**Apache Maven**

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. Using maven we can build and manage any Java based project.

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

**What is Maven?**

Maven is a project management and comprehension tool that 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 the following −

Builds

Documentation

Reporting

Dependencies

SCMs

Releases

Distribution

Mailing list

**Maven installation**

1. Maven is a Java based tool, so the very first requirement is to have JDK installed on your machine.
2. Download Maven 2.2.1 from <https://maven.apache.org/download.cgi>
3. Extract the archive, to the directory you wish to install Maven 3.3.1. The subdirectory apache-maven-3.3.1 will be created from the archive.
4. Add MAVEN\_HOME,M2\_HOME, M2, MAVEN\_OPTS to environment variables.
5. Set the environment variables using system properties.

MAVEN\_HOME=C:\Program Files\Apache Software Foundation\apache-maven-3.3.1

M2\_HOME=C:\Program Files\Apache Software Foundation\apache-maven-3.3.1

M2=%M2\_HOME%\bin

1. MAVEN\_OPTS=-Xms256m -Xmx512m
2. Now append M2 variable to System Path.

Append the string ;%M2% to the end of the system variable, Path.

**POM** stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml.The POM contains information about the project and various configuration detail 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, and 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

## **Elements of maven pom.xml file**

For creating the simple pom.xml file, you need to have following elements:

|  |  |
| --- | --- |
| Element | Description |
| **project** | It is the root element of pom.xml file. |
| **modelVersion** | It is the sub element of project. It specifies the modelVersion. It should be set to 4.0.0. |
| **groupId** | It is the sub element of project. It specifies the id for the project group. |
| **artifactId** | It is the sub element of project. It specifies the id for the artifact (project). An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, and WARs. |
| **version** | It is the sub element of project. It specifies the version of the artifact under given group. |

*File: pom.xml*

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
6. **<modelVersion>**4.0.0**</modelVersion>**
7. **<groupId>**com.javatpoint.application1**</groupId>**
8. **<artifactId>**my-app**</artifactId>**
9. **<version>**1**</version>**
11. **</project>**

## **Dependencies**

One of the core features of Maven is Dependency Management. Managing dependencies is a difficult task once we've to deal with multi-module projects (consisting 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 wants to use A, then that project requires to use library B too.

Maven helps to avoid such requirements 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.

|  |  |
| --- | --- |
| Sr.No. | Feature & Description |
| 1 | **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. |
| 2 | **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 dependency Management section and directly control which version of B is to be used when it is ever referenced. |
| 3 | **Dependency scope**  Includes dependencies as per the current stage of the build. |
| 4 | **Excluded dependencies**  Any transitive dependency can be excluded using "exclusion" element. As example, A depends upon B and B depends upon C, then A can mark C as excluded. |
| 5 | **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. |

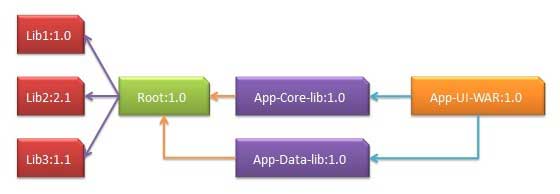
## Dependency Scope

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

|  |  |
| --- | --- |
| Sr.No. | Scope & Description |
| 1 | **compile**  This scope indicates that dependency is available in classpath of project. It is default scope. |
| 2 | **provided**  This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime. |
| 3 | **runtime**  This scope indicates that dependency is not required for compilation, but is required during execution. |
| 4 | **test**  This scope indicates that the dependency is only available for the test compilation and execution phases. |
| 5 | **system**  This scope indicates that you have to provide the system path. |
| 6 | **import**  This scope is only used when dependency is of type pom. This scope indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section. |

## Dependency Management

Usually, we have 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, the parent of sub-project's poms. Following example will help you understand this concept.



Following are the detail of the 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 the 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 detail.

**Update Project POM.xml:-**

We will 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 have used −

Sr.No. Element & Description

1.SCM

Configures the SVN location from where Maven will check out the source code.

