

Kai Uwe Bachmann

# Maven 2

Eine Einführung, aktuell zur Version 2.0.9



# 3 Das erste Projekt

## 3.1 Ein neues Projekt



### Hinweis

Da Maven ein Kommandozeilen-Tool ist, erfolgt der Aufruf immer aus der Kommandozeile bzw. Shell. Sofern nicht anders angegeben, müssen Sie sich in den Beispielen dazu im Projektverzeichnis befinden. Ebenso beziehen sich alle Pfadangaben auf dieses Verzeichnis.

Wenn Sie die Installation und Konfiguration erfolgreich hinter sich gebracht haben, geht es an das erste Projekt. Maven arbeitet mit einer Beschreibungsdatei (POM), in der alle benötigten Informationen über ein Projekt abgelegt werden. Zu diesen gehören neben dem Projektnamen und der Version im Wesentlichen die Art des Projektes sowie seine Abhängigkeiten. Sie können sich allerdings die Arbeit ersparen, jedes Mal die POM-Datei neu zu schreiben (oder aus einer Vorlage zu kopieren). Dazu bietet uns Maven ein simples Kommando an, mit dem wir beginnen können.

Zunächst wechseln wir in unser Projektverzeichnis und geben **mvn archetype:generate** ein. Was nun passiert, ist, dass Maven zunächst das Plugin `archetype` und einige andere nachlädt (sofern noch nicht geschehen). Danach fragt uns Maven, welche Art von Projekt wir erzeugen wollen. Dazu wird die Liste aller bekannten Projekttypen angezeigt. In dieser Liste suchen wir den Eintrag `maven-archetype-quickstart` und geben die Zahl, die davor steht, ein (diese dürfte auch als Default angeboten werden).

Nachdem Sie sich für einen Archetype entschieden haben, werden noch die weiteren benötigten Parameter abgefragt. Dazu gehört zunächst die GroupID, für die wir `de.meineGruppe.app` eingeben. Die Gruppen-ID ist so etwas wie der Familienname des Projektes. Auf diese Weise können Projekte auf verschiedenen Ebenen organisiert werden.

Nach der Gruppen-ID wird die Artefakt-ID (`artifaktId`) abgefragt. Für unser Beispiel verwenden wir `Kapite13`. Als vorletzter Pflichtparameter wird anschließend nach der Versionsnummer gefragt. Da es sich um ein neues Projekt handelt, wird »1.0-SNAPSHOT« angeboten, was wir hier auch verwenden. Der letzte anzugebende Parameter ist das Package, unter dem die ersten Klassen angelegt werden. Hier verwenden wir wie bei der Gruppe »`de.meineGruppe.app`«.

Sind alle Parameter eingegeben, zeigt uns Maven nochmals an, was alles ausgewählt wurde:

```
Confirm properties configuration:
groupId: de.meineGruppe.app
artifactId: Kapitel3
version: 1.0-SNAPSHOT
package: de.meineGruppe.app
```

Sind wir uns sicher, dass alles seine Richtigkeit hat, bestätigen wir dies mit der Eingabe von »Y« (oder einfach nur Enter), worauf alle benötigten Dateien und Verzeichnisse angelegt werden. Dabei erscheint folgende Ausgabe:

```
[INFO] -----
[INFO] Using following parameters for creating OldArchetype:
                                maven-archetype-quickstart:RELEASE
[INFO] -----
[INFO] Parameter: groupId, Value: de.meineGruppe.app
[INFO] Parameter: packageName, Value: de.meineGruppe.app
[INFO] Parameter: basedir, Value: D:\Projekte\beispiele\kap03
[INFO] Parameter: package, Value: de.meineGruppe.app
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: Kapitel3
[INFO] **** End of debug info from resources from generated POM ****
[INFO] OldArchetype created in dir: D:\Projekte\beispiele\kap03\Kapitel3
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 4 minutes 54 seconds
[INFO] Finished at: Sat Aug 02 21:49:05 CEST 2008
[INFO] Final Memory: 8M/15M
[INFO] -----
```

Was können wir hieraus ersehen? Zunächst einmal, dass das, was wir getan haben, erfolgreich (successful) war. Weiterhin sehen wir, dass unser Projekt unter der Gruppe (groupId) »de.meineGruppe.app« angelegt wurde und im Verzeichnis D:\Projekte\beispiele\kap03\Kapitel3 liegt. Die Projektversion ist »1.0-SNAPSHOT«.

