Build-Systeme für Java

Raphael Grund Ivan Kirchev

1 Vorwort

Das Ziel dieser Ausarbeitung ist es, dem Leser einen Überblick der drei meist genutzten Build-Systeme, die für Java verwendet werden, zu geben. Weiter möchten wir über einige der Vor- und Nachteile berichten und die Build-Skripte der verschiedenen Systeme selber vergleichen. Wir wollen jedoch nicht herausfinden, welches das beste Build-System ist, sondern wir wollen den Leser so weit über Stärken und Schwächen informieren, dass er oder sie für ein Projekt das richtige Werkzeug auswählen kann. Auch wollen wir denen, die noch nicht zuvor mit Build-Systemen gearbeitet haben, die Vorzüge dieser zeigen.

2 Welche Aufgaben erfüllen die Build-Systeme

Ein Build-System ist ein Programm zum automatisierten Erzeugen von Software. Es fasst unter anderem folgende Schritte zusammen: das Kompilieren des Source-Codes, das Erstellen einer ausführbaren Datei sowie die Verteilung des Programms. In kleineren Projekten ist es möglich, all diese Aufgaben manuell zu erfüllen. Das ist nicht praktisch für größere Projekte, bei denen der Überblick schnell verloren geht und es zu viel Zeit kostet, alles manuell durchzuführen. In einer solchen Situation ist ein Build-System sehr effizient und erleichtert enorm die Arbeit des Entwicklers. Die Build-Systeme entwickeln sich mit der Zeit, werden anspruchsvoller und bieten immer mehr Optionen zur Automatisierung des Build-Prozesses.

2.1 Herunterladen von Abhängigkeiten (eng. Dependencies)

Die Ausführungen in diesem Kapitel basieren auf [4. „Building Java Projects with Maven“]

Einer der großen Vorzüge von Build-Systemen ist, dass sie dem Entwickler das Verwalten von Abhängigkeiten abnehmen können. So kann man dem Build-System eingeben, dass man eine bestimmte Version einer Bibliothek benötigt. Des Weiteren ist es auch möglich anzugeben, welche Bandbreite von Versionen für ein bestimmtes Modul zulässig ist.

```
1 dependencies {
2     compile "joda-time:joda-time:2.3"
3     testCompile " junit:junit:4 .12"
4 }
```

So würde man z. B. in Gradle beschreiben, dass wir die Bibliothek Joda-time in Version 2.3 und zusätzlich zum Testen die Version 4.12 von Junit brauchen.

2.2 Weitere von Build-System unterstützte Funktionen

Die Ausführungen in diesem Kapitel basieren auf [1. „Java Build Tools: Part 1“]

Weitere Funktionen, die die Build-Systeme unterstützen, sind z. B.:
„Validieren und Kompilieren“: Beim Validieren wird die Projektstruktur nach Gültigkeit und Vollständigkeit geprüft. Beim Kompilieren wird der Quellcode kompiliert. Hier werden die zu kompilierenden Klassen vom Build-System an den Kompilierer übergeben. Dazu werden alle wichtigen Einstellungen in einem Build-Skript eingestellt.

„Testausführung“: Das Build-System kann mit einem Testframework (z. B. Junit, welches wir zuvor als Abhängigkeit angegeben haben) Testklassen verwenden, um zu überprüfen, ob der Code sich richtig verhält. Dazu muss im Build-Skript beschrieben werden, welche Klassen wir zum Testen verwenden möchten.
„Archiv-Erstellung“: Aus den zuvor kompilierten Klassen wird ein Archiv erstellt. Zusätzlich kann das Build-System noch weitere Dateien, die das Programm benötigt, in das Archiv kopieren.
„Verteilung“: Es ist möglich, dass Build-System anzuweisen, bei einem erfolgreichen Projekt-Build das erstellte Programm direkt zu deployen, also zu verteilen und auf einem anderen System zu installieren oder zum Download bereitzustellen.

3 Build-Systeme in Java

Es gibt eine Vielzahl von Build-Systemen, die für Java geeignet sind. Im Folgenden betrachten wir die drei meistverbreiteten davon: Maven, Ant+Ivy und Gradle.

3.1 Maven

Die Ausführungen in diesem Kapitel basieren auf [1. „Java Build Tools: Part 1“und 3. „Apache Maven“]

Maven ist bei Java Entwicklern das meist gewählte Build-System. Obwohl Maven schon 2002 veröffentlicht wurde, gewann es erst mit Version 2 an Beliebtheit.
Maven setzt das Prinzip "Konvention vor Konfiguration"über den gesamten Zyklus der Softwareerstellung durch. Das bedeutet, dass es von den Entwicklern nicht erfordert wird, den Build-Prozess selbst zu konfigurieren. Beim Erstellen eines Maven-Projekts wird eine Standard-Projektstruktur hinterlegt und wenn der Entwickler sich daran hält, sind nur wenige zusätzliche Konfigurationseinstellungen nötig, um den gesamten Lebenszyklus eines Softwareprojekts abzubilden. Der Lebenszyklus von Maven besteht aus verschiedenen Phasen, die die wesentlichen Aufgaben eines Build-Systems erfassen. Es müssen nicht alle Phasen durchlaufen werden. Standardmäßig sieht der Zyklus folgendermaßen aus:

