Ausgabe 05/2017

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 05

Java aktuell

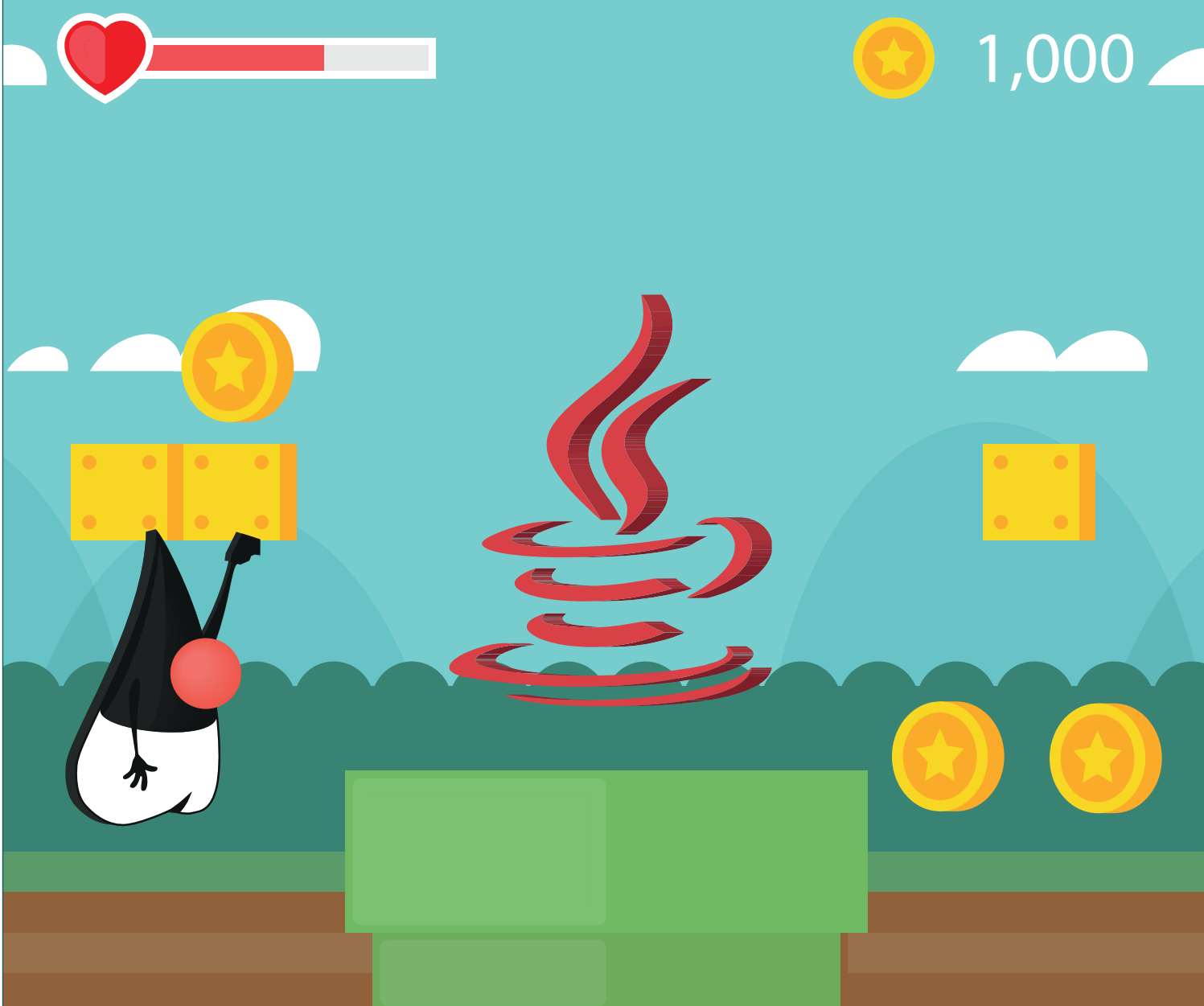
iJUG Verbund
www.ijug.eu

Tipps und Tricks
Projekte erfolgreich nach Java 9 migrieren

Modernisierung
Von Java Swing über JavaFX nach RISC-HTML

Aus der Praxis
Multicore-Programmierung mit Java

Next Level: Java 9



JVM Con

Die Konferenz für Java-Entwickler

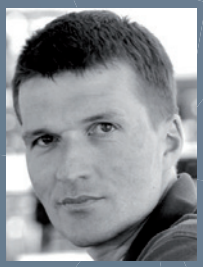
28.-29. November 2017 | Köln, Pullman Cologne Hotel

Bei Ticketbuchung mit dem Code jvm17ja sparen Sie 100,- €

Themenauswahl (u.a.):

- Java 9 - Features abseits von Jigswa und Jshel
- Wie Sie ihre Architektur am Leben erhalten
- Konfiguration von Java-Applikationen
- Reaktive Programmierung in Java
- Über (In-)Konsistenz in verteilten Systemen
- Pragmatischer Einstieg in Clojure
- JavaFX mit MVVM Pattern, Usability und Gestensteuerung
- Kollektion von Microservice Patterns
- REST-API für MongoDB mit Spring Boot & Spring Data

Referenten (u.a.):

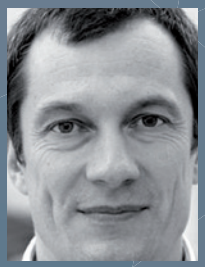


Adam Bien
adam-bien.com

David Delabassée
Oracle Corporation

Ivar Grimstad
Cybercom Group

Bernd Rücker
camunda services GmbH



Jens Schumann
open knowledge GmbH

Anmeldung und weitere Infos unter www.jvm-con.de #jvmcon17

Präsentiert von: web & mobile DEVELOPER

Veranstalter: Neue Mediengesellschaft Ulm mbH / developer media

Die Java Community rockt

Unabhängig davon, wie Oracle mit Java umgeht, ist die Java Community blendend unterwegs.

JavaLand 2017: Die dreitägige Java-Konferenz im Phantasialand verdoppelt innerhalb von vier Jahren die Teilnehmerzahl. Die Veranstaltung, in Kooperation mit Heise Medien und dem Interessenverbund der Java User Groups e.V. (iJUG) organisiert, feierte im Jahr 2014 mit 800 Java-Begeisterten ihren Auftakt.

Java Forum Stuttgart: Zu der eintägigen Konferenz mit Java als Leitmotiv kommen mehr als 1.600 Teilnehmer zusammen. Sie findet in diesem Jahr zum zwanzigsten Mal statt.

Java Forum Nord: Die nicht-kommerzielle Konferenz in Norddeutschland mit dem Themenschwerpunkt „Java für Entwickler und Entscheider" geht in die dritte Auflage. Sie wird von sieben Java User Groups gemeinsam organisiert.

JUG Saxony Day: Die Veranstaltung in der Nähe von Dresden ist restlos ausverkauft. Das Ziel von 450 Teilnehmern ist deutlich übertroffen.

Berlin Expert Days 2017: Hier dreht sich wieder alles rund um die Themen „Java", „Java EE", „Spring", „JavaScript", „Funktionale Sprachen", „DevOps" und „Stateless". Zahlreiche Experten geben in Talks ihr Wissen weiter.

Apropos Oracle: Im Rahmen der letzten iJUG-Mitgliederversammlung gibt es lange Diskussionen über die Zusammenarbeit mit Oracle. Viele User Groups befürchten einen Rückzug von Oracle aus den Community-Aktivitäten, da der Hersteller auf dem Java Forum Stuttgart sowie auf der diesjährigen JavaLand nicht mit einen Stand vertreten war und einige Oracle-Vertreter auch keine Vorträge zur JavaLand 2018 einreichen durften. Die User Groups haben auch wenig Verständnis dafür, dass Oracle auf der anderen Seite neue kommerzielle Veranstaltungen wie im Herbst in Köln als Hauptsponsor nachhaltig unterstützt. Ich bin gespannt, wie es hier weitergeht.

In eigener Sache: Alle Abonnenten der Zeitschrift ix erhalten einen Auszug aus dieser Ausgabe kostenlos beigelegt. Auf diesem Weg machen wir die Arbeit der Java Community noch bekannter und tragen so zu deren Erfolg bei. Noch eine erfreuliche Sache: Ab dem Jahr 2018 erscheint die Java aktuell mit sechs Ausgaben pro Jahr.

Ich wünsche Ihnen viel Spaß beim Lesen, viel Erfolg bei Ihren Projekten und freue mich wie immer auf Ihr Feedback an redaktion@ijug.eu.

Ihr

Wolfgang Taschner
Chefredakteur Java aktuell

Inhalt

Es besteht kein Grund, wegen der bevorstehenden Migration auf Java 9 in Panik zu verfallen

Mit JShell existiert jetzt endlich die lang erwartete Read Eval Print Loop (REPL) in Java 9

33

In der rein funktionalen Programmierung sind Daten unveränderlich – also kein gekapselter Zustand

54

Für die meisten Unternehmen führt aktuell kein Weg am Thema „Cloud" vorbei

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im dritten Quartal 2017.

4. Juli 2017

Java EE kurz vor dem Ziel

Linda DeMichiel, als Spec Lead des Umbrella-JSR quasi die Hüterin des Java-EE-Standards, hat den Proposed Final Draft der EE-8-Spezifikation auf GitHub hochgeladen *(siehe „https://github.com/javaee/javaee-spec/“).* In der offiziellen Java-EE-Gruppe schreibt sie dazu, dass die letzten Änderungen minimal sind und es im Wesentlichen nur noch um die Synchronisierung mit dem Security-JSR geht. Alexander Neumann drückt es auf Heise so aus: „EE 8 befindet sich auf der Zielgeraden“ – vorbehaltlich der finalen Zustimmung durch das Executive Committee. Das erscheint aber im Gegensatz zum kurzzeitigen Schock bei Java SE 9 vor Kurzem im Fall von EE 8 als sicher.
https://javaee.groups.io/g/javaee-spec/topic/5433704

10. Juli 2017

EE-8-Fortschritt im Juni

Die monatliche Zusammenfassung der EE-8-Aktivitäten von David Delabassee zeigt den Fortschritt auch klar. Alle Teil-JSRs, die mehr als reine Maintenance-Releases herausbringen, sind in den letzten Phasen des Prozesses („Public Review“ oder darüber hinaus) oder schon „final“. Als Konsequenz nimmt die Arbeit an den Spezifikationen stark ab, an der Referenz-Implementierung GlassFish 5 hingegen stark zu. Für alle, die immer den neuesten Fish im Container haben wollen, könnte diese Meldung zur Umstrukturierung der Docker-Images interessant sein, die jetzt unter der Organisation „oracle“ zu finden ist.
https://blogs.oracle.com/theaquarium/glassfish-docker-images-–-update

13. Juli 2017

Java-9-Features

In schneller Folge werden auf „blogs.oracle.com/java“ Zusammenfassungen der neuen Features sowie Links zu detaillierten Beschreibungen und Erklär-Videos wie von der Devoxx gepostet. Vor Infos zu SE 9 kann man sich ja sowieso kaum retten. Ein ganz interessanter Eintrag ist für mich aber dieser, in dem die 9er-Features nicht nur im Überblick gezeigt, sondern für eine schnelle Einordnung auch kategorisiert sind („behind the scenes“, „new functionlity“, „new standards“, „housekeeping“ etc.):
https://blogs.oracle.com/java/jdk-9-categories

1. August 2017

EE-8-Fortschritt im Juli

EE schreitet weiter voran: Bis auf Security 1.0 und EE 8 selber ist von allen Spezifikationen der „Proposed Final Draft“ vom Executive Committee akzeptiert worden. Auch die drei anderen sind zumindest in der finalen Review-Phase und sollten bald ebenfalls durch sein. Für GlassFish 5 wird inzwischen soweit möglich jeden Dienstag und Freitag ein „promoted build“ veröffentlicht.
https://blogs.oracle.com/theaquarium/java-ee-8-july-recap

8. August 2017

MicroProfile 1.1 ist freigegeben

Das neue MicroProfile-Minor-Release bringt als einzige größere Änderung ein Configuration-API. Da dieses das JDK 8 voraussetzt, ist auch MicroProfile 1.1 darauf festgelegt. Eine eigene Referenz-Implementierung hat „MicroProfile Config“ nicht, das GitHub-Projekt *(siehe „https://github.com/eclipse/microprofile-config“)* enthält nur die Doku, die API-Spezifikation und das Test-Compatibility-Kit; stattdessen wird auf Implementierungen der Partner Apache, IBM und Payara verwiesen. Das Configuration-API war ja schon mehrfach auch für Java EE avisiert, wurde aber nie als JSR angenommen. Es wäre interessant zu sehen, ob die „pre-standard collaboration“ (O-Ton David Blevins von Tomitribe) namens MicroProfile hier Früchte trägt und das Configuration-API demnächst auch in EE Einzug hält. In Folge-Releases ist die Aufnahme von Health Check, Security, Metrics und Fault-Tolerance-APIs geplant – also auch hier über aktuelle Java-EE-JSRs hinausgehend. Die APIs sind zumindest teilweise auch schon in Arbeit. Klingt gut – die Frage ist nur, wo die Grenze ist beziehungsweise ob irgendwann MicroProfile nur noch so heißt.
https://microprofile.io

20. August 2017

Java EE soll weiter geöffnet werden

Oracle überlegt, die Java Enterprise Edition (Java EE) an eine Open-Source-Organisation zu übergeben. Der Schritt soll nach der Bereitstellung von Java EE 8 erfolgen. EE-Evangelist David Delabassee schreibt in einem Blog-Eintrag, dass Oracle damit unter anderem agilere Prozesse, eine flexiblere Lizenzierung und eine breitere Beteiligung mit mehr Innovationskraft erreichen möchte. Oracle will die derzeitigen Überlegungen nun mit der Java-EE-Community, Lizenznehmern und möglichen Open-Source-Organisationen weiterführen. Die Übergabe der Java-EE-Technologien an eine Open-Source-Foundation könne der richtige nächste Schritt sein. Die Eclipse-Foundation hat bereits geäußert, Java EE gerne aufnehmen zu wollen. Ausdrücklich erwähnt Delabassee auch eine Übergabe des Test-Compatibility-Kit (TCK) – das wäre ein großer Schritt für das Java-Ökosystem (wenn auch wohl nur ein kleiner für die Menschheit). Natürlich bleibt neben der Freude auch ein bisschen Skepsis – schließlich macht ein Konzern wie Oracle nichts aus reiner Liebe zur Community; es dürften auch handfeste finanzielle Interessen dahinterstehen, etwa ein Abziehen

von Mitarbeitern aus der EE-Spezifikation und -Implementierung zugunsten der kommerziellen „Added value"-Produkte von Oracle. Dass Oracle sich ganz massiv aus der Basis zurückzieht, ist eigentlich nicht zu erwarten, sodass hoffentlich die positiven Aspekte durch mehr Engagement anderer großer Partner und der Community insgesamt überwiegen. WebLogic-Kunden werden übrigens in einem zeitgleichen Post dahingehend beruhigt, dass der geplante Strategiewechsel keine unmittelbaren Auswirkungen auf das kommerzielle Produkt hat.

https://blogs.oracle.com/theaquarium/opening-up-java-ee

21. August 2017

Ceylon jetzt bei Eclipse

Auch die maßgeblich von Red Hat getriebene und für die JVM, JavaScript-Engines und native Kompilierung gedachte Sprache Ceylon soll jetzt bei der Eclipse-Foundation unterschlüpfen. Bei Red Hat erhofft man sich dadurch ebenfalls mehr Beteiligung aus der Community. Vorher wurde noch ein letztes Update zu Release 1.3 veröffentlicht.

https://projects.eclipse.org/proposals/eclipse-ceylon

22. August 2017

EE 8 ist durch

Weitgehend kommentarlos und einstimmig – lediglich Intel hat sich enthalten – ist der Public-Final-Draft von Java EE 8 angenommen worden. Einem baldigen Release steht damit nichts mehr im Wege, da auch Security 1.0 als letzter Teil-JSR mit demselben Ergebnis durchgekommen ist – vielleicht ist die Erklärung, dass der Intel-Repräsentant im Executive Committee einfach ohne Vertretung in den Urlaub gegangen ist.

https://github.com/javaee/javaee-spec/tree/master/download

25. August 2017

OpenJDK erhält eigenes Team für Sicherheitslücken

Um eine koordinierte Behebung von Sicherheitslücken im OpenJDK zu fördern, schlägt Chief Platform Architect Mark Reinhold die Gründung einer „OpenJDK Vulnerability Group" vor. Mitglieder müssen neben einer Reihe von vertraglichen Bedingungen logischerweise eine gewisse Historie in der professionellen, vertrauensvollen Behandlung von Sicherheitslücken mitbringen und sind zur Verschwiegenheit bezüglich nicht veröffentlichter Schwachstellen verpflichtet. Als Kommunikationskanäle sollen drei E-Mail-Listen dienen: Eine zum Melden von Sicherheitslücken, eine rein interne für die Mitglieder des Teams und eine für öffentliche Meldungen durch das Team. Die Regeln seien nicht ganz kompatibel mit den OpenJDK Bylaws, aber das Governing Board des OpenJDK habe dies bereits diskutiert. Mark Reinhold geht daher davon aus, dass seinem Vorschlag zugestimmt wird. Der erste Leiter des Teams soll Andrew Gross sein, der bereits das Oracle-interne Java-Vulnerability-Team leitet. Auch später sollen immer Oracle-Mitarbeiter diese Rolle besetzen.

http://cr.openjdk.java.net/~mr/ojvg

26. August 2017

Wahlen zum JCP Executive Committee

Die nächsten Wahlen zum Executive Committee stehen an. Die Hälfte der Sitze aus jeder Kategorie wird neu vergeben – insgesamt 13 (die Nummer 25 – oder vermutlich Nummer 1 aus Oracle-Sicht – wird ja nicht gewählt). Mal sehen, welche Rolle das EC in Zukunft spielen wird, wenn die Oracle-Pläne für eine weitere Öffnung von Java EE konkret werden – und ob das schon Einfluss auf die Bewerbungen haben wird. Aber es gibt ja auch noch mehr als Java EE.

7. September 2017

Kürzere Release-Zyklen für Java SE

Oracle beziehungsweise das OpenJDK-Projekt wollen in Zukunft deutlich kürzere Release-Zyklen haben, nämlich zwei JDK-Releases pro Jahr: eines im März, eines im September. Das Thema wurde auf dem letzten Treffen des JCP Executive Committee diskutiert. Oracle möchte offensichtlich schon diesen Monat den JSR für Java SE 10 starten, mit einem „Freeze" im Dezember. Klingt wieder mal sportlich – Java SE 9 ist ja noch nicht mal freigegeben. Wie lange ist es noch mal verspätet gegenüber den ursprünglichen Plänen? Ungefähr zwei Releases nach dem neuen Schema … Aber laut Mark Reinhold führen die bisher langen Zyklen des „Release Train" zu den großen Problemen. Nachvollziehbar, wenn man vor der Wahl steht, etwa ein groß angekündigtes Java-Modulsystem um mindestens zwei Jahre zu verschieben oder den kompletten Zug deswegen anzuhalten. „Neue Features werden nur gemergt, wenn sie so gut wie fertig sind, sodass das in Entwicklung befindliche Release zu jedem Zeitpunkt feature-complete' ist", schreibt Reinhold.

https://mreinhold.org/blog/forward-faster

Andreas Badelt

stellv. Leiter der DOAG Java Community

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich in der DOAG Deutsche ORACLE-Anwendergruppe e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit dem Jahr 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").

Java 9 veröffentlicht

Sebastian Höing, DOAG Online, und André Sept, Leiter der DOAG Java Community

Java 9 ist endlich da und das Java Development Kit (JDK) steht für Entwickler zum Download bereit. Java SE 9 wurde somit fast drei Jahre nach Java SE 8 eingeführt und bringt einige wichtige Änderungen in der Architektur sowie eine Vielzahl von Verbesserungen mit. Die neuen, umstrittenen Modularitäts-Fähigkeiten, die auf Project Jigsaw basieren, sind sicherlich eines der interessanteren Features.

Die Modularität von Java 9 soll es Entwicklern erleichtern, anspruchsvolle Anwendungen einfacher zusammenzustellen. Außerdem soll Java besser in der Lage sein, auf kleinere Geräte herunter zu skalieren, während Sicherheit und Leistung verbessert werden. Die Modularität von Java 9 umfasst unter anderem die Anwendungspaketierung, die Modularisierung des JDK selbst und die Reorganisation des Quellcodes. Eine Vielzahl von Konfigurationen wird unterstützt, sodass Skalierbarkeit, Sicherheit und Anwendungsperformance verbessert werden können. Eine einfachere Skalierung von Java bis auf kleine Geräte ist ein wesentlicher Treiber des modularen Aufwands. Durch die Modularität können Entwickler Bibliotheken und große Anwendungen sowohl für Java SE als auch für Java EE besser aufbauen und verwalten.

Verbesserungen am Compiler und am Stream-API

Das Java-9-Update bietet mehrere neue Funktionen zum Kompilieren von Code, darunter vor allem die Ahead-of-Time-Kompilierung (AoT). Noch in einer experimentellen Phase, ermöglicht diese Fähigkeit die Kompilierung von Java-Klassen in nativem Code, bevor sie in der virtuellen Maschine gestartet werden. Diese Funktion soll die Startzeit von kleinen und großen Anwendungen verbessern, mit begrenzten Auswirkungen auf die Spitzenleistung.

Java 9 bietet auch die zweite Phase des intelligenten Kompilierungs-Deployments von Oracle. Diese Funktion beinhaltet die Verbesserung der Stabilität und Portabilität des Javac-Tools, sodass es standardmäßig in der Java Virtual Machine (JVM) verwendet werden kann. JDK 9 hat auch den Javac-Compiler aktualisiert, damit er Java-9-Programme kompilieren kann, um auf einigen älteren Versionen von Java laufen zu können. Auch das Stream-API von Java 9 wurde verbessert; es stehen neue Methoden zur Verfügung, um Aufgaben einfacher zu integrieren.

Read-Eval-Print-Loop in Java 9 integriert

Java 9 verfügt über ein Read-Eval-Print-Loop-Tool (REPL), ein weiteres langfristiges Ziel für Java, das in dieser Version nach Jahren der Entwicklung unter dem Projekt Kulia Wirklichkeit wird. Die REPL von Java 9 mit dem Namen jShell wertet deklarative Anweisungen und Ausdrücke interaktiv aus. Entwickler können vor der Kompilierung ein Feedback zu Programmen erhalten, indem sie einfach einige Zeilen Code eingeben. Das Kommandozeilen-Tool kann Tabulatoren vervollständigen und automatisch benötigte Semikolons hinzufügen. Das jShell-API erlaubt jShell-Funktionalität in IDEs und anderen Tools, obwohl das Tool selbst keine IDE ist.

Kürzere Releasezyklen geplant

Neben diesen Neuerungen bringt JDK 9 viele weitere Änderungen mit. Die Modularisierung soll auch der Grundstein für eine zukünftige Entwicklung sein. Wie Oracle auf seiner Java-Roadmap-Webseite *(siehe „http://www.oracle.com/technetwork/java/eol-135779.html")* schreibt, sollen in Zukunft alle sechs Monate Aktualisierungen erfolgen. Das Open JDK 9 steht unter *„http://jdk.java.net/9")* zum Download bereit.

JUGS
java user group stuttgart

Zwanzig Jahre Java User Group Stuttgart, zwanzig Mal Java Forum Stuttgart

Oliver Böhm, Java User Group Stuttgart

Das Java Forum Stuttgart findet dieses Jahr zum zwanzigsten Mal statt - eine sehr lange Zeit in der IT. Der Artikel lässt die wichtigsten Ereignisse der vergangenen Jahre nochmals Revue passieren und ruft die Tops und Flops der letzten zwei Jahrzehnte in Erinnerung.

1998

Im ersten Jahr ihre Bestehens beschließt die Java User Group Stuttgart, eine eigene Java-Konferenz auf die Beine zu stellen, als schwäbische Antwort auf die JavaOne. Dazu startet man gleich mit „300 % Java" – zumindest behauptete das damals noch Oracle in ihren Vorträgen.

1999

Kennt noch jemand das San-Francisco-Framework? Damit will IBM die komplette (Geschäfts-)Welt abbilden. Allerdings ist es so generisch, dass man damit nicht arbeiten kann. Mit der Verabschiedung von EJB 1.0 und 1.1 kommt das Aus für dieses Projekt. Jedes neue Projekt setzt damals auf EJB.

2000

Trotz aller anderslautenden Vorhersagen haben wir nicht nur den Jahrtausendwechsel überlebt, sondern auch den Wechsel des Java Forums vom Hotel am Schlossgarten ins SI-Centrum. Damit vervielfacht sich die Teilnehmerzahl auf 600. Die OSGi Alliance verabschiedet Release 1, mit dem es jetzt ein standardisiertes Komponenten-System für Java gibt.

2003

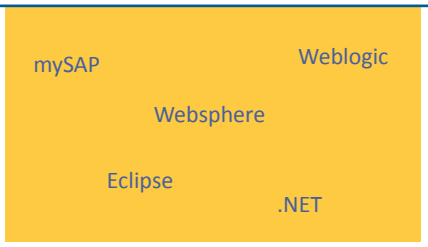
Spring 0.9 kommt als leichtgewichtige Alternative zu J2EE heraus, damals von der Firma Interface 21, später Spring Framework, danach SpringSource, heute Pivotal. Das alte Java-Logo wird entwirrt und klarer gestaltet. Sun versucht, mit J2ME im Mobile-Markt 1.0 Fuß zu fassen.

2001

Im Fokus 2001 steht der Euro, der am 1. Januar 2002 eingeführt wird. Nachdem sechzig bis achtzig Prozent aller Businesslogik in Cobol vorliegt, ist das Thema „Mainframe" natürlich auch auf dem Java Forum vertreten. Ein Vortrag lautet „Java-Programmierung unter CICS".

2004

Java ist nicht nur eine Insel, sondern eine Sprache, die mit dem Tiger-Release ihre erste große Erweiterung in Form von Generics und Annotationen erhält. Auch die Nummerierungslogik wird geändert, nach Java 1.4 kommt Java 5, gründlich überarbeitet und windschnittig: Java-Programme sind jetzt genauso schnell wie native Anwendungen.



2002

Der Enterprise-Bereich ist stark in Bewegung, was sich in Präsentationen zu WebSphere (IBM), WebLogic (BEA, jetzt Oracle) und mySAP niederschlägt. Auch zu „ .net" gibt es drei Vorträge. Eclipse löst langsam JBuilder als Standard-IDE ab und es gibt noch die legendären „Java – Quo Vadis?"-Vorträge von Sun.

2005

In diesem Jahr wird nicht nur der Begriff „Ajax" geprägt, sondern es findet auch das „AspectJ Winter Camp" in der Alten Scheuer in Degerloch statt. AspectJ ist eine aspektorientierte Erweiterung von Java, die unter anderem Bestandteil von Spring AOP ist.

2006

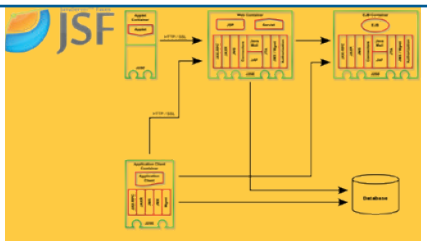
Java 6 unterstützt Skript-Sprachen und bringt mit Rhino eine eigene Scripting-Engine für JavaScript mit. Damit entwickeln sich neue Sprachen wie Groovy oder Scheme, aber auch viele alteingesessene Sprachen wie Python, PHP, Ruby oder TCL finden eine neue Heimat auf der Java-VM.

2009

Nachdem Sun kurz vor dem Aus steht und jeder erwartet, dass IBM Sun aufkauft, übernimmt Oracle im April überraschend die Firma. Lange Zeit ist nicht klar, wie es mit Java weitergehen soll. Anfang 2010 verlässt James Gosling, der Urvater von Java, Oracle. Im Herbst finden zum ersten Mal die Stuttgarter Testtage statt.

2007

Das Java Forum zieht vom SI-Centrum in die Stuttgarter Liederhalle um und kann die Teilnehmerzahl von 800 auf 1.100 steigern. Auch wenn es in den ersten Jahren ein finanzieller Verlust ist, hat es sich doch gelohnt, das Risiko einzugehen. Sollte man allerdings noch weiterwachsen, wird es langsam schwierig …



2010

Java 7 lässt immer noch auf sich warten. Dafür werden JEE 6 sowie JSF 2.0 verabschiedet und sind Bestandteil des Java Forums. SOA ist (immer) noch aktuell und mit zwei Vorträgen vertreten. Eclipse bekommt mit 4.0 einen neuen Unterbau und Steve Jobs verbannt Flash aus dem Apple-Universum.

2008

Im September bringt Google ein Jahr nach dem iPhone Android 1.0 auf Basis von Linux und Java heraus. „Android – Freund oder Feind" lautet ein Vortrag auf dem Java Forum dazu; Fazit: „Freund". Dies hält Oracle aber nicht davon ab, Google nach der Übernahme von Sun wegen angeblicher Lizenz-Verletzungen zu verklagen.

2011

Nach fünf Jahren Stillstand erscheint endlich Java 7. Es ist die letzte Version, die intern einen Codename (Dolphin) enthält, und die erste Version, die unter der Regie von Oracle herauskommt. Das Thema „Cloud" findet Einzug in die Java-Welt und in diverse Vorträge. Damit einhergehend tritt die Frage nach der besten UI-Technologie und deren automatisiertes Testen immer mehr in den Fokus.

2012

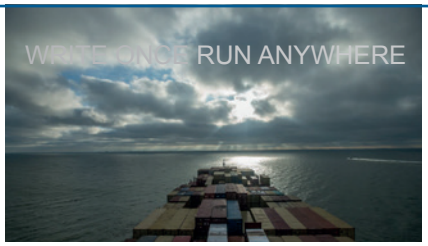
Eclipse Juno leitet den endgültigen Umstieg von Eclipse 3 auf Eclipse 4 ein. Version 3.8 ist die letzte Version der Dreier-Reihe. JavaFX tritt als Alternative zu Silverlight und Adobe Flash auf die Bühne – beides Frameworks, die bereits auf dem Rückzug sind. Maven bekommt mit Gradle einen neuen Konkurrenten – und GIT mit Gerrit einen neuen Begleiter zur Unterstützung des Clean-Code-Gedankens.

2015

Das Hype-Thema „Microservices" machte auch vor dem achtzehnten Java Forum nicht halt, sondern manifestiert sich in einigen Vorträgen. Weiterer Schwerpunkt neben Java 8 ist JavaFX, auf das große Hoffnung als Nachfolger von Swing gesetzt wird. Weitere Themen sind NoSQL und Big Data.

2013

Es findet die erste und einzige Mobile Forum Stuttgart mit zweihundert Teilnehmern und drei parallelen Tracks im Haus der Wirtschaft in Stuttgart statt, dessen Abkürzung „MFS" zu einigen Diskussionen innerhalb des JUGS-Boards führt. Java ist inzwischen auf mehr als einer Milliarde Desktops und drei Milliarden Mobil-Geräten installiert und gilt als das neue Cobol.



2016

Das Verschiffen von Java-Anwendungen in Containern wird immer populärer – und zwar direkt in die Cloud. Am unteren Ende ist Java auf vielen IoT-Geräten vertreten, sodass der damalige Slogan „Write once, run anywhere" von Sun heute aktueller denn je ist.

2014

Das neue Java 8 mit den Lambda-Funktionen und dem darauf aufbauenden Streaming-API erscheint. Funktionen lassen sich damit losgelöst von Klassen betrachten. Weitere Schwerpunkte sind mobile Anwendungen, TypeScript als das JavaScript für Java-Entwickler und JavaFX als die Alternative zu Flash.

2017

Das zwanzigste Java Forum ist bei fast zweitausend Teilnehmern angekommen.

Bildquellen

1999: *https://www.flickr.com/photos/yuyang226/2188321023*
2000: Java User Group Stuttgart
2006: *https://www.flickr.com/photos/shordzi/5360894205*
2007: Java User Group Stuttgart
2008: *http://www.flickr.com/photos/jdhancock/6051805616*
2009: *http://web.archive.org/web/20100124044749/http://blogs.sun.com:80/jag/entry/so_long_old_friend*
2010: *https://de.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition*
2011: *https://www.flickr.com/photos/clio1789/2721396826*
2013: Java User Group Stuttgart
2015: *https://www.flickr.com/photos/dskley/15558336487*
2016: *https://www.flickr.com/photos/oneeighteen/16048997759*

Oliver Böhm
ob@jugs.org

Oliver Böhm beschäftigt sich mit Java-Entwicklung unter Linux und Aspekt-Orientierter SW-Entwicklung. Neben seiner hauptberuflichen Tätigkeit als JEE-Architekt bei T-Systems ist er Buchautor, Projektleiter bei PatternTesting und Board-Mitglied der Java User Group Stuttgart.

Java EE: Wahl fällt auf Eclipse Foundation

DOAG-Online

Die Gespräche von Oracle mit Anbietern, der Community und Open-Source-Foundations bringen erste Fortschritte bezüglich der Zukunft von Java EE ans Tageslicht, wie Oracle in einem Blog-Beitrag *(siehe „https://blogs.oracle.com/theaquarium/opening-up-ee-update")* schreibt. IBM und Red Hat sollen demnach den Prozess der Umgestaltung stärker unterstützen. Zudem wurde die Eclipse Foundation als Partner ausgewählt, um die Weiterentwicklung der Plattform zu beschleunigen.

Oracle hat sich an IBM und Red Hat gewendet, um sich die Unterstützung für die neue Richtung der Java-EE-Plattform zu sichern. Beide Unternehmen sind bereits stark in den Weiterentwicklungsprozess eingebunden. Als neuer Partner wurde die Eclipse Foundation ausgewählt, weil sie laut Oracle bereits über langjährige Erfahrung mit Java EE und verwandten Technologien verfügt. Oracle verspricht sich davon eine beschleunigte und gemeinschaftliche Weiterentwicklung der Plattform.

André Sept, Leiter der DOAG Java Community, steht der Entscheidung zwiegespalten gegenüber: „Ich hätte eher auf Apache und nicht auf die Eclipse Foundation gehofft. Aber da auch das neue MicroProfile bei Eclipse organisiert wird, sind die richtigen Keyplayer wie IBM und Red Hat sowie – nicht zu vergessen – Tomitribe und Payara schon vereint. Wir von der DOAG Java Community (DJC) werden den Prozess begleiten und unterstützen. Denn das ist eine große Gelegenheit, die Plattform nach vorne zu bringen und sich vor allen Dingen mit der Community einzubringen."

Oracle betont in dem Blog-Eintrag, dass bestehende Java-EE-Lizenznehmer, einschließlich derer, die auf Java EE 8 umsteigen, sowie bestehende WebLogic-Server-Versionen weiterhin unterstützt werden. Bezüglich der zukünftigen Pläne hat Oracle eine Liste mit Vorschlägen unter Vorbehalt von Änderungen veröffentlicht:

- Bisherige Oracle-geführte Java-EE- und verwandte GlassFish-Technologien sollen an eine Foundation übergeben werden (einschließlich RIs, TCKs und der Projekt-Dokumentation)
- Demonstration der Fähigkeit, eine kompatible Implementierung unter Verwendung von Foundation-Ressourcen zu erstellen, die vorhandene Java EE 8 TCKs durchläuft
- Eine neue Marketing-Strategie mit einem neuen Namen für Java EE
- Entwicklung eines klaren Prozesses, um bestehende Spezifikationen weiterzuentwickeln und neue einfließen lassen zu können
- Developer und Sponsoren rekrutieren, um die Plattform innerhalb der Foundations voranzubringen. Dazu gehört auch eine mögliche Einbindung von Eclipse-MicroProfile-Technologien

Projekte erfolgreich nach Java 9 migrieren

Anton Epple, Eppleton IT Consulting

Das neue Modulsystem in Java 9 bietet endlich eine Lösung für die „Classpath Hell", aber was bedeutet das für bestehende Projekte? Dieser Artikel zeigt, was man tun muss, um ein bestehendes Projekt nach Java 9 zu portieren.

Es hat lange gedauert, bis das Java Platform Module System (JPMS) endlich Realität wurde. Vor zwölf Jahren wurde der erste Java Specification Request dazu formuliert. Nach vielen Diskussionen und einigen gescheiterten Anläufen ist es nun endlich soweit. Wer nun erwartet, dass das neue Modulsystem Frameworks wie OSGi ablöst, liegt allerdings falsch.

Zumindest in der derzeitigen Version bietet das Modulsystem nur minimale Features und kann ein ausgewachsenes Modulsystem wie OSGi nicht ersetzen. Der Fokus des JPMS liegt vielmehr darauf, eine „verlässliche Konfiguration" („reliable configuration") zu gewährleisten und „starke Kapselung" („strong encapsulation") zu ermöglichen. Ob diese Aufgaben wirklich erfüllt werden, wird derzeit noch

```
module de.eppleton.module1{
            // mit requires geben wir bekannt, dass
wir ein bestimmtes Modul benötigen
  requires de.eppleton.module2;
}
```

Listing 1

heftig diskutiert. Dieser Artikel konzentriert sich auf die praktischen Auswirkungen; deshalb zunächst nur ein kurzer Überblick über die wichtigsten neuen Features und darüber, was das in der Praxis für ein „normales" Projekt bedeutet.

Verlässliche Konfiguration

Das Ziel „verlässliche Konfiguration" wurde gewählt, um die bestehenden Probleme mit dem „Classpath" zu beheben. Bisher war der Entwickler dafür verantwortlich, der JVM die richtigen Abhängigkeiten über den Klassenpfad bekannt zu machen. Zur Laufzeit laden die „ClassLoader" dann bei Bedarf die nötigen Klassen aus den Ressourcen, die hier angegeben sind – fehlt eine Klasse, gibt es einen „NoClassDefFoundError".

