

APACHE MAVEN COOKBOOK

Hot Recipes for Maven Development

MavenTM



Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Apache Maven Cookbook

Contents

1	Maven Tutorial for Beginners	1
1.1	Introduction	1
1.2	Prerequisites	1
1.3	Installing maven	1
1.4	Creating an example project	2
1.5	Importing generated project into eclipse	2
1.6	Maven config files. POM file	5
1.7	Building the project	6
1.8	Conclusions	9
2	How to Use Maven For Dependency Management	10
2.1	Environment	10
2.2	Maven for Dependency Management	10
2.3	Transitive Dependency	10
2.4	Dependency Scope	11
2.5	Dependency Management	11
2.6	System Dependencies	14
2.7	Conclusion	14
2.8	Related Articles	14
3	Difference Between ANT and Maven	15
3.1	Introduction	15
3.2	High level comparision	15
3.2.1	High level comparision	16
3.2.2	Execution way	16
3.2.3	Lifecycle management	16
3.2.4	Dependency management	16
3.3	Ant project	16
3.4	Maven project	19
3.5	Projects difference	21
3.6	Conclusions	21
3.7	Download	21

4	Maven Project Structure Example	22
4.1	Introduction	22
4.2	Directory layout. Files	23
4.2.1	src/main/java	23
4.2.2	src/test/java	24
4.2.3	src/it	24
4.3	Directory layout. Resources	24
4.3.1	src/main/resources	24
4.3.2	src/test/resources	25
4.3.3	src/main/filters	25
4.3.4	src/test/filters	25
4.4	Directory layout. Misc	25
4.4.1	src/assembly	25
4.5	Directory layout. Webapp	25
4.6	Directory layout. Target	26
4.7	Directory layout. Pom file	26
4.8	Directory layout. Personal files	26
4.9	Conclusions	26
5	Maven Settings.xml example	27
5.1	Introduction	27
5.2	Single values	29
5.2.1	localRepository	29
5.2.2	interactiveMode	30
5.2.3	usePluginRegistry	30
5.2.4	offline	31
5.3	PluginGroups	31
5.4	Servers	32
5.5	Mirrors	32
5.6	Proxies	33
5.7	Profiles	34
5.8	Active profiles	35
5.9	Conclusions	35
5.10	Download the source code	37
6	Maven Local Repository example	38
6.1	Introduction	38
6.2	Local repository structure	38
6.3	Deploying artifacts to the local repository	39
6.4	Installing artifacts/dependencies in the local repository	39
6.5	Maven locate artifacts strategy	39
6.6	Conclusions	40

7	Maven Dependency Plugin Example	41
7.1	Introduction	41
7.2	Example project	42
7.3	See dependencies tree	44
7.4	Build classpath	46
7.5	Other features	47
7.6	Conclusions	47
7.7	Download the eclipse project	48
8	Maven Shade Plugin Example	49
8.1	Introduction	49
8.2	Example project	49
8.3	Include/Exclude dependencies	51
8.4	Shading packages	53
8.5	Attaching the shaded artifact	54
8.6	Conclusions	56
8.7	Download	56
9	Maven War Plugin Example	57
9.1	Introduction	57
9.2	Example project	58
9.3	Generate a exploded war	59
9.4	Filtering the war file content	59
9.5	Customizing manifest file	60
9.6	Conclusions	61
9.7	Download the eclipse project	61
10	Maven Compiler Plugin Example	62
10.1	Introduction	62
10.2	Example project	63
10.3	Plugin options	64
10.3.1	Set a different JDK to compile classes	64
10.3.2	Specify a compatible JDK	65
10.3.3	Set some arguments to the compiler	65
10.3.4	Set some specific arguments for the compiler that you have selected	66
10.4	Conclusions	67
10.5	Download the eclipse project	67

11 Maven jar plugin example	68
11.1 Introduction	68
11.2 Example project	69
11.3 Use a default <code>manifest file</code>	70
11.4 Custom <code>manifest file</code>	73
11.5 Include/Exclude files	75
11.6 Additional jars	77
11.7 Generate a jar artifact with test classes	79
11.8 Conclusions	81
11.9 Download the eclipse project	81
12 Maven assembly plugin example	82
12.1 Introduction	82
12.2 Example project	82
12.3 Assembly plugin predefined descriptors	84
12.4 Assembly plugin custom descriptors	85
12.5 Running the assembly plugin	86
12.6 See the result	88
12.7 Download the eclipse project	90

Copyright (c) Exelixis Media P.C., 2016

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

Preface

Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down.

An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache. This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated. (https://en.wikipedia.org/wiki/Apache_Maven)

In this ebook, we provide a compilation of Maven examples that will help you kick-start your own projects. We cover a wide range of topics, from project structure and configuration, to dependency management and plug-ins. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at <https://www.javacodegeeks.com/>

Chapter 1

Maven Tutorial for Beginners

In this tutorial we are going to see how we can install and use maven.

Maven is a build automation tool used mainly for java projects from apache.

We are going to see step by step how you can download and install maven, what are the prerequisites needed in order to do it and a project example.

For this example we use the following technologies:

- Windows 7
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits

1.1 Introduction

Maven is a java tool, so in order to introduce maven we are going to create a simple java project, and build it with maven.

The first thing we are going to do is install maven in your computer, to do that, follow the steps in the next bullet.

Let's begin!

1.2 Prerequisites

You must have installed Java in your computer in order to proceed because maven is a java tool. You can download a java JDK [here](#).

Once you have java installed on your system, you must install maven. You can download maven from [here](#).

1.3 Installing maven

In order to install maven you have to follow a few steps:

Ensure `JAVA_HOME` environment variable exists and is set to your JDK installation.

Extract the maven package to any directory you want. If you are under unix system, you can use the following commands:

unzip:

```
unzip apache-maven-3.3.9-bin.zip
```

or tar:

```
tar xzvf apache-maven-3.3.9-bin.tar.gz
```

If you are under windows or if you have any extract tool, use it to extract maven.

Add the `bin` directory to the system `PATH` environment variable. In unix system you can do the following in a shell session:

export:

```
export PATH=/opt/apache-maven-3.3.9/bin:$PATH
```

In a windows system, you can access through *start + right click on computer*, then choose properties. Click on *system advanced configuration*, click on *environment variables* button. Edit `path` environment variable and add the maven `bin` directory at the end.

Now you are ready, open a shell window and type `mvn -v`. You should see the maven version and many other things.

You can follow the full instructions in order to install maven [here](#).

1.4 Creating an example project

Maven has several archetypes predefined, you can use any of those archetypes and build a basic project in a few seconds. We are going to use a archetype called `maven-archetype-quickstart`, this archetype will create for us a basic java project with maven directories conventions that will be packaged as a jar file.

To create the project execute the following command in a directory that you will use as workspace

archetype generation:

```
mvn archetype:generate -DgroupId=com.javacodegeeks.example -DartifactId=jcg-example - ←  
  DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

If you are running maven for the first time, it will take a few seconds to accomplish the generate command because maven has to download all the required plugins and artifacts in order to make the generation task.

Notice now you have a new directory with the same name as the `artifactId` inside the choosen directory. Below you will see the new directory structure

1.5 Importing generated project into eclipse

You can build your project in shell environment, but is better work inside of a Java IDE such as eclipse.

To import the generated project into eclipse, open eclipse Mars and choose the workspace directory in which you generated the project in the last bullet.

Once eclipse is ready, import the project. Choose `File -> New -> Java project`. Type `jcg-example` as project name as you can see in the image below:

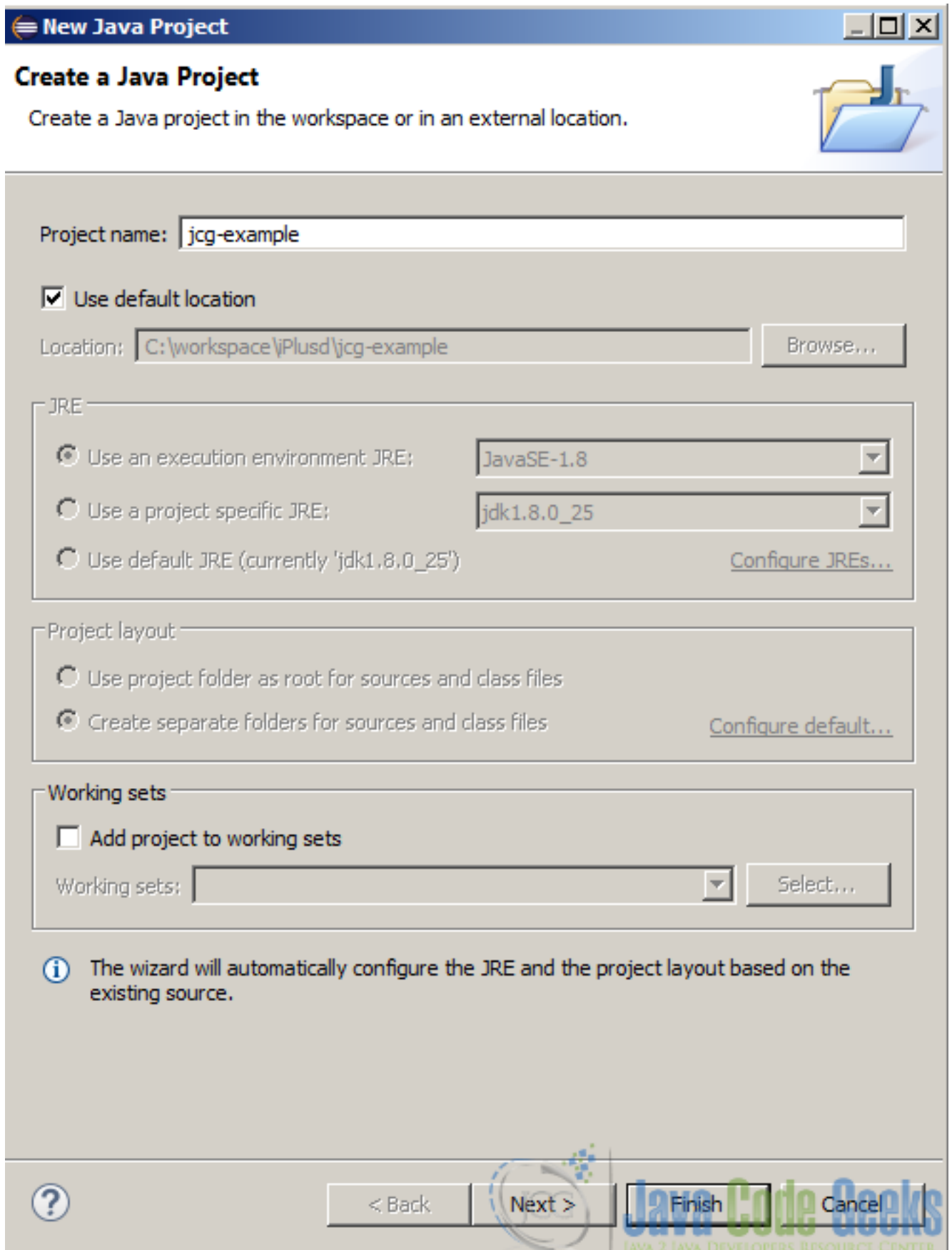


Figure 1.1: Importing project into eclipse, step 1

The project is imported but is not under maven nature. Eclipse Mars come out of the box with maven support, so activate it with right click on project and choose `Configure -> Convert to Maven project` as you can see in the image below:

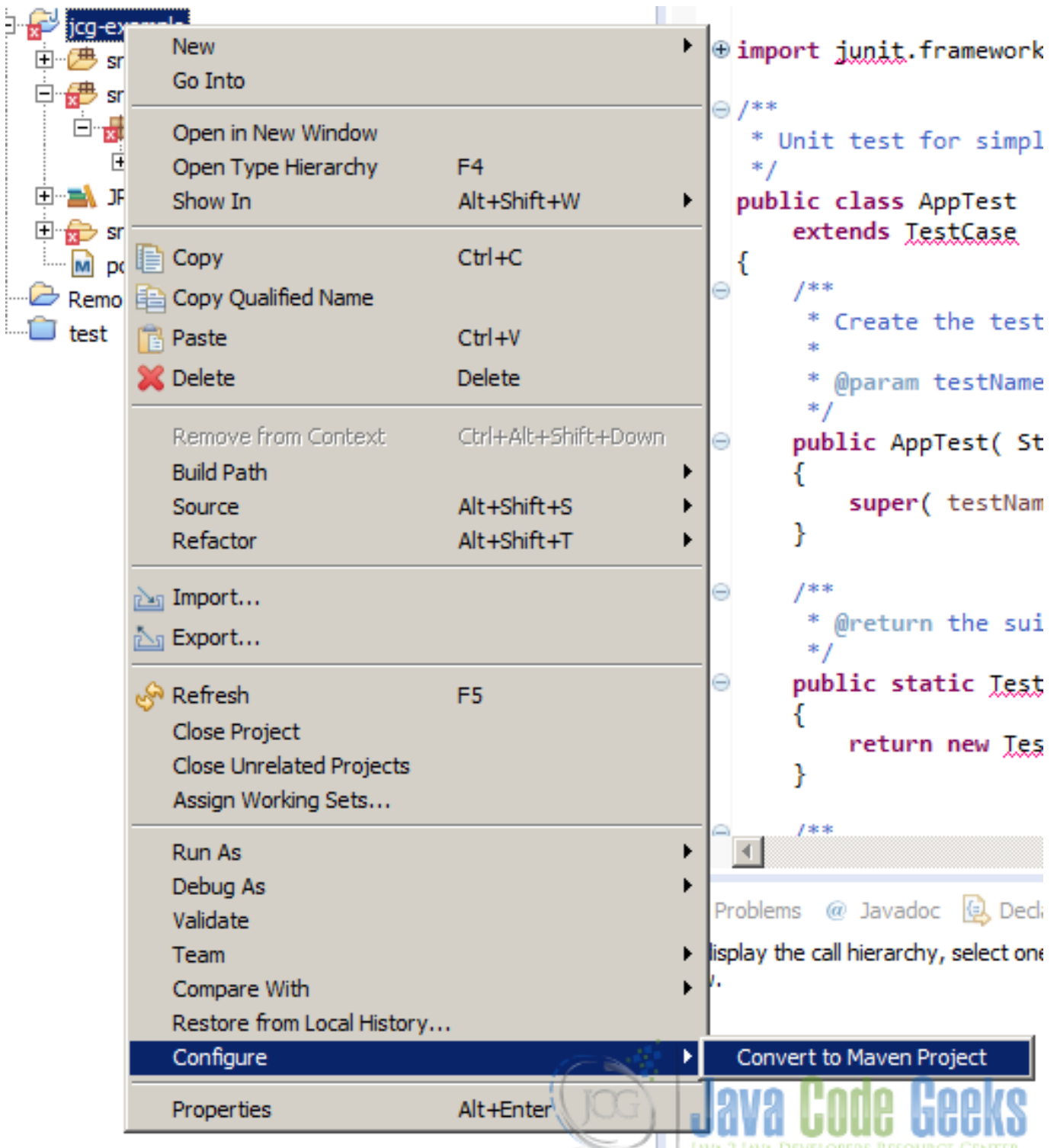


Figure 1.2: Importing project into eclipse, step 2

Now the project is recognized by eclipse as a maven project and all the dependencies will be resolved and added to the classpath automatically.

The project structure is the following one:



Figure 1.3: Example project

The dependencies and maven configuration is defined inside the maven configuration file pom.xml, pom means Project Object model. You can see more details about it in the next bullet.

1.6 Maven config files. POM file

Maven defines a file called pom.xml (Project Object Model) in which you can configure all project features, dependencies, repositories, plugins, etc. . . to use in order to build your project.

In our example project we have a very simple pom file:

pom.xml file:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.example</groupId>
  <artifactId>jcg-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>jcg-example</name>
  <url>https://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The file defines an artifact with `com.javacodegeeks.example` as `groupId`, `jcq_example` as `artifactId` and `1.0-SNAPSHOT` as `version`.

The pom file tells maven that the project will be packaged as a jar file in the `packaging` tag.

The pom file defines only one dependency in test scope, that is, `junit`. You can add all the dependencies you need in the pom file in order to use some other libraries in the desired scope.

You can see more details about the pom.xml file [here](#).

1.7 Building the project

To build the project you only have to do a right-click on the project in eclipse and choose `Run as -Maven build` as you can see in the image below

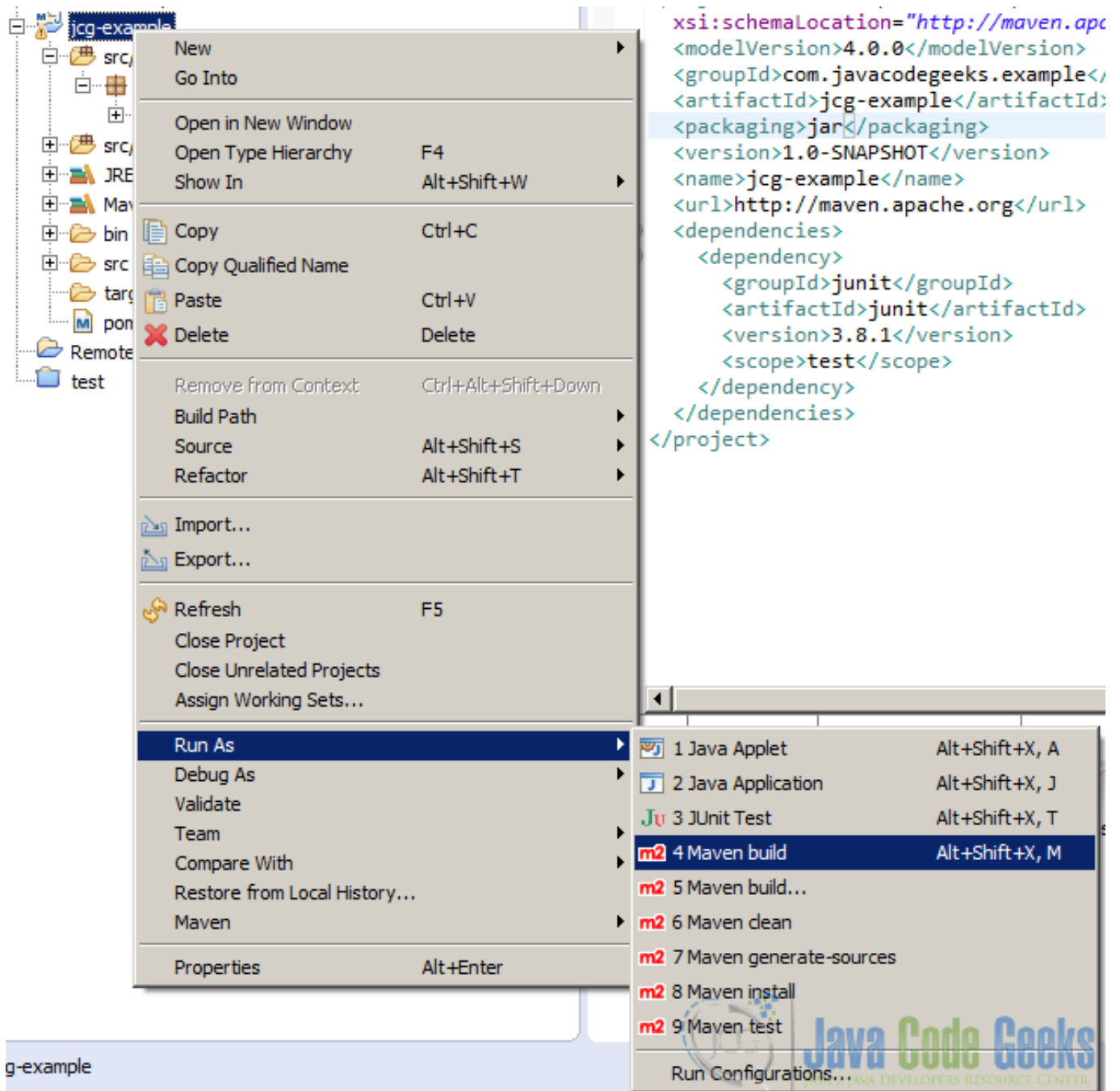


Figure 1.4: Building project, step 1

Type `clean install` as maven goals, then click on run.

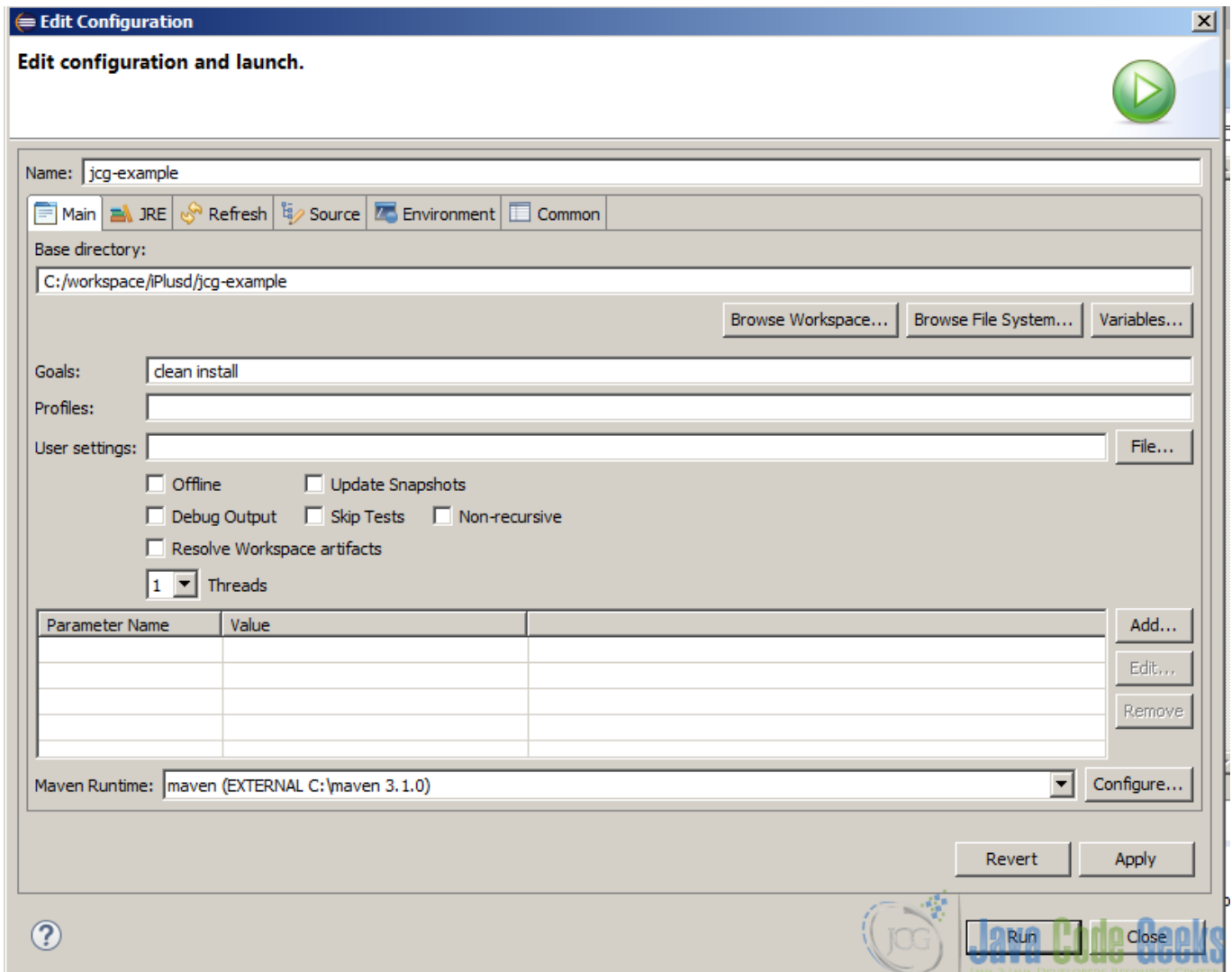


Figure 1.5: Building project, step 2

You can see a console output like this:

output:

```
[INFO] Scanning for projects...
[INFO]
[INFO] Building jcg-example 1.0-SNAPSHOT
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ jcg-example ---
[INFO] Deleting C:\workspace\iPlus\jcg-example\target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ jcg-example ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build ←
is platform dependent!
[INFO] skip non existing resourceDirectory C:\workspace\iPlus\jcg-example\src\main\ ←
resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ jcg-example ---
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is ←
platform dependent!
[INFO] Compiling 1 source file to C:\workspace\iPlus\jcg-example\target\classes
```

```
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ jcg-example ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\workspace\iPlusd\jcg-example\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ jcg-example ---
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 1 source file to C:\workspace\iPlusd\jcg-example\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ jcg-example ---
[INFO] Surefire report directory: C:\workspace\iPlusd\jcg-example\target\surefire-reports

T E S T S

Running com.javacodegeeks.example.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.051 sec

Results :

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

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ jcg-example ---
[INFO] Building jar: C:\workspace\iPlusd\jcg-example\target\jcg-example-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jcg-example ---
[INFO] Installing C:\workspace\iPlusd\jcg-example\target\jcg-example-1.0-SNAPSHOT.jar to C:\Users\fhernans\.m2\repository\com\javacodegeeks\example\jcg-example\1.0-SNAPSHOT\jcg-example-1.0-SNAPSHOT.jar
[INFO] Installing C:\workspace\iPlusd\jcg-example\pom.xml to C:\Users\fhernans\.m2\repository\com\javacodegeeks\example\jcg-example\1.0-SNAPSHOT\jcg-example-1.0-SNAPSHOT.pom
[INFO] BUILD SUCCESS
[INFO] Total time: 3.718s
[INFO] Finished at: Mon Apr 04 20:35:58 CEST 2016
[INFO] Final Memory: 13M/210M
```

You can run it from command line, you have to navigate to the project parent directory and type `mvn clean install`. You will see an output like the previous one.

1.8 Conclusions

In this tutorial you have seen the basic stuff to create, manage and build a simple jar project with maven. As you have seen, you can quickly create and build a java jar project with maven, delegating some things (like dependency management, life-cycle management, test execution, etc...) to maven so you can focus your efforts in create a very good code in your project.

Chapter 2

How to Use Maven For Dependency Management

In this example, we will see how to use Maven for dependency management. Maven is a build manager tool and mostly used in java projects. Maven was built on a central concept of project object model (POM). Maven addresses two steps for any project - first how a project is built and second how the dependencies are described.

2.1 Environment

- Windows 7 SP 1
- Eclipse Kepler 4.3
- Maven 3.0.4
- Java version 7

2.2 Maven for Dependency Management

Maven is used in large projects for one of its major feature and that is dependency management. Maven just not supports dependencies for a single project, but it easily manages multi-module projects. In such projects, maven helps in maintaining a high degree of control and stability.

2.3 Transitive Dependency

This feature of Transitive Dependency allows to avoid needing to discover and specify libraries that your own dependencies require, and what that means is that if your project is dependent on some libraries and those libraries are dependent on other libraries, you only specify your project's dependencies and this feature takes care of other dependencies. There is no limit to levels that dependencies can be gathered from, and will only cause a problem if cyclic dependency is discovered. If your project depends on A and A depends on C and C depends on B and B depends on A.

Following features help to limit the graph of included libraries using transitive dependencies' feature:

- **Dependency Mediation-** This feature determines what version of a dependency will be used when multiple versions of an artifact are encountered. You can always guarantee a version by declaring it explicitly in project's POM. If two dependency versions are at the same depth in the dependency tree, the order in the declaration that counts, the first declaration wins.
 - **Dependency Management-** With this feature, you can specify versions of artifacts to be used when they are encountered in transitive dependencies.
 - **Dependency Score-** This allows you to only include dependencies appropriate for the current stage of the build.
-

- **Excluded Dependencies**- If project X depends on project Y, and project Y depends on project Z, the project X can explicitly exclude project Z as a dependency using the `exclusion` element.
- **Optional Dependencies**- If project Y depends on project Z, and the owner of Project Y can mark project Z as optional dependency, using the `optional` element. When project X depends on project Y, X will depend only on Y and not on Y's optional dependency Z.

2.4 Dependency Scope

Dependency scope is used to restrict the transitivity of a dependency, and also to affect the `classpath` used for various build. There are 6 scopes available:

- **Compile** - This is the default scope if any is indicated. The dependencies are available in all classpaths of a project.
- **provided** - This is used when JDK or a web container is going to provide a dependency at runtime.
- **runtime** - This scope indicates that the dependency is not required for compilation, but for runtime.
- **test** - This scope indicates the dependency is required during test compilation and execution phases.
- **system** - This scope is similar to `provided` except that you have to provide JAR which contains in explicitly.
- **import** - This is used inside a `dependencymanagement` tag. It indicates that the specified POM should be replaced with the dependencies in that POM's `dependencymanagement` section.

2.5 Dependency Management

Dependency management is a mechanism to centralize the dependency information. In a multi-module project, you can specify in a parent project all the artifact version and it will be inherited by the child projects.

Below we will see an example where there are two POMs which extend the same parent.

project A

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.example</groupId>
  <artifactId>java-code-geeks-parent-A</artifactId>
  <version>1.0.1</version>
  <packaging>pom</packaging>

  <dependencies>
  <dependency>
  <groupId>group-a</groupId>
  <artifactId>artifact-a</artifactId>
  <version>1.0</version>
  <exclusions>
  <exclusion>
  <groupId> group-c </groupId>
  <artifactId>excluded-artifact</artifactId>
  </exclusion>
  </exclusions>
  </dependency>
  <dependency>
  <groupId>group-a</groupId>
  <artifactId>artifact-b</artifactId>
```

```

    <version>1.0</version>
    <type>bar</type>
    <scope>runtime</scope>
  </dependency>
</dependencies>
</project>

```

project B

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>java-code-geeks-parent-B</artifactId>
  <version>1.0.1</version>
  <packaging>pom</packaging>

  <dependencies>
<dependency>
  <groupId>group-c</groupId>
  <artifactId>artifact-b</artifactId>
  <version>1.0</version>
  <type>war</type>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>group-a</groupId>
  <artifactId>artifact-b</artifactId>
  <version>1.0</version>
  <type>bar</type>
  <scope>runtime</scope>
</dependency>
</dependencies>
</project>

```

These two POMs share a common dependency and each has one non-trivial dependency. Now parent POM will look like below

Parent Project

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.java.code.geeks</groupId>
  <artifactId>java-code-geeks-parent</artifactId>
  <version>1.0.1</version>
  <packaging>pom</packaging>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
      <version>1.0</version>

      <exclusions>
        <exclusion>
          <groupId>group-c</groupId>
          <artifactId>excluded-artifact</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
</dependencyManagement>

```

```

</dependency>

<dependency>
  <groupId>group-c</groupId>
  <artifactId>artifact-b</artifactId>
  <version>1.0</version>
  <type>war</type>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>group-a</groupId>
  <artifactId>artifact-b</artifactId>
  <version>1.0</version>
  <type>bar</type>
  <scope>runtime</scope>
</dependency>
</dependencies>
</dependencyManagement>

</project>

```

And two child POMs will be like below

Child A

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.example</groupId>
  <artifactId>java-code-geeks-child-A</artifactId>
  <version>1.0.1</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
    </dependency>

    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <type>bar</type>
    </dependency>
  </dependencies>
</project>

```

Child B

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.java.code.geeks</groupId>
  <artifactId>java-code-geeks-child-B</artifactId>
  <version>1.0.1</version>
  <packaging>pom</packaging>
  <dependencies>

```

```
<dependency>
  <groupId>group-c</groupId>
  <artifactId>artifact-b</artifactId>

  <type>war</type>
</dependency>

<dependency>
  <groupId>group-a</groupId>
  <artifactId>artifact-b</artifactId>

  <type>bar</type>
</dependency>
</dependencies>
</project>
```

2.6 System Dependencies

System dependencies are specified with `systemPath` under system scope and they are always available and are not looked up in repository. They tell Maven about dependencies which are provided by JDK or VM. A simple example as below shows that of JDBC extension:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.example</groupId>
  <artifactId>java-code-geeks-jdbcexample</artifactId>
  <version>1.0.1</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>javax.sql</groupId>
      <artifactId>jdbc-stdext</artifactId>
      <version>2.0</version>
      <scope>system</scope>
      <systemPath>${java.home}/lib/rt.jar</systemPath>
    </dependency>
  </dependencies>
</project>
```

2.7 Conclusion

Maven has provided a powerful capabilities to developer to manage their multi-projects under one roof. In this example, we saw how to use Maven for dependency management.

2.8 Related Articles

[Maven Introduction](#)

Chapter 3

Difference Between ANT and Maven

In this example we are going to see some differences between ant and maven.

Maven is a build automation tool used mainly for java projects from apache.

Apache **Ant** is a Java library and command-line tool whose mission is to drive processes described in build files

We are going to see an example of a project modeled with ant and the same project modeled with ant.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- Ant 1.9.3
- JDK 1.8.0_65 64bits

3.1 Introduction

Both tools come from apache software foundation, and both tools are designed to resolve the same thing, in other words, related to support the software build process. Ant was released in early 2000, it has a similar structure with unix make task. It is implemented in java language and is best situated to build java projects.

In front of it, Maven was originally released in 2004, and is suitable for manage and build java projects. Maven is an ant evolution and is capable of manage 'out of the box' some things like manage dependencies, build lifecycle events (such as compile, test, package, etc. . .) and so many things without any action needed by the user.

In this example we are going to see the same project under both technologies, and how we have to organize the project in order to achieve the same result.

The example project is a java project with an unique dependency (log4j) that is packaged as a jar file.

3.2 High level comparison

Ant and maven has several differences between themselves, there are some deep and important differences in how they manage the project build, dependencies, lifecycle management, the way that it execute task, and many other things.

Let's see the more important ones:

3.2.1 High level comparison

Project structure: Ant has not a defined project conventions, you can put things in any place, and later instruct ant for know where the things are. Maven has a project conventions and has several archetypes for predefined projects, so maven is easier because it knows where things are if you follow the project convention.

3.2.2 Execution way

Ant is a procedural tool, you have to tell it when, what and how it has to do all the things: compile, then copy, then package, then deploy, etc. . . Maven is a declarative tool, you only have to declare your project object model (pom) and put your source code and resources in the correct folder, mave will take care of the rest for you.

3.2.3 Lifecycle management

Ant do not have a lifecycle management, you have to declare some goals and define which of those goals run first, what run later and so on manually. Maven has a lifecycle management.

3.2.4 Dependency management

Ant does not have any mechanism to manage dependencies, you have to manage it manually: donwload all dependencies, place the dependencies in a folder and later copy it to the packaged artifact, maven has a mechanism to manage dependencies, you have a local repository that acts like a cache and you can define some remote repositories in order to download more dependencies, you only have to define all your dependencies needed and maven will download it for you.

3.3 Ant project

In this example we have a java project that will be packaged as a jar file. The project has an example class that defines a Main method that print Hello World! with log4j framework.

You can see below the ant project

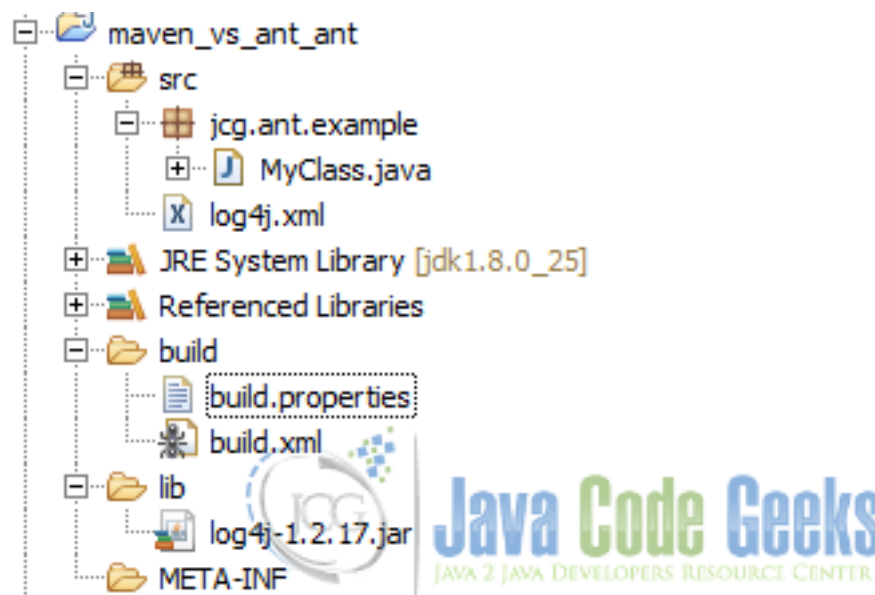


Figure 3.1: Ant project

We have a source package called `src`, inside of it you will find a package called `jcgc.ant.example` with a class called `MyClass`. You will find also a file called `log4j.xml` which is de log for java descriptor file.

Inside the `lib` folder you will find all the needed libs in order to compile the project. In this example we only need the `log4j` jar file. You have to manually instruct eclipse to add it to the compile and runtime claspath through `Project → Properties → Java Build Path`.

Inside `build` folder you will find the ant build script, called `build.xml` and a properties file called `build.properties`. Below you can see the ant build descriptor content

ant build file:

