, It is used within an artifact's repository to separate versions from each other. For example: *com.company.bank:consumer-banking:1.0* *com.company.bank:consumer-banking:1.1.*

Super POM

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.

An easy way to look at the default configurations of the super POM is by running the following command: **mvn help:effective-pom**

Create a pom.xml in any directory on your computer.Use the content of above mentioned example pom.

In example below, We've created a pom.xml in C:\MVN\project folder.

Now open command console, go the folder containing pom.xml and execute the following **mvn**command.

```
C:\MVN\project>mvn help:effective-pom
```

Maven will start processing and display the effective-pom.

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.project-group:project-name:jar:1.0
[INFO]    task-segment: [help:effective-pom] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [help:effective-pom {execution: default-cli}]
[INFO]

.....

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Thu Jul 05 11:41:51 IST 2012
[INFO] Final Memory: 6M/15M
[INFO] ------------------------------------------------------------------------
Effective POM displayed as result in console, after inheritance, interpolation,
and profiles are applied.
<?xml version="1.0" encoding="UTF-8"?>
<!-- ================================================================= -->
<!--                                                                   -->
<!-- Generated by Maven Help Plugin on 2012-07-05T11:41:51             -->
<!-- See: http://maven.apache.org/plugins/maven-help-plugin/           -->
<!--                                                                   -->
<!-- ================================================================= -->

<!-- ================================================================= -->
<!--                                                                   -->
<!-- Effective POM for project                                         -->
<!-- 'com.companyname.project-group:project-name:jar:1.0'              -->
<!--                                                                   -->
<!-- ================================================================= -->

<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 h
ttp://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.project-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <build>
    <sourceDirectory>C:\MVN\project\src\main\java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>C:\MVN\project\src\test\java</testSourceDirectory>
    <outputDirectory>C:\MVN\project\target\classes</outputDirectory>
    <testOutputDirectory>C:\MVN\project\target\test-classes</testOutputDirectory>
    <resources>
      <resource>
        <mergeId>resource-0</mergeId>
        <directory>C:\MVN\project\src\main\resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <mergeId>resource-1</mergeId>
        <directory>C:\MVN\project\src\test\resources</directory>
      </testResource>
    </testResources>
    <directory>C:\MVN\project\target</directory>
    <finalName>project-1.0</finalName>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-antrun-plugin</artifactId>
          <version>1.3</version>
        </plugin>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <version>2.2-beta-2</version>
        </plugin>
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>2.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-dependency-plugin</artifactId>
          <version>2.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId>
          <version>2.4</version>
        </plugin>
        <plugin>
          <artifactId>maven-ear-plugin</artifactId>
          <version>2.3.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-ejb-plugin</artifactId>
          <version>2.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-install-plugin</artifactId>
          <version>2.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>2.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-javadoc-plugin</artifactId>
          <version>2.5</version>
        </plugin>
        <plugin>
          <artifactId>maven-plugin-plugin</artifactId>
          <version>2.4.3</version>
        </plugin>
        <plugin>
          <artifactId>maven-rar-plugin</artifactId>
          <version>2.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-release-plugin</artifactId>
          <version>2.0-beta-8</version>
        </plugin>
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>2.3</version>
        </plugin>
        <plugin>
          <artifactId>maven-site-plugin</artifactId>
          <version>2.0-beta-7</version>
        </plugin>
        <plugin>
          <artifactId>maven-source-plugin</artifactId>
          <version>2.0.4</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.4.3</version>
        </plugin>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>2.1-alpha-2</version>
        </plugin>
      </plugins>
    </pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-help-plugin</artifactId>
        <version>2.1.1</version>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <url>http://repo1.maven.org/maven2</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
    </pluginRepository>
  </pluginRepositories>
  <reporting>
    <outputDirectory>C:\MVN\project\target/site</outputDirectory>
  </reporting>
</project>
```

In above pom.xml , you can see the default project source folders structure, output directory, plug-ins required, repositories, reporting directory which Maven will be using while executing the desired goals.

Maven pom.xml is also not required to be written manually.

Maven provides numerous archetype plugins to create projects which in order create the project structure and pom.xml

Details are mentioned in Maven plug-ins and Maven Creating Project Sections

CHAPTER

4

Maven Build Life Cycle

A *Build Lifecycle* is a well defined sequence of phases which define the order in which the goals are to be executed. Here phase represents a stage in life cycle. As an example, a typical *Maven Build Lifecycle* is consists of following sequence of phases

Phase	Handles	Description
prepare-resources	resource copying	Resource copying can be customized in this phase.
compile	compilation	Source code compilation is done in this phase.
package	packaging	This phase creates the JAR / WAR package as mentioned in packaging in POM.xml.
install	installation	This phase installs the package in local / remote maven repository.

There are always **pre** and **post** phases which can be used to register **goals** which must run prior to or after a particular phase.

When Maven starts building a project, it steps through a defined sequence of phases and executes goals which are registered with each phase. Maven has following three standard lifecycles:

- clean
- default(or build)
- site

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

The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The clean and package arguments are build phases while the *dependency:copy-dependencies* is a goal.

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

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

Clean Lifecycle

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.

In the following example, We'll attach maven-antrun-plugin:run goal to the pre-clean, clean, and post-clean phases. This will allow us to echo text messages displaying the phases of the clean lifecycle.

We've created a pom.xml in C:\MVN\project folder.

```
<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>
<build>
<plugins>
   <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-antrun-plugin</artifactId>
   <version>1.1</version>
   <executions>
      <execution>
         <id>id.pre-clean</id>
         <phase>pre-clean</phase>
         <goals>
            <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <echo>pre-clean phase</echo>
            </tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.clean</id>
         <phase>clean</phase>
         <goals>
          <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
```

```
                <echo>clean phase</echo>
             </tasks>
          </configuration>
       </execution>
       <execution>
          <id>id.post-clean</id>
          <phase>post-clean</phase>
          <goals>
             <goal>run</goal>
          </goals>
          <configuration>
             <tasks>
                <echo>post-clean phase</echo>
             </tasks>
          </configuration>
       </execution>
    </executions>
    </plugin>
</plugins>
</build>
</project>
```

Now open command console, go to the folder containing pom.xml and execute the following **mvn** command.

```
C:\MVN\project>mvn post-clean
```

Maven will start processing and display all the phases of clean life cycle

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [post-clean]
[INFO] ------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-clean}]
[INFO] Executing tasks
     [echo] pre-clean phase
[INFO] Executed tasks
[INFO] [clean:clean {execution: default-clean}]
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
     [echo] clean phase
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.post-clean}]
[INFO] Executing tasks
     [echo] post-clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012
[INFO] Final Memory: 4M/44M
[INFO] ------------------------------------------------------------------
```

You can try tuning **mvn clean** command which will display *pre-clean* and *clean*, nothing will be executed for *post-clean* phase.

Default (or Build) Lifecycle

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

Lifecycle Phase	Description
validate	Validates whether project is correct and all necessary information is available to complete the build process.
initialize	Initializes build state, for example set properties
generate-sources	Generate any source code to be included in compilation phase.
process-sources	Process the source code, for example, filter any value.
generate-resources	Generate resources to be included in the package.
process-resources	Copy and process the resources into the destination directory, ready for packaging phase.
compile	Compile the source code of the project.
process-classes	Post-process the generated files from compilation, for example to do bytecode enhancement/optimization on Java classes.
generate-test-sources	Generate any test source code to be included in compilation phase.
process-test-sources	Process the test source code, for example, filter any values.
test-compile	Compile the test source code into the test destination directory.
process-test-classes	Process the generated files from test code file compilation.
test	Run tests using a suitable unit testing framework(Junit is one).
prepare-package	Perform any operations necessary to prepare a package before the actual packaging.
package	Take the compiled code and package it in its distributable format, such as a JAR, WAR, or EAR file.
pre-integration-test	Perform actions required before integration tests are executed. For example, setting up the required environment.
integration-test	Process and deploy the package if necessary into an environment where integration tests can be run.
pre-integration-test	Perform actions required after integration tests have been executed. For example, cleaning up the environment.
verify	Run any check-ups to verify the package is valid and meets quality criteria's.
install	Install the package into the local repository, which can be used as a dependency in other projects locally.
deploy	Copies the final package to the remote repository for sharing with other developers and projects.

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

- When a phase is called via Maven command, for example 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).

In the following example, We'll attach maven-antrun-plugin:run goal to few of the phases of Build lifecycle. This will allow us to echo text messages displaying the phases of the lifecycle.

We've updated pom.xml in C:\MVN\project folder.

```
 <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>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
<executions>
   <execution>
      <id>id.validate</id>
      <phase>validate</phase>
      <goals>
         <goal>run</goal>
      </goals>
      <configuration>
         <tasks>
            <echo>validate phase</echo>
         </tasks>
      </configuration>
   </execution>
   <execution>
      <id>id.compile</id>
      <phase>compile</phase>
      <goals>
         <goal>run</goal>
      </goals>
      <configuration>
         <tasks>
            <echo>compile phase</echo>
         </tasks>
      </configuration>
   </execution>
   <execution>
      <id>id.test</id>
      <phase>test</phase>
      <goals>
         <goal>run</goal>
      </goals>
      <configuration>
         <tasks>
            <echo>test phase</echo>
         </tasks>
      </configuration>
   </execution>
   <execution>
         <id>id.package</id>
         <phase>package</phase>
         <goals>
            <goal>run</goal>
         </goals>
         <configuration>
         <tasks>
            <echo>package phase</echo>
         </tasks>
      </configuration>
   </execution>
   <execution>
      <id>id.deploy</id>
      <phase>deploy</phase>
      <goals>
         <goal>run</goal>
      </goals>
      <configuration>
      <tasks>
         <echo>deploy phase</echo>
      </tasks>
      </configuration>
   </execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

Now open command console, go the folder containing pom.xml and execute the following **mvn** command.

```
C:\MVN\project>mvn compile
```

Maven will start processing and display phases of build life cycle upto compile phase.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------
[INFO] [antrun:run {execution: id.validate}]
[INFO] Executing tasks
     [echo] validate phase
[INFO] Executed tasks
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\project\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [antrun:run {execution: id.compile}]
[INFO] Executing tasks
     [echo] compile phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Sat Jul 07 20:18:25 IST 2012
[INFO] Final Memory: 7M/64M
[INFO] ------------------------------------------------------------------
```

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

In the following example, We'll attach *maven-antrun-plugin:run* goal to all the phases of Site lifecycle. This will allow us to echo text messages displaying the phases of the lifecycle.

We've updated pom.xml in C:\MVN\project folder.

```
<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>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
   <executions>
      <execution>
         <id>id.pre-site</id>
         <phase>pre-site</phase>
         <goals>
            <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <echo>pre-site phase</echo>
            </tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.site</id>
         <phase>site</phase>
         <goals>
         <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <echo>site phase</echo>
            </tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.post-site</id>
         <phase>post-site</phase>
         <goals>
            <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <echo>post-site phase</echo>
            </tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.site-deploy</id>
         <phase>site-deploy</phase>
         <goals>
```

```
            <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <echo>site-deploy phase</echo>
            </tasks>
         </configuration>
      </execution>
   </executions>
</plugin>
</plugins>
</build>
</project>
```

Now open command console, go the folder containing pom.xml and execute the following **mvn** command.

```
C:\MVN\project>mvn site
```

Maven will start processing and display phases of site life cycle upto site phase.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [site]
[INFO] ------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-site}]
[INFO] Executing tasks
     [echo] pre-site phase
[INFO] Executed tasks
[INFO] [site:site {execution: default-site}]
[INFO] Generating "About" report.
[INFO] Generating "Issue Tracking" report.
[INFO] Generating "Project Team" report.
[INFO] Generating "Dependencies" report.
[INFO] Generating "Project Plugins" report.
[INFO] Generating "Continuous Integration" report.
[INFO] Generating "Source Repository" report.
[INFO] Generating "Project License" report.
[INFO] Generating "Mailing Lists" report.
[INFO] Generating "Plugin Management" report.
[INFO] Generating "Project Summary" report.
[INFO] [antrun:run {execution: id.site}]
[INFO] Executing tasks
     [echo] site phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Sat Jul 07 15:25:10 IST 2012
[INFO] Final Memory: 24M/149M
[INFO] ------------------------------------------------------------------
```

CHAPTER

5

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
Per User	Defined in Maven settings xml file (%USER_HOME%/.m2/settings.xml)
Global	Defined in Maven global settings xml file (%M2_HOME%/conf/settings.xml)

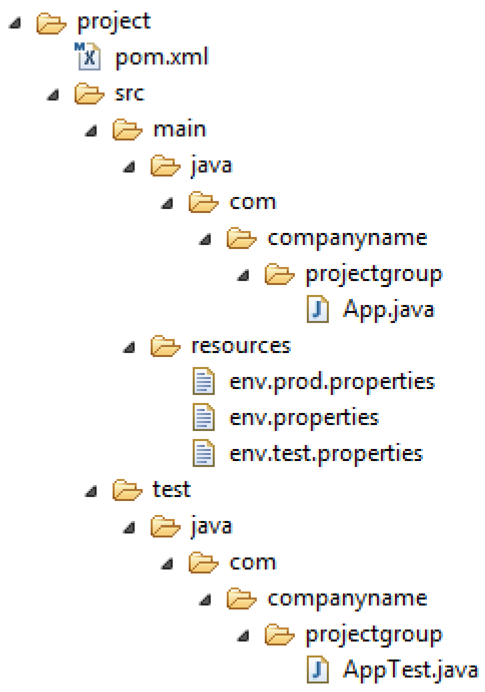
Profile Activation

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.

Profile Activation Examples

Let us assume following directory structure of your project:



Now, under *src/main/resources* there are three environment specific files:

File Name	Description
env.properties	default configuration used if no profile is mentioned.
env.test.properties	test configuration when test profile is used.
env.prod.properties	production configuration when prod profile is used.

Explicit Profile Activation

In the following example, We'll attach maven-antrun-plugin:run goal to test phase. This will allow us to echo text messages for different profiles. We will be using pom.xml to define different profiles and will activate profile at command console using maven command.

Assume, we've created following pom.xml in C:\MVN\project folder.

```
<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>
   <profiles>
      <profile>
      <id>test</id>
      <build>
      <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.1</version>
            <executions>
               <execution>
                  <phase>test</phase>
                  <goals>
                     <goal>run</goal>
                  </goals>
                  <configuration>
                  <tasks>
                     <echo>Using env.test.properties</echo>
                     <copy file="src/main/resources/env.test.properties"
tofile="${project.build.outputDirectory}/env.properties"/>
                  </tasks>
                  </configuration>
               </execution>
            </executions>
         </plugin>
      </plugins>
      </build>
      </profile>
   </profiles>
</project>
```

Now open command console, go to the folder containing pom.xml and execute the following **mvn**command. Pass the profile name as argument using -P option.

