The Maven Practical Guide

Karl Heinz Marbaise

Version 0.1.0-SNAPSHOT, 2023-09-24

Table of Contents

About Karl Heinz Marbaise

FIXME: Add more about me. something more about me ;-)

I'm … the xxx.

This is the Abstract of this document.

Colophon

Preface

Project Metadata

- Version control https://github.com/khmarbaise/the-maven-practical-guide
- Bug tracker: https://github.com/khmarbaise/the-maven-practical-guide/issues

This book has been created by using the following tools: Git, AsciiDoc and IDEA IntelliJ for text editing.

THIS IS WORK IN PROGRESS

This guide is an attempt to write a practical guide about Apache Maven. Its idea is to give practical hints how to use Apache Maven and describe and show what are the best practices and why you should follow them.

If you have any suggestions, improvements or found issues please report them via the bug tracker, mentioned in metadata area.

This is also in its early stage so a large number of areas are just a containing a couple of sentences or even less. Those simply reminders to my own.

Overview

This is a more or less full documentation about Apache Maven.

Chapter 1. The Basics

Requirement: Install Maven ? Download it from?

Download Maven https://maven.apache.org/download.cgi and install https://maven.apache.org/install.html.

Overview about the basics, components of a `pom.xml` file. What is a life cycle..

Checking that the installation has worked

```
mvn --version
```

This should print out something similar like the following on a MacOS:

```
$ mvn --version
Apache Maven 3.9.3 (21122926829f1ead511c958d89bd2f672198ae9f)
Maven home: /Users/khm/tools/maven
Java version: 17.0.7, vendor: Eclipse Adoptium, runtime: xxxxx
Default locale: en_DE, platform encoding: UTF-8
OS name: "mac os x", version: "13.4.1", arch: "aarch64", family: "mac"
```

On a linux like system it would print something like this:

TODO: Real output of a debian system:

```
$ mvn --version
Apache Maven 3.9.3 (21122926829f1ead511c958d89bd2f672198ae9f)
Maven home: /Users/khm/tools/maven
Java version: 17.0.7, vendor: Eclipse Adoptium, runtime: xxxxx
Default locale: en_DE, platform encoding: UTF-8
OS name: "mac os x", version: "13.4.1", arch: "aarch64", family: "mac"
```

On a Windows system it should look similar like this:

TODO: Real output of Windows:

```
c:\> mvn --version
Apache Maven 3.9.3 (21122926829f1ead511c958d89bd2f672198ae9f)
Maven home: /Users/khm/tools/maven
Java version: 17.0.7, vendor: Eclipse Adoptium, runtime: xxxxx
Default locale: en_DE, platform encoding: UTF-8
OS name: "mac os x", version: "13.4.1", arch: "aarch64", family: "mac"
```

1.1. The Foundation

Let us start with a minimal `pom` file. It's required that you put the content into a file called `pom.xml` in a separate directory on your system:

⊕ denotes a public type, ⊖

```
  <modelVersion>4.0.0</modelVersion>

  <!-- NEED TO REMOVE THE PARENT -->
  <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

1.2. The build life cycle

The life cycle phases...

1.2.1. Directory Structure

Describe the basic directory structure.

1.3. The Coordinates

groupId, artifactId, version.

1.4. Coordinates

Structure in java project separated/structured by using packages Higher level abstraction separation via groupId coordinates

groupId/artifactId/version

repositories? picturing to repositories? Remote repositories? How to find an artifact based on it coordinates? What about search https://central.sonatype.com

1.5. Versions

Base idea of versions? Why even needed?

- Semantic Versioning
- Calendar Versioning

1.5.1. SNAPSHOT

Version, SNAPSHOT vs. NON-SNAPSHOT

1.5.2. Releases

What is a release from Maven point of view? Immutability?

Chapter 2. Single Module Projects

An often used setup of a Maven project has a single `pom.xml` file which contains the definition for dependencies and plugins etc. The project has a single artifact as a result which is often a `jar` file, but of course you can create `war`, `ear` or alike. That will be defined by the given `<packaging>jar</packaging>` tag.

2.1. Directory Structure

The following directory structure shows the default directory layout of a Maven project. That a convention to go that way. I strongly recommend to keep that structure (more details ??).

TODO:

- Based pom.xml without any supplemental configuration (in particular plugins)
  - Consequences of that?
- The need for pluginManagement (pinning plugin versions?)
  - Why is that needed?
- More reasons?

