02-2017 | Sommer | www.ijug.eu

Java aktuell

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java im Mittelpunkt

Big Data
Predictive Analytics mit Apache Spark

Microservices
Lagom, das neue Framework

Achtung, Audit
Nutzung und Verteilung von Java SE

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

4 191978 304903 02

iJUG Verbund

JavaLand 2018

im Phantasialand | Brühl bei Köln

Save the Date

JavaLand 2018

20.-22. März

Wolfgang Taschner
Chefredakteur Java aktuell

Java steht im Mittelpunkt

Als vom Frühjahr 1991 bis Sommer 1992 unter der Federführung von James Gosling rund zehn Entwickler die Urversion von Java fertiggestellt hatten, konnte sich keiner der Protagonisten vorstellen, dass daraus einmal eine der wichtigsten Programmiersprachen entstehen würde. Heute nutzen weltweit mehr als neun Millionen Entwickler Java. Es ist jedoch nicht nur die Sprache Java, die sich so weit verbreitet hat; drumherum ist ein riesiges Biotop an Frameworks, Bibliotheken, APIs etc. entstanden.

Diese Ausgabe bietet einen guten Querschnitt durch die Java-Landschaft, beispielsweise mit dem Galen-Framework, das Layout-Testen leicht macht und dabei das moderne, responsive Webdesign berücksichtigt, oder mit Thymeleaf, einer neuen Template-Engine für Entwickler und Designer.

Vom aktuellen Big-Data-Geschehen handelt der Artikel „Predictive Analytics mit Apache Spark". Es geht um Verfahren für die Extraktion von Informationen aus Daten mit dem Ziel, aus den gewonnenen Informationen Trends oder Verhaltensmuster abzuleiten. Dazu passend haben zwei Autoren ihre Erfahrungen beim Aufsetzen eines Big-Data-Projekts beschrieben. Ein weiterer Artikel stellt mit Redis eine vielseitige und leichtgewichtige NoSQL-Datenbank vor.

Microservices sind ebenfalls in aller Munde. Mit Lagom betritt ein weiterer Player die Arena, in der sich schon Frameworks wie Spring Boot, Dropwizard und andere tummeln. Der Name „Lagom" stammt aus dem Schwedischen und meint „gerade richtig" oder „nicht zu viel, nicht zu wenig". Apache Kafka 101 hingegen ist ein Message Broker, dessen Architektur die Verarbeitung von Datenströmen mit sehr hohem Nachrichten-Durchsatz bei niedrigen Latenzen ermöglicht.

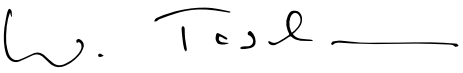
Auch Best Practices kommen in dieser Ausgabe nicht zu kurz. Ein Entwickler packt seine IntelliJ-IDEA-Trickkiste aus und ein anderer zeigt das JSON Schema-Driven Development mit JSSD. Ein erfahrener API-Designer berichtet, auf welche Entwurfsziele man bei Application Programming Interfaces achten sollte.

Java ist zwar Open Source, doch man darf es nicht immer frei benutzen und verteilen. Wenn ein Oracle-Auditor ins Haus kommt, sollten die Fallstricke bekannt sein. Markus Karg von der Java User Group Goldstadt hat sie zusammengestellt.

Überhaupt, was wäre Java ohne Community? Wie in jeder Ausgabe gibt es auch ein Interview mit einer Java User Group, diesmal mit der EuregJUG Maas-Rhine.

Ich wünsche Ihnen viel Spaß beim Lesen,

Ihr



Oracle behält sich das Recht vor, Kunden zu überprüfen

Mit Predictive Analytics lassen sich aus gewonnenen Informationen Trends oder Verhaltensmuster ableiten

Microservices sind in aller Munde und an entsprechenden Frameworks herrscht kein Mangel

Das Java-Tagebuch

Andreas Badelt, stellvertretender Leiter der DOAG SIG Java

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 vierten Quartal 2016.

27. September 2016

Interview mit Adam Bien: „Es ist noch nichts gestoppt"

Die Java-EE-Koryphäe Adam Bien hat sich in einem Interview zu den aktuellen Entwicklungen rund um den Standard und zu seinen Eindrücken von der JavaOne geäußert: „Es ist noch nichts gestoppt", sagt er mit Blick auf den von Oracle angekündigten Rauswurf von MVC, JMS 2.1 und Management API 2.0 aus EE 8. Über die Community-Umfrage könne man noch viel erreichen. Darüber hinaus empfindet er „das Fehlen von „Concurrency Utilities" auf den JavaOne-Slides als durchaus kritischer". Die geplante Aufnahme von Health Checking und Configuration in EE 8 begrüßt Bien. Interessant ist seine Aussage zum Verhalten des Herstellers: „Oracle verhält sich ein wenig wie Apple – es gibt sehr lange kein Feedback, die Mitarbeiter hören aber genau zu." Daher der Rat, neben der Teilnahme an der Umfrage auch verstärkt zu bloggen, und Artikel zu schreiben oder die Java EE Guardians zu unterstützen.

http://www.ijug.eu/home- ijug/aktuelle-news/article/interview-mit-adam-bien-es-ist-noch-nichts-gestoppt.html

29. September 2016

ARM-Quellcode für OpenJDK 9 freigegeben

Oracle gibt den Quellcode der JDK-9-Implementierungen (32 und 64 Bit) für ARM-Architekturen frei, auf denen unter anderem der Raspberry Pi basiert. Vor gut zwei Jahren hatte Henrik Ståhl von Oracle in seinem Blog noch gesagt, dass man neben Support-Einnahmen für Java auf „general purpose"-Plattformen insbesondere Lizenz-Einnahmen für Java-Embedded-Ports nutze, um die Java-Entwicklung zu finanzieren. Eine Freigabe des Codes sei daher erst denkbar, wenn ARM sich quasi als „general purpose"-Plattform weiterverbreite. Scheinbar ist dieser Moment jetzt gekommen. Ein bisschen kompliziert wird es aber schon, weil sich mit den AArch64- und AArch32-Projekten in den vergangenen Jahren zwei Open-Source-Implementierungen für Linux auf ARM entwickelt haben. Sie sind für das OpenJDK 8 entstanden (für das die Oracle-Freigabe nicht gilt), aber zumindest der AArch64-Port ist auch für OpenJDK 9 so gut wie fertig. Ein Merge dürfte das Entwicklerteam zumindest herausfordern. Die grundsätzliche Begeisterung bei den OpenJDK -Entwicklern ist daher, mit Verweis auf den Zeitpunkt, etwas gedämpft.

http://mail.openjdk.java.net/pipermail/aarch32-port-dev/2016-August/000406.html

4. Oktober 2016

NetBeans 8.2

NetBeans 8.2 ist da – für reine Java-Entwickler ist diesmal nicht viel dabei, vielleicht mit Ausnahme der verbesserten Docker-Unterstützung. Die größten Neuerungen gibt es bei JavaScript (Support von ECMAScript 6 und experimentell 7) sowie JavaScript-Frameworks (Node.js, Oracle JET).

https://netbeans.org/community/releases/82/

8. Oktober 2016

Ein paar Neuigkeiten vom JDK 9

Eine eher unscheinbare Sache, aber trotzdem eine Herausforderung: Mit JEP 299 („Java Enhancement Proposal") sollen die 22 (Linux) beziehungsweise 25 (Solaris) verschiedenen Sätze von Dokumenten des OpenJDK vereinheitlicht werden, also alle API-Dokus unter „docs/api/" und die Manual Pages unter „src/{share,}/man". War der Leidensdruck wirklich so groß oder ist das ein gutes Zeichen hinsichtlich der anderen Arbeiten am JDK 9? Neues gibt es auch von VisualVM. Das Java-Troubleshooting-Tool hat in der Version 1.3.9 jetzt experimentellen Support für das JDK 9. In Zukunft wird es jedoch nicht mehr mit dem Oracle JDK ausgeliefert, sondern muss von der projekteigenen Website bezogen werden. Diese ist mit dem geplanten Abschalten von java.net inzwischen auf GitHub migriert.

https://visualvm.github.io/

11. Oktober 2016

NetBeans-Team jetzt auf Slack

Für alle Slack-Nutzer oder diejenigen, die noch keinen Grund kennen, es auszuprobieren: Das NetBeans-Team ist jetzt auf Slack erreichbar.

https://netbeans.signup.team/

13. Oktober 2016

JCP-EC-Kandidaten stellen sich vor – die Zweite

Nach der öffentlichen Sitzung auf der JavaOne gibt es heute eine (ebenfalls öffentliche) Telefonkonferenz mit den Kandidaten für die Wahl zum JCP Executive Committee vom 1. bis 14. November. Es sind leider nicht alle Kandidaten vertreten, trotzdem ist es eine ziemlich große Runde – zum Glück auch viele, die selbst nicht kandidieren. Es ist nicht ganz einfach, den iJUG in der knapp bemessenen Redezeit vorzustellen (angeblich zwei Minuten, aber gefühlt waren es nur Sekunden). Parallel arbeiten wir mit Hochdruck an einem offiziellen Wahlprogramm. Das soll allerdings nur wenige zentrale Aussagen umfassen, sonst liest es eh niemand.