Neben dieser interaktiven Eingabe können Sie auch alle Parameter gleich beim Aufruf von Maven eingeben. So fragt z.B. Maven bei der Eingabe von

```
mvn archetype:generate -DgroupId=de.meineGruppe.app -DartifactId=Kapitel3
```

nicht mehr nach Gruppe und Artefakt.

Nun wollen wir einen kurzen Blick auf die erzeugten Verzeichnisse und Dateien werfen.

Als Erstes sehen wir die Datei `pom.xml` in unserem Projektverzeichnis. Dies ist für Maven die wichtigste Datei überhaupt. Sie sollte in etwa so aussehen:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>de.meineGruppe.app</groupId>
  <artifactId>Kapitel3</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Kapitel3</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>
```

Wichtig ist zunächst einmal die `artifactId`, dies ist unser Projektname, sowie die `Dependencies`. Wie Sie hier sehen, hat Maven für uns bereits eine Abhängigkeit zu JUnit eingetragen, hier in der Version 3.8.1.

Weiterhin wurden zwei Java-Dateien erzeugt, `/src/main/java/de/meineGruppe/app/App.java` und `/src/test/java/de/meineGruppe/app/AppTest.java`, welche das altbekannte »Hello World« sowie einen einfachen Test enthalten.

## 3.2 Kompilieren und Testen

`App.java` ist das gute alte »Hello World«, welches wir auch gleich einmal kompilieren und ausführen wollen. Dazu rufen wir vom Projektverzeichnis zunächst **`mvn compile`** auf. Auch hier werden wieder einige Plugins nachgeladen (keine Angst, das passiert nur beim ersten Mal, danach liegen sie in einem lokalen Repository) und wenn alles erfolgreich verlaufen ist, steht in der Ausgabe Folgendes:

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Kapitel3
[INFO]   task-segment: [compile]
[INFO] -----
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to ...\\Kapitel3\\target\\classes
[INFO] -----
```

```
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 2 seconds
[INFO] Finished at: Sat Aug 02 22:00:11 CEST 2008
[INFO] Final Memory: 3M/8M
[INFO] -----
```

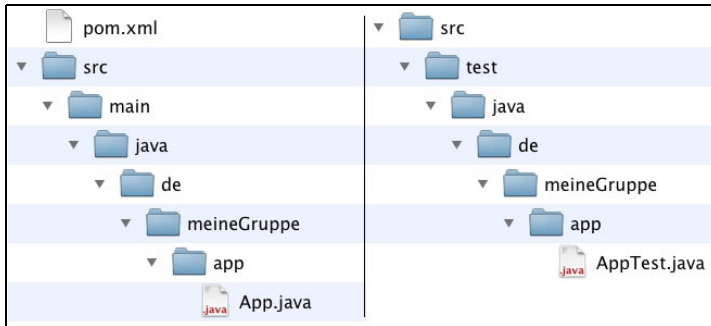


Abbildung 3.1: Struktur eines einfachen Projektes

Natürlich unterscheiden sich die statistischen Angaben zur Build-Zeit und zum Speicherverbrauch.