```
C:\MVN\project>mvn test -Ptest
```

Maven will start processing and display the result of test build profile.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] Copying 3 resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[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:\MVN\project\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\MVN\project\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
There are no tests to run.

Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

[INFO] [antrun:run {execution: default}]
```

```
[INFO] Executing tasks
     [echo] Using env.test.properties
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sun Jul 08 14:55:41 IST 2012
[INFO] Final Memory: 8M/64M
[INFO] ------------------------------------------------------------------
```

Now as an exercise, you can do the following steps

- Add another profile element to profiles element of pom.xml (copy existing profile element and paste it where profile elements ends).
- Update id of this profile element from test to normal.
- Update task section to echo env.properties and copy env.properties to target directory
- Again repeat above three steps, update id to prod and task section for env.prod.properties
- That's all. Now you've three build profiles ready (normal / test / prod).

Now open command console, go to the folder containing pom.xml and execute the following **mvn** commands. Pass the profile names as argument using -P option.

```
C:\MVN\project>mvn test -Pnormal

C:\MVN\project>mvn test -Pprod
```

Check the output of build to see the difference.

Profile Activation via Maven Settings

Open Maven **settings.xml** file available in %USER_HOME%/.m2 directory where **%USER_HOME%**represents user home directory. If settings.xml file is not there then create a new one.

Add test profile as an active profile using activeProfiles node as shown below in example

```
<settings 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/settings-1.0.0.xsd">
   <mirrors>
      <mirror>
         <id>maven.dev.snaponglobal.com</id>
         <name>Internal Artifactory Maven repository</name>
         <url>http://repo1.maven.org/maven2/</url>
         <mirrorOf>*</mirrorOf>
      </mirror>
   </mirrors>
   <activeProfiles>
      <activeProfile>test</activeProfile>
   </activeProfiles>
</settings>
```

Now open command console, go to the folder containing pom.xml and execute the following **mvn** command. Do not pass the profile name using -P option. Maven will display result of test profile being an active profile.

```
C:\MVN\project>mvn test
```

Profile Activation via Environment Variables

Now remove active profile from maven settings.xml and update the test profile mentioned in pom.xml. Add activation element to profile element as shown below.

The test profile will trigger when the system property "env" is specified with the value "test". Create a environment variable "env" and set its value as "test".

```
<profile>
   <id>test</id>
   <activation>
      <property>
         <name>env</name>
         <value>test</value>
      </property>
   </activation>
</profile>
```

Let's open command console, go to the folder containing pom.xml and execute the following **mvn** command.

```
C:\MVN\project>mvn test
```

Profile Activation via Operating System

Activation element to include OS details as shown below. This test profile will trigger when the system is windows XP.

```
<profile>
   <id>test</id>
   <activation>
      <os>
         <name>Windows XP</name>
         <family>Windows</family>
         <arch>x86</arch>
         <version>5.1.2600</version>
      </os>
   </activation>
</profile>
```

Now open command console, go to the folder containing pom.xml and execute the following **mvn** commands. Do not pass the profile name using -P option. Maven will display result of test profile being an active profile.

```
C:\MVN\project>mvn test
```

Profile Activation via Present/Missing File

Now activation element to include os details as shown below. Now test profile will trigger when *target/generated-sources/axistools/wsdl2java/com/companyname/group* is missing.

```
<profile>
   <id>test</id>
   <activation>
      <file>
         <missing>target/generated-
sources/axistools/wsdl2java/com/companyname/group</missing>
      </file>
   </activation>
</profile>
```

Now open command console, go to the folder containing pom.xml and execute the following **mvn** commands. Do not pass the profile name using -P option. Maven will display result of test profile being an active profile.

```
C:\MVN\project>mvn test
```

CHAPTER

6

Maven 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 repository 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:/MyLocalRepository</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).

CHAPTER

7

Maven Plugins

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:

```
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

```
mvn compiler:compile
```

Plugin Types

Maven provided following two types of Plugins:

Type	Description
Build plugins	They execute during the build and should be configured in the <build/> element of pom.xml
Reporting plugins	They execute during the site generation and they should be configured in the <reporting/> element of the pom.xml

Following is the list of few common plugins:

Plugin	Description
clean	Clean up target after the build. Deletes the target directory.
compiler	Compiles Java source files.
surefile	Run the JUnit unit tests. Creates test reports.
jar	Builds a JAR file from the current project.
war	Builds a WAR file from the current project.
javadoc	Generates Javadoc for the project.
antrun	Runs a set of ant tasks from any phase mentioned of the build.

Example

We've used **maven-antrun-plugin** extensively in our examples to print data on console. See Maven Build Profiles chapter. Let to understand it in a better way let's create a pom.xml in C:\MVN\project folder.

```
<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>
<build>
<plugins>
   <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-antrun-plugin</artifactId>
   <version>1.1</version>
   <executions>
      <execution>
         <id>id.clean</id>
         <phase>clean</phase>
         <goals>
            <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <echo>clean phase</echo>
            </tasks>
         </configuration>
      </execution>
   </executions>
   </plugin>
</plugins>
</build>
</project>
```

Next, open command console and go to the folder containing pom.xml and execute the following **mvn** command.

```
C:\MVN\project>mvn clean
```

Maven will start processing and display clean phase of clean life cycle

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
```

```
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [post-clean]
[INFO] ------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
     [echo] clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012
[INFO] Final Memory: 4M/44M
[INFO] ------------------------------------------------------------------
```

The above example illustrates the following key concepts:

- Plugins are specified in pom.xml using plugins element.
- Each plugin can have multiple goals.
- You can define phase from where plugin should starts its processing using its phase element. We've used clean phase.
- You can configure tasks to be executed by binding them to goals of plugin. We've bound echo task with run goal of maven-antrun-plugin.
- That's it, Maven will handle the rest. It will download the plugin if not available in local repository

CHAPTER

8

Create Java Project

Maven uses **archetype** plugins to create projects. To create a simple java application, we'll use maven-archetype-quickstart plugin. In example below, We'll create a maven based java application project in C:\MVN folder.

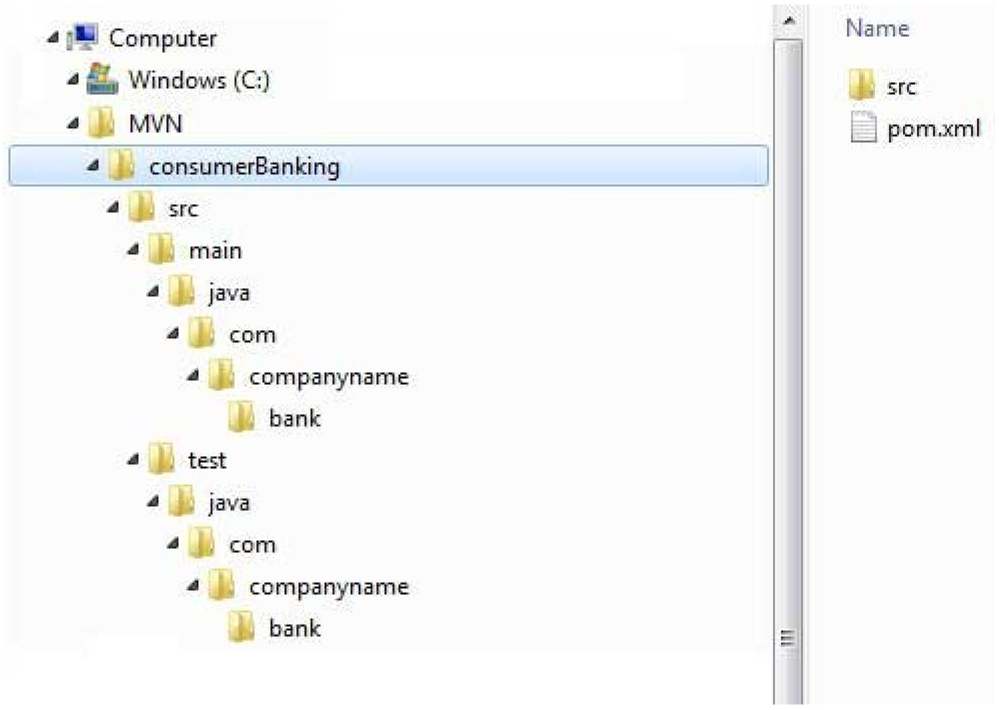
Let's open command console, go the C:\MVN directory and execute the following **mvn** command.

```
C:\MVN>mvn archetype:generate
-DgroupId=com.companyname.bank
-DartifactId=consumerBanking
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

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

```
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-quickstart:1.0
[INFO] -------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.bank
[INFO] Parameter: packageName, Value: com.companyname.bank
[INFO] Parameter: package, Value: com.companyname.bank
[INFO] Parameter: artifactId, Value: consumerBanking
[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\consumerBanking
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 14 seconds
[INFO] Finished at: Tue Jul 10 15:38:58 IST 2012
[INFO] Final Memory: 21M/124M
[INFO] ------------------------------------------------------------------
```

Now go to C:/MVN directory. You'll see a java application project created named consumerBanking (as specified in artifactId). Maven uses a standard directory layout as shown below:



Using above example, we can understand following key concepts

Folder Structure	Description
consumerBanking	contains src folder and pom.xml
src/main/java	contains java code files under the package structure (com/companyName/bank).
src/main/test	contains test java code files under the package structure (com/companyName/bank).
src/main/resources	it contains images/properties files (In above example, we need to create this structure manually).

If you see, Maven also created a sample Java Source file and Java Test file. Open C:\MVN\consumerBanking\src\main\java\com\companyname\bank folder, you will see App.java.

```
package com.companyname.bank;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Open C:\MVN\consumerBanking\src\test\java\com\companyname\bank folder, you will see AppTest.java.

```
package com.companyname.bank;

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()
    {
        assertTrue( true );
    }
}
```

Developers are required to place their files as mentioned in table above and Maven handles the all the build related complexities. In next section, we'll discuss how to build and test the project using Maven.

CHAPTER

9

Build & Test Java Project

What we have learnt in Project Creation chapter is how to create a Java application using Maven. Now we'll see how to build and test the application.

Go to C:/MVN directory where you've created your java application. Open *consumerBanking* folder.You will see the **POM.xml** file with following contents.

```
<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>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
         </dependency>
      </dependencies>
</project>
```

Here you can see, Maven already added Junit as test framework. By default Maven adds a source file**App.java** and a test file **AppTest.java** in its default directory structure discussed in previous chapter.

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

```
C:\MVN\consumerBanking>mvn clean package
```

Maven will start building the project.

```
[INFO] Scanning for projects...
[INFO] -------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]    task-segment: [clean, package]
[INFO] -------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\MVN\consumerBanking\target
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\
resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\MVN\consumerBanking\target\classes
[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:\MVN\consumerBanking\src\test\
resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\MVN\consumerBanking\target\test-classes
[INFO] [surefire:test {execution: default-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: 0.027 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\consumerBanking\target\
consumerBanking-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jul 10 16:52:18 IST 2012
[INFO] Final Memory: 16M/89M
[INFO] ------------------------------------------------------------------------
```

You've built your project and created final jar file, following are the key learning concepts

- We give maven two goals, first to clean the target directory (clean) and then package the project build output as jar(package).
- Packaged jar is available in consumerBanking\target folder as consumerBanking-1.0-SNAPSHOT.jar.
- Test reports are available in consumerBanking\target\surefire-reports folder.
- Maven compiled source code file(s) and then test source code file(s).
- Then Maven run the test cases.
- Finally Maven created the package.

Now open command console, go the C:\MVN\consumerBanking\target\classes directory and execute the following java command.

```
C:\MVN\consumerBanking\target\classes>java com.companyname.bank.App
```

You will see the result

```
Hello World!
```

Adding Java Source Files

Let's see how we can add additional Java files in our project. Open C:\MVN\consumerBanking\src\main\java\com\companyname\bank folder, create Util class in it as Util.java.

```
package com.companyname.bank;

public class Util
{
   public static void printMessage(String message){
            System.out.println(message);
   }
}
```

Update App class to use Util class.

```
package com.companyname.bank;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        Util.printMessage("Hello World!");
    }
}
```

Now open command console, go the C:\MVN\consumerBanking directory and execute the following **mvn** command.

```
C:\MVN\consumerBanking>mvn clean compile
```

After Maven build is successful, go the C:\MVN\consumerBanking\target\classes directory and execute the following java command.

```
C:\MVN\consumerBanking\target\classes>java com.companyname.bank.App
```

You will see the result

```
Hello World!
```

CHAPTER

10

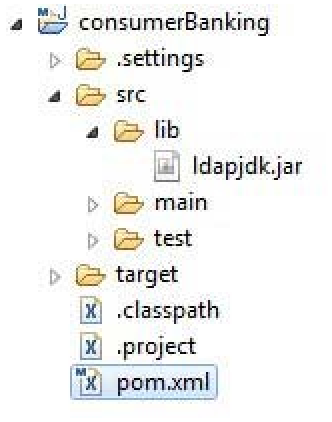
External Dependencies

Now as you know Maven does the dependency management using concept of Maven Repositories. But what happens if dependency is not available in any of remote repositories and central repository? Maven provides answer for such scenario using concept of **External Dependency**.

For an example, let us do the following changes to project created in Maven Creating Project section.

- Add lib folder to src folder
- Copy any jar into the lib folder. We've used ldapjdk.jar, which is a helper library for LDAP operations.

Now our project structure should look like following:



```
consumerBanking
    .settings
    src
        lib
            ldapjdk.jar
        main
        test
    target
    .classpath
    .project
    pom.xml
```

Here you are having your own library specific to project, which is very usual case and it can contain jars which may not be available in any repository for maven to download from. If your code is using this library with Maven then Maven build will fail because it cannot download or refer to this library during compilation phase.

To handle the situation, let's add this external dependency to maven **pom.xml** using following way.

```
<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.bank</groupId>
   <artifactId>consumerBanking</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>consumerBanking</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>ldapjdk</groupId>
         <artifactId>ldapjdk</artifactId>
         <scope>system</scope>
         <version>1.0</version>
         <systemPath>${basedir}\src\lib\ldapjdk.jar</systemPath>
      </dependency>
   </dependencies>

</project>
```

Look at the second dependency element under dependencies in above example which clears following key concepts about **External Dependency**.

- External dependencies (library jar location) can be configured in pom.xml in same way as other dependencies.
- Specify groupId same as name of the library.
- Specify artifactId same as name of the library.
- Specify scope as system.
- Specify system path relative to project location.

Hope now you are clear about external dependencies and you will be able to specify external dependencies in your Maven project.