https://jcp.org/en/whatsnew/elections

19. Oktober 2016

Verschiebung von Java 9 ist bestätigt

Die bereits im Vorfeld der JavaOne angekündigte Verschiebung von Java 9 ist jetzt von Mark Reinhold bestätigt worden. Nach dem neuen Zeitplan ist der „Final Release Candidate" (letzter Schritt vor der „General Availability") für den 6. Juli 2017 geplant.

http://mail.openjdk.java.net/pipermail/jdk9-dev/2016-September/004887.html

30. Oktober 2016

Das Wahlprogramm des iJUG

Gerade rechtzeitig vor der Wahl hat sich der iJUG auf die Kernpunkte verständigt, die er im Executive Committee vorantreiben möchte; sie sind nun auf den Wahlseiten für alle sichtbar. Die drei großen Bereiche sind: „Offenheit und Transparenz" – unter anderem verbesserter Repository-Zugang, insbesondere für Expert-Group-Mitglieder (was eigentlich ja selbstverständlich sein sollte) und die Veröffentlichung von TCKs. Die „Trennung von Standards und kommerziellen Interessen" – hier geht es nicht nur um rechtliche und Prozess-Verbesserungen, sondern auch darum, mehr Spec Leads zu haben, die den Interessen der Community verpflichtet sind. Und „Mehr Demokratie", insbesondere eine Stärkung der Rechte der Expert Groups und des Executive Committee gegenüber Spec Leads, die ihre Rolle nicht (mehr) ausreichend wahrnehmen.

http://www.ijug.eu/home-ijug/aktuelle-news/article/ijug-bewirbt-sich-fuer-sitz-im-executive-committee-des-jcp.html

18. November 2016

JCP-Executive-Committee-Wahlen beendet

Es hat nicht ganz gereicht. Der iJUG ist bei den EC-Wahlen am „Cut" gescheitert, hat aber immerhin aus dem Stand 8 Prozent der Stimmen geholt. Ich gratuliere den sechs gewählten Kandidaten: der Eclipse Foundation und der London Java Community (je 14 Prozent), Azul, Twitter, Tomitribe (neu im EC mit 10 Prozent) und Hazelcast (9 Prozent). Der iJUG wird seine zentralen Forderungen auch außerhalb des Executive Commitee verfolgen beziehungsweise in Zusammenarbeit mit den gewählten Organisationen.

https://jcp.org/aboutJava/communityprocess/elections/2016.html

28. November 2016

Auswertung der Java EE 8-Umfrage: erste Ergebnisse

Die Ergebnisse des „Java EE 8 Community Survey" lassen quälend lange auf sich warten. Schuld daran ist wohl nicht nur, dass es Freitext-Felder gab und die Teilnehmer ordentlich Gebrauch davon gemacht haben (zumindest nach allem, was ich von iJUG-Mitstreitern weiß), sondern auch, dass mit David Delabassee anscheinend nur ein einziger Oracle-Mitarbeiter an der Auswertung sitzt. Jetzt gibt es zumindest mal ein paar Informationshäppchen in einem Blog-Eintrag von David, der sich auf die Umfrage bezieht: Die JSRs zu Management API 2.0 und JMS 2.1 sollen zurückgezogen, also eingestellt werden. Soweit nichts Neues zum Stand der JavaOne-Keynote. Für MVC sucht Oracle nun nach einem anderen Community-Mitglied, das diesen außerhalb von Java EE 8 vorantreiben könnte (im Original: „as a stand-alone component"). Das wäre tatsächlich ein guter Schachzug, nicht nur für Oracle. Wobei ich mich in meiner Naivität frage, warum die Spezifikation dann nicht doch in Java EE 8 aufgenommen werden könnte – wenn der Zeitplan funktioniert. Aber vielleicht gibt es dazu ja auch demnächst mehr Details.

https://blogs.oracle.com/theaquariumentry/a_quick_update_on_java

5. Dezember 2016

„Inkubator"-Packages für das JDK

Um die Aufnahme neuer Features zu erleichtern, schlagen OpenJDK-Entwickler mit dem neuen JEP 11 vor, einen „Inkubator"-Prozess zu nutzen. Dieser soll es ermöglichen, in einem gewissen Maß auch unfertige Features aufzunehmen, um zunächst Erfahrungen zu sammeln, und ihre Standardisierung oder Finalisierung auf ein Folge-Release zu verschieben. Diese Features sollen im speziellen Package „jdk.incubator" liegen, bis sie den Inkubator-Status verlassen und ein eigenes Package bekommen oder wieder aus dem OpenJDK entfernt werden.

http://openjdk.java.net/jeps/11

9. Dezember 2016

OpenJDK: Vorschläge für Generics, Lambdas und Enumerations

Noch ein paar weitere JEPs: Die Verbesserungsvorschläge mit den Nummern 300, 301 und 302 sollen für besser lesbaren und klareren Code in Bezug auf Generics und Lambdas sorgen. Sie stehen aber bislang nur zur Diskussion und sind entsprechend auch nicht auf ein Release festgelegt. JEP 300 soll mit der expliziten Deklaration von kovarianten beziehungsweise kontravarianten Parameter-Typen dem Compiler ermöglichen, bei der Nutzung automatisch Wildcards („? super X" oder „? extends X") einzusetzen. JEP 301 will Konstanten innerhalb von Enumerations einen (jeweils) eigenen Typ ermöglichen. JEP 302 soll Mehrdeutigkeiten bei Lambda-Aufrufen verhindern, indem nicht genutzte Parameter durch Unterstriche ersetzt werden. Sie im Detail zu erläutern, würde den Rahmen sprengen, aber es gibt eine gute Zusammenfassung auf heise.de und die Details sind natürlich auf der openjdk-Seite verfügbar.

http://openjdk.java.net/jeps/

19. Dezember 2016

MicroProfile-Projekt bei Eclipse angenommen

Die Eclipse Foundation hat bekannt gegeben, dass der Vorstand das Projekt „MicroProfile" einstimmig angenommen hat. Diskussionen gibt es wohl noch bezüglich der Lizenzen: Momentan läuft MicroProfile unter der Apache License 2.0, was für die Eclipse Foundation sehr ungewöhnlich wäre. Mike Milinkovich, Chef der Eclipse Foundation, hat daher eine duale Lizenz mit der Eclipse Public License vorgeschlagen; dies trifft jedoch auch nicht auf uneingeschränkte Zustimmung. Trotzdem ist das Thema laut Milinkovich kein „Showstopper", weil die Apache License bereits akzeptiert wurde – notfalls auch allein.

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

21. Dezember 2016

Java EE 8: Die Umfrage-Ergebnisse

Endlich: Oracle hat die Auswertung der Community-Umfrage zu EE 8 veröffentlicht. Passend zur Oracle-Strategie belegen die JSRs Management API, JMS und MVC bei der Frage nach der Wichtigkeit von Features drei der vier letzten von 21 Plätzen – direkt hinter „Multi-Tenancy". „Configuration" und „Service Health" finden sich in der vorderen Hälfte. Die grafische Darstellung ist allerdings etwas suggestiv, wie sofort im iJUG bemängelt wird. Das Feature „MVC" ist, verglichen mit den ersten 16 Plätzen (von „REST Services" ganz

vorne bis „State Management"), den Befragten nicht absolut unwichtig, sondern einfach nur etwas weniger wichtig. Oracle sieht die Umfrage aber als Bestätigung der bereits eingeleiteten Roadmap-Änderung.
https://blogs.oracle.com/theaquarium/entry/java_ee_8_community_survey2

21. Dezember 2016

Aufregung um Java-Lizenzen

In der Java Community herrscht wieder Aufregung – diesmal wegen eines Beitrags im Online-Magazin „The Register" vor ein paar Tagen. Beim Titel „Oracle zielt auf Java Nicht-Zahler" und dem entsprechenden Bild, bei dem man in eine Gewehrmündung schaut, kann einem schon etwas mulmig werden, insbesondere wenn ein Lizenz-Audit bevorsteht. Inhaltlich geht es in dem Beitrag darum, dass Oracle weltweit zwanzig Neueinstellungen vorgenommen habe, die sich ausschließlich um Java-Lizenzverstöße kümmern sollen. Einige Kunden sollen bereits fünfstellige Beträge gezahlt haben. Weiter heißt es: „Experts are now advising extreme caution in downloading Java SE". Nun klingt ja die Zahl von zwanzig Auditoren erst einmal nicht furchterregend, wenn man mal ganz forsch annimmt, dass Java doch bei einigen Firmen weltweit im Einsatz ist. Diejenigen, die es trifft, dürfte das sicher nicht trösten. Nur: Der Text liefert zwar im Prinzip die entscheidenden Informationen, schürt aber beim unbedarften Leser doch mehr Angst als nötig. Was vielleicht damit zusammenhängt, dass der im Text zitierte Experte Gründer einer Firma ist, die mit Services rund um das Oracle-Lizenzmanagement Geld verdient … Jedenfalls wird das Thema im iJUG ausgiebig diskutiert. Es herrscht allerdings Einigkeit darüber, dass es abgesehen von lizenzpflichtigen Java-Embedded-Varianten erst mit dem Einsatz von „-XX:+UnlockCommercialFeatures" beim Java-Aufruf gefährlich wird. Dieser Schalter sollte auch dem leichtsinnigsten Nutzer zumindest eine Ahnung seiner möglichen Konsequenzen vermitteln.
https://www.theregister.co.uk/2016/12/16/oracle_targets_java_users_non_compliance/

