03-2016 | Herbst | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Java wächst weiter

Microservices
Schnell und einfach implementieren

Container-Architektur
Verteilte Java-Anwendungen mit Docker

Java-Web-Anwendungen
Fallstricke bei der sicheren Entwicklung

4 191978 304903 03

iJUG Verbund

Neues von den letzten Releases

Sich schnell einen Überblick über bestehende Strukturen und deren Beziehungen verschaffen

Daten in unterschiedlichen Formaten in der Datenbank ablegen

Groovy und Grails – quo vadis?

Falk Sippach, OIO Orientation in Objects GmbH

Die Turbulenzen des Jahres 2015 haben Groovy und Grails zwar gut überstanden, inhaltlich gingen die Neuerungen der letzten Releases allerdings etwas unter. Dabei hat sich trotz aller Probleme wieder eine Menge getan. Darum wollen wir in diesem Artikel einen Blick auf die Änderungen werfen, zunächst aber die typischen Eigenschaften zusammentragen und auch Kritikpunkte diskutieren.

Groovy hat seit jeher eine treue Fangemeinde (siehe Artikel in der letzten Ausgabe). Ein Großteil der Java-Entwickler meidet die Sprache aber, weil es sich ja „nur" um eine dynamisch typisierte Skriptsprache handelt. Für Irritation hat in Jahr 2009 auch die Aussage von James Strachan *[1]*, einem der Gründer, gesorgt: „I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy."

Man darf diese Aussage allerdings nicht falsch verstehen und sollte sich den gesamten Kontext des Blog-Beitrags anschauen. James Strachan hatte damals über einen Nachfolger von Java philosophiert. Groovy stand natürlich auch auf der Kandidatenliste. Durch die statische Typisierung und die mächtigen Typ-Inferenz-Mechanismen war Scala zu diesem Zeitpunkt für ihn aber die bessere Alternative zu Java. Das damals noch rein dynamisch typisierte Groovy konnte er sich nicht als kompletten Ersatz für Java vorstellen.

Auch heute lassen sich Groovy und Scala nur schwer vergleichen, auch wenn es Gemeinsamkeiten gibt. Letztlich verfolgen sie jedoch ganz unterschiedliche Ansätze und haben sich jeweils eine Nische in der Welt der alternativen JVM-Sprachen erkämpft. Als Ersatz oder Nachfolger für Java hat sich keine der beiden Sprachen durchsetzen können. Immerhin haben sie durch ihre mächtigere Syntax und den größeren Funktionsumfang Java vor sich hergetrieben. Das mündete letztendlich mit Version 8 im größten Update der Java-Geschichte. Und auch James Strachan sieht man in letzter Zeit wieder häufiger wohlwollend über sein früheres Baby (Groovy) twittern. Groovy lässt sich auf vier Hauptprinzipien reduzieren: Feature-Reichtum, Java-freundlich, dynamisches Groovy und robuste Plattform als Basis.

Feature-Reichtum: Auf der einen Seite bietet Groovy eine sehr ausdrucksstarke Syntax. Hat man sie einmal kennen und lieben gelernt, vermisst man sie schnell in reinen Java-Projekten. Zu nennen wären da stellvertretend die native Unterstützung für Collections inklusive der Wertebereiche (Ranges), die Verarbeitung von regulären Ausdrücken, GStrings, mehrzeilige Zeichenketten, der Property-Support, das Überladen von Operatoren und vieles mehr. Ebenfalls sehr praktisch sind die eingebauten Verarbeitungsmöglichkeiten für XML und JSON sowie die einfachen Datenbank-Zugriffsmöglichkeiten. Die Laufzeit-Meta-Programmierung (MOP), die Compiletime-Meta-Programmierung mit AST-Transformationen und Traits (Mehrfachvererbung) ermöglichen es, bestehenden Code sehr flexibel zu erweitern. Eine der grundlegenden Eigenschaften sind die anonymen Funktionen (Closures), auf denen viele andere Sprach-Konstrukte aufbauen. Sie geben der Sprache zudem einen funktionalen Anstrich.

Java-freundlich: Durch die große Ähnlichkeit mit der Java-Syntax lässt es sich mit Groovy sehr schnell loslegen, sei es in Tests, Skripten, als DSL (Spock, Gradle), zur Android-Entwicklung oder natürlich in Enterprise-Anwendungen (Grails). Als Superset der Java-Syntax ist gerade für Java-Entwickler die Lernkurve flach und die Migration von bestehenden Anwendungen einfach möglich. Groovy integriert sich nahtlos in Java und wird auch zu Bytecode kompiliert. Zudem können wechselseitig Klassen verwendet werden (Aufruf, Vererbung, Relationen). Dadurch kann Groovy als Basis das JDK wiederverwenden, erweitert es im GDK (Groovy Development Kit) aber auch um viele nützliche Funktionen.