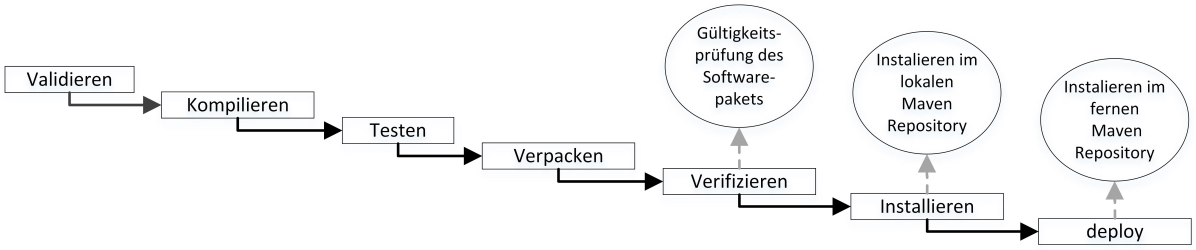

Abbildung 1: Mavens Lebenszyklus

Maven nutzt Plug-ins, um die verschiedenen Phasen aus seinem Lebenszyklus auszuführen. Z. B. für Kompilieren wird das compiler-Plug-in und für Verpacken das jar-Plug-in verwendet. Aber es gibt auch viele andere Plug-ins, die den Lebenszyklus noch wesentlich erweitern können. Eine revolutionäre Neuigkeit, die mit Maven entstanden ist, war die Möglichkeit Abhängigkeiten über das Netzwerk herunterzuladen, was eine sehr aufwändige Aufgabe war, wenn man in größeren Projekten arbeitete und alles manuell an verschiedenen Orten finden musste.

3.2 Ant+Ivy

Die Ausführungen in diesem Kapitel basieren auf [7. „8 Build-Tools im Vergleich“]

Ant wurde im Jahr 2000 entwickelt und ist dem Build-System Make sehr ähnlich, welches in Unix-Systeme verwendet wird. Mittlerweile ist Ant nicht mehr so populär, wird aber nach wie vor weiterentwickelt. Ant bietet sehr große Kontrolle über den Build-Prozess. Der Entwickler ist nicht gezwungen, eine bestimmte Verzeichnisstruktur zu nutzen, aber dafür muss er explizit definieren, wo alles sich befindet. Z. B. es muss definiert werden, wo die Java-Dateien zu finden sind oder wo die schon kompilierten Java-Klassen

gespeichert werden müssen. Zusammen mit der Verwendung von XML für seine Build-Skripte führt das sehr schnell zu sehr langen Konfigurationsdateien. Ein großer Unterschied zu Maven ist, dass Ant selbst keine Verwaltung von Abhängigkeiten bietet. Andererseits kann Ant, ähnlich wie Maven, durch Plug-ins erweitert werden. Ein Beispiel dafür ist das Plug-in Apache Ivy, welches ein agiles System zur Verwaltung von Abhängigkeiten ist.

3.3 Gradle

Die Ausführungen in diesem Kapitel basieren auf [1. „Java Build Tools: Part 1“und 6. „Java Build Tools: Ant vs Maven vs Gradle“]

Als drittes Build-System betrachten wir das neuere Gradle, welches 2012 veröffentlicht wurde. Gradle ist als Build-System für Projekte in Java, Groovy und Scala gedacht und auch besonders geeignet für Projekte, die aus mehreren Teil-Projekten bestehen.
Einer der Punkte, die Gradle am stärksten von den anderen Build-Systemen unterscheidet, ist, dass es nicht XML verwendet, sondern seine eigene DSL (Domänenspezifische Sprache) hat, die auf Groovy basiert ist. Groovy ist eine Skriptsprache für die Java Virtual Machine, die Java-ähnlichen Syntax hat. Auf die Vor- und Nachteile davon wird genauer im Abschnitt 4.1 ("XML vs DSL") eingegangen. Ein Vorteil, den Gradles DSL hat, ist, dass es sich beim Build-Skript um ausführbaren Code handelt. Somit muss man, um das Build-Skript zu verwenden, nicht unbedingt Gradle installiert haben, sondern es reicht ein nur wenige kb großer Wrapper. Mit Gradle hat man versucht, die Vorteile von Ant und Maven zu vereinen und aus ihren Fehlern zu lernen. So hat Gradle z.B. die Flexibilität von Ant, so dass man das Build-System an das Projekt anpassen kann und sich nicht nach dem Build-System richten muss, wie bei Maven. Dennoch enthält Gradle die Konventionen von Maven, so dass bei einem einfachen Java-Projekt, das diese Konventionen einhält, nur die Zeile **apply plugin: 'java'** für ein vollständiges Build-Skript genügt.
Komplexere Projekte mit vielen Unterprojekten brauchen ihre Zeit für den Build-Prozess. Um nicht mehr Zeit zu verbrauchen als unbedingt nötig, kann der Build-Prozess auf mehreren CPU Kernen laufen oder sogar auf mehreren Computern bzw. Servern. Zusätzlich versucht es, die Build-Zeit so klein wie möglich zu halten. So wird z.B. nur der geänderte Teil neu gebaut.
Gradle gewinnt immer mehr an Beliebtheit. Das wohl bekannteste Projekt, das Gradle verwendet, ist Android von Google, aber auch Hibernate, Grails, Groovy und Spring-Integration nutzen Gradle als Build-System.
Gradles Verwaltung von Abhängigkeiten ähnelt stark der von Maven. So ist es möglich, Maven-Repositories zu verwenden, aber auch der Zugang zu Ivy-Repositories bleibt Gradle nicht verwehrt. Außerdem kann ein Projekt, wie bei Maven, von anderen Bibliotheken oder sogar ganzen Projekten abhängig sein.
Gradle ist auch mit einer großen Menge von Plug-ins erweiterbar. Die Plug-ins für die Programmiersprachen Java, Groovy, Scala, und C++ werden von Haus aus mitgebracht. Mehr dazu kommt in Abschnitt 4.3 "Plug-ins".