23. Dezember 2016

Java-Champions reagieren auf Lizenz-Diskussion

Die Java-Champions, eine von Oracle unterstützte, aber grundsätzlich unabhängige Gruppe von Java-Experten, haben zur laufenden Lizenz-Diskussion ihre Sicht in einem Google-Dokument zusammengefasst. Es enthält im Wesentlichen die Dinge, die auch im iJUG schon diskutiert wurden. Der iJUG hat das Dokument ins Deutsche übersetzt.
http://bit.ly/2jpXiGO

18. Januar 2017

JSON-P JSR erreicht „Public Review"-Meilenstein

JSON-P 1.1 (JSR 374) ist jetzt im „Public Review" und wartet auf Feedback aus der Community.
https://blogs.oracle.com/theaquarium/entry/json_p_1_1_jsr2

19. Januar 2017

OpenJDK 9 ist „feature complete"

Die Entwicklung des JDK 9 geht im Rahmen des im Oktober 2016 aktualisierten Zeitplans voran. Mark Reinhold hat soeben das Erreichen des wichtigen Meilensteins „Feature Extension Complete" verkündet – alle JEPs und kleineren Verbesserungen sind also integriert. Weitere Änderungen, abgesehen von Bugfixes, kommen nun nur noch aus besonders wichtigen Gründen in das Release. Das angekündigte Release-Datum im Juli erscheint damit realistisch.
http://mail.openjdk.java.net/pipermail/jdk9-dev/2017-January/005505.html

23. Januar 2017

Reza Rahman zu Java-EE-Umfragen

Reza Rahman, Ex-Oracle-Mitarbeiter und Vorkämpfer der Java EE Guardians, hat in seinem Blog auf die Ergebnisse der Java-EE-8-Umfrage von Oracle reagiert und sie mit der Umfrage verglichen, die die Guardians gemeinsam mit DZone im Herbst 2016 durchgeführt hatten (die Ergebnisse der Guardians-Umfrage waren vor der JavaOne bereits mit Oracle geteilt worden). Kernpunkt seines Blog-Eintrags ist, dass sie relativ nah beieinander liegen – eine grundsätzliche Bestätigung des Oracle-Kurses also. Die Aufregung um Java EE könnte sich also langsam wieder legen, wenn alle Java EE JSRs endlich die nötige Fahrt aufnehmen. Am Ende seines Blog-Eintrags verweist Reza noch auf das MicroProfile, zu dem Oracle sich auch noch klar positionieren muss: „The idea is to make collaboration-based micro-services centric products from Java EE vendors available essentially before Java EE 8 is released. We can hope that the MicroProfile efforts will converge with Java EE 9 sooner rather than later." Das wäre schön.
http://blog.rahmannet.net/

27. Januar 2017

Stirb langsam: MD5

Der nächste Schritt, den seit Langem als unsicher eingestuften MD5-Algorithmus einzumotten, ist noch einmal um ein Quartal verschoben worden: Mit dem „Critical Patch Upgrade" im April soll das JRE aber endgültig mit MD5 signierte Jars als unsigniert ansehen. Eigentlich sollte das schon in diesem Monat der Fall sein, Oracle ist aber nach eigenen Angaben von einer Reihe von Kunden gebeten worden, ihnen noch etwas Zeit zu lassen. Auch SHA-1 soll dann nicht mehr in Zertifikatsketten akzeptiert werden, die an einem Default-Root-Zertifikat aus dem JDK hängen.
www.java.com/en/jre-jdk- cryptoroadmap.html

Andreas Badelt
Leiter der DOAG SIG Java

Er organisierte von 2001 bis 2015 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V. und war in dieser Zeit ehrenamtlich in der Development Community aktiv. Seit 2015 ist Andreas Badelt Mitglied in der neugegründeten Java Community der DOAG Deutsche ORACLE-Anwendergruppe e.V.

Achtung, Audit!

Markus Karg, JUG Goldstadt

Java ist Open Source, dann darf ich es doch wohl auch frei benutzen und verteilen? Weit gefehlt: Wenn ein Oracle-Auditor ins Haus kommt, sollte man Bescheid wissen! Die Fallstricke bei der Nutzung und Verteilung von Java

Varianten von Java SE

Oracle verpackt das eigentlich freie und kostenlose OpenJDK in verschiedene Pakete, die in überwiegender Mehrzahl kostenpflichtig sind. Alle Varianten enthalten ein JRE und ein JDK sowie eine Reihe zusätzlicher Beigaben:

- Das Projekt OpenJDK bietet JRE und JDK selbst nicht als kompilierten Download an, sondern verweist auf die Produkt-Palette von Oracle. Es ist jedoch in Debian und anderen Distributionen enthalten. Die JVM ist weniger leistungsstark als jene, die im Oracle-Java-SE-Download enthalten ist. Daher enthält beispielsweise Raspbian (ein speziell auf den Raspberry Pi zugeschnittenes Debian-Derivat) zusätzlich das proprietäre Oracle Java SE inklusive dessen leistungsfähiger Hot Spot Virtual VM (siehe unten).
- Oracle Java Platform Standard Edition (Java SE) ist das, was allgemein als „Java Download" bekannt ist und vom Normal-Anwender heruntergeladen wird. Es ist kostenlos (aber nicht frei!) für „general purpose computing" unter der Oracle Binary Code License (BCL). Es darf – unter Beachtung der in der BCL genannten Regeln– auch weitergegeben werden. Sehr wahrscheinlich ist davon auszugehen, dass Oracle mit „general purpose computing" nur PCs und Server meint. Wie beispielsweise der Rechtsstreit mit Google über Java auf Android zeigt, sind jedoch bereits die Nutzung auf Mobiltelefonen sehr vermutlich und jede Art von Embedded-Anwendung in Gerätschaften jeglicher Couleur – Stichwort „IoT" – sogar ganz explizit von der kostenfreien Nutzung ausgeschlossen. Für Mobiltelefone wirbt die Firma Gluon mit Oracles Segen mit der Gluon VM, einem OpenJDK-Port auf Android und iOS.
- Oracle Java SE Embedded enthält eine spezielle, für die Nutzung in SBCs und Mikro-Controllern optimierte JVM sowie Zusatzbibliotheken. Das Produkt steht unter OTNLA. Die Entwicklung damit ist kostenlos, die Weitergabe an den Endkunden ist allerdings zu bezahlen, und zwar pro Gerät, auf dem sich später die Embedded Runtime befindet.
- Oracle Java SE Advanced Desktop (optional for ISVs) sowie Oracle Java SE Advanced (optional for ISVs) enthalten die im Kasten „Kostenpflichtige Beigaben" genannten Entwicklungs- und Administrations-Werkzeuge und sind zunächst pro Entwickler zu bezahlen. Die Variante „Desktop" beschneidet die Client/Server-Fähigkeit der enthaltenen Monitoring-Werkzeuge. Der Zusatz „for ISVs" erlaubt dem Lizenznehmer darüber hinaus die Weitergabe von Java Flight Recorder und Java Mission Control sowie Advanced Management Console und Enterprise JRE Installer an Endkunden.
- Oracle Java SE Suite entspricht Oracle Java SE Advanced, enthält aber eine JVM mit Soft-Real-Time-Fähigkeit. Somit benötigt jeder Entwickler eine kostenpflichtige Lizenz.
- IBM und weitere Anbieter haben darüber hinaus eine Reihe kostenloser und kostenpflichtiger Produkte im Angebot, die wiederum proprietäre Erweiterungen gegenüber OpenJDK enthalten (etwa andere JVMs wie IBM J9 oder Unterstützung für spezielle Betriebssysteme wie z/OS).

Der geneigte Leser möge sich auf den entsprechenden Produktseiten selbst informieren, da dies sonst den Umfang dieses Artikels sprengen würde.

Kurz vor Weihnachten geriet die mediale Java-Welt in Aufruhr: Angeblich sei es dem Durchschnitts-Java-Programmierer unmöglich, sich ein JDK zu besorgen, ohne damit praktisch schon mit einem Fuß in der Oracle-Abzock-Falle zu stehen. Tatsächlich hat sich das Ganze schon bald als unbegründete Panikmache eines offenbar schlecht recherchierenden Online-Redakteurs herausgestellt. Trotzdem bleibt das flaue Gefühl, ob an der Sache nicht doch etwas dran ist. Tatsächlich: Wer nicht aufpasst, kann beim Besuch des Auditors in echte Argumentationsnot geraten. Dieser Artikel klärt auf und gibt Tipps, wie Lizenz-Risiken leicht zu vermeiden sind.

Was ist eigentlich ein Lizenz-Audit?

