Maven for building Java applications - Tutorial

Lars Vogel, Hendrik Still, Simon Scholz (c) 2013 - 2019 vogella GmbH – Version Unspecified, 11.01.2019

This tutorial describes the usage of Maven for building Java applications.

1. What is Apache Maven?

1.1. The Apache Maven build system

Apache Maven is an advanced build tool to support the developer at the whole process of a software project. Typical tasks of a build tool are the compilation of source code, running the tests and packaging the result into JAR_ files. In additional to these typical build capabilities, Maven can also perform related activities, e.g., create web sites, upload build results or generate reports.

Maven allows the developer to automate the process of the creation of the initial folder structure for the Java application, performing the compilation, testing, packaging and deployment of the final product. It is implemented in Java which makes it platform-independent. Java is also the best work environment for Maven.

1.2. Key features of Maven

Apache Maven can be used in environments where common build tools like GNU Make or Apache Ant were used. The key features of Maven are:

- Convention over configuration: Maven tries to avoid as much configuration as possible, by choosing real world default values and supplying project templates (archtypes).
- Dependency management: It is possible to define dependencies to other projects. During the build, the Maven build system resolves the dependencies and it also builds the dependent projects if needed.
- Repository: Project dependencies can be loaded from the local file system, from the Internet or public repositories. The company behind the Maven project also provides a central repository called *Maven Central*.
- Extensible via plug-ins: The Maven build system is extensible via plug-ins, which allows to keep the Maven core small. The Maven core does for example not know how to compile Java source code, this is handled by the compiler plug-in.

1.3. Maven Central

Maven Central is an open repository provided by the company Sonatype. This repository hosts libraries which can be used in your build. By default, a Maven build uses Maven Central to search for required libraries.

2. Installation of the Maven command line tools

2.1. Downloading and installing Maven

If you plan to use Maven only from within the Eclipse IDE, this installation is not required.

In case you want to be able to use Maven from the command line, you need to install the Maven command line support.

For a manual installation you can download Maven from the Maven Download (http://maven.apache.org/download.cgi) page. Extract the downloaded distribution to a selected folder on your computer and add the `M2_HOME` environment pointing to this directory. Add `M2_HOME/bin` to your path variable.

See Maven installation description (http://maven.apache.org/download.html#Installation) for a detailed installation guide.

2.2. Ubuntu

```
# if that does not work, try
sudo apt-get install maven
```

2.3. Validate installation

To validate that Maven was correctly installed, open a console and use the following command:

```
# command
mvn -version

# output should be similar to

Apache Maven 3.0.5
Maven home: /usr/share/maven
Java version: 1.7.0_55, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-7-openjdk-amd64/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.13.0-33-generic", arch: "amd64", family: "unix"
```

You should receive as output feedback which version of Maven you have installed.

3. Running a Maven build

3.1. Running a command line build

Maven provides a command line tool.

To build a Maven project via the command line, run the `mvn` command from the command line. The command should be executed in the directory which contains the relevant pom file. You need to provide the `mvn` command with the life cycle phase or goal to execute.

The Maven tooling reads the pom file and resolves the dependencies of the project. Maven validates if required components are available in a local repository. The local repository is found in the *.m2/repository* folder of the users home directory.

If the dependency is not available in the build reactor or the local repo, Maven downloads the depended artifacts from the central repository or the specified ones into the local repository.

Maven executes all life cycle phases until the specified one.

For example the `mvn clean install` command triggers the jar packaging. This includes compiling the sources, executing the tests and packaging the compiled files in a JAR file. As last step the `install` phase installs the resulting artifact into the local repository, so it can be used as dependencies by other Maven builds.

Maven creates the build result in the *target* folder.

CONSOLE

```
mvn install
```

1 compile, build and install the build result

To ensure that the build target is removed before a new build, add the `clean` target.

CONSOLE

```
mvn clean install
```

By default, Maven checks online if the dependencies have been changed. If you want to use your local repository, you can use the `-o` to tell Maven to work offline.

CONSOLE

```
mvn -o clean install
```

3.2. Dealing with build failure

If you are running a complex multi-module project build, you can define how the Maven build system should react to errors in one module.