CHAPTER

11

Create Project Documentation

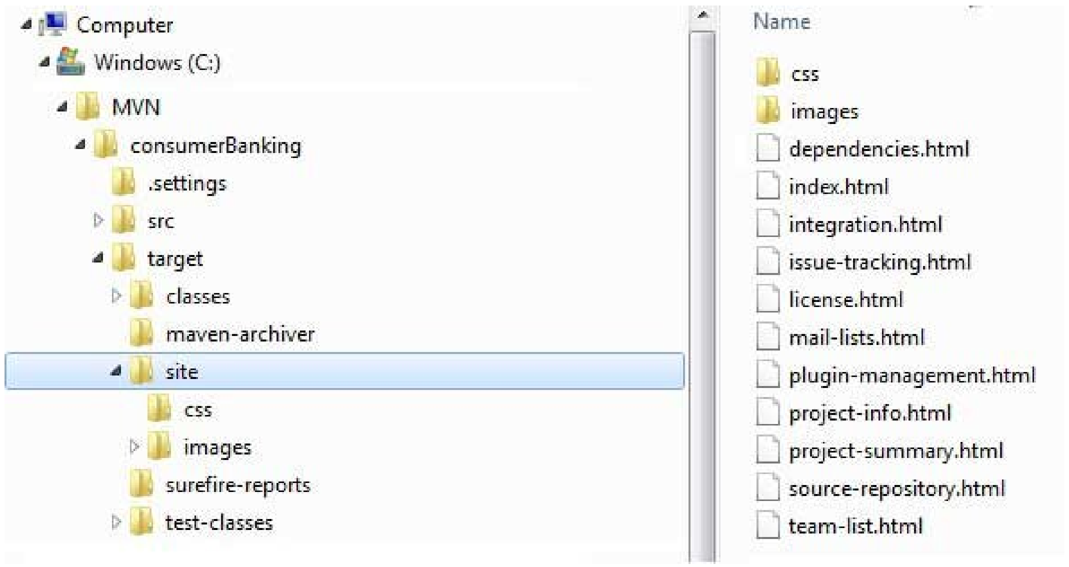
This chapter will teach you how to create documentation of the application in one go. So let's start, go to C:/MVN directory where you had created your java **consumerBanking** application. Open*consumerBanking* folder and execute the following **mvn** command.

```
C:\MVN>mvn site
```

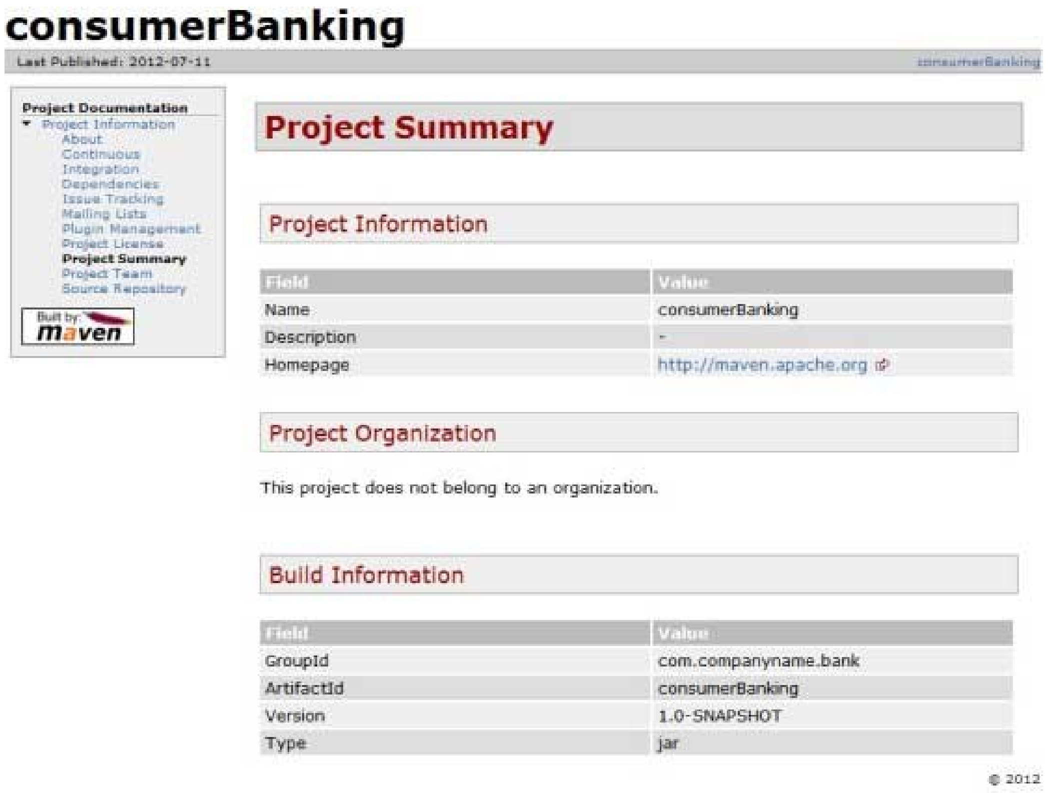
Maven will start building the project.

```
[INFO] Scanning for projects...
[INFO] -------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]    task-segment: [site]
[INFO] -------------------------------------------------------------------
[INFO] [site:site {execution: default-site}]
[INFO] artifact org.apache.maven.skins:maven-default-skin:
checking for updates from central
[INFO] Generating "About" report.
[INFO] Generating "Issue Tracking" report.
[INFO] Generating "Project Team" report.
[INFO] Generating "Dependencies" report.
[INFO] Generating "Continuous Integration" report.
[INFO] Generating "Source Repository" report.
[INFO] Generating "Project License" report.
[INFO] Generating "Mailing Lists" report.
[INFO] Generating "Plugin Management" report.
[INFO] Generating "Project Summary" report.
[INFO] -------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] -------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 11 18:11:18 IST 2012
[INFO] Final Memory: 23M/148M
[INFO] -------------------------------------------------------------------
```

That's it. Your project documentation is ready. Maven has created a site within target directory.



Open C:\MVN\consumerBanking\target\site folder. Click on index.html to see the documentation.


consumerBanking

Last Published: 2012-07-11 consumerBanking

Project Documentation
- Project Information
  - About
  - Continuous Integration
  - Dependencies
  - Issue Tracking
  - Mailing Lists
  - Plugin Management
  - Project License
  - **Project Summary**
  - Project Team
  - Source Repository

Built by: maven

Project Summary

Project Information

Field	Value
Name	consumerBanking
Description	-
Homepage	http://maven.apache.org

Project Organization

This project does not belong to an organization.

Build Information

Field	Value
GroupId	com.companyname.bank
ArtifactId	consumerBanking
Version	1.0-SNAPSHOT
Type	jar

© 2012

Maven creates the documentation using a documentation-processing engine called Doxia which reads multiple source formats into a common document model. To write documentation for your project, you can write your content in a following few commonly used formats which are parsed by Doxia.

Format Name	Description	Reference

APT	A Plain Text document format	http://maven.apache.org/doxia/format.html
XDoc	A Maven 1.x documentation format	http://jakarta.apache.org/site/jakarta-site2.html
FML	Used for FAQ documents	http://maven.apache.org/doxia/references/fml-format.html
XHTML	Extensible HTML	http://en.wikipedia.org/wiki/XHTML

CHAPTER

12

Maven Project Templates

Maven provides users, a very large list of different types of project templates (614 in numbers) using concept of **Archetype**. Maven helps users to quickly start a new java project using following command

```
mvn archetype:generate
```

What is Archetype?

Archetype is a Maven plugin whose task is to create a project structure as per its template. We are going to use *quickstart* archetype plugin to create a simple java application here.

Using Project Template

Let's open command console, go the **C:\ > MVN** directory and execute the following **mvn** command

```
C:\MVN>mvn archetype:generate
```

Maven will start processing and will ask to choose required archetype

```
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
...
600: remote -> org.trailsframework:trails-archetype (-)
601: remote -> org.trailsframework:trails-secure-archetype (-)
602: remote -> org.tynamo:tynamo-archetype (-)
603: remote -> org.wicketstuff.scala:wicket-scala-archetype (-)
604: remote -> org.wicketstuff.scala:wicketstuff-scala-archetype
Basic setup for a project that combines Scala and Wicket,
depending on the Wicket-Scala project.
Includes an example Specs test.)
605: remote -> org.wikbook:wikbook.archetype (-)
606: remote -> org.xaloon.archetype:xaloon-archetype-wicket-jpa-glassfish (-)
607: remote -> org.xaloon.archetype:xaloon-archetype-wicket-jpa-spring (-)
608: remote -> org.xwiki.commons:xwiki-commons-component-archetype
```

```
(Make it easy to create a maven project for creating XWiki Components.)
609: remote -> org.xwiki.rendering:xwiki-rendering-archetype-macro
(Make it easy to create a maven project for creating XWiki Rendering Macros.)
610: remote -> org.zkoss:zk-archetype-component (The ZK Component archetype)
611: remote -> org.zkoss:zk-archetype-webapp (The ZK wepapp archetype)
612: remote -> ru.circumflex:circumflex-archetype (-)
613: remote -> se.vgregion.javg.maven.archetypes:javg-minimal-archetype (-)
614: remote -> sk.seges.sesam:sesam-annotation-archetype (-)
Choose a number or apply filter
(format: [groupId:]artifactId, case sensitive contains): 203:
```

Press Enter to choose to default option(203: maven-archetype-quickstart)

Maven will ask for particular version of archetype

```
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:

1: 1.0-alpha-1

2: 1.0-alpha-2

3: 1.0-alpha-3

4: 1.0-alpha-4

5: 1.0

6: 1.1

Choose a number: 6:
```

Press Enter to choose to default option(6: maven-archetype-quickstart:1.1)

Maven will ask for project details. Enter project details as asked. Press Enter if default value is provided. You can override them by entering your own value.

```
Define value for property 'groupId': : com.companyname.insurance

Define value for property 'artifactId': : health

Define value for property 'version': 1.0-SNAPSHOT:

Define value for property 'package': com.companyname.insurance:
```

Maven will ask for project details confirmation. Press enter or press Y

```
Confirm properties configuration:

groupId: com.companyname.insurance

artifactId: health

version: 1.0-SNAPSHOT

package: com.companyname.insurance

Y:
```

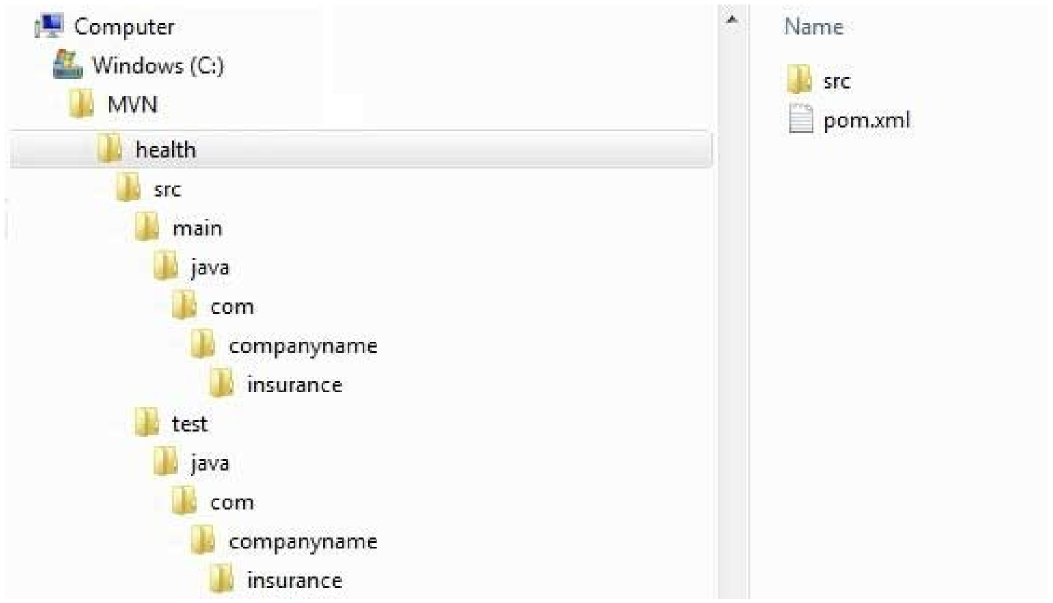
Now Maven will start creating project structure and will display the following:

```
[INFO] ---------------------------------------------------------------------
```

```
[INFO] Using following parameters for creating project
from Old (1.x) Archetype: maven-archetype-quickstart:1.1
[INFO] -------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.insurance
[INFO] Parameter: packageName, Value: com.companyname.insurance
[INFO] Parameter: package, Value: com.companyname.insurance
[INFO] Parameter: artifactId, Value: health
[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\health
[INFO] -------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] -------------------------------------------------------------------
[INFO] Total time: 4 minutes 12 seconds
[INFO] Finished at: Fri Jul 13 11:10:12 IST 2012
[INFO] Final Memory: 20M/90M
[INFO] -------------------------------------------------------------------
```

Created Project

Now go to **C:\ > MVN** directory. You'll see a java application project created named **health** which was given as *artifactId* at the time of project creation. Maven will create a standard directory layout for the project as shown below:



Created POM.xml

Maven generates a POM.xml file for the project as listed 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/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.companyname.insurance</groupId>
   <artifactId>health</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <name>health</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>
</project>
```

Created App.java

Maven generates sample java source file, App.java for the project as listed below:

Location: **C:\ > MVN > health > src > main > java > com > companyname > insurance > App.java**

```
package com.companyname.insurance;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Created AppTest.java

Maven generates sample java source test file, AppTest.java for the project as listed below:

Location: **C:\ > MVN > health > src > test > java > com > companyname > insurance > AppTest.java**

```
package com.companyname.insurance;

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()
    {
        assertTrue( true );
    }
}
```

That's it. Now you can see the power of Maven. You can create any kind of project using single command in maven and can kick-start your development.

CHAPTER

13

Maven Snapshot

A large software application generally consists of multiple modules and it is common scenario where multiple teams are working on different modules of same application. For example consider a team is working on the front end of the application as app-ui project (app-ui.jar:1.0) and they are using data-service project (data-service.jar:1.0).

Now it may happen that team working on data-service is undergoing bug fixing or enhancements at rapid pace and the they are releasing the library to remote repository almost every other day.

Now if data-service team uploads a new version every other day then following problem will arise

- data-service team should tell app-ui team every time when they have released an updated code.
- app-ui team required to update their pom.xml regularly to get the updated version

To handle such kind of situation, **SNAPSHOT** concept comes into play.

What is SNAPSHOT?

SNAPSHOT is a special version that indicates a current development copy. Unlike regular versions, Maven checks for a new SNAPSHOT version in a remote repository for every build.

Now data-service team will release SNAPSHOT of its updated code every time to repository say data-service:1.0-SNAPSHOT replacing a older SNAPSHOT jar.