Viele Softwarehersteller behalten sich das Recht vor, beim Kunden zu überprüfen, ob nicht etwa mehr Anwender die Software verwenden, als Lizenzen eingekauft wurden. Hierzu ist es, neben Selbstauskünften und Fernzugriff, teilweise auch gängige Praxis, einen Auditor vor Ort zu senden, der mit entsprechenden Werkzeugen ausgestattet die tatsächliche Nutzung im Firmennetz zählt. Diese Prüfung muss nicht immer negativ sein: Teilweise stellt sich dabei auch heraus, dass bestimmte Produkte schon lange nicht mehr im Einsatz sind und man sich deren Aktualisierung und Support künftig sparen kann. Trotzdem sind Administratoren und Geschäftsleitung gut beraten, mit solchen Prüfungen zu rechnen und ein ordentliches Lizenz-Management einzuführen. Denn je nach Grad des Verschuldens können unter Umständen auch strafrechtliche Konsequenzen drohen.

Steht der Auditor erst einmal vor der Tür, ist es meist zu spät. Eine Nutzung nicht lizenzierter Software lässt sich schwer begründen und ist kein Kavaliersdelikt. Ehrlichkeit ist sicherlich die bessere Lösung: Wer Tools verwendet, die Geld kosten, sollte die typischerweise sowieso moderaten Kosten nicht scheuen, sondern nutzenorientiert denken und aktiv entsprechende Lizenzen erwerben.

Leider hat es Oracle einem bislang sehr leicht gemacht, sich nicht lizenzierte Software mit dem JDK- und JRE-Download auf den Rechner zu ziehen. Das Gute ist, dass das kein Problem darstellt – solange man diese Software nicht regelmäßig verwendet. Doch um welche Software geht es dabei überhaupt und wieso ist im JRE nicht-lizenzierte Software drin?

Die Sache ist eigentlich ganz einfach. Sun Microsystems hat das JRE (also die von Sun entwickelte HotSpot JVM sowie die Standard-Bibliotheken, die zu Java SE zählen) als Open Source freigegeben und, wie auch Oracle heute noch, auf jegliche Lizenzgebühren verzichtet. Jeder darf diese freien

Bestandteile im Rahmen der vorliegenden Lizenzbedingungen nutzen, ändern und weitergeben. Sie werden durch das Projekt „OpenJDK" verwaltet und weiterentwickelt und sind beispielsweise in Debian wie auch in vielen anderen Linux-Distributionen enthalten. Wer also nur das OpenJDK verwendet, ist immer auf der sicheren Seite.

Oracle verteilt jedoch kein reines OpenJDK. Um den Entwicklern, Endanwendern und Administratoren das Leben zu vereinfachen, hat Oracle weitere Werkzeuge entwickelt beziehungsweise eingekauft (unter anderem durch die Übernahme der JRockit-Produktfamilie). Einige dieser Features und Werkzeuge sind in den Open-Source-Zweig eingeflossen, andere sind weiterhin Closed-Source, werden jedoch kostenfrei verteilt, und wiederum andere sind und bleiben kostenpflichtige Zusatzprodukte. Es wäre nett gewesen, Oracle hätte diese in einen separaten Download verpackt. Leider ist dem nicht so.

Der Original-Download des JRE und des JDK enthält daher kostenpflichtige Zusatzprodukte, die der Auditor solange ignoriert, wie sie nicht nachweislich verwendet wurden. Startet man diese allerdings regelmäßig, wird der Kaufpreis fällig! Daher ist es wichtig zu wissen, welche Tools kostenpflichtige sind und welche nicht.

Kostenpflichtige Beigaben

Oracle verteilt unter dem Namen „Java SE" OpenJDK mit weiteren Beigaben. Diese sind kostenpflichtig und sollten nur dann regelmäßig genutzt werden, wenn entsprechende Lizenzen erworben wurden:

- Java Mission Control
- Java Flight Recorder
- Java Advanced Management Console
  Microsoft Windows Installer (MSI)
  Enterprise JRE Installer
- Java Usage Tracker

Nähere Informationen sind im Web zu finden, unter anderem in der Oracle-Produktübersicht *(siehe „https://www.oracle.com/de/products/index.html"* und in der Java SE FAQ *(siehe „http://www.oracle.com/technetwork/articles/javase/faqs-jsp-136696.html")*.

Es besteht grundsätzlich keine Gefahr, wenn die normalen Downloads für Java SE JRE beziehungsweise Java SE JDK genutzt werden und man von den kostenpflichtigen Beigaben einfach die Finger lässt. Vermutlich wird sich auch kein Auditor daran stören, dass man sich diese Tools, da sie ja schon einmal auf der Platte sind, einmal in Ruhe anschaut, um über deren Kauf zu entscheiden. Ein Recht darauf, dass die ersten paar Klicks kostenlos sind, hat man jedoch keineswegs. Wer also gar nicht wirklich am Kauf dieser Produkte interessiert ist, sollte sie auch wirklich nicht starten.

Wer hingegen interessiert ist, dem bietet Oracle mehrere Optionen an. Teilweise bedingen diese zusätzliche Downloads, teilweise handelt es sich lediglich um das Abschließen eines zusätzlichen Lizenzvertrags. Gerade in Letzterem liegt der Grund, weshalb die Downloads des „eigentlich freien" Java SE die „eigentlich kostenpflichtigen" Tools bereits enthalten. Der Kasten „Varianten von Java SE" erklärt, welche Optionen derzeit angeboten werden und warum es so viele gibt.

Fazit

Auch wenn die Panikmache in den Medien übertrieben war – es gilt: „Augen auf beim Java-Kauf!" Selbst wenn kein Auditor seinen Besuch angekündigt hat, sollte man sich überlegen, welche kostenlosen/kostenpflichtigen Werkzeuge wirklich benötigt werden, und dann entweder auf OpenJDK umsteigen, nur die kostenlosen Oracle-Bestandteile nutzen und weitergeben oder einen (kostenpflichtigen) Lizenzvertrag mit Oracle abschließen.

Markus Karg
markus@headcrashing.eu

Markus Karg fasziniert das Programmieren, seit er in grauer Vorzeit einen Sinclair ZX Spectrum in die Hand bekam. Heute verantwortet er die Entwicklung eines unabhängigen Software-Herstellers. JAX-RS begleitet der gebürtige Pforzheimer seit Version 0.8 durch Feature Proposals, Mitarbeit an der Referenz-Implementierung „Jersey" und zuletzt in der Expert Group JSR 370. Einige Features von JAX-RS, aber auch von anderen Java-APIs, stammen aus der Feder des JCP-Mitglieds. Seine Freizeit widmet er der Kunst seiner Frau sowie einer nachhaltigen Gesellschaft.

Swift – the next big thing?

Krystof Beuermann, LVM Versicherung

Im September 2016 kam Swift 3 heraus, eine moderne Open-Source-Sprache, mit der sich Apple auf neues Terrain wagt. Swift verspricht, die besten Features aus diversen Programmiersprachen zu vereinen und dabei schnell, modern und sicher zu sein. Selbst große Player wie IBM springen auf den Swift-Zug auf und integrieren die Sprache in Server- und Cloud-Lösungen. Hat Swift das Potenzial, das nächste große Ding zu werden?

Im Sommer 2010 arbeitet der junge Apple-Mitarbeiter Chris Lattner nächtelang an einem geheimen Projekt: eine neue, revolutionäre Programmiersprache mit dem Codenamen „Swift". Ende 2010 präsentiert er der Apple-Geschäftsleitung den ersten Entwurf. Die Chefs sind beeindruckt. Schnell entsteht aus dem Hobby-Projekt ein ernsthaftes Unterfangen. Weitere Entwickler werden Swift zugewiesen und arbeiten mit Hochdruck an der Fertigstellung der neuen Sprache.

Auf der Entwicklerkonferenz WWDC 2014 zündet Apple die Swift-Bombe. Die Veröffentlichung sorgt für Furore: Plötzlich steht macOS- und iOS-Entwicklern neben dem angestaubten Objective-C eine zweite Sprache zur Verfügung. Mit Swift tritt Apple das Versprechen an, viele der alten Mängel und Unzulänglichkeiten von Objective-C zu verbessern.

Für einen Paukenschlag sorgt Apple im Jahr 2015 durch die Ankündigung, die Sprache und die zugehörige Tool-Chain unter eine Open-Source-Lizenz zu stellen. Ein weiteres Novum im Apple-Universum: Neben macOS wird offiziell auch Linux unterstützt. Die Community ist begeistert. Anfang 2016 steigt ein weiteres Schwergewicht in den Ring. IBM kündigt an, Swift auf den Server und in die Cloud zu bringen. In den jährlichen Entwickler-Umfragen der beliebten IT-Seite „stackoverflow.com" rangiert Swift auf den vordersten Plätzen in der Kategorie „Most loved language". Grund genug, sich die junge Sprache näher anzusehen.

Auf dem Spielplatz

Java-Entwickler, die Swift-Code vorgesetzt bekommen, fühlen sich gleich zu Hause. Die Syntax ist leicht lesbar und nutzt Konzepte, an die man sich bei modernen Programmiersprachen gewöhnt hat.

