



# Maven

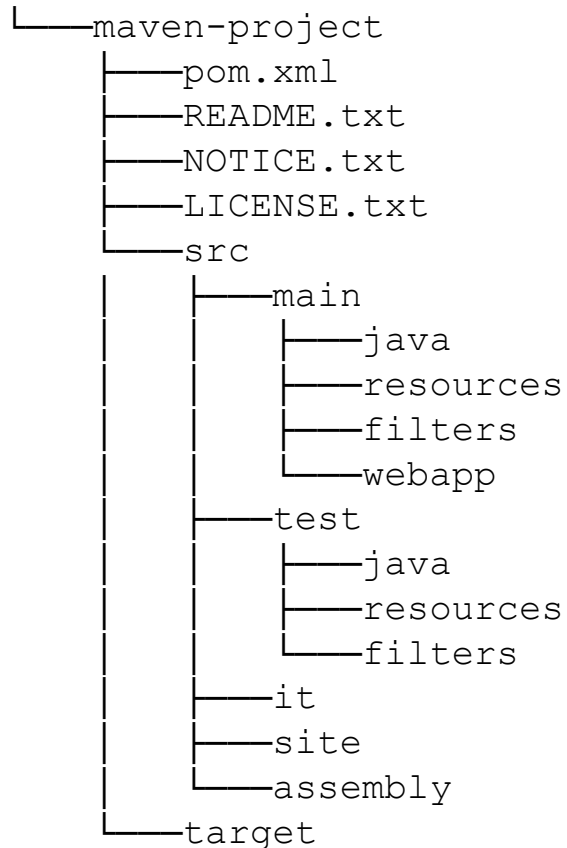
# Was ist Maven

- 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:
  - MAVEN\_HOME
  - JAVA\_HOME, muss auf ein JDK verweisen!  
(Maven 3.3+ requires JDK 1.7 or above to execute)
  - PATH setzen
- Test: `mvn -version`

# 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>javal1_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:javal1\_basis:1.0-SNAPSHOT***

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

# Projekt-Informationen

- Es können übliche Informationen zum Projekt festgehalten werden.
- Beispiele:  
[https://maven.apache.org/pom.html#More\\_Project\\_Information](https://maven.apache.org/pom.html#More_Project_Information)

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <name>javall_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>
```

# 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.
- 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	Description
validate	checks if the project is correct and all information is available
compile	compiles the source code into binary artifacts
test	executes the tests
package	takes the compiled code and package it, for example into a JAR file
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

- 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

# 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
<code>process-resources</code>	<code>resources:resources</code>
<code>compile</code>	<code>compiler:compile</code>
<code>process-test-resources</code>	<code>resources:testResources</code>
<code>test-compile</code>	<code>compiler:testCompile</code>
<code>test</code>	<code>surefire:test</code>
<code>package</code>	<code>ejb:ejb</code> or <code>ejb3:ejb3</code> or <code>jar:jar</code> or <code>par:par</code> or <code>rar:rar</code> or <code>war:war</code>
<code>install</code>	<code>install:install</code>
<code>deploy</code>	<code>deploy:deploy</code>

- siehe auch: <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

# 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      -->
    <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] -----
```

# Das Compiler-Plugin

Goal	Description
<code>compiler:compile</code>	Compiles application sources
<code>compiler:help</code>	Display help information on maven-compiler-plugin. Call <code>mvn compiler:help -Ddetail=true -Dgoal=&lt;goal-name&gt;</code> to display parameter details.
<code>compiler:testCompile</code>	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  
(ist oft auch für andere Plugins möglich)

# Das Compiler-Plugin: Konfiguration

- Wenn der Build mit einer anderen Java-Version 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](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

# Das Surefire-Plugin

- Goals und Parameter

- `mvn surefire:help` Liefert Goals des Plugins  
(generell `mvn <plugin>:help` Goals des Plugins)
- `mvn surefire:help -Ddetail=true -Dgoal=test` Auflistung aller Parameter  
(generell `mvn help:describe Dcmd=<phase>`)

- maven-failsafe-plugin ist im Gegensatz für reine Integrations-Tests gedacht  
(entkoppelt den Build vom Test-Ergebnis)

