Die Maven-Alternative?

Apache Buildr

Wohin mit dem Logo?
Keine Verlinkung im Text.

Die Automatisierung der Build-Prozesse ist eine feine und wichtige Sache. In der Regel freut man sich, wenn ein Projekt Apache Maven verwendet und man dank der Konvention das Projekt sofort bauen kann. Doch mit der Zeit stellt sich in der Regel Ernüchterung ein: Der Build schlägt plötzlich fehl, weil ein Artefakt nicht mehr in den bekannten Repositories gefunden werden kann, die Ergebnisse sind nicht exakt reproduzierbar oder für eine kleine Erweiterung des Prozesses müssen erst aufwändig Plug-ins entwickelt und verteilt werden. Muss das so sein? Apache Buildr hat sich zu einer mächtigen Alternative entwickelt, indem es den deklarativen Ansatz von Maven mit dem imperativen einer Skriptsprache kombiniert.

von Tammo van Lessen

Die Ursprünge von Buildr [1] liegen in der BPEL-Engine Apache ODE [2]. ODE ist ein komplexes Middleware-Projekt, das sehr hohe Anforderungen an das Build-System stellt. So besteht es aus über fünfunddreißig Modulen, unterstützt neun Datenbanken, wird in drei verschiedenen Editionen paketiert und hängt von über hundertzwanzig Bibliotheken ab. Die Datenbankanbindung erfolgt wahlweise über OpenJPA oder Hibernate – beide müssen im Build-Prozess berücksichtigt werden. Die Datenbankskripte sollen nicht nur für eine Datenbank, sondern gleichzeitig für alle neun erzeugt werden. Zusätzlich werden XML-Parser und -Serializer mittels XMLBeans generiert und eigene Annotations-Prozessoren für die Codegenerierung verwendet. Dazu kommen weitere „Kleinigkeiten“, wie die Anforderung, dass alle ausgelieferten Textdateien, auch die generierten, die Apache-Lizenz im Kopf führen müssen.

„Maven Uncertainty Principle“

Nach den guten Erfahrungen, die das ODE-Projektteam mit Maven 1 gemacht hatte, war man optimistisch, auch diese Anforderungen mit Maven – diesmal in Version 2 – umsetzen zu können. Das war auch tatsächlich möglich, allerdings mit deutlich höherem Aufwand als erwartet. Für eine so einfache Aufgabe wie das Zusammenführen von zwei SQL-Dateien benötigt man beispielsweise 34 Zeilen XML. Die SQL-Skripte für alle Datenbanken zu erzeugen, war mit Maven und Plug-ins gar nicht möglich, sodass man an dieser Stelle auf Ant-Skripte ausweichen musste. Im Ergebnis war die Build-Logik von Apache ODE um insgesamt 6739 Zeilen XML-Code gewachsen, verteilt auf 53 Dateien. Die Verteilung auf mehrere voneinander abhängige *pom.xml*-Dateien muss man in Kauf nehmen, wenn man sein Projekt auf mehrere Module verteilen möchte. Man kann sich vorstellen, dass das auf Kosten der Wartbarkeit geht. Zudem sind die Konfigurationsoptionen verschiedener Plug-ins nicht immer selbsterklärend und ändern sich mitunter zwischen Plug-in-Versionen. In der Praxis führt dies zu schwer aufzuspürenden Problemen und zu nicht reproduzierbaren Builds. Scherzhaft sprach man sogar vom „Maven Uncertainty Principle“, in Anlehnung an Heisenbergs Unschärferelation.

Notausgänge

Dass dies besser gehen muss, steht außer Frage – aber wo genau liegt eigentlich das Problem? Maven verfolgt einen rein deklarativen Ansatz mithilfe einer XML-basierten DSL. Diese DSL ist allerdings ausschließlich in der Lage, das „Was“ zu beschreiben, nicht jedoch das „Wie“. Für die tatsächliche Implementierung der gewünschten Funktionalität sind die Plug-ins verantwortlich. Sie allein bestimmen also das „Wie“. Die einzige Möglichkeit, Einfluss darauf zu nehmen, ist die Konfiguration der Plug-ins (im Rahmen der angebotenen Funktionalität) oder ein Notausgang. Ein solcher könnten z. B. ein selbst geschriebenes Maven-Plug-in, der Aufruf eines Ant-Scripts oder, wie in Maven 1, ein Jelly-Script sein. Die Lösung kann also nur die bessere Verbindung der deklarativen und imperativen Ansätze sein, die eine nahtlose Integration von Notausgängen erlaubt [3].