4 Tiefer Einblick in die Arbeit mit den drei Build-Systemen sowie ihre Vor- und Nachteile

Die Ausführungen in diesem Kapitel basieren auf [1. „Java Build Tools: Part 1“, 2. „Java Build Tools: Part 2“, 4. „Building Java Projects with Maven“und 5. „Building Java Projects with Gradle“]

In den folgenden Unterpunkten gehen wir ins Detail, um zu zeigen, wie die Build-Skripte von Maven, Ant+Ivy und Gradle aussehen und vertiefen uns in die Vor- und Nachteile sowie die wesentlichen Unterschiede der drei Build-Systemen.

4.1 XML vs DSL

Die hier betrachteten drei Build-Systeme für Java können in zwei Kategorien geteilt werden. Zur ersten Kategorie gehören die Build-Systeme, die XML verwenden, und zur zweiten die Build-Systeme, die DSL verwenden. XML ist eine streng strukturierte und stark standardisierte Sprache, die aus dem Jahr 1996 stammt. Aufgrund der standardisierten Natur ist es einfach von Maschinen zu verarbeiten. Zusätzlich gibt

es zahlreiche Bibliotheken, die mit XML umgehen können, sodass wir nicht alles selber implementieren müssen. Das liegt daran, dass Ant schon vor über 20 Jahren erfunden wurde und es eine gute Chance besteht, dass andere Entwickler mittlerweile die gleichen Probleme wie uns hatten.
Jedoch hat die hohe Standardisierung seinen Preis. Vergleichen wir XML mit DSL, fällt einem auf, dass man nur einen Bruchteil der Zeilen schreiben muss, um dasselbe Ergebnis zu erhalten. DSLs sind für ein bestimmtes Problemfeld entworfen und daher kürzer, weil sie alle für die Domäne irrelevante Probleme nicht darstellen können. DSL lässt sich auch eine beliebige Notation definieren. Z.B. Gradles Groovy weist gewisse Ähnlichkeiten zu Java auf, deshalb ist sie für Java Entwickler leicht zu lernen. Vieles, wofür man bei XML auf Plug-ins zurückgreifen muss, kann bei DSL direkt erledigt werden. Andererseits ist DSL durch seine Kompaktheit schwerer zu interpretieren und verlangt gewisse Erfahrung.

4.2 Build-Skripte

Um den Build-Prozess zu automatisieren, benötigen die Build-Systeme eine formale Beschreibung der durchzuführenden Funktionsaufrufe sowie der Abhängigkeiten dieser Aufrufe untereinander in der Form eines Skripts. Betrachten wir das folgende Programm:

```
1  package hello;
2  import org.joda.time.LocalTime;
3
4
5  public class HelloWorld {
6      public static void main(String[] args) {
7          LocalTime currentTime = new LocalTime();
8          System.out.println("The current local time is: " + currentTime.now());
9          Greeter greeter = new Greeter();
10         System.out.println(greeter.sayHello());
11     }
12 }
```

```
1  package hello;
2
3  public class Greeter {
4      public String sayHello() {
5          return "Hello world!";
6      }
7  }
```

Das Programm gibt zuerst das Datum und die lokale Zeit zurück und danach den String "Hello World". Für die Rückgabe des Datums und der Zeit wird nicht das java.time-Bibliothek, sondern die LocalTime-Klasse aus der joda.time-Bibliothek verwendet. Anhand dieses Beispiels werden wir die Skripte von den drei Build-Tools vergleichen.

4.2.1 Build-Skript für Maven

Jedes Maven-Projekt hat eine Standardverzeichnisstruktur. Wenn der Entwickler diese Struktur nicht ändert, müssen die Pfadnamen nicht spezifiziert werden. So wird die Konfigurationsdatei pom.xml (Project Object Model) nicht unnötig belastet. Innerhalb der Standardverzeichnisstruktur wird das Projekt in zwei Teile geteilt. In einem Ordner mit dem Namen „/src“ werden alle Dateien gespeichert, die als Eingabe für den Verarbeitungsprozess dienen. In einen zweiten Ordner, „/target“ kommen automatisch alle erzeugten Dateien. In beiden Ordnern werden zusätzliche Unterverzeichnisse erstellt, um die Dateien weiter zu differenzieren. Die folgende Struktur zeigt einige der wichtigsten Verzeichnisse:

src: alle Eingabedateien
src/main: Eingabedateien für die Erstellung des eigentlichen Produkts
src/main/java: Java-Quelltext
src/main/resources: andere Dateien, die für die Übersetzung oder zur Laufzeit benötigt werden, beispielsweise Java-Properties-Dateien
src/test: Eingabedateien, die für automatisierte Testläufe benötigt werden
src/test/java: JUnit-Testfälle für automatisierte Tests
target: alle erzeugten Dateien

target/classes: kompilierte Java-Klassen

Um Maven zu benutzen, muss es zuerst installiert werden. Mit dem Befehl mvn -v kann geprüft werden, ob und welche Version auf dem Rechner vorinstalliert ist. Nach der Installation, muss ein Maven-Projekt erstellt werden. Für unser Programm verwenden wir die Standardverzeichnisstruktur von Maven. Deshalb müssen wir als erstes in dem Projektordner folgendes Unterverzeichnis hinterlegen:
src/main/java/hello/

Das Maven–Projekt wird danach in einer XML-Datei mit dem Namen pom.xml konfiguriert. Dort werden alle Informationen für das Softwareprojekt, das von Maven unterstützt wird, gespeichert. Diese Datei nutzt die Konventionen von Maven, hat ein standardisiertes Format und muss in dem Root-Ordner unseres Softwareprojekts gespeichert werden. Die einfachste Version einer solchen Datei hat die folgende Struktur:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4      <modelVersion>4.0.0</modelVersion>
5      <groupId>org.proseminar.localTime</groupId>
6      <artifactId>nameOfOurJar</artifactId>
7      <packaging>jar</packaging>
8      <version>0.1.0</version>
9      <build>
10         <plugins>
11             <plugin>
12                 <groupId>org.apache.maven.plugins</groupId>
13                 <artifactId>maven-shade-plugin</artifactId>
14                 <version>2.1</version>
15                 <executions>
16                     <execution>
17                         <phase>package</phase>
18                         <goals>
19                             <goal>shade</goal>
20                         </goals>
21                         <configuration>
22                             <transformers>
23                                 <transformer
24                                     implementation="org.apache.maven.plugins.shade.resource.
25                                                                     ManifestResourceTransformer">
26                                     <mainClass>hello.HelloWorld</mainClass>
27                                 </transformer>
28                             </transformers>
29                         </configuration>
30                     </execution>
31                 </executions>
32             </plugin>
33         </plugins>
34     </build>
35 </project>
```

Am Anfang werden die folgende Elemente für die Konfiguration des Projekts angegeben:

<modelVersion>: Die Version der pom-Datei
<groupId>: die Gruppe oder die Organisation, zu der das Projekt gehört.
<artifactId>: der Name, der für das Projektartefakt (z.B. JAR oder WAR-Datei) verwendet wird.
<version>: Version von dem Projekt, das gebaut wird
<packaging>: wie soll das Projekt verpackt werden – JAR oder WAR

Nach diesen Voreinstellungen ist Maven bereit, den Build-Prozess auszuführen. Aber für unser Programm wird ein weiterer Schritt gebraucht. Grund dafür ist, dass die joda.time.LocalTime-Klasse für die Rückgabe der lokalen Zeit verwendet wird und sie noch nicht in unserem pom.xml als Abhängigkeit deklariert ist. Um das zu tun, muss unsere pom-Datei mit folgendem Code erweitert werden:

```
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.3</version>
</dependency>
```

Unter Version kann ein Intervall eingegeben werden, z.B. [2.0,3.0], damit Maven von mehreren Versionen die Passende automatisch auswählt. Problematisch wird es jedoch, wenn zwei Module Versionsbereiche verlangen, die sich nicht überschneiden. In dem Fall bricht Maven mit einer Fehlermeldung ab.
Standardmäßig sind alle Abhängigkeiten als compile-Abhängigkeiten definiert. Das bedeutet, dass sie zur compile-Zeit vorhanden sind. Zusätzlich können mit dem <scope>-Element andere Arten definiert werden, wie z. B. : <provided > für Abhängigkeiten, die ähnlich zu compile-Abhängigkeiten sind, aber die zur Laufzeit von Laufzeitumgebung bereitgestellt werden, oder <test> für Abhängigkeiten, die zum Kompilieren und zum Ausführen von Tests gebraucht werden, aber nicht zur Laufzeit.
Jetzt können verschiedene Ziele aus dem Build-Lebenszyklus ausgeführt werden. Mit **mvn compile** kann unser Code kompiliert werden. Nach dem Kompilieren wird der Ordner „target“ in dem Root-Verzeichnis des Projektes automatisch erstellt. Im „target“ findet man einen Ordner mit dem Namen „classes“, der alle .class-Dateien enthält. .class-Dateien sind die kompilierten Javaklassen unseres Projektes. Mit **mvn package** wird eine jar-Datei in dem target-Ordner hinterlegt. Eine jar-Datei ist eine Zip-Datei, die vor allem verwendet wird, um Java Anwendungen und Bibliotheken zu vertreiben. Um jar-Dateien ausführbar zu machen, kann Maven das plugin shade verwenden. Dieses plugin muss auch in der pom-datei spezifiziert werden.
Betrachten wir die folgende Testklasse, die wir unter **src/test/java/hello** speichern werden:

```
package hello;
import static org.hamcrest.CoreMatchers.containsString;
import static org.junit.Assert.*;
import org.junit.Test;
public class GreeterTest {
    private Greeter greeter = new Greeter();
    @Test
    public void greeterSaysHello() {
        assertThat(greeter.sayHello(), containsString("Hello"));
    }
}
```

Wollen wir den Test ausführen, brauchen wir zusätzlich noch die junit-Abhängigkeit in der pom.xml folgender Weise zu definieren:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Danach kann man mit **mvn test** den Test laufen lassen. Maven nutzt das Surfire-Plug-in zum Kompilieren und ausführen von allen Klassen, die unter **src/test/java** mit dem Namen ***Test** gespeichert sind.

4.2.2 Build-Skript für Ant + Ivy

Wie Maven verwendet Ant XML, um sein Build-Skript zu konfigurieren. Alle Konfigurationseinstellungen werden in einer build.xml-Datei gespeichert. Anders als Maven hat Ant keine Standardverzeichnisstruktur, d. h. der Entwickler muss alle Pfadnamen selber spezifizieren. Für unser Ant-Skript werden wir die gleiche Verzeichnisstruktur wie bei Maven hinterlegen. Grund dafür ist, dass eine Differenzierung zwischen den Quelldateien und den neu erzeugten Dateien gewünscht ist. So werden auch die Unterschiede zwischen den beiden Skripten leichter erkennbar.
Das Ant-Skript ist in verschiedenen Targets (dt. Ziele) geteilt. Targets sind vergleichbar mit den Funktionen in Programmiersprachen. Sie decken alle bei der Arbeit mit einem Softwareprojekt anfallenden Tätigkeiten ab. Z. B. das Kompilieren, Testen, die Erstellung von JAR-Dateien, Bereinigung der Projektordner und das Herunterladen von Abhängigkeiten werden als verschiedenen Targets von dem Entwickler

selbst implementiert. Wir haben eine build.xml-Datei für unseres kleines Projekt erstellt:

```
<project basedir="." default="run" name="LocalTime" xmlns:ivy="antlib:org.apache.ivy.ant">
    <target name="compile" description="Compiling">
        <mkdir dir="target/classes"/>
        <javac srcdir="src/" destdir="target/classes" includeantruntime="false"
      classpathref = "classpath"/>
    </target>
    <target name="jar" depends="compile">
        <mkdir dir="target/"/>
        <jar destfile ="target/nameOfOurFirstJar.jar" basedir="target/classes">
         <zipgroupfileset dir= "./lib" includes ="*.jar"/>
            <manifest>
                <attribute name="Main-Class" value="hello.HelloWorld"/>
            </manifest>
        </jar>
    </target>
    <target name="run" depends ="jar">
        <java jar="target/nameOfOurFirstJar.jar" fork="true"/>
    </target>
   <path id = "classpath">
      <fileset dir = "${basedir}/">
        <include name = "lib/*.jar"/>
      </fileset>
   </path>
   <target name="junit" depends="compile">
      <mkdir dir="target/test-classes/hello"/>
      <junit printsummary="yes" haltonfailure="no">
         <classpath refid="classpath"/>
         <classpath location="target/classes" />
         <test name="hello.GreeterTest"
            haltonfailure="no" todir="target/test-classes/hello/">
            <formatter type="plain" />
            <formatter type="xml" />
         </test>
      </junit>
   </target>
   <target name="clean">
        <delete dir="target"/>
    </target>
   <target name="resolve" description="Retrieve dependencies with Apache IVY">
      <ivy:retrieve type="jar" />
   </target>
</project>
```

Unter **<project>** findet man eine Beschreibung des Projekts, das als Defaultziel vordefinierte Target, sowie auch einen Namensraum, um Plug-ins wie Ivy leicht zu importieren. Danach werden alle benötigten Targets definiert. Das compile-Target wird zum Kompilieren des Codes aufgerufen. Dort wird erst der Ordner definiert, wo alle .class-Dateien gespeichert werden. Für unseres Beispiel nutzen wir das gleiche Verzeichnis wie bei Maven: „target/classes“. Der Vorteil von Ant ist, dass die Verzeichnisstruktur frei gewählt werden kann. Der Nachteil ist natürlich, dass man sie selber definieren muss und sie nicht wie bei Maven automatisch erstellen lassen kann. Unter **<javac>** sind die Quell- und Zielverzeichnisse definiert. Das zweite Target ist "jar". Wie der Name schon sagt, wird hier die jar-Datei konfiguriert. Quell- und Zielordner werden auch eingegeben. Mit **<Zipgroupfileset>** werden alle jar-Dateien aus dem lib-Ordner gruppiert und in der jar-Datei inkludiert. In dem lib-Ordner befinden sich alle Abhängigkeiten, die heruntergeladen wurden. Das **<run >**-Target ist für das Ausführen des kompilierten Code verantwortlich. Man sieht, dass run abhängig von jar und jar wiederum abhängig von compile ist. Dies bedeutet, dass beim Aufruf von run erst die compile- und jar-Targets ausgeführt werden und danach das run-Target. Ähnlich ist auch das **<junit >**-Target aufgebaut. Hier wird zusätzlich definiert, wo und mit welchem Dateityp die erzeugten Reports von dem ausgeführten Test gespeichert werden müssen. Haltonfailure besagt, dass der Test gestoppt werden muss, wenn er fehlschlägt. Das **<clean>**-Target bereinigt den Ordner, in dem sich alle erzeugten Dateien befinden. Letztendlich kommen wir zu der Verwaltung von Abhängigkeiten. Standardmäßig ist es für Ant möglich, Abhängigkeiten aus lokalen und Maven Central Repositories (dt. Quellen) zu laden. Unsere zwei Abhängigkeiten sind in Maven Central Repository verfügbar. Um die Abhängigkeiten zu laden, definieren wir das **<resolve >**-Target, wo **<ivy: retrieve type="jar“ >** angibt, nur die jar-Dateien zu speichern.
Alle benötigten Abhängigkeiten werden in einer zusätzlichen ivy.xml-Datei aufgelistet:

```
<ivy-module version="2.4">
    <info organisation="joda-time" module="LocalTime" />

    <dependencies>
        <dependency org="junit" name="junit" rev="4.12" />
        <dependency org="joda-time" name="joda-time" rev="2.3" />
    </dependencies>
</ivy-module>
```

Das Root-Element **ivy-module** gibt an, welche Version von Ivy verwendet wird. Danach folgt ein info-Tag mit Information für das Modul, für welches die Abhängigkeiten definiert werden. Es werden nur die Organisation, zu der das Modul gehört, und der Name des Moduls, das von Ivy beschrieben wird, eingegeben. Die Abhängigkeiten werden ähnlich zu Maven definiert. Mit org wird die Organisation eingegeben. Danach kommt der Name des Moduls und am Ende ist die benötigte Version unter dem Wert von rev gespeichert. Man kann hier auch Intervalle für die gesuchte Version eingeben. Wenn zwei Module verschiedene Versionsbereiche verlangen, die sich nicht überschneiden, wird Ant, anders als bei Maven, die Version mit der höchsten Versionsnummer nehmen.

Es ist deutlich zu erkennen, dass es bei Ant+Ivy wesentlich aufwändiger ist, alles zu konfigurieren. Vieles, was z. B. von Maven automatisch übernommen wird, muss bei Ant manuell konfiguriert werden, aber dafür hat der Entwickler die totale Kontrolle über den Erstellungsprozess.

4.2.3 Build-Skript für Gradle

Das Gradle-Skript ist eine build.gradle-Datei, in der alle nötige Konfigurationen eingetragen sind. Die Datei muss, wie bei den anderen zwei Build-Systemen, in dem Projektordner gespeichert werden. Anders als Maven und Ivy nutzt Gradle eine DSL für sein Skript. Mit DSL könnte nur eine Zeile Code ein gültiges Skript sein, wie z.B. **apply plugin: java**. Diese Zeile Code erweitert wesentlich die Funktionalität von Gradle. Es ist jetzt möglich, Tests laufen zu lassen, Java-Projekte zu unterstützen und JavaDoc zu erstellen. Das Gradle-Skript für unser Programm sieht so aus:

```
apply plugin: 'java'
apply plugin: 'application'
mainClassName = 'hello.HelloWorld'
sourceCompatibility = 1.8
targetCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    compile "joda-time:joda-time:2.3"
    testCompile " junit:junit:4 .12"
}
jar {
    baseName = 'nameOfOurFirstJar'
    version = '0.1.0'
}
```

Man erkennt sofort, dass das Skript von Gradle viel kürzer ist als die anderen zwei. Dies ergibt sich aus der deutlich kompakteren DSL, die Gradle für sein Skript benutzt. Die erste Zeile enthält, wie schon erwähnt, apply plugin: 'java'. In der zweiten Zeile steht apply plugin: 'Application' gefolgt von mainClassName = 'hello.HelloWorld', welche unsere Applikation ausführbar macht. Unter sourceCompatibility wird eingegeben, welche Java-Version für das Kompilieren des Codes verwendet werden soll. Entsprchechend wird unter targetCompatibility eingegeben, in welcher Java-Bytecode die erzeugten .class-Dateien gespeichert werden sollen. Die Abhängigkeiten in dem Projekt müssen wie bei Maven und Ant+Ivy zuerst deklariert werden. Gradle hat kein eigenes Repository. D.h., um externe Abhängigkeiten zu verwalten, muss mindestens ein Repository eingegeben werden. Der repositories-Block, in der Build-Datei gibt an, dass Gradle in unserem Fall die Maven Central Repository als Quellenbibliothek nutzt, um die Abhängigkeiten zu verwalten.

In dem Abhängigkeitsblock werden die Abhängigkeiten entsprechend deklariert. In unserem Fall haben wir zwei Einträge: einen für Joda Time und einen für junit. Von rechts nach links gelesen bedeutet der Eintrag für Joda Time, dass wir die Version 2.3 der Joda Time-Bibliothek in der Joda Time-Gruppe brauchen.

Für junit benötigen wir die Version 4.12 aus der junit-Gruppe. Auch hier sind Intervalle möglich. Ähnlich zu Ant nimmt Gradle die Version mit der höchsten Versionsnummer, falls die Versionsbereiche zweier Module sich nicht überschneiden. Ähnlich zu Maven können verschiedene Typen von Abhängigkeiten deklariert werden. Joda Time ist definiert als compile-Abhängigkeit und junit als testCompile.
In dem jar-Block wird der Name und die Version für unser Jar-Artefakt definiert.
Nach dem Ausführen mit dem Befehl **gradle build** wird in dem Quellverzeichnis unseres Projektes ein Ordner mit dem Namen Build erstellt. Dort findet man andere Unterordner, wie classes, reports und libs. Der classes-Ordner enthält die .class-Dateien des kompilierten Java-Code. In dem reports-Ordner werden Reports gespeichert, die während des Build-Prozesses erzeugt wurden, wie z. B. test reports. In dem libs-Ordner befinden sich die generierten Jar-Dateien.
Gradle ist aufgrund der DSL definitiv entwickler-freundlicher. Die Verzeichnisstruktur muss nicht explizit konfiguriert werden und die Abhängigkeiten werden von Gradle selbst verwaltet.

4.3 Plug-ins

An diesen Punkt wollen wir uns ansehen, wie viele Plug-ins für das jeweilige Build-System verfügbar sind und wie es uns möglich ist, das Build-System an unsere Wünsche anzupassen, wenn kein fertiges Plug-in verfügbar ist, das unseren Bedürfnissen genügt.
Für Maven sind Plug-ins von besonderer Wichtigkeit, denn aufgrund der sehr starken Standardisierung ist es in der Maven pom.xml selber kaum möglich, seine projektspezifischen Wünsche zu realisieren. In Maven ist es eigentlich nur möglich, seine eigenen Wünsche durch Plug-ins unterzubringen. Da es Maven jedoch schon seit geraumer Zeit gibt, ist das kein allzu großes Problem. Über die Zeit haben sich eine Vielzahl an guten Maven Plug-ins angesammelt. Mit dem AntRun-Plug-in ist es sogar möglich Ant-Tasks auszuführen. Ferner können wir mit diesem Plug-in sogar Skripte in die pom.xml schreiben. Uns ist es aber auch relativ einfach möglich, in Java selber Plug-ins für Maven zu schreiben. Eine einfache Anleitung dazu findet man im Handbuch-Abschnitt der Mavens Dokumentation[1].
Wie auch für Maven stehen uns, schon alleine aufgrund des Alters von Ant, viele Plug-ins zur Verfügung. Zusätzlich bieten Ant Targets schon von sich aus viele viele Anpassungsmöglichkeit. Natürlich ist es uns auch möglich, eigene Plug-ins oder Tasks zu schreiben und zusätzlich hat Ant auch den „exec“-Task. „Exec“ steht für Execute. Dieser Task ermöglicht uns also, mit Ant jedes ausführbare Programm zu starten und Argumente an dieses zu übergeben. Wenn wir aber z. B. einen Server oder ähnliches mit unseren neu gebauten Programmen versorgen möchten, ist es sehr wahrscheinlich, dass wir Ant nicht mal selber erweitern müssen, denn oft gibt es hier schon offizielle Unterstützung in Form eines Plug-ins, welches entweder mit der Software kommt oder auf der Herstellerseite verfügbar ist.
Gradle ist sehr modular programmiert, sodass Teile der Kern-Funktionalität als Plug-in realisiert sind. Diese Plug-ins werden mit Gradle selber ausgeliefert und müssen nur noch aktiviert werden. Zu diesen Plug-ins gehören unter anderem die Java- und Eclipse-Plug-ins. Es gibt leider noch nicht allzu viele Plug-ins für Gradle, da es besonders im Vergleich zu Ant oder Maven noch relativ jung ist. Das kann Gradle jedoch dadurch ausgleichen, dass wir mit Gradle sehr leicht unsere eigenen Plug-ins erstellen können. Hierzu müssen wir nicht einmal ein eigenes Plug-in-Projekt erstellen. Groovy-Code kann direkt in das Build-Skript geschrieben werden oder, um eine bessere Übersicht zu behalten, ist es auch möglich, einen BuildSrc-Ordner anzulegen und dort die Groovy-Dateien zu lagern. Für größere Plug-ins ist es natürlich auch möglich, diese in ein separates Projekt auszulagern. Dadurch ist es auch anderen Entwicklern möglich, an dem Plug-in zu arbeiten, auch wenn sie nicht an dem Projekt arbeiten, aus dem das Plug-in vielleicht eigentlich entstanden ist.