- -ff, –fail-fast - Stop at first module build failure
- -fn, –fail-never - NEVER fail the build, regardless of module build result

The -fn and -fae options are useful to verify builds that are running within a continuous integration tool like Jenkins and to see all errors in the build.

3.3. Scaffolding a project with Maven

Maven supports project scaffolding, based on project templates (called *archtype*). This is provided by the *archetype* plug-in. Maven provides archetypes for almost anything, from a simple Java application to a complex web application.

The goal of this scaffolding is to allow a fast start into the Maven world. It provides the "standardized" folder structure of software projects.

You can create a project by executing the generation *goal* on the archetype plugin via the following command: `mvn archetype:generate`.

This starts the generation process in the interactive mode and asks you for several settings.

4. Maven Wrapper

You can create a wrapper script which downloads automatically the correct version of Maven. This way users do not need to install Maven on their local machine.

4.1. Creating a Maven Wrapper

Create a Maven Wrapper for a project with the latest available Maven version.

```
cd {your-project}
mvn -N io.takari:maven:wrapper
```
CONSOLE

Create a Maven Wrapper for a project with a specified Maven version by using the maven property.

```
cd {your-project}
mvn -N io.takari:maven:wrapper -Dmaven=3.3.0
```
CONSOLE

When wrapper goal has been executed the following files will be created in the maven project.

- mvnw (shell script for unix systems)
- mvnw.cmd (batch file for windows)
- .mvn/wrapper/maven-wrapper.jar (Maven Wrapper JAR)
- .mvn/wrapper/maven-wrapper.properties (Maven Wrapper properties)

These Maven Wrapper files should be checked in into version control (e.g. GIT or SVN), so that others who checkout the sources are able to build the projects without the need to install Maven manually in the first place. And when using the Maven Wrapper there is no need to worry about the right version of Maven, since the project's Wrapper already specifies and downloads it automatically.

The `-N, --non-recursive` command line option specifies that only the project in the current directory is built without building it's submodules. So the Maven Wrapper will only be applied for the main project and not in every submodule.

4.2. Executing a Maven Wrapper

To run the Maven Wrapper the mvnw for unix systems or mvnw.bat for windows systems can be used.

UNIX:

```
./mvnw clean package
```
CONSOLE

WINDOWS:

5. Exercise: Create and build a Java project with Maven

In this exercise you create a Java project with the Maven command line and build this project.

5.1. Project generation

In this project we use the scaffolding functionality of Maven to create a Java project. To avoid the interactive mode, all required properties are passed directly to the command. Otherwise Maven asks for all the required parameters. Enter the following into one line in the command shell(the backslash masks the line breaks).

CONSOLE

```
mvn archetype:generate -DgroupId=com.vogella.build.maven.java \
-DartifactId=com.vogella.build.maven.java  \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DinteractiveMode=false
```

With this command Maven generates a Java project.

If this is the first time you execute this goal, this may takes some time. It also produces additional output compared to a second run. This is because Maven first loads all required plug-ins and artifacts for the project generation from the Maven central repository.

5.2. Review generated project

Validate that Maven generated a project on your file system similar to the following structure.

CONSOLE

```
com.vogella.build.maven.java/
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── vogella
    │               └── build
    │                   └── maven
    │                       └── java
    │                           └── App.java
    └── test
        └── java
            └── com
                └── vogella
                    └── build
                        └── maven
                            └── java
                                └── AppTest.java
```

You have generated a whole Maven project structure with Java source code. Maven created a App.java class in the *./src/main/* folder, which is just a simple "Hello World" program. It also created an example test class in *./src/test/*. In the root folder there is a *pom.xml* file.

CONSOLE

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.vogella.build.maven.java</groupId>
    <artifactId>com.vogella.build.maven.java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>com.vogella.build.maven.java</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

5.3. Compile your sources

Now you want to compile your Java sources. For this switch on the command line into our projects root directory and trigger the following Maven command.

CONSOLE

```
$ mvn compile

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building com.vogella.build.maven.java 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) ...
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources..
[INFO] skip non existing resourceDirectory .....
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile)...
[INFO] Nothing to compile - all classes are up to date
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.755s
[INFO] Finished at: Thu Oct 17 00:59:13 CEST 2013
[INFO] Final Memory: 7M/106M
[INFO] ------------------------------------------------------------------------
```

Maven now runs through all life cycle phases, which were needed by the `compile` phase. It resolves the dependencies, loads the JUnit artifact and built the sources. It even executed the JUnit test.

