PRAGTICS

FASTER SOFTWARE - BETTER BUSINESS

The good

Takeaway - Maven tutorial

The bad

May 2018

Agenda

1. Ziele
2. Wichtige Links
3. Übersicht und Ablauf
4. Build Lifecycles
5. Build Phases
6. Aufbau einer POM
7. Projektaufbau
8. Build Goals und Plugins
9. Best Practices

Ziel

Warum?

Software-Projekte werden häufig zwischen Entwicklern geteilt. Der Bau der Software sollte deshalb auf allen Rechnern gleich gehalten werden.

Womit?

Deklarativer
Wiederholbarer
Konfiguierbarer

Build-Prozess mit Abhängigkeitsmanagement

Möglichkeiten?

Maven by Apache Foundation
Gradle by Gradle Inc.
Pants by Twitter
Buck by Facebook

Wichtige Links

Root
http://maven.apache.org

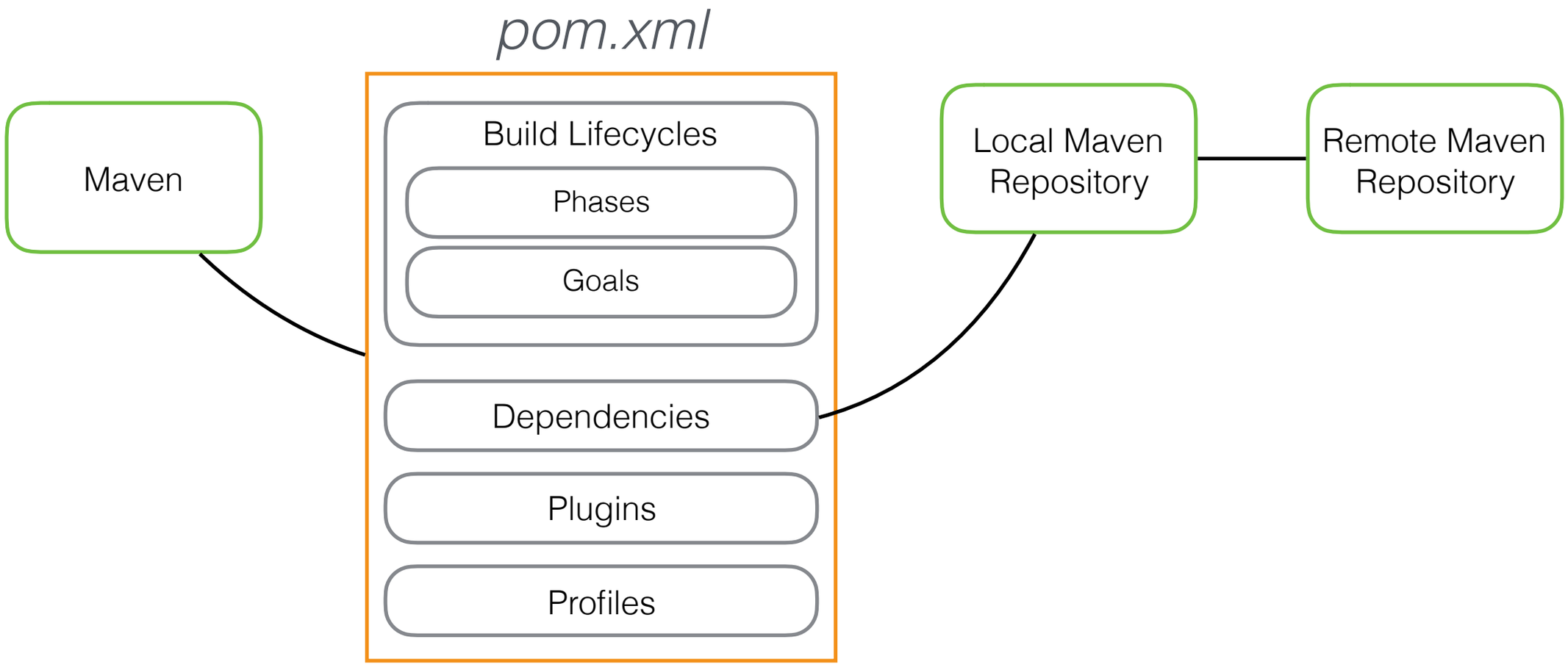
Local Repository
${user.home}/.m2/repository

