Java aktuell

iJUG Verbund
www.ijug.eu

Jakarta EE
So geht es jetzt weiter

Groovy
Nützliche, weniger bekannte Features

web3j
Ethereum-Blockchain für Java-Applikationen

Jakarta und der große Knall

Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
Ausgabe 05/2019
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 05

eclipsecon Europe 2019

OSGi™ Community Event 2019

Open Innovation and Collaboration

21. - 24. Oktober 2019
Ludwigsburg

Keynotes

Jan Leuridan
Sr. VP, Simulation and Test Solutions PLM Software
Siemens

Matt Rutkowski
CTO Serverless Technologies
IBM

Auszug Sprecherliste

Jens Reimann
Principal Software Engineer
Red Hat

Kamesh Sampath
Director of Developer Experience
Red Hat

Martin Lippert
Principal Software Engineer
Pivotal Software

Die Open Source Software Konferenz für Eclipse Technologien, offene Innovation und Industriekollaboration

Interesse an Cloud Native Java wie Jakarta EE, MicroProfile und Cloud IDEs, dem Eclipse IoT Stack und den erprobten Eclipse Technolgien wie der Eclipse IDE, RCP und Modeling? Auf der EclipseCon vernetzt Ihr Euch mit den Experten!

Frühbucherrabatt bis 1. Oktober - Ab 775 € (+ MwSt.) Anmeldung unter: www.eclipsecon.org

Jakarta und der große Knall

Die Java-Welt steht niemals still und wir halten Sie wie gewohnt auf dem Laufenden zu aktuellen Neuigkeiten rund um Ihre Lieblingstechnologie. Nachdem die Verhandlungen zwischen Oracle und der Eclipse Foundation zuletzt damit endeten, dass letztere den Namensraum „Java" nicht benutzen darf, hat sich einiges getan. Markus Karg berichtet in der Eclipse Corner, wie es mit dem Projekt „Jakarta EE" weitergeht und welche Maßnahmen die Eclipse Foundation jetzt umsetzt. In seinem Tagebuch hat auch Andreas Badelt, stellvertretender Leiter der DOAG Java Community, alles Wissenswerte der letzten Wochen für Sie zusammengefasst. Dmitrij Drandarov, msg David, berichtet, wie er mit seinem Team ein Legacy-Projekt zu neuem Leben erweckt hat, und teilt dabei seine Lessons Learned mit Ihnen. Weiterhin erwarten Sie spannende Artikel zu den Themen web3j, agile Führung, DRY-Prinzip und vieles mehr.

Wir wünschen Ihnen viel Spaß beim Lesen!

Ihre

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Groovy's nützliche Werkzeugkiste, die Sie vielleicht noch nicht kennen

Praxisbericht zur Reanimierung einer Software mit Gradle

Aus Fehlern lernen und davon profitieren

Die Vorteile der Ethereum-Blockchain für Java-Applikationen

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

25. März 2019

20 Jahre Apache Software Foundation

Die Apache Software Foundation feiert ihr 20-jähriges Bestehen. Wie sähen Open Source und die Software-Welt generell heute ohne die ASF aus? Kaum vorstellbar. Wir sagen: herzlichen Glückwunsch!

9. April 2019

Jakarta EE und die Namensrechte

Die Namensfindung für Jakarta EE geht weiter. Teilweise läuft dieser Prozess offen über die Mailing-Listen und Blogs, insbesondere was die Projektnamen angeht. Der aktuelle Vorschlag von Wayne Beaton, Director of Open Source Projects der Eclipse Foundation, lautet: Der Name Java wird konsequent durch Jakarta ersetzt, Camel Case wird durch Leerzeichen ersetzt und vorne steht immer Jakarta (auch wenn vorher kein Java im Namen war). Gewöhnen wir uns also schon mal an Jakarta Mail, Jakarta Enterprise Beans und die Jakarta Expression Language. Das ist auf jeden Fall schon mal konsistent. Parallel läuft – allerdings im Hinterzimmer – immer noch das juristische Tauziehen um Markenrechte und die Namensrechte an den „javax"-Packages.

9. April 2019

StackOverflow-Umfrage

Im Developer Survey 2019 von StackOverflow *[1]* ist Python an Java vorbeigezogen. Ein wesentlicher Faktor dürfte sein, dass Python den stark wachsenden Bereich Machine Learning dominiert. Es hatten fast 90.000 Personen aus 179 Ländern geantwortet, was für eine gewisse Repräsentativität spricht – die meisten Umfragen dieser Art erhalten deutlich weniger Antworten. Einige der Ergebnisse wurden auch nach Geschlechtern aufgeschlüsselt. Beispielsweise schreiben Männer mit durchschnittlich 15 1/2 Jahren etwas früher als Frauen, mit durchschnittlich 17 Jahren, ihre „erste Code-Zeile". Befragte nonbinären beziehungsweise diversen Geschlechts sogar mit 14 Jahren noch früher. Da mehr als die Hälfte der Befragten unter 30 Jahre alt waren, scheinen Geschlechterklischees also auch weiterhin eine große Rolle zu spielen. Übrigens, bei den IDEs liegen Visual Studio und Visual Studio Code gemeinsam vorn, gefolgt von, wer hat's erraten? Genau: Notepad++, und deutlich dahinter IntelliJ sowie vim – what?

10. April 2019

Google/Oracle – ein Stoßgebet

Amerikanische Rechtsprechung ist ja für Kontinentaleuropäer manchmal schwer zu verstehen – aber zumindest die Berichterstattung ist unterhaltsam. Nachdem Google 2018 den Akt des jahrelangen Rechtsstreits mit Oracle um Urheberrechtsverletzungen durch das Kopieren von Java APIs für Android verloren hat, hat der Konzern nun den Supreme Court der USA um eine Überprüfung dieses Urteils gebeten. Unter Gerichtsjournalisten wird dieser Antrag „Hail Mary Request" („Ave Maria") genannt – was sich auf die Erfolgsaussichten bezieht, vergleichbar mit dem „Hail Mary Shot" in der Schluss-Sekunde eines Basketballspiels (manchmal geht der Wurf aus 15 m allerdings auch in den Korb). Ob Googles Anwälte tatsächlich für den Erfolg beten, ist nicht überliefert.

17. April 2019

Red Hat übernimmt OpenJDK-Projektleitung

Red Hat hat jetzt offiziell die Projektleitung für die OpenJDK Releases 8 und 11 von Oracle übernommen. Der Konzern gibt sich optimistisch, was die Zukunft von Java angeht, und will mehr Verantwortung übernehmen. Das ADTMag zitiert Red Hats Senior Director of Product Management Rich Sharples, dass Java „noch weitere 20 bis 30 Jahre Leben vor sich habe". In der Pressemitteilung heißt es: „Java is in a renaissance moment. It continues to evolve and be a key component of new, emerging architectures." Darin verkündet das Unternehmen auch, dass in den nächsten Wochen das angekündigte Windows Release des OpenJDK mit kommerziellem Support herauskommen werde – zusammen mit IcedTea-Web, einer Open-Source-Implementierung von Java Web Start aus dem AdoptOpenJDK-Projekt.

25. April 2019

NetBeans verlässt den Inkubator

NetBeans darf nach dem Erscheinen von Version 11.0 den „Brutkasten" der Apache Software Foundation verlassen und ist zum Top-Level-Projekt befördert worden. Oracle-Mitarbeiter und Vice President of Apache NetBeans Geertjan Wielenga hat die Community aufgerufen, sich nun stärker zu engagieren: „Die Kehrseite von Freiheit ist Verantwortung." Alle Nutzer von NetBeans sollten sich fragen, wie sie das Projekt mit vorantreiben können – von BugFixing über Dokumentationsreviews hin zum Teilen von Tipps auf Twitter. Laut StackOverflow-Umfrage sind das „nur" noch 5,9 % der Entwickler, verglichen mit circa 15 % Eclipse- und 25 % IntelliJ-Nutzern, in absoluten Zahlen ist das aber immer noch eine Menge.

3. Mai 2019

Oracle und Eclipse: Verhandlungen gescheitert

Mike Milinkovich lässt in seinem Blog *[2]* unter „Update on Jakarta EE Rights to Java Trademarks" eine Bombe platzen. „Die Jakarta Community ist fleißig gewesen", berichtet er und zählt einige abgeschlossene sowie noch laufende Initiativen auf. Dann geht es jedoch schnell zum Kern: Die Eclipse Foundation und Oracle haben sich nicht auf Bedingungen für die Eclipse Foundation Community einigen können, um

Änderungen an „javax"-Package-Namespaces vorzunehmen oder um die derzeit in Java-EE-Spezifikationen verwendeten Java-Marken zu verwenden. Sprich: „javax"-Packages können von Jakarta EE nicht modifiziert und Marken wie beispielsweise die existierenden Spezifikationsnamen können dort nicht weiterverwendet werden. Damit bekommt nun auch der Vorschlag von „Insider" Wayne Beaton von Anfang April mehr Kontext. Über die Details der Verhandlungen beziehungsweise darüber, wie es zum Scheitern kam, haben laut Milinkovich beide Seiten Stillschweigen vereinbart – „aufgrund der Komplexität und Vertraulichkeit der Verhandlungen". Den Meeting Minutes der Eclipse Foundation lassen sich allerdings einige Details entnehmen, beispielsweise, dass die Eclipse Foundation befürchtet hat, die von Oracle geforderten Bedingungen könnten auch Nicht-Jakarta-Projekte wie die Eclipse IDE selbst gefährden (weil unter anderem der eigene Compiler ECJ nicht mehr mit ausgeliefert werden kann) oder sogar für den gemeinnützigen Status der Foundation bedrohlich werden; und dass das nicht absehbare Ende der Verhandlungen nach über 18 Monaten für das Vertrauen in die Foundation gefährlicher sei als ein Abbruch der Verhandlungen.

7. Mai 2019

Wie geht's weiter mit Jakarta?

Heulen und Zähneknirschen war gestern. Die Jakarta-Community hat sich kurz gesammelt und versucht, schnell eine Lösung für das Namensrechte-Dilemma zu finden. Blogs und E-Mail-Verteiler laufen heiß. „Big Bang" oder „Incremental Change" heißen die beiden Hauptvorschläge: Sollen alle „javax"-Packages auf einen Schlag in „jakarta" umbenannt werden (Jakarta EE 9), um dann in Release 10 auf sauberer Basis mit neuen Features zu starten? Oder soll die Umbenennung nur bei Bedarf erfolgen, das heißt pro Spezifikation und nur, wenn Packages angepasst werden müssen, sodass gegebenenfalls auch langfristig immer noch ein Mix existiert? Die Tendenz scheint in Richtung „Big Bang" zu gehen – lieber jetzt ein radikaler Schritt, genauso wie es die Verhandlungspartner Eclipse und Oracle gerade gemacht haben.

8. Mai 2019

Google I/O: Kotlin #1 für Android

Auf seiner Hauskonferenz hat Google verkündet, dass Kotlin die bevorzugte Programmiersprache für Android werden soll. Bereits jetzt sollen schon mehr als die Hälfte der Android-Entwickler die JVM-Sprache verwenden.

9. Mai 2019

GraalVM ist produktionsreif

Oracle hat die GraalVM mit Release 19.0 nach längerer Trial-Phase offiziell als produktionsreif angekündigt. Neben der quelloffenen Community Edition bietet der Hersteller die kostenpflichtige Enterprise Edition an, die noch einmal eine deutlich verbesserte Performance durch weitere Optimierungsverfahren aufweisen soll, sowie mehr Sicherheit durch einen „managed mode" für das Einbinden nativer Libraries – inklusive Entfernung nicht benötigten Codes, um die Angriffsoberfläche zu reduzieren. GraalVM Native Image, das für rasend schnelle Startup-Zeiten im niedrigen Millisekundenbereich sorgt, wird noch als „Early Adopter"-Feature angesehen, das zusätzlich installiert werden kann. Allerdings wird es ja bereits von diversen anderen Frameworks wie Quarkus, Micronaut und Oracles eigenem Helidon benutzt – ganz so „early" scheint es daher nicht mehr zu sein.

22. Mai 2019

Eclipse Jakarte EE Developer Survey

Die Ergebnisse des diesjährigen Jakarta EE Developer Survey, durchgeführt im März, sind da: Wenig überraschend hat die Nutzung von Java EE 8 unter den knapp 1.800 Teilnehmerinnen und Teilnehmern deutlich zugelegt – ganz vorn liegt jedoch immer noch EE 7. „Cloud Native" und Microservices stehen hoch im Kurs – auch keine Sensation. Bei den Frameworks für „Cloud Native Applications" liegt Spring Boot deutlich vor Kubernetes und MicroProfile (das aber in der Nutzung stark zugelegt hat). Die Top-Prioritäten der Community für die nähere Zukunft sind laut Survey bessere Unterstützung von Microservices und native Integration mit Kubernetes sowie produktionsreife Referenzimplementierungen (ein vermeintlicher Widerspruch zum trotzdem erfolgreichen MicroProfile, das ja bewusst auf die reine Spezifikation setzt). Mehr Details im Eclipse Foundation Blog *[3]*.

24. Mai 2019

Awesome Annotation Processing

Ein Hoch auf Annotations! Gunnar Morling (unter anderem Spec Lead von Bean Validation), hat eine kuratierte Liste von Ressourcen rund um das Thema Annotation Processing gestartet *[4]*.

29. Mai 2019

Jakarta: Java EE Guardians sind gespannt

Die „Java EE Guardians" haben sich offiziell zum Abbruch der Verhandlungen um die Jakarta-Namensrechte geäußert. Sie nehmen die Situation gelassen und begrüßen, dass der Stillstand ein Ende hat. Zumindest aktuell haben sie recht, es wird in der Community so viel debattiert wie lange nicht mehr – hoffentlich hält der Tatendrang noch etwas an.

29. Mai 2019

Graal schlägt Excelsior

Die GraalVM hat ein Opfer gefordert. Wie Heise berichtet, hat die Firma hinter dem seit fast 20 Jahren existierenden kommerziellen Ahead-of-time-Compiler „Excelsior JET" die Entwicklung eingestellt

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

und das Entwicklerteam bereits entlassen. Die teils namhaften Kunden erhalten nur noch ein Maintenance Release – und das war's. Schuld soll die GraalVM sein, die diesen Bereich neu aufrollt.

29. Mai 201

Pivotal Spring Runtime

Es gibt eine weitere neue, auf dem OpenJDK basierende Distribution: Pivotal bietet mit der kostenpflichtigen „Spring Runtime" Support für eine eigene Variante des OpenJDK, das Spring Framework plus Spring Boot und einen großen Teil der einzelnen Spring-Projekte sowie für Apache Tomcat und die Pivotal-Variante tc-Server.

1. Juni 2019

Clojure zu Java

Das Apache-Storm-Projekt hat Release 2.0 des Big-Data-Frameworks herausgebracht. Das Erstaunliche dabei: Der Clojure-Kern wurde durch eine Java-Implementierung ersetzt. Das Refactoring trägt laut Release-Nachrichten zu besserer Performance und erhöhter Wartbarkeit bei. Ein wesentlicher Grund war allerdings, dass das Nischendasein-fristende Clojure wohl eine Hürde beim Gewinnen von ausreichend vielen Mitstreitern darstellte. Wenn sie den letzten Trendmeldungen gefolgt wären, hätten sie doch eher Kotlin nutzen sollen…

11. Juni 2019

MicroProfile 3.0

Das neue Major Release von MicroProfile ist da. Es beinhaltet Updates zu den folgenden APIs: Rest Client (1.3), Metrics (2.0) und Health Check (2.0). Die letzten beiden Updates enthalten „breaking changes", wie schon aus dem Versionssprung ersichtlich, daher hat auch das MicroProfile insgesamt einen „Major"-Versionssprung gemacht.

Referenzen

[1] *https://insights.stackoverflow.com/survey/2019*
[2] *https://blogs.eclipse.org/blogs/mike-milinkovich*
[3] *https://eclipse-foundation.blog/2019/05/10/results-2019-jakarta-ee-developer-survey*
[4] *https://github.com/gunnarmorling/awesome-annotation-processing/*

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' eclipse-Corner

... und plötzlich ging alles ganz schnell: Nach monatelangem Gezerre um Trademarks, Projekt- und Package-Namen, Copyrights und Infrastrukturen sind nun endlich für uns Normalsterbliche sichtbare (und nutzbare) Fortschritte von allen Schauplätzen zu vermelden.

Aufgrund der gescheiterten Verhandlungen zwischen der Eclipse Foundation und Oracle über ein markenrechtliches Abkommen werden alle Projekte sowie der Package-Namensraum „javax.*" definitiv umbenannt – und zwar sofort. Hierzu wurden alle Projektbeteiligten aufgefordert, noch Anfang Juni einen neuen Namen sowie eine neue Zielbeschreibung vorzulegen und durch das federführende Project Management Committee (PMC) genehmigen zu lassen. Dieser Prozess ist derzeit in vollem Gange. Als erstes Projekt stürmte einmal mehr JAX-RS voran, das die Umbenennung bereits abgeschlossen hat: Im Internet firmiert Java's RESTful API bereits seit einigen Wochen als Jakarta RESTful Web Services *[1][2]*.

Aber auch in anderer Hinsicht ist JAX-RS ein Vorreiter: Der Quellcode für das anstehende Release 2.2 des API befindet sich seit geraumer Zeit nicht mehr im Namensraum „javax.ws.rs", sondern ist bereits nach „jakarta.ws.rs" gewandert. Andere APIs werden bald nachziehen, denn wie sich nach langer und offener Diskussion herauskristallisiert hat, ist die Community – Anwender wie Hersteller – mit großer Mehrheit dafür, diesen harten Schnitt sofort für alle Projekte durchzuführen. Der Gegenvorschlag, APIs erst dann in das neue Java-Package umzuziehen, wenn diese auch tatsächlich verändert wurden, wird nur von einer Minderheit unterstützt. Das entscheidende Argument hierbei: Lieber ein Ende mit Schmerzen als Schmerzen ohne Ende. Das heißt, besser jetzt sofort die Import-Deklarationen aller Bestandsanwendungen von „javax" auf „jakarta" umzuschreiben und ab dann für alle Zeit Frieden zu haben, als bei jedem Jakarta EE Release wieder und wieder die Importe anzufassen.

Jetzt geht es also definitiv los: Jeder, der seinen Bestandsanwendungen mehr als nur Bug Fixes angedeihen lassen möchte, darf schon mal in die Tasten greifen und einen Pull Request (PR) für den Tag des „Big Bang" vorbereiten, wie die Community diese Singularität beschreibt – den Tag, an dem Jakarta EE 9 releast wird und mit einem Schlag alle Dependencies brechen. Der Tag, an dem das WORA-Versprechen („Write Once, Run Anywhere") für einen kurzen Moment in der langen Java-Geschichte ernsthaft gebrochen wird.

Wer seine umgestellten Apps vorher kompilieren möchte, muss sich derzeit noch einer Menge hausgebrauter Dependencies bedienen. Um diesen Umstand zu beseitigen, sind Nightly Builds von unschätzbarem Wert – auch, um neue Features schon vor dem Release zu testen. Und wieder ist JAX-RS Erster im Bunde und bietet diese seit Juni auf dem Eclipse Maven Repository *[3]* an.

Um das Glück perfekt zu machen, sind nun auch TCK und Spezifikationen an der Reihe. Der bisherige TCK-Monolith wird aufgesplittet und an die einzelnen API-Projekte übergeben. Das ist eine so große Aufgabe, dass hierzu händeringend um Hilfe aus der Community aufgerufen wird. Bitte meldet euch bei uns! Auch für die Spezifikationsdokumente bittet die EF um Hilfe: Der AsciiDoc-Quellcode muss überarbeitet werden, um sicherzustellen, dass keine Namensrechte verletzt werden – der Kreis schließt sich.

Referenzen

[1] *https://github.com/eclipse-ee4j/jaxrs-api*
[2] *https://projects.eclipse.org/projects/ee4j.jaxrs*
[3] *https://github.com/eclipse-ee4j/jaxrs-api/wiki/Nightly-Builds*

Markus Karg
markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Unbekannte Kostbarkeiten des SDK Heute: Korrekte Ganzzahlarithmetik

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Javas Ganzzahlarithmetik zeigt Zahlenüberläufe nicht an und verstößt so gegen die zwölfte Regel der Unix-Philosophie, die *„Rule of Repair“*, die ausgeschrieben lautet: „When you must fail, fail noisily and as soon as possible.“ Mit Java 8 wurden Arithmetikoperationen eingeführt, die Zahlenüberläufe in Form einer Exception anzeigen.

Korrekte Ganzzahlarithmetik der Klasse „Math“

Javas Ganzzahlarithmetik rechnet falsch. Addiert man 1 zur größten darstellbaren Integer-Zahl (2147483647, `Integer.MAX_VALUE`), erhält man die kleinste darstellbare Integer-Zahl (-2147483648, `Integer.MIN_VALUE`). Java ist mit dieser Rechenschwäche jedoch nicht allein, viele andere Sprachen machen dies genauso. Mit Java 8 erhielt die Klasse `java.lang.Math` eine Reihe von Methoden, die „Exact“ im Namen tragen und die in *Listing 1* aufgeführt sind.

Das „Exact“ im Namen sollte allerdings besser nicht mit *„exakt“*, sondern eher mit *„korrekt“* übersetzt werden, da die Ergebnisse nunmehr tatsächlich korrekt sind. Zahlenüberläufe wie im obigen Beispiel führen zu einer `ArithmeticException` und sind somit kein Ergebnis.

Für uns Java-Entwickler stellt sich nun die Frage, ob wir alle Ausdrücke mit Integer-Arithmetik entsprechend umstellen beziehungsweise zukünftig ausschließlich so verwenden sollten. Aus `a + b` wird dann allerdings `Math.addExact(a, b)`, aus `i++;` wird `i = Math.incrementExact(i);` – sieht nicht gerade sehr verlockend aus.

Wie häufig in der Software-Entwicklung gibt es einen Trade-off und es muss die Frage beantwortet werden: Lieber schöneren, an die Mathematik angelehnten Code oder korrekten und damit sicheren Code?

Aber halt, da war doch noch was? Genau: Performanz. Um wie viel langsamer wird korrekte Ganzzahlarithmetik? Wir haben für die Integer-Addition einen JMH-Benchmark aufgesetzt, der eine um nur ca. ein Prozent erhöhte Laufzeit für die korrekte Arithmetik ausweist. Wie kann das sein? Ist es nicht wesentlich aufwendiger, einen Zahlenüberlauf zu erkennen? Die Auflösung ist recht trickreich, wie ein Blick in den Quell-Code für die Integer-Addition zeigt *(siehe Listing 2)*.

Da bitweise Operationen in der Regel schneller als Arithmetikoperationen sind, ist der Mehraufwand vernachlässigbar. Die Tests für die anderen Methoden sind ähnlich genial. Wir empfehlen hier dem Leser insbesondere einen Blick in die Implementierung der Long-Addition.

Zusammenfassung

Seit Java 8 unterstützt Java eine verlässliche Ganzzahlarithmetik, die Zahlenüberläufe durch eine Exception anzeigt. Der Gewöhnungsaufwand und Codierungsmehraufwand sind durchaus nicht zu unterschätzen, der Laufzeit-Overhead dagegen vernachlässigbar.

```
public static int addExact(int x, int y)
public static long addExact(long x, long y)
public static int subtractExact(int x, int y)
public static long subtractExact(long x, long y)
public static int multiplyExact(int x, int y)
public static long multiplyExact(long x, long y)
public static int incrementExact(int a)
public static long incrementExact(long a)
public static int decrementExact(int a)
public static long decrementExact(long a)
public static int negateExact(int a)
public static long negateExact(long a)
public static int toIntExact(long value)
```

Listing 1

Bernd Müller
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

```
public static int addExact(int x, int y) {
    int r = x + y;
    // HD 2-12 Overflow iff both arguments have the opposite sign of the result
    if (((x ^ r) & (y ^ r)) < 0) {
        throw new ArithmeticException("integer overflow");
    }
    return r;
}
```

Listing 2

Making Shell Scripts Groovy

Georg Berky, Valtech Mobility

Groovy ist eine dynamische Sprache auf der JVM und glänzte schon einige Jahre vor Java mit Features wie funktionaler Programmierung oder Literals für oft benutzte Datenstrukturen wie `List` *und* `Map`*. Weniger bekannt ist, dass Groovy auch mächtige Werkzeuge zum Arbeiten auf der Kommandozeile mitbringt, mit denen man – allein durch Kenntnis von Java – schon eigene Skripte schreiben kann, ohne sich gleich auf die komplizierte Syntax mancher Shells einlassen zu müssen.*

Getting Groovy – Installation und Setup

Der einfachste Installationsweg für Groovy ist `sdkman`, ein CLI-Tool, mit dem man eine Menge SDKs installieren und zwischen ihren Versionen hin- und herschalten kann. Der Installationsvorgang von `sdkman` wird in *Listing 1* gezeigt. Die Installation von Groovy ist in *Listing 2* zu finden.

A Groovy Kind of Java – Erste Schritte

Der erste Schritt in einer neuen Programmiersprache ist in unserer Zunft das „Hello World". Groovy ist eine syntaktische Obermenge von Java. Das bedeutet, dass wir einfach Java-Code schreiben können und es dadurch schon valides Groovy ist *(siehe Listing 3)*. Statt `.java` verwenden wir `.groovy` als Dateiendung *(siehe Listing 4)*.

```
curl -s "https://get.sdkman.io" | bash
```

Listing 1: sdkman installieren

```
sdk install groovy
```

Listing 2: Groovy installieren

Groovy wäre nicht „groovy", wenn es uns hier nicht einiges an Schreibarbeit abnehmen würde: Wir brauchen weder die Definition der Klasse noch die Main-Methode, sondern können einfach schreiben, als wären wir direkt in „main". Beide werden von Groovy generiert. Semikola können wir ebenfalls weglassen *(siehe Listing 5)*.

Um unser Skript zu einem First-Class-Citizen auf der Shell zu machen, muss es sich in die Unix-Philosophie der kleinen, zusammensetzbaren Programme einfügen. Dazu muss es wie andere Programme ausführbar sein und Daten über Pipes als Input akzeptieren.

Ausführbarkeit bekommen wir, indem wir eine Shebang `#!` an den Anfang unseres Skripts setzen. Damit signalisieren wir der Shell, dass sie einen neuen Prozess mit dem hinter der Shebang angegebenen Interpreter starten soll. Der Rest der Datei wird zum Input für den Interpreter *(Listing 6)*.

Danach machen wir die Datei noch für die Shell ausführbar. Um gänzlich zu verbergen, dass wir es mit Groovy zu tun haben, können wir noch die Dateiendung entfernen. Wir brauchen sie nicht mehr, wenn unser Skript mit einer Shebang anfängt *(Listing 7)*.

Pipes

Wir sind jetzt so weit, dass wir unsere Skripte auf der Kommandozeile starten können, aber wie interagieren wir mit anderen Prozessen auf der Shell? Das Unix-Bordmittel der Wahl sind Pipes.

Eigenschaften von Pipes

Pipes sind zeilengepufferte Kanäle zur Interprozesskommunikation. Sie verhalten sich wie reguläre Unix-Dateien mit einem schreibbaren und einem lesbaren Ende. Beide Enden haben einen eigenen Dateideskriptor. Zeilen, die am schreibbaren Ende geschrieben werden, kommen in derselben Reihenfolge am lesbaren Ende auch wieder aus der Pipe (FIFO: „First In, First Out").

Pipes haben eine interne Kapazität. Schreibt man zu viel, blockiert der Schreibvorgang, bis am anderen Ende gelesen wird. Ebenso blockieren lesende Operationen, wenn die Pipe leer ist.

Durch die Zeilenpufferung kann man aus einer Pipe erst lesen, wenn das Schreiben einer Zeile mit `\n` abgeschlossen wurde. Wenn sich der letzte schreibende Prozess von der Pipe trennt, empfangen die Konsumenten ein `EOF` (End of File). Auch das schließt die Verarbeitung einer Zeile ab.

Die Standard-Pipes: stdin, stdout, stderr

Die bekanntesten Pipes sind `stdin`, `stdout` und `stderr`. Bevor wir ein Programm starten, ist `stdin` auf der schreibbaren Seite der Pipe mit unserem Keyboard verbunden. Das Gegenstück `stdout` ist auf der lesbaren Seite mit dem Terminal verbunden, ebenso wie `stderr`, das zur Ausgabe von Fehlermeldungen getrennt vom normalen Output verwendet wird. Zwischen `stdin` und `stdout`/`stderr` sitzt verarbeitend die interaktive Shell.

Pipes und Groovy

In der JVM ist `stdin` mit dem `InputStream` unter `System.in` verbunden. Für die Ausgabe sind `stdout` und `stderr` mit einem `PrintStream` unter `System.out` und `System.err` verbunden.

```
public class Hello {
    public static void main(String [] args) {
        System.out.println("Hello World!");
    }
}
```

Listing 3: Hello World in Java und Groovy

```
➜ groovy HelloWorld.groovy
Hello World!
```

Listing 4: Hello World mit Groovy starten

```
println "Hello World!"
```

Listing 5: Hello World in Groovy

```
#!/usr/bin/env groovy
println "Hello World!"
```

Listing 6: Hello World mit Shebang

```
➜ chmod +x HelloWorld.groovy
➜ mv HelloWorld.groovy hello_world
➜ ./hello_world
Hello World!
```

Listing 7: Hello World ausführbar auf der Shell

Wir wollen als erstes Beispiel grob die Funktionalität von `grep` nachbauen. Unsere Eingabedatei `README.md` hat den in *Listing 8* angegebenen Inhalt.

Gebaut für die Unendlichkeit

Pipes können unendlich lange Streams von Daten sein. Wir wählen einen funktionalen Ansatz, um nicht in Speicherprobleme durch Zwischenspeichern zu laufen *(siehe Listing 9)*.

Die Methode `eachLine` ist eine Erweiterung des API von `InputStream` und gibt uns den Inhalt des Streams Zeile für Zeile. Die Variable `it` kommt ebenfalls von Groovy. Sie ist der Standardname bei Closures, die nur einen Parameter haben.

Dadurch, dass wir die Ergebnisse weiter nach `stdout` schreiben, können andere Programme auf der Shell diese ohne zusätzlichen Aufwand weiterverarbeiten, wie in *Listing 10* gezeigt wird. Hier geben wir die Ausgabe unseres Skripts mit einer Pipe an das Unix-Programm `wc` weiter, das mit dem Schalter `-l` die Anzahl der Zeilen zählt, die es liest. Die Eingabe bekommen wir über den Umleitungsoperator `<` für `stdin` aus der Datei `README.md`. Das funktioniert, weil sich Pipes wie reguläre Dateien verhalten. Alle Groovy-Erweiterungen des JDK können übrigens unter *[2]* nachgelesen werden.

Ausblick: Named Pipes

Will man mehr als zwei Prozesse miteinander kommunizieren lassen, verwendet man dafür normalerweise „Named Pipes". Details zu dieser Thematik sind im Blog des Autors zu finden.

```
# groovy-cli #
Examples demonstrating how to use groovy on the command line
This is a line without our keyword
this line is groovy
```

Listing 8: Eingabedatei für den Nachbau von grep

```
#!/usr/bin/env groovy
System.in.eachLine {
    if(it.contains("groovy")) {
        println it
    }
}
```

Listing 9: Unser Nachbau von grep

```
➜ ./3_pipes.groovy < README.md | wc -l
3
```

Listing 10: Interaktion mit anderen Shell Tools

```
.
├── hello
│   └── Hello.groovy
└── hello.groovy
```

Listing 11: Repository mit einem Package

```
#!/usr/bin/env groovy
import hello.Hello
def hello = new Hello()
hello.doSomething()
```

Listing 12: Klasse im Package verwenden

```
.
├── src
│   └── bar
│       └── Foo.groovy
└── test
    └── bar
        └── FooTest.groovy
```

Listing 13: Getrennte Test- und Produktionsklassen

```
➜ groovy -cp test:src test/bar/FooTest.groovy
JUnit5 launcher: passed=1, failed=0, skipped=0, time=51ms
```

Listing 14: Test- und Produktionscode auf dem Classpath

```
#!/usr/bin/env groovy
import org.junit.jupiter.api.Test

class MyTest {

    @Test
    void "simple JUnit 5 Test"() {
        assert 1 + 1 == 3
    }
}
```

Listing 15: Erster Test mit Groovy und JUnit 5

Test Driven Groovy Scripting

Skripte werden oft für Massen-Tasks oder kritische Administration verwendet. In solchen Fällen ist es sinnvoll, ihre Logik zu testen. Als Simulation echter Logik wollen wir ein Skript schreiben, das Zeilen aus einer Pipe liest, diese als Zahl interpretiert und sie inkrementiert. Dies wollen wir testen – einmal in einem isolierten und einmal in einem integrierten Test *[3]*. Groovy bringt dafür bereits viele Werkzeuge mit. Bevor wir mit dem Testing loslegen, möchte ich zunächst die grundlegenden Hilfsmittel für unsere Tests vorstellen.