In Java 9 wird der „Classpath" durch den „Modulepath" ersetzt. Mit dem neuen Modulsystem gibt jedes Modul seine Abhängigkeiten selbst bekannt. Das Modulsystem kann dann bereits beim Start prüfen, ob etwas fehlt. Dadurch werden Fehler früher erkannt und die Konfiguration wird verlässlicher. Das Modulsystem stellt ebenfalls sicher, dass jede Bibliothek nur genau einmal geladen werden kann. Um das zu gewährleisten, enthalten Module eine spezielle „module-info"-Klasse für die nötigen Informationen *(siehe Listing 1)*.

Starke Kapselung

Java bietet dem Entwickler ausreichende Möglichkeiten, um den Zugriff auf Typen und Methoden zu kontrollieren. Diese können als „public", „protected", „private", oder „package private" definiert sein. Auf Ebene der Packages fehlte jedoch bislang solch ein Mechanismus. In Java 9 kann man nun festlegen, wer auf Packages seines Moduls zugreifen darf. Das sind entweder nur das eigene Modul, eine Liste namentlich aufgeführter Module oder alle anderen Module. Die Konfiguration erfolgt wieder über die „module-info"-Klasse.

Jetzt kann man endlich mit Java-Bordmitteln verhindern, dass Entwickler auf Implementierungsdetails zugreifen und versehentlich eine Abhängigkeit darauf setzen, die beim nächsten Release Schwierigkeiten machen könnte. Der Entwickler eines Java-API kann dadurch die Implementierungsdetails ändern, ohne Rücksicht auf eventuelle Abhängigkeiten nehmen zu müssen *(siehe Listing 2)*.

Die modulare Plattform

Die Hauptarbeit bei der Modularisierung von Java steckt in der Aufteilung der Plattform selbst. Das Java-API wurde in Java 9 in kleinere Module unterteilt und der Zugriff auf Interna ist durch die Mechanismen der „starken Kapselung" beschränkt. Das ermöglicht es nun, nur die Teile mit einer Anwendung auszuliefern, die tatsächlich verwendet werden. Dies ist ein großer Vorteil für Desktop-Anwendungen, die eine eigene JVM mitbringen, und für „embedded" Anwendungen, die wenig Speicher zur Verfügung haben. Auch „containerisierte" Anwendungen profitieren von diesen Optimierungen. Gleichzeitig ist die Verkapselung der Interna eine der größten Herausforderungen für bestehende Anwendungen.

```
Module de.eppleton.module2{
            // nur diese Package ist von aussen für
andere Module zugänglich
  exports de.eppleton.module2.api;
            // Es ist auch möglich Exporte mit „to"
auf bestimmte Module zu beschränken
            // Nur diese können dann zugreifen
             exports de.eppleton.module2.impl
                              to de.eppleton.module3;
}
```

Listing 2

Um die Modularisierung möglich zu machen, wurden die Core-APIs massiv umgebaut. Für bestimmte Funktionen, die bislang nur durch den Zugriff auf interne Packages wie „sun.reflect" möglich waren, musste Ersatz geschaffen werden. Diese Änderungen muss man nun in den eigenen Anwendungen nachziehen.

Stufe 1: Test der bestehenden Anwendung auf Java 9

Wer nun sein Projekt auf Java 9 umziehen möchte, sollte zunächst einfach versuchen, die – mit Java 8 kompilierte – Anwendung unter Java 9 zu starten. Java 9 hat zwar den „Modulepath" eingeführt, um den „Classpath" zu ersetzen, es gibt diesen allerdings noch. Beide können sogar miteinander in der gleichen Anwendung zum Einsatz kommen.

Die Klassen des „Classpath" werden dann alle als Teil eines namenlosen Moduls betrachtet („Unnamed Module"). Dieses hat automatisch Abhängigkeiten auf alle exportierten Klassen der echten Module. Das werden wir später auch nutzen, um die Anwendung Schritt für Schritt zu migrieren. Umgekehrt haben die echten Module („named Modules") keinen Zugriff auf die Klassen des „Classpath" beziehungsweise des „Unnamed Module".

Falls man bislang die Regeln befolgt hat und die Anwendung keine internen Packages verwendet, die in Java 9 ersetzt wurden, stehen die Chancen gut, dass die Anwendung läuft. Es kann aber auch sein, dass sie interne APIs verwenden. In den meisten Fällen ist es dann eine Klasse aus einem Package aus dem „sun"-Namensraum, also zum Beispiel „sun.security.action.GetBooleanAction".

Es gibt auch ein paar Überraschungen. Zum Beispiel wurde die Methode „getPeer" aus der Klasse „java.awt.Component" entfernt, weil die Klasse „Peer" zum internen API zählt, obwohl das Package „java.awt.peer" zum Java-Namensraum gehört.

Wenn die Klasse nur zum internen API erklärt wurde, aber dennoch weiter existiert, lässt sich das Problem beheben, indem man mit-

```
swinggreeter-1.0-SNAPSHOT.jar -> java.desktop
   de.eppleton.swinggreeter.SwingGreeterService       ->
java.awt.peer.ComponentPeer                          JDK internal API (java.desktop)

Warning: JDK internal APIs are unsupported and private to JDK implementation that are
subject to be removed or changed incompatibly and could break your application.
Please modify your code to eliminate dependence on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API                          Suggested Replacement
----------------                          ---------------------
java.awt.peer.ComponentPeer               Should not use. See
https://bugs.openjdk.java.net/browse/JDK-8037739
```

*Listing 3: $ jdeps -jdkinternals libs/**

hilfe des Kommandozeilen-Parameters „-add-exports <Modul-mit-Interna>/<Name-der-internen-Package>=ALL-UNNAMED" die Import-Beschränkung aufhebt. Besser ist es jedoch, das Problem gleich richtig zu erledigen. Dazu kann man wieder das Tool „jdeps" verwenden. *Listing 3* zeigt, wie „jdeps" deutlich auf das Problem hinweist und einen Link mit Hinweisen zur Lösung des Problems liefert. Auch wenn die Anwendung läuft, sollte man das Tool verwenden, da es auch Fälle entdeckt, die beim Test der Anwendung vielleicht nicht abgedeckt sind. Sollte die Anwendung jetzt laufen, ist sie mit Java 9 kompatibel. Die Modularisierung kann nun Schritt für Schritt erfolgen.

Stufe 2: Kompilieren der Anwendung mit Java 9

Die nächste Hürde ist das Kompilieren der Anwendung mit Java 9. Auch wenn der Code Binär-kompatibel ist, ist er nicht automatisch Quelltext-kompatibel. Hier sind allerdings keine schlimmen Überraschungen zu befürchten. Die größte Änderung ist, dass der Unterstrich „_" von Java 9 an als Identifier verboten ist, da er eventuell einmal als Keyword verwendet wird. Man muss also gegebenenfalls einige Felder umbenennen. Auch hier liefert der Compiler Hinweise zur Lösung des Problems *(siehe Listing 4)*.

```
[ERROR] Failed to execute goal org.apache.maven.
plugins:maven-compiler-plugin:3.1:compile (default-
compile) on project hello: Compilation failure
[ERROR] /Users/antonepple/Entwicklung/Eppleton/train-
ings/Java9Migration/projects/legacyproject_porting/
hello/src/main/java/de/eppleton/hello/HelloWorld.
java:[8,9] as of release 9, '_' is a keyword, and may
not be used as an identifier
```

Listing 4

```
#vorher:
$JAVA9_HOME/bin/java -cp
lib1.jar;lib2.jar;lib3.jar;thirdpartylib.jar
de.eppleton.hello.HelloWorld

#nachher:
$JAVA9_HOME/bin/java -cp
lib1.jar;lib2.jar;lib3.jar -p thirdpartylib.jar
de.eppleton.hello.HelloWorld
```

Listing 5

Stufe 3: Third-Party-Bibliotheken werden Automatic Modules

Ist die Quellcode-Kompatibilität hergestellt, können wir mit dem eigentlichen Modularisieren beginnen. Als Erstes sollte man sich darum kümmern, externe Abhängigkeiten zu aktualisieren. Wenn es bereits eine Java-9-kompatible Version der Bibliothek gibt, dann sollte man diese auch verwenden. Wenn nicht, dann benutzt man die externe Bibliothek als „Automatic Module". Dazu wird die Bibliothek einfach vom „Classpath" auf den „Modulepath" verschoben.

Die JVM generiert sich dann automatisch eine „module-info"-Klasse. Da keine Information darüber vorliegt, welche Klassen exportiert werden sollen, werden einfach alle Packages freigegeben. Den Namen des Moduls legt die JVM anhand des Namens der JAR-Datei fest oder anhand eines bestimmten Eintrags in der Manifest-Datei („Automatic-Module-Name").

Wenn wir in einem späteren Schritt unsere eigenen JARs modularisieren, sind Abhängigkeiten auf die so erzeugten Module zu setzen. Zunächst ist das nicht nötig, da unsere Klassen auf dem Klassenpfad Teil des „Unnamed Module" sind und auf alle anderen Module, inklusive der automatischen Module, Zugriff haben.

Stufe 4: Migration von unten

Die empfohlene Vorgehensweise, um die eigenen Module zu portieren, ist die „Bottom-Up-Migration". Dazu muss man zunächst Kandidaten für die Modularisierung identifizieren, um sie aus dem „Unnamed Module" herauszulösen. Auch hier ist wieder das Tool „jdeps" sehr hilfreich. Es hilft bei der Suche nach einem Modul, das keine Abhängigkeiten auf die anderen nicht-modularen JARs hat. Das in *Listing 6* gezeigte JAR hat nur eine Abhängigkeit auf „java.base" und unser „thirdpartylib", aber auf kein anderes JAR des Classpath. Damit wäre es ein guter Kandidat.

Ein JAR mit weiteren Abhängigkeiten könnten wir nicht auf den „Modulepath" legen, denn dort hat es auf die Komponenten des „Classpath" („Unnamed Module") keinen Zugriff mehr. Haben wir ein passendes Modul identifiziert, legen wir eine Datei „module-info.

```
$ jdeps lib2.jar
lib2.jar -> java.base
lib2.jar -> thirdpartylib

   de.eppleton.lib2 -> java.lang         java.base
   de.eppleton.lib2 -> de.acme     thirdpartylib
```

Listing 6

```
$ jdeps lib1.jar
lib1.jar -> java.base
lib1.jar -> lib2.jar

de.eppleton.lib1   -> de.eppleton.lib2.api    lib2.jar
…
```

Listing 7

```
module de.eppleton.lib2{
  requires thirdpartylib;
           exports de.eppleton.lib2.api;
}
```

Listing 8

```
module module3{
            requires de.eppleton.module1;
  provides de.eppleton.module1.api.Service
                with de.eppleton.module3.services.
MyService;
}
```

Listing 9

```
module module2{
            requires de.eppleton.module1;
  uses de.eppleton.module1.api.Service;
}
```

Listing 10

java" im Wurzelverzeichnis des Quellcodes an. Als Modulname wird der Name des Basispackage empfohlen. Wenn unser Modul eine Third-Party-Bibliothek verwendet, müssen wir nun mit „requires" eine Abhängigkeit auf das entsprechende Modul setzen. Zusätzlich müssen wir die Packages exportieren, die von anderen Klassen verwendet werden sollen. Dazu nutzen wir wieder die Ausgabe des „jdeps"-Tools *(siehe Listing 7)*. Daraus ergibt sich dann die „module-info" *(siehe Listing 8)*.

Stufe 5: Schritt für Schritt modularisieren

Sobald alle Abhängigkeiten eines JAR vom „Classpath" auf den „Modulepath" gewandert sind, kann auch dieses JAR zum Modul werden. Die Abhängigkeiten, die wir zuvor mit „jdeps" identifiziert haben, müssen wir nun mit „requires" in der „module-info" angeben. So arbeiten wir uns von unten nach oben durch den Abhängigkeitsgraphen, bis der „Classpath" schließlich leer ist. Unsere Anwendung bleibt während dieses ganzen Prozesses immer lauffähig. So können wir notfalls auch die Zwischenschritte einrichten und Fehler bei der Migration schnell erkennen.

Stufe 6: Service-Abhängigkeiten migrieren

Seit Java 6 gibt es die Möglichkeit, Services zu registrieren und diese lose gekoppelt zu verwenden. Die Services werden bislang mithilfe einer Datei im Verzeichnis „META_INF/services" registriert. Um den Service zu konsumieren, wird die Klasse „ServiceLoader" verwendet. Solange sich die JARs auf dem „Classpath" befinden, funktioniert das auch in Java 9 problemlos. Auch nach der Modularisierung muss am Java-Code nichts geändert werden. Das Modul, das den Service implementiert, muss dies jedoch über die „module-info" mithilfe der Keywords „provides" und „with" bekannt geben *(siehe Listing 9)*. Das Modul, das den Service konsumiert, gibt dies mit dem Keyword „uses" ebenfalls bekannt *(siehe Listing 10)*.

Fazit

Dieser Überblicksartikel ist nicht auf jedes Detail eingegangen. In größeren Projekten wartet sicher noch die eine oder andere interessante Überraschung. Es besteht dennoch kein Grund, wegen der bevorstehenden Migration auf Java 9 in Panik zu verfallen. Es gab zwar viele Diskussionen und heftigen Widerstand gegen die Umsetzung. Sie kommen jedoch vor allem von den Herstellern der Build- oder Testing-Tools, JVMs oder Applikationsservern. Für normale Entwickler haben die Änderungen lange nicht so viele Auswirkungen. „Automatic Modules" und „Unnamed Module" bieten Möglichkeiten, die eigene Anwendung Schritt für Schritt zu portieren und zwischendurch immer wieder zu testen. Die Koexistenz von „Modulepath" und „Classpath" ermöglicht uns in vielen Fällen, Anwendungen ohne Änderungen auf Java 9 laufen zu lassen, sodass man der Migration relativ entspannt entgegensehen kann.

Anton Epple
toni.epple@eppleton.de

Anton „Toni" Epple ist Java-Entwickler der ersten Stunde und hat sich auf Client-Technologien spezialisiert. Er nutzt seine langjährigen Erfahrungen, um mit DukeScript eine moderne, Java-basierte Desktop-Technologie zu entwickeln. Toni ist Java Champion, JavaONE Rockstar, Mitglied im NetBeans Dream Team und Gewinner des Duke's Choice Award. Aktuell bereitet er Entwickler mit seinen Workshops auf den Umzug nach Java 9 vor.

JShell: REPL in Java 9

Thorsten Ludwig, inovex GmbH

Mit JShell hält endlich auch eine REPL in Java Einzug. Wer bereits aus anderen Sprachen die Vorzüge des interaktiven Entwickelns kennengelernt hat, wird das Release von Java 9 kaum noch abwarten können. Der Artikel zeigt, wie die REPL funktioniert, welche Möglichkeiten und Einschränkungen vorhanden sind, und gibt einen Ausblick, was mit passendem Tool-Support möglich sein wird.

Das Release von Java 9 steht kurz bevor, da lohnt sich ein Blick auf die neuen Features. Neben dem neuen Modulsystem namens „Jigsaw" ist JShell die interessanteste Neuerung. Damit gibt es endlich eine REPL in Java. REPL steht für „Read Eval Print Loop". Wer schon einmal Bekanntschaft mit funktionalen Programmiersprachen gemacht hat, wird das Konzept einer REPL kennen. Es handelt sich in der Regel um ein Command-Line-Tool, bei dem man Code eingibt, woraufhin dieser ausgewertet und das Ergebnis ausgegeben wird.

Damit lassen sich vor allem zwei Dinge erledigen: Zum einen eigener oder fremder Code erkunden, etwa das neue Projekt, das man übernimmt oder eine Bibliothek, die man noch nicht kennt; zum anderen kann man direkt in der REPL entwickeln, hier hat sich die Bezeichnung „REPL-Driven-Development" (RDD) etabliert. Der Vorteil: Man bekommt rasend schnell Rückmeldung, ob der Code das macht, was man sich vorgestellt hat. Mit einer guten REPL entfällt also die Notwendigkeit, eine Main-Methode oder Testfälle zu schreiben, nur um schnell etwas auszuprobieren.

Die ersten Gehversuche

Legen wir los, indem wir Java 9 installieren und dann im Terminal mit „jshell" die JShell starten. Es begrüßt uns ein blinkender Cursor, der auf Eingabe wartet. Um uns erstmal zurechtzufinden, tippen wir „/help" ein und bekommen eine Übersicht über alle Befehle. Ganz wichtig, um nicht in einer Falle wie in „emacs" zu landen: Mit „/exit" beenden wir die JShell wieder. Ansonsten gibt es einige Befehle, um verschiedene Einstellungen zu ändern oder unsere bisherigen Eingaben zu sehen. Praktisch ist die Anweisung „/history", die analog zum Linux-Befehl unsere komplette Eingabehistorie zeigt. Mit der Pfeiltaste nach oben können wir unsere bisherigen Eingaben wiederholen. Die restlichen wichtigen Befehle werden im Laufe des Artikels erwähnt. Geben wir jetzt erst mal ganz simpel „3+5;" ein, ein einfacher, aber gültiger Ausdruck in Java. *Listing 1* zeigt die Antwort.

Aufgrund unserer Mathematik-Kenntnisse wissen wir sofort, dass „8" das Ergebnis der Rechnung sein muss, aber was hat es mit „$1" auf sich? Hier handelt es sich um eine anonyme Variable, die JShell für uns angelegt hat. Jedes Mal, wenn wir etwas eingeben, das einen Wert zurückliefert, speichert die JShell das für uns unter „$x" ab, wobei „x" bei jedem Mal inkrementiert wird.

Die Variable lässt sich nun auch weiterverwenden *(siehe Listing 2)*. Beim letzten Listing sehen wir ein weiteres Killer-Feature der JShell: Man kann Semikolons weglassen! Wir wollen die JShell aber nicht nur als Taschenrechner benutzen, daher führen wir auch Methoden aus, etwa „substring" auf einem String *(siehe Listing 3)*.

Gibt man nur „.sub" ein und drückt dann Tabulator, hilft einem die JShell mit Vorschlägen *(siehe Listing 4)*. Tippt man nun ein weiteres „s" ein, wird einem mit einem Druck auf Tabulator der Name vervollständigt, ein erneutes Drücken schlägt einem die möglichen Signaturen vor *(siehe Listing 5)*.

Ein nochmaliges Tippen präsentiert die Dokumentation zu der Funktion. Wir können also nun mit der JShell die eingebauten Java-Funktionen erkunden. Schön wäre es auch, fremden Code auszuprobieren, etwa aus einer JAR.

I've got a JAR of code!

Als Beispiel nehmen wir die Commons-Lang-Library. Um sie zu benutzen, passen wir „class path" an *(siehe Listing 6)*. Die JShell spielt jetzt noch einmal alle Eingaben mit dem neuen „class path" durch. Nun können wir loslegen. Um uns Tipparbeit zu sparen, importieren wir erstmal „StringUtils". Schade – die Autovervollständigung funktioniert nur für eingebaute Packages, erst für den Klassennamen hilft uns die JShell: „jshell> import org.apache.commons.lang3.StringUtils". Wenn wir später einen Überblick über die Imports haben wollen, gibt uns „/imports" eine Liste über alle importierten Klassen aus.

Geben wir nun „StringUtils." ein, hilft uns die JShell mit Autovervollständigungen. Wir können die verschiedenen Methoden ausprobieren und bekommen die Signaturen vorgeschlagen. Leider bekommen wir von den importierten JARs keine Dokumentation angezeigt, selbst wenn wir die JAR mit den JavaDocs zusätzlich in den „class path" übernehmen.

Ähnlich funktioniert es, wenn wir unser eigenes Projekt in der JShell ausprobieren wollen. Als „class path" übernehmen wir dann einfach unser „build"-Verzeichnis. Um mehrere Verzeichnisse als „class path" zu übernehmen, trennen wir diese mit einem Doppelpunkt. Wollen wir Änderungen an unseren Klassen auch in der JShell sichtbar machen, müssen das Projekt neu gebaut und die Änderungen in die JShell übernommen werden, das geht am schnellsten mit dem Befehl „/reload". Die JShell startet übrigens mit dem Verzeichnis als „class path", in dem es gestartet wurde.

Ein Schritt Richtung RDD

Mit den bisher gesehenen Mitteln können wir nun vorhandenen Code ausprobieren. Der nächste Schritt wäre, direkt in der JShell zu entwickeln, um so ein schnelleres Feedback zu bekommen. Analog zum „Test-Driven-Development" hat sich hierfür der Name „REPL-Driven-Development" etabliert. Als Erstes wollen

```
jshell> 3+5;
$1 ==> 8
```

Listing 1

```
jshell> $1 * 2
$2 ==> 16
```

Listing 2

```
jshell> "Hallo Java aktuell".substring(6,10)
$4 ==> "Java"
```

Listing 3

```
jshell> "Hallo Java aktuell".sub
subSequence(   substring(
```

Listing 4

```
Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)
```

Listing 5

```
jshell>/env -class-path /home/thorsten/Downloads/com-
mons-lang3-3.6/commons-lang3-3.6.jar
|  Setting new options and restoring state.
```

Listing 6

```
jshell> public static String helloWorld() {
   ...> return "Hello World"
   ...> }
|  Error:
|  ';' expected
|  return "Hello World"
|
```

Listing 7

```
jshell> public static String helloWorld() {
   ...> return "Hello World";
   ...> }
|  Warning:
|  Modifier 'static'  not permitted in top-level
   declarations, ignored
|  public static String helloWorld() {
|  ^-----------^
|  created method helloWorld()
```

Listing 8

wir eine simple Methode schreiben, die uns „Hello World" zurückgibt. Also tippen wir „jshell> public static String helloWorld() { ...>" ein. Drücken wir auf Enter und bevor unsere Methode fertig ist, fordert ein freundlicher Pfeil uns auf, die Methode doch zu Ende zu schreiben *(siehe Listing 7)*. Schade, sobald wir Methoden schreiben, müssen wir die Semikolons doch setzen, also ein zweiter Anlauf *(siehe Listing 8)*.

Immerhin wurde unsere Methode erstellt, aber anscheinend kann man keine statischen Methoden erstellen? Da wir uns in der REPL befinden, können wir das direkt ausprobieren *(siehe Listing 9)*. Die Methode kann ohne Erstellen eines Objekts ausgeführt werden, also ist sie statisch, nur dass das Keyword „static" nicht erlaubt ist. Dementsprechend würde auch ein „this" in der Methoden-Deklaration zu einem Fehler führen. Wir können allerdings Variablen in der REPL definieren und sie in den Methoden verwenden *(siehe Listing 10)*.

Es lassen sich auch die anonymen Variablen verwenden, wobei sich hier natürlich die Frage der Sinnhaftigkeit stellt. Will man Änderungen an der Methode vornehmen, kann man sie einfach überschreiben *(siehe Listing 11)*.

Es ist mühsam, die Methode jedes Mal komplett einzugeben, da wäre es schön, wenn es einen einfacheren Weg gäbe, den Code zu verändern. Der Befehl „/edit" öffnet den eingebauten Editor, wir ändern die Methode und speichern sie mit einem Klick auf „Exit". Falls einem der eingebaute Editor nicht gefällt, lässt sich mit dem „/set"-Befehl auch ein beliebiger anderer Editor setzen. Stattdessen kann man aber auch jeden anderen Bash-Befehl schreiben *(siehe Listing 12)*.

Ist der Editor noch nicht gestartet, blockiert die JShell, bis der Editor beendet ist, im anderen Fall kann man direkt in der JShell weiterarbeiten. Dann werden allerdings die Änderungen nicht in die JShell übernommen – schade.

Diese REPL hat Klasse

Java wäre nicht Java, wenn man nicht auch ein paar schöne Klassen schreiben kann. Und natürlich geht das auch in der JShell *(siehe Listing 13)*. Wollen wir nun etwas an der Klasse ändern, gehen wir genauso vor wie bei einer Methode. Entweder wir geben die Klasse komplett neu ein oder wir starten den eingestellten Editor.

Wenn wir uns jetzt eine Übersicht darüber verschaffen wollen, was wir bisher an Code in die JShell eingegeben haben, helfen uns einige eingebaute Befehle. Mit „/list" sehen wir ähnlich wie bei „/history" unsere bisherigen Eingaben, allerdings wird nur der eingegebene Code angezeigt, Befehle werden herausgefiltert. Der Befehl „/methods" zeigt alle definierten Methoden an, „/vars" alle Variablen und „/types" alle Typen, also etwa Klassen, Interfaces oder Enums, „/imports" zeigt alle importierten Klassen an. Mit „/save" und „/open" können wir unsere Code-Snippets speichern und später wieder laden.

Da wir diese aber jedes Mal neu starten müssen, werden unsere Feedbackzyklen wieder so lang, dass sich der Vorteil von RDD ins nichts auflöst. Was also tun? Wir brauchen passende Tools und Integration in unsere Entwicklungsumgebung, damit wir die JShell effizient in unseren Entwicklungsprozess einbauen können. Zum Glück liefert die JShell alles mit, um die Integration zu ermöglichen.

```
jshell> helloWorld()
$3 ==> "Hello World"
```

Listing 9

```
jshell> int a = 42;
a ==> 42

jshell> public int getA() {
   ...> return a;
   ...> }
|  modified method getA()

jshell> getA()
$10 ==> 42

jshell> a = 23
a ==> 23
jshell> getA()
$12 ==> 23
```

Listing 10

```
jshell> public int getA() {
   ...> return a * 2;
   ...> }
|  modified method getA()

jshell> getA()
$14 ==> 46
```

Listing 11

```
jshell> /set editor /opt/idea/bin/idea.sh
|  Editor set to: /opt/idea/bin/idea.sh
```

Listing 12

Tool-Integration

Die JShell bietet ein API an, mit dem sie sich von außen komplett steuern lässt. Die Dokumentation dazu findet man unter *„http://cr.openjdk.java.net/~rfield/arch/doc/jdk/jshell/JShell.html"*. Somit kann jeder ein Programm oder ein Plug-in schreiben, das eine oder mehrere Instanzen der JShell aufruft. Eine Integration direkt in die IDE bietet sich hier an, daher ein kurzer Blick auf den Stand der Entwicklung bei den drei großen IDEs:

- **Eclipse**
  Eine direkte Integration in Eclipse wird es vermutlich nicht geben, allerdings wäre ein Plug-in für Eclipse möglich. Bisher gibt es allerdings noch keins, daher liebe Eclipse-User: Geht voran und integriert die JShell in Eclipse.
- **IntelliJ**
  Laut dem Issue-Tracker von JetBrains ist zumindest die grundlegende Integration abgeschlossen. Mit welcher Version das Feature veröffentlicht wird, ist noch nicht bekannt, es wird aber nicht vor der Version 2017.3 sein.

- **NetBeans**
  In den Nightly Builds kann man schon einen Ausblick auf die JShell-Integration in NetBeans werfen. Die JShell ist nun in die IDE integriert, der Code kann einfach bearbeitet werden – und wenn uns der Code aus der REPL gefällt, kann er per Knopfdruck in eine neue Klasse exportiert werden. Schade: Es wird wohl eine veraltete Version der JShell integriert, die Befehle weichen von der aktuellen JShell ab. Insgesamt ist NetBeans aber am weitesten mit der JShell Integration, auch wenn noch Raum für mehr Features ist (ausgewählten Code in der REPL laufen lassen, ausgewählten Code extrahieren, einzelne Methoden in vorhandene Klassen extrahieren etc.).

Bis die Integration der JShell in der Lieblings-IDE angekommen ist, kann man immerhin ein Terminal-Fenster in seiner IDE öffnen und dort die JShell laden. Man hat zwar keine Kommunikation zwischen IDE und JShell, aber immerhin alles in einem Fenster.

Fazit

Mit der JShell wandert endlich eine vollwertige, offizielle REPL in das Repertoire von Java-Entwicklern. Auch wenn es noch kleinere Schwächen gibt, etwa fehlende Package-Autovervollständigung für Nicht-Standardklassen, bietet einem die JShell alle wichtigen Features, die wir bei einer REPL sehen wollen. Ob sich die REPL im Entwickleralltag durchsetzen wird, hängt nun von der Tool-Unterstützung, insbesondere durch die IDEs, ab. Die nächsten Monate werden zeigen, wie die Entwickler der IDEs die Unterstützung der JShell mit neuen Features vorantreiben.

Hinweis: Listing 13 finden Sie unter *https://www.doag.org/de/home/news/java-aktuell-ausgabe-52017-jetzt-online/detail/*

Thorsten Ludwig
tludwig@inovex.de

Thorsten Ludwig arbeitet als Software-Entwickler bei der inovex GmbH. Er ist spezialisiert auf die Entwicklung komplexer Software-Systeme auf der JVM und freut sich über jedes neue Feature, das Java ein klein wenig funktionaler macht.

JAX-RS 2.1 in Action

Markus Karg, Java User Group Pforzheim

Es hat lange gedauert, bis Oracle JAX-RS 2.1 endlich fertig hatte. Doch der Schein trügt: Was rein deklarativ als einfaches Minor-Release daherkommt, wird Microservice-Autoren freudig stimmen.

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2 -DarchetypeGroupId=org.glassfish.jersey.
archetypes -DinteractiveMode=false -DgroupId=com.me -DartifactId=demo -Dpackage=com.me -DarchetypeVersion=2.26-b09
```

Listing 1

Ja, es hat wirklich sehr, sehr lange gedauert, bis JAX-RS 2.1 endlich fertig war, und ja, es ist nur die Hälfte der angesagten Themen drin. Aber nein, das macht gar nichts, denn was da nun tatsächlich an Features auf dem Tisch liegt, kann sich aufgrund des effektiven Nutzwerts wirklich sehen lassen. Es hat sich ja in den letzten Jahren architektonisch viel getan in der Software-Welt, gerade „Microservices" und das „Internet of Things" sind Themen, die im Cloud-Umfeld allgegenwärtig im Raum stehen – auch bei Java-Projekten. Die neuen Features von JAX-RS 2.1 gestalten solche Projekte wesentlich effizienter gegenüber dem originalen Release 2.0. Am einfachsten lässt sich dies mit einem kleinen Beispielprogramm demonstrieren.

Stellen wir uns vor, wir hätten ein Sensor-Netzwerk, beispielsweise zur Messung von Temperaturdaten. So etwas kennt man von den beliebten Wetterseiten im Internet. Natürlich können wir im Rahmen dieses Artikels nicht dem Deutschen Wetterdienst Paroli bieten, uns jedoch rein als Gedankenmodell vorstellen, wir möchten einen Service erstellen, der unsere Sensordaten per REST aggregiert zur Verfügung stellt und beispielsweise den Mittelwert aller Sensoren einmal pro Sekunde aktualisiert an die Clients sendet. Ja, sendet, denn mit JAX-RS 2.1 können wir nun endlich „pushen".

Fertig … los!

Fangen wir ganz langsam an, in „old school" JAX-RS 2.0 mit Java 7. Damit es nicht ganz so viel Hackerei wird, benutzen wir Maven sowie den Archetyp der JAX-RS-Referenz-Implementierung Jersey *(siehe „https://jersey.github.io")* und verzichten ansonsten ganz bewusst auf IDEs, Plug-ins, Wizards und den ganzen anderen Kram, der wenig bringt, aber schick aussieht. Tatsächlich entstand der Code in *Listing 1* komplett in Bash mit dem vi-Editor und der Autor hat die IDE kaum vermisst.

Nach wenigen Sekunden befindet sich auf unserer Platte (hat überhaupt noch jemand eine echte Platte?) im Ordner „demo/" das lauffähige Grundgerüst einer richtigen (wenngleich sinnentleerten) JAX-RS-2.- Anwendung, die sich per „mvn compile exec:java" erstellen und starten lässt und der wir mittels cURL die ersten (gleichfalls sinnlosen) Daten entlocken können: „curl http://localhost:8080/myapp/myresource". Diese beiden Befehlszeichen (für Maven und cURL) bitte gut merken. Wir brauchen sie nun laufend und es wäre sicherlich mühselig, sie ständig zu wiederholen.

Übrigens, wer sich fragt, wie man denn auf diese URL kommt: Dazu muss man keineswegs in den Quellcode schauen. Beim Start schreibt die Anwendung netterweise die fix programmierte Root-URL auf „STDOUT"; ein „GET" auf diese, etwa mit cURL, offenbart das gesuchte Geheimnis in Form von WADL – und einiges mehr. WADL ist zwar leider nicht Teil von JAX-RS, was seine diesbezügliche Nützlichkeit allerdings keineswegs mindert.

Im Folgenden nun wollen wir dieser Anwendung zunächst einen (zugegebenermaßen minimalen) Sinn einhauchen, um daraufhin Schritt für Schritt das Ganze mit den neuen Fähigkeiten von JAX-RS 2.1 anzureichern. Der Sinn, wie gesagt, soll sein, Sensoren abzufragen. Zwar haben wir keine echten Sensoren zur Hand, doch wir können diese mit einem kurzen Stück Java-Code simulieren *(siehe Listing 2)*. Auch die automatisch generierte Ressourcen-Klasse bauen wir kurz und schmerzlos um, sodass unnötiger Ballast wie Kommentare und „@Produces"-Annotationen durch die sinnvolle Abfrage des Sensors ersetzt werden *(siehe Listing 3)*. Das ist sicherlich kein Meisterwerk, aber es erfüllt seinen Zweck: Nach dem erneuten Kompilie-

```
@XmlRootElement public class SensorValue {
  public final long time = System.currentTimeMillis();
  public final double value =  100 * Math.random();

  @Override public String toString() {
     return "Time: " + time + " Value: " + value;
  }
}
```

Listing 2

```
@Path("myresource") public class MyResource {
    @GET public SensorValue getSensorValue() {
        return readSensor();
    }

    private static SensorValue readSensor() {
        return new SensorValue();
    }

}
```

Listing 3

```
@GET public Optional<SensorValue> getSensorValue() {
    return readSensor();
}

private static Optional<SensorValue> readSensor() {
    return Optional.of(new SensorValue());
}
```

Listing 4

```
public class InstantXmlAdapter extends
XmlAdapter<String, Instant> {
    public String marshal(Instant v) {
        return Optional.ofNullable(v).
map(Instant::toString).orElse(null);
    }

    public Instant unmarshal(String v) {
        return Optional.ofNullable(v).
map(Instant::parse).orElse(null);
    }
}
```

Listing 5

```
@Provider public class OptionalWriterInterceptor implements WriterInterceptor {
  public void aroundWriteTo(WriterInterceptorContext context) {
     if (context.getType().equals(Optional.class)) {
        Optional<?> optional = (Optional<?>) context.getEntity();
        Class<?> type = (Class) ((ParameterizedType) context.getGenericType()).getActualTypeArguments()[0];
        context.setEntity(optional.orElse(null));
        context.setType(type);
        context.setGenericType(type);
     }
     context.proceed();
  }
}
```

Listing 6

ren und Starten bestätigt cURL uns den Empfang eines kurzen XML-Dokuments mit Timestamp und zufällig simuliertem Sensorwert.

Java 8 ohne Klimmzüge

Was jedoch ärgert, ist der Typ „long" für den Timestamp. Seit Java 8 gibt es dafür einen vernünftigen Daten-Typ („Instant"), und den werden wir verwenden. Dann existieren noch Sensoren, die zwar immer mit einem reden, aber nicht immer einen sinnvollen Wert liefern. So etwas hätten wir in obiger Anwendung per „null" oder mit einer Exception umgesetzt. Da wir den Fehlercode nicht brauchen und da „null" immer die Angst einer „NullPointerException" mit sich bringt, wäre es schöner, hier die Java-8-Klasse „Optional" nutzen zu können.

Wir brauchen also Java 8, und das geht so: Zunächst ersetzen wir im POM kurzerhand den Wert „1.7" durch „1.8", damit Maven weiß, was wir haben wollen. Danach bauen wir den Sensor um auf Instant („public final Instant time = Instant.now();") und die Ressource auf „Optional" *(siehe Listing 4)*. Damit weiterhin XML herauskommt, brauchen wir einen JAXB-Adapter, das war schon in JAX-RS 2.0 so *(siehe Listing 5)*.

Bis hier dürfte das Ganze den meisten JAX-RS-Erfahrenen geläufig sein. Experten wissen natürlich, dass die Einführung von „Optional" ein paar mehr Tricks erfordert: Dazu brauchen wir einen „WriterInterceptor", sonst weiß kein existierender „MessageBodyWriter", was er tun soll – und wir müssten praktisch alle Zieldaten-Formate neu programmieren, einmal mit und einmal ohne „Optional". Da dieses Unterfangen höchst uneffektiv wäre, rüsten wir kurzerhand den Support für „Optional" nach, wodurch alle „MessageBodyWriter" dieser Welt bleiben können, wie sie sind *(siehe Listing 6)*.

Schon haben wir sowohl eine lesbare Uhrzeit als auch die Unterstützung für optionale Rückgabewerte aus Java 8. Der Trick an JAX-RS 2.1 ist also nicht, dass alle möglichen neuen Klassen direkt eingebaut sind (das sind nur wenige), sondern, in Analogie zu anderen erweiterbaren Frameworks wie JAXB, JSON-B und JPA, dass die in JSR 370 erarbeitete Spezifikation erzwingt, dass JRE 8 als Unterbau vorhanden sein muss – denn nur so kann sich die Anwendung darauf verlassen, die neuen JRE-8-Klassen zu finden. Ansonsten würde schließlich das Laden der Ressourcen-Klasse ebenso fehlschlagen wie das Laden des Interceptor oder des JABX-Adapters.

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-binding</artifactId>
</dependency>
```

Listing 7

```
curl -H Accept:application/json http://localhost:8080/
myapp/myresource
{"time":"2017-07-09T15:19:23.288Z","val
ue":57.3497420050259}
```

Listing 8

Good bye, JAXB!

