The Maven Practical Guide

Karl-Heinz Marbaise, 2021-01-10

Version 0.1.0-SNAPSHOT

Table of Contents

THIS IS WORK IN PROGRESS

This is work in progress.

If you have any suggestions, improvements or issues please use the following issue tracking system:

https://github.com/khmarbaise/the-maven-practical-guide/issues

Books are normally used to generate DocBook markup and the titles of the preface, appendix, bibliography, glossary and index sections are significant ('specialsections').

Example Preface

Optional preface.

Preface Sub-section

Preface sub-section body.

Chapter 1. The First Chapter

Chapters can contain sub-sections nested up to three deep. [1]

Chapters can have their own bibliography, glossary and index.

And now for something completely different: monkeys, lions and tigers (Bengal and Siberian) using the alternative syntax index entries. Note that multi-entry terms generate separate index entries.

Here are a couple of image examples: an [apache maven project] example inline image followed by an example block image:

[Maven Logo] | *images/maventxt_logo_200.png*

Figure 1. An Image with a Caption.

Followed by an example table:

Table 1. An example table

Option	Description
-a 'USER GROUP'	Add 'USER' to 'GROUP'.
-R 'GROUP'	Disables access to 'GROUP'.

Example 1. An example example

> Lorum ipum...

1.1. Sub-section with Anchor

Sub-section at level 2.

1.1.1. Chapter Sub-section

Sub-section at level 3.

Chapter Sub-section

Sub-section at level 4.

This is the maximum sub-section depth supported by the distributed AsciiDoc configuration. [2]

[1] An example footnote.

[2] A second example footnote.

Chapter 2. The Second Chapter

An example link to anchor at start of the first sub-section.

An example link to a bibliography entry [taoup].

Chapter 3. Dependencies

We should make some examples how dependencies are being solved (dependency puzzler).

The Basics

Let us start with a minimal `pom` file.

```
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.soebes.tmpg.examples.basics</groupId>
    <artifactId>basics-aggregator</artifactId>
    <version>0.1.0-SNAPSHOT</version>
  </parent>

  <groupId>com.soebes.tmpg.examples.basics</groupId>          ①
  <artifactId>simplest-pom</artifactId>                      ②
  <version>0.1.0-SNAPSHOT</version>                          ③

  <name>TMPG :: Examples :: Simplest POM</name>

</project>
```

① The groupId

② The artifactId

③ The version

Chapter 4. The build life cycle

Chapter 5. Profiles in Maven

Why and how to use Profiles.

Typical scenarios where to use profiles.

CI build (jenkins) etc.

Don’t use profiles to activate/deactive modules

Situations where you shouldn’t use profiles.

Different Environments

Chapter 6. Overview

In the wild a usual problem occurs having configurations for different environments like development, test, q&a and production. The differences between those environments are most likely things like username, password for an database connection or may be other things.

I have to admit that the example with the database connection is not the best, cause this would imply having such critical information within your application which you never should do in real life. This is chosen only as an example for information which are definitively different from environment to environment.

Let us make a more realistic example out of this to get more relationship to the real world. So we create an examples which consists of several files which are different from environment to environment.

get the whole lecture of GearConf2013

How to build for prod, dev, qa environment etc.

http://blog.soebes.de/blog/2011/07/29/maven-configuration-for-multipe-environments/

http://blog.soebes.de/blog/2011/08/11/maven-configuration-for-multipe-environments-ii/

6.1. The Solutions

6.2. The Obvious Solution

Many people using Maven would suggest to use profiles for such purposes. So you have different profiles which define the filtered values for the appropriate environments and you will build the appropriate artifacts.

This will result in calling Maven with the following commands to produce artifacts for the different environments.

```
mvn -Pdevelopment clean package
mvn -Ptest clean package
mvn -Pqa clean package
mvn -Pproduction clean package
```

But unfortunately this approach has one big drawback. How would you call Maven if you need the artifacts for development, test, q&a and production? So your answer might look like this?

```
mvn -Pdevelopment,test,qa,production clean package
```

The disadvantage of this approach is that you have to give all these parameters every time you call Maven maybe in several permutations depending for which environment you would like to build.

What does in practice happen? You simple forget them. Have you remembered to change the configuration of your CI solution? Have you informed all your team mates? I bet you missed something.

So solution should work by simply calling Maven like this:

```
mvn clean package
```

So in conclusion this approach is not ideal.

Picture of the application ?

What are the drawbacks of such a solution?