Dynamisches Groovy: Bekannt wurde Groovy als dynamische Skriptsprache. Dieses Image hat in den letzten Jahren allerdings mehr geschadet, denn die dynamische Typisierung und die darauf aufbauende Laufzeit-Metaprogrammierung können so manchen Java-Entwickler überfordern, wenn er nur statisch getypten und damit vom Compiler geprüften Code gewohnt ist.

Als Alternative gibt es mittlerweile aber auch Traits und die Compiletime-Meta-Programmierung mit AST-Transformationen. Damit werden die Erweiterungen schon im Kompiliervorgang hinzugefügt, was sich positiv auf die Laufzeit-Performance und die Stabilität gerade von großen Projekten auswirkt. Zudem kann auch noch die statische Typprüfung aktiviert und, sofern gewünscht, der Sourcecode statisch kompiliert werden.

Aber auch die Runtime-Meta-Programmierung hat weiterhin ihre Berechtigung, ermöglicht sie doch gerade für Domain Specific Languages (wie in Grails, Gradle, Spock) oder in kleinen Skripten das Schreiben von kompaktem und Boiler-Plate-freiem Code, der dynamisch zur Laufzeit ganz flexibel um neue Funktionalitäten erweitert wird. Die Builder (für XML, JSON, Swing etc.) oder das Überladen von Operatoren sind typische Beispiele dafür.

Robuste Plattform als Basis: Da der Sourcecode zu Java-Bytecode übersetzt wird, profitiert Groovy als JVM-Sprache von allen Vorteilen der Java-Laufzeit-Umgebung. Das

bedeutet auch, dass die Java-Klassenbibliothek (JDK) und überhaupt alle Java-Bibliotheken einfach wiederverwendet werden können. Als Entwickler muss man zudem nicht umdenken, da das gleiche Speicher-, Threading-, Vererbungs- und Sicherheitsmodell zur Anwendung kommt. Alle lieb gewonnenen Werkzeuge (Build-Tools, IDEs, Profiler, Debugger ...) können meist „1:1" oder mit kleinen Einschränkungen weiterverwendet werden.

Kritikpunkte

Da wo Licht ist, ist natürlich auch Schatten. Denn gerade wenn man Groovy als General Purpose Language in großen Projekten einsetzt, stößt man schnell auf einige Probleme. Wie so oft gilt: „There is no free lunch" (Herb Sutter). Man kann nicht die Vorteile von Groovy nutzen, ohne auch einige Kompromisse in Kauf zu nehmen. Diese sind nicht unbedingt als Nachteil zu werten, man muss sich aber mit den möglichen Konsequenzen auseinandersetzen und vor allem verstehen, warum es sich so verhält. Bei Java geht es uns nicht anders, auch da leben wir mit gewissen Nachteilen. Am häufigsten wird bei Groovy die dynamische Typisierung mit all ihren Folgeerscheinungen kritisiert:

- Fehler erst zur Laufzeit
- Fehlende Tool-Unterstützung bei Code-Vervollständigung
- Unsicherheit beim Refactoring
- Schlechtere Performance

Kommt man aus der Java-Welt, ist man natürlich gewohnt, dass der Compiler bereits viele Fehler entdeckt. Beim Refactoring erhält man so direktes Feedback, wenn sich der Code nicht mehr übersetzen lässt. Zudem können durch die statische Kompilierung Optimierungen vorgenommen werden, die die Performance positiv beeinflussen. Aber auch wenn die dynamische Typisierung zunächst ungewohnt ist, so stellt sie doch eine der wichtigsten Funktionen von Groovy dar. Dierk König sagt dazu: „In OO, static types limit your choice. That makes dynamic languages attractive ..."

Gerade bei kleineren Aufgaben (wie Skripten) helfen optionale Typen und Meta-Programmierung dabei, die Intention des Codes zu präzisieren. Vom Ballast befreiter Quellcode ist zudem viel besser verständlich. Das ist nicht zu unterschätzen, wenn man bedenkt, dass man ihn zu 80 Prozent der Zeit liest. Außerdem fällt es einfach leichter, gewisse Design- und Clean-Code-Prinzipien einzuhalten beziehungsweise umzusetzen.

Die Groovy-Entwickler haben die kritischen Stimmen vernommen und so gibt es seit einiger Zeit Alternativen zur dynamischen Typisierung und zur Laufzeit-Meta-Programmierung. Mit den AST-Transformationen „@TypeChecked" und „@CompileStatic" kann man den Groovy-Compiler zu strengen Prüfungen und sogar zur statischen Kompilierung zwingen. An den annotierten Codestellen lassen sich dann allerdings keine dynamischen Fähigkeiten nutzen, die beiden Vorgehensweisen können jedoch innerhalb einer Anwendung kombiniert werden. Und für die Meta-Programmierung stehen mit Traits (Mehrfachvererbung) und AST-Transformationen Compiler-basierte Alternativen zur Verfügung.

