Maven

Stephan Karrer

Motivation: Das Bauen einer Anwendung

Erzeugung eines deploybaren Artefakts bzw. Anwendung aus Programmcode, Bibliotheken und sonstigen Ressourcen.

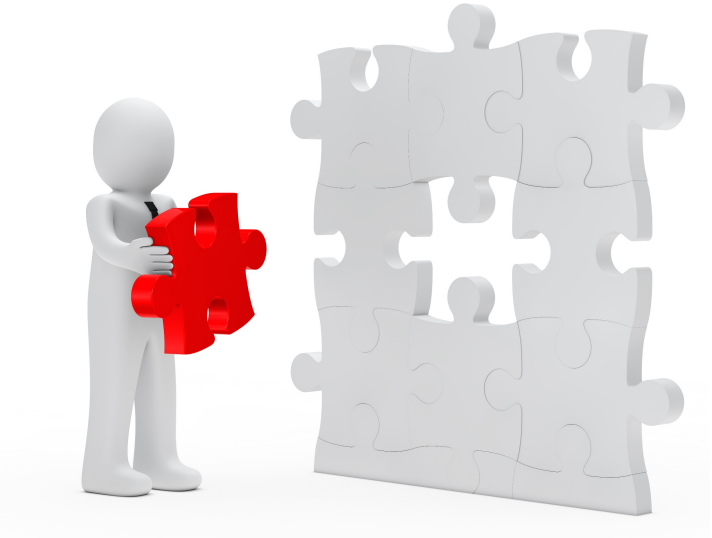


Bild von d3images auf Freepik

Problematik:
Komplexität aufgrund einer Vielzahl von Abhängigkeiten (Bibliotheken, Versionen, IDE, ...)

Lösung:
Automatisierung des Build-Vorgangs

Build-Automatisierung

- Organisation von Bibliotheken
- Kompilierung von Source-Code in Binär-Code
- Ausführung von Tests
- Paketierung des Binär-Codes und andere Dateien in ein deploybares Artefakt bzw. Produkt
- In der Java-Welt stehen dafür im wesentlichen 3 Werkzeuge zur Verfügung
  - ANT (veraltet)
  - Maven
  - Gradle
- Aktuell (JetBrains Developer Survey 2023) populärstes Build-Werkzeug:
  - Maven 74%
  - Gradle 46%
  - ANT 6%

Was ist Maven

- Apache Maven (aktuelle Version 3) ist das beste Beispiel für ein Konfigurations-basiertes Build-Management-Tool.
- Von der offiziellen Webseite (https://maven.apache.org/what-is-maven.html):
  - Making the build process easy
  - Providing a uniform build system
  - Providing quality project information
  - Encouraging better development practices
- Es folgt der Philosophie, dass Projekte in der Regel sehr ähnlich aufgebaut sind und deshalb auch der Build-Vorgang immer ähnlich ablaufen wird.
- Maven definiert zu diesem Zweck einige Standards und einen Build-Lifecycle, der für diesen Standard ausgelegt ist.
- Man muss nur das konfigurieren, was vom Standard abweicht bzw. projektspezifisch ist (Convention over Configuration).
- Maven ist eigentlich über die Kommandozeile zu nutzen, ist aber heute zusätzlich in den üblichen Entwicklungsumgebungen integriert.

Installation

- Herunterladen und entpacken.
- Umgebungsvariablen setzen:
  - JAVA_HOME, muss auf ein JDK verweisen!
    (Maven 3.3+ requires JDK 1.7 or above to execute)
  - PATH setzen
  - optional: MAVEN_HOME setzen
- Test: `mvn -version`
- Entwicklungsumgebungen (Eclipse, IntelliJ, …) bringen eigene Variante mit.

.

Begriffe – Teil 1

- **Project**:
  Das zu bauende Software-Projekt

- **Project Object Model (POM)**:
  Die Metadaten, welche Maven benötigt, um das Projekt zu bauen. Üblicherweise eine "pom.xml" Datei an oberster Stelle im Projektverzeichnis.

- **Artifact**:
  Etwas, was durch den Build-Prozess erzeugt oder konsumiert wird. Beispielsweise produziert der Build-Prozess ein JAR Artefakt als Output, gleichzeitig benötigt die Kompilierung auch Drittbibliotheken in Form anderer JAR Artefakte.

- **GAV**:
  Jedes Artefakt ist primär durch sogenannte GAV Koordinaten (GroupId, ArtifactId, Version) eindeutig identifiziert
  - **GroupId**: ein eindeutiger Kennzeichner für eine Gruppe/Unternehmen, z.B. "org.springframework"
  - **ArtifactId**: eine eindeutiger Kennzeichner für ein Artefakt einer Gruppe, z.B. "spring-core"
  - **Version**: ein eindeutiger Kennzeichner für die Version eines Artefakt, z.B. "5.3.22".

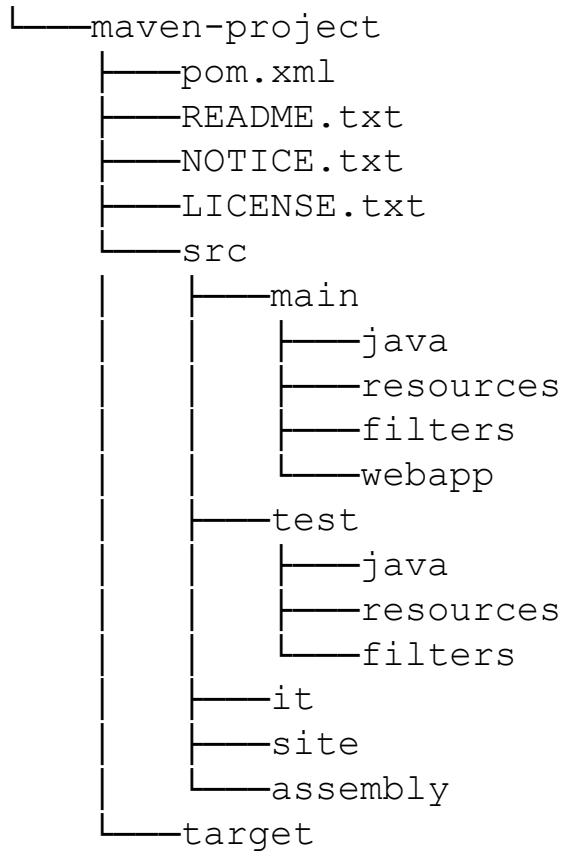
Begriffe – Teil 2

- **Dependency**:
  Eine Bibliothek, die zum Bauen und Ausführen eines Java Projekts benötigt wird, typischerweise Artefakte anderer Projekte

- **Plugin**:
  Eine im Kontext von Maven ausführbare Funktion, meistens in Java programmiert. Plugins definieren Goals und nutzen die Metadaten der POM, um ihre Aufgabe auszuführen (z.B. Kompilieren, Tests ausführen, JAR Datei erzeugen)

- **Repository**: Ein Ablageort für Artefakte. Kann lokal auf dem eigenen PC, im Intranet oder im Internet existieren.