6.3. The next

Multi Module Builds

Chapter 7. Overview

Sometimes it is enough having a single pom file and a limited number of java classes which are combined into a single jar which is produced to get a full fledged project. But this is only in a limited number of cases a solution for all kind of project types. If we take a look at JEE projects which often contains several kinds of artifacts which will be combined into a final single artifact usually an EAR file than the single module setup is not really a good idea. For this kind of purposes it's verify useful having a so called multi module build which combines the different artifacts under a single hood.

The idea of a multi-module-build is having multiple maven modules which are released under the same version which are directly related.

Let us start with a simple example project which contains several modules like the following:

```
module-core
module-client
module-server
```

Before we know about multi-module-builds you should have created three separated maven projects and had to define dependencies between those modules and work on them without any relationship. Obviously you can imagine that the above modules have relationship to each other, cause the `module-client` module has a dependency to the `module-core` whereas the `module-server` has a dependency to `module-core` and so on.

Wouldn't it be the best if all the above module live within a single location (git repository or SVN trunk for example) where you could simply checkout those modules and work with your IDE on the whole project, cause if you need to change something in your `module-core` it's very likely that you need to change the depending module `module-client` as well? Exactly for such purposes a multi-module-build exists in Maven.

7.1. Structure

Based on the idea of the relationship of the modules it is necessary to create an appropriate folder structure which more or less shows the relationship of the module to each other.

```
+--- pom.xml
  +--- module-core
  +--- module-client
  +--- module-server
```

There are some parts which you need to pay attention to, to get a good working environment. The first thing is the pom on the root level of this structure. This pom file contains no code nor does it produce an artifact as the usual maven project. This means in other words it does not produce an `jar`-File. This is the reason why this Maven project defines it's packaging as `pom`.

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

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.soebes.smpp</groupId>
    <artifactId>smpp</artifactId>
    <version>2.1.0</version>
  </parent>

  <groupId>com.soebes.mpg.examples.mmb</groupId>
  <artifactId>parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  ..
  <modules>
    <module>module-core</module>
    <module>module-client</module>
    <module>module-server</module>
  </modules>
  ..
</project>
```

Apart from the above you need to define the list of modules which you would like include in this parent. It is best practice to name the folders as their appropriate `artifactId`. So now let us take a look at a module how its pom file looks like:

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

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.soebes.mpg.examples.mmb</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>

  <artifactId>module-core</artifactId>
...
</project>
```

See here

What is the basic idea of a multi module build? Same version? Same time of releasing them.

- Multiple Modules
- mvn install (first why?)
- Unit Tests (mvn test)
- Integration Test (mvn integration-test)
- packaging
- use of an module from a reactor build in other projects?

Pro's and Cons' http://stackoverflow.com/questions/23584429/releasing-a-modular-maven-project

Jenkins support for separated maven projects to be released: https://wiki.jenkins-ci.org/display/JENKINS/Maven+Cascade+Release+Plugin

Aggregator ? Difference.

http://stackoverflow.com/questions/23936339/maven-parent-project-structure

What if only a single modules code has been changed? Can i release only a single module from the multi module build? Draw backs?

Testing with Maven

Chapter 8. Overview

Basic Unit Testing, Responsibilities of Plugins, Using JUnit, TestNG, multi module builds and unit tests.

This chapter will give some practical hints how you can use unit- and integration tests in relationship with Maven. Furthermore it will give you tipps how to prevent several issues with testing.

Think about some examples about the following: http://stackoverflow.com/questions/23588707/maven-layout-how-to-be-sure-that-src-main-does-not-depend-on-src-test

http://stackoverflow.com/questions/23659829/maven-run-class-before-test-phase-exec-maven-plugin-execjava-not-executing-cla

Information: http://labs.carrotsearch.com/randomizedtesting.html http://stackoverflow.com/questions/8295100/how-to-re-run-failed-junit-tests-immediately

https://blog.42.nl/articles/keeping-integration-tests-isolated/

8.1. Unit Testing

Unit testing can be done out-of-box in Maven which means you just have to locate your unit tests into 'src/test/java' and follow the naming conventions. The resulting and recommended folder structure will be show in the followin example.

```
.
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- soebes
    |               `-- training
    |                   `-- maven
    |                       `-- simple
    |                           `-- BitMask.java
    `-- test
        `-- java
            `-- com
                `-- soebes
                    `-- training
                        `-- maven
                            `-- simple
                                `-- BitMaskTest.java
```

The folder 'src/main/java' plus an approriate package structure will contain your production code whereas the 'src/test/java' plus the package structure will contains your unit test area.