Source Code organisieren

Die meisten meiner Skripte sind recht klein und gehen nicht über wenige Klassen hinaus. Ab einer gewissen Größe kann es jedoch sinnvoll sein, den Code in Packages aufzuteilen.

Packages

Angenommen, mein Repository sähe so aus, wie in *Listing 11* gezeigt, dann könnte ich mit diesem Skript-Layout `hello.groovy` auf die Klasse `Hello`, wie in *Listing 12* beschrieben, zugreifen.

Tests von Produktionsklassen trennen

Wenn das Repository größer wird, lohnt es sich, Test- und Produktionscode zu trennen, um nicht aus Versehen Test-Code in Produktionsklassen zu verwenden.

Unser Produktionscode liegt in Packages unter `src`. Wie in Maven und Gradle bekommt jedes Package ein eigenes Verzeichnis. Unter-Packages werden zu Unterverzeichnissen. Mit den Tests verfahren wir genauso.

Angenommen unser Repository hätte den Inhalt, der in *Listing 13* zu sehen ist, dann können wir den Test **aus dem Root-Verzeichnis** des Repository starten *(siehe Listing 14)*. Der Aufruf mit `-cp path1:path2` sagt Groovy, wo nach Klassen gesucht werden soll. Wenn wir testen, brauchen wir sowohl den Baum unter `test` als auch den unter `src`. Wenn wir nur Produktionscode starten wollen, nehmen wir `test` einfach nicht in den Classpath auf. So ist dann sichergestellt, dass sich kein Test-Code unter den Produktionscode gemischt hat.

Batteries Included: JUnit + Power Assertions

Groovy bringt inzwischen ganze drei Versionen von JUnit mit: 3, 4 und 5. Man muss keine zusätzlichen Bibliotheken einbinden, sondern kann sofort loslegen. Wir beschränken uns hier auf JUnit 5.

JUnit 5

Auf der Shell ist der einzige Unterschied zu regulären JUnit-Tests, dass wir die bekannte Shebang an den Anfang der Datei setzen müssen. Danach folgen die bekannten Imports und eine Testklasse mit den Annotations von JUnit 5 *(siehe Listing 15)*. Groovy erlaubt Strings als Methodennamen. Dies machen wir uns bei Tests zunutze, um besonders sprechende Namen zu vergeben.


ORAWORLD
Das e-Magazine für alle Oracle-Anwender!
EOUC
EMEA ORACLE USERGROUP COMMUNITY
• Spannende Geschichten aus der Oracle-Welt
• Technologische Hintergrundartikel
• Leben und Arbeiten heute und morgen
• Einblicke in andere User Groups weltweit
• Neues (und Altes) aus der Welt der Nerds
• Comics, Fun Facts und Infografiken
Jetzt Artikel einreichen oder Thema vorschlagen!
Jetzt e-Magazine herunterladen
www.oraworld.org

```
➜ ./5_junit5.groovy
JUnit5 launcher: passed=0, failed=1, skipped=0, time=51ms

Failures (1):
  JUnit Jupiter:MyTest:simple JUnit 5 Test()
    MethodSource [className = 'MyTest', methodName = 'simple JUnit 5 Test', methodParameterTypes = '']
    => Assertion failed:

assert 1 + 1 == 3
         |   |
         2   false
```

Listing 16: Fehlgeschlagener Test

Wenn wir den Test starten, bekommen wir eine Fehlermeldung *(Listing 16)*. Die sprechenden Fehlermeldungen sind ein Feature von Groovy, die „Power Assertions". Wir sehen alle Teile der fehlgeschlagenen Assertion auf einen Blick: die ganze Expression, ihre Teile und das Ergebnis ihrer (Teil-)Auswertungen. Bei komplexeren Evaluationen hilft uns das enorm bei der Analyse des Problems.

Stubs und Mocks

Groovy kann sowohl interpretiert als auch kompiliert ausgeführt werden. Skripte und auch Tests in Skripten werden vorkompiliert. Deswegen können sie dynamische Features wie `StubFor` und `MockFor` nicht verwenden. Stattdessen werden wir das Feature „Map Coercion" benutzen, das uns erlaubt, Maps mit dem Coercion-Operator `as` zu Implementierungen von Interfaces zu machen. Dies funktioniert auch im kompilierten Kontext. Jetzt haben wir alle Hilfsmittel zusammen, um mit dem Testing zu starten.

Unit-Tests

Unit-Tests sollen isoliert von der Außenwelt laufen und nichts von Dateien, Netzwerk, Datenbanken oder – frei nach Robert Martin – anderen „Implementierungsdetails" *[4]* aus der Außenwelt unseres Systems wissen. Um unsere Komponenten isoliert testbar zu machen, müssen wir das Design unseres Skripts entsprechend wählen. Zum Schluss werden wir das ganze System in einem integrierten Test prüfen.

Incrementer

Die reine Logik unseres Skripts beschränken wir deswegen darauf, eine Zahl zu nehmen und die inkrementierte Zahl zurückzuliefern. Woher die Zahl kommt, ist irrelevant. Die Außenwelt ist ein Implementierungsdetail. Das entkoppelt das Holen der Zahl vom Inkrementieren. Wir beachten das Single Responsibility Principle und unser Skript wird dadurch ohne einen anderen Prozess oder eine Datei als Input für den Stream testbar *(Listing 17)*. Auch TDD funktioniert auf der Kommandozeile gut. Getrieben von unseren Tests sieht die Logik der Produktionsklasse zum Schluss wie in *Listing 18* beschrieben aus.

Sie haben vielleicht bemerkt, dass ich keine Shebang an die Klassen geschrieben habe, obwohl dies möglich wäre. Shebangs schreibe ich normalerweise nur an den Einstiegspunkt unseres Systems, um diesen als solchen zu markieren. Den Test starte ich auf traditionelle Weise *(siehe Listing 19)*.

Deserializer

Der Deserialisierer soll eine Zeile lesen, die Zeile in eine Zahl umwandeln und diese an den Incrementer weitergeben. Letzteres werden wir durch einen Interaktionstest gegen einen Mock verifizieren. Die Zeile selbst wird später aus der Pipe kommen. Den Mock erzeugen wir durch die oben erwähnte Map Coercion: Wir weisen dem Key `increment` der Map eine Closure zu. Durch die Coercion der Map wird sie zu einer Interface-Implementierung mit einer Methode `increment(int number)`.

Der Mock nimmt das übergebene Argument an und speichert es im Field `receivedDeserialized`. Dieses prüfen wir in unseren Tests. Wenn nichts gespeichert wird, bleibt das Field `null` und ist damit nach „Groovy Truth" `false` *(siehe Listing 20)*. Der Produktionscode sieht getrieben von unseren Tests so aus, wie in *Listing 21* gezeigt.

```
import org.junit.jupiter.api.*

class IncrementerTest {

    private def sut = new Incrementer()

    @Test
    void "increment 1"() {
        assert sut.increment(1) == 2
    }

    //gleicher Test für 2 und 3 ...

}
```

Listing 17: Tests für den Incrementer

```
class Incrementer {
    public int increment(int number) {
        return number + 1
    }
}
```

Listing 18: Testgetriebener Produktionscode des Incrementer

```
➜ groovy IncrementerTest.groovy
JUnit5 launcher: passed=3, failed=0, skipped=0, time=13ms
```

Listing 19: Tests des Incrementer starten

```
import org.junit.jupiter.api.*

class DeserializerTest {

    private def receivedDeserialized

    private def incrementerMock
    private def sut

    @BeforeEach
    void "setup"() {
        //map coercion!
        incrementerMock = [increment: { int number ->
            receivedDeserialized = number }] as Incrementer

        sut = new Deserializer(incrementerMock)
    }

    @Test
    void "deserialize 1 on one line"() {
        def line = "1\n"

        sut.deserialize(line)

        assert receivedDeserialized == 1
    }

    //gleicher Test für 2 und 3...

    @Test
    void "deserialize 1 on one line without newline"() {
        def line = "1"

        sut.deserialize(line)

        assert receivedDeserialized == 1
    }

    @Test
    void "deserialize non-numbers does not call increment"() {
        def garbage = "lkajdfoiaer"

        sut.deserialize(garbage)

        assert !receivedDeserialized //groovy truth
    }
}
```

Listing 20: Testcode Deserializer

```
class Deserializer {

    private def incrementer

    Deserializer(Incrementer incrementer) {
        this.incrementer = incrementer
    }

    Integer deserialize(String line) {
        try {
            def deserialized = Integer.parseInt(line?.trim())

            return incrementer.increment(deserialized)
        }
        catch(NumberFormatException e) {
            System.err.println("Malformed message: '${line}'")
        }
    }
}
```

Listing 21: Produktionscode Deserializer

```
#!/usr/bin/env groovy
import java.util.stream.*

def numbers = Stream.iterate(0G) { i ->
    i.add(1G)
}

numbers.each {
    println it
    sleep 1000
}
```

Listing 22: Produktionscode Producer

```
import org.junit.jupiter.api.*

class PipeConsumerTest {

    @Test
    void "consumer increments text from stdin"() {
        def process = "./pipe_consumer.groovy".execute()
        def stdin = new PrintStream(
            process.getOutputStream(), true) //autoFlush
        def stdout = process.getInputStream()

        stdin.println("1")
        stdin.println("2")
        stdin.println("3")
        stdin.close() //EOF

        assert stdout.readLines() == ["2", "3", "4"]
    }
}
```

Listing 23: Tests des Consumer

```
#!/usr/bin/env groovy
def deserializer = new Deserializer(new Incrementer())

System.in.eachLine {
   println deserializer.deserialize(it)
}
```

Listing 24: Produktionscode Consumer

```
➜ ./pipe_producer.groovy | ./pipe_consumer.groovy
1
2
3
...
```

Listing 25: Producer und Consumer zusammen starten

Die Fehler geben wir auf `System.err` aus. Damit können wir sie – falls nötig – gesondert weiterverarbeiten. Sie landen also nicht in den Pipes, die wir mit dem `System.out` unseres Skripts verbinden.

Producer

Der Producer produziert durch `BigInteger` und `Stream.iterate()` einen potenziell unendlich langen Stream von Zahlen, die er direkt nach `System.out` schreibt. Das `G` hinter den Zahlen macht sie zu `BigInteger` *(siehe Listing 22)*. Da der Producer unendlich viele Zahlen produziert, ist es nicht möglich, ihn direkt zu testen.

Consumer

Um den Consumer integriert zu testen, müssen wir seine Eingaben kontrollieren und seine Ausgaben verifizieren können. Groovy hat das API von `String` um die Methode `execute()` erweitert. Diese führt den Befehl im String in einer Instanz von `Process` aus, auf der wir `stdin` und `stdout` setzen können.

Um in die Eingabe bequem schreiben zu können, verwenden wir einen `PrintStream`. Die Ausgabe lesen wir mit der Methode `readLines()` aus dem Ausgabe-Stream des Prozesses. Auch diese Methode ist eine Erweiterung von Groovy *(siehe Listing 23)*.

Nach dem Aufruf von `execute()` läuft der Prozess eigentlich schon. Da er aber keine Eingaben über `stdin` bekommt, blockiert das Lesen und wir müssen uns nicht darum kümmern, den Prozess erst dann zu starten, wenn wir in sein `stdin` geschrieben haben – ein weiterer Vorteil des Arbeitens mit Pipes.

Die Bezeichnungen der Methoden können anfangs etwas verwirrend sein: Um das `stdin` des Prozesses zu erhalten, muss man `getOutputStream` aufrufen. Die Blickrichtung ist hier aus der Sicht des aufrufenden Prozesses. Gleiches gilt in Gegenrichtung für `stdout` und `getInputStream`. Der Consumer selbst ist einfach gebaut *(siehe Listing 24)*. Der gemeinsame Start von Producer und Consumer ist in *Listing 25* zu sehen.

Dependencies

Das letzte und gleichzeitig mächtigste Werkzeug in unserem Scripting-Koffer ist Groovys Dependency Management Tool „Grape". Es erlaubt uns, zur Laufzeit Bibliotheken nachzuladen und sie in unsere Skripte einzubinden.

Kernstück von Grape ist die Annotation `@Grab`, die als Parameter entweder (wie in Gradle) einen String oder (wie in Maven) die drei Teile der aus Maven bekannten GAV-Parameter nimmt, das Artefakt hinter den Parametern aus Maven Central oder einem

```
//alternativ: @Grab('org.apache.commons:commons-lang:3.8.1')
@Grab(
    group='org.apache.commons',
    module='commons-lang3',
    version='3.8.1')
import org.apache.commons.lang3.*

println StringUtils.joinWith(" ", "Hello", "Java", "Aktuell")
```

Listing 26: Dependencies einbinden

konfigurierten Repository zwischenspeichert und es zur Laufzeit auf den Classpath legt.

In *Listing 26* binden wir die Bibliothek `commons-lang3` aus der Gruppe `org.apache.commons` in Version `3.6` als Dependency ein, um daraus die Klasse `StringUtils` verwenden zu können.

`@Grab` kann überall verwendet werden, wo Annotations erlaubt sind, also insbesondere an Imports, Fields und Klassen. Man kann beliebig viele `@Grab` hintereinander verwenden. Wir können jetzt jede beliebige Bibliothek aus dem JVM-Ökosystem in unseren Skripten verwenden.

Fazit

Groovy ist ein mächtiges Werkzeug für Skripte auf der Kommandozeile. Durch seine starke Ähnlichkeit zu Java hat es eine sehr flache Lernkurve und man kann auch als Shell-Neuling schnell Skripte in einer für Java-Teams les- und wartbaren Sprache entwickeln. Die Skripte sind durch die in Groovy eingebauten Werkzeuge leicht zu testen und können sich mit Grape auch die große Anzahl von Bibliotheken im JVM-Ökosystem zunutze machen.

Quellen und Referenzen

[1] Georg Berky (2018): „Groovy Shell Scripting - Pipes und FIFOs". Never Code Alone

[2] *http://groovy-lang.org/gdk.html*

[3] J.B. Rainsberger (2009): „Integrated tests are a scam". The Code Whisperer

[4] Robert C. Martin (2012): „NO DB". Clean Coder Blog

Georg Berky
georg.berky@valtech-mobility.com

Georg Berky ist leidenschaftlicher Programmierer, hauptsächlich unterwegs in JVM-Sprachen wie Java, Groovy oder Clojure. Seine ersten Schritte hat er in QBASIC, dann unter Linux auf der Kommandozeile und mit C gemacht und konnte bis heute nicht aufhören zu programmieren. Im Arbeitsleben entwickelt er Connected-Car-Dienste bei der Valtech Mobility GmbH.
In seiner Freizeit ist er Co-Organisator der Softwerkskammergruppen Düsseldorf und Ruhrgebiet. Wenn er mal nicht programmiert, praktiziert er Aikido oder spielt Trompete.

Functional Core für einen seiteneffektfreien Anwendungskern

Thomas Ruhroth, Msg Systems und Kai Schmidt, selbstständig

In Programmiersprachen vermischen sich mehr und mehr funktionale, imperative und objektorientierte Aspekte. Java erhält Funktionalitäten wie Lambdas, Streams oder ein Flow-API, die funktional ausgerichtet sind. JavaScript dagegen erhält Konstrukte wie Klassen und Interfaces und scheint sich von dem Fundament der Prototype-Struktur abzukapseln. Man hört von Vorteilen funktionaler Programmierung, was die Lesbarkeit, Testbarkeit und die Fehlersuche in der Anwendung betrifft. Dennoch bergen funktionale Programmiersprachen schwer zu verstehende und schwer greifbare Konzepte wie Monaden. Wie kann man, wenn man imperativ aufgewachsen ist, einen Einstieg in diese Welt finden und mit dem Wandel der Programmiersprachen mitgehen? Wie nutzt man die Vorteile der imperativen und funktionalen Welten und wie setzt man sie in der Anwendungsarchitektur ein? Eine Möglichkeit ist der Architekturstil „Functional Core/Imperative Shell", der imperative Objektorientierung von funktionalen Elementen der Anwendung trennt.

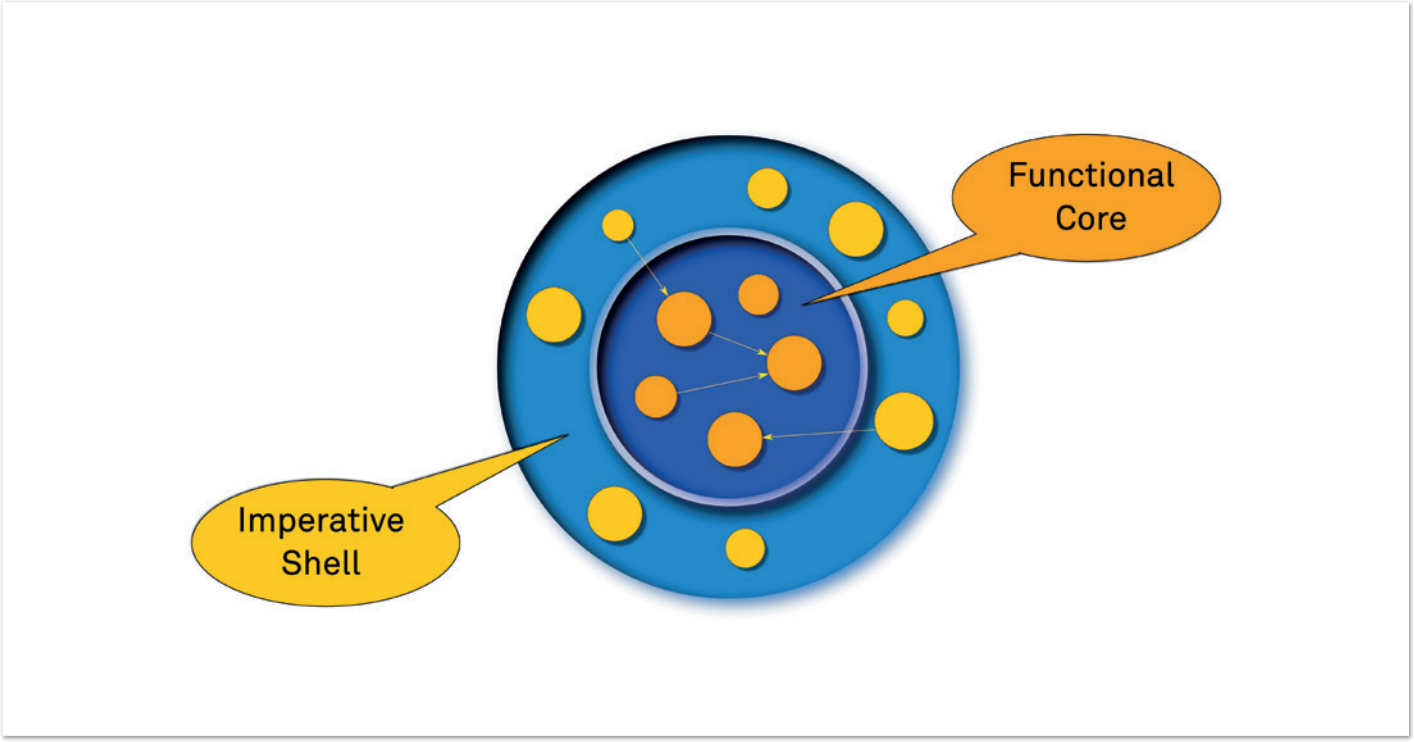

Abbildung 1: Architektur-Schema Functional Core/Imperative Shell (Quelle: Kai Schmidt)

Objektorientierte und funktionale Prinzipien scheinen sich zu widersprechen. Funktionen als Kern der funktionalen Welt sollen keinen Zustand und keine Seiteneffekte haben, wohingegen diese Prinzipien zum Kern der Objektorientierung gehören. Beide lassen sich jedoch sehr elegant vereinen. Objekte lassen sich sehr gut in funktional aufgebauten Methoden verwenden. In diesem Artikel möchten die Autoren zeigen, wie gerade die Kombination dieser beiden Welten entscheidende Vorteile in der eigenen Anwendungsentwicklung haben kann.

Wesentliche Grundlagen funktionaler Programmierung

Während die Objektorientierung als Idee auf der Gruppierung von Daten und direkten Datenmanipulationen aufbaut, ist die wesentliche Idee der funktionalen Sprachen, alle Programmfunktionen als Funktionen aufzufassen (referenzielle Transparenz). Komplexe Abläufe können durch Rekursion und Pattern Matching gestaltet werden. Zentral sind die `puren` Funktionen, die zwei Regeln einhalten müssen:

- Gleiche Eingabewerte liefern das gleiche Ergebnis
- Sie haben keine Seiteneffekte

Die gleichen Ergebnisse können nur sichergestellt werden, wenn die Funktion frei von Zuständen ist. So werden weder Daten aus der Datenbank gelesen noch dürfen die Eingabewerte bei der Verarbeitung verändert werden. Mit Immutable-Klassen stellt Java sicher, dass ihre Instanzen durch eine Funktion nicht verändert werden. Beispielsweise ist jedes String-Objekt „immutable", bei der mittels Funktionen neue Instanzen der Klasse „String" entstehen – die eigentliche Instanz bleibt unverändert.

Seiteneffekte beschreiben Effekte, die Aktionen außerhalb der eigentlichen Berechnungslogik ausführen. Das beinhaltet verschiedene Arten, angefangen bei Veränderungen von Variablen in Objekten bis hin zu Änderungen in I/O-Streams oder in einer Datenbank.

Functional Core/Imperative Shell

Die Idee von Functional Core/Imperative Shell ist es, die Regeln der funktionalen Programmierung für einen Teil der Anwendung zu verwenden (Functional Core), während in dem anderen Teil die zustandshaltenden und seiteneffektbehafteten Techniken aufzufinden sind (Imperative Shell). *Abbildung 1* zeigt, wie die Imperative Shell die Anwendungsfunktionalität unter Ausnutzung des Functional Core bereitstellt. Eine Abhängigkeit des Core auf die Shell ist nicht zulässig. Dieser in der Java-Welt weitgehend noch unbekannt zu scheinende Stil wurde von Gary Bernhardt bereits 2012 auf der RubyConf *[1]* vorgestellt.

Über eine Beispiel-Anwendung zur Vortragsverwaltung einer Konferenz soll dieses Konzept veranschaulicht werden. Der vollständige Quellcode ist unter GitHub *[2]* verfügbar und wird hier auszugsweise und teilweise in abgewandelter Form in den Listings verwendet.

In *Listing 1* ist die Funktion `addNewTalk` dargestellt, die einer Liste (`list`) einen Vortrag (`talk`) nur hinzufügt, wenn dieser nicht `null` ist. Das Erstellen einer neuen, veränderlichen Liste (`mList`) stellt sicher, dass die übergebene Liste (`list`) nicht verändert wird und die Methode somit seiteneffektfrei ist. Die Wertgleichheit des Resultats ergibt sich hier daraus, dass es ausschließlich von den Funktionsparametern abhängt. Die Rückgabe von `List.of` gibt eine Immutable-List zurück, sodass weitere Veränderungen an dieser neuen Liste ausgeschlossen sind.

Die Imperative Shell benutzt den Functional Core, um daraus die Anwendung zu orchestrieren. Sie kann beispielsweise für die Haltung des Anwendungszustandes in Form der vom Core erhaltenen Objekte zuständig sein.

```
public List<Talk> addNewTalk(List<Talk> list, Talk talk){
     var mutableList mList = new ArrayList<Talk>(list)
     if (talk != null) {
        mList.add(talk);
     }
     return List.of(mList);
}
```

Listing 1: Beispiel-Methode im Functional Core

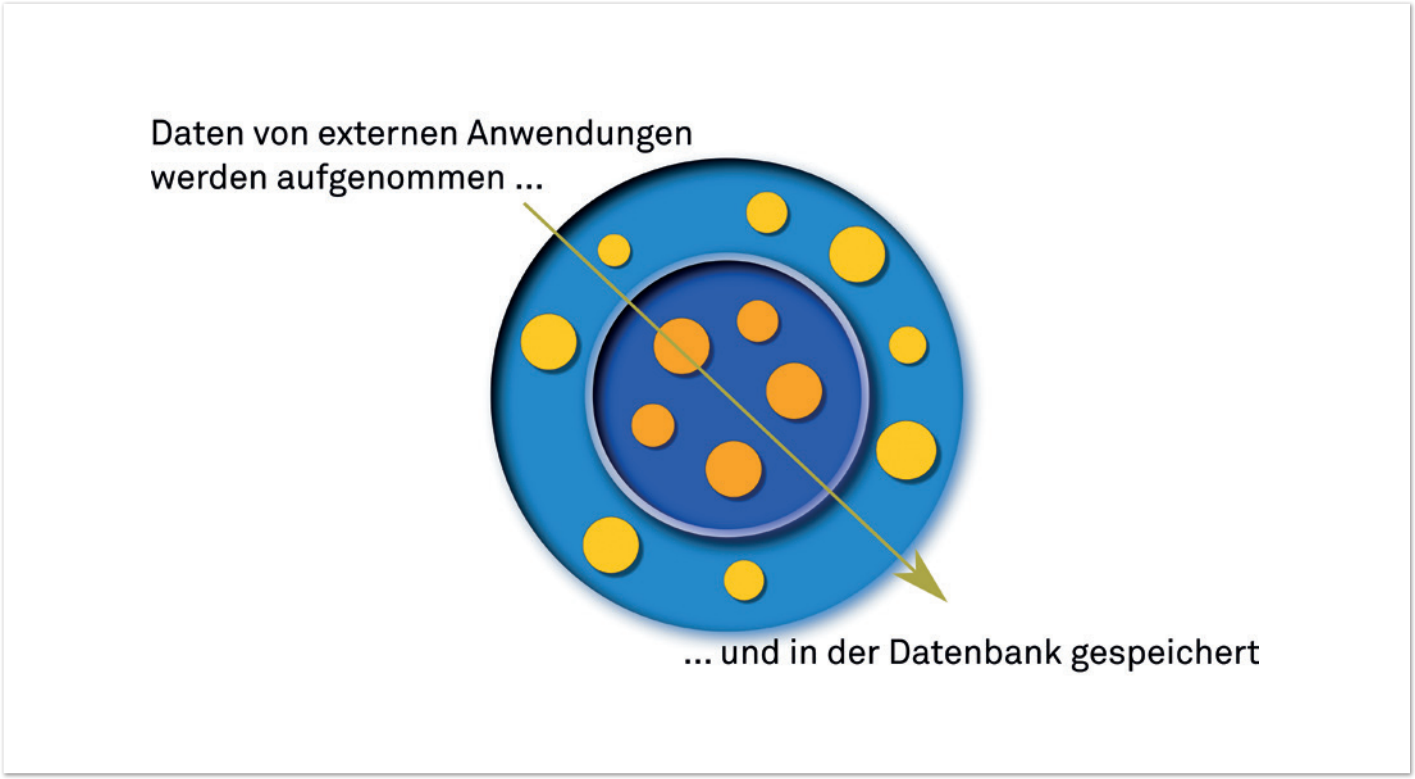

Abbildung 2: Integrationstests als Durchstich (Quelle: Kai Schmidt)

In *Listing 2* wird eine zu *Listing 1* ähnliche Funktion auf einem komplexen Agenda-Objekt ausgeführt. Der neue Zustand nach Ablauf der Methode ist abhängig vom vorigen Zustand. Zusätzlich wird durch das Persistieren auf die Datenbank die Systemumgebung verändert.

Durch die Ausnutzung der Objektorientierung kann eine vereinfachte Schreibweise eingeführt werden. Dabei profitiert man davon, dass das Objekt, das in der rein funktionalen Schreibweise als Parameter übergeben werden muss, bereits die zugehörigen Daten hält. Da es „immutable" ist, kann man direkt auf den State zugreifen und gleichzeitig sicherstellen, diesen nicht zu verändern. So verkürzt sich die Signatur aus *Listing 2* und damit auch dessen Nutzung von `agenda = agenda.addNewTalk(agenda, talk);` zu `agenda = agenda.addNewTalk(talk);`.

Gary Bernhardt nennt dieses Vorgehen in Anlehnung an OO (Objektorientierung) daher FauxO (vom französischen „faux") – also falsche beziehungsweise künstliche Objektorientierung: Die Daten sind zwar nicht vom Code getrennt, es findet jedoch keine Änderung eines Übergabewertes statt.

Testbarkeit im Kern

Die Tatsache, dass der Functional Core nach funktionalen Paradigmen aufgebaut ist, lässt sich für das Testen leicht und in angenehmer Weise nutzen. Bei gleichen Eingabewerten erhält man dasselbe Resultat. Seiteneffekte sind ausgeschlossen und können daher für einen Test nicht relevant sein. Nach dem typischen Muster von „Arrange, Act, Assert" sind die Tests alle gleich aufgebaut: Sie bereiten die Eingabeparameter vor, rufen die zu testende Logik auf und prüfen die Ausgabewerte.

```
public void addNewTalk (Talk talk) {
     agenda = agenda.addNewTalk(agenda, talk);
     someFancyORM.persist(ListEntity.from(agenda.talks));
}
```

Listing 2: Beispiel-Methode der Imperative Shell

```
public class Agenda_When_adding_new_talk {
        @Test
     public void If_talk_does_not_exist_Then_it_is_created() {

     //Arrange
     Agenda agenda = Agenda.initializeAgenda();
     final String topicName = "NewTopic";

     //Act
     agenda = agenda.addNewTalk(topicName);

     //Assert
     AssertThat(agenda.getTalks().iterator().next()
     .getTopic()).isEqualTo(topicName);
     }
}
```

Listing 3: Functional Core Test

```
public class TalkController_When_new_talk_is_created {
     private MockMvc mockMvc;

     @Test
     public void Then_talk_is_shown_in_list() throws Exception
     {

     //Arrange: Neuer Talk ist noch nicht vorhanden
     String newTalk = "TestTalk";
     MvcResult result = mockMvc.perform(get("/"))
                              .andReturn();

     assertThat(result
                     .getResponse()
                     .getContentAsString()
                     .contains(newTalk))
                 .isFalse();

     //Act: Neuen Talk erstellen
     mockMvc
        .perform(post("/talk")
           .content("Vortragsthema=" + newTalk);

     //Assert: Prüfen ob Talk angezeigt wird
     mockMvc.perform(get("/"))
                     .andExpect(status().isOk())
                     .andExpect(content()
                     .string(containsString(newTalk)));
     }
}
```

Listing 4: Imperative Shell Test

Wer bereits Tests für Redux-Reducer oder EventSourcing-Anwendungen geschrieben hat, weiß diesen einfachen Aufbau bereits zu schätzen: Ein bestehender State wird über eine Action beziehungsweise ein Event in einen neuen überführt. Der Test bereitet die Eingabewerte vor, löst das Ereignis aus und gleicht das Resultat mit dem erwarteten Ergebnis ab. Einen beispielhaften Test für den Functional Core zeigt *Listing 3*.

Wie solide ist die Hülle?

Der Test des Functional Core gestaltet sich einfach, jedoch ist die Imperative Shell voll von Seiteneffekten. Die Tests des Functional Core decken glücklicherweise bereits weite Teile (die meisten berechneten Werte) der Anwendung ab. Was fehlt, ist die Sicherstellung der gewollten Seiteneffekte. Folglich müssen noch die Verantwortlichkeiten der Imperative Shell sichergestellt werden; sie hat im Idealfall keine Verzweigungen im Code. Diese sind häufig bereits in den Functional Core gerutscht. Daher müssen nur lineare Pfade

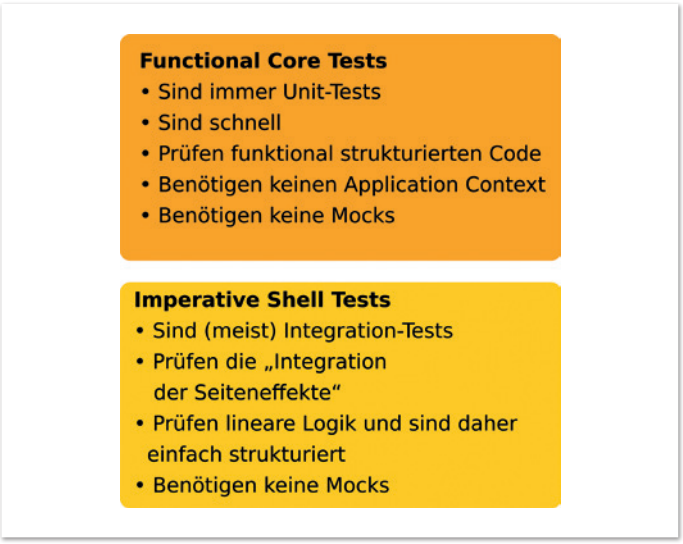

Abbildung 3: Core und Shell Tests im Vergleich (Quelle: Kai Schmidt)

durch die Imperative Shell getestet werden – bei Bedarf mit parametrisierbaren Eingabewerten. Diese Tests kann man wie einen Durchstich durch die Anwendung sehen *(siehe Abbildung 2)*. *Listing 4* zeigt beispielhaft einen solchen Test.