Will man weiterhin mit dynamisch typisierten Anteilen umgehen, empfiehlt sich zur Qualitätssicherung natürlich auch eine gute Testabdeckung. In Java-Programmen haben wir diesen Anspruch doch ebenfalls. Groovy zwingt uns durch die fehlende Compiler-Unterstützung also nur stärker zu dieser Vorgehensweise. Gleichzeitig lassen sich in Groovy die Tests einfacher schreiben. Dadurch wird der Test-Code kürzer, ausdrucksstärker und besser verständlich, zudem sind Mocking-Funktionalitäten bereits eingebaut und Tools wie Spock oder Geb verwenden sowieso Groovy-DSLs. Durch eine hohe Testabdeckung werden auch die Fehler aufgedeckt, die in einer statisch getypten Sprache sonst durch den Compiler gefunden worden wären. Zusätzlich werden aber auch viele Szenarien getestet, bei denen der Compiler nicht helfen würde. Das klingt doch irgendwie nach einer Win-Win-Situation.

Durch die statische Kompilierung entfallen auch die Performance-Nachteile, da der erzeugte Bytecode ähnlich optimiert werden kann wie der von Java. Aber auch bei dynamisch typisiertem Groovy-Code wurden in den letzten Versionen immer wieder kleinere Optimierungen vorgenommen, sodass sich die Situation in den vergangenen Jahren stetig verbessert hat. Der größte Geschwindigkeitszuwachs wird die noch ausstehende Überarbeitung des Meta Object Protocol auf Basis von Invoke Dynamic bringen. Leider ist aktuell nicht klar, ob und wann dieser Umbau erfolgen wird. Allerdings geht es beim Großteil der Anwendungsfälle selten um das Herauskitzeln der letzten Millisekunden; die Flaschenhälse liegen oftmals an ganz anderen Stellen (wie Datenbank-Abfragen) und für die Implementierung von Real-Time-Systemen wird man Groovy eher nicht verwenden.

Groovy hebt sich von Java nicht nur durch die dynamischen Fähigkeiten, sondern auch durch viele kleine nützliche Funktionen ab. Man könnte jede Menge nennen, stellvertretend sind hier die inoffiziellen Top 5 kurz an Beispielen illustriert *(siehe Listing 1)*:

- Mehrzeilige Strings und Zeichenketten mit Platzhaltern
- Elvis-Operator
- Vereinfachte Objekt-Navigation und sicheres De-Referenzieren
- Eingebauter Support für das Lesen und Schreiben von Datenformaten (XML, JSON etc.)
- Power-Asserts

Das war jedoch nur die Spitze des Eisbergs, genauso erwähnenswert wären zum Beispiel:

- Multiple Assignments
- Null Object Pattern
- Spaceship-Operator
- Diverse GDK-Methoden und -Operatoren
- Diverse AST-Transformationen
- Dynamische Method-Interception
- Default-Parameter in Methoden

Eine Aufstellung dieser Hidden Features steht unter *[2]*. Aufgrund der Mächtigkeit der Sprache darf ein Punkt jedoch auch nicht vergessen werden: „With great power comes great responsibility". Man kann Dinge machen, von denen Java-Entwickler nur träumen können. Aber gerade deshalb sollte man Vorsicht walten lassen, insbesondere in gemischten Teams mit reinen Java-Entwicklern. Was für einen Groovy-Fan elegant aussieht, ist für einen anderen Kollegen ein Buch mit sieben Siegeln. Deshalb sollte man zum Beispiel einheitliche Code-Konventionen festgelegt. Über den Einsatz statischer Code-Analyse-Tools (CodeNarc) und die statische Typprüfung empfiehlt es sich ebenfalls nachzudenken.

Neuerungen der letzten Releases

Dass Groovy kein One-Hit-Wonder ist, sieht man an den regelmäßig veröffentlichten Versionen (mindestens einmal pro Jahr). Da die Sprache mittlerweile mehr als zwölf Jahre alt ist, darf man allerdings keine bahnbrechenden Neuerungen mehr erwarten. Es gibt jedoch stetig kleine Fortschritte, um die Verwendung einfacher und effizienter zu machen. Das letzte große Release war

```
String s = """Dieser GString
reicht über mehrere Zeilen und
kann Platzhalter verwenden: ${person.foobar}"""

// von if/else zum Elvis-Operator
def username = getUsernameFromAnywhere()
if (username) { username } else { 'Unknown' }
username ? username : 'Unknown'
username ?: 'Unknown'

// Null-Safe Dereferenz Operator
bank?.kunden?.konten?.guthaben?.sum()

// XML parsen
def text = '''
  <list>
     <technology>
        <name>Groovy</name>
     </technology>
     <technology>
        <name>Grails</name>
     </technology>
  </list>'''

def list = new XmlSlurper().parseText(text)
assert list.technology.size() == 2

// Power Asserts
groovy> a = 10
groovy> b = 9
groovy> assert 89 == a * b
Exception thrown

Assertion failed:

assert 89 == a * b
            |  | | |
            |  10| 9
            |    90
            false
```