Table 2. Naming Schema for Unit Tests

Name Pattern	Example
Test*.java	TestBitMask.java
*Test.java	BitMaskTest.java
*TestCase.java	BitMaskTestCase.java

The following simple example will show how a basic unit test can look like.

```
package com.soebes.training.maven.simple;

import static junit.framework.Assert.assertEquals;

import org.junit.Test;

public class BitMaskTest {

    @Test
    public void checkFirstBitTest() {
        BitMask bm = new BitMask(0x8000000000000000L);
        assertEquals(true, bm.isBitSet(63));
    }

    @Test
    public void checkNumberBitTest() {
        for (int bitNumber = 0; bitNumber < 64; bitNumber++) {
            long bitMask = Long.rotateLeft(1, bitNumber);
            BitMask bm = new BitMask(bitMask);
            assertEquals(true, bm.isBitSet(bitNumber));
        }
    }

    @Test
    public void setBitNumberTest() {
        BitMask bm = new BitMask();
        for (int bitNumber = 0; bitNumber < 64; bitNumber++) {
            bm.setBit(bitNumber);
            assertEquals(true, bm.isBitSet(bitNumber));
        }
    }

    @Test
    public void unsetBitNumberTest() {
        BitMask bm = new BitMask();
        for (int bitNumber = 0; bitNumber < 64; bitNumber++) {
            bm.setBit(bitNumber);
        }
        for (int bitNumber = 0; bitNumber < 64; bitNumber++) {
            bm.unsetBit(bitNumber);
```

```
            assertEquals(false, bm.isBitSet(bitNumber));
        }
    }

    @Test
    public void adhocBitTest() {
        BitMask bm = new BitMask(0xffffffffffffffffL);
        bm.unsetBit(10);
        bm.unsetBit(20);
        bm.unsetBit(30);
        bm.unsetBit(40);
        bm.unsetBit(50);
        bm.unsetBit(60);

        assertEquals(false, bm.isBitSet(10));
        assertEquals(false, bm.isBitSet(20));
        assertEquals(false, bm.isBitSet(30));
        assertEquals(false, bm.isBitSet(40));
        assertEquals(false, bm.isBitSet(50));
        assertEquals(false, bm.isBitSet(60));
    }
}
```

So if you follow the conventions in Maven and put your tests into the appropriate locations 'src/test/java' those tests will automatically be picked up and executed as unit tests. The plugin which is responsible for execution of those unit tests is the Maven Surefire Plugin.

Make an example output here....

An important thing to think of is sometimes which test framework you would like to use? There are things like JUnit, TestNG, Spock and of course many other opportunities.

In the case you would like to use JUnit within your unit tests you just simply add the appropriate dependency to your 'pom.xml' and that's it.

```
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.soebes.tmpg.examples.testing</groupId>
    <artifactId>tmpg-examples-aggregator</artifactId>
    <version>0.1.0-SNAPSHOT</version>
  </parent>

  <groupId>com.soebes.tmpg.examples.testing.ut-example</groupId>
  <artifactId>unit-test-example</artifactId>
  <version>0.1.0-SNAPSHOT</version>

  <name>TMPG :: Testing :: Unit Test Example</name>

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

</project>
```

If you prefer TestNG to use for your unit tests you can simply add the dependency for TestNG and your unit tests can be run as well without any supplemental change except the changes based on the differences between JUnit and TestNG itself.

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

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.soebes.tpmg.examples.testing</groupId>
    <artifactId>tpmg-examples-aggregator</artifactId>
    <version>0.1.0-SNAPSHOT</version>
  </parent>

  <groupId>com.soebes.tpmg.examples.testing.ut-example</groupId>
  <artifactId>unit-test-example-testng</artifactId>
  <version>0.1.0-SNAPSHOT</version>

  <name>TMPG :: Testing :: Unit Test Example (TestNG)</name>

  <dependencies>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>6.8.7</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

Location of unit tests

'src/test/java' is the correct location for unit tests.

Packaging of unit tests execution of unit tests Support of testing frameworks JUnit, TestNG, Spock?, BDD ?

8.2. Integration Testing

What is integration tests? How to use? Naming convention? In which cases should be used a separate module?

Test an web application with Selenium? (Examples).

8.2.1. Importance of Separation

Why is it important to separate between unit tests and integration tests?