Snapshot vs Version

In case of Version, if Maven once downloaded the mentioned version say data-service:1.0, it will never try to download a newer 1.0 available in repository. To download the updated code, data-service version is be upgraded to 1.1.

In case of SNAPSHOT, Maven will automatically fetch the latest SNAPSHOT (data-service:1.0-SNAPSHOT) everytime app-ui team build their project.

app-ui pom.xml

app-ui project is using 1.0-SNAPSHOT of data-service

```
<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>app-ui</groupId>
   <artifactId>app-ui</artifactId>
   <version>1.0</version>
   <packaging>jar</packaging>
   <name>health</name>
   <url>http://maven.apache.org</url>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>
   <dependencies>
      <dependency>
      <groupId>data-service</groupId>
         <artifactId>data-service</artifactId>
         <version>1.0-SNAPSHOT</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

data-service pom.xml

data-service project is releasing 1.0-SNAPSHOT for every minor change

```
<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>data-service</groupId>
   <artifactId>data-service</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <name>health</name>
   <url>http://maven.apache.org</url>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>
   </project>
```

Although, In case of SNAPSHOT, Maven automatically fetches the latest SNAPSHOT on daily basis. You can force maven to download latest snapshot build using -U switch to any maven command.

```
mvn clean package -U
```

Let's open command console, go the **C:\ > MVN > app-ui** directory and execute the following **mvn**command.

```
C:\MVN\app-ui>mvn clean package -U
```

Maven will start building the project after downloading latest SNAPSHOT of data-service.

```
[INFO] Scanning for projects...
[INFO] -------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]    task-segment: [clean, package]
[INFO] -------------------------------------------------------------------
[INFO] Downloading data-service:1.0-SNAPSHOT
[INFO] 290K downloaded.
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\MVN\app-ui\target
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\app-ui\src\main\
resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\MVN\app-ui\target\classes
[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:\MVN\app-ui\src\test\
resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\MVN\app-ui\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\MVN\app-ui\target\
surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.companyname.bank.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\app-ui\target\
app-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jul 10 16:52:18 IST 2012
[INFO] Final Memory: 16M/89M
[INFO] ------------------------------------------------------------------------
```

CHAPTER

14

Maven Build Automation

Build Automation defines the scenario where dependent project(s) build process gets started once the project build is successfully completed, in order to ensure that dependent project(s) is/are stable.

Example

Consider a team is developing a project bus-core-api on which two other projects app-web-ui and app-desktop-ui are dependent.

app-web-ui project is using 1.0-SNAPSHOT of bus-core-api project

```
<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>app-web-ui</groupId>
   <artifactId>app-web-ui</artifactId>
   <version>1.0</version>
   <packaging>jar</packaging>
   <dependencies>
      <dependency>
      <groupId>bus-core-api</groupId>
         <artifactId>bus-core-api</artifactId>
         <version>1.0-SNAPSHOT</version>
      </dependency>
   </dependencies>
</project>
```

app-desktop-ui project is using 1.0-SNAPSHOT of bus-core-api project

```
<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>app-desktop-ui</groupId>
   <artifactId>app-desktop-ui</artifactId>
   <version>1.0</version>
   <packaging>jar</packaging>
   <dependencies>
      <dependency>
      <groupId>bus-core-api</groupId>
         <artifactId>bus-core-api</artifactId>
         <version>1.0-SNAPSHOT</version>
      </dependency>
   </dependencies>
</project>
```

bus-core-api project

```
<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>bus-core-api</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
</project>
```

Now teams of app-web-ui and app-desktop-ui projects require that their build process should kick off whenever bus-core-api project changes.

Using snapshot ensures that the latest bus-core-api project should be used but to meet above requirement we need to do something extra.

We've two ways

- Add a post-build goal in bus-core-api pom to kick-off app-web-ui and app-desktop-ui builds.
- Use a Continuous Integration (CI) Server like Hudson to manage build automation automatically.

Using Maven

Update bus-core-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>bus-core-api</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <build>
   <plugins>
   <plugin>
   <artifactId>maven-invoker-plugin</artifactId>
   <version>1.6</version>
      <configuration>
         <debug>true</debug>
         <pomIncludes>
            <pomInclude>app-web-ui/pom.xml</pomInclude>
            <pomInclude>app-desktop-ui/pom.xml</pomInclude>
         </pomIncludes>
      </configuration>
      <executions>
```

```
          <execution>
             <id>build</id>
             <goals>
                <goal>run</goal>
             </goals>
          </execution>
       </executions>
    </plugin>
    </plugins>
    <build>
</project>
```

Let's open command console, go the **C:\ > MVN > bus-core-api** directory and execute the following **mvn** command.

```
C:\MVN\bus-core-api>mvn clean package -U
```

Maven will start building the project bus-core-api.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
[INFO] Building bus-core-api
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------------
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\bus-core-ui\target\
bus-core-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
```

Once bus-core-api build is successful, Maven will start building app-web-ui project

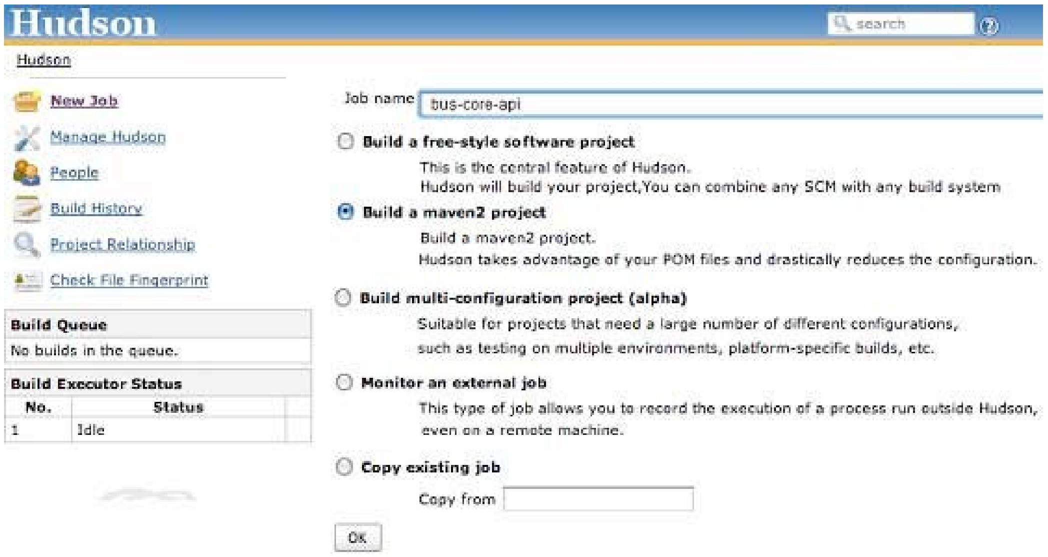
```
[INFO] ------------------------------------------------------------------
[INFO] Building app-web-ui
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\app-web-ui\target\
app-web-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
```

Once app-web-ui build is successful, Maven will start building app-desktop-ui project

```
[INFO] ------------------------------------------------------------------
[INFO] Building app-desktop-ui
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\app-desktop-ui\target\
app-desktop-ui-1.0-SNAPSHOT.jar
[INFO] -------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] -------------------------------------------------------------------
```

Using Continuous Integration Service with Maven

Using a CI Server is more preferable as developers are not required to updated bus-core-api project pom every time a new project, for example app-mobile-ui is added as dependent project on bus-core-api project. Hudson automatically manages build automation using Maven dependency management.



Build Queue

No builds in the queue.

Build Executor Status

No.	Status
1	Idle

Job name: bus-core-api

( ) **Build a free-style software project**
This is the central feature of Hudson.
Hudson will build your project,You can combine any SCM with any build system

[✓] **Build a maven2 project**
Build a maven2 project.
Hudson takes advantage of your POM files and drastically reduces the configuration.

( ) **Build multi-configuration project (alpha)**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

( ) **Monitor an external job**
This type of job allows you to record the execution of a process run outside Hudson, even on a remote machine.

( ) **Copy existing job**
Copy from

Hudson considers each project build as job. Once a project code is checked-in to SVN (or any Source Management Tool mapped to Hudson), Hudson starts its build job and once this job get completed, it start other dependent jobs(other dependent projects) automatically.

In above example, when bus-core-ui source code is updated in SVN, Hudson starts its build. Once build is successful. Hudson looks for dependent projects automatically, and starts building app-web-ui and app-desktop-ui projects.

CHAPTER

15

Dependency Management

One of the core features of Maven is Dependency Management. Managing dependencies become difficult task once we've to deal with multi-module projects (consists of hundreds of modules/sub-projects). Maven provides a high degree of control to manage such scenarios.

Transitive Dependencies Discovery

It is pretty often a case, when a library say A depends upon other library say B. In case another project C want to use A then that project requires to use library B too.

Maven helps to avoid such requirement to discover all the libraries required. Maven does so by reading project files(pom.xml) of dependencies, figure out their dependencies and so on.

We only need to define direct dependency in each project pom. Maven handles the rest automatically.

With transitive dependencies, the graph of included libraries can quickly grow to a large extent. Cases can arise when there are duplicate libraries. Maven provides few features to control extent of transitive dependencies

Feature	**Description**
Dependency mediation	Determines what version of a dependency is to be used when multiple versions of an artifact are encountered. If two dependency versions are at the same depth in the dependency tree, the first declared dependency will be used.
Dependency management	Directly specify the versions of artifacts to be used when they are encountered in transitive dependencies. For an example project C can include B as a dependency in its dependencyManagement section and directly control which version of B is to be used when it is ever referenced.
Dependency scope	Includes dependencies as per the current stage of the build
Excluded dependencies	Any transitive dependency can be exclude using "exclusion" element. As example, A depends upon B and B depends upon C then A can mark C as excluded.
Optional dependencies	Any transitive dependency can be marked as optional using

	"optional" element. As example, A depends upon B and B depends upon C. Now B marked C as optional. Then A will not use C.

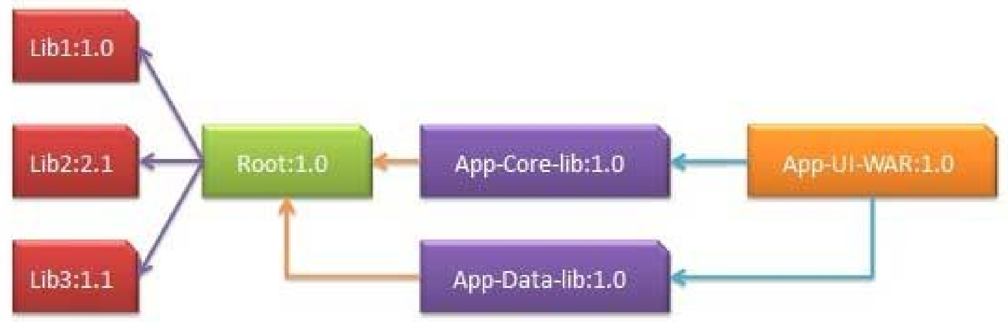
Dependency Scope

Transitive Dependencies Discovery can be restricted using various Dependency Scope as mentioned below

Scope	Description
compile	This scope indicates that dependency is available in classpath of project. It is default scope.
provided	This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime.
runtime	This scope indicates that dependency is not required for compilation, but is required during execution.
test	This scope indicates that the dependency is only available for the test compilation and execution phases.
system	This scope indicates that you have to provide the system path.
import	This scope is only used when dependency is of type pom. This scopes indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section.

Dependency Management

Usually, we've a set of project under a common project. In such case, we can create a common pom having all the common dependencies and then make this pom parent of sub-project's poms. Following example will help you understand this concept



Following are the details of above dependency graph

- App-UI-WAR depends upon App-Core-lib and App-Data-lib.
- Root is parent of App-Core-lib and App-Data-lib.
- Root defines Lib1,lib2, Lib3 as dependencies in its dependency section.

App-UI-WAR

```
<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.groupname</groupId>
      <artifactId>App-UI-WAR</artifactId>
      <version>1.0</version>
      <packaging>war</packaging>
      <dependencies>
         <dependency>
            <groupId>com.companyname.groupname</groupId>
            <artifactId>App-Core-lib</artifactId>
            <version>1.0</version>
         </dependency>
      </dependencies>
      <dependencies>
         <dependency>
            <groupId>com.companyname.groupname</groupId>
            <artifactId>App-Data-lib</artifactId>
            <version>1.0</version>
         </dependency>
      </dependencies>
</project>
```

App-Core-lib

```
<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">
      <parent>
         <artifactId>Root</artifactId>
         <groupId>com.companyname.groupname</groupId>
         <version>1.0</version>
      </parent>
      <modelVersion>4.0.0</modelVersion>
      <groupId>com.companyname.groupname</groupId>
      <artifactId>App-Core-lib</artifactId>
      <version>1.0</version>
      <packaging>jar</packaging>
</project>
```

App-Data-lib

```
<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">
      <parent>
         <artifactId>Root</artifactId>
         <groupId>com.companyname.groupname</groupId>
         <version>1.0</version>
      </parent>
      <modelVersion>4.0.0</modelVersion>
      <groupId>com.companyname.groupname</groupId>
      <artifactId>App-Data-lib</artifactId>
      <version>1.0</version>
      <packaging>jar</packaging>
</project>
```

Root

```
<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.groupname</groupId>
      <artifactId>Root</artifactId>
      <version>1.0</version>
           <packaging>pom</packaging>
      <dependencies>
         <dependency>
            <groupId>com.companyname.groupname1</groupId>
            <artifactId>Lib1</artifactId>
            <version>1.0</version>
         </dependency>
      </dependencies>
      <dependencies>
         <dependency>
            <groupId>com.companyname.groupname2</groupId>
            <artifactId>Lib2</artifactId>
            <version>2.1</version>
         </dependency>
      </dependencies>
      <dependencies>
         <dependency>
            <groupId>com.companyname.groupname3</groupId>
            <artifactId>Lib3</artifactId>
            <version>1.1</version>
         </dependency>
      </dependencies>
</project>
```

Now when we build App-UI-WAR project, Maven will discover all the dependencies by traversing the dependency graph and build the application.

From above example, we can learn following key concepts

- Common dependencies can be placed at single place using concept of parent pom. Dependencies of App-Data-lib and App-Core-lib project are listed in Root project (See the packaging type of Root. It is POM).
- There is no need to specify Lib1, lib2, Lib3 as dependency in App-UI-WAR. Maven use the Transitive Dependency Mechanism to manage such details.

CHAPTER

16

Deployment Automation

In project development, 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.

Update Project POM.xml

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

For Example: bus-core-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>bus-core-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>[bus-core-api-release-checkin]-<
               /scmCommentPrefix>
            </configuration>
         </plugin>
      </plugins>
   </build>
</project>
```

In Pom.xml, following are the important elements we've 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.