```
<project name="jcgc" basedir="." default="generate_jar">
  <!-- Fichero Properties -->
  <property file="build.properties"/>

  <!-- Create folders target -->
  <target name="create_folders">
    <echo>Crearing needed folders...</echo>
    <mkdir dir="${app_dir}"/>
    <mkdir dir="${app_dir}/META-INF"/>
    <echo>Done!</echo>
  </target>

  <!-- Compilarion Target -->
  <target name="compile">
    <echo>Compiling classes...</echo>
    <javac
      encoding="UTF-8"
      classpath="${classpath}"
      srcdir="${source_dir}"
      destdir="${app_dir}/${classes_dir}"
      debug="true"/>
    <echo>Done!</echo>
  </target>

  <!-- Copy Target -->
  <target name="copy">
    <echo>Copying files...</echo>
    <copy todir="${app_dir}/${meta_dir}">
      <fileset dir="${root_dir}/${meta_dir}/" includes="*.xml"/>
    </copy>
    <echo>Copying META-INF</echo>

    <copy file="${root_dir}/${source_dir}/log4j.xml" todir="${app_dir}" />

    <echo>Cpoying classes...</echo>
    <echo>Done!</echo>
  </target>

  <!-- Clean Target -->
  <target name="clean">
    <delete dir="${app_dir}"/>
  </target>

  <target name="generate_jar">
    <echo>Generating jar...</echo>
    <antcall target="create_folders"/>
    <antcall target="compile"/>
    <antcall target="copy"/>
    <jar destfile="jcgc.jar" basedir="${app_dir}"/>
    <echo>Done!</echo>
    <antcall target="clean"/>
  </target>
</project>
```

```
</target>
</project>
```

ant build properties file:

```
generated_dir=generated
root_dir=.
app_dir=app
meta_dir=../META-INF
classes_dir=.
lib_dir=../lib
jars_dir=jars
source_dir=../src
compiled_classes=classes
classpath=../lib/log4j-1.2.17.jar;
```

The project has a default task called `generate_jar`. This task is based on other tasks

- `create_folders` → Create the needed folders in order to build the jar structure. This task creates the app directory inside the build directory and the app/META-INF directory, that is the base structure for the jar to build
- `compile` → This task compiles all the classes inside the source classes directory. It will put the compiled classes inside the output app folder
- `copy` → This task is responsible for copy all the needed files inside the output folder

After those task are executed, the main task will pack the output folder as a jar file, after all, will execute the `clean` task in order to delete all the temporary folders used for build the jar file.

If you run the ant descriptor file, you will see an output like this

ant run output:

```
Buildfile: C:\workspace\i+d\maven_vs_ant_ant\build\build.xml
generate_jar:
  [echo] Generating jar...
create_folders:
  [echo] Creating needed folders...
  [mkdir] Created dir: C:\workspace\i+d\maven_vs_ant_ant\build\app
  [mkdir] Created dir: C:\workspace\i+d\maven_vs_ant_ant\build\app\META-INF
  [echo] Done!
compile:
  [echo] Compiling classes...
  [javac] C:\workspace\i+d\maven_vs_ant_ant\build\build.xml:21: warning: '↔
    includeantruntime' was not set, defaulting to build.sysclasspath=last; set to false ↔
    for repeatable builds
  [javac] Compiling 1 source file to C:\workspace\i+d\maven_vs_ant_ant\build\app
  [echo] Done!
copy:
  [echo] Copying files...
  [echo] Copying META-INF
  [copy] Copying 1 file to C:\workspace\i+d\maven_vs_ant_ant\build\app
  [echo] Copying classes...
  [echo] Done!
  [jar] Building jar: C:\workspace\i+d\maven_vs_ant_ant\build\jcg.jar
  [echo] Done!
clean:
  [delete] Deleting directory C:\workspace\i+d\maven_vs_ant_ant\build\app
BUILD SUCCESSFUL
Total time: 979 milliseconds
```

Now, inside the build folder you will find the `jcg.jar` file.

3.4 Maven project

In this example we have a java project that will be packaged as a jar file. The project has an example class that defines a Main method that print Hello World! with log4j framework.

You can see below the maven project



Figure 3.2: Maven project

You will see the maven folder convention for code and resources: `src/main/java` and `src/main/resources` and also for test environment: `src/test/java` and `src/test/resources`.

Inside main java you will find the `MyClass` class, and inside the java resources you will find the `log4j` descriptor file.

You only have to define the needed dependencies inside maven descriptor file `pom.xml` in the project root folder. You can see it below

`pom:`

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven_vs_ant_maven</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven VS Ant :: example</name>
  <url>https://maven.apache.org</url>

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

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
  </dependencies>
</project>
```

```
        </dependency>
    </dependencies>
</project>
```

Maven will take care of download and cache the required dependencies, compile the code and generate the jar file. If you run the maven descriptor file with `clean install` goals, you will see an output like this

maven run output:

```
[INFO] Scanning for projects...
[INFO]
[INFO]
[INFO] Building Maven VS Ant :: example 1.0.0-SNAPSHOT
[INFO]
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ maven_vs_ant_maven ---
[INFO] Deleting C:\workspace\i+d\maven_vs_ant_maven\target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven_vs_ant_maven ←
---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ maven_vs_ant_maven ---
[INFO] Compiling 1 source file to C:\workspace\i+d\maven_vs_ant_maven\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ ←
maven_vs_ant_maven ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ ←
maven_vs_ant_maven ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven_vs_ant_maven ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ maven_vs_ant_maven ---
[INFO] Building jar: C:\workspace\i+d\maven_vs_ant_maven\target\maven_vs_ant_maven-1.0.0- ←
SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ maven_vs_ant_maven ---
[INFO] Installing C:\workspace\i+d\maven_vs_ant_maven\target\maven_vs_ant_maven-1.0.0- ←
SNAPSHOT.jar to C:\Users\fhernans\.m2\repository\com\javacodegeeks\examples\ ←
maven_vs_ant_maven\1.0.0-SNAPSHOT\maven_vs_ant_maven-1.0.0-SNAPSHOT.jar
[INFO] Installing C:\workspace\i+d\maven_vs_ant_maven\pom.xml to C:\Users\fhernans\.m2\ ←
repository\com\javacodegeeks\examples\maven_vs_ant_maven\1.0.0-SNAPSHOT\ ←
maven_vs_ant_maven-1.0.0-SNAPSHOT.pom
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 3.329s
[INFO] Finished at: Mon Mar 21 12:58:33 CET 2016
[INFO] Final Memory: 14M/209M
[INFO]
```

After running the maven command you will find the jar file under target folder.

3.5 Projects difference

As you can see ant require that you indicates all the needed task to do, in the correct order, and depends on you to put all the things in the correct place, to compile the classes and so on. . .

In the other side, maven take care for you of compile classes, copy resources, generate jars file and so on. . .

Maven project is more standard, is easier to create and to maintain and is better for manage dependencies than ant project.

3.6 Conclusions

As we have saw the maven project is much easier than the ant project. Maven will take care of some parts for you automatically, in front of it, you have to instruct ant in how it has to make those kind of things, like compile the code, copy files, execute tests, build the result structure, package it as a jar file and so on. With maven, you only have to be worried about tell it what are the needed dependencies.

3.7 Download

Download

You can download the full source code of this example here: [maven vs ant](#)

Chapter 4

Maven Project Structure Example

In this example we are going to see maven project structure and how the projects are organized.

Maven is a build automation tool used mainly for java projects from apache.

We are going to see some examples of maven project structure.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits

4.1 Introduction

Maven is an universal software project management, in order to get maven users familiar with maven projects, maven defines some conventions or directory layouts.

Through those directory layouts maven achieves an uniform way to organize projects and files inside of it. This a very good approach because you can work on several projects and you always will have the same project structure, so you will switch between projects and you don't have to expend time in order to learn how the project is organized.

You can see a typical `jar` maven project structure here



Figure 4.1: Jar structure

You can see a typical `war` maven project structure here

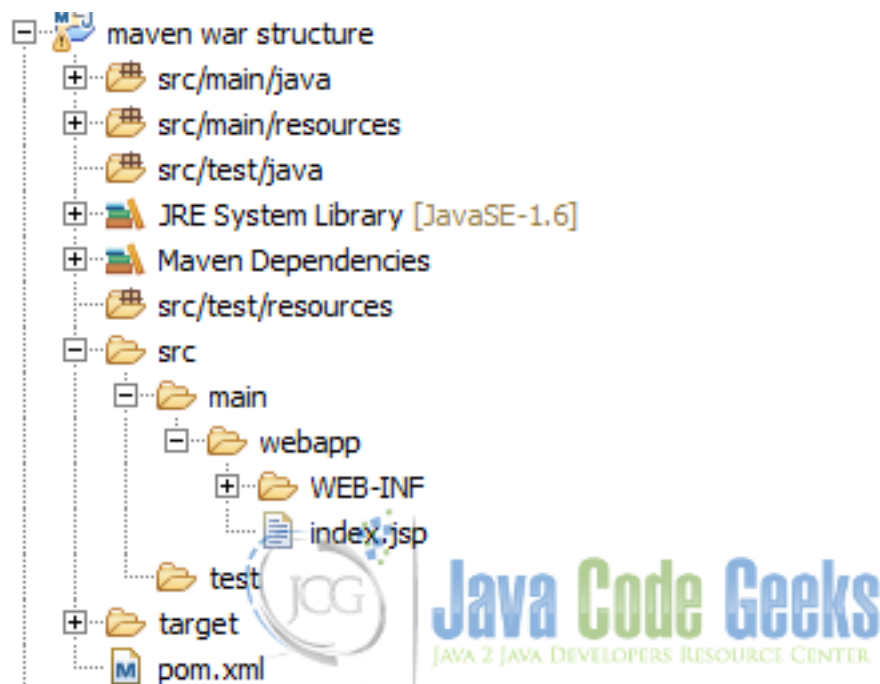


Figure 4.2: War structure

4.2 Directory layout. Files

Maven defines some conventions in order to organize the normal files inside a project. In this directories you can put all application sources files.

The directories are the following

- `src/main/java`
- `src/test/java`
- `src/it`

4.2.1 `src/main/java`

Inside this folder you can put all the application source files. Classes and packages for the main (real) artifact should be put in this folder.

All the content inside of this directory will be put in the classpath of the generated artifact. If the artifact is a `jar` file, all the classes and packages will be in the root folder of the generated `jar`, so it will be available by default on the runtime classpath.

If the artifact is a `war`, all the classes and packages will be put inside the `WEB-INF/classes` directory, so it will be available on the runtime classpath by default.

When the project is build or packaged all those classes and packages will be put in the `target` folder.

If you use `eclipse` as your IDE, this directory will be put inside the `java build path` automatically when you give the `maven nature` to the project.

4.2.2 `src/test/java`

Inside this folder you can put all the application test source files. Classes and packages for the test artifact should be put in this folder.

All the content inside of this directory will NOT be put in the classpath of the generated artifact.

When the project is build or packaged all those classes and packages will be put in the `target` folder.

When you run your test you must be aware that `maven surefire plugin` will run the classes from the `target` directory.

If you use `eclipse` as your IDE, this directory will be put inside the `java build path` automatically when you give the `maven nature` to the project.

4.2.3 `src/it`

Inside this folder you can put all the application integration test source files. Classes and packages for the integration test artifact should be put in this folder.

All the content inside of this directory will NOT be put in the classpath of the generated artifact.

When the project is build or packaged all those classes and packages will be put in the `target` folder.

When you run your integration test you must be aware that the implicated plugin will run the classes from the `target` directory.

If you use `eclipse` as your IDE, this directory will be put inside the `java build path` automatically when you give the `maven nature` to the project.

4.3 Directory layout. Resources

Maven defines some conventions in order to organize the normal files inside a project. In this directories you can put all application sources files.

The directories are the following

- `src/main/resources`
- `src/test/resources`
- `src/main/filters`
- `src/test/filters`

4.3.1 `src/main/resources`

Inside this folder you can put all the application resource files. Resources for the main (real) artifact should be put in this folder.

All the content inside of this directory will be put in the classpath of the generated artifact. If the artifact is a `jar` file, all the resources will be in the root folder of the generated `jar`, so it will be available by default on the runtime classpath.

If the artifact is a `war`, all resources will be put inside the `WEB-INF/classes` directory, so it will be available on the runtime classpath by default.

When the project is build or packaged all those resources will be put in the `target` folder.

If you use `eclipse` as your IDE, this directory will be put inside the `java build path` automatically when you give the `maven nature` to the project.

4.3.2 `src/test/resources`

Inside this folder you can put all the application test resource files. Resources for the test artifact should be put in this folder.

All the content inside of this directory will NOT be put in the classpath of the generated artifact.

When the project is build or packaged all those test resources will be put in the `target` folder.

When you run your test you must be aware that `maven surefire plugin` will use resources from the `target` directory.

If you use `eclipse` as your IDE, this directory will be put inside the `java build path` automatically when you give the `maven nature` to the project.

4.3.3 `src/main/filters`

Inside this folder you can put all the application filters files. Filters for the artifact should be put in this folder.

- You can see more details [here](#).

4.3.4 `src/test/filters`

Inside this folder you can put all the application test filters files. Filters for the test artifact should be put in this folder.

- You can see more details [here](#).

4.4 Directory layout. Misc

Maven defines some conventions for several purposes, like

- `src/assembly`
- `LICENSE.txt`: This file represents the project license file.
- `NOTICE.txt`: This file are notes, notices and attributions for the project, as third party libraries mentions, licenses, etc...
- `README.txt`: Project readme file.

4.4.1 `src/assembly`

Inside this folder you can put all the maven assembly plugin file. This files will be used by the maven assembly plugin.

- You can see an example of the maven assembly plugin [here](#).

4.5 Directory layout. Webapp

Maven project structure defines a folder in order to store all resources and files needed by a web application.

- `src/main/webapp`

Inside this folder you can put all the required files for a web application like `jsp` files, `js` files, `html` files, `css` files, `template` files, `reports` files, `WEB-INF` files (like `web.xml`), `META-INF` files, etc...

All the content inside of this directory will be put in the classpath of the generated `war` artifact, all resources will be put inside the `WEB-INF` directory, so it will be available on the runtime classpath by default.

When the project is build or packaged all those resources will be put in the `target/WEB-INF` folder.

4.6 Directory layout. Target

The target folder is the maven default output folder. When a project is build or packaged, all the content of the sources, resources and web files will be put inside of it, it will be used for construct the artifacts and for run tests.

You can delete all the target folder content with `mvn clean` command.

4.7 Directory layout. Pom file

The pom (Project Object Model) file is a maven special file that describe how the project have to be build and from when maven should download artifacts/dependencies, what are those dependencies and so many more things.

This file is placed on the root project folder.

4.8 Directory layout. Personal files

You can put more folders inside of the project structure, but in those cases you have to instruct maven on the build tag of the pom file in order to manage those folder.

4.9 Conclusions

As you have seem, maven defined a good and a clear project structure in order to familiar users across projects. This is a very important point if you planned to work in several projects, avoiding the need to learn how different project are structured.

Chapter 5

Maven Settings.xml example

In this example we are going to see the maven settings xml file and most of its features. Maven is a build automation tool used mainly for java projects from apache. You can access to the maven settings reference [here](#). We are going to see some examples of the maven settings possibilities.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits

5.1 Introduction

Maven has a high level of customization, you can define several things in order to define how maven works. Maven provides a configuration file called `settings.xml` in which you can customize the `settings` tag. We are going to see several things you can configure inside the `settings` tag in the next bullets.

The `settings.xml` could be in two different places:

- **Maven installation:** `$M2_HOME/conf/settings.xml` (unix notation), where `M2_HOME` is the maven install directory.
- **User directory:** `${user.home}/.m2/settings.xml` (unix notation), where `user.home` is the user home directory.