If look into the formal definition of unit tests you will read things like independent from any resource etc. So you can by definition parallelize unit tests in general. If you don't have real unit tests you can't go that simple path to improve your build time.

In controdiction integration tests are not independent and could not be parallized by default. Under special circumstances you might change cause you know your code and of course your tests. This means to parallelize integration tests is always a task which should be done separately.

http://tempusfugitlibrary.org/documentation/ http://labs.carrotsearch.com/randomizedtesting.html

http://zeroturnaround.com/rebellabs/the-correct-way-to-use-integration-tests-in-your-build-process/

Maven Assemblies

Chapter 9. Overview

During the usage you will often be faced with the situation to create a kind of a distribution archive for example 'dist.zip' or 'dist.tar.gz' or other kind of archive flavors. Things which also happen are to create a so called 'ueber' jar which you can use to call your java application from the command line (There are other opportunities as well see Chapter…). Furthermore you often have the requirement to create archives with different configurations for different environments this also achievable.

This chapter will give a wide overview of the possibilities how you can create the different flavors of archives which you need to fullfil the requirements of your builds furthermore we will take a look what kind of mistakes you can make and how to prevent them.

Chapter 10. The Maven Assembly Plugin

The Maven Assembly Plugin is especially created for such purposes to create any kind of archive type.

10.1. Single Executable Artifact

One of the requirements you will often be confronted with is to create an archive which can simply be executed on command line. This is often called an 'ueber-jar' or 'fat-jar' (or Maven tongue: jar-with-dependencies). This can simply be accomplished by using Maven Assembly Plugin's pre-defined descriptors.

The following pom.xml example will give you an impression how the configuration for Maven Assembly Plugin needs to look like to get a `jar-with-dependencies`.

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

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.soebes.tmpg.examples.assemblies</groupId>
    <artifactId>tmpg-assemblies-aggregator</artifactId>
    <version>0.1.0-SNAPSHOT</version>
  </parent>

  <artifactId>assembly-jar-with-dependencies</artifactId>
  <name>TMPG :: Assemblies :: JAR With Dependencies</name>
  <dependencies>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>6.8.8</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <executions>
          <execution>
            <id>make-jar-with-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
              </descriptorRefs>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

By using the above pom you will get a jar which contains all the dependencies you have defined in your pom file as part of the resulting jar which is named by using a classifier `jar-with-dependencies`

to make it distinguishable from the other artifacts. The other aspect of this example project is that you can see how simple it is to create such an artifact. One thing which should be mentioned about the Maven Assembly Plugin. It is not bound to any Build Life Cycle Phase by default which means you need to bind it to the life-cylce explicitly if you like to use it.

A note about the given example. In real life you should find this example suspicious, cause it will use a typical test dependency (TestNG) without the scope `test` which is usually wrong. In this case it is only use for example purposes.

The created `jar-with-dependencies` can simply be used by the following:

```
java -jar target/assembly-jar-with-dependencies-0.1.0-SNAPSHOTS-jar-with-
dependency.jar
```

10.2. Creating an ZIP Archive

The idea of an archive is have particular content which is defined by the project in the way you like to do that and define how it will look like within the archive. This means having a folder structure within the archive or not.

10.3. Default Assemblies

- Default assemblies
- Archive types
- Component Descriptors
- Distribution archive
- empty folders etc.

Typical scenarios which occur often.

Create a jar with all dependencies which might be used to call simply java program as a command line tool. Alternatives Maven Shade Plugin.

You would like to create an distribution archive which contains the resulting components of your application. Examples.

You would like to create an archive with all sources of your project as well as the sources of all the used dependencies:

```
mvn dependency:sources
mvn dependency:copy-dependencies -Dclassifier=sources -DoutputDirectory=target/sources
```

Add here all the examples from my example-assemblies

Special requirements which you can fulfil with maven-assembly-plugin https://stackoverflow.com/

questions/24311053/how-to-get-the-content-of-a-directory-inside-of-war-that-is-inside-of-an-ear-tha

10.4. Predefined Descriptors

The predefined descriptors fulfil the need of often requests archive types which should be usable in a very short time. You could use them in a very simple form.

10.5. Module Sets

10.6. Dependency Sets

10.7. Sources

10.8. Predefined Descriptors

Currently there are four of them:

1. bin
2. jar-with-dependencies
3. src
4. project

Plugins

Chapter 11. Overview

The plugins of Maven building the foundation of the Maven eco system, cause if you download Maven itself it's a relative small archive (less than 8 MiB) and Maven itself is more or less only a IoC container which supports the life-cycle and other small things. If you like to compile for example your source code this is provided by the Maven Compiler Plugin with the appropriate functionality.