Remote Repository
http://mvnrepository.com/

Project Dependencies
${project.root}/external libraries

Installation
Maven
Maven Wrapper (portabler)

Übersicht und Ablauf



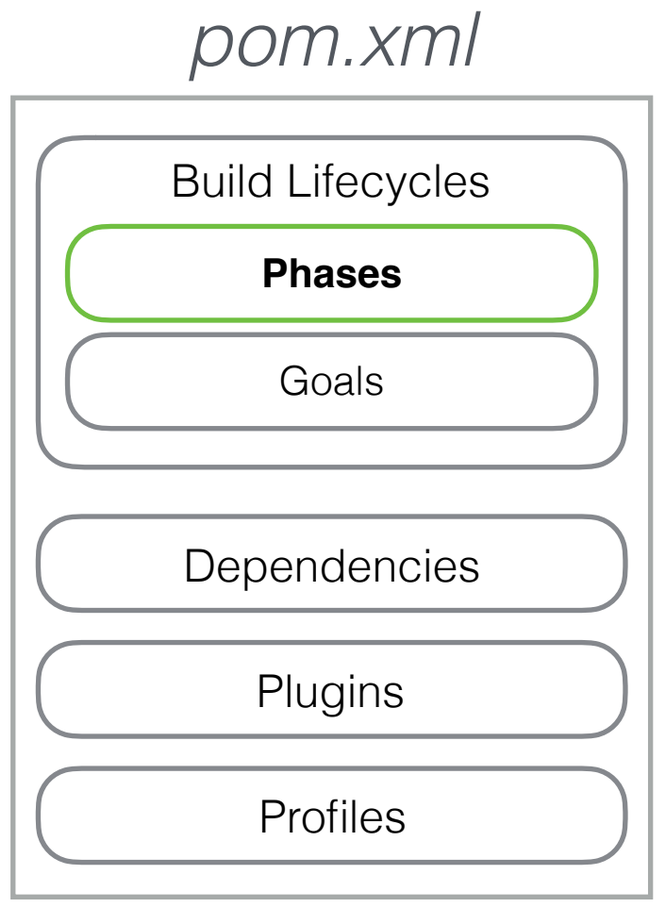
1. **Ausführung mit $mvn [options] [<goal(s)>] [<phase(s)>].**
2. pom.xml wir eingelesen.
3. Das über die Options gesetzte Profile wird bestimmt.
4. Dependencies werden geladen. Lokal prüfen, ob Abhängigkeit schon vorhanden. Wenn nicht, dann an die das Standard-Remote Repository oder weitere, konfigurierbare Repositories gehen.
5. Build Lifecycles, Phases und/ oder Goals, werden ausgeführt.

Standard Phasen des „Build Lifecycles“

1. **Default**
   Deployment des Projektes.

2. **Clean**
   Projekt aufräumen.

3. **Site**
   Projektdokumentation erstellen.



Übung 1.)

1. **Im Projekt die Projektdokumentation erzeugen**
   Dokumentation anschauen.

3. **Projekt aufräumen**
   Überprüfen, dass Projekt aufgeräumt wurde.

4. **Im Projekt die Projektdokumentation erzeugen**
   Dokumentation anschauen.

5. **Zusatz**
   Wie wird der „default“-Lifecycle aufgerufen?

„Default Lifecycle Phase“ im Detail

Die unten genannten Phasen werden in der genannten Reihenfolge bis zum angegeben Ziel ausgeführt. Bei „$ mvn test“ wird also „validate“, compile“ und abschließend „test“ durchgeführt.

1. **validate**
   Einfache Validierung, ob das Projekt richtig konfiguriert ist und gebaut werden kann.

2. **compile**
   Erzeugen der Java-Binares (Klassen).

