Ausgabe 03/2019

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

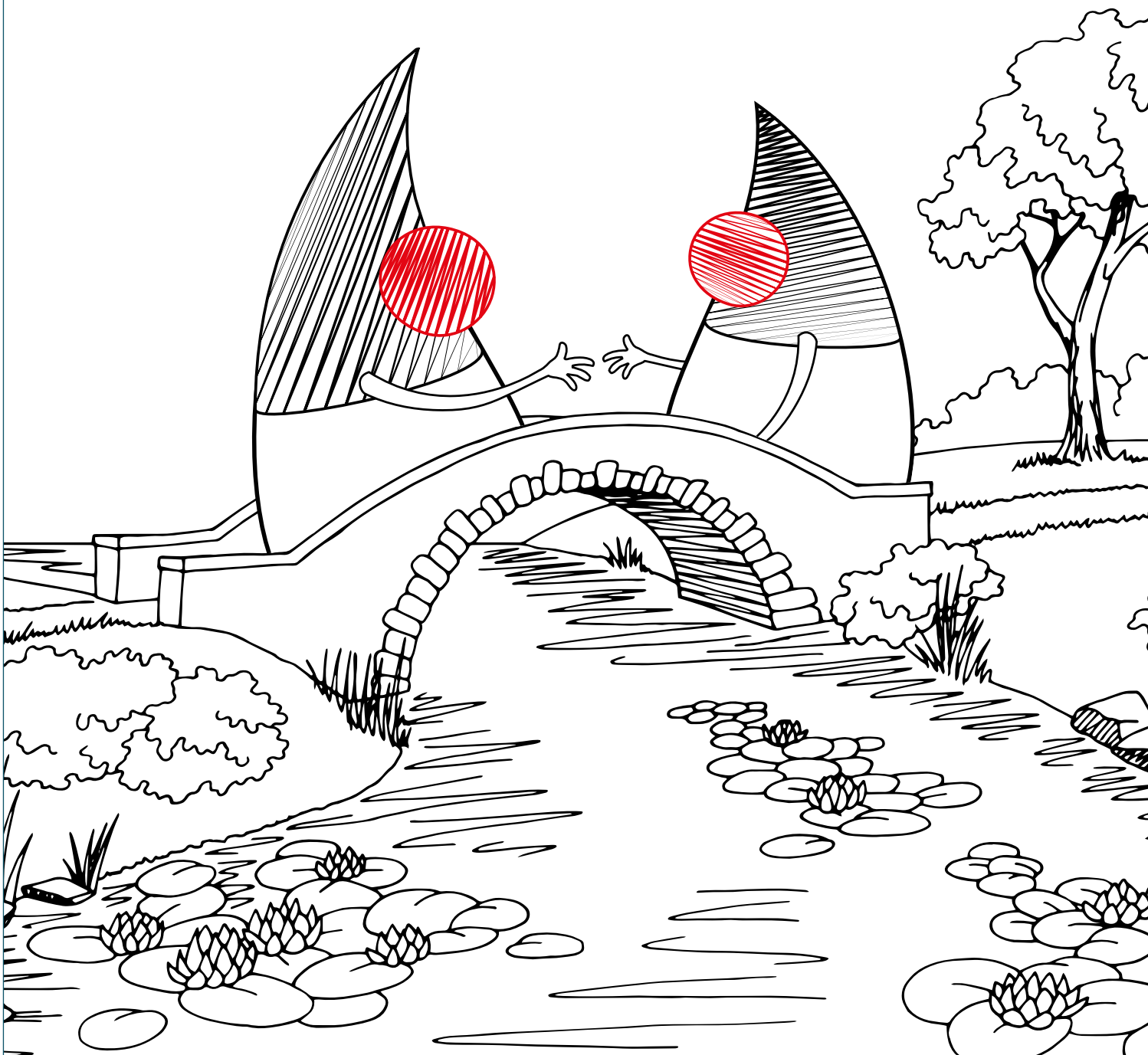
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 03

Java aktuell

iJUG Verbund
www.ijug.eu

Java verbindet



Donnerstag
4. Juli 2019

Kultur- und Kongresszentrum
Liederhalle Stuttgart

Besucheranmeldung
seit 14. April 2019
www.java-forum-stuttgart.de

JAVA FORUM stuttgart 2019

Donnerstag, 4. Juli 2019
JFS - Java Forum Stuttgart

Mit nun 22 Jahren Tradition ist das JFS zur festen Institution geworden. Das Forum bietet den Teilnehmern die Möglichkeit, sich umfassend über Themen zu Java bzw. im JVM-Umfeld zu informieren. Die Breite wird erreicht durch Grundlagenvorträge, Erfahrungsberichte und Informationen über konkrete Projekte. Produkte werden sowohl in Form von Vorträgen als auch als Demonstrationen an den Ausstellungsständen präsentiert.

Mittwoch, 3. Juli 2019
Workshop „Java für Entscheider“
Eintägige Überblicksveranstaltung für Entscheider aus der IT oder IT-nahen Einsatzfeldern wie Abteilungs- oder Teamleiter, deren Mitarbeiter im Java Umfeld tätig sind oder sein sollen.

Möchten Sie beim JFS 2020 einen interessanten Fachvortrag einreichen? Dann melden Sie sich bitte rechtzeitig bei uns!

ca. 40 Aussteller

Dieses Jahr waren wir bereits mit den Ständen Ende Januar ausgebucht.

max. 48 Fachvorträge
& zusätzlich einige
„Pecha-Kucha“ Kurz-Vorträge

NOVATEC avono MIMACOM Digital. Innovation. Engineers.

Eine Veranstaltung der
Java User Group Stuttgart e.V.

JUG STUTTGART JAVA USER GROUP

Hallo, Java aktuell

Wie in der vergangenen Ausgabe bereits angekündigt, tritt Wolfgang Taschner im März 2019 in den wohlverdienten Ruhestand. Wir bedanken uns herzlich für die langjährige Zusammenarbeit und Unterstützung und wünschen ihm eine erholsame Zeit.

Ich freue mich sehr, die Redaktionsleitung der „Java aktuell" für ihn zu übernehmen und mit einer großartigen Community zu vielen interessanten Themen rund um Java zusammenarbeiten zu dürfen – eine spannende aber auch anspruchsvolle Aufgabe, die mit dieser Ausgabe beginnt. Ein wichtiger Umbruch ist auch bei Java zu verzeichnen: Die Änderungen der Lizenzregelung für Java und JDK durch Oracle sorgt bei vielen Nutzern für Verunsicherung. Michael Paege, OPITZ Consulting, klärt in seinem Artikel detailliert darüber auf, wie Entwickler darauf reagieren können und zeigt Handlungsalternativen auf. Zum zehnjährigen Jubiläum des iJUG findet ihr von Stefan Koospal, Mitglied des iJUG-Vorstands, unter dem Motto dieser Ausgabe "Java verbindet" interessante Informationen zur Vereinsgeschichte. Weiterhin erwarten euch Artikel zu aktuellen Trends aus dem Java-Themenkosmos, wie beispielsweise zum Projekt Helidon, Axon und CQRS. Ich wünsche euch viel Spaß beim Lesen und freue mich auf die zukünftige Zusammenarbeit mit vielen Autoren und Java-Fans!

Ihre

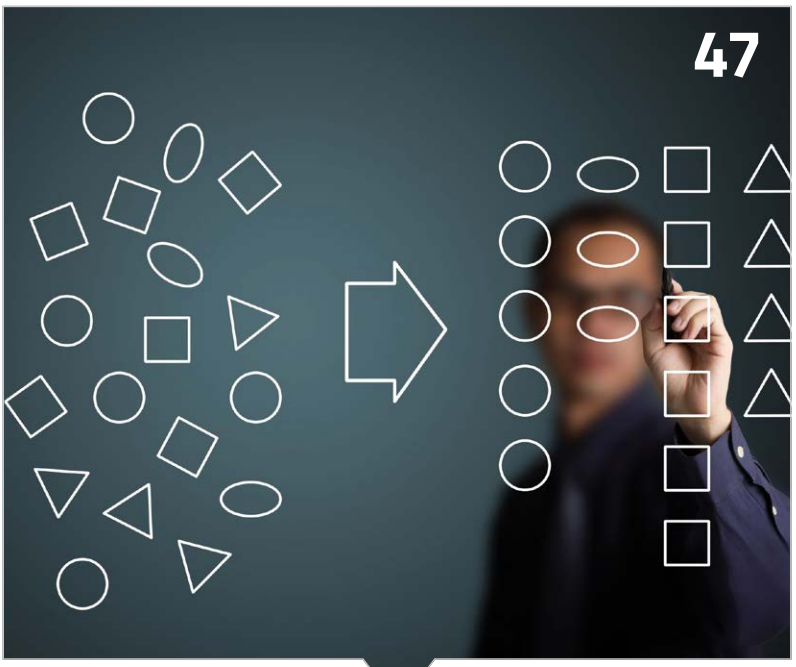
Lisa Damerow

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Das Ende kostenfreier Java-Updates – Was Entwickler jetzt tun können

Microservices mit dem Oracle-Open-Source-Projekt Helidon



Natürliche Ordnung von Java-Objekten

Mobile Apps mit Multi-OS Engine erstellen

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

14. Dezember 2018

WildFly 15 und Java 11

WildFly hat inzwischen einen vierteljährlichen Release Train. Das Hauptthema der ersten drei Releases im Jahr 2018 war der vollständige und – seit WildFly 14 – offizielle Java EE 8 Support, diesmal stand der Sprung in der SE-Version im Vordergrund. Das neue Release 15 unterstützt weiterhin Java 8 und wurde laut Red Hat ausgiebig darauf getestet. Die „Zwischen-Releases" 9 und 10 wurden ebenfalls getestet. Der Fokus war jedoch darauf gerichtet, WildFly für das erste neue Long-Term-Support-Release – Java 11 – vorzubereiten. Die Modularisierung seit SE 9 („Jigsaw") habe dabei großen Einfluss auf die interne Funktionalität gehabt (Classloading, Reflection). Wirklich praktische Auswirkungen für Nutzer von WildFly hat sie noch nicht – solange es keine EE-Spezifikation gibt, in der sie genutzt wird.

19. Dezember 2018

Red Hat: Kommerzieller OpenJDK-Support für Windows

Red Hat erweitert seinen kommerziellen Support für das OpenJDK auf Microsoft Windows. Bislang lediglich das hauseigene Enterprise Linux unterstützt, allerdings verlangen viele Kunden nach Betriebssystem-übergreifenden Angeboten.

27. Dezember 2018

Apache NetBeans 10

NetBeans 10 wurde veröffentlicht. Auch wenn sich am Inkubator-Status des Projekts bei der Apache Software Foundation noch nichts geändert hat, liefert es doch ein knappes halbes Jahr nach Release 9 die nächste Version. Sie enthält Unterstützung für Java 11 und JUnit 5 sowie für andere Sprachen, die in 9 ausgeklammert wurden (PHP, JavaScript, Groovy). Im März soll bereits Version 11 herauskommen und sechs Monate später 12, jeweils mit Unterstützung des aktuellen JDK mit der um eins höheren Versionsnummer.

10. Januar 2019

Spring Cloud Stream und Functions

Bei Spring wird weiter mit Hochdruck an verbesserter Microservice- und „Serverless"-Unterstützung gearbeitet. Vor Kurzem wurde Spring Cloud Functions 2.0.0 freigegeben, unter anderem mit Verbesserungen für die Nutzung von Azure Functions und Kotlin-Support. Ebenso wurde Spring Cloud Stream 2.1.0, Release-Name „Fishtown", freigegeben. Das Framework für eventgetriebene Microservices bietet jetzt das Programmiermodell von Cloud Functions 2.0.0 als Alternative. Dazu wurde die unterstützte Spring-Boot-Version auf 2.1.x angehoben und es gibt eine Reihe von Verbesserungen, insbesondere für die RabbitMQ- und Kafka-Anbindung.

11. Januar 2019

Simon Ritters Java-Kristallkugel für 2019

Simon Ritter schaut wieder in seine Kristallkugel, um das Jahr 2019 für Java vorherzusagen. Hier eine Zusammenfassung dessen, was er gesehen hat:

Die Anzahl von Embedded Applications auf Java werde trotz fehlender Unterstützung durch Oracle weiterwachsen, weil andere Firmen in die Bresche springen, inzwischen kostengünstige Geräte mit ausreichend Speicher verfügbar sind und das Plattform-Modulsystem maßgeschneiderte Runtimes ermöglicht. Java ME sei Javas „Norwegian Blue" (der „mausetote" Papagei aus dem Monty-Python-Sketch). Im Bereich Enterprise Java werde die Eclipse Foundation noch eine Weile mit der Ausarbeitung des neuen Jakarta-EE-Spezifikationsprozesses als Ersatz für den JCP beschäftigt sein; hier also wenig wirkliche Weiterentwicklung – mit Ausnahme von Ansätzen für Microservices, insbesondere MicroProfile. Bei Java SE habe sich der Release Train schnell stabilisiert und es werde zwei pünktliche Releases geben, allerdings ohne große neue Features (Raw String Literals dürften es auch nicht in Release 13 schaffen), dafür mit Vorbereitungen für Valhalla („Value Types", „Generics over Primitive Types") und Loom („high-throughput lightweight concurrency"). Der JCP, auf die Spezifikation von Java SE reduziert, wovon wiederum einiges im OpenJDK abläuft, bleibe eben für SE wichtig; nur die Anzahl der direkt im JCP involvierten Firmen werde sich verringern. Aufgrund der Vielzahl von OpenJDK-basierten Distributionen und der Tatsache, dass das Oracle JDK für den Einsatz in Produktion kostenpflichtig ist, sollten sich die meisten Java-Nutzer bis Ende des Jahres vom Oracle JDK wegbewegt haben.

Bei Letzterem steht Simon mit Azul in direktem Wettbewerb mit Oracle, also könnte hier seine Glaskugel auch eine deutliche Verzerrung aufweisen – aber eine große Zahl von Anwendern wird voraussichtlich auf kostenlose Varianten setzen. Hier passt eine andere Zahl ins Bild, die vom AdoptOpenJDK-Projekt getwittert wurde: Nach einer Million Downloads, verkündet im Oktober 2018, sind jetzt die sechs Millionen fast erreicht – und die Liste wächst.

18. Januar 2019

Der Vorteil von Open Source TCKs

Die Jakarta EE TCKs sind noch nicht fertig, aber schon länger frei verfügbar *[2]*, und jeder kann damit „herumspielen". Wie zum Beispiel ein Apache-Mitstreiter, der es auf TomEE loslässt und auf Twitter darüber schreibt. Auch wenn es noch dauern wird, bis TomEE offiziell Jakarta-EE-zertifiziert ist – die unkomplizierte Form der Zusammenarbeit ist für alle Seiten positiv. Oder, wie Eclipse-Foundation-Chef Mike Milinkovich es in seiner Antwort auf Twitter ausdrückt: „Wir haben es schon eine Weile gesagt: Von der Verfügbarkeit des TCK als Open Source wird die Jakarta EE Community stark profitieren."

23. Januar 2019

JBoss 7.2 und EE 8

JBoss EAP, ist nun Java-EE-8-zertifiziert. Version 7.1 liegt ein Jahr zurück, entsprechend hat sich einiges getan. Von MicroProfile werden ebenfalls einige APIs unterstützt: REST Client, Config, OpenTracing und Health.

31. Januar 2019

GlassFish und Jakarta EE

GlassFish 5.1 ist da, genauer: Eclipse GlassFish 5.1. Die neue Version ist die erste, die vollständig bei der Eclipse Foundation liegt. Java-EE-8-zertifiziert ist es, die Zertifizierung für Jakarta EE 8 wird hoffentlich in naher Zukunft folgen. Von Payara, die den gleichnamigen GlassFish-Fork mit kommerziellem Support liefern und massiv zum Projekt beigetragen haben, heißt es: „Ein weiterer wichtiger Schritt ist getan, um Jakarta EE 9 zu starten."

1. Februar 2019

Amazon Corretto

Corretto 8, Amazons kostenlose Variante des OpenJDK 8, ist jetzt „Generally Available", so verkündet es Arun Gupta im AWS-Blog. Die aktuelle Version 1.8.0_202 kann heruntergeladen werden *[3]* und unterstützt neben Amazon Linux verschiedene andere Linux-Varianten sowie neuere Windows- und MacOS-Versionen. Daneben steht sie auch als Image auf DockerHub zur Verfügung. Amazon nutzt Corretto laut Arun Guptas Blog-Eintrag für „Tausende Production Services" (ob das für Amazon ein signifikanter Anteil ist, wird nicht gesagt). Kostenlose Sicherheitsupdates soll es bis mindestens Juni 2023 geben. Eine weitere Version, Corretto 11 (entsprechend dem LTS Release OpenJDK 11) soll noch vor April 2019 freigegeben werden.

5. Februar 2019

Jakarta Blogs

Für Jakarta EE gibt es einen neuen Blog-Aggregator *[4]*. Jede(r) kann ein eigenes Blog per GitHub Issue hinzufügen, die Bedingungen und Anleitung stehen hier: *[5]*.

7. Februar 2019

MicroProfile wird groß

MicroProfile ist inzwischen groß, was die Anzahl der einzelnen Spezifikationen angeht. Bevor es eine ähnliche Wahrnehmung ereilt wie Java EE, nämlich wenig „micro", sondern „big & heavy" zu sein, hat John Clingan eine Diskussion darüber angeregt, wie dem vorgebeugt werden kann. Die Diskussion ist offen für alle *[6]* und verspricht, interessant zu werden.

7. Februar 2019

MicroProfile Starter ist in der Beta

MicroProfile Starter ist unter *start.microprofile.io* jetzt als Beta-Version verfügbar. Man kann ein MicroProfile-Projekt mit allen Dependencies und Beispielen generieren und herunterladen – für verschiedene Versionen des Frameworks und verschiedene Server. Bislang werden nur Build-Definitionen für Maven generiert, was jedoch aufgrund der vorherrschenden Stellung in diesem Segment kein wesentliches Hindernis sein dürfte – sorry, Gradle-Fans, habt Geduld.

10. Februar 2019

MicroProfile 2.2 ist da

Wir beenden den MicroProfile-Lauf mit einem kurzen Update: Version 2.2 ist da (basierend auf Java EE 8 wie alle 2.x Releases), mit einigen Updates der APIs: Enthalten sind jetzt die Versionen Fault Tolerance 2.0, OpenAPI 1.1, OpenTracing 1.3 und Rest Client 1.2.

Referenzen

[1] *https://dzone.com/articles/staring-into-my-java-crystal-ball-2019*
[2] *github.com/eclipse-ee4j/jakartaee-tck*
[3] *aws.amazon.com/corretto*
[4] *jakartablogs.ee*
[5] *https://github.com/jakartaee/jakartablogs.ee*
[6] *https://groups.google.com/forum/#!searchin/microprofile/microprofile$20growing$20pains%7Csort:date/microprofile/bxKDRVr-LXHI/ZlO0QsZ7BQAJ*

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

Endlich ist GlassFish 5.1 da, alle freuen sich und sind aus dem Häuschen – oder doch nicht? Jedenfalls hat die Eclipse Foundation – halt, sagen wir es doch ganz offen – jedenfalls hat Oracle es nun doch endlich mal geschafft, die gleiche Software durch die Tür zu bringen, die sie schon vor über anderthalb Jahren veröffentlicht hat. Juhu! Da kommt doch mal Partylaune auf! Und was sollen wir nun damit? Ja, es ist keine Vaporware – die Software kann tatsächlich downgeloadet werden *[1]*. Dass das keiner an die große Glocke hängt und man den Link kaum findet, hat einen Grund: Es ist nämlich die falsche. Ja, richtig gelesen: Oracle hat es doch tatsächlich dank des, umschreiben wir es mal vorsichtig mit dem Wort „seltsamen", Staging-Prozesses geschafft, die falsche Software zu veröffentlichen! „LOL", wie man das online kommentieren würde! GlassFish 5.1 ist de-facto nicht die Menge all der ganzen Teile, die in monatelanger Kleinarbeit „staged", geprüft, „released" und letztendlich auf Maven Central *[2]* veröffentlicht wurden. Nein, es ist irgendein unbedeutender, veralteter Zwischenstand aus dem vierten Quartal 2018, dessen Download sich nicht lohnt. Wieso das passiert ist, konnte mir bislang niemand sagen, jedoch wurde mir eine Person aus dem Hause Oracle benannt (ich möchte betonen: eine), die wohl beauftragt ist, das zu fixen. Einen Termin konnte man mir nicht nennen (das ist, nach der Erfahrung der vergangenen Monate, wohl auch besser so).

Was tut eigentlich der Rest der Eclipse Foundation? Die EF selbst kämpft noch mit ihrer überlasteten Infrastruktur und streitet sich nach wie vor mit Oracle um die Nutzungsrechte der Namen „Java", „JAX-RS", etc. Stand der Dinge ist hier, dass Oracle es wohl nicht erlauben will und alles daher einen neuen Namen erhält. Wir „freuen" uns schon darauf, dass es nun also bald nicht nur „Jakarta EE" statt „Java EE" heißt, sondern auch „Jakarta API for RESTful WebServices" – denn das Akronym bleibt vermutlich tabu! Dass dies implizit auch bedeutet, dass es keine Java-Standards mehr sind, sondern nur noch vereinsinterne Eclipse-APIs, möchte ich mal ganz am Rande erwähnen. Mag sein, dass dies vielen egal ist, vielen aber eben auch nicht! Neben Oracle sind Red Hat, IBM, Payara und weitere Stakeholder. Zumindest namentlich. Inhaltlich habe ich nicht viel entdecken können, was diese Firmen unter dem Jakarta-Dach an echter Code Contribution geleistet hätten, also von regelmäßigem Marketing und Selbstorganisation abgesehen. Schade eigentlich, wo doch alle unbedingt mehr Einfluss auf Java EE gefordert hatten!

Also alles Mist? Nein, bei Weitem nicht! Sobald Oracle seine Handvoll involvierter Mitarbeiter dank des hoffentlich finalen GlassFish 5.1.1 entlastet hat, können diese endlich den Suspend Mode in den EE4J-Projekten beenden und sich den angestauten Pull Requests *[4]* widmen. Das heißt ganz konkret, dass dann beispielsweise JAX-RS 2.2 oder Jersey 2.29 endlich wieder Fahrt aufnehmen können und wir uns auf der JavaLand über eine ganze Reihe neuer Features und veröffentlichter Bug Fixes unterhalten können, die eigentlich schon fertig, aber eben nicht integriert sind. Da ist einiges in der Pipe *[3]*, das wirklich Partylaune aufkommen lässt (etwa die Transition von angestaubten Application-Servern auf moderne Architekturen), und darauf lohnt es sich auch noch ein paar Wochen mehr zu warten! Wem es nicht schnell genug geht, der sei auch bei dieser Ausgabe aufgefordert, sich an den zahlreichen Open-Source-Projekten zu beteiligen – jeder ist herzlich willkommen!

Referenzen

[1] *https://www.eclipse.org/downloads/download.php?file=/glassfish/glassfish-5.1.0.zip*
[2] *https://search.maven.org/search?q=g:jakarta.**
[3] *https://github.com/eclipse-ee4j/jaxrs-api/wiki/Roadmap*
[4] *https://github.com/eclipse-ee4j/jersey/pulls*

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.

Java: Was ändert sich? Risiken und Handlungsalternativen für Java-Nutzer

Michael Paege, Opitz Consulting Deutschland GmbH

Den 31. Januar 2019 haben sich manche Java-Nutzer schon lange rot in ihren Kalendern angestrichen: Zu diesem Stichtag, so hatte Oracle es im März 2018 angekündigt, wird der Hersteller Updates für das Oracle Java Development Kit 8 (JDK 8) nicht mehr kostenfrei zur Verfügung zu stellen. Wer also ab Februar 2019 weiterhin neue Updates und Security Patches für das Oracle JDK 8 braucht oder haben möchte, benötigt die kostenpflichtige Java SE Subscription. Im Zusammenhang mit der Entscheidung von Oracle, JDK 11 kostenfrei nur noch für Development, Test, Proof of Concept und Demo zur Verfügung zu stellen, und zusammen mit der Entscheidung des Java Community Process (JCP), die generelle Release-Kadenz von drei Jahren auf sechs Monate zu verkürzen, bedeutet dies gewaltige Veränderungen für Java-Kunden. Zudem trifft es mit Java ein Umfeld, das in der Vergangenheit aufgrund der freien Verfügbarkeit nicht, beziehungsweise kaum, gemanagt werden musste und wurde.

Hintergrund:

Java ist eine Programmiersprache, bestehend aus einer Entwicklungsumgebung (Java Development Kit, kurz: JDK) und einer Laufzeitumgebung (Java Runtime Environment, kurz: JRE), die 1995 von Sun Microsystems eingeführt wurde. Nachfolgend verwende ich nur noch die Kurzform JDK, was das JRE immer mit einschließt.

Aufgrund der Tatsache, dass Laufzeitumgebungen für nahezu jegliche Plattformen entwickelt wurden, von der Smartcard über mobile Endgeräte bis hin zu Servern, erfreute sich Java schon bald großer Beliebtheit

und ist heute entsprechend weit verbreitet. Im Jahr 2007 veröffentlichte Sun den Java Source Code und stellte ihn unter Open-Source-Lizenz, geboren war das „Open JDK". Aus diesem OpenJDK hat Oracle nach der Übernahme von Sun ein eigenes JDK, das Oracle JDK, entwickelt. Aus dem OpenJDK wurden weitere Forks abgeleitet und beispielsweise gemeinsam mit Linux-Distributionen mitgeliefert oder als eigenständige Distributionen kostenpflichtig (zum Beispiel Azul Zulu) oder kostenfrei (zum Beispiel Adopt OpenJDK) angeboten *(siehe Abbildung 1)*.

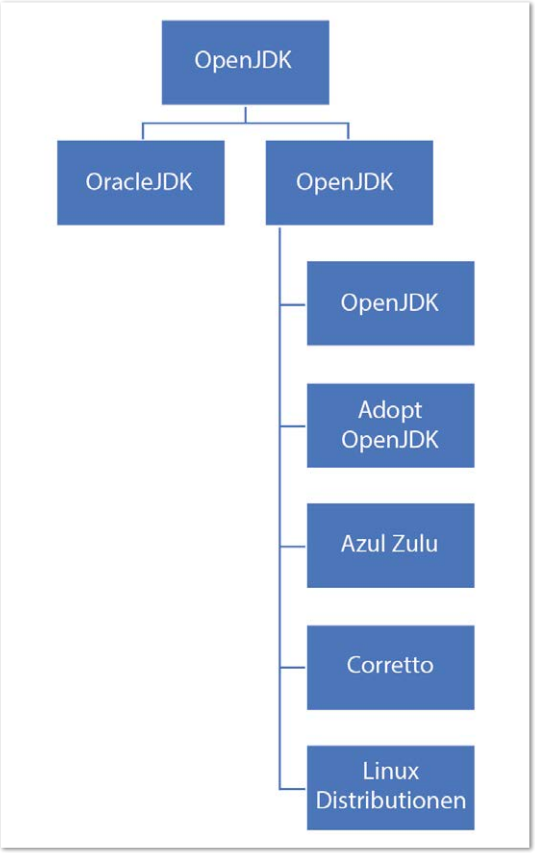


Abbildung 1: JDK-Distributionen

Das Oracle JDK wurde später um zusätzliche Features angereichert, wie Flight Recorder, Mission Control oder den Enterprise Installer, deren Nutzung auch in der Vergangenheit bereits lizenzpflichtig war. Wollten Kunden diese Features nutzen, mussten sie die Produkte Java SE Desktop für den Desktop-Einsatz oder Java SE Advanced für den Server-Einsatz lizenzieren.

Oracle JDK und OpenJDK

Für alle Java-Distributionen gilt: Sie müssen dem vom Java Community Process (JCP) festgelegten Standard entsprechen. Die Kompatibilität wird über das Technology Compatibility Kit (TCK) geprüft. Daher sollten Oracle JDK und OpenJDK bezüglich der Standardfunktionalitäten austauschbar sein.

Generelle Änderungen im Java-Standard: Ein Überblick

Welche Funktionalitäten sind abgekündigt?
Generell ist der Einsatz von Java auf dem Client bei zukünftigen Versionen als nicht mehr zukunftssicher zu betrachten. Java 8 (sowohl OpenJDK als auch Oracle JDK) ist die letzte Version, die die Features Java Applets und Servlets, Web Start und JavaFX-Graphics-Bibliothek unterstützt. Wer also eines dieser Features aktuell einsetzt und aus welchen Gründen auch immer weiterhin einsetzen muss, ist gezwungen, weiterhin Java 8 einzusetzen.

Welche Änderungen sind bezüglich der Release-Kadenz zu erwarten?
Um die Attraktivität von Java als Entwicklungsumgebung zukünftig weiter zu fördern, beschloss das JCP, die Release-Kadenz von drei Jahren auf sechs Monate zu verkürzen. Das bedeutet, dass nach Java 8 alle sechs Monate ein neues Java Major Release veröffentlicht

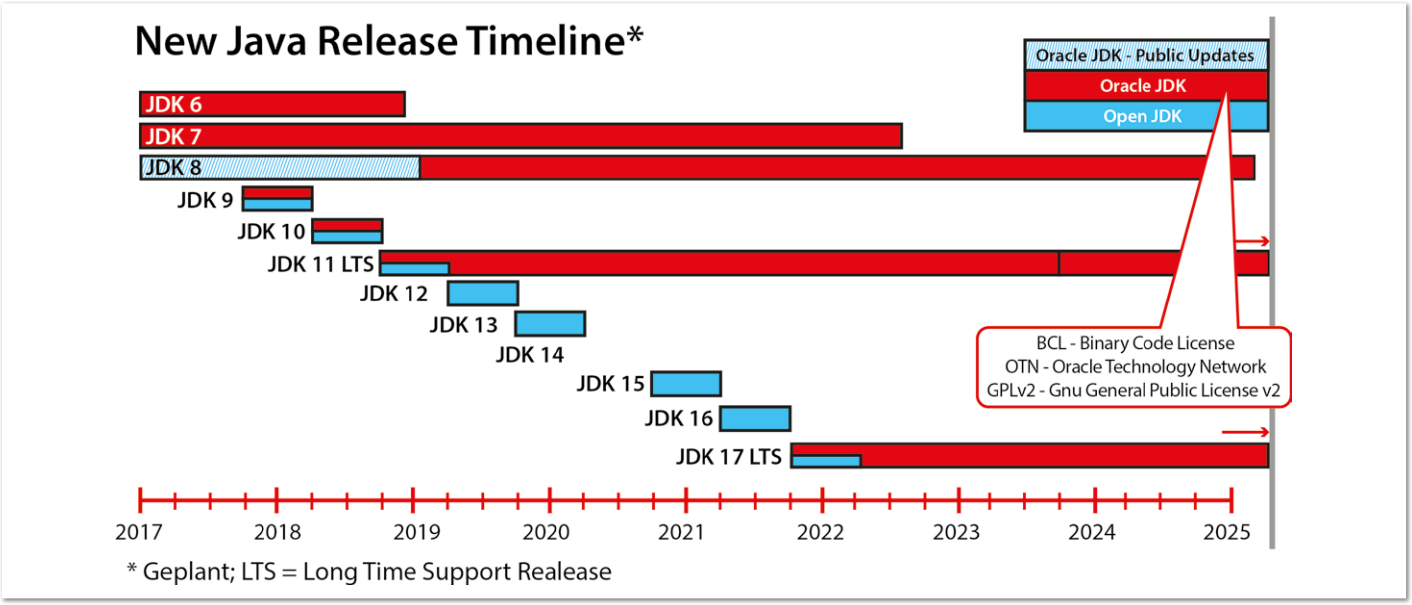


Abbildung 2: Java Roadmap/Release Timeline

wird. Man will damit erreichen, dass Release-Wechsel leichter vonstattengehen, weil die Änderungen zwischen zwei Releases weniger umfassend sind als bei größeren Zeitabständen. Zudem möchte man in der Lage sein, den Entwicklern neue Features so früh wie möglich zur Verfügung zu stellen. Das bedeutet aber, dass Unternehmen nach sechs Monaten keine Updates und Security Patches mehr für ein Major Release bekommen, sondern sie müssen auf das neue Release migrieren. Da dies aber in produktiven Umgebungen aus Test- und Deployment-Gründen in der Regel nicht durchführbar ist, wird es Java 8, Java 11 und zukünftig Java 17 als sogenannte Long Time Support Releases (LTS) von bestimmten Distributoren geben, zum Beispiel Oracle JDK (kostenpflichtig), Adopt OpenJDK (kostenfrei), Azul Zulu (kostenpflichtig), und auch Amazon hat mit Corretto eine kostenfreie Distribution mit LTS angekündigt *(siehe Abbildung 2)*.

Welche Änderungen gibt es in Bezug auf das Oracle JDK?

Am 25. März 2018 hat Oracle angekündigt, Updates und Security Patches für Oracle JDK 8, das aktuelle Release mit LTS und – wie oben schon ausgeführt – das Release, auf dem viele Kunden wegen danach wegfallender Features noch eine Zeitlang werden bleiben müssen, nach Januar 2019 nicht mehr kostenfrei zur Verfügung zu stellen. Im November hat Oracle hierzu noch eine Unterscheidung hinzugefügt: Für private Nutzung werden Updates und Security Patches für Oracle JDK 8 noch bis zum 31. Dezember 2020 kostenfrei zur Verfügung gestellt, für kommerzielle Nutzer des Oracle JDK 8 bleibt es bei der Entscheidung, dass es keine kostenfreien Updates und Security Patches nach dem 31. Januar 2019 mehr geben wird. Darüber hinaus hat Oracle angekündigt, ab Oracle JDK 11 kein separates JRE mehr zur Verfügung zu stellen, sondern das JRE nur noch zusammen mit dem JDK verfügbar zu machen. Entsprechend dem OTN-Lizenzvertrag für Java SE *[1]*, den man beim Download von Oracle JDK 11 akzeptieren muss, stellt Oracle das Oracle JDK 11 nur noch für Entwicklung, Test, Demo und Proof of Concept kostenfrei zur Verfügung. Dies wird vermutlich auch für folgende Releases gelten.

Wen betreffen die Änderungen und wen nicht?

Zunächst die guten Nachrichten: Nicht betroffen von der Notwendigkeit, für die Nutzung von Oracle JDK 8 und Oracle JDK 11 die Subscription zu beziehen und sich Gedanken über eine Migration in Richtung OpenJDK machen zu müssen, sind Nutzer, die Oracle-Produkte lizenziert und unter Support haben, die Java SE enthalten. Das gilt für die Produkte WebLogic Standard Edition, WebLogic EE, WebLogic Suite, Internet Application Server Enterprise Edition, Oracle Forms, GlassFish Server, Coherence Standard Edition, Coherence EE, Coherence Grid Edition, WebCenter Content und WebCenter Universal Content Management. Bei diesen ist die Nutzung des Oracle JDK also auch zukünftig durch den Support des darüberliegenden Produktes mit abgedeckt, allerdings beschränkt sich diese auf die Benutzung des darüberliegenden Produktes. Das heißt, Oracle JDK 11 dürfte beispielsweise produktiv für die Nutzung des WebLogic Servers Standard Edition verwendet werden, aber nicht für die Nutzung auf einem anderen Application Server auf diesem Server. Unklar ist aktuell noch, wie es sich mit kostenfreien und damit auch

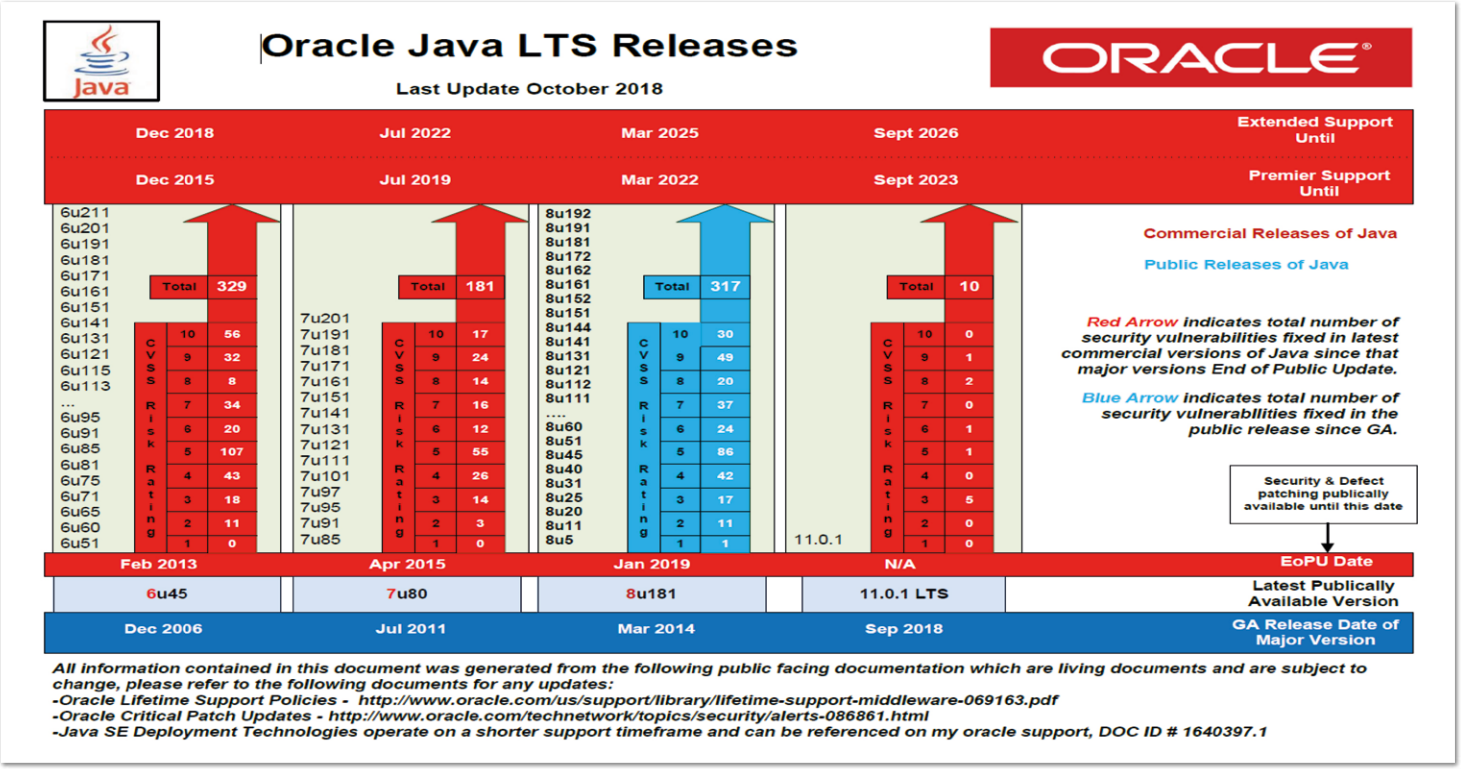


Abbildung 3: Anzahl der entdeckten Vulnerabilities unter Angabe der jeweiligen Severity für Java 6, Java 7 und Java 8 im Zeitraum des jeweiligen Releases.

supportfreien Oracle-Produkten verhält, die Java nutzen, wie der Oracle Database XE und dem SQL Developer. Entsprechende Fragen an Oracle blieben bis dato unbeantwortet.