5.4. Create a JAR file

Now you want to create an executable JAR file out of our project. The `package` goal creates a deployable JAR file.

To ensure previous build artifacts are removed, you can use the `clean` goal.

CONSOLE

```
mvn clean package
```

Afterwards you can run the packed program.

CONSOLE

```
$ java -cp target/com.vogella.build.maven.java-1.0-SNAPSHOT.jar \
com.vogella.build.maven.java.App
Hello World!
```

CONSOLE

```
$ cd com.vogella.build.maven.java
$ mvn package
...

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.vogella.build.maven.java.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.014 sec

Results :

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

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ com.vogella.build.maven.java ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.631s
[INFO] Finished at: Thu Oct 17 01:05:11 CEST 2013
[INFO] Final Memory: 11M/102M
[INFO] ------------------------------------------------------------------------
```

5.5. Running the test

Instead of running a full build with packaging, it is also possible to just run the test phases of the Maven life cycle.

```
....
 T E S T S
-------------------------------------------------------
Running com.vogella.build.maven.java.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec

Results :

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

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
.....
```

5.6. Remove all build results / Clean the project

To clean the project and so remove the generated files in the *./target/* folder, run the following command.

CONSOLE

```
$ mvn clean
```

6. Exercise: execute a Java program with Maven

If you want to execute a program you can use the `exec-maven-plugin`. This is demonstrated in the following *pom.xml* file. To trigger this use the `exec:java` target in maven.

XML

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.vogella.build.maven.intro</groupId>
    <artifactId>com.vogella.build.maven.intro</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>mavenintroduction</name>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
                <configuration>
                    <mainClass>com.vogella.build.maven.intro.Main</mainClass>
                </configuration>
            </plugin>
        </plugins>

    </build>
</project>
```

```
    <artifactId>com.vogella.build.maven.intro</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>mavenintroduction</name>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
                <configuration>
                    <mainClass>com.vogella.build.maven.intro.Main</mainClass>
                </configuration>
            </plugin>
        </plugins>