Listing 1

Version 2.4 Anfang 2015. Danach gab es nur noch kleinere Bugfix-Releases, bei denen sehr viel Aufwand in die Erfüllung der Bedingungen der Apache Software Foundation gesteckt werden musste. Die markantesten Neuerungen der letzten drei Minor-Releases waren:

- Neue AST-Transformationen
- Traits
- Android-Support

Der Groovy-Compiler bietet bereits seit der Version 1.6 die Möglichkeit, in den Kompilierungslauf einzugreifen, um beispielsweise zusätzliche Anweisungen einzufügen oder zu löschen („AST" steht für „Abstract Syntax Tree"). Damit können klassische Querschnittsprobleme wie Autorisierung, Tracing oder die Transaktionssteuerung gelöst werden, ohne den Sourcecode mit unnötigem Ballast aufzublähen. Vergleichbar ist dieser Mechanismus mit der Vorgehensweise der Java-Bibliothek Lombok, bei Groovy ist er allerdings direkt in die Sprache integriert.

Regelmäßig kommen neue Implementierungen von AST-Transformationen hinzu. Eigene zu schreiben, ist auch möglich. Man unterscheidet zwischen globalen und lokalen AST-Transformationen, bei letzteren werden die anzupassenden Codestellen durch Annotationen markiert. Der Compiler kann dann an diesen Stellen zusätzliche Informationen in den Bytecode generieren. Der Sourcecode bleibt so schlank, also gut les- und wartbar.

```
@Builder
class Person {
  String firstName, lastName
  int age
}

def person = Person.builder()
        .firstName("Dieter")
        .lastName("Develop")
        .age(21)
     .build()

assert person.firstName == "Dieter"
assert person.lastName == "Develop"
```

Listing 2

Da die generierten Informationen im Bytecode landen, können aber auch andere (Java-) Klassen diese Funktionalitäten aufrufen. Es folgt mit dem Builder Design Pattern ein Beispiel für eine lokale AST-Transformation. Die Implementierung der Builder-Klasse existiert dabei nur im Bytecode *(siehe Listing 2)*.

Neben dem Builder gibt es eine große Anzahl von fertigen AST-Transformationen *[3]* und es kommen ständig neue dazu. Die Transformationen können in verschiedene Kategorien einsortiert werden: Code-Generation („@ToString", „@EqualsAndHashCode" etc.), Klassen-Design („@Immutable", „@Singleton" etc.), Logging-Erweiterungen („@Log", „@Log4j" etc.), deklarative Nebenläufigkeit („@Synchronized" etc.), Compiler-Hinweise („@TypeChecked", „@CompileStatic", „@PackageScope" etc.), Auflösen von Abhängigkeiten (Dependency Management mit „@Grab") und viele mehr.

In der Dokumentation finden sich weitere Informationen. Auch Grails macht mittlerweile starken Gebrauch davon und kann seine Magie so sehr stabil und auch performant umsetzen. Das Schreiben eigener Transformationen ist übrigens nicht ganz trivial, es gibt jedoch einige Tools (wie AST-Browser) und mittlerweile auch Tutorials, die bei der Umsetzung unterstützen.

Traits wurden Anfang 2014 in Version 2.3 in Groovy eingeführt. Seit Version 8 gibt es mit Default-Methoden in Java eine vergleichbare, aber nicht ganz so mächtige Alternative. Letztlich handelt es sich um einen Mehrfachvererbungs-Mechanismus; in Groovy gab es mit der AST-Transformation „@Mixin" bereits seit Version 1.6 eine Vorgänger-Umsetzung. Diese hatte allerdings einige Probleme und so wurde das Konzept mit Traits direkt in die Sprache integriert. Das Ziel ist es, einzelne Funktionalitäten in eigene Interfaces herauszuziehen („Sing-

```
trait Fahrbar {
  int geschwindigkeit
  void fahren() {
     "println Fahren mit ${geschwindigkeit} km/h!"}
}
```

Listing 3

```
class Bobbycar implements Fahrbar {
}
new Bobbycar(geschwindigkeit:100).fahren()
```

Listing 4

le Responsibility"-Prinzip) und später über Vererbung wieder zu kombinieren.

Bis zur Einführung der Default-Methoden war in Java keine Mehrfachvererbung möglich – genaugenommen auch nicht gewollt. In Java 8 wurde sie aus der Not heraus hinzugefügt. Durch das Stream-API mussten die Interfaces des Collection-Frameworks erweitert werden. Ohne Default-Methoden wären alle bisherigen Implementierungen kaputtgegangen, die Abwärtskompatibilität wäre gebrochen gewesen. Traits können im Gegensatz zu Java-Interfaces nicht nur Methoden, sondern auch Zustände enthalten *(siehe Listing 3)*. Subklassen können dann von beliebig vielen Traits ableiten und erben sowohl die Variablen als auch die Methoden *(siehe Listing 4)*.