2.Repositories

Location where built WAR/EAR/JAR or any other artifact will be stored after code build is successful.

3.Plugin

maven-release-plugin is configured to automate the deployment process.

Maven Release Plug-in

The Maven does the 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, such as −

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 the command console, go to the C:\ > MVN >bus-core-api directory and execute the following mvn command.

>mvn release:prepare

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

>mvn release:perform

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

What is a Plugin?

"Maven" is really just a core framework for a collection of Maven Plugins. In other words, plugins are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on. Almost any action that you can think of performing on a project is implemented as a Maven plugin.

## What are 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, which can be executed using the 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 the following command.

mvn compiler:compile

## Plugin Types

Maven provided the following two types of Plugins −

|  |  |
| --- | --- |
| Sr.No. | Type & Description |
| 1 | **Build plugins**  They execute during the build process and should be configured in the <build/> element of pom.xml. |
| 2 | **Reporting plugins**  They execute during the site generation process and they should be configured in the <reporting/> element of the pom.xml. |

Following is the list of few common plugins −

|  |  |
| --- | --- |
| Sr.No. | Plugin & Description |
| 1 | **clean**  Cleans up target after the build. Deletes the target directory. |
| 2 | **compiler**  Compiles Java source files. |
| 3 | **surefire**  Runs the JUnit unit tests. Creates test reports. |
| 4 | **jar**  Builds a JAR file from the current project. |
| 5 | **war**  Builds a WAR file from the current project. |
| 6 | **javadoc**  Generates Javadoc for the project. |
| 7 | **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. Refer Build Profiles chapter. Let us understand it in a better way and 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 the 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 displaying the 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*.
* Maven will then download the plugin if not available in local repository and start its processing.

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 the command console, go to 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] ------------------------------------------------------------------

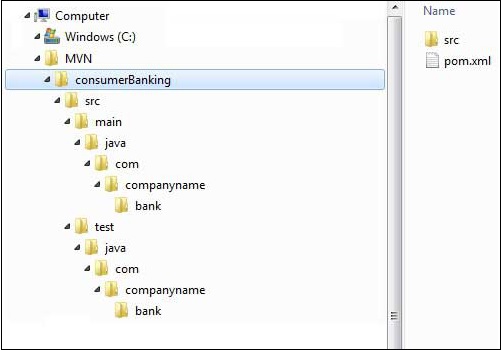
[INFO] Total time: 14 seconds

[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 consumer Banking (as specified in artifactId). Maven uses a standard directory layout as shown below −



Using the above example, we can understand the following key concepts −

|  |  |
| --- | --- |
| Sr.No. | Folder Structure & Description |
| 1 | **consumerBanking**  contains src folder and pom.xml |
| 2 | **src/main/java**  contains java code files under the package structure (com/companyName/bank). |
| 3 | **src/main/test**  contains test java code files under the package structure (com/companyName/bank). |
| 4 | **src/main/resources**  it contains images/properties files (In above example, we need to create this structure manually). |

If you observe, you will find that 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 to 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 all the build related complexities.

In the next chapter, we'll discuss how to build and test the project using maven Build and Test Project.

Run Selective Reports

To run the reports selectively, you can configure it to include only the reports that you prefer. Use mvn site to generate the selected reports.

<project>

...

<reporting>

<plugins>

<plugin>

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

<artifactId>maven-project-info-reports-plugin</artifactId>

<version>3.0.0</version>

<reportSets>

<reportSet>

<reports>

<report>dependencies</report>

<report>team</report>

<report>mailing-lists</report>

<report>ci-management</report>

<report>issue-management</report>

<report>licenses</report>

<report>scm</report>

</reports>

</reportSet>

</reportSets>

</plugin>

...

</plugins>

</reporting>

...

</project>

**create documentation of the application**

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 using the examples given in the previous chapters. 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]-----------------------------------------------

Your project documentation is now ready. Maven has created a site within the target directory.

documentation site pages

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

consumer web page

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

XDoc A Maven 1.x documentation format

https://jakarta.apache.org/site

FML Used for FAQ documents

https://maven.apache.org