Standard-Verzeichnisstruktur



- maven-project/pom.xml — Projektkonfiguration
- maven-project/LICENSE.txt — Lizenz-Bedingungen
- maven-project/NOTICE.txt — Infos zu Fremdresourcen
- maven-project/src/main — Sourcecode
- maven-project/src/test — Tests
- maven-project/src/it — Integrationstests
- maven-project/src/site — Generierte Dokumentation
- maven-project/src/assembly — Deskriptoren für Binärcode
- maven-project/target — Generierter Code

- Abänderung ist unüblich, kann aber konfiguriert werden.
- Aktuell nicht benötigte Verzeichnisse können auch fehlen.

POM (Project Object Model): Die projektspezifische Konfigurationsdatei

```
<?xml version="1.0" encoding="UTF-8"?>
<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>sk.train</groupId>
    <artifactId>java11_basis</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

- Für eine minimale Projektkonfiguration ist nur erforderlich:
  - project root — Sauberes XML
  - modelVersion — aktuell Version 4.0.0
  - groupId — eindeutiger Identifier der Organisation/Projektgruppe (empfohlen: Java Package Naming)
  - artifactId — eindeutiger Identifier für das Projekt innerhalb der „groupId“ (gibt jar-Namen vor)
  - version — Versionsnummer (Empfehlung Semantic Versioning 1.0.0: https://semver.org/spec/v1.0.0.html)

- group, artifact and version (GAV) sind die sogenannten Maven-Koordinaten und identifizieren das Maven-Artefakt eindeutig.
  Folgende GAV-Schreibweise ist üblich: groupId:artifactId:version
  Hier z.B: ***sk.train:java11_basis:1.0-SNAPSHOT***

POM (Project Object Model): Struktur und Hierarchie

- Jedes Projekt ohne expliziten Parent erbt implizit von einer sog. Super-POM.

  (siehe z.B. für Version 3.9.9: https://maven.apache.org/ref/3.9.9/maven-model-builder/super-pom.html)

- Diese „Super-POM“ ist fest eingestellt und bildet die Spitze der Vererbungshierarchie bzgl. der Projektkonfiguration.

- Die POM-Struktur (XML) ist selbstverständlich ebenfalls festgelegt.
  (siehe: https://maven.apache.org/ref/3.9.9/maven-model/maven.html)

- ***`mvn help:effective-pom`*** zeigt uns die Effektive POM, die Maven tatsächlich benutzt.

POM Anpassungen

```
<?xml version="1.0" encoding="UTF-8"?>
<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>sk.train</groupId>
    <artifactId>java11_basis</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>java11_basis</name>
    <description>sample project</description>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

- Wir erben von unserer Eltern-POM , default „Maven Super-POM“, und via Voreinstellungen, können aber durchaus Anpassungen vornehmen (Convention over Configuration).

Arbeitsweise

- Verschiedene Build-Vorgänge (Build Lifecycle) sind durch Maven definiert:
  - Clean — Löscht alle Dateien, die bei einem vorherigen Build-Vorgang erzeugt wurden.
  - Site — Generiert die Projektdokumentation und stellt diese auf dem Zielserver bereit (Deployment).
  - Default — Der Standard-Vorgang, wenn keiner der anderen beiden gewählt ist.
- Jeder Build-Vorgang besteht aus einzelnen Phasen.
- Abhängig vom Paketierungsformat sind Arbeitsschritte (Goals) an die Phasen gebunden, sprich: werden innerhalb der Phase ausgeführt.
- Sogenannte Plugins realisieren ein oder mehrere Goals, die Goals werden realisiert durch MOJOs (MOJO ist ein Wortspiel für POJO (Plain-old-Java-object)).
- Es lassen sich auch neue Plugins schreiben und damit neue Goals hinzufügen.
- Siehe auch: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
  https://maven.apache.org/plugins/index.html

Phasen des Default-Lifecycle im Überblick

Build-Phase	Relevant Plugins	Description
validate		validate the project is correct and all necessary information is available
compile	compiler	compile the Java source code of the project
test	surfire (for Junit -tests)	test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
package	*different, chosen by format*	take the compiled code and package it in its distributable format, such as a JAR
integration-test	failsafe (for JUnit-tests)	takes the packaged result and executes additional tests, which require the packaging, in an appropriate environment
verify	*different*	run any checks on results of integration tests to ensure quality criteria are met
install	install	install the package into the local repository, for use as a dependency in other projects locally
deploy	deploy	done in the build environment, copies the final package to the remote repository for sharing with other developers and projects

- Die Zuordnung von Plugins zu Phasen ist nicht 1:1, insbesondere können Goals verschiedenen Phasen, auch mehrfach zugeordnet sein (z.B. kompilieren des Source-Codes und des Test-Codes)

Packaging

- Entscheidet, welche Goals standardmäßig im Build-Lifecycle abgearbeitet werden.
- Mögliche Packaging-Formate:
  - jar (Standard)
  - war Web-Archiv (Web-Applikation)
  - ear Enterprise-Archiv (Web-Applikation + Backend-Code)
  - rar Resource-Archiv
  - par OSGI-Archiv
  - ejb3 EJB3-Archiv
  - pom nur Default-Vorgaben
  - maven-plugin Plugin
  - eigene Formate ...