Wie bei Interfaces üblich, dürfen Traits abstrakte Methoden definieren, sodass Subklassen diese überschreiben müssen. Das kann beispielsweise für die Implementierung des Template Method Pattern interessant sein. Dass in Java lange Zeit keine Mehrfachvererbung angeboten wurde, hing in erster Linie mit der Angst vor Konflikten in der Vererbungshierarchie zusammen. Das allgemein bekannte Diamond-Problem wird in Groovy aber relativ einfach umgangen *(siehe Listing 5)*.

Es gewinnt nämlich automatisch das zuletzt in der „implements"-Liste stehende Trait, wenn die aufzurufende Methode ermittelt wird. Möchte man die Auswahl steuern, kann man entweder die Reihenfolge anpassen oder explizit den Super-Aufruf implementieren *(siehe Listing 6)*.

Traits können übrigens auch ganz dynamisch zur Laufzeit von Instanzen einer Klasse implementiert werden. Dann ist man zwar wieder bei der langsameren Runtime-Meta-Programmierung, kann aber sehr flexibel bestehende Objekte um neue Funktionalitäten ergänzen. Übrigens kann ein Trait mit einer abstrakten Methode (SAM-Type) auch über eine Closure instanziiert werden, ähnlich wie das bei Java 8 mit Lambda-Ausdrücken und Functional Interfaces der Fall ist. Weitere Beispiele finden sich unter *[4]*.

Seit Anfang 2015 (Version 2.4) lassen sich Android-Apps mit Groovy entwickeln. Android-Entwickler profitieren dabei auch wiederum von Groovys prägnanter Syntax und den Meta-Programmierungsmöglichkeiten. In Java entwickelte Android-Anwendungen enthalten typischerweise viel Boiler Plate Code. Zudem stehen die modernen Errungenschaften aus Java 8 noch nicht zur Verfügung. Groovy bringt mit der SwissKnife-Bibliothek zudem einige AST-Transformationen mit, sodass häufig wiederkehrende Funktionen (Behandeln von UI-Events, Verarbeitung in Background-Threads) sehr schlank implementiert werden können.

Wir haben gesehen, dass es in Groovy in den vergangenen Jahren keine technischen Revolutionen gab. Das liegt allerdings hauptsächlich an der Ausgereiftheit der Sprache, die nur wenige Wünsche offenlässt. Trotzdem wurden evolutionsartig kleine Verbesserungen und Neuerungen eingebaut. Wie sich das unter der Obhut von Apache ohne kommerzielle Firma im Rücken entwickeln wird, muss sich noch zeigen. Ganz anders ist es bei Grails verlaufen. Da gab es mit Version 3.0 im Frühjahr 2015 geradezu ein Feuerwerk an Änderungen, damals noch unter der Ägide von Pivotal. Aber durch die Übernahme durch die Firma OCI ist weiterhin Zug in dem Open-Source-Projekt, wie auch das Release 3.1 und die in der Roadmap *[5]* geplanten Versionen beweisen.

Im Jahre 2005 erblickte Grails das Licht der Welt und war letztlich die Antwort der Groovy-Welt auf das Web-Application-Framework Ruby on Rails. In Anlehnung daran wurde ursprünglich der Name gewählt („Groovy auf Schienen"), durch die Abkürzung ist man nun aber eher auf der Suche nach dem heiligen Gral.

Grails ist geprägt von einigen Prinzipien für sauberes, objektorientiertes Software-Design wie „DRY" (Don't Repeat Yourself), „SRP" (Single Responsibility) und „CoC" (Convention over Configuration). Das heißt zum Beispiel, dass statt einer komplexen Konfiguration einfach Konventionen für die Namensgebung von Objekten einzuhalten sind, aus denen sich das Zusammenspiel zur Laufzeit automatisch ergibt. Diese Eigenschaften ermöglichen eine rasche Umsetzung von Anforderungen, wodurch Grails seit jeher ein beliebtes Prototyping-Framework ist, das zudem auch optisch sehr ansprechende Ergebnisse liefert. Vor mehr als zehn Jahren hatten die klassischen Java-Webanwendungen einige Tücken:

- Aufwändiges Editieren von Konfigurationsdateien
- Notwendige Anpassungen des Deployment-Deskriptors („web.xml")
- Aufwändige Konfiguration der Abhängigkeiten
- Umständliche Build-Skripte
- Primitive Template- und Layout-Mechanismen
- Neustarten der Anwendung beziehungsweise des Webcontainers nach Änderungen

```
trait A {
  String exec() { 'A' }
}

trait B extends A {
  String exec() { 'B' }
}

trait C extends A {
  String exec() { 'C' }
}

class D implements B, C {}

def d = new D()
assert d.exec() == 'C'
```

