Java

Buildsysteme

Marcus Köhler

1. Februar 2019

Java-Kurs

Überblick

1. Buildsysteme
    - Warum Buildsysteme?
    - Lifecycles
2. Maven
    - Was ist Maven?
    - Die `pom.xml`
3. Gradle
    - Was ist Gradle?
    - Die `build.gradle`
4. Demo

Buildsysteme

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.
Dazu kommt noch das sogenannte *Dependency management*, d.h. das Management von benötigten Bibliotheken etc., was von Hand nur schwer umzusetzen ist.

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.
Dazu kommt noch das sogenannte *Dependency management*, d.h. das Management von benötigten Bibliotheken etc., was von Hand nur schwer umzusetzen ist.
Um das alles weitestgehend zu automatisieren, wurden Buildsysteme geschrieben. Sie sorgen automatisiert dafür, dass:

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.
Dazu kommt noch das sogenannte *Dependency management*, d.h. das Management von benötigten Bibliotheken etc., was von Hand nur schwer umzusetzen ist.
Um das alles weitestgehend zu automatisieren, wurden Buildsysteme geschrieben. Sie sorgen automatisiert dafür, dass:

- Alle nötigen Bibliotheken und Ressourcen verfügbar sind

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.
Dazu kommt noch das sogenannte *Dependency management*, d.h. das Management von benötigten Bibliotheken etc., was von Hand nur schwer umzusetzen ist.
Um das alles weitestgehend zu automatisieren, wurden Buildsysteme geschrieben. Sie sorgen automatisiert dafür, dass:

- Alle nötigen Bibliotheken und Ressourcen verfügbar sind
- Unit- & Integrationtests ausgeführt werden, sofern vorhanden

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.
Dazu kommt noch das sogenannte *Dependency management*, d.h. das Management von benötigten Bibliotheken etc., was von Hand nur schwer umzusetzen ist.
Um das alles weitestgehend zu automatisieren, wurden Buildsysteme geschrieben. Sie sorgen automatisiert dafür, dass:

- Alle nötigen Bibliotheken und Ressourcen verfügbar sind
- Unit- & Integrationtests ausgeführt werden, sofern vorhanden
- Der Code korrekt kompiliert wird

Warum Buildsysteme?

Der Java-Compiler `javac` hat eine riesige Menge an Optionen, die nötig sind, um größere Programme mit mehreren Verzeichnissen zu verarbeiten. Allerdings heißt das auch, dass für große Projekte entsprechend lange Compiler-Aufrufe nötig sind.
Dazu kommt noch das sogenannte *Dependency management*, d.h. das Management von benötigten Bibliotheken etc., was von Hand nur schwer umzusetzen ist.
Um das alles weitestgehend zu automatisieren, wurden Buildsysteme geschrieben. Sie sorgen automatisiert dafür, dass:

- Alle nötigen Bibliotheken und Ressourcen verfügbar sind
- Unit- & Integrationtests ausgeführt werden, sofern vorhanden
- Der Code korrekt kompiliert wird
- Das fertige Programm quasi „Release-fertig“ gemacht wird

Einstellungen

Allen Buildsystemen ist gemein, dass die Einstellungen, wie ein Projekt gebaut werden muss, in einer zentralen Datei gespeichert sind. Hierzu gehören u.a.:

- Der Name & die Version des Projekts

Einstellungen

Allen Buildsystemen ist gemein, dass die Einstellungen, wie ein Projekt gebaut werden muss, in einer zentralen Datei gespeichert sind. Hierzu gehören u.a.:

- Der Name & die Version des Projekts
- In welchen Verzeichnissen Quelldateien und Ressourcen zu finden sind

Einstellungen

Allen Buildsystemen ist gemein, dass die Einstellungen, wie ein Projekt gebaut werden muss, in einer zentralen Datei gespeichert sind. Hierzu gehören u.a.:

- Der Name & die Version des Projekts
- In welchen Verzeichnissen Quelldateien und Ressourcen zu finden sind
- Welche Dependencies gebraucht werden

Einstellungen

Allen Buildsystemen ist gemein, dass die Einstellungen, wie ein Projekt gebaut werden muss, in einer zentralen Datei gespeichert sind. Hierzu gehören u.a.:

- Der Name & die Version des Projekts
- In welchen Verzeichnissen Quelldateien und Ressourcen zu finden sind
- Welche Dependencies gebraucht werden
- Wie das Programm gepackt werden muss

Einstellungen

Allen Buildsystemen ist gemein, dass die Einstellungen, wie ein Projekt gebaut werden muss, in einer zentralen Datei gespeichert sind. Hierzu gehören u.a.:

- Der Name & die Version des Projekts
- In welchen Verzeichnissen Quelldateien und Ressourcen zu finden sind
- Welche Dependencies gebraucht werden
- Wie das Programm gepackt werden muss
- ...