```
+--- pom.xml
  +--- src
        +--- main
              +--- java
              +--- resources
        +--- test
              +--- java
              +--- resources
```

2.1.1. Different dependent projects

Having several projects? Identifying duplication for example plugin configuration, pluginManagement etc. TODO: Add chapter about parents…

This should be moved to a location after testing setup and single module build.

Chapter 3. Testing with Maven

create a general summary (overview). The goal which should be reached in this section.

3.1. Project Setup

Explain the basic project setup..

Using fraction example.

3.1.1. Unit Testing

Basic unit testing, naming conventions, skip execution of unit tests

Using JUnit Jupiter as example because it state of the art.

3.1.2. Integration Testing

Basic combination of unit- and integration testing, naming conventions, skip execution of integration tests or both.

3.1.3. Testing Suites

With JUnit Jupiter

3.1.4. Testing Frameworks

JUnit 4

Test NG

JUnit Jupiter

Combining

Combination of all of them within a single build.

This part 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/

3.1.5. 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 following 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 appropriate package structure will contain your production code whereas the `src/test/java` plus the package structure will contain your unit test area.

https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html#includes

Table 1. Naming Schema for Unit Tests

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

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

This is an Example

```
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 location `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.

Unit Testing Example(1)

```
  <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>4.13.1</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.

Unit Testing Example(2)

```
<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 ?

3.1.6. 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).

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 of 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 contradiction integration tests are not independent and could not be parallelized 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/

3.1.7. Combining Unit- and Integration Testing

?? Controlling what should be executed and what not?

??

End To End Testing

How? Using a profile? Better solutions?

Chapter 4. Parent

the intention of a parent in general, super-pom, corporate parent? What should be part of a parent? What should NOT being part of a parent?

4.1. Overview

What is the target of a parent?

Which kind of parents do exist?

Corporate parent, multi-module-parent? Is there a difference?

Chapter 5. Multi Module Builds

Sometimes it is sufficient having a single pom file and a limited number of Java classes which are combined into a single jar file.

This is a solution in a number of cases but not for all kind of project types. In times of microservices projects becoming smaller even though it happens that you would like to separate out parts. For example having a generation of OpenAPI based code, or things like your jpa repositories (from a Spring Boot app) etc.

If you think about creating a command line app including a web application where you share code. That means you would have a command line part (module), a web application part(module) and common code (module). This results in a nice multi-module-build.

In the JEE area where several parts of an enterprise applications make sense to separate the creation of those kind of archive like Enterprise Application Archive(ear), the web application archive(war) etc.

Nevertheless, a project can become larger, when you realize it would make it easier to separate several parts out into a higher level of groupings.

In such cases it makes sense to create a multi-module-build. So start with a look on the basic structure of a multi-module-build.

5.1. Basic Structure

Lets us take the example from the Overview chapter about the command line app. We could name those modules like this:

- module-cli
- module-web
- module-common

5.1.1. Directory Structure

Based on this assumption a directory structure could look like this. We assume further that the `root` directory is the name of the git repository which will be created:

```
root
 +-- module-cli
 +-- module-common
 +-- module-web
```

Now let us dive into the details about such a project structure. What needs to be done to create such structure (apart from creating the directories). How could we build such a project and what kind of consequence are following from it.

Based on the idea of the relationship of the modules it is useful to create an appropriate directory structure.

```
 +--- pom.xml
   +--- module-cli
   +--- module-web
   +--- module-common
```

5.1.2. The Multi-Module-Parent

There are some parts which you need to pay attention to, to get a good working experience. The first thing is the `pom.xml` file on the root level of this structure. This is usually called the "parent" or multi-module-parent. This looks very similar like this:

pom.xml

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

  <modelVersion>4.0.0</modelVersion>

  ...

  <groupId>com.soebes.mpg.examples.mmb</groupId>
  <artifactId>parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  ..
  <modules>
    <module>module-cli</module>
    <module>module-common</module>
    <module>module-web</module>
  </modules>
  ..
</project>
```

This `pom.xml` file contains the usual parts like groupId, artifactId, version but at least one very important difference to other project which is the `<packaging>pom</packaging>`. And of course the module defined by the usage of the `<modules>..</modules>` tag including the list of module which are defined by `<module>⋯</module>`. The names like `module-cli` etc. correspond to the directories on the file system.