- Testausführung

- 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>

# 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.

# 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

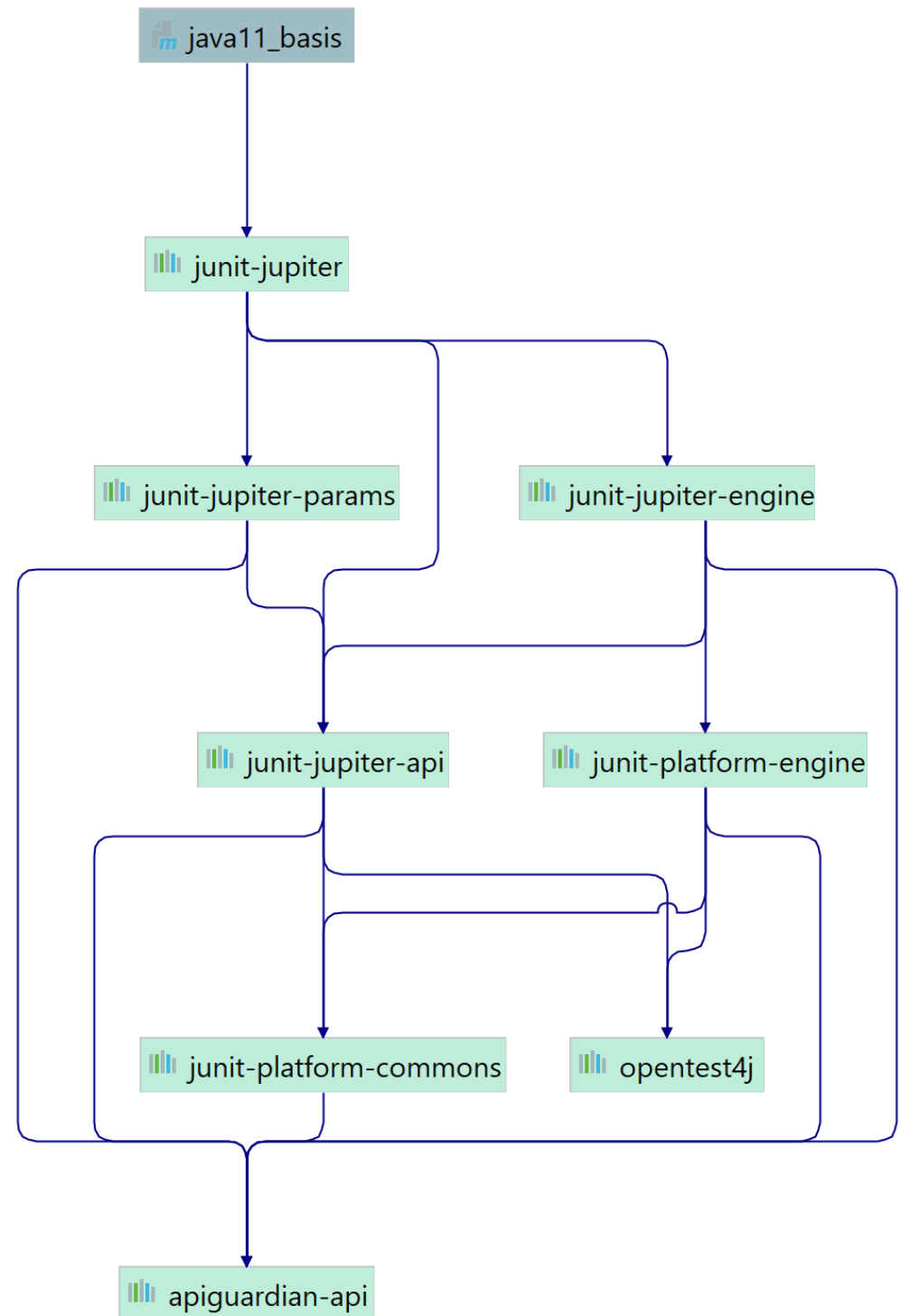
```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependency>
    <groupId>mylib</groupId>
    <artifactId>mylib</artifactId>
    <scope>system</scope>
    <version>1.0</version>
    <systemPath>${basedir}\libs\mylibrary.jar</systemPath>
  </dependency>
  ...
</project>
```

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

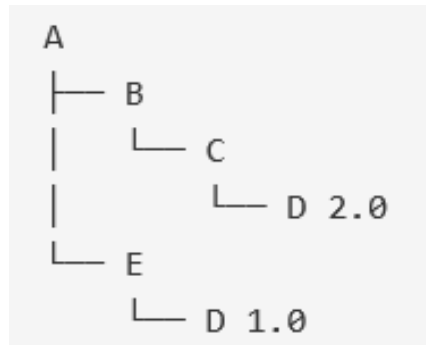


## Transitive Dependencies

- 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)



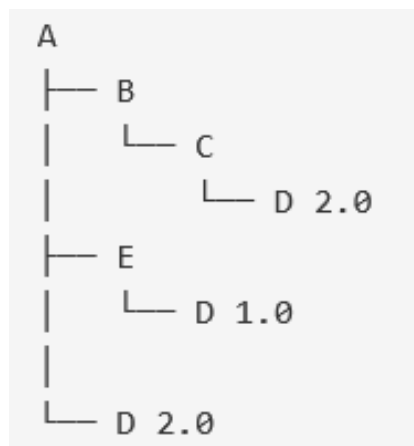
## Transitivität und Konflikte



- Maven wählt die bzgl. der Ebenen und innerhalb einer Ebene nach der Reihenfolgen am nächsten (nearest) liegende Dependency 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

- Dependencies können auch gezielt ausgeschlossen (excluded) werden bzw. als optional markiert werden.