This chapter will give you an overview of the different areas of plugins sources, the different ideas of the plugins and their typical usage within a Java project build.

11.1. The Plugin Sources

In general there are two big sources of plugin. The first source is the area under the umbrella of the Apache Software Foundation. I will call them the **Core Maven Plugins**. The reason for this is that you will find plugins like Maven Compiler Plugin, Maven Jar Plugin etc. in that area which provide the most basic functionality for your build.

The second source is the MojoHaus area which also provides a large number of maven plugins.

Other sources of PLugins (JBoss, Tomcat, Antlr, google code (Maven Processor Plugin)

jaxws-maven-plugin ? (URL?)

Describe more sources and other plugins

Groovy Plugins

Chapter 12. The Different Plugins

In this chapter we will take a look on more or less every plugin which is a participant of a usual Java build or to be more accurate a participant of the {link-build-life-cycle}.

The lifecycle contains already bindings for usual plugins so in the majority of the cases its enough to build usual projects.

12.1. Clean Everything

If you want to be sure your build will start from scratch you need to wipe out everything which has been created by previous operations or the build itself. So the Maven Clean Plugin is your friend which will delete the 'target' folder of your project or in every module in case of a multi-module build. This can be simply achieved by calling maven like this:

```
mvn clean
```

Usualy you won't ever think about the Maven Clean Plugin, cause by default it's bound to the 'clean' Build Life Cycle Phase and there is no reason to change the configuration of the Maven Clean Plugin or something similar. In rare situations it could happen that you need to change the configuration and add supplemental folders or files which should be deleted during a 'mvn clean' call.

HINT: Something about the clean life cylce of the maven super pom!!

In the clean life cycle the following phases exist: pre-clean, clean, post-clean.

12.2. Resources

Often it occurs that your java code needs some kind of configuration files. One of the most famous examples for this kind of configurations is one of those numerous logging frameworks like log4j, logback, log4j2 etc. So the question is where to locate such configuration files? The Default Folder Layout gives you the hint to put such things into 'src/main/resources' which is of course intended for the production code (in other words which is packaged later into the jar file). Furthermore it is often the case as well as having different configuration files for your unit tests cause you would like having a different logging level in your unit test so you need a different set of files which should be located into 'src/test/resources'. This means in other words those files will not be packaged into the resulting jar file.

TODO: Move the following to test phase

HINT about super pom !

So usually you can simply put your appropriate configuration files into 'src/test/resources' or 'src/main/resources' and they will automatically be copied into 'target/classes' or 'target/test-classes'. But why are they copied ? The most important point about this is that you can use such resources by the usual java resources way like this:

code example (getClass().getResourcesAsStream("/log4j.properties"); This works for unit tests and for your production code! (good example?)

One important thing to mention is that the order on your class path is that the resources from your test resources coming first before your production code which means you can give a different configurtation file for every file which you already use in your production code and so you can change the behaviour in your unit tests to change things for example the logging level or something else.

- Maven Resources Plugin

12.3. Let The Source Be With You

You usually write Java source code (ok ok sometimes you write difference sources like Groovy, Scala, Ceylon or whatever) and of course you would like to compile such code into usable class files which can be used to run your application or to run your unit tests.

This is the purpose of the Maven Compiler Plugin which will compile your source code into class files.

The source code is located in 'src/main/java' and will be compiled into the 'target/classes' folder. Apart from that the Maven Compiler Plugin is also responsible to compile your unit/integration test code which is located in 'src/test/java' into 'target/test-classes'.

12.4. Let’s See If The Code Is Working?

After we have compiled the whole code we should run the unit tests to check our code. This is done before the code will be packaged into a jar file, cause if one of your unit tests will fail your build will fail and no packages are being built.

For this purpose the Maven Surefire Plugin is responsible to run those unit tests.

12.5. Let The Jar’s Come To Me

After the production code has been compiled into the appropriate '.class' files they will be packed into a jar file which is the base unit to be distributed. The jar will contain only the files from 'target/classes'. So if you don’t do something special your unit tests will never be packed into jar files.

The Maven Jar Plugin is bound to the 'package' build life cycle phase to create a jar file. This jar file contains only the files from the 'src/main/java' inclusive the resources from 'src/main/resources' area (Let use call it the production code area).