Zuordnung der Plugin-Goals zum Default-Lifecycle abhängig vom Package-Format

Phase	plugin:goal
`process-resources`	`resources:resources`
`compile`	`compiler:compile`
`process-test-resources`	`resources:testResources`
`test-compile`	`compiler:testCompile`
`test`	`surefire:test`
`package`	`ejb:ejb` *or* `ejb3:ejb3` *or* `jar:jar` *or* `par:par` *or* `rar:rar` *or* `war:war`
`install`	`install:install`
`deploy`	`deploy:deploy`

- Für die üblichen Paketierungsformate ist die Zuordnung identisch.
- siehe auch: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Phasen und Kommandos

- Wird eine Phase ausgeführt, so werden automatisch alle davor liegende Phasen gemäß obiger Reihenfolge ausgeführt und damit natürlich alle relevanten Goals der jeweiligen Phase:
  - ***`mvn clean`*** Aufruf des clean-Cycle
  - ***`mvn verify`*** Aufruf der verify-Phase und damit aller vorhergehenden Phasen
  - ***`mvn clean install`*** Aufruf des clean-Cycle und anschließender install-Phase
  - ***`mvn compiler:compile`*** Aufruf eines Goals innerhalb eines Plugins ohne vorhergehende Schritte

- Mit Hilfe des Help-Plugins funktioniert üblicherweise für alle Plugins:
  - ***`mvn help:describe -Dcmd=compile`*** Listet die zugehörigen Goals und Plugins zu einer Phase, hier compile-Phase
    (generell mvn help:describe -Dcmd=<phase>)
  - ***`mvn surefire:help`*** Liefert die Goals des Plugins, hier surefire-Plugin
    (generell mvn <plugin>:help)
  - ***`mvn compiler:help -Ddetail=true -Dgoal=compile`***
    Liefert eine Auflistung aller möglichen Parameter des Plugins bzgl. des Goals, hier compiler-Plugin
    (generell mvn <plugin>:help -Ddetail=true -Dgoal=<goal>)

Explizite Plugin-Konfiguration

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.10.1</version>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

- Durch einen Eintrag in der POM (für default Lifecycle in der build-Sektion) kann die Standardkonfiguration angepasst werden.
- Beispiel: aktuelle Versionen der Plugins verwenden.
- Die Konfigurationsmöglichkeiten eines Plugins sind der „hoffentlich brauchbaren“ Dokumentation des Plugins zu entnehmen.

Plugins einbinden

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.1.0</version>
                <configuration>
                    <mainClass>sk.train.Main</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

- Es gibt sogenannte Core Plugins und eine Vielzahl Optionale, siehe
  https://maven.apache.org/plugins/index.html

- Es können auch zusätzliche Plugins eingebunden werden, z.B. das exec-Plugin von MojoHaus:
  mvn exec:java führt das konfigurierte Java-Programm aus

Plugin-Goal einer Phase zuordnen

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.1.0</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>java</goal>
                        </goals>
                        <phase>verify</phase>
                    </execution>
                </executions>
                <configuration>
                    <mainClass>sk.train.Main</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Maven Properties (value placeholders)

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>First_Maven</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <finalName>${project.groupId}-${project.artifactId}</finalName>
    </build>
    ...
</project>
```

- Properties können aus verschiedenen Quellen stammen:
  - Umgebungsvariablen: ***${env.*}***
  - Einträge in der POM: ***${project.*}*** (*Ältere Zugriffsvarianten* ***${pom.*}*** *oder* ***${*}*** *sind deprecated)*
  - Einträge in der settings.xml: ***${settings.*}***
  - System Properties: ***${java.*}***
  - Eigene Property-Definitionen: ***${*}***

- ***mvn help:system*** — Listet die System- und Umgebungsvariablen (Properties) auf
- ***mvn -Dfile.encoding=UTF-8*** — System- bzw. Maven-Parameter können beim Aufruf gesetzt werden

- Die effektive POM zeigt die aufgelösten Referenzen

Verwendung von Properties

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <properties>
        <!--    Standard-Properties    -->
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--    Eigene Properties    -->
        <jupiter-version>5.9.2</jupiter-version>
     </properties>

     <dependencies>
          <!--  testing dependencies  -->
          <dependency>
               <groupId>org.junit.jupiter</groupId>
               <artifactId>junit-jupiter</artifactId>
               <version>${jupiter-version}</version>
               <scope>test</scope>
          </dependency>
          ...
  </dependencies>
...
</project>
```

Ausgaben des Build-Cycles

- Einige Ausgaben werden direkt auf die Standardausgabe geschrieben, ansonsten wird SLF4J als Logging-Fassade benutzt.

- Logging erfolgt standardmäßig auf die Standardausgabe und nur für die Log-Level ***info***, ***warning***, und ***error*** . Dies kann durch Kommando-Optionen angepasst werden oder aber durch direkte Konfiguration der verwendeten Logging-Implementierung (default: <*MAVEN_HOME*>/conf/logging/simplelogger.properties). siehe z.B: https://www.baeldung.com/maven-logging

```
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< sk.train:Maven_Basis_Java11 >---------------------
[INFO] Building Maven_Basis_Java11 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.2.0:clean (default-clean) @ Maven_Basis_Java11 ---
[INFO] Deleting E:\stephan\workspaces\intellij2\Maven_Basis_Java11\target
[INFO]
[INFO] --- maven-resources-plugin:3.3.0:resources (default-resources) @ Maven_Basis_Java11 ---
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.10.1:compile (default-compile) @ Maven_Basis_Java11 ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to E:\stephan\workspaces\intellij2\Maven_Basis_Java11\target\classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.953 s
[INFO] Finished at: 2023-01-25T22:40:13+01:00
[INFO] ------------------------------------------------------------------------
```

Ausgaben des Build-Cycles

- Standardmäßig werden bei Laufzeitfehlern im Build-Vorgang nur kurze Fehlermeldungen ausgegeben.
- Um den vollen Stacktrace bei Laufzeitfehlern anzuzeigen, steht die Option ***-e*** oder ***–errors*** zur Verfügung
  - Bsp: ***mvn clean -e compile***
- Logging erfolgt standardmäßig auf die Standardausgabe und nur für die Log-Level ***info***, ***warning***, und ***error*** .
- Dies kann durch Kommando-Optionen angepasst werden
  - Bsp: ***mvn X clean compile*** setzt Log-Level auf „debug“
- oder aber durch direkte Konfiguration der verwendeten Logging-Implementierung
  (default: <*MAVEN_HOME*>/conf/logging/simplelogger.properties).
  siehe z.B: https://www.baeldung.com/maven-logging

Das Help-Plugin

- ***mvn help*** Listet die Optionen auf. Ist selbst durch das Help-Plugin realisiert.
- Help-Plugin: Goals und Parameter
  - ***mvn help:help*** Liefert Goals des Plugins
    (generell mvn <plugin>:help Goals des Plugins)
  - ***mvn help:system*** Liefert System-Properties und Umgebungsvariablen
  - ***mvn help:effective-pom*** Zeigt effektive POM
  - ***mvn help:effective-settings*** Zeigt effektive Voreinstellungen
  - ***mvn help:…*** Siehe:
    https://maven.apache.org/plugins/maven-help-plugin/

Das Compiler-Plugin

Goal	Description
compiler:compile	Compiles application sources
compiler:help	Display help information on maven-compiler-plugin. Call `mvn compiler:help -Ddetail=true -Dgoal=<goal-name>` to display parameter details.
compiler:testCompile	Compiles application test sources.