Einstellungen

Allen Buildsystemen ist gemein, dass die Einstellungen, wie ein Projekt gebaut werden muss, in einer zentralen Datei gespeichert sind. Hierzu gehören u.a.:

- Der Name & die Version des Projekts
- In welchen Verzeichnissen Quelldateien und Ressourcen zu finden sind
- Welche Dependencies gebraucht werden
- Wie das Programm gepackt werden muss
- ...

Allerdings ist von System zu System unterschiedlich, wie genau das alles abgespeichert wird; dazu später mehr.

Lifecycles

Der *Lifecycle* eines Programms ist(im Kontext von Buildsystemen) die Folge von Prozessen, die für ein gewünschtes Ziel notwendig sind. Die Buildsysteme, mit denen wir uns heute beschäftigen, haben in etwa den gleichen `default`-Lifecycle:

Lifecycles

Der *Lifecycle* eines Programms ist(im Kontext von Buildsystemen) die Folge von Prozessen, die für ein gewünschtes Ziel notwendig sind.
Die Buildsysteme, mit denen wir uns heute beschäftigen, haben in etwa den gleichen `default`-Lifecycle:

`validate`	Prüfen, ob alle notwendigen nformationen vorhanden sind
`compile`	Den Quellcode kompilieren
`test`	ggf. Tests ausführen
`package`	Das Programm in ein JAR-Archiv packen
`verify`	Die Ergebnisse der `test`-Phase überprüfen
`install`	Das Programm in ein Zielverzeichnis kopieren
`deploy`	Das fertige Programm in ein Repository hochladen

Maven

Was ist Maven?

Maven ist ein Buildsystem von der Apache-Foundation, welches der Nachfolger von *Ant* ist. Mavens Build-Datei(d.h. die Einstellungen) ist in XML geschrieben.

Was ist Maven?

Maven ist ein Buildsystem von der Apache-Foundation, welches der Nachfolger von *Ant* ist. Mavens Build-Datei(d.h. die Einstellungen) ist in XML geschrieben. Maven wird folgendermaßen aufgerufen(hierzu muss man im Verzeichnis sein, in dem die Build-Datei liegt):

```
mvn <target>
ODER
mvn <plugin>:<target>
```

Wobei *<target>* eine der vorhin genannten Phasen eines Lifecycles und *<plugin>* eines von Mavens Plugins ist.

Was ist Maven?

Maven ist ein Buildsystem von der Apache-Foundation, welches der Nachfolger von *Ant* ist. Mavens Build-Datei(d.h. die Einstellungen) ist in XML geschrieben. Maven wird folgendermaßen aufgerufen(hierzu muss man im Verzeichnis sein, in dem die Build-Datei liegt):

```
mvn <target>
ODER
mvn <plugin>:<target>
```

Wobei *<target>* eine der vorhin genannten Phasen eines Lifecycles und *<plugin>* eines von Mavens Plugins ist.
Maven führt dann alle Phasen aus, die im zugehörigen Lifecycle bis hin zu *<target>* vorkommen.

Maven-Verzeichnisstruktur

Sofern nicht anders in der Build-Datei angegeben, geht Maven von der folgenden Verzeichnisstruktur innerhalb des Projekts aus:

```
<project>
├── pom.xml(Build-info)
├── src
│   ├── main
│   │   ├── java
│   │   │   └── <sourcecode>
│   │   └── resources
│   └── test
│       ├── java
│       │   └── <test-sources>
│       └── resources
└── out(Compiled classes)
```

Die `pom.xml`

Die Build-Datei von Maven heißt für gewöhnlich `pom.xml`. In ihr sind(in XML) alle Informationen zum Build abgespeichert, die von der sogenannten *Master-POM* abweichen. Die Master-POM ist sozusagen die Standardkonfiguration von Maven, die alle nicht explizit angegebenen Informationen zum Build liefert.