Buildr hat sich genau das zum Ziel gesetzt und setzt dabei auf bewährte Mittel. So sollen die guten Eigenschaften von Maven, wie z. B. die Abhängigkeitsverwaltung, beibehalten werden. Als Basis für die DSL wird aber auf XML verzichtet und stattdessen Ruby eingesetzt. Dadurch hat man jederzeit die Möglichkeit, die Mächtigkeit der Skriptsprache als Notausgang zu verwenden, wenn die deklarativen Elemente nicht mehr ausreichen. Der Buildr-basierte Build von Apache ODE besteht nun nur noch aus 912 Zeilen Ruby-Code, der Übersichtlichkeit halber auf drei Dateien verteilt. Ein wichtiger Nebeneffekt: Der Build-Prozess ist sogar doppelt so schnell.

Abb. 1: Auf der Basis von Ruby setzt Buildr auf Rake auf

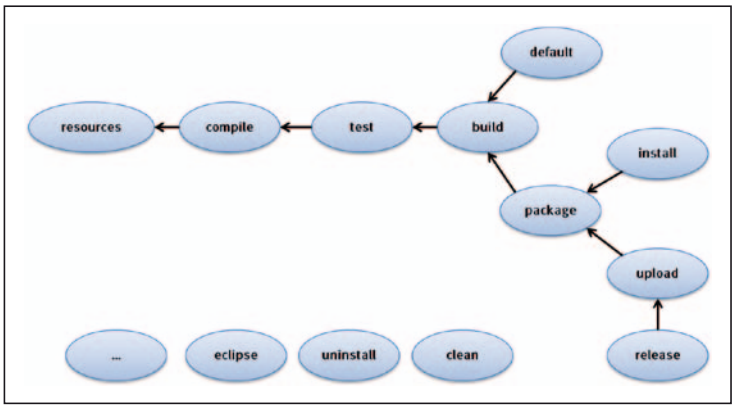

Abb. 2: Abhängigkeiten der wichtigsten globalen Tasks

Auf der Basis von Ruby setzt Buildr auf Rake auf (**Abb. 1**). Rake ist ein populäres Build-Werkzeug in der Ruby-Welt. Es operiert auf einem gerichteten azyklischen Graphen, um Abhängigkeiten zwischen Tasks zu definieren. Dafür stellt es eine einfache DSL zur Verfügung. Die Tasks selbst werden in Ruby implementiert und können dementsprechend komplexe Aufgaben bearbeiten. Zusätzlich können so genannte *FileTasks* kausale Abhängigkeiten zwischen Dateien definieren. Wird eine solche Abhängigkeit zwischen einer *.java*- und *.class*-Datei definiert, so weiß Rake, dass es die *.class*-Datei nur dann neu erzeugen muss, wenn die *.java*-Datei ein neueres Änderungsdatum als die *.class*-Datei hat.

Dennoch wurde Rake für die Build-Prozesse von Ruby-Projekten entwickelt. Um effizient Java-Projekte bauen zu können, liefert Buildr die nötigen Erweiterungen der DSL, um die wichtigsten Elemente eines Builds, nämlich Kompilieren, Testen, Paketieren und Releasen, deklarativ in der Skriptsprache verwenden zu können.