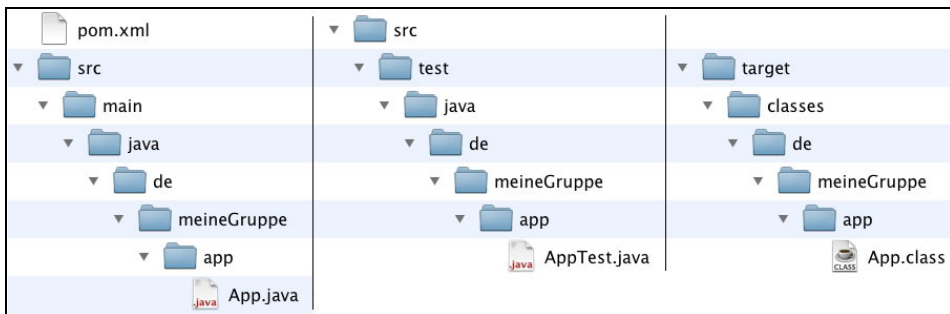


Abbildung 3.2: Struktur eines einfachen Projektes nach dem Kompilieren

Was ist nun geschehen?

Maven hat die Java-Dateien unter `src/main/java` kompiliert (in diesem Fall nur eine) und die Ergebnisse nach `target/classes` geschrieben. Wir können nun die Applikation einfach durch den Aufruf mittels `java <classname>` starten, aber wir haben ja auch eine Testklasse (`AppTest.java`), was ist mit der?

Um sie zu kompilieren, bemühen wir nochmals Maven, jetzt jedoch mittels `mvn test`.

Wie wir sehen, wird jetzt ebenfalls die Testklasse kompiliert und im Verzeichnis `target/test-classes` abgelegt. Aber wir sehen noch mehr. Zum einen erscheint in der Konsole eine Ausgabe der Art:

```

...
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Compiling 1 source file to ...\\Kapitel3\\target\\test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: ...\\Kapitel3\\target\\surefire-reports

```

```

-----
T E S T S
-----
Running de.meineGruppe.app.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec

Results :

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

[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Sat Aug 02 22:01:47 CEST 2008
[INFO] Final Memory: 5M/11M
[INFO] -----

```

Wie wir sehen, wurde genau ein Test erfolgreich ausgeführt.

Zum anderen gibt es ein weiteres Verzeichnis `target/surefire-reports` mit einer Text- und einer XML-Datei. Maven hat nicht nur die Testklasse kompiliert, sondern auch gleich ausgeführt und das Ergebnis in diesem Verzeichnis abgelegt. Die zwei Dateien haben prinzipiell den gleichen Inhalt, nur in unterschiedlichen Formaten. Während die Textdatei einfach die Logausgabe des Unit-Tests enthält, ist die XML-Datei zusätzlich mit einer Reihe statistischer Werte und Angaben zur Laufzeitumgebung gefüllt. Dadurch kann diese von entsprechenden Tools zur Protokollierung und Auswertung weiterverwendet werden.

Ein weiterer Punkt fällt auf, wenn man sich die Konsolenausgabe anschaut. Maven versucht, auch die Datei `App.java` zu kompilieren, da diese aber aktuell ist, wird sie kurzerhand übersprungen.

Wie sieht das Ganze aus, wenn die Tests Fehler enthalten? Dazu verändern wir einfach den Test von `assertTrue( true );` nach `assertTrue( false );`

Wie wir nun in der Ausgabe sehen, ist der Build fehlgeschlagen.