Let's open command console, go the **C:\ > MVN >bus-core-api** directory and execute the following **mvn** command.

```
C:\MVN\bus-core-api>mvn release:prepare
```

Maven will start building the project. Once build is successful run the following **mvn** command.

```
C:\MVN\bus-core-api>mvn release:perform
```

Once build is successful you can verify the uploaded JAR file in your repository.

CHAPTER

17

Maven Web Application

This chapter will teach you how to manage a web based project using version control system **Maven**. Here you will learn how to create/build/deploy and run a web application:

Create Web Application

To create a simple java web application, we'll use maven-archetype-webapp plugin. So let's open command console, go the C:\MVN directory and execute the following **mvn** command.

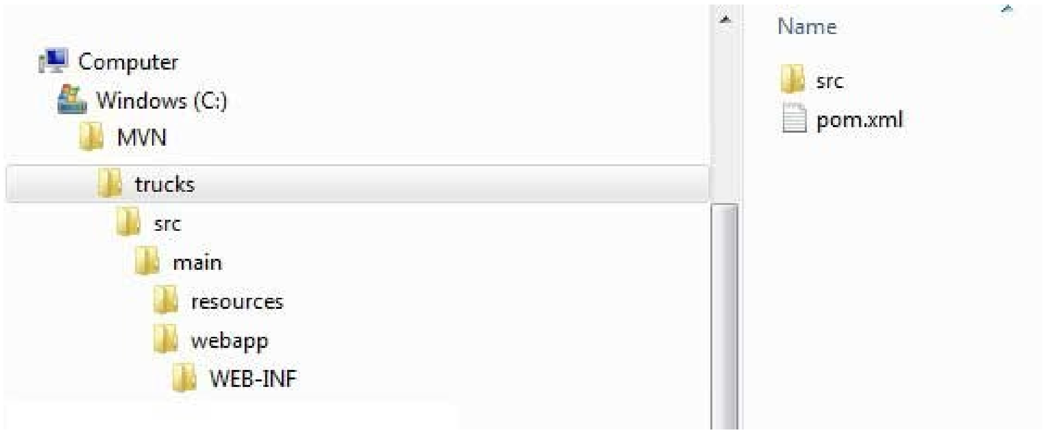
```
C:\MVN>mvn archetype:generate

-DgroupId=com.companyname.automobile

-DartifactId=trucks

-DarchetypeArtifactId=maven-archetype-webapp

-DinteractiveMode=false
```

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

```
[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] -------------------------------------------------------------------
```

Now go to C:/MVN directory. You'll see a java application project created named trucks (as specified in artifactId).



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

Folder Structure	**Description**
trucks	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>trucks</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>trucks</finalName>
   </build>
```

```
</project>
```

If you see, Maven also created a sample JSP Source file

Open **C:\ > MVN > trucks > src > main > webapp >** folder, you will see index.jsp.

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

Build Web Application

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

```
C:\MVN\trucks>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:\MVN\trucks\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:\MVN\trucks\target\trucks]
[INFO] Processing war project
[INFO] Copying webapp resources[C:\MVN\trucks\src\main\webapp]
[INFO] Webapp assembled in[77 msecs]
[INFO] Building war: C:\MVN\trucks\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] -------------------------------------------------------------------
```

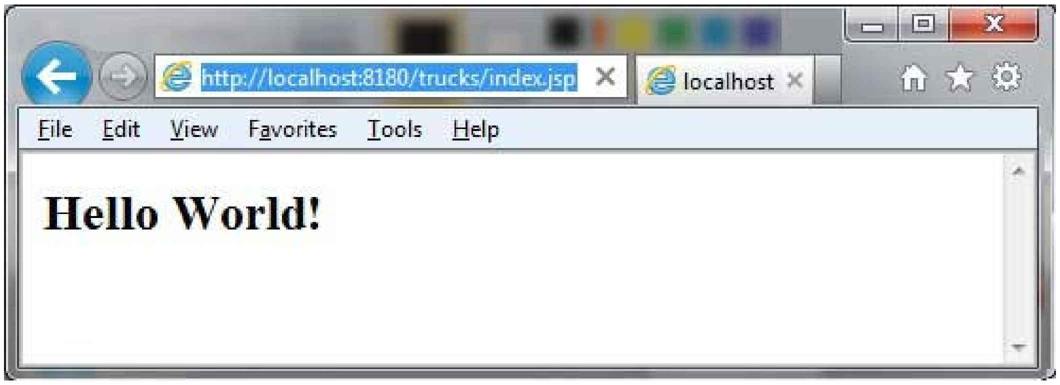
Deploy Web Application

Now copy the **trucks.war** created in **C:\ > MVN > trucks > 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.


http://localhost:8180/trucks/index.jsp
localhost
File Edit View Favorites Tools Help

Hello World!

CHAPTER

18

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.
- It provides wizards for creating new Maven projects, pom.xml and to enable Maven support on existing projects
- It provides quick search for dependencies in remote Maven repositories

Installing m2eclipse plugin

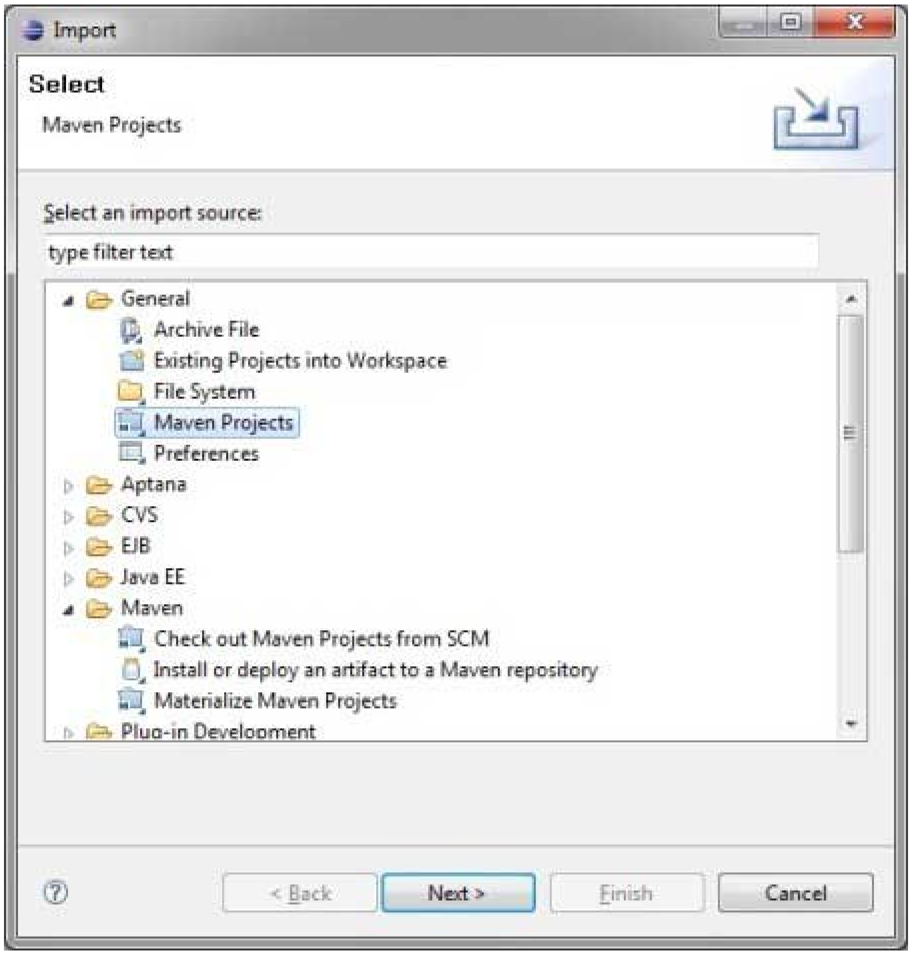
Use one of the following links to install m2eclipse:

Eclipse	**URL**
Eclipse 3.5 (Gallileo)	Installing m2eclipse in Eclipse 3.5 (Gallileo)
Eclipse 3.6 (Helios)	Installing m2eclipse in Eclipse 3.6 (Helios)

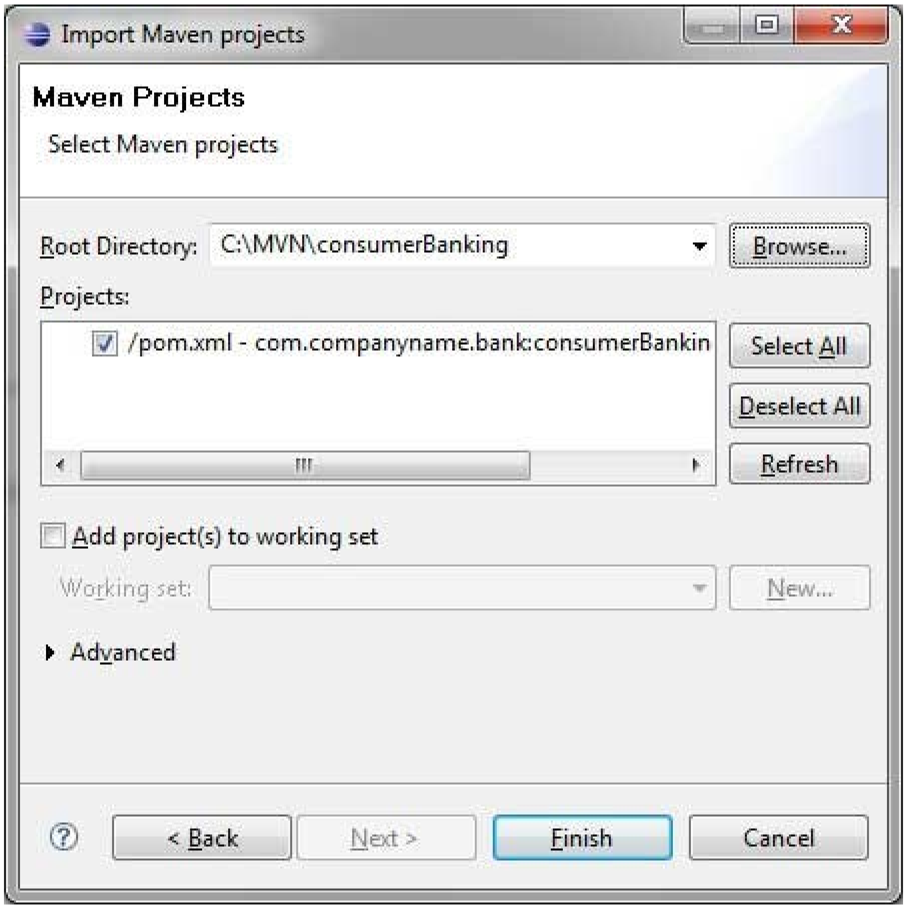
Following example will help you to leverage benefits of integrating Eclipse and maven.

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.


Import Maven projects

Maven Projects

Select Maven projects

Root Directory: C:\MVN\consumerBanking

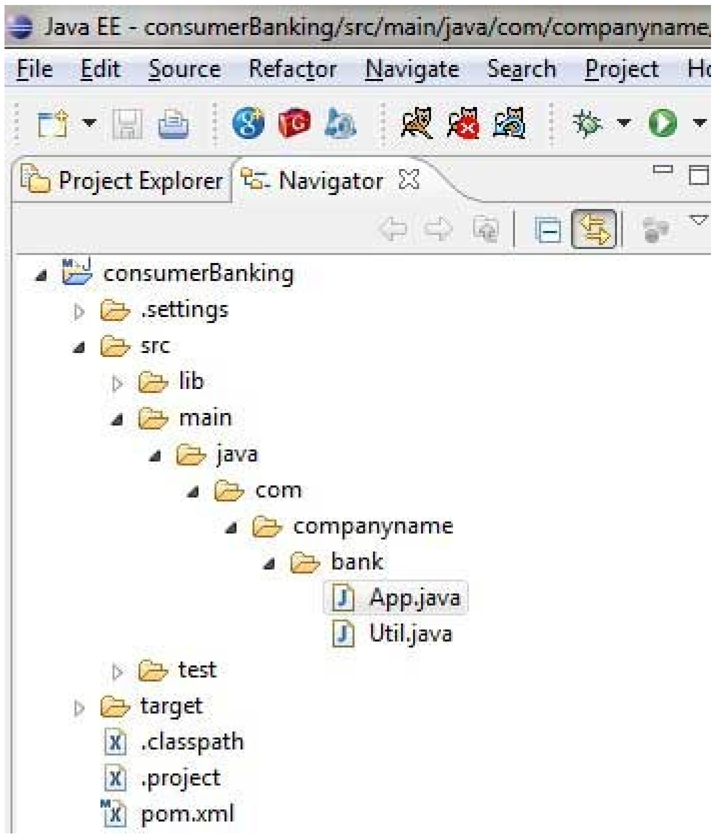
Projects:

[✓] /pom.xml - com.companyname.bank:consumerBankin

[ ] Add project(s) to working set

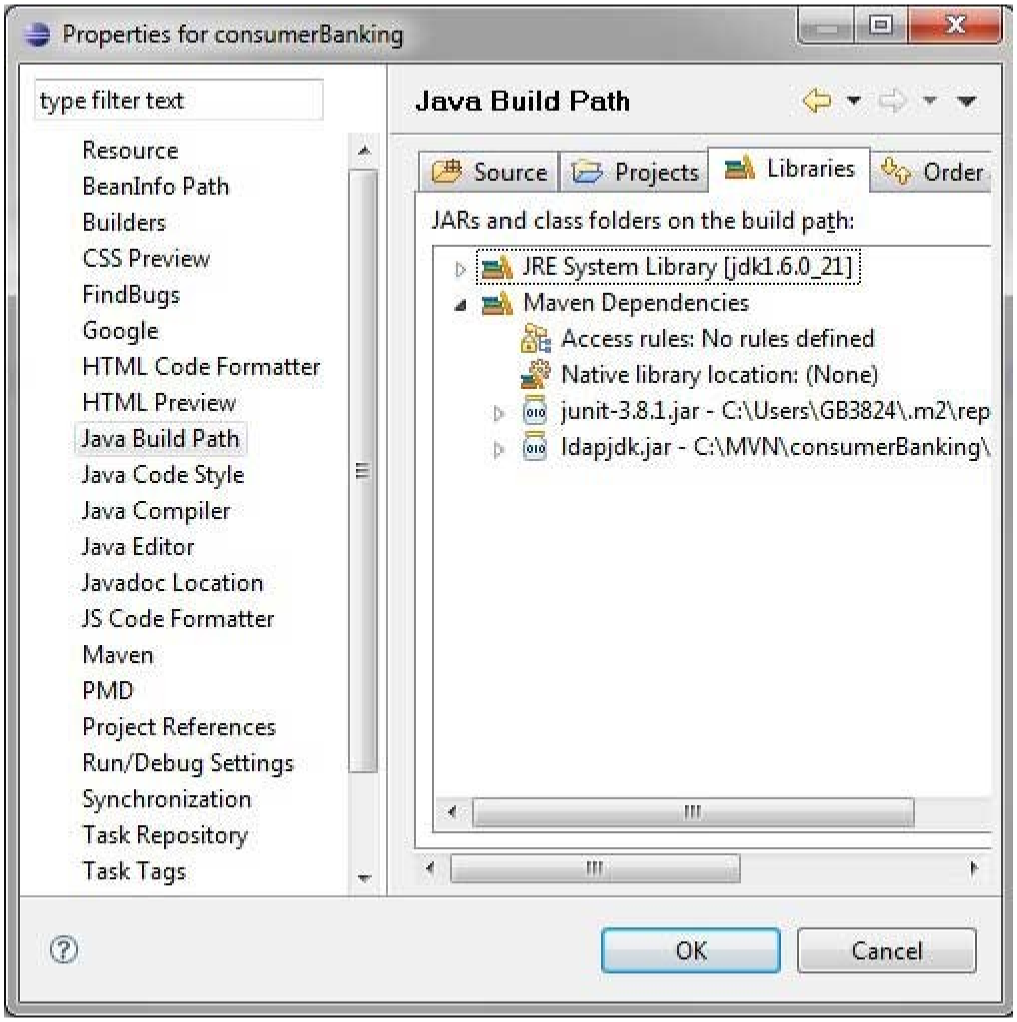
Working set:

Now, you can see the maven project in eclipse.


Java EE - consumerBanking/src/main/java/com/companyname/
File Edit Source Refactor Navigate Search Project H
Project Explorer
Navigator