Apropos JAXB-Adapter: Falls es noch nicht bekannt sein sollte, JAXB wird zukünftig nicht mehr automatisch im JRE enthalten sein. Zwar ist es in Java 9 noch da, doch dank des Projekts „Jigsaw" ist es optional – man muss es also extra einschalten. Ob dies in Java 10 noch der Fall sein wird, ist fraglich, denn offiziell ist JAXB ab Java 9 als „deprecated for removal" markiert. Es ist also damit zu rechnen, dass es in Java 10 oder Java 11 verschwunden ist. Die Lösungsstrategie darf sich jeder selbst aussuchen, hier einige zur Auswahl:

- In Java 9 per „--add-modules java.xml.bind" schlicht JAXB wieder an der Kommandozeile einschalten oder die eigene Anwendung mit einer entsprechenden Klausel in „module-info.java" ausstat-

```
@GET @Produces(MediaType.SERVER_SENT_EVENTS)
public void getSensorValue(@Context SseEventSink sseEventSink, @Context Sse sse, @HeaderParam("X-Accept") MediaType
mediaType) {
  timer.schedule(
    () -> sseEventSink.send(sse.newEventBuilder().mediaType(mediaType).data(readSensor().get()).build()).thenRu-
nAsync(() -> getSensorValue(sseEventSink, sse, mediaType)),
    1, TimeUnit.SECONDS
  );
}
```

Listing 9

ten. Annahme: Java 9 wird diese Möglichkeiten in der vorliegenden Form im Final Release enthalten, wonach es bis Redaktionsschluss aussieht.
- In Java 10 als eigenständige Bibliothek der Anwendung hinzufügen wie jede andere Third-Party-Bibliothek auch. Annahme: Oracle führt JAXB, derzeit zu finden auf https://github.com/javaee/jaxb-v2, als selbstständiges Projekt weiter oder übergibt es, ähnlich wie im Falle des MVC-API, einem neuen Maintainer. Nach der kürzlichen Oracle-Ankündigung zur Öffnung von Java EE ist davon auszugehen.
- Oder einfach zugunsten von modernen Formaten wie JSON auf XML verzichten.

Die Cloud spricht JSON

Stichwort JSON: Neben vielen kleineren Änderungen (wie beispielsweise dem Verzicht auf die ohnehin seit JAX-RS 2.0 sinnlose Methode „getSize" in „MessageBodyWritern") bringt JAX-RS 2.1 auch ein paar tolle, größere Sachen mit. Gerade im Hinblick auf Microservices und Cloud-Anwendungen muss an dieser Stelle natürlich erwähnt werden, wie leicht es für den Anwendungsprogrammierer wird, von XML auf JSON – die „Lingua Franca" der Cloud-Welt – umzustellen oder es parallel zu nutzen, denn XML weiter zu unterstützen, erfordert keine Ressourcen. JSON bekommt man nun quasi geschenkt. Derzeit muss es in der Referenz-Implementierung allerdings noch manuell eingeschaltet werden, was über simples Auskommentieren der entsprechenden Dependency im POM geschieht *(siehe Listing 7)*. Das war's! *Listing 8* zeigt den kurzen Test.

Augenscheinlich erfolgreich! Theoretisch könnte man nun also an dieser Stelle XML ausbauen. Die einzige Stelle, an der XML explizit in unserer Anwendung genannt ist, ist die Annotation „@XmlRootElement" über dem SensorValue-POJO. Wird diese entfernt, ist XML effektiv stillgelegt.

Push it!

Das nächste größere Schmankerl ist die Unterstützung für Server-Sent Events (SSE, nicht zu verwechseln mit WebSockets, Comet oder Long Polling). Diese Technologie erweitert reines HTTP um die Möglichkeit, nicht nur eine, sondern beliebig viele Responses pro Request zu senden – und zwar zu einem beliebigen, späteren Zeitpunkt. Die Push-Möglichkeit ist gerade im IoT-Umfeld sinnvoll, etwa um unnötige Network-Roundtrips zu vermeiden.

Nicht verschwiegen werden soll, dass auch JAX-RS nichts daran ändern kann, dass SSE durch die Hintertür Sessions wieder einführt, die man ja in REST-Anwendungen gerade nicht möchte. Wen dies jedoch nicht weiter stört, der bekommt nun das Handwerkszeug in Form der injizierten Ereignisrutsche „SseEventSink" und der zugehörigen Event-Factory „Sse" geliefert – muss sich dann prinzipiell aber um alles Weitere selbst kümmern: JAX-RS-Ressourcen-Methoden müssen „void" als Ergebnis-Typ deklarieren und alle Responses, auch die initiale, selbst produzieren. Ebenso muss sich die Anwendung selbst darum kümmern, wann eine Response geschickt wird, etwa mittels einer Schleife, eines Timers oder wenn ein anderer Client Daten hochgeladen hat *(siehe Listing 9)*. Hierzu gibt es ein Broadcast-API, das in unserem Beispiel aber nicht vorkommt. Um das auszuprobieren, muss allerdings bei Nutzung von Jersey der SSE-Support im POM eingeschaltet sein *(siehe Listing 10)*.

Das Code-Beispiel macht augenscheinlich, dass für SSE einiges Beiwerk notwendig ist. Zum einen darf der initiale Thread, der die SSE-Session ursprünglich öffnet, nicht blockiert werden. Die Erzeugung der Responses erfolgt daher im Beispiel über einen „ScheduledExecutor". Auf eine initiale Response wird der Einfachheit wegen verzichtet, sie könnte aber auf technisch gleichem Wege erfolgen, wenn man den initialen Delay auf „0" setzt statt auf „1 Sekunde". Die Schleife wird im Beispiel über die Reaktion auf das Ende des Sendevorgangs gebildet, indem der „CompletionStage", die von „SseEventSink.send()" geliefert wird (und die per Hintergrund-Thread sendet), befohlen wird, einfach wieder die gleiche Methode („getSensorValue") aufzurufen.

Jeder einzelne Event, der dem Client gesendet werden soll, kann einen anderen MIME-Type haben. Möchte man HTTP-konform dem Client die Wahl über den MIME-Type per Accept-Header geben, steht einem die SSE-Spezifikation im Weg, die besagt, dass der initiale SSE-Session-Aufbau nur erfolgt, wenn der Client per „Accept:text/event-stream" dies fordert. Zwar können mehrere Accept-Header-Values im Request enthalten sein, dies jedoch nur, um dem Server die Auswahl über einen Typ zu lassen, was hier aber explizit ja eine SSE-Session sein soll. Damit steht der Accept-Header nicht mehr

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-sse</artifactId>
</dependency>
```

Listing 10

```
curl -N -H X-Accept:application/json http://localhost:8080/myapp/myresource
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--
0data: {"time":"2017-07-09T17:58:05.303Z","value":23.254565347099575}

100    70    0    70    0     0     31      0 --:--:--  0:00:02 --:--:--
31data: {"time":"2017-07-09T17:58:06.307Z","value":88.88601882225326}

100   139    0   139    0     0     43      0 --:--:--  0:00:03 --:--:--
43data: {"time":"2017-07-09T17:58:07.310Z","value":17.351233704968173}

100   209    0   209    0     0     49      0 --:--:--  0:00:04 --:--:--
49data: {"time":"2017-07-09T17:58:08.315Z","value":23.409041205977122}
```

Listing 11

```
@GET public CompletionStage<SensorValue> getAver-
ageSensorValue() {
  CompletionStage<SensorValue> sensorA =
readSensor("A");
  CompletionStage<SensorValue> sensorB =
readSensor("B");
  return sensorA.thenCombine(sensorB,
SensorValue::average);
}
```

Listing 12

für den gewünschten MIME-Type der Events zur Verfügung. Unser Beispiel befreit sich aus diesem Dilemma, indem es HTTP-konform einen „X-Accept"-Header erfindet. Daher läuft es auch nur dann fehlerfrei, wenn cURL nun statt mit „-H Accept" mit „-H X-Accept" aufgerufen wird *(siehe Listing 11)*.

Der zusätzliche Parameter „-N" verbietet es cURL, die Ausgabe zu puffern und hat weder mit JAX-RS noch mit SSE zu tun: Damit wird lediglich erreicht, dass wir die eingetroffenen Responses sofort auf der Konsole sehen können und nicht warten müssen, bis der entsprechende STDOUT-Puffer voll ist.

Reaktivität

Temperatur-Sensoren sind nicht sonderlich flott: Je nach Modell kann die Abfrage schon eine Weile dauern. Dies fällt beim Pushen nicht auf, da die empfangende Anwendung ja letztendlich gar nicht weiß, wann die Anfrage an den Sensor gestartet wurde. Wenn wir jedoch nicht nur einen einzigen Sensor abfragen, sondern beispielsweise den Mittelwert einer ganzen Sensoren-Batterie (etwa um den Schnitt einer größeren, überwachten Fläche zu erhalten), und wenn wir die Messung zudem explizit anstoßen, werden wir die ewige Wartezeit sicherlich ärgerlich finden. Zudem ist in so einem Fall von langsamer Query anzumerken, dass der Front-End-Thread (also der, der die Netzwerkkarte des Servers bedient) blockiert – und davon hat der Webserver leider nicht unbegrenzt viele. Entsprechend findet keine Kommunikation mehr statt, der Server fühlt sich überlastet an. Gut, ein Raspberry PI ist da jetzt sowieso anderweitig am Limit. Insbesondere in Cloud-Umgebungen stellt dies allerdings ein spürbares Skalierungshemmnis dar.

Um Front-End-Threads von Back-End-Threads zu entkoppeln, gab es schon in JAX-RS 2.0 die Möglichkeit der asynchronen Programmierung. Diese ist aber zugegebenermaßen recht umständlich. Mit dem reaktiven API von JAX-RS 2.1 wird dies sehr, sehr viel übersichtlicher, *Listing 12* ist zur Veranschaulichung auf zwei Sensoren reduziert.

Interessant dabei ist, dass „CompletionStage" nicht die einzige Klasse ist, mit der JAX-RS umzugehen weiß: Über diese Pflichtübung hinaus steht es jeder Implementierung frei, auch weitere reaktive Frameworks zu unterstützen, beispielsweise RxJava. Damit eine Anwendung jedoch portabel bleibt (also nicht nur mit Jersey, sondern beispielsweise auch mit CXF läuft), sollte man sich nicht auf diese spezielle Fähigkeit verlassen und lieber die entsprechende Bibliothek selbst mitbringen sowie das Ergebnis etwa von Observable auf CompletionStage umformen. Dazu gibt es im Web zahlreiche Beispiele. Freuen darf man sich bereits auf das nächste Release von JAX-RS: Dort wird es dann auch Unterstützung für das Flow-API geben, das mit Java SE 9 eingeführt wurde, womit SSE dann erst richtig Spaß macht.

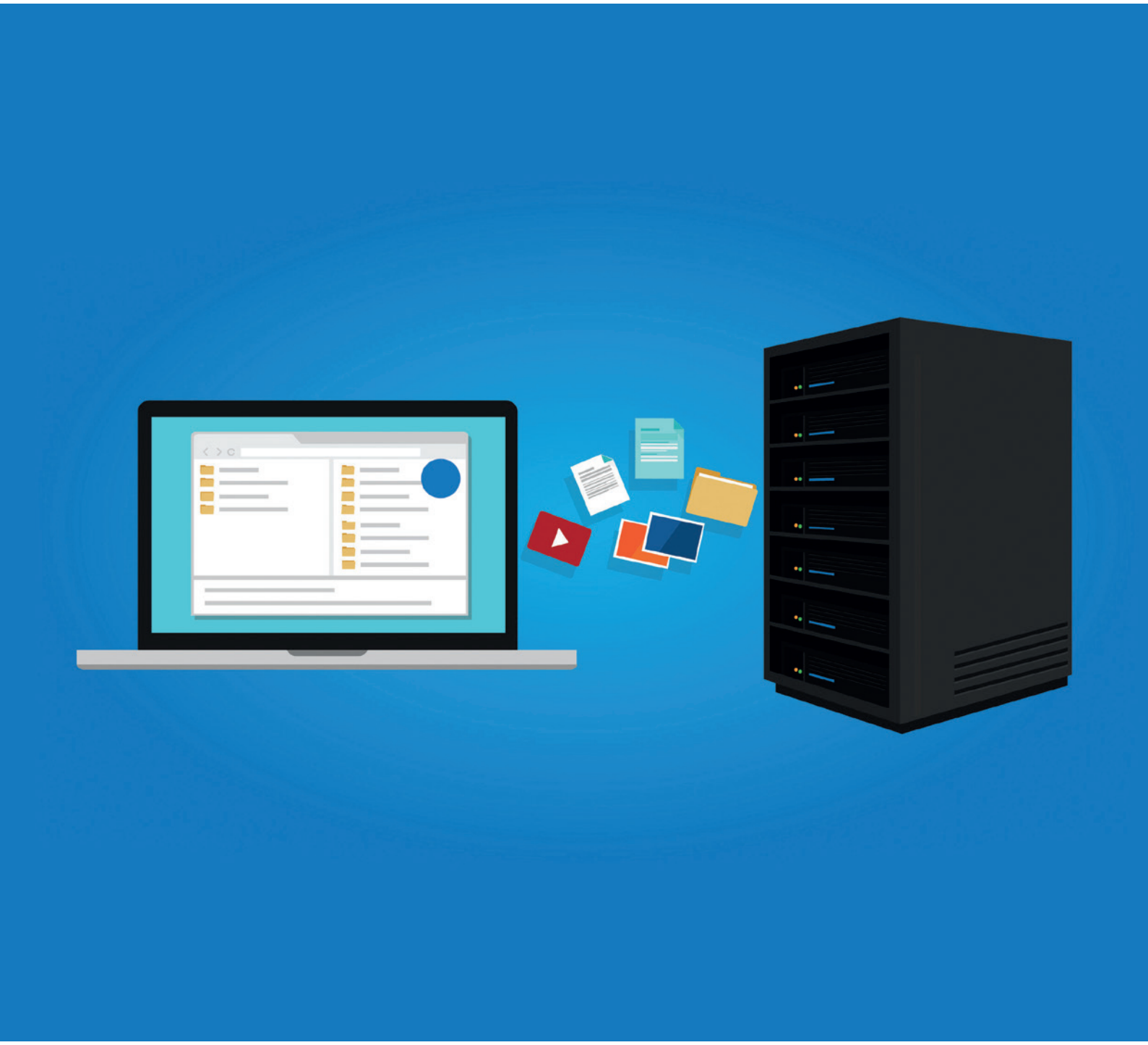
Fazit

Mit JAX-RS 2.1 wurde wertvolle Detailarbeit am REST-API geleistet, das vor allem den Architekturwechsel von Application Server auf Microservices unterstützen soll. Mit der Konzentration auf die Aspekte Java 8, JSON, SSE und reaktive Programmierung folgt der neue Standard diesem Trend, ohne Vorhandenes über Bord zu werfen, das API unnötig aufzublasen oder die Portabilität zu opfern – Stichwort „WORA".

Wer Lust hat, kann das Ganze gerne mal auf einem Raspberry PI mit echter Wettersensorik ausprobieren. Tatsächlich würde es problemlos laufen, denn JAX-RS 2.1 ist leichtgewichtig genug, um nicht nur auf der Cloud-Seite, sondern auch auf der „Things"-Seite zu laufen, und der Raspberry PI ist zugegebenermaßen nicht gerade schwachbrüstig.

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.



GraphQL als Alternative zu REST

Manuel Mauky, Saxonia Systems AG

GraphQL ist eine Abfragesprache für Web-APIs, mit der Web- und Mobile-Anwendungen Daten vom Server beziehen könnten. Es stellt damit eine Alternative zu REST-Schnittstellen dar, die in diesem Kontext häufig eingesetzt werden. Der Artikel zeigt, wie GraphQL einige der Schwierigkeiten von REST in diesem Einsatz-Szenario löst und darüber hinaus interessante neue Ideen bei der Frontend-Entwicklung ermöglicht.

Der Aufbau und die Architektur vieler Anwendungsarten hat sich in den letzten Jahren in mehrfacher Hinsicht gewandelt. Auf der Backend-Seite findet man heute beispielsweise immer häufiger auf Modularität ausgerichtete Ansätze wie Microservices vor. Auch die Frontend-Seite hat sich gewandelt. Während früher das Rendering größtenteils vom Server übernommen wurde und der Client lediglich einfache Hilfsaufgaben übernommen hat, sind heute komplexe Single-Page-Anwendungen weitverbreitet. Diese Entwicklung ist für viele Anwendungsfälle auch sinnvoll, da damit eben auch komplexere Anwendungen mit höheren Anforderungen an das Interaktionsdesign realisiert werden können.

Eine ähnliche Entwicklung konnte man auf mobilen Geräten beobachten, bei denen heute immer mehr Anwendungen durch eigenständige Apps umgesetzt werden. Hieraus ergibt sich eine klare Aufteilung der Zuständigkeiten zwischen dem Server und den Client-Anwendungen. Der Server stellt Daten bereit und bietet Schnittstellen für fachliche Aktionen. Von der Interaktion mit dem Nutzer hingegen hat der Server keine Kenntnis.

Für Entwickler bleibt die Frage, wie konkret nun die Daten vom Server abgeholt, zum Client transportiert und dort verarbeitet werden können. Die klassische Antwort lautet „REST". Doch obwohl REST mittlerweile omnipräsent ist und auch in sehr vielen anderen Szenarien wie zur Kommunikation zwischen Microservices erfolgreich zum Einsatz kommt, ist es nicht unbedingt immer die beste Wahl. Einer dieser Anwendungsfälle, für die REST zwar gegenwärtig häufig zum Einsatz kommt, aber gleichzeitig für einige Probleme und so manches Kopfzerbrechen sorgt, ist der Datenaustausch zwischen dem Server und Client-Anwendungen. Genau aus diesem Grund und für diesen Einsatzzweck hat Facebook die Abfragesprache „GraphQL" entwickelt, um einige Probleme von REST zu lösen. Dieser Artikel stellt deshalb GraphQL als Alternative zu REST für derartige Szenarien vor.

REST ist Ressourcen-orientiert. Jede Ressource besitzt einen eindeutigen Identifier, den Unified Resource Identifier (URI). Unter diesem können Repräsentationen dieser Ressource, beispielsweise als JSON oder XML, abgeholt werden. Ein weiteres Merkmal von REST ist die Verwendung von Hyperlinks zwischen Ressourcen. Damit lassen sich unter anderem Verbindungen und Abhängigkeiten zwischen Daten ausdrücken.

Als Beispiel dient eine Blog-Anwendung. Diese könnte die Ressourcen „Article", „Author" und „Comment" besitzen. Ein Artikel wurde von einem oder mehreren Autoren verfasst und kann Kommentare enthalten. Als Besucher des Web-Blogs landet man zunächst auf der Artikel-Übersichtsseite. Sie zeigt die letzten zehn Artikel in einer Kurz-Ansicht mit den Namen der Autoren und der Anzahl der Kommentare. Anschließend kann man über einen Klick auf eine Überschrift zu einer Detailseite des jeweiligen Artikels gelangen.

Aus Frontend-Sicht beginnt die Kommunikation mit einem GET-Request auf den Einstiegspunkt des API, woraufhin eine Liste der verfügbaren Ressourcen geliefert wird. Anschließend liefert ein weiterer GET-Request auf die „Article"-Ressource eine Liste von Artikeln. Hierbei stößt man auf die erste Schwierigkeit, denn es sollen ja nur die letzten zehn Artikel abgefragt werden. REST bietet aber kein Konzept für die Sortierung und Begrenzung der gelieferten Daten. Allerdings existieren Best-Practices und Lösungsansätze zur Gestaltung des API, um auch diese Funktionalitäten anbieten zu können.

```
type Article {
  id: ID!
  releaseDate: String
  teaser: String
  text: String
  permalink: String
  authors: [Author]
  comments: [Comment]
}

type Author {
  id: ID!
  name: String!
  articles: [Article]
}

type Comment {
  id: ID!
  text: String!
  author: Author
]
```

Listing 1

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
    authors {
       name
    }
    comments {
       id
    }
  }
}
```

Listing 2

Nun hat das Frontend zwar eine Liste der Artikel, es fehlen jedoch noch bestimmte Informationen zur Anzeige. Da Autoren und Kommentare jeweils eigene Ressourcen darstellen, beinhalten die ausgelieferten Artikel-Daten diese Informationen nicht selbst, sondern verweisen lediglich per Hyperlink auf die verbundenen Ressourcen. Um an die eigentlichen Daten zu gelangen, sind für jeden Artikel in der Liste zwei weitere GET-Requests auf die jeweilige Autoren- beziehungsweise Kommentar-Ressource durchzuführen.

Für dieses einfache Beispiel der Artikel-Übersichtsseite sind also bereits 22 GET-Requests erforderlich. Dieser Umstand kann ein ernstes Performance-Problem darstellen, da die Anzahl notwendiger Requests ein wesentlicher Faktor für die Geschwindigkeit der Anwendung sein kann. Das ist insbesondere der Fall für mobile Anwendungen, die auch in Gegenden mit schlechter Netzabdeckung noch benutzbar sein sollen.

Die Anzahl der Requests ist jedoch nicht das einzige Problem. Ein weiteres ist die Menge der zu übertragenden Daten. Im Beispiel soll ein Artikel ein Feld für den eigentlichen Text und eines für einen kurzen Teaser enthalten. Für die Startseite ist nur der Teaser inte-

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum...",
        "releaseDate": "2017-04-23",
        "authors": [
           {
              "name": "Hugo"
           }
        ],
        "comments": []
      },
      {
        "id": "sdöf234ds2",
        "title": "Other Title",
        "permalink": "other_title",
        "teaser": "Lorem Ipsuns.sd",
        "releaseDate": "2017-03-24",
        "authors": [
           {
              "name": "Luise"
           },
           {
              "name": "Elfriede"
           }
        ],
        "comments": [
            {
              "id": "13424"
            }
        ]
      }
    ]
  }
}
```

Listing 3

ressant, der ausführliche Text soll dagegen erst auf der Detailseite erscheinen. Je nachdem, wie das REST-API gestaltet ist, könnte es allerdings sein, dass auch der ausführliche Text bereits bei dem Request für die Artikel-Liste für jeden dieser Artikel mitgeliefert wird und damit unnötige Daten übertragen werden. Als Entwickler wünscht man sich also auch hier eine Möglichkeit, um das Ergebnis eines REST-Requests zu beeinflussen.

Auch für diese Probleme gibt es Lösungsansätze bei REST. Beispielsweise könnten Query-Parameter mitgeschickt werden, die bestimmen, welche Daten zurückgeliefert werden. Wenn man beispielsweise den Parameter „withFullText=true" an die URL anhängt, könnte der Server entsprechend den vollständigen Text für jeden Artikel mit in die Antwort packen. Eine weitere Option wäre, verschiedene Repräsentationen für ein und dieselbe Ressource bereitzustellen. Bei einem GET-Request könnte dann etwa als ACCEPT-Header nicht mehr nur „application/json" sondern „application/full+json" oder „application/with.comments+json" angegeben werden.

Allerdings lösen beide Ansätze nicht das Problem, dass für eine Ansicht manchmal Daten von mehreren Ressourcen benötigt werden. Um diesem zu begegnen, sieht man deshalb in der Praxis manchmal View-spezifische Ressourcen. Ein GET-Request auf „/api/article_overview" könnte beispielsweise die Kurzfassung der Artikel liefern, genauso wie es für die Übersichtsseite notwendig ist. Hierbei werden allerdings die Ideen von REST ziemlich verwässert. Denn eigentlich ist „article_overview" keine eigenständige, echte Ressource. Vor allem sorgt dieser Ansatz aber für eine deutliche Kopplung zwischen Server und Clients. Ist eine neue Ansicht im Frontend geplant, müssen dafür entsprechende Änderungen am Server vorgenommen werden, selbst dann, wenn sich die eigentlichen Daten gar nicht geändert haben.

Wie man es also dreht und wendet, eine wirklich zufriedenstellende Lösung scheint unmöglich zu sein. Will man ein sauberes REST-API umsetzen, erschwert dies die Benutzung auf Frontend-Seite. Weicht man dagegen die Regeln für REST auf und verzichtet beispielsweise auf saubere Hyperlinks zwischen den Ressourcen, muss man eine stärkere Kopplung zwischen Server und Clients in Kauf nehmen. Selbst dann ist die Benutzung des API auf Frontend-Seite alles andere als optimal, da immer noch überflüssige Daten übertragen werden können.

GraphQL als Alternative

Basierend auf diesen Schwierigkeiten beim Umgang mit REST-Schnittstellen für die Abfrage von Daten und den Transport zum Client, hat Facebook die Abfragesprache GraphQL entwickelt. Sie ist dort seit dem Jahr 2012 intern im Einsatz und wurde Ende 2015 der Allgemeinheit präsentiert und zur Verfügung gestellt. Seitdem hat GraphQL vor allem in der React-Community, aber auch darüber hinaus für großes Interesse gesorgt. Beispielsweise hat GitHub seine neue Version des offiziellen API erst kürzlich auf GraphQL umgestellt *[1]* und verarbeitet darüber laut eigenen Angaben täglich mehr als 125 Millionen Queries.

Doch was ist GraphQL nun genau und wie hilft es bei den oben beschriebenen Problemen? Bei GraphQL stellt der Server ein statisch typisiertes Schema bereit. In diesem Schema ist festgelegt, welche Daten der Server kennt und wie diese Daten miteinander in Beziehung stehen. In *Listing 1* ist beispielhaft ein Ausschnitt aus einem Schema für unseren Blog-Anwendungsfall zu sehen. Es enthält für unsere drei Entitäten „Author", „Article" und „Comment" Typdefinitionen.

GraphQL selbst stellt zunächst einmal nur eine Spezifikation dar. Die im Code-Ausschnitt verwendete Syntax ist die GraphQL Schema Language *[2]*; sie dient vor allem der Beschreibung des Schemas in einer Programmiersprachen-unabhängigen Art und Weise. Wie das Schema konkret implementiert wird, hängt von der verwendeten Programmiersprache ab. Hier macht GraphQL selbst keine Vorgaben, die Macher stellen aber eine Referenz-Implementierung in JavaScript zur Verfügung. Vom Frontend aus können nun Queries gegen dieses Schema geschrieben und zum Server geschickt werden. *Listing 2* zeigt eine solche Beispiel-Query für unsere Artikel-Übersichtsseite.