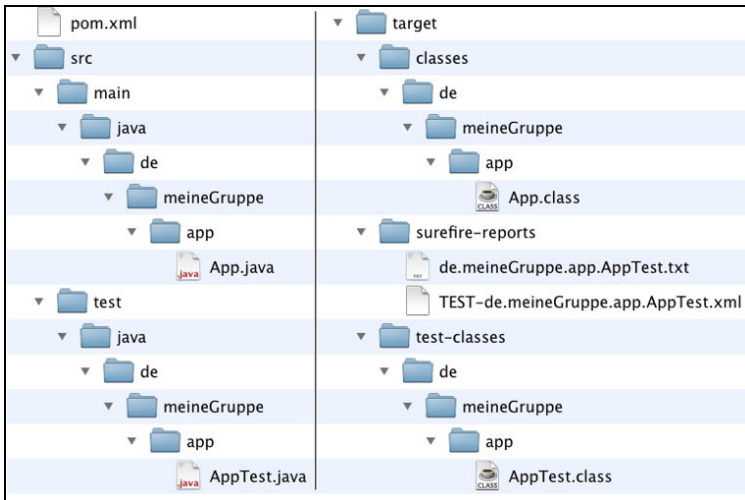


Abbildung 3.3: Struktur eines einfachen Projektes nach dem Testen

```

...
Running de.meineGruppe.app.AppTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0,          \
                                     Time elapsed: 0.036 sec <<< FAILURE!

Results :

Failed tests:
  testApp(de.meineGruppe.app.AppTest)

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

[INFO] -----
[ERROR] BUILD FAILURE
[INFO] -----
[INFO] There are test failures.

Please refer to ...\\Kapitel13\\target\\surefire-reports          \
                                     for the individual test results.

[INFO] -----
[INFO] For more information, run Maven with the -e switch
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Sat Aug 02 22:02:50 CEST 2008
[INFO] Final Memory: 5M/10M
[INFO] -----

```

Näheres zum Grund finden wir in der Datei `/target/surefire-reports/de.meine-Gruppe.app.AppTest.txt`

```
-----  
Test set: de.meineGruppe.app.AppTest  
-----  
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: \\  
    0.11 sec <<< FAILURE!  
testApp(de.meineGruppe.app.AppTest) Time elapsed: 0.02 sec <<< FAILURE!  
junit.framework.AssertionFailedError  
at junit.framework.Assert.fail(Assert.java:47)  
at junit.framework.Assert.assertTrue(Assert.java:20)  
at junit.framework.Assert.assertTrue(Assert.java:27)  
at de.meineGruppe.app.AppTest.testApp(AppTest.java:36)
```

Wie wir erwartet haben, hat unsere geänderte Assertion zugeschlagen.

Damit wir mit unserer Einführung weitermachen können, beheben wir den Fehler im Test wieder.



#### Tipp

Wollen Sie nur die Testklassen kompilieren, ohne die Tests auszuführen, können Sie dies mit **mvn test-compile** erreichen.

## 3.3 JAR-Erzeugung

Wir haben unsere Klassen nun kompiliert und mit der Test-Suite überprüft. Nun müssen wir für eine ordentliche Java-Applikation das Ganze noch in eine JAR-Datei packen. Auch dieses erledigt Maven wieder für uns.

Durch den Aufruf von **mvn package** erzeugt Maven die gewünschte JAR-Datei (sofern die Tests fehlerfrei laufen), und wenn wir sie uns näher anschauen (**jar -tvf target/Kapitel3-1.0-SNAPSHOT.jar**), sehen wir, dass sich nur die Applikationsklasse, nicht aber die Testklasse darin befindet. Wollen wir eine JAR-Datei für die Testklassen bauen, können wir dies einfach über **mvn jar:test-jar** erledigen. Diese wird dann auch in das Verzeichnis `target/` geschrieben.

Grundsätzlich erzeugt jedes Maven-Projekt nur eine Hauptdatei (Artefakt, z.B. die JAR-Datei), es kann aber eine Reihe von begleitenden Artefakten wie Testdateien, Dokumentation (z.B. über JavaDoc) oder auch Webseiten erzeugen.

Ein weiterer Aufruf von **mvn install** befördert nun noch die JAR-Datei in unser lokales Repository, von wo aus sie durch andere Projekte benutzt werden kann.