Die `pom.xml`

Die Build-Datei von Maven heißt für gewöhnlich `pom.xml`. In ihr sind(in XML) alle Informationen zum Build abgespeichert, die von der sogenannten *Master-POM* abweichen. Die Master-POM ist sozusagen die Standardkonfiguration von Maven, die alle nicht explizit angegebenen Informationen zum Build liefert.
Eine „leere“ `pom.xml` sieht folgendermaßen aus:

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
      w3.org/2001/XMLSchema-instance"
2 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
      .org/xsd/maven-4.0.0.xsd">
3 <modelVersion>4.0.0</modelVersion>
4
5   <groupId>com.mycompany.app</groupId> <!-- Basic info -->
6   <artifactId>my-app</artifactId>
7   <version>1.0-SNAPSHOT</version>
8 </project>
```

Die **pom.xml**

In Maven hat ein Projekt immer eine `groupId`, welche mehrere Projekte zu einer *Gruppe* zusammenfassen, und eine `ArtifactId`, welche das genaue Projekt identifiziert.

Die `pom.xml`

In Maven hat ein Projekt immer eine `groupId`, welche mehrere Projekte zu einer *Gruppe* zusammenfassen, und eine `ArtifactId`, welche das genaue Projekt identifiziert.
In der `pom.xml` werden normalerweise außer Basisinfos noch Java-Versionen und Dependencies angegeben:

```
<properties>
    <maven.compiler.source>1.7</maven.compiler.source> <!-- Define Java
     version -->
    <maven.compiler.target>1.7</maven.compiler.target>
</properties>
<dependencies>
    <dependency>
        <groupId>junit</groupid>        <!-- Define Dependency -->
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

Gradle

Was ist Gradle?

Gradle ist ein Buildsystem, welches eine eigene Sprache auf Basis von Groovy für die Build-Datei verwendet. Der Vorteil von dieser Herangehensweise ist, dass die Build-Dateien um einiges kürzer sind, als das Maven-Äquivalent. Allerdings verwendet Gradle die gleiche Standard-Verzeichnisstruktur wie Maven.

Was ist Gradle?

Gradle ist ein Buildsystem, welches eine eigene Sprache auf Basis von Groovy für die Build-Datei verwendet. Der Vorteil von dieser Herangehensweise ist, dass die Build-Dateien um einiges kürzer sind, als das Maven-Äquivalent. Allerdings verwendet Gradle die gleiche Standard-Verzeichnisstruktur wie Maven.
Gradle wird folgendermaßen aufgerufen:

```
1 gradle <task>
```

Wobei *<task>* das Gradle-Äquivalent von Maven-Targets ist. Welche Tasks verfügbar sind, kann mit `gradle tasks` angezeigt werden.

Die `build.gradle`

Gradle speichert seine Einstellungen in einer Datei namens `build.gradle` ab, wobei eine eigene Sprache auf Basis von Groovy verwendet wird(Syntaxdetails werden hier nicht weiter erklärt).

Die **build.gradle**

Gradle speichert seine Einstellungen in einer Datei namens **build.gradle** ab, wobei eine eigene Sprache auf Basis von Groovy verwendet wird(Syntaxdetails werden hier nicht weiter erklärt). Eine „leere“ **build.gradle** für Java besteht nur aus einer Zeile:

```
1 apply plugin: 'java'
```

Die `build.gradle`

Gradle speichert seine Einstellungen in einer Datei namens `build.gradle` ab, wobei eine eigene Sprache auf Basis von Groovy verwendet wird(Syntaxdetails werden hier nicht weiter erklärt). Eine „leere“ `build.gradle` für Java besteht nur aus einer Zeile:

```
1 apply plugin: 'java'
```

Dies lädt alle Java-spezifischen Einstellungen und Konventionen, wie sie von Maven vorgegeben sind.

Dependencies

Das Dependency-Management in Gradle funktioniert fast genauso wie in Maven, d.h. man sagt, was man braucht und Gradle erledigt, soweit möglich, den Rest.

Dependencies

Das Dependency-Management in Gradle funktioniert fast genauso wie in Maven, d.h. man sagt, was man braucht und Gradle erledigt, soweit möglich, den Rest.
Anders als bei Maven muss man neben der Dependency an sich noch ein Repository angegeben werden, in dem danach gesucht werden soll:

```
repositories{
    mavenCentral(); //The maven repo
}
dependencies{
    test 'junit:junit:4.1.2' //require jUnit for the test phase
}
```

Dependencies

Das Dependency-Management in Gradle funktioniert fast genauso wie in Maven, d.h. man sagt, was man braucht und Gradle erledigt, soweit möglich, den Rest.
Anders als bei Maven muss man neben der Dependency an sich noch ein Repository angegeben werden, in dem danach gesucht werden soll:

```
repositories{
    mavenCentral(); //The maven repo
}
dependencies{
    test 'junit:junit:4.1.2' //require jUnit for the test phase
}
```

Dependencies werden, wie bei Maven auch, für bestimmte Phasen definiert, wobei die Syntax folgendermaßen aussieht:

```
dependencies{
    <phase> '<groupId>:<artifactId>:<version>'
}
```

Demo

DEMO