```
 +--- pom.xml
   +--- module-cli
   +--- module-web
```

```
    +--- module-common
```

In a usual Maven project you have a single `pom.xml` file, and then you start with the `src/` directory where your code lives. The directory is often enriched with other files like a `README.md` (or alike) etc.

The `README.md`(or alike) of course are often being found in multi-module-build as well.

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`.

But on the level of the parent `pom.xml` you will not find any `src` directory.

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:

pom.xml

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

  <modelVersion>4.0.0</modelVersion>

  <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.

Reasons to create a multi module setup:

- 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?
- Release all modules/projects at one point in time
- All the modules are related to each other ?

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?

5.2. Releasing a Multi module Project

From root, single module? ?

5.2.1. Examples

XXX

5.3. Spring Boot

A multi module project based on Spring Boot.

spring boot project setup.

```
root
 +--- pom.xml
   +--- jpa
   +--- controllers
   +--- application
```

Chapter 6. Code Coverage

What is code coverage…

6.1. JaCoCo

see article about jacoco..

6.1.1. single Module Setup

6.1.2. Multi Module Setup

6.2. Mutation Testing

Using pitest.

Chapter 7. Maven Assemblies

7.1. Overview

During the usage you will often be faced with the situation to create a kind of 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 fulfill the requirements of your builds furthermore we will take a look what kind of mistakes you can make and how to prevent them.

7.1.1. The Maven Assembly Plugin

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

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 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-cycle explicitly if you like to use it.

A note about the given example. In real life you should find this example suspicious, because 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
```

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.

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

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.

Module Sets

Dependency Sets

Sources

Predefined Descriptors

Currently there are four of them:

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

4. project

Chapter 8. Plugins

The plugins of Maven building the foundation of the Maven ecosystem, 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.

8.1. 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 (for example versions-maven-plugin, build-help-maven-plugin, buildnumber-maven-plugin).

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

jaxws-maven-plugin ? (URL?)

Describe more sources and other plugins

Groovy Plugins

8.1.1. 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.

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
```

Usually 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 cycle of the maven super pom!!

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

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 because 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 configuration 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

Let The Source Be With You

You usually write Java source code. Ok, ok sometimes you write source code in different languages Groovy, Kotlin, Scala 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. We will focus here on Java source code first.

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`.

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 (jar files) are being built.

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

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 exist 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).

- {link-maven-jar-plugina}

  Creating test-jars no transitive behaviour of test-jar artifact. Solution create a usual separate module.

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.

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 Exec Maven Plugin
- Mojo's SQL Maven Plugin
- Mojo's Templating Maven Plugin
- Mojo's Versions Maven Plugin
- More ?

http://mojo.codehaus.org/clirr-maven-plugin/ (really up-to-date? Not yet anymore.)

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

(DOES NOT EXIST ANYMORE: http://mvnplugins.fusesource.org/maven/1.4/maven-uberize-plugin/compared-to-shade.html)

maven-graph-plugin

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

(Looks interesting) 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/2.4.2/maven-plugin/usage.html spring-boot-maven-plugin

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

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 here

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>3.8.7</maven>
</prerequisites>
```

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

Example Configuration to define minimum Maven version

```
<project ...>

  <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>
```

Example Configuration to require same version

```
<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>
```

8.1.2. Maven compiler

Just a try. It's related to this blog post:

- https://blog.soebes.io/posts/2023/06/2023-06-24-how-to-use-jdk21-preview-features-incubator/

Chapter 9. Making Releases

Here you get an overview of the different options how to do a release with Maven.

9.1. Single Module Build

9.2. Multi Module Build

9.2.1. In One

The usual way?

Only releasing changed modules? What's behind this question?

Pros/Cons?

9.2.2. Single Childs

Why needed?

9.3. Maven Release Plugin

pros/cons

9.4. The Traditional Maven Way

- maven-release-plugin

How it works? What is needed (how to configure it?)

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

Change the preparation goal in maven-release-plugin

9.5. Releases The CD Way

Releases in the time of Continuous Delivery. How to solve this via Maven? Is this possible? Maven 3.8.7, 3.9.0, 4.0.0…

Using Ci Friendly. with maven-scm-plugin.