- Die Goals und die Zuordnung des Plugins kann ausgegeben werden:
  - ***mvn compiler:help*** — Liefert obige Information
    (generell mvn <plugin>:help Goals des Plugins)
  - ***mvn help:describe -Dcmd=compile*** — Plugin-Zuordnung zur Phase
    (generell mvn help:describe Dcmd=<phase>)

- ***mvn compiler:help -Ddetail=true -Dgoal=compile***
  Liefert eine Auflistung aller möglichen Parameter

Das Compiler-Plugin: Konfiguration

- Wenn der Build mit einer anderen Java-Version erfolgen soll als der, die Maven selbst benutzt, muss „fork“ gesetzt sein.
- Zur Parametrisierung siehe: https://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
     ...
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <JAVA_HOME_8>C:\Program Files\Java\jdk1.8.0_291</JAVA_HOME_8>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.10.1</version>
                <configuration>
                    <verbose>true</verbose>
                    <fork>true</fork>
                    <executable>${JAVA_HOME_8}/bin/javac</executable>
                    <compilerVersion>1.8</compilerVersion>
                    <meminitial>128m</meminitial>
                    <maxmem>512m</maxmem>
                    <verbose>true</verbose>
                    <compilerArgs>
                        <arg>-Xlint:all</arg>
                    </compilerArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>
     ...
</project>
```

Das Compiler-Plugin: JPMS-Parameter

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
     ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.10.1</version>
                <configuration>
                    <verbose>true</verbose>
                    <compilerArgs>
                        <arg>--add-exports</arg>
                        <arg>java.base/jdk.internal.misc=ALL-UNNAMED</arg>
                    </compilerArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>
     ...
</project>
```

- Zur Übersteuerung der Restriktionen des Java Platform Module System (JPMS, ab Java 9) existieren die entsprechenden Compiler-Argumente (https://maven.apache.org/plugins/maven-compiler-plugin/examples/jpms_args.html#jpms-args)
  - --upgrade-module-path
  - --add-exports
  - --add-reads
  - --add-modules
  - --limit-modules
  - --patch-module

Junit-Tests: Das Surfire-Plugin

- Goals und Parameter
  - ***mvn surefire:help*** Liefert Goals des Plugins (nur „test“ neben „help“)
  - ***mvn surefire:help -Ddetail=true -Dgoal=test*** Auflistung aller Parameter

- Testausführung
  - Die Tests werden via eigenen Classloader ausgeführt bzw. mit entsprechenden Optionen auch mit eigener Java-Version
  - Es werden kurze Berichte zum Testergebnis im Text- und XML-Format in ${basedir}/target/surefire-reports/TEST-*.xml.
    Via Surefire Report Plugin kann auch HTML-Format erzeugt werden.
  - Es werden sowohl JUnit4, JUnit5 als auch TestNG als Test-Frameworks unterstützt.
  - schlägt ein Test fehl, führt das im Standardfall zum Abbruch des Build
  - Inklusion bzw. Ausschluß von Tests kann anhand der Namensregeln für Tests erfolgen (https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html)
  - ***mvn test -Dtest="TheFirstUnitTest"*** Führt nur einen einzelnen Test aus
  - ***mvn -DskipTests verify*** Die Tests können auch übersprungen werden

    siehe z.B: https://www.baeldung.com/maven-skipping-tests#bd-introduction

Integrations-Tests: Das Failsafe-Plugin

- Der Maven Lifecycle hat 4 Detail-Phasen für die Ausführung der Integrationstest:
  - pre-integration-test — Aufsetzen der Testumgebung.
  - integration-test — Ausführen der Integrations-Tests.
  - post-integration-test — Bereinigen der Testumgebung.
  - verify — Prüfen der Testergebnisse.

- maven-failsafe-plugin ist im Gegensatz für reine Integrations-Tests gedacht und setzt üblicherweise Paketierung voraus.
  Kommt potentiell für „integration-test“ und „verify“ zum Einsatz.

- greift intern auf das Surefire-Plugin zurück und unterstützt von daher die gleichen Test-Frameworks.

- Konfiguration erfolgt analog zum Surefire-Plugin
  siehe: https://maven.apache.org/surefire/maven-failsafe-plugin/

- Testausführung
  - Wie beim normalen Test werden kurze Berichte zum Testergebnis im Text- und XML-Format in ${basedir}/target/failsafe-reports/TEST-*.xml erzeugt.
    Via Surefire Report Plugin kann auch HTML-Format erzeugt werden.
  - Aber entkoppelt den Build vom Testergebnis:
    Ein fehlgeschlagener Test in der Integration-Test-Phase führt nicht direkt zum Abbruch, sondern erlaubt die Ausführung der (internen) Phase „post-integration-test“ zum Aufräumen.
  - Deshalb sollte der Aufruf via ***mvn verify*** erfolgen, was ja die Phase „integration-test“ inkludiert.

Einbindung des Failsafe-Plugin

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
     ...
    <build>
        <plugins>
            <plugin>
               <artifactId>maven-failsafe-plugin</artifactId>
               <version>3.1.2</version>
               <executions>
                  <execution>
                    <goals>
                         <goal>integration-test</goal>
                         <goal>verify</goal>
                    </goals>
                    <configuration>
                         ...
                    </configuration>
                  </execution>
               </executions>
          </plugin>
        </plugins>
    </build>
     ...
</project>
```

- Das Failsafe-Plugin ist nicht automatisch dabei (via Super-POM) sondern muss explizit eingebunden werden.
- Beispiel zur Verwendung: https://www.torsten-horn.de/techdocs/maven.htm

Das Resource-Plugin

- Das Resources-Plugin kopiert die Resourcen aus dem Resource-Ordner in ein Output-Verzeichnis, so dass diese später mit eingepackt werden können, , z.B. in das jar.

- Es gibt neben „help“ 3 Goals:
  - resources:resources — Kopiert die Resourcen des Main-Source-Codes. Wird automatisch in der Phase „process-resources“ ausgeführt
  - resources:testResources — Kopiert die Resourcen des Test-Source-Codes. Wird automatisch in der Phase „process-test-resources“ ausgeführt
  - resources:copy-resources — Kopiert als Parameter anzugebende Resourcen in ein ebenfalls anzugebendes Zielverzeichnis.

- Somit sind die Resourcen, z.B. Konfig-Dateien, für die normale Anwendung und die Tests getrennt

- Interessant ist hierbei die Möglichkeit der sog. Resourcen-Filterung anhand von Properties:
  - Ein einfaches Beispiel in der offiziellen Dokumentation: https://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html
  - Ein komplexeres Beispiel mit Profilen: https://www.sonatype.com/maven-complete-reference/properties-and-resource-filtering#mavenref9-3

Dependency-Management

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.9.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
...
</project>
```