The following image shows the maven installation directory structure:

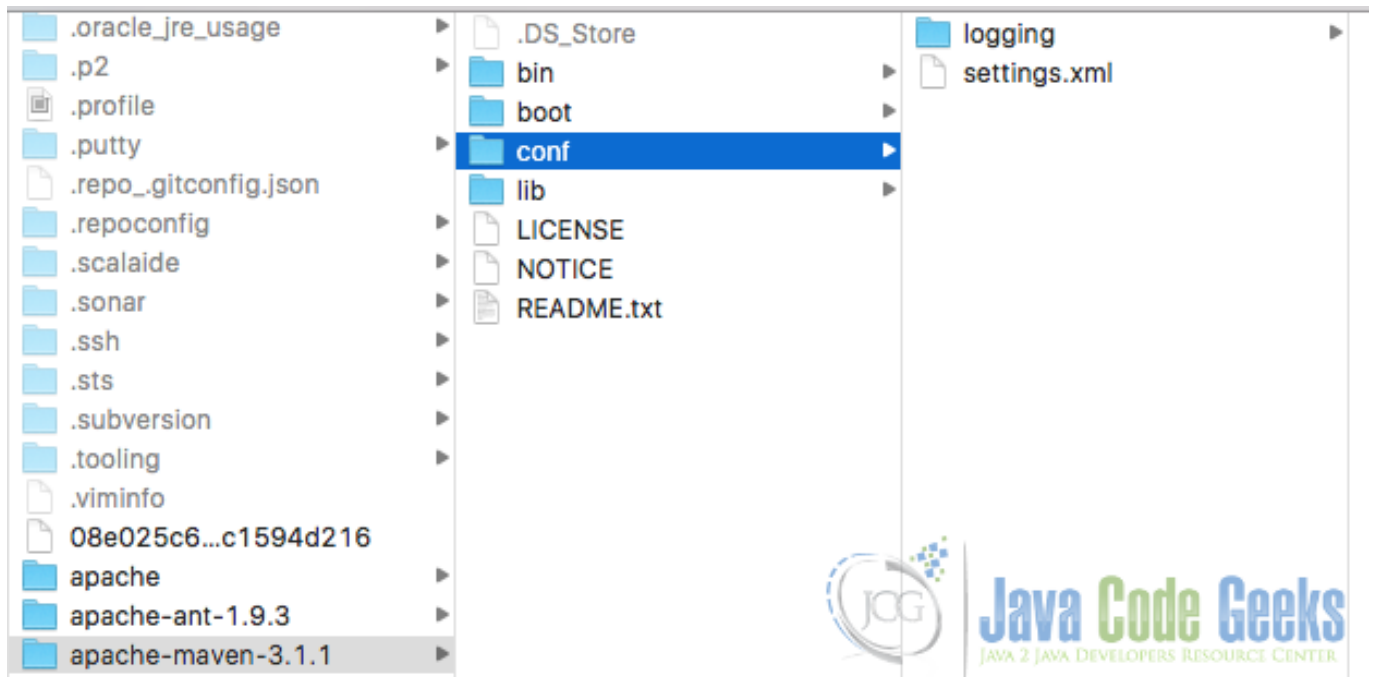


Figure 5.1: Maven installation directory

The following image shows the user local repository structure:

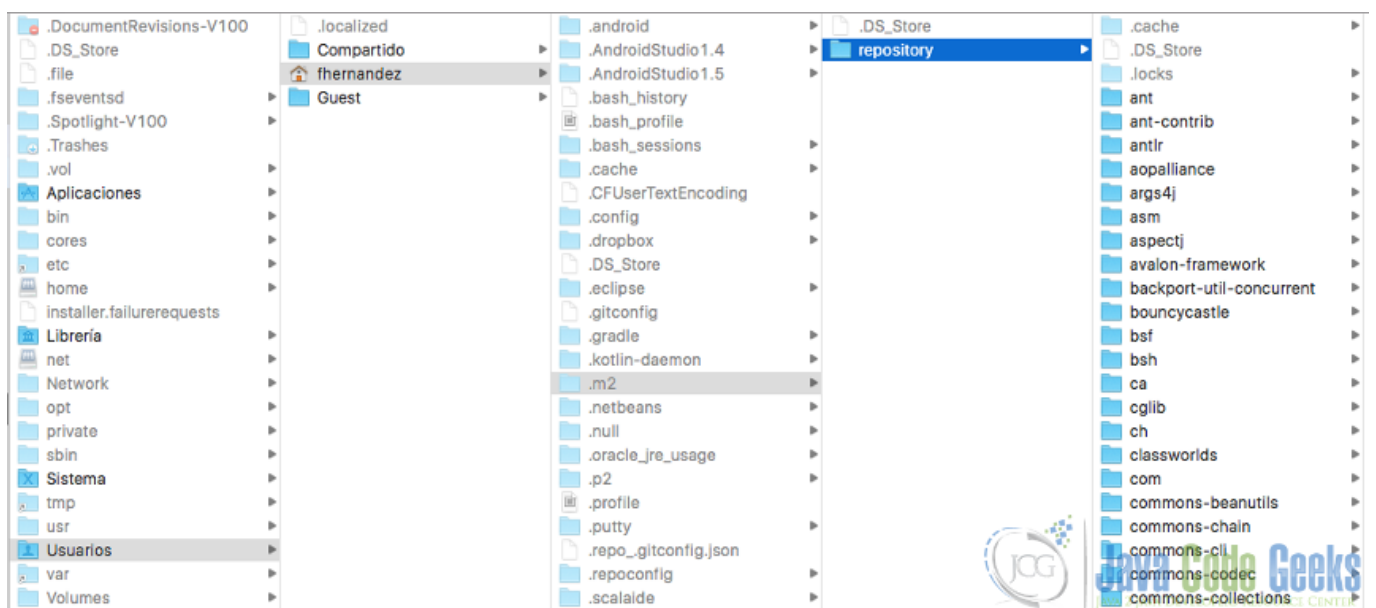


Figure 5.2: User local repository

You can customize both of them. If this is the case, both files will get merged, but be aware that the user-specific `settings` file has more priority than the other one.

The `settings` tag defines the following things

- `localRepository`

- interactiveMode
- usePluginRegistry
- offline
- pluginGroups
- servers
- mirrors
- proxies
- profiles
- activeProfiles

Let's see all of them in more detail:

5.2 Single values

The first four elements are simple properties that accept single values. All those fields have default value, so you can skip the definition and the default values will be used. Let's see it one by one:

5.2.1 localRepository

Indicates where is located the local maven repository. By default, this local repository is located under the user home folder in `${user.home}/.m2/repository`, but with this property you can define another location. This is useful when you need to share a local repository along several users inside a project or organization.

You can see the local repository below:

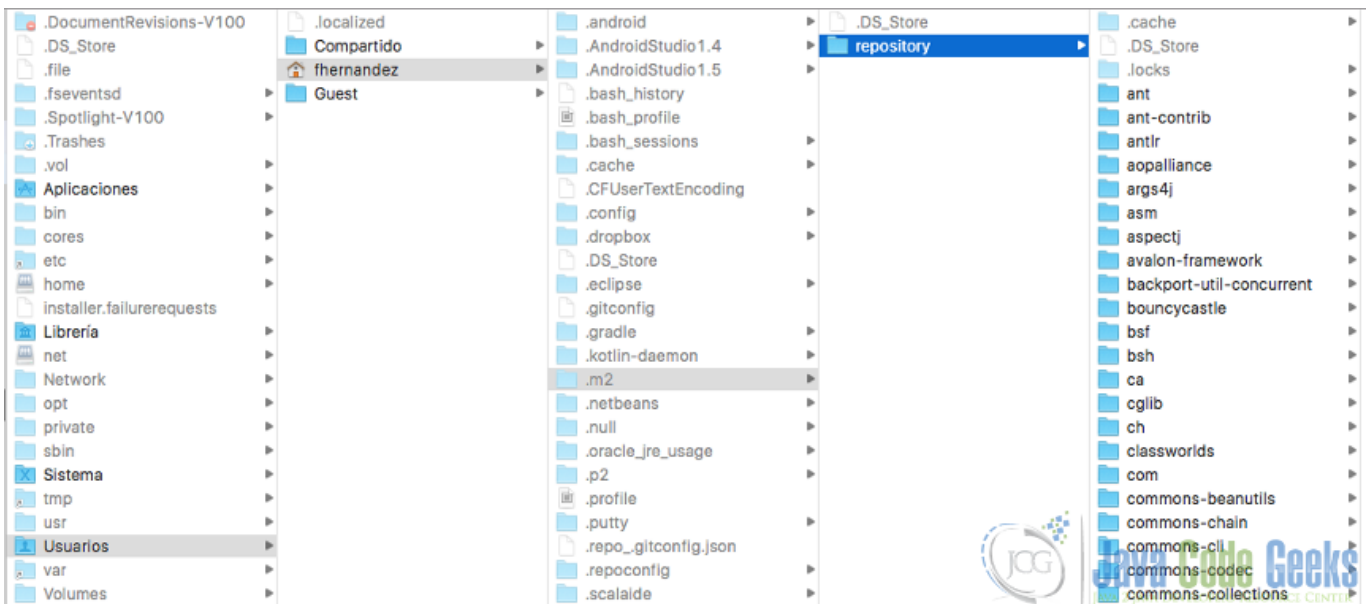


Figure 5.3: User local repository

The local repositories allow you to work in offline mode, and act as a cache for your artifacts, plugin and all other stuffs needed. Use local repositories as much as you can, maven will use it by default, but you should use it anyway to improve your builds and maven operations.

5.2.2 interactiveMode

Indicates if maven should interact with the user for input. Is a true/false field. Defaults to true.

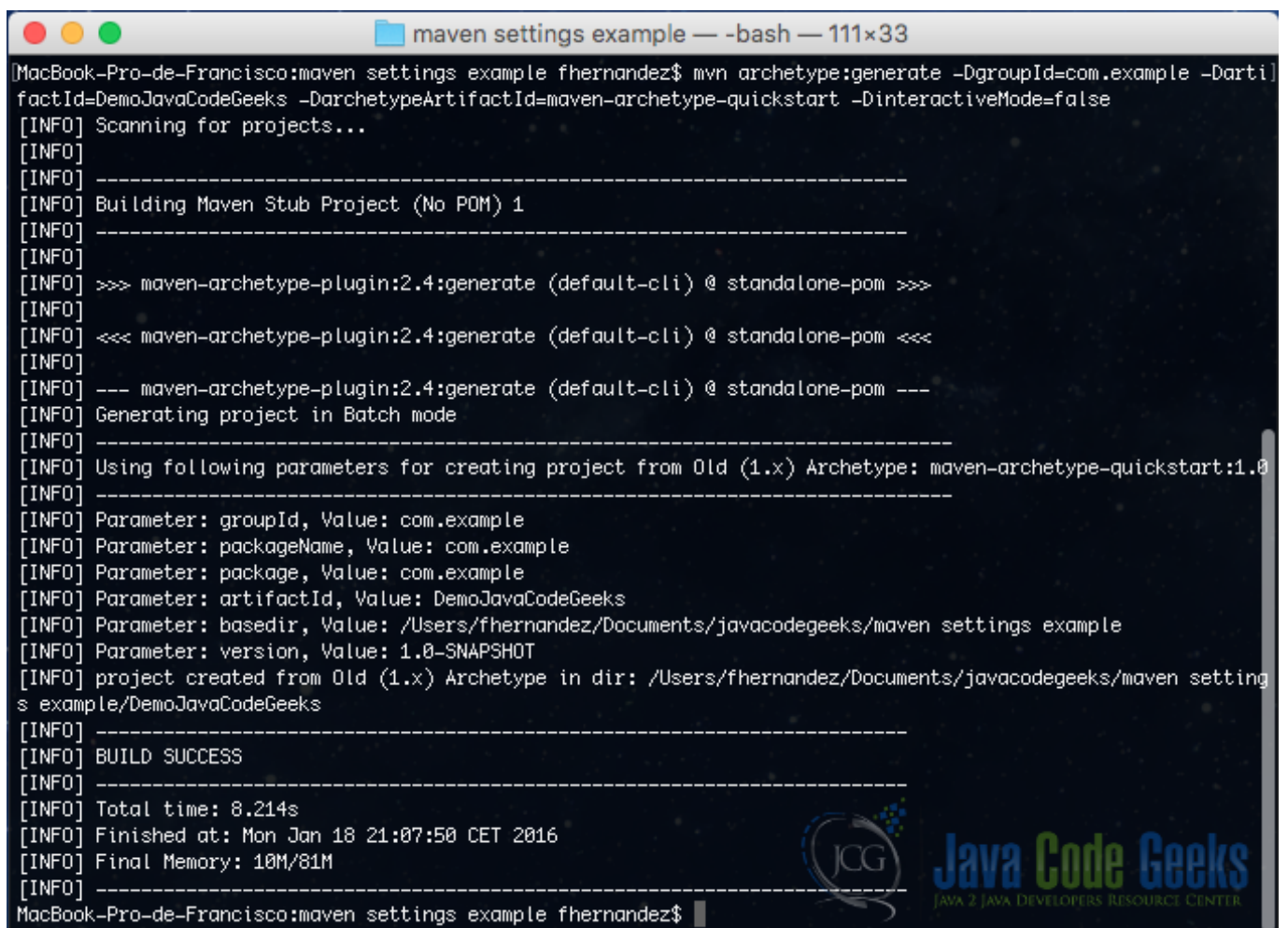
This option can be useful when we want to create an empty and default java project. With this option activated, maven will not ask us anything and the process will be faster. We can test it with the following instruction:

Invocation example

```
mvn archetype:generate -DgroupId=com.example -DartifactId=DemoJavaCodeGeeks -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Note that we have defined `interactiveMode` in the command line, this is only necessary if we have set `interactiveMode` in `settings.xml` to true.

The following is the console output:



```
MacBook-Pro-de-Francisco:maven settings example fhernandez$ mvn archetype:generate -DgroupId=com.example -DartifactId=DemoJavaCodeGeeks -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO]
[INFO] -----
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] -----
[INFO] Parameter: groupId, Value: com.example
[INFO] Parameter: packageName, Value: com.example
[INFO] Parameter: package, Value: com.example
[INFO] Parameter: artifactId, Value: DemoJavaCodeGeeks
[INFO] Parameter: basedir, Value: /Users/fhernandez/Documents/javacodegeeks/maven settings example
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /Users/fhernandez/Documents/javacodegeeks/maven settings example/DemoJavaCodeGeeks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 8.214s
[INFO] Finished at: Mon Jan 18 21:07:50 CET 2016
[INFO] Final Memory: 10M/81M
[INFO] -----
MacBook-Pro-de-Francisco:maven settings example fhernandez$
```

Figure 5.4: Non InteractiveMode output

5.2.3 usePluginRegistry

There is a file called `plugin-registry.xml` in `${user.home}/.m2` folder. This field indicates if maven should use that file to manage plugins versions. Defaults to false.

The Maven 2 plugin registry (`~/m2/plugin-registry.xml`) is a mechanism to help the user exert some control over their build environment. Rather than simply fetching the latest version of every plugin used in a given build, this registry allows the user to

peg a plugin to a particular version, and only update to newer versions under certain restricted circumstances. There are various ways to configure or bypass this feature, and the feature itself can be managed on either a per-user or global level.

You can see more options and possibilities from the plugin registry mechanism [here](#).

5.2.4 offline

Indicates if maven should work in offline mode, this is, maven can not connect to remote servers. Defaults to false.

Now, we can see an example of those four fields in the following `settings.xml` example with the default values:

single Values example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>>true</interactiveMode>
  <usePluginRegistry>>false</usePluginRegistry>
  <offline>>false</offline>

</settings>
```

You have to be careful if you need to use some stuff that you had not used before in your machine, because with the offline mode activated, maven is unable to download those stuffs.

This is a common problem related to offline mode.

5.3 PluginGroups

The `pluginGroup` accepts multiples values, when a plugin is invoked, maven will search along this element in order to find the `groupId` for the plugin. It makes more easy the maven execution. You can define several plugins `groupId`, by default, it contains the following ones:

- `org.apache.maven.plugins`
- `org.codehaus.mojo`

Let's see an example:

Group Plugin example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <pluginGroups>
    <pluginGroup>org.mortbay.jetty</pluginGroup>
    <pluginGroup>your.own.plugin.groupId</pluginGroup>
  </pluginGroups>

</settings>
```


Now, you can invoke the goals defined in plugins that belong to those `groupId` without specify it. For example:

Invocation example

```
mvn jetty:run
```

5.4 Servers

The `servers` tags allow us to define some informations that should not be distributed inside of our `pom.xml` files like server username, password, private keys, etc... We can define our repositories and our `distributionManagement` with references to the server configuration in our `settings.xml` or `pom.xml` file. Let's see an example:

Server example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <servers>
    <server>
      <id>server_repo_java_code_geeks</id>
      <username>john</username>
      <password>doeIsMyPass</password>
      <privateKey>${user.home}/.ssh/dsa_key</privateKey>
      <passphrase>my_passphrase</passphrase>
      <filePermissions>774</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
    <server>
      <id>server_repo_java_code_geeks_2</id>
      <username>steve</username>
      <password>steve_password</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>steve_passphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
  </servers>
</settings>
```

As you can see we have defined two servers. The `id` field is the key to reference this server in our `pom.xml` files. We can define some fields related to the server like username and password to connect to the server, permission for files and directories, private keys, etc... Most of the elements are optional but be aware that if you use a private key, you can not use a password, otherwise, private key will be ignored.

Since maven 2.1.10, a mechanism for encrypt password has been added, see [this](#) for more information about it.

5.5 Mirrors

Sometimes a good approach is to create a mirror of a repository, in order to reduce the traffic over the network in a big organization, or to optimize the build operations. The mirror is like a cache of a specific repository. We can define in `settings.xml` those mirrors, so maven will improve its operations. Let's see an example:

Mirror example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <mirrors>
    <mirror>
      <id>centralmirror</id>
      <name>Apache maven central mirror Spain</name>
      <url>https://downloads.centralmirror.com/public/maven</url>
      <mirrorOf>maven_central</mirrorOf>
    </mirror>
    <mirror>
      <id>jcg_mirror</id>
      <name>Java Code Gueeks Mirror Spain</name>
      <url>https://downloads.jcgmirror.com/public/jcg</url>
      <mirrorOf>javacodegeeks_repo</mirrorOf>
    </mirror>
  </mirrors>

</settings>
```

As you can see we have defined two mirrors, one for Apache maven central repository and another one for a fictional java code geeks repository. The field `mirrorOf` should point to a `id` of a defined repository.

The `id` field must not match the field `mirrorOf` value.

You can see more details about the mirrors of repositories [here](#)

5.6 Proxies

We can define a HTTP proxy to allow to maven be able to get access to internet, reaching the needed repositories. Let's see an example:

Proxy example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <proxies>
    <proxy>
      <id>jcg_proxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.javacodegeeks.com</host>
      <port>9000</port>
      <username>proxy_user</username>
      <password>user_password</password>
      <nonProxyHosts>*.google.com|javacodegeeks.com</nonProxyHosts>
    </proxy>
  </proxies>

</settings>
```

As you can see we have defined a HTTP proxy server which is in a host called `proxy.javacodegeeks.com`, listening in port 9000, with a specific user and password and some proxy excluded URL patterns.