Tests des Functional Core entsprechen charakteristisch Unit-Tests, da es ein Ziel des Functional Core ist, unabhängig von der umgebenden Infrastruktur zu laufen. Demgegenüber stehen die Tests der Imperative Shell, die Integrationstests darstellen. Diese testen beispielsweise, ob Benutzereingaben des Benutzers korrekt in der Datenbank landen und wieder korrekt auf der UI widergespiegelt werden. Sie weisen daher die in *Abbildung 3* genannten Charakteristiken auf.

Die Jagd nach Fehlern

Als Programmierer stellt man fest, dass schwer nachvollziehbare Fehler zu einem hohen Anteil vom Zustand der Anwendung abhängig sind. Häufig lässt sich in imperativen Sprachen nur schwer feststellen, wo es zu Zustandsänderungen innerhalb der Anwendung kommt. Finale Objektreferenzen schützen ausschließlich vor Änderung der Referenz selbst – jedoch nicht vor Veränderung ihrer Inhalte. Diese Änderbarkeit kann praktisch sein, hat aber seine Schattenseiten: Man verliert Hinweise darüber, an welchen Stellen im Code Zustandsänderungen herbeigeführt werden.

Durch die Trennung der Anwendung in funktionale und imperative Bereiche lässt sich diese Eigenschaft einschränken: Allein durch die Lage des Codes innerhalb der Anwendungsarchitektur (also im umgebenden Java-Package) ist erkennbar, wo Zustandsänderungen vorkommen. Innerhalb des Functional Core erleichtert dies die Fehlersuche. Im Programmcode selbst kann man nachvollziehen, wo Änderungen an den Argumenten durchgeführt wurden. Im Debugger kann man deren Werte im Stacktrace zurückverfolgen und die Stellen betrachten, an denen Werte eventuell fehlerhaft zugewiesen wurden.

Lesbarkeit

Durch die Anwendungsarchitektur Functional Core/Imperative Shell ist jede Anwendung in zwei Bereiche unterteilt. Beide Bereiche haben ihre eigenen Charakteristika, sodass man schnell zuordnen kann, in welchem Bereich sich ein Logik-Bestandteil befindet oder wo welche Anpassungen (beispielsweise durch Feature Requests) durchgeführt werden müssen. Außerdem lässt sich im Functional Core durch die Eigenschaften der Unveränderlichkeit der Programmfluss leichter nachverfolgen, wie wir im Abschnitt über die Auffindbarkeit von Fehlern betrachtet haben.

In den Abschnitten über die Testbarkeit konnten wir feststellen, dass die geschriebenen Tests nicht nur die Funktionalität der geschriebenen Anwendung sicherstellen, sondern den Anwendungscode gleichzeitig dokumentieren. Sie teilen sich nahezu automatisch in Unit-Tests und Integrationstests und beschreiben diese meist ohne Zuhilfenahme von Abkapselungstechniken, die zum Beispiel über Mocking-Frameworks erreicht werden. So bleiben Tests leicht nachvollziehbar und konzentrieren sich auf das Wesentliche.

In den funktionalen Teilen der Anwendung dokumentieren bereits die Signaturen der Methoden den Quellcode. Man findet keine Methoden, in denen sich ein „void" in der Methodensignatur eingeschlichen hat. Da man Seiteneffekte ausschließt, würden diese keinen Mehrwert darstellen – welche Aufgabe sollten diese Methoden schon erledigen? Die Tests der Imperative Shell dokumentieren typische Anwendungsflüsse (oder sogar Use Cases), beispielsweise vom Dialog über die Datenbank und gegebenenfalls wieder zurück zu den Daten, die über die Benutzerschnittstelle zur Anzeige gebracht werden.

Rand- und Sonderfälle

Wie häufig in Architekturfragen ist Functional Core/Imperative Shell nicht die Antwort auf alle Probleme. Trotz der beschriebenen Vorteile sollte man sich für jede Anwendung überlegen, ob dieser Stil der passende für das aktuelle Projekt ist. Die Entscheidung hängt von vielen Faktoren ab, angefangen von der zu implementierenden Fachlichkeit bis hin zu den Fähigkeiten des Teams oder sogar zur Organisationsstruktur.

Beispielsweise ist die Trennung von Logik für den Functional Core von der Logik für die Imperative Shell nicht so einfach, wie es anfangs klingt. Angefangen hatten die Autoren mit der Annahme, dass die Fachlichkeit Bestandteil des Functional Core ist, während alles, was nicht zur reinen Fachlichkeit gehört, in die Imperative Shell passt. Es blieb unumstritten, dass die reine Fachlichkeit Bestandteil des Functional Core ist. Der praktikablere Ansatz ist jedoch, dass all jenes, das funktional abbildbar ist, in den Functional Core gehört. Dies kann ebenso querschnittliche Aspekte wie beispielsweise Teile der Autorisierung/Authentifizierung betreffen. Auf diese Weise erhöht sich ebenfalls die Linearität des Imperative-Shell-Codes. Eine hohe Diskussionsbereitschaft im Team und eine gesunde Affinität zu Refactorings schärfen die zugehörigen Sinne.

Der Stil eignet sich gut, sobald auf unveränderlichen Datenstrukturen gearbeitet wird, also beispielsweise in CQRS-Systemen *[3]*. Dort ist durch die Trennung der Schreib- und Lese-Seite der Umgang mit unveränderlichen Strukturen natürlicher. Auf der Schreibseite wird eine Entität beziehungsweise ein Aggregat durch die Anwendung der bisherigen Events rekonstituiert. Änderungen am Aggregat werden nicht direkt am Aggregat durchgeführt, sondern führen zu neuen Events, die wiederum unveränderlich in den Append-Only-Event-Store geschrieben werden. Die verändernde Schnittstelle, die basierend auf den Events die Leseseite aktualisiert, wird schmal, sodass die Datenbank direkt (beispielsweise über Insert und Update-Statements) aktualisiert werden kann.

Bei klassischen CRUD-Anwendungen wird häufig mit Entitäten gearbeitet, die basierend auf den Geschäftsregeln und dem Nutzungsfall direkt verändert werden, bevor sie wieder persistiert werden. Jedoch dürfen Entitäten nicht von Operationen des Functional Core verändert werden. Die im Functional Core ermittelten neuen Ziel-Zustände müssen also in einem separaten Schritt mit der Persistierung zusammengeführt werden. Dies geschieht entweder durch das Überschreiben der Werte der Entität oder über Techniken aus dem ORM-Framework, wie einem `merge` in Hibernate. Sicherlich nicht zu empfehlen ist diese Architektur daher bei Anwendungen, die sich ausschließlich um Datenveränderung kümmern, ohne viel Geschäftslogik zu beinhalten.

Wenn man die Aufgaben des Functional Core betrachtet, kommt die Frage auf, ob im entsprechenden Code Log-Ausgaben erlaubt sein sollten. Klar ist das nach der Theorie, dass dies eine Veränderung des Systemzustands ist. Jedoch wird ein Log nicht die Funktionalität verändern, weil es sich aus Sicht der Anwendung wie ein „write-only"-Speicher verhält. In wissenschaftlichen Dokumenten arbeitet man daher häufig mit einer Abschwächung der puren Funktionen: Bei `effektiv` puren Funktionen beispielsweise sind Seiteneffekte erlaubt, solange diese sich nicht auf das Programm selbst auswirken *[4]*.

Fazit

Die vorgestellte Architektur erlaubt es, Vorteile der funktionalen Welt in Java strukturiert zu nutzen. Selbst wenn man sie nur für Teile verwendet, vereinfacht es die Programmierung. Insbesondere die Regeln aus dem Functional Core können auch in anderen Architekturen vorteilhaft eingesetzt werden, wenn man etwa komplexe Daten durch eigene Datentypklassen nach den Prinzipien von Functional Core gestaltet definiert. Gute, aber sehr einfache Beispiele sind im Java-Framework die Klassen String und LocalDate.

Gleichzeitig ist die Architektur kein Allheilmittel und man muss teilweise einige Erfahrungen sammeln und Refactorings durchführen, bis man sich in diesem Stil zurechtgefunden hat. Die Zuordnung von Code in den Functional Core beziehungsweise in die Imperative Shell sowie die Verwendung unveränderlicher Datentypen benötigt eine gewisse Erfahrung. Mindestens in der Anfangszeit wunderte sich wahrscheinlich jeder, wieso ein String nicht verändert wurde, obwohl man die Methode „substring" oder „replace" darauf aufgerufen hatte. Man gewinnt jedoch leicht testbaren Code mit verständlichen und schnellen Unit-Tests und wenigen, von Natur aus langsameren Integrationstests.

Java wurde bereits mehrfach in Richtung funktionaler Aspekte erweitert und wird sicherlich auch zukünftig weitere Fortschritte machen. Strings haben seit Langem als unveränderliche Datenstruktur in Java ihren Platz gefunden: Unveränderliche Dates (Java 8), Immutable Collections im Stream-API (Java 10) und Erweiterungen Richtung Pattern Matching durch die Erweiterung des Switch-Konstrukts (Java 12) bringen Java den funktionalen Vorteilen näher. Dennoch fehlen weitere Konzepte wie beispielsweise Value Types, die bisher noch keine Berücksichtigung in der Sprache gefunden haben.

Wer weiter in das Thema einsteigen möchte, dem sei insbesondere die Link-Liste von Kasper B. Graversen *[5]* empfohlen. Funktionale Erkenntnisse kann man selbst als Javaianer aus dem sehr gut aufbereiteten Tutorial „Try Haskell!" *[6]* gewinnen.

Quellen

[1] *https://www.youtube.com/watch?v=yTkzNHF6rMs*
[2] *https://github.com/electronickai/functional-core-demo*
[3] *https://martinfowler.com/bliki/CQRS.html*
[4] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif and Mira Mezini (2018) „A Unified Lattice Model and Framework for Purity Analyses" ASE '18, ACM
[5] *https://gist.github.com/kbilsted/abdc017858cad-68c3e7926b03646554e*
[6] *https://tryhaskell.org/*

Thomas Ruhroth
Thomas.Ruhroth@msg.group

Thomas Ruhroth ist Lead IT Consultant bei msg systems AG. In seiner industriellen Arbeitserfahrung arbeitete er als Entwickler, Software-Architekt und Business-Analyst in verschieden Bereichen wie Geographische Informationssysteme und Logistik. In der Forschung liegt sein Fachgebiet in der Softwarespezifikation und in der Entwicklung langlebiger Informationssysteme. Der Wissenstransfer aus der Kombination von Forschungsarbeiten mit industrieller Anwendung ist in vielen seiner Projekte eine treibende Kraft.

Kai Schmidt
mail@kai-schmidt.hamburg

Kai Schmidt ist freiberuflicher Software-Entwickler und -Architekt. Zuvor war er bei den IT-Beratungsunternehmen msg systems AG und Capgemini angestellt und in seiner über zehnjährigen Projekterfahrung größtenteils in Java- und C#-Projekten in den Bereichen Logistik, Flugzeugbau sowie Handel tätig. In seinem letzten Projekt konnte er den hier vorgestellten Architekturstil über ein Jahr lang kennenlernen. Heute berät und beteiligt er sich gern an betrieblichen Anwendungssystemen und ist in der JUG sowie für Kids4IT aktiv.

Wiederbeleben von Legacy-Projekten mit Gradle

Dmitrij Drandarov, msg DAVID GmbH

In diesem Artikel möchte ich einen Ausschnitt unserer Erfahrungen geben, die wir beim Wiederbeleben eines fast drei Jahrzehnte alten Projekts gesammelt haben und noch immer sammeln. Dabei werde ich sowohl auf unser organisatorisches Vorgehen als auch stichprobenartig auf einige Aspekte unserer Nutzung von Gradle eingehen.

Vor ziemlich genau einem Jahr wurde das Projekt initialisiert. Eine fast drei Jahrzehnte alte Software sollte in ein neues Rechenzentrum wandern, aber war aufgrund einer Vielzahl technischer Schulden nicht vom Platz zu bewegen. Wir wurden mit der Wiederbelebung der Software beauftragt, um diese wieder transportfähig zu machen. Zudem sollten mehrere Subprojekte aus Sicherheitsgründen auf aktualisierte Versionen von Programmiersprachen und Frameworks migriert werden. Unsere Vorstellung der Software war von Anfang an die eines totkranken Patienten, den wir versuchen,

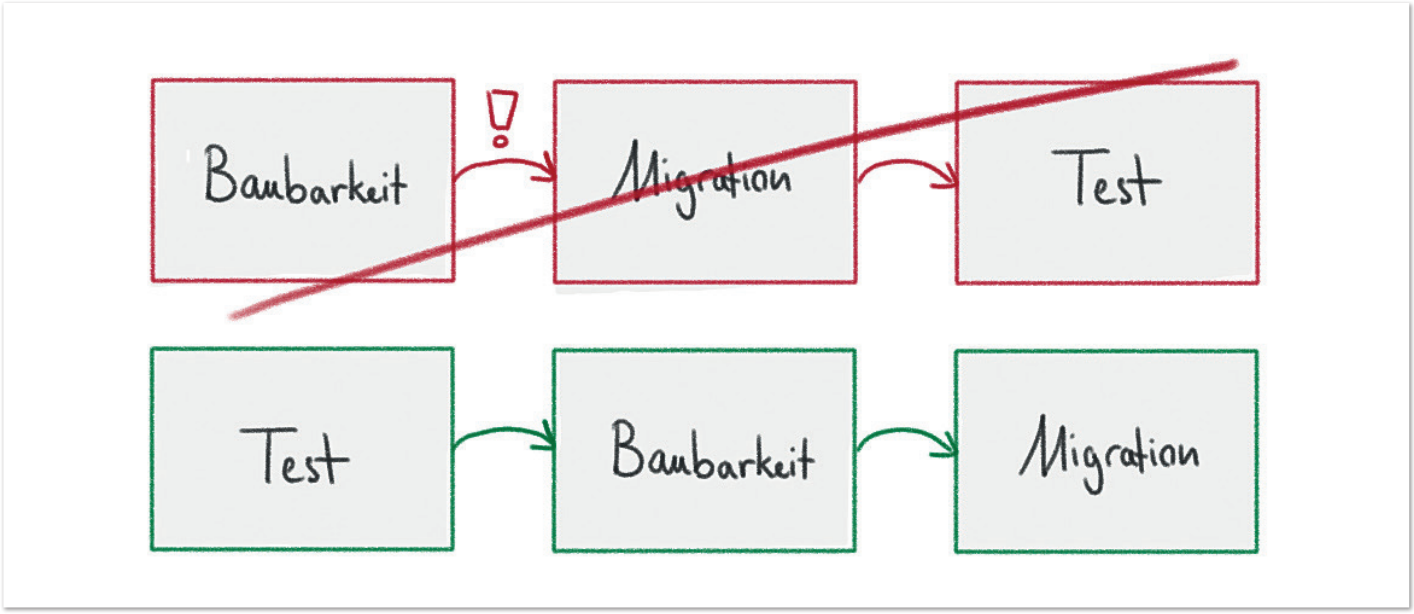

Abbildung 1: Der Workflow des Projekts (Quelle: Dmitrij Drandarov)

auf eine Trage zu hieven und in sein neues Rechenzentrum zu tragen. Die Trage und Stabilisierung wird dabei aus unserem organisatorischen Vorgehen und der Nutzung von Gradle als Build-Management-Tool „gebaut". Diese „Trage" will ich stichprobenartig in diesem Artikel vorstellen.

Unser Vorgehen

Die Codebasis der Software sollte fachlich und inhaltlich möglichst nicht verändert werden. Eine frühzeitige Analyse ergab, dass von den uns überlassenen 80 Subprojekten höchstens eine Handvoll ohne großen Aufwand baubar ist, ca. 30 sind in der Migration von Ant zu Maven stecken geblieben und man kann mehr oder weniger raten, ob die „build-final.xml", „build-finalfinal.xml" oder die „pom.xml" das aktuelle Build-Skript ist. Dabei sind sowohl viele Artefakt-Inhalte, Server-Konfigurationen, als auch zu wählende Build-Skripte abhängig von der Ziel-Stage (Entwicklung, QA, produktiv, etc.). Eine Jar-File muss nicht dieselbe Jar-File auf einem anderen Server oder einer anderen Stage sein. Neben Ant und Maven finden sich auch viele Shell-, Bat- und Make-Skripte, die alle gemeinsam von 20.000 Zeilen Perl-Code orchestriert werden. Ein Blick in den nicht dokumentierten Perl-Code zeigte, dass dieses Konzept nicht zu retten war.

Eine Qualitätssicherung unserer Migration ist ohne Baubarkeit und, daraus folgend, ohne ordentliche Testbarkeit nicht gegeben. Da eine Wiederherstellung der Baubarkeit auf Grund des komplexen Build-Prozesses viel Zeit in Anspruch nehmen würde, haben wir uns folgendes Vorgehen überlegt: Wir invertieren unsere organisatorischen Abhängigkeiten, indem wir auf Black-Box-Integrationstests beziehungsweise End-to-End-Tests setzen und unsere Qualitätssicherung auf fachliche Ebene abstrahieren. Da die Oberfläche eine Web-Anwendung ist, können wir Selenium und JUnit für Oberflächentests nutzen, um in einzelnen Tests fachliche Belange abzubilden. Unser Workflow ist dadurch statt „Baubarkeit zu Migration zu Test", „Test zu Baubarkeit zu Migration" *(siehe Abbildung 1)*.

Bis zur Herstellung der Baubarkeit kopieren wir uns von der Entwicklungs- und QA-Stage des Kunden Stück für Stück die nötigen Artefakte und Konfigurationen auf unsere eigenen Server, um für die Integrationstests eine Referenz-Instanz aufzubauen. Gegen diese schreiben wir unsere Tests. Dadurch wird das Test-Team unabhängig von der Migration. Ebenso Stück für Stück stellen wir die Baubarkeit einzelner Subprojekte wieder her und migrieren sie. Auf einer Migrations-Instanz mit aktualisierten Servern und Infrastruktur können wir dann die migrierten Artefakte deployen und die schon geschriebenen Tests stattdessen gegen diese laufen lassen.

Dadurch haben wir sozusagen ein riesiges Golden-Master-Testsystem *[1]* aufgebaut *(siehe Abbildung 2)*, mit dem wir die Qualität der gesamten Migration sichern können. Unser finales Vorgehen im Überblick sieht folgendermaßen aus:

1. Das Test-Team baut ein Test-Framework auf und läuft auf der Referenz-Instanz voraus
2. Das Build-Team baut die Infrastruktur auf und stellt die Baubarkeit wieder her
3. Das Migrations-Team migriert die Software
4. Die Ausführung der Tests auf der Migrations-Instanz und Vergleich mit der Referenz-Instanz

Letztendlich stellt die Migrations-Instanz die Basis für das neue Deployment beim Kunden dar. Dabei schneiden wir die Software fachlich und migrieren und liefern die Software in Inkrementen aus.

Um diese komplexen und vielfältigen Anforderungen umzusetzen und das richtige Framework zu schaffen, in dem wir arbeiten können, brauchten wir ein mächtiges und flexibles Build-Management-Tool. Es muss aber auch die nötige Integrität und Skalierbarkeit für so ein riesiges Projekt haben. Maven ist etabliert und skalierbar, aber bietet uns nicht die nötige Flexibilität. Da die Software verschiedenste Technologien, Sprachen in unterschiedlichen Versionen und komplexe (und fragwürdige) Deployment- und Bauverhalten enthält, hätten wir für die nötigen Erweiterungen wahrscheinlich dutzende Plugins schreiben müssen. Eine Integration von Inline-Ant in den Maven-Skripten oder Ant allein kam für uns wegen der nötigen Integrität und Skalierbarkeit nicht in Frage. Daher fiel unsere

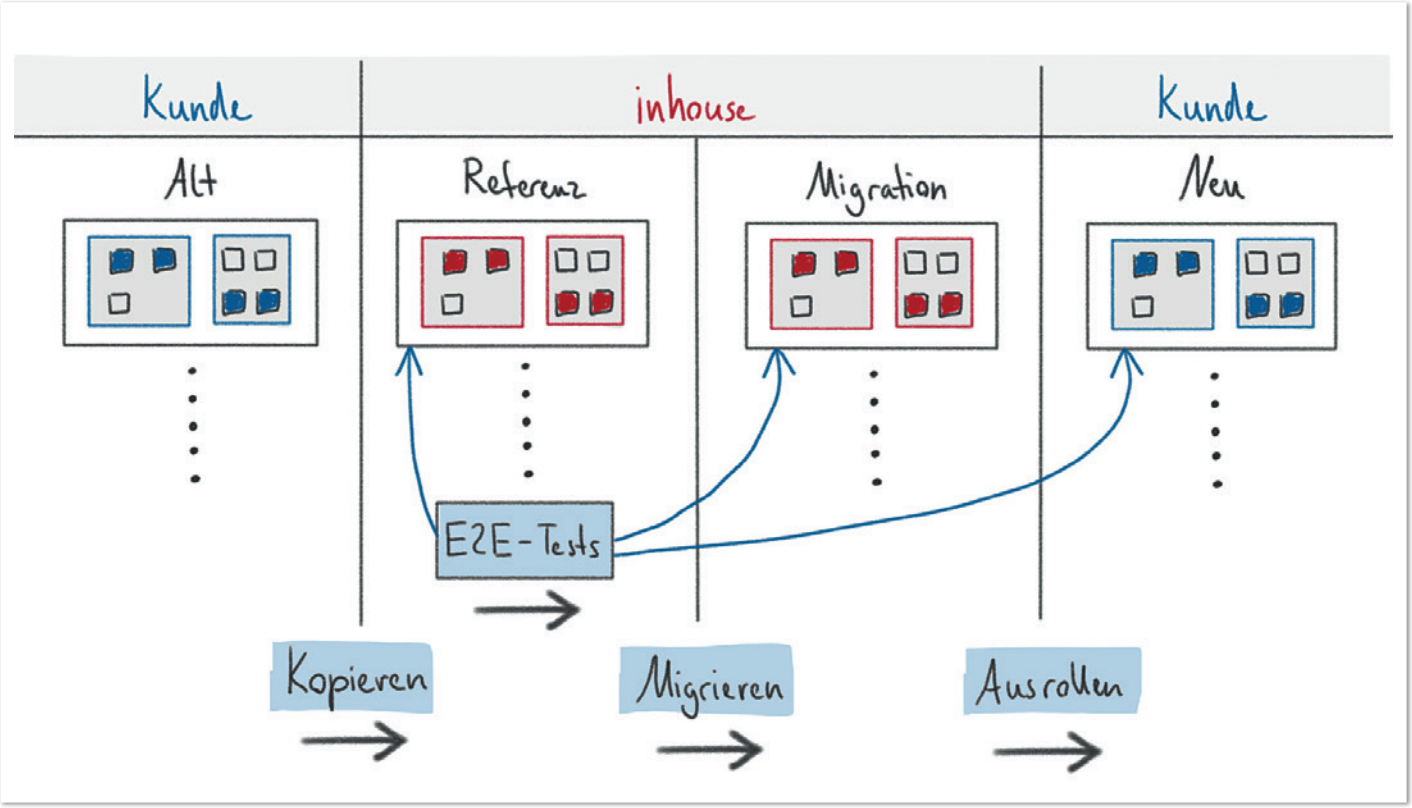

Abbildung 2: Unser Golden-Master-Vorgehensmodell (Quelle: Dmitrij Drandarov)

```
$ ./gradlew build -Pinfrastr=kunde -Pstage=qa -Pmigrated=false

buildscript {
  ext {
    infrastr = project.hasProperty("infrastr") ? infrastr:"inhouse"
    stage = project.hasProperty("stage") ? stage:"local"

    migrated = project.hasProperty("migrated") ? migrated:"false"
  }
  apply from: "config/${infrastr}-infrastructure.gradle"
}
```

Listing 1: Inklusion einer bestimmten Infrastruktur (build.gradle im Projekt-Root)

```
buildscript {
  ext {
    String propertiesPath = "config/$infrastructure-$stage/
                $infrastructure-$stage-stage.properties"

    // Load, Log and override with local properties
    infrConfig = new Properties()
    infrConfig.load new FileInputStream("$propertiesPath")
    ...
    // Repositories...
    ...
    // Weitere Konfigurationen...
}
```

Listing 2: Beispiel einer >infrastructure<.gradle Datei

Wahl auf Gradle. Es ermöglicht das Schreiben von Build-Skripten in Groovy bzw. Kotlin und ist damit sehr einfach erweiterbar. Gleichzeitig hat es eine klar definierte API und man kann ein für unsere Software angepasstes, stabiles und skalierbares Framework schreiben.

Abbildung der Stageabhängigkeit

Eine zentrale Herausforderung war es, die Stage-Abhängigkeit, die sich durch viele Teile der Software zog, abzubilden. Dafür haben wir die nötige Konfiguration der Infrastruktur ausgelagert und diese konfigurierbar gemacht *(siehe Listing 1)*.

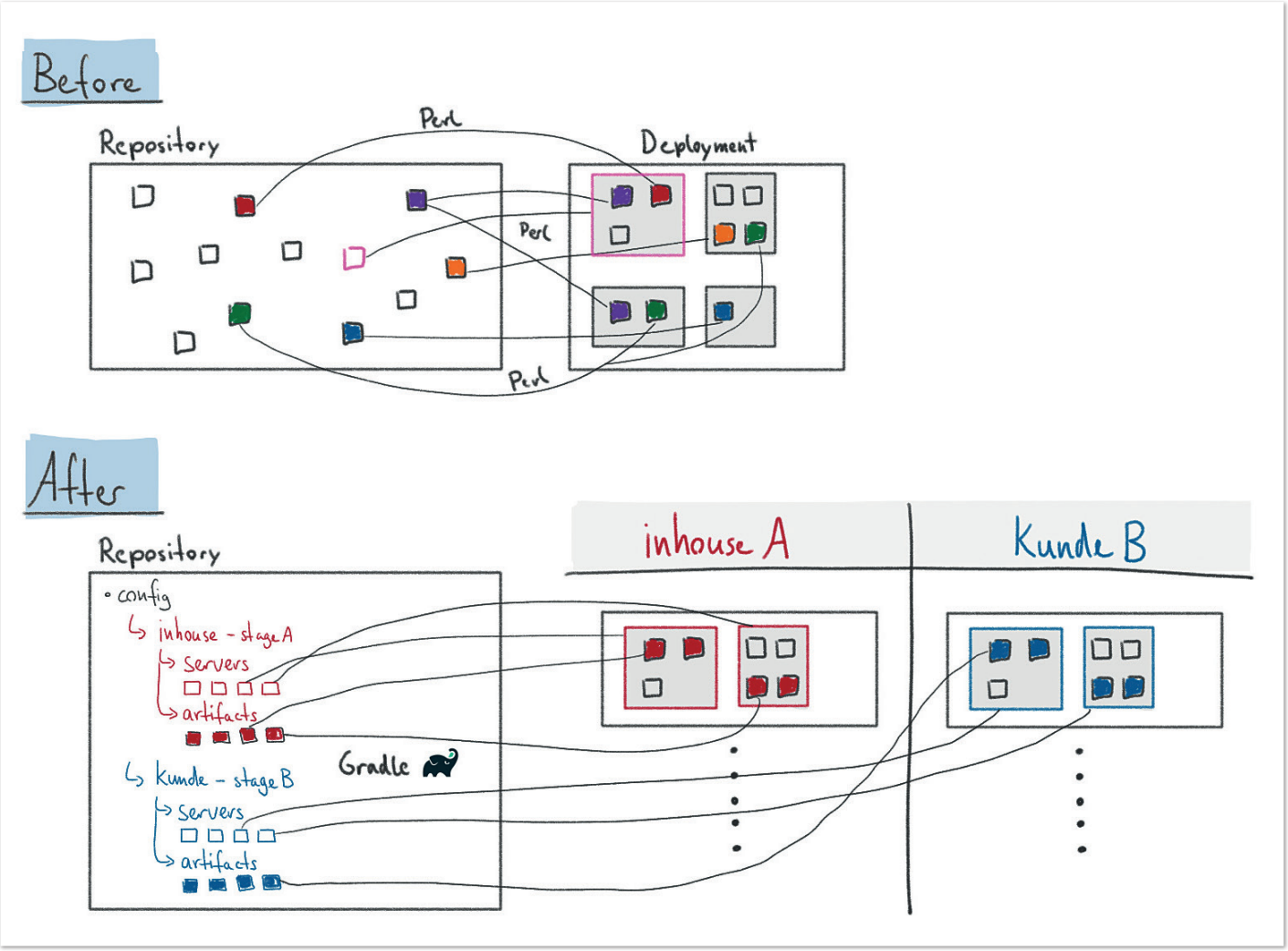

Abbildung 3: Unser Konzept zur Abbildung der Stage-Abhängigkeit (Quelle: Dmitrij Drandarov)

Wir definieren hier zuerst zwei Properties, die diese Informationen für uns abbilden. Die Property „infrastructure" bestimmt unter anderem welche Maven-Repositories genutzt werden sollen. Beim Kunden haben wir beispielsweise keinen freien Internetzugriff, sondern ein intern gehostetes Maven Repository. Die zweite Property ist „stage". Sie entscheidet, welche Properties-Datei geladen wird. Darin befinden sich beispielweise Datenbank-Konfigurationen, Pfade zu Applikations-Servern und URLs der zu testenden Web-Applikationen. Diese unterscheiden sich natürlich per Stage. Außerdem haben wir eine „migrated"-Property, die entscheidet, ob gegen die Migrations- oder Referenz-Infrastruktur gebaut und getestet wird *(siehe Listing 2)*.

Durch das Laden der "infrastructureConfig" in einem `ext{ }`-Block ist die Variable in allen Subprojekten zur einfachen Verwendung verfügbar. Aus ihr lassen sich alle nötigen, für die Stage relevanten Properties lesen. Beispielsweise kann sie auch an unsere JUnit-Integrationstests als System-Properties übergeben werden *(siehe Listing 3)*.

Da wir auf verschiedenen Infrastrukturen testen, bauen und deployen – was nicht gerade trivial ist – wollen wir die meisten Belange in Gradle abbilden. Damit machen wir uns unabhängiger von den verschiedenen Infrastrukturen, auf denen wir mit dem Projekt arbeiten wollen. Damit haben wir alle nötigen Informationen und Logik schon da, wo wir sie brauchen. Es gibt zudem einen „config"-Ordner im Repository, der für jede Stage einen „<infrastructure>-<stage>"-Ordner mit Stage-abhängigen Artefakt-Inhalten und Server-Konfigurationen beinhaltet. Ein Ausschnitt aus der Implementierung befindet sich weiter unten im Artikel. Mit diesem Modell *(siehe Abbildung 3)* wollen wir die Stage-Abhängigkeit der Artefakte und Server umsetzen.

```
database-name = DB2-LOKAL
database-host = server-220.kunde.de
database-port = 50000

application-deploy-host = http://server-112.kunde.de
application-url = http://server-112.kunde.de:1080/app
apache-document-root = /var/www/project/apache/htdocs
```

Listing 3: Beispiel einer >infrastructure<->stage<.properties Datei

Einfache Erweiterung der Build-Pipeline

In Gradle ist es, ebenso wie in Maven, möglich, Plugins zu schreiben. Diese müssen einfach das Plugin-Interface implementieren und im Root-Ordner in „buildSrc/src/main/" als Java- oder Groovy-Klassen abgelegt werden. In der „buildSrc/build.gradle"-Datei werden sie zudem registriert und sind damit in allen Build-Skripten verfügbar. Um möglichst wenig Komplexität in unseren Build-Prozess zu bringen, nutzen wir die Plugins primär zur Sicherung der Integrität und Skalierbarkeit sowie zur Reduktion von Redundanzen.

Ein sehr einfacher Use-Case für Plugins war die Erweiterung des Lifecycles von Gradle. In Gradle gibt es beispielsweise einen „build"-Task, der unter anderem vom „test"-Task abhängig ist, der wiederum von „compileTest" abhängt, etc. Wir haben das genutzt, um unsere Pipeline nach oben hin zu erweitern: Es gibt einen „integrationTest"-Task, der von einem „deploy"-Task abhängig ist, der wiederum vom „build" abhängt. Hierbei kann man beliebige zusätzliche Logik in das Plugin einbauen. So sieht unser triviales „IntegrationTestPlugin" aus *(siehe Listing 4)*.

```
class IntegrationTestPlugin implements Plugin<Project> {
  void apply(Project project) {
    project.getPluginManager().apply(JavaPlugin.class)
    project.getPluginManager().apply(DeployPlugin.class)

    project.task("integrationTest", type: Test)

    // Mögliche zusätzliche Erweiterungen...
  }
}
```

Listing 4: Einfache Erweiterung des Lifecycles

Umsetzung der stageabhängigen Artefakte

Ein weiteres Plugin ist z.B. unser „ConfigurationPlugin", welches die Stageabhängigkeit der Artefakte umsetzt. Es sieht (in gekürzter Form) aus, wie in *Listing 5* abgebildet.