```
consumerBanking
  .settings
  src
    lib
    main
      java
        com
          companyname
            bank
              App.java
              Util.java
    test
  target
  .classpath
  .project
  pom.xml
```

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



JARs and class folders on the build path:

- JRE System Library [jdk1.6.0_21]
- Maven Dependencies
  - Access rules: No rules defined
  - Native library location: (None)
  - junit-3.8.1.jar - C:\Users\GB3824\.m2\rep
  - ldapjdk.jar - C:\MVN\consumerBanking\

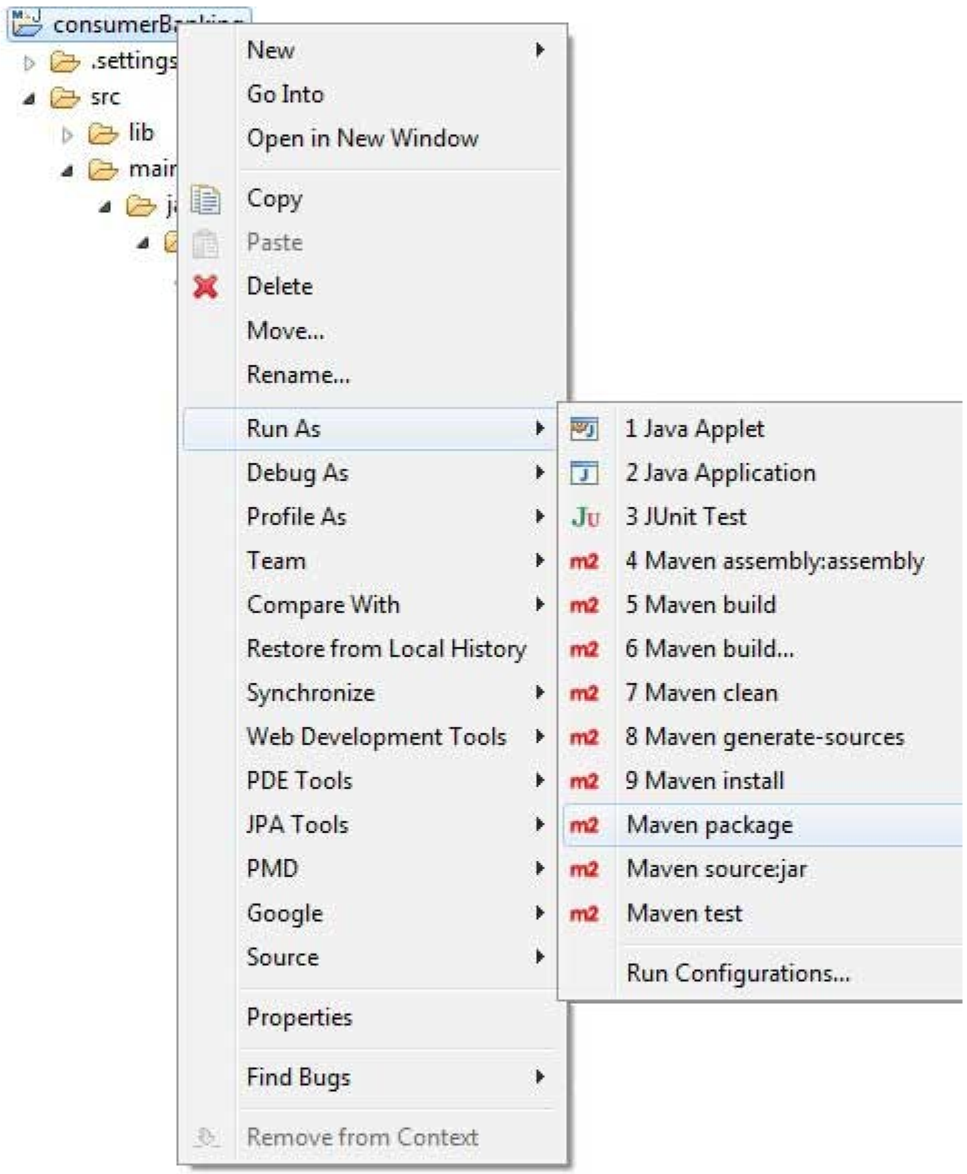
Now, Its time to build this project using maven capability of eclipse.

- 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

```
Hello World!
```

CHAPTER

19

Maven NetBeans Integration

NetBeans 6.7 and newer has inbuild support for Maven. In case of previous version, Maven plugin is available in plugin Manager. We're using NetBeans 6.9 in this example.

Some of features of NetBeans are listed below

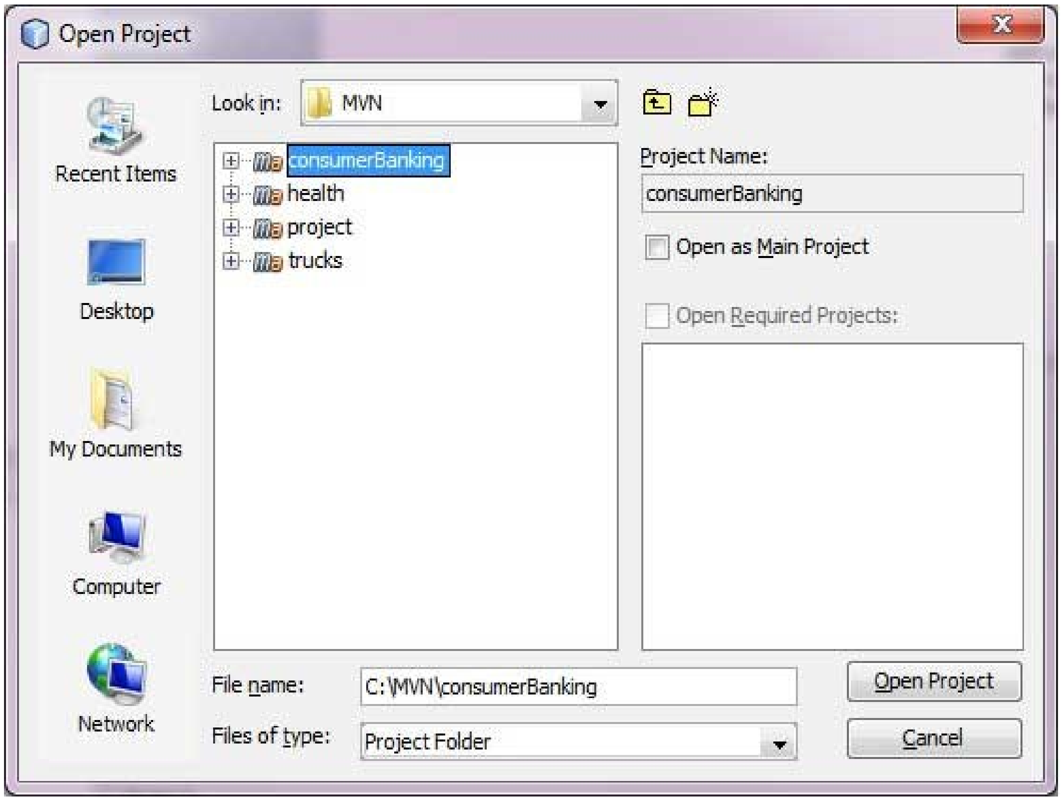
- You can run Maven goals from NetBeans.
- You can can view the output of Maven commands inside the NetBeans using its own console.
- You can update maven dependencies with IDE.
- You can Launch Maven builds from within NetBeans.
- NetBeans does the dependency management automatically based on Maven's pom.xml.
- NetBeans resolves Maven dependencies from its workspace without installing to local Maven repository (requires dependency project be in same workspace).
- NetBeans automatic downloads required dependencies and sources from the remote Maven repositories.
- NetBeans provides wizards for creating new Maven projects, pom.xml
- NetBeans provides a Maven Repository browser that enables you to view your local repository and registered external Maven repositories.

Following example will help you to leverage benefits of integrating NetBeans and Maven.

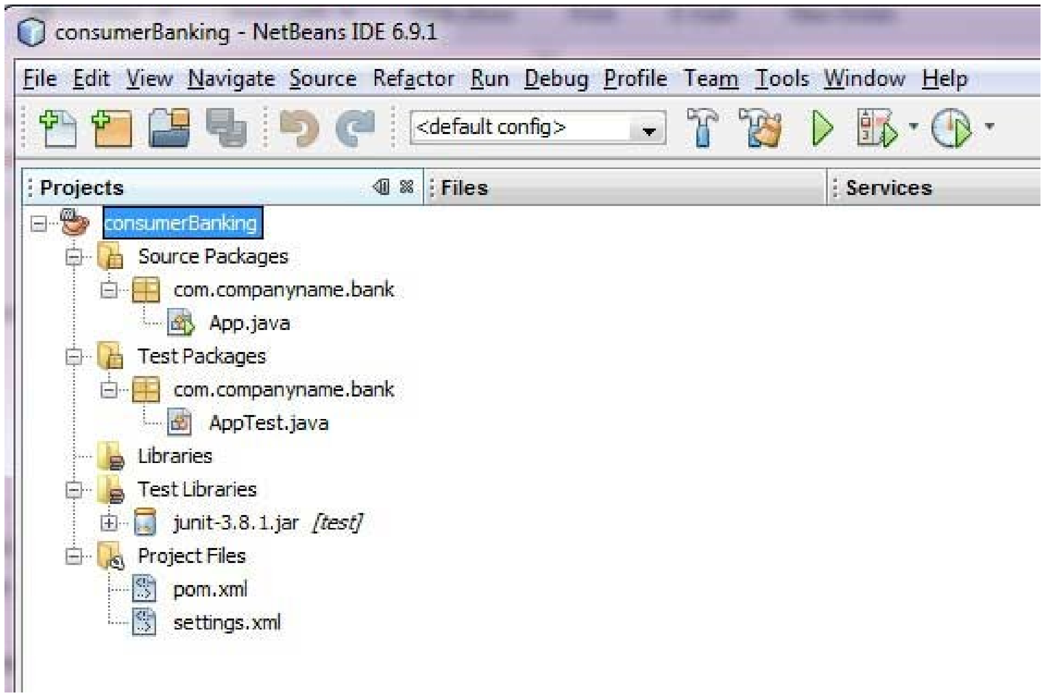
Open a maven project in NetBeans

- Open NetBeans.
- Select File Menu > Open Project option.

- 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.



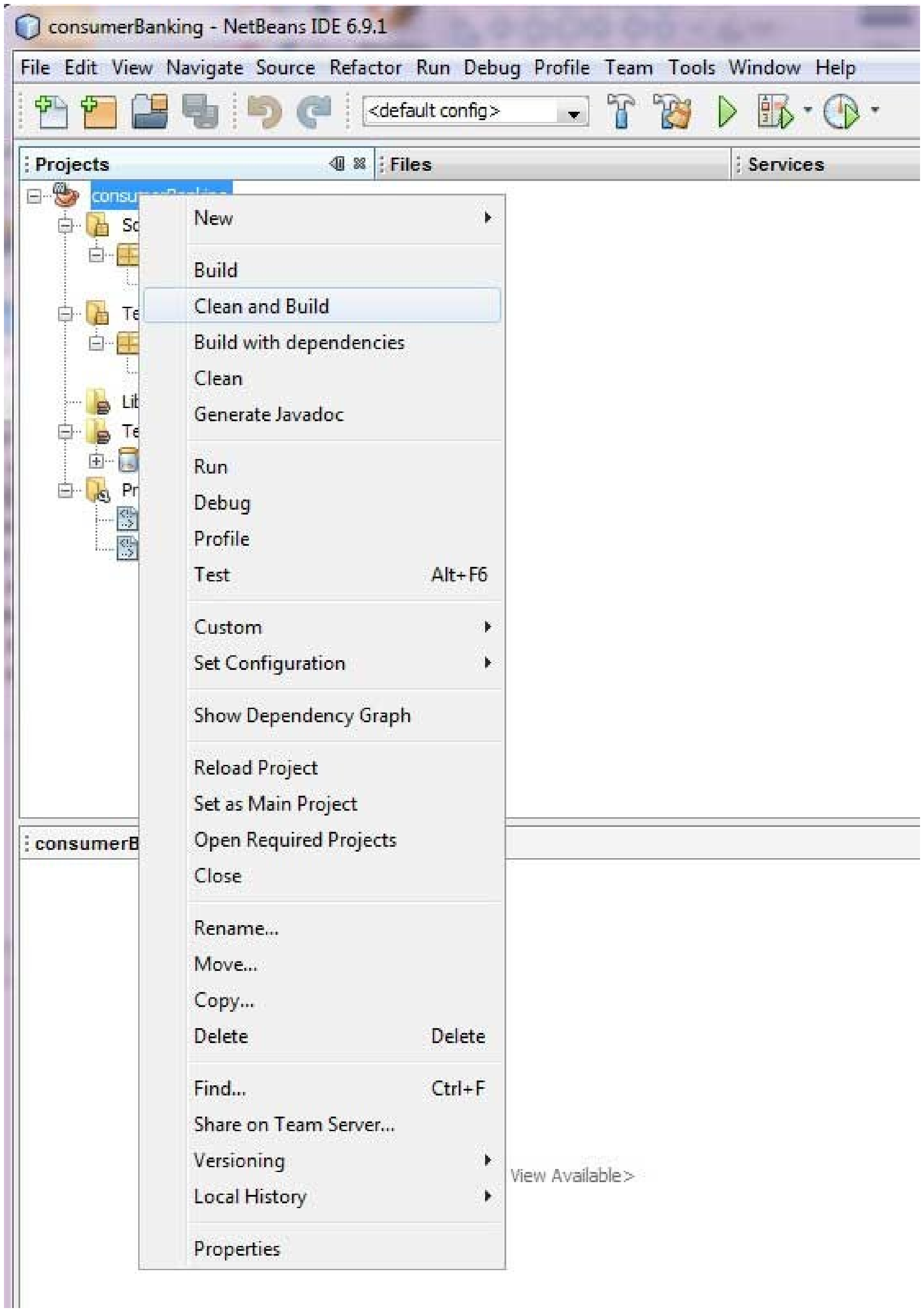
Now, you can see the maven project in NetBeans.Have a look at consumerBanking project Libraries and Test Libraries.You can see that NetBeans has added Maven dependencies to its build path.



Build a maven project in NetBeans

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

- Right Click on consumerBanking project to open context menu.
- Select Clean and Build as option



Maven will start building the project. You can see the output in NetBeans Console