- Eine der wichtigsten Eigenschaften von Maven: Dependency-Management
- Anhand der GAV (groupId, artifactId and version) der benötigten Artefakte wird versucht, diese bereit zu stellen.
- Als Quellen werden in dieser Reihenfolge benutzt:
  - Andere Projekte im selben Maven Build (Maven reactor)
  - Lokales Repository (default: %HOME%\.m2\repository
  - Globales Repository (default: Maven Central)
- Statt globale Repositories im Web direkt zu verwenden, werden oft „Mirrors“ konfiguriert.

Dependency-Management: Auflösungs-Mechanismus

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.9.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
...
</project>
```

1) Benötigtes Artefakt: **org.junit.jupiter:junit-jupiter:5.9.2:jar**

2) Intern wird daraus folgender Verzeichnis-Pfad:
   org/junit/jupiter/junit-jupiter/5.11.0/junit-jupiter-5.11.0.jar

3) Sofern das Artefakt nicht ein anderes Projekt des Maven Build ist, wird anhand dieses Pfads gesucht:
   1) Lokales Repository (default: <MAVEN_HOME>\.m2\repository)
   2) Globale Repositories (default: Maven Central) und/oder Mirror

Scope der Dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.9.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
...
</project>
```

- Nicht jede Dependency wird für jeden Arbeitsschritt im Build benötigt.
- Folgende Scopes sind definiert:
  - compile (default) — in allen Phasen verfügbar und mit deployed (propagated)
  - provided — nur beim Compile und Test verfügbar, zur Laufzeit in der Zielumgebung verfügbar (nicht transitiv)
  - runtime — nur zur Laufzeit bereit zu stellen, d.h. verfügbar bei Test und Runtime, aber nicht bei Compile
  - test — nur bei Test verfügbar (nicht transitiv)
  - system — wird durch das aktuelle System bereit gestellt und nicht in einem Repository gesucht
  - import — Spezialfall bei Übernahme aus Parent-POM
- siehe auch: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