Die Kernkonzepte

Projektstruktur: Anders als Maven werden Multi-Modul-Builds nicht auf mehrere POMs verteilt, sondern in einer einzigen Build-Datei namens *buildfile* beschrieben. Dabei können Unterprojekte beliebig häufig verschachtelt sein. Buildr geht zunächst davon aus, dass die einzelnen Projekte der Maven-Konvention, also *src/main/java* für Java-Code, *src/main/resource* für Ressourcen, *src/test/java* für Java-Tests, *src/test/resources/* für Testressourcen etc. folgen. Es lassen sich aber auch andere Projektlayouts konfigurieren. Analog zu den GAV-Koordinaten der Maven-Projekte (Group, Artifact, Version) werden jedem Projekt eine *artifactId*, eine *groupId* und eine Version zugewiesen. Für jedes Projekt (und die Unterprojekte) definiert Buildr automatisch eine Reihe von Tasks, die den Lifecycles von Maven sehr nahe kommen:

- *clean*: löscht alle während des Builds erzeugten Dateien
- *compile*: kompiliert das Projekt und seine Unterprojekte
- *test*: testet das Projekt und seine Unterprojekte
- *build*: kompiliert und testet das Projekt und seine Unterprojekte
- *package*: erzeugt JARs, WARs, EARs, AARs, Zips oder OSGi Bundles
- *install*: kopiert die erzeugten Artefakte in das lokale Maven-Repository
- *upload*: lädt die erzeugten Artefakte in ein entferntes Maven-Repository
- *eclipse/idea*: erzeugt die Hilfsdateien, um die Projekte in Eclipse oder IDEA öffnen zu können
- *cc*: wartet kontinuierlich auf Änderungen der Quelldateien und startet dann automatisch die Kompilierung
- *release*: erzeugt ein Release mit Tag in der Versionskontrolle und lädt die Artefakte in ein Maven-Repository hoch
- *junit:report*: erzeugt die HTML-Reports für die JUnit-Testfälle

Abb. 2 zeigt die Abhängigkeiten der wichtigsten Tasks.

Abhängigkeiten: Eines der wichtigen Features von Maven ist die Möglichkeit, Projektabhängigkeiten automatisch verwalten lassen zu können. Buildr übernimmt dieses Feature und arbeitet mit den gleichen Repositories wie Maven zusammen. Die GAV-Koordinaten der Artefakte werden in der Form *group:id:type:version* angegeben und in normalen Ruby-Variablen definiert. Mit einigen Hilfsmethoden lassen sich hier elegante Schreibweisen finden, um komplexe Abhängigkeitsstrukturen einfach auszudrücken. Wichtig ist zu erwähnen, dass Buildr keine komplexe Abhängigkeitsauflösung wie Maven unterstützt. Zwar kann man mithilfe der *transitive()*-Methode die weiteren Abhängigkeiten herunterladen und verwenden, allerdings ist diese Funktionalität nur rudimentär implementiert. Das ist allerdings auch so gewollt, denn die automatische Auflösung der Abhängigkeiten hat auch bei Maven immer wieder zu Problemen geführt. Besser ist es, die Abhängigkeiten manuell festzulegen. Wer dennoch nicht auf dieses Feature verzichten möchte, kann die Aether- und Ivy-Plug-ins verwenden. Anders als bei Maven ist es jedoch auch sehr leicht möglich, lokale Bibliotheken, beispielsweise aus einem *libs*-Ordner im Projektverzeichnis, zu verwenden.

Bauen: Buildr unterstützt nicht nur Java, sondern auch Groovy, Scala und Ruby, zusammen mit ihren wichtigsten Testframeworks. So können Java-Projekte ohne weiteres Zutun mit JUnit, TestNG oder JBehave getestet werden. Die Kompilierung der Quelldateien lässt sich sehr leicht um zusätzliche Tasks, etwa zur Codegenerierung, erweitern. Ebenfalls Teil des Bauens ist das Kopieren und Verarbeiten der Ressourcen. Wie auch Ant und

Maven erlaubt es Buildr, die Dateien während dieses Schrittes zu manipulieren, beispielsweise, um die aktuelle Versionsnummer in das Produkt einzubinden.