Ebenfalls nicht betroffen sind Nutzer anderer Produkte, die Java SE (also Oracle JDK) mit lizenziert haben, wie IBM WebSphere und SAP NetWeaver.

Auch Nutzer, die bereits Java SE lizenziert haben, sind von den Änderungen nicht betroffen, weil sie zum Beispiel eines oder mehrere der kostenpflichtigen Features nutzen.

Kunden, die eine ältere Java-Version wie zum Beispiel Java 5, Java 6 oder Java 7 einsetzen, sind von den Änderungen nicht akut betroffen. Diese Kunden gehen mit der Nutzung der Altversionen auf den entsprechenden Servern allerdings ein Sicherheitsrisiko ein, weil es für diese Versionen schon seit Jahren keine Security Patches mehr gibt.

Und nun die nicht so guten Nachrichten: Von den Änderungen sind alle Kunden betroffen, die Oracle JDK nutzen und weiterhin nutzen müssen oder möchten und die Java als eigenständige Installation auf Client, Server oder in Containern nutzen, die eine stabile Major-Version nutzen müssen und/oder weiterhin Sicherheitsupdates haben möchten.

Wer das Oracle JDK 11 und seine Folgeversionen produktiv nutzen möchte, ist ebenfalls betroffen.

Auch die Nutzer von „reinem" OpenJDK sind betroffen, weil sich hier die Release-Kadenz ändert. Das heißt, um aktuelle Updates und Security Patches zu bekommen, müssten sie alle sechs Monate auf ein neues Major Release wechseln.

Handlungsalternativen

Anwendern, die von den Änderungen betroffen sind, bleiben mehrere Optionen:

- Nichts tun und Oracle JDK 8 ohne Updates weiterbetreiben
- Die Java SE Subscription beziehen
- Auf OpenJDK umsteigen

Nachfolgend werde ich die einzelnen Handlungsalternativen weiter beleuchten.

Nichts tun und Oracle JDK 8 ohne Updates weiter betreiben

Dies ist zunächst erst mal die am wenigsten aufwendige und zunächst kostensparendste Lösung. Sie birgt jedoch ein Sicherheitsrisiko, das jeder Kunde für sich, seine Architektur und seine entsprechenden weiteren Schutzmaßnahmen selbst bewerten muss. Grundsätzlich gilt aber Java, auch aufgrund seiner Verbreitung, als eine hoch gefährdete Umgebung, insbesondere, wenn es – beispielsweise durch einen Browser-Plug-in – über das Internet erreichbar ist *(siehe [2])*. Die Anzahl der beseitigten Vulnerabilities, vor allem auch die Anzahl der Fälle mit hoher Severity in *Abbildung 3* zeigt, dass man das Security-Thema bei Java nicht unterschätzen sollte.

Die Java SE Subscription beziehen

Wer weiterhin Oracle JDK 8 mit aktuellen Security Patches oder Oracle JDK 11 – und damit auch Oracle JRE 11 – für Produktionszwecke nutzen möchte, braucht die Java SE Subscription.

Die bisher lizenzierbaren Produkte „Java SE Advanced", „Java SE Advanced Desktop" und „Java Suite" wurden im Januar 2019 von der Oracle-Preisliste entfernt und sind – laut bisherigen Aussagen von Oracle – nur noch bei Lizenzerweiterungen für Kunden kaufbar. Sie wurden ersetzt durch die „Java SE Subscription" für die Nutzung auf Servern und die „Java SE Desktop Subscription" für die Nutzung auf dem Desktop. Beide Subscriptions sind befristet auf die jeweilige Laufzeit, die typischerweise ein Jahr beträgt. Sie enthalten die Lizenz für die Nutzung der kostenpflichtigen Features sowie von Oracle JDK 11 für Produktionszwecke. Auch der Support ist enthalten, sodass neue Bugs und Probleme über My Oracle Support gemeldet werden können und neue Builds mit Updates und Security Patches zur Verfügung stehen. Achtung, die auf der Oracle Website aufgeführte Java SE Subscription Global Pricelist enthält den Subscription-Preis in US-Dollar und auf monatlicher Basis. Für den ungefähren Jahrespreis ist dieser Wert also mit zwölf zu multiplizieren und in Euro umzurechnen. Die Oracle-Partner verfügen jedoch über eine lokalisierte Preisliste in Euro.

Im Rahmen der Java SE Subscriptions erhält der Kunde für die Long Time Support Releases Oracle JDK 8, Oracle JDK 11 und zukünftig auch Oracle JDK 17 Support im Rahmen der Oracle Lifetime Support Policy. Das heißt: Premier Support für fünf Jahre und danach, wenn gewünscht, Extended Support mit Zusatzkosten für weitere drei Jahre *(Details siehe Tabelle 1)*.

Java-Version	Premier Support bis	Extended Support bis
8	März 2022	März 2025
11	September 2023	September 2026
17	Voraussichtlich Sept. 2026	Voraussichtlich Sept. 2029

Tabelle 1: Die Laufzeiten für den Support von Oracle JDK 8, Oracle JDK 11 und Oracle JDK 17

Die Java SE Subscription ist für den Java-Einsatz auf Servern gedacht. Die Metrik ist „Prozessor" und es werden zur Ermittlung der zu lizenzierenden/subskriptionierenden Anzahl der Prozessoren dieselben Regeln angewendet wie beispielsweise bei der Datenbank Enterprise Edition. Auch die Regelungen beim Einsatz in virtualisierten Architekturen sind dieselben wie bei allen übrigen Oracle-Produkten (vergleiche *[3]*).

Die Java SE Desktop Subscription ist für den Java-Einsatz auf Desktop- oder Laptop-Computern gedacht. Die Metrik ist „Named User Plus" und auch hier werden zur Ermittlung der zu lizenzierenden/

subskriptionierenden Anzahl der User dieselben Regeln angewendet wie bei den übrigen Oracle-Produkten.
Zu zählen sind also

- **alle berechtigten Personen,** die auf Java zugreifen können, also nicht nur diejenigen, die es zu einem bestimmten Zeitpunkt tatsächlich tun, sondern alle, die es tun könnten. Eine Applikation zur Essensbestellung in der Kantine ist also theoretisch von allen Mitarbeitern benutzbar.
- **sowie alle nicht-benutzerbedienten Geräte** (wie z.B. Temperaturfühler, Kommissionierroboter etc.), die ihre Daten aus der Java-Applikation bekommen oder dort hineinschreiben. Bei einer speicherprogrammierbaren Steuerung (SPS) muss jeder Sensor oder Motor als separater NUP gezählt werden, da die SPS selbst die Daten nur bündelt, also als Multiplexer arbeitet. Dazu mehr unter dem nächsten Punkt.
- **und gezählt wird am Multiplexing-Frontend** Als Multiplexer wertet Oracle nicht nur technische Multiplexer wie beispielsweise Application Server, Transaktionsmonitore oder Connection Pooling, sondern gegebenenfalls auch Datentransfers von und zu anderen (internen oder externen) Systemen.

Es reicht also nicht, alle Java-nutzenden PCs und Laptops zu zählen, sondern es müssen alle berechtigten Personen gezählt werden, nicht-benutzerbediente Geräte, und das vor einem etwaigen Multiplexer. Im Unterschied zu anderen Produkten treten die Problematiken des Multiplexings im Java-Umfeld allerdings eher selten auf; auch nicht-benutzerbediente Geräte sind hier nicht die Regel, weil PCs und Laptops typischerweise vor allem von Menschen bedient werden.

Die Subskriptionierung von Java SE in einer Farm für Desktop-Emulation (beispielsweise Citrix XenDesktop) muss laut Oracle durch das Produkt „Java SE Subscription" in der Metrik „Prozessor" erfolgen und nicht als Desktop Subscription in der Metrik „Named User Plus".

Auf OpenJDK umsteigen

Will man weiterhin aktuelle Patches für Java 8 haben oder Java 11 produktiv einsetzen, ohne jedoch die Java SE Subscription von Oracle zu beziehen, so bleibt als Möglichkeit der Umstieg auf OpenJDK. Bei OpenJDK handelt es sich, wie zu Beginn dieses Artikels bereits ausgeführt, um einen parallelen Entwicklungsstrang der Java JDK. Die Austauschbarkeit der JDKs ist durch die Zertifizierung über das Technology Compatibility Kit gewährleistet, sofern man keine kostenpflichtigen Zusatzfeatures von Oracle verwendet. Wobei Oracle angekündigt hat, die Zusatzfeatures in den nächsten Java Releases ebenfalls Open Source zur Verfügung zu stellen, sodass sie ins OpenJDK integriert werden können.

Beim OpenJDK muss der Kunde nun entscheiden, ob er das „reine" OpenJDK einsetzen möchte oder OpenJDK-Distributionen mit Long Time Support (LTS). Ersteres kann direkt im Internet heruntergeladen werden *(siehe [4])*. Bei dieser Variante muss der Kunde alle sechs Monate einen Wechsel auf das jeweils aktuelle OpenJDK Release mitmachen.

OpenJDK-Distributionen, die mit LTS angeboten werden, können also durchaus von Vorteil sein. Diese gibt es als kostenfreie oder kostenpflichtige Varianten.

Kostenpflichtige OpenJDK-Distributionen mit LTS wären beispielsweise das schon lange angebotene Azul Zulu OpenJDK sowie die mit den kommerziellen Linux-Distributionen ausgelieferten Open JDKs von Red Hat, Suse oder Debian. Diese bieten aktuell für die jeweiligen Java-LTS-Versionen Supportzeiträume über 2023 hinaus an *(siehe Tabelle 2)*.

Distribution (kostenpflichtig)	Java 8	Java 11
Azul Zulu	März 2026	September 2027
Red Hat	Juni 2023	Oktober 2024
Zum Vergleich: Oracle JDK	März 2025	September 2026

Tabelle 2: Supportzeiträume für die kostenpflichtigen Java-LTS-Versionen

Kostenfreie OpenJDK-Distributionen mit LTS sind das von Microsoft und IBM unterstützte Adopt OpenJDK sowie das im November 2018 von Amazon angekündigte Amazon Corretto *(siehe Tabelle 3)*.

Distribution (kostenfrei)	Java 8	Java 11
Adopt OpenJDK	Mind. bis September 2023	Mind. bis September 2022
Amazon Corretto	Mind. bis Juni 2023	Mind. bis August 2024
Oracle	Januar 2019 (Oracle JDK)	März 2019 (OpenJDK von Oracle)

Tabelle 3: Supportzeiträume für die kostenfreien Java-LTS-Versionen

Compliance-Risiken im Überblick

Viele Unternehmen bewegen sich im Java-Umfeld rechtlich auf sehr unsicherem Terrain. Wie wir gesehen haben, machen es die unterschiedlichen Regelungen für verschiedene Produkte und die unterschiedlichen Supportzeiträume für Anwender nicht unbedingt einfacher. Doch welche Risiken ergeben sich in der Praxis und wie können sie vermieden werden?

Compliance-Risiko: Lizenzierung

Die ersten Anfragen zum Thema Java SDK erreichten mich im Sommer 2018 von Lizenzmanagern unserer Kunden. Man war sich unsicher und befürchtete ein Lizenzierungsproblem. Meine Antwort darauf lautete damals: „Eigentlich nicht." – Zumindest war dies nicht durch die Ankündigung von Oracle gegeben, den Support für Oracle JDK 8 nicht mehr kostenfrei zur Verfügung zu stellen. Weitere Builds von Oracle JDK 8 werden nur noch in

My Oracle Support zur Verfügung gestellt, sodass meiner Meinung nach kein Risiko bestand, dort ohne Subscription heranzukommen. Diese Aussage muss ich heute relativieren. Mittlerweile gibt es Äußerungen von Oracle, die Compliance-Risiken im Lizenzumfeld zukünftig einräumen. So können Mitarbeiter noch bis zum 31. Dezember 2020 eine Version des Oracle JDK 8 privat herunterladen und diese in der Firma im kommerziellen Umfeld einspielen. Riskant ist ebenfalls die Verwendung von Oracle JDK 11 in Produktionsumgebungen und natürlich, falls kostenpflichtige Features aus dem Oracle JDK verwendet werden. Aber dieses Thema ist nicht neu, sondern wurde bereits vor gut zwei Jahren (auch in den Medien) thematisiert *siehe [5]*.

Compliance-Risiko: Security
Aus den oben ausgeführten Überlegungen lässt sich ableiten, dass Unternehmen auch ein Security-Risiko eingehen, wenn sie keine aktuellen Updates und Security Patches einspielen. Die in den Java-Umgebungen auch heute noch vorhandenen Security-Lücken könnten von Angreifern genutzt werden. Hier hilft nur, wie auch bei Betriebssystemen, Virenscannern, Browsern und etlichen anderen Systemen, die jeweils aktuellsten Versionen und Builds der JDKs zu verwenden.

Kostenrisiko:
Ein Kostenrisiko ist auf jeden Fall gegeben. Entweder durch Bezug der Java Subscriptions, durch Aufwände der Migration auf OpenJDK oder durch Aufwände zur Beseitigung etwaiger Schäden durch Angriffe auf veraltete Java-Versionen.

Fazit

Unternehmen sind sicherlich gut beraten, die jetzige Situation zu nutzen, um ihre Java-Architekturen, JDKs und Versionen im Unternehmen zu analysieren und für die zukünftige Java-Nutzung sowie für zukünftiges Deployment- und die Patch-Strategie zeitnah die richtigen Weichen zu stellen.

Referenzen

[1] *https://www.oracle.com/technetwork/java/javase/terms/license/javase-license.html*
[2] *https://www.heise.de/download/blog/Wie-sicher-ist-Java-3632920*
[3] *https://www.doag.org/de/home/news/vmware-vsphere-versionen-bis-51-ebenfalls-von-oracles-geaenderten-lizenzbestimmungen-betroffen/detail*, hier: *https://www.doag.org/de/home/news/ein-grosser-schritt-fuer-die-oracle-lizenzierung-in-virtuellen-umgebungen/detail* und hier: *https://www.doag.org/de/home/news/support-ende-vsphere-55-wie-geht-es-weiter/detail*
[4] Download unter *https://openjdk.java.net bzw. https://jdk.java.net*
[5] *https://www.doag.org/de/home/news/java-se-kommerzielle-features-muessen-explizit-aktiviert-werden/detail*

Der Autor verwendete für seine Recherche darüber hinaus diese Quellen:
Svend Back, Techdata: Java SE Präsentation
Rob van der Aar, Oracle: Java SE Präsentation
https://adoptopenjdk.net/support.html
https://aws.amazon.com/de/corretto/faqs/
https://www.azul.com/products/azul_support_roadmap/
https://www.heise.de/developer/meldung/Oracle-Ende-fuer-oeffentliche-Updates-von-Java-8-ab-Februar-2019-4035059.html
https://www.heise.de/developer/artikel/Wird-Java-jetzt-kostenpflichtig-4144533.html?seite=2
https://www.java.com/en/download/release_notice.jsp
https://access.redhat.com/articles/1299013

Michael Paege
michael.paege@opitz-consulting.com

Michael Paege studierte BWL mit Schwerpunkt Wirtschaftsinformatik an der Westfälischen Wilhelms-Universität in Münster. Bereits während des Studiums hat er Anwendungssysteme mit den relationalen Datenbanken Informix und vor allem Oracle entwickelt. Nach dem Studium 1993 begann er bei OPITZ CONSULTING als PL/SQL- und Forms/Reports-Entwickler und übernahm bald Projektleitungsaufgaben. 1999 gründete er die Hamburger Niederlassung von OPITZ CONSULTING, die er bis 2010 leitete. Parallel dazu wurde das Thema Lizenzvertrieb und Lizenzberatung zu einem immer größeren Aufgabenbereich, den er in 2010 zu seinem Hauptaufgabenfeld machte. Ehrenamtlich ist Michael Paege seit langem in der DOAG aktiv und gründete das Competence Center Lizenzfragen, das er bis heute leitet.

Gemeinsam großartige Teams schaffen

gelesen von Michael Fritz , msg systems AG

In Zeiten des digitalen Wandels und steigenden Veränderungsdrucks auf Unternehmen aller Branchen sind Agilität und neue Formen der Zusammenarbeit kein Hype mehr, sondern in vielen Organisationen implementiert oder werden als Unterstützung für Veränderung betrachtet. Die Einführung und der Einsatz agiler Methoden in der digitalen Transformation werden in der Literatur hinreichend behandelt und Scharen von Beratern und Coaches haben sich auf diese Disziplin spezialisiert. Die Autoren Sandy Mamoli und David Mole haben dieses Praxishandbuch bereits im Jahr 2015 in der englischen Originalfassung veröffentlicht; nun steht es auch in der deutschen Erstauflage zur Verfügung.

Dieses Buch wendet sich an Führungskräfte, Entscheider und an Teambildungsprozessen Beteiligte, die sich mit der Umgestaltung von Teams und Prozessen in Unternehmen befassen. Unternehmen, die Produkte entwickeln, stark wachsen und dynamischen Veränderungen unterliegen sowie bereit sind, neue Herangehensweisen zu betrachten, sind eine Zielgruppe. Das Buch kann als „Kochbuch" oder Trainerleitfaden eingesetzt werden.

Im Vorwort geht Esther Derby auf die Vor- und Nachteile herkömmlicher Vorgehensweisen zur Auswahl von Mitarbeitern und auf das Zusammenstellen von Teams ein.

In sieben Kapiteln beschreiben die Autoren entlang eines Praxisbeispiels des neuseeländischen E-Commerce-Anbieters Trade Me, wie sie 2013 einen umfangreichen Self-Selection-Prozess mit 22 Teams und 150 beteiligten Mitarbeitern durchgeführt haben.

Im ersten Kapitel stellen die Autoren einige klassische Herangehensweisen zur Teamzusammenstellung und Mitarbeiterauswahl vor und gehen auf die Chancen und Risiken dieser Vorgehensweisen ein. Anhand von Beispielen und Herausforderungen aus der heutigen Arbeitswelt leiten sie die Motivation für die Verwendung des Self-Selection-Prozesses ab. Sie erklären, warum aus ihrer Sicht in schnell wachsenden Organisationen und bei wechselnden Herausforderungen Selektion durch Führungskräfte scheitert. Die Kapitel zwei und drei widmen sich ausführlich der Vorbereitung und Planung eines Self-Selection-Events. Die Herausforderungen

lagen bei Trade Me in der großen Anzahl der Mitarbeiter, die in den Prozess eingebunden werden sollten, sowie daran, dass es den Autoren an Erfahrungen in dieser Größenordnung mangelte. Mithilfe eines Bereitschafts-Checks und eines durchgeführten Tests wurden erste Erkenntnisse gewonnen. Weitere Erfolgsfaktoren waren eine transparente und dauerhafte Kommunikation über das Vorhaben. Die Autoren geben nützliche Hinweise zur Definition von Regeln und weisen auf die Koordination der Logistik hin.

In Kapitel drei sind die Phasen, Checklisten und Beispiele für Flipcharts und eingesetzte Hilfsmittel sowie Planungsbeispiele abgebildet. Alle Fallstudien und Gratis-Checklisten stellen die Autoren auf ihrer Webseite zur Verfügung. Im vierten Kapitel wird ausführlich und detailliert mit einer schrittweisen Anleitung beschrieben, wie ein Self-Selection-Event durchgeführt werden sollte. Auch in diesem Kapitel werden Checklisten und Best Practices zur Verfügung gestellt. Alle Hilfsmaterialien werden um wichtige Hinweise und Bilder sowie nützliche Raumpläne ergänzt. Die einzelnen Schritte und Phasen des Prozesses werden durch kurze Erfahrungsberichte und Statements der Teilnehmer oder Trainer ergänzt.

In Kapitel fünf beschreiben die Autoren, wie es nach dem Event weitergehen sollte und welche Schritte und Maßnahmen nützlich sind. Hilfreiche Tools und Besprechungsstrukturen, wie zum Beispiel das Lean Coffee, werden vorgestellt. Die Autoren weisen auf die unverzügliche Etablierung der neuen Teamorganisation hin und stellen auch Fehler und Erfolgsfaktoren vor, die in dieser Phase zu betrachten sind. In den abschließenden Kapiteln sechs und sieben wird kurz auf die Erkenntnisse und Erfahrungen des Prozesses eingegangen. Die möglichen Langzeiteffekte von Self-Selection, wie zum Beispiel auf die Produktivität und auf die Zufriedenheit der Mitarbeiter, werden hier beschrieben. Das Buch endet mit dem Verweis auf die Webseite der Autoren, wo weitere Informationen, Fallstudien, Videos und Gratis-Checklisten zum Herunterladen bereitgestellt werden.

Fazit

Das Buch ist einerseits ein Erfahrungsbericht zur Durchführung einer großen Veränderungsmaßnahme in einem Software-Unternehmen und andererseits eine Art Kochbuch oder Trainerleitfaden für Menschen, die Veränderungen in Organisationen begleiten und operativ durchführen werden. Die Materialsammlungen, Checklisten und Hilfsmaterialen sind übersichtlich dargestellt. Erfahrenen, agilen Coaches oder Scrum-Mastern dürften viele Hilfsmittel bekannt sein. Auch die Organisation und Planung von Veranstaltungen ist nicht neu, aber gut und strukturiert aufbereitet. Hilfreich wären Erfahrungsberichte aus anderen Unternehmen und Beispiele zu Vorher/Nachher-Unterschieden.

Mithilfe dieses Handbuchs ist es möglich, Self-Selection-Events durchzuführen und sofort anzufangen. Es setzt darauf auf, dass Führungskräfte, Geschäftsführer und Entscheider sich bereits für eine andere Art von Veränderungsmaßnahmen entschieden haben und bereit für die Transformation sind.

Autoren:	**Sandy Mamoli, David Mole**
Titel:	**Gemeinsam großartige Teams schaffen**
Verlag:	**Carl Hanser Verlag**
Umfang:	**107 Seiten**
Preis:	**22,00 Euro**
ISBN:	**978-3-446-45673-0**

Das Buch steht als Druck-Version und ePaper zur Verfügung.

Michael Fritz
michael.fritz@msg.group

Zehn Jahre iJUG – eine Erfolgsgeschichte

Stefan Koospal, iJUG

Der iJUG – der Interessenverbund der Java User Groups im deutschsprachigen Raum – feiert im Jahr 2019 sein zehnjähriges Bestehen. Mittlerweile zählt der Verein 38 Java User Groups als Mitglieder und ist in der Java-Welt bekannt. Doch das war nicht von Anfang an so. In diesem Artikel finden Sie spannende Fakten und Zahlen zur Vereinsgeschichte des iJUG, Veranstaltungen sowie den Community-Projekten JavaLand und Java aktuell.

Kleine Anfänge

2009 war ein wichtiges Jahr für Java. Oracle übernahm SUN und der iJUG – der Interessenverbund der Java User Groups im deutschsprachigen Raum – wurde gegründet. Fried Saacke von der DOAG machte damals den ersten Schritt und lud einige bestehende Java User Groups nach Frankfurt zur Gründungsversammlung ein.

Die Gründungsmitglieder des iJUG e.V. bestanden aus DOAG e.V. (Fried Saacke und Andreas Badelt,), Java User Group Stuttgart e.V. (Dr. Michael Paus und Tobias Frech), JUG Erlangen-Nürnberg (Oliver Szymanski), JUG Köln (Michael Hüttermann), JUG München (Andreas Haug) und JUG Deutschland e.V. (Frank Schwichtenberg). Im April 2010 erschien der erste Newsletter, der vor allem die Vortragstermine in den einzelnen JUGs bekannt machen sollte. Die iJUG-Gründung setzte vieles in Bewegung: Neue JUGs wurden gegründet und stießen bald dazu. Von Anfang an versuchte der iJUG einen direkten Kontakt zu wichtigen Playern im Java-Bereich aufzubauen und zu pflegen. Vertreter von Oracle waren regelmäßig zu den Mitgliederversammlungen eingeladen und lieferten fundierte Informationen. Dazu gehörten auch Vortragsreihen von Oracle zu aktuellen Java-Themen in den einzelnen JUGs.

Abbildung 1: Mitgliederversammlung des iJUG im März 2010

Schon vor der Gründung des iJUG versuchten die einzelnen User Groups der Community, zu niedrigen Kosten aktuelle Informationen in Form von Tagungen zu liefern. Die größte Konferenz, das Java Forum Stuttgart, zieht seit 1998 jedes Jahr immer mehr Java-Entwickler in die Liederhalle. Seit 2009 hat sich die Teilnehmerzahl mit aktuell knapp 2.000 Besuchern fast verdoppelt. Die DOAG Konferenz + Ausstellung in Nürnberg mit ca. 2.000 Teilnehmern gibt es schon seit 1987, allerdings erst seit 2010 mit einem signifikanten Anteil an Java-Talks. Die Source-Talk-Tage in Göttingen, veranstaltet vom JUG Deutschland e.V. und SUN User Group Deutschland e.V., boten zwischen 2005 und 2014 eine Mischung aus SUN- und Java-Themen in der Tradition der SUG-Konferenzen in den 90er Jahren. Der Nachfolger ist das Java Forum Nord seit 2015 in Hannover. Der Herbstcampus in Nürnberg legt seit 2007 den Schwerpunkt auf Softwareentwicklung mit vielen Java-Themen. Ins Leben gerufen wurde er von der JUG Erlangen-Nürnberg. Inzwischen hat der Heise-Verlag die Organisation übernommen.

Abbildung 2: Java User Group Stuttgart im Jahr 1999

Java aktuell

Das erste große Projekt des iJUG war die Herausgabe der Zeitschrift „Java aktuell". Viele Mitglieder waren skeptisch, ob es gelingt, eine Printzeitschrift zu Java aus eigener Kraft am Leben zu erhalten.

Fried Saacke konnte die Zweifler – den Autor dieses Artikels eingeschlossen – überzeugen. Mit Wolfgang Taschner übernahm ein versierter technischer Journalist die Chefredaktion und brachte das Journal mit Herzblut voran. Er wird dieses Jahr auf der JavaLand in den Ruhestand verabschiedet. Danach übernimmt Lisa Damerow von der DOAG Dienstleistungen GmbH diesen Aufgabenbereich. Die erste Ausgabe erschien im Herbst 2010, die ersten Jahre mit vier Ausgaben, seit 2018 mit sechs Ausgaben pro Jahr. Das Besondere der Zeitschrift ist zweifelsohne die enge Verknüpfung mit der Community. Inzwischen erscheint die „Java aktuell" in einer Auflage von mehr als 10.000 Exemplaren, sowohl auf dem Postweg an Abonnenten als auch am Kiosk.

Abbildung 3: Cover der Java aktuell Ausgabe 1/2010

JavaLand

Das zweite große Community-Projekt trägt den Namen „JavaLand". Auch hier galt es erst mal, Bedenkenträger zu überzeugen. Wird die Location funktionieren? Gibt es nicht schon genug Konferenzen? Für viele Entwickler hat sich die JavaLand seit 2013 zum Highlight des Jahres entwickelt. Die Mischung aus besonderer Location und geballten Informationen, gepaart mit reichlich Austausch innerhalb der Community, lockt inzwischen mehr als 2.000 Besucher in das Phantasialand nach Brühl.

Abbildung 4: Der Vorstandsvorsitzender des iJUG, Fried Saacke, bei der Eröffnung der JavaLand 2018

Die zuletzt über 600 Vortragseinreichungen werden von einer Gruppe aus ehrenamtlichen Helfern aus den JUGs bewertet und zu einem Programm zusammengestellt. Das Newcomer-Programm für Erstreferenten ermöglicht es zudem, mit Unterstützung eines Mentors aus der Community Vortragserfahrungen zu sammeln. Ein weiterer Arbeitskreis aus den JUGs kümmert sich um die Community-Aktivitäten – eine Mischung aus Workshops, Networking und Sport, bei der für jeden Teilnehmer etwas dabei ist.

Abbildung 5: Plenum auf der JavaLand 2018

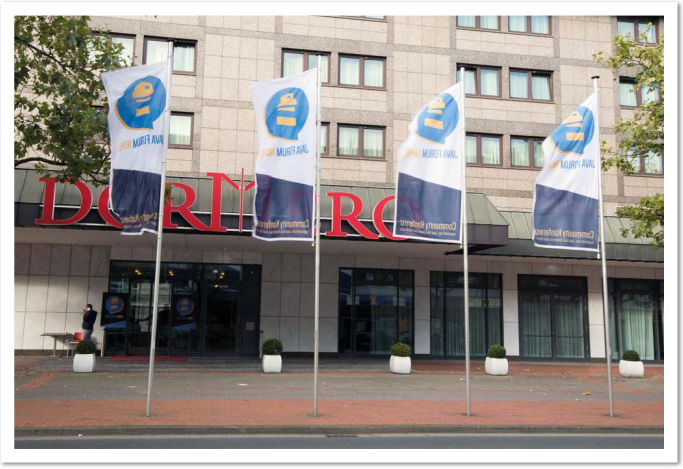
Seit 2015 findet nach dem Vorbild der belgischen „Devoxx4Kids" am Tag vor der JavaLand die „JavaLand 4 Kids" statt. Schüler aus der Umgebung im Alter von acht bis achtzehn Jahren werden dazu eingeladen, einen ganzen Tag lang zu forschen und spielerisch an das Programmieren herangeführt zu werden. Treibende Kraft der Aktion ist Uwe Sauerbrei von der JUG Hamburg, die sich auch in der Hansestadt um den Nachwuchs für die digitale Welt kümmert.

Abbildung 6: Beim „JavaLand 4 Kids" werden Kinder spielerisch ans Programmieren herangeführt

Aufschwung der Community

Aktuell zählt der iJUG im deutschsprachigen Raum 38 Java User Groups als Mitglieder. Die meisten haben sich nach 2009 zusammengefunden und wurden bei der Gründung tatkräftig unterstützt. Jeder iJUG-Newsletter zeigt mit einer Liste der Vorträge in den verschiedenen Regionen, wie aktiv die Community sich neue Inhalte erarbeitet und diese verbreitet. In einigen Regionen zählen regelmäßig stattfindende Stammtische zum festen Programm. Neue

JUG-Konferenzen sind entstanden und werden immer erfolgreicher: Die Berlin Expert Days (BED) Con seit 2011, der JUG Saxony Day in Dresden seit 2014 und seit 2018 in Hausham bei München die JSpirit-Konferenz, die Java, Skifahren und Whiskey verbindet. Durch die regelmäßige Zusammenarbeit im iJUG entstand unter den JUGs in Norddeutschland ein so guter Zusammenhalt, dass 2014 eine gemeinsame Konferenz – das Java Forum Nord – in Hannover ins Leben gerufen wurde. Zur fünften Auflage in diesem Jahr erwarten die Organisatoren rund 600 Entwickler.



[Foto: Mike Meier]

Java ist in Bewegung

Seit der Gründung des iJUG hat sich in der Java-Welt einiges getan. Die Sprache wurde wesentlich erneuert. Viele neue Elemente wie Lambdas wurden eingeführt, manche alten Konstrukte werden nicht mehr unterstützt. JavaScript-Applikationen verdrängen Java-Appletts auf dem Desktop. Die Release- und Supportpolitik von Oracle bereitet dabei vielen Entwicklern und Anwendern Kopfschmerzen. Der iJUG versucht, dabei Orientierungshilfe zu liefern und im Sinne der Community Einfluss auszuüben. Auf Seite 9 finden Sie zu dieser Thematik einen ausführlichen Artikel von Michael Paege. Nachdem Jakarta EE offiziell die Nachfolge von Java EE 8 unter der Regie und Lizenzierung durch die Eclipse Foundation übernommen hat, hat der iJUG eine intensive Zusammenarbeit mit der Eclipse Foundation vereinbart. Die Umstellung des Support-Modells und der Update-Zyklen durch Oracle verunsichert einige Anwender. Ist Amazon Coretto eine Alternative? Diese und andere Fragen werden im iJUG kontrovers diskutiert. Entscheidend ist dabei nicht, zu einer einheitlichen Meinung zu kommen, sondern die Argumente für verschiedene Standpunkte darzustellen und den Anwendern Entscheidungshilfen zu liefern.

Kommunikation im iJUG

Jede User Group im iJUG benennt mindestens zwei Vertreter oder Vertreterinnen. Am dritten Tag der JavaLand treffen sich alle zur Mitgliederversammlung. Neben organisatorischen Fragen stehen inhaltliche Punkte, wie die Zusammenarbeit mit der Eclipse Foundation, auf der Tagesordnung. Ein zweites Treffen findet am Vortag des Java Forum Stuttgart statt. Die Verbindung mit großen und wichtigen Community-Konferenzen erleichtert den meisten Mitgliedern die Teilnahme. Neben persönlichen Zusammenkünften erfolgt die Zusammenarbeit hauptsächlich über E-Mails. Dauerhaftes wird in einem Redmine-Wiki festgehalten; dort werden über das Ticketsystem Aufträge verteilt und abgearbeitet. Es passiert einiges, ohne dass die Community es direkt wahrnimmt.

Stefan Koospal
stefan.koospal@mathematik.uni-goettingen.de

Stefan Koospal ist Systemverwalter am Mathematischen Institut der Universität Göttingen. Nach dem Mathematikstudium in Göttingen arbeitete er für eine Unternehmensberatung im Bereich Netzwerk der AS/400 und C-Programmierung auf dem PC. Die Aufnahme der Tätigkeit am Mathematischen Institut 1992 brachte ihn frühzeitig in Kontakt mit dem Internet auf Basis von SUN Workstations. Seit 1994 ist er Mitglied des Vereines SUN User Group Deutschland e.V., seit 2006 der Vorsitzende. Beim Schwesterverein Java User Group Deutschland e.V., der 1996 aus der SUG heraus gegründet wurde, arbeitet er im Vorstand mit. Von 2005 bis 2014 veranstalteten die beiden Vereine zusammen die Source-Talk-Tage in Göttingen. Diese Tagung wurde 2015 abgelöst durch die gemeinsame Konferenz der JUGs im Nord- und Mitteldeutschland – dem Java Forum Nord in Hannover. Der SUG Deutschland e.V. ist seit 2010 Mitglied im iJUG e.V., seitdem ist er auch in dessen Vorstand aktiv. In Göttingen organisiert er den monatlichen Java-Stammtisch. Weitere Interessen gelten dem e-Learning im Bereich Mathematik.

Calliope-Workshop

Tino Sperlich, Kids4IT

Das Schülerforschungszentrum Hamburg (SFZ) ist mittlerweile zu einer zweiten Heimat für Kids4IT geworden: In einer anregenden Umgebung haben wir wunderbare Räume und sogar einen Lagerraum.

Der letzte dreistündige Workshop stand ganz im Zeichen des Einplatinenrechners „Calliope". Mit fünf Mentoren wurden vierzehn Schüler zwischen acht und vierzehn Jahren betreut. Die Vorstellungsrunde war sehr gut geeignet, das Eis zu brechen, und hat wirklich geholfen, sich (fast) alle Namen einzuprägen: Lisa liest, Tobias tippt, Sophie sagt, Charlotte chattet, Jakob mag Joghurt, Johann joggt, Hauke hilft, Patrick mag Peperoni, Magnus malt, Ilias isst keine süßen Sachen, Tino tanzt ...

Dann ging's gleich in die Vollen: WLAN-Passwort eingeben und den Online-Editor unter *„http://mini.pxt.io"* aufrufen, das war schnell erledigt. Es gibt zwar einen groben, gedruckten Leitfaden, aber, wie immer, war es toll zu sehen, was den Schülern selbst einfiel: Spiele, Geheimnisse, Mathehilfe, Lichtshow. Der Autor als Calliope-Neuling war wirklich elektrisiert. Nachfolgend ein paar Beispiele, die man mit dem Code von GitHub *(siehe „https://github.com/kids4it/workshopresults/tree/master/20181027k4ithh/calliope")* und dem Simulator auch ohne Hardware nachbauen kann.

Magnus und Ilias – Reaktionsspiel: Starte das Spiel mit beiden Knöpfen A und B. Die Lichtkette biegt zufällig nach links oder rechts ab. Drücke rechtzeitig den richtigen Knopf (Knopf A links, Knopf B rechts), sonst ist das Spiel verloren und ein Ton ertönt.

Jakob – Binärzähler: Jakob hat mit den LEDs einen Zähler auf Zweier-Basis implementiert. Bei den ersten Versionen kam der Verdacht

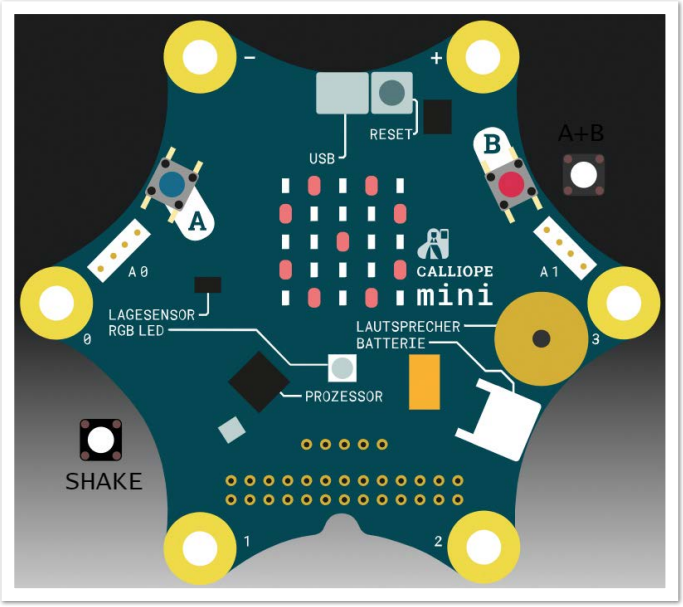


Abbildung 1: Der Calliope

auf, dass eine LED defekt ist. Was tun? Es wurde ein kleines Testprogramm erdacht, das auf Knopfdruck alle LEDs aufleuchten lässt. Die LEDs waren ok, also musste es ein Programmfehler sein. Daraufhin haben Jakob und Daniel debuggt und den Fehler gefunden.

Henry und Henri – elektronischer Würfel: Man drückt abwechselnd Knopf B, um einen neuen Wert zu erhalten. Wer eine „6" würfelt, sieht und hört eine Belohnung.