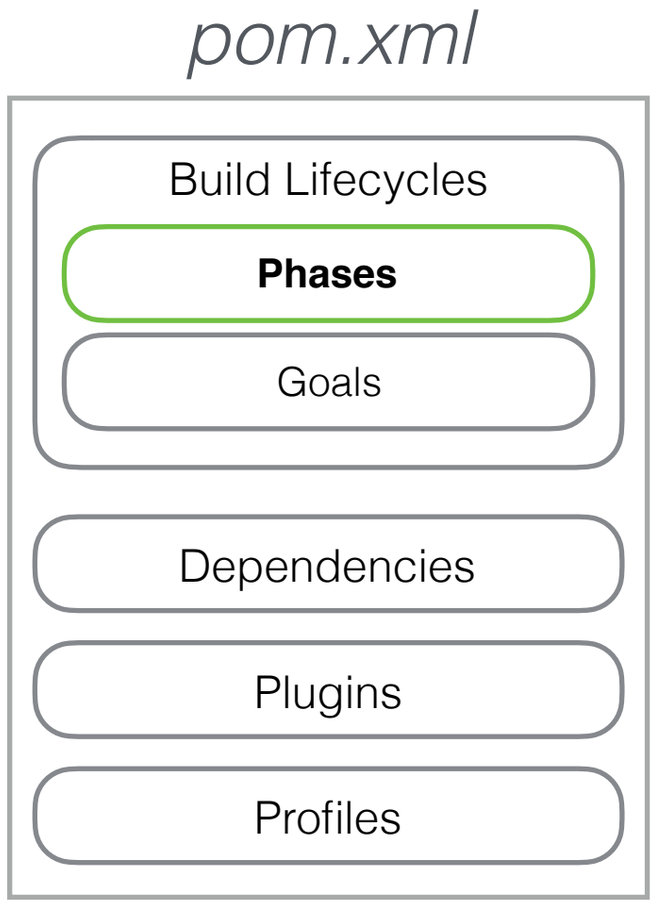
3. **test**
   Ausführen der Tests.

4. **package**
   Packen der Java-Binaries in das gewählte Format (z.B. WAR oder JAR).

5. **verify**
   Ausführen der Integrations-Tests.

6. **install**
   Installiert das Paket in das lokale Maven Repository.

7. **deploy**
   Kopieren in das remote Repository.



Übung 2.)

1. **Projekt aufräumen und Java Binaries erzeugen**
   Java Binaries anschauen.

2. **Java Binaries Testen**
   Test-Ausgabe überprüfen.

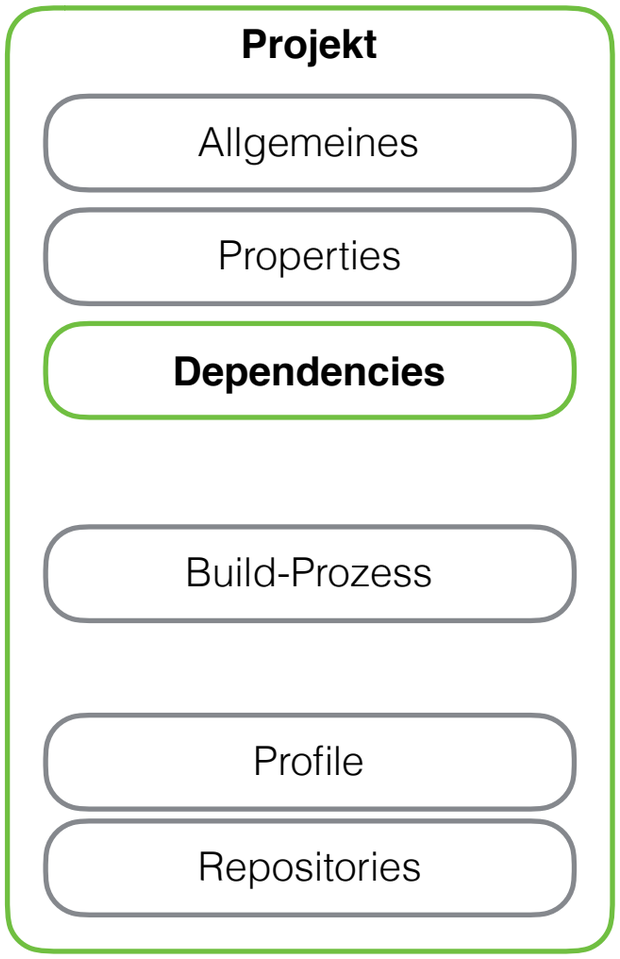
3. **Java Binaries im Maven Debug-Modus testen**
   Debug-Ausgabe überprüfen.