```
// schema
type Mutation {
    addAuthor(name: String): Author
}

// query
mutation {
    addAuthor("Manuel") {
        id
    }
}
```

Listing 4

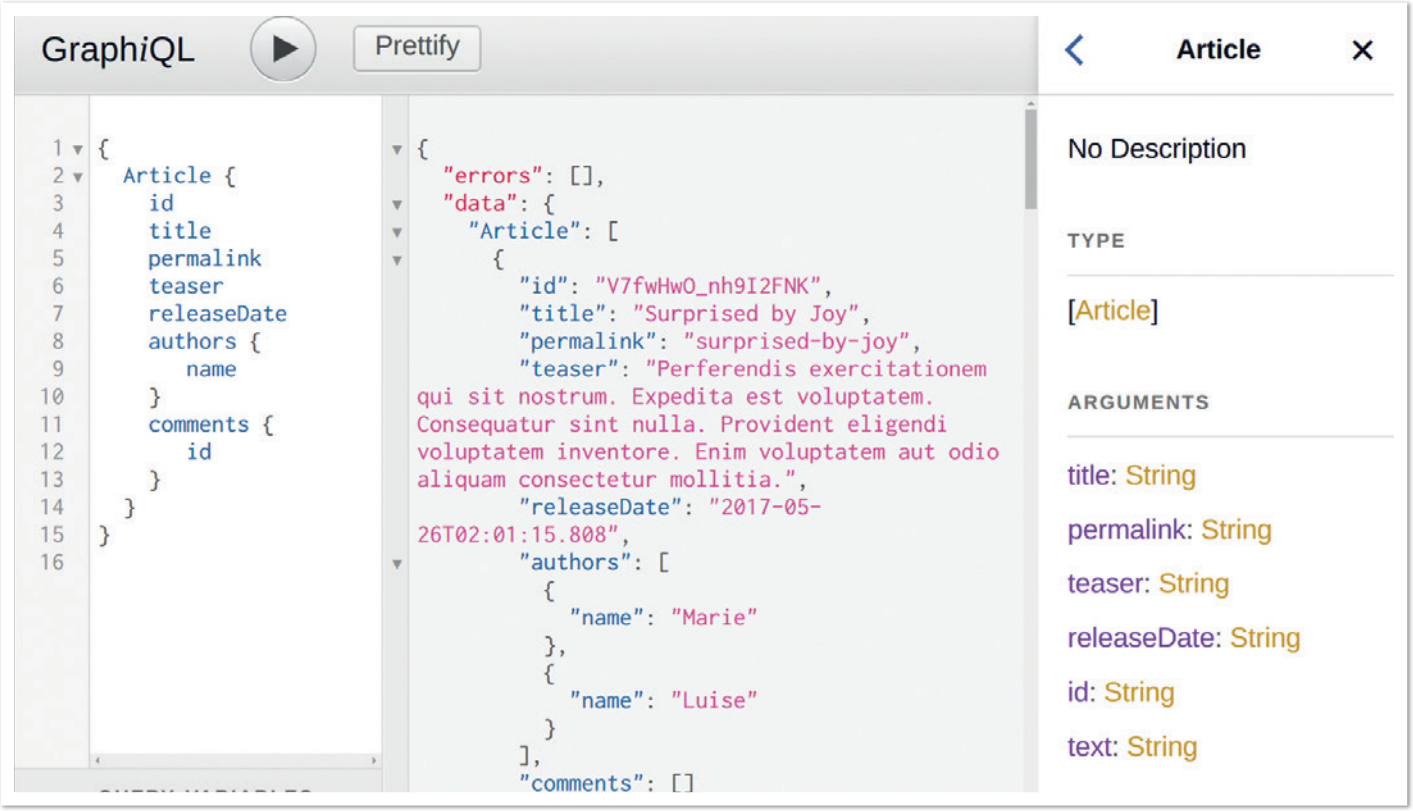


Abbildung 1: Das Tool GraphiQL zum Ausprobieren von APIs

Die Query erinnert in ihrer Struktur ein wenig an JSON. Sie beschreibt genau, welche Daten erforderlich sind. Wir interessieren uns für Artikel. Von diesen interessieren uns aber nur die aufgeführten Attribute. Außerdem können Abfragen beliebig geschachtelt werden. Dies machen wir uns für die Autoren des Artikels zunutze, da wir von den Autoren für die Übersichtsseite lediglich die Namen benötigen. Auch hier können wir also wieder genau bestimmen, welche Attribute von Interesse sind und welche nicht. *Listing 3* zeigt eine Antwort, wenn man diese Query zum Server schickt.

Die Antwort liegt als JSON vor und entspricht in ihrer Struktur genau der gestellten Anfrage. Nutzer des API können also exakt bestimmen, welche der angebotenen Daten sie interessieren und welche nicht. Wie der Server die Daten für die jeweilige Query beschafft, hängt von der Implementierung ab. Auf unterster Ebene bietet GraphQL Einstiegspunkte auf jeder Verschachtelungsebene des Schemas. Bei der Verarbeitung der Query aus *Listing 2* entspricht beispielsweise „Article" letztlich einem Funktionsaufruf, mit dem alle Artikel aus der Datenbank oder aus einer sonstigen Quelle geladen werden müssen. Auf der nächsten Ebene entsprechen wiederum alle Felder, also „id", „title" etc., letztlich Funktionsaufrufen, die für einen gegebenen Artikel einen konkreten Wert liefern müssen. Für skalare Typen wird hier standardmäßig einfach ein „Getter" aufgerufen, es können jedoch auch andere Implementierungen gewählt werden.

Die Varianten, um ein GraphQL-API zu implementieren, sind vielfältig. Für Java-Entwickler existiert zum Beispiel eine von der Community entwickelte Bibliothek *[3]*, um aus vorhandenen JPA-Entitäten automatisch ein passendes GraphQL-Schema zu generieren. Dabei wird auch das Beschaffen der Daten über den EntityManager von JPA automatisch von der Bibliothek übernommen.

Mutationen

Bisher wurde nur die Abfrage von Daten betrachtet. REST stellt aber auch Möglichkeiten für deren Manipulation und Veränderung bereit. Dazu werden bei REST die wohldefinierten HTTP-Verben verwendet. Ein POST-Request auf eine Ressource sorgt beispielsweise für das Anlegen einer neuen Entität, ein PUT-Request für das Editieren und ein DELETE-Request für das Löschen einer Ressource, wobei das REST-API definiert, welche Operationen für welche Ressourcen erlaubt sind.

Auch GraphQL erlaubt verändernde Operationen, jedoch unterscheidet sich der Ansatz ebenfalls fundamental von REST. Die möglichen Operationen sind bei REST durch die vorhandenen Verben vorgegeben und fest mit den Ressourcen verknüpft. Bei GraphQL werden dagegen sogenannte „Mutationen" im Schema definiert, die allerdings vollkommen losgelöst von der Abfrageseite sind.

Mutationen sind im Prinzip als ausführbare fachliche Funktionen zu verstehen, die Daten übergeben bekommen und Daten zurückliefern können, wobei üblicherweise die veränderten Daten selbst als Rückgabewerte angeboten werden. In *Listing 4* ist eine Mutation „addAuthor" definiert, die zum Anlegen von neuen Autoren benutzt werden kann. Zusätzlich zum übergebenen Namen lässt sich eine GraphQL-Query angeben, die im Beispiel die ID des neu angelegten Autors selektiert.

Ohne es explizit zu erwähnen, setzt die GraphQL-Spezifikation also im Prinzip das „Command-Query-Responsibility-Segregation"-Pattern (CQRS) um. Es wird strikt zwischen Abfrage und Veränderung der Daten getrennt. Auf diese Weise lassen sich sehr einfach auch fachliche Intentionen im API ausdrücken, indem zum Beispiel Mutationen wie „editText" und „changeAuthor" angeboten werden.

Obwohl beide letztlich den Artikel verändern, stecken doch unterschiedliche Absichten dahinter, die vom Server auch unterschiedlich behandelt werden könnten.

Statische Typisierung und Introspektion

GraphQL erlaubt nicht nur das Abfragen von konkreten Daten, sondern auch von Informationen über das Schema selbst. Diese als „Introspektion" bezeichnete Funktion sowie die Tatsache, dass GraphQL statisch typisiert ist, ermöglichen interessante und mächtige Entwicklertools zur Unterstützung der Programmierer. Das beste Beispiel dafür ist das Abfrage-Tool „GraphiQL" – man beachte das kleine „i" im Namen *[4] (siehe Abbildung 1)*. Es ermöglicht nicht nur das dynamische Ausführen von Queries gegen ein Schema, sondern unterstützt auch durch Autovervollständigung, automatische Hinweise beim Schreiben von Queries sowie durch eine automatisch bereitgestellte Dokumentation des Schemas. Auch weitere Anwendungsfälle sind denkbar. So könnte man beispielsweise bereits zum Entwicklungszeitpunkt durch ein in den Build-Prozess integriertes Tool prüfen, ob die im Client verwendeten Queries auch wirklich zum Schema des Servers passen, und bei Fehlschlägen frühzeitig Warnungen ausgeben *[5]*.

Aber auch die andere Richtung ist möglich: Ein Tool im Build-Prozess könnte prüfen, ob eine Änderung im Server-Code versehentlich zu einer inkompatiblen Änderung des API führt, indem alle vorhandenen Queries aller Client-Anwendungen automatisch gegenüber dem neuen API geprüft werden *[6]*.

Neue Ideen und Patterns im Frontend

Neben der Möglichkeit zur Vermeidung von unnötigen Requests durch zielgenau formulierte Queries ermöglicht GraphQL auch einige neue Ideen und Patterns bei der Frontend-Entwicklung. Diese resultieren nicht nur in einer weiteren Optimierung der zu übertragenden Daten, sondern ermöglichen darüber hinaus auch eine bessere Entkopplung von UI-Komponenten.

Dazu nochmal ein Blick auf das Blog-Beispiel. Für die Übersichtsseite sind einige, aber nicht alle Daten der Artikel erforderlich und man formuliert eine entsprechende GraphQL-Query. Bei einem Klick des Nutzers auf einen konkreten Artikel sollen zu einer Detail-Seite navigiert und dazu weitere Daten zu diesem konkreten Artikel nachgeladen werden, was wiederum durch eine entsprechende GraphQL-Query ausgedrückt wird. Die Menge der benötigten Daten überschneidet sich jedoch bei beiden Seiten. Informationen wie der Titel, der Teaser und die Namen der Autoren werden auf beiden Seiten benötigt. Andere Daten, vor allem der ausführliche Artikeltext, sind nur für die Detail-Seite relevant.

Beim Betreten der Detail-Seite liegen also bestimmte Daten bereits vor und müssen nicht erneut vom Server geladen werden. Als Entwickler könnte man diese Überlegung also nutzen, die Query für die Detailseite entsprechend kürzen und Daten von der Übersichtsseite wiederverwenden. Dieser naive Ansatz birgt allerdings einige Nachteile: Die Detailseite besitzt dann schließlich eine direkte Abhängigkeit zur Übersichtsseite. Ändert sich später die Darstellung auf der Übersichtsseite, könnten plötzlich bestimmte Informationen nicht mehr zur Verfügung stehen. Außerdem ist die Detailseite auch direkt über eine URL ansprechbar. In diesem Fall war die Übersichtsseite nicht vorher aktiv und hat noch keine Daten geladen. Ein händisches Optimieren nach diesem Muster ist also kompliziert und fehleranfällig.

Jedoch könnte ein entsprechend intelligentes Framework diese Optimierung im Hintergrund selbstständig übernehmen. Entwickler schreiben dann für jede Komponente eine komplette GraphQL-Query mit sämtlichen benötigten Daten und sind damit vollständig entkoppelt von anderen Komponenten. Erst das Framework übernimmt die Optimierung der Queries und setzt darüber hinaus ein Caching im Client um. Auf diese Weise müssen bestimmte Queries unter Umständen überhaupt nicht mehr neu ausgeführt werden, wenn beispielsweise später zu einer Seite zurücknavigiert wird. Der Umfang der zu übertragenden Daten verringert sich dadurch noch weiter, vor allem aber wird ein vergleichsweise einfaches Programmiermodell mit sehr starker Entkopplung ermöglicht: Entwickler einer Komponente definieren einfach ihre Datenanforderung mittels GraphQL, ohne sich Gedanken über das konkrete Laden der Daten und Aspekte wie Caching Gedanken machen zu müssen. Ein weiterer Vorteil dieser als „Query Co-Location" bezeichneten Methode ist, dass so auch verschachtelte Komponenten besser voneinander entkoppelt werden können.

Die Detail-Seite könnte beispielsweise aus einer Titel-Komponente, die den Titel des Artikels sowie die Informationen zum Autor darstellt, einer Haupt-Komponente, die den vollständigen Text des Artikels anzeigt, und einer Kommentar-Komponente zusammengesetzt sein. Jede dieser Unter-Komponenten definiert ihre jeweils benötigten Datenanforderungen mittels GraphQL. Es wäre allerdings eine schlechte Idee, wenn jede Komponente auch selbst dafür verantwortlich wäre, ihre Query auch tatsächlich auszuführen.

Stattdessen könnte jede Komponente ihre Query an die jeweilige Eltern-Komponente weiterreichen. Diese könnte die Queries der Kind-Komponenten kombinieren und ihrerseits wieder an die eigene Eltern-Komponente übergeben. Irgendwann, an der Wurzel des Komponentenbaums angelangt, müsste die vollständige Query einmalig ausgeführt und die Ergebnis-Daten wiederum auf gleichem Weg zurück in die Komponenten durchgereicht werden.

Dieser Mechanismus sorgt für eine weitere Entkopplung der UI-Komponenten, denn Eltern-Komponenten haben keine konkrete statische Abhängigkeit mehr zu den Datenanforderungen ihrer Kind-Komponenten. Sie sammeln lediglich die Queries ein, ohne Kenntnis davon, was in der Query jeweils steht. Benötigt eine Kind-Komponente beispielsweise wegen eines neuen Features neue Daten, sind in der Eltern-Komponente keinerlei Anpassungen im Code notwendig.

Interessant ist dabei auch, dass beim Kombinieren der einzelnen Queries durchaus auch Dopplungen enthalten sein können. Mehrere Komponenten könnten die gleichen Daten benötigen. Auch hier könnte ein intelligentes Framework also Optimierungen durchführen. Tatsächlich existieren mit „Relay" *[7]* und „Apollo Client" *[8]* mindestens zwei JavaScript-Frameworks, die solche Ansätze unterstützen. Relay wurde von Facebook selbst als umfassendes Framework für die Kombination von GraphQL mit React.js speziell für datengetriebene Anwendungen entwickelt. Apollo Client dagegen ist ein etwas leichtgewichtigerer Ansatz, der neben React.js auch andere Plattformen wie Angular, iOS und Android unterstützt.

Nachteile von GraphQL und Vorteile von REST

Natürlich gibt es auch bei GraphQL nicht nur Vorteile, sondern auch Nachteile und Einschränkungen. Und auch REST hat nach wie vor seine Berechtigung und in vielen Situationen Vorteile gegenüber GraphQL. Da wären zum einen ganz praktische Aspekte, wie die weite Verbreitung von REST, die damit verbundene Akzeptanz und die Verfügbarkeit von Wissen und Werkzeugen. Ein Beispiel ist das Spring-Data-REST-Projekt, mit dem Java-Entwickler extrem einfach REST-APIs aufsetzen können. Im GraphQL-Umfeld existieren zwar auch erste Projekte, die etwa eine Integration mit Spring-Boot und JPA bieten, aber bezüglich Ausgereiftheit, Stabilität und Dokumentation können diese Community-Ansätze noch nicht mithalten.

REST hat jedoch auch inhärente Vorteile, vor allem dann, wenn der Hypermedia-Gedanke konsequent verfolgt wird. So können beispielsweise mittels Hyperlinks ohne Weiteres auch Verbindungen zu ansonsten unabhängigen Ressourcen auch auf fremden APIs hergestellt werden, was mit GraphQL aufgrund des Fokus auf ein einziges Schema nicht so einfach möglich ist. Dies ist besonders im Microservices-Umfeld interessant, wo häufig ein API-Gateway einkommende Requests auf unterschiedliche Microservices umleitet und dieser Prozess auch anhand der Hyperlinks gesteuert werden kann.

Da die Beschaffung der Daten auch bei einem GraphQL-Schema letztlich frei implementiert werden kann, sind ähnliche Weiterleitungen auch mit GraphQL umsetzbar, allerdings ist der nötige Aufwand etwas höher. Ein weiterer Aspekt ist die Steuerung von Clients von der Serverseite aus durch intelligent gesetzte Hyperlinks. Dadurch kann das Verhalten der Clients manipuliert werden, ohne dass Anpassungen an der eigentlichen Software der Clients notwendig sind.

Ein Nachteil bei GraphQL ist, dass Requests nicht ohne Weiteres auf der Netzwerk-Ebene gecacht werden können. In vielen Fällen dürfte das allerdings wenig problematisch sein, da über das GraphQL-API in der Regel ja vor allem dynamische Daten abgefragt werden und diese im Vergleich zu statischen Inhalten wie HTML-, JavaScript- und Bild-Dateien, die weiterhin ganz normal cachbar sind, häufig einen geringen Umfang haben dürften.

Ein weiterer Nachteil ist, dass die Größe der Requests selbst, also der Daten, die vom Client zum Server geschickt werden müssen, in der Regel größer ist. Auch hier sind ebenfalls Optimierungen denkbar, die beispielsweise mit der aktuellen Version des oben erwähnten Relay-Frameworks eingeführt wurden, diese sind aber Framework-spezifisch.

Fazit

Als Fazit lässt sich sagen, dass GraphQL eine wirklich spannende Technologie mit interessanten Ideen ist und in einigen Anwendungsfällen eine echte Alternative zu REST-Schnittstellen darstellt. REST wird dadurch aber nicht obsolet, sondern behält seine Berechtigung durch seine Flexibilität und eine Vielzahl von Einsatzmöglichkeiten weiterhin bei.

Vor allem dann, wenn die Menge der zu übertragenden Daten und die Anzahl der Requests kritisch ist, beispielsweise bei Consumer-Web-Apps und mobilen Anwendungen, kann GraphQL seine Stärken ausspielen. Die Möglichkeiten, die durch das statisch typisierte Schema entstehen, beispielsweise mächtige Entwickler-Tools, sind vielversprechend. Auch die starke Entkopplung von Komponenten, die durch GraphQL ermöglicht bzw. vereinfacht wird, verspricht einen Mehrwert bezüglich Wiederverwendbarkeit und Wartbarkeit. Es wird spannend sein zu sehen, welche weiteren Entwicklungen und Ideen wir in dieser Richtung in Zukunft erwarten können.

Quellen

[1] *https://developer.github.com/v4*
[2] *https://github.com/sogko/graphql-schema-language-cheat-sheet*
[3] *https://github.com/jcrygier/graphql-jpa*
[4] *https://github.com/graphql/graphiql*
[5] *https://github.com/apollographql/eslint-plugin-graphql*
[6] *https://github.com/creditkarma/graphql-validator*
[7] *https://facebook.github.io/relay*
[8] *http://dev.apollodata.com*

Manuel Mauky
manuel.mauky@saxsys.de

Manuel Mauky arbeitet seit dem Jahr 2010 als Software-Entwickler bei der Saxonia Systems AG in Görlitz. Er beschäftigt sich mit allen Aspekten der Anwendungsentwicklung mit einem Fokus auf das Frontend, zum Beispiel mit JavaFX und JavaScript. Außerdem interessiert sich für funktionale Programmierung und neue Programmiersprachen. Er ist einer der Organisatoren der Görlitzer Java User Group und steuert dafür und auch für andere User Groups und Konferenzen regelmäßig Vorträge bei.

Goodbye CRUD, hello Immutability: der Umgang mit Daten in Clojure

Michael Sperber, Active Group GmbH

In der rein funktionalen Programmierung sind Daten unveränderlich – also kein gekapselter Zustand, keine Setter, kein CRUD-Pattern. Dieses Paradigma setzt Clojure auf der Java-Plattform konsequent um. Auf den ersten Blick scheint es so, als bedeute der Verzicht auf Veränderung primär eine Einschränkung. Tatsächlich aber befreit gerade dieser Verzicht die Programmierung von vielen Problemen, schafft neue Möglichkeiten und eröffnet eine andere Sicht auf die Datenmodellierung.

Nahezu jedes Standardbeispiel für die objektorientierte Programmierung setzt auf zwei Ideen:

- Ein Objekt repräsentiert eine Entität aus der Problemstellung
- Ein Objekt kapselt unveränderlichen Zustand

In einem Telefonbuch beispielsweise könnten „Personen" gespeichert sein. Entsprechend diesen Ideen gibt es eine Klasse, die in Java anfangen könnte wie in *Listing 1*. Der Bezug zwischen der Idee „Person" aus der Problemstellung und den Attributen der Klassen-Definition ist klar ersichtlich. Außerdem werden die Attribute durch Veränderung manipuliert: Es gibt einen Setter für „firstName", wahrscheinlich auch noch einen für „lastName". An „phones" wird zwar nicht zugewiesen, stattdessen sorgt die Methode „put" unmittelbar dort für eine Veränderung. So ist die Klasse allerdings noch nicht fertig; sie sollte noch eine sinnvolle Definition für „equals" enthalten *(siehe Listing 2)*.

Diese Definition offenbart zwei Probleme:

- Sie ist umständlich, weil die Klasse das CRUD-Pattern benutzt und es damit sein kann, dass „firstName" und „lastName" jeweils „null" sein können.
- Sie ist komplett mechanisch – wahrscheinlich wurde sie von einer IDE generiert. Warum muss sie überhaupt programmiert werden?

Außerdem fehlen noch Definitionen für „hashCode" und „toString", die beide ähnliche Symptome aufweisen. Die Definition weist zudem auf ein anderes Problem hin: Sie implementiert die Idee, dass zwei Personen gleich sind, wenn sie gleiche Namen und gleiche Telefonnummern haben *(siehe Listing 3)*.

Der Vergleich „p1.equals(p2)" liefert „true". Was ist aber nach der Anweisung „p1.setLastName("Ferber");"? Vielleicht hat „p1" geheiratet? Plötzlich sind „p1" und „p2" verschieden: „p2" hat nicht geheiratet; „p1" und „p2" sind nicht dieselbe Person. Ob „das Gleiche" oder „dasselbe" die richtige Grundlage für die Implementierung von „equals" ist, ist Ermessensfrage und manchmal gar nicht so einfach zu entscheiden.

Joshua Blochs Klassiker „Effective Java“ *[1]* hat dementsprechend eine klare Empfehlung, nämlich sogenannte „value objects“ zu verwenden, die unveränderlich („immutable“) sind: „Classes should be immutable unless there's a very good reason to make them mutable … If a class cannot be made immutable, limit its mutability as much as possible.“ Dieser Ratschlag ist in der Java-Welt aus einer Reihe von pragmatischen Gründen weithin akzeptiert:

- „Value objects“ sind problemlos als Schlüssel in Maps und anderen Datenstrukturen verwendbar
- „Value objects“ sind „thread“-sicher
- „Value objects“ benötigen keinen Copy-Konstruktor und auch keine „clone“-Methode

Warum Schlüssel in Maps? Bei Hash-Maps beispielsweise ändert sich der Hash-Code, wenn der Schlüssel verändert wird – dieser steht dann plötzlich nicht mehr an der richtigen Stelle. Dies sind alles gute Gründe. Sie verstellen jedoch den Blick auf die fundamentale Ursache der Probleme, die durch veränderliche Objekte verursacht werden; die Welt funktioniert nämlich nicht so, wie die objektorientierte Programmierung sie mit veränderlichen Objekten modellieren möchte.

Dies erscheint erst einmal nicht intuitiv. In der realen Welt ändert sich alles ständig – die Welt und die Objekte in ihr haben einen Zustand. Allerdings suggeriert das objektorientierte Modell des gekapselten Zustands, dass jedes Objekt seinen eigenen isolierten Zustand hat. Hier ein einfaches Beispiel: Ein Elefant geht von einem Zimmer in den Flur. Ein typisches objektorientiertes Programm macht in dem Beispiel drei Entitäten aus (Elefant, Zimmer, Flur) und aus dem Vorgang möglicherweise diese Folge von Anweisungen: „room1.exit(elephant)“ und „hallway.enter(elephant)“.

Das sind also zwei Schritte; in der Realität braucht der Elefant allerdings nur einen Schritt für beides: Der Schritt aus dem Zimmer ist derselbe Schritt wie der Schritt in den Flur. Das mag als Spitzfindigkeit erscheinen, ist es jedoch spätestens dann nicht mehr, wenn das Programm nebenläufig wird. Das muss es, wenn es die hochgradig nebenläufige Realität modellieren möchte. Dann können andere Threads den Zustand zwischen den beiden Anweisungen beobachten, und da ist der Elefant kurz mal nirgendwo: Der Zustand ist inkonsistent. Es ist möglich, das objektorientierte Modell zu ändern, aber das fundamentale Problem bleibt.

Dass also die reale Welt nicht mit gekapseltem Zustand funktioniert, ist eine Ursache des Problems. Noch fundamentaler ist der Umstand, dass Computerprogramme von Menschen geschrieben werden. Die Objekte in einem Computerprogramm repräsentieren deshalb nicht die Welt, sondern unsere Wahrnehmung der Welt. Dies ist eine der Motivationen für Domain-driven Design.

Auch dazu ein Beispiel: Bei einem Bundesligaspiel gibt es enorm viele Entitäten: 22 Spieler/innen, Schiedsrichter, den Ball, Tausende von Zuschauern. Ein Zuschauer nimmt den Zustand des Spiels immer konsistent wahr; in der visuellen Wahrnehmung befinden sich niemals zwei Personen gleichzeitig am selben Ort oder verschwinden kurz, um dann woanders wieder aufzutauchen. Außerdem funktioniert der Prozess der Wahrnehmung selbst so, dass wir konsistente Schnappschüsse aufzeichnen, die das Gehirn dann analysiert.

```
public class Person {
    private String firstName;
    private String lastName;
    private Map<String, String> phones =
      new HashMap<String, String>();
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void newPhone(String key, String to) {
        this.phones.put(key, to);
    }
   ...
}
```

Listing 1

```
@Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof Person))
            return false;

        Person person = (Person) o;

        if (firstName != null
            ? !firstName.equals(person.firstName)
            : person.firstName != null)
            return false;
        if (lastName != null
            ? !lastName.equals(person.lastName)
            : person.lastName != null)
            return false;
        if (!phones.equals(person.phones))
            return false;
        return true;
    }
```

Listing 2

```
Person p1 = new Person();
p1.setFirstName("Mike");
p1.setLastName("Sperber");
p1.addPhone(new Phone("555"));
Person p2 = new Person();
p2.setFirstName("Mike");
p2.setLastName("Sperber");
p2.newPhone("work", "(123) 555-1234"));
```

Listing 3

```
(ns phonebook
  (:require [active.clojure.record :refer :all]))
```

Listing 4

```
(define-record-type Person
  (make-person first last phones)
  person?
  [first person-get-first
   last person-get-last
   phones person-get-phones])
```

Listing 5

Dies steht in Gegensatz zum Standardmodell der objektorientierten Programmierung für die Wahrnehmung von Ereignissen, dem Observer-Pattern: Ein Zuschauer schickt nicht jedem Objekt im Stadion eine Nachricht, die darum bittet, über Zustandsänderungen informiert zu werden. Außerdem sind die Schnappschüsse häufig noch präsent, wenn die Welt sich weiterbewegt hat. Der Mensch hat ein Gedächtnis und es ist oft enorm nützlich, sich an die Vergangenheit zu erinnern.

Das Problem mit veränderlichen Daten war schon früh bekannt. Alan Kay, der den Begriff „objektorientiert" geprägt hat, schrieb im Jahr 1993 *[2]*: „Though OOP came from many motivations, two were central. ... [T]he small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether." Irgendwie ist dieses Ziel in der Weiterentwicklung von OOP seit dieser Zeit abhandengekommen.

Daten in Clojure

Der vorige Abschnitt hat gezeigt, warum es sinnvoll ist, auf Veränderlichkeit von Daten zu verzichten. Wie machen die funktionalen Programmierer das, ohne ständig das Gefühl zu haben, dass ihnen etwas Wichtiges fehlt? Die folgenden Code-Beispiele verwenden die Library Active Clojure *[3]*. An den Anfang gehört folgende Präambel für die entsprechenden Importe *(siehe Listing 4)*. Das „ns" steht für „namespace", den Clojure-Namen für „package". *Listing 5* zeigt das Clojure-Pendant zur „Person"-Klassendefinition von oben.

„Record" ist das Clojure-Pendant zu Java-„Pojos" – zusammengesetzten Objekten, deren Bestandteile Namen haben. Clojure hat auch noch ein anderes eingebautes Konstrukt zur Deklaration von Records („defrecord), das sich aber nicht so gut für Erläuterungen eignet. Diese Deklaration definiert den Typ für Personen, außerdem einen Konstruktor „make-person" sowie „Getter"-Funktionen für die Felder „first", „last" und „phones" namens „person-get-first", „person-get-last" und „person-get-phones". Benutzen lässt sich das wie in *Listing 6*.

Hier sind zwei Variablen „mike-sperber" und „sabine-ferber" deklariert – der Konstruktor „make-person" akzeptiert für jedes Feld ein Argument. Die Syntax „{...}" erzeugt eine Map, in dem Fall mit den Schlüsseln „:home" und „:work". Dabei handelt es sich um sogenannte „Keywords", ein spezielles Clojure-Konstrukt, das besonders schnellen Map-Zugriff erlaubt. Das Definieren von „equals" und „hashCode" (oder gar eines Copy-Konstruktors oder „clone") ist unnötig, da Records unveränderlich sind. Die Definitionen werden von Clojure automatisch im Hintergrund erzeugt. Die Getter sind ganz normale Funktionen *(siehe Listing 7)*.

Angenommen, die beiden heiraten. Das Verändern der Felder ist nicht möglich. Stattdessen muss ein neues „Person"-Objekt erzeugt werden. Die Funktion in *Listing 8* erledigt das. Die Funktion „marry" akzeptiert Argumente für die zwei Parameter „person-1" sowie „person-2" und übernimmt (etwas willkürlich) für das neue Objekt den Vornamen von „person-1", den Nachnamen von „person-2" und die Telefonnummern von „person-1". Clojure druckt das Ergebnis wie in *Listing 9* aus. Das Beispiel zeigt, dass ein „Person"-Objekt nicht „eine Person" repräsentiert, sondern den Zustand (beziehungsweise einen Schnappschuss des Zustands) zu einem bestimmten Zeitpunkt. Tritt ein neuer Zustand ein, muss ein neues

```
(def mike-sperber
  (make-person "Mike" "Sperber"
               {:home "(123) 555-1234"
                :work "(321) 555-4321"}))

(def sabine-ferber
  (make-person "Sabine" "Ferber"
               {:home "(123) 555-1234"
                :work "(444) 555-4444"}))
```

Listing 6

```
(person-first mike-sperber) => "Mike"
(person-last sabine-ferber) => "Ferber"
```

Listing 7

```
(defn marry [person-1 person-2]
  (make-person (person-get-first person-1)
               (person-get-last person-2)
               (person-get-phones person-1)))
```

Listing 8

```
(marry mike-sperber sabine-ferber) =>
#crud.phone_book.Person{
  :first "Mike"
  :last "Ferber"
  :phones {:home "(123) 555-1234" :work "(321) 555-4321"}}
```

Listing 9

```
(assoc {:home "(123) 555-1234" :work "(321) 555-4321"}
       :home "(123) 555-2345")
=> {:home "(123) 555-2345", :work "(321) 555-4321"}
```

Listing 10

```
(defn new-phone [person key to]
  (make-person (person-get-first person)
               (person-get-last person)
               (assoc (person-get-phones person) key to)))
```

Listing 11

Objekt her. Angenommen, eine Person bekommt eine neue Telefonnummer. Dazu muss aus der Map mit den schon vorhandenen Nummern eine neue Map gemacht werden (Verändern geht nicht), in der alle Einträge aus der alten Map sowie ein weiterer stehen. In Clojure erledigt das die Funktion „assoc". *Listing 10* zeigt ein Beispiel. Eine Funktion, die das auf „Person"-Objekten erledigt, muss die Map mit den Telefonnummern erst extrahieren, eine neue Map und damit ein neues Objekt erzeugen *(siehe Listing 11)*.

Linsen zur Daten-Manipulation

Die Funktion „new-phone" sieht etwas umständlich aus: Im Java-Programm steht einfach „person.phones.put(key, to) ". Geht das nicht einfacher? Es geht, und zwar mit einem Konzept namens „Linsen"

```
(ns phonebook
  (:require [active.clojure.record :refer :all]
            [active.clojure.lens :as lens]))
```

Listing 12

```
(define-record-type Person
  (make-person first last phones)
  person?
  [(first person-get-first person-first)
   (last person-get-last person-last)
   (phones person-get-phones person-phones)])
```

Listing 13

```
(lens/yank mike-sperber person-first) => "Mike"
(lens/shove mike-sperber person-last "Müller")
=> #phone_book.Person{
     :first "Mike"
     :last "Müller"
     :phones {:home "(123) 555-1234" :work "(321) 555-
4321"}}
```

Listing 14

```
(defn marry [person-1 person-2]
  (lens/shove person-1 person-last
              (lens/yank person-2 person-last)))
```

Listing 15

```
(defn new-phone [person key to]
  (lens/overhaul person
                 person-phones
                 (fn [old]
                   (assoc old key to))))
```

Listing 16

```
(defn new-phone [person key to]
  (lens/shove person
              (lens/>> person-phones key)
              to))
```

Listing 17

(lenses). Es erlaubt, bei zusammengesetzten Daten auf einen bestimmten Bestandteil der Daten zu fokussieren und sich nur um den zu kümmern. Eine Bibliothek für Linsen ist bei Active Clojure dabei. Die Präambel zum Code muss sie noch einbinden *(siehe Listing 12)*.

Das „:as lens" bedeutet, dass die Linsen-Funktionen unter dem Präfix „lens/" verfügbar sind. Für „Person"-Objekte sind Linsen für alle drei Felder erforderlich. Dazu wird die Record-Definition etwas aufgebohrt *(siehe Listing 13)*. Um jedes Feld herum sind also Klammern; an letzter Stelle steht jeweils der Name der Linse. Eine Linse kann sowohl dafür benutzt werden, den fokussierten Teil der Daten zu extrahieren, als auch dafür, ein neues Objekt zu erzeugen, das beim fokussierten Teil einen anderen Wert hat. Das erledigen die Linsen-Funktionen „lens/yank" und „lens/shove" *(siehe Listing 14)*. Mithilfe dieser Funktionen lässt sich „marry" kompakter als vorher schreiben *(siehe Listing 15)*.

Bei „new-phone" ist es nützlich, eine weitere Linsen-Funktion namens „lens/overhaul" zu benutzen. Sie erlaubt es, den fokussierten Teil zu ändern, also den neuen Wert abhängig vom alten zu bestimmen. Die Funktion akzeptiert eine Funktion, die den alten Wert übergeben bekommt und den neuen produziert *(siehe Listing 16)*.

Aber es geht noch besser: Der Zugriff auf einen Eintrag einer Map ist auch eine Art Fokus: Keywords fungieren auch als Linsen. Damit ist es möglich, „(assoc old key to)" durch „(lens/shove old key to)" zu ersetzen. Das allein bringt noch keine Verbesserung. Allerdings ist es möglich, Linsen zu komponieren – auf den fokussierten Teil eines Objekts kann wiederum ein Teil fokussiert werden. Die Linsen-Funktion „lens/>>" erledigt das – sie macht aus zwei (oder mehr) Linsen eine. Die Linse „(lens/>> person-phones key)" fokussiert also auf einen Eintrag innerhalb der Telefonnummern einer Person. Damit lässt sich nun „new-phone" kompakt und intuitiv schreiben *(siehe Listing 17)*. Linsen können noch eine Menge mehr; wer nach „lenses functional programming" im Internet sucht, findet es.

Fazit

Unveränderliche Daten zu benutzen, ist mehr als nur eine gute Idee: Sie eröffnen eine neue Sichtweise auf die Datenmodellierung, die auf konsistenten Schnappschüssen statt auf gekapseltem Zustand aufbaut. Sie sind eine konsequente Fortführung der ursprünglichen Idee der objektorientierten Programmierung und werden von funktionalen Sprachen besonders gut unterstützt – auch auf der Java-Plattform.

Weitere Informationen

[1] Joshua Bloch, Effective Java, 3rd edition, Addison-Wesley, 2017

[2] Alan Kay, The early history of Smalltalk, Proceedings HOPL-II The second ACM SIGPLAN conference on History of programming languages, ACM, 1993 Seiten 69 –95: *http://worrydream.com/EarlyHistoryOfSmalltalk*

[3] Active Clojure: *https://github.com/active-group/active-clojure*

Michael Sperber

michael.sperber@active-group.de

Dr. Michael Sperber ist Geschäftsführer der Active Group GmbH in Tübingen, die Individualsoftware entwickelt. Er ist seit mehr als zwanzig Jahren in Forschung über und industrieller Anwendung von funktionaler Programmierung tätig. Er hat zahlreiche Fachartikel zum Thema verfasst, Anfänger-Ausbildungen in Programmierung an den Universitäten Tübingen, Freiburg und Kiel konzipiert und (in Tübingen) mehrfach durchgeführt. Michael Sperber gehört zu den Mitinitiatoren und Autoren des Blogs „funktionale-programmierung.de" und der Entwicklerkonferenz BOB.

Von Java Swing über JavaFX nach RISC-HTML

Björn Müller, CaptainCasa

Wir – eine Community europäischer Software-Hersteller von Geschäftsanwendungen – waren lange Zeit ganz gut mit Java Swing gefahren. Aber der Druck zu einer Modernisierung wurde zunehmend stärker. Ein Umstieg in Richtung JavaFX war zwar technologisch interessant, doch so recht brachte uns dieser auch nicht weiter … Dann kam uns eine Idee, die für uns zum „Game Changer" wurde und uns den Weg in den Browser bahnte: RISC-HTML.

Unser Swing-Client ist im Jahr 2007 entstanden. Eine Zeit, in der die einen uns rieten, ActionScript mit dem Macromedia/Adobe Player zu verwenden. Die anderen rieten uns, auf HTML-Frameworks zu setzen. Wir entschieden uns für Swing – und dies aus guten Gründen. Swing war (und ist es auch noch …) „rock-solid", performant und die Existenz langfristig gesichert.

Geschäftsanwendungen haben in ihren Kernteilen sehr lange Lebenszyklen von teilweise weit mehr als Jahren und werden in der Regel von Teams überschaubarer Größe entwickelt und gewartet. Die Lust und Zeit, die Dialoge solcher Anwendungen alle paar Jahre mit anderen Technologien neu zu entwickeln, ist begrenzt. Dies gilt genauso für die Lust und Zeit, sich mit Problemen wie Browser-Kompatibilität herumzuschlagen. Deswegen liegt der größte Fokus – neben der Verkaufbarkeit und Attraktivität der Anwendung – auf einer langfristigen Stabilität *(siehe Abbildung 1)*.

So ganz direkt wurde Swing nicht auf die Anwendungsentwicklung losgelassen: Unser Swing-Frontend lief im Client des Benutzers und bekam vom Server das Layout als XML-Beschreibung der Dialoge zugesendet. Diese Dialogbeschreibungen wurden also vom Swing-Client ausgerendert. Eingaben des Benutzers wurden zu passenden Zeitpunkten an den Server übermittelt und dort in die Java-Anwendung eingearbeitet, worauf der Server dann ein Dialogupdate zum Client zurücksendete. Der Swing-Client war somit

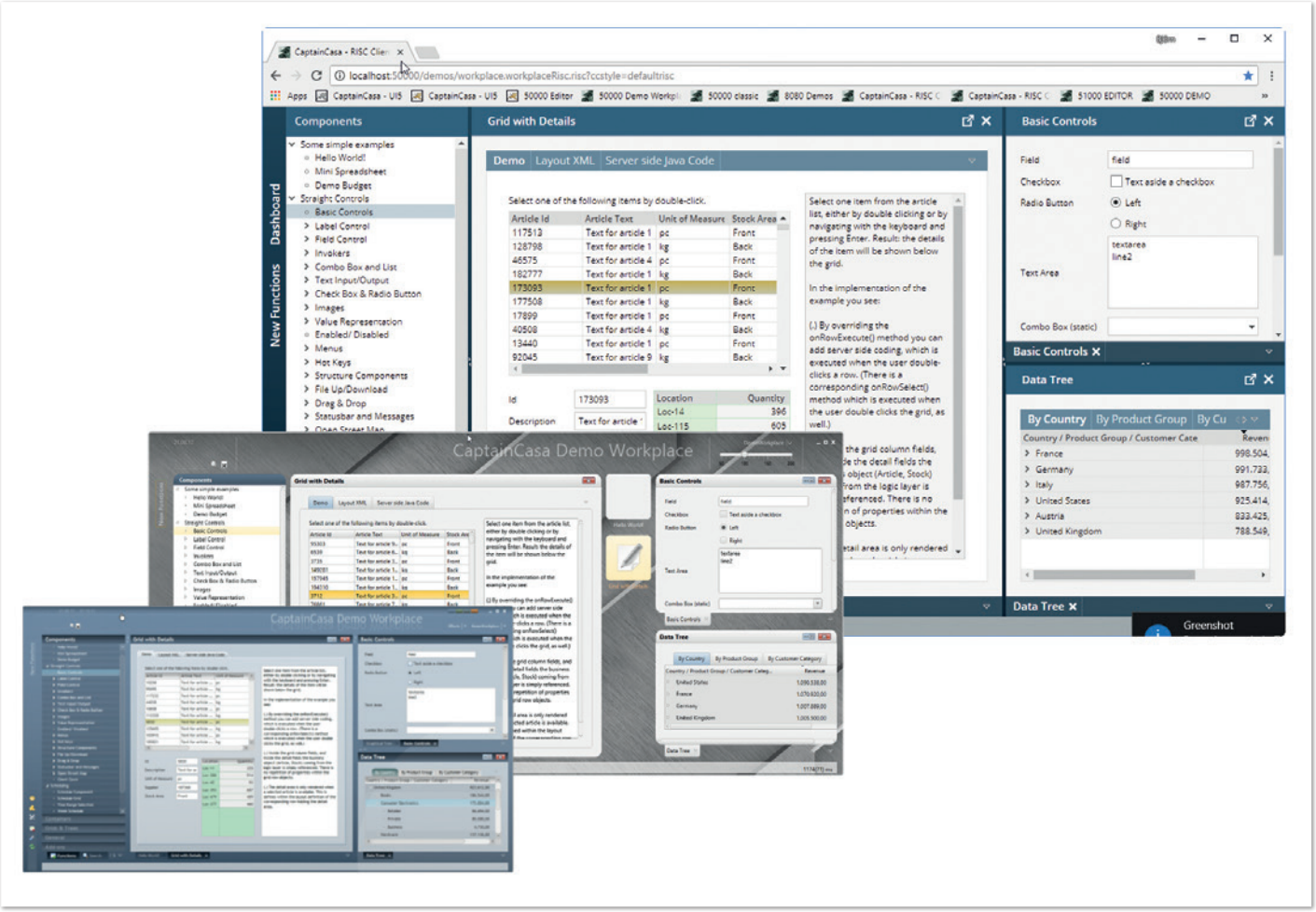


Abbildung 1: Eine Anwendung, drei Clients: Swing (2007), FX (2012) und RISC-HTML (2016)

kein konkret implementierter Anwendungsdialog, sondern eine generische Rendering Engine *(siehe Abbildung 2)*.

Modernisierungsdruck

Natürlich gingen drei Entwicklungen nicht an uns vorüber: Zum einen war da die zunehmende Qualität der Browser und die zunehmende Steigerung der Performance von JavaScript. Zum anderen gab es zunehmende generelle Zweifel an Java im Frontend – vor allem getrieben durch Sicherheitsprobleme der Java VM. Hinzu kam das Erscheinen von JavaFX als Ablösung von Java Swing – das spätestens mit JavaFX 2.0 eine gewisse Ernsthaftigkeit zeigte. Alle drei Entwicklungen nagten an der Attraktivität unserer Swing-Client-Lösung und damit an der Attraktivität der Anwendungslösungen unserer Community-Mitglieder.

Umstieg auf JavaFX

Unsere Versuche, mithilfe existierender Web-Frameworks unseren generischen Client in HTML/JavaScript zu reimplementieren, scheiterten regelmäßig. Gründe waren die üblichen Verdächtigen: mangelnde Browser-Kompatibilität, mangelnde Performance bei großen Szenarien und mangelnde Layout-Fähigkeiten. Es zeigte sich, dass die Frameworks für überschaubare Szenarien funktionierten, ab einer gewissen Komplexität jedoch immer an ihre Grenzen stießen.

Für uns war somit die Richtung gegeben: Wir blieben Client-seitig auf dem Java-Zug und setzen darauf, dass das Thema „Java im Client" über JavaFX wieder salonfähig wird. Mit dem Erscheinen von JavaFX 2.0 im Jahr 2011 starteten wir also die Reimplementierung unseres Clients. Beim Umstieg kam uns natürlich unsere entkoppelte Client-Struktur zu Hilfe – letztlich mussten wir ja nur den generischen Rendering-Client in JavaFX reimplementieren.

JavaFX hat uns dabei von Release 2.0 an technologisch gut gefallen. Wir haben schnell einen Weg gefunden, um effektiv damit zu entwickeln, und blieben innerhalb dieses Wegs auch von Stabilitätsproblemen verschont. Noch konkreter: Wir fanden JavaFX klasse.

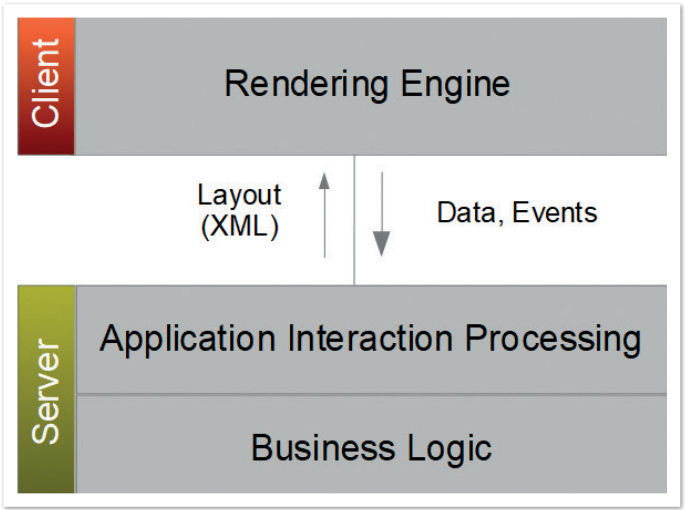


Abbildung 2: Abgekoppelter Client

Einige Probleme, die wir im Swing-Client funktional hatten, wurden in JavaFX solide gelöst: das freiere Styling der Komponenten, die bessere Integrierbarkeit von HTML-Inhalten und – auch im Bereich von Geschäftsanwendungen nicht ganz unwichtig – die Möglichkeit, professionell anmutende Animationen zu verwenden. Natürlich gab es auch technologische Herausforderungen: Ein Hauptkampfgebiet war dabei eine gewisse Trägheit des Clients beim Aufbau großer Dialoge, etwa mit einer hohen Anzahl von Textfeldern.

Summa summarum waren wir aber technologisch zufrieden und konnten nach einem Jahr einen Client zur Verfügung stellen, der eine weitgehend kompatible Alternative zu unserem Swing-Client darstellte – und dazu optisch noch eine Ecke attraktiver war. Nun standen wir da, mit einem frisch aufpolierten Client in moderner JavaFX-Technologie und mussten feststellen, dass JavaFX bei Weitem nicht in dem Maße abhob, wie wir es erwartet oder vielmehr gehofft hatten. Im Gegenteil: JavaFX wurde (und wird weiterhin) gerade aus dem Java-Lager heraus äußerst kritisch beurteilt und hat nie ein gewisses Hype-Potenzial erreicht.

Das bedeutete für die Software-Hersteller unserer Community: Die Ressentiments gegenüber Java im Frontend blieben unvermindert groß und wurden teilweise durch Zweifel an der Zukunftsfähigkeit von JavaFX noch verstärkt. Der Druck in Richtung HTML-Client stieg, aber unsere technologischen Bedenken, mit bestehenden HTML-Frameworks langfristig unsere Probleme zu lösen, blieben bestehen.

RISC-HTML: Eine etwas andere Nutzung von HTML und JavaScript

Genau in dieser Phase entwickelten wir eine Idee, die als Laborversuch startete und binnen kurzer Zeit zum Selbstläufer wurde. Die Idee bekam den Namen „RISC-HTML", in Anlehnung an den Paradigmenwechsel, der in den 1990er-Jahren durch die Entwicklung von RISC-Prozessoren im Bereich der Hardware vollzogen wurde. „RISC" steht bei uns für „Reduced Instruction Set Client".

Die Idee lautet: Zum flexiblen Erstellen von Oberflächen-Komponenten benötigt man eigentlich nur ganz wenige optische Grundelemente, die ein Browser zur Verfügung stellen muss. Die beiden wichtigsten Grundelemente sind dabei Rechtecke, die gestylt werden können, und Texteingabe-Felder. Diese Grundelemente müssen nun so zusammengesetzt werden, dass aus ihnen vollständige, funktionale Komponenten werden, also Buttons, Combo-Boxen, Grids, Dialoge etc. Für das Zusammensetzen wird allein die absolute Positionierung per „x,y,width,height,(z)"-Koordinate verwendet. Beispiel: Ein Button ist nichts anderes als eine Schachtelung von Rechtecken. Gleiches gilt für die Combo-Box; sie ist eine Zusammenschachtelung von Rechtecken um ein Textfeld und beinhaltet zusätzlich ein Dialog-Rechteck, das sich per z-Koordinate in den Vordergrund drängt *(siehe Abbildung 3)*.

Das Zusammensetzen der Grundelemente geschieht über JavaScript. Es werden also Komponentenklassen programmiert, die zur Laufzeit die Grundelemente im DOM-Tree anlegen und arrangieren sowie Events verarbeiten. Jede Klasse hat ein standardisiertes Set an Methoden und muss beispielsweise nach außen berichten, was ihre minimale beziehungsweise präferierte Größe ist. Jede Komponente ist für die Ausrichtung ihrer beinhalteten Grundelemente zuständig; im Falle eines Buttons ist die Logik hierfür überschaubar, bei komplexen Layout-Komponenten, etwa mit einer Mischung aus prozentualen und festen Größenangaben, ist die Logik entsprechend komplexer.

Abbildung 3: Alles baut sich aus Rechtecken zusammen

Im RISC-HTML-Ansatz werden also alle Komponenten auf Basis minimaler Grundelemente zusammengesetzt. Ganz konkret im Browser sind dies zwei Elemente: das DIV- und das INPUT-beziehungsweise TEXTAREA-Element. Das CSS-Styling sorgt dafür, dass die zusammengesetzten Elemente am Ende auch wie die betreffende Komponente aussehen.

Das Verfahren kann man recht gut veranschaulichen, wenn man sich den DOM-Tree eines konkret mit RISC-HTML aufgebauten Dialogs im Entwicklermodus des Browsers anschaut. Man sieht vor allem eines: eine tiefe Schachtelung von absolut positionierten DIV-Elementen *(siehe Abbildung 4)*.

Die Vorteile des RISC-HTML

Warum wurde der RISC-HTML-Ansatz vor dem Hintergrund einer Vielzahl existierender Web-Ansätze überhaupt erfunden? Der erste Hauptgrund war die Schaffung einer langfristig stabilen Grundlage für die Erstellung von Komponenten mit hoher Komplexität einerseits und mit technisch garantierter Browser-Kompatibilität andererseits. Vom Browser selbst werden nur einfachste Elemente erwartet („DIV", „INPUT"/„TEXTAREA"), diese kommen zudem nur auf einfachste Art und Weise zum Einsatz (absolute Positionierung). Damit ergibt sich eine dramatische Reduktion der Abhängigkeiten und damit der Kompatibilitätsprobleme.

Der zweite Hauptgrund war die Schaffung einer Grundlage, auf deren Basis beliebige Layout- und Gestaltungs-Strategien entwickelt werden können. Als Beispiel sei hier genannt, dass gängige HTML-Frameworks immer Probleme im Bereich des flexiblen vertikalen Ausrichtens haben: Ein Dialog hat einen festen Header-Bereich, einen scrollbaren Innenbereich und einen festen Footer-Bereich – für gängiges HTML ist so eine Konstruktion immer noch eine Herausforderung. Wir alle kennen die HTML-Seiten, auf denen man immer ganz nach unten zu den Verarbeitungsbuttons scrollen muss. Eigene Layouting-Verfahren sind im Browser nur über den Umweg des nachträglichen Ausmessens von Inhalten zu realisieren; der Browser rendert also erst aus; danach weiß man, wie man das Layout anpassen muss. Das Layout richtet sich dann vor dem Auge des Benutzers aus.

Was uns in der Praxis von Anfang an mehr als überraschte, das war die Performance, mit der über den RISC-HTML-Ansatz erstellte Dialoge aufgebaut und verarbeitet wurden. Ehrlich gesagt hatten wir

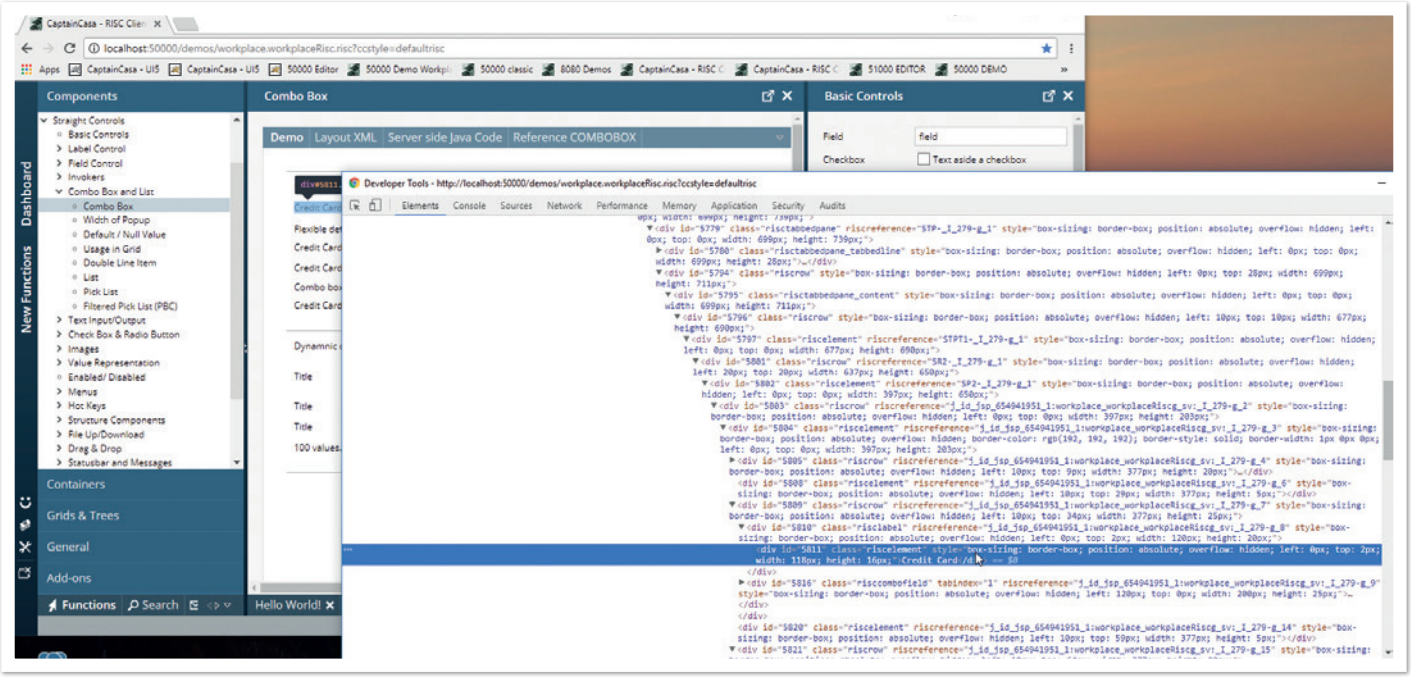


Abbildung 4: Schachtelung von einfachsten Grundelementen, absolut positionierten DIVs

hier massive Probleme erwartet, da bei RISC-HTML doch ein erheblicher Teil von Rendering-Logik, die klassisch im nativen Bereich des Browsers abläuft, nun auf einer JavaScript-Ebene abläuft.

Aber im Gegenteil: Auch wenn die Anzahl der zusammengesteckten Grundelemente durchaus hoch ist und die aufgebauten DOM-Element-Bäume deswegen groß sind, kommen die Browser damit hervorragend zurecht. Der Browser wird ja auch nur noch zum Zeichnen absolut positionierter Rechteck-Bereiche verwendet. Er rechnet nicht mehr selbst aus, wo welches Element nun genau hinkommt, sondern führt nur noch das aus, was ihm die entsprechende JavaScript-basierte Layout-Komponente vorgibt. JavaScript ist mittlerweile einfach schnell geworden und auf einem Level, der es erlaubt, auch Algorithmen einer gewissen Komplexität darin abzubilden.

Eine Anwendung mit RISC-HTML bauen

RISC-HTML ist zunächst also eine JavaScript-Bibliothek, in der über JavaScript-Klassen Komponenten (Button, Feld, Grid, Dialoge etc.) instanziiert und zusammengesetzt werden können. Komponenten haben ganz klassische Properties und Events. Jede Komponente zerlegt sich dabei in andere Komponenten oder – auf unterster Ebene – rendert sich in entsprechende Grundelemente aus (Rechtecke, Texteingabe-Felder).

Wie nun konkret eine Anwendung an diese Library anschließt, das ist prinzipiell frei gestaltbar. Freunde einer direkten, Client-seitigen JavaScript-Anwendungsverarbeitung können nun direkt Single-Page-Applications bauen.

Innerhalb unserer CaptainCasa-Community ist der Ansatz natürlich ein anderer: Hier wurde der generische Rendering-Client, der nichts anderes macht, als vom Server kommende Dialogbeschreibungen auszurendern, auf Basis der RISC-HTML-Library reimplementiert. Vom Server kommende Layoutbeschreibungen werden also in entsprechende RISC-HTML-Komponenten umgesetzt.

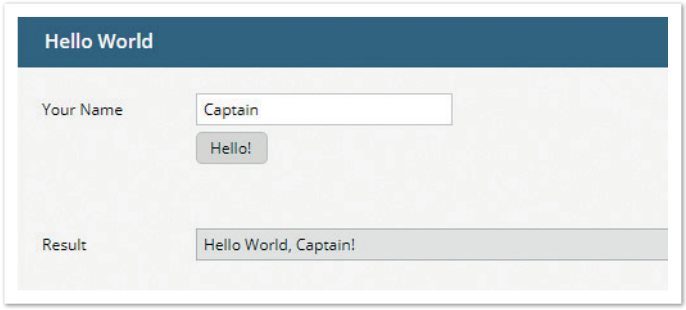


Abbildung 5: „Hello world"-Anwendung