## 3.4 Ressourcen

Ein Java-Programm besteht natürlich nicht nur aus den Klassen, sondern auch aus Properties, XML-Dateien und vielen anderen. Der Ort, diese zu speichern, ist das Verzeichnis `/src/main/resources/` bzw. `/src/test/resources/` für die Tests. Alle Dateien und Verzeichnisse, die Maven dort findet, kopiert es in die jeweiligen Targets, wobei hierauf noch Filter angewendet werden können. So ist es möglich, Einstellungen in die Dateien zu schreiben oder verschiedene Dateien zusammenzuführen.

Um dies einmal auszuprobieren, erzeugen wir einfach zwei Dateien, `/src/main/resources/App.txt` und `/src/test/resources/TestApp.txt`. Der Inhalt soll dabei zuerst einmal egal sein.

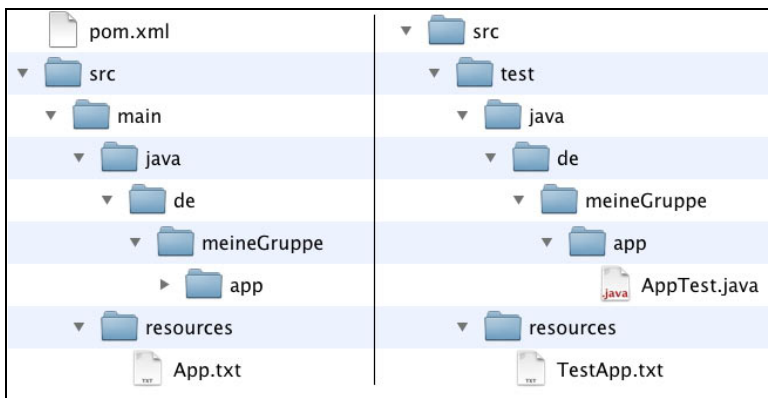


Abbildung 3.4: Struktur eines einfachen Projektes mit Ressourcen

Nach dem Kompilieren und Packen mittels `mvn package` schauen wir uns an, was passiert ist.

Zunächst sehen wir die zwei Dateien im Filesystem unter `/target/classes/App.txt` und `/target/test-classes/TestApp.txt`. In der JAR-Datei finden wir jedoch nur die Datei `App.txt`. Das ist logisch, da wir ja nur das reguläre Artefakt gebaut haben und dort keine Testklassen, aber auch keine Testressourcen enthalten sind. Wenn wir das Test-Artefakt mittels `mvn jar:test-jar` bauen, finden wir dort auch die Datei `TestApp.txt`.

Neben den erwarteten Dateien tauchen noch drei weitere Dateien in der erzeugten JAR-Datei auf. `MANIFEST.MF`, `pom.xml` sowie `pom.properties`. Die Manifest-Datei ist dabei nichts Überraschendes, sie wird von den meisten Packaging-Programmen selbstständig erzeugt, wenn sie nicht vorhanden ist, aber was sollen die zwei POM-Dateien?

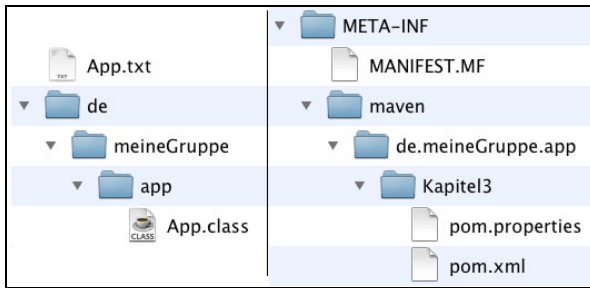


Abbildung 3.5: JAR-Datei eines einfachen Projektes

Die Erklärung ist recht einfach: Maven legt dort eine Kopie der POM-Datei ab, die zur Erzeugung dieses Artefaktes geführt hat. Sinn des Ganzen ist es, jederzeit nachvollziehen zu können, mit welchen Einstellungen und Abhängigkeiten das Artefakt entstanden ist. In der Properties-Datei ist dagegen die Gruppe, der Name und die Version des Artefaktes nochmals festgehalten, so dass diese einfach vom Programm ausgelesen werden können. Dadurch ist es möglich, z.B. in einer About-Box die eingesetzten Libraries und Versionen anzuzeigen.