Hier wird zuerst der Pfad der entsprechenden Konfiguration bestimmt. Dann wird ein Task „copyConfig" erstellt, von dem „build" abhängt. Der Task kopiert die stageabhängigen Teile aus dem o.g. „config"-Ordner, z.B. einfach in einen bestimmten Teil des „build"-Ordners, der im entsprechenden Projekt konfiguriert werden kann. In dem stageabhängigen Projekt muss man als Nutzer dann nur noch das Plugin anwenden und den Pfad setzen *(Listing 6)*.

Mobile Ausführung manueller Tests

Ein weiteres Problem bestand darin, dass Teile der Software nicht zuverlässig und deterministisch testbar sind. Daher haben wir

```
void apply(Project project) {
  // Path to stage-dependent config files (/config/infrastructure-stage/artifacts/$project.name/)
  def artifactConfigPath =
  "/config/$project.infrastr-$project.stage/artifacts/$project.name"

  project.afterEvaluate {
    Task copyConfig = project.task("copyConfig", type: Copy) {
      from project.fileTree(artifactConfigPath)
      // Path to config files target directory (/build/...)
      into artifactConfig.buildPath
    }

    project.tasks.build.dependsOn copyConfig
  }
}
```

Listing 5: Verkürztes ConfigurationPlugin

```
apply plugin: ConfigurationPlugin
artifactConfig.buildPath = "$buildDir/resources/main"
```

Listing 6: Anwendung des ConfigurationPlugins

```
java -jar junit-plaform-console.jar -cp unsereFatTestJar.jar
```

Listing 7: Ausführung von Standalone-JUnit-Tests

```
task testJar(type: Jar) {
  from {
    configurations.testCompile.collect { it.isDirectory() ?
      it:zipTree(it) }
  }
  from {
    sourceSets.test.output
  }
}
```

Listing 8: Bauen einer Fat-Test-Jar

ebenfalls manuelle Tests geschrieben, die zum Testen, z.B. an die Anwender der Applikation, gehen können. Die JUnit-Tests öffnen eine kleine, selbstgeschriebene JavaFX-Oberfläche, die dem Anwender Anweisungen zum manuellen Test gibt und einen Approve- und Fail-Button besitzt, um das Testergebnis zu entscheiden. Die JUnit-Test-Ergebnisse können uns dann zugeschickt werden. Das bedeutet, dass wir unsere Tests mobil machen müssen. Die Tests sind mit JUnit geschrieben, daher können wir den Standalone-ConsoleLauncher von JUnit 5 *[2]* nutzen und mit ihm die Tests an einem beliebigen Ort auszuführen, ohne eine ganze Entwicklungsumgebung mitzuliefern *(Listing 7)*.

Dafür müssen wir lediglich eine Jar bauen, die alle Tests und die entsprechenden Abhängigkeiten beinhaltet. Gradle macht das sehr einfach, indem wir in einer Test-Jar alle „testCompile"-Abhängigkeiten und aus dem Test-SourceSets alle Klassen inkludieren *(siehe Listing 8)*.

Weitere Use Cases

Das ist alles in allem nur ein kleiner Ausschnitt der Probleme, die wir mit Gradle gelöst haben. Nachstehend findet sich eine kurze Auflistung weiterer Probleme, die wir mit Gradle gelöst haben:

- Dynamische Anpassung von Stage-Properties, damit diese über Docker parallelisiert getestet werden können
- Stage-abhängige Server-Konfigurationen
- Lokal liegende proprietäre Artefakte als Dependencies
- Proprietäre ZIP-Dependencies aus Maven Repositories entpacken
- Schnelle Dockerisierung, zum Beispiel von Spring-Boot-Applikationen
- Konzept zum Überschreiben von Properties in der „local"-Stage durch lokale, nicht versionierte Properties im User-Ordner
- Datenbankversionierung über Flyway/Liquibase
- Sichere Exklusion von Mocks auf bestimmten Infrastrukturen
- AsciiArt unserer Projektpflanze zur Aufrechterhaltung der Motivation in einem Legacy-Projekt

Unser Fazit nach einem Jahr

Nach einem Jahr können wir sagen, dass vieles von dem, was wir mit Gradle umgesetzt haben, nicht mit Maven oder Ant möglich gewesen wäre. Da wir aber alle bis zum Start des Projekts mit Gradle wenig bis keine Erfahrungen hatten, haben wir regelmäßig bestimmte Mechanismen und Eigenheiten dazugelernt. Das hat dazu geführt, dass sich unsere Build-Skripte iterativ weiterentwickelt haben und wir oft Konzepte überdenken mussten. Außerdem hat sich gezeigt, dass wenn man viele Aspekte unseres Vorgehens überhaupt durch Gradle ermöglicht und sie damit davon abhängig macht, Hürden aufkommen können. Eine Änderung z.B. durch das Test-Team kann schwierig sein, wenn die entsprechenden Kollegen aus dem Build-Team nicht verfügbar sind und das Wissen nicht genug im Team verteilt ist. Auf diese Punkte sollte man frühzeitig Wert legen und die Kollegen schulen.

Referenzen

[1] Johannes Seitz, 2017, *https://www.maibornwolff.de/blog/techniken-der-it-sanierung-der-golden-master-test*

[2] JUnit 5 User Guide, 2019, *https://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher*

Dmitrij Drandarov
dmitrij.drandarov@outlook.com

Dmitrij Drandarov arbeitet als IT-Consultant bei der msg DAVID in Braunschweig und macht parallel seinen Master an der TU Braunschweig. Sowohl im Berufsalltag als auch privat befasst er sich mit verschiedensten Technologien und versucht regelmäßig Neues dazuzulernen. Bei weiteren Fragen steht er gerne zur Verfügung.

Ostereier und jede Menge Wissen bei den Stuttgarter Testtagen 2019

Oliver Böhm, Java User Group Stuttgart

Wahre Programmierer machen keine Fehler, testen ist nur was für Schwächlinge und die Ostereier bringt der Osterhase – dies sind die Mythen, mit denen man sich als Java-Entwickler herumschlagen muss. Grund genug, um der Sache auf den Grund zu gehen und uns auf die Suche nach den Ostereiern (siehe Abbildung 1) [1] zu machen.

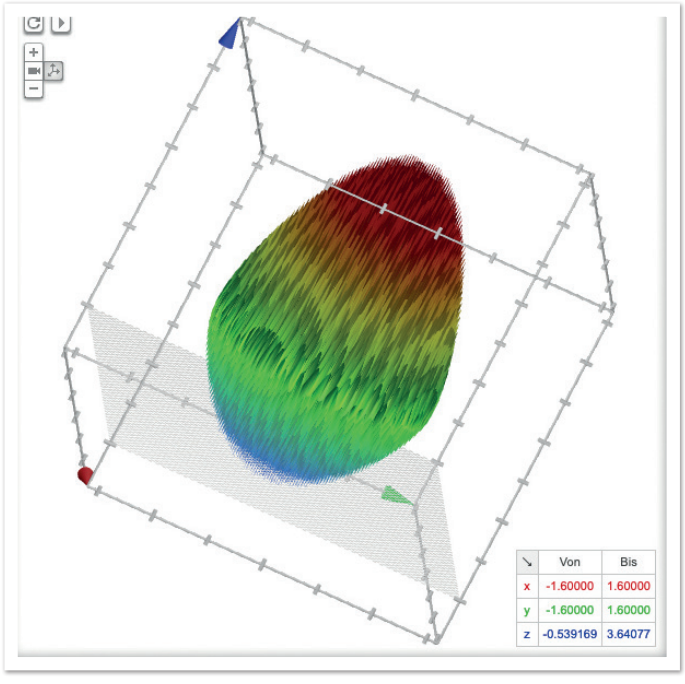

Abbildung 1 (Quelle: Google)

In der Woche vor Ostern fanden bereits zum sechsten Mal die „Stuttgarter Testtage" *[2]* der Java User Group Stuttgart statt. Dieses Mal waren wir wieder im Waldheim Möhringen zu Gast, in dem wir bereits in den Jahren 2013 und 2015 waren. Im Gegensatz zu 2015 hatte das Waldheim einige Schwachstellen beseitigt und uns mit einem lichtstarken Beamer sowie verbessertem WLAN versorgt. Letzteres war zwar stabil, aber dennoch recht langsam (DSL-Standard). Ein Beispiel dafür, dass Deutschland bezüglich schnellen Internets international nur Platz 30 belegt *[3]*. Im Zeichen von Docker-Images und Maven-Downloads ist das leider nicht mehr

Die Teilnehmer hören gespannt einem Vortrag zu. (Quelle: Oliver Böhm)

Die interaktiven Workshops regten zum Mitmachen und Mitdenken an. (Quelle: Oliver Böhm)

Kriterium	sehr gut (1)	gut (2)	befr. (3)	4	5	Schnitt
Umfang	6	6		1		1,69
Stoffvermittlung	6	4	3			1,77
Lernklima	9	4				1,31
Übungen	3	8	2			1,92
Räumlichkeiten	7	6				1,46
Gesamteindruck	8	5				1,38

Tabelle 1: Teilnehmerbewertung der Stuttgarter Testtage 2019

zeitgemäß und lässt die obligatorische Maven-Gedenkminute beim Start eines Builds eine gefühlte Ewigkeit dauern. Trotz Standard-DSL und Akustik-Schwierigkeiten kamen die Räumlichkeiten mit einer Gesamtnote von 1,5 recht gut weg – sicherlich auch ein Grund dafür, dass das Lernklima mit 1,3 überdurchschnittlich gut bewertet wurde *(siehe Tabelle 1)*.

Einen großen Anteil am positiven Gesamtbild hatten vor allem die Referenten, die interessante Themen, viel Enthusiasmus sowie USB-Sticks gegen die schwierigen WLAN-Verhältnisse im Gepäck hatten. Die Test-Themen selbst waren breit gefächert: Von Microservices bis Frontend, mit oder ohne Testautomatisierung, von Security bis Application Monitoring war für jeden etwas dabei. Während am Vormittag die Referenten das Thema vorstellten, war der Nachmittag für Workshops reserviert. Dort konnten die Teilnehmer sich eins von zwei Themen aussuchen und sich den Aufgaben stellen, die die Referenten mitgebracht hatten, diskutieren oder sich gegenseitig austauschen. Für weitere Gelegenheiten zum Austausch sorgte am Abend ein Barbecue, bei dem man sich mit anderen vernetzen und interessante Dinge erfahren konnte. Die Alterspanne lag zwischen 27 und 58 Jahren, die weiteste Anreise war aus Düsseldorf, gefolgt von München.

Das Konzept der vergangenen Testtage hat sich auch dieses Mal bewährt und trotz mancher Herausforderung waren sowohl Teilnehmer als auch Referenten hoch zufrieden. So wird es auch im Jahr 2021 wieder die Stuttgarter Testtage geben – beim nächsten Mal vielleicht zusätzlich mit einem Netzwerk-Sponsor für schnelles Internet (Bewerbungen dafür gerne an *ob@jugs.org*). Alle, die dabei sein wollen, merken sich am besten schon mal den 29. und 30. April 2021 vor.

Quellen

[1] Die Grafik erhält man, wenn man „1.2+(sqrt(1-(sqrt(x^2+y^2))^2) + 1 - x^2-y^2) * (sin (10000 * (x*3+y/5+7))+1/4) from -1.6 to 1.6" in Google eingibt.

[2] *https://www.jugs.org/tt2019/*

[3] *https://www.techbook.de/easylife/web/internet-speedtest-ranking-international-weltweit*

Oliver Böhm
ob@jugs.org

Oliver Böhm studierte Informatik an der Universität Stuttgart. Nach C++-Entwicklung im Unix-Bereich beschäftigt er sich seit 1999 mit Java-Entwicklung unter Linux und Aspekt-orientierter SW-Entwicklung. Er ist unter anderem Autor der Bücher „JavaSoftware Engineering unter Linux" und „Aspekt-Orientierte Programmierung mit AspectJ 5". Neben seiner hauptberuflichen Tätigkeit als Java-Statiker und -Archäologe beim Optica Abrechnungszentrum gibt er AOSD-Vorlesungen und ist Board-Mitglied der JUG Stuttgart.

Die Rückkehr zu einer positiven Fehlerkultur

Sabine Wojcieszak, getNext IT

In der agilen Welt und besonders im DevOps-Ansatz wird immer wieder darüber gesprochen, dass wir aus Fehlern lernen müssen. Gleichzeitig besagt das Modell der VUCA-Welt (Volatility, Uncertainty Complexity, Ambiguity), dass unsere (Arbeits-)Welt nicht nur immer komplexer wird, sondern sich auch immer rasanter verändert und Fehler daher passieren werden. Doch wie passt die bislang praktizierte Idee der Fehlervermeidung in diese neue Welt? Und wie schaffen wir eine Kultur, die positiv mit Fehlern umgeht? Wir alle – Organisationen wie Individuen – müssen lernen, positiv aus Fehlern zu lernen!

Abbildung 1: Jeder Fehler beinhaltet die Chance für das Team, an ihm zu wachsen! (© Kyle Glenn - Unsplash)

Grundsätzlich sind Fehler nichts Besonderes und gehören, seitdem wir auf der Welt sind, zu unserem Leben. In unseren ersten Lebensjahren pflastern Fehlversuche und Fehler unseren Weg; sie werden von den Eltern, Großeltern und anderen in der Entwicklung bereits fortgeschrittenen Personen sogar auch noch „gefeiert", belohnt oder führen zumindest zum Spenden von Trost. Experimente – so können die Versuche, das Laufen zu lernen oder einen Turm aufzubauen, genannt werden – werden in der Regel von den Älteren begeistert unterstützt. Die Motivation wächst, früher oder später kommt der Erfolg und mit dem Erfolg wächst das Selbstbewusstsein. Das wachsende Selbstbewusstsein wiederum ist gut für das Meistern neuer Herausforderungen. „Aus Fehlern lernen" ist hier positiv hinterlegt und ein Erfolgsrezept.

Dann kommt die Schul- und Studienzeit. Abhängig von Schulform, Schule und Lehrkraft wird schnell klar, dass Fehler mit schlechten Noten bestraft werden, die wiederum auch zuhause für schlechte Stimmung sorgen können. „Fehler sind doof!", ist das Ergebnis und das führt häufig dazu, dass Schüler nur noch lernen, um Fehler und damit „Bestrafung" in Form von schlechten Noten zu vermeiden, nicht aber, um den Lernstoff zu verstehen. So berichteten Studierende, sie müssten für eine Klausur 278 Folien auswendig lernen, da der Professor nur Antworten in exakt seinem Wortlaut akzeptiere – leider kein Einzelfall. Auf die Frage, was die Studierenden davon im Kopf behalten würden, antworten sie: „Dass wir 278 Folien auswendig lernen mussten." Welche Verschwendung von Zeit und Motivation ein derartiges Vorgehen beinhaltet, lässt sich nur erahnen. In einem so organisierten Lernumfeld – in Schulen wie in Unternehmen – wird die Bedeutung von „aus Fehlern lernen" dahingehend verändert, dass Aufgaben, die potenziell Fehler verursachen könnten, gemieden werden und nur bereits bewährte Lösungsansätze – ob passend oder nicht – genutzt werden. Nachdenken, Kreativsein, Hinterfragen und Experimentieren verlieren an Bedeutung.

Auf ins Arbeitsleben!

In vielen Organisationen wird dieses Vorgehen in der Unternehmenskultur gehegt und gepflegt. Fehlervermeiden ist ein Prinzip, das dabei hilft, in der Deckung zu bleiben, keine Aufmerksamkeit zu erregen und sich dem Blame Game und den sich daraus ergebenden Konsequenzen zu entziehen. Doch was bedeutet das eigentlich für Unternehmen, wenn sie Mitarbeitern neue, komplizierte und komplexe Aufgaben stellen wollen? Neu, kompliziert und komplex bedeutet, dass die Wahrscheinlichkeit, einen Fehler zu machen, zunimmt. Wenn jedoch gelernt wurde, dass Fehler per se schlecht sind, wird es immer schwieriger, jemanden zu finden, der sich an diese komplizierten und komplexen Aufgaben herantraut und sogar noch innovative Lösungen dafür findet. Etwas wagen zu können, etwas auszuprobieren oder Mut zu zeigen für neue Ansätze, sind in solchen Kulturen und Lernumfeldern nicht die gefragten Skills. Gleichzeitig hat das „Nicht-über-Fehler-Sprechen" einen weiteren, für das Unternehmen unter Umständen sehr teuren Aspekt. Entstandene Fehler werden versteckt; eine offene und rechtzeitige Kommunikation darüber erfolgt in der Regel nicht. Wenn diese Fehler dann „entdeckt" worden sind, haben sich die Auswirkungen über einen Prozess hinweg schon summiert oder das Beheben eines Fehlers ist einfach aufgrund der Abhängigkeiten gar nicht oder nur noch mit einem enormen Aufwand möglich. Das hat direkte Auswirkungen auf die Qualität und das Unternehmen. Gleichzeitig aber führt es zu

Abbildung 2: Entwickeln Teams und Führungskraft ein wahres „Wir"-Gefühl, so hilft das beim Aufbau einer positiven Fehlerkultur! (© Kaboompics .com - Pexels)

Frustration bei Mitarbeitern und Führungskräften. Am Ende stehen dann auf der einen Seite mehr Kontrolle und auf der anderen Seite das völlige Loslassen jeglicher Verantwortung für die Qualität der eigenen Arbeit, da „ja jemand anderes das schon kontrollieren wird". Der Frust auf Seiten der Führungskräfte und die damit verbundene Überzeugung, dass die Mitarbeiter „immer Fehler machen", kann zu einer negativen Haltung führen, die den wissenschaftlich untersuchten *Golem-Effekt* nach sich zieht. Dieser Effekt besagt, dass, wenn eine Führungskraft davon überzeugt ist, dass Mitarbeiter „immer Fehler machen" oder „nicht in der Lage sind, Aufgaben zu erledigen", diese Führungskraft ihr Verhalten entsprechend anpasst. Eine „Command & Control"-Umgebung könnte das Ergebnis sein. Dieses geänderte Verhalten führt dazu, dass die Mitarbeiter – wohl aus Verunsicherung und mangelndem Selbstvertrauen – wiederum genauso agieren: Sie machen Fehler und performen schlecht! Die sich *selbsterfüllende Prophezeiung* nimmt ihren Lauf.

Ein weiterer negativer Effekt einer derartigen Fehlerkultur ist der entgehende Mehrwert, wenn Prozesse und Abläufe dadurch verbessert werden könnten, dass über Fehler nicht nur gesprochen wird, sondern auch gemeinsam geprüft wird, was die wirklichen Gründe und Auslöser dafür waren, und nicht nur oberflächliche Kosmetik betrieben wird. Gleichzeitig wird jedoch auch das interne, tägliche Innovations- und Veränderungspotenzial nicht voll ausgeschöpft. Spätere, in großen Aktionen betriebene Change-Prozesse wirken dagegen oft sperrig und anstrengend und sind meist nicht sehr beliebt.

Und dann kommen die Agilisten, Digitalisierer, VUCA-Prediger und New-Work-Jünger…

…und sprechen von „Welcome failure", „Fail fast", „Fail early" und „Fail often"! Plötzlich gibt es „Fuck up Nights", in denen Menschen offen über ihre häufig sehr beträchtlichen Fehlschläge berichten; und sie werden auch noch dafür gefeiert! Nun soll genau das auch im Unternehmenskontext stattfinden: Mitarbeiter sollen über ihre Fehler sprechen, damit andere und das Unternehmen daraus lernen können. Fehler sollen „gefeiert" werden – und es geht noch weiter; nicht nur darüber sprechen sollen die Teams, sie sollen auch noch gemeinsam in „Blameless Postmortems" ausführlich analysieren, welche einzelnen Schritte und Entscheidungen zu dem Fehler geführt haben! Transparenz in den Arbeitsschritten lassen klar erkennen, wann was passiert ist. Passenderweise heißt diese Funktion im Git auch noch „blame". Wer sich mit DevOps beschäftigt, weiß, dass im Idealfall unverzüglich eine Warnmeldung und ein Stopp für das ganze Team erscheinen (Andon Cord), wenn durch das Einchecken von Code die Continuous-Delivery-Pipeline crasht – sofort sind alle informiert!

Doch egal wie positiv die Fehlerkultur im Unternehmen auch sein mag, es fühlt sich einfach nicht gut an, einen Fehler verursacht zu haben. Darüber zu sprechen, kann zu einer echten Tortur werden, die jeder einzelne ertragen können muss. Nicht selten versuchen Teammitglieder mit Aussagen wie „mein Fehler" oder „ich nehme es auf meine Kappe", diesen ungewohnten Prozess zu verkürzen. Nichtsdestotrotz ist es für Unternehmen heutzutage zwingend notwendig, darüber nachzudenken, wie sie mit Fehlern umgehen, wenn sie in der VUCA-Welt überleben wollen. Das Akronym VUCA steht für Volatility (Volatilität, Unbeständigkeit), Uncertainty (Unsicherheit), Complexity (Komplexität) und Ambiguity (Mehrdeutigkeit). Selbstverständlich haben sich der Markt und das Kundenverhalten und damit die Anforderungen schon immer verändert: Neu ist dabei definitiv die Geschwindigkeit, mit der Veränderungen auftreten, und damit auch die Häufigkeit. Hinzu kommen die kaum absehbaren komplexen Zusammenhänge, die die Einschätzungen massiv erschweren. Dies wird durch die Mehrdeutigkeit, zum Beispiel bei der Auswertung von Daten oder Ergebnissen, noch zusätzlich verstärkt. Das aktuelle Zeitalter der Digitalisierung ist genau von diesen Aspekten geprägt, und eines wird klar: Fehler werden einen festen und wichtigen Bestandteil der Arbeit darstellen! Durch stete Veränderungen, zunehmende Komplexität und eine große Mehrdeutigkeit besteht bei neuen Lösungen – und das ist es, was regelmäßig entwickelt werden muss – eine große Unsicherheit. Ansätze und Vorgehensweisen, die gestern noch einwandfrei funktioniert haben, müssen nicht zwangsläufig zu einer neuen Aufgabe passen – auch dann nicht, wenn diese auf den ersten Blick als identisch erscheint.

Wenn also Fehler ein fester Bestandteil der Arbeit sein werden, kann dann jeder so viele Fehler machen, wie er will? Nein! Gerade weil die VUCA-Welt jeden Tag neues Fehlerpotenzial mit sich bringt, ist es ein erklärtes Ziel einer positiven Fehlerkultur, bekannte Fehler zu vermeiden – allerdings nicht, um einer „Bestrafung" oder anderen negativen Konsequenzen zu entgehen. Um jedoch Fehler für alle vermeiden zu können, muss herausgefunden werden, was tatsächlich zu dem Fehler geführt hat. So kann die Struktur von bestimmten Prozessen oder auch sich wiederholenden Tätigkeiten zu Fehlern führen. Gemeinsam im Team mit dem Verursacher als dem Experten für diesen Fehler (siehe *„Denn zum Lernen sind sie da – wie Fehler Systeme widerstandsfähiger machen"* Teil 1 *[1]* & 2 *[2]*) wird nach dem Root Cause gesucht, um danach eine Entscheidung über eine werthaltige Lösung treffen zu können. Allerdings zeigt sich gerade in der heutigen Zeit, dass Fehler oft das Ergebnis einer Verkettung mehrerer kleinerer Dysfunktionalitäten sind und es den EINEN Root Cause gar nicht mehr gibt.

Umso wichtiger erscheint es zu ergründen, wodurch der Fehler tatsächlich entstanden ist. Dieses Vorgehen wird als „Culture of Causality" bezeichnet. Häufig kommt dann der Aspekt der Automatisie-

rung von Prozessen mit integrierten Quality Gates zum Einsatz, die nicht nur dazu beitragen, Fehler zu vermeiden, sondern auch dazu, ein definiertes Level an Qualität zu standardisieren.

Solch ein offener Umgang mit Fehlern ist aber nur möglich, wenn die „Psychologische Sicherheit" im Team gegeben ist. Ein hohes Maß an Vertrauen untereinander ist eine unbedingte Voraussetzung. Damit ist eine positive Fehlerkultur nicht nur die Aufgabe der Unternehmen, sondern eines jeden einzelnen Individuums innerhalb einer Organisation.

Lernen, aus Fehlern zu lernen!

Schließen Sie bitte einmal die Augen und beantworten Sie ehrlich für sich folgende Frage: Wie reagieren Sie, wenn ein Teammitglied davon berichtet, dass ein Fehler aufgetreten ist? Rollen Sie genervt mit den Augen? Stöhnen Sie leise, aber wahrnehmbar auf? Denken Sie „Der schon wieder!" oder „War ja klar!"? oder neigen Sie tatsächlich zu deutlicheren Unmutsäußerungen wie: „Was hast du denn da schon wieder gemacht?" Ein wichtiger Schritt ist es zu verstehen, dass Teamwork „Wir" bedeutet: Nur *zusammen* erreichen wir *gemeinsame Ziele* – in guten wie in schlechten Zeiten. Nur gemeinsam können wir uns entwickeln, unter anderem auch daraus, was wir aus den Fehlern gelernt und wie wir sie behoben haben.

Zu diesem „Wir" sollte auch die Führungskraft gehören. „Mein Team und ich" oder „der Boss und wir" sind Reliquien aus dem letzten Jahrtausend! Diese „Wir"-Einstellung funktioniert übrigens unabhängig von den hierarchischen Strukturen eines Unternehmens – sie findet nämlich in den Köpfen statt. Wird das „Wir" entwickelt und gelebt, sind einerseits Schuldzuweisungen schwieriger, gleichzeitig wird es jedoch wesentlich einfacher, offen über Fehler zu reden. Die gegenseitige Wertschätzung wird in solchen Teams großgeschrieben. Ruft zum Beispiel ein Kunde bei der Führungskraft an, um sich über ein Problem oder einen Fehler zu beschweren, wird bei einer praktizierten „Wir"-Haltung die Führungskraft eher zu einer Aussage wie „Wir werden das im Team besprechen und eine Lösung finden!" tendieren. Aussagen wie „Ich werde klären, wer dafür verantwortlich ist", „Ich werde das Team zur Rechenschaft ziehen" oder Ähnliches haben ausgedient. In einem weiteren Schritt kann dann diese Transparenz – auch in Bezug auf Fehler – auf den Kunden ausgeweitet werden, sodass auch dieses Verhältnis von mehr Vertrauen geprägt ist und ein gegenseitiger Mehrwert geschaffen werden kann. Gleichzeitig ist es allerdings auch hilfreich, wenn auch die Mitarbeiter die Idee vom „Wir" inklusive der Führungskraft leben.

Dazu gehört auch die Akzeptanz, dass die unterschiedlichen Rollen ebenso bedeuten, verschiedene Aufgaben aus den unterschiedlichsten Blickwinkeln betrachten und erledigen zu müssen. Je mehr Offenheit, Vertrauen und Transparenz in Teams herrschen, desto besser kann jeder einzelne seine Rolle annehmen und erfüllen.

Wird das „Wir" gelebt, ist auch eine positive Einstellung gegenüber den Teamkollegen sehr wahrscheinlich. Tatsächlich hat auch die positive Einstellung eine direkte Auswirkung auf die Leistung. So wie der Golem-Effekt sich negativ auswirkt, erzielt der *Pygmalion-Effekt* genau das Gegenteil. Ist eine Führungskraft davon überzeugt, dass Mitarbeiter ihre Aufgaben erfüllen werden, wird sie sich entsprechend verhalten. Dieses Verhalten wiederum nimmt positiven Einfluss auf das Verhalten der Mitarbeiter, die dadurch besser performen. Weniger Fehler, bessere Ergebnisse und kreative Lösungen führen zu mehr Erfolg. Dieser gemeinsam erzielte Erfolg zusammen mit der gegenseitigen Wertschätzung erzeugt bei den Teammitglieder mehr Selbstvertrauen. Hier setzt sich der positive Kreislauf fort: Der *Galatea-Effekt* beschreibt, dass Mitarbeiter, deren Selbstvertrauen gestärkt ist, auch bessere Ergebnisse erzielen. Die aus gemeinsamen Erfolgen gezogene Motivation schafft im Team einen zusätzlichen Mehrwert. Zum einen entstehen weniger Fehler, zum anderen wird es jedoch auch leichter, über Fehler zu sprechen, wenn das Selbstbewusstsein stimmt, das Team Wertschätzung erfährt und Erfolge die Motivation erhöhen. Zusätzlich wirkt es sich positiv aus, Fehler als Bestandteil der Arbeit zu akzeptieren, weil dadurch Fehler nicht mehr auf das Podest des „außergewöhnlichen Events" gehoben werden und somit eher der Normalität angehören.

Mit einem gelebten „Wir", gegenseitiger Wertschätzung, einem offenen und modernen Leadership und einer positiven Haltung gegenüber Fehlern können Teams die Grundlage für eine positive Fehlerkultur legen – ganz ohne Tools. Mit dieser Änderung im Mindset können wir wieder dorthin zurückkehren, wo wir alle einmal gestartet sind: zu der Zeit, in der „aus Fehlern lernen" DAS Erfolgsrezept für uns war, um komplexe Aufgaben wie Gehen oder Sprechen zu erlernen! Auch für Unternehmen ist genau dieses Erfolgsrezept der Schlüssel, um die Herausforderung der VUCA-Welt zu meistern, sich kontinuierlich zu verbessern und weiterzuentwickeln!

Referenzen

[1] *https://www.entwicklung-komplexer-systeme.de/single-post/2018/02/21/denn-zum-Lernen-sind-sie-da---Wie-Fehler-die-Widerstandsf%C3%A4higkeit-von-Systemen-und-Prozessen-verbessern-k%C3%B6nnen-Teil-1*

[2] *https://www.entwicklung-komplexer-systeme.de/single-post/denn-zum-Lernen-sind-sie-da-Fehlerkultur-fuer-mehr-Sicherheit-Teil-2*

Sabine Wojcieszak
sabine@getnext-it.com

Als Enthusiastic Agile & DevOps Enabler bei getNext IT führt Sabine Wojcieszak Projekte für Unternehmen erfolgreich durch. Sie adressiert Thematiken wie Kommunikation, Führung, Meetings und Innovation und coacht sowohl Teams als auch Führungskräfte. Als Sprecherin ist sie auf internationalen Konferenzen unterwegs und schreibt Artikel für führende Fachmagazine und Blogs. Weiterhin lehrt und motiviert sie an der Fachhochschule internationale Master-Studierende in den Bereichen DevOps, Agiles PM und Open Source. Auf Twitter ist sie unter @SabineBendixen zu finden.

Jenkins, Jenkins: Don't Repeat Yourself (DRY)!

Lukas Pradel, Conciso GmbH

In modernen Microservice-Architekturen wird Software vollautomatisch über CI/CD-Pipelines ausgeliefert. Wie lassen sich diese Pipelines bei wachsender Komplexität und steigender Anzahl von Services beherrschen, zumal sie ja mitunter das Deployment in die Produktion steuern? Man tut gut daran, sich einen alten Bekannten unter den Prinzipien der Softwareentwicklung in Erinnerung zu rufen.

Microservices sind nach wie vor in aller Munde. Oftmals werden sie über eine eigene Continuous Integration Pipeline gebaut. Im Falle von Continuous Deployment werden sie auch automatisch ausgeliefert und deployt. Ein wichtiger Bestandteil von Microservice-Architekturen ist die Unabhängigkeit der Services. Auch wenn die Services in der Praxis immer in Kombination eingesetzt werden, soll jeder Microservice eine abgeschlossene Einheit sein, die vollkommen unabhängig von allen anderen gebaut, deployt und betrieben werden kann. Daher wird jeder Microservice mit einer eigenen, separaten Datenbankinstanz oder Message-Broker-Instanz versehen. So ist sichergestellt, dass bei einem Ausfall einer Datenbank- oder Message-Broker-Instanz nicht automatisch alle Systemkomponenten ausfallen. Aus demselben Grund verfügt jeder Microservice auch über eine eigene separate CI/CD-Pipeline. Die separate Pipeline ist auch deshalb wichtig, weil sich Pipelines von Service zu Service unterscheiden können, wenn die Servicelandschaft insgesamt eher heterogen ist.