4.4 Integration

Im Folgenden wollen wir betrachten, inwieweit unsere Build-Systeme in IDEs sowie App- und CI-Server integriert sind.
Maven, Ant und Gradle sind bestens in die großen IDEs integriert. So unterstützen z. B. Eclipse, Intellij und NetBeans alle Funktionen unserer drei Build-Systeme, entweder durch ein Plug-in oder schon von Haus aus.
Bei der App Server-Integration genießen sowohl Ant als auch Maven den Altersvorteil, den sie auch bei den Plug-ins hatten. So haben Ant und Maven Unterstützung in den meisten App-Server. Wollen wir mit Gradle unsere Web App auf einem Server Installieren, so sieht unsere Auswahl schon etwas spärlicher aus und wir müssen uns mit Tomcat oder Jetty begnügen. Bei unseren beiden anderen Build-Systemen

[1] http://maven.apache.org/guides/plugin/guide-java-plugin-development.html+

können wir zusätzlich noch auf JBoss, GlassFish, WebLogic und mehr zurückgreifen.
CI steht für „Continuous intergration“ und ist ein nützliches Werkzeug, um die Programmqualität zu sichern, beziehungsweise um diese zu verbessern. Auf CI-Servern können beispielsweise die Testausführungen aus Abschnitt 2.3 ausgeführt werden. Diese ist besonders sinnvoll, sobald man viele komplexere Tests hat, die zu viel Zeit auf einem normalen Computer verbrauchen würden. So können die Tests über Nacht auf dem CI-Server durchlaufen. Das ist jedoch bei Weitem nicht alles, mit dem uns der CI Server helfen kann. So kann der CI-Server die fertigen Builds für alle Nicht-Entwickler bereitstellen oder Softwaremetriken, wie Code-Coverage messen.
Nun ist die Frage, inwieweit können wir CI-Server mit unseren drei Build-Systemen verwenden.
Maven funktioniert mit den meisten CI-Servern, ohne das man selber etwas tun muss. So gibt es entweder Plug-ins oder der Server wird mit Unterstützung ausgeliefert.
Für Ant und Gradle ist die Unterstützung nicht ganz so weit ausgeweitet wie für Maven. Dies ist für beide Build-Systeme jedoch nicht sonderlich tragisch, da Gradle seinen eigenen Wrapper mitbringen kann, der sogar verwendet werden kann, um Gradle installieren zu lassen. Auch wenn Ant wesentlich weiter als Gradle verbreitet ist, ist auch hier volle Unterstützung nicht nötig, da zum Starten von Ant nur ein einzelnes CLI-Kommando benötigt wird.

4.5 Repositories

Repositories bieten die Möglichkeit, die vom Build-System gebauten Archive zu veröffentlichen, ohne das ein ganzer eigener CI-Server nötig ist.
Maven hat die Möglichkeit in alle Repositories-Archive hochzuladen, die dies akzeptieren. Das können sowohl öffentlich zugängliche Repositories sein, als auch Private, die man zuvor selbst erstellt hat. Bekannte öffentliche Maven-Repositories sind Apache, Ibiblio, Codehaus oder Java.Net. Mittels Software wie Apache Archiva, Nexus oder Artifactory lassen sich gekaufte oder selbst entwickelte Bibliotheken und Frameworks firmenweit verbreiten.
Möchte man mit Ant Archive auf Repositories hochladen, greift man wieder auf Ivy zurück. Ivy kann auf Maven Central zugreifen. Möchte man darüber hinaus auf weitere Repositories zugreifen, muss man diese in einer ivysettings.xml-Datei definieren. Diese Datei kann entweder im home-Verzeichnis des Benutzers platziert werden oder es kann ein Pfad oder eine URL zu ihr angegeben werden.
Gradle besitzt noch keine eigene Möglichkeit, gebaute Projekte zu teilen. Jedoch sind sowohl 'ivy-publish' als auch 'maven-publish' als Plug-ins verfügbar.

5 Zusammenfassung

Abschließend können wir festhalten, dass es allgemein kein bestes Build-System gibt. Hat man beispielsweise viele kleine Projekte mit kurzer Lebenszeit, so ist Gradle besonders geeignet. Die durch die DSL sehr kompakte Build-Scripte, ermöglichen einen schnellen Start in neue Projekte. Hat man jedoch nur wenige große Projekte und ist auf viele Werkzeuge angewiesen, so ist Maven, auf Grund der weiten Integration in Entwicklungswerkzeugen, besser geeignet. Ant ist von Anfang an leicht zu verwenden, deshalb eignet sich Ant für Leute, die noch keine Erfahrung mit Build-Systemen hatten. Wir sollten also immer Abwägen was für unsere Projekte am wichtigsten ist und welches Build-System das bieten kann.

6 Quellen

1. Java Build Tools: Part 1, unter https://zeroturnaround.com/rebellabs/java-build-tools-part-1-an-introductory-crash-course-to-getting-started-with-maven-gradle-and-ant-ivy, [Stand: 13.06.2017]
2. Java Build Tools: Part 2, unter https://zeroturnaround.com/rebellabs/java-build-tools-part-2-a-decision-makers-comparison-of-maven-gradle-and-ant-ivy, [Stand: 13.06.2017]
3. Apache Maven, unter https://de.wikipedia.org/wiki/Apache_Maven, [Stand: 13.06.2017]
4. Building Java Projects with Maven, unter https://spring.io/guides/gs/maven, [Stand: 13.06.2017]
5. Building Java Projects with Gradle, unter https://spring.io/guides/gs/gradle, [Stand: 13.06.2017]
6. Java Build Tools: Ant vs Maven vs Gradle, unter https://technologyconversations.com/2014/06/18/build-tools, [Stand: 13.06.2017]
7. 8 Build-Tools im Vergleich: Ant – Buildr – Maven – Bazel – Buck – Gradle – Pants – sbt, unter https://jaxenter.de/8-build-tools-im-vergleich-ant-buildr-maven-bazel-buck-gradle-pants-sbt-41627, [Stand: 13.06.2017]