Eine REPL-Umgebung, ähnlich der für Java 9 angekündigten JShell, wird auch für Swift angeboten. REPL steht für „Read Eval Print Loop" und ist eine interaktive Shell zur Evaluierung und Ausgabe von Ausdrücken. Sie erleichtert den Einstieg in die Sprache und eignet sich, um Code-Schnipsel ohne großen Aufwand zu testen. Auf Linux-Systemen ist die REPL-Umgebung nach der Installation der Tool-Chain über den Konsolenbefehl „swift" erreichbar *(siehe Abbildung 1)*.

```
bash-3.2$ swift
Welcome to Apple Swift version 3.0.1 (swiftlang-800.0.58.6 clang-800.0.42.1). Type :help for assistance.
  1> let äpfel = 3
  2. let birnen = 5
  3. let äpfelText = "Ich habe \(äpfel) Äpfel."
  4. let obstText = "Ich habe \(äpfel + birnen) Stücke Obst."
äpfel: Int = 3
birnen: Int = 5
äpfelText: String = "Ich habe 3 Äpfel."
obstText: String = "Ich habe 8 Stücke Obst."
  5>
```

Abbildung 1: Die REPL-Umgebung von Swift

Mac- und iPad-Besitzer bekommen es sogar noch bunter. Apple legt sich ins Zeug, die Sprache leicht erlernbar zu machen. So soll eine iPad-App namens „Swift Playgrounds" den Programmier-Nachwuchs spielerisch mit den Konzepten der Sprache vertraut machen. Als Nebeneffekt wird dadurch die junge Kundschaft an das eigene Ökosystem gebunden. Die macOS-Entwicklungsumgebung „Xcode" bietet ebenfalls sogenannte „Playgrounds" (Spielplätze) zum Ausprobieren von Code an. Im Gegensatz zur Terminal-Version der REPL-Umgebung bietet Xcode sogar eine grafische Auswertung an *(siehe Abbildung 2)*.

Man kann die Sprache auch online ausprobieren. IBMs Swift Sandbox *(siehe „https://swiftlang.ng.bluemix.net")* bietet eine Spielwiese für einfache Konsolen-Anwendungen *(siehe Abbildung 3)*.

Die verschiedenen Playgrounds sind eine gute Möglichkeit, um die Sprache kennenzulernen und damit zu experimentieren. Ähnlich wie Java setzt Swift auf starke Typisierung. Der Compiler unterstützt die Entwickler bei der Typensuche: Ist der Typ einer Variablen eindeutig erkennbar, muss er nicht zwingend angegeben werden *(siehe Listing 1)*. Anhand des Literals erkennt der Compiler den Typ „String" für die Variable „grußwort". Compilerbauer nennen dies „Typ-Inferenz".

Syntaktischer Zuckerguss

Beim Entwurf der Sprache wurden viele alte Zöpfe abgeschnitten. Das Semikolon hat als Zeilenbegrenzer ausgedient; wer es liebgewonnen hat, kann es trotzdem weiterverwenden. Zum direkten Ausführen von Code sind keine umhüllenden Klassen oder „main"-Methoden notwendig. Die eingängige Syntax ist nur einer der Vorzüge.

Unter der Haube verfolgt Swift einen Multiparadigmen-Ansatz. Konzepte der Objektorientierung sind angereichert mit funktionalen Elementen wie Closures, generischen Typen und Funktionen. Nützliche Konstrukte wie Tupel, die in Beziehung stehende Daten bündeln, sind von Haus aus dabei.

Ein besonderer Fokus liegt auf dem Thema „Sicherheit". Variablen müssen, wie bei Java, vor ihrer Benutzung stets initialisiert sein. Arrays und Integer genießen automatischen Schutz vor Überläufen. Bei der Deklaration von Variablen können sich Scala-Geübte entspannt zurücklehnen. Die Syntax ist fast identisch. Variablen werden über das Schlüsselwort „var" deklariert, Konstanten anstatt über „val" über das Wort „let" *(siehe Listing 2)*.

Im Gegensatz zu Java können Variablen standardmäßig nicht nil - das Äquivalent zu

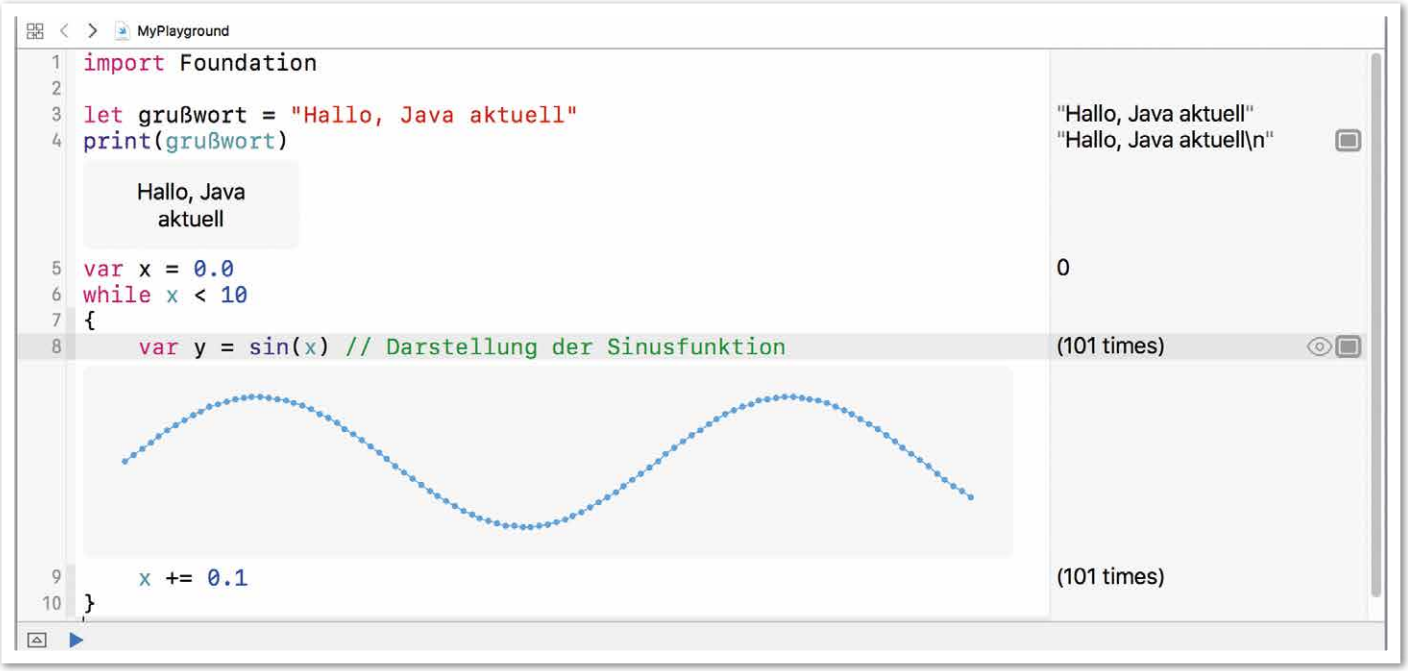


Abbildung 2: Playgrounds in der Entwicklungsumgebung Xcode

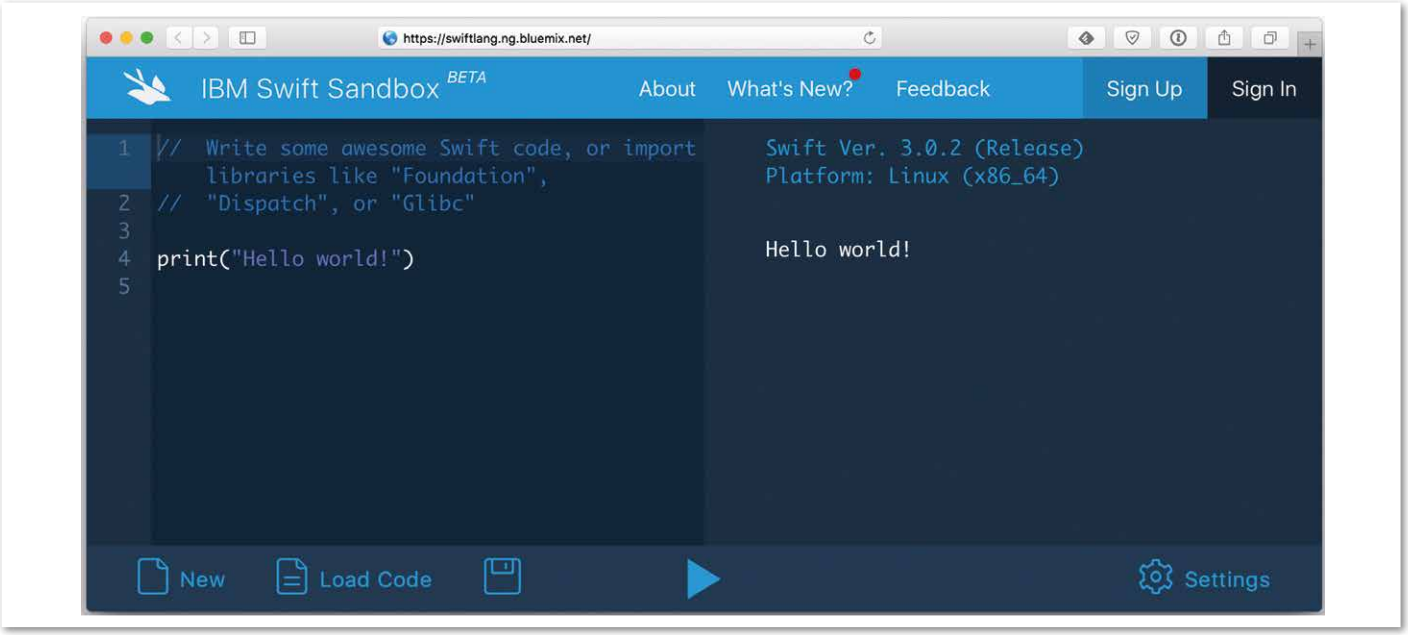


Abbildung 3: Die IBM Swift Sandbox zum einfachen Ausprobieren von Code-Schnipseln

null - sein. Erst die auch aus Java 8 bekannten Optionals bieten dafür eine Möglichkeit: Sie verpacken die eigentlichen Variablen in einer zusätzlichen Hülle. Optionals kennzeichnen sich durch ein Fragezeichen hinter der Typendefinition *(siehe Listing 3)*.