5.7 Profiles

Profiles are a maven mechanism that adds the ability to modify some values or properties under certain circumstances. The profile defined in the `settings.xml` file are a reduced version of the profile that we can define inside `pom.xml` file. We can define activation conditions, repositories, `pluginRepositories` and `properties` elements. Beware that if the same profile id are defined in `pom.xml` and `settings.xml`, the values from `settings.xml` will override the values defined in `pom.xml`. Let's see an example:

Profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>>false</activeByDefault>
        <jdk>1.6</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.3200</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>3.0.3</value>
        </property>
        <file>
          <exists>${basedir}/windows.properties</exists>
          <missing>${basedir}/windows_endpoints.properties</missing>
        </file>
      </activation>

      <properties>
        <user.project>${user.home}/your-project</user.project>
        <system.jks>${user.home}/your_jks_store</system.jks>
      </properties>

      <repositories>
        <repository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>https://snapshots.maven.codehaus.org/maven2</url>
          <layout>default</layout>
        </repository>
      </repositories>
    </profile>
  </profiles>
</settings>
```

```
        <pluginRepositories>
          <pluginGroup>your.own.plugin.groupId</pluginGroup>
        </pluginRepositories>

      </profile>
    </profiles>

  </settings>
```

If you see the activation tag, we have defined some elements in order to activate this profile, this profile is not activated by default as we have indicated in `activeByDefault` field. If some of those criterias that we have defined are matched, maven will activate this profile.

We have defined some properties inside `properties` tag. When this profile is active, we can access to those properties anywhere inside `pom.xml` file with `${prop}` notation where `prop` is the name that we have gave to the property.

We have defined some `repositories` and `pluginRepositories` too. This elements can be used when this profile is activated.

5.8 Active profiles

We can place inside of the `activeProfiles` some defined profiles and all of them will be activated regardless of its activation conditions or configuration elements. Let's see an example:

Activate profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <activeProfiles>
    <activeProfile>test</activeProfile>
  </activeProfiles>

</settings>
```

5.9 Conclusions

As we have seen in this example, maven `settings` file allow us to customize maven execution in several different ways and we can accomplish a lot of things with `settings.xml` file.

We can see the entire `settings.xml` file below:

Activate profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="https://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/ ←
    settings-1.0.0.xsd">

  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <usePluginRegistry>false</usePluginRegistry>
  <offline>false</offline>
```

```
<pluginGroups>
  <pluginGroup>org.mortbay.jetty</pluginGroup>
  <pluginGroup>your.own.plugin.groupId</pluginGroup>
</pluginGroups>

<servers>
  <server>
    <id>server_repo_java_code_gueeks</id>
    <username>john</username>
    <password>doeIsMyPass</password>
    <privateKey>${user.home}/.ssh/dsa_key</privateKey>
    <passphrase>my_passphrase</passphrase>
    <filePermissions>774</filePermissions>
    <directoryPermissions>775</directoryPermissions>
    <configuration></configuration>
  </server>
  <server>
    <id>server_repo_java_code_gueeks_2</id>
    <username>steve</username>
    <password>steve_password</password>
    <privateKey>${user.home}/.ssh/id_dsa</privateKey>
    <passphrase>steve_passphrase</passphrase>
    <filePermissions>664</filePermissions>
    <directoryPermissions>775</directoryPermissions>
    <configuration></configuration>
  </server>
</servers>

<mirrors>
  <mirror>
    <id>centralmirror</id>
    <name>Apache maven central mirror Spain</name>
    <url>https://downloads.centralmirror.com/public/maven</url>
    <mirrorOf>maven_central</mirrorOf>
  </mirror>
  <mirror>
    <id>jcg_mirror</id>
    <name>Java Code Gueeks Mirror Spain</name>
    <url>https://downloads.jcgmirror.com/public/jcg</url>
    <mirrorOf>javacodegueeks_repo</mirrorOf>
  </mirror>
</mirrors>

<proxies>
  <proxy>
    <id>jcg_proxy</id>
    <active>>true</active>
    <protocol>http</protocol>
    <host>proxy.javacodegueeks.com</host>
    <port>9000</port>
    <username>proxy_user</username>
    <password>user_password</password>
    <nonProxyHosts>*.google.com|javacodegueeks.com</nonProxyHosts>
  </proxy>
</proxies>

<profiles>
  <profile>
    <id>test</id>
    <activation>
      <activeByDefault>>false</activeByDefault>
      <jdk>1.6</jdk>
    </activation>
  </profile>
</profiles>
```

```
<os>
  <name>Windows XP</name>
  <family>Windows</family>
  <arch>x86</arch>
  <version>5.1.3200</version>
</os>
<property>
  <name>mavenVersion</name>
  <value>3.0.3</value>
</property>
<file>
  <exists>${basedir}/windows.properties</exists>
  <missing>${basedir}/windows_endpoints.properties</missing>
</file>
</activation>

  <properties>
  <user.project>${user.home}/your-project</user.project>
    <system.jks>${user.home}/your_jks_store</system.jks>
  </properties>

  <repositories>
<repository>
  <id>codehausSnapshots</id>
  <name>Codehaus Snapshots</name>
  <releases>
    <enabled>>false</enabled>
    <updatePolicy>always</updatePolicy>
    <checksumPolicy>warn</checksumPolicy>
  </releases>
  <snapshots>
    <enabled>>true</enabled>
    <updatePolicy>never</updatePolicy>
    <checksumPolicy>fail</checksumPolicy>
  </snapshots>
  <url>https://snapshots.maven.codehaus.org/maven2</url>
  <layout>default</layout>
</repository>
</repositories>

  <pluginRepositories>
  <pluginGroup>your.own.plugin.groupId</pluginGroup>
</pluginRepositories>

</profile>
</profiles>

<activeProfiles>
  <activeProfile>test</activeProfile>
</activeProfiles>

</settings>
```

5.10 Download the source code

This was an example about Maven Settings.xml.

Download

You can download the full source code of this example as a text file here: [settings.zip](#)

Chapter 6

Maven Local Repository example

In this example we are going to see some of the capabilities from the maven local repository.

Maven is a build automation tool used mainly for java projects from apache.

We are going to see some examples of the capabilities of the maven local repository.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits

6.1 Introduction

Maven uses spaces for store artifacts and dependencies. Those spaces are called repositories. There are two types of repositories: Remotes and Locals.

Remote repositories are accesed in differentes ways (http, ftp, etc. . .) and contains artifacts and dependencies provided by a third party, one example of this kind of repositories is repo.maven.apache.org which is maven central repository.

Local repositories are copies of a remote repository inside your own installation and acts as a cache. It also can have the copy of your not-deployed-yet artifacts and dependencies.

The structure of both local and remote repositories is the same, there are not differencies between them.

6.2 Local repository structure

The local repository by default is located under the `.m2/repository` folder under the user home folder. Inside of it you will find all the artifacts and dependencies organized in folders for each `group_id`, `artifact_id` and `version`.

You can see the typical structure of a local repository in the following image

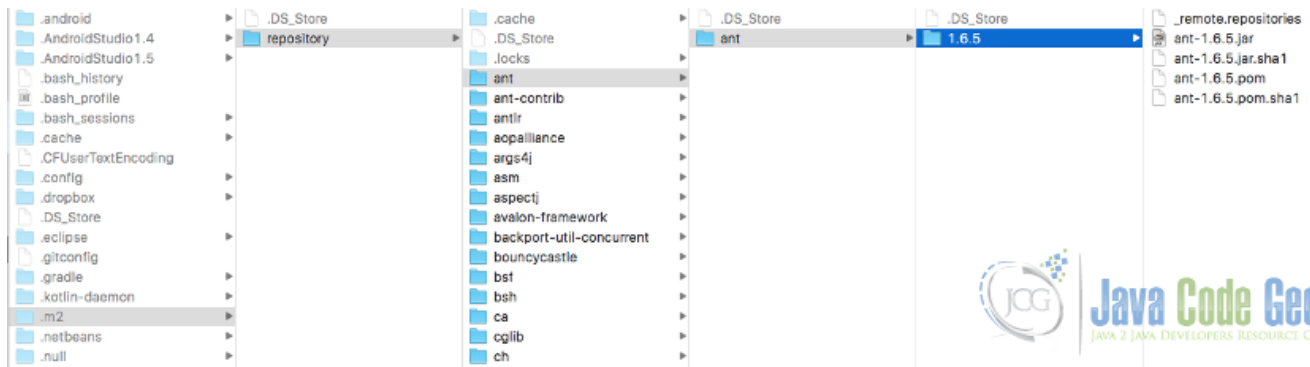


Figure 6.1: Local repository structure

Each folder usually has the jar file, the pom file and meta-files that allow maven to manage the repository status. The files would vary depending on the type of the artifact/dependency.

6.3 Deploying artifacts to the local repository

You can deploy artifacts to the local repository, launching the `mvn install` command, after running it, you can go to the local repository and search a folder for your `group_id`, navigate down inside the folder structure and you will find a folder with the artifact version, inside of it you will see the artifact itself.

You can find more information [here](#)

6.4 Installing artifacts/dependencies in the local repository

You can install a `jar` into your local repository that is not managed by maven and is not under any other remote repository.

Use the following command in order to do that

command:

```
mvn install:install-file -Dfile=./jar_file-1.0.0.jar -DgroupId=your_group -DartifactId= ↵
your_artifact -Dversion=1.0.0 -Dpackaging=jar -DgeneratedPom=true
```

After running the command, you will find inside the local repository a folder called `your_group/your_artifact/1.0.0` and inside of it the jar file, the associated pom and all the related meta-file.

You can find more information [here](#).

6.5 Maven locate artifacts strategy

Maven will always try to find an artifact or dependencies into the local repository first, in order to improve the building process. If you do not have internet connectivity or you are under a very poor internet connection you should install your dependencies/artifacts in your local repository as described in points 3 and 4.

If you are in a organization when only one machine has internet connection you can download all the artifacts/dependencies needed in order to build your project in that machine and then zip the local repository and distribute it among all your team members, so all of you can work offline without problems.

6.6 Conclusions

As we have seen, the maven local repository is an important part of maven, and it acts as a cache for remote artifacts and dependencies. The maven local repository allows you to work offline once you have downloaded all the needed artifacts and dependencies into your local repository.

Chapter 7

Maven Dependency Plugin Example

In this example we are going to see some of the capabilities from the maven dependency plugin.

Maven is a build automation tool used mainly for java projects from apache.

You can access to the maven dependency plugin [here](#).

We are going to see some examples of the capabilities of the maven dependency plugin.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits
- Junit 4.12
- Maven dependency plugin 2.10

7.1 Introduction

The maven dependency plugin provides the capability to manipulate artifacts besides some other capabilities like analyze the project and search unused dependencies. We can call to `mvn dependency:analyze` for example and we don't have to define the maven dependency plugin inside `pom.xml` it will be downloaded and executed when maven needs it.

Despite that, we can define it inside `pom.xml` in order to control some features.

The maven dependency plugin has severals goals defined (from plugin's [page](#)):

- `dependency:analyze` analyzes the dependencies of this project and determines which are: used and declared; used and undeclared; unused and declared.
 - `dependency:analyze-dep-mgt` analyzes your projects dependencies and lists mismatches between resolved dependencies and those listed in your `dependencyManagement` section.
 - `dependency:analyze-only` is the same as `analyze`, but is meant to be bound in a pom. It does not fork the build and execute `test-compile`.
 - `dependency:analyze-report` analyzes the dependencies of this project and produces a report that summarises which are: used and declared; used and undeclared; unused and declared.
-

- `dependency:analyze-duplicate` analyzes the `and` tags in the `pom.xml` and determines the duplicate declared dependencies.
- `dependency:build-classpath` tells Maven to output the path of the dependencies from the local repository in a classpath format to be used in `java -cp`. The classpath file may also be attached and installed/deployed along with the main artifact.
- `dependency:copy` takes a list of artifacts defined in the plugin configuration section and copies them to a specified location, renaming them or stripping the version if desired. This goal can resolve the artifacts from remote repositories if they don't exist in either the local repository or the reactor.
- `dependency:copy-dependencies` takes the list of project direct dependencies and optionally transitive dependencies and copies them to a specified location, stripping the version if desired. This goal can also be run from the command line.
- `dependency:display-ancestors` displays all ancestor POMs of the project. This may be useful in a continuous integration system where you want to know all parent poms of the project. This goal can also be run from the command line.
- `dependency:get` resolves a single artifact, eventually transitively, from a specified remote repository.
- `dependency:go-offline` tells Maven to resolve everything this project is dependent on (dependencies, plugins, reports) in preparation for going offline.
- `dependency:list` alias for `resolve` that lists the dependencies for this project.
- `dependency:list-repositories` displays all project dependencies and then lists the repositories used.
- `dependency:properties` set a property for each project dependency containing the to the artifact on the file system.
- `dependency:purge-local-repository` tells Maven to clear dependency artifact files out of the local repository, and optionally re-resolve them.
- `dependency:resolve` tells Maven to resolve all dependencies and displays the version.
- `dependency:resolve-plugins` tells Maven to resolve plugins and their dependencies.
- `dependency:sources` tells Maven to resolve all dependencies and their source attachments, and displays the version.
- `dependency:tree` displays the dependency tree for this project.
- `dependency:unpack` like `copy` but unpacks.
- `dependency:unpack-dependencies` like `copy-dependencies` but unpacks.

Now, we are going to see some of the capabilities in action with some examples.

7.2 Example project

For this example, we are going to use a java project with maven nature that will be packaged as a jar file. Eclipse Mars comes with maven support out of the box, so you don't have to install anything. Our project will look like this



Figure 7.1: Initial project

At this point, we have an empty maven project. We are going to define the maven dependency plugin inside `pom.xml` in order to test the plugin capabilities.

The `pom.xml` will look like this

`pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/
  maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-dependency-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven dependency :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.2.4.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

```

    </dependencies>

    <build>
      <plugins>
        <plugin>
          <artifactId>maven-dependency-plugin</artifactId>
          <version>2.10</version>
        </plugin>
      </plugins>
    </build>
  </project>

```

The project has a dummy class, and two dependencies: `spring-core` and `junit` (test scoped).

7.3 See dependencies tree

The maven dependency plugin allow us to show the dependencies as a tree. You can see an example in the following `pom.xml`:

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-dependency-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven dependency :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.2.4.RELEASE</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>2.10</version>
        <executions>
          <execution>
            <id>tree</id>
            <phase>generate-sources</phase>
            <goals>

```

```

        <goal>tree</goal>
      </goals>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
</project>

```

Note you can add the `org.eclipse.m2e:lifecycle-mapping` plugin in order to avoid eclipse errors, to do this put the cursor above the error mark over execution in the plugin definition and choose the *Permanently mark goal tree in pom.xml as ignored in Eclipse build* option as you can see in the imagen below

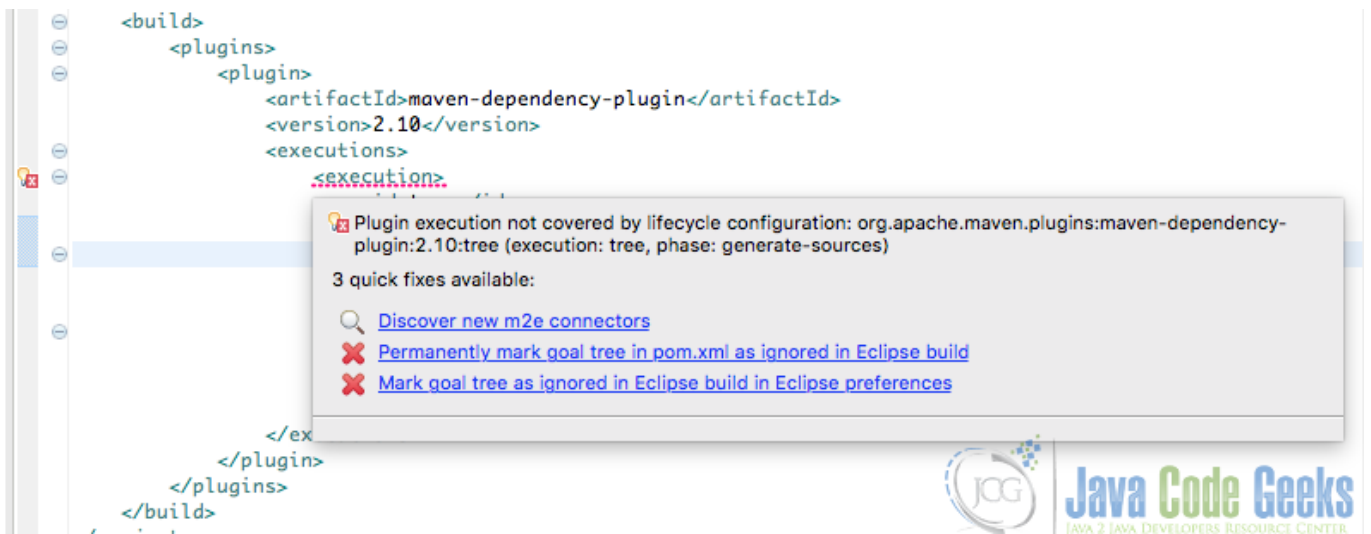


Figure 7.2: Eclipse maven lifecycle errors management

Do this for all the situations you need. This operation will add some code to our `pom.xml` at the end.