Nun haben wir die Ressourcen im Artefakt, aber wie können wir sie von unserem Programm aus verwenden?

Da die Ressourcen im Classpath liegen, ist der Zugriff sehr einfach durch

```
InputStream in = getClass().getResourceAsStream("/App.txt");
```

möglich. Natürlich können diese Ressourcen nur gelesen werden.

Seine Stärken kann Maven beim Filtern von Ressourcen ausspielen. Dabei werden die Ressourcendateien nicht einfach nur kopiert, sondern es finden Ersetzungen innerhalb der Ressourcendateien statt. Ersetzt werden dabei alle Stellen, die in der Art `${<property name>}` aussehen, wobei »property name« den Namen einer Property angibt, deren Inhalt an diese Stelle geschrieben werden soll.

Für die Quelle der Properties kennt Maven mehrere Möglichkeiten. Da ist zunächst der Inhalt der POM-Datei und der Maven-Settings. Hier kann auf jedes Element zugegriffen werden, in dem als Property-Name die Tags verwendet werden, die zu dem Eintrag führen. Beispiele hierfür sind »project.name« für den Projektnamen oder »project.version« für die Projektversion. Zusätzlich gibt es einige Properties, die Maven von sich aus zur Verfügung stellt.

Um das Ganze einmal auszuprobieren, erzeugen wir einfach eine Datei `/src/main/resources/App.properties` mit folgendem Inhalt:

```
application.name=${project.name}
application.version=${project.version}
application.src=${basedir}
```

und ändern unser POM ein wenig ab:

```
...
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
...
```

Nach Aufruf von **mvn process-resources** erhalten wir unter `/target/classes/App.properties` folgenden Inhalt:

```
application.name=Maven Quick Start Archetype
application.version=1.0-SNAPSHOT
application.src=D:\Projekte\beispiele\kap03\Kapite13
```

Wobei der Eintrag unter »application.src« natürlich vom Pfad im eigenen Filesystem abhängt.

Eine andere Möglichkeit ist die Verwendung einer eigenen Konfigurationsdatei. Auch hierzu legen wir eine neue Datei `/src/main/filters/filter.properties` mit dem Inhalt `mein.filter=Foo` an.

Hier ist ebenfalls eine kleine Änderung am POM nötig:

```
...
<build>
  <filters>
    <filter>src/main/filters/filter.properties</filter>
  </filters>
  ...
</build>
```

In der Datei `App.properties`, natürlich diejenige im Ressourcen-Verzeichnis, fügen wir noch die Zeile `message=${mein.filter}` ein, damit der Filter auch etwas zu ersetzen hat, und rufen Maven wieder auf.

Nun sehen wir in der erzeugten Datei die neue Zeile `message=Foo`.

Die nächste Möglichkeit ist die Angabe von Properties in der POM-Datei. Hier kann ein Bereich der Form

```
<project>
  ...
  <properties>
    <mein.wert.im.filter>Hallo</mein.wert.im.filter>
```

```

    </properties>
    ...
</project>

```

eingefügt werden, wodurch die Property mit dem Namen `mein.wert.im.filter` zur Verfügung steht. Sinnvoll ist dieses Vorgehen insbesondere bei Multi-Module-Projekten, wo die Property in einem übergeordneten Projekt definiert wird oder bei der Verwendung eines Firmen-POMs.

Die letzte Möglichkeit, eine Property zu setzen, ist über die Kommandozeile. Der Aufruf

```
mvn ... -Dcommand.value=Wert
```

bringt die Property `command.value` mit dem Inhalt »Wert« ins Spiel. Beachten sollte man, dass zur Angabe von Leerzeichen im Text der ganze Parameter in Anführungszeichen zu setzen ist.