Moritz und Kit – Pacman 2018: Man benutzt alle vier Eingabemöglichkeiten, damit sich der Pacman-Punkt über das Spielfeld bewegt: Knopf A nach links, Knopf B nach rechts, A+B nach oben, Schütteln nach unten, aber Vorsicht am Rand!

Johann und Eric – Verschlüsselung light: In bester Forschermanier wurden erst mal die Zahlenräume getestet und eine möglichst große Multiplikation gerechnet. Dann eine fast endlose Zeichenkette ausgeben – wird der Calliope das aushalten? Und es musste eine Verschlüsselung her, um den Code zu schützen: Erst eine bestimmte Kombination von Knopf A und B gibt den gewünschten Text aus. Als Variante entstand noch ein „Keylogger", der die Tastendrücke grafisch ausgab.

Weitere Informationen unter *„https://www.kids4it.de/calliope-workshop".*

Tino Sperlich
tino.sperlich@gmx.de

Tino Sperlich arbeitet als Software-Entwickler und Requirements Engineer in Hamburg. Er unterstützt regelmäßig Kids4IT, eine gemeinnützige User Group, und veranstaltet monatliche Workshops in der Hansestadt, um Schülern die Faszination von Technik allgemein und Programmierung im Speziellen zu vermitteln.

Microservices mit dem Helidon Framework

Marcel Amende, Oracle Deutschland B.V. & Co. KG

Von moderner Java-Entwicklung wird heute viel Agilität verlangt. Man will schnell und einfach neue Projekte starten, diese von der ersten Minute an automatisiert testen und früh in Produktion bringen. Es braucht die richtigen Frameworks, damit man sich auf die funktionalen Anforderungen konzentrieren kann, ohne Abstriche bei Codequalität und -leistung zu machen. Das Spring-Ökosystem [1] mit dem im Jahr 2014 veröffentlichten Spring-Boot-Projekt [2] ist sicherlich ein Pionier der modularen Entwicklung in Java. Es eignet sich gut für die Entwicklung von Microservices, wobei es in Bezug auf Funktionsumfang und Größe am oberen Ende des Werkzeugspektrums liegt. Mit dem Open-Source-Projekt Helidon [3] gibt es nun eine leichtgewichtige Alternative, die insbesondere Enterprise-Java-Entwicklern einen leichten Einstieg ermöglicht.

Klein ist fein

Neue technische Möglichkeiten bringen Veränderung. Die Veränderungsgeschwindigkeit nimmt zudem immer weiter zu. Auch die „Kunst" des Programmierens ist dadurch im steten Fluss. Java ist in dieser Hinsicht sicherlich keine Experimentierplattform, versucht allerdings konsequent, Trends aufzugreifen und diese zu professionalisieren, sobald sie sich bewähren. Für die Entwicklung von mehrschichtigen, monolithischen Webapplikationen ist mit Java EE eine beeindruckende Plattform entstanden. Heute ist jedoch die Entwicklung von (Cloud-) Applikationen in Form von kleinen, modularen und ereignisgetriebenen Diensten, sogenannten Microservices, gefragt. Die funktional reichhaltigen, aber schwergewichtigen Applikationsserver-Plattformen verlieren vor diesem Hintergrund trotz aller Optimierungen an Attraktivität. Zumal sich auch die Art der Ausbringung verändert. Statt eine reine Java-Applikation in ein Applikationsserver-Cluster auszubringen und alle weiteren Ressourcen zu referenzieren, packt man in Microservice-Architekturen in sich geschlossene Funktionalität mit allen benötigten Abhängigkeiten, Bibliotheken und Ressourcen in unabhängig lauffähige (Docker *[4]* -) Container. Diese werden wiederum auf einer verwalteten Containerplattform, zum Beispiel mit Kubernetes *[5]*, betrieben.

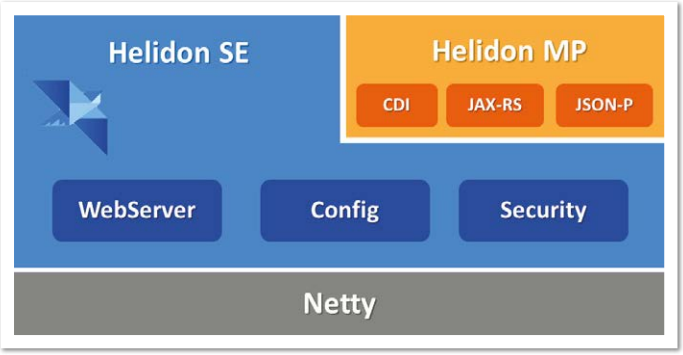


Abbildung 1: Helidon-Architektur

Dadurch ändern sich die Anforderungen an die Java-Plattform und an die Java Frameworks. Bei Ausbreitung in Hunderte Funktionen und Container ist die Größe und Startzeit der virtuellen Maschine entscheidender als das Deployment-Modell. Die Antwort aus Sicht von Java EE auf die neuen Anforderungen ist die Verschlankung zum MicroProfile, bei dem alle für die Entwicklung von Microservices irrelevanten Bibliotheken herausgestrichen werden. Letztlich ist aber auch eine Beschränkung auf Java SE und somit ein Verzicht auf sämtliche Pack- und Ausbringungsfunktionalität absolut möglich.

Helidon-Einführung

Das Open Source Framework Helidon ist eine Sammlung von Java-Bibliotheken, die das Schreiben von Microservices vereinfacht. Im Kern nutzt es das Netty Framework *[6]*, das speziell für die schnelle und einfache Erstellung von asynchronen, ereignisgetriebenen Netzwerkapplikationen geschaffen ist. Durch seine Nutzung der Java NIO (non-blocking IO) APIs werden bei Netzwerkoperationen keine Threads blockiert, was einen deutlich besseren Durchsatz und eine geringere Latenz bei gleichzeitig minimiertem Ressourcenbedarf ermöglicht. Helidon stellt auf dieser Basis einen reaktiven Webserver bereit, bei dem ein einzelner Thread mehrere (TCP-) Verbindungen gleichzeitig abhandeln kann. Grundlage dieser Implementierung ist zudem das Reactive Pattern *[7]*, das die Registrierung der Event Handler für die ereignisbasierte Verarbeitung der Serviceanfragen übernimmt. Dadurch wird letztlich eine gute vertikale Skalierung innerhalb einer virtuellen Maschine erreicht.

Helidon unterstützt zudem das Java MicroProfile. Weithin bekannte APIs wie JAX-RS für die Erstellung von RESTful Web Services, JSON-P für das Parsen von JSON-Nachrichten und Dependency Injection (CDI) sind nutzbar.

Mit Helidon erstellte Microservices werden als Docker-Container gepackt. Die Nutzung in Kombination mit einer, zum Beispiel durch Kubernetes, verwalteten Containerplattform bietet sich daher an. Helidon bietet darüber hinaus Integrationsmöglichkeiten beispielsweise mit dem Prometheus Framework für die Visualisierung operationaler Metriken oder mit dem Zipkin Framework für Nachverfolgbarkeit und Latenzanalysen in verteilten Microservice-Umgebungen.

Programmiermodell der Wahl

Bei Helidon kann man sich zwischen zwei Programmiermodellen entscheiden: Man spricht von Helion SE bei Verwendung des MicroFramework oder von Helidon MP bei Verwendung des MicroProfile. Das MicroFramework folgt einem rein funktionalen Programmierstil, bei dem man die von Helidon zur Verfügung gestellten APIs für WebServer, Konfigurationsverwaltung und Sicherheitsfunktionalität direkt nutzt. *Listing 1* zeigt ein Beispiel. Diese Variante bietet volle Transparenz und Kontrolle.

Für Entwickler, die auf ihre Erfahrung mit Enterprise Java (JEE) zurückgreifen möchten, bietet sich das MicroProfile mit seinem eher deklarativen Programmiermodell an. Hier werden Annotationen genutzt, um aus einfachen Java-Klassen Services und aus den Methoden der Klasse Service-Endpunkte zu machen. *Listing 2* zeigt die Deklaration eines einfachen REST-Service.

Schnelleinstieg in Helidon

Der Einstieg in Helidon gelingt schnell und mit wenigen Voraussetzungen. Man braucht am Entwicklerarbeitsplatz als Minimalvoraussetzung nur Java SE 8 oder Open JDK 8 als Laufzeitumgebung, Maven (ab Version 3.5) für den Buildprozess und Docker (ab Version 18.02) als Container für die Ausbringung des Microservice. Für den Fall, dass ein durch Kubernetes verwaltetes Containercluster Ziel der Ausbringung ist, benötigt man zusätzlich Kubectl (ab Version 1.7.4) für den administrativen Zugriff vom Client aus.

Bei der händischen Erstellung eines RESTful-Service mit Helidon beginnt man mit der Erstellung eines Maven-Projekts in einer Java-IDE der Wahl oder durch die Ausführung eines Standard-Maven-Archetyps auf Kommandozeilenebene *(siehe Listing 3)*.

Das dabei entstehende pom.xml muss um zwei Einträge ergänzt werden, um den Helidon-Web-Server und YAML als Helidon-Konfigurationsformat heranzuziehen *(siehe Listing 4)*.

Bei Verwendung einer IDE sollten die Main-Klasse und die Pfade abhängiger Bibliotheken automatisch eingefügt werden. Ansonsten

```
WebServer.create(
  Routing.builder()
     .get("/greet", (req, res)
        -> res.send("Hello MicroFramework!"))
     .build())
  .start();
```

Listing 1: Programmierbeispiel MicroFramework

```
public class GreetService {
  @GET
  @Path("/greet")
  public String getMsg() {
     return "Hello MicroProfile!";
  }
}
```

Listing 2: Programmierbeispiel MicroProfile

```
$ mvn archetype:generate
     -DarchetypeGroupId=org.apache.maven.archetypes
     -DarchetypeArtifactId=maven-archetype-quickstart
     -DarchetypeVersion=1.3
```

Listing 3

ist das detaillierte Vorgehen in der Sektion „Guides" der Helidon-Dokumentation beschrieben *[8]*.

Noch einfacher findet man den Einstieg, wenn man spezielle Helidon-„Quickstart"-Archetypen für Maven verwendet. Diese gibt es für beide Programmiermodelle. Für Helidon SE kann das in *Listing 5* gezeigte Kommando genutzt werden, wobei die Parameter „groupId", „artifactId" und „package" frei wählbar sind.

Dadurch wird ein kompilierbares und sofort lauffähiges Maven-Projekt generiert, das einen auf dem Helidon Framework basierenden RESTful-Service enthält. Zentraler Teil des Projekts ist eine ausführbare Klasse „Main.java", die den Web-Server startet und das Routing für den Service-Endpunkt erstellt *(siehe Listing 6)*.

Eine weitere Klasse „GreetService.java" stellt die auf dem Interface „io.helidon.webserver.Service" basierende Serviceimplementierung dar. Sie registriert sich über ein Update der Routingregeln und generiert für die verschiedenen Servicemethoden JSON-Ausgaben *(siehe Listing 7)*.

Den Build-Prozess für das Projekt stößt man über Maven mit $ mvn package an. Das Resultat ist eine Beispielapplikation in einem JAR-Archiv; alle zur Laufzeit benötigten abhängigen Bibliotheken werden in einem Unterverzeichnis „target/libs" gespeichert. Die Applikation kann durch Ausführen der JAR-Datei lokal gestartet und getestet werden: `$ java -jar target/quickstart-se.jar`

Die Applikation stellt einen RESTful-Service bereit, der exemplarisch einige GET- und POST-Methoden unterstützt, um Grußbotschaften in Form von JSON-Nachrichten auszugeben. Folgender Aufruf ist z.B. möglich: `$ curl -X GET http://localhost:8080/greet/Java%20Aktuell`
`{"message":"Hello Java Aktuell!"}`

Ein Microservice ist mit Helidon also in wenigen Sekunden erstellt: Beispielprojekt mit Maven aus einem Archetyp erstellen, packen und starten. Nun kann man mit der funktionalen Erweiterung und Anpassung fortfahren. Bleibt die Frage, wie man den Microservice auf einfache Art und Weise ausbringen und betreiben kann.

Helidon in der Cloud

Docker-Container sind heute typischerweise das Mittel der Wahl für das Packen und Ausbringen von Microservices, vor allem in privaten oder öffentlichen Cloud-Umgebungen. Dazu trägt bei, dass die Docker-Container mit Werkzeugen wie Kubernetes (K8s) auch in großer Anzahl und in großen Clustern effektiv verwaltet werden können.

An dieser Stelle werden wir die Oracle Container Native Services *[9]* der Oracle Cloud Infrastructure (OCI) heranziehen, um Helidon in einer Containerumgebung zu betrachten. Diese beinhalten eine verwaltete Kubernetes-Umgebung und eine Docker Registry. Mit einem kostenlosen Testzugang *[10]* lässt sich das Beispiel praktisch nachvollziehen. Lokale Installationen von Docker und Kubernetes sowie Docker Hub *[11]* und Cloud-Umgebungen vieler anderer Anbieter sollten ebenfalls nutzbar sein. Die in den folgenden Beispielen verwendeten Docker- und Kubernetes-Kommandos sind allgemeingültig. Nur die APIs für Zugang und Konfiguration sind anbieterspezifisch.

Der erste Schritt in Richtung der Ausbringung des Helidon-Beispielservice ist das Packen als Docker Image. Das mit dem Schnellstartbeispiel ausgelieferte Dockerfile sorgt dafür, dass Applikation und Blibliotheken in ein online frei beziehbares Basisimage mit bereits enthaltenem OpenJDK kopiert werden. `$ docker build -t quickstart-se target`

```
<dependencies>
     <dependency>
        <groupId>io.helidon.bundles</groupId>
        <artifactId>helidon-bundles-webserver</artifactId>
     </dependency>
     <dependency>
        <groupId>io.helidon.config</groupId>
        <artifactId>helidon-config-yaml</artifactId>
     </dependency>
</dependencies>
```

Listing 4

```
$ mvn archetype:generate
     -DinteractiveMode=false \
     -DarchetypeGroupId=io.helidon.archetypes \
     -DarchetypeArtifactId=helidon-quickstart-se \
     -DarchetypeVersion=0.11.0 \
     -DgroupId=io.helidon.examples.javaaktuell \
     -DartifactId=quickstart-se \
     -Dpackage=io.helidon.examples.javaaktuell.quick-
start.se
```

Listing 5

```
...
WebServer server =
     WebServer.create(serverConfig, createRouting());
server.start().thenAccept(ws -> {...});
...
```

Listing 6

```
...
@Override
public final void update(final Routing.Rules rules) {
  rules
     .get("/", this::getDefaultMessage)
     .get("/{name}", this::getMessage)
     .put("/greeting/{greeting}", this::updateGreeting);
}
...
```

Listing 7

```
$ docker images
REPOSITORY      TAG       IMAGE ID      CREATED      SIZE
quickstart-se   latest    1234567abcde  2 hours ago  88.6MB
```

Listing 8

```
$ kubectl create secret docker-registry <SecretName> \n
--docker-server=fra.ocir.io \n
--docker-username='/<TenantName>/<UserName>' \n
--docker-password=<AuthToken> \n
--docker-email='<AnyDummyEmail>'
```

Listing 9

Daraus resultiert ein ca. 88,6 MB großes und mit dem Tag „latest" versehenes Image namens „quickstart-se" im lokalen Docker Repository *(siehe Listing 8)*.

Dieses lässt sich bereits lokal ausführen und mit denselben Endpunkten und Aufrufen testen wie beim vorherigen Start der Applikation als JAR-Datei. `$ docker run --rm -p 8080:8080 quickstart-se:latest`

Die Reise ist hier allerdings noch nicht zu Ende, schließlich soll die Applikation in einer vollverwalteten Umgebung im Cluster bereitgestellt werden. Erster Schritt dafür ist die Anmeldung bei der entfernten Docker Registry von der Docker-Kommandozeile aus. In diesem Fall handelt es sich um eine private Oracle Cloud Infrastructure Registry (OCIR) in der Rechenzentrumsregion Frankfurt („fra.ocir.io"): `$ docker login fra.ocir.io -u <TenantName>/<UserName> -p <AuthToken>`

Im nächsten Schritt wird das lokale Image mit einem Tag für den folgenden Push in die Registry in der Cloud versehen: `$ docker tag quickstart-se:latest fra.ocir.io/<TenantName>/<RepoName>/quickstart-se:latest`

Nun kann der „Push", das heißt der Upload des Docker Image, in die Cloud Registry erfolgen: `$ docker push fra.ocir.io/<TenantName>/<RepoName>/quickstart-se:latest`

Da es sich um eine private Registry handelt, muss dem Verwaltungswerkzeug Kubernetes nun eine Authentifizierung durch ein sogenanntes „Pull Secret" ermöglicht werden, um Docker Images per „Pull" für die direkte Ausbringung heranzuziehen. Dieses „Pull Secret" kann über die Kubernetes-Kommandozeile generiert werden *(siehe Listing 9)*.

Das Secret wird als Datei im Home-Verzeichnis des Betriebssystembenutzers abgelegt.

Mit dem Maven-Projekt wurde bereits ein Deskriptor „app.yaml" für die Ausbringung mit Kubernetes generiert. Dieser wird nun angepasst. In der Deployment-Sektion unter `spec: containers:` wird der Link auf das Docker Image in der privaten Cloud Registry gesetzt: image: `fra.ocir.io/<TenantName>/<RepoName>/quickstart-se:latest`

Zusätzlich wird für die Authentifizierung noch einen Verweis auf das generierte „Pull Secret" ergänzt:

```
imagePullSecret:
- name: <SecretName>
```

In der Datei ließe sich auch die Anzahl der Replikate erhöhen oder man könnte Memory-Limits für den Service setzen. Grundsätzlich ist jedoch bereits ohne weitere Anpassungen alles für die Ausbringung über die Kubernetes-Kommandozeile („kubectl") bereitet:

```
$ kubectl create -f target/app.yaml
```

Ist der Kubernetes-Proxy gestartet, kann man sich das Ergebnis der Ausbringung im Kubernetes Dashboard ansehen. Dort sieht man ein Deployment „quickstart-se", das als Service auf einem Kubernetes-Knoten („Pod") läuft *(siehe Abbildung 2)*.

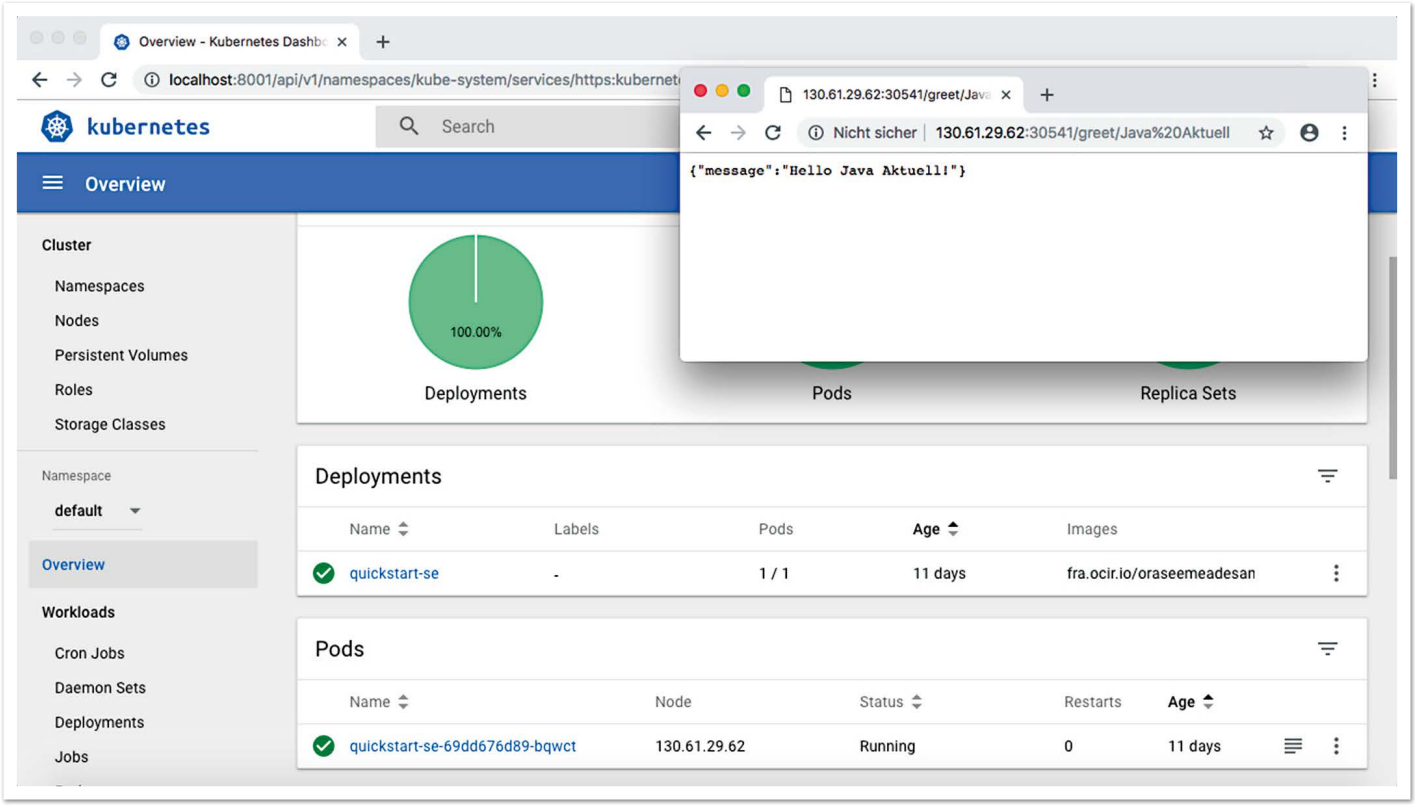


Abbildung 2: Kubernetes Dashboard - Overview

Fazit und Ausblick

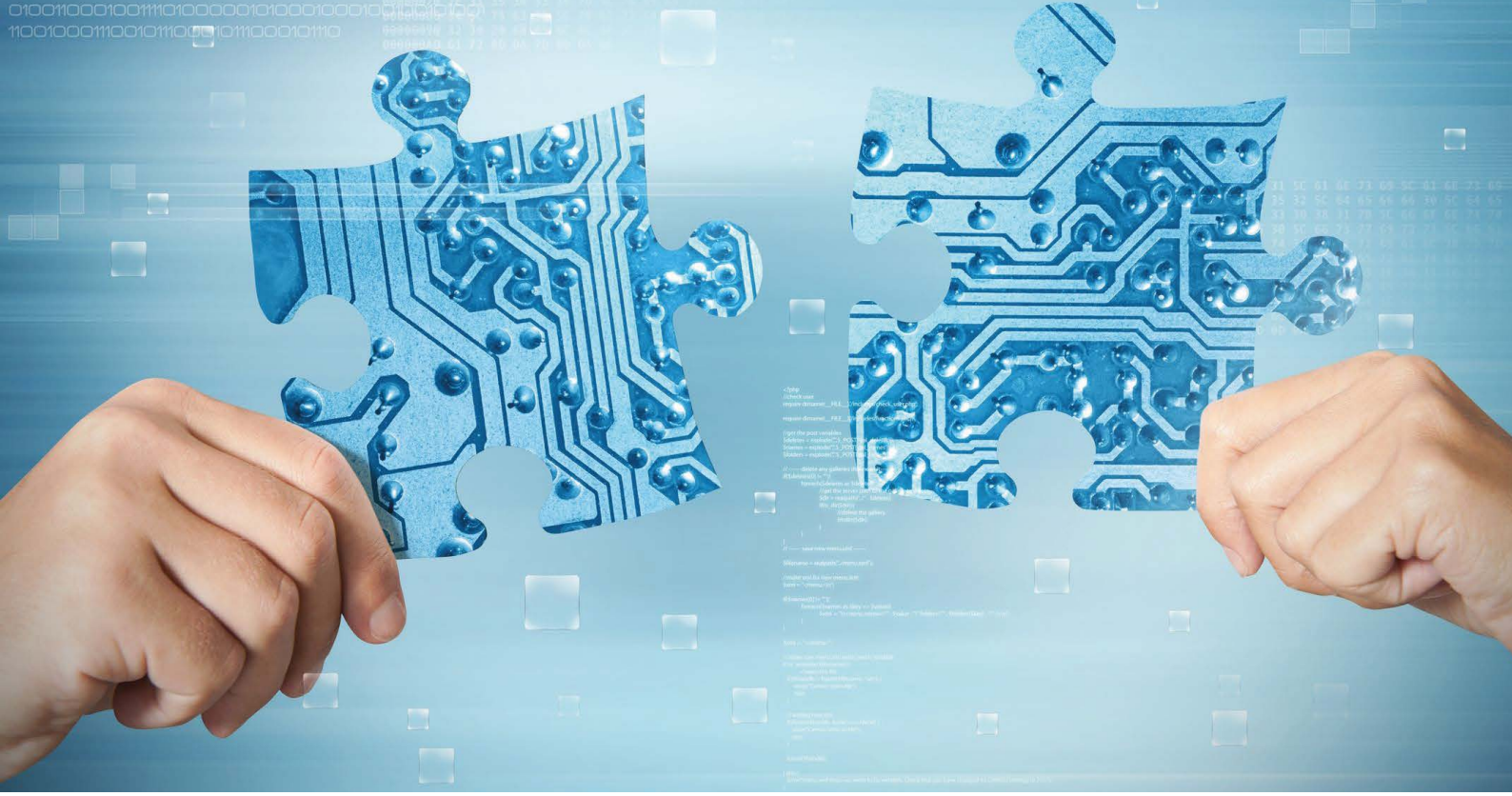
Beim Entwickeln in und für die Cloud gehen die Veränderungen Hand in Hand. Vermehrte Anforderungen an Flexibilität und Agilität in der IT bereiten den Boden für neue Entwicklungsmethodiken wie die Mico-service-Entwicklung. Die Entwicklergemeinde schafft sich im dynamischen Open-Source-Umfeld die Werkzeuge selbst, um die anstehenden Entwickungsaufgaben zu vereinfachen. Mit Helidon reiht sich hier ein neues, schlankes und leicht zu erlernendes Framework ein, das auch von der Interoperabilität lebt. Denn in der Coud entstehen auf Basis von Docker, Kubernetes, Prometheus & Co. die passenden Ablauf-, Verwaltungs- und Überwachungsumgebungen für Microservices. Diese Art der Quasi-Standardisierung auf Basis von Open Source eröffnet dem Entwickler ungeahnte Möglichkeiten der Portabilität zwischen den Cloud-Anbietern. Was kann man von Helidon in Zukunft erwarten? Helidon stünde ein Eventsystem gut zu Gesicht, um aus einzelnen Microservices flexibel modulare Applikationen zu erstellen. Unterstützung für kommende Web-Standars wie HTTP/2 und MicroProfile-Versionen ist naheliegend. Zudem besteht großes Potenzial in einer engeren Integration mit dem JDK, aber auch zum Beispiel mit der Graal VM.

Referenzen

[1] *https://spring.io*
[2] *https://spring.io/projects/spring-boot*
[3] *https://helidon.io*
[4] *https://docker.com*
[5] *https://kubernetes.io*
[6] *https://netty.io/*
[7] *https://java-design-patterns.com/patterns/reactor/*
[8] *https://helidon.io/docs/latest/#/guides/01_SE_REST_web-service*
[9] *https://cloud.oracle.com/containers*
[10] *http://cloud.oracle.com/tryit*

Marcel Amende
Marcel.Amende@oracle.com

Geboren als Ingenieur, aufgewachsen bei Oracle, zu Hause in der Cloud. Als Solution Engineer repräsentiert Marcel die „Cloud Native"-Entwicklergemeinde von Oracle, die aus den Diensten der Oracle Cloud kreative Kundenlösungen gestaltet. Neben der Serviceentwicklung und Integration gilt sein besonderes Interesse auch dem Internet der Dinge.



CQRS und Event Sourcing mit dem Axon Framework

Christian Iwanzik, tarent solutions GmbH

Klassische CRUD-Modelle kommen in puncto Erweiterbarkeit schnell an ihre Grenzen. CQRS gepaart mit Event Sourcing kann in solchen Fällen ein gutes Werkzeug sein, um aus der Komplexitätsfalle zu kommen. Exemplarisch mit dem Axon Framework umgesetzt, zeigt der Artikel neben der Theorie auch ein praktisches Beispiel.

Naiver Ansatz mit CRUD

Moderne Software basiert oft auf iterativen Modellen, um auf sich ändernde Anforderungen schnell und kurzfristig reagieren zu können. Daher sind die ersten Umsetzungen von Software oft mit einer einfachen CRUD-Architektur versehen, die schnell Resultate bringt. CRUD (Create, Read, Update, Delete) beschränkt sich auf eine zentrale Datenstruktur, für die die vier Operationen in einer Serviceklasse implementiert werden. Zusätzliche Controller und Repositories sorgen für die Serialisierung und Persistierung.

In *Abbildung 1* sehen wir eine klassische Kundenverwaltung. Die Customer-Entität wird für alle Belange der Software herangezogen. Customer, inklusive Adressen, werden immer ganz gelesen und bei Änderungen komplett überschrieben.

Allerdings erweitern sich die Anforderungen erwartungsgemäß sehr schnell und die vier Operationen reichen bald nicht mehr aus. So kann das System durch eine Zweifaktor-Registrierung per Mail erweitert werden, um Fake-Registrierungen entgegenzuwirken. Eine Analystin hätte zusätzlich gerne eine Abfrage über noch nicht bestätigte Kunden, um die Conversion Rate auszurechnen. Allein damit haben wir nun aber bereits einen ersten Prozess etabliert, dem unsere Architektur nicht mehr gewachsen ist.

1. Wer verschickt die E-Mail bei der Registrierung?
2. Wie muss eine Customer-Entität aussehen, die einmal einen Zustand eines nicht bestätigten Accounts darstellt und später den eines bestätigten Accounts?
3. Wo wird ein Bestätigungstoken gespeichert?
4. Ist eine E-Mail-Bestätigung ein Update der gesamten Entität?

So kann man sich schnell immer mehr Erweiterungen vorstellen. Kunden sollen ihre E-Mail-Adresse über denselben Zweifaktor-Mechanismus ändern können. Gleichzeitig sollen jedoch während des Prozesses etwaige E-Mails noch an die alte Adresse gesendet werden. Die Entität bläht sich schnell mit optionalen und Null-

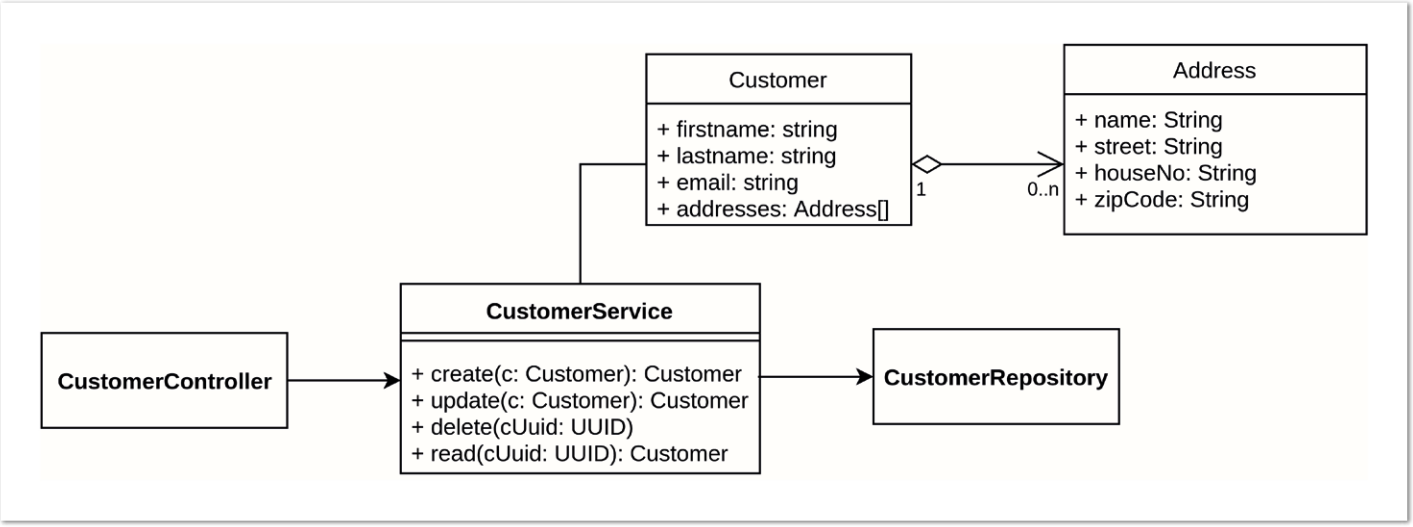


Abbildung 1: Simple CRUD-Architektur

Attributen auf, die eventuell genau einmal in ihrem Lebenszyklus für einen speziellen Prozess ausgewählt werden.

Zu allem Überfluss lagern sich Prozesse und Businesslogik in verschiedene Serviceklassen aus und haben nicht mehr den Zusammenhalt mit der eigentlichen Entität. Das Ergebnis eines solch unkontrollierten Wachstums ist der sogenannte „Big Ball of Mud" *[1]*. Also ein System chaotischer Beziehungen. *Abbildung 2* zeigt ein reales Beispiel einer Komponente, die mit CRUD gestartet ist und sich mit immer weiteren Anforderungen konfrontiert sah. Ein Refactoring auf CQRS konnte hier im Nachhinein einen Großteil der Komplexität eliminieren.

CQRS und Axon

CQRS steht für „Command Query Responsibility Segregation" *[2]*. Indem das System ändernde und lesende Zugriffe auf Daten hart voneinander trennt, können diese besser auf die Anforderungen reagieren und sind dabei leichter erweiterbar. Auch werden per Commands nicht mehr alle Daten überschrieben, sondern Änderungswünsche sauber von der eigentlichen Verarbeitung getrennt.

Abbildung 3 beschreibt eine grobe Skizze der Architektur. Im Zentrum steht immer das Domain Model. Es ist der eigentliche Kern der Software und beschreibt alle für die Software geschäftsrelevanten Daten und Methoden. Also den Grund, aus dem die Software überhaupt entwickelt wird. Das Model bietet nach außen Methoden an, über die Änderungsbefehle, sogenannte Commands, übergeben werden können. Diese Commands können von den Domain-Methoden entweder angenommen oder abgelehnt werden. Angenommene Commands erzeugen erst dann eine Zustandsänderung der Daten.

Commands selbst sind kleine Datenpakete, die so einen Befehl repräsentieren. Sie werden durch den Command Handler angenommen und in Aufrufe für das Domain Model übersetzt.

Ist innerhalb des Domain Model eine Zustandsänderung passiert, so schickt die Domain ein Business-Event in eine Event Queue. Diese Events wiederum werden von einem Event Handler in ein Read Model überführt, das einen speziellen Aspekt der Domain repräsentiert und für Abfragen bereitsteht.

Das Axon Framework

Das Axon Framework ist eine in Java geschriebene Bibliothek für die JVM, welches das CQRS Pattern umsetzt. Die folgenden Beispiele beschreiben das Framework in der Version 3.4 in Kombination mit Spring Boot 2.1.2. Als Sprache wurde Kotlin gewählt.

Die Einbindung in den eigenen Java-Code ist simpel über Maven oder Gradle Dependencies möglich. Eine Interoperabilität mit Spring Boot ist über die „axon-spring-boot-starter"-Dependency bereits vom Framework vorgesehen.

Commands

Am Anfang einer Datenveränderung stehen immer Commands. Dies sind kleine Datenobjekte, die ein bestimmtes Änderungsprädikat darstellen. In *Listing 2* sehen wir ein „CreateCustomer"-Command. Dieses beinhaltet alle Daten, um einen Customer initial anzulegen. Aber eben auch nicht mehr.