Um an den Inhalt eines Optional zu gelangen, muss man es auspacken. Dafür fungiert das Ausrufezeichen *(siehe Listing 4)*. Beim expliziten Auspacken lauert Gefahr: Versucht man, ein Optional auszupacken, das „nil" ist, wird dies mit einem Laufzeitfehler quittiert.

Funktional verschachtelt

Neben Variablen können auch Funktionen optional sein. Eine Funktion ist über das Stichwort „func" definiert. Funktionen können dabei global, lokal und sogar ineinander verschachtelt sein. Die Syntax ähnelt der Methoden-Definition von Java. Schön anzusehen sind die bei Java vermissten benannten Parameter, die auch der Aufrufer benutzt. Mathematisch Versierte freuen sich über den Pfeil als Kennzeichner für den Rückgabe-Typ *(siehe Listing 5)*.

Optionale Funktionen definieren sich mit einem Fragezeichen hinter dem Rückgabe-Typ. Nur diesen Funktionen ist es erlaubt, nil als Rückgabewert zu verwenden *(siehe Listing 6)*.

Java-Geübte heben bereits den Zeigefinger, wenn sie den „=="-Operator für String-Vergleiche sehen. In Swift ist dies die übliche Praxis, da dieser für viele Typen vordefiniert ist, so auch für Strings. Im Gegensatz zu Java ist sogar das Überladen von Operatoren mög-

```
var grußwort = "Hallo, Java aktuell"
```

Listing 1

```
let konstanterString = "Foo"
var variablerInteger :Int = 42
```

Listing 2

```
var optionalerString: String? =  nil
optionalerString = "Foo"
```

Listing 3

```
print(optionalerString) // Gibt Optional("Foo") aus
print(optionalerString!) // Gibt Foo aus
```

Listing 4

lich, sodass komplexe Klassen eigene Vergleichsoperatoren implementieren können.

Doch Funktionen können noch mehr. Swift beherrscht Closures, also anonyme Funktionen, die sich nahtlos in das Typ-System einreihen. Somit kann in einer Variablen eine Referenz auf eine Funktion abgelegt sein. Im Beispiel in *Listing 7* ist eine anonyme Funktion der Variablen „addition" zugewiesen. Sie bekommt zwei „Int"-Werte als Eingabe und gibt die Summe der Werte zurück.

Closures können auch Parameter in Funktionen sein. Die Funktion „ausführenUndAusgeben" in *Listing 8* erwartet drei Parameter als Eingabe: zwei Ganzzahlen und eine Funktion, die ebenfalls zwei Zahlen als Parameter erwartet. „ausführenUndAusgeben" ruft die übergebene Funktion auf und zeigt die Rückgabe auf der Konsole an.

Strukturen: Eine Klasse für sich

Neben den altbekannten Klassen gibt es in Swift ein weiteres Konstrukt, das man als Bauplan für Objekte verwenden kann: die aus C bekannten Strukturen. Klassen und Strukturen unterscheiden sich fundamental voneinander. Klassen sind sogenannte „Reference-Types", bei denen nur die Referenz verwaltet und übergeben wird. Sie sind wie bei Java mit dem Schlüsselwort „class"

```
func grüßen(person: String) -> String {
    let gruß = "Hallo " + person + "!"
    return gruß
}
grüßen(person: "Heinz") // Gibt Hallo Heinz! aus
```

Listing 5

```
func peterGrüßen(person: String) -> String? {
    if (person == "Peter")
    {
        return "Hallo Peter!"
    }
    else {
        return nil
    }
}
```

Listing 6

```
let addition = { (summand1: Int, summand2: Int) -> (Int) in
    return summand1+summand2
}
print(addition(2,2)) // Gibt 4 aus
```

Listing 7

```
func ausführenUndAusgeben(a: Int, b: Int, closure: (Int,Int) -> Int) ->
Void
{
    print(closure(a,b))
}
ausführenUndAusgeben(a: 4,b: 4,closure: addition) // Gibt 8 aus
```

Listing 8

```
class ApplicationWindow {
    var title = "Hauptfenster"
}
```

Listing 9

```
struct Ostfriese {
    var vorname = "Hein"
}
```

Listing 10

```
protocol Mensch  {
    var vorname: String {get set}
    func grüßen()
}
```

Listing 11

```
struct Bayer: Mensch
{
    var vorname: String
    func grüßen() {
        print("Grüß Gott. Ich bin \(vorname)")
    }
}
Bayer(vorname: "Franz").grüßen() // Gibt Grüß Gott. Ich bin Franz aus
```

Listing 12

definiert *(siehe Listing 9)*. Strukturen hingegen sind als Value-Types bezeichnet und mit dem Wort „struct" angelegt. Bei ihnen wird stets der Inhalt kopiert und weitergegeben *(siehe Listing 10)*.

Man kann sich nun fragen, wann man Klassen und wann Strukturen einsetzt. Die Antwort lautet: „Es kommt darauf an". Strukturen sollten dann eingesetzt werden, wenn sie simple, kopierbare Daten enthalten. Die in *Listing 10* angeführte Struktur „Ostfriese" ist ein Beispiel dafür. Strukturen verringern zudem das Problem von Speicherlecks in Multithread-Anwendungen.

Klassen sollten den Vorzug bekommen, wenn die Referenz auf sie eine besondere Rolle spielt. Ein Anwendungsfenster oder eine Netzwerk-Verbindung würde man eher als Klasse implementieren, da hier die Referenz auf die konkrete Ressource wichtig ist. Klassen sind zudem vererbbar, Strukturen nicht.

Etikette und Protokoll

Eine Sache haben Klassen und Strukturen gemeinsam: Sie können Protokollen entsprechen. Protokolle stellen Baupläne dar und ähneln den Interfaces aus Java. Unserem Ostfriesen-Beispiel folgend, wäre ein Protokoll „Mensch" denkbar *(siehe Listing 11)*.

Offensichtlich lassen sich nicht nur Funktionen über Protokolle erfordern, sondern auch Variablen. Über den Doppelpunkt können Klassen und Strukturen einem Protokoll entsprechen *(siehe Listing 12)*.

Alles ist erweiterbar

Swift kennt ein für Java-Fans völlig neues Konzept: Extensions. Durch sie sind Klassen und Strukturen um zusätzliche Funktionalität erweiterbar. Ähnlich zur Vererbung lassen sich neue Variablen und Methoden hinzufügen. Ruby-Geübten mag es nun kalt den Rücken hinunterlaufen. Doch keine Sorge: Das Überschreiben bestehender Funktionalität ist bei Swift nicht vorgesehen. Extensions sind so mächtig, dass sie Typen erweitern können, für die man gar keinen Zugriff auf den Quelltext hat. So kann man sich leicht Hilfsfunktionen programmieren, beispielsweise eine Funktion verdoppeln() für den Typ „Int" *(siehe Listing 13)*. Die Funktion lässt sich jetzt für jede Ganzzahl aufrufen *(siehe Listing 14)*. Selbst die eben erwähnten Protokolle können durch Extensions um „default"-Implementierungen erweitert werden *(siehe Listing 15)*.

In der Struktur „Person" ist die Implementierung der Methode nicht mehr zwingend notwendig und der „grüßen()"-Aufruf

funktioniert trotzdem *(siehe Listing 16)*. Extensions sind somit ein handliches Werkzeug, um Code einfach wartbar zu machen.

```
extension Int{
    func verdoppeln () -> Int {return self+self}
}
```

Listing 13

```
print(5.verdoppeln()) // Gibt 10 aus
```

Listing 14

```
extension Mensch
{
    func grüßen() {
        print("Guten Tag. Ich bin \(vorname)")
    }
}
```

Listing 15

```
struct Westfale: Mensch
{
    var vorname: String
}
Westfale(vorname: "Klaus").grüßen() // Gibt Guten Tag. Ich bin Klaus aus
```

Listing 16

Swift auf dem Server

Ursprünglich war Swift für die Entwicklung von macOS- und iOS-Apps auf Basis der von Apple bereitgestellten Frameworks vorgesehen. Mit der Gründung einer Server-API-Arbeitsgruppe im Herbst 2016 lenken die Schöpfer die Sprache in eine weitere Richtung mit dem Ziel, die Sprache für Server und Cloud fit zu machen. Die Vision ist, Backend-Frameworks komplett in Swift zu schreiben, ohne auf C-Bibliotheken zurückzugreifen.