```
<t:rowbodypane>
  <t:row>
    <t:label text="Your Name"
       width="100" />
    <t:field
       text="#{DemoHelloWorld.name}" width="200" />
  </t:row>
  <t:row>
    <t:coldistance width="100" />
    <t:button actionListener="#{DemoHelloWorld.on-
Hello}" text="Hello" />
  </t:row>
  <t:rowdistance  height="20" />
  <t:row>
    <t:label text="Result" width="100" />
    <t:field enabled="false" text="#{DemoHelloWorld.
output}" width="100%" />
  </t:row>
</t:rowbodypane>
```

Listing 1

CaptainCasa Enteprise Client

Wie hat man sich das aus Sicht der Anwendungsentwicklung vorzustellen? Dazu das berühmte Hello-World-Beispiel *(siehe Abbildung 5)*. Im einfachsten Falle definiert die Anwendung entweder direkt oder per Wysiwyg-Layout-Editor einen konkreten Dialog als XML-Definition *(siehe Listing 1)*.

```
package demo;

import javax.faces.event.ActionEvent;

public class DemoHelloWorld
{
    String m_name;
    String m_output;

    public void setName(String value){ m_name = value;
}
    public String getName() { return m_name; }

    public String getOutput() { return m_output; }

    public void onHello(ActionEvent ae)
    {
        if (m_name == null)
            m_output = "No name set.";
        else
            m_output = "Hello World, "+m_name+"!";
    }
}
```

Listing 2

In der XML-Definition sind die Auswahl und das Arrangement der Komponenten definiert. Attribute der Komponenten werden entweder direkt gesetzt oder sie werden per Expression an eine Java-Bean („Managed Bean") gebunden. Diese Java-Bean wird nun implementiert *(siehe Listing 2)*. Das war´s.

Die Laufzeitumgebung, auf Server-Seite übrigens JSF-basiert, übernimmt nun den Rest: Der generische Client fordert das Layout an, der Server liest das XML-Layout, baut den Kontakt zur „Managed Bean" auf und sendet das gefüllte XML-Layout zum Client, der dieses dann optisch für den Benutzer ausrendert. Der Benutzer sieht den Dialog und tätigt Eingaben. Beim Drücken des Buttons werden Eingaben und Events per Request zum Server geschickt, der sie dann der „Managed Bean" weitergibt. Änderungen in der Bean werden dann wieder zurück zum Client übertragen. Die Kommunikation zwischen Client und Server-Verarbeitung verläuft dabei inhaltlich immer so, dass nur Änderungen übertragen werden; dies gilt speziell für die Übermittlung der Layouts.

RISC-HTML ist doch gar kein richtiges HTML ...

Wie sieht der DOM-Element-Baum eines mit RISC-HTML erstellten Dialogs aus? Es gibt praktisch nur noch „DIV" und ein paar „INPUT" beziehungsweise „TEXTAREA", kein „TABLE", „TR", „BUTTON", „SPAN", „UL" oder „LI". Für diejenigen, die aus dem klassischen HTML-Umfeld kommen, ist das zunächst einmal gegen die Lehre: Die Semantik eines Dialogs lässt sich nicht mehr durch direktes Anschauen auf das gerenderte HTML erkennen.

Wir können dem entgegenhalten, dass das gängige Praxis ist. Jede Control-Bibliothek gibt sich nicht mit den doch recht einfach gestrickten HTML-Elementen zufrieden, sondern baut kräftig eigene, höherwertige Komponenten (etwa eine Kalender-Komponente), die sich dann wieder in echte HTML-Elemente zerlegt. Zugegeben, CaptainCasa treibt das etwas mehr auf die Spitze, aber das Prinzip ist das gleiche. Für bestimmte Bereiche, in denen die Semantik von Komponenten hervorstechen muss, gibt es als Folge dieser Entwicklung entsprechende Möglichkeiten, die Semantik explizit zu definieren. Der wichtigste Bereich ist „Accessibility". Dort werden im RISC-Client beispielsweise die semantischen Informationen über die ARIA-Schnittstelle übergeben.

RISC-HTML wurde erfunden, um durchaus komplexe Sachbearbeiter-Dialoge von Geschäftsanwendungen effektiv und langfristig stabil abbilden zu können. Es ist dort geeignet, wo Oberflächen programmiert werden. Dies sind typischerweise Dialoge mit einem gewissen Umfang und einer gewissen Komplexität. Bei der grafischen Ausgestaltung der Dialoge geht es mehr darum, ein konsistentes „Look & Feel" über die vielen Dialoge einer Anwendung effizient zu gestalten, und nicht darum, jeden einzelnen Dialog individuell zu designen. Der RISC-HTML-Ansatz ist naturgemäß nicht dort zu Hause, wo Oberflächen über HTML-Design beziehungsweise HTML-Templating entstehen. Hier steht ja gerade das klassische HTML und damit das explizite Designen mit HTML-Mitteln im Vordergrund.

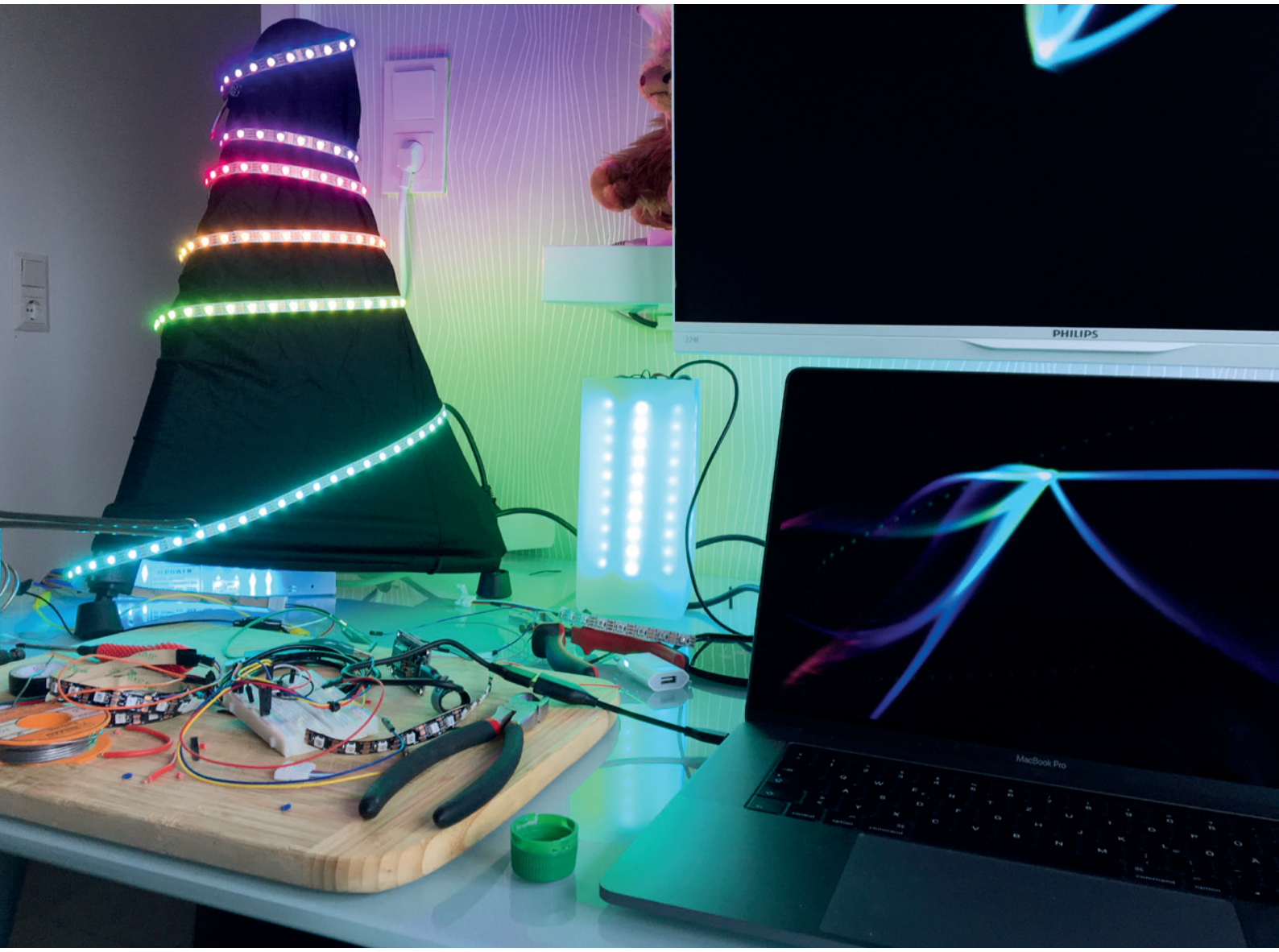
Fazit

Wir kamen von Swing, standen etwas ratlos mit einer JavaFX-Umsetzung herum und landeten über die Entwicklung einer neuen Methode, RISC-HTML, im Browser. Dort fühlen wir uns nun angekommen. Der RISC-HTML-Ansatz bietet uns im Browser eine Architektur-bedingte Stabilität und Flexibilität, auf deren Basis wir die langfristig orientierten Geschäftsanwendungen der Community-Mitglieder mit gutem Gewissen aufsetzen. Ein Austausch der UI-Technologie war in unserem Framework vergleichsweise einfach, weil von Anfang an die Client-Verarbeitung in Form einer generischen Rendering-Engine komplett von der Anwendungsverarbeitung getrennt war. Für die Anwendungsentwicklungen bedeutete dies, dass sie von dem Wechsel der Client-Technologie weitestgehend unberührt waren.

Und zu guter Letzt: Der „CaptainCasa-Enterprise-Client" – so heißt das Client-Framework unserer Community – kann kostenfrei (dies gilt sowohl für die Entwicklung als auch für die Nutzung) von der Seite *„http://www.CaptainCasa.org"* bezogen werden. Dort finden sich auch weitergehende Informationen und Online-Demos.

Björn Müller
bjoern.mueller@CaptainCasa.com

Björn Müller arbeitete zunächst zehn Jahre in der Anwendungsentwicklung, Basisentwicklung und Architekturentwicklung für SAP. 2001 erfolgte die Gründung der Casabac Technologies GmbH als Pionier im Bereich von Rich Internet Applications auf Basis von Client-Scripting (AJAX). 2007 gründete er die CaptainCasa Community, eine Verbindung mittelständischer, deutschsprachiger Softwarehäuser mit dem Fokus auf Rich Client Frameworks für umfangreiche, langlebige Anwendungssysteme.



Oracle JET on Speed mit socket.io

Enno Schulte, virtual7

Echtzeit-Applikationen definieren sich durch bezifferte, kurze Antwortzeiten. Diese sollen in der Regel so kurz sein, dass für die menschliche Wahrnehmung keine Verzögerung bemerkbar ist. Um diese Performance zu erreichen, müssen Server über entsprechende Ressourcen verfügen und das Netzwerk niedrige Latenzzeiten ermöglichen. Durch den geringeren Overhead und permanente Duplex-Verbindungen kann das WebSocket-Protokoll hier einen sinnvollen Beitrag leisten.

Vorwiegend findet man Echtzeit-Applikationen in der Prozess-Automatisierung oder Daten- und Telekommunikation. Mit dem Internet of Things (IoT) kommt ein weiterer Bereich hinzu, in dem die Verbindungsgeschwindigkeit eine große Rolle spielt. Niemand möchte mehrere Sekunden warten, bis das Licht angeht oder eine Tür sich öffnet. Läuft die Kommunikation zwischen Geräten und Clients über eine Cloud-Instanz, wird das Szenario noch verschärft.

Die Praxis

Viele Entwickler haben das eine oder andere IoT-Projekt in ihrem Haushalt. Das letzte Projekt des Autors entstand, als ihm ein Set von Philips HUE zu teuer erschien. Damit lassen sich Lampen einzeln oder als Gruppe via Smartphone oder Sprachassistent steuern.

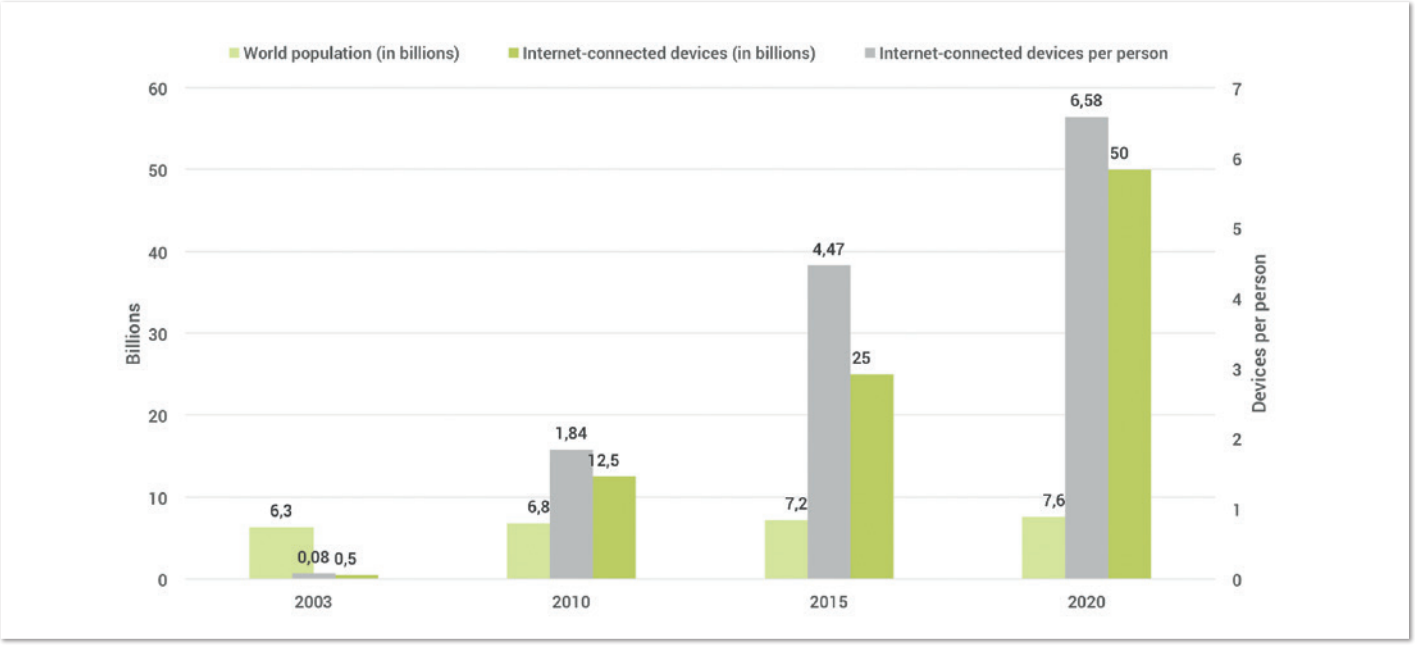


Abbildung 1: Entwicklung internetverbundener Geräte

Der hohe Preis für ein paar Lampen kam nicht infrage; da lag es nahe, ein solches System selbst zu bauen.

Das Konzept ist simpel. Ein Raspberry Pi steuert einen Streifen digitaler LEDs und stellt eine REST-Schnittstelle mittels „node.js" zur Verfügung. Um die Schnittstelle zu bedienen, kommt ein JavaScript-Client mit Oracle JET zum Einsatz. Die Verbindung zwischen Client und Server via HTTP ist in diesem Szenario eventuell etwas schwerfällig, um Lampen zu steuern. Niemand möchte ein bis zwei Sekunden warten, wenn man das Licht einschaltet. Von Lichtschaltern sind wir das auch nicht gewohnt. Hier kommt die Verbindung mittels WebSocket-Protokoll zum Zuge.

Der Artikel beschreibt alle notwendigen Teilbereiche zum Bau eines solchen Projekts. Zuerst betrachten wir den technischen Aufbau, um einen digitalen LED-Streifen mit einem Raspberry Pi und „node.js" leuchten zu lassen. Anschließend wird eine Schnittstelle über das WebSocket-Protokoll mit „socket.io" bereitgestellt. Daraufhin integrieren wir die „socket.io"-Library in Oracle JET und bauen einen Client zur Steuerung der LEDs.

Aber welchen Wert könnte ein solches Konstrukt in unser aller Berufsleben haben? Mit steigender Anzahl internetfähiger Geräte erhöht sich auch die Menge der Geräte, die sensorische Daten innerhalb unserer Produktions- oder Lieferprozesse sammeln. Diese Integration mit modernen Informations- und Kommunikationstechniken wird oftmals als „Industrie 4.0" bezeichnet. Dadurch steigert sich die Anzahl der Clients und der Bedarf, Informationen in Echtzeit auszuwerten *(siehe Abbildung 1)*.

Um diese Performance zu erreichen, eignet sich das WebSocket-Protokoll eher als eine reine HTTP-Implementierung, da bei HTTP die Verbindung immer wieder erneut aufgebaut werden muss. Durch den geringeren Overhead und die permanenten Duplex-Verbindungen kann das WebSocket-Protokoll wesentlich schneller Nachrichten am Server empfangen und zusätzlich Nachrichten vom Server ausgehend senden.

Abbildung 2: Ein LED-Streifen

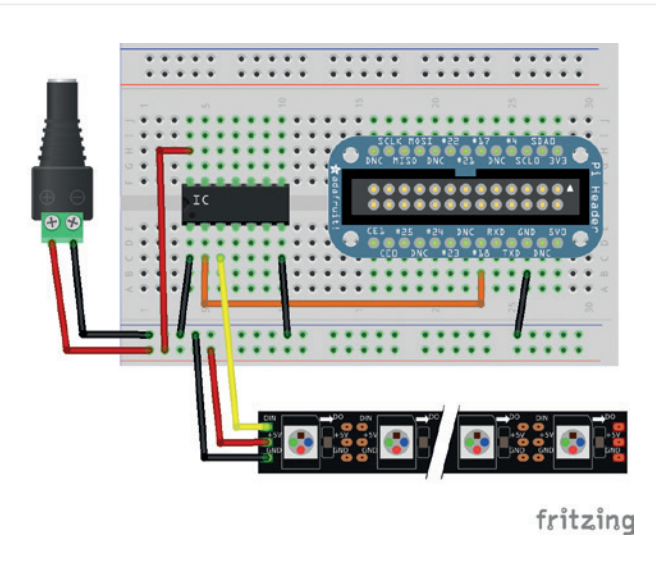


Abbildung 3: Schaltung der LEDs mit IC (https://learn.adafruit.com/assets/19658)

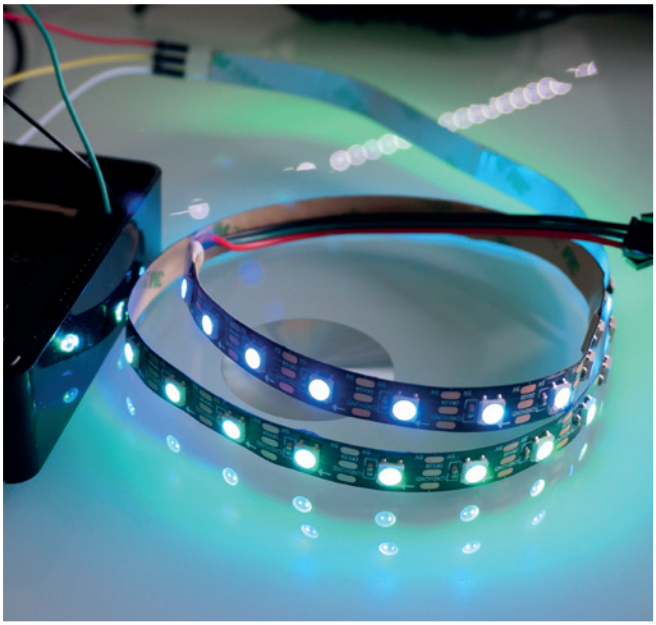


Abbildung 4: Das erste Ergebnis

Der technische Aufbau

Bei nahezu jedem Discounter bekommt man inzwischen farbig leuchtende LED-Streifen (fünf Meter lang), bei denen man per Fernbedienung die LEDs in zwanzig unterschiedlichen Farben leuchten lassen kann. Man erkennt diese Streifen daran, dass sie vier Adern besitzen, jeweils eine für rot, grün und blau sowie einen negativen Pol.

Mit einer solchen analogen Bauweise ist es nur möglich, alle LEDs in derselben Farbe leuchten zu lassen; die Wiedergabe von Mustern oder nur einzeln leuchtenden LEDs ist nicht möglich. Viel spannender wird es mit digitalen LEDs. Bei diesen befindet sich auf jeder LED ein Mikrocontroller, der es ermöglicht, die LEDs individuell zu schalten. Wer sich solche LEDs beschaffen möchte, sollte nach „WS2812b"-LEDs Ausschau halten *(siehe Abbildung 2)*.

```
// Initialisieren der LEDs
var numLeds = 200;
var ws281x = require('rpi-ws281x-native');
ws281x.init(numLeds);

// Ein Array aus RGB Werten erzeugen
// in diesem Fall würden alle LEDs weiß leuchten
var ledData = new Array(numLeds);
ledData.fill([255,255,255]);

// Funktion aus RGB Werten die Integer Repräsentation
zu erzeugen
function rgb2Int(r, g, b) {
    return ((r & 0xff) << 16) + ((g & 0xff) << 8) + (b
& 0xff);
};

// Funktion um das RGB Array in Integerwerte umzuwan-
deln
function ledData2pixelData(ledData){
    var pixels = new Uint32Array(numLeds);
    for(var i = 0; i < ledData.length; i++){
        pixels[i] = rgb2Int(ledData[i][0]
,ledData[i][1]
,ledData[i][2]);
    }
    return pixels;
};

// Dieser Aufruf lässt den LED Streifen leuchten
ws281x.render(ledData2pixelData(ledData));
```

Listing 1

Eine dieser LEDs benötigt 60 mA und fünf Volt Spannung. Um diese risikofrei an einem Raspberry Pi zu betreiben, ist ein IC empfohlen, um den mit drei Volt betriebenen Raspberry Pi von den LEDs zu trennen. Allerdings hat der Autor zweihundert LEDs problemlos direkt an den Raspberry Pi angeschlossen und dabei keine Probleme bemerkt *(siehe Abbildung 3)*.

Der node.js-Server

Sind die technischen Voraussetzungen geschaffen, können wir uns dem „node.js"-Server widmen. Glücklicherweise gibt es dafür be-

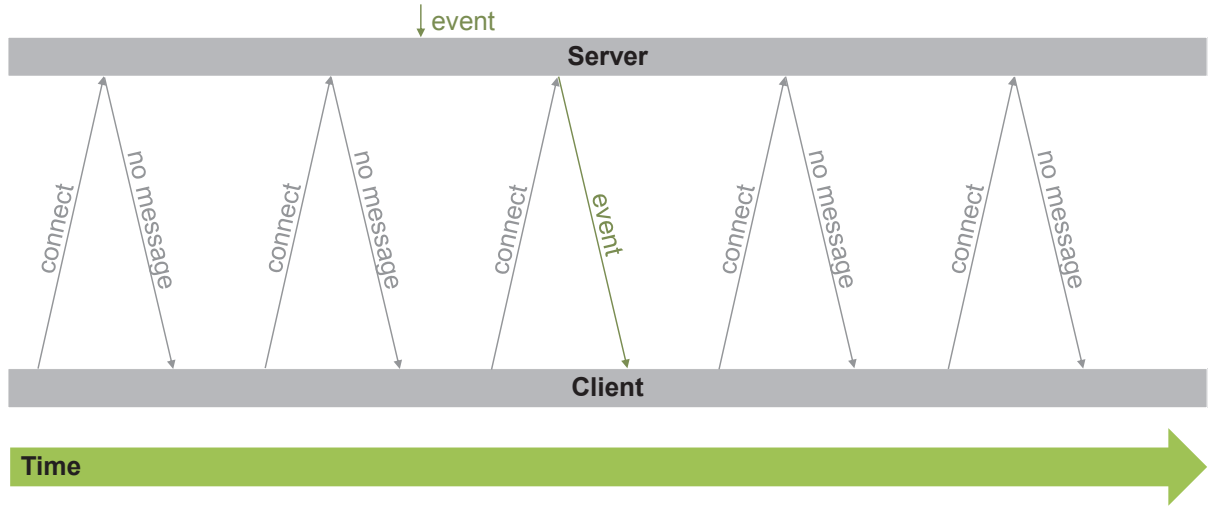


Abbildung 5: Polling

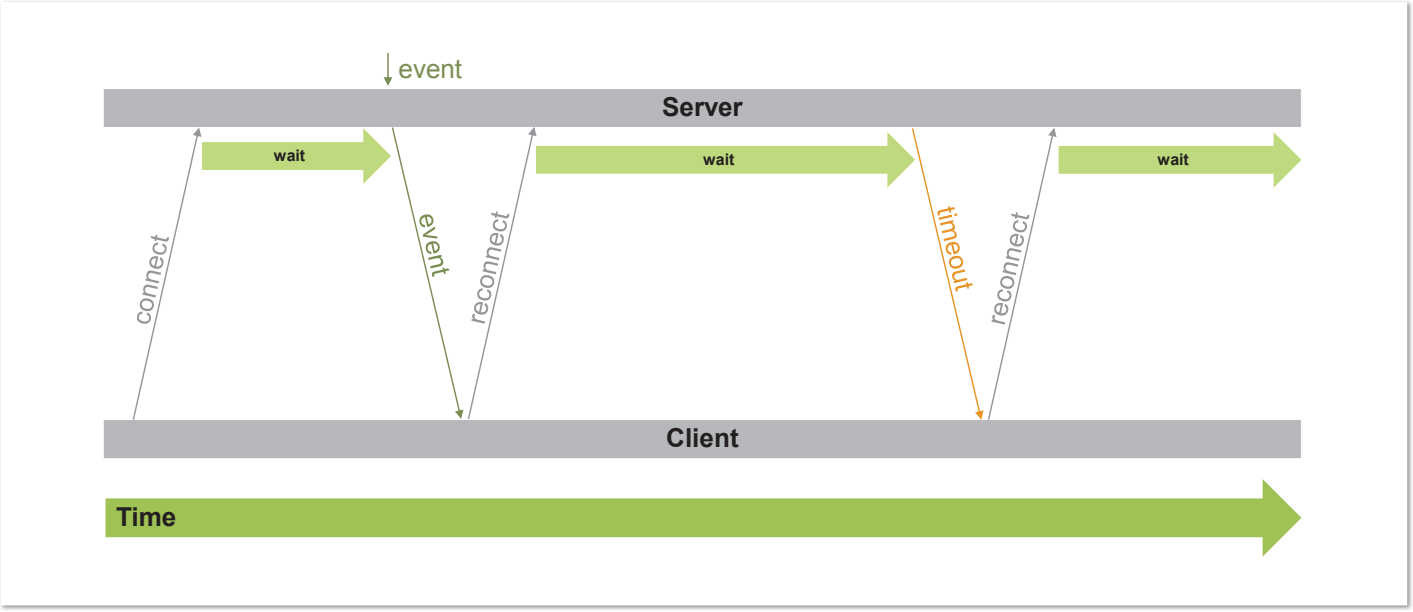


Abbildung 6: Long Polling

reits ein Package, das es sehr einfach macht, die LEDs zu steuern. In der Datei „package.json" müssen wir dazu die Zeile „"rpi-ws281x-native": "~0.8.1"" hinzufügen. Weitere Infos zum Package stehen unter *„https://www.npmjs.com/package/rpi-ws281x-native".*

Bevor wir dieses Package einsetzen können, ist es wichtig, die interne Soundkarte des Raspberry zu deaktivieren, denn das Package und die Soundkarte nutzen den gleichen GPIO-Pin. Dazu einfach in der Datei „/etc/modprobe.d/alsa-base.conf" die Zeile „options snd-usb-audio index=-2" auskommentieren.

Ist alles angeschlossen und „rpi-ws281x-native" installiert, empfiehlt es sich, den Aufbau mit den mitgelieferten „examples" zu testen. Im Ordner „node_modules" sollte sich ein Unterordner „rpi-ws281x-native/examples" befinden. In diesem finden wir verschiedene Beispiele, die einfach über „sudo node rainbow.js 200" mit Root-Privilegien und der Anzahl der LEDs als Parameter gestartet werden können. Nun sollten alle LEDs wie ein Regenbogen leuchten und die Farben permanent wechseln *(siehe Abbildung 4)*.

Sobald alles korrekt angeschlossen und betriebsbereit ist, kann man seine eigene Implementierung innerhalb eines „node.js"-Servers erstellen. Mit wenigen Zeilen könnte man beispielsweise im Falle eines Request den kompletten Streifen in Weiß leuchten lassen *(siehe Listing 1)*.

WebSockets

Um nun via REST die LEDs steuern zu können, müsste man noch die URLs definieren und den obigen Code zur Ausführung bringen, wenn die URLs aufgerufen werden. Was passiert allerdings, wenn wir mehrere Clients haben? Wenn ein Client die Farbe ändert, würden wir es an einem anderen Client nicht sehen können. Um dies in einem HTTP-Request/Response-Szenario zu ermöglichen, gibt es eine Vielzahl von Techniken. Alex Russell hat diese unter dem Begriff „COMET" zusammengefasst. Alle haben gemeinsam, dass sie mit regelmäßigen Requests abfragen, ob es eine Zustandsänderung gegeben hat. Dabei kommen am häufigsten Polling und Long Polling zum Einsatz *(siehe Abbildungen 5 und 6)*.

Eine Alternative dazu bietet das WebSocket-Protokoll. Es stellt keinen „1:1"-Ersatz für HTTP dar, sondern ist eine auf HTTP aufsetzende, effiziente, bidirektionale Kommunikationsmethode. Im Gegensatz zu HTTP gibt es einen sehr geringen Overhead. Inzwischen ist das Protokoll Bestandteil von HTML5 und wird von nahezu allen Browsern unterstützt. Ein wesentlicher Unterschied zu HTTP liegt darin, dass die Verbindung bestehen bleibt und somit auch Nachrichten vom Server initiiert werden können. In unserem Fall ist es also möglich, beim Ändern der Farbe sofort alle Clients darüber zu informieren.

Für die meisten Programmiersprachen gibt es inzwischen Frameworks, die eine WebSocket-Verbindung unterstützen, für *„.net"* beispielsweise SignalR, bei Java ist Atmosphere beliebt und für JavaScript ist „socket.io" ein geeignetes Hilfsmittel. Es wird empfohlen, eines dieser Frameworks zu verwenden, anstatt die nativen Methoden für den Aufbau einer WebSocket-Verbindung zu nutzen. Falls ein Client das WebSocket-Protokoll nicht unterstützt, nutzen diese Frameworks automatisch einen COMET-Mechanismus.

„socket.io"

Mit unserem JavaScript-basierten Duo aus „node.js"-Server und Oracle-JET-Client liegt es nahe, „socket.io" für die WebSocket-Kom-