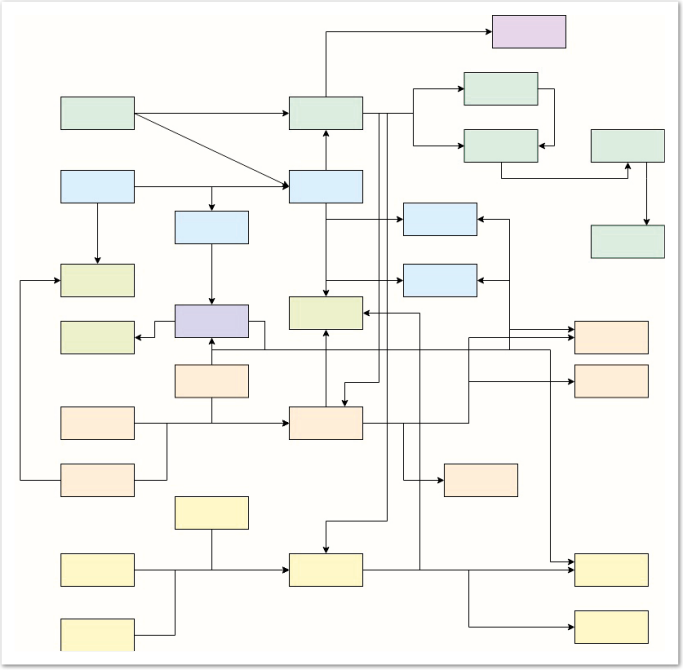


Abbildung 2: Reales Beispiel eines Big Ball of Mud

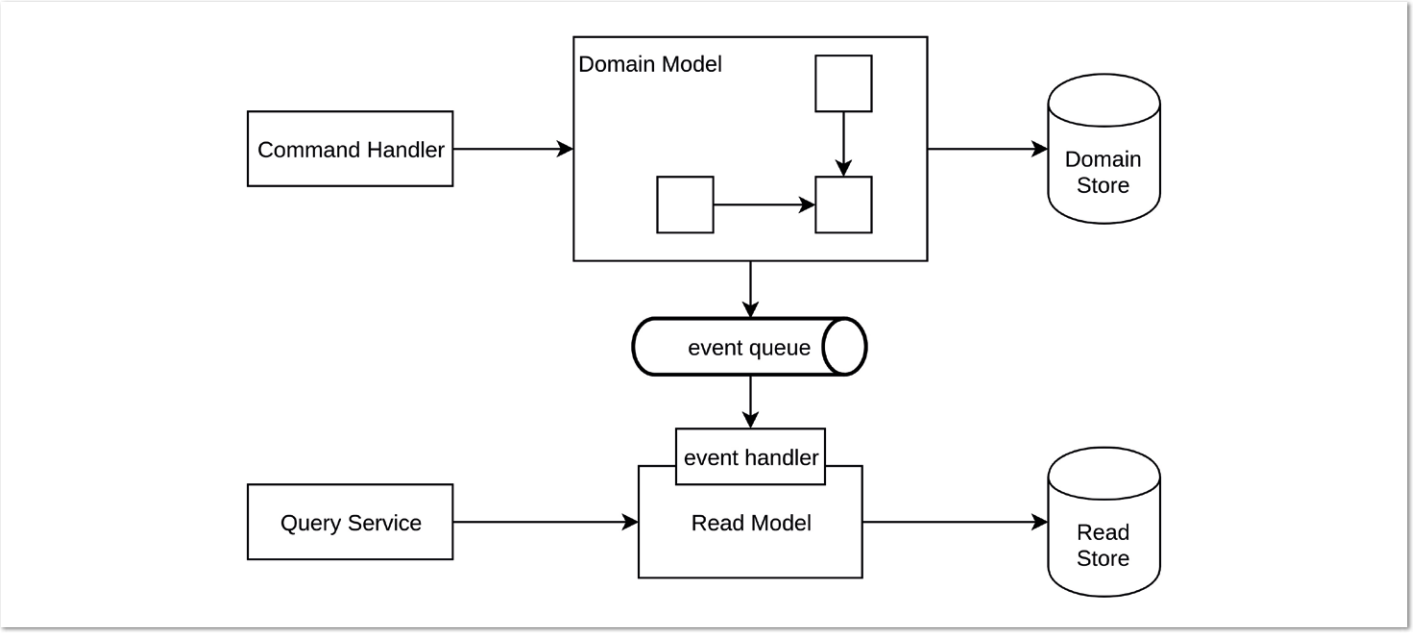


Abbildung 3: Übersicht CQRS

Anders als bei CRUD modellieren wir hier nicht die komplette Entität, die überschrieben werden soll, sondern nur die zu ändernden Elemente. Der angelegte Customer könnte später noch eine Adresse hinzubekommen. Die Adresse würde dann aber mit einem AddAddress-Command hinzugefügt werden, das nur die Daten der Adresse beinhaltet. Commands sind also keine direkten Zustandsbeschreibungen, sondern erst mal nur Absichtserklärungen an das Domain Model. Um diesem Sachverhalt mehr Ausdruck zu verleihen, sollten Commands im Namen immer ein Prädikat gefolgt vom Namen der zu ändernden Entität beinhalten. Als Quelle von Commands können alle eingehenden Datenquellen auftreten. Entweder ein UI oder REST Controller oder ein Message Subscriber.

```
@RequestMapping(value = ["/customer"], method = arrayOf(POST))
fun createCustomer(@RequestBody command: CreateCustomer): ... {
  val result = DeferredResult<Any>()

  // Sending the command into the CQRS system.
  val future: CompletableFuture<UUID> =
     commandGateway.send<UUID>(createCustomerCommand)

  // Return the uuid, when the command handling is finished.
  future.exceptionally { err → … }
     .thenAccept { customerUuid -> ...}

  return result
}
```

Listing 3: Ein REST-Endpunkt

```
@CommandHandler
fun createCustomer(customer: CreateCustomer): UUID {
  return repository
     .newInstance {
        Customer(
           firstname = FirstName( customer.firstname),
           lastname = LastName(customer.lastname),
           email = UnverifiedEmail(customer.email, ...)
        )
     }
     .identifier() as UUID
}
```

Listing 4: Ein Command Handler

Listing 3 zeigt als Beispiel einen Spring-MVC-REST-Endpunkt. Das Command kann hier direkt zur JSON-Deserialisierung benutzt werden. Der Einstieg in das Axon Framework geschieht über das „commandGateway", das als Spring Bean direkt über die Axon Dependencies erzeugt wurde. Das Command Gateway übernimmt alle Commands und vermittelt sie an Command Handler weiter. Diese sind simple Spring Beans, die aber Methoden besitzen, die mit @CommandHandler annotiert sind. Die Registrierung geschieht automatisch.

Listing 4 zeigt den Command Handler, der das Anlegen eines neuen Customer übernimmt. Das hier erwähnte „repository" ist ebenfalls eine Dependency, die von Axon gestellt wird, um das Domain Model zu persistieren. Im weiteren Verlauf erzeugt der Handler ein neues Domain Model und übergibt es dem „repository". Andere Handler würden zunächst das Model mithilfe des „repository" laden und darauf eine Methode anwenden. *Listing 5* zeigt dies anhand einer Adresse.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.2.RELEASE</version>
</parent>

<dependencies>
  <dependency>
     <groupId>org.axonframework</groupId>
     <artifactId>axon-spring-boot-starter</artifactId>
     <version>3.4.2</version>
  </dependency>
  …
```

Listing 1: Maven Dependencies für Axon

```
data class CreateCustomer (
  val name: String,
  val surname: String,
  val email: String
)
```

Listing 2: Ein simples Command

Das Domain Model

Die Domain ist das eigentliche Herz der Software. Sie beschreibt unabhängig von Datenbankschema oder technischen Abhängigkeiten die fachliche Gesamtheit des zu lösenden Problems. Also Daten und Methoden, um diese zu ändern. Wichtig ist hier, dass Daten stark in der Domain gekapselt sein müssen. Änderungen werden lediglich durch Methoden der Domain durchgeführt. Diese haben die Aufgabe, von außen eingegebene Änderungswünsche zu validieren und gegebenenfalls mit einem Fehler abzuweisen. Änderungswünsche, die den Anforderungen entsprechen, wie zum Beispiel Pre- und Postconditions innerhalb der Domain, führen dann zu einer Zustandsänderung der Domain. Wichtig ist aber, dass nur die Domain darüber entscheidet, ob sie die Änderung durchführt oder nicht.

```
@CommandHandler
fun addAddress(addrCmd: AddAddress) {
  repository
      .load(addrCmd.customerUuid.toString())
      .invoke { customer ->
         val address = Address.fromCommand(addrCmd)
         customer.addAddress(address)
      }
}
```

Listing 5: Ein Command Handler auf einem bestehenden Model

```
@Aggregate
class Customer() {
  @AggregateIdentifier
  private var customerUuid: UUID = UUID.randomUUID()
  private var addresses: List<Address> = listOf()
  private lateinit var firstname: FirstName
  private lateinit var lastname: LastName
  private lateinit var email: Email

  constructor(...) {
      this.firstname = firstname
      this.lastname = lastname
      this.email = unverifiedEmail
      AggregateLifecycle.apply(CustomerCreated(...))
  }

  fun addAddress(address: Address) {
      this.addresses = addresses + address
      AggregateLifecycle.apply(AddressAdded(...))
  }
…
```

Listing 6: Das Customer Domain Model – in Axon: Das Aggregat

```
data class CustomerCreated(
  val customerUuid: UUID,
  val name: String,
  val surname: String,
  val email: String
)
```

Listing 7: Das CustomerCreated-Eventobjekt

```
@Component
class AddressAggregator() {
  private val allAddresses: MutableList<Address> = ...
  @EventHandler
  fun handleAddressAdded(event: AddressAdded) {
      allAddresses + event.address
  }
  fun getAllAddresses(): List<Address> {
      return allAddresses.toList()
  }
}
```

Listing 8: Ein Event Handler am Beispiel eines Address-Aggregators

Listing 6 zeigt nun das Customer Domain Model. Wichtig zu wissen ist, dass das Domain Model bei Axon „Aggregate" genannt wird. Das Aggregat existiert nicht als Bean im System, sondern wird immer vom Repository geladen und instanziiert beziehungsweise von einem Command Handler erzeugt. Zur Serialisierung durch Axon werden lediglich die beiden Annotationen „@Aggregate" und „@AggregateIdentifier" benötigt.

Domain Events

Sich ändernde Domains beziehungsweise Aggregate werden in der Folge ein Domain Event emittieren, um der „restlichen Welt" zu sagen, dass sich etwas geändert hat. Domain Events lassen sich auf zwei grundlegende Weisen beschreiben. Als idempotente Objekte, die den gesamten augenblicklichen Zustand der Domain beschreiben, oder als Deltaobjekte, die nur die Änderung der Domain darstellen.

Auf Code-Ebene unterscheiden sich Commands und Events nicht sonderlich. Allerdings haben sie eine grundlegend andere Semantik. Während ein Command eine Änderung herbeiführen will, beschreibt ein Event eine bereits vergangene Änderung innerhalb des Model. Daher wird für Eventnamen auch immer die Vergangenheitsform gewählt, um diesen Aspekt zu unterstreichen. *Listing 7* zeigt als Beispiel das CreatedCustomer-Event, das bei der Generierung eines neuen Customer erzeugt wird. *Listing 6* zeigt auch, wie Events aus dem Model emittiert werden können. Über das statische „AggregateLifecycle"-Objekt können neue Events erzeugt und in das Axon-System gebracht werden.

Event Handler

Die emittierten Events werden nun durch registrierte Event Handler weiterverarbeitet. Auch hier macht es die Spring Integration wieder sehr einfach. Handler-Klassen müssen lediglich als Spring Bean im Context vorhanden sein. Methoden, die Events verarbeiten, werden dann mit einer „@EventHandler"-Annotation versehen. *Listing 8* zeigt als Beispiel einen Adress-Sammler, der sich auf „AddressAdded"-Events registriert. Axon ermittelt den passenden Eventtyp für den Handler dabei per Reflection.

Read Model und Queries

CQRS sieht nicht vor, das Domain Model bei der Verarbeitung eines Command direkt an einen Aufrufer zurückzugeben. Für diese Fälle bietet es die Read-Modelle. Dabei muss das Read-Modell nicht eins zu eins das Domain-Modell abbilden, sondern kann und soll immer das Bedürfnis der Abfrage erfüllen. Daher ist es durchaus üblich, mehrere Read-Modelle in einem System zu haben. Die mögliche Datenredundanz wird hier gegenüber der besseren und unabhängigen Entwicklung und Wartung in Kauf genommen.

In großen Systemen kann das Read-Modell auch aus einer Kombination von Events unterschiedlicher Domain-Modelle bestehen, um z.B. KPIs oder andere Analysen um funktionsübergreifende Daten zu erheben. In *Listing 8* haben wir bereits ein erstes Beispiel für ein Read-Modell. Hier interessiert sich die Abfrage nicht für einzelne Kunden, sondern für die Gesamtheit aller Adressen.

Queries sind nun gezielte Abfragen auf ein Read-Modell. Im Beispiel des Listings 8 könnte dies beispielsweise die Anzahl aller Adressen sein. Oder eine disjunkte Menge aller Städte. Andere Abfragen wür-

```
@Aggregate
class Customer() {
...
  @EventHandler
  fun handleAddressAdded(event: AddressAdded) {
     this.addresses = addresses + event.address
  }

  fun addAddress(address: Address) {
     this.addresses = addresses + address
     AggregateLifecycle.apply(AddressAdded(...))
  }
...
}
```

Listing 9: Das Domain Model konsumiert die eigenen Events

den auf andere Aspekte des Systems abzielen und dafür gegebenenfalls andere Read-Modelle benötigen.

Event Sourcing

CQRS kann man per se auch ohne Events betreiben. Wichtig ist allein, dass Commands und Queries getrennt sind und ein zentrales Domain Model die Hoheit über die Daten hat. Allerdings bietet eine eventbasierte Architektur einige Vorteile. Dadurch, dass jede Zustandsänderung des Domain Model in einem entsprechenden Event resultiert, ist es möglich, den aktuellen Zustand des Domain Model allein durch die Zusammenführung aller Events wiederherzustellen. Ein Event Handler könnte so eine exakte Kopie des Domain Model erzeugen, ohne über das eigentliche Domain Model Bescheid zu wissen.

Speichert man dabei noch alle Events mit einem Timestamp, so lassen sich Read-Modelle auch in die Vergangenheit projizieren oder Simulationen mit alten Ständen betreiben. Auch ein Rollback im Falle eines Fehlers oder einer Simulation ist hier möglich.

Axon bietet Event Sourcing als festen Bestandteil und sogar in der Standardkonfiguration an. Jedes Event, das von der Domain emittiert wird, wird nicht nur an die Event Handler weitergeleitet, sondern auch in einem Event Store persistiert. In *Listing 5* wird die „load"-Methode des Repository aufgerufen, um das Aggregate, also das Domain Model, zu laden. Technisch wird hier direkt auf den Event Store zugegriffen und das Domain Model aus allen bisherigen Events aggregiert. Daher auch der Axon-Begriff „Aggregate" für das Domain Model. Damit nun aber das Domain Model aus den Events zusammengebaut werden kann, muss es seine eigenen Events konsumieren, um entscheiden zu können, wie es sich selbst aufbaut. *Listing 9* zeigt die Command-Methode „addAddress" und den Event Handler „handleAddressAdded".

Das Event Sourcing erklärt auch, warum in *Listing 5* das geänderte Aggregat nicht gespeichert wird. Durch den Aufruf von „addAddress" wird ein Event „AdressAdded" emittiert, das im Event Store gespeichert wird. Beim erneuten Laden wird dieses Event auf das Aggregat angewendet, um den aktuellen Zustand wiederherzustellen. Um nicht jedes Mal alle Events zu laden und zu verarbeiten, bietet Axon hier Abhilfe in Form von Snapshots.

Vor- und Nachteile

CQRS als Ganzes muss immer als ein Werkzeug in einem Werkzeugkasten verstanden werden. Nicht alle Probleme lassen sich mit CQRS lösen und selbstverständlich hat auch CRUD seine Daseinsberechtigung in einfachen Domänen ohne viel Fachlogik.

Den Aufwand der harten Trennung zwischen Domain Model und Read Model ist die größte Hürde, da man bei Änderungen gegebenenfalls zwei Modelle anpassen muss oder sogar zwei Persistenzschichten einbezieht.

Aber gerade wenn es darum geht, verschiedene Sichten auf eine Domäne zu haben, hat es seine Vorteile. Durch die sehr lose Kopplung forcieren Änderungen in der Domäne nur selten eine Query, sofern die Events abwärtskompatibel bleiben. Auch bietet Event Sourcing eine interessante Möglichkeit, Datenbestände zu versionieren und diese abfragbar zu machen. Mit Axon 4 bietet das Framework darüber hinaus auch einen externen EventBus, um verteilte Systeme aufzubauen.

Die Trennung von Domäne und Commands bietet ebenfalls eine interessante Entkopplung, da man „Nutzerwünsche" als solche im System abbilden kann, ohne direkt das eigentliche Datenmodell zu verändern und dieses in sich geschlossen zu halten.

Weitere und nicht behandelte Themen

Axon bietet über Command- und Eventhandler noch Unterstützung zu eventbasierten Queries, um die Abfragen noch weiter von den Modellen zu trennen.

Über Sagas können zusammenhängende Events betrachtet werden. Im Normalfall stehen Events immer für sich. In komplexen Geschäftstransaktionen können Events jedoch aufeinander aufbauen. Hier helfen Sagas als spezielle Form von Eventhandlern.

Weiterführende Quellen

Beispielprojekt: *https://gitlab.com/lwanzik/axon-spring*
http://www.laputan.org/mud/
https://martinfowler.com/bliki/CQRS.html
https://docs.axoniq.io/reference-guide/v/3.4/
https://axoniq.io/

[1] *Vgl. http://www.laputan.org/mud/*
[2] *Vgl. https://martinfowler.com/bliki/CQRS.html*

Christian Iwanzik
christian.iwanzik@gmail.com

Christian Iwanzik, 33 Jahre aus Siegburg, ist nun seit 2010 bei der tarent Solutions GmbH in Bonn tätig. Hauptsächlich auf der JVM arbeitet er mit Java, Kotlin und Scala und entwickelt moderne Microservices in Spring oder Play. Gerade die innere Architektur der Microservices liegt dabei mit Domain Driven Design mit CQRS oder hexagonaler Architektur im Fokus. Außerhalb des Codes arbeitet er beireits seit einigen Jahren in modernen Microservicearchitekturen und kümmert sich um die Intergration in CI/CD Systeme.

Objektorientierte Programmierung als bewusste Entscheidung

Torben Fojuth, neuland – Büro für Informatik

Java gilt als objektorientierte Programmiersprache – dies ist hinlänglich bekannt. Das Beherrschen der Syntax allein geht jedoch nicht automatisch mit einem Verständnis der zugrunde liegenden Paradigmen einher. Dass durch die bloße Wahl von Java als Programmiersprache automatisch auch objektorientierter Code entsteht, ist ein Irrtum.

Bei der objektorientierten Programmierung (nachfolgend kurz OOP genannt) handelt es sich um ein Programmierparadigma. Fragt man Wikipedia *[1]*, so ist OOP nur ein Paradigma unter vielen aus dem Bereich der imperativen Programmierung. Etwas kompakter dagegen ist die Sicht von Robert „Uncle Bob" Martin *[2]* auf die Welt der Programmierparadigmen. Aus seiner Sicht ist OOP eines von insgesamt drei fundamentalen Paradigmen:

1. objektorientierte Programmierung
2. funktionale Programmierung
3. strukturierte Programmierung

Für Martin definiert sich ein Paradigma vor allem durch das Beschränken der Möglichkeiten, die dem Entwickler bei der Program-

mierung zur Verfügung stehen. Im Falle der OOP wird die Möglichkeit entfernt, beliebige Funktionen aufzurufen, da Funktionen – in Form von Methoden – in der Idee der OOP im Kontext von Objekten existieren und entsprechend verwendet werden müssen.

Objektorientierte Programmiersprachen bieten in der Regel viele weitere Features wie beispielsweise Polymorphie und Vererbung. Diese Techniken sowie deren sachgemäße Verwendung werden aufgrund des engen Rahmens jedoch nicht behandelt. Es soll im Folgenden vielmehr um das Konzept der Bündelung von Daten sowie Operationen auf diesen Daten gehen, also um die Idee des Objektes. Im Kontrast dazu nun ein Beispiel *(Listing 1)* für nicht objektorientierten Code in Java.

```
public static void main(String[] arguments) {
  List<String> lines = readLinesFromFile();
  lines = sortLinesByFirstCharacter(lines);
  print(lines);
}
```

Listing1: Code ohne Anwendung von OOP

Rein prozeduraler Code in Java

Dass man in Java mit Leichtigkeit Code schreiben kann, der nichts mit OOP zu tun hat, zeigt *Listing 1*. Man erkennt auf den ersten Blick, unter anderem am fehlenden Schlüsselwort new, dass OOP hier keine Rolle spielt. Es werden ausschließlich statische Hilfsmethoden ohne einen umgebenden Objektkontext aufgerufen. So handelt es sich zwar um imperativen, nicht aber um objektorientierten Code.

Listing 1 ist bewusst darauf getrimmt, rein imperativen Code zu zeigen. Bei der täglichen Arbeit an echtem Produktiv-Code fällt es etwas schwerer zu erkennen, wo Chancen für objektorientierte Modellierung verpasst wurden.

Welche Vorteile eine objektorientierte Modellierung mit sich bringt und welche potenziellen Gefahren der Verzicht von OOP birgt, soll nun näher betrachtet werden.

Geschäftslogik außerhalb des Objektes

In *Listing 2* folgt ein weiteres, weniger offensichtliches Beispiel für nicht konsequent objektorientierten Code, der aus einem regulären Softwareprojekt stammen könnte:

Es geht um einen Service, der einen Artikel zu einem Warenkorb hinzufügt. Obiger Code enthält viel Potenzial für Verbesserungen, es kommt aber zunächst auf die enthaltene Geschäftslogik an: Offenbar ist es in diesem System wichtig, dass der Gesamtwert eines Warenkorbs 500 Währungseinheiten nicht übersteigen darf. Konsequenterweise findet sich diese Regel auch im Code wieder. Worin besteht also die Kritik?

Das Problem an der gezeigten Implementierung besteht darin, dass diese Geschäftsregel außerhalb des Objektes Cart implementiert ist. Verwendet ein Mitarbeiter nicht diesen CartService zum Hinzufügen des Artikels, sondern beispielsweise direkt den Cart, umgeht er oder sie ungewollt die obige Regel und könnte so größere Warenkörbe erstellen als vorgesehen.

Geschäftslogik in das Objekt verschieben

Die Lösung des Problems besteht darin, diese Logik in das Objekt Cart zu verlagern, wie in *Listing 3* gezeigt. Durch diese Maßnahme ist gewährleistet, dass die Logik an der richtigen Stelle steht und der Warenkorb stets in sich konsistent bleibt.

Mit dem Umbau gehen ganz automatisch weitere Änderungen einher, die der Idee der OOP entgegenkommen. So ist in der neuen Variante zum einen das Warenkorblimit nun Bestandteil der richtigen Klasse Cart - zuvor war die Konstante außerhalb definiert. Zum anderen muss der Warenkorb (zumindest zu diesem Zweck) nun keine Methode zum Abfragen seines aktuellen Gesamtwerts mehr bereitstellen und kann so seine Interna für sich behalten.

Beide Veränderungen sorgen dafür, dass Logik in die Objekte hineinwandert, statt außerhalb implementiert zu werden. Dieses Grundprinzip der OOP ist auch unter dem Namen „Tell, don't ask" *[3]* bekannt und besagt, dass man Objekten per Methodenaufruf sagen soll, was sie tun sollen, statt sie nach ihren Daten zu fragen, um es selbst zu tun:

„Procedural code gets information then makes decisions. Object-oriented code tells objects to do things."
— Alec Sharp

```
public void addItemToCart(Item item, CartId cartId) {
  Cart cart = loadCart(cartId);
  if(cart.total() + item.price() > 500.00f){
     throw new CartLimitExceeded();
  }
  cart.addItem(item);
}
```

Listing 2: Hinzufügen eines Artikels zum Warenkorb

```
public void addItemToCart(Item item, Cart cart) {
  Cart cart = loadCart(cartId);
  cart.addItem(item);
}

public class Cart {
  private static final Float LIMIT = 500.00f;

  List<Item> items = new ArrayList<>();

  public void addItem(Item item) {
     if(this.total() + item.price() > LIMIT){
        throw new CartLimitExceeded();
     }
     items.add(item);
  }
}
```

Listing 3: Domänenlogik liegt nun innerhalb des Cart-Objekts

```
public class Cart {
  private static final Money LIMIT = new Money(500, 0, EURO);
  ...
  public void addItem(Item item) {
     if(this.total().plus(item.price()).exceeds(LIMIT)){
        throw new CartLimitExceeded();
     }
     items.add(item);
  }
```

Listing 4: Währungsbeträge als Value Objects

```
public void processOrder(Order order){
  String orderId = order.id();
  if(orderId.endsWith("-23")){
     germanLogisitcs.process(order);
  } else {
     austrianLogistics.process(order);
  }
}
```

Listing 5: Bestellnummern als String

```
public void transferOrder(Order order){
  OrderId orderId = order.id();
  if(orderId.isGermanOrder()){
     germanLogisitcs.process(order);
  } else {
     austrianLogistics.process(order);
  }
}
```

Listing 6: Die Bestellnummer als Value Object

```
public void updateAverageCartValueForCurrentMonth(){
  Orders orders = ordersCurrentMonth();
  AverageCartValue average = new AverageCartValue(orders);
  monitoring.updateAverageCartValueKPI(average.toMon-
ey());
}

public class AvergageCartValue(){
  public AverageCartValue(Orders orders) {
     this.orders = orders;
  }
  public Money toMoney(){
     Money sum = new Money(0,0, EUR);
     orders.forEach(order -> sum = order.addTotalTo(sum));
     Money average = sum.divideBy(orders.size());
     return average;
  }
}
```

Listing 7: Durchschnittlicher Warenkorbwert als Klasse

Von allen Prinzipien zum Erstellen von gut wartbarem objektorientierten Code ist dieses Prinzip sicher eines der wichtigsten.

Werden, bei vorgeblich objektorientiertem Code, Prinzipien der OOP konsequent nicht beachtet, spricht man in der Softwareentwicklung von sogenannten Anti-Patterns. Das Anti-Pattern für die Nichtbeachtung des „Tell, don't ask"-Prinzips wird als anämisches Datenmodell *[4]* bezeichnet. In einem anämischen Datenmodell besteht ein Großteil der Klassen lediglich aus Feldern sowie deren Settern und Gettern. Sämtliche Logik ist hingegen außerhalb der Klassen implementiert. Diese Klassen stellen also reine Daten-Konstrukte dar und verdienen es nicht, als Objekte im Sinne der OOP bezeichnet zu werden. Beim Antreffen eines solchen anämischen Datenmodells in einem Projekt ist höchste Vorsicht geboten. Bevor ein Beispiel diese Gefahr erläutert, soll aber zunächst der Begriff der Value Objects eingeführt werden.

Werte als Objekte: Value Objects

Zurück zum Code rund um den Warenkorb: Nicht nur Dinge wie der Warenkorb, die einem Lebenszyklus in Form von Zustandsänderungen unterliegen, können und sollten als Objekte modelliert werden. Ebenso gewinnen Eigenschaften von Objekten an Bedeutung und Sicherheit, wenn sie als Objekte modelliert werden. In obigem Beispiel könnten etwa die Währungsbeträge als Objekte mit eigenen, sprechenden Methoden dargestellt werden. Diese Art von Objekten wird auch Value Object genannt *(siehe [5] und [6])*. *Listing 4* zeigt den Code nach einem weiteren Umbau.

Welcher Vorteil ergibt sich daraus? Nun, speziell über das Thema Währungsbeträge und deren Modellierung mit primitiven Typen wurde – unter anderem in *[7]* – schon viel geschrieben. Im Kern bietet die Modellierung mit Value Objects vor allem folgende Vorteile:

1. die Absicherung der Wertebereiche
2. die Kontrolle über verfügbare Operationen
3. ein zentraler Ort für Geschäftslogik, die sich auf den Wert bezieht

In obigem Kontext sind negative Zahlen für Preise nicht sehr sinnvoll. Dies kann in einer konkreten Implementierung direkt im Konstruktor sichergestellt werden, sodass es nicht mehr möglich ist, in diesem Sinne ungültige Währungsbeträge zu erstellen. Als Entwickler/in sollte man sich vor der Verwendung von primitiven Typen stets die Frage stellen: Stimmt der fachliche Wertebereich mit dem technischen Wertebereich des verwendeten Typs überein? Ist dies nicht der Fall, ist eine Modellierung des Wertes als Value Object sinnvoll.

Des Weiteren sind insbesondere die Addition und Subtraktion von Fließkommazahlen nie exakt. Das in *Listing 4* gezeigte Value Object Money verlangt deshalb separate, ganzzahlige Werte für den Vor- und den Nachkommateil des Preises und definiert eine eigene Methode für die Addition, um sicherzustellen, dass die Operation stets exakt funktioniert. Glücklicherweise steht dem JDK mit *[8]* mittlerweile ein Modul zur Verfügung, das keine Wünsche im Umgang mit Währungsbeträgen offen lässt. Probleme bei der Modellierung von Währungsbeträgen gehören damit der Vergangenheit an.

Beim dritten Vorteil der Value Objects soll es, wie versprochen, noch einmal um die Gefahren des anämischen Datenmodells gehen. Angenommen in einem System existieren zwei sogenannte Mandanten: Deutschland und Österreich. Beide haben unterschiedliche Arten von Bestellnummern, die durch ein magisches Kürzel am Ende der Bestellnummer („-23" für Deutschland und „-42" für Österreich) gekennzeichnet sind. Das System muss an verschiedenen Stellen zwischen beiden Versionen unterscheiden, so beispielsweise beim Übergeben der Bestellung an die Logistik.

In obigem Beispiel ist erneut Geschäftslogik aus dem Modell „herausgesickert" und wird stattdessen vom Benutzer der Klasse implementiert. Das Wissen zur Berechnung der Mandanten befindet sich nun an jeder Stelle im System, die diese Unterscheidung benötigt. Eventuell geschieht dies sogar in leicht abgewandelter Form, je nachdem, wer es implementiert hat, was das Auffinden und Erkennen dieser Stellen zusätzlich erschwert.

Was passiert nun, wenn sich die Kürzel ändern oder die Schweiz als weiterer Mandant mit eigenem Kürzel eingeführt wird? Es muss jede dieser Stellen gefunden und korrekt angepasst werden. Ein solches Vorgehen ist nicht nur sehr aufwendig, sondern gleichzeitig auch fehleranfällig.

Modelliert man hingegen die Bestellnummer als Value Object, so wird die Logik an genau einer Stelle implementiert: der Bestellnummer selbst *(siehe Listing 6)*.

Modellierung abstrakter fachlicher Konzepte als Objekt

Bei der Modellierung eines Geschäftsfeldes durch Klassen sollte man sich, gerade mit Blick auf das „Tell, don't ask"-Prinzip, nicht darauf beschränken, ausschließlich greifbare Subjekte der Domäne wie beispielsweise den Kunden, den Warenkorb oder den Artikel zu modellieren.

Das Modell gewinnt durch die Modellierung abstrakter fachlicher Konzepte, wie etwa spezieller Kennzahlen, an Bedeutung. So könnte der durchschnittliche Warenwert der Warenkörbe pro Monat eine wichtige Kennzahl für ein Unternehmen sein. *Listing 7* zeigt, wie dies aussehen könnte.

Einmal für diesen Gedanken sensibilisiert, lassen sich in fast jeder Domäne weitere Konzepte identifizieren, die einer separaten Modellierung würdig sind. Weitere Inspiration sowie einige streitbare Thesen zur Implementierung von Klassen finden sich in dem empfehlenswerten Erstlingswerk von Yegor Bugayenko *[10]*.

Fazit

Die Java-Syntax und die Klassen des JDK zu kennen, sind eine solide Basis für eine Tätigkeit als Softwareentwickler/in. Will man das Handwerk der Softwareentwicklung professionell betreiben – und dabei OOP einsetzen –, gibt es Konzepte hinter dem Code, die man sich aneignen sollte.

Neben den Referenzen, die schon im Verlauf des Artikels erwähnt wurden, ist ein Verständnis der SOLID-Prinzipien *[11]* ein guter Schritt. Ebenso gibt es – subjektiv betrachtet – Standardwerke, die man gelesen haben sollte. Zu diesen zählen meiner Meinung nach mindestens „Clean Code" *[12]*, das POODR-Buch *[13]* sowie das GOOS-Buch *[14]*.

Diese Bücher eignen sich auch hervorragend für eine kooperative Lektüre, beispielsweise in Form einer Leserunde unter Kollegen und Kolleginnen. Um das neu erworbene Wissen zu verstetigen, haben sich die Ideen des gemeinsamen Code Review als Termin mit dem gesamten Team sowie Peer Reviews (falls das Team mit Merge Requests arbeitet) als hilfreich erwiesen.

Abschließend noch ein Hinweis zur Einordnung der in diesem Artikel behandelten Themen: Der Rahmen eines Artikels ermöglicht natürlich keine erschöpfende Behandlung aller Themen aus dem Themenkomplex der OOP. Des Weiteren bleiben Themen unerwähnt, die auf den höheren Abstraktionsebenen liegen oder einen gänzlich eigenen Themenkomplex darstellen:

- Organisation von Klassen innerhalb einer Anwendung
- Organisation von Anwendungen innerhalb einer System-Infrastruktur
- Modellierung des Geschäftsfeldes mithilfe von Domain-driven Design

Der Artikel vermittelt einen ersten Eindruck davon, wie viel es über objektorientierte Programmierung, abseits von Syntax und Sprachfeatures, zu lernen gibt. Bei Fragen oder Anmerkungen kontaktieren Sie mich gern.

Literaturverzeichnis

[1] *https://de.wikipedia.org/wiki/Programmierparadigma*
[2] Robert C. Martin, Three Paradigms, 2012, *https://blog.cleancoder.com/uncle-bob/2012/12/19/Three-Paradigms.html*
[3] *https://pragprog.com/articles/tell-dont-ask*
[4] Martin Fowler, , Stand 2018, *https://www.martinfowler.com/bliki/AnemicDomainModel.html*
[5] Eric J. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003
[6] Vaughn Vernon, Implementing Domain-Driven Design, 2013
[7] Joschua Bloch, Effective Java, 28.5.2008
[8] *https://jcp.org/en/jsr/detail?id=354*
[9] *https://dzone.com/articles/solid-principles-by-examples-liskov-substitution-p*
[10] Yegor Bugayenko, Elegant Objects, 2016
[11] 2018, *https://en.wikipedia.org/wiki/SOLID*
[12] Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 2008
[13] Nat Pryce, Steve Freeman, Growing Object-Oriented Software, Guided by Tests , 2009
[14] Sandi Metz, Practical Object-Oriented Design in Ruby: An Agile Primer, 2012

Torben Fojuth
torben.fojuth@posteo.de

Seit seinem Abschlss an der Universität Bremen in 2007 arbeitet Torben Fojuth als Softwarentwickler im Web-Umfeld. Aktuell ist er bei der "neuland - Büro für Informatik GmbH" als Softwareentwickler und Ausbilder tätig. Seine Schwerpunkte liegen in den Bereichen Softwarearchitektur, Clean Code und Domain-driven Design. Seine Rolle in der Lehre und Fortbildung ist eine gute Ergänzung zur Projektarbeit.

BootsFaces: JavaServer Faces in Zeiten von Angular, React und Co.

Stephan Rauh, Opitz Consulting Deutschland GmbH

Wir schreiben das Jahr 2019: Alle Welt verwendet moderne JavaScript-Frameworks für Benutzeroberflächen. JavaServer-Faces(JFS)-Entwickler werden nur müde belächelt oder gar bedauert. JSF ist aus der Mode gekommen. Es gilt als altbacken, kompliziert und außerdem noch langsam. Genau in dieser Zeit tritt unser Open-Source-Projekt BootsFaces an, die Lücke zu schließen und Java-Entwicklern das Tor zu modernen, responsiven UIs aufzustoßen.

Allen Unkenrufen zum Trotz kennen die Downloadzahlen von BootsFaces nur eine Richtung: Sie stürmen nach oben – und das schon seit Jahren. Das Gleiche gilt für die Downloadzahlen von OmniFaces, die in unregelmäßigen Abständen veröffentlicht werden *[1]*. Bei den Kollegen vom PrimeFaces-Team frage ich nur unregelmäßig nach, sie sind aber erfolgreich genug, um von ihrer Cash-Cow PrimeFaces komfortabel zu leben und gleichzeitig in die JavaScript-Welt zu expandieren: PrimeNG, PrimeReact und seit Neuestem auch PrimeVue.

Nichtsdestotrotz wirken die Downloadzahlen geradezu vernachlässigbar, wenn wir sie mit Angular (sechs Millionen Downloads pro Monat) oder React.js (elf Millionen Downloads pro Monat) vergleichen. BootsFaces kommt auf rund 3.000 Downloads pro Monat. Selbst wenn man unterstellt, dass npm-stat.com anders zählt als Maven Central, ist der Unterschied beeindruckend. Warum ist BootsFaces trotzdem interessant? Was motiviert die Mannschaft hinter BootsFaces, weiterzumachen? Welche Antworten hat BootsFaces auf die Herausforderungen Angular, React und Vue.js?

Feedback der Community

Die zweite Frage ist schnell beantwortet: Die BootsFaces-Community motiviert uns immer wieder aufs Neue. Wir bekommen eine Menge positives Feedback, und das macht Spaß. Noch dazu, wenn es von Leuten kommt, die wir ansonsten nie kennengelernt hätten. Manchmal gewinnen wir dadurch sogar Einblick in fremde Kulturen

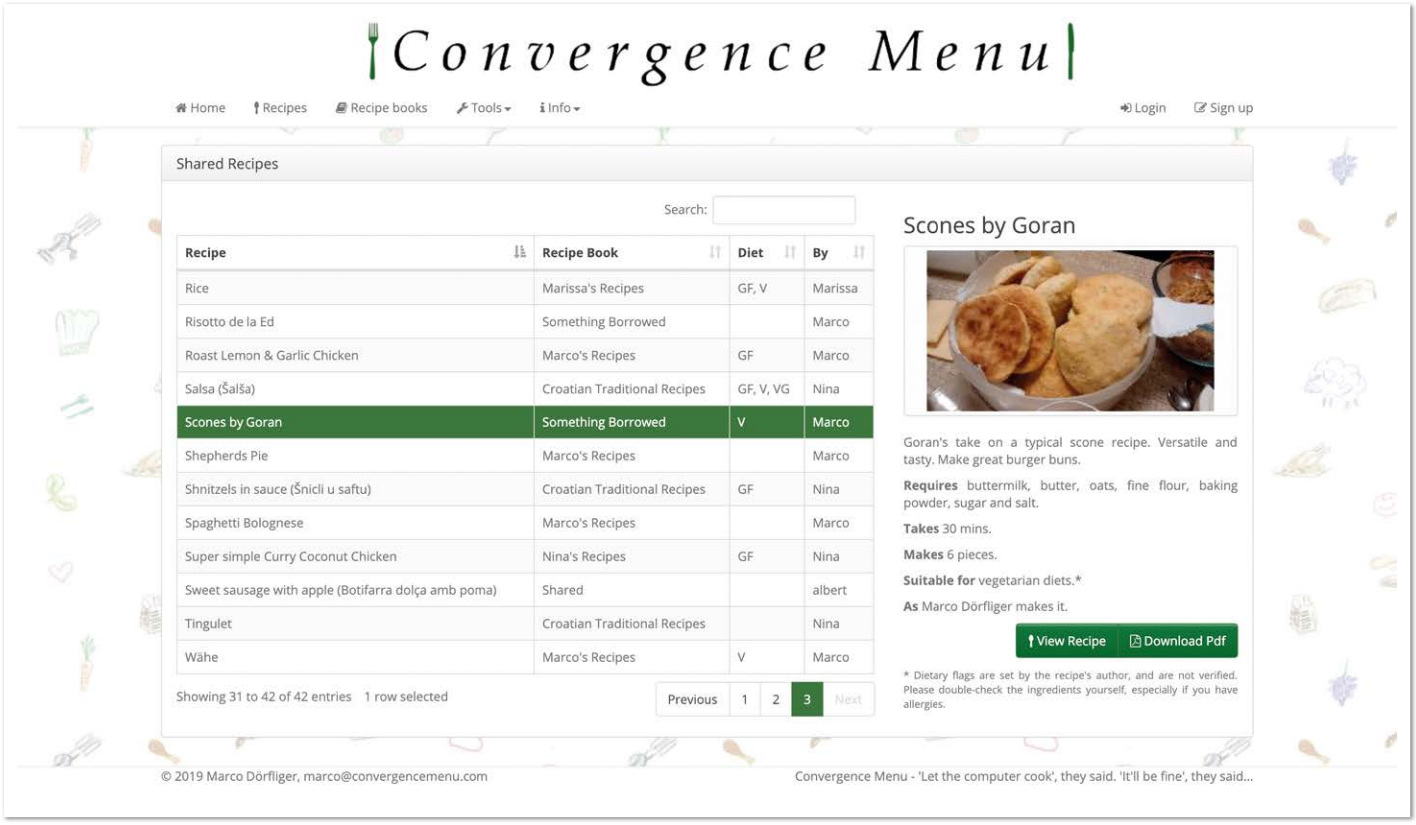


Abbildung 1: Das Convergence Menu von Marco Dörfliger zeigt viele populäre Elemente von BootsFaces, wie z.B. Layout, Tabellen, Menüs

oder sehen Projekte, die mit unserem Framework realisiert wurden. Das ist und bleibt spannend!
Für die Beantwortung der zweiten Frage möchte ich ein wenig ausholen.

Was ist BootsFaces?

BootsFaces ermöglicht es JSF-Entwicklern, das Layout-Framework Bootstrap zu verwenden. Bootstrap wiederum ist ein CSS-Framework, das mithilfe einiger Ideen aus dem Buchdruck beinahe automatisch für gutaussehende Webseiten und Webanwendungen sorgt. Wer einmal versucht hat, eine Visual-Basic-Anwendung Pixel für Pixel sauber auszurichten, weiß die Vorteile von Bootstrap zu schätzen.

Streng genommen ist BootsFaces spätestens seit JSF 2.2 überflüssig. CSS-Klassen und natives HTML konnte man schon immer in JSF verwenden, und seit JSF 2.2 können die nativen Eingabefelder von HTML auch in einer JSF-Anwendung eingesetzt werden („Pass-Through-Elemente"). Ich habe diesen Ansatz in einem großen Projekt benutzt. Der große Vorteil ist die Flexibilität. Wir konnten beliebige JavaScript-Komponenten aus dem Internet in unserer Anwendung verwenden. Das Ergebnis war eine schnelle und gutaussehende Anwendung mit ansprechender Ergonomie.

Der Nachteil war, dass unsere Quelltexte sehr umfangreich wurden. Sie waren auch nicht leicht zu lesen. Bootstrap nutzt sehr viele -Tags, die sich nur durch eine CSS-Klasse weiter hinten in der Zeile unterscheiden. Komplexere Elemente bestehen aus einer ganzen Reihe von HTML-Tags, die erst in der Kombination als modaler Dialog oder Reiterkarten erkennbar werden *(siehe Listing 1)*.

Mission von BootsFaces

Spätestens seit JSF 2.2 lautet unser Motto daher: JSF mit Bootstrap soll einfacher werden. Wenn wir schon dabei sind: Auch JSF selbst soll einfacher werden, egal ob mit oder ohne Bootstrap. BootsFaces bietet einen einfacheren Zugang zu AJAX, einen ganzen Schwung neuer Search Expressions, eine optionale, HTML-artige Syntax und vieles mehr. Natürlich auch eine Reihe von UI-Komponenten. Schauen wir uns unser Beispiel in der BootsFaces-Variante an *(siehe Listing 2)*.

Acht Zeilen anstelle der ursprünglich benötigten 24 Zeilen sind ein deutlicher Vorteil. Noch wichtiger: Diese acht Zeilen zeigen die Rolle des jeweiligen Tags gleich am Anfang der Zeile, statt sie – wie in der HTML-Lösung – in den class-Attributen zu verstecken. Die verbesserte Lesbarkeit kommt Entwicklern in der Wartungsphase ihrer Anwendung zugute. Die meisten Programme werden nur einmal geschrieben, aber später während einer oft hektischen Fehlersuche häufig gelesen. Deswegen legen wir großen Wert auf die Lesbarkeit der Quelltexte.

UI-Komponentenbibliothek

Die Komponentenbibliothek von BootsFaces umfasst 82 Widgets. Wie wir gerade gesehen haben, ersparen viele dieser Widgets Ihnen eine Menge Schreibarbeit, indem sie aus kompaktem XHTML-Code mehrere Zeilen HTML und JavaScript erzeugen. Gleichzeitig ist es eine Besonderheit von BootsFaces, dass es – im Vergleich zu anderen JSF-Komponentenbibliotheken – relativ kompakten HTML-Code generiert. BootsFaces kann sich das leisten, weil es jünger ist als die meisten Mitbewerber. Wir setzen konsequent auf HTML5 und können damit viele Features nutzen, die anderen JSF-Frameworks aus Kompatibilitätsgründen verwehrt bleiben.