Beispiel Scope system

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <dependency>
          <groupId>mylib</groupId>
          <artifactId>mylib</artifactId>
          <scope>system</scope>
          <version>1.0</version>
          <systemPath>${basedir}\libs\mylibrary.jar</systemPath>
</dependency>
...
</project>
```

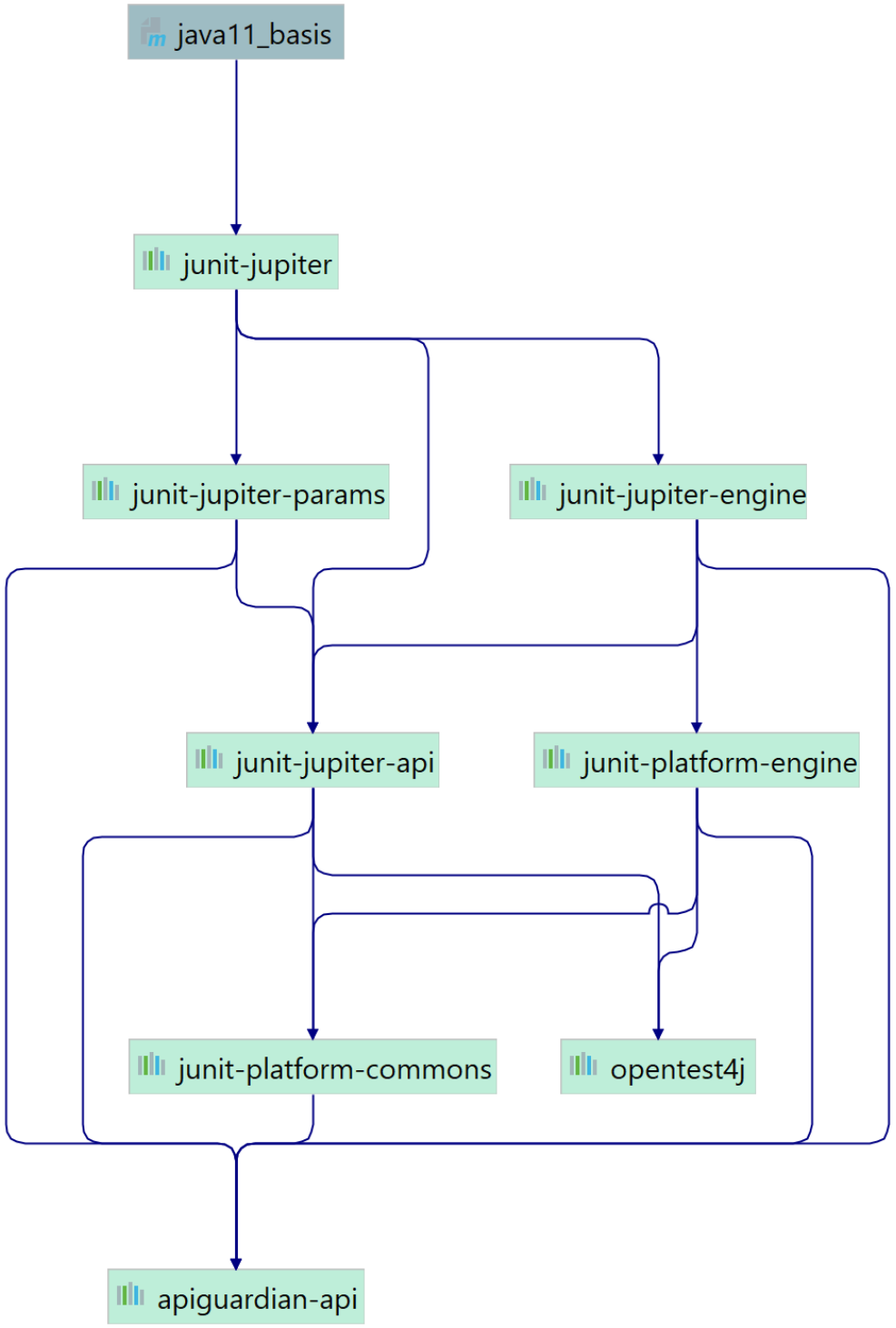
- Damit können lokal verfügbare Artefakte adressiert werden.
- Es muss dann via systemPath-Element der Pfad angeben werden (***${basedir}*** ist Standard-Property)

Transitive Dependencies

- Maven versucht auch die Artefakte bereit zu stellen, die wiederum von unseren direkten Dependencies benötigt werden.
- Zu Beginn wird Maven deshalb erstmal eine Menge von Artefakten in das lokale Repository laden!
- Anzeige des aktuellen Dependency-Tree:

  `mvn dependency:tree`

  (geht in der IDE meist hübscher)

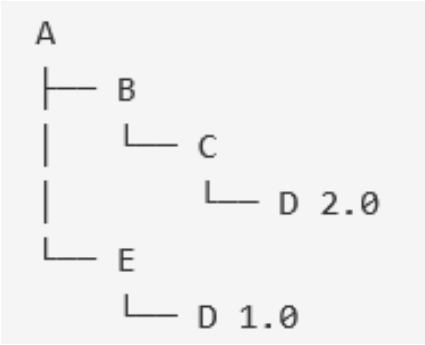


Das Dependency-Plugin

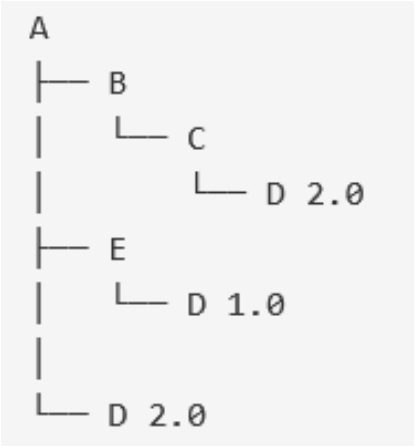
- Unterstützung für das Dependency-Management bietet das Dependency-Plugin welches via Super-POM stets eingebunden ist.
  (https://maven.apache.org/plugins/maven-dependency-plugin/index.html)

- Darüber lassen sich z.B. genauer die Dependencies analysieren bzw. filtern
  - ***`mvn dependency:analyze`*** Welche Dependencies werden deklariert/genutzt
  - ***`mvn dependency:tree`*** Zeigt den Dependency-Baum an
  - ***`mvn dependency:tree -Dincludes=*:apiguardian-api:*:*`***
    Zeigt nur den Teilbaum bzgl. der Dependency an
  - ***`mvn dependency:help`*** Liefert alle Goals (26 in aktuellen Versionen)

Transitivität und Konflikte

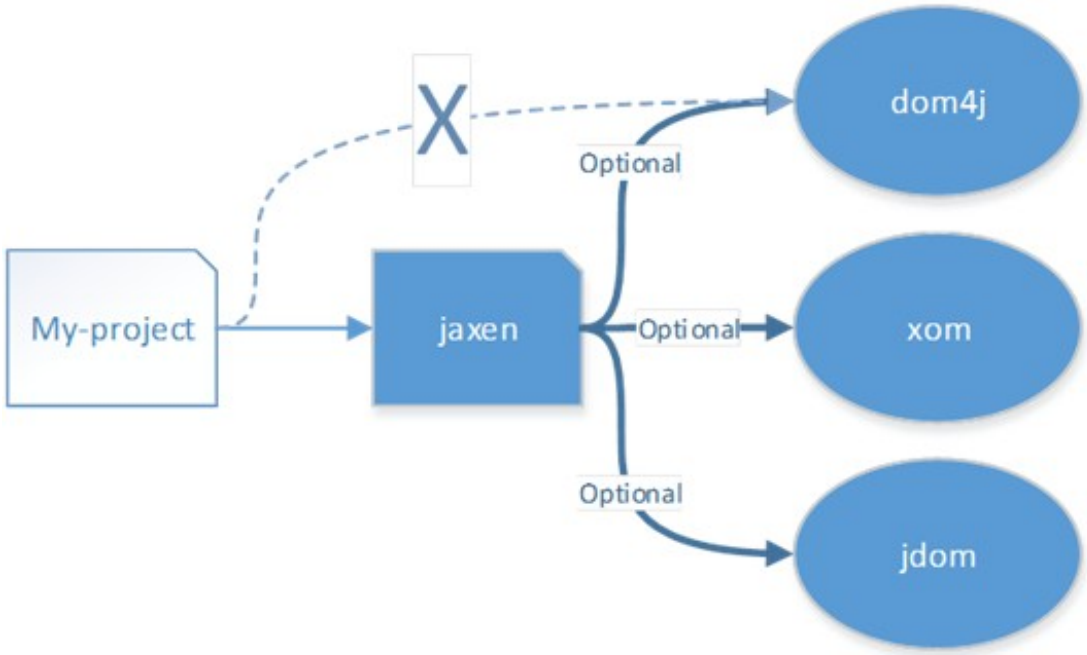


- Maven wählt die bzgl. der Ebenen und innerhalb einer Ebene nach der Reihenfolge (sprich abhängig von der Reihenfolge in der POM) am nähesten (nearest) liegende Dependency unabhängig von der Version aus.
  Im Beispiel: D 1.0
- Das ist nicht immer die gewünschte Version!!



- Wir können aber stets die gewünschte Version als direkte Dependency formulieren (auch wenn wir diese nicht direkt benötigen).
  Im Beispiel: D 2.0
- Das funktioniert aber nur, wenn Version 2.0 von D abwärtskompatibel ist!
- Das Dependency-Plugin kann im Vorfeld auf Konflikte prüfen:
  mvn dependency:tree -Dverbose zeigt Konflikte an
- Das Enforcer-Plugin kann hier ebenfalls hilfreich sein:
  https://maven.apache.org/enforcer/maven-enforcer-plugin/index.html

Optionale Dependencies



- Das Artefakt „jaxen“ benötigt alle seine Dependencies („dom4j“, „xom“, „jdom“) hat diese aber als optional markiert.

- Wenn dann ein weiteres Projekt, hier „My-project“, „jaxen“ als Dependency listet, werden die optionalen Artefakte nicht transitiv bereit gestellt.
  Stattdessen muss das weitere Projekt explizit die benötigten Artefakte als Dependencies deklarieren!
  Details siehe:
  https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html

Dependencies ausschließen