Die Community hat dafür einiges an Vorarbeit geleistet. Derzeit ringen mehrere Kandidaten um die Vorherrschaft als führendes Swift-Backend-Framework. Dass es um viel Geld geht, wird deutlich, wenn man sich die Konkurrenten näher ansieht. Den ersten Wurf macht das junge kanadische Startup-Unternehmen PerfectlySoft Inc. im Herbst 2015 mit der Kombination namens „Perfect"aus Bibliothek und HTTP-Server. Gesponsert mit mehr als drei Millionen US-Dollar Risikokapital entwickelt sich das Gespann schnell zur Nummer eins unter den Server-Frameworks.

Auch IBM sieht viel Potenzial in der Sprache Swift und kündigt Anfang 2016 eine Swift-Laufzeitumgebung für den hauseigenen Cloud-Dienst Bluemix an. Dazu veröffentlicht IBM die erste Version des in Swift geschriebenen, quelloffenen Web-Frameworks Kitura. Ähnlich wie Perfect ist Kitura eine Kombination aus HTTP-Server und Bibliothek, angereichert mit den entsprechenden Entwicklungstools.

Das dritte Framework im Bunde nennt sich Vapor und verspricht nichts weniger, als die Zukunft der Web-Entwicklung zu sein. Mit einer ausdrucksstarken Sprach-Syntax und der großen Community möchte man besonders Swift-Anfänger und -Umsteiger anlocken.

Die großspurigen Versprechen täuschen darüber hinweg, dass alle genannten Frameworks noch in den Kinderschuhen stecken. Dies liegt nicht zuletzt daran, dass auch an der Sprache noch stark geschraubt wird. Es mangelt unter Linux an produktionsreifen Frameworks zur Verarbeitung von Nebenläufigkeit und Parallelität. Die Community versucht, dies durch Eigenentwicklungen zu kompensieren. Die Erwartungshaltung ist jedoch, dass die Sprache selbst Implementierungen für Netzwerk, Security und Nebenläufigkeit mitbringt. Sollte man deswegen noch einen Bogen um Swift im Backend machen? Keinesfalls – für kleine agile Web-Projekte eignet sich Swift heute schon, insbesondere wenn der Client ein iOS- oder macOS-Gerät ist und man Logik „1:1" wiederverwerten kann.

Fazit

Die Sprache Swift hat viele Gemeinsamkeiten mit anderen Sprachen. Für Java-Entwickler fällt der Einstieg nicht zuletzt aufgrund der ähnlichen Syntax leicht. Dennoch verläuft die Lernkurve nicht komplett flach. Swift bringt frische Konzepte wie Extensions und Protokolle mit, die bekannte Pfade der Objektorientierung verlassen.

Eine schöne Programmiersprache allein reicht heutzutage nicht mehr aus, um sich auf dem Markt erfolgreich zu behaupten. Entwickler erwarten Frameworks, die sie bei der Arbeit unterstützen und ihnen tägliche Aufgaben erleichtern. Die Schritte in Richtung Open Source und Linux geben Swift Aufschwung. Beinahe täglich entstehen auf GitHub neue Frameworks und Module. Zudem steht mit Apple hinter Swift ein Unternehmen mit enormer Marktmacht und dem unbändigen Willen, die Sprache weiter zu pushen. Auch Firmen wie IBM sehen großes Potenzial in der Sprache und richten ihre Cloud-Plattformen auf Swift aus.

Trotz dieser Entwicklungen steht Swift noch am Anfang. Der Pioniergeist der Swift-Erfinder erinnert an die Anfänge von Java in den Neunzigern. Damals wurden die Grundsteine für eine Sprache gelegt, die heute auf Milliarden von Geräten läuft. Die Swift-Macher haben die Vision, diese Erfolgsgeschichte zu wiederholen. Das könnte ihnen durchaus gelingen, wenn die klugen Köpfe so viele Nachtschichten wie Chris Lattner einlegen.

Krystof Beuermann
krystof.beuermann@gmx.de

Krystof Beuermann ist IT-Architekt und Entwickler bei der LVM Versicherung in Münster. Er arbeitet dort im Spannungsfeld zwischen monolithischen und leichtgewichtigen Enterprise-Anwendungen. In der übrigen Zeit beschäftigt er sich mit Swift und iOS-Entwicklung.

Herausforderung „Multi Channel Architecture“

Lars Röwekamp, open knowledge GmbH

Die letzten Jahre standen ganz im Zeichen der Digitalisierung. Damit einhergehend findet ein „Mindshift“ im Denken und Handeln der Konsumenten statt. Zu jedem Zeitpunkt sollen deren digitale Bedürfnisse automatisch erkannt und via Web oder Mobile App befriedigt werden – wenn möglich, individuell auf den eigenen Kontext zugeschnitten. Multi-Channel ist das Schlagwort der Stunde. Aber wie sieht die dazu passende Architektur aus?

Mobile Devices wie Smartphones und Tablets gewinnen in den letzten Jahren mehr und mehr an Bedeutung. Kein Wunder also, dass auch immer mehr Unternehmen den mobilen Kanal als zusätzliche Chance zur Umsatzsteigerung und Kundenbindung für sich entdecken. Natürlich möchte man nicht für jeden Kanal das Rad, also Front- und Backend, neu erfinden müssen. Stattdessen wird versucht, eine Rundum-sorglos-Lösung für alle Kanäle zu schaffen: die Multi-Channel-Architektur.

Multi-Channel

Egal, ob „Graceful Degradation“, „Progressive Enhancement“, „Mobile First“ oder „Responsive Design“, das Prinzip ist immer gleich: Multi-Channel-Lösungen versuchen, Content beziehungsweise Funktionalität für die verschiedenen zu bedienenden Kanäle gemeinsam zu nutzen. Unterschiede gibt es lediglich in der visuellen Aufbereitung sowie dem Grad der zur Verfügung gestellten Funktionalität. Die Anwendungen werden dabei so konzipiert, dass das Backend spezifische Bildformate, UI-Elemente und Inhalte an den jeweiligen Client ausliefert beziehungsweise der Client den vom Backend erhaltenen Content entsprechend seinen funktionalen Möglichkeiten und UX-Spezifika adaptiert *(siehe Abbildung 1)*.

Architektonisch und technologisch stellen Multi-Channel-Lösungen keine große Herausforderung dar. Wir haben es nach wir vor mit dem klassischen Modell von Client und Server beziehungsweise Frontend und Backend zu tun. Lediglich die clientspezifi-

sche Anpassung des Contents ist neu. Sie kann, wie oben beschrieben, entweder auf dem Client oder auf dem Server erfolgen.

Multi-, Cross- oder gar Omni-Channel?

Auch wenn der eben beschriebene Lösungsansatz auf den ersten Blick recht reizvoll erscheint – „one content to rule them all" – und technologische Helferlein wie SaaS, LESS, OOCSS, Layout Breakpoints oder Grid-Systeme die entsprechenden Mittel zur Umsetzung an die Hand geben, hat das Ganze doch einen kleinen, aber entscheidenden Haken: Der Multi-Channel-Ansatz geht davon aus, dass es prinzipiell sinnvoll ist, auf allen Kanälen die gleichen fachlichen Use Cases anzubieten. Anders formuliert, die Stärken und Eigenheiten der unterschiedlichen Kanäle werden nicht wirklich genutzt und so wird durch einen zusätzlichen Kanal wie Mobile kein wirklicher fachlicher Mehrwert geschaffen.

Im günstigsten Fall führt eine Multi-Channel-Lösung lediglich dazu, dass bereits bestehende Nutzer mehrere der angebotenen Kanäle parallel nutzen oder aber von einem bereits bestehenden Kanal, etwa dem Web, auf einen neuen Kanal mit Mobile App wechseln. Neue Kunden werden so in der Regel nicht gewonnen, neue Geschäftsfelder nicht erschlossen. Wie auch? Schließlich ist die Funktionalität ja identisch zu der bereits existierenden.

Im ungünstigsten Fall wird genau das Gegenteil von dem erreicht, was sich der Anbieter ursprünglich erhofft hat. Bietet der neue Kanal, in unserem Beispiel die Mobile App, nicht die gewünschte Qualität, verliert der Nutzer das Vertrauen in den Anbieter und nutzt als Konsequenz auch die bis dato angebotenen Kanäle nicht mehr.

Was aber muss man tun, um das ursprüngliche Ziel – neue Kunden, neue Geschäftsfelder – zu erreichen, und wie wirkt sich dies auf die Anwendungs-Architektur aus? Wirklich sinnvolle Lösungen nutzen die Vorteile der verschiedenen Kanäle sowie den Kontext, in dem sie genutzt werden, konsequent aus und schaffen so echte Mehrwerte. Tablets werden zum Beispiel häufig zum Surfen auf dem heimischen Sofa verwendet. Smartphones dagegen auf dem Weg zur Arbeit, beim Anstehen in einer Schlange, im Café etc.