```
<div class="tab-panel" role="tabpanel">
  <ul class="nav nav-tabs" role="tablist">
     <li role="presentation" class="active">
        <a role="tab" data-toggle="tab"
           href="#j_id_c_pane">
           First tab
        </a>
     </li>
     <li role="presentation">
        <a role="tab" data-toggle="tab"
           href="#j_id_e_pane">
           Another tab
        </a>
     </li>
  </ul>
  <div class="tab-content" role="tablist">
     <div id="j_id_c_pane" class="tab-pane active">
        First tab
     </div>
     <div id="j_id_e_pane" class="tab-pane">
        Second tab
     </div>
  </div>
</div>
```

Listing 1

```
<b:tabView>
  <b:tab title="First tab">
     First tab
  </b:tab>
  <b:tab title="Another tab">
     Second tab
  </b:tab>
</b:tabView>
```

Listing 2

```
<div class="container">
  <div class="row">
     <div class="col-sm-4">left-hand-side column</div>
     <div class="col-sm-8">right-hand-side column</div>
  </div>
</div>
```

Listing 3

```
<b:container>
  <b:row>
     <b:column colSm="4">
        left-hand-side column
     </b:column>
     <b:column small-screen="two-thirds">
        right-hand-side column
     </b:column>
  </b:row>
</b:container>
```

Listing 4

Layout-Komponenten

Historisch gesehen bilden die Layout-Komponenten den Kern von BootsFaces. Die meisten Bootstrap-Seiten bestehen aus Containern, Zeilen und Spalten. Ein einfaches HTML-Layout sieht mit Bootstrap wie in *Listing 3* beschrieben aus.

BootsFaces übernimmt dieses Schema und verbessert die Lesbar-

keit durch spezielle Tags. Sie sehen auf den ersten Blick, ob es sich um einen Container, eine Spalte oder eine Zeile handelt. Wenn Sie wollen, können Sie die etwas kryptischen Ausdrücke „sm-4“ durch Klartext ersetzen. Das nächste Beispiel zeigt in der oberen Hälfte die kompakte Schreibweise, die einen erfahrenen Webdesigner anspricht, und in der unteren Hälfte die wortreiche Syntax, die neuen Mitgliedern in Ihrem Team das Leben erleichtert *(siehe Listing 4)*.

Variationen der Syntax

Es hat ein paar Jahre gedauert, bis die Entwicklergemeinschaft die neuen Variationen der Syntax angenommen hat. Das Beispiel zeigt drei der neuen Möglichkeiten:

- BootsFaces erlaubt den Bindestrich als Alternative zum Binnenmajuskel. Einfacher formuliert: Viele JSF-Attribute bestehen aus mehreren Wörtern. Wie in Java weithin üblich, werden die Wortgrenzen durch einen Großbuchstaben abgegrenzt. Im Beispiel wäre das also „„smallScreen“ oder (weniger leicht ersichtlich) „colSm“. Wenn Sie viel mit Bootstrap oder generell viel mit CSS arbeiten, kennen Sie auch eine andere Konvention: Wortgrenzen werden in CSS-Klassen durch einen Bindestrich markiert. Das stellt einen BootsFaces-Entwickler vor eine unnötige Hürde. Mal gilt der Bindestrich, mal die Großschreibung. Es gibt eine klare, einfache Regel dafür – aber warum sollten wir unsere Anwender zwingen, sie zu lernen? Unsere Antwort: Wir erlauben einfach beides.
- In eine ähnliche Richtung geht die Vereinfachung des Attributnamens. Im Prinzip hat Bootstrap mit den T-Shirt-Größen schon eine ziemlich raffinierte Metapher gefunden. Wir bieten mit „small-screen“ trotzdem eine – hoffentlich – ausdrucksstärkere Alternative zu „col-sm“ an.
- Für den Wert der Spaltenbreite können Sie wahlweise eine Zahl oder einen englischen Klartext angeben. Unsere Idee dahinter ist, dass es für einen Laien sofort klar ist, was „one-third“ oder „half“ bedeutet. Die traditionelle Angabe „col-sm-4“ bzw. „col-sm-6“ erfordert etwas mehr Einarbeitung. Wie immer im Leben gibt es natürlich auch die andere Sichtweise: Erfahrene Web-Designer fühlen sich wahrscheinlich wohler mit der kompakten Schreibweise, die sie schon seit Jahren kennen. Also erlauben wir auch in diesem Fall beides.

Die Flexibilität von BootsFaces hat leider eine Kehrseite. Bisher haben wir keine der großen IDEs Eclipse, IntelliJ und NetBeans überzeugen können, speziellen Support für BootsFaces anzubieten. Daher werden fast alle Attribute der Komponenten doppelt angezeigt – einmal mit Bindestrich und einmal mit Großbuchstaben. Das hat in der Vergangenheit schon einige Entwickler verwirrt.

```
<container>
  <row>
     <column small-screen="one-third">
        left-hand-side column
     </column>
     <column small-screen="two-thirds">
        right-hand-side column
     </column>
  </row>
</container>
```

Listing 5

HTML-artige Syntax

Die fehlende Unterstützung der IDEs verhindert womöglich auch den Durchbruch der HTML-artigen Syntax. BootsFaces erkennt auch Komponenten ohne das Präfix „b:“ als BootsFaces-Komponente. Unser Beispiel wird damit noch eine Kleinigkeit kompakter und ausdrucksstärker *(siehe Listing 5)*.

Leider markieren die meisten IDEs diesen Quelltext als fehlerhaft und bieten keine Codevervollständigung mehr an. Das wiederum hindert die meisten Entwickler daran, das Feature zu verwenden.

Vielleicht ist es aber auch einfach so, dass die meisten Entwickler meine Allergie gegen XML nicht teilen. An und für sich würde ich auch gerne eine kompaktere, weniger zeremonielle Alternative zu XHTML anbieten. Beispielsweise könnte die Liste der Namespaces am Beginn jeder XHTML-Datei wegfallen, wenn man mit Default-Werten für die gängigen Namespaces arbeitet. Der JSF-Standard würde das durchaus erlauben. Leider ist es sehr schwierig, die JSF-Basisbibliotheken Mojarra und MyFaces entsprechend anzupassen – vor allem, wenn die Erweiterung mit beiden Bibliotheken funktionieren soll.

Zusätzliche Breakpoints

Der große Vorteil eines Frameworks wie Bootstrap ist, dass sich Anwendungen dynamisch an unterschiedliche Bildschirmgrößen anpassen. Eine BootsFaces-Anwendung kann je nach Displaygröße unterschiedlich aussehen. Wir haben dafür schon ein Beispiel gesehen.

Dieses Codefragment *(siehe Listing 6)* stellt die Seite in zwei nebeneinanderliegenden Spalten dar. Aber nur auf großen Bildschirmen. Genauer gesagt, nur auf Bildschirmen ab dem Breakpoint „sm“. Bei Bootstrap 3 (und damit bei BootsFaces 1.x) sind das 768 Pixel, bei Bootstrap 4 576 Pixel. Bei beiden Versionen von Bootstrap liegt der größte Breakpoint bei 1200 Pixeln.

So genial dieses Konzept ist, hat es doch einen großen Nachteil. Bisher haben alle meine Anwendungen andere Breakpoints erfordert, als sie von Bootstrap angeboten werden. Das liegt daran, dass Bootstrap hauptsächlich für mobile Geräte entworfen wurde – und zudem für mobile Geräte, die inzwischen vom Markt verschwunden sind.

Als JSF-Framework richtet sich BootsFaces in erster Linie an Desktop-Anwendungen. Dort sind mittlerweile viel größere Bildschirme üblich. Bei Laptops findet sich häufig die Breite 1440 Pixel. Full-HD (1920 Pixel) gehört längst zum Mainstream und seit einigen Jahren sind 4K-Bildschirme en vogue.

Ich würde mir daher ein paar zusätzliche und vor allem praxisnähere

```
<column small-screen="one-third">
  left-hand-side column
</column>
<column small-screen="two-thirds">
  right-hand-side column
</column>
```

Listing 6

Breakpoints wünschen. Material Design macht es vor: Dort gibt es Breakpoints bei 480, 600, 840, 960, 1280, 1440, 1600 und 1920 Pixeln. Ich würde noch 2560 Pixel und die 4K-Auflösung 3840 Pixel hinzunehmen.

Als einer der Committer von BootsFaces brauche ich es nicht bei Wünschen zu belassen. Ich kann die Dinge auch umsetzen. Als ersten Schritt habe ich den Breakpoint „XXL" mit 1400 Pixeln hinzugefügt. Seitdem warte ich vergeblich auf das Feedback der Community. Offensichtlich ist das Problem nicht so drängend, wie ich dachte. Oder wir haben es in der Dokumentation zu gut versteckt. Wie dem auch sei, in einer der nächsten Versionen plane ich, die Breakpoints „XXXL" mit 1600 Pixeln, „full-hd" mit 1920 Pixeln und eventuell noch „QHD" mit 2560 Pixeln sowie „4K" mit 3840 Pixeln zu ergänzen.

AJAX

Die ersten Versionen von JSF unterstützen AJAX noch nicht. Das kam erst relativ spät dazu. Der gewählte Ansatz in *Listing 7* ist sehr flexibel, aber auch sehr wortreich.

Das muss doch auch einfacher gehen!

BootsFaces greift ein paar Ideen von PrimeFaces und Angular auf und entwickelt sie konsequent weiter. Der AJAX-Aufruf wird jetzt direkt am zugehörigen JavaScript-Event definiert. Das ist kompakter, verbessert die Lesbarkeit und kommt damit den Kollegen, die Ihr Projekt in einigen Jahren pflegen müssen, entgegen *(siehe Listing 8)*.

In eine ähnliche Kerbe schlägt das Attribut „auto-update". Jedes Bildschirmelement mit diesem Attribut wird automatisch bei jedem AJAX-Request neu gezeichnet. Typische Beispiele sind die Fehlermeldungen in der Maske oder Eingabefelder, die rot markiert werden, wenn die Eingabe fehlerhaft war.

Automatic AJAX?

Es gibt auch ein paar Dinge, an denen wir gescheitert sind. Bei JSF muss der Programmierer explizit angeben, was nach einem AJAX Request neu gezeichnet wird. Das war seinerzeit ein genialer Schachzug, um AJAX nachträglich zu JSF zu ergänzen. Gleichzeitig ist das auch eine große Fehlerquelle. Daher stellt sich die Frage, ob es nicht einfacher geht. Computer sind sehr gut darin, solche Veränderungen zu erkennen. Viel besser als wir Menschen. Also sollten wir dem Computer diese Aufgabe überlassen.

Dieser Ansatz funktioniert. Die JSF-Bibliothek ICEFaces hat dieses Feature seinerzeit implementiert und ich habe es auch selbst schon erfolgreich viele Jahre in produktiven Anwendungen verwendet. Damals habe ich ein UI-Framework entwickelt, dass JSF in vielerlei Hinsicht verblüffend ähnelt. Nur, dass es etliche Jahre früher entwickelt wurde und wir nie die Chance hatten, es in der Open-Source-Gemeinde zu veröffentlichen.

Vor ein paar Jahren habe ich gemeinsam mit Thomas Andraschko dieses „automatic AJAX" mit unserer Bibliothek BabbageFaces in die JSF-Welt übertragen. Das Projekt erregte einiges Aufsehen, in der Praxis funktionierte es aber nie richtig. Es erfordert, dass man extrem sauberen HTML-Code schreibt. JSF ist ein XML-Dialekt und zwingt damit schon zu einem unnatürlich sauberen Programmierstil. Aber noch nicht einmal das genügte den Ansprüchen unseres Vergleichsalgorithmus. Sehr schade. Ich hätte diese Idee sehr gerne in BootsFaces verwendet und es den JSF-Entwicklern damit erspart, manuell die richtige ID für das Update der AJAX-Requests zu definieren.

JSF soll einfacher werden!

Anhand dieser Beispiele wird deutlich, was wir vom BootsFaces-Team mit unserem Ziel „JSF soll einfacher werden!" meinen. Wir haben uns über viele Eigenheiten von JSF in jahrelanger täglicher Arbeit geärgert und beschlossen, aktiv etwas zu tun. Unsere Hoffnung ist, diese Ideen später auch in den JSF-Standard einfließen zu lassen. Bei einem Thema ist uns das auch gelungen. Thomas Andraschko vom PrimeFaces-Team hat sich von unseren „Search Expressions" anspornen lassen, das Feature in den JSF-Standard einzubringen. Seit JSF 2.3 profitiert die ganze JSF-Community davon – und nicht mehr nur die Anwender von PrimeFaces und BootsFaces.

Search Expressions

Ursprünglich waren die Search Expressions ein Feature von PrimeFaces. Ich hätte die Search Expressions von PrimeFaces gerne direkt für BootsFaces verwendet. Das hätte gut gepasst, weil ich seinerzeit BootsFaces als reines Layout-Framework definieren wollte, das quasi als Erweiterung von PrimeFaces genutzt werden kann. Beides ist anders gekommen. Kaum ein Architekt ist bereit, BootsFaces und PrimeFaces im gleichen Projekt einzusetzen. Wir konnten die Search-Expressions-Engine aus lizenzrechtlichen Gründen nicht ohne Weiteres verwenden.

Die Alternative war, selbst ein Search-Expressions-Framework zu schreiben, dafür zu sorgen, dass das API zu PrimeFaces kompatibel ist, und einen deutlich größeren Funktionsumfang anzubieten. Das passte insofern gut in meine Strategie, als ich ohnehin ein paar neue Search Expressions geplant hatte:

- `@next` ist sehr praktisch, um die Fehlermeldung hinter dem Eingabefeld zu referenzieren
- `@previous` zeigt (in vielen Anwendungen) auf das Label vor dem Eingabefeld
- `@**:id` erlaubt eine rekursive Suche nach einer ID – im Prinzip wie im Dateisystem von Unix. JSF hat die Besonderheit, dass jedes Bildschirmelement eine ID hat und dass diese ID vom JSF-Framework verändert wird. Je nachdem, ob das Element in einer Tabelle, einem Reiter oder einem Formular steht, wird ein Präfix vor die ID, die wir Programmierer definiert haben, gesetzt. Die Wildcard-Suche löst das Problem. „@**:myField" findet alle Felder mit der JSF-ID „myField".

```
<h:commandButton value="Hover over me!">
  <f:ajax render="@form :messages"
     event="mouseover"
        listener="#{AJAXBean.hoverAction}"/>
</h:commandButton>
```

Listing 7

```
<b:commandButton  value="Hover over me!"
           update="@form :message"
           onmouseover="ajax:AJAXBean.hoverAction()"/>
```

Listing 8

Reality Check

Aus meiner Sicht sind die neuen Search Expressions ein großer Erfolg. Sie lösen ein Problem, mit dem ich als JSF-Entwickler massiv zu kämpfen hatte. Jedes Mal, wenn ich JSF-Code kopiert oder verschoben hatte, musste ich einige sehr technische IDs ändern. Üblicherweise in einem Moment, in dem ich den Kopf nicht wirklich für solche technischen Imponderabilien frei hatte. `@next`, `@previous` und `@**:id` lösen das Problem und, wie gesagt, ein großer Teil dieses Features hat sogar den Weg in den JSF-Standard gefunden.

Zu meinem großen Erstaunen gab es lange Zeit überhaupt kein Feedback der Community. Im besten Fall liegt das einfach daran, dass die Features klaglos funktionieren. Es könnte aber auch daran liegen, dass JSF-Entwickler einfach in anderen Bahnen denken als ich. Zum Beispiel so: „Der Quelltext muss nicht elegant, einfach und schön sein. Er muss einfach nur funktionieren." Noch eine andere Möglichkeit ist, dass frustrierte Entwickler gleich zu Angular, React oder Vue.js wechseln.

Aus meiner Perspektive ist das natürlich eine sehr interessante Option. So interessant, dass ich selber eine Angular-Schulung anbiete und – wenn ich die Wahl habe – normalerweise Angular oder Vue.js statt JSF verwende.

Nichtsdestotrotz bin ich weiterhin ein sehr aktiver und begeisterter Committer für ein JSF-Framework. Etwas pointiert ausgedrückt: Was bringt mich dazu, mich bei einem Legacy-Framework zu engagieren? Noch wichtiger: Warum sollten Sie sich im Jahr 2019 noch für JSF und BootsFaces interessieren, wo doch jeder Architekt und Entwickler, der etwas auf sich hält, im Frontend längst auf JavaScript und HTML5 setzt?

Was ist mit Angular, React und Vue.js?

Kein Zweifel, der Trend geht eindeutig Richtung JavaScript. Technologien wie JSF haben ihre besten Zeiten hinter sich. Die Dynamik der JavaScript-Community ist überwältigend. Das ist auch kein Wunder: Es gibt viel mehr JavaScript-Entwickler, als es jemals Java-Frontend-Entwickler gegeben hat. Wer sich ernsthaft und intensiv mit Frontend-Entwicklung beschäftigen möchte, dem rate ich schweren Herzens von BootsFaces ab. Heutzutage ist es schwer, als Full-Stack-Entwickler alle Bereiche von der Datenbank bis zum UX-Design abzudecken. Die Technologie hat sich auf allen Ebenen ausdifferenziert, wird komplexer, und es ist nicht abzusehen, dass sich das so bald ändert.

BootsFaces kann auf anderen Gebieten punkten. Es gibt sie noch, die Full-Stack-Entwickler. Entwickler, die gerne im Backend programmieren und trotzdem eine schöne Benutzeroberfläche mögen oder den Abstraktionslevel, den JSF bietet, zu schätzen wissen. Als JSF-Entwickler muss ich mich nicht zwingend intensiv mit HTML5 auseinandersetzen.

BootsFaces hat – im Vergleich zu anderen JSF-Komponentenbibliotheken – den Vorteil, einen besonders einfachen Zugang zu JavaScript und zu CSS zu bieten. Als Entwickler haben Sie die Wahl. Sie können den Komfort der Standardbibliothek verwenden, Sie können diese Standardbibliotheken mit CSS sehr einfach an Ihr Corporate Design anpassen oder Sie nutzen die nahtlose Integration mit JavaScript.

Ein Blick in die Glaskugel

Hinter den Kulissen tut sich bei BootsFaces einiges. Das kurzfristige Ziel ist es, auf Bootstrap 4 upzudaten. Wir haben schnell festgestellt, dass das einen verblüffend tiefgreifenden Umbau unseres Frameworks bedeutet. Das ist ein interessanter Unterschied zur normalen Anwendungsentwicklung: Dort gibt es einen relativ einfachen und unkomplizierten Migrationspfad. Als Framework-Entwickler wollen wir Ihnen aber den vollen Funktionsumfang von Bootstrap 4 zur Verfügung stellen und der ist nun einmal sehr viel größer – und in vielen Details auch anders – als das Feature Set der Vorgängerversion.

Daher haben wir uns für ein komplettes Redesign entschieden. Die Komponenten von BootsFaces werden jetzt nicht mehr in Java, sondern in der Template-Sprache von Angular definiert. Das Ergebnis ist um ein Vielfaches kürzer, sehr viel besser lesbar und – hoffentlich – leichter zu warten. Der Java-Code wird aus dem Angular-Template generiert.

Das eröffnet neue, spannende Möglichkeiten. Das Angular-Template definiert, wie der fertige HTML-Code aussehen soll. Es sagt nicht mehr, wie wir den HTML-Code erzeugen. Er muss nicht unbedingt von Java generiert werden!

Wir glauben, dass wir aus dem gleichen Quelltext auch eine Angular-Bibliothek, eine React-Bibliothek oder eine Vue.js-Bibliothek generieren können. Oder auch alle drei Varianten gleichzeitig.

Fazit

Wenn diese Idee funktioniert – und warum sollte sie nicht funktionieren! –, haben wir eine sehr elegante Antwort auf unsere ursprüngliche Frage gefunden. Welche Existenzberechtigung hat ein JSF-Framework wie BootsFaces im Jahr 2019? Ganz einfach: Erstens erfreut sich das totgesagte JSF auch heutzutage noch einer erstaunlichen Gesundheit und zweitens ist BootsFaces gleichzeitig auch eine Angular-Bibliothek, eine React-Bibliothek und eine Vue.js-Bibliothek – und damit voll und ganz auf der Höhe der Zeit! Zugegeben – das ist Zukunftsmusik. Um dorthin zu kommen, brauchen wir noch die Unterstützung von ein oder zwei Entwicklern.

Referenzen

[1] *http://balusc.omnifaces.org/2018/07/)*

Stephan Rauh
Stephan.Rauh@opitz-consulting.com

Stephan Rauh arbeitet als Senior Consultant bei der OPITZ CONSULTING Deutschland GmbH und befasst sich seit Jahren mit Angular, JSF und generell mit UI-Technologien. Er ist durch seinen Blog http://www.beyondjava.net und sein Open-Source-Framework BootsFaces bekannt geworden.

GraphQL für Java-Anwendungen

Nils Hartmann

GraphQL ist eine Sprache zur Abfrage von Daten, die mit dem Versprechen antritt, sowohl einfach in der Entwicklung als auch effizient in der Laufzeit zu sein. Dieser Artikel stellt die Sprache vor und zeigt, wie man dafür ein API für die eigene Java-Anwendung bauen kann.

Die Abfragesprache GraphQL steht seit 2015 als Open-Source-Lösung zur Verfügung. Ursprünglich von Facebook konzipiert und veröffentlicht, wird die Sprache seit Ende 2018 von einem Konsortium, der GraphQL Foundation *[1]*, weiterentwickelt und spezifiziert. GraphQL selbst ist kein fertiges Produkt; in der Spezifikation ist im Wesentlichen beschrieben, wie Abfragen aussehen müssen und wie diese von einem Server bearbeitet und beantwortet werden müssen. Möchte man für die eigene Anwendung eine GraphQL-Schnittstelle bereitstellen, kann man auf Frameworks und Tools für diverse Sprachen zurückgreifen, die einem bei der Implementierung helfen. Für Java gibt es das „graphql-java"-Projekt *[2]*, das später in diesem Artikel vorgestellt wird.

Zunächst aber noch zu den konzeptionellen und implementierungsunabhängigen Aspekten von GraphQL. Die zentrale Idee von GraphQL ist, dass ein Client immer genau die Daten von einer Anwendung abfragen kann, die er für einen Use-Case, zum Beispiel eine Ansicht im Frontend, benötigt. Dadurch können Netzwerk-Requests und übertragenes Datenvolumen optimiert werden, da der Client prinzipiell niemals zu viele oder zu wenige Daten abfragen muss. Gleichzeitig soll durch den Einsatz der Sprache aber auch die Entwicklung möglichst einfach sein, da sie es erlaubt, Anbieter und Verwender des API zu entkoppeln. Der Server kann nämlich einfach beliebige Daten zur Verfügung stellen, der Client ist aber nicht gezwungen, diese ganz zu konsumieren. Vielmehr pickt sich der Client je nach Anwendungsfall genau die Teile der Daten heraus, die er für den Anwendungsfall benötigt. Der Server kann somit Daten „auf Vorrat" zur Verfügung stellen und auch bestehende Daten erweitern, ohne dass davon bestehende Clients betroffen wären.

Ein praktisches Beispiel

Um die Konzepte von GraphQL praktisch zu verdeutlichen, soll eine kleine Beispiel-Anwendung, der BeerAdvisor, dienen. Der Quellcode der Anwendung steht auf GitHub zur Verfügung. Mit dieser Anwendung können Nutzer eine Reihe von Bieren bewerten und sich ansehen, wo diese Biere erhältlich sind. Andere Nutzer können die abgegebenen Bewertungen einsehen *(siehe Abbildung 1)*.

Der BeerAdvisor besteht clientseitig aus drei Ansichten, die jeweils auf unterschiedliche Ausschnitte aus dem Domain Model zugreifen. Die Übersichtsseite zeigt beispielsweise nur den Namen der Biere (Entity „Beer") und deren durchschnittliche Bewertung an. Die Detailansicht eines Bieres stellt neben diesen Informationen auch den Preis des Bieres dar, darüber hinaus aber auch dessen Bewertungen (aus den Entities „Rating" und „User" im Domain Model) und die Namen der Geschäfte (Entity „Shop"), in denen es erhältlich ist. Die Detail-Seite eines Geschäfts wiederum zeigt Informationen zu einem Geschäft (etwa dessen Adresse) sowie die dort erhältlichen Biere – davon aber nur jeweils den Namen und nicht deren Preis, Bewertungen etc. Mit GraphQL kann die Anwendung für jede der Ansichten genau die Informationen aus dem Model abfragen, die sie dafür jeweils benötigt.

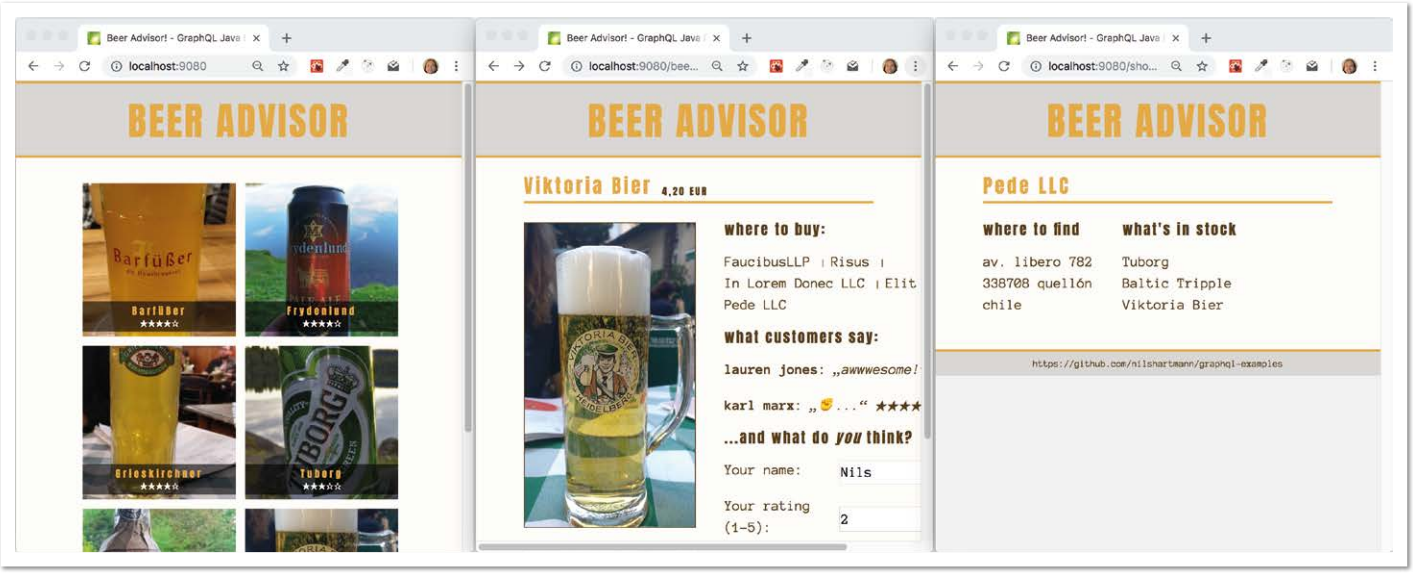


Abbildung 1: Die Beispielanwendung BeerAdvisor

```
query OverviewPageQuery {
  beers {
     name
     averageStars
  }
}
```

Listing 1

```
query BeerViewQuery {
  beer(beerId: "B1") {
     name
     ratings {
        comment
        author {
           name
        }
     }
     shops { . . . }
  }
}
```

Listing 2

```
curl -X POST -H "Content-Type: application/json" \
     -d ,{"query":"{ beers { id name } }"}‘
     http://localhost:9000/graphql
```

Listing 3

Mit GraphQL Daten abfragen

Eine GraphQL-Abfrage besteht immer aus einer oder mehreren „Operationen".

Eine Operation kann eine „Query" sein, um Daten zu lesen, eine „Mutation", um Daten zu schreiben bzw. zu verändern, oder eine „Subscription", wenn der Client sich für Events registrieren möchte, die der Server veröffentlicht.

Die Abfrage wird in einer JSON-artigen Notation formuliert, in der Daten vom API selektiert werden, in GraphQL „Felder" genannt.

Listing 1 zeigt die Query, die Namen (Feld „name") und durchschnittliche Bewertung (Feld „averageStars") für alle „Beer"-Objekte lädt, um die Übersichtsseite der Anwendung darzustellen:

Wie dort beschrieben, benötigt die Detailansicht eines Bieres andere Daten, beispielsweise die Bewertungen sowie deren Autoren. Die folgende Query *(Listing 2)* fragt diese Daten ab. Neu in dieser Query ist, dass Felder aus einer Hierarchie von Objekten abgefragt werden. Außerdem erhält das Feld „beer" ein Argument (ähnlich wie das in Java bei Methoden möglich ist).

Eine Abfrage in GraphQL besteht folglich aus einer hierarchischen Struktur von Feldern, die an einen GraphQL-fähigen Server geschickt werden muss. Das erfolgt in der Regel per HTTP-POST-Request an einen zentralen Endpunkt; eine Versionierung des API gibt es nicht, da dieses abwärtskompatibel erweitert werden kann. Der Endpunkt nimmt die Anfrage entgegen, verarbeitet sie und liefert das Ergebnis zurück. *Listing 3* zeigt beispielhaft, wie man mit dem Kommandozeilen-Tool „curl" eine GraphQL-Abfrage ausführen kann.

Das Ergebnis der Abfrage ist ein JSON-Objekt, das auf Root-Ebene ein „data"-Feld enthält. Dieses Feld enthält die abgefragten Daten, deren Hierarchie identisch mit der Abfrage ist. Der Client weiß daher durch die Formulierung seiner Abfrage, wie die Antwort vom Server strukturell aussieht. Eine exemplarische Antwort auf die oben gezeigte „BeerViewQuery" ist in *Abbildung 2* zu sehen – die Hierarchie unterhalb von „data" entspricht genau der abgefragten Struktur der Query.

Um Daten auf dem Server zu verändern, führt der Client eine Mutation aus. Die Abfrage dazu ist fast identisch zu einer Query, nur dass sie als Operation-Typ „mutation" angibt. Als Argumente übergibt der Client alle Informationen, die der Server benötigt, um die Aktion erfolgreich auszuführen. Aus dem spezifischen Ergebnis einer Mutation kann der Client wiederum – genau wie bei einer Query – einzelne Felder auswählen, die er vom Server als Antwort benötigt. Im BeerAdvisor können Nutzer über eine Mutation eine neue Bewertung („Rating") für ein Bier abgeben. Dazu muss der Client Informationen wie den Kommentar und den Benutzer übergeben. Der Ser-

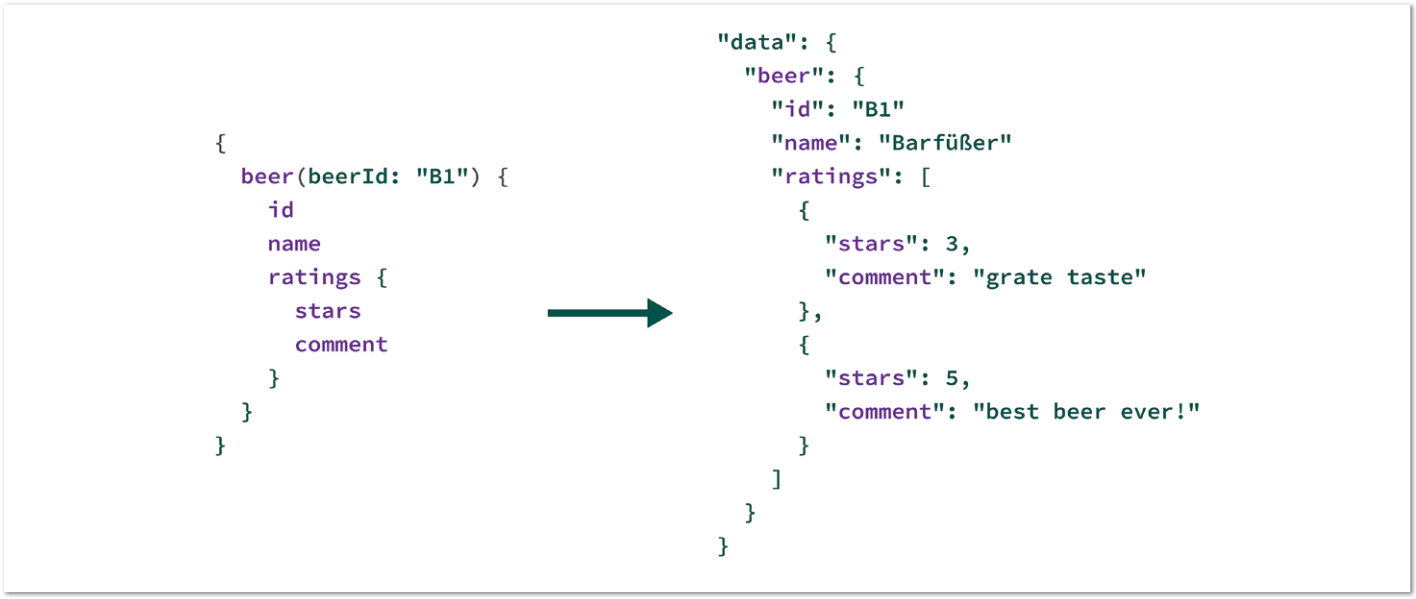


Abbildung 2: Frage und Antwort einer GraphQL Query

ver liefert für die „addRating"-Mutation das neue, auf dem Server angelegte Rating-Objekt zurück. Aus diesem Objekt kann sich der Client erneut die Felder herauspicken, die er benötigt – in unserem Fall ist das lediglich das „id"-Feld, da alle anderen Informationen der Bewertung auf dem Client schon bekannt sind *(siehe Listing 4)*.

```
mutation AddRatingMutation {
  addRating(beerId: „B1", userId: „U2", comment:
"tasty", stars: 3) {
      id
  }
}
```

Listing 4

Die API-Definition

Woher weiß der Client (oder ein Entwickler) nun, welche Objekte und Felder es auf dem Server überhaupt gibt? Welche Queries er ausführen kann und welche Mutations mit welchen Parametern vorhanden sind? Dazu muss man das GraphQL API mit einem Schema beschreiben.

Ein gängiger Weg zur Beschreibung des Schemas ist die Schema Definition Language (SDL), die im Juni 2018 den Weg in die GraphQL-Spezifikation geschafft hat und die auch in Java-Anwendungen verwendet wird.

Ein Schema beschreibt die Objekte (in GraphQL „Object Types" genannt) mitsamt ihren Feldern, die Clients abfragen können. Da GraphQL eine typsichere Sprache ist, muss man die Felder mit Typen versehen. Es gibt einige eingebaute Typen wie ID, String, Int, Boolean und außerdem Arrays beziehungsweise Listen. Die Definition eines eigenen, fachlichen Objekttyps könnte wie in *Listing 5* aussehen.