Details siehe:

<https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html>

# 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 angegeben 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/>

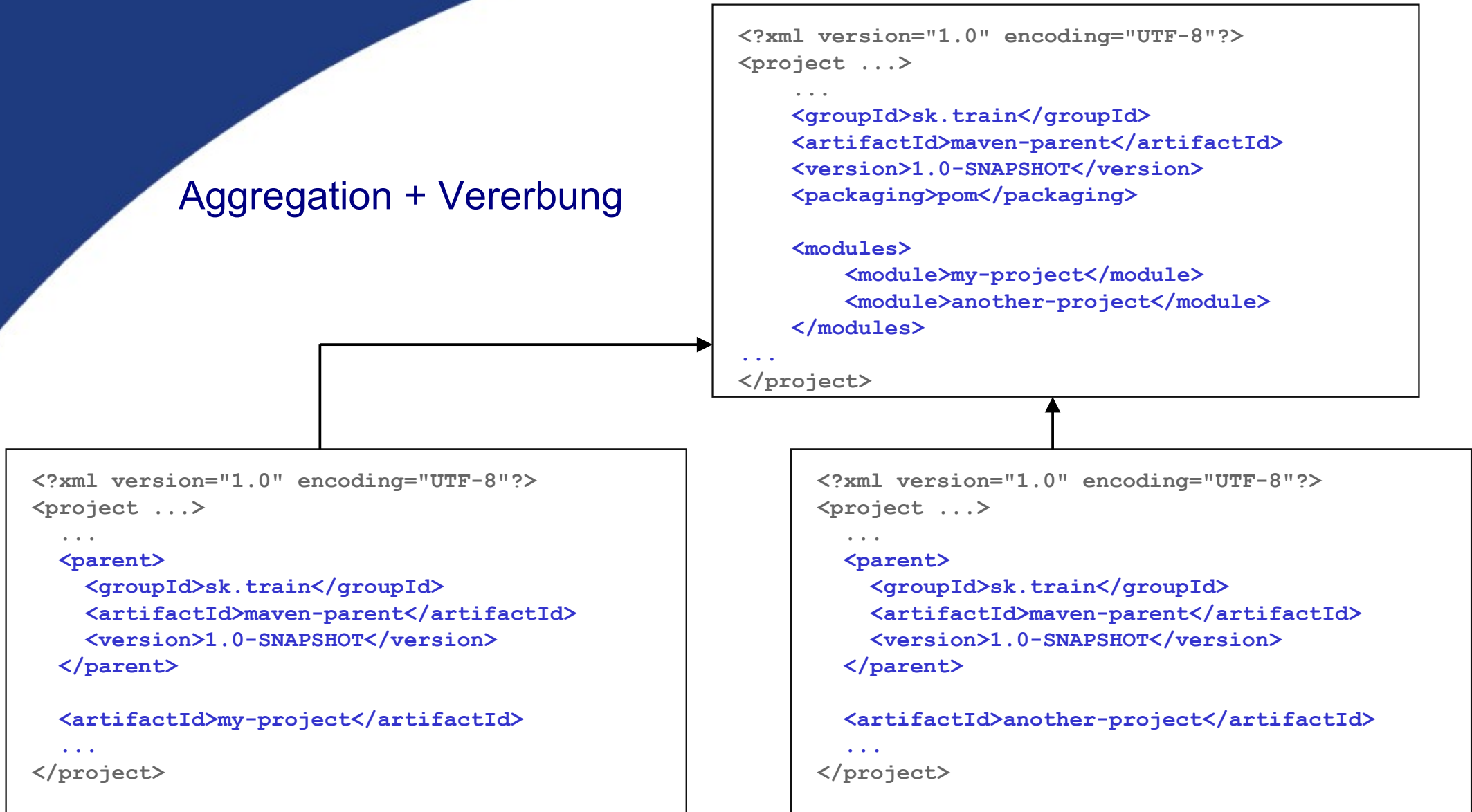
# POM Aggregation: Multi-Modul-Projekt

```
<?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.

## Aggregation + Vererbung



- 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>
```

# Projekt-Dokumentation

```
+-- src/
  +- site/
    +- apt/
      | +- index.appt
      |
    +- 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.

# 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/>

# 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)
- Effektive Einstellungen via ***mvn help:effective settings***

## 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

# 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.