Stages sind Quality Gates

CI/CD-Pipelines bestehen aus mehreren Schritten (sogenannten „stages"), die sequenziell oder parallel durchlaufen werden können *(siehe Abbildung 1)*. Dabei ist jeder Schritt ein „Quality Gate". Wenn beispielsweise ein Kompilieren des Codes nicht möglich ist oder ein Integrationstest scheitert, wird die Software nicht ausgeliefert. Ein Software-Artefakt, das alle Pipeline-Stufen durchlaufen hat, kann potenziell in Produktion ausgeliefert werden. Pipelines können sowohl grafisch-konfigurativ als auch deklarativ über Code definiert werden. So kommt zum Beispiel bei GitLab CI die Programmiersprache Ruby zum Einsatz, während Jenkins-Pipelines mit Groovy formuliert werden.

Komplexität vs. Duplizierung

Bei komplexen Softwaresystemen mit konsequentem Domänenschnitt kann die Anzahl der Microservices schnell zweistellig werden. Damit einhergehend wird oft auch die Anzahl der Pipeline-Schritte groß. Je nachdem, wie homogen die Microservices sind, schleichen sich dann schnell Redundanzen und Code-Duplizierungen ein, die man im Anwendungscode (vollkommen zu Recht) peinlichst zu vermeiden versucht. Im Großen und Ganzen funktionieren die Pipelines in allen Services ähnlich: Code auschecken, kompilieren, testen, ausliefern und deployen. Daher wird häufig der Code eines Microservice für einen neuen Microservice übernommen und – wo nötig – angepasst.

Deklarative Jenkins-Pipelines

Ein Beispiel für eine simple deklarative Jenkins-Pipeline zeigt das `Jenkinsfile` in *Listing 1*. Diese Pipeline ist denkbar einfach und besteht aus lediglich zwei Stages: in der ersten wird der Code aus dem Source Control Management System (in diesem Fall Git) ausgecheckt und in der zweiten Stage die Software mit Maven gebaut.

Stellen wir uns vor, dass diese Pipeline nicht in einem Microservice, sondern in zehn Microservices zum Einsatz kommt. In diesem Szenario bereitet es den Engineers genau wie bei Code-Duplizierungen im Anwendungscode Kopfzerbrechen, wenn nun beispielsweise eine querschneidende neue Anforderung eine Änderung am Build-Prozess erforderlich macht. Beispiele für solche Änderungen finden sich schnell und sehen erst einmal sehr unschuldig aus:

- Vor einem neuen Release des Microservice soll die Softwareversion über den Befehl `sh "mvn -e -DnewVersion=${version} versions:set"` gesetzt werden
- In manchen Services soll eine andere Maven-Settings-Datei verwendet werden
- In den Maven-Build-Schritt muss eine neue Umgebungsvariable aufgenommen werden

In all diesen Fällen bleibt einem nichts anderes übrig, als in allen Microservices das Jenkinsfile zu öffnen, nach der entsprechenden Stage und der jeweiligen Stelle zu suchen und dort die gewünschten Änderungen einzupflegen. Eine mühsame Arbeit, die man sich gerne ersparen würde. Die Ursache des Problems besteht darin, dass wir mit dem Code unserer CI/CD-Pipeline gegen ein bewährtes Prinzip der Softwareentwicklung verstoßen haben.

Das DRY-Prinzip

Vor ziemlich genau 20 Jahren haben Andy Hunt und Dave Thomas in ihrem legendären Buch „The Pragmatic Programmer" unter der Überschrift „The Evils of Duplication" das bekannte DRY-Prinzip („Don't repeat yourself" – „Wiederhole dich nicht") formuliert, das besagt, dass **jedes Wissensfragment genau eine eindeutige, unmissverständliche und maßgebliche Repräsentation in einem System haben muss** *[1]*. Die Autoren schlagen vor, das Prinzip in allen Bereichen der Softwareentwicklung anzuwenden: bei Datenbankschemata, Testplänen, Dokumentation und eben auch beim Build-System. Es gibt in der Softwareentwicklung kaum ein Prinzip, das so unumstritten und verbreitet ist wie das DRY-Prinzip. Aus gutem Grund, denn die Folgen bei Verstößen sind äußerst schmerzhaft und kostspielig.

Sobald Informationen redundant vorgehalten werden, werden Anpassungen aufwendig, da diese immer an mehreren Stellen vorgenommen werden müssen. Darüber hinaus wird das Softwaresystem

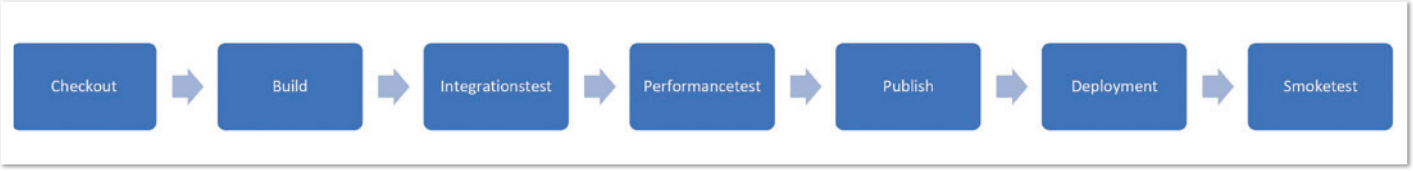

Abbildung 1: Eine einfache CI/CD-Pipeline (© Lukas Pradel)

```
#!/usr/bin/env groovy

pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        git branch: pipelineParams.branch, credentialsId: 'GitCredentials', url: pipelineParams.scmUrl
      }
    }

    stage('Build') {
      steps {
        withMaven(mavenSettingsFilePath: '/var/jenkins_home/settings.xml') {
          sh "mvn -e -ff clean install"
        }
      }
    }
  }
}
```

Listing 1: Ein Jenkinsfile einer einfachen Pipeline

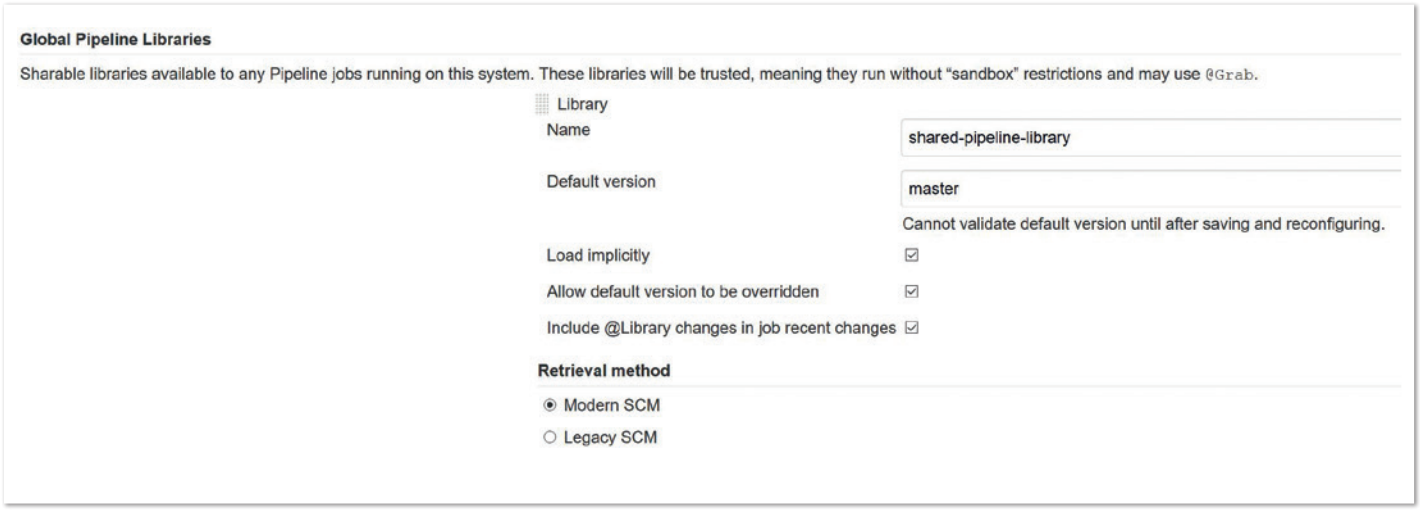

Abbildung 2: Die Jenkins-Konfiguration für eine Shared Pipeline Library (Quelle: Lukas Pradel)

```
#!/usr/bin/env groovy

@Library('shared-pipeline-library@2.1.5')

pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        git branch: pipelineParams.branch, credentialsId: 'GitCredentials', url: pipelineParams.scmUrl
      }
    }

    stage('Build') {
      steps {
        buildService(service: 'FooService', version: '1.2.7')
      }
    }
  }
}
```

Listing 2: Die vereinfachte, DRY-konforme Jenkins-Pipeline.

anfällig für Bugs, weil mit wachsender Anzahl von Redundanzen auch die Wahrscheinlichkeit steigt, dass Stellen übersehen oder fehlerhafte Anpassungen vorgenommen werden. Häufig finden sich bei Code-Duplikaten auch geringfügige Abwandlungen, sodass es auch in modernen Softwareentwicklungsumgebungen immer schwieriger wird, alle Stellen zu identifizieren.

Umgekehrt sind Softwaresysteme, bei denen das Prinzip erfolgreich angewendet wird, leicht zu warten, denn eine Änderung an einer einzelnen Komponente des Systems erfordert keine zusätzlichen Anpassungen an anderen Komponenten.

Shared Pipeline Groovy Libraries

Bezogen auf unsere Pipeline würde sich der geübte Engineer wünschen, für jede der einzelnen Stages eine parametrisierbare Funktion oder Methode extrahieren und diese dann einfach in jeder Pipeline mit den jeweils passenden Argumenten aufrufen zu können.

An genau dieser Stelle schafft das „Pipeline Shared Groovy Libraries Plug-in" für Jenkins Abhilfe *[2,3]*. Den Jenkins-Maintainern scheint der Wert des Plug-ins bewusst zu sein, da es mittlerweile fester Bestandteil des Pipeline-Plug-ins und somit in neueren Jenkins-Installationen standardmäßig installiert ist. Mit dem Plug-in ist es möglich, Pipeline-Code versioniert in eine Bibliothek auszulagern, die man dann in den Pipelines einzelner Microservices einbinden und verwenden kann.

Dazu muss man lediglich im SCM die Bibliothek ablegen. Dann kann man diese in der globalen Jenkins-Konfiguration unter Angabe der SCM-Koordinaten referenzieren. *Abbildung 2* zeigt die entsprechende Konfiguration in Jenkins. In den Pipeline-Skripten der einzelnen Microservices kann die Bibliothek nun sehr einfach verwendet werden. Unsere ursprüngliche Pipeline aus *Listing 1* wird deutlich vereinfacht, wie man *Listing 2* entnehmen kann.

Die Bibliothek wird über die Zeile `@Library('shared-pipeline-library@2.1.5')` in die Version 2.1.5 eingebunden. In unserer Beispielversion ist die Bibliothek in Git versioniert. Das Jenkins-Plug-in klont die Bibliothek mit dem Tag "2.1.5". So ist es uns auch möglich, „breaking changes" an der Bibliothek vorzunehmen, ohne die Pipelines der einzelnen Services alle sofort umstellen zu müssen. Diese können so lange eine beliebige Version der Bibliothek

```
import de.conciso.ServiceMavenSettings

def call(args) {

  assert args.service: "Service name must be provided."
  assert args.version: "Service version must be provided."

  def serviceMavenSettings = new ServiceMavenSettings()
  def mavenSettings = serviceMavenSettings.getMavenSettingsForService(args.service)

  withMaven(mavenSettingsFilePath: mavenSettings) {
    sh "mvn -e -DnewVersion=${args.version} versions:set"
    sh "mvn -e -ff clean install -Dbuild.environment=ci"
  }
}
```

Listing 3: Die extrahierte Build-Stage in der Datei buildService.groovy

verwenden, bis die Pipeline-Skripte entsprechend für die neue Version angepasst wurden – und sie können unabhängig weiterentwickelt werden.

Die gesamte Build-Stage ist nun im Aufruf der Funktion `buildService` verschwunden, wie man es aus der Java-Entwicklung gewohnt ist. Wir wollen uns nun natürlich noch ansehen, wie die eigentliche Bibliothek im Detail aussieht.

Shared Pipeline Library

Die Struktur der Bibliothek ist in *Abbildung 3* dargestellt. Im Ordner `src` liegen Groovy-Klassen, im Ordner `test` die entsprechenden Unit-Tests. Funktionen, die im Pipeline-Skript aufgerufen werden können, liegen unter exakt dem Namen der Funktion als Groovy-Datei im Ordner `vars`. Dementsprechend gibt es dort eine Datei `buildService.groovy`. In dieser findet sich nun der extrahierte und parametrisierte Code der Build-Stage, wie man *Listing 3* entnehmen kann.

Funktionen im Ordner `vars` beginnen immer mit `def call(args)` und können über das `args`-Feld auf ihre Aufrufargumente zugreifen. In diesem Fall haben wir im Vergleich zur ursprünglichen Pipeline in *Listing 1* somit den Namen und die Version des Service parametrisiert und das Setzen der Service-Version sowie eine Maven-Variable ergänzt. Das Ermitteln der korrekten Maven-Settings-Datei haben wir in eine separate Groovy-Klasse mit dem Namen `ServiceMavenSettings` ausgelagert, die über ein einfaches `import`-Statement verwendet wird.

In allen Microservices wird nunmehr lediglich die Funktion `buildService` aufgerufen. Wenn nun die Pipeline-Bibliothek weiterentwickelt wird, muss in den Pipelines der einzelnen Microservices im Idealfall nur noch die Versionsnummer angepasst werden. Alle übrigen Änderungen sind in der Bibliothek weggekapselt.

Objektorientierte Bibliotheken

Abgesehen von den Groovy-Skript-Funktionen kann man, wie bereits erwähnt, auch objektorientiert in der Bibliothek programmieren. Dazu muss man nur mit entsprechenden Groovy-Klassen im Ordner `src` arbeiten. Diese können dann wiederum in den Skripten im Verzeichnis `vars` über ein einfaches `import`-Statement verwendet werden, wie unser Beispiel zeigt. Bezogen auf unsere Pipeline könnte die Klasse `ServiceMavenSettings` dann so aussehen wie in *Listing 4*.

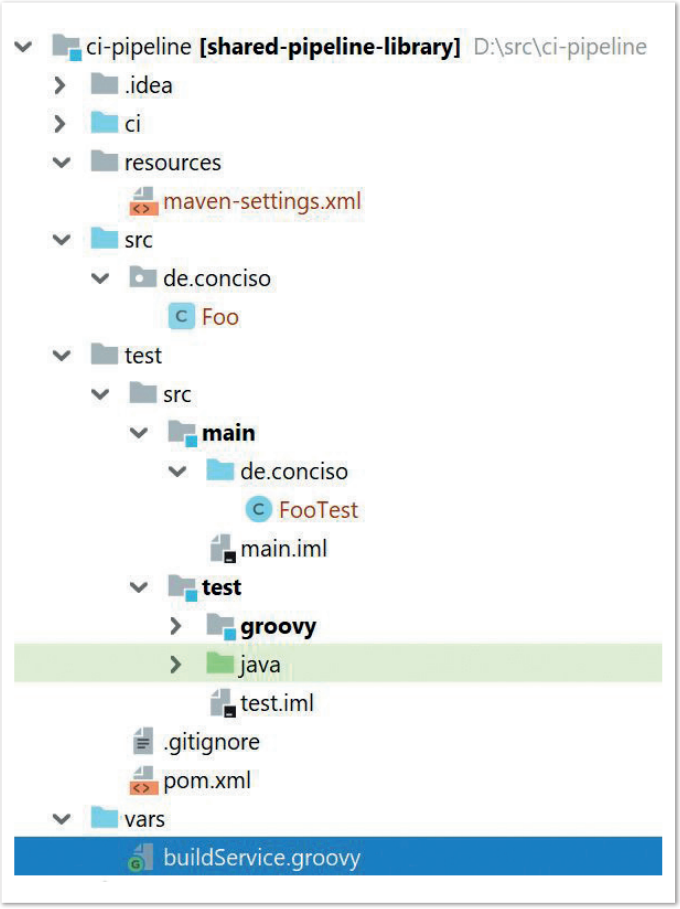

Abbildung 3: Die Struktur einer Shared Pipeline Library (Quelle: Lukas Pradel)

Wir haben hier natürlich ein absolutes Minimalbeispiel betrachtet. Bei realistischen Pipelines mit 20-30 Stages für 15 oder mehr Microservices besteht der erste Schritt in Richtung DRY-Konformität daher darin, für jede Stage, die in einer der Pipelines vorkommt, eine entsprechende Funktion in der Pipeline Library bereitzustellen und diese in den Pipelines zu referenzieren.

Das Auslagern von Pipeline-Schritten in die Bibliothek kann sogar so weit getrieben werden, dass ein komplettes Pipeline-Skript in der Bibliothek abgelegt werden kann (ebenfalls im Verzeichnis `vars`). Somit wandert dann ein komplettes parametrisiertes Jenkinsfile in eine entsprechend benannte Groovy-Datei. Das eigentliche Jenkinsfile sieht dann nur noch wie in *Listing 5* aus.

```
package de.conciso

class ServiceMavenSettings implements Serializable {

  ServiceMavenSettings() {
  }

  getMavenSettingsForService(String serviceName) {
    if (serviceName == 'FooService') {
      return '/var/jenkins_home/settings.xml'
    } else {
      return '/var/jenkins_home/other_settings.xml'
    }
  }
}
```

Listing 4: Eine Groovy-Klasse für Pipeline Libraries.

```
#!/usr/bin/env groovy

@Library('shared-pipeline-library@3.0.0')
servicePipeline(service: 'foo', ..)
```

Listing 5: Die extreme Variante: Die Pipeline verschwindet komplett in der Bibliothek

In diesem Beispiel liegt der gesamte Pipeline-Code in der Bibliothek in der Datei `vars/servicePipeline.groovy`. Diese Variante sollte man allerdings nur nach sorgfältiger Überlegung und mit Vorsicht einsetzen. Sie bietet sich nur an, wenn die Microservices vollkommen homogen sind (identische Programmiersprache, identische Frameworks, identische Prozesse, identische Organisationen etc.). Anderenfalls führt dieses Vorgehen dazu, dass die externalisierte Pipeline so sehr parametrisiert werden muss, dass sie selbst unwartbar wird.

Fazit

Wir beobachten in unseren Projekten immer wieder, dass Pipeline-Code stiefmütterlich behandelt wird, da er kein Teil des Anwendungscodes ist und sich gerade Engineers im Bereich Java-Backend oftmals nicht für Infrastruktur-Themen begeistern.

Diese Vernachlässigung ist jedoch überaus gefährlich: Je mehr Continuous Delivery und DevOps Einzug halten, desto wichtiger werden CI/CD-Pipelines im Softwareentwicklungsprozess. Bei gelebtem Continuous Delivery umfassen sie das Deployment in die Produktion und manuelle Schritte entfallen teilweise komplett – der gesamte Prozess ist vollkommen automatisiert.

Wir plädieren daher dafür, dem Pipeline-Code mit denselben Qualitätsansprüchen zu begegnen, die man auch an den Anwendungscode hat. Dabei ist das DRY-Prinzip nur eines der wichtigen Prinzipien der Softwareentwicklung. So kann auch Pipeline-Code automatisiert getestet und sogar testgetrieben entwickelt werden, auch wenn wir es in diesem Beitrag nur andeuten.

Quellen

[1] Andrew Hunt, David Thomas (1999): The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, USA.

[2] *https://wiki.jenkins.io/display/JENKINS/Pipeline+Shared+Groovy+Libraries+Plugin*

[3] *https://jenkins.io/doc/book/pipeline/shared-libraries/*

Lukas Pradel

lukas.pradel@conciso.de

Lukas Pradel arbeitet als Senior Consultant bei der Conciso GmbH. Die Zeit, die er mit der Beachtung des DRY-Prinzips einspart, nutzt Lukas zum Motorradfahren.

Wir lassen fliegen! Multiclient-App mit Rico, Java FX, Spring Boot & Polymer

Tanja Stuchels, Sprouts

Praxisreport: In diesem Artikel wird beschrieben, wie wir bei Sprouts ein System aufbauen, das für verschiedene Rollen unterschiedliche Frontends anbietet. So haben wir ein Web-Frontend, das auf Polymer und Rico aufgebaut ist, und einen vollwertigen JavaFX-Client, der ebenfalls auf dem Rico-Framework basiert. Außerdem werden Tools vorgestellt, die wunderbar unseren Solution-Stack vervollständigen, aber wenig bekannt sind.

Kurz über uns

Die Sprouts GmbH besteht aus zwei Geschäftsbereichen: Dem Software-Development und unserem operativen Geschäft – das sind Flugdienstleistungen im weitesten Sinne. Wir verstehen uns als Software-getriebenes Unternehmen. Unsere *„fly"-Produktfamilie* verwenden wir für unser Tagesgeschäft, lizenzieren diese jedoch ebenfalls an Kunden mit eigenen Werksflugbetrieben.

Überblick

Unsere Desktop-Anwendung setzt auf JavaFX, das Web-Frontend ist mit Googles Polymer implementiert. Zurzeit befindet sich eine iOS-App in der Entwicklung, die Ende dieses Jahres veröffentlicht werden soll *(siehe Abbildung 1)*. Im Backend setzen wir auf Spring Boot. Jedes Frontend hat seinen eigenen Server, der eine gewisse Anzahl von Clients bedient. Bei Bedarf besteht die Möglichkeit, weitere Server hochzufahren. Alle Server kommunizieren via Hazelcast miteinander. Das System basiert auf Java 8; wir haben zuletzt auf Amazon Corretto Java 8 migriert.

Tools und Abhängigkeiten

Folgende Tools und Abhängigkeiten werden für die *sprouts „fly"*-Produktreihe verwendet:

- Keycloak
- Apache Camel
- Hazelcast
- MongoDB
- CalendarFX

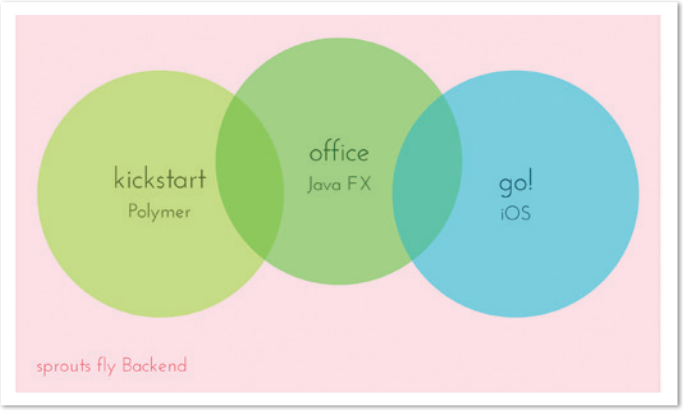

Abbildung 1: Unsere sprouts „fly"-Produktfamilie (Quelle: Tanja Stuchels)

- FlexGantFX
- Lombok
- JavaMail
- MigLayout
- JFXtras
- ControlsFX
- Guava
- mime-util (eu.medsea.mimeutil)
- GMapsFX
- Spring Boot
- Spring Data Mongo
- Migramongo
- C10N - Cosmopolitan
- JaVers
- Rico
- Polymer
- Docker

Einige dieser Tools sind nicht sonderlich stark verbreitet, deshalb wird im späteren Verlauf deren Funktionalität detaillierter vorgestellt.

Technik-Universum

Es folgt eine kurze Beschreibung der System-Komponenten:

1. „fly office" Update Channel Switcher

Dieses kleine Tool ermöglicht es dem Anwender, auszuwählen, mit welcher Konfiguration das System gestartet werden soll. Wenn zum Beispiel ein potenzieller Kunde eine Vorführung der Software bekommen soll, wird an dieser Stelle das Demo-System ausgewählt *(siehe Abbildung 3)*. Außer dieser Auswahlmöglichkeit gibt es dort den *„Future-Channel"* und den *„Production-Channel"* zur Auswahl. Der gewünschte Channel muss nicht bei jedem Start ausgewählt werden. Lediglich wenn ein anderer als der zuletzt ausgewählte Channel gewünscht wird, muss hier eine Auswahl getroffen werden.

2. „fly office" Launcher

Der Launcher ist eine kleine Anwendung, die den JavaFX-Client startet und gegebenenfalls aktualisiert. Mit dem JavaFX-Client („fly office") arbeiten unsere Flugdienstberater. Der Launcher selbst wird mit dem Java Packager Tool erzeugt und ist eine unter Windows und MacOS installierbare Anwendung. Diese lädt bei jedem Start die aktuellste Version des *„fly office"* herunter und startet diese.

3. „fly office"-Server

Dies ist das Backend für den JavaFX-Client *„fly office"*.

4. „fly kickstart"-Server

Dies ist das Backend für die mit Polymer umgesetzte *„kickstart"*-Web-Applikation.

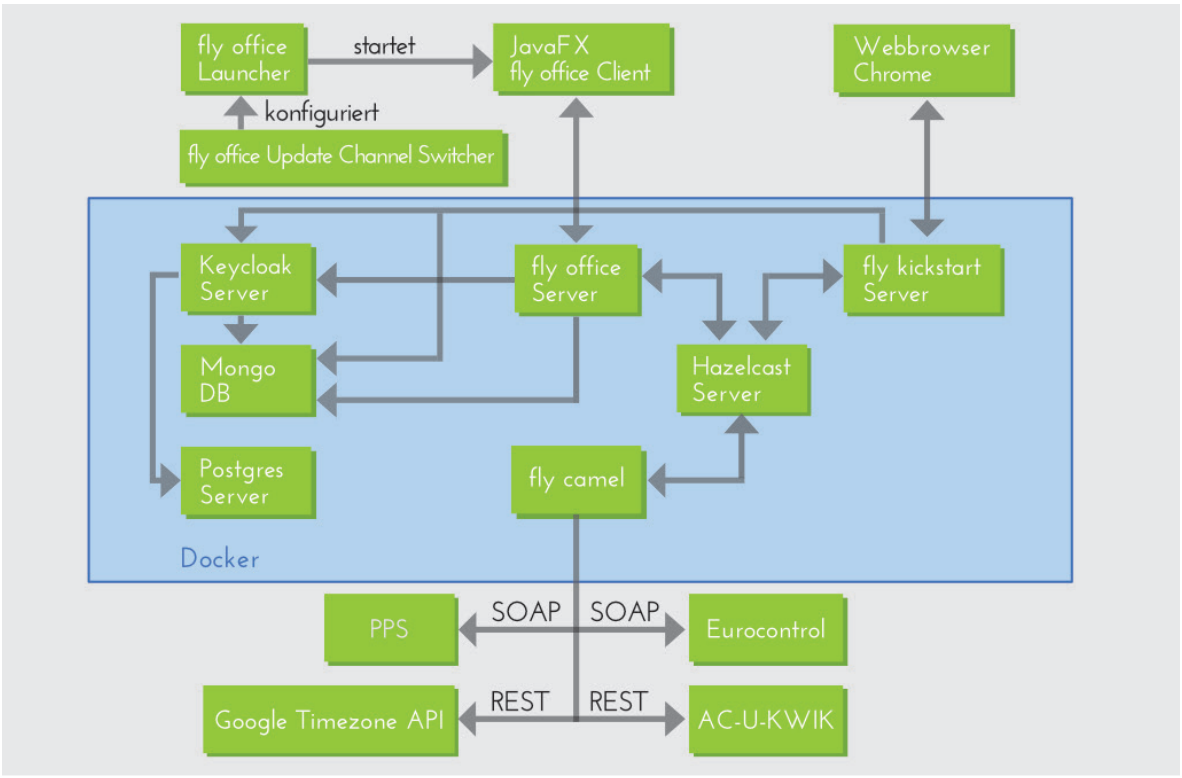

Abbildung 2: Solution-Stack unserer „fly"-Produktfamilie (Quelle: Tanja Stuchels)

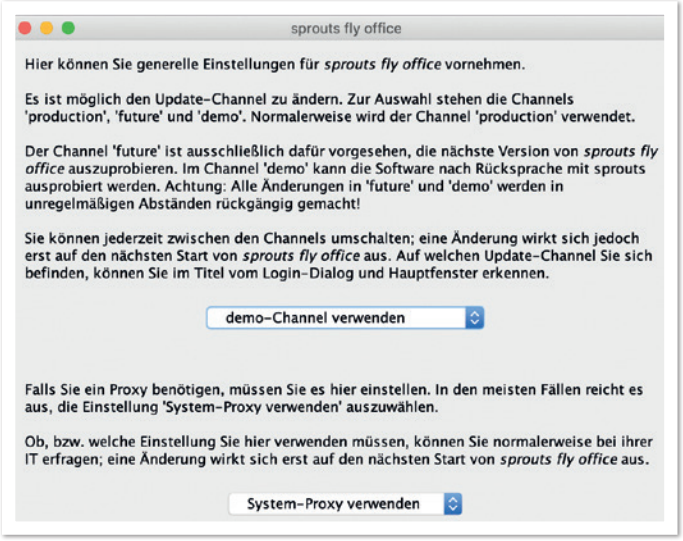

Abbildung 3 (Quelle: Tanja Stuchels)

5. MongoDB
MongoDB stellt die Datenbank für das *„fly"*-System bereit. Keycloak-Server, *„fly office"*-Server, *„fly kickstart"*-Server und *„fly camel"* legen hier ihre Daten ab.

6. Keycloak-Server
Wir verwenden Keycloak für die Authentifizierung unserer Anwender. Unsere eigene Keycloak-Erweiterung macht es möglich, dass auf die Benutzerdatenbank von *„fly office"* zugegriffen werden kann.

7. Hazelcast-Server
„fly office"-Server, *„fly kickstart"*-Server und *„fly camel"* verwenden Hazelcast-Server, um sich gegenseitig Nachrichten zu schicken. Hazelcast dient hier als Event-Bus. Auf diese Art informieren sich die einzelnen Komponenten des Systems gegenseitig über Änderungen an einzelnen Flügen oder anderen Datenbank-Entitäten.

8. Postgres-Server
Der Keycloak-Server verwendet den Postgres-Server, um darin Keycloak-interne Informationen zu speichern. Das fly-System verwendet Postgres jedoch nicht.

9. „fly camel"
Basiert auf Apache Camel und ist ein ESB (Enterprise Service Bus). Wer noch nie mit einem ESB gearbeitet hat, dem ist es sehr zu empfehlen, sich dieses Projekt einmal näher anzuschauen. Es können zum Beispiel Routen definiert werden, die ein Verzeichnis überwachen. Wird in diesem Verzeichnis eine Datei abgelegt, so wird der Inhalt eingelesen, in ein Event verpackt und über den ESB verschickt. Der nächste Teil der Route könnte dieses Event nehmen und transformieren. Diese Route könnte so konfiguriert werden, dass eine andere Komponente den Inhalt des Events annimmt und als Datensatz abspeichert. Wir haben verschiedene Apache-Camel-Routen an unseren Hazelcast-Server gebunden. Eine dieser Routen reagiert beispielsweise auf Änderungen an Flügen. Diese Route ruft die SOAP-Schnittstelle eines externen Programmes (PPS) auf und aktualisiert dort den Flug.

Rico vom Karakun Developer Hub

Rico ist ein Client-Server-Framework, das die unterschiedlichsten Funktionen und Schnittstellen anbietet, um Enterprise-Applikationen umzusetzen. Eine der Hauptfunktionen ist die Bereitstellung eines Datenmodells, das sich automatisch zwischen Client und Server aktualisiert.

In *Abbildung 4* starten wir mit der Eingabe eines Flughafen-Codes in das Suchfeld, das an das Rico-Modell gebunden ist. Die Eingabe wird in Echtzeit mit dem Modell im Server synchronisiert. Bei Eingabe von *„Enter"* wird Client-seitig eine Action aufgerufen. Das Rico-Framework ruft die dazugehörende Action auf der Server-Seite auf. Im Server wird die Eingabe weiterverarbeitet beziehungsweise die eigentliche Suche durchgeführt. Der Controller füllt nun das Modell im Server mit den Suchergebnissen. Die Änderungen am Modell werden vom Rico-Framework automatisch zum Client synchronisiert. An den entsprechenden Teil des Modells ist die Tabelle im UI gebunden, sie zeigt nun die Suchergebnisse an.

Im Folgenden wird mit Pseudo-Code beschrieben, wie mit dem Rico-Framework ein einfaches *„Hello World!"*-Programm erstellt wird.

Die Anwendung zeigt einen *„Hier klicken"*-Button. Sobald der Benutzer diesen anklickt, wird im Server die dazugehörige onClick-Action aufgerufen. Diese setzt im Modell einen Begrüßungstext. Rico synchronisiert den Begrüßungstext in den Client, in dem ein Label an das Property gebunden ist. Dieses zeigt die von Rico synchronisierte Begrüßung an. Alle gezeigten Annotationen stammen aus dem Rico-Framework.