    </build>
</project>
```

```
Problems @ Javadoc Declaration Console
<terminated> com.vogella.build.maven.intro [Maven Build] /usr/lib/jvm/ja
[INFO] Installing /home/vogella/workspace/vogella/com.vogella
[INFO] Installing /home/vogella/workspace/vogella/com.vogella
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ com.v
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ com.v
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ com.v
Hello Maven!
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
[INFO] Total time: 1.608s
[INFO] Finished at: Sat Jan 04 13:58:14 CET 2014
[INFO] Final Memory: 15M/301M
[INFO] ------------------------------------------------------
```

7. Configuration and coordinates of a Maven project

7.1. Project Object Model (POM)

The configuration of a Maven project is done via a *Project Object Model* (POM). This model is typically represented by a *pom.xml* file. This Maven configuration file is called the pom file.

The pom file describes the project, configures plugins, and declares dependencies. It names the project and provides a set of unique identifiers (called coordinates) for a project. It also defines the relationships between this project and others through dependencies, parents, and prerequisites.

A multi project pom file includes a *modules* section. This section tells Maven which project are part of the build.

In the *build* section of the pom you can define plugins for which are needed for the build.

7.2. The GAV as project unique identifier - project coordinates

Maven coordinates and defines a set of identifiers which can be used to uniquely identify a Maven component. This can for example be used to define the exact version of the JUnit test library which should be used for the project. These are defined via the groupId, artifactId, version and packaging property.

Table 1. Coordinate attributes

Name	Description

groupId	Defines a unique base name of the organization or group that created the project. This is normally a reverse domain name. For the generation the groupId also defines the package of the main class.
artifactId	Defines the unique name of the project. If you generate a new project via Maven this is also used as root folder for the project.
packaging	Defines the packaging method. This could be e.g. a jar, war or ear file. If the packaging type is pom, Maven does not create anything for this project, it is just meta-data.
version	This defines the version of the project.

The project `groupId:artifactId:version` (also known as GAV) makes that project unique.

The full Maven coordinates are often written in the following format: groupId:artifactId:packaging:version.

By default, this is the only configuration file required for the build process.

Maven always executes against an *effective pom*. This is a combination of settings from this project's pom.xml, all parent pom, a super-pom defined within Maven, user-defined settings, and active profiles.

The result of a build is called *artifact*. An artifact can be for example an executable or an archive of documents.

8. Maven plug-ins, goals and the life cycle

8.1. Maven Plug-ins and goals

A Maven plugin is a collection of one or more *goals*. A goal is a "unit of work" in Maven. It is possible to execute goals independently or a part of a larger chain of goals.

Goals can define parameters, which may have default values. Plugin goals can be attached to a life cycle phase. The goals are executed based on the information found in the pom of the project, e.g., the compiler:compile goal checks the POM for relevant parameters.

8.2. Maven life cycle

Every build follows a specified *life cycle*. Maven comes with a default life cycle that includes the most common build *phases* like compiling, testing and packaging.

The following lists gives an overview of the important Maven life cycle phases.

- validate - checks if the project is correct and all information is available
- compile - compiles source code in binary artifacts
- test - executes the tests
- package - takes the compiled code and package it, for example
- integration-test - takes the packaged result and executes additional tests, which require the packaging
- verify - performs checks if the package is valid
- install - install the result of the package phase into the local Maven repository
- deploy - deploys the package to a target, i.e. remote repository

If you instruct Maven to execute a phase, it executes all existing phases in the pre-defined sequence until it has executed the defined phase. All relevant goals are executed during this process. A goal is relevant for a phase if the Maven plug-in or the pom binds this goal to the corresponding life cycle phase.

8.3. Packages and goal bindings

Each packaging contains a list of bindings for goals to a particular life cycle phase. For example, the jar packaging binds the following goals to the life cycle phases.

Table 2. Packaging goal

Life cycle phase	Goal binding
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

8.4. Adding goals to life cycle phases

You can add goals to life cycle phases by configuring more Maven plug-ins and adding them to a life cycle in your pom file. You need to specify which goal should be executed. If the plug-in does not specify the default life cycle it should run, you must also specify the life cycle phase it should run.

```xml
<plugin>
   <groupId>com.vogella.example</groupId>
   <artifactId>vogella-some-maven-plugin</artifactId>
   <version>1.0</version>
   <executions>
     <execution>
       <phase>verify</phase>
       <goals>
         <goal>checklinks</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
```

9. Maven repositories and dependency resolution

9.1. Maven repositories

During the initial phase of a build Maven validates if you have the specified version of all required artifact dependencies and Maven plug-ins. If required, it retrieves them from a Maven repository. A repository is a collection of project artifacts stored in a directory structure similar to the Maven coordinates of the project.

If necessary, Maven downloads these artifacts and plug-ins into a local repository. The default local repository is located in the home directory of the user in the *.m2/repository* folder. If an artifact or a plug-in is available in the local repository Maven uses it.

Maven uses a default remote repository location (http://repo1.maven.org/maven2) from which it downloads the core Maven plugins and dependencies. You can configure Maven to use more repositories and replace the default one.

If you run the `mvn install` command, the generated artifacts are installed into the local Maven repository.

9.2. Maven dependency resolution and Maven reactor

```xml
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

During a build, the Maven system tries to resolve the dependencies of the modules which are build. To resolve dependencies, Maven uses the following sources in the given order:

- Projects which are included in the same Maven run (the so called Maven *reactor*)
- Local repository
- Maven central repository

10. Multi module projects (Aggregator)

Maven supports building multiple projects. A multi module project (aggregator) is defined by a parent POM referencing one or more projects. This aggregator can contain also the build configuration or include another parent POM to get this configuration.