```
NetBeans: Executing 'mvn.bat -Dnetbeans.execution=true clean install'
NetBeans:      JAVA_HOME=C:\Program Files\Java\jdk1.6.0_21
Scanning for projects...
------------------------------------------------------------------------
```

```
Building consumerBanking
   task-segment: [clean, install]
------------------------------------------------------------------------
[clean:clean]
[resources:resources]
[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources
[compiler:compile]
Compiling 2 source files to C:\MVN\consumerBanking\target\classes
[resources:testResources]
[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\MVN\consumerBanking\src\test\resources
[compiler:testCompile]
Compiling 1 source file to C:\MVN\consumerBanking\target\test-classes
[surefire:test]
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: 0.023 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[jar:jar]
Building jar: C:\MVN\consumerBanking\target\consumerBanking-1.0-SNAPSHOT.jar
[install:install]
Installing C:\MVN\consumerBanking\target\consumerBanking-1.0-SNAPSHOT.jar
to C:\Users\GB3824\.m2\repository\com\companyname\bank\consumerBanking\
1.0-SNAPSHOT\consumerBanking-1.0-SNAPSHOT.jar
------------------------------------------------------------------------
BUILD SUCCESSFUL
------------------------------------------------------------------------
Total time: 9 seconds
Finished at: Thu Jul 19 12:57:28 IST 2012
Final Memory: 16M/85M
```

```
---------------------------------------------------------------------------------
```

Run Application in NetBeans

Now, right click on App.java. Select **Run File** As option. You will see the result in NetBeans Console.

```
NetBeans: Executing 'mvn.bat -Dexec.classpathScope=runtime
-Dexec.args=-classpath %classpath com.companyname.bank.App
-Dexec.executable=C:\Program Files\Java\jdk1.6.0_21\bin\java.exe
-Dnetbeans.execution=true process-classes
org.codehaus.mojo:exec-maven-plugin:1.1.1:exec'
NetBeans:      JAVA_HOME=C:\Program Files\Java\jdk1.6.0_21
Scanning for projects...
------------------------------------------------------------------------
Building consumerBanking
   task-segment: [process-classes,
   org.codehaus.mojo:exec-maven-plugin:1.1.1:exec]
------------------------------------------------------------------------
[resources:resources]
[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources
[compiler:compile]
```