```
Project-A
   -> Project-B
        -> Project-D <! -- This dependency should be excluded -->
              -> Project-E
              -> Project-F
   -> Project C
```

- Aus was für Gründen auch immer soll „Project-D“ keine transitive Dependency von „Project-A“ werden, wurde aber nicht auf der Ebene von B als optional markiert (sollte nur in speziellen Fällen nötig sein).
- Details siehe:
  https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html
- Ein Beispiel mit Nutzung des Dependency- und des Enforcer-Plugins:
  https://www.baeldung.com/maven-version-collision

Dependencies ausschließen: Lösung

- „Project-D“ kann als transitive Dependency in „Project-A“ ausgeschlossen werden!
- Die „Exclusion“ erfolgt bei der Dependency „Project-B“

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample.ProjectA</groupId>
  <artifactId>Project-A</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
  <dependencies>
    <dependency>
      <groupId>sample.ProjectB</groupId>
      <artifactId>Project-B</artifactId>
      <version>1.0-SNAPSHOT</version>
      <exclusions>
        <exclusion>
          <groupId>sample.ProjectD</groupId> <!-- Exclude Project-D from Project-B -->
          <artifactId>Project-D</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
</project>
```

POM Inheritance

- Das Packaging muss ***pom*** sein im Parent-Projekt
- Elemente, die vererbt werden:
  - Dependencies
  - Developers und Contributors
  - Plugin-Listen
  - Plugin-Executions mit korrespondierenden IDs
  - Plugin Konfiguration
  - ...
- Sofern das Parent-Projekt nicht das Elternverzeichnis bzw. im lokalen Repository abgelegt ist, muss der Pfad angeben werden.
- Details siehe: https://maven.apache.org/pom.html

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>maven-parent-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!-- This needs to be set -->
    <packaging>pom</packaging>
...
</project>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <!-- Here we define the parent -->
    <parent>
        <groupId>sk.train</groupId>
        <artifactId>maven-parent-project</artifactId>
        <version>1.0-SNAPSHOT</version>
        <!-- This is optional -->
        <relativePath>../maven-parent-project</relativePath>
    </parent>

    <artifactId>maven-inheritance-example</artifactId>
    <!-- version und groupId können vererbt werden -->
...
</project>
```

<dependencyManagement>

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <parent>
      <groupId>sk.train</groupId>
      <artifactId>maven-parent</artifactId>
      <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>maven-child-project</artifactId>
    <packaging>jar</packaging>

    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
      </dependency>
    </dependencies>
...
</project>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>maven-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <dependencyManagement>
      <dependencies>
        <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.11</version>
          <scope>test</scope>
        </dependency>
        <dependency>
          <groupId>org.apache.commons</groupId>
          <artifactId>commons-lang3</artifactId>
          <version>3.9</version>
        </dependency>
      </dependencies>
    </dependencyManagement>
...
</project>
```

- Hierbei erbt das Kind nur das, was es referenziert. Die Version wird übernommen. (quasi Lookup-Table)
  - Dadurch können optionale Dependency-Vorgaben für alle Kinder gemacht werden!

- Analogon existiert für Plugins: <pluginManagement>
  siehe z.B: https://devflection.com/posts/2020-04-12-maven-part-3/

POM Aggregation: Multi-Modul-Projekt

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
      <groupId>sk.train</groupId>
      <artifactId>my-aggregator</artifactId>
      <version>1.0</version>
      <!-- This needs to be set -->
      <packaging>pom</packaging>

      <modules>
        <!-- relative path to project directory -->
        <module>my-project</module>
        <module>another-project</module>
        <!-- relative path to POM -->
        <module>third-project/pom-example.xml</module>
      </modules>
    ...
</project>
```

- Hier geht es nicht um Vererbung, sondern dass ein Build gegen das Aggregator-Projekt, den Build aller Module bewirkt!
- Die Module sind deshalb im Aggregator mittels Angabe ihrer Top-Level-Verzeichnisse bzw. direkter Angabe der POM-Position gelistet.
- Allerdings wird das dann als Ganzes paketiert und deployed, wobei hier Sonderlocken möglich sind. Siehe z.B. https://stackoverflow.com/questions/63567442/how-to-deploy-only-part-of-a-multi-module-maven-project

Aggregation + Vererbung

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>maven-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
        <module>my-project</module>
        <module>another-project</module>
    </modules>
...
</project>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <parent>
    <groupId>sk.train</groupId>
    <artifactId>maven-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>my-project</artifactId>
  ...
</project>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <parent>
    <groupId>sk.train</groupId>
    <artifactId>maven-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>another-project</artifactId>
  ...
</project>
```

- Aggregation und Vererbung werden oft zusammen eingesetzt.
- Ideale Verzeichnisstruktur: Aggregator/Parent liegt im Elternverzeichnis.

Bill Of Materials (BOM)

- Eine BOM ist eine Zusammenstellung von Dependencies, die bzgl. Versionen und Features zusammen passen.
- Eine BOM wird durch eine POM mit einer entsprechenden ***<dependencyManagement>***-Sektion bereit gestellt.
- Paketierungsformat ist ***pom***.
- Viele Frameworks nutzen diese Art Bereitstellungs-Mechanismus, z.B:
  - spring-data-bom:
    BOM for Spring Data project
  - jackson-bom:
    BOM for Jackson dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>maven_sampleBOM</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <dependencyManagement>
      <dependencies>
        <!--  testing dependencies  -->
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>5.9.2</version>
          <scope>test</scope>
        </dependency>
        <dependency>
          <groupId>org.mockito</groupId>
          <artifactId>mockito-core</artifactId>
          <version>5.0.0</version>
          <scope>test</scope>
        </dependency>
      </dependencies>
    </dependencyManagement>
...
</project>
```

Nutzung einer BOM via Parent-Mechanismus

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>maven-sampleBOM</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
...
</project>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <parent>
    <groupId>sk.train</groupId>
    <artifactId>maven-sampleBOM</artifactId>
    <version>1.0</version>
  </parent>

  <groupId>sk.train</groupId>
  <artifactId>child-project</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

- Bei einer bereit gestellten BOM wird diese nicht im Elterverzeichnis liegen, sondern muss via Repository gezogen werden.

Nutzung einer BOM via Import-Mechanismus

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <groupId>sk.train</groupId>
    <artifactId>maven-sampleBOM</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
...
</project>
```

- Der import-Scope ist ein Spezialfall für den Import einer Projektdefinition.