Paketieren: Ohne weitere Plug-ins kann Buildr ZIPs, TARs, TGZs, JARs, WARs, AARs, EARs und OSGi Bundles erzeugen. Zudem stellt die DSL Tasks zum Erzeugen von Source-Distributionen und Javadoc-Archiven bereit. Die DSL erlaubt es auch, sehr einfach Einfluss auf den Inhalt der Archive nehmen zu können. Das folgende Codebeispiel erzeugt z. B. eine ZIP-Datei namens *api-docs-1.0.zip*, die die generierte Dokumentation sowie eine Readme-Datei enthält:

```
package(:zip).path("#{id}-docs-#{version}").tap do |path|
  path.include _('target/docs')
  path.include _('README')
end
```

Erweiterbarkeit: Eines der wichtigsten Entwurfsziele von Buildr war die Erweiterbarkeit, um das Schaffen von Notausgängen möglichst unkompliziert zu gestalten. Buildr bietet dazu verschiedene Ansätze. Zum einen lässt sich an jeder Stelle im Build File jederzeit Ruby-Code ausführen, zum anderen können beliebige Ant-Tasks ausgeführt werden. Diese lassen sich dann in wiederverwendbare Tasks kapseln und entweder in Form eines Gems als Plug-in verwenden oder aber direkt in das *tasks*-Verzeichnis legen und von dort aus verwenden. Die Liste solcher Tasks wächst ständig. Auf diese Weise lassen sich Werkzeuge wie Checkstyle, Cobertura, Emma, Sonar, ANTLR, XMLBeans, OpenJPA, Hibernate, Jetty, GWT etc. einfach einbinden.

Ein Blick in die Praxis

Obwohl Buildr sowohl mit Ruby als auch mit JRuby genutzt werden kann, ist die Verwendung von Letzterem zu empfehlen. Falls noch nicht vorhanden, muss eine aktuelle JRuby-Version installiert und in den Path aufgenommen werden. Das gelingt am besten mit RVM (unter Linux/Mac) oder Pik (unter Windows). Die Installation von Buildr erfolgt dann mithilfe von Gem:

```
gem install buildr
```

Mit *buildr --version* erfahren Sie, welche Version installiert wurde. Zur Drucklegung ist Version 1.4.9 aktuell. Wechseln Sie nun in ein Projektverzeichnis Ihrer Wahl. Für den Anfang sollte es kein zu kompliziertes Projekt sein. Rufen Sie Buildr auf und lassen Sie ein Build File erstellen. Buildr versucht nun anhand einer existierenden *pom.xml* oder der Verzeichnisstruktur zu erkennen, um was für ein Projekt es sich handelt. Das funktioniert meist nicht besonders gut, stellt aber einen guten Startpunkt dar. Ein funktionierendes Beispielprojekt können Sie unter [4]

auschecken, das soll uns nun auch als Grundlage dienen. Es handelt sich dabei um ein sehr vereinfachtes Multimodulprojekt, bestehend aus einem Modul für ein API (*api*) und einem Modul für die Implementierung (*impl*). Nachdem Buildr ein neues Build File erzeugt hat, versucht er, das Projekt direkt zu bauen. Das schlägt allerdings fehl, weil die Implementierung die Logback-Bibliothek verwendet und diese Abhängigkeit Buildr noch nicht bekannt ist.

Listing 1 zeigt, wie das vollständige Build File für dieses Projekt aussieht. Zu Beginn wird die aktuelle Versionsnummer für das Projekt festgelegt. Der Variablenname ist eine Konvention und wird von Buildrs Release-Task verwendet, um die nächsthöhere Version zu bestimmen und zu setzen. Danach wird das Maven-Central-Repository in die Liste der bekannten Repositories aufgenommen. Die Variable *LOGBACK* wird als Array definiert und referenziert die Core- und Classic-Artefakte der Bibliothek. Alternativ hätte man auch schreiben können:

```
LOGBACK = group('logback-classic', 'logback-core,
:under=>'ch.qos.logback', :version=>'1.0.7')
```