```
var server = http.createServer(app);
var io = require('socket.io')(server);
// To prevent CORS Problems
io.origins('*:*');

io.on('connection', function(socket){
    socket.on('message', function(msg){
      ledData.fill(msg);

      ledData2pixelData(ledData);
      ws281x.render(ledData2pixelData(ledData));
      // Nachricht an alle Clients
      io.emit('update', msg);
    });
});
```

Listing 2

```
copyCustomLibsToStaging: {
  fileList: [{cwd: 'node_modules/socket.io-client/
dist/',
  src: ['**', '!test.js'],
  dest: 'web/js/libs/socket.io-client'}]
},
```

Listing 3

```
'socket.io': {
                exports: ['io']
            },
```

Listing 4

munikation zu verwenden. Zur Integration in den „node.js" Server binden wir einfach „socket.io" in „package.json" ein und lassen NPM den Rest erledigen. Mit dem Code in *Listing 2* können wir den Server nun so konfigurieren, dass er beim Erhalt einer Nachricht die LEDs entsprechend leuchten lässt.

Oracle JET

Das Oracle JavaScript Extension Toolkit (JET) ist, wie der Name auch suggeriert, mehr ein Toolkit als ein Framework. Daraus resultiert einer der größten Vorteile der Technologie: Die eingesetzten Bibliotheken sind nicht in Stein gemeißelt und können nach Belieben ausgetauscht oder erweitert werden. Oracle hat mit den bereits im Toolkit enthaltenen Paketen eine gut aufeinander abgestimmte Kombination vorgelegt. Wer sich die vielen verschiedenen JavaScript-Bibliotheken auf dem Markt anschaut, sieht, dass das kein einfaches Unterfangen ist. Mit Version 2.2 ist noch Bower als Komponenten-Manager hinzugekommen. Zudem befand sich NPM als Paket-Manager im Einsatz. Mit Version 3 hat Oracle sich von Bower verabschiedet und setzt nun einheitlich auf NPM.

Um „socket.io" in unserer JET-Applikation nutzen zu können, müssen wir zuerst die notwendigen Pakete via NPM installieren. Dazu erweitern wir unsere „package.json" um den Eintrag „"socket.io-client": "*"". Leider sind noch ein paar Schritte mehr erforderlich. Der wohl schwierigste: Unter „/scripts/grunt/config" steht eine Datei namens „oraclejet-build.js". Dort müssen wir unsere „socket.io-client"-Bibliothek in den Schritt „copyCustomLibsToStaging" integrieren *(siehe Listing 3)*. Dadurch werden die Dateien aus dem Ordner „node_modules" in das Distributionsverzeichnis der JET-Applikation kopiert. Weitere Informationen dazu stehen unter *„https://docs.oracle.com/middleware/jet300/jet/developer/GUID-EC40DF3C-57FB-4919-A066-73E573D66B67.htm#JETDG-GUID-EC40DF3C-57FB-4919-A066-73E573D66B67".*

Jetzt haben wir die Pakete heruntergeladen und in ein anderes Verzeichnis kopiert. Um diese nutzen zu können, müssen wir sie allerdings noch bei JET anmelden. Als Erstes hinterlegen wir dazu den Pfad zu den Dateien in der Datei „main-release-paths.json". In unserem Fall „"socket.io": „libs/socket.io-client/dist/socket.io.slim"".

Oracle JET macht starken Gebrauch von RequireJS, um Bibliotheken zur Laufzeit zu laden. Um unsere „socket.io-client"-Bibliothek einsetzen zu können, müssen wir einen Verweis zu dieser Bibliothek in RequireJS hinterlegen. Dies geschieht in der Datei „main.js". Dort muss in dem Block „require.config {...}" der Pfad „'socket.io': 'libs/socket.io-client/socket.io.slim'" hinzugefügt und im Block „shim" „socket.io" ebenfalls noch mal referenziert werden *(siehe Listing 4)*.

Sobald alle diese Referenzen gesetzt sind, kann man in der JET-Applikation Gebrauch von der neuen Bibliothek machen. Dazu ist lediglich im „define" des jeweiligen Moduls die „socket.io-client"-Bibliothek zu laden. *Listing 5* zeigt ein Beispiel.

In dem Beispiel kann man sehen, dass als dritter Parameter die „socket.io-client"-Bibliothek und ein entsprechender Parameter an dritter Stelle in der „function" geladen wird. Mit der Variablen „io" können wir nun im Modul unsere Socket-Verbindung aufbauen und auf Events reagieren, die wir geschickt bekommen, beziehungsweise eigene Events auslösen *(siehe Listing 6)*.

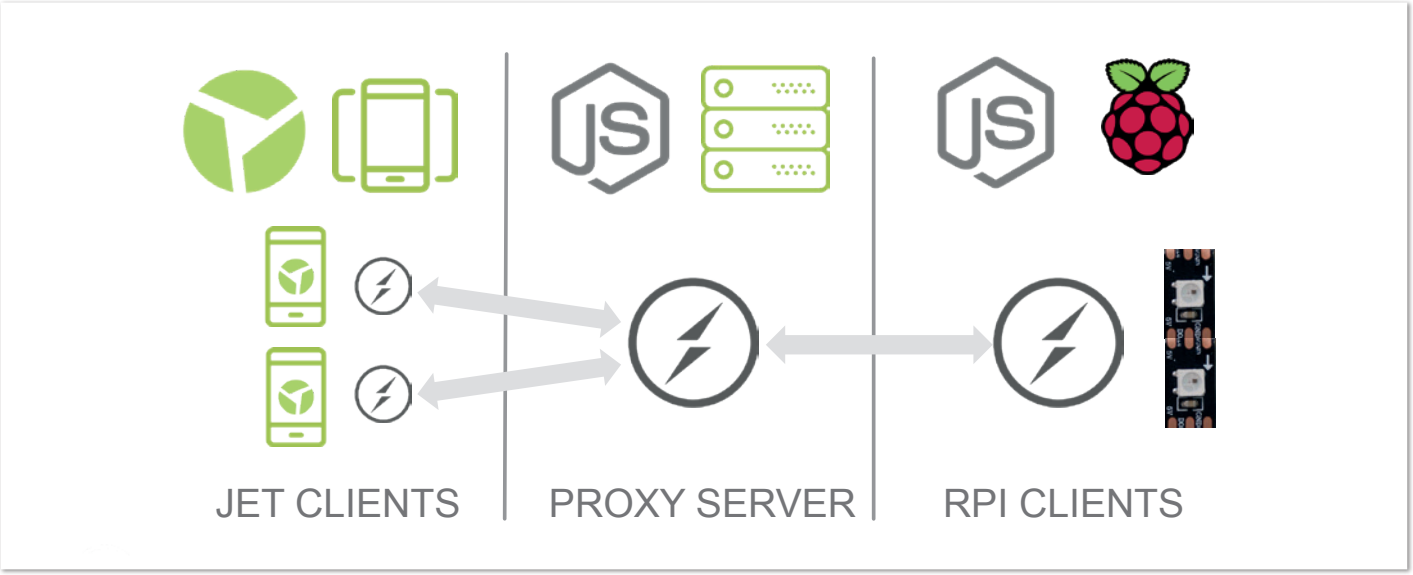


Abbildung 7: Die Gesamt-Architektur

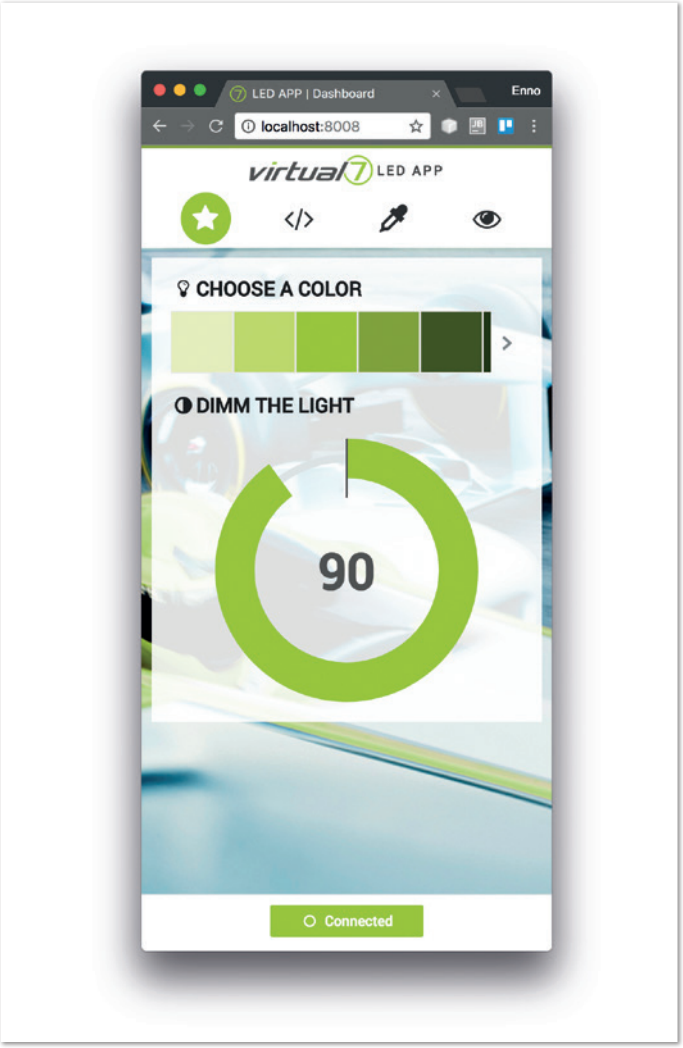


Abbildung 8: Die Lampen steuern

```
define(['ojs/ojcore', 'knockout', 'socket.io', 'color-
Model', 'ojs/ojrouter', 'ojs/ojknockout', 'ojs/ojar-
raytabledatasource', 'ojs/ojbutton'],
    function (oj, ko, io, colorModel) {…
```

Listing 5

```
self.socket = io.connect('api.meinedomain.de');
self.socket.on('update',function(payload){
                        console.log(payload);
                    });
```

Listing 6

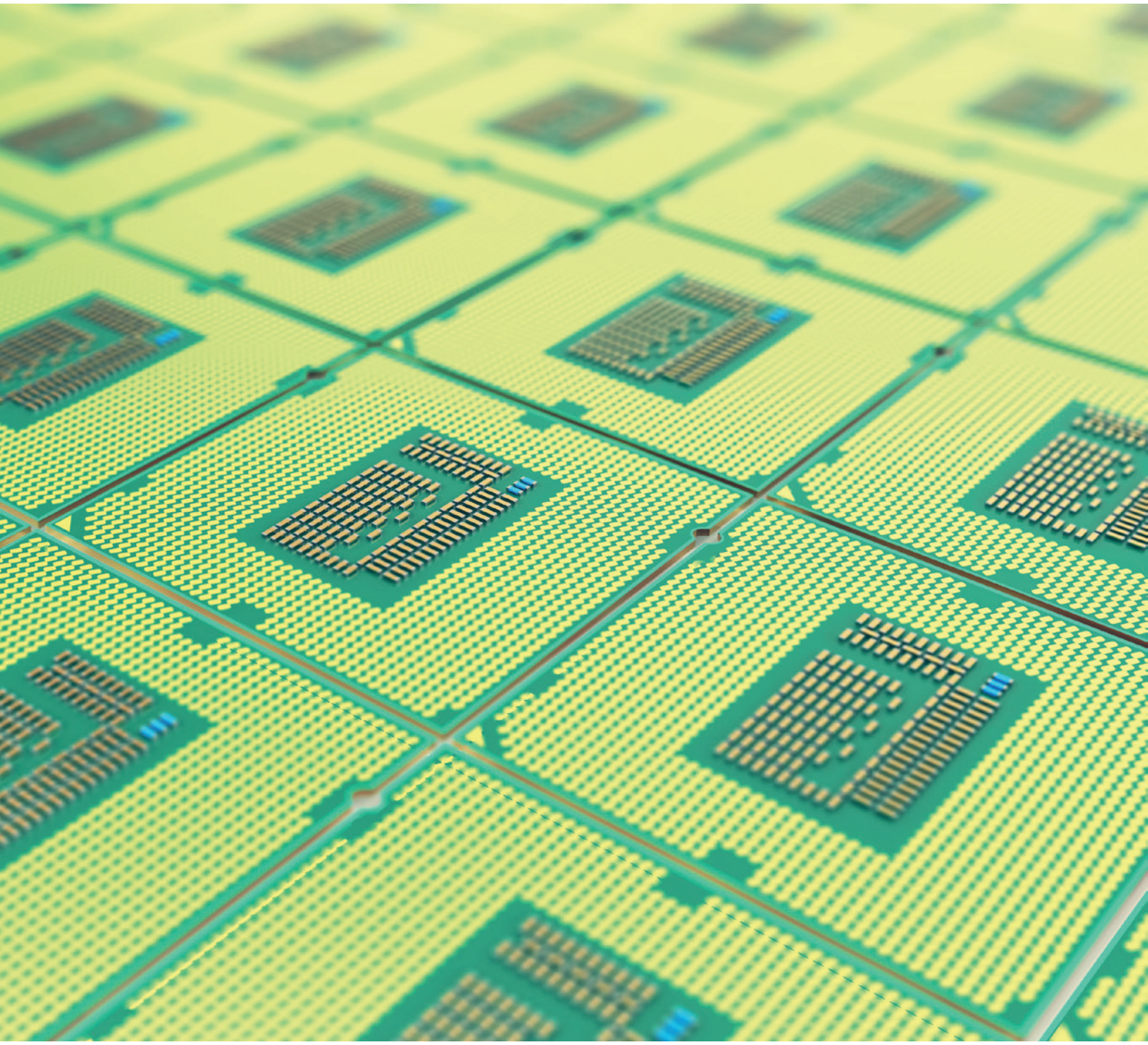
Soviel zur Theorie. Wie können nun diese ganzen Code-Schnipsel zu einem funktionierenden Konzept für eine eigene Lichtsteuerung werden? Wir haben einen Raspberry Pi mit einem „node.js"-Server. Diesen können wir mit unserer JET-Client-Applikation steuern. Damit hätten wir eine Lampe im Griff. Problematisch wird es allerdings, wenn wir unsere Lampe so schön finden, dass wir eine weitere integrieren wollen. Dann müsste man zwangsläufig den Client anpassen und noch eine Verbindung zum zweiten „node.js"-Server aufbauen. Um dieses Problem zu umgehen, wurde ein weiterer „node.js"-Server hinzugefügt. Dieser kommuniziert nun mit jedem Raspberry Pi und jedem Client über WebSockets *(siehe Abbildung 7)*.

Dieser Proxy-Server ist auch sehr gut geeignet, um die Client-Applikationen unter einer Domain zu hosten. Andernfalls müsste man sich noch genauer mit der Distribution einer solchen App auf den einzelnen Endgeräten beschäftigen. In der angezeigten Gesamt-Architektur befindet sich der Proxy-Server bei einem Cloud-Hoster in Frankfurt. Wenn nun also ein Client die Farbe der LEDs ändert, wird ein Signal nach Frankfurt und von dort zum entsprechenden Raspberry Pi gesendet. Dieser setzt dann den Befehl um und meldet die aktuell dargestellten Farben an den Proxy-Server zurück, der schlussendlich alle Clients über die neuen Farben informiert.

Trotz des Umwegs und des Hin und Hers kann man die Verzögerung nicht wahrnehmen. Somit lassen sich die Lampen in Echtzeit steuern und es kann an jedem Client in Echtzeit verfolgt werden, was die Lampe gerade anzeigt *(siehe Abbildung 8)*. Bei einer einzigen Farbe ist das noch unspektakulär. Programmiert man sich jedoch ein paar Algorithmen, die Farbmuster wie Regenbogen, Farbverläufe und Ähnliches darstellen, ist das Ganze schon wesentlich interessanter. Ebenso wie die Lampen zu tracken, bestünde in einem IoT-Szenario im industriellen Sinne die Möglichkeit, Maschinendaten live in einem Dashboard zu verfolgen.

Enno Schulte
enno.schulte@virtual7.de

Enno Schulte (Dipl.-Wirt.-Inform.) arbeitet seit mehr als zehn Jahren im Consulting mit einem starken Fokus auf Oracle-Technologien. Ein Hauptaugenmerk hat er dabei auf ADF, dem JEE-Framework von Oracle. Mit einem intensiven Engagement in der ADF-Community der DOAG sowie mit guten Kontakten zu Oracle gestaltet er die Entwicklung und Verbreitung dieser Technologie. Darüber hinaus beschäftigt er sich bei seiner Tätigkeit als Senior Technical Consultant bei virtual7 mit Microservice-Architekturen und der Implementierung leichtgewichtiger JavaScript-Lösungen in IoT-und Container-basierten Umgebungen.



Multicore-Programmierung mit Java

Jörg Hettel, Hochschule Kaiserslautern

Java ist für die Entwicklung von Multicore-Software gut gerüstet. Neben zahlreichen Concurrency-Tools stehen den Entwicklern heute mit den parallelen Streams und der „CompletableFuture"-Klasse zwei mächtige Frameworks zur Verfügung, um den Ablauf von Programmen durch Parallelisierung zu beschleunigen. Dieser Artikel stellt die für die Anwendungsentwicklung zur Verfügung stehenden Concurrency-Konzepte kurz vor und zeigt den Einsatz der beiden Parallelisierungs-Frameworks anhand von Beispielen.

Der im Zeitalter der Einzelkern-Prozessoren geltende Grundsatz, dass sich mit der Einführung neuer Prozessor-Generationen auch automatisch die Laufzeit-Performance von Anwendungen erhöht, gilt nicht mehr uneingeschränkt. Bei neuen Prozessor-Generationen wird nicht mehr die Taktrate erhöht, sondern die Anzahl der zur Verfügung stehenden Rechenkerne. Eine Beschleunigung der Anwendung erhält man somit nur noch dann, wenn die zusätzlichen Rechen-Ressourcen auch genutzt werden. Java ist für die Entwicklung paralleler Programme gut gerüstet. So wurde das mit Java 5 eingeführte Concurrency-Paket seither sukzessive erweitert und mit leitungsfähigen Parallelisierungsframeworks ausgestattet.

Die in Java vorhandenen Concurrency-Konzepte lassen sich grob in zwei Kategorien einteilen. Zum einen sind dies Werkzeuge (Concurrency-Tools), um Anwendungen „threadsafe" zu entwickeln, und zum anderen Frameworks, mit denen man Anwendungen durch Ausnutzung von Parallelität beschleunigen kann.

Low-Level-Konzepte

Java besitzt bereits von Beginn an rudimentäre Konzepte für die Thread-Programmierung *(siehe Abbildung 1)*. Die Klasse „Thread" beziehungsweise das Interface „Runnable" bilden die Abstraktion für einen eigenständigen Programmfluss. Das Schlüsselwort „synchronized" und die Objektmethoden „wait" und „notify" beziehungsweise „notifyAll" koordinieren den parallelen Zugriff auf Objekte.

Da Java-Programme auf verschiedenen Plattformen laufen, ist die Java-Laufzeit-Umgebung (virtuelle Maschine) auch mit einem eigenen Speichermodell ausgerüstet, das die Sichtbarkeit von Variablenänderungen für parallel ausgeführte Threads festlegt. So wurden den Schlüsselwörtern „synchronized" und „volatile" Sichtbarkeitsgarantien zugeordnet.

Der Praxiseinsatz dieser rudimentären Konzepte ist sehr komplex und somit fehleranfällig. Zudem sind die Konzepte stark limitiert. So gibt es beispielsweise keinerlei Möglichkeiten, an einer „synchronized"-Barriere wartende Threads zu unterbrechen oder diese nach einem fairen Prinzip wie „first-in-first-out" durchzulassen.

Java-Concurrency-Tools

Mit Java 5, dessen Release-Datum mit dem breiten Aufkommen von Multicore-Rechnern zusammenfällt, wurde das Paket „java.util.concurrency" eingeführt. Es enthält verschiedene Klassen für den erleichterten Umgang mit Threads. So hat man ergänzend zur „synchronized"-Barriere verschiedene Lock-Klassen eingeführt. Neben der Klasse „ReentrandLock", die auch eine faire Lockvergabe und unterbrechbares Warten unterstützt, wurde auch ein Lock bereitgestellt, bei dem zwischen Lese- und Schreib-Zugriffen unterschieden werden kann.

Mithilfe der Klasse „StampedLock" (seit Java 8) ist nun auch die Implementierung von optimistischem Locking möglich. Zur Steuerung mehrerer Threads dienen die Klassen „CountDownLatch", „CyclicBarrier" und „Phaser" (seit Java 7). An einer „CountDownLatch" können beliebig viele Threads angehalten werden. Erst wenn diese durch ein Signal geöffnet wird, dürfen die wartenden Threads weiterlaufen. Eine „CyclicBarrier" funktioniert ähnlich, nur dass hier eine vorher festgelegte Anzahl (Schwellwert) von Threads angestaut wird. Ist der Schwellwert erreicht, dürfen die wartenden Threads weiterlaufen. Danach wird sie wieder geschlossen, sodass ankommende Threads wieder bis zur nächsten Freigabe, also dem Erreichen des Schwellwerts, warten müssen. Eine offene „CountDownLatch" kann nicht wiederverwendet und der Schwellwert einer „CyclicBarrier" nicht verändert werden. Die „Phaser"-Klasse hebt diese beiden Limitierungen auf.

Eine weitere Klasse zur Koordinierung mehrerer Threads ist „Semaphore". Sie entspricht einem Durchlasszähler, um in einfacher Weise den Zugang zu Ressourcen-Pools zu verwalten. Ergänzend wurden „atomare" Datenklassen wie „AtomicInteger" oder „AtomicReference" eingeführt. Damit lassen sich Variablen beziehungsweise

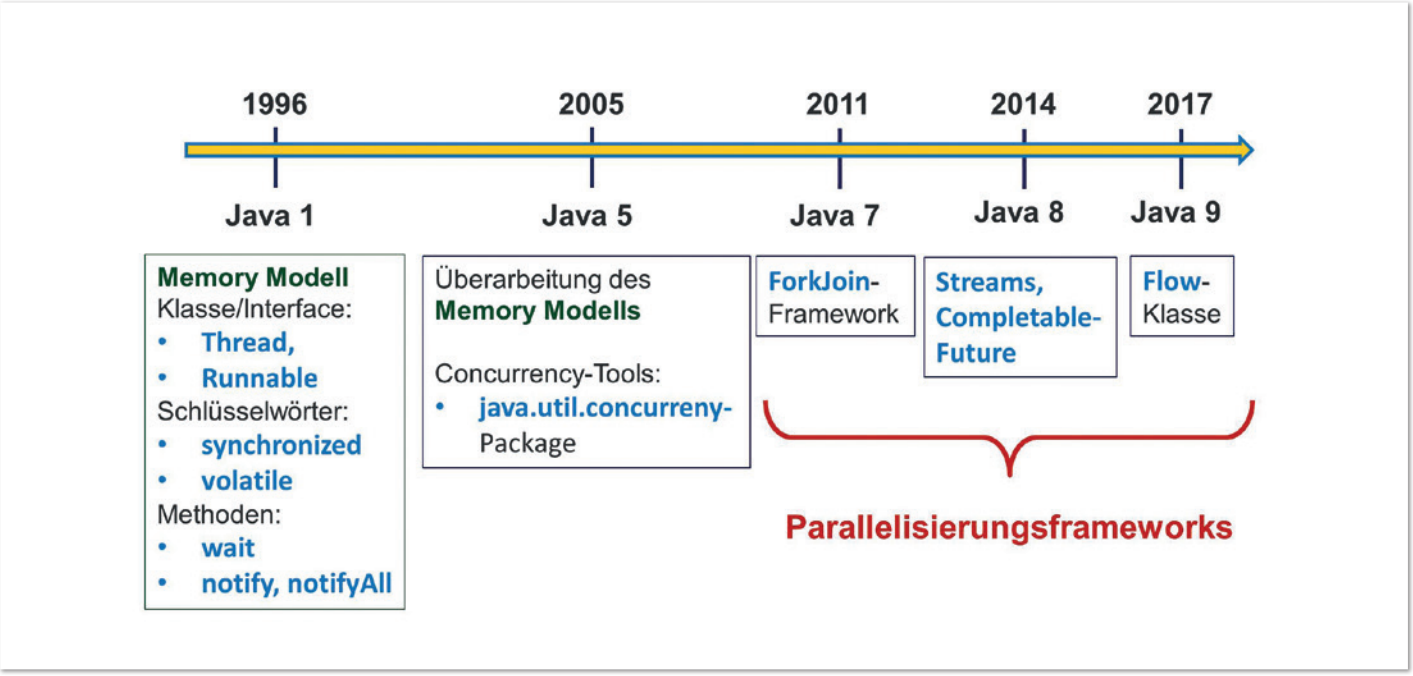


Abbildung 1: Einführung verschiedener Concurrency-Konzepte

$$\sum_{k=0}^{n} Bin(n,k;p) = 1$$

Mit:

$$Bin(n,k;p) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

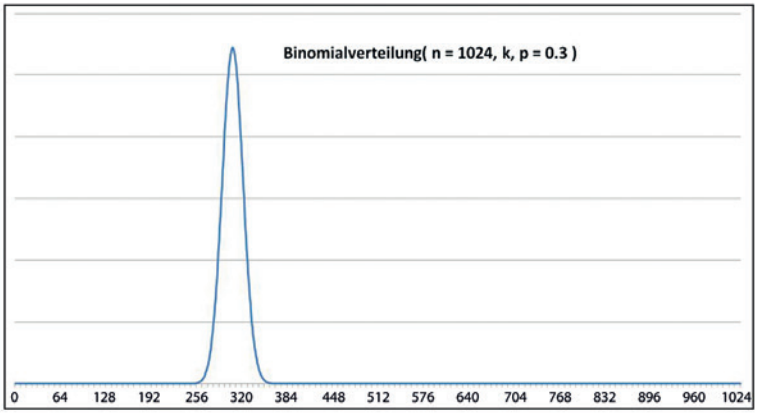


Abbildung 2: Die Binomialverteilung für n=1024 und p=0,3. Auf der x-Achse ist der Parameter „k" aufgetragen

Referenzen atomar lesen und ändern, ohne dass (in der Regel) die virtuelle Maschine ein explizites Locking durchführen muss. Möglich wird dies, indem hier direkt die von moderner Hardware angebotenen „compare-and-swap"-Operationen zum Einsatz kommen. Mithilfe der atomaren Klassen können zum Beispiel threadsichere, lockfreie Datenstrukturen implementiert werden. Eine Auswahl verschiedener threadsicherer Datenstrukturen wird auch gleich mit angeboten, etwa „ConcurrentSkipListSet" oder „ConcurrentHashMap". Datenstrukturen wie „BlockingQueue" oder „Exchanger" können für den sicheren Datenaustausch zwischen Threads benutzt werden.

Die mit Java 5 für die Entwicklung paralleler Anwendungen eingeführte Abstraktion sind der „Threadpool" beziehungsweise „Executor" sowie das „Future"-Pattern. Die Einführung dieser Konzepte befreit Entwickler von der rudimentären Thread-Programmierung. Man implementiert jetzt lediglich „Tasks", die dann einem geeigneten „Executor" zur asynchronen Ausführung übergeben werden. Bei den Tasks unterscheidet man zwei Arten: ohne oder mit Rückgabe. Tasks ohne Rückgabe implementieren das „Runnable"-, Tasks mit Rückgabe das „Callable"-Interface. Zur Ausführung von Tasks stehen verschiedene „Executor"-Varianten zur Verfügung: solche, die intern die benötigte Anzahl von Threads dynamisch anpassen, oder solche, die Tasks periodisch ausführen.

Mithilfe des Executor-Mechanismus lassen sich parallele Abläufe schon einfacher implementieren, wobei aber immer noch einiges an Handarbeit notwendig ist. Ein Beispiel ist die Summation einer Binomial-Verteilung, deren Ergebnis immer den Wert „Eins" liefert *(siehe Abbildung 2)*. Sie beschreibt die Erfolgswahrscheinlichkeit in einer Serie von unabhängigen Versuchen mit zwei möglichen Ergebnissen *[4]*. Die Berechnung der Summe ist relativ rechenaufwendig. Zum einen führt die Fakultät in den einzelnen Summanden zu sehr großen Zahlen, zum anderen liefert dagegen die Berechnung der Potenzen sehr kleine. Damit die Summe exakt ausgerechnet werden kann, muss auf „BigDecimal" zur Zahlendarstellung zurückgegriffen werden *(siehe Listing 1)*.

Die Berechnung lässt sich durch die Parallelisierung der „for"-Schleife beschleunigen („Parallel-For-Pattern"). Hierzu werden der

```
final int n = ...;
final BigDecimal p = new BigDecimal("...");

BigDecimal res = BigDecimal.ZERO;
for(int k=0; k<=n; k++)
{
  res = res.add( Bin(n, k, p) );
}
```

Listing 1

```
final int n = ...;
final BigDecimal p  = new BigDecimal(...);
BigDecimal[] values = new BigDecimal[n+1];

// Parallele Berechnung der Einzelwerte
Arrays.parallelSetAll(values, k -> Bin(n, k, p) );

BigDecimal res = BigDecimal.ZERO;
for(BigDecimal bd : values )
{
   res = res.add(bd);
}
```

Listing 2

```
BigDecimal res = Arrays.stream(values)
                       .parallel()
                       .reduce( BigDecimal.ZERO,
                                BigDecimal::add );
```

Listing 3

Zahlenbereich, über den in der „for"-Schleife iteriert wird, aufgeteilt und entsprechende Task-Objekte (mit Rückgabe) erstellt. Danach werden die Task-Objekte an einen geeigneten „Executor" („Threadpool") zur Ausführung übergeben. Am Ende der Berechnungen werden die Teilergebnisse eingesammelt und addiert *(siehe Abbildung 3)*.

Eine Parallelisierung mithilfe von Executoren erfordert die Implementierung von sehr viel Verwaltungscode. Neben dem, was getan werden soll (Berechnung einer Teilsumme), muss auch, wie es getan

werden soll (Aufspaltung des Intervalls, Erzeugung der Task-Objekte, Einsammeln der Teilergebnisse und Berechnung der Gesamtsumme), explizit implementiert sein. Diese Vorgehensweise ist sehr fehleranfällig und somit wenig praxistauglich.

Parallele Arrays und Streams

Die in Java 8 neu eingeführten Konzepte vereinfachen die Implementierung paralleler Abläufe sehr stark. So wurde die Klasse „Arrays" um Methoden ergänzt, mit denen beispielsweise Arrays parallel initialisiert oder sortiert werden können. *Listing 2* zeigt, wie sich mithilfe einer parallelen Array-Initialisierung die Berechnung der Einzelwerte einer Binomialverteilung implementieren lässt.

Die Initialisierung des Arrays erfolgt hier durch eine deklarative Beschreibung. Der Lambda-Ausdruck gibt lediglich an, welche Vorschrift auf die einzelnen Array-Elemente angewendet werden soll. Der Vorteil bei dieser Vorgehensweise ist, dass die Parallelisierung der Initialisierung nun intern erfolgen kann und nicht explizit implementiert werden muss.

Eine analoge Vorgehensweise findet man auch bei der Stream-Verarbeitung. Streams bilden, grob gesagt, eine Abstraktion über iterierbare Datensammlungen. Die klassische Iteration über ein Array oder eine Collection entspricht dem imperativen Beschreibungsparadigma, die Art und Weise der Iteration ist also explizit im Code verankert. Im *Listing 2* entspricht dies der „foreach"-Schleife über die Array-Elemente. Bei Streams erfolgt die Iteration „intern" und die Verarbeitungsschritte werden wieder deklarativ, in der Regel durch Lambda-Ausdrücke, formuliert. Mithilfe des Stream-Mechanismus kann nun auch die Iteration über das Array (Summation) parallelisiert werden *(siehe Listing 3)*.

Bei der internen Parallelisierung kommt das „ForkJoin"-Framework zum Einsatz, das bereits mit Java 7 eingeführt wurde. Dabei wird das Problem, in unserem Fall der Datenbereich, rekursiv aufgeteilt, bis eine gewisse Anzahl, die von den zur Verfügung

```
final int n = ...;
final BigDecimal p  = new BigDecimal("...");
BigDecimal res = IntStream.rangeClosed(0, n)
                          .parallel()
                          .mapToObj( k -> Bin(n, k, p)
)
                          .reduce( BigDecimal.ZERO,
                                   BigDecimal::add );
```

Listing 4