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.vogella.tychoexample</groupId>
    <artifactId>com.vogella.tycho.aggregator</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <parent>
        <groupId>com.vogella.tychoexample</groupId>
        <artifactId>com.vogella.tycho.parent</artifactId>
        <version>1.0.0-SNAPSHOT</version>
        <relativePath>../com.vogella.tycho.parent</relativePath>
    </parent>

    <modules>
        <module>../com.vogella.build.targetdefinition</module>
        <module>../com.vogella.tycho.plugin1</module>
        <module>../com.vogella.tycho.plugin2</module>
        <module>../com.vogella.tycho.rcp</module>
        <module>../com.vogella.tycho.product</module>
        <module>../com.vogella.tycho.feature</module>
        <module>../com.vogella.tycho.update</module>
        <module>../com.vogella.tycho.unittests</module>
    </modules>
</project>
```

The packaging type of such a POM is pom, as such a project will not result in any build output.

11. Using profiles and properties in Maven

11.1. Using profiles

Maven supports the usage of profiles to define different configurations. If you start Maven, you can instruct it to use a certain profile. For this you specify the `-P` parameter followed directly (without whitespace) by the profile, e.g. `-PyourProfile`.

```
      <activation>
      </activation>
    <properties>
      <db.location>URL_to_dev_system</db.location>
      <logo.image>companylogo.png</logo.image>
    </properties>
  </profile>
  <profile>
    <id>production</id>
        <properties>
            <db.location>URL_to_prod_system</db.location>
            <logo.image>companylogo2.png</logo.image>
        </properties>
    </profile>
  </profiles>
```

11.2. Using properties

You can specify properties in your build files. You can override them via the command line with the `-D` parameter followed directly (without whitespace) by its value. See the next chapter for an example.

11.3. Example for properties: Skipping the tests in a Maven build

You can specify via a property that your tests should be skipped during the build. This property (among with several others for demo purposes) is defined in the following snippet.

```xml
<properties>
<skipTests>true</skipTests>
<maven.build.timestamp.format>yyyyMMdd-HHmm</maven.build.timestamp.format>
<buildTimestamp>${maven.build.timestamp}</buildTimestamp>
<buildId>${buildType}${buildTimestamp}</buildId>
</properties>
```

How to override these parameters via the command line is demonstrated by the following listing.

```xml
mvn clean install -DskipTests=false
```

11.4. Useful properties

Table 3. Useful properties

Property	Description
skipTests	true or false, specifies if tests should be executed or not
showWarnings	true or false, defines if the Maven build shows the compiler warnings

12. Maven and version control systems

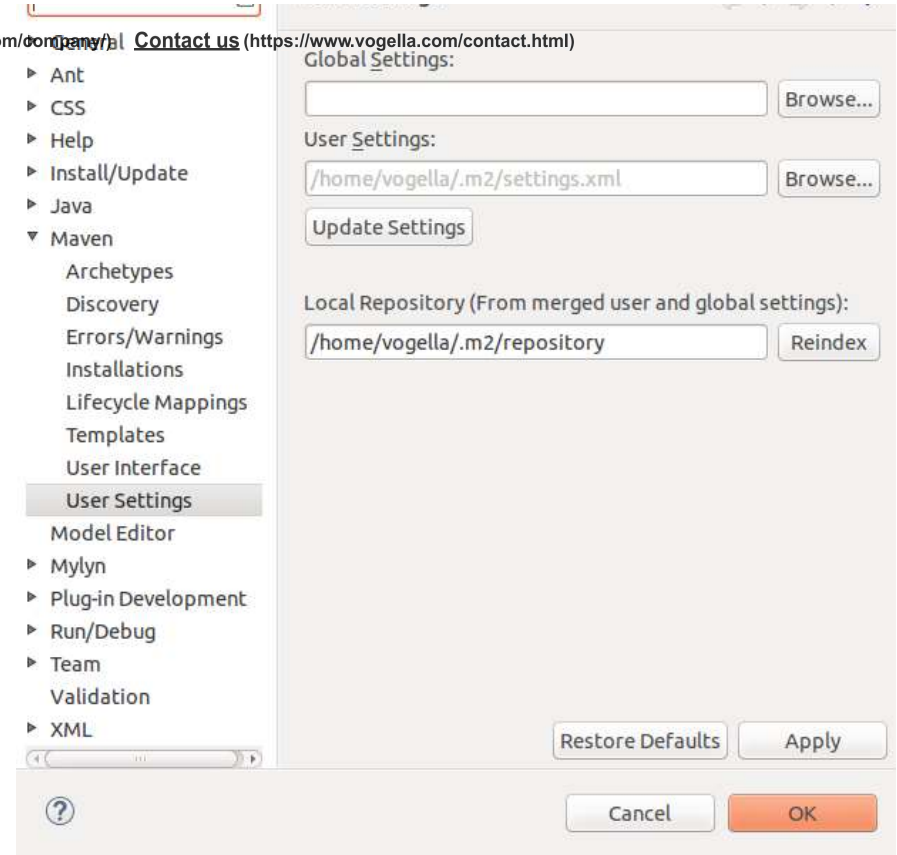
Maven generates its output into the *target* folder of each project. This build output should not get included into your version control system.

Add this directory to your ignore resources. For example, if you use Git as version control system, add the "target/" entry to your *.gitignore* file in the root of each project.

13. Maven settings

Maven allows to define settings on a global, user and project level. The common case is to define on a user level, settings like a proxy server or passwords to upload build artifacts to a server.

You can view the file locations in the Eclipse IDE via Windows > Preferences > Maven > User settings.



The following *settings.xml* file defines a proxy server. If this snippet is located in the *.m2* folder of the users home, Maven uses this proxy.