```
Nothing to compile - all classes are up to date
[exec:exec]
Hello World!
------------------------------------------------------------------------
BUILD SUCCESSFUL
------------------------------------------------------------------------
Total time: 1 second
Finished at: Thu Jul 19 14:18:13 IST 2012
Final Memory: 7M/64M
-------------------------------------------------
```

CHAPTER

20

Maven IntelliJ Integration

IntelliJ IDEA has inbuild support for Maven. We're using IntelliJ IDEA Community Edition 11.1 in this example.

Some of features of IntelliJ IDEA are listed below

- You can run Maven goals from IntelliJ IDEA.
- You can can view the output of Maven commands inside the IntelliJ IDEA using its own console.
- You can update maven dependencies within IDE.
- You can Launch Maven builds from within IntelliJ IDEA.
- IntelliJ IDEA does the dependency management automatically based on Maven's pom.xml.
- IntelliJ IDEA resolves Maven dependencies from its workspace without installing to local Maven repository (requires dependency project be in same workspace).
- IntelliJ IDEA automatic downloads required dependencies and sources from the remote Maven repositories.
- IntelliJ IDEA provides wizards for creating new Maven projects, pom.xml

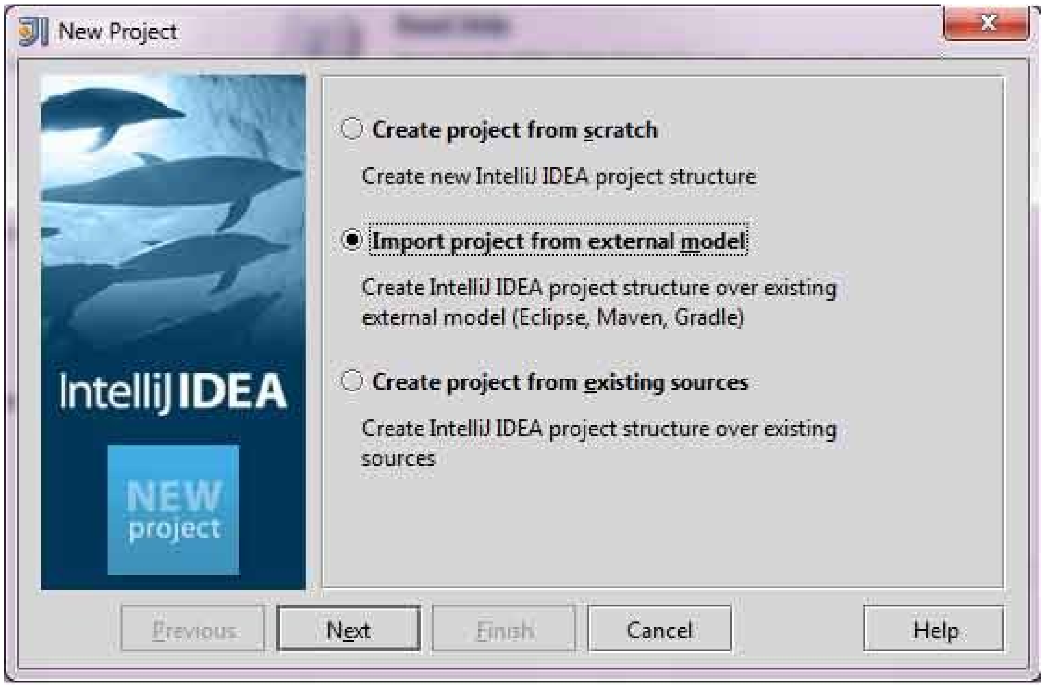
Following example will help you to leverage benefits of integrating IntelliJ IDEA and Maven.

Create a new project in IntelliJ IDEA

We'll import Maven project using New Project Wizard.

- Open IntelliJ IDEA.
- Select File Menu > New Project Option.

- Select import project from existing model.



New Project

○ Create project from scratch

Create new IntelliJ IDEA project structure

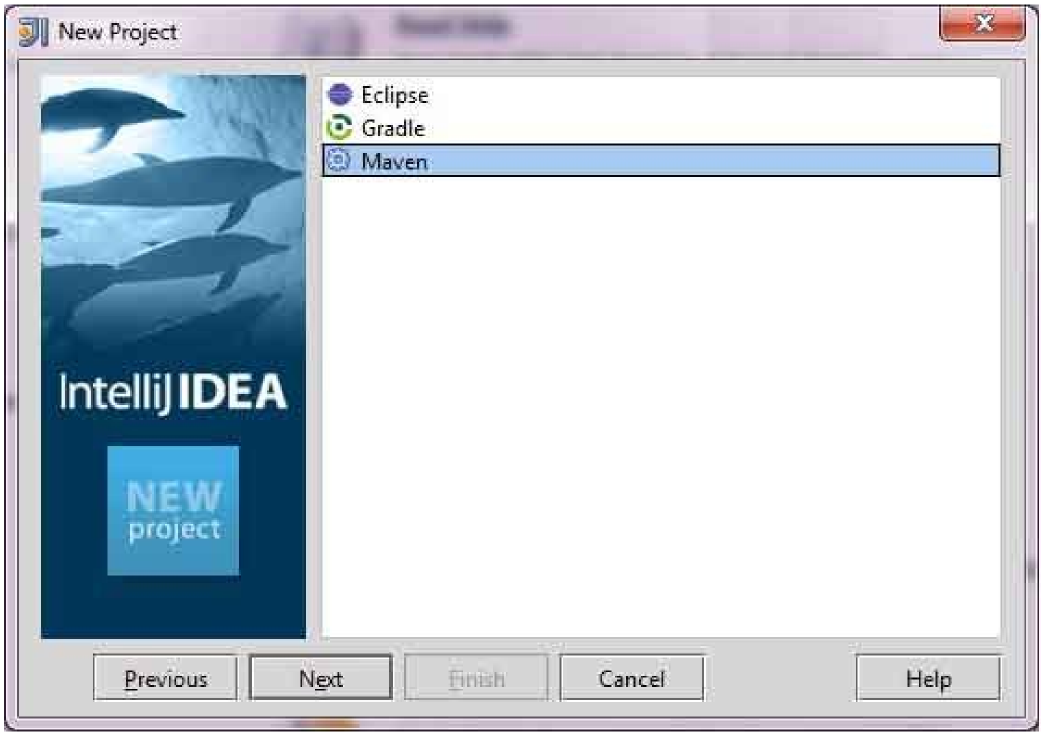
● Import project from external model

Create IntelliJ IDEA project structure over existing external model (Eclipse, Maven, Gradle)

○ Create project from existing sources

Create IntelliJ IDEA project structure over existing sources

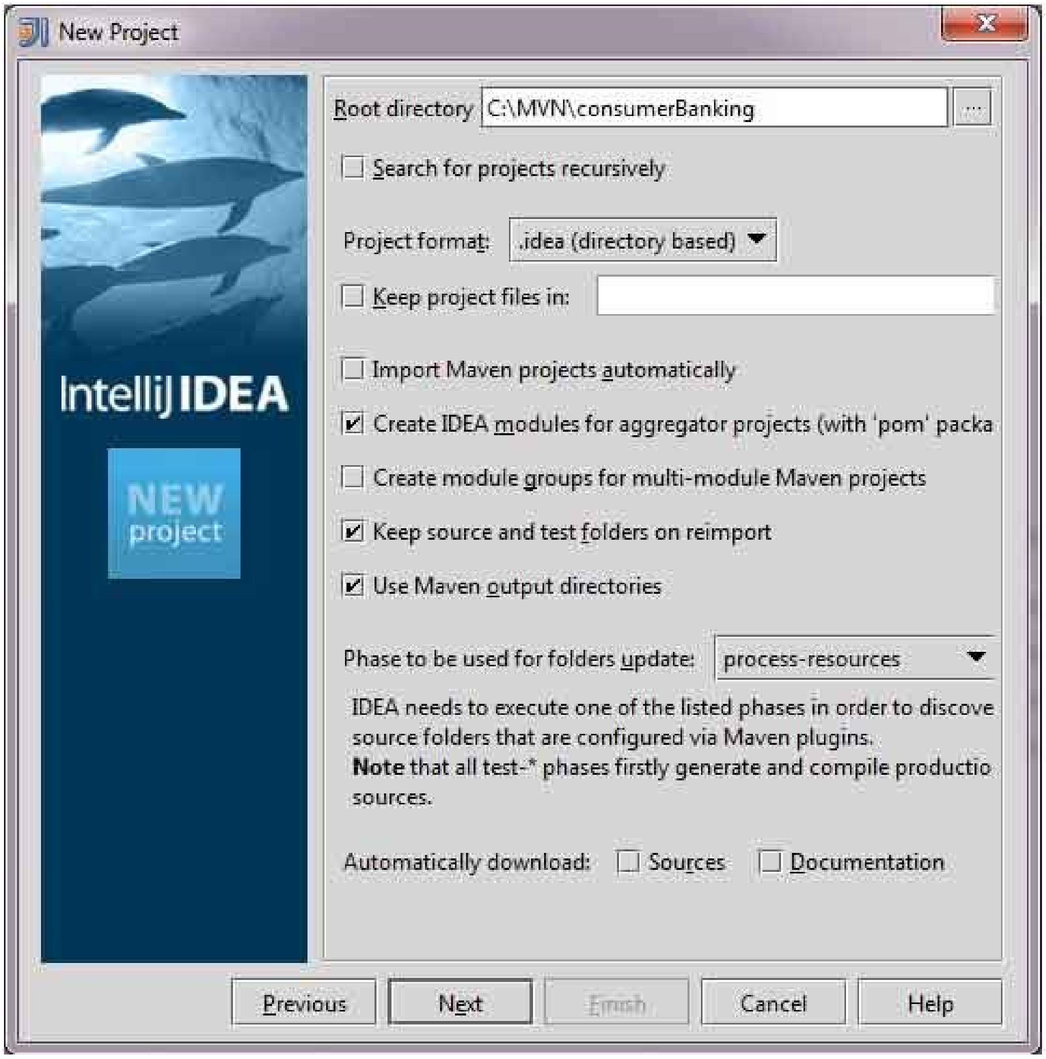
- Select Maven option



New Project

Eclipse
Gradle
Maven

- 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.



Root directory C:\MVN\consumerBanking

☐ Search for projects recursively

Project format: .idea (directory based)

☐ Keep project files in:

☐ Import Maven projects automatically

☑ Create IDEA modules for aggregator projects (with 'pom' packa

☐ Create module groups for multi-module Maven projects

☑ Keep source and test folders on reimport

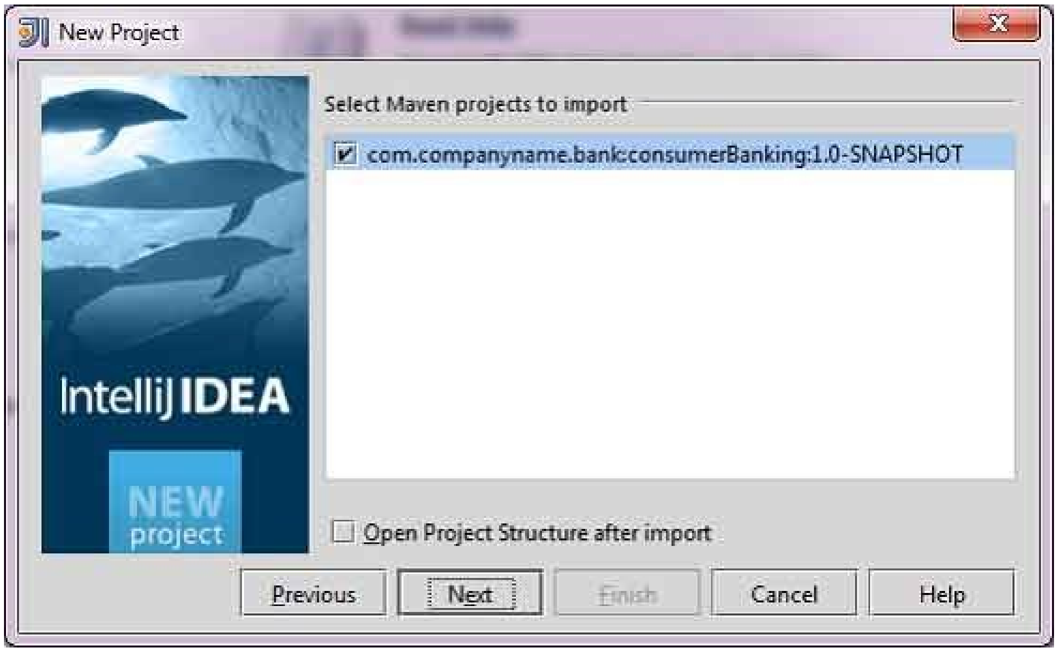
☑ Use Maven output directories

Phase to be used for folders update: process-resources

IDEA needs to execute one of the listed phases in order to discove source folders that are configured via Maven plugins.
Note that all test-* phases firstly generate and compile productio sources.

Automatically download: ☐ Sources ☐ Documentation

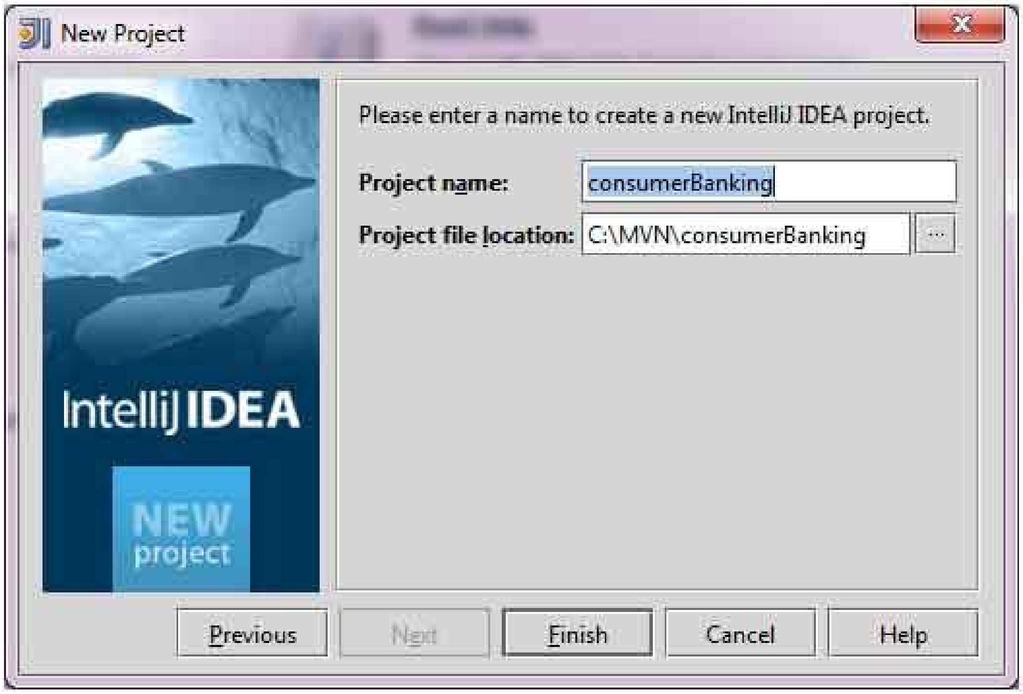
- Select Maven project to import.


Select Maven projects to import

☑ com.companyname.bank:consumerBanking:1.0-SNAPSHOT

☐ Open Project Structure after import

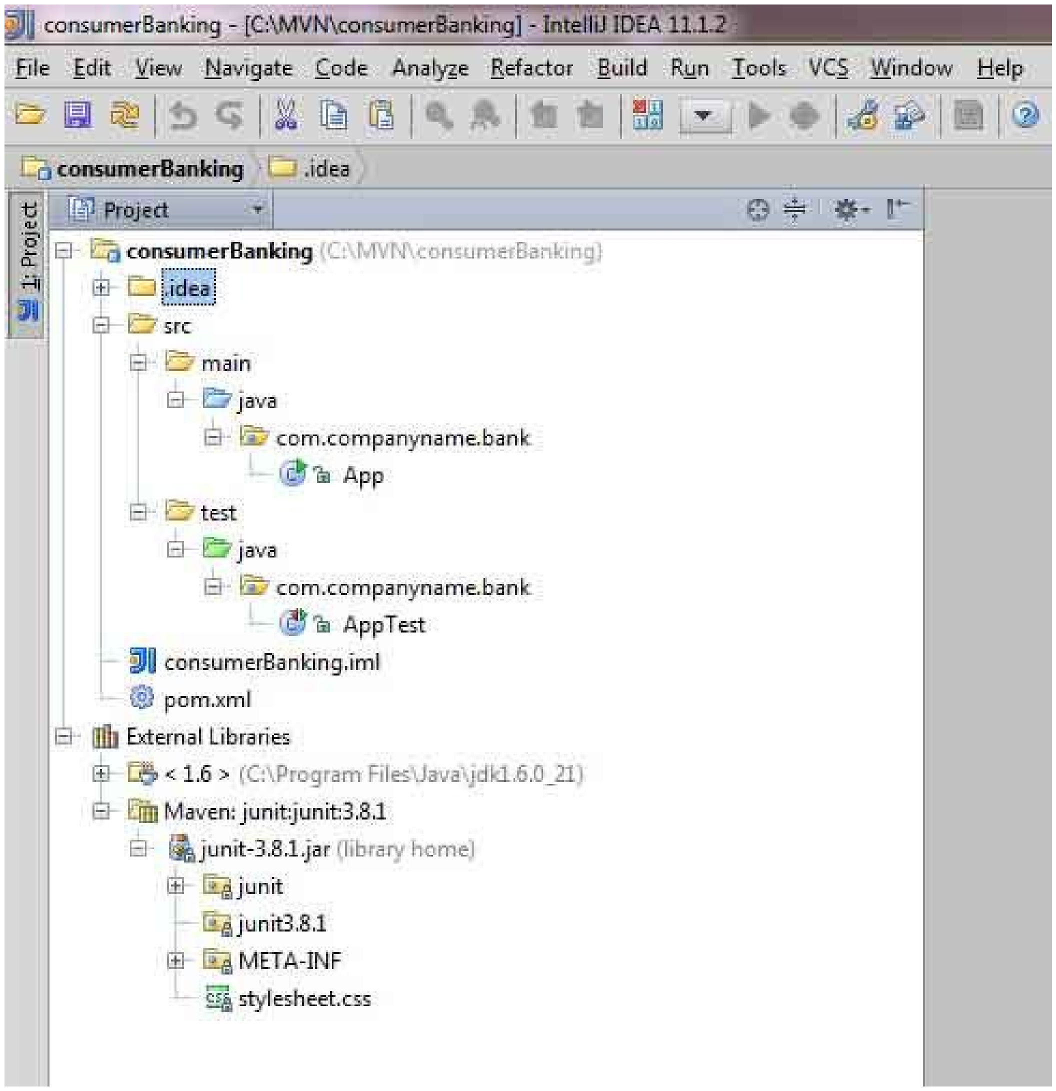
- Enter name of the project and click finish.


Please enter a name to create a new IntelliJ IDEA project.

Project name: consumerBanking

Project file location: C:\MVN\consumerBanking

Now, you can see the maven project in IntelliJ IDEA.Have a look at consumerBanking project external libraries.You can see that IntelliJ IDEA has added Maven dependencies to its build path under Maven section.


consumerBanking - [C:\MVN\consumerBanking] - IntelliJ IDEA 11.1.2
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
consumerBanking .idea
1: Project
Project
consumerBanking (C:\MVN\consumerBanking)
.idea
src
main
java
com.companyname.bank
App
test
java
com.companyname.bank
AppTest
consumerBanking.iml
pom.xml
External Libraries
< 1.6 > (C:\Program Files\Java\jdk1.6.0_21)
Maven: junit:junit:3.8.1
junit-3.8.1.jar (library home)
junit
junit3.8.1
META-INF
stylesheet.css

Build a maven project in IntelliJ IDEA

Now, Its time to build this project using capability of IntelliJ IDEA.

- Select consumerBanking project.
- Select Buid menu > Rebuild Project Option

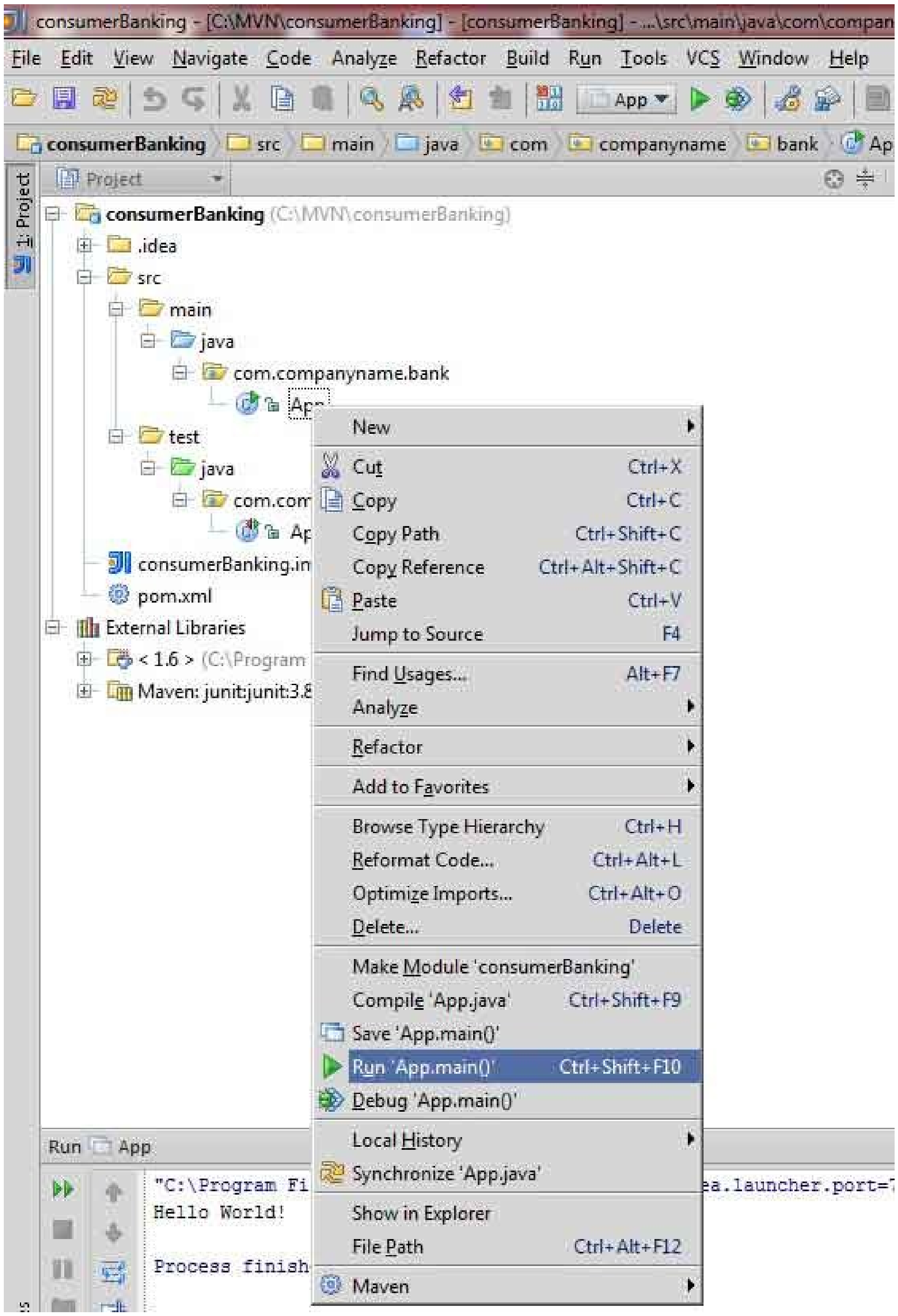
You can see the output in IntelliJ IDEA Console

```
4:01:56 PM Compilation completed successfully
```

Run Application in IntelliJ IDEA

- Select consumerBanking project.

- Right click on App.java to open context menu.
- select Run App.main()



consumerBanking - [C:\MVN\consumerBanking] - [consumerBanking] - ...\src\main\java\com\compan
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
App
consumerBanking src main java com companyname bank Ap
Project
consumerBanking (C:\MVN\consumerBanking)
.idea
src
main
java
com.companyname.bank
App
test
java
com.com
Ap
consumerBanking.in
pom.xml
External Libraries
< 1.6 > (C:\Program
Maven: junit:junit:3.8
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Jump to Source F4
Find Usages... Alt+F7
Analyze
Refactor
Add to Favorites
Browse Type Hierarchy Ctrl+H
Reformat Code... Ctrl+Alt+L
Optimize Imports... Ctrl+Alt+O
Delete... Delete
Make Module 'consumerBanking'
Compile 'App.java' Ctrl+Shift+F9
Save 'App.main()'
Run 'App.main()' Ctrl+Shift+F10
Debug 'App.main()'
Local History
Synchronize 'App.java'
Show in Explorer
File Path Ctrl+Alt+F12
Maven
Run App

```
"C:\Program Fi                                  ea.launcher.port=
Hello World!

Process finish
```

You will see the result in IntelliJ IDEA Console.

```
"C:\Program Files\Java\jdk1.6.0_21\bin\java"
-Didea.launcher.port=7533
"-Didea.launcher.bin.path=
C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 11.1.2\bin"
-Dfile.encoding=UTF-8
-classpath "C:\Program Files\Java\jdk1.6.0_21\jre\lib\charsets.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\deploy.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\javaws.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\jce.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\jsse.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\management-agent.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\plugin.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\resources.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\rt.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\dnsns.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\localedata.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunjce_provider.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunmscapi.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunpkcs11.jar
C:\MVN\consumerBanking\target\classes;
C:\Program Files\JetBrains\
IntelliJ IDEA Community Edition 11.1.2\lib\idea_rt.jar"
com.intellij.rt.execution.application.AppMain com.companyname.bank.App
Hello World!

Process finished with exit code 0
```