```
CompletableFuture.supplyAsync( () -> sensor.getData()
)
                 .thenApplyAsync( data ->
                       /* Daten in Chart eintragen */,
                       Platform::runLater );
```

Listing 5

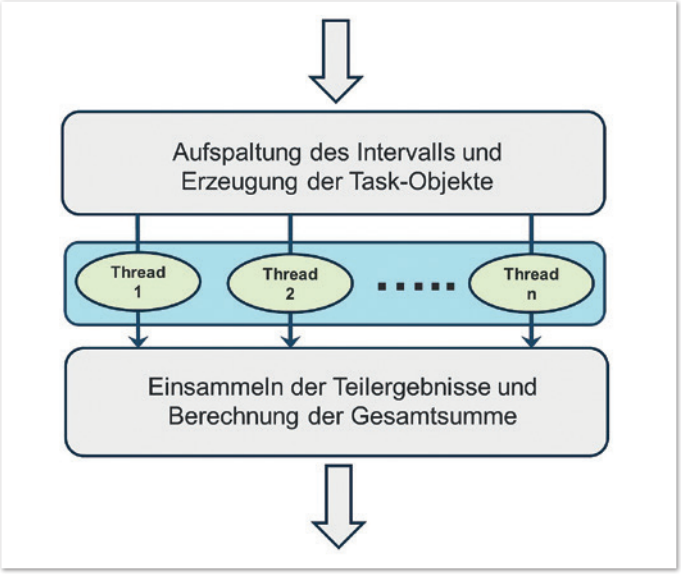


Abbildung 3: Schematische Umsetzung der Parallelisierung der „for"-Schleife

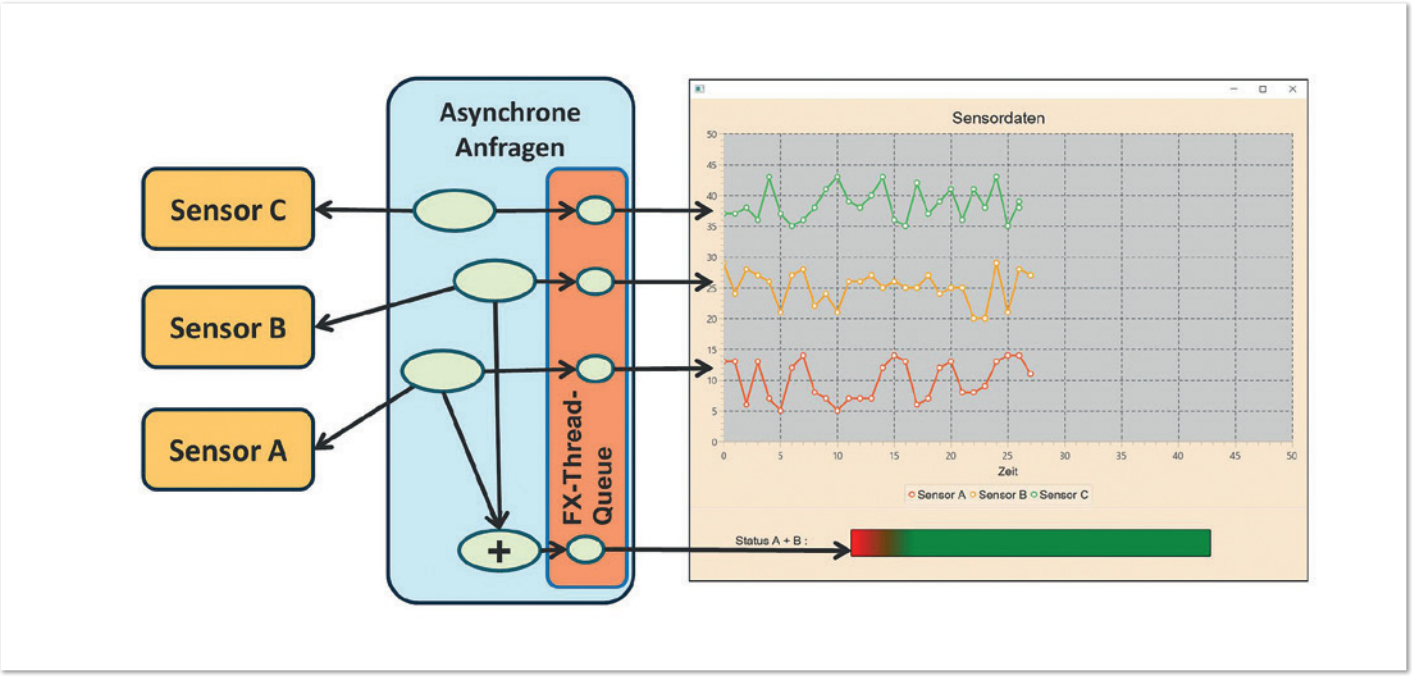


Abbildung 4: Darstellung verschiedener Sensorwerte (Liniendiagramm) und ihrer akkumulierten Werte (Farbbalken im unteren Bereich)

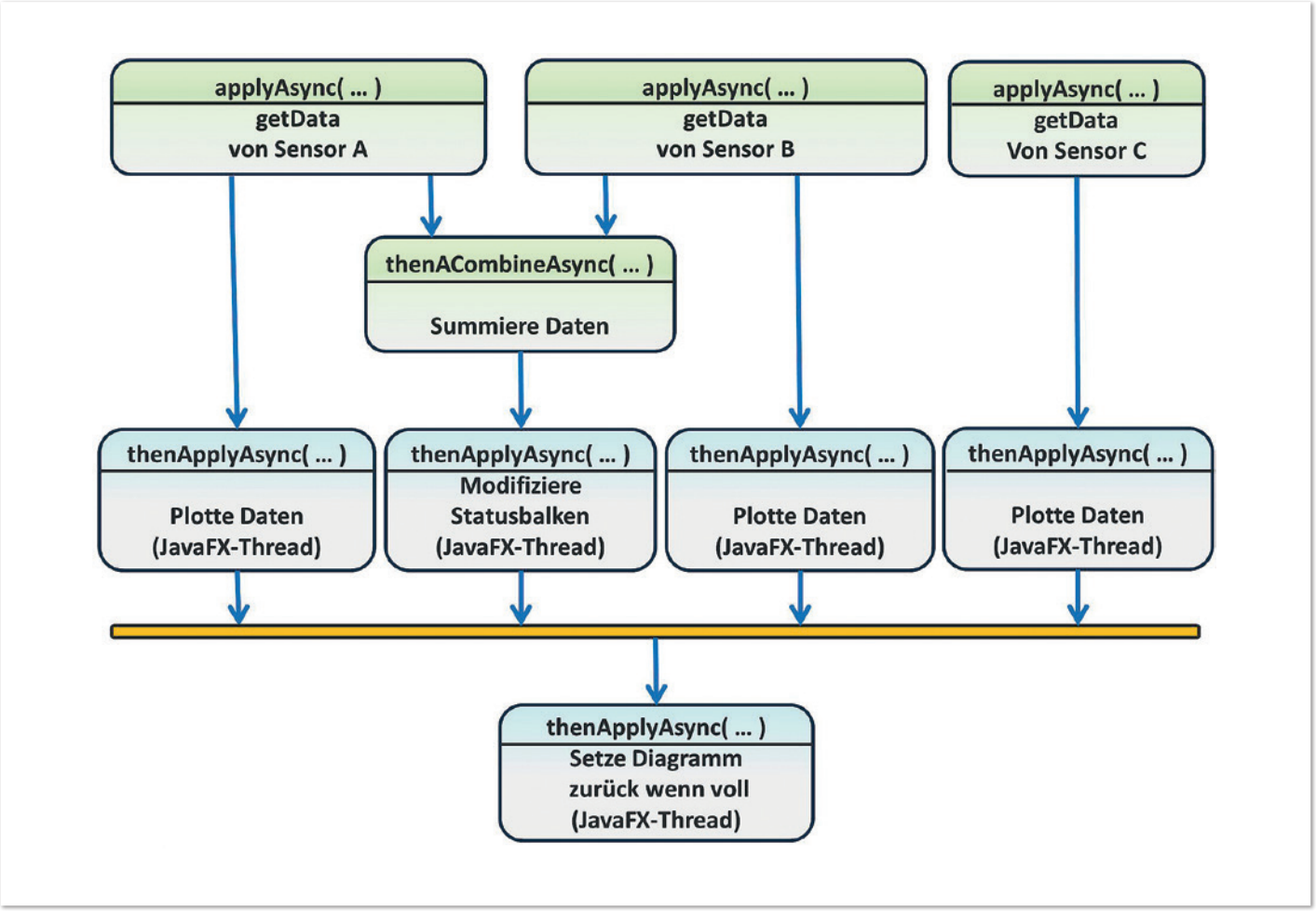


Abbildung 5: Konfiguration der asynchronen Datenverarbeitung mithilfe der „CompletableFuture"-Objekte

stehenden Rechenkernen abhängt, von Teilproblemen entstanden ist. Die Teilprobleme werden an die Rechenkerne verteilt und die Teilergebnisse wieder rekursiv zusammengesetzt. All dies übernimmt das Stream-Framework. Das hier betrachtete Beispiel kann auch ohne den Umweg über die Erstellung eines Arrays direkt mit Streams formuliert werden. *Listing 4* zeigt eine entsprechende Implementierung.

In dem gezeigten Beispiel wird der Stream durch eine Reduktion abgeschlossen, es wird also ein Wert ermittelt. Neben Reduktionen gibt es auch die Möglichkeit, Streams durch eine Sammlung abzuschließen. In diesem Fall erhält man als Ergebnis einen Datencontainer.

Auf den ersten Blick scheint der Einsatz dieser Parallelitäts-Konzepte sehr verlockend. Man muss aber doch einige Punkte dabei beachten *[5]*. Zum einen müssen die angewendeten Operationen (Lambda-Ausdrücke) frei von Seiteneffekten sein; die Operationen dürfen in der Regel also nur lesend auf gemeinsam benutzte Daten zugreifen beziehungsweise deren Status nicht unkoordiniert verändern.

Darüber hinaus lohnt sich der Einsatz einer Parallelisierung, die ja auch einen Overhead verursacht, nicht in jedem Fall. Es muss genug Arbeit für die einzelnen Kerne geben, der Datenbereich sollte also entsprechend groß sein und die ausgeführten Operationen müssen einen gewissen Rechenaufwand verursachen. Einen weiteren Einfluss auf die Effizienz der Parallelisierung haben die zugrunde liegende Datenquelle, die Art der Auswertung sowie das Speicher-Layout der Daten.

Nicht bei allen Datenquellen lässt sich die Iteration gut parallelisieren. Im Prinzip funktioniert das nur gut, wenn man einen wahlfreien Zugriff auf Elemente hat, wie bei Arrays. Wird ein Stream mit einer Datensammlung abgeschlossen, müssen im parallelen Fall die zugrunde liegenden Datencontainer gut vereinigt werden können. So ist das parallele Sammeln in einer „ArrayList" viel effizienter als das Sammeln in einem „HashSet". Ein ungünstiges Speicher-Layout der Streamdaten kann zu „cache misses" führen, was sich ebenfalls negativ auf die Laufzeit-Performance auswirkt.

Die Klasse „CompletableFuture"

Neben der parallelen Verarbeitung von Datensammlungen kann die „CompletableFuture"-Klasse parallele Verarbeitungsvorgänge implementieren. Dazu ein Beispiel: Eine JavaFX-Anwendung soll periodisch verschiedene Sensoren abfragen und deren Werte visualisieren *(siehe Abbildung 4)*.

Das periodische Abfragen der Sensoren, das in der Regel über das Netzwerk erfolgt, kann hier jeweils parallel stattfinden, wobei die eigentliche Datenvisualisierung dann allerdings vom JavaFX-Thread übernommen werden muss. *Listing 5* zeigt die asynchrone Abfrage von Sensordaten, die anschließend in ein JavaFX-Diagramm eingetragen werden.

```
// Abfragen der Sensordaten
CompletableFuture<Integer> cfVal1 =
     CompletableFuture.supplyAsync( () -> sensor1.
getData() );
CompletableFuture<Integer> cfVal2 =
     CompletableFuture.supplyAsync( () -> sensor2.
getData() );

// Daten in das Diagramm eintragen
...

// Daten von Sensor A und B addieren
// und Farbstatus des Balkens setzen
CompletableFuture<Boolean> cfPlotStatus =
    cfVal1.thenCombineAsync(cfVal2, (d1, d2) -> d1 +
d2 )
          .thenApplyAsync( data -> {
              /* Statusbar modifizieren */},
              Platform::runLater );
```

Listing 6

```
CompletableFuture.allOf(
   cfPlotVal1, cfPlotVal2, cfPlotVal3, cfPlotStatus)
                .thenRunAsync( () -> {
                   /* Lösche Daten,
                      wenn Diagramm voll ist */
            }, Platform::runLater).join();
```

Listing 7

Man beachte, dass der JavaFX-Thread die Übernahme der Daten in das Diagramm übernimmt. Man erkennt dies daran, dass bei der „thenApplyAsync"-Methode als zweiter Parameter ein expliziter Thread-Pool, nämlich der von JavaFX, angegeben wurde. Neben der Formulierung solch einfacher asynchroner Abläufe lassen sich auch Ergebnisse asynchron weiterverarbeiten. So werden beispielsweise in der Monitor-Anwendung die von Sensor A und B erhaltenen Werte zusätzlich akkumuliert dargestellt. Hierzu werden einfach die beiden Ergebnisse der Sensorabfragen, sobald sie vorliegen, entsprechend verknüpft. *Listing 6* zeigt eine Implementierung und *Abbildung 5* schematisch den asynchronen Verarbeitungsablauf.

Die Klasse „CompletableFuture" kennt zahlreiche Methoden, mit deren Hilfe sich auch komplexe asynchrone Abläufe spezifizieren lassen. Man kann dies dann wahlweise synchron oder asynchron tun und im asynchronen Fall auch einen Thread-Pool angeben, an den die Aufgabe delegiert wird. Zudem stehen auch Barrieren zur Verfügung, um mehrere Abläufe zu koordinieren. In *Listing 7* werden zum Beispiel alle asynchron ausgeführten Tasks an der „allOf"-Barriere synchronisiert und bei Bedarf die Daten aus der Anzeige gelöscht. Das komplette Beispiel steht im Web zur Verfügung *[6]*.

Auch im Umgang mit der „CompletableFuture"-Klasse gilt, dass die übergebenen Aktionen frei von Seiteneffekten sein müssen. Hat man erst mal die Funktionsweise der Klasse verinnerlicht, lassen sich mit „CompletableFuture" sehr einfach asynchrone Abläufe oder asynchrone APIs implementieren.

Fazit

Java biete heute zahlreiche Konzepte für die Implementierung von Multicore-Software. Neben umfangreichen Concurrency-Tools stehen den Entwicklern vor allem mit den Streams und der „CompletableFuture"-Klasse zwei mächtige Frameworks zur Implementierung paralleler Abläufe zur Verfügung. Der Artikel konnte die Konzepte und Möglichkeiten dieser Frameworks nur ansatzweise aufzeigen, ohne dass jeweils auf die genauen Funktionsdetails eingegangen wurde. Wer sich näher mit der Implementierung von Multicore-Software mit Java beschäftigen möchte, dem seien zwei Bücher *[1]* und *[2]* und eine Artikelserie *[3]* empfohlen. Um die modernen Parallelisierungskonzepte gewinnbringend einzusetzen, ist ein fundiertes Verständnis der nebenläufigen Programmierung in jedem Fall notwendig.

Literatur

[1] Brain Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea, Java Concurrency in Practice, Addison-Wesley, 2006

[2] Jörg Hettel und Manh Tien Tran, Nebenläufige Programmierung mit Java, dpunkt.verlag, 2016

[3] Klaus Kreft und Angelika Langer, Verschiedene Artikel zu Java-Concurrency: *http://www.angelikalanger.com/Articles/EffectiveJava.html*

[4] Binomialverteilung: *https://de.wikipedia.org/wiki/Binomialverteilung*

[5] Doug Lea et al, When to use parallel streams: *http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html*

[6] Sourcecode der Beispiele: *https://github.com/hettel/JavaAktuell*

Jörg Hettel
joerg.hettel@hs-kl.de

Jörg Hettel promovierte am Institut für Informationsverarbeitung und Kybernetik an der Universität Tübingen. Nach seiner Promotion war er als Berater bei nationalen und internationalen Unternehmen tätig. Er begleitete zahlreiche Firmen bei der Einführung von objektorientierten Technologien und übernahm als Software-Architekt Projektverantwortung. Seit 2003 ist er Professor an der Hochschule Kaiserslautern am Standort Zweibrücken.

Cloud Native Java – from Zero to Hero!

Christian Schwörer und Johannes Dilli, NovaTec Consulting GmbH

Für die meisten Unternehmen führt aktuell kein Weg am Thema „Cloud" vorbei. Neben der Fragestellung, wie sich dadurch die Bereitstellung und der Betrieb von Applikationen verändern, sind bei der Anwendungsentwicklung auch einige Rahmenbedingungen wichtig.

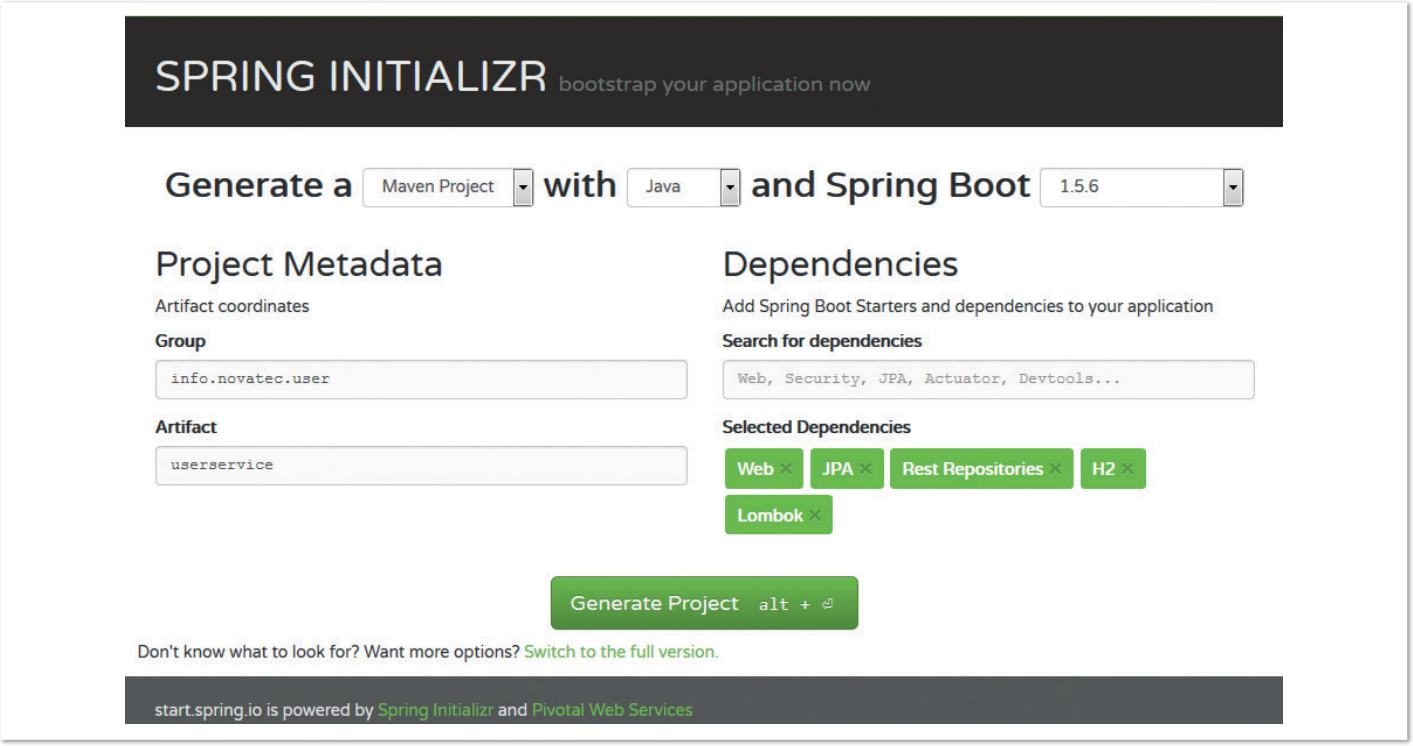


Abbildung 1: Bootstrapping des Users-Service mit dem Spring Initializr

Typischerweise werden im Zusammenhang mit der Cloud die Grundsätze der 12-Factor-Apps *[1]* genannt, auch wenn bisher keine allgemeingültige Definition dafür existiert, was unter dem Begriff „cloud native" zu verstehen ist. Das zugrunde liegende Architekturprinzip der 12-Factor-Apps basiert auf lose gekoppelten, individuell skalierbaren Microservices, um die Cloud-Topologien optimal nutzen zu können.

Mit Spring Boot und Spring Cloud stehen zwei aufeinander aufbauende JVM-Frameworks zur Verfügung, um Cloud-native Applikationen erstellen und betreiben zu können. Dieser Artikel zeigt, wie man mit diesen Frameworks ein Microservice-System entwickelt, um es anschließend in der Cloud-Foundry-Plattform in Betrieb zu nehmen.

Microservices mit Spring Boot

Spring Boot hat sich als das Framework zur effizienten Entwicklung von Microservices etabliert. Der Erfolg begründet sich auf zahlreiche Eigenschaften, um die Anwendungsentwicklung drastisch zu vereinfachen. Drei ganz wesentliche Aspekte dabei sind:

- **Erstellen von Stand-alone-Spring-Applikationen**
  Durch die einfache Möglichkeit, Servlet Container wie Apache Tomcat oder Jetty in die erstellte Anwendung einzubinden, ist kein zusätzlicher Web- oder Applikationsserver mehr erforderlich. Das paketierte JAR-Archiv beinhaltet alles, sodass lediglich eine JRE zur Ausführung notwendig ist.
- **Convention over Configuration (CoC)**
  Die konsequente Umsetzung dieses Paradigmas deckt mit sehr wenigen Annotationen einen Großteil der typischen Anwendungsfälle ab. Dies zeigt sich auch im später vorgestellten Demoprojekt: Hier liegt eine H2-Datenbank im Klassenpfad. Da keine andere Datenquelle definiert ist, folgt die Anwendung der Konvention, diese H2 zu verwenden.
- **Starter Dependencies**
  Die meisten Entwickler sind beim Einsatz von Build-Tools wie Apache Maven bereits durch die „Dependency Hell" gegangen. Spring Boot umgeht dies, indem Abhängigkeiten in sogenannten „Starter Dependencies" gekapselt sind, die alle erforderlichen Bibliotheken bündeln. So reicht es für die Verwendung des Java-Persistence-API aus, die Starter Dependency „spring-boot-starter-data-jpa" zu verwenden – die zugrunde liegenden Hibernate- und Spring-Data-Abhängigkeiten bleiben transparent.

Wie erwähnt, handelt es sich hierbei nur um einen Auszug aus den vielfältigen Vereinfachungen von Spring Boot. Weitere Vorzüge stehen in der zugehörigen Fachliteratur *[2, 3]*. Im Folgenden wird ein einfaches Demoprojekt mit Spring Boot Microservices entwickelt: Es soll im ersten Schritt möglich sein, Nutzer anzulegen, zu editieren und zu entfernen. Nutzer können Kommentare erstellen, bearbeiten und löschen. Somit sind zwei Microservices erforderlich: ein Users-Service und ein Comments-Service. Beide Dienste sollen RESTful sein, für die CRUD-Operationen sind also die HTTP-Methoden „get", „post", „put" und „delete" verfügbar.

Das Bootstrapping eines neuen Service erfolgt am einfachsten mit dem Spring Initializr *(siehe „https://start.spring.io"). Abbildung 1* zeigt die benötigten Starter Dependencies für den Users-Service. Die Web-Dependency wird für die Fullstack-Entwicklung mit Tomcat benötigt, JPA für die objektrelationale Persistierung und Rest Repositories für die Bereitstellung per REST. Für den rudimentären Service soll vorerst eine In-memory H2-Datenbank verwendet werden. Die Dependency zum Projekt Lombok dient zur Vermeidung von Boilerplate-Code *[4]*. Ein Klick auf „Generate Project" erzeugt ein ZIP-Archiv, das direkt in die IDE importiert werden kann. Nach dem Import lohnt sich ein erster Blick in die erzeugte Startklasse UserserviceApplication *(siehe Listing 1)*.

```
@SpringBootApplication
public class UserserviceApplication {
  public static void main(String[] args) {
    SpringApplication.run(UserserviceApplication.
class, args);
  }
}
```

Listing 1

```
@Entity
@Getter
@Setter
@NoArgsConstructor
class User {
    @Id
    @GeneratedValue
    private Long id;
    private String nickname;
    private String firstname;
    private String lastname;
}
```

Listing 2

```
@RepositoryRestResource
interface UserRepository extends JpaRepository<User,
Long> {}
```

Listing 3

```
@Getter
@Setter
@NoArgsConstructor
public class Comment {
    @Id
    private String id;
    private String text;
    private String userNickname;
}
```

Listing 4

Auffällig ist die Annotation „@SpringBootApplication". Damit wird die Anwendung gemäß dem CoC-Paradigma grundlegend konfiguriert. Ein User hat die Eigenschaften „Nickname", „Vorname" und „Nachname". Abgebildet wird dies in einer JPA-Entity, erweitert um das technische Attribut „id" *(siehe Listing 2)*. Die Persistierung des Users ist über ein Spring-Data-Repository realisiert *(siehe Listing 3)*.

Auch bei den Repositories zeigt sich die konsequente Realisierung des CoC-Paradigmas: Es ist lediglich erforderlich, vom Framework-Interface „JpaRepository" abzuleiten, und schon stehen die CRUD-Operationen für einzelne User oder auch Listen von Usern implizit zur Verfügung. Natürlich können im Bedarfsfall eigene, spezifische Operationen ergänzt werden.

Eine besondere Bedeutung kommt der Annotation „@RepositoryRestResource" zu: Darüber erhalten alle Repository-Operationen eine REST-Fassade und sind über die HTTP-Methoden aufrufbar. Somit ist der erste Microservice fertig und kann gestartet werden. Das erfolgt entweder direkt in der IDE durch Ausführen der

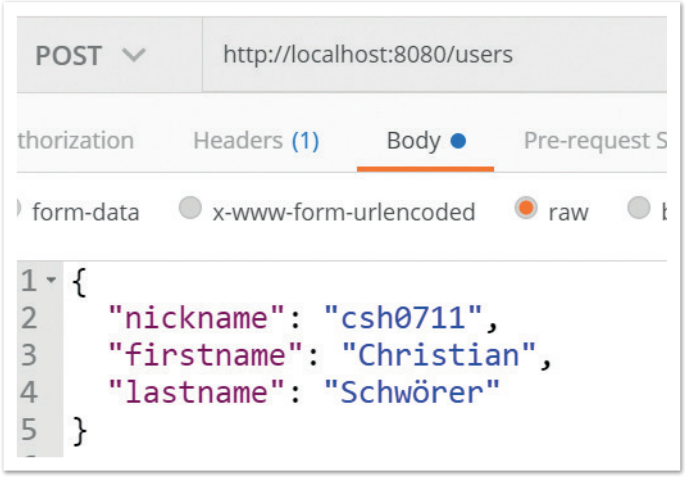


Abbildung 2: Erstellen eines Users per „POST"

Startklasse „UserserviceApplication" oder mit Maven über die Kommandozeile mit „mvn spring-boot:run". Dadurch wird die Anwendung auf dem Default-Tomcat-Port 8080 gestartet. Anschießend lässt sich, wie in *Abbildung 2* zu sehen, mit „POST" über den REST-Endpoint *„http://localhost:8080/users"* ein neuer User anlegen. *Abbildung 3* zeigt, wie per „GET"-Aufruf die Liste aller User ausgelesen wird.

Erwähnenswert ist, dass das erzeugte JSON bereits die Hypermedia-Links gemäß der HATEOAS-Struktur enthält, also dem höchsten Reifegrad von REST entspricht *[5]*. Auch Paginierung und Sorting der Ergebnislisten ist out of the box unterstützt. Ein einzelner User lässt sich folglich REST-konform über *„http://localhost:8080/users/1"* auslesen und gegebenenfalls per „PUT" ändern oder per „DELETE" entfernen.

Für den zweiten Service zur Verwaltung der Kommentare wird analog vorgegangen. Allerdings verwendet dieser Service anstelle einer relationalen Datenbank mit MongoDB eine NoSQL-Lösung. Nach der Erzeugung des Comments-Service Projekts mit dem Spring Initializr wird die Klasse Comment erstellt *(siehe Listing 4)*. Im Anschluss daran wird auch hierfür ein Spring Data Repository definiert, das nun aber vom Interface MongoRepository ableitet *(siehe Listing 5)*.

Somit ist auch dieser Microservice fertiggestellt und lauffähig. Da Port 8080 bereits vom Users-Service belegt ist, wird der Comments-Service auf Port 8081 ausgeführt. Dazu ist die Konfigurationsdatei „application.yml" zu ergänzen *(siehe Listing 6)*. Nach dem Start können per „POST" ein Kommentar hinzugefügt und mit „GET" alle Kommentare ausgelesen werden *(siehe Abbildungen 4 und 5)*.

Neben den beiden vorgestellten sind weitere Microservices denkbar. So kann zum Beispiel ein Images-Service zum Bild-Upload ergänzt werden. Für die Umrechnung der Bilder, etwa in eine Fullsize- und eine Thumbnail-Größe, kommt ein technischer Image-Resizing-Service zum Einsatz. Zudem ist ein Recommendations-Service vorstellbar, bei dem User Empfehlungen abgeben können. Als Service Consumer kann beispielsweise eine Angular-4-Single-Page-Application (SPA) alle Dienste zusammenführen, sodass sich der in *Abbildung 6* gezeigte Aufbau ergibt.

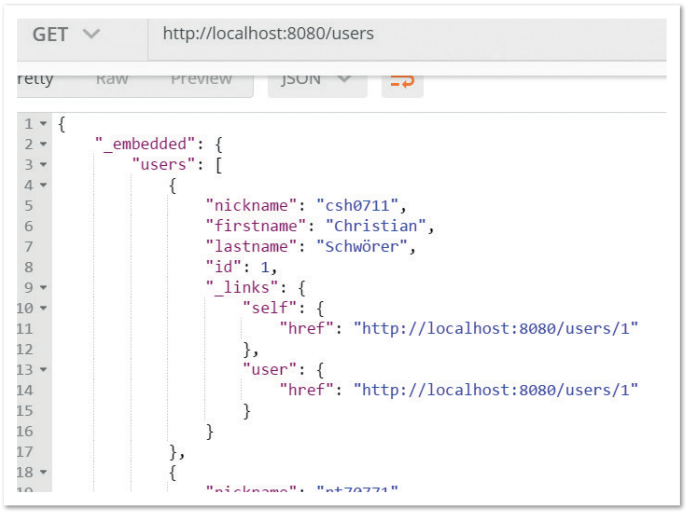


Abbildung 3: Ermitteln aller User per „GET"

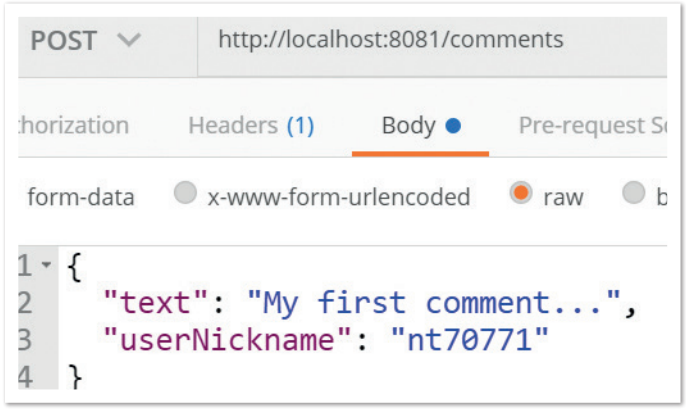


Abbildung 4: Erstellen eines Kommentars per „POST"

```
@RepositoryRestResource
interface CommentRepository extends
MongoRepository<Comment, String> {
}
```

Listing 5

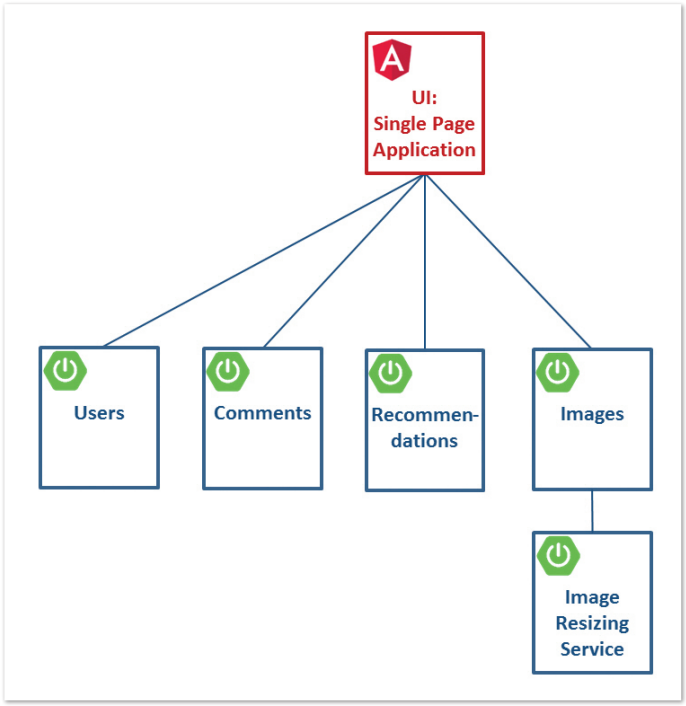


Abbildung 6: Struktur der Spring Boot Microservices mit Angular-SPA

```
server:
  port: 8081
```

Listing 6

```
@EnableZuulProxy
@SpringBootApplication
public class ZuulEdgeServiceApplication {
  public static void main(String[] args) {
  SpringApplication.run(ZuulEdgeServiceApplication.
class, args);
  }
}
```

Listing 7

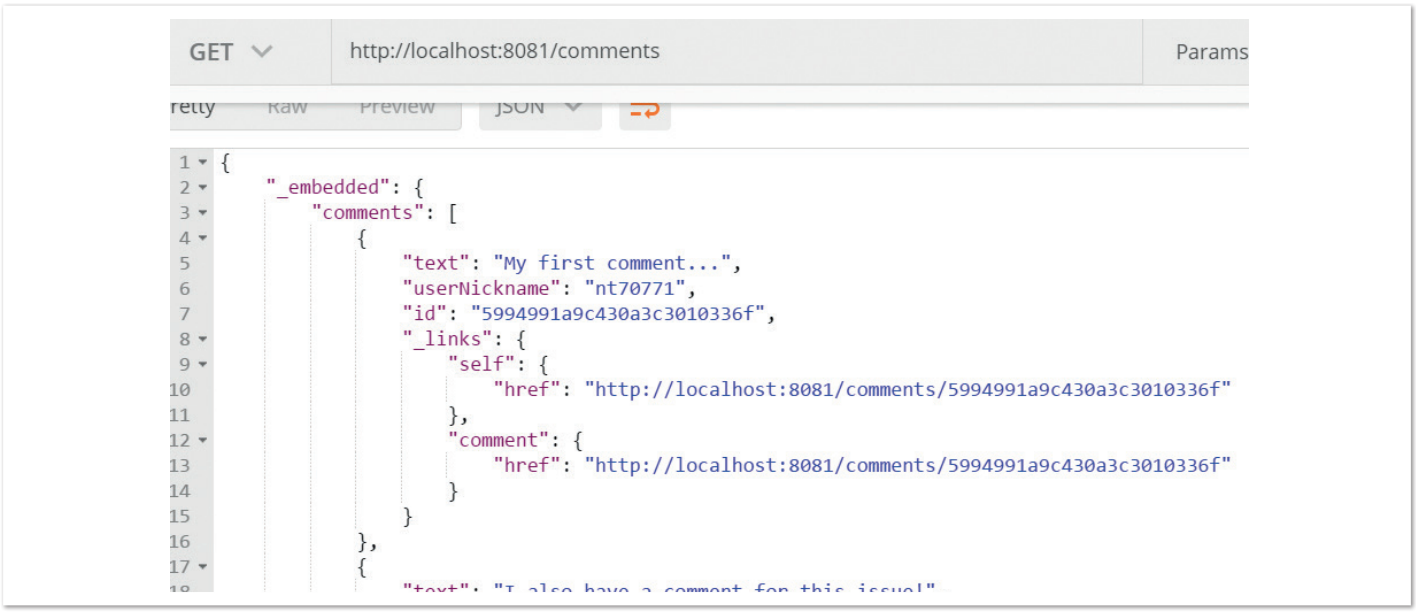


Abbildung 5: Auslesen aller Kommentare per „GET"

Edge-Service mit Zuul

Nachteil dieses Aufbaus ist, dass die Angular-SPA nun die Adressen aller Microservices kennen muss. Dies führt zu mehreren Problemen: Zum einen müssen alle Services öffentlich erreichbar sein; zum anderen verhindert die Same-Origin-Policy den Zugriff per JavaScript auf andere Webseiten, also auch auf die REST-Endpoints anderer Microservices *[6]*.

Abhilfe schafft hier ein Edge-Service, auch als API-Gateway bekannt. Er dient als Reverse-Proxy und bietet die APIs der einzelnen Microservices unter einem einheitlichen Zugangspunkt an. Zusätzlich können Filter definiert werden, die auf die gerouteten Requests angewendet werden. Spring Cloud Netflix bietet Zuul als JVM-basierten Edge-Service *[7]*. Der Edge-Service setzt auf Spring Boot auf und kann ebenfalls mit dem Spring Initializr erstellt werden kann. Es wird nur eine Dependency benötigt: Zuul. Um diese beim Anwendungsstart zu aktivieren, ist „@EnableZuulProxy" der Startklasse hinzuzufügen *(siehe Listing 7)*.

Das Mapping einer URL des Zuul-Edge-Service zu einem Microservice wird in der „application.yml" eingetragen *(siehe Listing 8)*. Es sorgt dafür, dass alle Anfragen an *„http://localhost:8888/users"* zu *„http://localhost:8080/users"* geroutet werden. Kommt Zuul in Kombination mit Eureka (siehe nächstes Kapitel) zum Einsatz, sind die Routen nicht mehr als statische URLs einzutragen, sondern es reicht der Name der Anwendung. Zuul löst diesen mithilfe von Eureka anschließend zu einer Adresse auf *(siehe Listing 9)*.

Service Discovery mit Eureka

Um es Microservices zu ermöglichen, sich gegenseitig zu erreichen, muss die Adresse des aufgerufenen Service bekannt sein. Sollen die Services in der Cloud betrieben werden, ist eine statische Zieladresse jedoch nur noch bedingt nutzbar.

Zu den Eigenschaften der Cloud zählt es, dass Anwendungen jederzeit beendet und auf einer anderen (virtuellen) Maschine erneut gestartet werden können. Zusätzlich werden bei Last neue Instanzen einer Applikation gestartet oder bei geringer Last auch wieder beendet. Dieses Verhalten ermöglicht keine feste Zuordnung zwischen Service Consumer und Service Provider.

Trotzdem müssen Anwendungen untereinander kommunizieren können. Um sicherzustellen, dass sich Applikationen trotz der dynamischen Skalierung innerhalb der Cloud gegenseitig finden, ist eine Service Registry erforderlich. Sie stellt ein Adressbuch dar: Anhand des Namens einer Applikation lässt sich deren aktuelle Adresse ermitteln. Wird eine neue Anwendung gestartet, teilt sie dies der Service Registry mit. Wird sie beendet, so wird sie wieder aus der Registry entfernt.

Eureka ist eine Service-Registry-Implementierung von Netflix. Spring Cloud Netflix bietet eine einfache Integration des Eureka-Servers. Um diesen zu erstellen, muss im Spring Initializr die Dependency „Eureka-Server" gewählt, der Startklasse muss anschließend „@EnableEurekaServer" hinzugefügt und schon kann der Eureka-Server gestartet werden *(siehe Listing 10)*.

Um es einzelnen Microservices zu ermöglichen, sich in die Service Registry einzutragen und Daten aus dieser abzufragen, ist in den

```
server:
  port: 8888
zuul:
  routes:
    users:
      path: /users/**
      url: http://localhost:8080/users/
    comments:
      path: /comments/**
      url: http://localhost:8081/comments/
```

Listing 8

```
zuul:
  routes:
    users:
      path: /users/**
      serviceId: Users-Service
    comments:
      path: /comments/**
      serviceId: Comments-Service
```

Listing 9

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
  public static void main(String[] args) {  SpringAp-
plication.run(EurekaApplication.class, args);
  }
}
```

Listing 10

```
spring:
  application:
    name: Users-Service
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Listing 11