- Prevent problems with using property in distributionManagement

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

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

CI Delivery

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

Chapter 10. Continuous Integration Solution

10.1. More Details

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

Chapter 11. Build Smells

11.1. 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

11.2. Not Part of the Life Cycle

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

11.3. Multi Module Builds

accessing other modules via `${project.basedir}/../..` ?

11.3.1. Module Structure

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

11.3.2. The Install Hack

The Install Hack

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

11.3.3. Separation of Concerns

Multiple Purposes of a Module

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

Trying to build different artifacts with a single project setup instead of going for multi-module-setup. Clean separation.

11.4. Testing

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

11.5. 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!)

11.6. 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/

11.7. What about dependencies by profiles?

Describe why and how and what the drawbacks are?

Chapter 12. Plugin Development

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

How to test plugins?

Think about a good example here?

12.1. 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?

12.2. Basics

What is a Mojo?

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

12.2.1. Building a plugin

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

12.3. 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

12.3.1. Compatibility

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

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

Chapter 13. Plugin Configuration

How to configure plugins for the life cycle

13.1. General Configuration

Let us start with a minimal `pom` file.

Basic Example of Plugin Configuration

```
  <modelVersion>4.0.0</modelVersion>

  <!-- NEED TO REMOVE THE PARENT -->
  <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

There are several options to configure a plugin.

- xx
- xxx

13.2. The build life cycle

Chapter 14. Performance tipps

14.1. 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…

Chapter 15. Repository Manager

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.

15.1. Reasons

TODO:

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

Chapter 16. Best Practices

keep the defaults…

16.1. Plugin Management

Define the plugins via pluginManagement…

16.2. 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 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.

16.3. Dependencies / DependencyManagement

Scope only in real dependencies not in dependencyManagement

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

16.4. 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.

16.5. Company wide parent(s)

Manifest setup with master parent.

```
<configuration>
  <archive>
    <addMavenDescriptor>true</addMavenDescriptor>
    <index>true</index>
    <manifest>
      <addDefaultImplementationEntries>true</addDefaultImplementationEntries> ①
      <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries> ②
    </manifest>
    <manifestEntries>
      <artifactId>${project.artifactId}</artifactId> ③
      <groupId>${project.groupId}</groupId>
      <version>${project.version}</version>
      <buildNumber>${buildNumber}</buildNumber>
      <scmBranch>${scmBranch}</scmBranch>
    </manifestEntries>
  </archive>
</configuration>
```

① The groupId

② The artifactId

③ The version

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

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?

16.6. 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.

16.7. 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.

16.8. Nexus

why the order of the repositories does really matter…

16.9. 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

Chapter 17. Exceptions from Best Practices

This part will describe/discuss exceptions from the best practices or in other words ignoring conventions over configuration.

17.1. Layout

Sometimes people to say I don’t want to follow best practices. For example using a different directory layout? You can do that if you really need that, but it depends…

The first question I ask in such a situation is: Why do you like to do that? What kind of problem are you trying to solve?

Chapter 18. Site

Reporting in Maven. How to configure it? What can be done?

generating sites with Maven The maven `site` life cycle.

18.1. Maven Site Plugin

- Configure the site?
- What is needed?

Chapter 19. Profiles

Why and how to use Profiles.

Typical scenarios where to use profiles.

CI build (jenkins) etc.

Situations where you shouldn't use profiles.

19.1. Basics

19.1.1. Environment Dependent

19.2. Bad Practices

Don't use profiles to activate/deactive modules

Chapter 20. Different Environments

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.

https://blog.soebes.io/posts/2016/05/2016-05-08-building-for-multiple-environments/

https://blog.soebes.io/posts/2011/2011-07-29-maven-configuration-for-multipe-environments/

https://blog.soebes.io/posts/2011/2011-08-11-maven-configuration-for-multipe-environments-ii/

20.1. The Solutions

20.2. 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 on which environment you would like to build. What does in practice happen? You simply forget them. Have you remembered to change the configuration of your CI solution? Have you informed all your teammates? 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?

20.3. The next

Chapter 21. Maven 4

What kind of things have been enhanced, changed, improved. Compatibility.?

Plugins?

21.1. Consumer vs. Build POM

21.2. Improved Reactor Behaviour

21.3. Caching

21.4. Plugins

Appendix A: Example Appendix

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

A.1. Appendix Sub-section

Sub-section body.

A.2. Example Glossary

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

Chapter 22. 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.

Glossary

Maven

What is Maven

GAV

groupId, artifactId, version

Index