```xml
<settings>
 <proxies>
  <proxy>
   <active>true</active>
   <protocol>http</protocol>
   <host>proxy</host>
   <port>8080</port>
   <username>your_user</username>
   <password>your_password</password>
   <nonProxyHosts>www.google.com|*.test.com</nonProxyHosts>
  </proxy>
 </proxies>
</settings>
```

14. Useful Maven parameters

The following table lists useful Maven parameters.

Table 4. Maven build parameters

Parameter	Description
–log-file log.txt	Maven build output is written to the specified file
–debug	Outputs more information during the build process

15. Useful maven plugins to analyse a project

15.1. Show the dependency tree

Maven provides a plugin, which can be used to visualize a dependency tree to either the console or an output file.

```
mvn dependency:tree -Dverbose -DoutputFile=/home/simon/maven-dependencies
```

```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.vogella</groupId>
    <artifactId>com.vogella.maven.dependencies</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.10</version>
        </dependency>
    </dependencies>

</project>
```

The output of the `mvn dependency:tree -Dverbose` command can for example look like the following.

```
com.vogella:com.vogella.maven.dependencies:pom:0.0.1-SNAPSHOT
\- org.slf4j:slf4j-log4j12:jar:1.7.10:compile
   +- org.slf4j:slf4j-api:jar:1.7.10:compile
   \- log4j:log4j:jar:1.2.17:compile
```

15.2. Versions plugin

The `display-dependency-updates` goal of the *Versions Maven Plugin* can be used to determine, if there are newer versions of a dependency available.

For example, running `mvn versions:display-dependency-updates` for a project might result in the following output.

```
--- versions-maven-plugin:2.1:display-dependency-updates (default-cli) @ manifest-maven-plugin ---
artifact biz.aQute:bnd: checking for updates from central
artifact biz.aQute:bnd: checking for updates from release
artifact commons-io:commons-io: checking for updates from central
artifact commons-io:commons-io: checking for updates from release
artifact junit:junit: checking for updates from central
artifact junit:junit: checking for updates from release
artifact org.apache.maven:maven-plugin-api: checking for updates from central
artifact org.apache.maven:maven-plugin-api: checking for updates from release
artifact org.apache.maven.plugin-tools:maven-plugin-annotations: checking for updates from central
artifact org.apache.maven.plugin-tools:maven-plugin-annotations: checking for updates from release
The following dependencies in Dependencies have newer versions:
  junit:junit ..................................... 3.8.1 -> 4.12
  org.apache.maven:maven-plugin-api ..... 3.2.3 -> 3.2.5

------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 4.214s
Finished at: Tue Feb 10 16:00:31 CET 2015
Final Memory: 13M/297M
------------------------------------------------------------------------
```

16. Apache Maven resources

Five things you did not know about Maven (http://www.ibm.com/developerworks/library/j-5things13)

Maven deployment artifact (http://maven.apache.org/plugins/maven-deploy-plugin/examples/deploy-ftp.html)

Maven tools examples on GitHub (https://github.com/pkainulainen/maven-examples)

Maven Wrapper GitHub repo (https://github.com/takari/maven-wrapper)

17. vogella training and consulting support

Appendix A: Copyright, License and Source code

Version unspecified
Last updated 2019-05-21 15:07:00 +0200