```
applications:
  - name: Users-Service
    buildpack: java_buildpack
    path: target/userservice-0.0.1-SNAPSHOT.jar
    memory: 512MB
    instances: 1
    routes:
     - route: users-service.cfapps.io
```

Listing 12

Microservice-Projekten mit „Eureka Discovery" eine weitere Dependency zu ergänzen. Damit sich eine Anwendung am Eureka-Server registrieren kann, müssen der Anwendung ihr eigener Name und die Adresse des Eureka-Servers bekannt sein. Beides kann über die „application.yml" gesetzt werden *(siehe Listing 11)*.

Die zusätzliche Annotation „@EnableEurekaClient" in der Startklasse aktiviert den Eureka-Client und sorgt dafür, dass sich der Microservice beim Starten selbstständig beim Eureka-Server mit

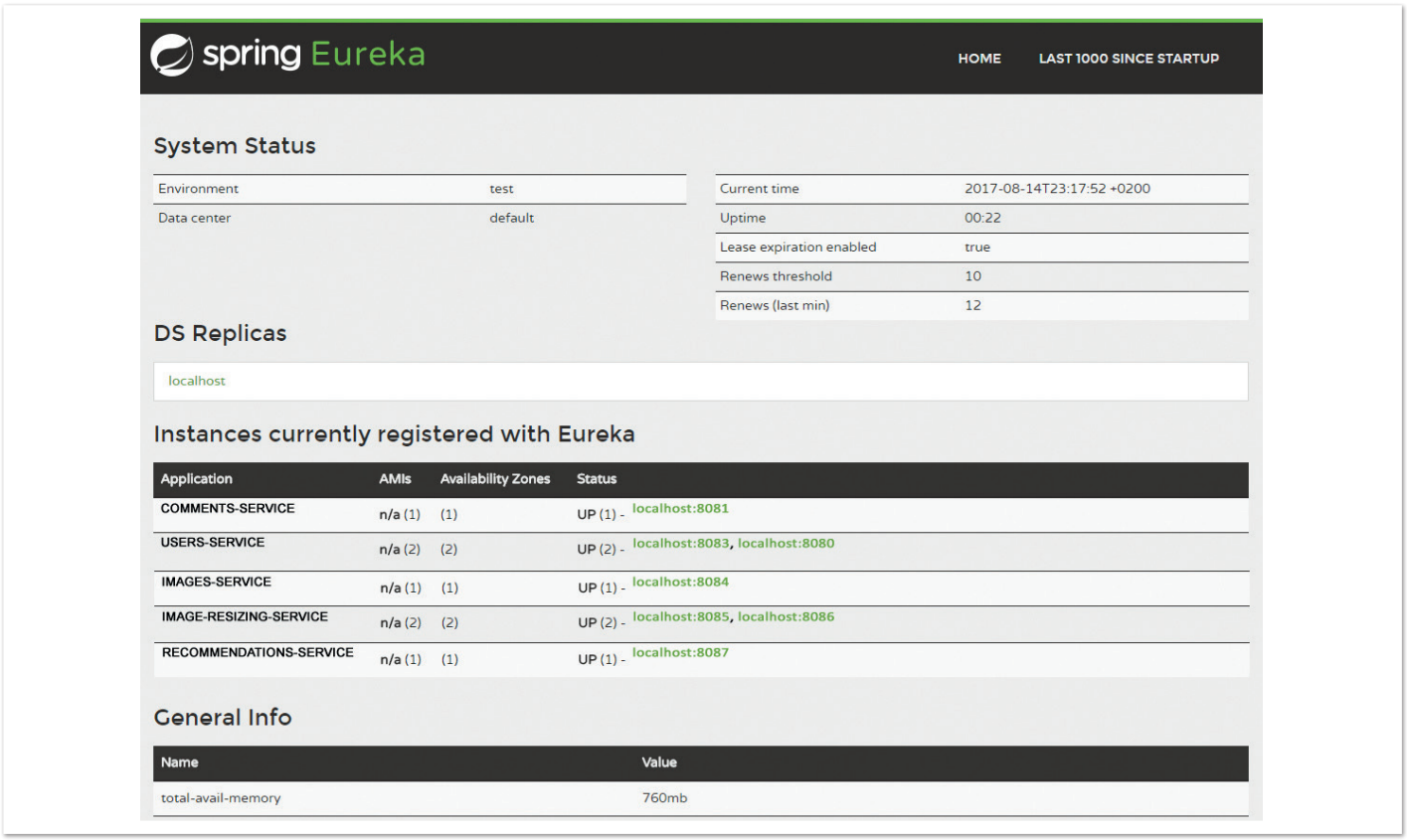


Abbildung 7: Eureka-Web-Oberfläche mit allen registrierten Microservices

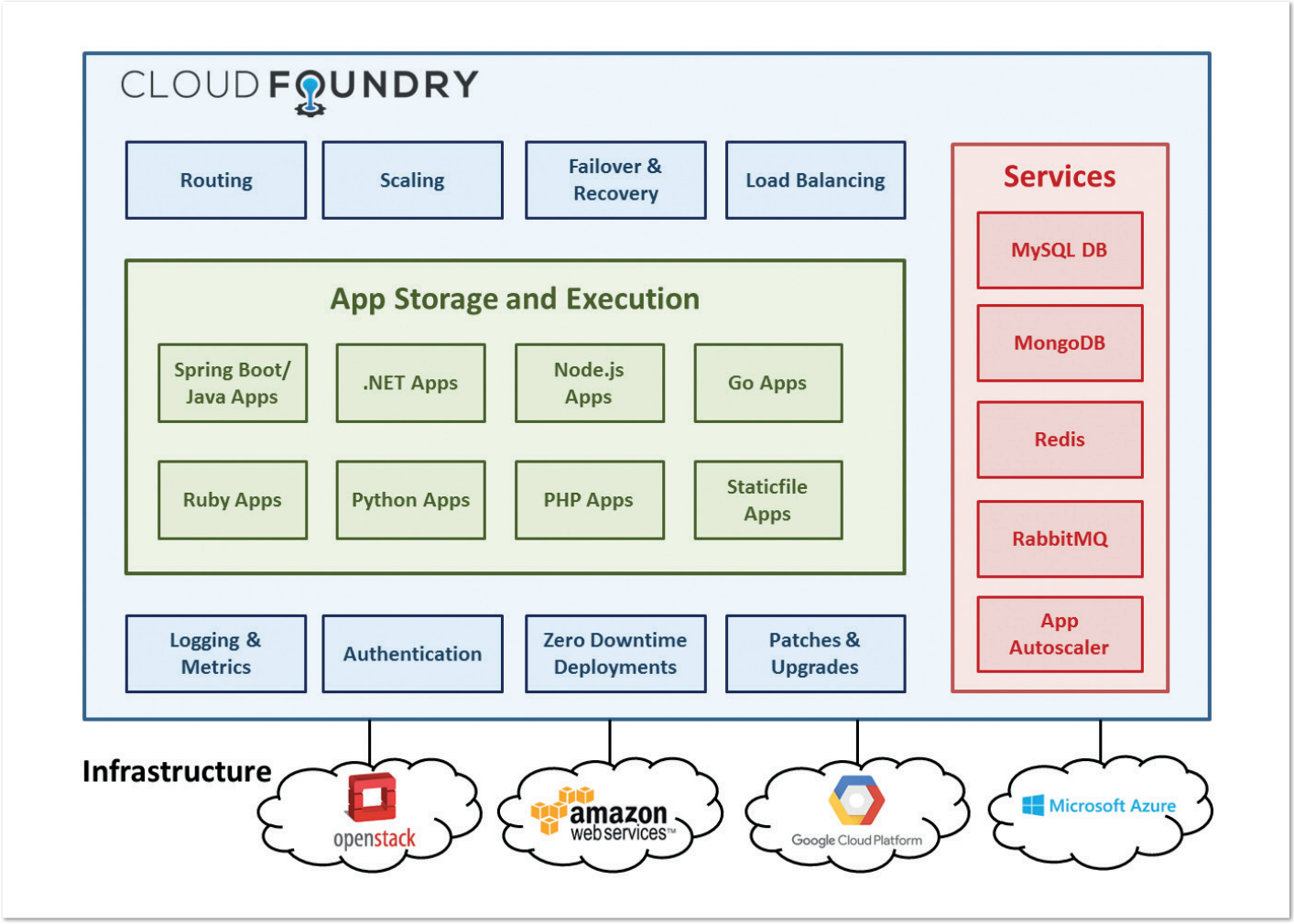


Abbildung 8: Grundlegende Komponenten von Cloud Foundry

seiner aktuellen Adresse registriert. Beim Beenden meldet sich die Anwendung wieder ordnungsgemäß vom Eureka-Server ab. Da ein geordnetes Beenden einer Anwendung nicht immer sichergestellt werden kann, markiert Eureka Anwendungen nach einer gewissen Zeit selbstständig als nicht mehr verfügbar. Um zu verhindern, dass Services fälschlicherweise als nicht verfügbar markiert werden, müssen diese regelmäßig ein Heartbeat-Signal an Eureka senden. Eureka bietet eine Web-Oberfläche, auf der alle registrierten Anwendungen mit ihren Instanzen sichtbar sind, sodass man sich leicht eine Übersicht über alle aktiven Microservices machen kann. Wie *Abbildung 7* zeigt, können unter einem Namen auch mehrere Instanzen eines Microservice registriert sein.

Cloud Foundry als Cloud Native PaaS

Spring Boot und Spring Cloud erhöhen die Produktivität, indem sie es dem Entwickler ermöglichen, sich auf die Anwendungserstellung zu konzentrieren und sich nicht mit technischen Rahmenbedingungen beschäftigen zu müssen. Wie bekommt man nun eine Applikation möglichst schnell und einfach in eine produktive Cloud-Umgebung, sodass die gewonnene Effizienz bei diesem Schritt nicht wieder verloren geht? Hier kommt Cloud Foundry als Platform as a Service (PaaS) für Cloud-native Anwendungen ins Spiel. Die Entwicklung der quelloffenen Plattform wird von der Cloud Foundry Foundation geleitet *[8]*.

Es gibt mehrere Public-Cloud-Foundry-Provider, darunter Pivotal Web Services, IBM Bluemix und SAP Cloud Platform *[9]*. Aber natürlich ist es ebenfalls möglich, eine private On-Premise-Plattform aufzubauen. *Abbildung 8* zeigt den schematischen Aufbau der wesentlichen Cloud-Foundry-Komponenten. Da an dieser Stelle nur eine vereinfachte Darstellung möglich ist, wird für Details auf weiterführende Literatur verwiesen *[10]*.

Elementarer Bestandteil jeder Cloud-Foundry-Instanz ist die Komponente „App Storage and Execution", die für das Vorhalten und die Ausführung der eingerichteten Applikationen verantwortlich ist. Daneben stellt die Plattform eine ganze Reihe weiterer Funktionalitäten zur Verfügung: Neben dem Routing gehört die Möglichkeit, Anwendungen zu skalieren, dazu. Ebenso sind Failover- und Recovery-Strategien implementiert, die den Re-Start von Anwendungen sicherstellen, falls diese ungeplant beendet werden. Auch das generelle Load Balancing zwischen verschiedenen App-Instanzen übernimmt die Plattform. Darüber hinaus bietet Cloud Foundry Logging sowie Metriken zur Anwendungsüberwachung und unterstützt Praktiken zur Ausfallszeit-Minimierung bei Deployments.

Die meisten Anwendungen benötigen bestimmte Dienste: Die zuvor beschriebenen Users-Service und Comments-Service erfordern zum Beispiel eine relationale Datenbank beziehungsweise MongoDB. Diese Dienste werden vom Betreiber der Cloud-Foundry-Instanz zur Verfügung gestellt und können über die Plattform durch Service-Bindings von der Anwendung genutzt werden.

Letztendlich benötigt jede Cloud-Foundry-Instanz eine Infrastruktur, auf der sie betrieben wird. Hier ist neben den üblichen Cloud-Computing-Anbietern auch das Aufsetzen einer eigenen Cloud-Infrastruktur möglich, beispielsweise mit OpenStack. Aus Entwicklersicht zeichnet sich Cloud Foundry durch seine Anwenderfreundlichkeit aus. So gibt es neben einer grafischen Web-Oberfläche mit der Cloud Foundry CLI einen einfach nutzbaren Kommandozeilen-Client, über den Apps eingerichtet, skaliert und verwaltet werden *[11]*.

Mit dem CF-CLI-Befehl „cf push" wird eine Anwendung in einer Cloud-Foundry-Instanz eingerichtet. Soll beispielweise der Users-

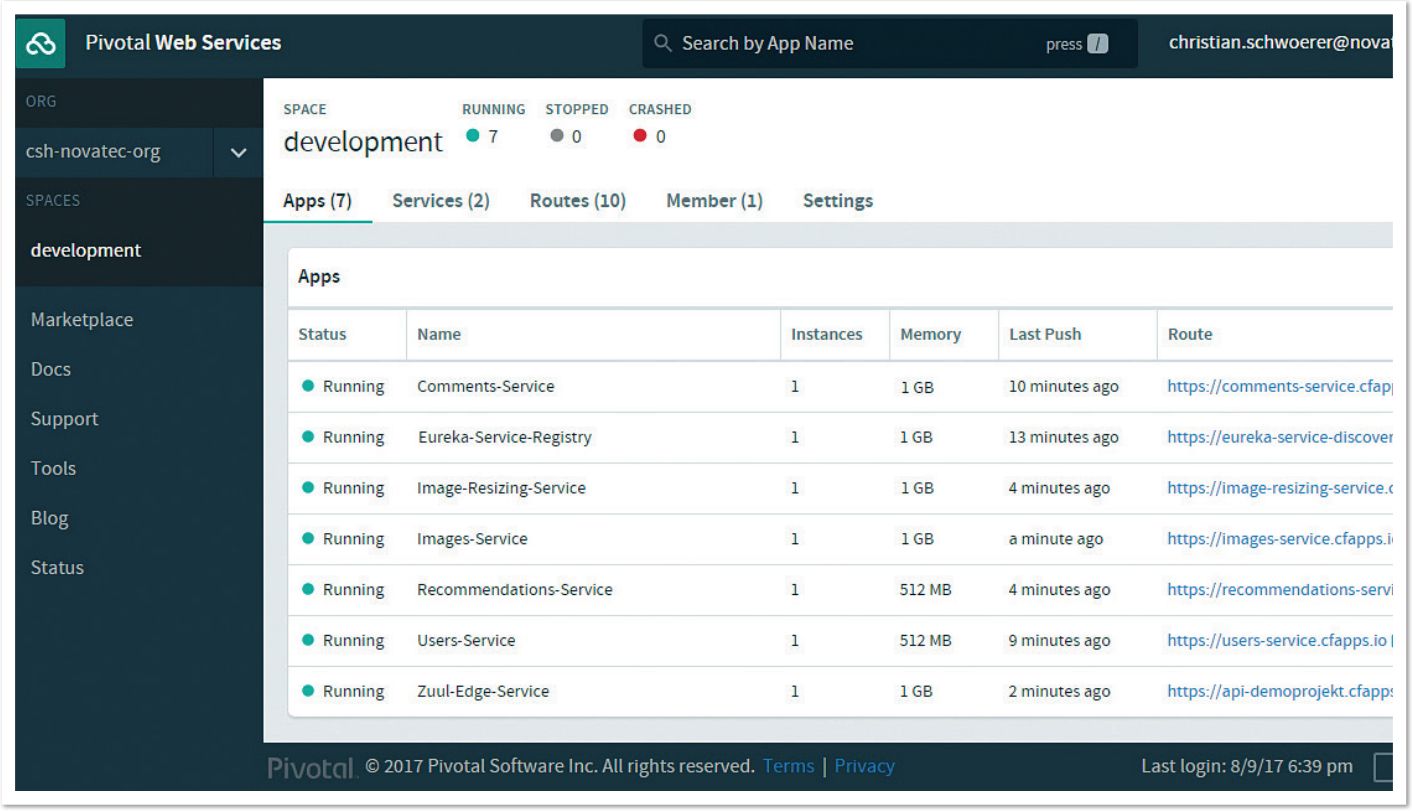


Abbildung 9: Die in Pivotal Web Services gepushten Apps

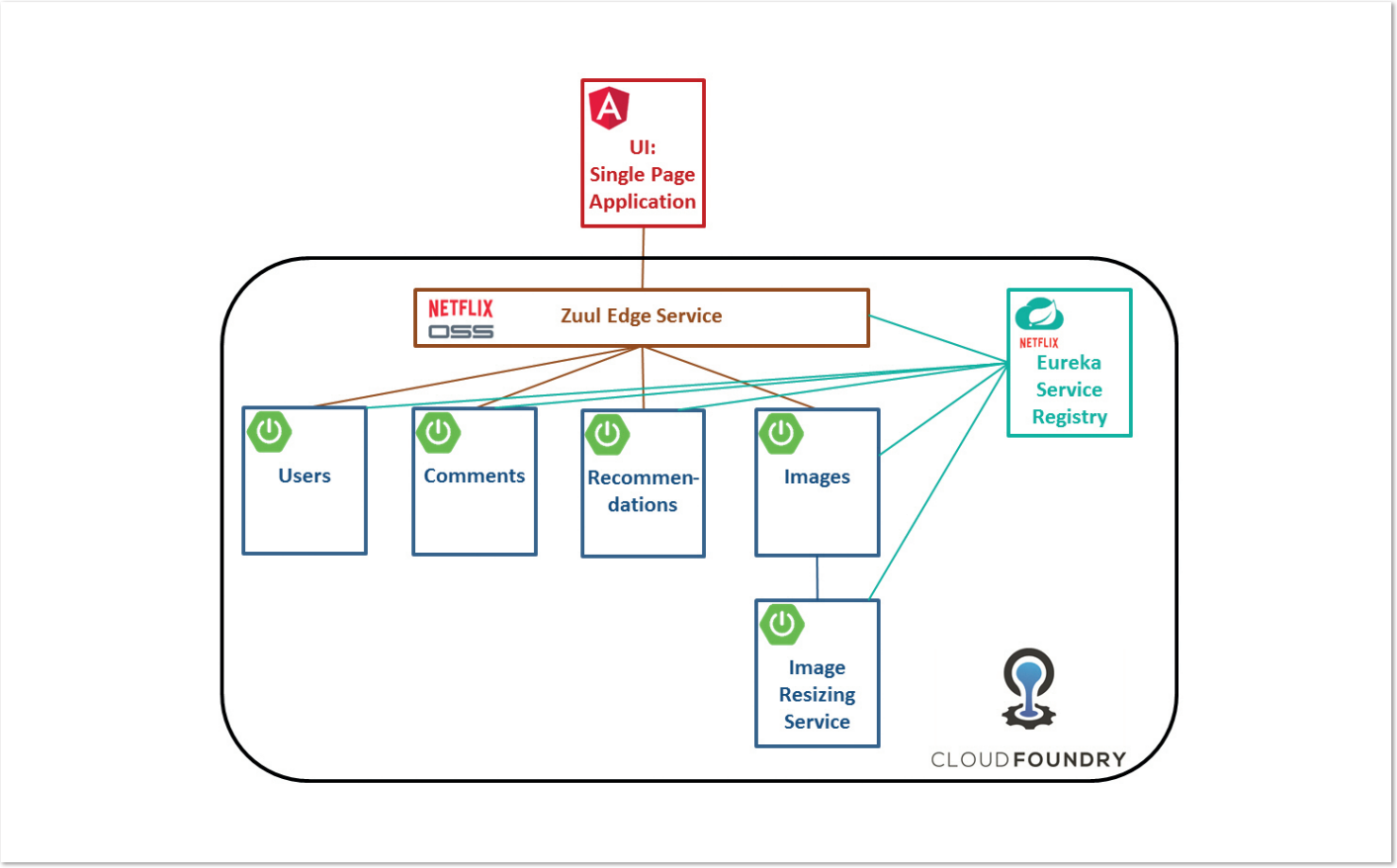


Abbildung 10: Cloud-native Microservice-Architektur des Demoprojekts

Service aus Abschnitt 1 gepusht werden, lautet der Befehl „cf push Users-Service –p .\target\userservice-0.0.1-SNAPSHOT.jar". Mit dessen Ausführung wird zunächst das JAR hochgeladen. Sobald die Cloud-Foundry-Instanz die Datei erhält, sucht sie ein passendes Buildpack – für Spring Boot das Java-Buildpack. Ein Buildpack ist vereinfacht gesagt ein Verzeichnis mit Skripten, die auf das Deployable angewendet werden. Es gibt unterschiedliche Buildpacks für die verschiedensten Sprachen und Technologien wie „.net", „node.js", „Go", „Python", „Ruby" oder PHP.

Das Java-Buildpack sorgt unter anderem dafür, dass das aktuelle OpenJDK heruntergeladen und aus JAR und JDK ein sogenanntes Droplet erzeugt wird. Dieses Droplet wird als Linux-Container im Cloud Foundry Cluster gestartet. Anstatt die Parameter bei jedem „cf push" mitzugeben, empfiehlt es sich, diese in der „manifest.yml" im Projektverzeichnis abzulegen *(siehe Listing 12)*.

Ist diese Datei vorhanden, reicht ein einfaches „cf push" aus, und die Parameter werden ausgelesen. Im obigen Beispiel sind das neben dem Anwendungsnamen der Pfad zum JAR, das zu verwendende Buildpack, der zuzuordnende Hauptspeicher, die Anzahl der zu startenden App-Instanzen und die Route beziehungsweise URL, unter der die App erreichbar sein soll. Nachdem alle Microservices aus unserem Demoprojekt auf diese Weise gepusht wurden, stehen sie als Apps in der Cloud bereit *(siehe Abbildung 9)*.

Noch verwenden die Apps keine von Cloud Foundry bereitgestellten Dienste: Der Users-Service nutzt beispielsweise noch die eingebettete H2. Das ist natürlich für ein produktives Cloud-Szenario in keiner Weise ausreichend – insbesondere wenn mehrere Instanzen einer App gleichzeitig betrieben werden. Demzufolge sollte ein Provided Service mit einer relationalen Datenbank genutzt werden.

Der Befehl „cf marketplace" listet alle Services auf, die im Marktplatz der Plattform verfügbar sind. Nutzt man die Pivotal Web Services, gibt es dort mit dem Dienst „ClearDB" eine MySQL-Datenbank. Eine Service-Instanz, die von der App verwendet werden kann, wird über die Befehle „cf create-service cleardb spark users-db" und „cf bind-service Users-Service users-db" angelegt.

Zunächst wird die Service-Instanz erzeugt. Als Parameter wird nach dem Namen des Provided Service („cleardb") der Serviceplan vorgegeben („spark2). Der Serviceplan ist Dienst-spezifisch; je nach Lizenz-Modell stehen verschiedene Pläne zur Verfügung. Die verfügbaren Pläne eines Dienstes sind im Marketplace aufgeführt. Als letzter Parameter wird der Name der zu erstellenden Serviceinstanz angegeben („users-db").

Der zweite Befehl bindet die erzeugte „users-db" an die App Users-Service. Da keine andere Datenquelle definiert ist, nutzt der Spring-Boot-Microservice dem CoC-Paradigma entsprechend die nun gebundene Service-Instanz. Genauso wird auch für den Comments-Service vorgegangen – in diesem Fall mit „mLab" als MongoDB as a Service.

Neben vielen weiteren Befehlen bietet die CF CLI auch die Möglichkeit, Applikationen zu skalieren: Eine horizontale Skalierung, also der Betrieb von beispielsweise vier parallelen Instanzen, erfolgt über den Befehl „cf scale Users-Service -i 4". Die Erhöhung des Speichers für jede laufende Instanz, also eine vertikale Skalierung, wird mit „cf

scale Users-Service -m 1024M" vorgenommen. Damit steht nun das Demoprojekt mit allen Microservices, dem Zuul-Edge-Service sowie dem Eureka-Server in der Cloud-Foundry-Instanz zur Verfügung. *Abbildung 10* zeigt die finale Cloud-native Microservice-Architektur.

Weitere Funktionen

Abschließend wird noch auf einige andere Komponenten des Spring-Cloud-Frameworks eingegangen, die eine weitergehende Umsetzung des Cloud-Native-Ansatzes erlauben *[12]*:

- **Spring Cloud Config**
  Anstelle die Konfigurationen der Microservices in Properties-Files wie der „application.yml" zu verwalten, kann ein zentraler Server verwendet werden, von dem sich alle Anwendungen ihre jeweilige Konfiguration laden. Die Properties werden versioniert in einem Git-Repo verwaltet und bei einer Änderung benachrichtigt der Config-Server alle Anwendungen etwa per RabbitMQ, sodass diese jeweils die aktuelle Konfiguration erhalten *[13]*.
- **Spring Cloud Sleuth**
  Werden in einem verteilten System mehrere Microservices hintereinander aufgerufen und es tritt ein Fehler auf, so kann sich die Fehlerauswertung schwierig gestalten. Sleuth fügt jeder Nachricht eine eindeutige ID hinzu, die für einen Aufruf über Systemgrenzen gleich bleibt, und schreibt diese ID in alle Log-Nachrichten. Anhand dieser ID lässt sich eine Nachricht in den Logs erkennen und nachverfolgen *[14]*.
- **Zipkin**
  Für eine grafische Darstellung von systemübergreifenden Aufrufen und deren Laufzeiten bietet sich Zipkin an. Es verwendet die eindeutigen IDs von Sleuth, um Nachrichten zu korrelieren und den Service-übergreifenden Verlauf darzustellen *[15]*.

Aktuell entsteht mit dem Spring-Cloud-Gateway ein vielversprechender Nachfolger für Zuul. Das erste Release, das auf Spring 5, Spring Boot 2.0 und der Reactive-Bibliothek Reactor basiert, ist für Dezember 2017 angekündigt *[16]*. Schlussendlich noch die Anmerkung, dass – je nach Cloud-Foundry-Provider – die vorgestellten Dienste Eureka und Cloud Config bereits durch den Anbieter, also als Provided Services, zur Verfügung stehen können und nicht selbst erstellt werden müssen *[17]*.

Fazit

Zusammenfassend lässt sich festhalten, das Spring Boot und Spring Cloud die einfache Entwicklung und Bereitstellung von lose gekoppelten Microservices ermöglichen. Cloud Foundry als PaaS erlaubt den effizienten Cloud-Betrieb im besten DevOps-Sinne. Zusätzlich steht mit der CF CLI eine Schnittstelle zur Realisierung von Continuous Delivery zur Verfügung. Mit all diesen Komponenten lässt sich somit der Cloud-native Ansatz umfassend abbilden.

Weiterführende Informationen

[1] *https://12factor.net*
[2] Josh Long & Kenny Bastani, Cloud Native Java – Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry, O'Reilly Media, 2017
[3] Craig Walls: Spring Boot in Action, Manning, 2015
[4] *https://projectlombok.org*
[5] *https://spring.io/understanding/HATEOAS*
[6] *https://de.wikipedia.org/wiki/Same-Origin-Policy*
[7] *https://github.com/Netflix/zuul*
[8] *https://www.cloudfoundry.org/members*
[9] *https://www.cloudfoundry.org/certified-platforms*
[10] Duncan Winn, Cloud Foundry, The Definitive Guide – Develop, Deploy, and Scale, O'Reilly Media, 2017
[11] *https://github.com/cloudfoundry/cli*
[12] *http://blog.novatec-gmbh.de/spring-cloud-sprint-a-fast-and-comprehensive-spring-cloud-services-tutorial beziehungsweise http://blog.novatec-gmbh.de/service-discovery-eureka-cloud-foundry*
[13] *https://cloud.spring.io/spring-cloud-config*
[14] *http://cloud.spring.io/spring-cloud-sleuth*
[15] *http://zipkin.io*
[16] *https://spring.io/blog/2017/07/06/spring-cloud-finchley-m1-is-available beziehungsweise https://github.com/spring-cloud/spring-cloud-gateway*
[17] *http://docs.pivotal.io/spring-cloud-services/1-4/common*

Christian Schwörer
christian.schwoerer@novatec-gmbh.de

In seiner Tätigkeit für die NovaTec Consulting GmbH begleitet Christian Schwörer in verschiedenen Kundenprojekten den Wandel von der klassischen JEE-Welt hin zu verteilten, Cloud-basierten Microservice-Architekturen. Er ist der Überzeugung, dass sich die praktische Auseinandersetzung mit dem Thema für jeden Entwickler lohnt – auch wenn Microservices nicht die Silver Bullet für alle Probleme des Software Engineering sein werden.

Johannes Dilli
johannes.dilli@novatec-gmbh.de

Als Consultant für die NovaTec Consulting GmbH beschäftigt sich Johannes Dilli schon lange mit dem Spring Framework und weiß dessen Stärken zu schätzen. Durch die Einfachheit von Spring Boot erlebt Spring in den letzten Jahren einen Boom. Er sieht in Spring – insbesondere durch Projekte wie Spring Cloud und die gute Integration mit Cloud Foundry – ein sehr geeignetes Framework zur Entwicklung von Microservices.

Praxisbuch Usability und UX

gelesen von Dennis Stritzke

Als Entwickler kommen wir mit vielen Aspekten der Gestaltung in Kontakt, dabei handeln wir im besten Fall jedoch nach vagen Vorgaben, nach der eigenen Intuition oder gar nach dem eigenen Geschmack. Jens Jacobsen und Lorena Meyer bieten in ihrem „Praxisbuch Usability und UX" einen differenzierten und methodischen Ansatz für die Gestaltung von gebrauchstauglichen Anwendungen und deren Nutzungserlebnis.

Im ersten Teil legen sie die Grundlage, um über Usability und UX zu sprechen, dabei grenzen sie die Begriffe ab und belegen diese Abgrenzung mit Beispielen sowie sinnvollen deutschen Übersetzungen: die Gebrauchstauglichkeit und das Nutzungserlebnis.

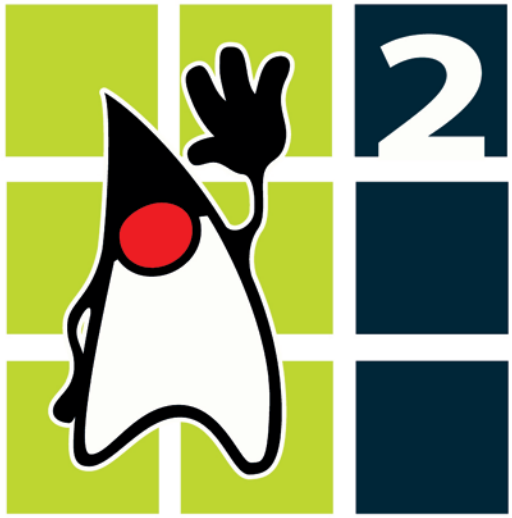
Der zweite Teil vermittelt uns die Methodik, um ein System zu gestalten, dies aus verschiedenen Blickwinkeln zu betrachten und beurteilen zu lassen, unter anderem sind „Personas", „Wireframes" und „A/B-Tests" systematisch vorgestellt. Damit eignet sich dieser Teil besonders für (agile) Produktteams, denen eine Gesamtverantwortung übertragen wurde. Durch sinnvolle Wahl der Methoden kann jedoch jeder Projektkontext von diesem Buchteil profitieren.

Die Autoren schließen das Buch mit einem 240 Seiten starken Abschnitt über konkrete Richtlinien und Empfehlungen zu einzelnen Bereichen von Oberflächen und Abläufen ab. Dabei gehen sie von der Navigationslogik über die Gestaltung einzelner Seiten bis hin zur Inhaltsebene einer Anwendung. Da wir als Entwickler in vielen Fällen keine konkreten Details zur Gestaltung erhalten, ist dieser Abschnitt besonders für unsere tägliche Arbeit relevant. Die Autoren vermitteln die Richtlinien für den Standardfall und erklären die relevanten Rahmenbedingungen, sodass wir selbst entscheiden können, was uns zum gesetzten Ziel führt.

Fazit: Jacobsen und Meyer haben ein Arbeitsbuch geschaffen, das viele Richtlinien, Denkanstöße und ein strukturiertes Herangehen an die Usability bietet. Die im Titel enthaltene User Experience wird allerdings kaum behandelt, was der Ausführlichkeit der Usability sehr zugutekommt. Das Buch eignet sich weniger dazu, in einem Stück gelesen zu werden, als dazu, immer wieder zum Nachlesen für eine aktuelle Herausforderung herangezogen zu werden.

Dennis Stritzke
dennis@stritzke.me

Titel:	**Praxisbuch Usability und UX**
Autor:	**Jens Jacobsen, Lorena Meyer**
Verlag:	**Rheinwerk Computing**
Umfang:	**511 Seiten**
Preis:	**39,90 Euro**
ISBN:	**9783836244237**



„Die Entwicklung von Java sollte in eine unabhängige Stiftung ausgelagert werden …“

Usergroups bieten vielfältige Möglichkeiten zum Erfahrungsaustausch und zur Wissensvermittlung unter den Java-Entwicklern. Sie sind aber auch ein wichtiges Sprachrohr in der Community und gegenüber Oracle. Wolfgang Taschner, Chefredakteur der Java aktuell, sprach darüber mit Gregor Trefs von der majug.

Wie ist die majug organisiert?

Gregor Trefs: Die Mannheimer Java User Group (majug) wurde im Jahr 2008 von Alexander Hanschke als Interessengruppe gegründet. Damals war die Veranstaltung eher studentisch an der Universität Mannheim angesiedelt. Mit der Zeit änderten sich das Publikum und verschiedene Personen wie Oliver Gierke und Lars Vogel halfen in der Organisation mit. Momentan besteht das Organisationsteam aus fünf Leuten: Mark, Markus, Walery, Wolfgang und ich. Wir führen die majug gleichberechtigt und demokratisch. Das bedeutet,

Zur Person:
Gregor Trefs

Gregor ist freiberuflicher Software-Entwickler. Sein erstes Programm war ein in BASIC geschriebenes Text Adventure. Er ist Mitorganisator der Java User Group Mannheim (majug) und schreibt regelmäßig auf seinem Blog *„https://gtrefs.github.io".*

dass Entscheidungen mehrheitlich getroffen werden, etwa die Bewertung von Vortrags- und Workshop-Vorschlägen, aber auch unsere Organisationsform betreffend. So sind wir zwar kein Verein, jedoch seit dem Jahr 2015 Mitglied des iJUG-Verbundes. Wir kündigen unsere Veranstaltungen über Meetup an und sind weiterhin an der Universität Mannheim.

Was sind die Ziele der majug?

Gregor Trefs: Für uns, als Leitung der majug, ist es wichtig, dass wir Themen anbieten, die uns und unsere Besucher interessieren. Damit wollen wir die Möglichkeit schaffen, Fähigkeiten und Wissen aufzubauen und Besucher für Java und die JVM zu begeistern.

Wie viele Veranstaltungen gibt es pro Jahr?

Gregor Trefs: Für Vorträge treffen wir uns ungefähr einmal im Monat, zumeist an der Universität Mannheim. Weiterhin veranstalten wir ein- bis zweimal im Jahr Workshops. Hierbei unterstützen uns örtliche Unternehmen mit Räumlichkeiten und Verpflegung. Zuletzt hat Michael Plöd einen Workshop zum Thema „Domain-Driven Design" gehalten.

Was bedeutet Java für euch?

Gregor Trefs: Für viele von uns ist Java die Sprache, die wir hauptsächlich beruflich einsetzen und am längsten kennen. Für manche ist sie sogar die erste Programmiersprache. Die Vertrautheit mit der Syntax erlaubt ein einfacheres Ausprobieren von Konzepten wie funktionaler Programmierung. Dadurch werden die Grenzen von Java erfahrbar und die Wege anderer Sprachen nachvollziehbarer.

Welchen Stellenwert besitzt die Java-Community für euch?

Gregor Trefs: Mark war Mitglied der Expert Group für CDI 2.0 (JSR 365). Außer ihm stehen wir eher auf der konsumierenden Seite des JCP, finden es jedoch prinzipiell gut, dass solch ein Prozess existiert und die Mitgestaltung der Sprache möglich ist. Oft führt dies zu viel Feedback und sehr offenen Diskussionen, leider aber auch zu Verzögerungen bei Uneinigkeit. Projekt Jigsaw ist ein gutes Beispiel dafür. Für Java 7 angekündigt, wurde es erst im zweiten Public Review Ballot im JCP für Java 9 akzeptiert.

Wie sollte sich Java weiterentwickeln?

Gregor Trefs: Ich bin mit der aktuellen Entwicklung sehr zufrieden. Es gibt eine vieldiskutierte Hypothese in der Linguistik, die besagt, dass die Sprache das Denken formt. Ich glaube, dass dies auch für Programmiersprachen gilt. Die Elemente und grundlegenden Konzepte einer Programmiersprache bestimmen, wie wir die Lösungen eines Problems gestalten. In Java ist deshalb der imperative Denkansatz vorherrschend. Lambdas und die Typen „Optional", „Stream" und „CompletableFuture" sind erste Möglichkeiten, Seiteneffekte wie asynchrone Berechnungen funktionaler zu gestalten. Java 9 erweitert die APIs der Typen und bringt mit dem Flow-API reaktive Programmierung in die Standard-Bibliothek. Im Rahmen von Projekt Amber werden interessante Erweiterungen wie Pattern Matching *(siehe „http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html")* und lokale Typinferenz *(siehe „http://openjdk.java.net/jeps/286")* diskutiert. Es fehlt die Möglichkeit, bestehende Klassen um neue Funktionalität zu erweitern. In C# und Kotlin gibt es hierfür Extensions und in Scala implizite Konvertierungen.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?

Gregor Trefs: Die Entwicklung von Java sollte in eine unabhängige Stiftung ausgelagert werden. Damit lassen sich Interessenkonflikte besser lösen und eine gewisse Neutralität wird gewahrt. Bei der Kommunikation und Transparenz der einzelnen JSRs gibt es Verbesserungsbedarf. Oft gibt es bei weniger aktiven JSRs über einen längeren Zeitraum kein Update. Beim Review sind dann Themen enthalten, an denen die Community nicht partizipiert hat, sondern die durch den Spec Lead im Alleingang eingebracht wurden.

Wie sollte sich die Community gegenüber Oracle verhalten?

Gregor Trefs: Die Community sollte mehr Neutralität seitens Oracle einfordern. Der JCP und auch die Java-9-Entwicklungen haben gezeigt, dass die Oracle-Interessen einen großen Einfluss auf den Prozess hatten. Auch Java EE ist sehr stark von Oracle und weniger von den Bedürfnissen geprägt.

info@majug.de
http://www.majug.de

iJUG Verbund

Alle Mitglieder auf einen Blick

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

www.ijug.eu

Impressum

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

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

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

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
Chefredakteur (ViSdP): Wolfgang Taschner
Kontakt: redaktion@doag.org

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt; Daniel van Ross, FIZ Karlsruhe; André Sept, Freiberufler; Jan Diller, Triestram und Partner

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

Fotonachweis:
Titel: Original © Freepik/freepik.com
S. 8: © BillionPhotos.com/Fotolia
S. 14: © Andrey Armyagov/123RF
S. 18: © PHUCHONG CHOKSAMAI/123RF
S. 22: © belchonock/123RF
S. 27: © Bakhtiar Zein/123RF
S. 37: © Natalia Bratslavsky/123RF
S. 48: © Rostislav Zatonskiy/123RF
S. 54: © Kenny Kiernan/123RF

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

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

Inserentenverzeichnis

<DevCamp 2018/>

30. - 31. Januar in Hamburg

DOAG

devcamp.doag.org ->


JavaLand 2018
13. - 15. März 2018 in Brühl bei Köln
Ab sofort Ticket & Hotel buchen!
www.javaland.eu
Programm online!
JavaLand