```
type Rating {
  id: ID!
  beer: Beer!
  author: User!
  comment: String!
  stars: Int!
}
```

Listing 5

```
type Query {
  beer(beerId: ID!): Beer
  beers: [Beer!]!
}

type Mutation {
  addRating(ratingInput: AddRatingInput): Rating!
}

type Subscription {
  onNewRating(beerId: ID!): Rating!
}
```

Listing 6

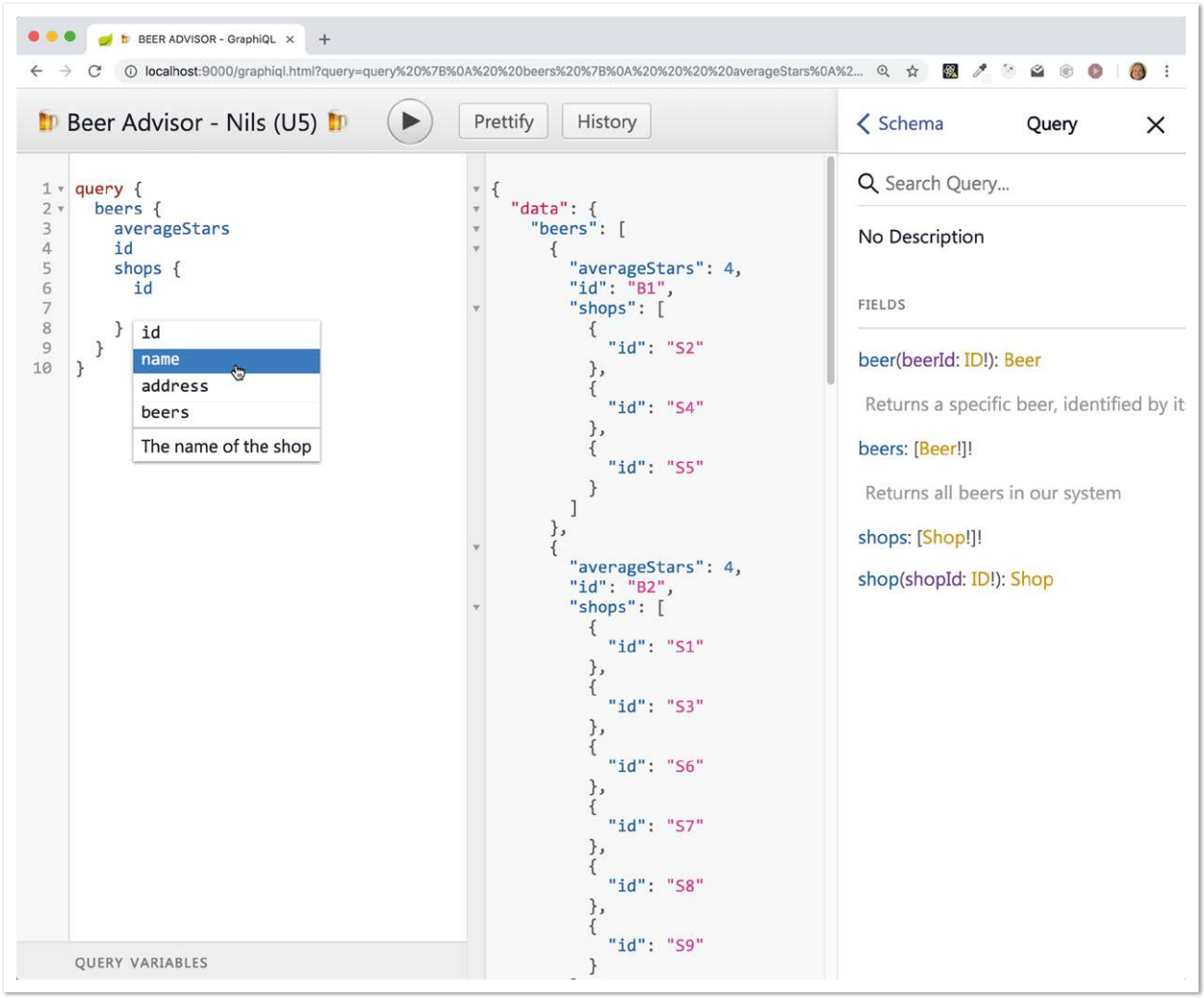


Abbildung 3: Der GraphQL API Explorer „GraphiQL"

Listing 5 definiert den Typ „Rating", der aus fünf Feldern besteht. Jedes Feld erfordert neben dem Namen den jeweiligen Typ. Dazu gehören auch die Angaben, ob das Feld ein Pflichtfeld ist oder nicht (gekennzeichnet mit einem Ausrufezeichen) oder ob es sich um ein Array handelt (dann wäre der Feld-Typ mit eckigen Klammern umschlossen). Die Felder „beer" und „author" zeigen auf weitere fachliche Objekttypen.

Neben den Objekttypen besteht ein Schema noch aus den Operationen, die über das API ausgeführt werden können – das sind die angesprochenen Queries, Mutations und Subscriptions. Die Operationen heißen „Root Operation Types", werden syntaktisch genauso wie Object Types in der SDL beschrieben und bestehen ebenfalls aus einer Menge von Feldern *(siehe Listing 6)*.

Am Query Type ist ein Feld „beer" definiert. Daran ist die Beschreibung von Argumenten für Felder zu sehen. Für die einzelnen Argumente müssen ebenfalls Name und Typ angegeben werden, ähnlich wie bei Argumenten in Java-Methoden.

Während Mutation und Subscription in einem Schema optional sind, ist zumindest die Definition des Query Type notwendig.

GraphQL-Abfragen beginnen immer bei einem Feld, das innerhalb eines der Root Operation Types definiert ist. Von dort aus kann dann weiter transitiv über alle referenzierten Types gewandert werden.

Bei der Ausführung einer Query stellt die GraphQL-Laufzeitumgebung auf dem Server sicher, dass nur Queries, die mit dem Schema kompatibel sind, verarbeitet werden. Queries, die ungültig sind, weil sie zum Beispiel nichtexistierende Felder abfragen, werden vom Server zurückgewiesen. Ähnlich verhält es sich mit dem Ergebnis der Query: Der Client bekommt die Daten nur zurückgeschickt, wenn die Antwort dem Schema entspricht. Ebenfalls wird überprüft, ob Pflichtfelder wirklich befüllt sind und ob die zurückgegebenen Felder den erwarteten Typen entsprechen. Dadurch können sowohl Server als auch Client bei der Verarbeitung von Queries und deren Ergebnissen sicher sein, dass sie es nur mit syntaktisch korrekten Daten zu tun haben.

Entwickler-Werkzeuge

Das Schema selbst kann zur Laufzeit ebenfalls durch eine reguläre GraphQL Query abgefragt werden, ähnlich wie dies in Java mittels Reflection möglich ist.

Die Möglichkeit, das Schema zur Laufzeit abzufragen, hat eine Reihe von Werkzeugen für die Arbeit und Entwicklung von GraphQL-Anwendungen hervorgebracht. Ein prominentes Beispiel ist GraphiQL. Dabei handelt es sich um einen Editor, der im Browser läuft und mit dem Nutzer GraphQL-Abfragen schreiben und ausführen können. GraphiQL fragt das jeweilige Schema eines API ab und kann dadurch Features wie Code-Completion und Syntax-Highlighting anbieten, so wie man sie auch aus der Java-Entwicklung in IDEs gewohnt ist. Der Editor lässt sich auch in die eigene GraphQL-Anwendung integrieren, um zum Beispiel zur Entwicklungszeit Queries zu testen *(siehe Abbildung 3)*.

Auch für die JetBrains-Produktfamilie (u.a. IDEA, WebStorm) stehen Plug-ins zur Verfügung, mit denen sich ebenfalls aus der IDE heraus Queries gegen ein GraphQL API absetzen lassen können und die Unterstützung bei der Formulierung des Schemas in der SDL bieten (Refactorings etc.).

GraphQL-Server in Java

Wir haben jetzt gesehen, wie Abfragen formuliert und ausgeführt werden und wie das Schema des API definiert wird. Nun gilt es, das API serverseitig in Java zu implementieren und für Clients bereitzustellen.

Grundlage dafür ist das Open-Source-Projekt graphql-java. Mit diesem Projekt wird zunächst das Schema des API definiert (per SDL oder programmatisch). Dann werden „DataFetcher" implementiert, die festlegen, welche Daten für welches Feld des API zurückgeliefert werden. Das Projekt hat keine Abhängigkeit auf Spring oder Java EE, kann aber sehr einfach in beiden Umgebungen eingesetzt werden.

```
class BeerAdvisorDataFetchers {
  // gesetzt z.B. durch DI-Mechanismus
  private BeerRepository beerRepository;

  public DataFetcher beerDataFetcher() {
     return new DataFetcher<List<Beer>>() {
        @Override
           public List<Beer> get(DataFetchingEnvironment env) {
              return beerRepository.findAll();
        }
     }
  }

  public DataFetcher beerDataFetcher() {
     return new DataFetcher<Beer>() {
        @Override
        public Beer get(DataFetchingEnvironment env) {
           // Argument "beerId" aus der gerade ausgeführten
           // Query auslesen
           String beerId = env.getArgument("beerId");
           return beerRepository.getBeer(beerId);
        }
     }
  }
}
```

Listing 7

Für die Integration in Spring und Spring Boot stehen zwei weitere Projekte, graphql-java-spring *[2]* und graphql-spring-boot *[2]* zur Verfügung.

Das Schema des API wird üblicherweise in einer oder mehreren Text-Dateien abgelegt und so eingebunden, dass es zur Laufzeit über den Klassenpfad eingelesen werden kann (zum Beispiel im „src/main/resources"-Ordner).

Damit der Server zur Laufzeit weiß, welche Informationen für welche Anfrage geliefert werden sollen, müssen „DataFetcher" implementiert und an das Schema gebunden werden. Ein DataFetcher liefert dabei immer den Wert für ein Feld aus dem Schema. Es handelt sich dabei um ein Interface, das nur über eine einzige Methode verfügt, die implementiert werden muss. Zur Laufzeit bekommt diese Methode über den Parameter „environment" Informationen über die aktuelle Abfrage übergeben. Darüber lassen sich dann zum Beispiel die in der Query angegebenen Argumente eines Feldes auslesen.

In *Listing 7* sind exemplarisch zwei DataFetcher zu sehen. Zum einen der DataFetcher für das „beers"-Feld (das eine Liste aller Biere zurückliefert) und zum anderen der Fetcher für das „beer"-Feld (das ein einzelnes Bier, das über das Argument „beerId" vom Aufrufer angegeben wird, zurückliefert).

Da es sich bei dem DataFetcher um ein funktionales Interface handelt, kann die Implementierung auch über Lambdas erfolgen *(siehe Listing 8)*.

Die DataFetcher müssen zwingend für alle Felder angelegt werden, die in einem der Root Operation Types (also Query, Subscription oder Mutation) definiert sind. Für alle darunterliegenden bzw. referenzierten Objekte sind die DataFetcher optional. Ist für ein Feld kein DataFetcher angegeben, wird per Default der „PropertyDataFetcher" verwendet. Dieser probiert ein Feld entweder über Reflektion und Namenskonventionen zu ermitteln (bei POJOs) oder liest den Wert aus einer Map aus. Aus diesem Grund muss für die Felder „name", „id" und „price", die im Schema für den „Beer"-Typ definiert sind, auch kein DataFetcher angegeben werden. Das Root-Objekt (Beer) wird über einen der beiden oben gezeigten DataFetcher ermittelt, die jeweils ein „Beer"-Objekt (bzw. eine Liste davon) zurückgeben. Da in der Java-Klasse „Beer" ebenfalls die Felder „name", „id" und „price" vorhanden sind, muss dafür kein expliziter DataFetcher angegeben werden, da der PropertyDataFetcher diese Felder automatisch findet und abfragen kann. (Dabei ist übrigens sichergestellt, dass nur auf solche Felder zugegriffen wird, die auch im Schema definiert sind. Ein Feld, das im Schema nicht definiert ist, aber an einer POJO-Instanz existiert, ist trotzdem nicht abfragbar.)

Anders sieht die Sache für das Feld „averageStars" am „Beer"-Objekt aus. Dieses Feld ist in der Java-Klasse „Beer" nicht definiert, da der Wert dynamisch berechnet werden soll. Also muss dafür ein eigener DataFetcher implementiert werden. DataFetcher für (Unter-)Objekte sehen genauso aus wie die vorher gezeigten DataFetcher für die Root Operation Types. Über die Methode „getSource" am übergebenen Environment kann die Instanz des Objektes abgefragt werden, das der übergeordnete Fetcher ermittelt hat. In diesem Beispiel geht es um ein Feld, das im API-Schema auf dem „Beer"-Objekt definiert ist, folglich liefert „getSource" eine Instanz der Java-Klasse „Beer" *(siehe Listing 9)*.

```
class BeerAdvisorDataFetchers {
  private BeerRepository beerRepository;

  public DataFetcher<Beer> beerFetcher() {
     return environment -> {
        String beerId = environment.getArgument("beerId");
        return beerRepository.getBeer(beerId);
     };
  }

  public DataFetcher<List<Beer>> beersFetcher() {
     return environment -> beerRepository.findAll();
  }
}
```

Listing 8

```
import nh.graphql.beeradvisor.domain.Beer;

public class BeerDataFetchers {
  public DataFetcher<Integer> averageStarsFetcher() {
     return environment -> {
        // Instanz ermitteln, von der das Feld 'averageStars'
        // abgefragt wurde
        Beer beer = environment.getSource();

        // Wert für das Feld berechnen und zurückliefern
        return (int)
           Math.round(beer.getRatings().stream()
              .mapToDouble(Rating::getStars)
              .average()
              .getAsDouble());
     };
  }
}
```

Listing 9

```
SchemaParser schemaParser = new SchemaParser();

TypeDefinitionRegistry typeRegistry =
  schemaParser.parse(/* (Klassen-)Pfad zur SDL-Datei */);

// Zuordnung von Typen/Feldern zu DataFetchern
RuntimeWiring runtimeWiring = RuntimeWiring.newRuntimeWiring()
  .type(newTypeWiring("Query")
     .dataFetcher("beer", beerAdvisorDF.beerFetcher())
     .dataFetcher("beers", beerAdvisorDF.beersFetcher()))
  .type(newTypeWiring("Beer")
     .dataFetcher("averageStars",
        beerDFs.averageStarsFetcher()))
  .type(newTypeWiring("Mutation")
     .dataFetcher("addRating",
        beerAdvisorDF.addRatingFetcher()))
  .build();

// Das "ausführbare" GraphQL Schema
GraphQLSchema schema =
  new SchemaGenerator().makeExecutableSchema
     (typeRegistry, runtimeWiring);
```

Listing 10

```
GraphQL graphQL = GraphQL.newGraphQL(schema).build();

ExecutionInput executionInput =
  ExecutionInput.newExecutionInput()
     .query
     ("query { beers { id name ratings { stars } }
}").build();

ExecutionResult executionResult =
  graphQL.execute(executionInput);

Map<String, Object> data = executionResult.getData();
```

Listing 11

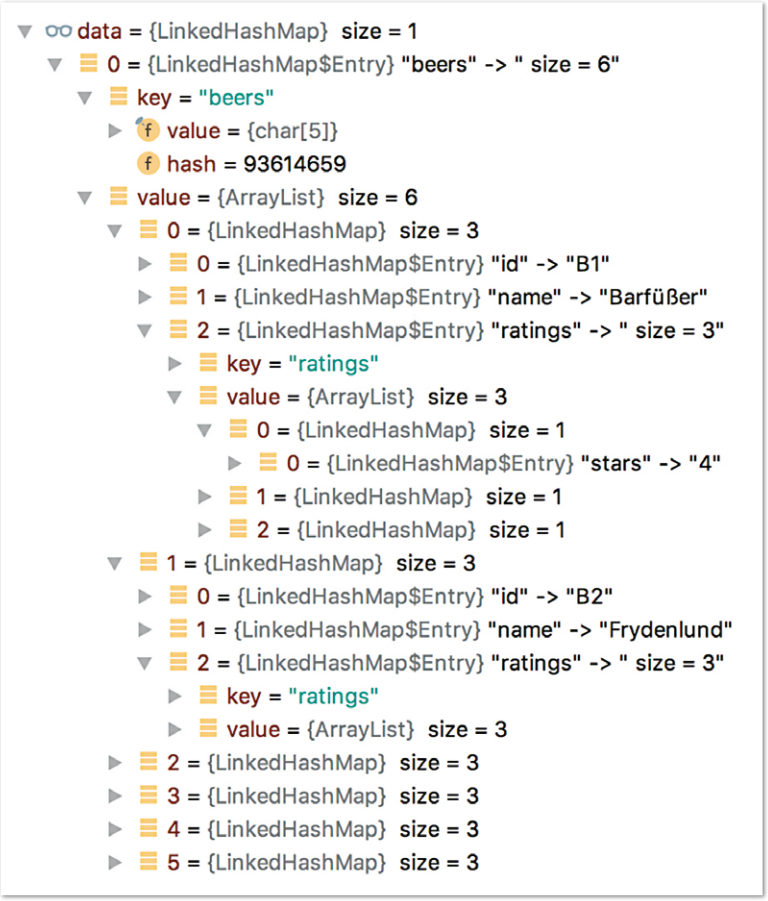


Abbildung 4: Das Ergebnis eine GraphQL-Abfrage in Java

DataFetcher für Mutations und Subscriptions werden technisch identisch implementiert, sie unterscheiden sich lediglich in dem, was sie fachlich machen, von den gezeigten Fetchern. Eine Mutation etwa könnte ein Domain-Objekt verändern und über ein Repository in der Datenbank speichern. Bei Subscriptions ist zu beachten, dass der vom DataFetcher zurückgelieferte Wert ein Reactive-Streams-Publisher-Objekt sein muss. Beispielhafte Implementierungen dafür finden sich jeweils im BeerAdvisor Source Code.

Bereitstellen des API

In der Anwendung ist jetzt das Schema des API definiert und mit den DataFetchern ist beschrieben, wie Daten zu einzelnen Feldern ermittelt werden können. Im letzten Schritt muss das Schema mit den DataFetchern verknüpft werden. Dies erfolgt beim Start der Anwendung über das „RuntimeWiring". Mit diesen Informationen wird dann eine Instanz von „GraphQLSchema" erzeugt, mit der es dann möglich ist, Abfragen auszuführen. Das gekürzte *Listing 10* zeigt das exemplarisch.

Über die erzeugte Schema-Instanz lassen sich jetzt aus der Java-Anwendung heraus GraphQL Queries ausführen. Das Ergebnis der (erfolgreichen) Queries *(Listing 11)* ist in der Regel eine verschachtelte Map mit den Werten der abgefragten Felder *(siehe Abbildung 4)*.

Das Projekt graphql-java-tools *[2]* bietet darauf aufbauend eine Abstraktionsschicht an, mit der es möglich ist, GraphQL-Abfragen und -Antworten auf Java Beans zu mappen.

In der Regel werden GraphQL-Abfragen aber nicht aus der Java-Anwendung heraus ausgeführt, sondern sollen über einen HTTP-Endpunkt angeboten werden. Dazu muss die oben gezeigte Logik in ein Servlet eingebunden werden, das die Query aus dem HTTP Request entgegennimmt und an die GraphQL-Instanz weiterleitet. Ein fertiges Servlet steht in den Projekten graphql-java-servlet *[2]* und graphql-java-spring *[2]* zur Verfügung.

Fazit

GraphQL bietet eine interessante Möglichkeit, Daten gezielt vom Server zu laden, ohne dass sich Client und Server sehr eng aneinander koppeln müssen. Auf der anderen Seite ist die Sprache aber von der Mächtigkeit her nicht mit SQL oder Ähnlichem zu vergleichen: Sortierungen, Aggregationen oder eine Suche etwa fehlen der Sprache. Werden solche Features benötigt, müssen sie manuell programmiert werden, zum Beispiel durch entsprechende Felder oder Argumente an Feldern.

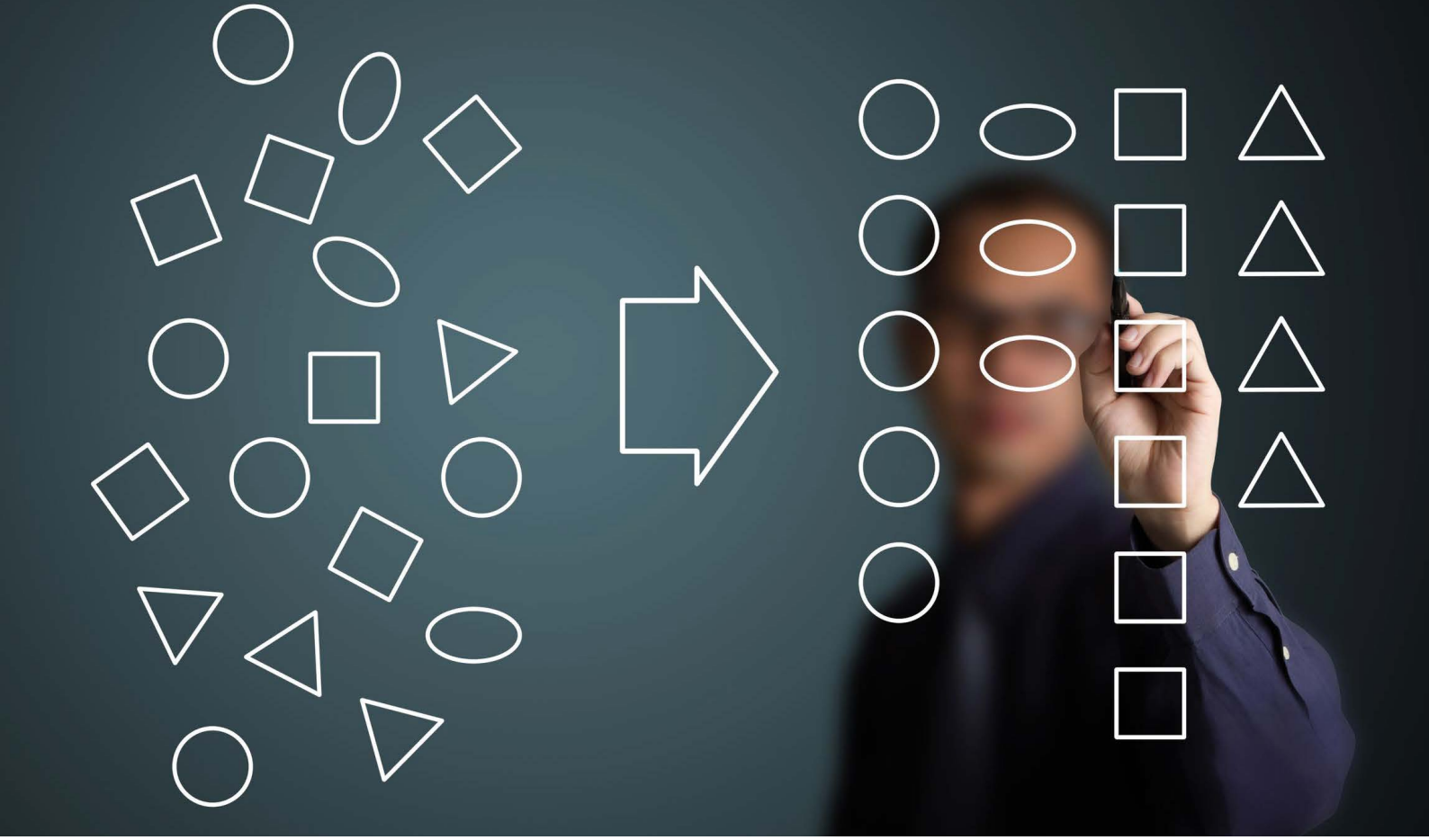
Ein GraphQL API für die eigene Anwendung anzubieten, ist mit dem graphql-java-Projekt relativ einfach. Darüber hinaus macht das typisierte Schema sehr gute Entwicklertools möglich, sodass sich eigene bestehende Anwendungen zügig mit einem GraphQL API versehen lassen. Es lohnt sich auf jeden Fall, die Sprache zumindest einmal auszuprobieren und die weitere Entwicklung in diesem Bereich im Auge zu behalten.

Referenzen

[1] *https://foundation.graphql.org/*
[2] *https://www.graphql-java.com/*
[3] *https://github.com/nilshartmann/graphql-examples/tree/master/java/beeradvisor_graphql-java/backend*

Nils Hartmann
nils@nilshartmann.net

Nils Hartmann ist freiberuflicher Softwareentwickler, Architekt und Trainer aus Hamburg. Er programmiert sowohl in Java als auch in JavaScript/TypeScript und beschäftigt sich mit der Entwicklung von Single-Page-Anwendungen. Nils ist Co-Autor des Buchs „React – Die praktische Einführung" (dpunkt-Verlag) und gibt sein Wissen auf Konferenzen, Schulungen und Workshops weiter.



Eine moderne und konsistente Implementierung natürlicher Ordnung bei Java-Objekten – Teil 1

Christian Heitzmann, SimplexaCode AG

Eigene Java-Klassen sortierbar, also komparabel zu machen, ist eine elementare und wichtige Aufgabe. Gemessen an ihrer Bedeutung wird dieses Thema in der Grundlagen-Literatur jedoch nur sehr spärlich behandelt. Hinzu kommt, dass mit neuen Java-Versionen und somit neuen Sprachmerkmalen immer wieder neue Varianten entstanden sind, wie sich die „compareTo"- beziehungsweise „compare"-Methoden mehr oder weniger elegant implementieren lassen. Dieser zweiteilige Artikel in dieser und in der nächsten Ausgabe zeigt die Evolution dieser Methoden über die verschiedenen Java-Versionen auf und bietet eine Code-Vorlage für eine saubere Implementierung ab Java 8 an.

Auch wenn die Implementierung einer natürlichen Ordnung in eigenen Java-Klassen durchaus bedeutsam sein mag, so ist es zumindest nicht notwendig, diese Definition für alle eigenen Klassen vorzunehmen. Hingegen ergibt es durchaus Sinn, dies für sogenannte „Werteklassen" zu tun, also für Klassen, deren Schwerpunkt mehr auf Daten als auf Verhalten liegt.

Die Definition spezifischer natürlicher Ordnung wird spätestens dann unvermeidbar, wenn die Klassen in Algorithmen oder Datenstrukturen (vor allem „Collections") verwendet werden, die auf irgendeine Art eine Sortierung vornehmen. Das wird ganz klar bei Methoden wie „Arrays.sort", „Collections.sort" oder „List#sort" (seit Java 8), kann aber bei Collections wie „TreeSet" oder „TreeMap" auch weniger offensichtlich sein, weil diese Datenstrukturen ihre Sortierung (und sogar Balancierung) intern und üblicherweise für den API-Benutzer unsichtbar durchführen.

Zum Glück kann ein Programmierer nicht aus Versehen eine Methode oder Collection benutzen, die ein vergleichbares (komparables)

Objekt benötigt, ohne dass das Objekt auch tatsächlich komparabel ist. Entweder implementiert die eigene Klasse die generische Schnittstelle „Comparable<T>“, die den Programmierer wiederum zwingt, ihre einzige Methode „compareTo(T)“ mit Funktionalität zu füllen, oder der Programmierer kann einen externen generischen „Comparator<T>“ implementieren, der dann der sortierenden Methode oder Datenstruktur zusammen mit den eigentlichen Klassenobjekten übergeben wird. Mit anderen Worten, die Implementierung von „Comparable“ stellt eine „eingebaute interne“ Definition der Klassenreihenfolge dar, während „Comparator“ einer „nachgerüsteten externen“ Definition entspricht.

Streng genommen sollte man nur bei Ersterem, also bei der internen Implementierung, von „natürlicher Ordnung“ sprechen, denn das „natürlich“ bezieht sich darauf, dass die Objekte „von selbst“, also „natürlich“ wissen, in welcher Reihenfolge sie sich anzuordnen haben. Bei der externen Implementierung wird dies von außen „auferdiktiert“, was nicht zwingend „natürlich“ sein muss. Nachdem nun auf diese Finesse hingewiesen wurde, wird in diesem Artikel auf eine derart strenge Unterscheidung verzichtet.

Die alte Art bis Java 6

Abgesehen vom aufgezeigten Hauptunterschied zwischen „Comparable“ und „Comparator“ bleibt die eigentliche Implementierung der natürlichen Ordnung einer Klasse für beide Varianten ähnlich. Hier muss man unterscheiden zwischen der Art, wie „Comparable compareTo(T)“- und „Comparator compare(T, T)“-Methoden in der Vergangenheit, bis und mit Java 6, implementiert wurden, wie sie dann in Java 7 implementiert werden dürfen und wie sie letztlich heute, seit Java 8, am konsistentesten implementiert werden können.

Um die unterschiedlichen Arten zu demonstrieren, wird die folgende Klasse eingeführt. Sie stellt eine künstliche Werteklasse dar, die alle möglichen Datentypen für ihre Attribute (englisch „fields“) ent-

```
public final class ComparableJava6DemoClass implements Comparable<ComparableJava6DemoClass> {
  private boolean booleanField;
  private byte byteField;
  private char charField;
  private short shortField;
  private int intField;
  private long longField;
  private float floatField;
  private double doubleField;
  private OtherComparableClass comparableClassField;

  /* [...] Constructors omitted. */
  @Override
  public int compareTo(ComparableJava6DemoClass otherDemoClass) {
     int difference;
     /* 1. booleanField */
     if (!this.booleanField && otherDemoClass.booleanField) return -1;
     if (this.booleanField && !otherDemoClass.booleanField) return +1;
     /* 2. byteField */
     if (this.byteField < otherDemoClass.byteField) return -1;
     if (this.byteField > otherDemoClass.byteField) return +1;
     /* 3. charField */
     if (this.charField < otherDemoClass.charField) return -1;
     if (this.charField > otherDemoClass.charField) return +1;
     /* 4. shortField */
     if (this.shortField < otherDemoClass.shortField) return -1;
     if (this.shortField > otherDemoClass.shortField) return +1;
     /* 5. intField */
     if (this.intField < otherDemoClass.intField) return -1;
     if (this.intField > otherDemoClass.intField) return +1;
     /* 6. longField */
     if (this.longField < otherDemoClass.longField) return -1;
     if (this.longField > otherDemoClass.longField) return +1;
     /* 7. floatField */
     difference = Float.compare(this.floatField, otherDemoClass.floatField);
     if (difference != 0) return difference;
     /* 8. doubleField */
     difference = Double.compare(this.doubleField, otherDemoClass.doubleField);
     if (difference != 0) return difference;
     /* 9. comparableClassField */
     if (this.comparableClassField == null) {
        difference = (otherDemoClass.comparableClassField == null ? 0 : +1);
     } else {
        difference = (otherDemoClass.comparableClassField == null ? -1
           : this.comparableClassField
           .compareTo(otherDemoClass.comparableClassField));
     }
     if (difference != 0) return difference;
     return 0;
  }
  /* [...] Other methods omitted. */
}
```

Listing 1

hält, es gibt also ein Attribut für jeden primitiven Java-Datentyp, wie „short" oder „double" (mit den entsprechenden Namen „shortField" und „doubleField"), sowie ein Attribut für einen Referenztyp (mit dem Namen „comparableClassField", der in diesem Fall eine andere komparable Klasse namens „OtherComparableClass" referenziert).

Es wurde darauf verzichtet, auch Arrays zu referenzieren, da die Implementierung von Ordnung auf Arrays normalerweise das Durchlaufen all seiner Elemente in einer Schleife benötigt, den ganzen Code aufbläht, in der Praxis nicht sehr geläufig ist und daher auch nicht viel zur Kernaussage dieses Artikels beiträgt *(siehe Listings 1 und 2)*.

Ein paar Punkte sind erwähnenswert:

- Die Reihenfolge, in der die Attribute verglichen werden, spielt auf jeden Fall eine Rolle. In dieser künstlichen Demoklasse werden die Klassen-Attribute in derselben Reihenfolge verglichen, in der sie als Felder deklariert sind. Es gibt keinen Grund, dies zwingend so zu tun. In realen Klassen wird es viel wahrscheinlicher sein, dass eine andere als die Deklarations-Reihenfolge verwendet wird. Der erste Vergleich (hier kommentiert mit „/* 1. booleanField */") entspricht dem höchstwertigen Attribut. Nur wenn sich die zwei booleschen Werte als gleich herausstellen, wird das zweit-höchstwertige Attribut (in diesem Fall „byteField", kommentiert mit „/* 2. byteField */") geprüft und verglichen. Man sieht, dass es in der Demoklasse insgesamt neun Vergleichsschritte gibt. Erst wenn sich alle neun Attribute als gleich herausstellen, dann, und nur dann, wird die „compareTo"-Methode „0" zurückgeben und somit Gleichheit anzeigen.
- Einen externen „Comparator" zu implementieren, funktioniert ähnlich, ist jedoch nicht exakt das Gleiche. Seine „compare(T, T)"-Methode benötigt zwei Parameter: die zwei Klassen, die verglichen werden sollen. Die „Comparable compareTo(T)"-Methode benötigt nur einen Parameter, da sie sich selbst bereits kennt und darum nur eine weitere Referenz auf das andere Objekt benötigt. Der gezeigte Code-Ausschnitt verwendet daher „demoClass1" und „demoClass2" anstelle von „this" und „otherDemoClass" *(siehe Listing 3)*.