Listing 5

```
class D implements B, C {
  String exec() { B.super.exec() }
}
def d = new D()
assert d.exec() == 'B'
```

Listing 6

Mit Grails wurde den Entwicklern die Arbeit deutlich erleichtert. Aus dem ursprünglichen Einsatzzweck für CRUD-Webanwendungen hat es sich mittlerweile zu einem gestandenen Fullstack-Web-Application-Framework gemausert. Zu den Kernprinzipien zählen die gesteigerte Produktivität, die einfachere Entwicklung, die Erweiterbarkeit und die ausgereiften Bibliotheken als stabile Basis.

Gesteigerte Produktivität

Grails eignet sich wunderbar für das Bauen von Prototypen. Durch sinnvoll gewählte Grundeinstellungen und einfache Konventionen lassen sich Redundanzen vermeiden und in kurzer Zeit beachtliche Ergebnisse erzielen. Hinzu kommen die mittlerweile auf Gradle aufbauende Entwicklungskonsole, die einheitliche Projektstruktur, die durch Metaprogrammierung hineingemixten Enterprise-Funktionalitäten und die Verwendung von Groovy.

Die Einfachheit ergibt sich durch die prägnantere Syntax von Groovy, die einheitliche Persistenz-Schnittstelle (GORM, ursprünglich für Objekt-relationales Mapping erfunden, mittlerweile aber auch für NoSQL-Datenbanken geeignet), die wohldefinierte Projektstruktur mit hohem Wiedererkennungseffekt, Verzicht auf unnötige Konfigurationen, die automatische Injektion von fertig mitgelieferten Enterprise-Services und die Validierung von Daten anhand flexibler und leicht zu erweiternder Regeln. Grails folgt dem KISS-Prinzip („keep it simple and stupid") mit dem Ziel, unnötigen Boiler-Plate-Code zu vermeiden. Zudem kann per Scaffolding statisch oder dynamisch ein initiales Code-Gerüst generiert werden, das bereits die CRUD-Funktionalitäten vom Frontend bis zur Datenspeicherung abdeckt und sogar Restful-Schnittstellen anbietet. Man kommt somit in wenigen Minuten zu einer lauffähigen Applikation. Nur bei der Geschäftslogik kann und muss letztlich noch Hand angelegt werden.

Die Erweiterbarkeit zeigt sich durch den modularen Aufbau von Grails mit der Austauschbarkeit von einzelnen Komponenten (Hibernate/RDBMS <=> NoSQL, Tomcat <=> Jetty etc.). Zudem kann der Kern von Grails über Plug-ins beliebig um weitere Funktionalitäten ergänzt werden. Auch die eigene Anwendung lässt sich über Plug-ins sauber strukturieren und einzelne Teile sind entsprechend leichter wiederverwendbar. Aufbauend auf den gleichen Ideen wie dem Java EE Standard (Integration von Spring und Hibernate) bringt es zudem seine Ablaufumgebung in Form eines Webcontainers direkt mit und war durch die Entwicklungskonsole seit jeher quasi eine Art Vorgänger der heute so stark beworbenen Self-Contained-Systems. Da die aktuelle Version auf Spring Boot aufsetzt, können mittlerweile auch direkt ausführbare Fat-JARs erzeugt werden.

Stabile Basis

Grails ist in erster Linie eine Menge Gluecode um bestehende mächtige Bibliotheken wie Spring, Hibernate und Sitemesh. Durch die Entwicklung in Groovy kann man den syntaktischen Zucker von Grails voll ausnutzen. Als Build-Artefakt wird ein klassisches Java WAR erzeugt, das in Webcontainern eingesetzt beziehungsweise mittlerweile als Fat-JAR einfach ausgeführt werden kann. Leider ist nichts perfekt und so werden auch an Grails immer wieder Kritikpunkte laut:

- Problematisch in großen Projekten
- Gebrochene Abwärtskompatibilität und großer Aufwand bei Upgrades
- Aktualität der Plug-ins
- Unleserliche Stack-Traces

Auch wenn einige Erfolgsstories für große Projekte existieren, eignet sich Grails eher für überschaubare Projekte mit kleineren und möglichst homogenen Teams (etwa für Microservices). Gerade ein Mix von erfahrenen Java- und enthusiastischen Groovy-Entwicklern kann schnell zu Konflikten und auf Dauer zu Unverständnis führen. Aus eigener Erfahrung empfiehlt es sich auf jeden Fall, projektübergreifende Code-Konventionen (Typ-Angaben hinschreiben, Verwendung von „return" etc.) festzulegen, damit auch die Java-Entwickler den Groovy-Code lesen und ändern können. Zudem ist eine gute Testabdeckung wichtig, um die Auswirkungen der dynamischen Funktionalitäten abfangen zu können und bei Refactorings wenigstens durch die Testläufe Feedback zu erhalten.