You can run the plugin with the dependency:tree `-Doutput=./file.graphml -DoutputType=graphml` command, you will see a file at the root project folder with the graph content called `file.graphml`

output:

```

[INFO] Scanning for projects...
[INFO]
[INFO]
[INFO] Building Maven dependency :: example 1.0.0-SNAPSHOT
[INFO]
[INFO]
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @ maven-dependency-plugin-
example ---
[WARNING] The parameter output is deprecated. Use outputFile instead.
[INFO] Wrote dependency tree to: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven
dependency plugin/file.graphml
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.113 s
[INFO] Finished at: 2015-12-20T20:16:49+01:00
[INFO] Final Memory: 11M/309M
[INFO]

```

You can use some other formats and you can see the tree in the output console rather than in a file, like this

output:

```
[INFO] Scanning for projects...
[INFO]
[INFO]
[INFO] Building Maven dependency :: example 1.0.0-SNAPSHOT
[INFO]
[INFO]
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @ maven-dependency-plugin- ←
    example ---
[INFO] com.javacodegeeks.examples:maven-dependency-plugin-example:jar:1.0.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:compile
[INFO] | \- org.hamcrest:hamcrest-core:jar:1.3:compile
[INFO] \- org.springframework:spring-core:jar:4.2.4.RELEASE:compile
[INFO]     \- commons-logging:commons-logging:jar:1.2:compile
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.023 s
[INFO] Finished at: 2015-12-20T20:11:22+01:00
[INFO] Final Memory: 11M/309M
[INFO]
```

7.4 Build classpath

Another interesting maven dependency plugin feature is the capability to build the project classpath as a string

The following pom.xml shows an example:

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ←
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ←
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-dependency-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven dependency :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.2.4.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

```

    </dependencies>

    <build>
      <plugins>
        <plugin>
          <artifactId>maven-dependency-plugin</artifactId>
          <version>2.10</version>
          <executions>
            <execution>
              <id>build-classpath</id>
              <phase>generate-sources</phase>
              <goals>
                <goal>build-classpath</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </project>

```

You can run the plugin with the `mvn generate-sources` command, you will see an output result like this

output:

```

[INFO] Scanning for projects...
[INFO]
[INFO]
[INFO] Building Maven dependency :: example 1.0.0-SNAPSHOT
[INFO]
[INFO]
[INFO] --- maven-dependency-plugin:2.10:build-classpath (build-classpath) @ maven- ←
    dependency-plugin-example ---
[INFO] Dependencies classpath:
/Users/fhernandez/.m2/repository/junit/junit/4.12/junit-4.12.jar:/Users/fhernandez/.m2/ ←
    repository/org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar:/Users/fhernandez/.m2/ ←
    repository/org/springframework/spring-core/4.2.4.RELEASE/spring-core-4.2.4.RELEASE.jar:/ ←
    Users/fhernandez/.m2/repository/commons-logging/commons-logging/1.2/commons-logging-1.2. ←
    jar
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.107 s
[INFO] Finished at: 2015-12-20T20:46:28+01:00
[INFO] Final Memory: 11M/309M
[INFO]

```

7.5 Other features

As we had seen this plugin have several goals and we have saw a few examples, you can see [here](#) the usage of all other features.

7.6 Conclusions

As you have seen with this example, the maven dependency plugin allow you to do several things in order to fit your dependency management requirements.

7.7 Download the eclipse project

This was an example about Maven dependency plugin.

Download

You can download the full source code of this example here: [maven-dependency-plugin-example.zip](#)

Chapter 8

Maven Shade Plugin Example

In this example we are going to see some of the capabilities from the maven shade plugin.

Maven is a build automation tool used mainly for java projects from apache.

You can access to the maven shade plugin [here](#).

We are going to see some examples of the capabilities of the maven shade plugin.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits
- Maven shade plugin 2.4.3

8.1 Introduction

The maven shade plugin allow us to generate an `uber-jar` and allow us to rename the packages of some dependencies. Uber is a German word that means above or over. In this case, an `uber-jar` is an `over-jar`, in another words, one level up from a simple `jar`. You can generate one jar that contains your package and all its dependencies in one single `jar` file. You can distribute your `uber-jar` as a independent stuff with all its requirements inside of it. The maven shade plugin is beyond maven assembly plugin and is capable to do more things than the maven assembly plugin. The maven shade plugin has one goal defined:

- `shade`: Invoked during the `package` phase

8.2 Example project

For this example, we are going to use a java project with maven nature that will be packaged as a jar file. Eclipse Mars comes with maven support out of the box, so you don't have to install anything. Our project will look like this



Figure 8.1: Example project

At this point, we have an empty maven project. We are going to define the maven shade plugin inside `pom.xml` in order to test the plugin capabilities.

The `pom.xml` will look like this

`pom`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-shade-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven shade plugin :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <spring.version>4.2.2.RELEASE</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
```

```

        <version>${spring.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.4.3</version>
        </plugin>
    </plugins>
</build>
</project>

```

The project has one dummy class called Main. Also, the project defines some dependencies in pom.xml like log4j and spring. In the following bullets, we are going to see some of the maven shade plugin capabilities applied to this project.

8.3 Include/Exclude dependencies

The plugin allow you to control the dependencies that can be included and excluded in the generated jar. The following pom.xml shows how we can exclude the log4j dependency in the result jar file

pom:

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
    XMLSchema-instance"
    xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
        maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.javacodegeeks.examples</groupId>
    <artifactId>maven-shade-plugin-example</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>Maven shade plugin :: example</name>
    <url>https://maven.apache.org</url>

    <properties>
        <spring.version>4.2.2.RELEASE</spring.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>

    <build>

```

```

        <plugins>
          <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
          </plugin>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.4.3</version>
            <executions>
              <execution>
                <phase>package</phase>
                <goals>
                  <goal>shade</goal>
                </goals>
                <configuration>
                  <artifactSet>
                    <excludes>
                      <exclude> ←
                        log4j:log4j:jar: ←
                      </exclude>
                    </excludes>
                  </artifactSet>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </project>

```

After running it with `mvn package` will see an output like this

output:

```

[INFO] Scanning for projects...
[INFO]
[INFO]
[INFO] Building Maven shade plugin :: example 1.0.0-SNAPSHOT
[INFO]
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-shade-plugin- ←
example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ maven-shade-plugin-example ←
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
shade plugin/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven-shade- ←
plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ maven-shade-plugin ←
-example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-shade-plugin-example ←
---
```

```
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ maven-shade-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven shade plugin/ ↵
target/maven-shade-plugin-example-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:2.4.3:shade (default) @ maven-shade-plugin-example ---
[INFO] Excluding log4j:log4j:jar:1.2.17 from the shaded jar.
[INFO] Including org.springframework:spring-core:jar:4.2.2.RELEASE in the shaded jar.
[INFO] Including commons-logging:commons-logging:jar:1.2 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven shade plugin/ ↵
target/maven-shade-plugin-example-1.0.0-SNAPSHOT.jar with /Users/fhernandez/Documents/ ↵
workspaceJavaCodeGeeks/maven shade plugin/target/maven-shade-plugin-example-1.0.0- ↵
SNAPSHOT-shaded.jar
[INFO] Dependency-reduced POM written at: /Users/fhernandez/Documents/ ↵
workspaceJavaCodeGeeks/maven shade plugin/dependency-reduced-pom.xml
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.868 s
[INFO] Finished at: 2016-01-30T00:15:30+01:00
[INFO] Final Memory: 21M/230M
[INFO]
```

The generated jar does not include any class from the `log4j` jar and include all the classes from `spring` jar. You can see it inside target folder.

You can see more capabilities in order to include/exclude dependencies [here](#).

8.4 Shading packages

The plugin allows you to shade a package into another package. The following `pom.xml` shows how we can migrate the classes from `com.javacodegeeks` package to `com.shaded.javacodegeeks`. We can use `include/exclude` tags to define how the migration should work

pom:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-shade-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven shade plugin :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <spring.version>4.2.2.RELEASE</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
```

```

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.4</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.4.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <relocations>
                                <relocation>
                                    <pattern>com. ←
                                        javacodegeeks</ ←
                                        pattern>
                                    <shadedPattern>com. ←
                                        shaded. ←
                                        javacodegeeks</ ←
                                        shadedPattern>
                                    <includes>
                                        <include> ←
                                            com. ←
                                            javacodegeeks ←
                                            .Main</ ←
                                            include>
                                    </includes>
                                </relocation>
                            </relocations>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>

```

After running `mvn package` you will find inside the generated jar a new package with the content of the old one, with the includes/excludes instructions applied to it.

- You can find more details of how use this feature [here](#).

8.5 Attaching the shaded artifact

The plugin by default will replace the original jar with the shaded jar. You can however generate the shaded jar with a qualifier if you need to distribute both of them. The following `pom.xml` shows how we can include the shaded jar inside the original jar.

pom:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-shade-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven shade plugin :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <spring.version>4.2.2.RELEASE</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.4.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <shadedArtifactAttached>true</ ↵
              shadedArtifactAttached>
            <shadedClassifierName>jcg</ ↵
              shadedClassifierName>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Now you can see inside target folder a jar with the `jcg` prefix, you can distribute it in a separate way from the original one.

- You can find more details of how use this feature [here](#).

8.6 Conclusions

As we have seen the maven shade plugin offers some interesting capabilities which we can take advantage of in order to build jar files, you can get more details in the link above at the introduction of this example.

8.7 Download

Download

You can download the full source code of this example here: [maven shade plugin](#)

Chapter 9

Maven War Plugin Example

In this example we are going to see some of the capabilities from the maven war plugin.

Maven is a build automation tool used mainly for java projects from apache.

You can access to the maven war plugin [here](#).

We are going to see some examples of the capabilities of the maven war plugin.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits
- Maven war plugin 2.6

9.1 Introduction

The maven war plugin is called implicitly by the maven lifecycle in the appropriate phase so it is a *special* plugin. We don't need to define it inside `pom.xml` it will be downloaded and executed when maven needs it.

Despite that, we can define it inside `pom.xml` in order to build our project as a war file.

The maven war plugin has some goals defined:

- `war`: Default goal. Invoked during the `package` phase for projects with a `packaging` of `war`
- `exploded`: This goal creates a exploded web app in a specified directory
- `inplace`: This goal is an `exploded` goal variant which generates a exploded web app inside the web application folder in `src/main/webapp`
- `manifest`: this goal generates a manifest file for this web app

There is no need to define that goal inside `pom.xml`, as we said before, maven will invoke that goal when the maven lifecycle have to build the war file.

9.2 Example project

For this example, we are going to use a java project with maven nature that will be packaged as a war file. Eclipse Mars comes with maven support out of the box, so you don't have to install anything. Our project will look like this:

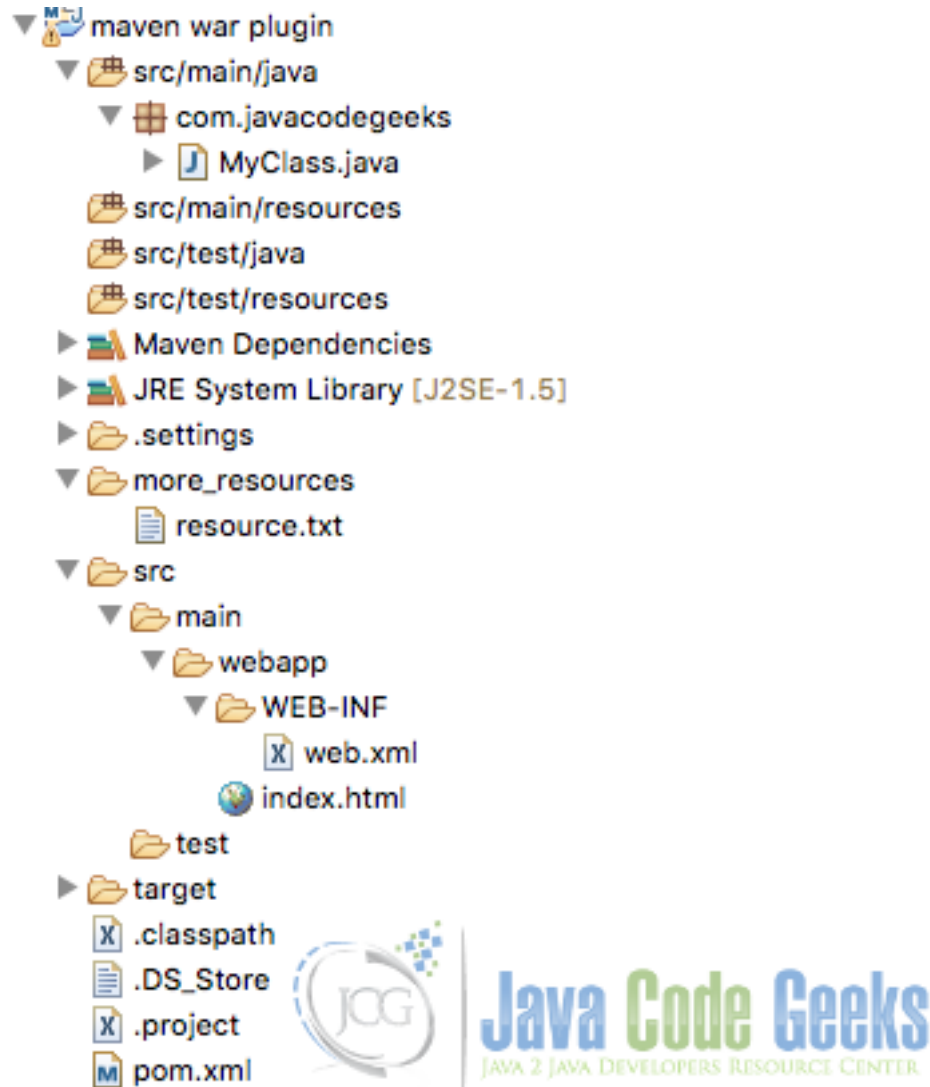


Figure 9.1: Initial project

At this point, we have an empty maven project. We are going to define the maven war plugin inside `pom.xml` in order to test the plugin capabilities.

The `pom.xml` will look like this:

`pom`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/
  maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-war-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
```

```
<name>Maven war :: example</name>
<url>https://maven.apache.org</url>
<packaging>war</packaging>

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

<build>
  <plugins>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.6</version>
    </plugin>
  </plugins>
</build>
</project>
```

The project has one dummy class called `MyClass`, a dummy `web.xml` file inside `src/main/webapp/WEB-INF` folder and a *hello world* `index.html` file on the root `webapp` folder (`src/main/webapp`). Also, the project contains a folder called `more_resources` with a dummy file called `resources.txt`.

In the following bullets, we are going to see some of the maven war plugin capabilities applied to this project.

9.3 Generate a exploded war

The plugin allow you to generate a exploded war as a folder, you can do it running the `mvn war:exploded` command. After running it, you will see a new folder under `target` folder with the generated war exploded, this is, as a normal directory with all of its files inside of it.

9.4 Filtering the war file content

The plugin allow you to filter the war content, you can include/exclude resources from the output war file. The following `pom.xml` shows how to include some stuffs to the war file

`pom:`

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-war-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven war :: example</name>
  <url>https://maven.apache.org</url>
  <packaging>war</packaging>

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

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.6</version>
```

```

        <configuration>
            <webResources>
                <resource>
                    <!-- Relative path to the pom.xml ↵
                    <!-- directory -->
                    <directory>more_resources</ ↵
                    <!-- directory -->
                </resource>
            </webResources>
        </configuration>
    </plugin>
</plugins>
</build>
</project>

```

After running `mvn clean install`, inside the generated war structure you will find the `resource.txt` file in the root folder that comes from the `more_resources` folder in the example project. This is useful when we have other resources (like reports or whatever kind of resources needed) and we want to include it inside war file.

You can find more details of how use this feature [here](#).

9.5 Customizing manifest file

The plugin allow you to control the manifest file, you can include the classpath inside the manifest file for example. This is useful when the war file is under a more complex structure like an ear file and you want to share the dependencies along several modules.

The following `pom.xml` shows how to use this feature

`pom`:

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
  maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-war-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven war :: example</name>
  <url>https://maven.apache.org</url>
  <packaging>war</packaging>

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

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.2.4.RELEASE</version>
      <optional>true</optional>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.6</version>

```

```
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

In order to test the classpath inclusion in `manifest.mf` file we have added the `spring` dependency in `pom.xml`, after run `mvn clean install` we can see a `manifest.mf` file like this:

manifest.mf:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: fhernandez
Class-Path: spring-core-4.2.4.RELEASE.jar commons-logging-1.2.jar
Created-By: Apache Maven 3.3.3
Build-Jdk: 1.8.0_65
```

As you can see, the `manifest` file include a `Class-Path:` property with the classpath entry based on the dependencies defined in `pom.xml` in a transitive way.

9.6 Conclusions

As we have saw the maven war plugin offers some interesting capabilities which we can take advantage of in order to build war files, you can get more details in the link above at the introduction of this example.

9.7 Download the eclipse project

Download

You can download the full source code of this example here: [maven-war-plugin-example.zip](#)

Chapter 10

Maven Compiler Plugin Example

In this example we are going to see most of the capabilities from the maven compiler plugin.

Maven is a build automation tool used mainly for java projects from apache.

You can access to the maven compiler plugin [here](#).

We are going to see some examples of the capabilities of the maven compiler plugin.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits
- Maven compiler plugin 3.3

10.1 Introduction

The maven compiler plugin is called implicitly by the maven lifecycle in the appropriate phase so it is a *special* plugin. We don't need to define it inside `pom.xml` it will be downloaded and executed when maven needs it.

Despite that, we can define it inside `pom.xml` in order to configure the way that maven should compile our classes.

The maven compiler plugin has two goals defined:

- `compile`: Compile the classes under `src/main/java`
- `test-compile`: Compile the classes under `src/test/java`

There is no need to define these goals inside `pom.xml`, as we said before, maven will invoke these goals when the maven lifecycle have to compile our classes.

Since maven 3 `javax.tools.JavaCompiler` (JDK 6 or newer) is used to compile java classes. The default source settings and also the default target settings is JDK 1.5 independently of the JDK you are using maven with.

We are going to see how we can change and control those things below.

10.2 Example project

For this example, we are going to use a java project with maven nature that will be packaged as a jar file. Eclipse Mars comes with maven support out of the box, so you don't have to install anything. Our project will look like this

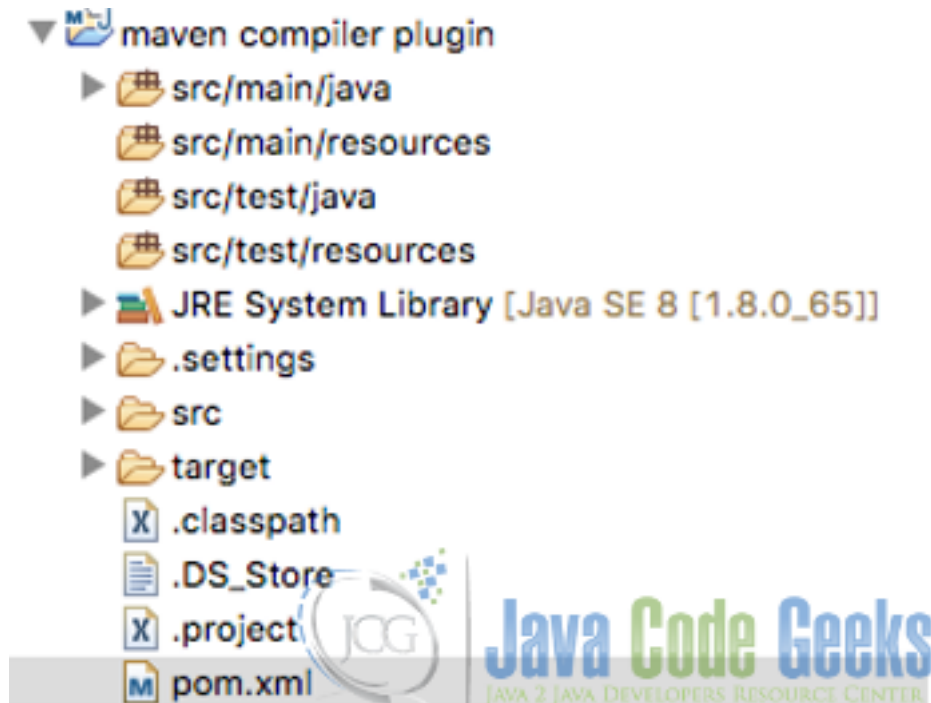


Figure 10.1: Initial project

At this point, we have an empty maven project. We are going to define the maven compiler plugin inside `plugin.xml` in order to test the plugin capabilities.