```
public abstract class OtherComparableClass implements
Comparable<OtherComparableClass> {
  /* [...] Body omitted. */
}
```

Listing 2

Ein weiterer Unterschied ist die Verwendung von Getter-Methoden, da ein externer „Comparator" normalerweise nicht direkt auf die Klassenattribute, die verglichen werden sollen, zugreifen kann (das nennt sich „Kapselung"). Wann immer möglich, sollte man versuchen, die interne Methode zu implementieren, anstatt eine externe „compare"-Methode zu schreiben.

Es gibt verschiedene Arten, die Attribute zu vergleichen, abhängig von ihrem Typ:

- Primitive, ganzzahlige Typen (also „byte", „char", „short", „int" und „long", siehe Vergleiche 2 bis 6) scheinen die einfachsten zu sein, sollten aber ab Java 8 trotzdem geändert werden (siehe unten). Es ist einfach, ihre Funktionalität zu verstehen. Im Falle der ganzzahligen „compareTo"-Methode versetzt man sich selbst in die Position der Klasse, die sie implementiert. Dann sagt man: „Wenn mein (also „this") Wert kleiner als der Wert des anderen (also „Parameters") ist, dann gib eine negative Zahl zurück. Wenn mein Wert größer als der Wert des anderen ist, dann gib eine positive Zahl zurück."
- Man kann auch ähnlich bei der externen „compare"-Methode vorgehen: „Wenn der Wert des ersten Parameters kleiner als der Wert des zweiten Parameters ist, dann gib eine negative Zahl zurück. Wenn der Wert des ersten Parameters größer als der Wert des zweiten Parameters ist, dann gib eine positive Zahl zurück."
- Der Vergleich von „boolean" (siehe Vergleich 1) ist weniger offensichtlich, aber immer noch einfach nachzuvollziehen, wenn man den

```
import java.util.*;

public final class DemoComparator implements Comparator<ComparableJava6DemoClass> {
  @Override
  public int compare(ComparableJava6DemoClass demoClass1, ComparableJava6DemoClass demoClass2) {
     int difference;
     /* 1. booleanField */
     if (!demoClass1.isBooleanField() && demoClass2.isBooleanField()) return -1;
     if (demoClass1.isBooleanField() && !demoClass2.isBooleanField()) return +1;
     /* 2. byteField */
     if (demoClass1.getByteField() < demoClass2.getByteField()) return -1;
     if (demoClass1.getByteField() > demoClass2.getByteField()) return +1;
     /* [...] Other comparisons omitted. */
     /* 9. comparableClassField */
     if (demoClass1.getComparableClassField() == null) {
        difference = (demoClass2.getComparableClassField() == null ? 0 : +1);
     } else {
        difference = (demoClass2.getComparableClassField() == null ? -1
           : demoClass1.getComparableClassField()
           .compareTo(demoClass2.getComparableClassField()));
     }
     if (difference != 0) return difference;
     return 0;
  }
}
```

Listing 3

```
public final class ComparableJava7DemoClass implements Comparable<ComparableJava7DemoClass> {
  /* [...] Fields omitted. */

  /* [...] Constructors omitted. */
  @Override
  public int compareTo(ComparableJava7DemoClass otherDemoClass) {
     int difference;
     /* 1. booleanField */
     difference = Boolean.compare(this.booleanField, otherDemoClass.booleanField);
     if (difference != 0) return difference;
     /* 2. byteField */
     difference = Byte.compare(this.byteField, otherDemoClass.byteField);
     if (difference != 0) return difference;
     /* 3. charField */
     difference = Character.compare(this.charField, otherDemoClass.charField);
     if (difference != 0) return difference;
     /* 4. shortField */
     difference = Short.compare(this.shortField, otherDemoClass.shortField);
     if (difference != 0) return difference;
     /* 5. intField */
     difference = Integer.compare(this.intField, otherDemoClass.intField);
     if (difference != 0) return difference;
     /* 6. longField */
     difference = Long.compare(this.longField, otherDemoClass.longField);
     if (difference != 0) return difference;
     /* 7. floatField */
     difference = Float.compare(this.floatField, otherDemoClass.floatField);
     if (difference != 0) return difference;
     /* 8. doubleField */
     difference = Double.compare(this.doubleField, otherDemoClass.doubleField);
     if (difference != 0) return difference;
     /* 9. comparableClassField */
     if (this.comparableClassField == null) {
        difference = (otherDemoClass.comparableClassField == null ? 0 : +1);
     } else {
        difference = (otherDemoClass.comparableClassField == null ? -1
           : this.comparableClassField
           .compareTo(otherDemoClass.comparableClassField));
     }
  if (difference != 0) return difference;
     return 0;
  }
  /* [...] Other methods omitted. */
}
```

Listing 4

Quelltext liest. In diesem Fall kommt „false" vor „true", also eine Instanz von ComparableDemoClass, die das „booleanField" auf „false" gesetzt hat, ist kleiner als eine Instanz der gleichen Klasse, die „booleanField" auf „true" gesetzt hat. Man beachte, dass dies lediglich eine Definitionssache und nicht in Stein gemeißelt ist.

- Fließkommazahlen (also „float" und „double", siehe Vergleiche 7 und 8) sollten die statische „compare"-Methode ihrer Wrapper-Klassen „Float" beziehungsweise „Double" verwenden. „float" und „double" sind viel komplizierter als ganzzahlige Typen, weil sie besondere Werte besitzen können: positive und negative Nullen, positive und negative „Unendlichkeiten" (englisch „infinities") sowie NaN („Not-a-Number") als Resultat gewisser ungültiger Operationen, wie die Division von Null durch Null. Da man sich wahrscheinlich nicht mit all diesen Sonderfällen herumschlagen möchte, verwendet man einfach die oben erwähnte „compare"-Methode, die einem all die schwere Arbeit abnimmt.
- Die alte Art, Referenzattribute zu vergleichen (siehe Vergleich 9), ist langatmig und fehleranfällig. Es lohnt sich nicht, durch die Details des Code-Blocks zu gehen, weil er nicht mehr relevant ist. Die Grundidee ist, dass die „compareTo"-Methoden der Referenz-Typen rekursiv aufgerufen werden. Da mein Referenz-Attribut, das andere Referenz-Attribut (also das Referenz-Attribut des Parameters) oder beide „null" sein können, muss eine komplizierte Fallunterscheidung durchgeführt werden, die den Code aufbläht. In obigem Quelltext ist „null" größer als jeder andere (gültige) Wert. Mit anderen Worten, „null" kommt in der Reihenfolge zuletzt.

Die Zwischenlösung in Java 7

In Java 7 wurden alle Wrapper-Klassen für boolesche Werte („Boolean" für „boolean") und alle ganzzahligen primitiven Typen („Byte" für „byte", „Character" für „char", „Short" für „short", „Integer" für „int" und „Long" für „long") mit statischen „compare"-Methoden ausgestattet. Ihre Verwendung ist konsistent zu denjenigen von „Float" und „Double". Joshua Bloch, der Autor des großartigen Buchs „Effective Java", schreibt, die Verwendung von „relational operators „<" and „>" in compareTo-methods is verbose and error-prone and no longer recommended."

Der Autor stimmt dieser Aussage nicht ganz zu. Wie man in unten stehendem Code-Beispiel sehen kann, hat sich die „Code-Verbosität" – sofern man überhaupt von „Verbosität" sprechen kann – nicht wirklich geändert. In Bezug auf die behauptete Fehleranfälligkeit warf er einen Blick in die JDK-Implementierungen der statischen „compare"-Methoden der Wrapper-Klassen, um herauszufinden, ob er irgendwelche Fallstricke im Zusammenhang mit den Vergleichs-

```
import java.util.*;
public final class ComparableJava8DemoClass implements Comparable<ComparableJava8DemoClass> {
  private static final Comparator<ComparableJava8DemoClass> DEMO_CLASS_COMPARATOR
     = Comparator.comparing((ComparableJava8DemoClass dc)
        -> Boolean.valueOf(dc.booleanField))                          /* 1. */
     .thenComparingInt(dc -> dc.byteField)                            /* 2. */
     .thenComparingInt(dc -> dc.charField)                            /* 3. */
     .thenComparingInt(dc -> dc.shortField)                           /* 4. */
     .thenComparingInt(dc -> dc.intField)                             /* 5. */
     .thenComparingLong(dc -> dc.longField)                           /* 6. */
     .thenComparingDouble(dc -> dc.floatField)                        /* 7. */
     .thenComparingDouble(dc -> dc.doubleField)                       /* 8. */
     .thenComparing(dc -> dc.comparableClassField,
        Comparator.nullsLast(OtherComparableClass::compareTo));       /* 9. */
  /* [...] Fields omitted. */

  /* [...] Constructors omitted. */
  @Override
  public int compareTo(ComparableJava8DemoClass otherDemoClass) {
     return DEMO_CLASS_COMPARATOR.compare(this, otherDemoClass);
  }
  /* [...] Other methods omitted. */
}
```

Listing 5

operatoren „<" und „>" für ganzzahlige Typen übersehen hat, aber nein, in diesen Methoden finden wirklich keinerlei Zaubereien statt. In dieser Hinsicht sieht er also keinen Vorteil in der Verwendung der statischen „compare"-Methoden der Wrapper-Klassen, aus Sicht der Konsistenz ist es jedoch absolut sinnvoll. Die Benutzung statischer „compare"-Methoden ist nun für alle primitiven Typen in Java gleich, was mehr als Grund genug ist, daran festzuhalten *(siehe Listing 4)*.

Die neue Art seit Java 8

Wie man sehen kann, haben sich die Dinge für den Vergleich des Referenz-Typs (siehe Vergleich 9) noch immer nicht geändert. Sein Code-Block ist wirklich langatmig und fehleranfällig. Java 8 führte „Optional" ein, die als kleine Container für einzelne Objekte aufgefasst werden können und entweder leer sind oder eben das Objekt enthalten. Ihr Vorteil beruht auf der Tatsache, dass ihr Inhalt nie „null" sein kann.

Dieses Konzept könnte vielleicht hilfreich sein, um die Fallunterscheidung ein wenig zu vereinfachen. Man denke dabei an die Verwendung der „Optional"-Methoden „ifPresent" oder „ifPresentOrElse" (seit Java 9), die die Menge an „if-else"-Befehlen reduzieren könnten. Da ich jedoch einen komplett anderen Ansatz vorstellen werde, müssen wir diese Idee nicht weiter verfolgen.

Beginnend mit Java 8 können Schnittstellen nun auch statische und Default-Implementierungen enthalten. Das ist genau das, was „Comparator" macht: Er wurde mit einer Vielzahl von Methoden ausgestattet, die ganz grob als „Fabrik-Methoden" verstanden werden können. Sie erlauben das schrittweise Erstellen von „Comparatoren", die wiederum entweder direkt oder innerhalb der „compareTo"-Methode einer Klasse aufgerufen werden können *(siehe Listing 5)*.

Erneut bedürfen ein paar Punkte einer Erläuterung:

- Die Klasse enthält nun ein statisches und finales „Comparator"-Attribut (mit dem Namen „DEMO_CLASS_COMPARATOR"). Es wird innerhalb der Klassenmethode „compareTo" aufgerufen, was die „compareTo"-Methode auf eine einzige Zeile reduziert. Wenn der „Comparator" als „innere Klasse" deklariert wird (am besten als „statische innere Klasse"), kann er auf alle (üblicherweise privaten) Attribute der äußeren Klasse zugreifen. Wird der „Comparator" irgendwo außerhalb deklariert, dann ist er normalerweise abhängig von Getter-Methoden, um auf die Attribut-Werte zugreifen zu können. Es ist allerdings nicht wirklich entscheidend, ob solch ein „Comparator" nun „intern" oder „extern" verwendet wird – so oder so ist seine Benutzung deutlich konsistenter geworden.
- Der „Comparator" wird Schritt für Schritt erstellt. Man beginnt mit einer der statischen „Comparator.comparing[XXX]"-Methoden für das höchstwertige Attribut und hängt dann die anderen Attribute in der gewünschten Reihenfolge unter Benutzung der „thenComparing[XXX]"-Methoden an. Für „boolean" und alle Referenz-Attribute verwendet man „comparing" und „thenComparing". Für „byte", „char", „short" und „int" verwendet man „[then]comparingInt". Für „long" nimmt man „[then]comparingLong", für „float" und „double" kommt „[then]comparingDouble" zum Einsatz . Wenn man die Methoden für primitive Typen nicht überall dort verwendet, wo sie möglich wären, dann wird Java Autoboxing durchführen, was fehleranfällig sein und die Performance signifikant beeinträchtigen kann. Leider stellt Java keine „[then]comparingBoolean"-Methode zur Verfügung, sodass man sich für diesen Typ ausnahmsweise mit Autoboxing abfinden muss (oder das Boxing manuell vornimmt, so wie ich im obigen Code-Beispiel).
- Jede „[then]comparing[XXX]"-Methode benötigt eine sogenannte „Key-Extractor-Funktion" als Parameter, was üblicherweise mithilfe von Lambdas implementiert wird. Die Eingabe für so einen Key-Extractor ist diejenige Klasse, mit der verglichen werden soll (hier die Demoklasse, referenziert durch „dc"); die Ausgabe ist das extrahierte Attribut. Wenn der Lambda-Ausdruck direkten Zugang zu den privaten Attributen hat, dann ist es einfach; wenn nicht, dann muss man die entsprechenden Getter-Methoden aufrufen, wie oben erwähnt.
- Der Java-Compiler ist in der Regel sehr gut darin, die korrekten Typen bei Lambda-Ausdrücken herauszufinden. Allerdings scheint es hier Probleme zu geben. Aus diesem Grund muss der Typ der Eingabe zumindest im ersten Befehl ausdrücklich spezi-

fiziert sein, wie im Beispiel zu sehen ist: „(ComparableJava8DemoClass dc) -> Boolean.valueOf(dc.booleanField))". Die Typen der darauffolgenden Lambdas sollten danach automatisch bestimmt werden können.

- Die „[then]comparing"-Methoden (nicht für „int", „long" oder „double") erlauben die Angabe eines weiteren „Comparator" im zweiten Methoden-Parameter. Wenn man sich Vergleich 9 im Code-Beispiel anschaut, sieht man die zusätzliche Spezifikation „Comparator.nullsLast(OtherComparableClass::compareTo)". „Comparator.nullsLast" ist eine statische Methode, die einen „Comparator" mit weiterer Funktionalität im Umgang mit „null"-Werten ausstattet – anstatt eine „NullPointerException" zu werfen und ohne langatmige Fallunterscheidungen, wie in den Code-Beispielen zu sehen war. Wenn „Comparator.nullsLast" verwendet wird, dann kommt „null" zuletzt. Wer möchte, dass „null" zuerst kommt, verwendet einfach „Comparator.nullsFirst".

Zusammenfassung und Ausblick

Das war's für den ersten Teil. Die Art der Implementierung natürlicher Ordnung in Java 8 sieht modern und prägnant aus. Ihre Verwendung ist konsistent, sicher und – am wichtigsten – einfach. Die vollständigen Quelltexte zu diesem Artikel können unter *link.simplexacode.ch/8sk8* zur privaten Nutzung heruntergeladen werden. Im Teil 2 zu diesem Thema werden in der nächsten Ausgabe einige praxisrelevante Beispiele für die Implementierung von Ordnung und die Sortierung realistischerer Werteklassen gezeigt.

Hinweis: Die englische Version des Artikels finden Sie unter *link.simplexacode.ch/vq29*.

Christian Heitzmann
christian.heitzmann@simplexacode.ch

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Software-Entwicklung, -Schulung und -Beratung vor allem für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit fünfzehn Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.

Implementierung von Event-Storming-Modellen mit Axon

Sven-Torben Janus, Conciso GmbH

Seit einigen Jahren erfreut sich Domain-driven Design (DDD) großer Beliebtheit. Damit einhergehend rücken leichtgewichtige Modellierungsmethoden in den Fokus. Eine dieser Methoden ist Event Storming. Es hilft Teams, ein gemeinschaftliches, interdisziplinäres Verständnis des Problem- und Lösungsraums zu entwickeln, das sich in Form vieler Post-its an einer Wand manifestiert. Aber wie überführt man eine Menge von Post-its in ausführbaren Code? Ein patternbasierter Ansatz kann hier helfen.

Anderthalb Jahrzehnte nach dem Erscheinen von Eric Evans „Big Blue Book" *[1]* ist Domain-driven Design (DDD) präsenter als je zuvor. Dies ist nicht zuletzt auf den anhaltenden Microservice-Hype zurückzuführen. Die Zerlegung großer Systeme in viele kleinere Teile erfordert in der Regel eine strikte Trennung von Funktionalitäten unter fachlichen Gesichtspunkten. Insbesondere die strategischen Patterns des DDD werden daher vielfach für entsprechend notwendige Designentscheidungen angeführt. Die Anwendung dieser Patterns erfordert jedoch immer ein gutes Verständnis des fachlichen Problem- und Lösungsraums. Leichtgewichtige Modellierungsmethoden rücken dabei immer stärker in den Fokus.

Wer schon einmal gesehen hat, wie Kollegen diskutieren und währenddessen eine Wand mit orangenen Post-its tapezieren, hat mit hoher Wahrscheinlichkeit eine Event Storming *[2]* Session beobachtet. Mit Event Storming hat sich eine Workshop-Methode etabliert, die es den beteiligten Personen – von Fachexperten, strategischen Entscheidern über User Experience Designer, Product Owner bis hin zu Entwicklern – ermöglicht, genau dieses Verständnis interdisziplinär und gemeinschaftlich zu erarbeiten. Das gemeinsame Verständnis spiegelt sich dabei in Form eines Domänenmodells wider. Diese können je nach Ausrichtung und Ziel des Workshops von grob granularen „Big Picture"-Modellen bis hin zu detaillierten, implementierungsnahen Modellen variieren. Aber wie kann man behaupten, dass eine Menge von Post-its implementierungsnah sei? Wie kann man Post-its in ausführbaren Code überführen?

Aus anderen Bereichen der Modellierung werden hierzu häufig patternbasierte Ansätze genutzt. Das Überführen von formalen Modellen, wie zum Beispiel UML, in eine objektorientierte Programmiersprache oder ein relationales Datenbankmodell funktioniert im Wesentlichen patternbasiert. Entsprechend sind auch (Meta-)Pattern im Sinne der „Gang of Four" *[3]* recht eindeutig in Code überführ- und ausführbar.

Event Storming kennt im Kern ebenfalls Patterns. Anhand einiger ausgewählter wird nachfolgend auf Basis des Axon Frameworks *[4]* gezeigt, wie sie zur Implementierung von Event-Storming-Modellen

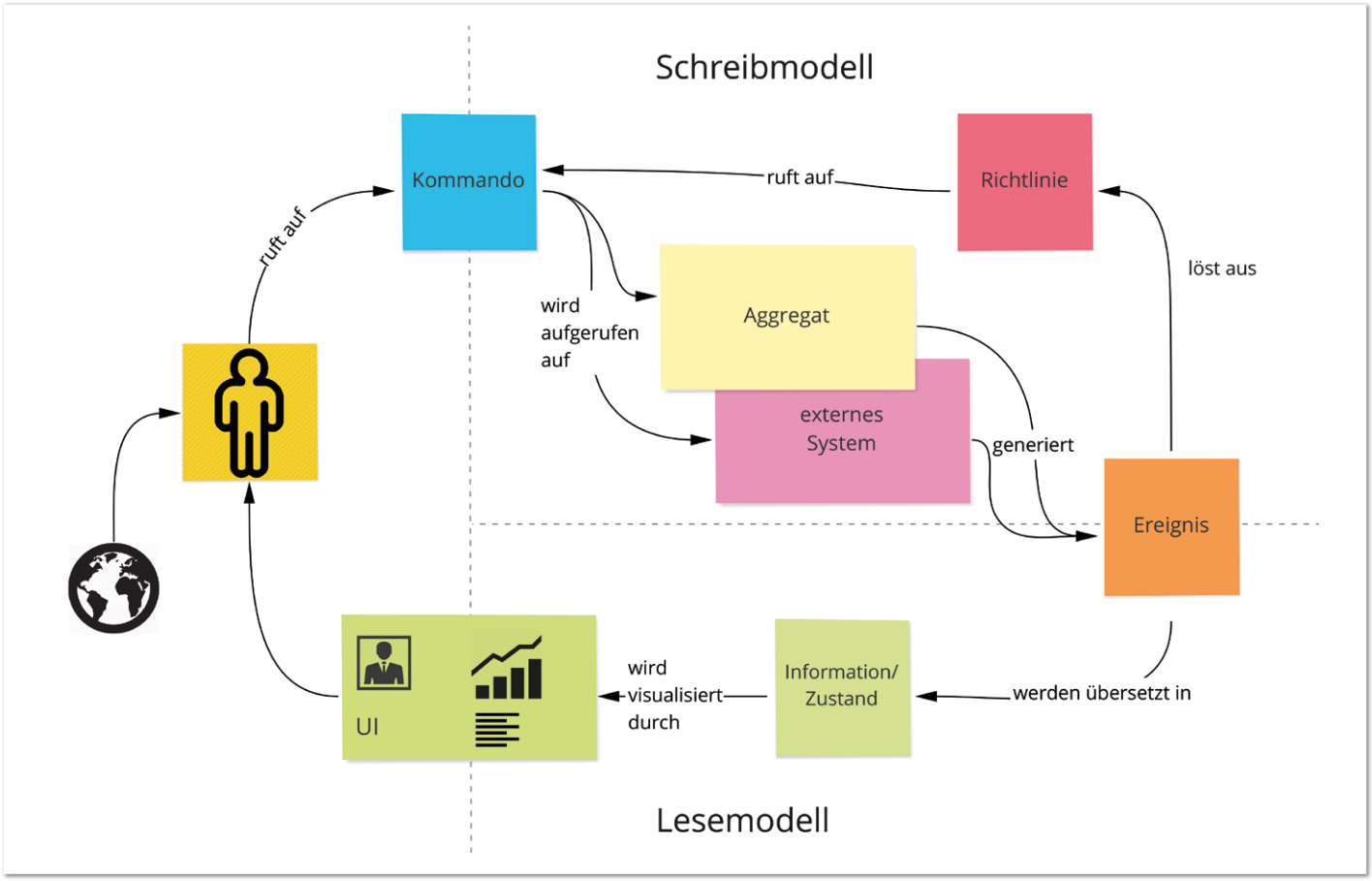


Abbildung 1: Event Storming – Big Picture

nutzbar sind. Ziel dabei soll sein, das gemeinsame Verständnis im Sinne einer Ubiquitous Language *[5]* möglichst eins zu eins in Code zu überführen.

Event Storming

Beim Event Storming startet die Modellierung in der Regel mit dem Aufschreiben von Schlüsselereignissen der Fachdomäne auf orangefarbige Post-its. Die Ereignisse sind dabei in zeitlicher Reihenfolge sortiert. Ziel ist es vor allem, den Prozess und den gesamten Kontext des Problems und nicht jedes Detail zu modellieren.

Stehen die wesentlichen Ereignisse einmal fest, stellt sich die Frage, durch welche Aktionen die Ereignisse ausgelöst werden. Um diese Frage zu beantworten, müssen die Teilnehmer einen rückwärtsgewandten Blick einnehmen. Aktionen werden im Event Storming durch sogenannte „Kommandos" auf blauen Post-its modelliert. Oftmals sind sie durch einen Benutzer initiiert. Sie können allerdings auch durch externe Systeme oder Richtlinien ausgelöst werden. Damit ein Benutzer ein Kommando auslösen beziehungsweise eine Entscheidung treffen kann, benötigt er entweder Informationen aus der realen Welt oder über den Zustand (grüne Post-its) der Anwendung. Letzterer kann aus den in der Domäne aufgetretenen Ereignissen abgeleitet werden.

Ist der wesentliche Prozess einmal definiert und klar, können vor allem die Entwickler in ein implementierungsnäheres Design einsteigen. Hierzu werden Aggregate (breite gelbe Post-its) und externe Systeme (breite pinke Post-its) identifiziert, die Verantwortlichkei-

Abbildung 2: Beispielprozess: Kunde wählt Produkte aus

ten für die Verarbeitung von Kommandos übernehmen. Im Zusammenhang ergibt sich somit ein Bild wie in *Abbildung 1* dargestellt.

Ein Modellbeispiel

Zur Veranschaulichung der Implementierung eines durch Event Storming entstandenen Modells soll ein klassisches Shop-Beispiel dienen. Zu Beginn des betrachteten Prozesses steht ein Kunde, der einen Shop betritt. Immer, wenn der Kunde den Shop betritt, soll ihm ein Warenkorb bereitgestellt werden, in den er Produkte legen kann *(siehe Abbildung 2)*.

Nachdem der Kunde einige Produkte ausgewählt hat, bekommt er seinen Warenkorb, den er anschließend bestellen kann, inklusive Gesamtpreis und möglicher Rabatte angezeigt. Daraufhin wird eine Bestellung erzeugt und das System quittiert dies mit dem Ereignis, dass der Warenkorb bestellt wurde *(Abbildung 3)*.

Im Anschluss an die Bestellung erfolgen einige automatische Schritte. Per Richtlinie wird festgelegt, dass bei einem Versand außerhalb der EU die Waren zum Zollverfahren angemeldet werden müssen. Gleichzeitig werden immer die Versandart bestimmt sowie eine Rechnung gestellt *(siehe Abbildung 4)*.

Nach der Rechnungsstellung kann der Kunde seine Rechnung bezahlen. Das Rechnungssystem quittiert dies mit einem entsprechenden Ereignis. Durch die Bezahlung der Rechnung wird die Ware automatisch zum Versand freigegeben *(Abbildung 5)*.

Abbildung 3: Beispielprozess: Kunde bestellt Waren

Abbildung 4: Beispielprozess: Rechnungsstellung, Versandart und Zollanmeldung

Abbildung 5: Beispielprozess: Kunde bezahlt Rechnung

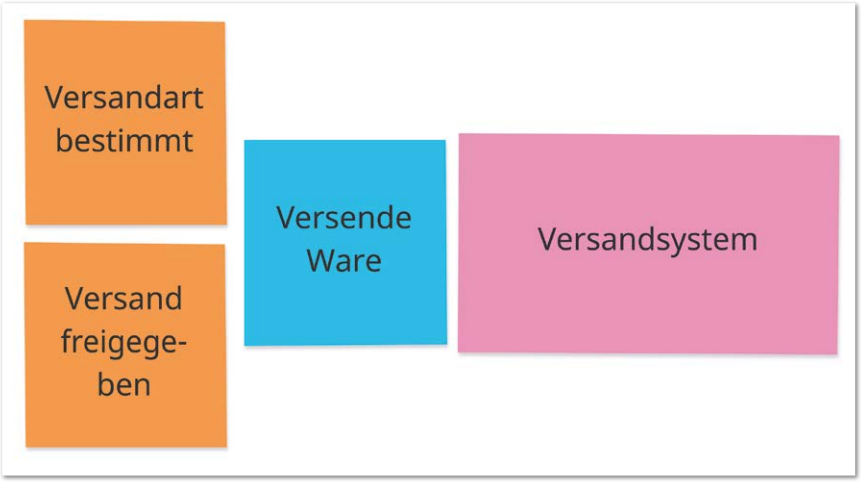


Abbildung 6: Beispielprozess: Warenversand

Wenn der Versand freigegeben ist und die Versandart bestimmt wurde, wird die Ware versendet *(siehe Abbildung 6)*.

Das Axon Framework

Für die Implementierung kommt das Axon Framework zum Einsatz. Axon ist ein auf Java basierendes Command Query Responsibility Segregation (CQRS) *[6]* Framework. CQRS trennt lesende Anfragen, die den Zustand der Anwendung nicht verändern, von schreibenden, zustandsverändernden Aktionen. Im Kern seiner Architektur folgt Axon dabei dem Zyklus in *Abbildung 1*. Es stellt einige Building Blocks bereit, die eine Implementierung des Zyklus recht einfach gestalten. Über sogenannte „Command Handler" können Kommandos eines Benutzers ausgeführt werden. Die Command Handler agieren auf dem Domänenmodell, das im Kern aus Aggregaten besteht. Aggregate können mittels Repositories gespeichert werden. Dabei unterstützt Axon neben dem Speichern des aktuellen Zustands eines Aggregats auch die Speicherung der Ereignisse, die zu diesem Zustand geführt haben. Axon kombiniert an dieser Stelle CQRS mit Event Sourcing *[7]*. Ereignisse werden zugleich auf einen Event Bus publiziert. Von dort können sie weitere Aktionen auf den Aggregaten auslösen. Gleichzeitig können sie jedoch auch mit Event Handlern verarbeitet werden. Diese Event Handler erlauben es, weitere Modelle, die für lesende Anfragen optimiert sind, zu persistieren. Diese wiederum können durch Query Handler abgefragt und somit Informationen auf der UI angezeigt werden.

Mit diesen Building Blocks bietet Axon alle Voraussetzungen, um gängige Patterns aus Event-Storming-Modellen zu implementieren. Die am häufigsten anzutreffenden sind neben vielen weiteren vermutlich die nachfolgenden drei *(siehe Abbildung 7)*.

Pattern: Automation

Automation ist eines der einfachsten Patterns im Event Storming. Das Pattern besagt, dass bei Eintritt eines Ereignisses automatisch ein weiteres Ereignis oder Kommando ausgelöst wird. Dieses Pattern lässt sich im Beispielprozess gut erkennen. Der Versand

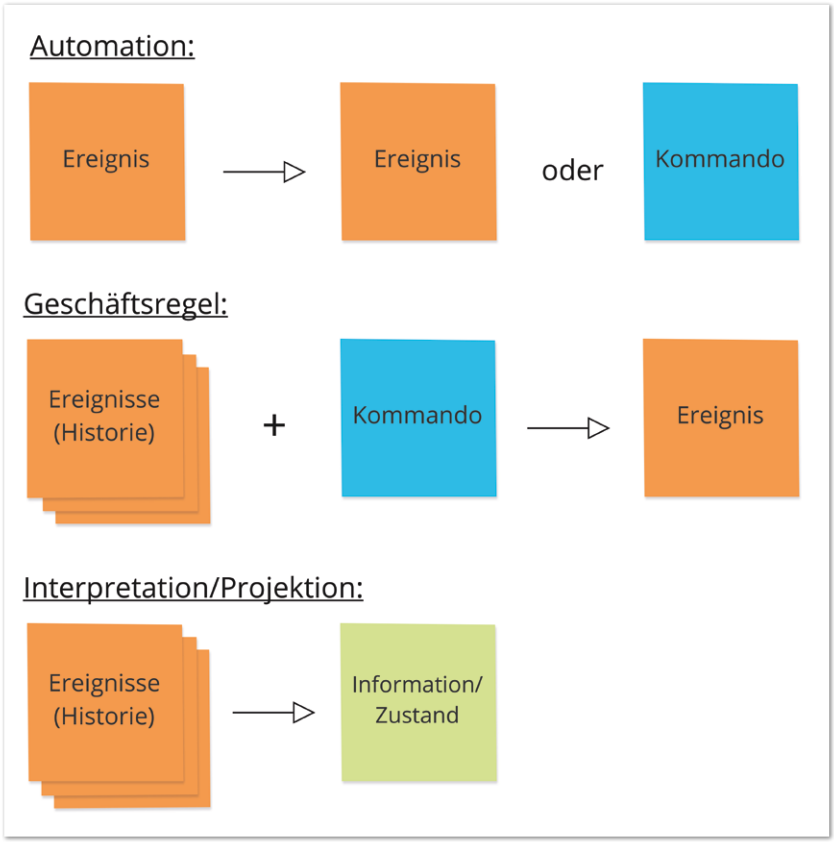


Abbildung 7: Auswahl häufig auftretender Patterns beim Event Storming

von Waren wird automatisch freigegeben (Ereignis), wenn die Rechnung bezahlt wurde (Ereignis).

In Axon lässt sich dieses Pattern mit einem einfachen Event Handler in Kombination mit Axons Event Bus implementieren. Mit Event Handlern kann auf Ereignisse reagiert werden. Der Event Bus erlaubt das Versenden weiterer Ereignisse. Für einen solchen Event Handler muss lediglich eine Methode mit @EventHandler annotiert werden. Sie nimmt als Parameter ein Ereignis vom Typ „RechnungBezahlt" entgegen. Der Typ lässt sich einfach als POJO implementieren und stellt ereignisbezogene Informationen bereit. In diesem Fall enthält das Ereignis die „WarenkorbId", um mitzuteilen, für welchen Warenkorb die Rechnung bezahlt wurde. Um zu signalisieren, dass automatisch der Versand freigegeben wurde, kann einfach ein entsprechendes Ereignis mithilfe des Event Bus publiziert werden *(siehe Listing 1)*.

Anstelle eines Ereignisses kann bei der Automation auf ein Ereignis auch ein Kommando folgen. Zum Beispiel soll einem Kunden, wann immer er den Shop betreten hat (Ereignis), automatisch ein Warenkorb bereitgestellt werden (Kommando). Das Pattern lässt sich mit Axon ebenfalls recht einfach implementieren *(siehe Listing 2)*.

Auch an dieser Stelle muss einfach nur ein Event Handler wie zuvor implementiert werden. Die dazugehörige Methode nimmt als Parameter ein Ereignis vom Typ „KundeHatShopBetreten" entgegen. Anstelle in der Methode ein weiteres Ereignis zu publizieren, kann Axon über das sogenannte „Command Gateway" aber auch ein Kommando auslösen. Das Gateway lässt sich wie auch der Event Bus einfach injizieren. Mit der Methode „sendAndWait" können ein Kommando ausgelöst und auf dessen Verarbeitung gewartet werden. Es stehen allerdings auch Methoden zur asynchronen Verarbeitung bereit. Auch Kommandos sind einfache POJOs und enthalten die Informationen, die benötigt werden, um ein Kommando auszuführen. In diesem Falle die „KundenId", um zu signalisieren, für welchen Kunden ein Warenkorb bereitgestellt werden soll.

```
class Shop {

  @Inject
  EventBus eventBus;

  @EventHandler
  void on(RechnungBezahlt event) {
     eventBus.publish(GenericEventMessage.asEventMessage(new
VersandFreigegeben(event.getWarenkorbId())));
  }

}
```

Listing 1

```
class Shop {

  @Inject
  CommandGateway commandGateway;

  @EventHandler
  void on(KundeHatShopBetreten event) {
     UUID warenkorbId = commandGateway.sendAndWait(new
StelleWarenkorbBereit (event.getKundenId()));
     // …
  }

}
```

Listing 2

Pattern: Geschäftsregel

Kommandos sind in der Regel auch Auslöser einer Geschäftsregel. Eine Geschäftsregel besteht immer aus einer Reihe von Ereignissen, der sogenannten „Historie", und einem auslösenden Kommando. Als Ergebnis der Geschäftsregel wird ein weiteres Ereignis signalisiert.

Wie Kommandos durch das Command Gateway ausgelöst werden können, wurde bereits gezeigt. Zur Verarbeitung von Kommandos

```
public class Warenkorb {

  @AggregateIdentifier
  private UUID id;
  private UUID kundenId;

}

public class WarenkorbHandler {

  @Inject
  Repository<Warenkorb> warenkoerbe;

  @CommandHandler
  public UUID handle(StelleWarenkorbBereit command) throws Exception {
     UUID warenkorbId = UUID.randomUUID();
     Aggregate<Warenkorb> warenkorb = warenkoerbe.newInstance(() -> new Warenkorb(warenkorbId, command.getKunde-
nId()));
     return (UUID) warenkorb.identifier();
  }

  @CommandHandler
  public void handle(LegeProduktInWarenkorb command) {
     Aggregate<Warenkorb> warenkorb = warenkoerbe.load(command.getWarenkorbId().toString());
     // ...
  }

}
```

Listing 3

```
public class Warenkorb {

  @AggregateIdentifier
  private UUID id;
  private UUID kundenId;

  @CommandHandler
  public Warenkorb(StelleWarenkorbBereit command) {
     this(command.getWarenkorbId(), command.getKundenId());
  }

  @CommandHandler
  public void handle(LegeProduktInWarenkorb command) {
     if (command.getAnzahl() > 0) {
        // …
     } else {
        throw new IllegalArgumentException("Anzahl
muss größer 0 sein.");
     }
  }
}
```

Listing 4

```
public interface LegeProduktInWarenkorb {

  @TargetAggregateIdentifier
  UUID getWarenkorbId();

  UUID getProduktId();

  int getAnzahl();

}
```

Listing 5

ebenfalls nur eine Methode annotiert werden – diesmal mittels @CommandHandler. Die Methode nimmt als Parameter ein Kommando vom Typ „StelleWarenkorbBereit" entgegen *(siehe Listing 3)*.
Die Methode implementiert die Logik, die zur Ausführung des Kommandos notwendig ist. Im Beispiel werden eine „WarenkorbId" erzeugt und ein Warenkorb in Form eines Aggregats mit dieser ID angelegt. Durch die Annotation `@AggregateIdentifier` wird dem Framework signalisiert, welcher Wert als Identifier des Aggregats (hier des Warenkorbs) dient. Die eigentliche Anlage des Aggregats erfolgt über Axons Repository Interface. Das Repository stellt Methoden zum Speichern und Laden bereit.

Bei dieser Implementierungsvariante fällt auf, dass recht viel Boiler-Plate-Code für den Zugriff auf das Repository geschrieben werden muss. Die zweite Möglichkeit der Implementierung ist deutlich weniger geschwätzig. Wann immer ein Aggregat erzeugt werden soll, kann nämlich auch ein Konstruktor als Command Handler genutzt werden *(siehe Listing 4)*.

Allerdings fällt auf, dass die Generierung der „WarenkorbId" nun bereits in das Kommando kodiert werden muss, da diese Form von Command Handler keinen Rückgabewert erlaubt. Soll das Aggregat weitere Kommandos ausführen, ist es für den Zugriff auf das Aggregat außerdem notwendig, innerhalb des Kommandos festzulegen, welche Eigenschaft den Identifier des Aggregats definiert. Dies erfolgt, wie in *Listing 5* gezeigt, einfach mit der Annotation `@TargetAggregateIdentifier`.

```
public class Warenkorb {

  @AggregateIdentifier
  private UUID id;
  private UUID kundenId;
  private Map<Produkt, Integer> produkte;

  @CommandHandler
  public Warenkorb(StelleWarenkorbBereit command) {
     this(command.getWarenkorbId(), command.getKundenId());
  }

  public Warenkorb(UUID warenkorbId, UUID kundenId) {
     AggregateLifecycle.apply(new WarenkorbBereitgestellt(warenkorbId, kundenId));
  }

  @CommandHandler
  public void handle(LegeProduktInWarenkorb command) {
     if (command.getAnzahl() > 0) {
        AggregateLifecycle.apply(new ProduktInWarenkorbGelegt(command.getProduktId(), command.getAnzahl()));
     } else {
        throw new IllegalArgumentException("Anzahl muss größer 0 sein.");
     }
  }

  @EventSourcingHandler
  public void on(WarenkorbBereitgestellt event) {
     this.id = event.getWarenkorbId();
     this.kundenId = event.getKundenId();
     this.produkte = new HashMap<>();
  }

  @EventSourcingHandler
  public void on(ProduktInWarenkorbGelegt event) {
     Produkt neuesProdukt = new Produkt(event.getProduktId());
     produkte.compute(neuesProdukt, (produkt, anzahlImWarenkorb) ->
        anzahlImWarenkorb == null ? event.getAnzahl() : anzahlImWarenkorb + event.getAnzahl());
  }

}
```

Listing 6

Nach Verarbeitung des Kommandos muss als Ergebnis der Geschäftsregel wieder ein Ereignis signalisiert werden. Auch hier kennt Axon zwei grundlegende Mechanismen. Zum einen lässt sich auch hier einfach der Event Bus, wie zuvor gezeigt, nutzen. Zum anderen stellt Axon aber auch einen Event-Sourcing-Mechanismus bereit. Für das zuvor gezeigte Beispiel sieht das wie in *Listing 6* beschrieben aus.

Anstatt in einem Command Handler den Zustand eines Aggregats direkt zu verändern, wird im Command Handler nur geprüft, ob das Kommando (z.B. basierend auf dem aktuellen Zustand des Aggregats) angewendet werden kann oder darf. Wenn dies so ist, wird einfach ein Ereignis signalisiert. Diesmal wird dazu jedoch nicht der Event Bus direkt verwendet, sondern die AggregateLifecycle.apply-Methode. Diese bewirkt zum einen, dass das Ereignis mittels Event Bus signalisiert wird, speichert aber zugleich, dem Event-Sourcing-Ansatz entsprechend, das Ereignis in der Historie aller Ereignisse des Aggregats. Dies erlaubt Axon, den Zustand des Aggregats jederzeit aus der Historie der Ereignisse wiederherzustellen. Dazu müssen, ähnlich zu Event Handlern, sogenannte „Event Sourcing Handler" implementiert werden. Auch dies geschieht analog zu Event Handlern einfach per Annotation `@EventSourcingHandler`. Beim Wiederherstellen des Zustands des Aggregats wird Axon die Event Sourcing Handler in der Reihenfolge der Ereignisse wieder aufrufen.

Neben diesen sehr einfachen Geschäftsregeln kommt es in der Praxis natürlich auch vor, dass langlebigere Geschäftsprozesse automatisiert werden müssen. Axon stellt hierzu sogenannte „Sagas" bereit. Eine Saga erlaubt es, eine Reihe lokaler Transaktionen oder in diesem Fall Kommandos auszuführen (und gegebenenfalls zu kompensieren). Aus dem Beispielprozess bietet sich hier eine Folge von Ereignissen und Kommandos an, die zur Abwicklung der Bestellung dienen. Wurde ein Warenkorb bestellt, sollen die Versandart bestimmt und die Rechnung gestellt werden. Wenn die Rechnung bezahlt wurde, wird die Ware automatisch zum Versand freigegeben. Wurden die Ware zum Versand freigegeben und die Versandart bestimmt, soll die Ware versendet werden. Dies lässt sich mit einer Saga wie in *Listing 7* beschrieben abbilden.

Sagas nutzen spezielle Event Handler, die mithilfe der `@SagaEventHandler`-Annotation definiert werden. Mit der Annotation `@StartSaga` kann definiert werden, welcher der Event Handler eine neue Instanz der Saga startet. Die Saga wird durch eine mithilfe von associationProperty definierte Property des

```
public class Bestellabwicklung {

  private UUID warenkorbId;
  private boolean rechnungBezahlt = false;
  private boolean versandartBestimmt = false;

  @Inject
  private transient CommandGateway;

  @StartSaga
  @SagaEventHandler(associationProperty = "warenkorbId")
  public void handle(WarenkorbBestellt event) {
     warenkorbId = event.getWarenkorbId();
     String versandnummer = erzeugeVersandnummer();
     String rechnungsnummer = erzeugeRechnungsnummer();
     SagaLifecycle.associateWith("versandnummer", versandnummer);
     SagaLifecycle.associateWith("rechnungsnummer", rechnungsnummer);
     commandGateway.send(new BestimmeVersandart(warenkorbId, versandnummer));
     commandGateway.send(new StelleRechnung(warenkorbId, rechnungsnummer));
  }

  @SagaEventHandler(associationProperty = "versandnummer")
  public void handle(VersandartBestimmt event) {
     versandartBestimmt = true;
     if (rechnungBezahlt) {
        commandGateway.send(new VersendeWare(warenkorbId));
     }
  }

  @SagaEventHandler(associationProperty = "rechnungsnummer")
  public void handle(RechnungBezahlt event) {
     rechnungBezahlt = true;
  if (versandartBestimmt) {
        commandGateway.send(new VersendeWare(warenkorbId));
     }
  }

  @SagaEventHandler(associationProperty = "warenkorbId")
  public void handle(WareVersand event) {
     SagaLifecycle.end();
  }

  // ...