Letztlich gilt wie so oft das Pareto-Prinzip. 80 Prozent der Anwendung (zumeist CRUD) lassen sich in Grails sehr einfach umsetzen. Die letzten 20 Prozent brauchen dann jedoch auf einmal einen Großteil der Zeit und erscheinen im Vergleich umständlicher, aber natürlich nicht unlösbar. In Java fällt das nur nicht so auf, weil schon die ersten 80 Prozent deutlich mehr Aufwand bedeuten.

Die mangelnde Abwärtskompatibilität schlägt sich meist darin nieder, dass man einigen Aufwand investieren muss, um bestehende Anwendungen auf die neueste Grails-Version zu aktualisieren. Geschuldet ist dies dem Innovationsdrang der Grails-Entwickler, die neuesten Techniken und Ideen rasch zu integrieren. Das wird umso aufwändiger, je länger man wartet. Darum sollte man immer frühzeitig updaten. Die Grails-Dokumentation unterstützt durch einen Migrationsguide. Und hat man doch mal einige Versionsupdates ausgelassen, empfiehlt es sich, alle Änderungen Schritt für Schritt und von Version zu Version abzuarbeiten.

Der modulare Aufbau von Grails hat zu einem riesigen Plug-in-Ökosystem geführt *[6, 7]*. Leider schwankt die Qualität sehr stark. Guten Gewissens kann man die Plug-ins der Core-Entwickler einsetzen, sie werden bei neuen Grails-Versionen zeitnah angepasst und regelmäßig gewartet. Erkennen kann man sie über positive Bewertungen des Voting-Systems und durch eine große Anzahl von aktiven Nutzern. Bei anderen, oft von Einzelkämpfern erstellten Modulen gilt es zunächst, einen Blick auf die Aktualität des Sourcecodes (letzte Commits), die Antworthäufigkeit bei Fragen und die Bearbeitung der Tickets zu werfen. Leider existieren nämlich einige Karteileichen, die durch die hochfrequente Update-Politik mit den aktuellen Versionen gar nicht mehr funktionieren. Da bleiben dann bei Interesse nur der Fork und die eigene interne Pflege des Plug-ins oder natürlich eine Weiterentwicklung des Originals, sofern man sich das zutraut.

Aufgrund des großen Funktionsumfangs fällt es schwer, die wichtigsten Grails-Features zu benennen. Es gibt einfach zu viele nützliche Eigenschaften. Zudem setzt jeder Entwickler auch andere Prioritäten. Folgende Punkte sollen aber stellvertretend für viele andere stehen:

- Automatische Dependency Injection (etwa durch Namenskonventionen)
- Validierung der Domänen-Klassen (ähnlich wie Java-EE-Validation-API)
- Sehr einfache, aber trotzdem mächtige Erweiterung durch Tag-Libs
- Automatisch generierte CRUD-Methoden
- Viele Persistenz-Abfrage-Varianten wie die gut lesbaren „Where"-Queries

Bei jedem dieser Punkte könnte man wieder ins Detail gehen. An dieser Stelle sei aber auf die Dokumentation verwiesen. In-

teressanter sind die neuesten Funktionen des letzten Release. Neue Minor-Versionen kommen ähnlich wie bei Groovy mindestens einmal pro Jahr heraus. Das letzte Major-Update (3.0) wurde im Frühjahr 2015 veröffentlicht. Die Liste der Neuerungen ist lang und beeindruckend. Die Maßnahmen versprechen eine verbesserte Stabilität und leichtere Integration in bestehende System-Landschaften:

- Spring Boot als Basis
- Neues Interceptor-API
- Anwendungsprofile
- Gradle als Build-System
- API-Redesign

Über Spring Boot braucht man mittlerweile nicht mehr viel zu schreiben. Viele seiner Eigenschaften hatte Grails in ähnlicher Form schon sehr lange im Portfolio. Um sich aber den unnötigen Pflegeaufwand zu sparen, baut man nun auf dem vielfach bewährten Framework auf. Von Spring Boot profitiert Grails besonders durch die Einbettung des ausführenden Containers, der in einem lauffähigen JAR inkludiert wird. Damit kann man Grails-Anwendungen tatsächlich überall laufen lassen, es muss nur Java installiert sein. In der Entwicklung ist man zudem nicht mehr auf eine speziell angepasste IDE angewiesen, da eine Grails-Anwendung nun wie jede beliebige Java-Anwendung gestartet, auf Fehler überprüft und mit Profilen eingerichtet werden kann.