The `pom.xml` will look like this

`pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/
XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-
  v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-compiler-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven compiler :: example</name>
  <url>https://maven.apache.org</url>

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

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
      </plugin>
    </plugins>
  </build>
</project>
```



```
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.3</version>
        </plugin>
      </plugins>
    </build>
  </project>
```

10.3 Plugin options

We are going to see how we can do several things with the maven compiler plugin:

10.3.1 Set a different JDK to compile classes

We can set a different JDK in order to compile our classes, the following pom.xml shows how we can do it

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-compiler-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven compiler :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <JAVA_HOME_6>/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home</JAVA_HOME_6 <
  >
  </properties>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <executable>${JAVA_HOME_6}/bin/javac</executable>
          <compilerVersion>1.6</compilerVersion>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

You can set the JAVA_HOME_6 in maven settings.xml file or in another properties file in order to make your pom.xml more portable.

10.3.2 Specify a compatible JDK

If you want the compiled classes be compatible with a specific java version, you can set a specific JDK target and source, as you can see below

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven- ↵
    v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-compiler-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven compiler :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <JAVA_HOME_6>/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home</JAVA_HOME_6 ↵
  >
  </properties>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <executable>${JAVA_HOME_6}/bin/javac</executable>
          <compilerVersion>1.6</compilerVersion>
          <source>1.4</source>
          <target>1.4</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

After compile your classes, it will be compatible with JDK 4.

10.3.3 Set some arguments to the compiler

You can pass arguments to the compiler, as you can see below

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven- ↵
    v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-compiler-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven compiler :: example</name>
```

```

<url>https://maven.apache.org</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <JAVA_HOME_6>/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home</JAVA_HOME_6 ←
  >
</properties>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.4</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <executable>${JAVA_HOME_6}/bin/javac</executable>
        <compilerVersion>1.6</compilerVersion>
        <source>1.4</source>
        <target>1.4</target>
        <fork>true</fork>
        <meminitial>128m</meminitial>
        <maxmem>512m</maxmem>
        <verbose>true</verbose>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

10.3.4 Set some specific arguments for the compiler that you have selected

If you have selected a specific compiler you can pass arguments to it with `compilerArgs`, as you can see below

pom.xml:

```

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ←
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven- ←
  v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-compiler-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven compiler :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <JAVA_HOME_6>/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home</JAVA_HOME_6 ←
    >
  </properties>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>

```

```
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <executable>${JAVA_HOME_6}/bin/javac</executable>
    <compilerVersion>1.6</compilerVersion>
    <source>1.4</source>
    <target>1.4</target>
    <fork>true</fork>
    <meminitial>128m</meminitial>
    <maxmem>512m</maxmem>
    <verbose>true</verbose>
    <compilerArgs>
      <arg>-verbose</arg>
      <arg>-Xlint:all,-options,-path</arg>
    </compilerArgs>
  </configuration>
</plugin>
</plugins>
</build>
</project>
```

You can see all the arguments that you can pass to the maven compiler plugin [here](#).

10.4 Conclusions

As we have saw the maven compiler plugin offers some interesting capabilities which we can take advantage of in order to make our code more portable and compatible.

10.5 Download the eclipse project

Download

You can download the full source code of this example here: [maven compiler plugin.zip](#)

Chapter 11

Maven jar plugin example

In this example we are going to see some of the capabilities from the maven jar plugin.

Maven is a build automation tool used mainly for java projects from apache.

You can access to the maven jar plugin [here](#).

We are going to see some examples of the capabilities of the maven jar plugin.

For this example we use the following technologies:

- MAC OSX
- Eclipse Mars.1
- Maven3
- JDK 1.8.0_65 64bits
- Junit 4.12
- Maven jar plugin 2.6

11.1 Introduction

The maven jar plugin provides the capability to build jars files, if you define that your project is packaged as a jar file, maven will call implicitly to this plugin. We don't need to define it inside `pom.xml` it will be downloaded and executed when maven needs it.

Despite that, we can define it inside `pom.xml` in order to control some features of the generated jar file.

The maven jar plugin has two goals defined:

- `jar`: Allow to package our main classes as a jar file
- `test-jar`: Allow to package our test classes as a jar file

The default goal is `jar`, there is no need to define that goal inside `pom.xml`, as we said before, maven will invoke that goal when maven needs it.

You can do several things with the maven jar plugin

- Use a default `manifest` file
 - Custom the `manifest` file
-

- Include/Exclude content from jar file
- Create an additional jar file in the project
- Create a jar with test classes

Let's see all those things in more details.

11.2 Example project

For this example, we are going to use a java project with maven nature that will be packaged as a jar file. Eclipse Mars comes with maven support out of the box, so you don't have to install anything. Our project will look like this

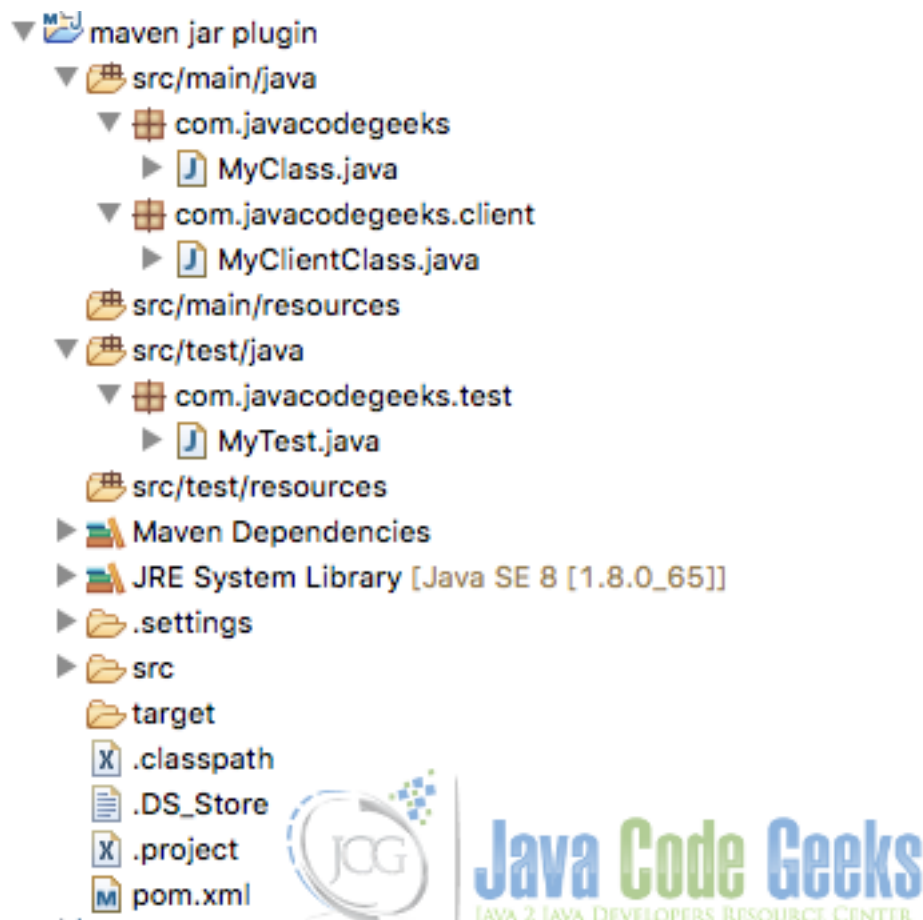


Figure 11.1: Initial project

At this point, we have an empty maven project. We are going to define the maven jar plugin inside `pom.xml` in order to test the plugin capabilities.

The `pom.xml` will look like this

`pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.javacodegeeks.examples</groupId>
<artifactId>maven-jar-plugin-example</artifactId>
<version>1.0.0-SNAPSHOT</version>
<name>Maven jar :: example</name>
<url>https://maven.apache.org</url>

<properties>
  <junit.version>4.12</junit.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.6</version>
    </plugin>
  </plugins>
</build>
</project>
```

The project has a dummy class, a dummy client class and a dummy test.

11.3 Use a default manifest file

The jar plugin allow us to define a default manifest file. With that option, the file located at `${project.build.outputDirectory}/META-INF/MANIFEST.MF` will be the manifest file in the jar. You can see below a pom.xml using this

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-jar-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven jar :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
```

```

    </dependencies>

    <build>
      <plugins>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>2.6</version>
          <configuration>
            <useDefaultManifestFile>>true</ ←
              useDefaultManifestFile>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>

```

You can run the plugin with the `mvn package` command, you will see an output result like this

output:

```

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven jar :: example 1.0.0-SNAPSHOT
[INFO] -----
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-jar ←
  -plugin/2.6/maven-jar-plugin-2.6.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-jar- ←
  plugin/2.6/maven-jar-plugin-2.6.pom (6 KB at 5.5 KB/sec)
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-jar-plugin- ←
  example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ maven-jar-plugin-example ←
  ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
  jar plugin/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven-jar- ←
  plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ maven-jar-plugin- ←
  example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-jar-plugin-example ---
[INFO] Surefire report directory: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
  jar plugin/target/surefire-reports

T E S T S

Running com.javacodegeeks.test.MyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.06 sec

Results :

```



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

[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ maven-jar-plugin-example ---
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver ↵
/2.6/maven-archiver-2.6.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver ↵
/2.6/maven-archiver-2.6.pom (5 KB at 23.3 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus- ↵
archiver/2.8.1/plexus-archiver-2.8.1.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-archiver ↵
/2.8.1/plexus-archiver-2.8.1.pom (5 KB at 23.9 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io ↵
/2.3.2/plexus-io-2.3.2.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io ↵
/2.3.2/plexus-io-2.3.2.pom (3 KB at 15.4 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/commons/commons- ↵
compress/1.9/commons-compress-1.9.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/commons/commons-compress ↵
/1.9/commons-compress-1.9.pom (12 KB at 47.9 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus- ↵
interpolation/1.21/plexus-interpolation-1.21.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus- ↵
interpolation/1.21/plexus-interpolation-1.21.pom (2 KB at 9.3 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus- ↵
archiver/2.9/plexus-archiver-2.9.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-archiver ↵
/2.9/plexus-archiver-2.9.pom (5 KB at 24.7 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io/2.4/ ↵
plexus-io-2.4.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io/2.4/ ↵
plexus-io-2.4.pom (4 KB at 21.7 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver ↵
/2.6/maven-archiver-2.6.jar
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver ↵
/2.6/maven-archiver-2.6.jar (23 KB at 90.7 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus- ↵
archiver/2.9/plexus-archiver-2.9.jar
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-archiver ↵
/2.9/plexus-archiver-2.9.jar (142 KB at 321.9 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io/2.4/ ↵
plexus-io-2.4.jar
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io/2.4/ ↵
plexus-io-2.4.jar (80 KB at 274.4 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/commons/commons- ↵
compress/1.9/commons-compress-1.9.jar
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/commons/commons-compress ↵
/1.9/commons-compress-1.9.jar (370 KB at 526.9 KB/sec)
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven-jar-plugin/ ↵
target/maven-jar-plugin-example-1.0.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 5.812 s
[INFO] Finished at: 2015-12-12T20:35:48+01:00
[INFO] Final Memory: 17M/188M
[INFO] -----
```

After the execution, you will have a jar artifact generated under `target` folder and inside of it under the `META-INF` folder you will find a file called `MANIFEST.MF` with the following content

default MANIFEST file:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: fhernandez
Created-By: Apache Maven 3.3.3
Build-Jdk: 1.8.0_65
```

This is a default manifest file that the plugin has generated for you.

11.4 Custom manifest file

The default contents of the manifest file is described [here](#). Since version 2.1 the maven jar plugin uses Maven Archiver 2.1 so it no longer adds the specification and implementation details in the manifest file, if you want to add those details, you have to manually define it.

You can alter the default content of the manifest file with the archive configuration element. You can see all the possibilities in this [link](#).

We are going to define some elements in the manifest file. You can see below a pom.xml using this

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-jar-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven jar :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <archive>
            <index>>true</index>
            <manifest>
              <addClasspath>>true</addClasspath>
            </manifest>
            <manifestEntries>
              <javacodegeeks>maven jar plugin ↵
                example</javacodegeeks>
```

```

                                <codification>${project.build. ↵
                                    sourceEncoding}</codification>
                                <key>value from javacodegeeks ↵
                                    author</key>
                                </manifestEntries>
                            </archive>
                        </configuration>
                    </plugin>
                </plugins>
            </build>
        </project>

```

As you can see, we define some keys inside of `manifestEntries` tag like `javacodegeeks` as maven jar plugin example, `codification:${project.build.sourceEncoding}` and `purpose:example` from javacodegeeks author. We define inside the tag `archive` the `index` tag as `true`, so a file called `INDEX.LIST` will be generated with the jar files listed inside of it.

You can run the plugin with the `mvn package` command, you will see an output result like this

output:

```

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven jar :: example 1.0.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-jar-plugin- ↵
    example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ maven-jar-plugin- ↵
    ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven-jar- ↵
    plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ maven-jar-plugin- ↵
    example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-jar-plugin-example ---
[INFO] Surefire report directory: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ↵
    jar plugin/target/surefire-reports

T E S T S

Running com.javacodegeeks.test.MyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.06 sec

Results :

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

[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ maven-jar-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven jar plugin/ ↵
    target/maven-jar-plugin-example-1.0.0-SNAPSHOT.jar

```

```
[INFO] JarArchiver skipping indexJar /Users/fhernandez/Documents/workspaceJavaCodeGeeks/ ↵
      maven jar plugin/target/classes because it is not a jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.113 s
[INFO] Finished at: 2015-12-12T20:54:25+01:00
[INFO] Final Memory: 11M/245M
[INFO] -----
```

After the execution, you will have a jar artifact generated under `target` folder and inside of it under the `META-INF` folder you will find a file called `MANIFEST.MF` with the following content

custom MANIFEST file:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
purpose: example from javacodegeeks author
Built-By: fhernandez
codification: UTF-8
javacodegeeks: maven jar plugin example
Created-By: Apache Maven 3.3.3
Build-Jdk: 1.8.0_65
```

and a `INDEX.LIST` file with the following content

index LIST file:

```
JarIndex-Version: 1.0

maven-jar-plugin-example-1.0.0-SNAPSHOT.jar
META-INF
META-INF/maven
META-INF/maven/com.javacodegeeks.examples
META-INF/maven/com.javacodegeeks.examples/maven-jar-plugin-example
com
com/javacodegeeks
```

11.5 Include/Exclude files

If you need to include or exclude some classes in your jar you can use the maven jar plugin to achieve it, as you can see below the following `pom.xml` excludes the content of all `client` folders inside of class folders.

`pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-jar-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven jar :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
```

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <excludes>
          <exclude>**/client/**</exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

You can run the plugin with the `mvn package` command, you will see an output result like this

output:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven jar :: example 1.0.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-jar-plugin- ←
example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ maven-jar-plugin- ←
example ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven-jar- ←
plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ maven-jar-plugin- ←
example ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-jar-plugin-example ---
[INFO] Surefire report directory: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/surefire-reports

T E S T S
```

```
Running com.javacodegeeks.test.MyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.06 sec

Results :

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

[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ maven-jar-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven jar plugin/ ↵
target/maven-jar-plugin-example-1.0.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.173 s
[INFO] Finished at: 2015-12-12T21:14:44+01:00
[INFO] Final Memory: 18M/229M
[INFO] -----
```

After the execution, you will notice that there is not any class inside the `com.javacodegeeks.client` folder.

11.6 Additional jars

You can generate an additional jar artifact besides the main jar artifact with the maven jar plugin. You have to define the phase and the goal in order to not override the main jar artifact generation, also the `classifier` tag is important for this. You can see below a `pom.xml` using this

`pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-jar-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven jar :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <executions>
```

```

        <execution>
            <phase>package</phase>
            <goals>
                <goal>jar</goal>
            </goals>
            <configuration>
                <classifier>client</classifier>
                <includes>
                    <include>**/client/*</include>
                </includes>
            </configuration>
        </execution>
    </executions>
</plugin>
</plugins>
</build>
</project>

```

You can run the plugin with the `mvn package` command, you will see an output result like this

output:

```

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven jar :: example 1.0.0-SNAPSHOT
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-jar-plugin- ←
example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ maven-jar-plugin-example ←
---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven-jar- ←
plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ maven-jar-plugin- ←
example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-jar-plugin-example ---
[INFO] Surefire report directory: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/surefire-reports

T E S T S

Running com.javacodegeeks.test.MyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.059 sec

Results :

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

[INFO]

```

```
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ maven-jar-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven jar plugin/ ↵
target/maven-jar-plugin-example-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default) @ maven-jar-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven jar plugin/ ↵
target/maven-jar-plugin-example-1.0.0-SNAPSHOT-client.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.734 s
[INFO] Finished at: 2015-12-13T12:20:57+01:00
[INFO] Final Memory: 10M/245M
[INFO] -----
```

After the execution, you will have two jars files under `target` folder as you can see in the output above.