4. **Eigenen Test schreiben, der fehlschlägt**
   Java Binaries anschauen.

5. **Zusatz**
   Warum werden beim Ausführen von „verify“ auch die Unit-Tests ausgeführt?

Aufbau der POM

```xml
<?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>

    <contributors>
        <contributor>
            <name>Jan Bartkowiak</name>
            <email>j.bartkowiak@pragtics.de</email>
        </contributor>
    </contributors>

    <organization>
        <name>pragtics - faster software, better business</name>
        <url>https://www.pragtics.de</url>
    </organization>

    <licenses>
        <license>
            <distribution>Apache License 2.0</distribution>
            <url>https://www.apache.org/licenses/LICENSE-2.0</url>
        </license>
    </licenses>

    <groupId>de.pragtics.trainings</groupId>
    <artifactId>maven</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>10</maven.compiler.source>
        <maven.compiler.target>10</maven.compiler.target>
    </properties>


    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>

    </dependencies>
</project>
```



Übung 3.)

1. **Google Guava als Dependency einbinden**
   Dependency und ggf. transitive Abhängigkeiten anschauen.

2. **Guava nutzen**
   Duplikate einer Liste entfernen (Link).

3. **Guava Dependency Scope auf „test“ setzen**
   Was passiert, warum?

4. **Zusatz**
   Truck-Factor von Dependencies bestimmen (Docker Maven Plugin vs. Docker Maven). Warum sollte dies beachtet werden?

Standard-Projektaufbau nach Maven

Es gilt „Convention over Configuration“. Deswegen ergibt sich mit Maven - sofern nicht anders konfiguriert folgende Projektstruktur:

- **Source Code**
  - ${basedir}/src/main/java
- **Resources**
  - ${basedir}/src/main/resources
- **Tests**
  - ${basedir}/src/test
- **Complied Byte Code**
  - ${basedir}/target/classes
- **Distributable JAR**
  - ${basedir}/target/

Übung 4.)

1. **„Spring Boot“-Anwendung über den Initializer erzeugen**
   Link folgen und Anwendung erstellen. Abhängigkeit „Spring Boot Starter Web“ und „Spring Boot Acutator“ auswählen. Ergebnis anschauen und in die IDE importieren. Ggf. direkt starten.

2. **Anwendung testen**
   Was passiert beim Testen genau?

3. **Anwendung bauen und das JAR starten**
   Starten z.B. über „java -jar target/JAR-NAME.jar“

4. **Zusatz**
   Was sind Maven BOMs?