Das neue Interceptor-API ergänzt beziehungsweise ersetzt die Grails-Filter bei der Entwicklung von Querschnittsbelangen (Security, Tracing, Logging etc.). Gewisse Basis-Funktionalitäten erben Interceptoren von einem Trait. Im Gegensatz zu den Filtern, die ähnlich Servlet-Filtern über Muster auf bestimmte URLs reagieren, werden Interceptoren in der Regel über Namenskonventionen jeweils einem Controller zugeordnet. Gemeinsam ist beiden Ansätzen, dass sie verschiedene Callbacks („before", „after" und „afterView") anbieten, über die Querschnittsfunktionalität implementiert wird. Interceptoren sind dazu noch kompatibel mit der statischen Kompilierung („@CompileStatic"), ein nicht zu verachtender Performance-Vorteil. Schließlich werden sie potenziell bei jedem Request aufgerufen. Weitere Informationen finden sich unter *[8]*.

Eine andere Neuerung ist der Support für Anwendungsprofile. Gemeint ist hier die Struktur, die beim Anlegen eines neuen Grails-Projekts gleich auf das gewünschte Szenario (Web-, REST-, RIA-Anwendung etc.) zugeschnitten werden kann. Es ist ein bisschen vergleichbar mit den Java-EE-Profilen („Web" und „Full"), geht aber noch einige Schritte weiter. So kapselt ein Profil die aufrufbaren Kommandos, Plug-ins, Skeletons/Templates und Ressourcen. Der Default ist übrigens weiterhin die klassische Web-Anwendung. Man kann auch eigene, firmenspezifische Profile anlegen, die dann in einem Repository (im User-Verzeichnis) gespeichert werden.

Grails hatte seit jeher ein auf Groovy, Ant und Ivy basierendes, selbst entwickeltes Build-Management, auf dem auch die Kommandos für die Entwicklungskonsole aufgebaut haben. Aber gerade die Integration in eine bestehende Systemlandschaft mit klassischen Java-Build-Tools wurde dadurch unnötig verkompliziert. Zudem war das Bauen mit Gant immer etwas problematisch. Seit Grails 3 wird als Build-System Gradle verwendet. Somit ist man auch hier nicht mehr abhängig von speziell integrierten Funktionen der IDEs. Das erleichtert den Abschied von der Eclipse-basierten „Groovy und Grails Toolsuite", die seit dem Rückzug durch Pivotal nicht mehr weiterentwickelt wird. Es reicht theoretisch wieder ein einfacher Text-Editor mit der Kommandozeile, so wie Grails bereits vor mehr als zehn Jahren gestartet ist.

In Grails passieren viele Dinge automatisch, manchmal magisch. Das wurde in den älteren Versionen vor allem durch Laufzeit-Meta-Programmierung erreicht, war jedoch immer wieder die Quelle für Probleme und Instabilitäten im Entwicklungsprozess. Mittlerweile setzt man bei der Erweiterung von Domain-Klassen, Controllern, Services, Tag-Libs und Tests verstärkt auf Compiletime-Meta-Programmierungsmechanismen mit AST-Transformationen und auf Traits. Im Rahmen dieser Umbauarbeiten wurden gleich das API umstrukturiert und eine saubere Trennung in öffentliche („grails.*") und private/interne Pakete („org.grails.*") vorgenommen.

Die stetig steigenden Downloadzahlen zeigen das wachsende Interesse. Wer nach dem hier Gelesenen (erneut) Lust auf Groovy und Grails bekommen hat, sollte sich selbst ein Bild machen, die neuesten Versionen installieren und einfach mal loslegen. Die Online-Dokumentation hilft beim Starten und zudem gibt es mittlerweile jede Menge Literatur und Tutorials zum Thema.

Referenzen

[1] Blog James Strachan: *http://macstrac.blogspot.de/2009/04/scala-as-long-term-replacement-for.html*

[2] Groovy Hidden Features: *http://stackoverflow.com/questions/303512/hidden-features-of-groovy*

[3] Available AST transformations: *http://groovy-lang.org/metaprogramming.html#_available_ast_transformations*

[4] Traits: *http://blog.oio.de/2014/07/04/unterstutzung-von-traits-ab-groovy-2-3/*

[5] Grails Roadmap: *https://github.com/grails/grails-core/wiki/Roadmap*

[6] Grails-Plug-ins bis 2.x: *https://grails.org/plugins*

[7] Grails-Plug-ins ab 3.0: *https://bintray.com/grails/plugins*

[8] Grails Interceptor: *http://www.ociweb.com/resources/publications/sett/september-2015-grails-3-interceptors*

Falk Sippach
falk.sippach@oio.de

Falk Sippach hat mehr als fünfzehn Jahre Erfahrung mit Java und ist bei der Mannheimer Firma OIO Orientation in Objects GmbH als Trainer, Software-Entwickler und Projektleiter tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit Anderen die örtliche Java User Group. Falk Sippach twittert unter @sippsack.