There exsist situations where you like to package your test code into a jar as well. This can be achieved by using the `test-jar` goal of the Maven Jar Plugin.

See examples (testing with common code).

- Maven Jar Plugin

Creating test-jars non transitive behaviour of test-jar artifact. Solution create a usual separate module.

12.6. Install The Archive

After the jar archive has been created the archive can be installed into the local repository to be consumed by other projects on the same machine. For such a purpose the Maven Install Plugin is responsible.

12.7. Distribute It To The World

To break the limits of your machine you can distribute an jar archive to a remote repository which can be used by other users. For this the Maven Deploy Plugin.

- Maven WAR Plugin
- Maven EAR Plugin
- Maven EJB Plugin
- Maven Shade Plugin
- Maven Deploy Plugin
- Maven Install Plugin

Idea and usage? Why?

- Mojo's Buildnumber Maven Plugin
- Mojo's Build Helper Maven Plugin
- Mojo's Appassembler Maven Plugin
- Mojo's Cargo Maven 2 Plugin
- Mojo's Exec Maven Plugin
- Mojo's SQL Maven Plugin
- Mojo's Templating Maven Plugin
- Mojo's Versions Maven Plugin
- ??ß

http://mojo.codehaus.org/clirr-maven-plugin/

Google Code: maven-processor-plugin http://stackoverflow.com/questions/24345920/could-i-use-java-6-annotation-processors-jsr-269-to-produce-code-for-gwt-in-ma

http://mvnplugins.fusesource.org/maven/1.4/maven-uberize-plugin/compared-to-shade.html

maven-graph-plugin

https://github.com/fusesource/mvnplugins/

(Looks intresting) http://site.kuali.org/maven/plugins/graph-maven-plugin/1.2.3/dependency-graphs.html Can add the graphs a reports to the build. Take a deeper look into it. http://site.kuali.org/maven/plugins/

Checksums http://nicoulaj.github.io/checksum-maven-plugin/

nar-maven-plugin: https://github.com/maven-nar/nar-maven-plugin

https://github.com/marceloverdijk/lesscss-maven-plugin

Take a deeper look into this http://docs.spring.io/spring-boot/docs/1.1.4.RELEASE/maven-plugin/usage.html spring-boot-maven-plugin

Very interesting plugin: http://www.javacodegeeks.com/2014/08/maven-git-release.html

12.8. Let The Force Be With You

The larger a build becomes the more you need to control what happens within your build otherwise the {link-broken-window-problem} occurs and will likely result in later problems you should prevent.

How can you force rules within in your build? Sometimes it is not enough to suggest the best practice you need to force the best practices within a build. The tool to do so is the Maven Enforcer Plugin.

One of the basic things is to force your build is built with the correct Maven version, cause htere exist some things which don't work with older Maven versions etc. The way to prevent building with the wrong Maven version was to use the `prerequsites` tag like this:

```
<prerequisites>
  <maven>2.2.1</maven>
</prerequisites>
```

but based on the improvements in Maven within Maven 3 the 'prerequsites' part in the pom has been marked as depcrecated and will not be checked. So to make sure a build will only works with a particular Maven version for example 3.1.1 you need to go the following path:

```
<project ...>

  <!-- This marked as deprecated for Maven 3.X. This is checked by maven-enforcer-
plugin -->
  <!--  https://issues.apache.org/jira/browse/MNG-4840 -->
  <!--  https://issues.apache.org/jira/browse/MNG-5297 -->
  <prerequisites>
    <maven>${maven.version}</maven>
  </prerequisites>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-enforcer-plugin</artifactId>
        <executions>
          <execution>
            <id>enforce-maven</id>
            <goals>
              <goal>enforce</goal>
            </goals>
            <configuration>
              <rules>
                <requireMavenVersion>
                  <version>${maven.version}</version>
                </requireMavenVersion>
              </rules>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ..
</project>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <executions>
    <execution>
      <id>enforce-maven</id>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <requireSameVersions>
            <plugins>
              <plugin>org.apache.maven.plugins:maven-surefire-plugin</plugin>
              <plugin>org.apache.maven.plugins:maven-failsafe-plugin</plugin>
              <plugin>org.apache.maven.plugins:maven-surefire-report-plugin</plugin>
            </plugins>
          </requireSameVersions>
        </rules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Making Releases

Chapter 13. Overview

Chapter 14. The Traditional Maven Way

maven-release-plugin etc. How it works. Pro’s and con’s.

Chapter 15. Releases The CD Way

Releases in the time of Continious Delivery. How to solve this via Maven? Is this possible? Maven 3.2.1.

http://www.youtube.com/watch?v=McTZtyb9M38

http://maven.40175.n5.nabble.com/Continuous-Delivery-and-Maven-td3245370.html

http://stackoverflow.com/questions/18456111/what-is-the-maven-way-for-project-versions-when-doing-continuous-delivery

An other kind of doing releases.. http://danielflower.github.io/2015/03/08/The-Multi-Module-Maven-Release-Plugin-for-Git.html

Continious Integration Solution

Chapter 16. More Details

Jenkins how to work in relationship with Jenkins. Which Plugins of Jenkins can be usefull

Build Smells

Chapter 17. Creating Multiple Artifacts

Multiple Artifacts The Wrong Way

Creating multiple jars from a single module? (create a jar from package a.b.c and create an other jar from packages a.b.d)?

Examples for build smells:

http://stackoverflow.com/questions/11448184/maven-jar-plugin-include-upper-dir

Chapter 18. Not Part of the Life Cycle