In *Listing 1* sehen wir das Property, in dem die Begrüßungsnachricht übertragen wird. Zusätzlich sind aus Convenience-Gründen *„Getter"* und *„Setter"* definiert. In *Listing 2* wird gezeigt, wie die Business-Logik auf der Server-Seite aussieht. Beim Aufruf der onClick-Action wird im Modell die Begrüßungsnachricht gesetzt. In *Listing 3* werden einem BorderPane der entsprechende Button und das Label hinzugefügt. Die Button-Action ruft Server-seitig die onClick-Action auf. Die statische Methode FXBinder.bind() stellt das eigentliche Binding der Message-Property an das Label dar.

```
// Geteilt zwischen Client & Server

@RemotingBean
public class Model {
  private Property<String> message;
  public String getMessage() { return message.get(); }
  public void setMessage(String message) { message.set(message); }
  public Property<String> messageProperty() { return message; }
}
```

Listing 1: Beispiel-Code Shared | Rico-Framework

Polymer

Das Google-Polymer-Projekt ist ein Framework zur Entwicklung von Webapplikationen unter Verwendung von Webkomponenten. Zum Projekt gehören fertig einsetzbare Elemente, jedoch ist die eigene Entwicklung von Webkomponenten ebenfalls einfach umzusetzen.

Da sich die Polymer-Komponenten an das Material-Design-System halten, ist eine benutzerfreundliche Oberfläche schnell entwickelt *(siehe Abbildung 5)*. Das Rico-Framework stellt eine Polymer-Erweiterung zur Verfügung, die es ermöglicht, das Rico-Modell an Polymer-Komponenten zu binden.

Das folgende Listing zeigt eine *„Hello World!"*-Anwendung mit Polymer unter Verwendung des Rico-Frameworks. Wir müssen hier nur den Polymer-Teil programmieren; den Controller und das Modell können wir einfach aus *Listing 2* bzw. *3* weiterverwenden.

```
// Im Server

@RemotingController("HelloWorld")
public class HelloWorldController {

    @RemotingModel
    private Model model;

    @RemotingAction
        private void onClick() {
          model.setMessage("Hello World");
    }
}
```

Listing 2: Beispiel-Code Server | Rico-Framework

JaVers – Objektüberwachung und Änderungsaufzeichnung

Wir verwenden JaVers für das Auditing und zur Erstellung einer Änderungshistorie. JaVers ähnelt dem bekannteren Hibernate Envers, im Unterschied dazu kann es jedoch mit SQL und mit nicht-relationalen Datenbanken wie zum Beispiel MongoDB arbeiten. JaVers zeichnet die Änderungen an Datenbank-Entitäten auf und bietet Möglichkeiten zur Abfrage der Änderungen. Es zeichnet sich durch eine einfache Handhabung aus. Eine Besonderheit ist, dass es Spring-Data-Repositories mit automatischem Auditing erweitern kann.

In *Listing 5* sehen wir, wie JaVers manuell verwendet werden kann. Die daraus resultierende Änderungshistorie ist in *Listing 6* zu sehen.

```
// Im Java FX-Client

public class HelloWorldComponent extends BorderPane {

  private Button button = new Button();
  private Label label = new Label();

  public HelloWorldComponent(ControllerProxy controller, Model model) {
       button.setText("Hier klicken");
       button.setOnAction(event->controller.invoke("onClick"));
       FXBinder.bind(label.textProperty()).to(model.messageProperty());
       setTop(button);
       setCenter(label);
  }
}
```

Listing 3: Beispiel-Code Client | Rico-Framework

```
<dom-module id="hello-view">
    <template>
        <!-- styles weggelassen -->
        <remoting-controller id="controller" name="HelloWorld" model="{{model}}"/>
        <div>
            <button on-tap="helloWorldAction">Hier klicken</button>
            <h2>[[model.message]]</h2>
        </div>
    </template>
    <script>
        class HelloView extends Polymer.Element {
            static get is() {
                return 'hello-view';
            }

            helloWorldAction(event) {
                this.$.controller.invoke("onClick");
            }
        }
        window.customElements.define(HelloView.is, HelloView);
    </script>
</dom-module>
```

Listing 4: „Hello World!"-Anwendung mit Polymer | Rico-Framework

Migramongo

Migramongo ist ein Java-Tool, mit dessen Hilfe das Deployment von Datenbankänderungen realisiert werden kann. Der größte Unterschied zu anderen existierenden Tools, wie zum Beispiel dem bekannteren Liquibase, ist, dass die Migrations-Skripte in Java-Klassen oder Spring-Beans, statt wie oft beispielsweise mit JavaScript

```
User user = new User("Id-0", "Robert", 43);
javers.commit("user", robert);
user.setAge(44);
javers.commit("user", robert);
```

Listing 5: Manuelle Verwendung von JaVers

```
List<Change> changes = javers.findChanges(
                       QueryBuilder.byInstanceId("Id-0", User.class).build());

changes.forEach(change -> System.out.println("- " + change));

Ausgabe:

- ValueChange{ 'age' changed from '43' to '44' }
```

Listing 6: Abfrage der Änderungshistorie einer Entität

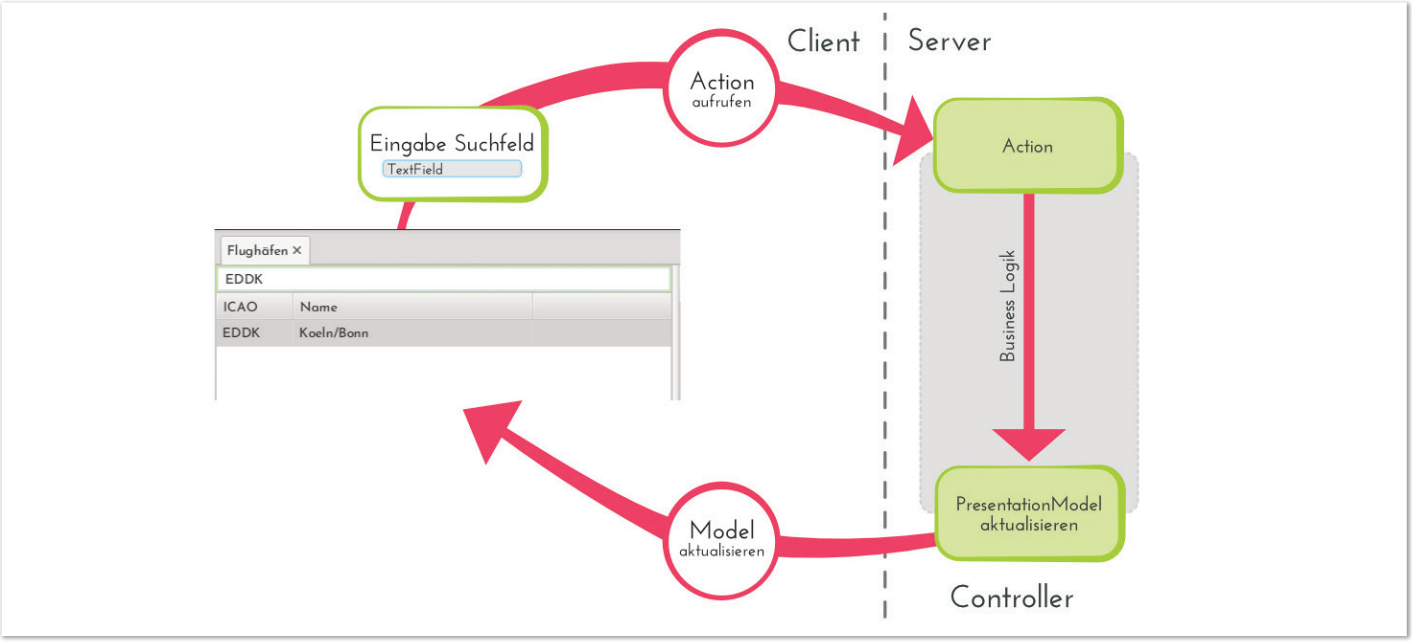

Abbildung 4: Funktionsweise Datenmodell des Rico-Frameworks (Quelle: Tanja Stuchels)

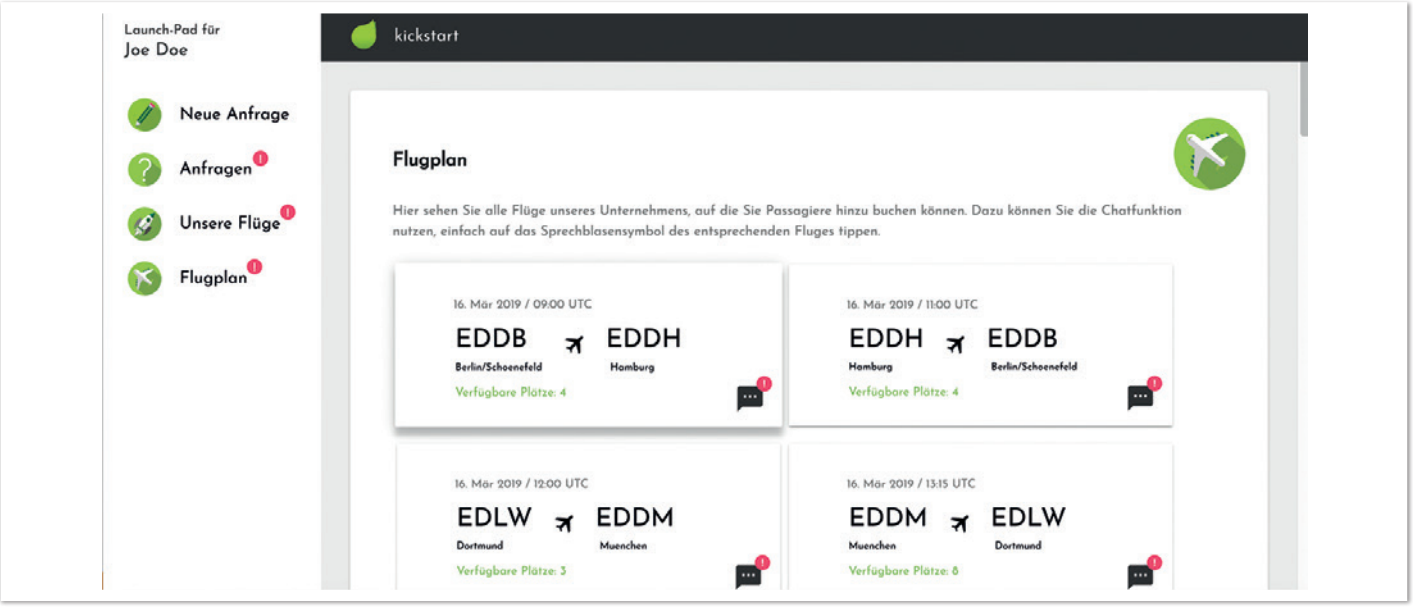

Abbildung 5: Bildschirmfoto „fly kickstart" (Quelle: Tanja Stuchels)

```
@PostConstruct
public void executeMigrationScripts() {
    MigraMongoStatus status = migraMongo.migrate();
    if (status.status == MigraMongoStatus.MigrationStatus.ERROR) {
        throw new IllegalStateException("Data migration failed: " + status.message);
    }
}
```

Listing 7: Aufruf von Migramongo

```
@Component
public class Migration_017_018 implements MongoMigrationScript {

    @Override
    public MigrationInfo getMigrationInfo() {
        return new MigrationInfo("017", "018") {
            @Override
            public String getModule() { return "Server"; }

            @Override
            public String getInfo() { return "Flights ergänzt um das Feld now"; }
        };
    }

    @Override
    public void migrate(MongoDatabase database) {
        mongoTemplate.findAll(DBObject.class, "flight").forEach(flight -> {
                flight.put("now", Instant.now());
                mongoTemplate.save(flight, "flight");
        });
    }
}
```

Listing 8: Beispielmigration mit Migramongo

```
Window msg = C10N.get(Window.class);
System.out.println(msg.title("James"));

Ausgabe:
Hallo, James!
```

Listing 9: Aufruf von C10N

```
public interface Window {
  @En("Hello, {0}!")
  @De("Hallo, {0}!")
  String title(String userName);

  @En("Show help")
  @De("Hilfe öffnen")
  String showHelp();
}
```

Listing 10: Textablage/Internationalisierung mit C10N

oder XML, geschrieben werden. Außerdem ist es, wie der Name vermuten lässt, mit der MongoDB kompatibel. Migramongo kann mit oder ohne Spring verwendet werden. Im Beispiel-Code verwenden wir es mit Spring.

Listing 7 zeigt, wie das injizierte Migramongo aufgerufen wird, um die gegebenenfalls notwendige Migration durchzuführen. Dabei wendet Migramongo nur diejenigen Änderungen an, die neu hinzugekommen sind.

Listing 8 zeigt, dass zu einem Migramongo-Changeset eine generelle MigrationInfo gehört und die Methode migrate() implementiert sein muss. Dort findet die eigentliche Migrationsarbeit statt.

C10N – Cosmopolitan

Für die Internationalisierung verwenden wir das Tool Cosmopolitan – kurz C10N. In *Listing 9* ist ein Aufruf von C10N zu sehen. Die Übersetzungen werden in Java-Sourcecode gekapselt *(siehe Listing 10)* und sind durch die Modularisierung sehr gut zu warten – Refactoring und Vererbung sind möglich.

Tanja Stuchels
tanja.stuchels@sprouts.aero

Tanja Stuchels arbeitet als Softwareentwicklerin bei der sprouts GmbH. Sie hat Erfahrungen in der Implementierung und Konzeption von Desktop- und Webanwendungen mit Java. Ihr weiterer Schwerpunkt ist die Gestaltung und Entwicklung grafischer Benutzeroberflächen.

Agile Führung ist ein Mindset – Doch wie kann man das lernen?

Dominic Lindner, agile-unternehmen.de

Agile Führung ist in aller Munde und soll laut moderner Management-Literatur die „Best Practice der Führung" sein. Während der Suche in Internetblogs, Magazinen oder Fachbüchern finden sich immer mehr Tipps und Hinweise, wie eine Führungskraft agil führen kann. Dabei fällt auf, dass diese Tipps oft von Coaches, Forschern und Experten zusammengestellt worden sind. Doch wie kann eine Führungskraft diese Art der Führung im Alltag umsetzen und erlernen?

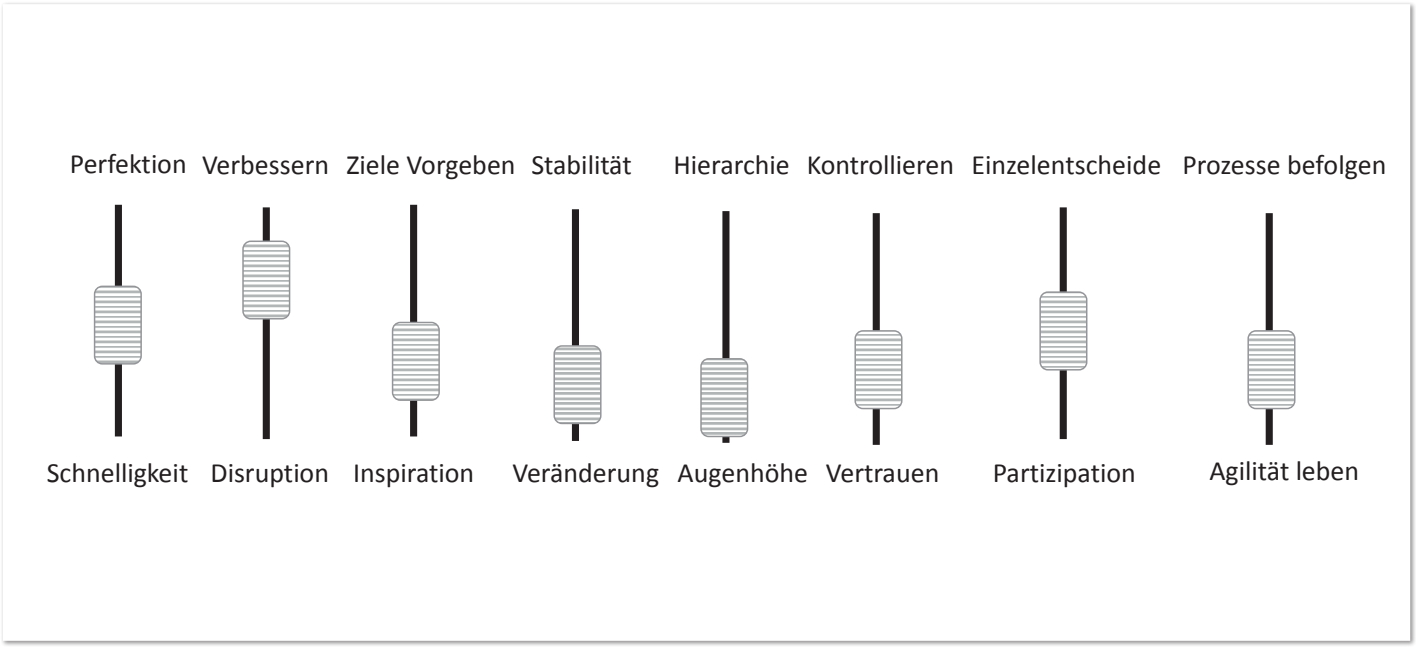

Abbildung 1: Führungsleitbild zur agilen Führung (© Dominic Lindner)

Agilität beschreibt im allgemeinen Sinne die Fähigkeit eines Individuums oder Objekts, flexibel und dynamisch auf seine Umwelt zu reagieren und sich schnell an Veränderungen anzupassen. Übertragen wir dies auf agile Führung, könnte es also bedeuten, dass eine Führungskraft flexibel auf die Anforderungen ihrer Mitarbeiter reagiert und diese damit in jeder Situation optimal unterstützt.

„In der agilen Welt ist das Mindset wichtiger als Tools und Methoden"

Dieses Zitat aus meiner aktuellen Forschung (Lindner 2019) steht sinnbildlich für die schwammige Definition des Begriffs der agilen Führung. Auch in der Literatur wird diese Art der Führung als eine „Verhaltensweise" beziehungsweise ein „Mindset" und eine „persönliche Haltung" beschrieben. Genau diese Tatsache macht den Begriff sehr schwammig und kaum greifbar. Was genau ist dieses „Mindset" und wie kann man ein solches erlernen? Auch sind Fragen, ob es sich um eine universelle Verhaltensweise handelt oder ob es für bestimmte Situationen gewisse Verhaltensweisen gibt, nicht beantwortet. Einig ist sich die Literatur nur bei der Tatsache, dass es darum geht, durch das „Mindset" oder Wertebild einer Führungskraft die Selbstorganisation von Mitarbeitern zu fördern. Hierdurch werden Aufgaben besser verteilt und Entscheidungen schneller getroffen, womit dynamische Projekte eine höhere Erfolgswahrscheinlichkeit haben.

Das Wertebild einer Führungskraft

In der Forschung wird auf die Persönlichkeit der Führungskraft abgezielt. Dabei werden vor allem Softskills und eigene Werte in den Vordergrund gerückt. Hilfe kann dabei ein persönliches Wertebild schaffen, das jede Führungskraft hat. Dabei gilt nicht jedes Wertebild als agil, weswegen die Literatur zahlreiche Tipps und Blaupausen bereitstellt, die sich oftmals stark unterscheiden. Beispielsweise werden folgende Ratschläge gegeben:

- Vertrauen Sie Mitarbeitern,
- inspirieren Sie Ihr Umfeld,
- helfen Sie den Mitarbeitern, sich zu entwickeln,
- geben Sie Rahmen vor und
- führen Sie nach Zielen.

Ein agiles Wertebild

Es gilt, gewisse Verhaltensweisen und Werte zu befolgen, die mit einer agilen Führung verbunden werden. Ich habe dazu in meiner aktuellen Forschung die am meisten genannten Werte genommen und sie in einer Onlinebefragung 66 Führungskräften präsentiert (Lindner 2019). Die Datenerhebung habe ich gemeinsam mit Tobias Greff vom AWS Institut durchgeführt. Mithilfe von Schiebereglern konnten die Befragten die Werte gewichten. Ich habe dabei immer gegensätzliche Werte genommen, etwa Perfektion und Geschwindigkeit. Das Ergebnis ist in *Abbildung 1* zu sehen.

An *Abbildung 1* ist erkennbar, dass es nicht darum geht, dass eine Führungskraft vollkommen agil oder nicht agil ist. Es ist wichtig, wie ein DJ an einem Mischpult, die Werte in die richtige Balance zu bringen. Sie können diese Abbildung als eine Art Blaupause nehmen und die Werte nun für sich selbst gewichten. Als Ergebnis haben Sie ein Zielbild, wie Ihre persönliche agile Führung aussehen kann.

Agile Führung: Learning by Doing

Nun bleibt die Frage, wie ein agiles Führungsleitbild erlernt werden kann. Bevor ich praktische Tipps gebe, habe ich die Befragung der 66 Führungskräfte ebenfalls genutzt und diese gefragt, wie ein agiles Wertebild erlernt werden kann und wie die Befragten vorgehen. Die Antworten sind in *Abbildung 2* zu sehen.

Die befragten Führungskräfte präferieren dabei Learning by Doing, Konferenzbesuche sowie Internetblogs/Magazine. Die Antworten erscheinen sinnvoll, da sicherlich viel Wissen in Schulungen und Internetblogs erlangt wird, aber es am Ende an der Führungskraft selbst liegt, dieses Wissen im Alltag zu erproben.

Learning by Doing bedeutet das Lernen durch unmittelbares Anwenden im Alltag. Dazu ist vor allem Selbstreflexion wichtig. Fragen Sie sich dabei: Was war gut? Was hat sofort funktioniert? Wo gab es

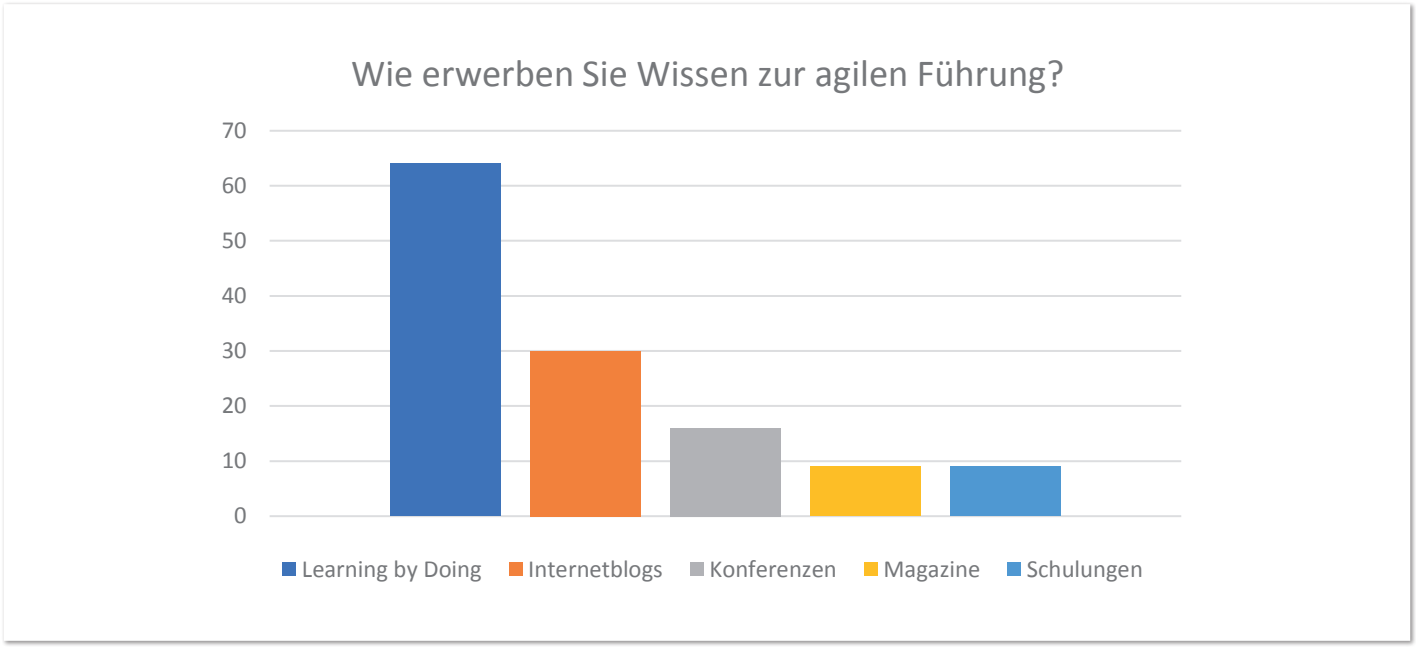

Abbildung 2: Wie erwerben Sie Wissen zur agilen Führung? (© Dominic Lindner)

Probleme? Was würde ich beim nächsten Mal anders machen? Mithilfe dieser Selbstreflexion können Sie schrittweise theoretisches Wissen in der Praxis anwenden und erproben.

Agile Führung im Arbeitsalltag erlernen

Nun stellt sich die Frage, ob eine Führungskraft so einfach Gewohnheiten und Charakter ändern kann. Natürlich sind Menschen in der Lage, gewisse Änderungen an der Persönlichkeit vorzunehmen, allerdings benötigt dies einiges an Durchhaltevermögen. Unsere Persönlichkeit ändert sich schon seit unserer Geburt ständig und wird durch unser Umfeld sowie unser eigenes Handeln und Tun beeinflusst. In der Praxis zeigt sich, dass schon kleine Eingriffe in den eigenen Charakter Menschen dabei unterstützen, glücklicher und erfolgreicher zu leben.

Der Mensch ist ein Gewohnheitstier und hat tägliche Routinen (z. B. der Weg ins Büro, Zähneputzen vor dem Frühstück etc.) und Verhaltensweisen (zum Beispiel emotionale Diskussion bei Problemen oder Rückzug bei starken Diskussionen). Nun sollen diese festgelegten Automatismen verändert werden. Dabei können Sie nach drei Schritten vorgehen: Routinen erkennen, eine Motivation zur Veränderung suchen und diese Routine ändern. Die Schritte werden in *Abbildung 3* dargestellt.

1. Routinen erkennen

Der erste Schritt ist, dass Sie eine gewisse Gewohnheit oder Routine erkennen und diese beschreiben können. Dies ist nicht einfach, da vieles unterbewusst passiert. Am besten ist dazu Feedback von guten Freunden oder Arbeitskollegen. Solche Gewohnheiten können sein:

- Ich höre meinen Mitarbeitern kaum zu.
- Ich kontrolliere meine Mitarbeiter zu stark.
- Ich gehe zu wenig Risiken ein und setze auf Stabilität.

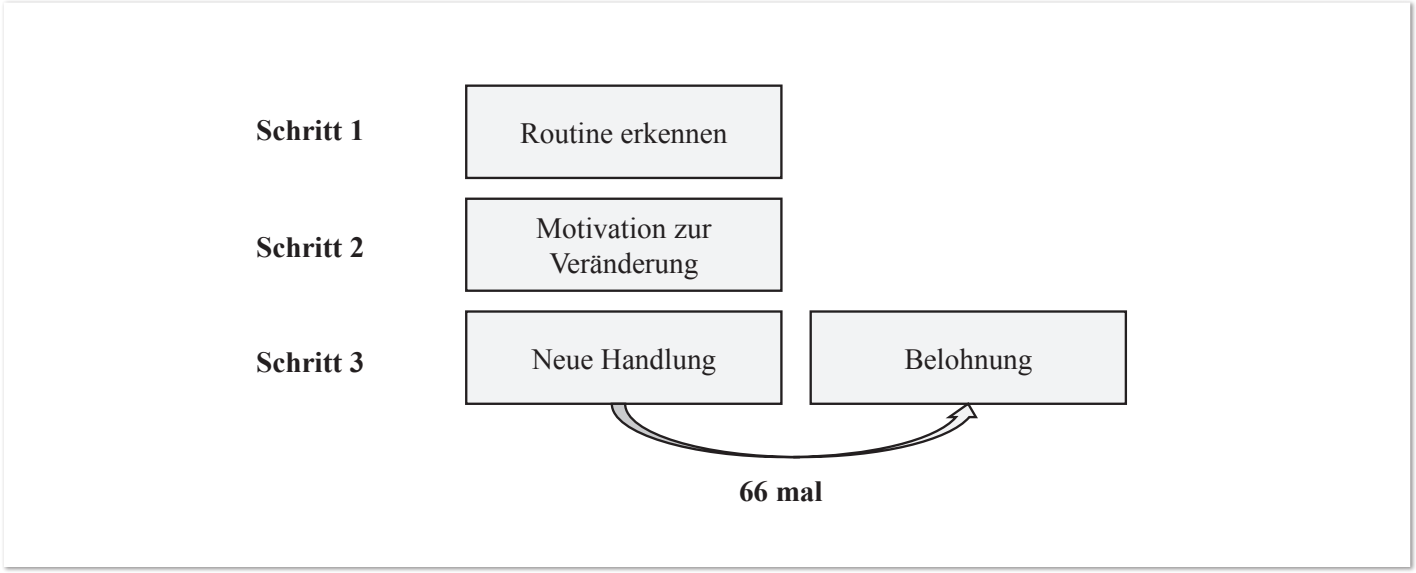

Abbildung 3: Zyklus der Veränderung (Quelle: Dominic Lindner)

2. Motivation zur Veränderung

Als Nächstes sollten Sie eine Motivation zur Veränderung finden. Nicht jede Gewohnheit ist automatisch wichtig genug, um verändert zu werden, und auch nicht jeder Mensch muss perfekt sein. Was ist also der Auslöser dafür, dass Sie eine gewisse Situation ändern wollen? Beispiele könnten sein:

- Ihre Verhaltensweise bringt Sie ständig in kritische Situationen.
- Ein Mitarbeiter hat wegen einer bestimmten Verhaltensweise von Ihnen gekündigt.
- Sie verärgern Kollegen durch bestimmte Routinen.
- Merken Sie sich immer: Gewohnheiten zu verändern, ist kein „Muss", sondern eher ein „Könnte".

3. Neue Handlungen werden zu Routinen

Im dritten Schritt werden diese Verhaltensweise oder Routinen durch neue ersetzt. Auch ich habe dies erprobt und empfehle, die Ziele, also Werte, die man verändern will, aufzuschreiben. Beispiele sind:

- Ich möchte Entscheidungen schneller treffen.
- Ich will Mitarbeitern mehr zuhören.
- Ich will Mitarbeitern mehr vertrauen.

Ich empfehle dabei, sich in jeder Situation diese Ziele kurz erneut ins Bewusstsein zu rufen und selbst zu bewerten, ob diese umgesetzt worden sind. Haben Sie beispielsweise heute im Meeting die Entscheidung schnell genug getroffen oder haben Sie das Team durch Wartezeit frustriert? Haben Sie heute im Einzelgespräch mit Herrn Müller genug zugehört oder haben Sie wieder zu viel geredet?

Ein wichtiger Tipp ist, dass Sie bei Einhaltung der neuen Gewohnheiten eine Belohnung einplanen. Beispiele sind ein Stück Schokolade oder etwas anderes, dass Ihnen Freude bereitet. Die Idee hinter dieser kleinen Belohnung ist, dass in Ihrem Gehirn eine Art Auslöser gesetzt wird: Wenn das nächste Mal die gleiche Aktion passiert, dann führe ich folgendes Routineprogramm durch, da es zu einer Belohnung führt. Laut wissenschaftlichen Studien müssen Sie die neue Handlung übrigens 66 Mal ausführen, bis diese zur Routine wird. Die Psychologin Phillippa Lally vom University College in London hat dies in einer Untersuchung erforscht. Sie gab dazu 96 Probanden die Aufgabe, sich drei neue Gewohnheiten (u. a. 15 Minuten Spaziergang jeden Morgen) anzueignen. Die Anzahl der bewussten Ausführungen der Tätigkeit bis zur automatisierten Gewohnheit war im Durchschnitt 66.

Fazit

Agile Führung ist ein Verhalten sowie ein „Mindset". Gerade diese Tatsache macht es sehr schwer, agile Führung wirklich treffend zu beschreiben und zu definieren. Zentraler Punkt ist eine Änderung der eigenen Persönlichkeit. Dabei gilt es, Gewohnheiten und eigene Werte zu verändern. Dies ist nicht einfach und es wird empfohlen, diese Veränderung durch Learning by Doing im Alltag zu erproben. Dazu ist es wichtig, aus gewissen Routinen auszubrechen und eigene Werte sowie Charaktereigenschaften zu verändern.

Quellen

Lindner, D. (2019). KMU im digitalen Wandel: Ergebnisse empirischer Studien. Wiesbaden: Springer Gabler.