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <groupId>sk.train</groupId>
  <artifactId>new-project</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>sk.train</groupId>
        <artifactId>maven-sampleBOM</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

Archetypes

- Projekt-Templates können als Archetypes zur Verfügung gestellt werden.
- ***`mvn archetype:generate`*** Dialog für Template-Suche und Projektkonfiguration

  `mvn archetype:generate -Dfilter=org.apache:struts` mit Filterung
- Es ist nicht so einfach, das passende zu finden und insbesondere die vom Maven-Quellprojekt selbst bereit gestellten sind veraltet!
- In den IDEs ist das in der Regel in einen entsprechenden GUI-Dialog eingepackt.
- Details hierzu, insbesondere zur Erstellung:
  https://maven.apache.org/archetype/maven-archetype-plugin/

Profile

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
   <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <maven.compiler.showWarnings> true
        </maven.compiler.showWarnings>
        <maven.compiler.showDeprecation> true
        </maven.compiler.showDeprecation>
        <maven.compiler.verbose> true </maven.compiler.verbose>
      </properties>
    </profile>
    <profile>
      <id>noverbose</id>
      <properties>
        <maven.compiler.showWarnings> false
        </maven.compiler.showWarnings>
        <maven.compiler.showDeprecation> false
        </maven.compiler.showDeprecation>
      </properties>
    </profile>
  </profiles>
  ...
</project>
```

- Statt verschiedene POMs zu benutzen, können Profile für unterschiedliche Einstellungen verwendet werden.
- Die Profil-Einstellungen überschreiben die Standardwerte.
- Profile werden definiert
  - in der POM
  - mit Einschränkungen in settings.xml (global oder lokal)
- Aktiviert durch
  - Explizit durch Optionen, z.B. **`mvn compile -P noverbose`**
  - Aktivierung per XML (siehe Bsp.)
  - Basierend auf Umgebungsvariablen
  - Existenz von Dateien
- siehe auch: https://maven.apache.org/guides/introduction/introduction-to-profiles.html

Welche Profile werden benutzt?

```
$ mvn help:active-profiles

Active Profiles for Project 'My Project':

The following profiles are active:

- my-settings-profile (source: settings.xml)
- my-external-profile (source: profiles.xml)
- my-internal-profile (source: pom.xml)
```

- Via Maven Help-Plugin können unter anderem die aktiven Profile angezeigt werden.

Settings

Grundgerüst der settings.xml

```
<settings
xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
       http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <localRepository/>
  <interactiveMode/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

- Einstellungsebenen:
  - settings.xml in <*MAVEN_HOME*>\conf (global)
  - settings.xml in <*HOME*>\.m2 (lokal, je User)
- Effektive Einstellungen via ***mvn help:effective settings***

Wichtige Einstellungen

- localRepository
  - wohin sollen die heruntergeladenen Artefakt gelegt werden
  - Standard: <*HOME*>\.m2\repository, sollte geändert werden, wenn <HOME> auf einem Netzlaufwerk liegt oder bei Anmeldung kopiert wird
- proxies
  - Netzwerk-Proxy-Einstellungen
- servers
  - Nutzername/Passwort
  - Key-File/Passphrase
  - Zuordnung zu Rechnerdefinitionen über die id
  - Sollte in den Nutzer-Settings gesetzt werden
  - Seit Maven 2.1 auch verschlüsselt möglich

Settings: Proxy-Konfiguration

```
<settings
xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
       http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <proxies>
    <proxy>
    <id>companyProxy</id>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.company.com</host>
    <port>8080</port>
    <username>proxyusername</username>
    <password>proxypassword</password>
    <nonProxyHosts />
    </proxy>
  </proxies>
</settings>
```

Settings: Mirrors

```
<settings
xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
       http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <mirrors>
    <mirror>
        <id>mynexus</id>
        <mirrorOf>*</mirrorOf>
        <name>Global Mirror</name>
        <url>http://mynexus.local/repo/path</url>
    </mirror>
  </mirrors>
</settings>
```

- Alternative Server für Repositories
- Für einzelne Server oder alle

Projekt-Informationen

- Es können übliche Informationen zum Projekt festgehalten werden.
- Beispiele: https://maven.apache.org/pom.html#More_Project_Information

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
     ...
    <name>java11_basis</name>
    <description>sample project</description>
    <url>www.sampleproject.de</url>
    <inceptionYear>2023</inceptionYear>
    <licenses>
      <license>  <!--   ...   -->     </license>
    </licenses>
    <developers>
      <developer>
        <id>mm</id>
        <name>Max Musterfrau</name>
        <email>muster@example.de</email>
        <!--       ...      -->
      </developer>
    </developers>
    <contributors>  <!--   ...   -->  </contributors>
    <organization>  <!--   ...   -->  </organization>
    <issueManagement>  <!--   ...   -->  </issueManagement>
    <ciManagement>  <!--   ...   -->  </ciManagement>
    <mailingLists>  <!--   ...   -->  </mailingLists>
    <scm>  <!--   ...   -->  </scm>
</project>
```

Projekt-Dokumentation

```
+- src/
   +- site/
      +- apt/
      |  +- index.apt
      !
      +- markdown/
      |  +- content.md
      |
      +- fml/
      |  +- general.fml
      |  +- faq.fml
      |
      +- xdoc/
      |  +- other.xml
      |
      +- site.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <version>4.0.0-M4</version>
      </plugin>
      ...
    </plugins>
  </build>

  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>3.4.2</version>
      </plugin>
    </plugins>
  </reporting>
...
</project>
```

- Mit Hilfe des Site Build Lifecycle kann die Dokumentation zum Projekt erzeugt werden: ***`mvn site`***
- Die Ausgabeformate (default ist HTML) und die Struktur der Seite kann durch Vorgaben angepasst werden.
- Neben dem Project Info Reports Plugin (default) stehen weitere spezielle Report-Plugins zur Verfügung.

Aktualisierung von Maven

- Da Maven weder plattform-spezifische Installationsverfahren noch Update-Mechanismen vorsieht, muß eine Aktualisierung manuell erfolgen:

1) Installation der neuen Version.

2) Aktualisierung der Pfad-Variablen (und optional der „MAVEN_HOME“-Variablen).

3) Sofern die bisherigen Einstellungen beibehalten werden sollen, müssen die bisherigen Konfigurations-Einstellungen in die Verzeichnisstruktur der aktuellen Installation kopiert werden:
   - in der Regel zumindest die globale settings.xml in <*Maven_HOME*>/conf