```
Calling mvn assembly:single or assembly:assemble ? Why not being part of the build?
and use mvn package?
```

Chapter 19. Multi Module Builds

19.1. Module Structure

- Parent of a multi-module is **not located** at root level of the structure?

19.2. The Install Hack

The Install Hack

You need to do `mvn install` in a multi-module build but `mvn clean package` will not work?

19.3. Separation of Concerns

Multiple Purposes of a Module

Use a module for only one purpose not for many. (Separation of Concern)

Chapter 20. Testing

- Not separated unit- and integration tests Configuration by using profile for unit and integration tests
- Typical indicator having TestSuite class file etc.

Chapter 21. Assemblies

- Looking on the file system instead of using the reactor Here: http://stackoverflow.com/questions/23951547/how-to-create-single-target-from-multi-module-maven-project
- Warngings in relationship with maven-assembly-plugin (dir format!)

Chapter 22. Problem with Profiles and Dependencies

http://blog.soebes.de/blog/2013/11/09/why-is-it-bad-to-activate-slash-deactive-modules-by-profiles-in-maven/

Chapter 23. What about dependencies by profiles?

Describe why and how and what the drawbacks are?

Plugin Development

Chapter 24. Overview

How to develop a plugin. Basics. Annotations? Example project.

24.1. Reasons

- Reasons to develop a plugin?
- Why is it better to write a plugin instead using scripts/external execution of Java/Groovy/Kotlin whatever code?

Often I see people developing maven plugins which are superfluous, cause the functionality is already provided by one of the existing plugins or a combination of other plugins.

So the question is: When should I start to think about creating my own plugin? The simple answer to this: If the needed functionality is not being provided by any existing plugin.

Example when to create a plugin?

24.2. Basics

What is a Mojo?

- The annotations for plugins which are needed?
- How to build a plugin?
- How to test a plugin?

24.3. Building a plugin

- What is needed to build a plugin?
- How does a Maven project look like for building a plugin?

24.4. Testing

Testing a plugin is one of the most challenging thing cause a plugin which is running inside a container (Maven Runtime) has several aspects of testing.

In general there at least three different typs of tests you usually (should) write:

1. The Java code (more or less independent) functionality you would like to put into a plugin.
    - Usually covered by unit/integration tests which you (should) already know.
2. The Mojo itself?
3. The interaction with a real project?
    - How does a plugin behave within a real project setup?

Testing frameworks / Support

- Maven Invoker Plugin
- https://github.com/khmarbaise/maven-it-extension
- http://maven.apache.org/plugins/maven-invoker-plugin/index.html

https://github.com/asciidoctor/asciidoctor-maven-plugin

24.5. Compatibility

There a different aspect of compatibility. The first one is:

- Which minimum Java version you should support?
- Minimum Maven version you will require?

Performance tipps

Chapter 25. Incremental Builds

incremental builds in Maven.

Improve the performance of your build.

http://grumpyapache.blogspot.de/2014/05/build-system-performnce-on-windows.html

Don’t use NFS neither do use NTFS…

Repository Manager

Chapter 26. Overview

This chapter will give you an overview of the idea the intention of a repository manager. It will also show and describe the advantages of the usage of a repository manager.

Chapter 27. Test

TODO:

- Why do you need a repository manager?
- Proxy to Central, XXx
- Using more external repositories than Central repository.

Best Practice

Chapter 28. Generate Into Source Folder

generating code into src folder instead of 'target'. Pro/Cons on that approach..

In Maven the convention exists to put everything which is generated, compiled etc. into the `target` folder of the appropriate module. Unfortunately in the wild you will find builds which do not follow the convention and for example generate things into `src folder which is a bad practice (Hint why?).

So let use think about this a little bit more. The first thing is if you change something in `src` folder means your version control system will be alarmed about such a change which on the other hand means you will be alarmed about a change which is not really a change, cause generated code will usually generate the same code from the same source but usually with some changed time stamp within the generated code. The consequence on the above is you must exclude some areas from your version control view to suppress such irritations.

The next thing is you need to change the configuration of your appropriate plugins, cause more or less all plugins follow that conventions (ok there exist some exceptions). This implies your configuration in your pom gets larger and of course does not follow the conventions over configuration paradigm.

Other 'solutions' which fall into the same category do something different. They generate into a folder within 'target' things like 'target/generated-code' but they usually missed that plugins usually add the generated code folder to the project sources folders automatically already. But in contradiction they explicitly add the generated folder via supplemental plugins like 'build-helper-maven-plugin' to the sources folder.

Chapter 29. Dependencies / DependencyManagement

Scope only in real dependencies not in dependencyManagement

means always use scope:test in the real project and not in dependencyManagement

Chapter 30. Deps via Props

Using properties to control the dependencies with a so called company pom?

Result: Complete build is necessary instead of using released within a build.

Chapter 31. Company wide parent(s)

Manifest setup with master parent.