Dominic Lindner
dominic.lindner@agile-unternehmen.de

Dominic Lindner ist seit 2016 externer Doktorand der FAU Erlangen-Nürnberg am Lehrstuhl für IT-Management und Associate Manager bei der noris network in Nürnberg. Sein Forschungsinteresse liegt im Bereich Arbeit 4.0, Agilität und Digital Leadership. Vorher absolvierte Dominic Lindner einen Master in Wirtschaftsinformatik in Nürnberg und Schweden und war 2 Jahre als IT-Consultant tätig. 2016 wurde seine Masterarbeit mit dem deutschen Studienpreis für Projektmanagement der GPM ausgezeichnet. Dominic Lindner bloggt aktiv auf agile-unternehmen.de.

web3j – das Tor zur Blockchain für die Java-Welt

Tim Zöller, ilum:e informatik ag

Bis zum Beginn des Jahres 2018 waren Blockchains in aller Munde. Neue Anwendungsfälle für Software wurden enthusiastisch diskutiert, die Kurse von Kryptowährungen wie Bitcoin und Ethereum erreichten unerwartete Höhen und waren über Wochen hinweg Nachrichtenthema – bis sich viele Anwendungsfälle als impraktikabel erwiesen und die Blase platzte. Doch die Technik bleibt interessant und kann sinnvoll eingesetzt werden. Dieser Artikel erörtert, wie man Java-Applikationen mit der Ethereum-Blockchain verbindet und für wen dies interessant sein kann.

Ethereum basiert auf dem gleichen Prinzip wie die meisten Blockchains: Es ist ein verteiltes Netzwerk, in dem die Netzwerkteilnehmer mit öffentlichen Transaktionen den globalen Zustand ändern. Da sich dieser Zustand komplett aus dem Transaktionslog ergibt, sind einmal in die Blockchain geschriebene Daten unveränderlich. Dabei werden diese Transaktionen nicht von einer zentralen Autorität im Netzwerk überprüft, sondern durch ein kryptographisches Verfahren, das den Netzwerkteilnehmern erlaubt, einen Konsens über valide Transaktionen auszuhandeln. Dieses Verfahren wird oft als „Mining" bezeichnet. Dabei werden Transaktionen gesammelt und in Blöcken zusammengefasst. Um den Inhalt der Blöcke und deren Reihenfolge unveränderlich zu machen, wird jeder Block mit einem Hash versehen, der aus den enthaltenen Transaktionen und dem Hash des vorhergehenden Blocks besteht. Das Berechnen des Hash ist aufwendig, die Überprüfung auf Korrektheit jedoch simpel. Versucht ein Netzwerkteilnehmer, den Inhalt eines zurückliegenden Blocks zu manipulieren, werden die Hashes aller nachfolgenden Blöcke ungültig und müssten ebenfalls neu berechnet werden. Dies würde eine derart hohe Rechenleistung benötigen, dass es als nicht praktikabel angesehen wird. Die Teilnahme am Netzwerk steht jedem offen, hierzu muss lediglich ein Account erstellt werden, der aus einem privaten und einem zugehörigen öffentlichen Schlüssel besteht. Basierend auf dem Schlüsselpaar ergibt sich eine 42-stellige Adresse, mit der der Account im Netzwerk angesprochen werden kann. Möchten Anwender mit ihrem Account eine Transaktion veranlassen, wird sie mit dem privaten Schlüssel signiert, das Netzwerk verifiziert die Herkunft der Transaktion mit dem öffentlichen Schlüssel. Verliert ein Nutzer seinen privaten Schlüssel, kann er nicht mehr auf seinen Account zugreifen. Das Veranlassen von Transaktionen muss im Ethereum-Netzwerk mit der Kryptowährung Ether bezahlt werden, die einen realen Gegenwert in einer echten Währung besitzt. Hierbei handelt es sich um einen Mechanismus, der das Netzwerk vor Spam schützt.

Smart Contracts – dezentral ausgeführte Programme

Ethereum ermöglicht darüber hinaus die Ausführung von Programmen im Netzwerk. Diese werden dezentral installiert und basieren auf sogenannten Smart Contracts, die meistens in der Programmiersprache „Solidity" geschrieben werden. Wird ein Smart Contract in das Ethereum-Netzwerk installiert, empfangen ihn alle Netzwerkteilnehmer. Um den Aufruf zu ermöglichen, erhält jedes Programm einen eigenen Ethereum-Account mit öffentlicher Adresse. Für diesen Code gilt dieselbe Voraussetzung wie für alle Transaktionen im Netzwerk: Er ist unveränderlich. Ein einmal ausgebrachtes Programm kann nicht mehr entfernt oder nachträglich geändert werden. Nach der Verteilung im Netzwerk sind alle Knoten in der Lage, den Code lokal auszuführen. Dies geschieht, wenn ein Netzwerkteilnehmer eine Smart-Contract-Funktion mit einer Transaktion aufruft. Jeder Netzwerkknoten, der die Transaktion empfängt, führt den Programmcode aus und wertet den Aufruf aus. Der Konsens über ausgelöste Seiteneffekte und errechnete Ergebnisse wird danach wieder dezentral im Netzwerk ausgehandelt. Da der Code auf vielen verschiedenen Knoten ausgeführt wird, muss er zwingend deterministisch sein, damit alle Knoten zum selben Ergebnis kommen. Die Kombination dieser Faktoren kann ein Vertrauen in die Autoren von Software obsolet machen: Der Netzwerkteilnehmer weiß, dass ein Programm unter der ihm bekannten Adresse vom Ersteller nicht geändert werden kann, beispielsweise um ein Verhalten zu seinem Nachteil zu verändern. Außerdem ist es niemandem möglich, einmal in Smart Contracts gespeicherte Werte nachträglich zu ändern.

Kommunikation mit Ethereum-Nodes

Es existiert nicht lediglich eine einzelne Clientsoftware für Ethereum, sondern eine Vielzahl verschiedener Implementierungen gegen denselben Standard. Es gibt unter anderem Projekte auf Basis von C++ (cpp-ethereum), Go (geth), Rust (Parity) und Java (Harmony). Jeder Client muss die Ethereum Virtual Machine (EVM) implementieren, in der der Smart Contract als Bytecode ausgeführt wird. Die homogene Clientlandschaft bringt Vorteile mit sich: Durch eine theoretische Sicherheitslücke ist immer nur ein kleiner Teil der Netzwerkteilnehmer kompromittierbar. Es ist beispielsweise unwahrscheinlich, denselben Fehler im Go-Client und im C++-Client zu finden. Darüber hinaus macht diese Vielfalt die Standardisierung der Clientschnittstellen nötig, um Drittsoftware die Kommunikation mit ihnen zu ermöglichen. Dieser Standard mit dem Namen „JSON-RPC" erlaubt es, auch Java-Applikationen mit beliebigen Ethereum-Knoten im selben lokalen Netzwerk zu kommunizieren. JSON-RPC benötigt kein spezielles Transportmedium, es kann unter anderem über HTTP oder Sockets genutzt werden. Entwickler können beispielsweise ein Web-Frontend entwerfen, das mit JSON-RPC über HTTP mit einem Ethereum-Client kommuniziert, um eine Benutzeroberfläche für einen Smart Contract anzubieten oder um automatisiert Transaktionen in der Ethereum-Blockchain zu veranlassen.

Entwickeln und Testen von Smart Contracts

Wie im Vorfeld bereits beschrieben, muss das Auslösen von Transaktionen in der Ethereum-Blockchain mit der Kryptowährung Ether bezahlt werden, die einen realen Gegenwert besitzt. Da jede Interaktion, lesende Zugriffe ausgenommen, mit der Blockchain eine Transaktion benötigt, würde das Testen auf der produktiven Blockchain schnell teuer werden. Um eine einfachere Entwicklung zu ermöglichen, wurden in der Vergangenheit mehrere Testnetzwerke gestartet, etwa Ropsten, Rinkeby oder Kovan. Diese verhalten sich zwar wie die produktive Blockchain, stellen Entwicklern aber einfache Möglichkeiten bereit, kostenlos ausreichend Ether zu erhalten. Da in diesen Netzwerken keine realen Kosten mit dem Veranlassen von Transaktionen verbunden sind, sind sie sehr anfällig für Überlastung. Weiterhin sind diese Blockchains öffentlich und global verfügbar – eventuelle Testdaten und frühe Versionen der entwickelten Software sind für alle Netzwerkteilnehmer einsehbar. Im Gegenzug müssen Entwickler alle bisherigen Transaktionen des Netzwerks lokal auf ihren Rechner synchronisieren, obwohl sie für sie gar nicht relevant sind. Das Einrichten eines lokalen Ethereum-Netzwerks auf dem Laptop kann also durchaus sinnvoll sein. Zur Vereinfachung habe ich 2018 auf GitHub das Repository „geth-dev" unter der Eclipse Public License 2.0 bereitgestellt *[1]*. Dieses beinhaltet eine „docker-compose"-Datei, die ein Ethereum-Netzwerk mit drei aktiven Clients startet. Dabei handelt es sich um ein reines Entwick-

```
<dependency>
<groupId>org.web3j</groupId>
<artifactId>core</artifactId>
<version>4.3.1</version>
</dependency>
```

Listing 1: Maven Dependency zur aktuellen Version von web3j

```
Web3j web3j = Web3j.build(new HttpService("http://localhost:8545"));
try {
Web3ClientVersion clientVersion = web3j.web3ClientVersion().send();
System.out.println("Connected to Ethereum client: " + clientVersion.getWeb3ClientVersion());
} catch(IOException e) {
System.err.println(e.getMessage());
}
```

Listing 2: Verbindungsaufbau zum Client

```
// Asynchroner Aufruf
web3j.web3ClientVersion()
 .sendAsync()
 .thenApply(Web3ClientVersion::getWeb3ClientVersion)
 .thenApply("Connected to Ethereum client: "::concat)
 .thenAccept(System.out::print);

// Reaktiver Aufruf
web3j.web3ClientVersion()
 .flowable()
 .map(Web3ClientVersion::getWeb3ClientVersion)
 .map("Connected to Ethereum client: „::concat)
 .forEach(System.out::println);
```

Listing 3: Asynchrone und reaktive Kommunikation mit dem Ethereum-Node

lungsnetzwerk, das nicht die Grundlage einer produktiven privaten Blockchain sein sollte. Um die Entwickler-Hardware nicht durch aufwendiges Mining zu belasten, nutzt das Netzwerk einen ressourcenschonenden Algorithmus zur Konsensfindung. Um das Netzwerk auf einem Rechner mit Docker zu starten, muss lediglich das Repository geklont und im Hauptverzeichnis der Befehl `docker-compose up` ausgeführt werden.

web3j – ein Java Wrapper für JSON-RPC und mehr

Da JSON-RPC über HTTP oder Sockets angesprochen werden kann, können Java-Applikationen theoretisch mit einem Ethereum-Node kommunizieren, ohne spezielle Bibliotheken einzubinden. Um diesen Aufwand nicht betreiben zu müssen, wurde web3j ins Leben gerufen, eine leichtgewichtige Library, die eine simple Integration von JVM-Applikationen mit der Ethereum-Blockchain bietet. Sie existiert seit Ende 2016 und steht unter der Apache License 2.0. Zum Umfang des Projekts gehören neben dieser Kernbibliothek auch ein Spring Boot Starter, ein Maven- und ein Gradle-Plug-in, um Wrapper-Klassen für Smart Contracts zu erstellen, und ein Adapter zu Hyperledger Fabric, ein Distributed Ledger der Linux Foundation.

Java und web3j – die erste Kommunikation

Wie die Interaktion einer Java-Anwendung über web3j mit einer Ethereum-Blockchain funktioniert, lässt sich am besten mit einem Beispiel veranschaulichen. Um das Beispiel ausführen zu können, wird neben Maven ein lokaler Ethereum-Knoten benötigt, der über http://localhost:8545 eine JSON-RPC-Verbindung über HTTP akzeptiert. Die Abhängigkeit zum web3j-Projekt wird zunächst in der „pom.xml" des Maven-Projekts hinzugefügt *(siehe Listing 1)*.

Der Verbindungsaufbau und die Kommunikation mit dem Client werden im *Listing 2* dargestellt. Zunächst wird eine web3j-Instanz erstellt, die sich über eine HTTP-Verbindung zum lokalen Ethereum-Node verbindet. Ist die Verbindung erfolgreich, kann sie im Anschluss über die Variable `web3j` referenziert werden. Kann keine Verbindung hergestellt werden, wird eine `java.net.ConnectException` geworfen. Im nächsten Schritt wird eine Anfrage über die Version

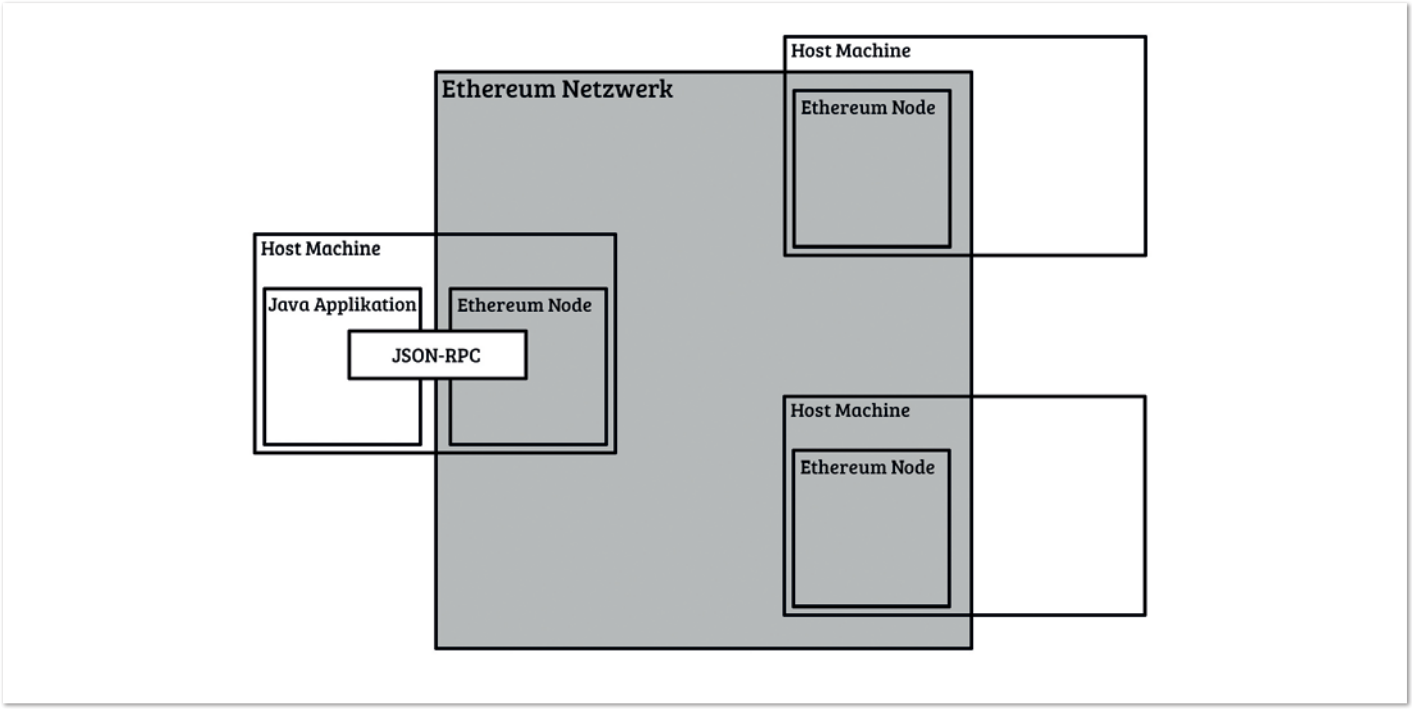

Abbildung 1: Struktur einer web3j-Applikation mit Java (© Tim Zöller)

```
pragma solidity ^0.5.2;

contract HashSaver {

    struct HashEntry {
        address owner;
        uint256 created;
    }

    mapping(string=> HashEntry) hashes;

    constructor() public {
    }

    function saveHash(string memory fileHash) public {
        assert(hashes[fileHash].owner == address(0x0));
        hashes[fileHash].owner = msg.sender;
        hashes[fileHash].created = now;
    }

    function checkOwnership(string memory fileHash) public view returns(address, uint256) {
       return(hashes[fileHash].owner, hashes[fileHash].created);
    }
}
```

Listing 4: Ein Smart Contract in der Sprache Solidity, der Hashes von Dateien speichert

```
<plugin>
    <groupId>org.web3j</groupId>
    <artifactId>web3j-maven-plugin</artifactId>
    <version>4.1.0</version>
    <configuration>
        <packageName>de.tutorial.web3.generated</packageName>
        <sourceDestination>src/main/java/generated</sourceDestination>
        <nativeJavaType>true</nativeJavaType>
        <outputFormat>java</outputFormat>
        <soliditySourceFiles>
            <directory>src/main/resources</directory>
            <includes>
                <include>/.sol</include>
            </includes>
        </soliditySourceFiles>
        <outputDirectory>
            <java>src/main/java</java>
        </outputDirectory>
    </configuration>
</plugin>
```

Listing 5: Konfiguration des web3j-Maven-Plug-ins in der „pom.xml“

```
public class HashSaverClient {

    private final static String CONTRACT_ADDRESS = "0x99cf8ae9e2aaae69cfc1038271f1a13307662063";

    public static void main(String... args) {
        Web3j web3j = Web3j.build(new HttpService("http://localhost:8545"));
        Credentials credentials = Credentials.create("bc5b578e0dcb2dbf98dd6e5fe62cb5a28b84a55e15fc112d4ca88e1f62bd7c35");

        HashSaver contract = HashSaver.load(CONTRACT_ADDRESS, web3j, credentials, new DefaultGasProvider());
        String fileHash = new FileHasher().getSHA("~/Desktop/MyFile.pdf");

        try {
            TransactionReceipt receipt = contract.saveHash(fileHash).send();
            if (receipt.isStatusOK()) {
                System.out.println("Stored file hash with TX " + receipt.getTransactionHash());
            } else {
                System.out.println("Was not able to store file hash");
            }
        } catch (Exception e) {
            System.out.println("Was not able to store file hash, " + e.getMessage());
        }

    }

}
```

Listing 6: Java Client, der mit dem Smart Contract interagiert

des lokalen Knotens gesendet und das Ergebnis ausgegeben: `Connected to Ethereum client: Geth/v1.8.27-stable-4bcc0a37/linux-amd64/go1.10.4`. Die Verbindung war erfolgreich. Das Senden der Anfrage über die Methode `send` geschieht synchron und blockiert die Ausführung. In diesem Beispiel erfolgt keine Kommunikation mit dem Ethereum-Netzwerk mit einer Transaktion, sondern lediglich ein lokaler Lesezugriff. Ein synchroner Aufruf wird demnach nicht lange blockieren. Da eine Transaktion aber vom Netzwerk bestätigt werden muss, können die Antwortzeiten für sie drastisch höher liegen. Aus diesem Grund bietet das web3j-API noch zwei weitere Möglichkeiten, mit Ethereum-Nodes zu kommunizieren – asynchron und reaktiv *(siehe Listing 3)*. Die reaktiven Aufrufe funktionieren mit RxJava.

Entwicklung und Aufruf eines Smart Contract

Der Smart Contract in *Listing 4* stellt Funktionen bereit, die ein Objekt „HashEntry" speichern können. Ein `HashEntry` besteht aus der Senderadresse der Transaktion und einem Zeitstempel, der das Erstelldatum markiert. Gespeichert werden `HashEntry`-Instanzen in einer Map `hashes`, die einen beliebigen String als Schlüssel enthält. Dieses Konstrukt ermöglicht Nutzern, einen Datei-Hash mit ihrer Ethereum-Adresse und einem Zeitstempel zu assoziieren, um nachzuweisen, dass sie zu einem bestimmten Zeitpunkt in Besitz einer Datei waren. Gespeichert werden die Einträge in der Funktion `saveHash`. Nachdem mit einem `assert` sichergestellt wird, dass für den Hash einer Datei noch kein Eintrag existiert, wird der Eintrag inklusive Absenderadresse und aktuellem Zeitstempel gespeichert. Gelesen werden diese Daten, indem die Funktion `checkOwnership` aufgerufen wird. Diese Funktion hat keine Seiteneffekte und erlaubt lesenden Zugriff ohne Transaktion, deshalb wird sie mit dem Keyword `view` versehen.

Beim Aufruf dieser Funktionen aus einem Java-Programm heraus kann web3j ebenfalls unterstützen. Mit dem zugehörigen Maven-Plug-in ist es möglich, einen Java-Wrapper für den Smart Contract generieren zu lassen. Die Konfiguration des Plug-ins lässt sich *Listing 5* entnehmen. Das Maven-Plug-in kompiliert, sofern der Solidity Compiler `solc` in der richtigen Version auf dem System installiert ist, mit dem Befehl `mvn web3j:generate-sources` den Solidity Source Code und generiert eine Java-Klasse `HashSaver.java`, die Methoden anbietet, um Transaktionen an den Smart Contract zu senden. Darüber hinaus bietet die Klasse `HashSaver` eine statische Methode `deploy` an, mit der der Smart Contract im Netzwerk installiert werden kann. Sie liefert die Ethereum-Adresse zurück, unter welcher der Contract aufgerufen werden kann.

Unter Nutzung der generierten Klasse ist es Entwicklern jetzt möglich, mit dem Smart Contract zu interagieren, ohne viel Boilerplate-Code schreiben zu müssen *(Listing 6)*. Nachdem die web3j-Verbindung mit dem lokalen Node hergestellt wird, muss ein Credentials-Objekt erstellt werden, das den privaten Schlüssel eines Ethereum-Accounts hält. Im Anschluss wird der Smart Contract mit der statischen Methode `load` der Klasse `Hashsaver` unter Angabe der Contract-Adresse, der web3j-Verbindung und der Credentials geladen. Die Methode `saveHash` bildet die gleichnamige Methode des Smart Contract ab, der Aufruf `contract.saveHash(fileHash).send()` sendet also direkt eine Transaktion an das Ethereum-Netzwerk und speichert den übergebenen Hash in der Blockchain. Das Ergebnis der Transaktion kann man am resultierenden `TransactionReceipt` überprüfen. Die Java-Applikation hat erfolgreich mit dem Ethereum-Netzwerk kommuniziert.

Fazit

Anhand der Beispiele lässt sich zeigen, dass web3j einen Weg gefunden hat, viel Komplexität von Java-Entwicklern, die mit der Ethereum-Blockchain arbeiten möchten, fernzuhalten. Dies trifft nicht nur auf den Verbindungsaufbau zu, auch das Installieren und Interagieren mit Smart Contracts wird durch web3j deutlich vereinfacht und kann so beispielsweise in Build Pipelines automatisiert erfolgen. Zusätzlich sind Entwicklern alle Möglichkeiten gegeben, web3j in ihre eigene Architektur zu integrieren – sei sie synchron, asynchron oder sogar reaktiv.

Quellen

[1] *https://github.com/javahippie/geth-dev*

Tim Zöller
mail@tim-zoeller.de

Tim Zöller arbeitet als IT-Consultant bei ilum:e informatik AG in Mainz und entwickelt seit 10 Jahren Software. Er hilft Unternehmen dabei, ihre Prozesse mit Java zu digitalisieren, und ist Mitgründer der JUG Mainz.

Identity und Access Management leicht gemacht

Mathias Conradt, Auth0

Spring Security ist nicht nur ein mächtiges Framework zur Authentifizierung und Autorisierung von Benutzern, sondern auch der De-facto-Standard zur Absicherung von Java-Applikationen. In Verbindung mit einer Identity-as-a-Service-Plattform (IDaaS) lassen sich sicherheitsrelevante Features in kurzer Zeit und mit wenig eigenem Code abbilden.

Mit Spring 5.1 wurden *OAuth 2.0* und die *OpenID-Connect-(OIDC)*-Unterstützung rundum erneuert beziehungsweise re-implementiert. Während das *OAuth-2.0*-Protokoll zur Autorisierung oder Access Delegation auf Ressourcen wie APIs oder sonstige Backends dient, handelt es sich bei *OpenID* Connect um einen Identity Layer, der auf *OAuth 2.0* aufsetzt und rein der Benutzer-Authentifizierung dient.

Im Folgenden wollen wir uns anschauen, wie wir Benutzern einer Spring-MVC-Applikation ermöglichen können, sich mittels *OpenID Connect* über einen eigenen *Identity-Provider (IdP)* zu authentifizieren, ohne selbst einen *Autorisierungsserver* implementieren zu müssen. Hierfür nutzen wir Auth0 als IDaaS-Plattform. Technisch wird hier – für den Entwickler transparent – eine Mongo-Datenbank in der Auth0-AWS-Cloud verwendet. Im zweiten Schritt schauen wir uns an, wie wir als Entwickler mühelos weitere Social IDPs (Facebook, Twitter, LinkedIn etc.) ohne viel Mehraufwand föderiert anbinden können.

OpenID Connect und der OAuth 2.0 Authorization Code Grant

Konzeptionell sieht der Ablauf einer *OpenID-Connect*-basierten Authentifizierung gemäß dem *„OAuth 2.0 Authorization Code Grant"* wie folgt aus *(siehe Abbildung 1)*:

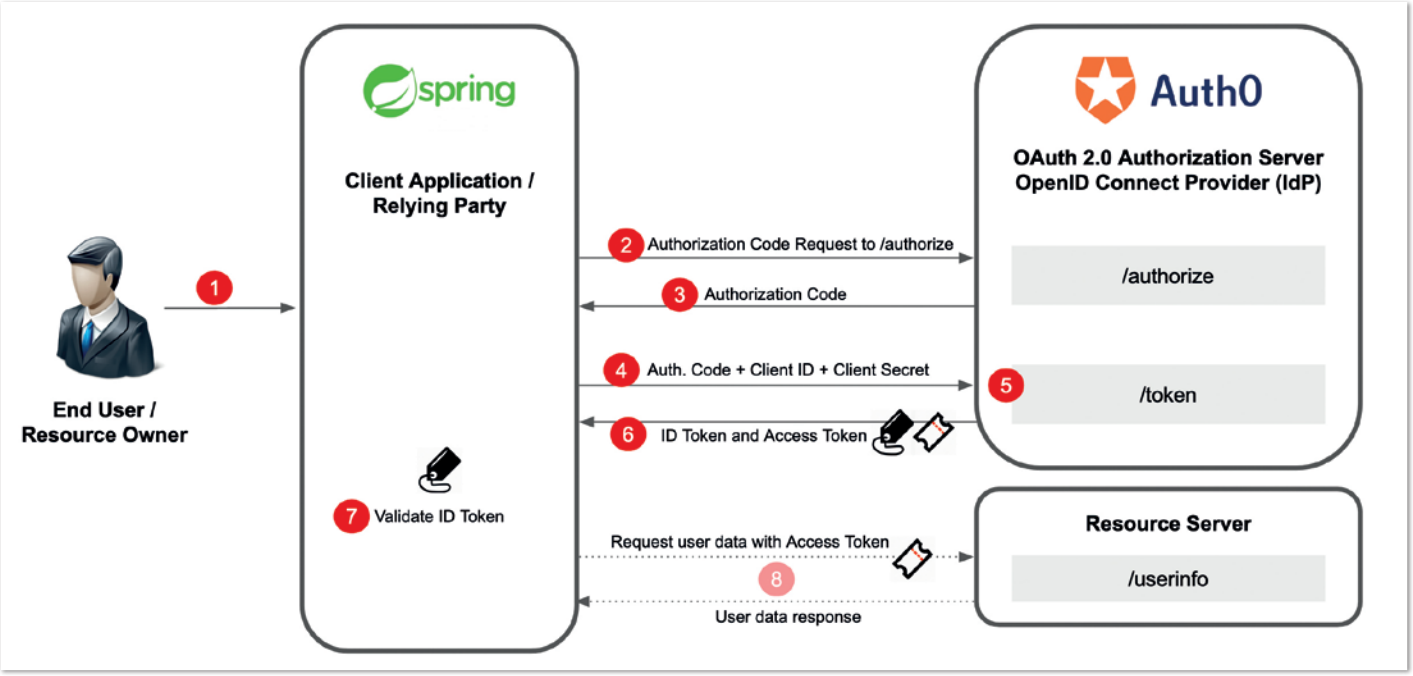

Abbildung 1: OpenID Connect mit Authorization Code Grant [1]

1. Der Benutzer ruft die Spring-Applikation im Browser auf.
2. Die Spring-Applikation beziehungsweise Spring Security leitet den Benutzer aufgrund unserer Sicherheitskonfiguration zum Autorisierungsserver/IdP (`/authorize`-Endpunkt) weiter, in unserem Fall Auth0. Sofern keine aktive Session beim IdP besteht, wird dem Benutzer eine Login-Seite und/oder ein Consent-Dialog angezeigt. Der Benutzer authentifiziert sich mit einer der konfigurierten Login-Optionen (zum Beispiel Benutzername/Passwort) und sieht daraufhin einen Consent-Dialog, der die Berechtigungen (beispielsweise das Auslesen des Benutzerprofils) auflistet, die der Autorisierungsserver/IdP (Auth0) der anfragenden Spring-Applikation gewähren wird.
3. Der Autorisierungs-Server/IdP (Auth0) leitet den User mit einem *„Authorization Code"* zurück an unsere Spring-Applikation.
4. Spring Security sendet den Code an den Autorisierungsserver/IdP. (`/oauth/token`-Endpunkt) zusammen mit Client ID und Client Secret.
5. Der Autorisierungsserver/IdP (Auth0) verifiziert den Code, Client ID und Client Secret.
6. Der Autorisierungsserver/IdP (Auth0) gibt *„ID-Token"* sowie *„Access-Token"* (und optional *„Refresh-Token"*) an die Spring-Applikation zurück.
7. Unsere Spring-Applikation validiert und dekodiert den *„ID-Token"* und kann dessen Inhalt (Payload) zur Anzeige des Benutzerprofils nutzen.
8. Optional kann die Spring-Applikation den *„Access-Token"* nutzen, um weitere Benutzerinformationen vom – gemäß OpenID-Connect-standardisierten – *UserInfo*-Endpunkt des Autorisierungsservers/IdP abzurufen. Für unsere Demo reicht uns jedoch der *„ID-Token"*.

Eine Anmerkung am Rande: Theoretisch könnten wir für eine reine OpenID-Connect-Authentifizierung, bei der wir lediglich den *„ID-Token"* und keinen *„Access-Token"* verwenden, auch den *„Implicit Grant mit Form Post Response Mode"* *[2]* in Erwägung ziehen, allerdings wird dieser von Spring Security 5.1 derzeit nicht unterstützt. Dies würde die generelle Komplexität verringern und das Verwalten von Client Secrets unnötig machen.

Die Implementierung erfolgt in zwei Schritten. Zuerst konfigurieren wir unseren Identity-Provider seitens Auth0, danach erfolgt die Integration in ein Spring-Boot-beziehungsweise Spring-Security-Projekt.

IdP-Konfiguration seitens Auth0

Wir erstellen zunächst auf *https://auth0.com* einen kostenfreien Account sowie Mandanten und registrieren unsere geplante Java-Applikation im Auth0-Dashboard unter dem Punkt *Applications*. Als Technologie-Stack wählen wir *Regular Web App* und *Java Spring MVC* aus. Nach erfolgreichem Anlegen der Client-Applikation und Wechsel zum *Settings*-Tab erhalten wir eine *Client ID* sowie ein *Client Secret*. Dieses benötigen wir später bei der Konfiguration des Auth0-IdP in der Spring-Boot-Applikation. In den Settings konfigurieren wir noch die erlaubten Callback-URLs auf *http://localhost:3000/login/oauth2/code/auth0* sowie die erlaubten Logout-URLs auf *http://localhost:3000*. Optional kann ein Icon vergeben werden. Ich verwende in meinem Beispiel das Java-Logo *(siehe Abbildung 2)*.

Nach den getroffenen Vorbereitungen können wir nun mit der eigentlichen App-Entwicklung beginnen.

Spring-Boot-Projektinitialisierung

Wir erstellen ein Spring-Boot-Projekt mit wesentlichen, für *„OpenID Connect"* relevanten Abhängigkeiten *(siehe Listing 1)*. Das gesamte Beispielprojekt ist auf GitHub verfügbar *[3]*.