Ein Problem bleibt. Wenn Binärdaten wie z. B. Bilder zufällig die passende Byte-Kombination enthalten, kann das Ergebnis sehr merkwürdig aussehen und die Dateien unverwendbar machen. Auch kann es vorkommen, dass Sie bestimmte Textdateien von dieser Filterung ausnehmen möchten.

Hierzu ist eine weitere Modifikation der POM-Datei nötig.

```

...
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
    <excludes>
      <exclude>images/**</exclude>
    </excludes>
  </resource>
  <resource>
    <directory>src/main/resources</directory>
    <includes>
      <include>images/**</include>
    </includes>
  </resource>
</resources>
...

```

In diesem Beispiel werden alle Dateien gefiltert mit Ausnahme der Dateien, die im Unterverzeichnis `images` liegen. Hier können natürlich auch andere Such-Pattern wie z. B. `**/*.jpg` angegeben werden.

## 3.5 Weitere Schritte

Maven kann natürlich noch viel mehr, als einfach nur Projekte zu kompilieren. So z.B.:

- Erzeugen einer Webseite: **mvn site**
- Bereinigen des Projektes: **mvn clean**
- Erzeugen von Projektdateien für eine IDE: **mvn idea:idea** oder **mvn eclipse:eclipse** oder ...

... um nur einige zu nennen, doch dazu später mehr.

## 3.6 Einsatz von Plugins

Maven hat bereits eine Reihe von Plugins vorkonfiguriert, so dass Sie in kleineren Projekten nur selten in die Verlegenheit kommen, andere Plugins einzusetzen. Doch keine Regel ohne Ausnahme, und so soll hier kurz gezeigt werden, wie man Maven dazu bringt, nicht nur Java 1.4-Sourcen zu verarbeiten, sondern die erweiterten Möglichkeiten von Java 5 zu verwenden.

Maven benutzt zum Kompilieren das Plugin »maven-compiler-plugin« aus der Gruppe »org.apache.maven.plugins«. Um dieses Plugin auf Java 5 einzustellen muss die Default-Konfiguration überschrieben werden.

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

Hier wird das Plugin der Version 2.0 verwendet. Lässt man den Versionseintrag weg, so sucht sich Maven die aktuellste Version, die verfügbar ist. Lässt man die Angabe zur Group-ID weg, versucht Maven das Plugin in den Gruppen »org.apache.maven.plugins« und »org.codehaus.mojo« zu finden. Diese Default-Gruppen können durch den Eintrag `pluginGroups` in den Settings erweitert werden.

## 3.7 Abhängigkeiten

Ein Vorteil von Maven ist die einfache Verwaltung von Abhängigkeiten. Sie brauchen in einem Projekt nur die direkt benötigten Libraries angeben, ohne sich um deren Abhängigkeiten zu kümmern. Um dies zu erreichen, benutzt Maven eine Art Vererbungshierarchie, d.h. für jede Library im Repository sind die direkten Abhängigkeiten in einer POM-Datei angegeben, so dass Maven die indirekten Abhängigkeiten selbst auflösen kann.

Um die Abhängigkeiten einzustellen, hat unsere POM-Datei eine eigene Sektion: `dependencies`. Ein typischer Eintrag sieht folgendermaßen aus:

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

Die `groupId` gibt dabei einen Sammelbegriff an, z. B. eine Organisation oder eine Firma, unter der alle Libraries dieser Gruppe zu finden sind. Das Ganze wird im Repository wie eine Package-Bezeichnung gehandelt, d.h. es werden die entsprechenden Unterverzeichnisse verwendet. Üblicherweise finden Sie hier Einträge wie »org.apache« oder »de.meineFirma«.

Die `artifactId` gibt nun die genaue Library an, die verwendet werden soll, wohingegen mit dem Eintrag `version` noch die Versionsnummer bezeichnet wird.