Danach wird das Wurzelprojekt definiert. Das DSL-Schlüsselwort *desc* gibt der Projekttask eine natürlichsprachliche Beschreibung, während *define* die Artefakt-ID (Multi-Java) als Parameter übergeben bekommt. In dem folgenden Block werden die Group-ID und die Versionsnummer gesetzt. Damit sind alle GAV-Koordinaten bestimmt. Die Unterprojekte *api* und *impl* „erben" diese Koordinaten, die Artefakt-IDs werden dabei zusammengesetzt. Dieses Verhalten kann aber umkonfiguriert werden. Als Nächstes wird das API-Projekt definiert. Dem Compiler wird übergeben, dass er Java-5-Bytecode erzeugen soll, danach werden ein JAR, sowie Javadocs und eine Source-Distribution erzeugt. Die Implementierung benötigt zur Compile-Zeit und zur Laufzeit die Logback-Bibliothek, deshalb wird sie in der *compile.with*-Direktive zusammen mit der Abhängigkeit zu dem API-Projekt angegeben. Die Methode *transitive()* berechnet aus den POMs der beiden Logback-Artefakte die transitiven Abhängigkeiten und übergibt sie dem Compiler. Danach werden die Tests ausgeführt und dann ein JAR sowie die Javadocs erzeugt.

Damit ist der Build-Prozess definiert. Mit *buildr -T* erhalten Sie eine Übersicht über die Tasks, die auf dem Projekt ausgeführt werden können. *buildr artifacts* beispielsweise lädt die referenzierten Artefakte aus dem Maven-Repository herunter. *buildr eclipse* erzeugt *.classpath*- und *.project*-Dateien für den Import in Eclipse.

Um den eigentlichen Build auszuführen, rufen Sie *buildr* auf. Daraufhin werden die beiden Unterprojekte kompiliert und die Tests ausgeführt. Möchten Sie nur das Implementierungsprojekt testen, können Sie in das *impl*-Verzeichnis wechseln und dort *buildr test* oder in dem Wurzelverzeichnis *buildr multi-java:impl:test* aufrufen.

Um die Projekte zu paketieren, rufen Sie *buildr package* auf. Danach finden Sie in den *target*-Verzeichnissen die JAR-Dateien. *buildr install* kopiert die Artefakte in Ihr lokales Maven-Repository.

Listing 1

```
# Version number for this release
VERSION_NUMBER = "1.0"

# Specify Maven 2.0 remote repositories here, like this:
repositories.remote << " http://repo.maven.apache.org/maven2/"

LOGBACK = ['ch.qos.logback:logback-classic:jar:1.0.7', 'ch.qos.logback:logback-core:jar:1.0.7']

desc "The Multi-java project"
define "multi-java" do

  project.version = VERSION_NUMBER
  project.group = "org.example"

  define "api" do
    compile.using(:source => '1.5', :target => '1.5')
    package(:jar)
    package(:javadoc)
    package(:sources)
  end

  define "impl" do
    compile.with(project("api"), transitive(LOGBACK))
          .using(:source => '1.5', :target => '1.5')
    test
    package(:jar)
    package(:javadoc)
  end
end
```

Fazit

Das Ziel dieses Artikels war es, Apache Buildr mit seinen grundlegenden Konzepten vorzustellen und dem Leser den Einstieg zu erleichtern. Es wurden die Probleme von rein deklarativen Build-Systemen diskutiert und mit Buildr eine Lösung präsentiert, die die deklarativen und imperativen Ansätze geschickt kombiniert. Wer sich intensiver mit dem Thema beschäftigen möchte, sei auf die gute Dokumentation auf der Projektwebseite verwiesen.

Tammo van Lessen arbeitet als Senior Consultant bei der innoQ Deutschland GmbH im Bereich SOA/BPM und promoviert im gleichen Bereich. Er ist PMC Chair Apache ODE und Member der Apache Software Foundation und hat aktiv an der Standardisierung von BPMN 2.0 mitgewirkt. Er ist Mitautor des Buchs „Geschäftsprozesse automatisieren mit BPEL".

Links & Literatur

[1] http://buildr.apache.org/

[2] http://ode.apache.org/

[3] Fowler, Martin: JRake. 2006. http://martinfowler.com/bliki/JRake.html

[4] https://github.com/vanto/buildr-jm-demo

[5] http://svn.apache.org/repos/asf/ode/trunk/Rakefile