Um unsere Applikation vor nicht authentifizierten Zugriffen zu schützen, erstellen wir zunächst eine Konfigurations-Klasse, die von `WebSecurityConfigurerAdapter` *[4]* ableitet. In ihr überschreiben wir die `configure()`-Methode und definieren die Sicherheitsregeln für eingehende HTTP-Requests. In unserem Fall wollen wir für die

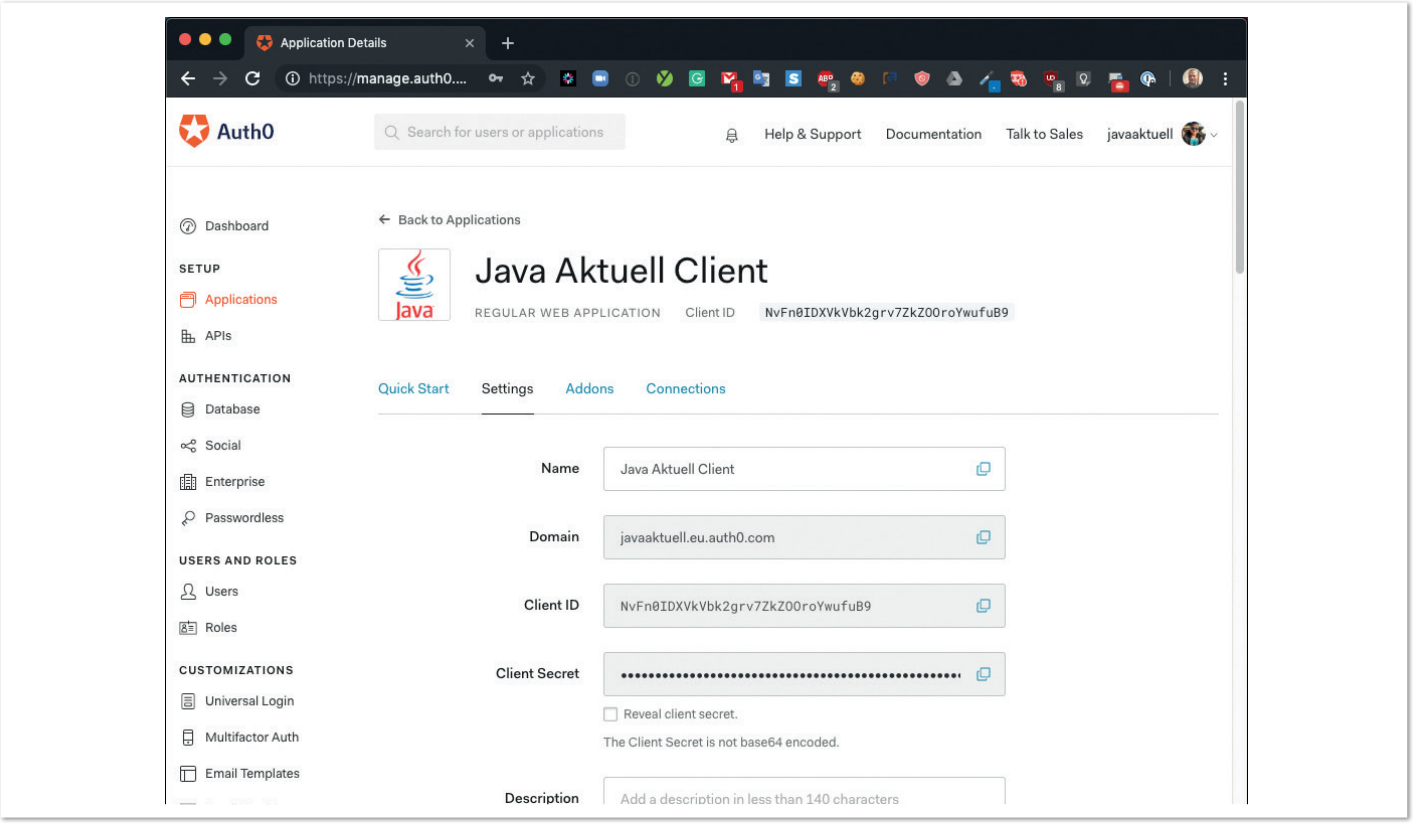

Abbildung 2: Client-Application-Settings im Auth0-Dashboard

gesamte Applikation eine Authentifizierung voraussetzen. Außerdem wollen wir OAuth2-Login aktivieren. Zu guter Letzt definieren wir noch einen eigenen `LogoutHandler`, der den Default-Handler überschreibt *(siehe Listing 2)*.

Der `oauth2Login` holt sich die Konfiguration des zu verwendenden Identity-Providers aus der Applikationskonfiguration `application.yml`. In *Listing 3* definieren wir zunächst Auth0 als Identity-Provider.

Betrachten wir die einzelnen Konfigurationsabschnitte genauer: Die in *Listing 3* gezeigte Konfiguration unterteilt sich in zwei Teile. Der Identity-Provider wird im fett markierten Bereich definiert.

Über die Issuer-URI ist Spring imstande, sich über das als *„OpenID Connect Discovery" [5]* standardisierte Verfahren die notwendigen OAuth-2.0-Endpunkte wie Authorization-Endpoint sowie Token-Endpoint zur Kommunikation zwischen Client-Applikation (Spring) und Autorisierungsserver (Auth0) selbstständig herauszusuchen.

Der obere Teil registriert die zugehörige Client-Applikation *(siehe Listing 4)*. Der `client-name` kann hierbei beliebig gewählt werden; es bietet sich der eigenen Übersicht halber an, den gleichen Namen zu verwenden, wie in Auth0 bei der Registrierung der Applikation gewählt. Als `scope` fragen wir `openid profile email offline_access` an. Hiermit zeigen wir an, dass es sich um einen OpenID

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

Listing 1: Wesentliche Dependencies in der pom.xml

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepository;
    private final LogoutController logoutController;

    public SecurityConfig(LogoutController logoutController) {
        this.logoutController = logoutController;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2Login()
            .and()
            .logout()
            .addLogoutHandler(logoutController);
        }
    }
}
```

Listing 2: SecurityConfig.java

```
[...]
spring:
  security:
    oauth2:
      client:
        registration:
          auth0:
            client-name: Java Aktuell Client
            client-id: {clientID}
            client-secret: {clientSecret}
            scope:
            - openid
            - profile
            - email
            - offline_access
            authorization-grant-type: authorization_code
            logout-uri: https://javaaktuell.eu.auth0.com/v2/logout?client_id=[...]
        provider:
          auth0:
            issuer-uri: https://javaaktuell.eu.auth0.com/
            user-name-attribute: name
```

Listing 3: Konfiguration des Identity-Providers in der application.yml

```
registration:
  auth0:
    client-name: Java Aktuell Client
    client-id: {clientID}
    client-secret: {clientSecret}
    scope:
      - openid
      - profile
      - email
      - offline_access
    authorization-grant-type: authorization_code
    logout-uri: [...]
```

Listing 4: Registrierung der zugehörigen Client-Applikation

Connect Request handelt, wir die Profildaten des Users zurückerhalten möchten und dass wir neben Access- und ID-Token auch ein Refresh-Token erhalten möchten (`offline_access`). Da es sich um eine reguläre Webapplikation mit Backend handelt, verwenden wir hier den Authorization Code Grant von OAuth 2.0 – zumal Spring Security aktuell auch nur diesen unterstützt: `authorization-grant-type: authorization_code`.

Als Nächstes benötigen wir einen Controller, wir nennen ihn `HomeController`, der authentifizierte Requests an *http://localhost:3000/*

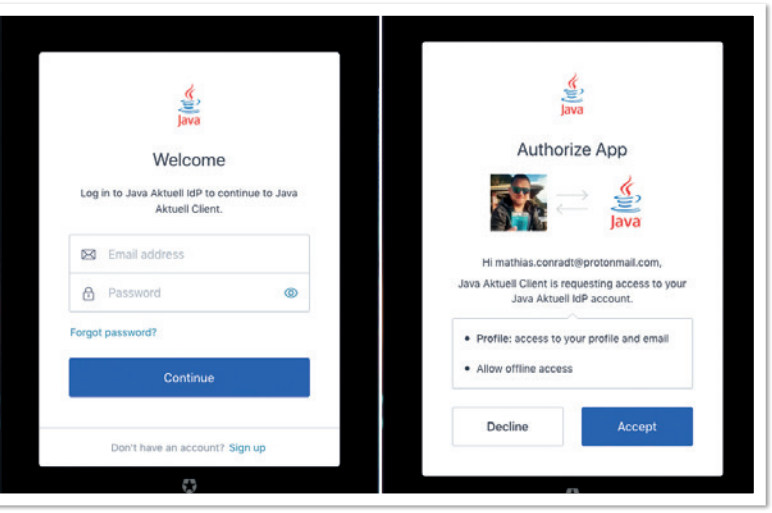

Abbildung 3: Login- und Registrierungs-Seite des Identity-Providers

entgegennimmt und das Benutzerprofil (extrahierte Claims des ID-Tokens) darstellt. Hierfür lassen wir eine Methode `index()` die GET-Anfragen am Root-Pfad entgegennehmen. In ihr haben wir sowohl ein `Model`-Objekt als auch den erhaltenen `OAuth2AuthenticationToken` *[6]*, der eine OAuth2-Authentifizierung repräsentiert und die Token-Information beinhaltet *(siehe Listing 5)*.

Die Claims aus dem ID-Token, die letztlich die Benutzerattribute und somit das Benutzerprofil darstellen, erhalten wir mittels `authentication.getPrincipal().getAttributes()`; wir können diese direkt an die View (index.html) unter dem von uns gewählten Attribut-Namen `userInfo` übergeben. Wie wir hier sehen, müssen wir uns um den OAuth-2.0-gemäßen Tausch von Autorisierungscode gegen einen Access- und ID-Token nicht kümmern, dies übernimmt Spring Security's OAuth- beziehungsweise OpenID-Support automatisch. Auch wird das Dekodieren und Validieren des ID-Tokens,

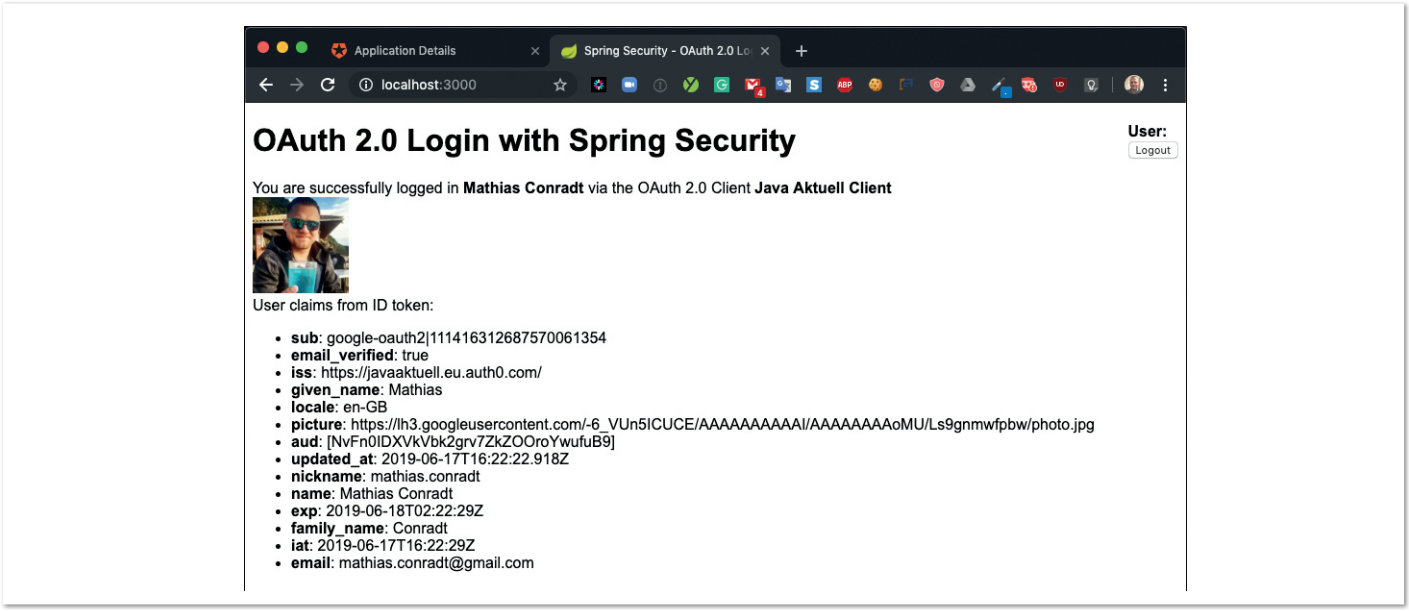

Abbildung 4: Benutzerprofil

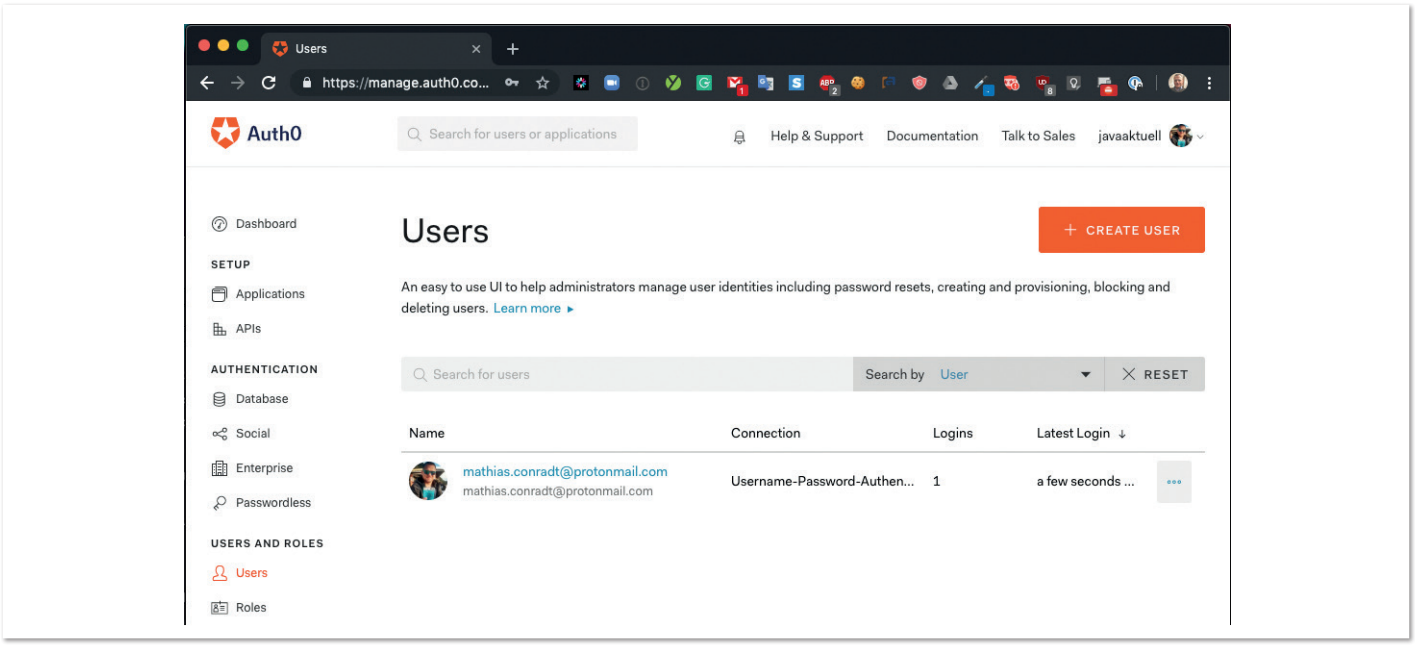

Abbildung 5: Benutzerübersicht im Auth0-Dashboard

der vom Identity-Provider als Base64Url-dekodierter und signierter *JWT (JSON Web Token)* *[7]* zurückkommt, automatisch vorgenommen und nimmt dem Entwickler bereits einiges an manueller Implementierungsarbeit ab.

Die zugehörige View, die Thymeleaf als Template Library verwendet und an die wir das Benutzerprofil über das `Model` übergeben, sieht wie in *Listing 6* gezeigt aus. Sie zeigt neben dem Namen der Client-Applikation das Avatar-Foto des Benutzers sowie dessen Profil an. Das Benutzerfoto wird seitens Auth0 automatisch vom Gravatar-Dienst auf Basis der bei der Registrierung verwendeten Benutzer-E-Mail-Adresse aufgesucht, sofern vorhanden.

Wir starten die Client-Applikation mit `mvn spring-boot:run` und rufen die URL *http://localhost:3000* im Browser auf. Da wir aktuell noch nicht authentifiziert sind, werden wir sofort zum konfigurierten Identity-Provider, Auth0, weitergeleitet und aufgefordert, uns einzuloggen beziehungsweise gegebenenfalls einen Account zu erstellen, sofern wir noch keinen haben. Außerdem erscheint ein Consent-Screen, über den wir einwilligen, dass unsere Spring-Applikation Zugriff auf unser Profil bei Auth0 erhalten darf *(siehe Abbildung 3)*.

Nach erfolgreicher Authentifizierung wird das Benutzerprofil, konkret die sogenannten „Claims" des ID-Tokens, dargestellt *(siehe Abbildung 4)*.

Bezüglich des Logout haben wir als Entwickler die Möglichkeit zu entscheiden, ob wir beim Klicken des Logout-Buttons lediglich die Session unserer Applikation beenden oder den Benutzer auch am IdP ausloggen wollen. Ist Letzteres der Fall, müssen wir einen eigenen `LogoutController` erstellen und in unserer `SecurityConfig` registrieren, der den entsprechenden Logout-Endpunkt des IdP, in unserem Fall Auth0, aufruft. Der IdP-seitige Logout-Endpunkt wird aus unserer `application.yml` ausgelesen und ist als `logout-uri` definiert: `https://javaaktuell.eu.auth0.com/v2/logout?client_id={clientID}&returnTo=http://localhost:3000` *(siehe Listing 7)*.

Andernfalls, sofern es uns ausreicht, beim Logout lediglich die aktuelle Session unserer eigenen Applikation zu beenden, nicht jedoch die des IdP, könnten wir ansonsten auf den `LogoutController` sowie die `logout()` und `addLogoutHandler()` in unserer `SecurityConfig`-Konfigurationsklasse verzichten.

Auth0 Dashboard, Social Login und MFA-Konfiguration

Wenn wir ins Auth0-Dashboard schauen, sehen wir unter dem Punkt `Users & Roles` nun auch den soeben registrierten Benutzer *(siehe Abbildung 5)*.

Um nun neben der Auth0-Datenbank weitere föderierte Social-Identity-Provider wie Google und Facebook zu integrieren sowie beispielsweise Multi-Factor-Authentication (MFA) zu aktivieren, können wir dies rein über die Konfiguration innerhalb von Auth0 vornehmen, ohne dabei unsere Java-Applikation anfassen zu müssen *(siehe Abbildungen 6 und 7)*.

```
@Controller
public class HomeController {
    public HomeController(OAuth2AuthorizedClientService authorizedClientService) {
        this.authorizedClientService = authorizedClientService;
    }
    @RequestMapping("/")
    public String index(Model model, OAuth2AuthenticationToken authentication) {
        model.addAttribute("userName", authentication.getName());
        model.addAttribute("clientName", authorizedClient.getClientRegistration().getClientName());
        model.addAttribute("userinfo", authentication.getPrincipal().getAttributes());
        return "index";
    }
}
```

Listing 5: HomeController.java

```
[...]
<h1>OAuth 2.0 Login with Spring Security</h1>
<div>
  You are successfully logged in <span style="font-weight:bold" th:text="${userName}"></span>
  via the OAuth 2.0 Client <span style="font-weight:bold" th:text="${clientName}"></span>
</div>
<div><img th:src="${userinfo.get('picture')}" height="100" width="100"/></div>
<div>
  User info from ID token:
  <ul>
    <li th:each="attribute : ${userinfo}">
      <span style="font-weight:bold" th:text="${attribute.key}"/>: <span th:text="${attribute.value}"></span>
    </li>
    </ul>
</div>
[...]
```

Listing 6: index.html

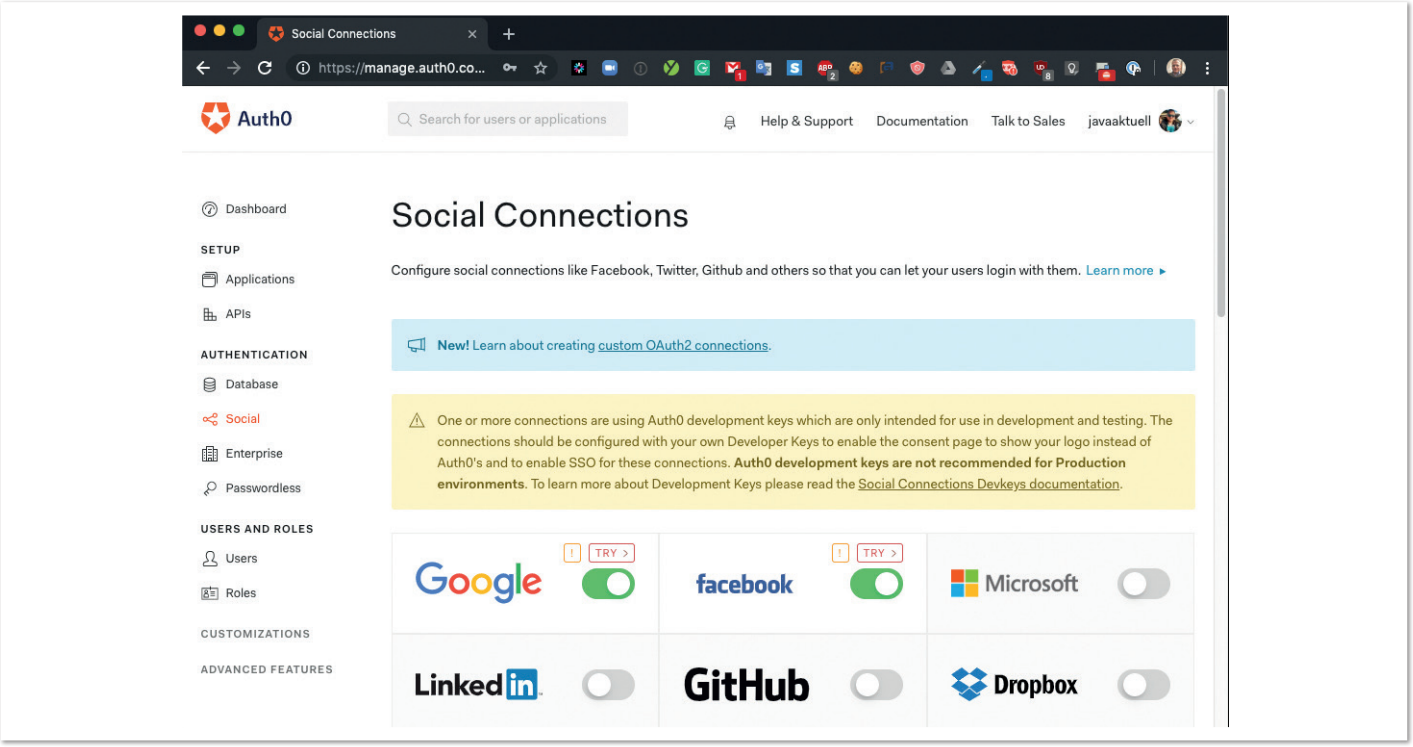

Abbildung 6: Aktivierung weiterer föderierter Social-Identity-Provider

```
@Controller
public class LogoutController extends SecurityContextLogoutHandler {

    private final ClientRegistrationRepository clientRegistrationRepository;
    private Logger logger = LoggerFactory.getLogger(LogoutController.class);

    @Value("${spring.security.oauth2.client.registration.auth0.logout-uri}")
    private String logoutUrl;

    public LogoutController(ClientRegistrationRepository clientRegistrationRepository) {
        this.clientRegistrationRepository = clientRegistrationRepository;
    }

    @Override
    public void logout(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentica-
tion authentication) {
        super.logout(httpServletRequest, httpServletResponse, authentication);
        try {
            httpServletResponse.sendRedirect(logoutUrl);
        } catch (IOException ioe) {
            logger.error("Error logging out.", ioe);
        }
    }
}
```

Listing 7: LogoutController.java

Wenn wir uns nun aus unserer Webapplikation unter *http://localhost:3000* ausloggen und erneut einzuloggen versuchen, stellen wir fest, dass wir nun sowohl zwei weitere Buttons für Facebook und Google als Authentifizierungs-Möglichkeit haben; des Weiteren werden wir nach dem Login aufgefordert, unsere Multi-Factor-Authentication (MFA) über das Scannen eines QR-Codes zu initialisieren *(siehe Abbildung 8)*.

Fazit

Ein OAuth-2.0- bzw. OpenID-Connect-basierter Login ist in Spring Security 5.1 mit sehr wenig Zeilen Code und minimaler Konfiguration möglich. Das Framework nimmt dem Entwickler einiges an Implementierungsarbeit ab, was die Details des OAuth2-Protokolls angeht.

Statt weitere Social-Identity-Provider direkt in der Java-Applikation zu konfigurieren, haben wir diese stattdessen mittels Auth0 als Broker föderiert integriert. Dies hat den Vorteil, dass die Konfiguration an zentraler Stelle vorgenommen werden kann. Bei der Entwicklung weiterer Client-Applikationen skaliert dies hervorragend und minimiert doppelten Aufwand. Dasselbe gilt ebenso für die Bereitstellung von MFA. Ein netter Seiteneffekt ist, dass wir so ebenfalls ein Single Sign-on über alle Applikationen hinweg bereitstellen können.

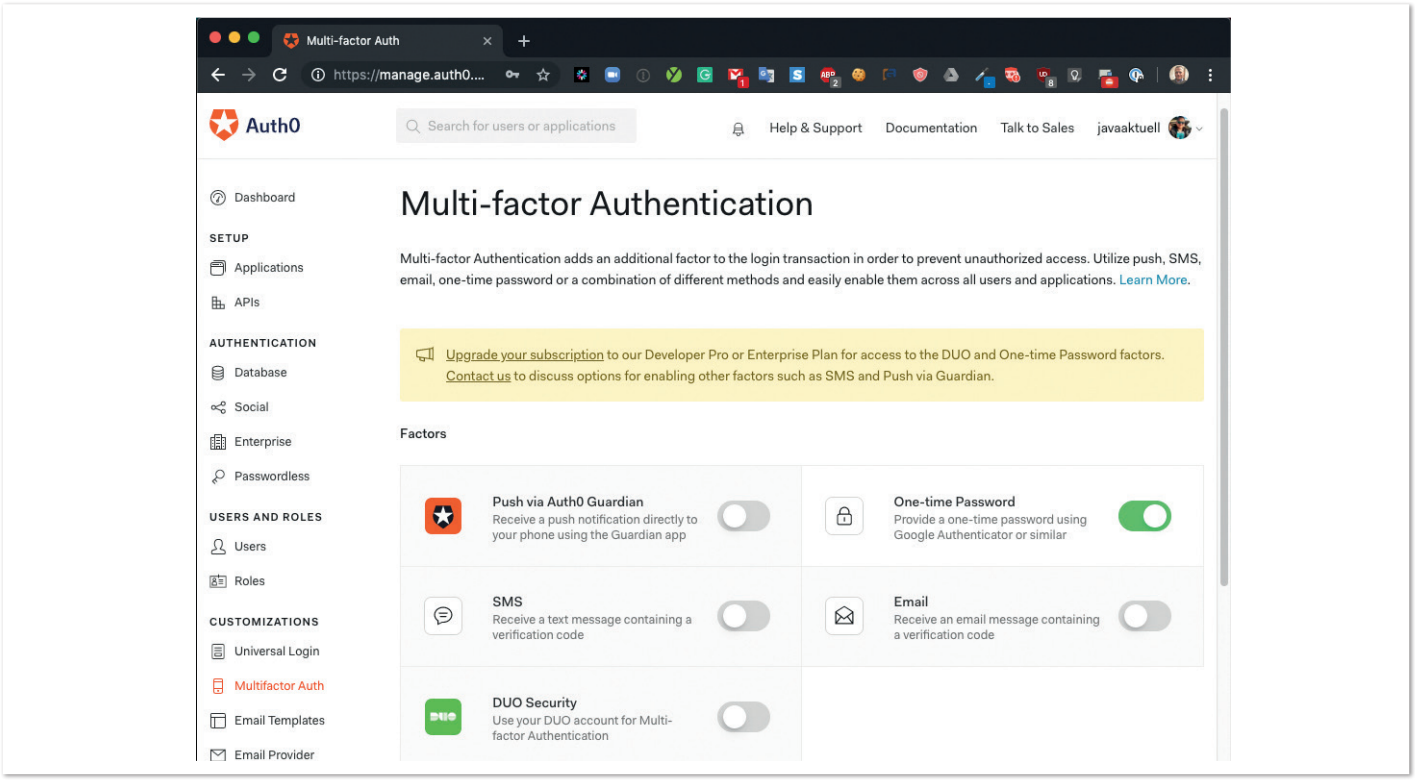

Abbildung 7: Aktivierung von MFA mittels Google Authenticator

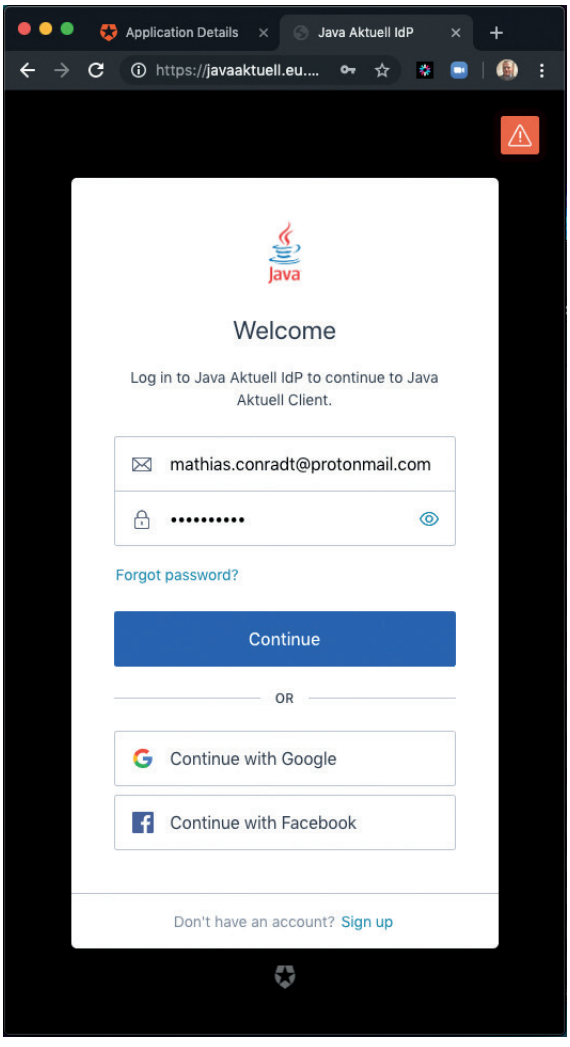

Abbildung 8: Login-Widget mit aktivierten Social Connections

Quellen

[1] *https://auth0.com/docs/flows/concepts/auth-code*

[2] *https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html*

[3] *https://github.com/mathiasconradt/java-aktuell*

[4] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html*

[5] *https://openid.net/specs/openid-connect-discovery-1_0.html*

[6] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/oauth2/client/authentication/OAuth2AuthenticationToken.html*

[7] *https://jwt.io/*

Mathias Conradt

mathias.conradt@auth0.com

Mathias Conradt ist Senior Solutions Engineer bei Auth0 und beschäftigt sich vorwiegend mit Identity & Access Management Lösungen rund um OAuth und OpenID Connect.

iJUG Verbund

Alle Mitglieder auf einen Blick

Der iJUG möchte alle Java-Usergroups unter einem Dach vereinen. So können sich alle Java-Usergroups in Deutschland, Österreich und der Schweiz, die sich für den Verbund interessieren und ihm beitreten möchten, gerne unter office@ijug.eu melden.

www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Stefan Kinnen. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Mylène Diacquenod
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@doag.org

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: Original © macrovector | *https://de.123rf.com*
S. 10: Bild © wrightstudio | *https://de.123rf.com*
S. 12: Bild © Anastasiia Nevestenko | *https://de.fotolia.com*
S. 20: Bild © Andrei Krauchuk | *https://de.123rf.com*
S. 26: Bild © PYRAMIS | *https://de.fotolia.com*
S. 32: Bild © Oliver Böhm
S. 34: Bild © vectorpocket | *https://de.freepik.com*
S. 38: Bild © Chris Titze Imaging | *https://de.fotolia.com*
S. 43: Bild © *www.humaaans.com*
S. 49: Bild © Sebastien Decoret | *https://de.123rf.com*
S. 53: Bild © sashkin7 | *https://de.123rf.com*
S. 58: Bild © BillionPhotos.com | *https://de.fotolia.com*

Anzeigen:
Simone Fischer, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

Mediadaten und Preise unter:
www.doag.org/go/mediadaten

Druck:
adame Advertising and Media GmbH,
www.adame.de

Inserentenverzeichnis

Early Bird
bis 21. Januar 2020

JavaLand 2020

17. - 19. März 2020 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Red Hat
OpenShift

The Kubernetes platform for big ideas

Interactive Learning Portal

https://learn.openshift.com

Launch a pre-configured OpenShift instance with an integrated command-line interface using your web browser!

Our guided training scenarios will help you experiment and learn by solving real-world problems using Kubernetes and other advanced, container-centric tooling.

Red Hat

openshift.com