Als letzte Angabe kann nun noch ein `scope` folgen, der Maven mitteilt, für welchen Build-Schritt die Library von Bedeutung ist. Lässt man diese Angabe weg, so geht Maven davon aus, dass die Library für den Bau des Hauptartefaktes benötigt wird.

Unser Beispieleintrag sagt nun, dass wir im Repository die Library `/junit/junit/3.8.1/junit-3.8.1.jar` für das Ausführen unserer Tests benötigen. Wenn wir uns den Eintrag in unserem lokalen Repository ansehen, finden wir im gleichen Verzeichnis die Datei `junit-3.8.1.pom`, in welcher noch mal die `groupId`, `artifactId` und `version` angegeben sind. Diese Datei ist nichts weiter als die POM-Datei, mit der diese Library erstellt wurde (kann natürlich auch nachträglich angelegt worden sein, wenn es kein Maven-Projekt war). Weiterhin fallen noch zwei weitere Dateien mit der Endung `sha1` auf. Diese Dateien sind wichtig, wenn Sie mit öffentlichen Repositories arbeiten. Sie enthalten Signaturen, mit deren Hilfe Maven überprüfen kann, ob die Library und die POM-Datei unverändert sind und nicht durch manipulierte Dateien ersetzt wurden.

Schauen wir uns jetzt einmal die erzeugten Dateien unseres Projektes an. Dazu gehen wir im Repository in das Verzeichnis `/de/meineGruppe/app/Kapitel3/1.0-SNAPSHOT`.

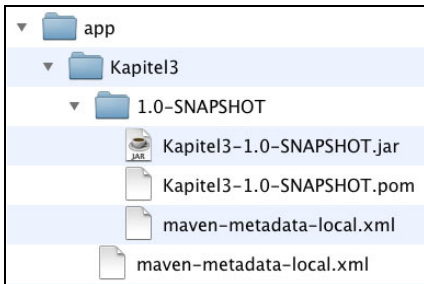


Abbildung 3.6: Repository – Eintrag eines einfachen Projektes

Hier finden wird die erzeugte JAR-Datei sowie eine Kopie unserer POM-Datei. Wenn wir diese mit der POM-Datei aus unserem Projekt vergleichen, fällt uns auf, dass es keine Eins-zu-eins-Kopie ist, sondern dass Maven einige kleinere Änderungen vorgenommen hat. Im Wesentlichen ist es der folgende Eintrag:

```
...
<distributionManagement>
  <status>deployed</status>
</distributionManagement>
...
```

Dieser besagt nichts weiter, als dass die Applikation ins Repository deployed wurde.

Neben der JAR-Datei und unserem POM finden wir noch eine weitere Datei, `maven-metadata-local.xml`. In dieser Datei merkt sich Maven den Status der Applikation. Gehen wir ein Verzeichnis weiter hinauf, so finden wir diese Datei nochmals. Hier merkt sich Maven die bekannten Versionen unserer Applikation. Dies ist derzeit nur eine. Um das auszuprobieren, gehen wir in unser Projekt und ändern in der POM-Datei die Version von »1.0-SNAPSHOT« auf »1.1-SNAPSHOT«. Nach einem Aufruf von `mvn install` finden wir neben dem neuen Verzeichnis im Repository einen weiteren Eintrag in der Datei `maven-metadata-local.xml`.

Was ist nun der tiefere Sinn dieser Datei? Hier merkt sich Maven, welche Versionen eines Artefaktes bekannt sind, welche davon die aktuellste ist, und wichtiger, welche das aktuellste Release ist. Eingesetzt werden diese Informationen z.B. wenn Maven die aktuelle Version eines Plugins sucht.

Dies soll für den Einstieg erst mal reichen. Maven bietet noch viele weitere Möglichkeiten, auf die aber erst später eingegangen werden soll.