11.7 Generate a jar artifact with test classes

The maven jar plugin allow us to generate a jar file with the test classes. You can see below a `pom.xml` using this `pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↵
    maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-jar-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven jar :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <executions>
          <execution>
            <goals>
              <goal>test-jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```



```
        </plugins>
    </build>
</project>
```

You can run the plugin with the `mvn package` command, you will see an output result like this

output:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven jar :: example 1.0.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-jar-plugin- ←
example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ maven-jar-plugin-example ←
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven-jar- ←
plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ maven-jar-plugin- ←
example ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-jar-plugin-example ---
[INFO] Surefire report directory: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven ←
jar plugin/target/surefire-reports

T E S T S

Running com.javacodegeeks.test.MyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.057 sec

Results :

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

[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ maven-jar-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven jar plugin/ ←
target/maven-jar-plugin-example-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:test-jar (default) @ maven-jar-plugin-example ---
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven jar plugin/ ←
target/maven-jar-plugin-example-1.0.0-SNAPSHOT-tests.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.109 s
```

```
[INFO] Finished at: 2015-12-13T12:29:47+01:00
[INFO] Final Memory: 18M/226M
[INFO] -----
```

After the execution, you will have two jars files under `target` folder as you can see in the output above. One of the jars artifact is the main class and the other one is a jar artifact with your test classes.

11.8 Conclusions

As you have seen with this example, the maven jar plugin allow you to do several things in order to fit your deploy requirements.

11.9 Download the eclipse project

Download

You can download the full source code of this example here: [maven-jar-plugin-example.zip](#)

Chapter 12

Maven assembly plugin example

In this example we are going to see how we can use the assembly maven plugin in order to control how maven generates our output packages.

Maven is a build automation tool used mainly for java projects from apache.

You can access to the maven assembly plugin information [here](#).

We are going to use the assembly plugin in order to generate different packages for different usages.

For this example we use the following technologies:

- MAC OSX
- Eclipse Luna
- Maven3
- JDK 1.8.0_65 64bits
- Maven assembly plugin 2.6

12.1 Introduction

For this example we are going to show how we can generate several packages in order to use it in different environments: for different servers, distribute source code, organize files for different audiences, etc. . .

In this kind of situations the maven assembly plugin will help us to achieve it.

12.2 Example project

For this example, we are going to use a java project with maven nature that will be packaged as a jar file. Eclipse Luna comes with maven support out of the box, so you don't have to install anything. Our project will look like this:

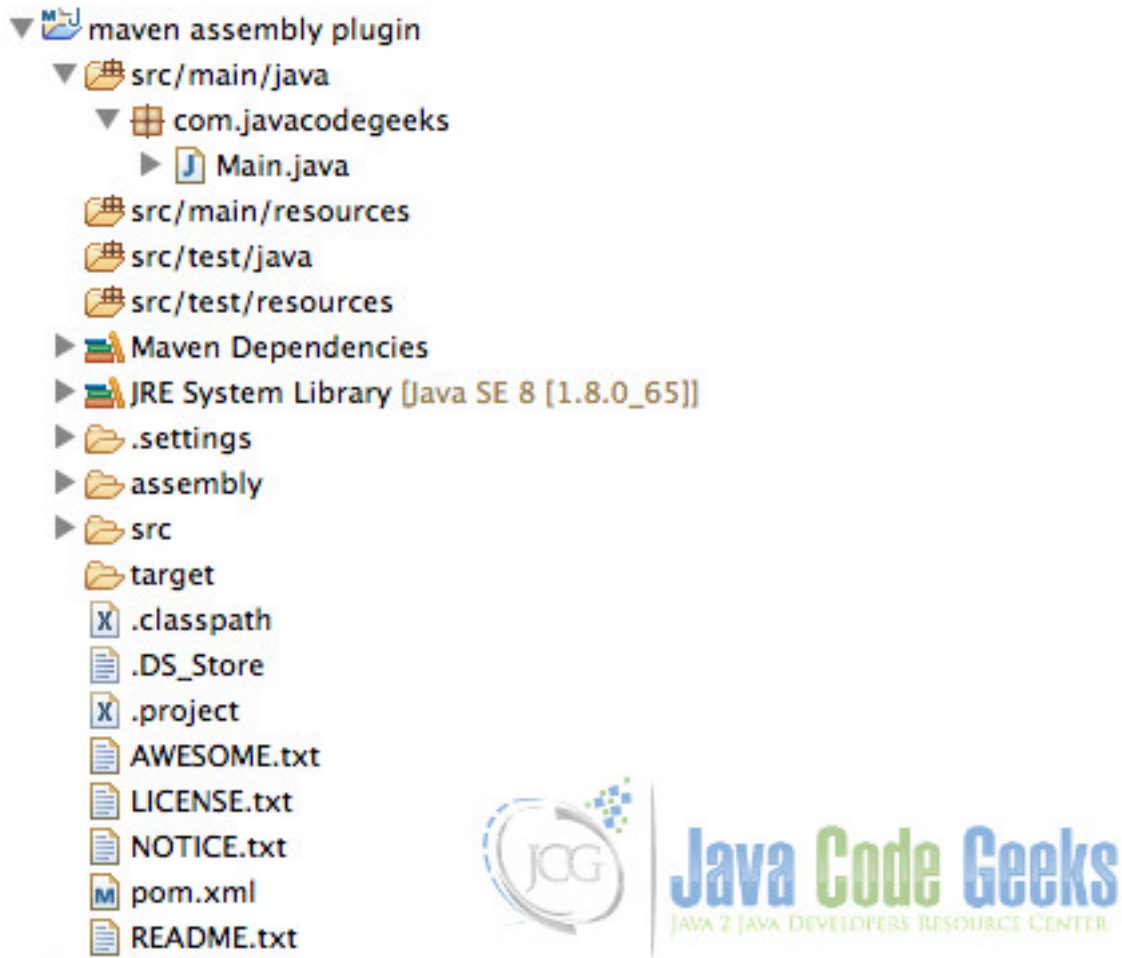


Figure 12.1: Initial project, ready for work

At this point, we have an empty maven project. Note that the project has several text files like `NOTICE.txt`, `LICENSE.txt`, `NOTICE.txt` and `README.txt`. We are going to change the way those files are stored in the output packages with the assembly plugin help.

The `pom.xml` has some dependencies in order to show how we can decide if those dependencies will be in the output package, or not, or only a part of them.

The `pom.xml` will look like this:

`pom.xml`:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/
  maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.examples</groupId>
  <artifactId>maven-assembly-plugin-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven assembly :: example</name>
  <url>https://maven.apache.org</url>

  <properties>
    <spring.version>4.2.2.RELEASE</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
```

```

<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.4</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</ ←
            descriptorRef>
          <descriptorRef>bin</descriptorRef>
          <descriptorRef>src</descriptorRef>
          <descriptorRef>project</descriptorRef>
        </descriptorRefs>
        <descriptors>
          <descriptor>assembly/ourAssembly.xml</ ←
            descriptor>
        </descriptors>
      </configuration>
      <executions>
        <execution>
          <id>trigger-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>

```

Note that the project includes `spring` framework and `log4j` logger framework, also defines the assembly plugin in lines 35 to 59 with several descriptor references and a custom descriptor too. The next sections will show how all those descriptors work.

12.3 Assembly plugin predefined descriptors

The assembly maven plugin comes out of the box with some predefined descriptors, Let's see it:

- `jar-with-dependencies` → Allow us to generate a jar package with all the dependencies defined in `pom.xml` file inside of it. This is useful when we plan to deliver an auto-executable jar.
- `bin` → Use this predefined descriptor in order to create a binary distribution of your package.
- `src` → Use this predefined descriptor in order to distribute your source code. The output package will have the `src` folder content inside it.
- `project` → (since 2.2) Use this predefined descriptor in order to distribute your entire project minus the `target` folder content

You can see more details about those predefined descriptors [here](#).

The example project use all the predefined descriptors in order to show how they work. We can see the result output in later sections.

12.4 Assembly plugin custom descriptors

The assembly plugin allow us to create a custom assembly in which we can define how our package will be.

The example project references a custom descriptor in `assembly/ourAssembly.xml` at line 47. So you have to create a folder called `assembly` in project root folder, and create inside it a file called `ourAssembly.xml`. The content of the new file will be the following:

`ourAssembly.xml`:

```
<assembly
  xmlns="https://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/plugins/maven-assembly-plugin/assembly ←
    /1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>ourAssembly</id>
  <formats>
    <format>jar</format>
  </formats>
  <dependencySets>
    <dependencySet>
      <excludes>
        <exclude>commons-logging:commons-logging</exclude>
        <exclude>log4j:log4j</exclude>
      </excludes>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>${basedir}</directory>
      <includes>
        <include>*.txt</include>
      </includes>
      <excludes>
        <exclude>AWESOME.txt</exclude>
        <exclude>LICENSE.txt</exclude>
      </excludes>
    </fileSet>
  </fileSets>
  <files>
    <file>
      <source>AWESOME.txt</source>
      <outputDirectory>/MyFiles</outputDirectory>
      <filtered>>true</filtered>
    </file>
    <file>
```

```

                <source>LICENSE.txt</source>
                <outputDirectory>/License</outputDirectory>
                <filtered>true</filtered>
            </file>
        </files>
    </assembly>

```

Our assembly uses a `dependencySets` tag in order to exclude the following dependencies: `commons-logging:commons-logging` and `log4j:log4j`. The `commons-logging` was not defined as a dependency in our `pom.xml`, but it is indirectly referenced by `spring`.

Our assembly uses a `fileSets` tag in order to include some files (all `txt` files) and exclude two of them: `AWESOME.txt` and `LICENSE.txt`. We can use this tag in order to include files that conform to predefined patterns.

Our assembly uses a `files` tag in order to include some files in custom folders. In this case the assembly includes `AWESOME.txt` file inside of `MyFiles` folder and `LICENSE.txt` file inside of `License` folder. We can use this tag in order to include some concrete files in concrete locations.

For more details about how we can define custom assemblies you can go to [here](#).

12.5 Running the assembly plugin

The assembly plugin only has one non-deprecated target called `single`. The `pom.xml` file has defined the assembly plugin to run in package phase, so we can run it with `mvn package`.

Output:

```

[INFO] Scanning for projects...
[INFO]
[INFO] Using the builder org.apache.maven.lifecycle.internal.builder.singlethreaded. ←
    SingleThreadedBuilder with a thread count of 1
[INFO]
[INFO] -----
[INFO] Building Maven assembly :: example 1.0.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maven-assembly-plugin ←
    -example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ maven-assembly-plugin- ←
    example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maven- ←
    assembly-plugin-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ maven-assembly- ←
    plugin-example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven-assembly-plugin-example ←
    ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ maven-assembly-plugin-example ---
[INFO]
[INFO] --- maven-assembly-plugin:2.6:single (trigger-assembly) @ maven-assembly-plugin- ←
    example ---

```

```
[INFO] Reading assembly descriptor: assembly/ourAssembly.xml
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /MyFiles
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /License
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-ourAssembly.jar
[INFO] Building jar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-jar-with-dependencies.jar
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[INFO] Building tar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-bin.tar.gz
[WARNING] Entry: maven-assembly-plugin-example-1.0.0-SNAPSHOT/maven-assembly-plugin-example ←
-1.0.0-SNAPSHOT-jar-with-dependencies.jar longer than 100 characters.
[WARNING] Resulting tar file can only be processed successfully by GNU compatible tar ←
commands
[WARNING] Entry: maven-assembly-plugin-example-1.0.0-SNAPSHOT/maven-assembly-plugin-example ←
-1.0.0-SNAPSHOT-ourAssembly.jar longer than 100 characters.
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[INFO] Building tar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-bin.tar.bz2
[WARNING] Entry: maven-assembly-plugin-example-1.0.0-SNAPSHOT/maven-assembly-plugin-example ←
-1.0.0-SNAPSHOT-jar-with-dependencies.jar longer than 100 characters.
[WARNING] Resulting tar file can only be processed successfully by GNU compatible tar ←
commands
[WARNING] Entry: maven-assembly-plugin-example-1.0.0-SNAPSHOT/maven-assembly-plugin-example ←
-1.0.0-SNAPSHOT-ourAssembly.jar longer than 100 characters.
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[INFO] Building zip: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-bin.zip
[INFO] Building tar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-src.tar.gz
[INFO] Building tar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-src.tar.bz2
[INFO] Building zip: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-src.zip
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[INFO] Building tar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-project.tar.gz
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[INFO] Building tar: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-project.tar.bz2
[WARNING] The assembly descriptor contains a filesystem-root relative reference, which is ←
not cross platform compatible /
[INFO] Building zip: /Users/fhernandez/Documents/workspaceJavaCodeGeeks/maven assembly ←
plugin/target/maven-assembly-plugin-example-1.0.0-SNAPSHOT-project.zip
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.104 s
[INFO] Finished at: 2015-11-23T11:16:00+01:00
```



```
[INFO] Final Memory: 17M/309M  
[INFO] -----
```

12.6 See the result

After run the maven command, we can see the different packages generated by the plugin like this:

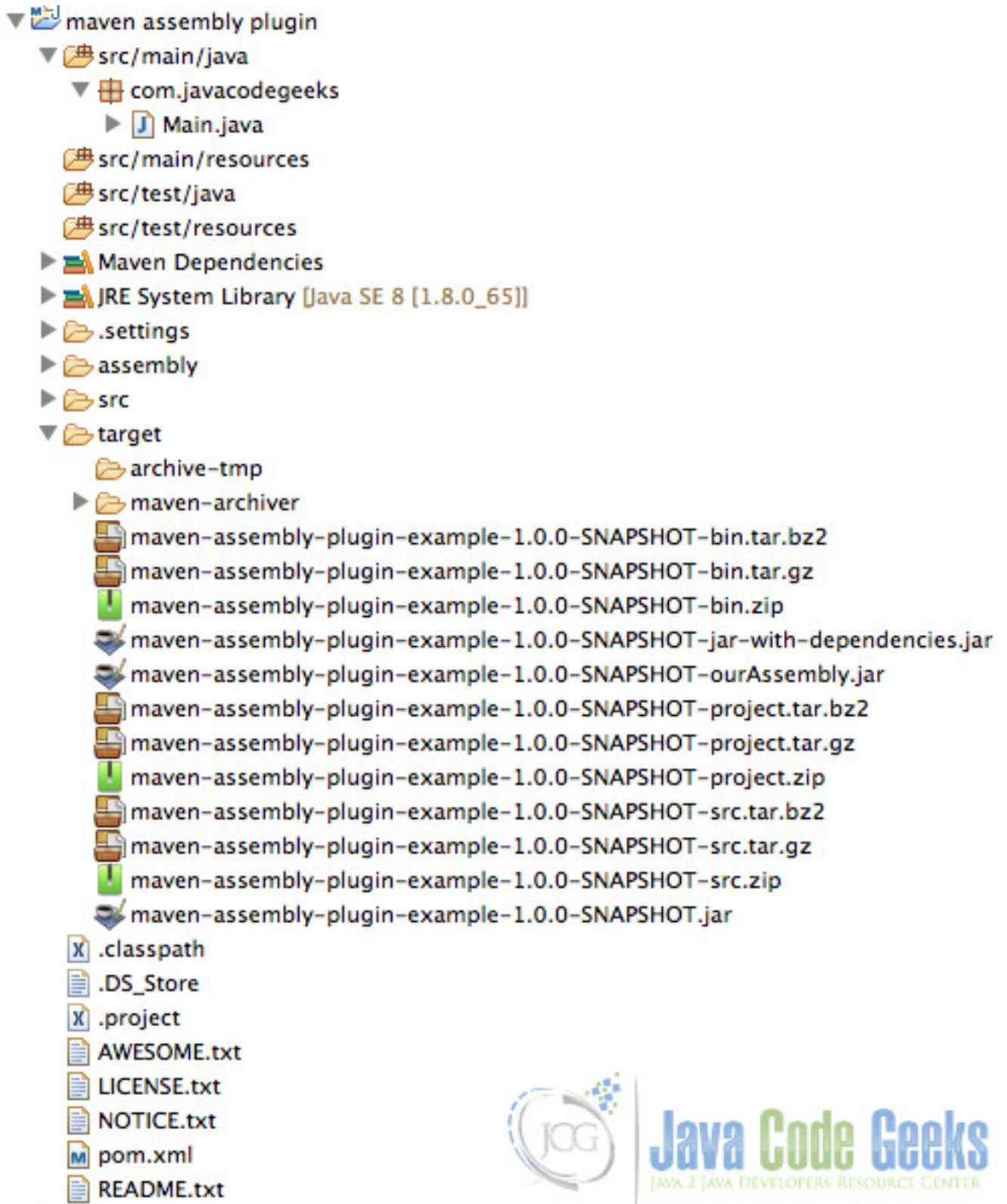


Figure 12.2: Different packages generated by the assembly plugin

For each different descriptor we have a different package, we can relate each package with the descriptor that generates it because the descriptor name is added to the end of the file name.



For example, we can see the `src` result package here:



Figure 12.3: Jar generated by the `src` descriptor content

We can see our classes inside it.

We can also see the `ourAssembly` package result here

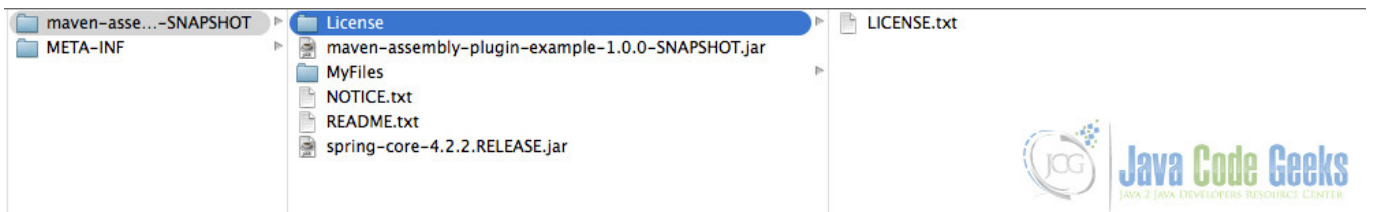


Figure 12.4: Jar generated by `ourAssembly` content

We can see how the files are distributed how we want, and how there are no jars for `log4j` and `common-logging` inside it as we specified in `ourAssembly.xml` file.

12.7 Download the eclipse project

Download

You can download the full source code of this example here: [maven-assembly-plugin-example.zip](#)