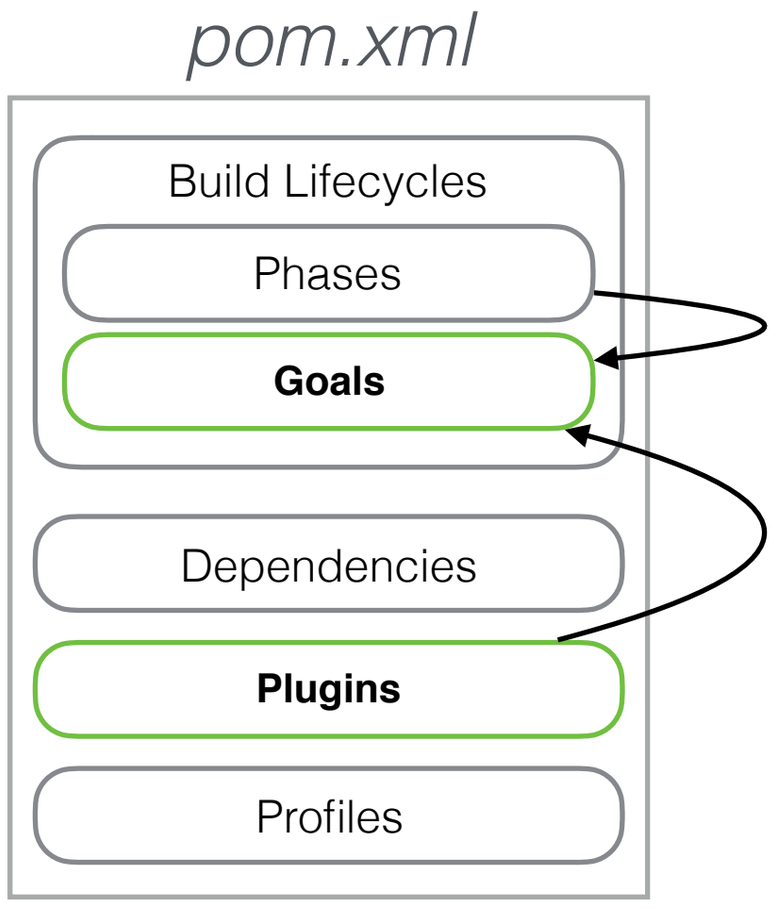
Build Goals und Plugins

Warum?

Ergänzen den Build um weitere Funktionalitäten in Form von „goals“.

Wie?

Goals - und damit die Plugins - hängen an einer, oder mehrere „Build Lifecycle“-Phasen. Plugins können - und müssen - häufig explizit an eine „Lifecycle“-Phase gebunden werden.



Möglichkeiten?

Statische Source Code Analyse
Kopieren von Ressourcen
Changelog erstellen
Auf den Server deployen
C++-Sourcen kompilieren
Tomcat starten
…

Übung 5.)

1. **JaCoCo für die Testabdeckung verwenden**
   Link folgen und Plugin in die POM einbinden. Bitte dran denken, eine passende Version der Dependency zu wählen!

2. **Anwendung bauen**
   Wann genau wird der Report erstellt und wo liegt dieser?

3. **Report auswerten**
   Plugin manuell ansteuern.

Best Practices

1. **IDE-spezifische Dateien werden nur in Ausnahmefällen committet**
   Maven-reicht für ein Setup vollkommen aus und wird von allen gängigen IDEs unterstützt.

2. **Dependencies werden niemals committet**
   Sollte der Bedarf bestehen, dass Dependencies im Firmennetz zur Verfügung stehen, wird auf ein eigenes „Remote Repository“ zurückgegriffen. Nexus oder Artifactory sind hier mögliche (kommerzielle) Vertreter. Apache archiva ist Open Source.

3. **Versionen von Abhängigkeiten sind so genau wie möglich zu Pflegen**
   Wird anstelle von 1.1.0 nur 1 verwendet, so sucht Maven automatisch nach einer passenden Version mit 1.*.*. Im schlimmsten Fall werden damit neue Schnittstellen in ein Projekt gebracht, ohne dass das jemand merkt.

4. **Versionen von Abhängigkeiten werden in den Properties gepflegt**
   Das vereinfacht die schnelle Anpassung, da die Abhängigkeit an mehreren Stellen benutzt wird.

5. **Copyrights und Lizenzen sind zu beachten**
   Nicht alles, was frei verfügbar ist, darf in einer kommerziellen Software auch genutzt werden. Um Projekte ohne Lizenzen sollte ein großer Bogen gemacht werden.

6. **Automatisiere das Bauen, Testen und Deployen**
   Maven Projekte können - und sollten - durch eine sogenannte Build-/ und Deplyo-Pipeline verwaltet werden. Damit lassen sich typische Fehler vermeiden und viel Zeit sparen.

Für den nächsten Start

1. **Maven in 5 Minuten**
   Reicht, um nach dem Tutorial sein Wissen aufzufrischen und einfach loszulegen.

2. **Neues Problem, welches beim Build gelöst werden könnte?**
   Maven-Plugins anschauen. Ansonsten die pom.xml so einfach wie möglich halten. Ein Plugin weniger, bedeutet auch ein Thema weniger, was ein Neuling lernen muss.

3. **In die „Dependency Hell“ geraten?**
   Einfach einmal hier vorbeischauen.

4. **Maven kann nicht installiert werden?**
   Kein Problem, am Besten beim "Maven Wrapper“ vorbeischauen.