}
```

Listing 7

startenden Ereignisses eindeutig identifiziert. Wie im startenden Event Handler und den weiteren `@SagaEventHandler`-Annotationen zu sehen, können aber durch die SagaLifecycle.associateWith-Methode auch andere Properties und deren Werte zur Assoziation von Ereignissen mit einer bestimmten Saga-Instanz herangezogen werden. Zum Auslösen weiterer Kommandos wird auch innerhalb von Sagas einfach das Command Gateway genutzt. Zum Beenden einer Saga steht in SagaLifecycle die „end"-Methode zur Verfügung. Mithilfe des unterstützten Lebenszyklus von Sagas ist es somit möglich, auch langlebige Geschäftsprozesse abzubilden und zu automatisieren.

Pattern: Interpretation/Projektion

An Stellen, an denen ein Geschäftsprozess manuellen Eingriff beziehungsweise Benutzerinteraktion erfordert, ist es in der Regel notwendig, den Zustand des Systems oder eines Teils des Systems zu visualisieren. Im Beispiel ist das der Fall, wenn ein Kunde seine Rechnung bezahlt oder Waren in den Warenkorb legt. Hier wären beispielsweise die Rechnung oder die Produkte im Warenkorb zu visualisieren. Wie zuvor gesehen, können diese Informationen einfach über Aggregate, die durch Repositories in Axon zugreifbar sind, ermittelt werden.

Manchmal ist der Zugriff auf ein Aggregat jedoch nicht ausreichend. Soll einem Kunden beispielsweise eine Produktempfehlung gemacht werden, die alle Produkte umfasst, die er aus einem Warenkorb entfernt und letztlich noch nie bestellt hat, müssen prinzipiell alle Warenkörbe und Bestellungen durchsucht werden. Als Alternative bietet Axon das Konzept von Query Handlern an. *Listing 8* zeigt eine einfache Implementierung.

Query Handler sind ähnlich wie Event Handler, lösen aber keine Ereignisse oder Kommandos aus. Vielmehr stellen sie Informationen bereit, die zuvor über Event Handler in Lesemodelle persistiert wurden. Dies folgt einem konsequenten Command-Query-Responsibility-Segregation(CQRS)-Ansatz. Aufwendige Datenbankabfragen sind somit nicht nötig. Stattdessen wird der Zustand aus einer Teilmenge der aufgetretenen Ereignisse projiziert. Die benötigten Informationen können so direkt aus dem Lesemodell abgefragt werden.

Technische Integration

Es zeigt sich, wie einfach mit einigen Building Blocks eine patternbasierte Überführung eines „Post-it-Modells" in eine ausführbare Implementierung ist. Sicherlich sind hier die wichtigen fachlichen Details wegabstrahiert. Dennoch wird deutlich, wie ein solcher Ansatz mindestens ein schnelles Prototyping des Modells erlaubt. Ein solcher Ansatz erfordert sicherlich noch nicht den Einsatz eines Frameworks. Mit einfachen Java-Board-Mitteln lassen sich solche Building Blocks auch schnell selbst entwickeln *[8]*. Spätestens, wenn man aber eine ernsthafte Implementierung auf diese Weise in Erwägung zieht, kann Axon weitere Stärken ausspielen. Denn auch im Bereich der technischen Integration bietet Axon aufgrund seiner guten Schnittstellenabstraktionen vielfältige Konfigurationsmöglichkeiten und eine gute Integration mit anderen Frameworks und Technologien. AMQP-basierte Messaging-Systeme, aber auch Kafka, können ebenso einfach angebunden werden wie JPA-fähige Datenbanken oder eine MongoDB. Aber auch querschnittliche Themen wie Transaktionssicherheit oder Schemamigration werden durch ein Unit-of-Work-Pattern und Event Upcaster gelöst. Vor allem steht jedoch eine sehr gute Integration mittels Spring und Support von Spring Boot zur Verfügung.

Fazit

Event Storming ist ein Tool, mit dem es möglich ist, auch implementierungsnahe Domänenmodelle zu erarbeiten. Anhand der vorgestellten, immer wiederkehrenden Patterns können diese einfach in eine Implementierung überführt werden. Dies ist besonders einfach mit Frameworks wie Axon zu realisieren, die dafür notwendige Building Blocks bereitstellen und zugleich technische Aspekte nicht außer Acht lassen. Sie erleichtern es, eine allgegenwärtige Sprache (Ubiquitous Language) zu etablieren, die sich vom gemeinsam entwickelten Domänenmodell während des Event Storming bis in die Implementierung wiederfindet.

Referenzen

[1] *http://dddcommunity.org/book/evans_2003/*
[2] *https://www.eventstorming.com/*
[3] *http://wiki.c2.com/?GangOfFour*
[4] *https://axoniq.io/*
[5] *https://martinfowler.com/bliki/UbiquitousLanguage.html*
[6] *https://martinfowler.com/bliki/CQRS.html*
[7] *https://martinfowler.com/eaaDev/EventSourcing.html*
[8] *https://conciso.de/blog/2018/05/23/kombination-cqrs-event-sourcing-java-teil3/*

Hinweis: Unter *„http://www.ijug.eu/go/javaaktuell/201903/listings"* finden Sie das fehlende *Listing 8*.

Sven-Torben Janus
sven-torben.janus@conciso.de

Sven-Torben Janus arbeitet als Senior Software Architect bei der Conciso GmbH, wo er den Bereich Softwarearchitektur mitverantwortet. Er befürwortet einen agilen und praktikablen Entwurf von Softwarearchitekturen. Sein Unbehagen über technologiegetriebene Designs führte ihn zum Domain-driven Design und zur Gründung des DDD Meetups Rhein/Main. Er twittert unter @sventorben und teilt sein Wissen auf Konferenzen, Artikeln und im Blog auf https://conciso.de/blog/.

iOS-Apps in Java

Thomas Künneth, Mathema Software GmbH

Das Open-Source-Projekt Multi-OS Engine ermöglicht die Erstellung nativer iOS-Anwendungen in Java. Dieser Artikel zeigt die Nutzung des Frameworks anhand einer einfach nachvollziehbaren Beispiel-App. Er behandelt neben Installationsvoraussetzungen und Setup die Implementierung der Geschäftslogik sowie der Benutzeroberfläche. Das vollständige Projekt steht in einem GitHub-Repository zur Verfügung [1].

Apps für iOS werden in der Regel in Swift geschrieben. Nur selten kommt noch Objective-C zum Einsatz. Dem Entwickler steht mit Xcode eine mächtige IDE zur Verfügung. Sie unterstützt beim Debuggen, Profilen und Testen, archiviert und paketiert die App und lädt sie in den Store hoch. Auch die Verwaltung von Geräte- und Team-Zertifikaten findet in Xcode statt. Apple-typisch ist die Nutzung des Werkzeugs komfortabel und intuitiv. Dennoch ist für einen effizienten Einsatz einiges an Know-how erforderlich. Dies gilt noch viel mehr für die eigentliche Programmierung. Die bloße Kenntnis der Swift-Syntax ist bei Weitem nicht genug. Wie tickt iOS? Welche Klassen gibt es? Wie wird die Benutzeroberfläche erstellt? In einem Unternehmen, das voll auf iOS-Apps setzt, führt an einem entsprechenden Wissenserwerb kein Weg vorbei. Was aber, wenn eine längerfristige Strategie noch nicht festgelegt wurde? Oder wenn abzusehen ist, dass nur wenige Apps entstehen werden? Kommt eine externe Vergabe nicht in Betracht, sollte vorhandenes Wissen so gut wie möglich genutzt werden. In den letzten 20 Jahren war Java eine der am häufigsten genutzten Technologien für die Erstellung von Unternehmensanwendungen. Es liegt also nahe, iOS-Apps in Java zu schreiben. Damit das funktioniert, muss der Java-Quelltext in native ARM-Maschinensprache übersetzt werden. Außerdem werden eine Art Laufzeitumgebung benötigt, die unter anderem die Speicherverwaltung und die Kommunikation mit iOS übernimmt, und natürlich geeignete Klassenbibliotheken. Sicher keine leichte Aufgabe. Im Laufe der Jahre sind mehrere solcher Frameworks entstanden. Das vielleicht bekannteste ist RoboVM. Im Jahr 2015 wurde das als Open Source gestartete Produkt von der damals noch eigenständigen Firma Xamarin gekauft und nach der Übernahme durch Microsoft im Jahr 2016 eingestellt. Bis heute haben ein paar RoboVM-Forks überlebt.

Überblick

Die ursprünglich von Intel entwickelte Multi-OS Engine *[2]* ist ebenfalls Open Source. Sie steht seit 2016 unter der Apache License Version 2.0 zur Verfügung. Die Firma Migeran ist Project Lead und Core Developer. Sie bietet kommerziellen Support, kundenspezifische Entwicklung sowie Schulungen an. Multi-OS Engine basiert auf der aktuellen Android-Laufzeitumgebung ART (Android Runtime) und verwendet

Abbildung 1: Suche nach dem Multi-OS Engine-Plugin in IntelliJ

müssen Sie noch ein Plug-in herunterladen. Unter Android Studio und IntelliJ erreichen Sie den Plug-in-Manager zum Beispiel auf dem Willkommensbildschirm über „Configure/Plugins". Klicken Sie dann auf „Marketplace" und suchen Sie nach „Multi-OS Engine". Ein Klick auf „Install" *(siehe Abbildung 1)* startet den Installationsvorgang.

In Eclipse öffnen Sie mit „Help/Install New Software" den Installationsassistenten, geben bei „Work With" die Adresse *„https://dl.bintray.com/multi-os-engine/eclipse"* ein und drücken dann die Enter-Taste. Setzen Sie nun ein Häkchen vor „Multi-OS Engine", klicken auf „Next" und folgen dann den weiteren Installationsanweisungen.

Nach der Installation des Plug-ins und dem Neustart der IDE können Sie mit den jeweiligen Assistenten Multi-OS-Engine-Projekte anlegen. Wie das in IntelliJ aussieht, ist in den *Abbildungen 2, 3 und 4* zu sehen. Für einen ersten Test bietet sich die Vorlage „Single View Application" an. Auf der zweiten und dritten Seite des Wizard konfigurieren Sie das Projekt, indem Sie beispielsweise den Namen der App und des Projekts, das Java-Basispaket sowie den Speicherort angeben. Nachdem der Assistent beendet wurde, lädt die IDE fehlende Abhängigkeiten und baut die App. Sie können diese nun starten. In IntelliJ geschieht dies am einfachsten durch Anklicken des grünen Play-Buttons in der Toolbar.

dieselben Java-Bibliotheken wie Android. Die Java-Objective-C-Brücke „Nat/J" erlaubt den Zugriff auf beliebigen Objective-C-Code. Auf diese Weise werden native Benutzeroberflächen aus Java heraus angesprochen. Dazu später mehr. Die Entwicklung erfolgt unter macOS oder Windows. In diesem Fall muss ein macOS-Build-Rechner zur Verfügung stehen. Wie dieser eingerichtet wird, kann in der Multi-OS-Engine-Dokumentation nachgelesen werden *[3]*. Der Einsatz unter Linux ist derzeit nicht möglich. Auf Wunsch kann Multi-OS Engine unter macOS mit brew und repo aus dem Source Code gebaut werden *[4]*. Das ist allerdings nicht nötig. Denn als IDE kommen Android Studio, IntelliJ IDEA oder Eclipse ab Version 4.5 zum Einsatz. Die Installation des Frameworks erfolgt bequem mit den Plug-in-Managern der jeweiligen Entwicklungsumgebung.

Sicher ist Ihnen aufgefallen, dass im vorherigen Absatz öfter das Wort Android gefallen ist. Wieso kommt Googles mobile Plattform ins Spiel, wenn es doch eigentlich um native iOS-Anwendungen geht? Android-Apps wurden von Anfang an in Java geschrieben. Deshalb ist Android Studio (wie auch IntelliJ und Eclipse) optimal auf Java eingestellt. Die Laufzeitumgebung ART wurde für die bestmögliche Ausführung von Java-Code auf mobilen Geräten entwickelt, einschließlich Ahead-of-Time-Compiler und hochoptimierter Speicherverwaltung. Auch die Android-Java-Klassenbibliotheken haben sich in Millionen von Apps bewährt. Multi-OS Engine fußt also auf einem grundsoliden Fundament und kann sich auf iOS-spezifische Anpassungen und Erweiterungen konzentrieren, zum Beispiel auf das Zurverfügungstellen von iOS-API-Bindings. Für die Erstellung der Benutzeroberfläche, das Bauen sowie die Signierung und Paketierung der fertigen App wird Apples Xcode mit den dazugehörenden Kommandozeilentools verwendet. Aus diesem Grund ist bei der Entwicklung unter Windows der bereits angesprochene zusätzliche macOS-Rechner erforderlich.

Installation und Projekt-Setup

Nachdem Sie Xcode und eine der von Multi-OS Engine unterstützten Entwicklungsumgebungen installiert haben,

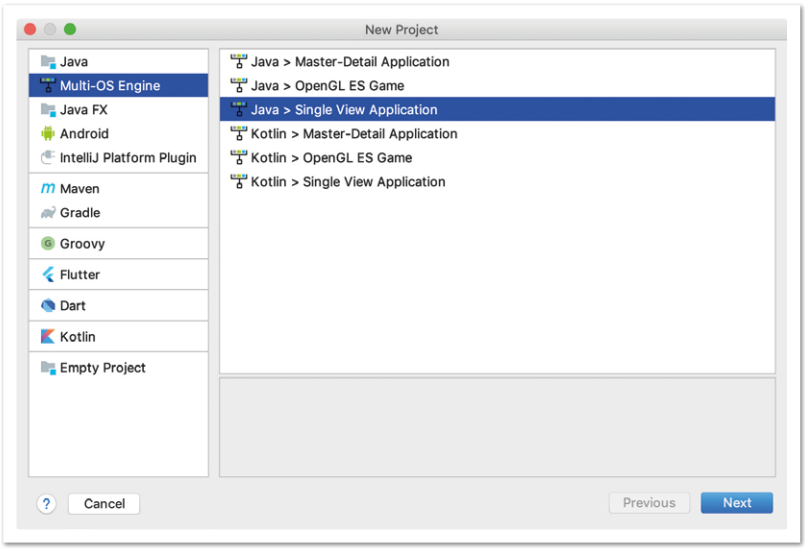


Abbildung 2: Mehrere Projektarten stehen zur Auswahl

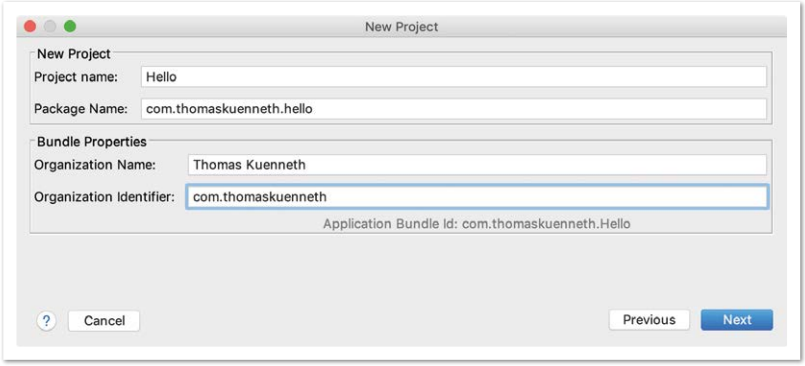


Abbildung 3: App-Informationen erfassen

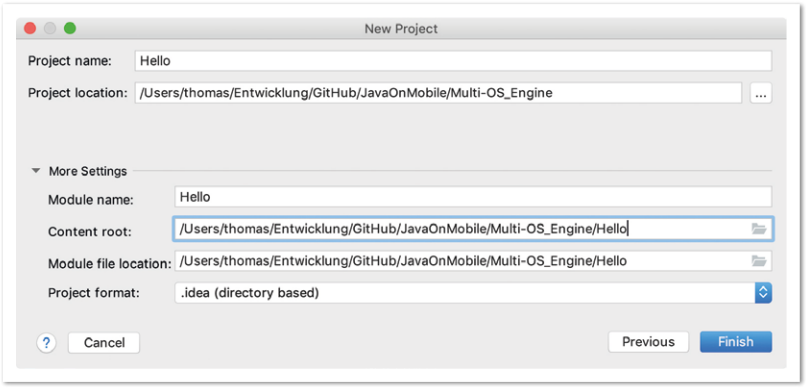


Abbildung 4: Projektname und Speicherort eintragen

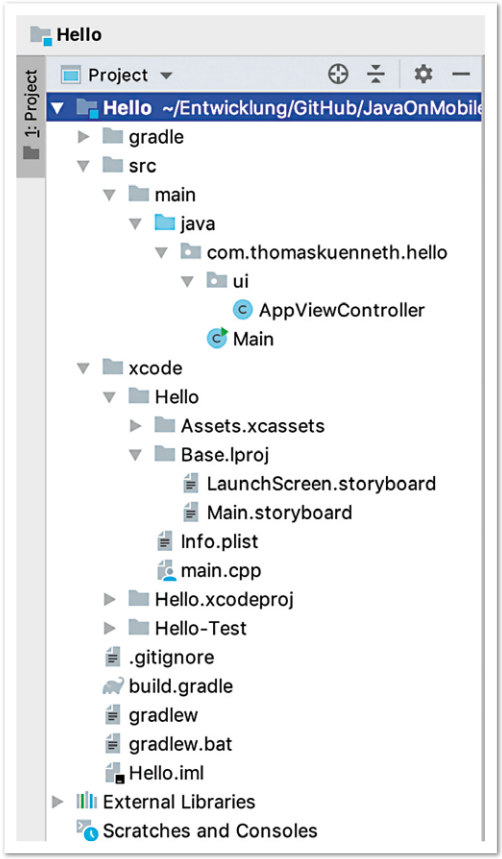


Abbildung 5: Die Projektstruktur

Bitte werfen Sie einen Blick auf die Projektstruktur in *Abbildung 5*. Unterhalb von „src/main/java" befinden sich die beiden Klassen AppViewController und Main. Letztere bildet den Einstiegspunkt in die App und muss im Normalfall nicht verändert werden. AppViewController wird nach dem Hinzufügen von UI-Controls nicht mehr benötigt. Bitte löschen Sie diese dennoch nicht. Sie kann gegebenenfalls als Vorlage für eigene View Controller dienen – dazu gleich mehr. Unterhalb von Xcode befinden sich Dateien, die von Apples IDE verwendet werden. Das ist nötig, weil Multi-OS Engine auf native Werkzeuge zum Bauen der App setzt. Außerdem wird die Benutzeroberfläche in Xcode erstellt. Um das Projekt in Xcode zu öffnen, klicken Sie mit der rechten Maustaste auf „xcode" und wählen dann „Multi-OS Engine Actions/Open Project in Xcode". Falls die Fehlermeldung „Neither the Xcode project nor the workspace is set in the Gradle plugin" erscheint, öffnen Sie das Projekt einfach im Finder. Klicken Sie hierzu mit der rechten Maustaste auf „xcode/Hello" und wählen dann „Reveal in Finder". Ein Doppelklick auf „Hello.xcodeproj" öffnet das Projekt.

Eine Benutzeroberfläche erstellen

In der aktuellen Version enthält die Projektvorlage „Single View Application" keine Bedienelemente. Wir werden im Folgenden eine einfache Benutzeroberfläche erstellen. Bitte legen Sie zunächst mit „File/New/File" eine neue „Cocoa Touch Class" an. Geben Sie dieser den Namen MyViewController. Sie leitet von UIViewController ab und muss in Objective-C implementiert werden. Vor „Also create XIB file" darf kein Häkchen gesetzt sein. Achten Sie darauf, dass die Datei im selben Verzeichnis wie „main.cpp" abgelegt wird und dem Target „Hello" zugeordnet ist. Beides sollte standardmäßig eingestellt sein. Xcode wird nun die beiden Files „MyViewController.h" und „MyViewController.m" erstellen. Als Nächstes müssen Sie den neu erstellten View Controller dem sogenannten „Storyboard" zuordnen. Klicken Sie hierzu als Erstes auf die Datei „Main.storyboard". Im Anschluss daran markieren Sie den Knoten „App View Controller Scene". Wie in *Abbildung 6* zu sehen, müssen Sie im Identity Inspector unter „Custom Class/Class" den neu angelegten MyViewController eintragen. Der Name des Knotens ändert sich zu „My View Controller Scene".

Anschließend müssen Sie aus der Komponentenbibliothek, die mit „View/Libraries/Show Library" geöffnet wird, einen Button sowie ein Label auf das Storyboard ziehen. Danach haben Sie es fast geschafft. Nun definieren Sie für den Button eine sogenannte „Action" und für das Label ein „Outlet". Solche Verbindungen ermöglichen den Zugriff auf die UI-Elemente in Ihrem Code. Klicken Sie auf „View/Assistant Editor/Show Assistant Editor". Neben dem Storyboard sehen Sie nun einen Texteditor. Er enthält in seinem oberen Bereich eine Brotkrümelnavigation. Mit dieser wechseln Sie zur Datei „MyViewController.h". Um ein Outlet oder eine Action zu erstellen, halten Sie die Steuerungstaste gedrückt und ziehen dann das entsprechende UI-Element auf den Assistant Editor *(siehe Abbildung 7)*. Es erscheint ein Pop-up-Menü, mit dem Sie die Verbindung konfigurieren. Legen Sie auf diese Weise eine Action für den Button und ein Outlet für das Label an *(siehe Abbildung 8 und 9)*.

Logik in Java

Bevor Sie Xcode beenden, können Sie dem Button einen aussagekräftigeren Namen geben. Damit die Benutzeroberfläche auf allen möglichen Geräten richtig dargestellt wird, müssen unter Umständen sogenannte „Layout Constraints" angepasst oder hinzugefügt werden. Dies ist ein umfangreicheres Thema und wird im Folgenden nicht weiter dargestellt. Alternativ ist es für dieses Beispiel ausreichend, das Label etwas größer zu ziehen, damit der anzuzeigende Text vollständig dargestellt wird. Speichern Sie Ihre Änderungen und verlassen dann Xcode. Um auf die eben angelegten Verbindungen zuzugreifen, legen Sie die Klasse MyViewController *(Listing 1)* an.

MyViewController referenziert eine Reihe von iOS-spezifischen Klassen, unter anderem UIViewController, UIButton und UILabel. Um gute Benutzeroberflächen erstellen zu können, müssen Sie sich mit deren Verwendung vertraut machen (das gilt auch für die Verwendung von Xcode und des Storyboard-Designers). Zum Beispiel wird die mit `@Override` versehene Methode viewDidLoad()aufgerufen, wenn eine View geladen wurde. Sie eignet sich deshalb prima für bestimmte Initialisierungsaufgaben. Im Vergleich dazu sind die Multi-OS-Engine-spezifischen Ergänzungen schnell erlernbar. Mit der Annotation `@Selector` kennzeichnen Sie Elemente aus nativen Klassen, zum

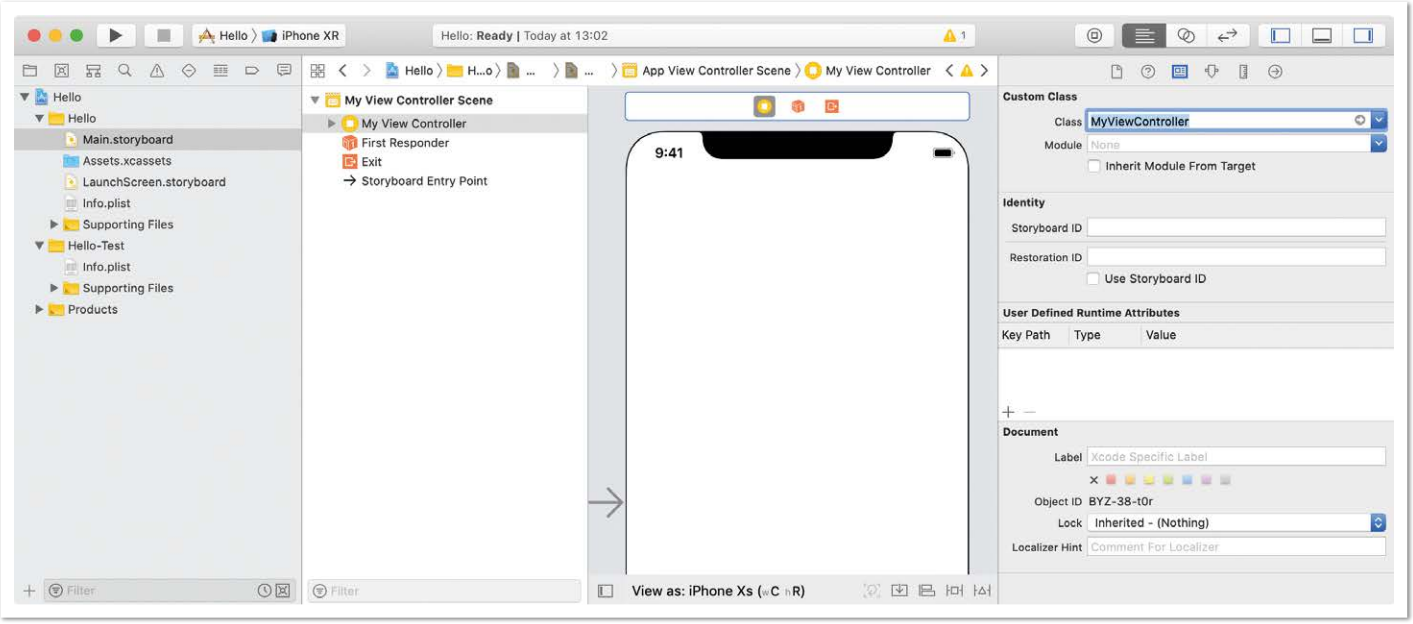


Abbildung 6: Einen neuen View Controller zuordnen

Beispiel die Action der Schaltfläche und das Outlet des Labels. In jedem Fall lohnt ein Blick auf die Dokumentation und das Forum *[5]*. Dort kann unter anderem die Erstellung eigener Bindings nachgelesen werden.

Zusammenfassung

Die Integration echter nativer Benutzeroberflächen ist zweifellos ein großer Pluspunkt. Auf der anderen Seite ist gerade der Wunsch, kein oder nur wenig natives Wissen aufbauen zu müssen, ein Grund für die Suche nach Alternativen. Insofern wirbt Multi-OS Engine auch mit dem Argument, vorhandene Android-Apps als Basis zu nehmen. Die Idee ist, die beiden nativen Teile sowie die gemeinsame Geschäftslogik in separate Module zu packen. Wie das funktioniert, ist in der Dokumentation beschrieben. Eine solche Aufteilung findet sich übrigens in den meisten aktuellen Cross-Platform-Frameworks wieder. Auch der Zugriff auf die jeweiligen nativen Build Tools und die Notwendigkeit mindestens eines Macs sind bewährte Praxis.

Bleibt also die Bewertung der Multi-OS Engine selbst. Die Integration in weitverbreitete IDEs macht die Entwicklung angenehm und komfortabel. Dokumentation und Beispiele sind anschaulich und helfen bei der Einar-

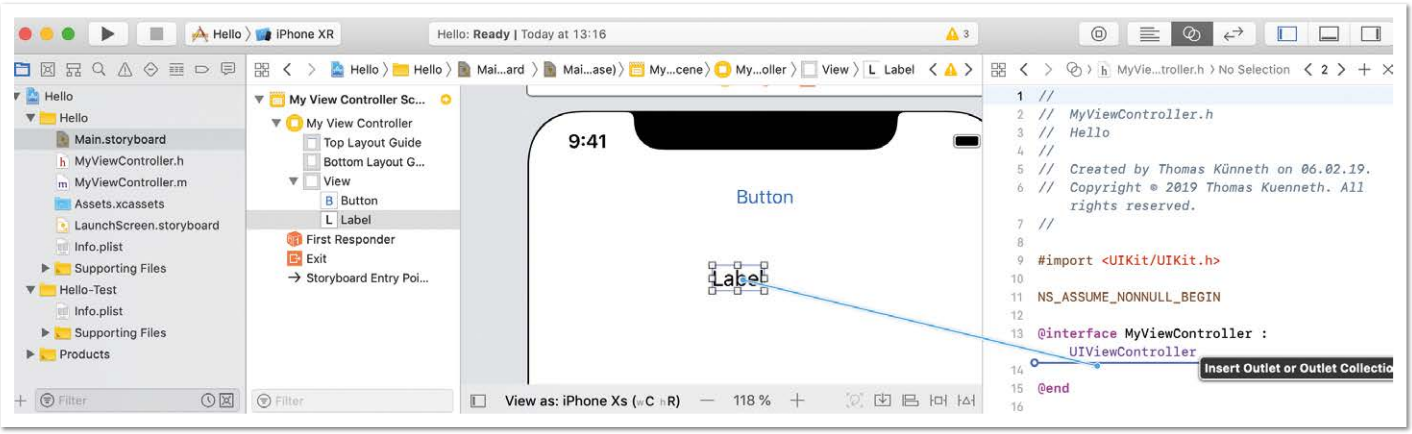


Abbildung 7: Eine Verbindung anlegen

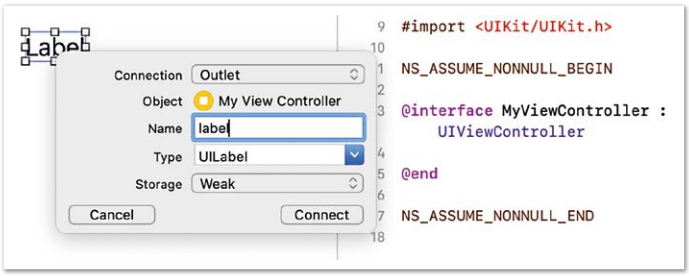


Abbildung 8: Ein Outlet konfigurieren

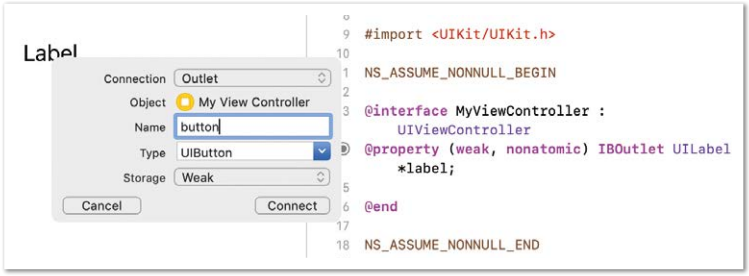


Abbildung 9: Eine Action konfigurieren

```
package com.thomaskuenneth.ui;

import apple.uikit.UIButton;
import apple.uikit.UILabel;
import apple.uikit.UIViewController;
import org.moe.natj.general.Pointer;
import org.moe.natj.general.ann.Owned;
import org.moe.natj.general.ann.RegisterOnStartup;
import org.moe.natj.objc.ObjCRuntime;
import org.moe.natj.objc.ann.IBOutlet;
import org.moe.natj.objc.ann.ObjCClassName;
import org.moe.natj.objc.ann.Property;
import org.moe.natj.objc.ann.Selector;
import java.util.Locale;

@org.moe.natj.general.ann.Runtime(ObjCRuntime.class)
@ObjCClassName("MyViewController")
@RegisterOnStartup
public class MyViewController extends UIViewController {
  public UILabel label = null;
  private int count;

  @Override
  public void viewDidLoad() {
     label = getLabel();
     label.setText("Noch nicht geklickt");
     count = 0;
  }

  @Selector("label")
  @Property
  @IBOutlet
  public native UILabel getLabel();

  @Owned
  @Selector("alloc")
  public static native MyViewController alloc();

  @Selector("init")
  public native MyViewController init();

  protected MyViewController(Pointer peer) {
     super(peer);
  }

  @Selector("button:")
  public void buttonClicked(UIButton sender) {
     label.setText(String.format(Locale.US,
        "%d mal geklickt",
        ++count));
  }
}
```

Listing 1: Die Klasse „MyViewController"

beitung. Sie sind aber leider nicht in allen Punkten aktuell. Gegebenenfalls ist deshalb Forum-Recherche erforderlich. Die kontinuierliche Weiterentwicklung von Android und iOS macht regelmäßige Aktualisierungen von Cross-Platform-Frameworks erforderlich. Denken Sie an neue APIs. Außerdem „schrauben" Apple und Google gerne an den Tool-Schnittstellen. Wenn sich hier inkompatible Änderungen ergeben, bricht der Bauvorgang ab. Probleme kann es auch bei der IDE-Integration geben. Im schlechtesten Fall funktionieren Plug-ins nicht mit aktuellen Versionen einer Entwicklungsumgebung. Hier hilft leider nur ausprobieren.

Insbesondere in der zweiten Jahreshälfte 2018 war es recht still um Multi-OS Engine. Daher bleibt abzuwarten, ob und wann die nächste Version erscheint. Da das Framework quelloffen ist, kann man bei Bedarf einen Blick hinter die Kulissen werfen und zumindest theoretisch selbst Erweiterungen vornehmen oder Fehler beheben.

Links

[1] *https://github.com/tkuenneth/JavaOnMobile/tree/master/Multi-OS_Engine/Hello*

Technologieauswahl

Dieser Artikel geht davon aus, dass eine App ausschließlich unter iOS betrieben wird oder eine bereits vorhandene native Android-Anwendung auf iPhones oder iPads portiert werden soll. Bei der Neuentwicklung einer mobilen App für beide Ökosysteme können „klassische" Cross-Platform-Frameworks eine interessante Alternative sein. Sie versprechen die Entwicklung von in Stores hochladbaren Apps mit einheitlicher Code-Basis. Erreicht wird dies durch eine gemeinsame Programmiersprache, Klassenbibliothek und Werkzeuge sowie eine abstrahierte Sicht auf die Benutzeroberfläche. Wie nativ eine solche App ist, hängt von der Funktionsweise des Frameworks ab. Wird Quellcode transpiliert oder zur Laufzeit interpretiert? Entstehen aus der generalisierten Oberflächenbeschreibung native UI-Elemente oder werden die Widgets selbst gezeichnet? Wie vollständig ist der Zugriff auf native Ressourcen? Wieviel Overhead bringt das Framework mit sich, das heißt, um wie viel größer werden Apps? Eine Technologie, die „weniger native" Apps erzeugt, ist nicht zwangsläufig schlechter geeignet. Denn die Nähe zum Original steht in einem Spannungsverhältnis zu Kosten und vorhandenem Know-how. Hat ein Unternehmen noch keine Software in Swift entwickelt, muss es die Implementierung entweder extern vergeben, Wissen einkaufen oder aufbauen. Ob Letzteres lohnt, hängt unter anderem davon ab, wie viele weitere Apps entstehen sollen. Das gilt natürlich auch für die Programmiersprache eines Cross-Platform-Frameworks. In diesem Bereich dominieren derzeit JavaScript/TypeScript und C#. Ist entsprechendes Wissen nicht vorhanden, muss es eingekauft oder aufgebaut werden. Eine Ausnahme bildet Gluon Mobile. Dieses Framework setzt auf Java und JavaFX.

[2] *https://multi-os-engine.org/*
[3] *https://doc.multi-os-engine.org/multi-os-engine/3_getting_started/3_remote_build/remote_build.html*
[4] *https://github.com/multi-os-engine/multi-os-engine*
[5] *https://discuss.multi-os-engine.org/*

Thomas Künneth
thomas.kuenneth@mathema.de

Thomas Künneth ist Principal Consultant und Head of Mobile bei der MATHEMA Software GmbH. Seine Schwerpunkte sind die Architektur großer Unternehmensanwendungen, Native und Cross-Platform-Apps sowie das Mobile Enterprise. Seit annähernd 20 Jahren beschäftigt sich Thomas intensiv mit Java. Außerdem ist er Android-Entwickler der ersten Stunde und Autor zahlreicher Artikel und Bücher zu Java, Eclipse, Android und Mobile Computing. Thomas spricht regelmäßig auf nationalen und internationalen Konferenzen.

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:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: © Elena Akifeva www.123rf.com
S. 9: Bild © Le Moal Olivier www.123rf.com
S. 15: Bild © Václav Mach www.123rf.com
S. 16: Buch © Carl Hanser Verlag www.hanser.de
S. 18: Bild 1 © iJUG www.ijug.eu
S. 18: Bild 2 © java user group stuttgart
S. 19: Bild 4-6 © Schuchrat Kurbanov
S. 20: Bild © Mike Meier
S. 22: Logo © helidon https://helidon.io
S. 27: Bild © alphaspirit www.123rf.com
S. 32: Bild © scanrail https://123RF.com
S. 36: Logo © BootsFaces www.bootsfaces.net
S. 47 Bild © Dusit Panyakhom https://123RF.com
S. 53 Bild © rawpixel https://123RF.com
S. 61 Bild © Galina Peshkova https://123RF.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

JUG SAXONY DAY

13.09.

PROGRAMM (AB MAI 2019)
jug-saxony-day.org

ORT
Radisson Blu Park Hotel & Conference Centre
Radebeul b. Dresden

TICKETBUCHUNG
Fast Lane bis 15. Mai 2019
Early Bird bis 15. Juli 2019
Freitickets für Studierende
(Solange der Vorrat reicht.)

backoffice.jugsaxony.org

Eine Veranstaltung des JUG Saxony e. V.

Kontakt
team@jugsaxony.org
jugsaxony.org

Folge uns auf

adesso business. people. technology.

SOFTWARE DEVELOPMENT@adesso

Von Anfang an Teil des Java-Teams!

Entwickelt bereits kluge IT-Lösungen bei adesso:
Ihr neuer Kollege Kristof Zalecki | Software Engineer

Sie wollen dort einsteigen, wo Zukunft programmiert wird? Dann sind Sie mit einem Start in unserem Java-Team bei adesso genau richtig. Gemeinsam setzen wir herausfordernde Projekte für unsere Kunden um. Dafür brauchen wir Menschen, die Lust haben, ihr Wissen, ihre Talente und ihre Fähigkeiten einzubringen.

Planen und realisieren Sie in interdisziplinären Projektteams anspruchsvolle Anwendungen und Unternehmensportale auf Basis von Java/JavaScript-basierten Technologien als

→ (Senior) Software Engineer (m/w) Java
→ Software Architekt (m/w) Java
→ (Technischer) Projektleiter Softwareentwicklung (m/w) Java

CHANCENGEBER – WAS ADESSO AUSMACHT

Kontinuierlicher Austausch, Teamgeist und ein respektvoller, anerkennender Umgang sorgen für ein Arbeitsklima, das verbindet. So belegen wir nach 2016 auch 2018 den 1. Platz beim Wettbewerb „Deutschlands Beste Arbeitgeber in der ITK"!

Mehr als 650 Software Engineers Java bei adesso, über 120 Schulungen und Weiterbildungen – zum Beispiel in Angular2 oder Spring Boot – sowie ein Laptop und ein Smartphone ab dem ersten Tag warten auf Sie!

GREAT PLACE TO WORK® 2018 Beste Arbeitgeber™ ITK

IHRE BENEFITS – WIR HABEN EINE MENGE ZU BIETEN:

- Welcome Days
- Choose your own Device
- Weiterbildung
- Events: fachlich und mit Spaß
- Sportförderung
- Mitarbeiterprämien
- Auszeitprogramm

Es wird Ihnen bei uns gefallen! Mehr Informationen auf **www.karriere.adesso.de**.
Olivia Slotta aus dem Recruiting-Team freut sich auf Ihre Kontaktaufnahme:
adesso AG // Olivia Slotta // T +49 231 7000-7100 // jobs@adesso.de