```
<configuration>
  <archive>
    <addMavenDescriptor>true</addMavenDescriptor>
    <index>true</index>
    <manifest>
      <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
      <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>
    </manifest>
    <manifestEntries>
      <artifactId>${project.artifactId}</artifactId>
      <groupId>${project.groupId}</groupId>
      <version>${project.version}</version>
      <buildNumber>${buildNumber}</buildNumber>
      <scmBranch>${scmBranch}</scmBranch>
    </manifestEntries>
  </archive>
</configuration>
```

What should be defined in such a parent? http://stackoverflow.com/questions/24409889/where-should-i-keep-my-companys-parent-pom

Style Guide for POM files. SortPom (default style for pom files)

Why you should never use version ranges?

No different dependencies via profiles! Why ? The consequences?

Naming modules based on their artifactId's.

Don't do this: http://stackoverflow.com/questions/23901560/how-to-handle-different-dependencies-requirements-for-web-servers-in-pom-xml

Ideas like this: http://developer-blog.cloudbees.com/2013/03/playing-trade-offs-with-maven.html

This is a bad idea: http://stackoverflow.com/questions/24104735/embedding-dependencies-resources-in-maven

What is a good solution for such kind of questions: http://stackoverflow.com/questions/24248873/maven-package-resources-with-classes Answer: create a mod-core, mod-war and that's it?

Chapter 32. Building for different Environments

You are often face with the problem having different environments like dev, test, prod this is just a simple example how real life is.

Chapter 33. How to do good integration tests for maven plugins

One of the final tests should be to clean your local repository and start your integration tests of your plugin from scratch

rm -fr $HOME/.m2/repository mvn -Prun-its clean verify

This should work without any problem.

Chapter 34. Nexus

why the order of the repositories does really matter…

Chapter 35. Branching Strategies

http://stackoverflow.com/questions/24420474/do-you-really-need-to-version-the-trunk-of-a-maven-project

https://github.com/lewisd32/lint-maven-plugin

Example Appendix

One or more optional appendixes go here at section level 1.

Appendix Sub-section

Sub-section body.

Example Glossary

Glossaries are optional. Glossaries entries are an example of a style of AsciiDoc labeled lists.

Example Colophon

Text at the end of a book describing facts about its production.

This books has been created by using the following tools: Git, AsciiDoc and Vim for text editing.

Example Bibliography

The bibliography list is a style of AsciiDoc bulleted list.

Books

- [taoup] Eric Steven Raymond. 'The Art of Unix Programming'. Addison-Wesley. ISBN 0-13-142901-9.
- [walsh-muellner] Norman Walsh & Leonard Muellner. 'DocBook - The Definitive Guide'. O’Reilly & Associates. 1999. ISBN 1-56592-580-7.

Articles

- [abc2003] Gall Anonim. 'An article', Whatever. 2003.

Example Index