Eine Anwendung für die Kanäle Smartphone oder Mobile sollte also Mehrwerte schaffen, indem sie zu jedem Zeitpunkt weiß, wo der Nutzer sich gerade aufhält und

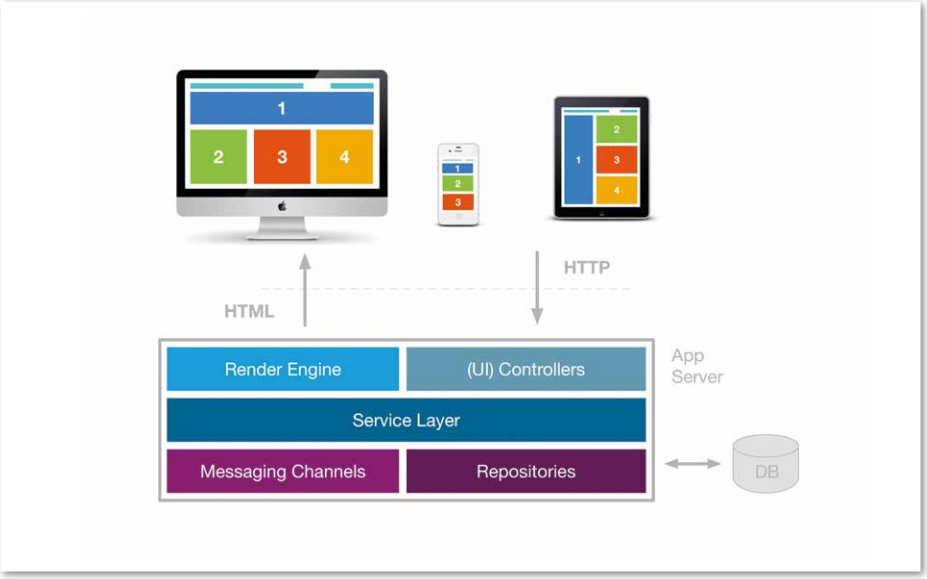


Abbildung 1: Responsive Design

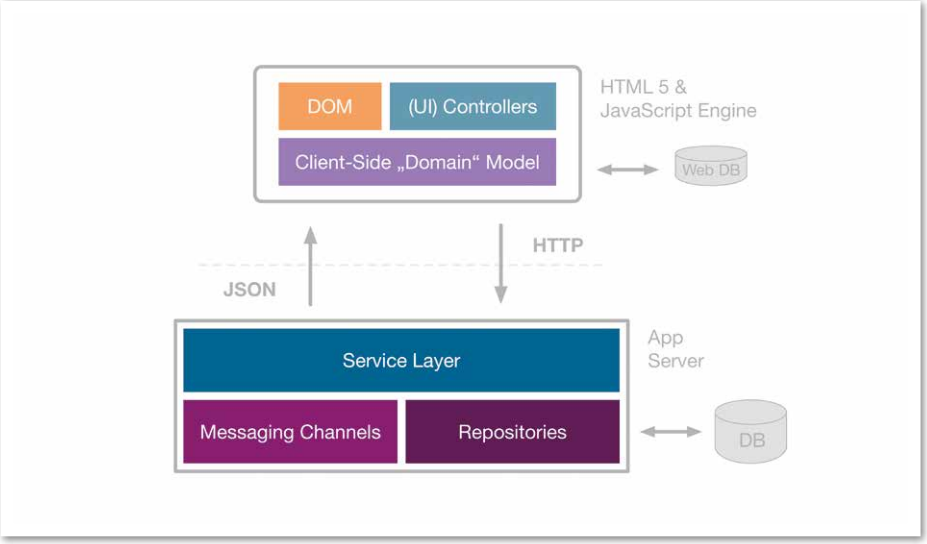


Abbildung 2: Ressourcen statt Views

wer er überhaupt ist, und aus diesen und einigen anderen Daten antizipieren, was der Nutzer gerade tun möchte. Genau diese – und keine andere Funktionalität – sollte die App dann prominent anbieten.

Dabei gilt es zusätzlich, das Device-spezifische Nutzerverhalten zu beachten. Beim Smartphone zum Beispiel holt der Nutzer das Gerät aus der (Hosen-)Tasche, führt eine Aktion aus beziehungsweise fragt eine Information ab und steckt es dann wieder weg. Auf einem Tablet dagegen surft der Nutzer mittels Gesten etwas länger durch die App. Im Web beziehungsweise bei Desktop-Anwendungen wiederum ist der Nutzer durchaus bereit, umfangreiche Formulare auszufüllen oder mehrseitige Prozesse in Form von Wizards zu bearbeiten.

Jeder Kanal sollte, abgesehen von den allgemeinen und auf allen Kanälen sinnvollen Funktionen, seine eigenen Use Cases abdecken und dabei den ihm bekannten Kontext bestmöglich berücksichtigen. So wird die Summe der Einzelteile – also der einzelnen Funktionalitäten der verschiedenen Kanäle – größer als das Ganze. Zusätzlich sollte ein fließender Wechsel von einem Kanal auf einen anderen, also zum Beispiel das Fortsetzen eines im Web begonnenen Kaufprozesses in einer Mobile App, möglich sein. Ist dies gegeben, sprechen wir nicht mehr von „Multi Channel", sondern von „Cross Channel".

Stehen am Ende nicht die einzelnen Kanäle und ihre jeweiligen Funktionalitäten, sondern der Nutzer selbst und seine Be-

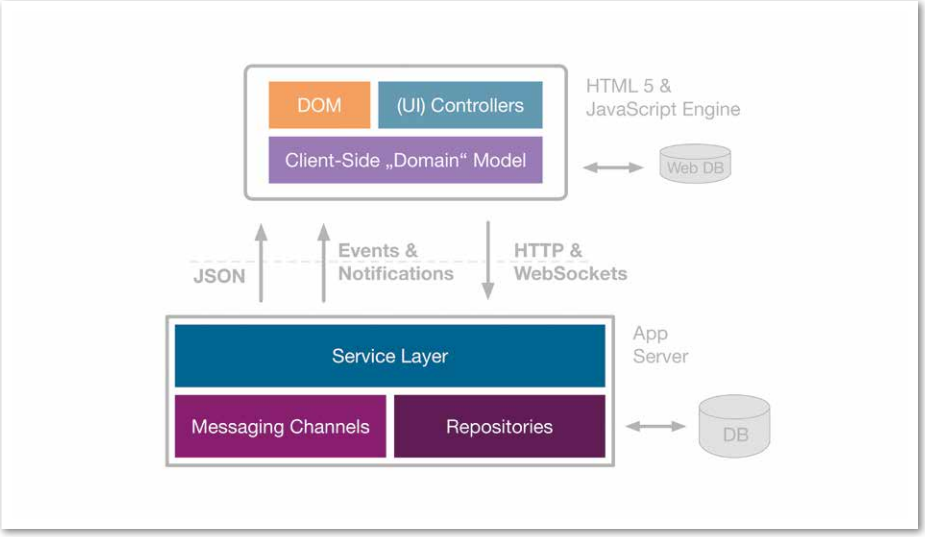


Abbildung 3: Agieren statt reagieren

dürfnisse im Fokus, haben wir den heiligen Gral – Omni-Channel – erreicht. Dort sind nicht selten auch SMS, Messanger oder Social-Media-Kanäle wie Facebook und Twitter eingebunden.

Während sich Multi-Channel-Lösungen noch relativ einfach durch leichte Modifikation bestehender Web-Anwendungen beziehungsweise deren Architekturen realisieren lassen, stellt sich dies bei Cross- beziehungsweise Omni-Channel schon etwas komplizierter dar. Klassische beziehungsweise monolithische sind häufig zu unflexibel, um auf die unterschiedlichen, sich permanent ändernden Anforderungen der verschiedenen Kanäle zeitnah reagieren zu können. Time-to-Market steht im direkten Widerspruch zur Architektur. Aber wo genau liegt das Problem und wie kann man dem entgegenwirken?

Schritt für Schritt zum Ziel

Ein wesentliches Problem klassischer Web-Architekturen ergibt sich durch das Ausliefern vorgerenderter HTML-Views. Dies gilt selbst dann, wenn die Views bereits optimal für die verschiedenen Kanäle wie Web und Mobile aufbereitet sind. Das beschriebene Vorgehen funktioniert nämlich nur dann, wenn das Backend sämtliche Use Cases aller Clients kennt. Neue Use Cases auf einem der Clients beziehungsweise einem der zu bedienenden Kanäle bedeuten automatisch auch immer eine Erweiterung des Backend. Dass dies dem Wunsch nach einer Optimierung von „Time-to-Market“ entgegensteht, ist leicht nachvollziehbar.

Besser wäre es, statt Views lediglich Daten (Ressourcen) zur Verfügung zu stellen und die visuelle Aufbereitung sowie die UI-Logik auf die Clients zu verlagern. So könnte eine unabhängige Evolution der verschiedenen Client-Typen – also Kanäle – erfolgen und somit auch eine bessere Skalierung in der Entwicklung erreicht werden *(siehe Abbildung 2)*. Als Austauschformat bietet sich JSON an, da dies die natürliche Objekt-Notation des Web ist, die gleichzeitig auch von allen anderen Clients wie nativem Android oder iOS-Apps verarbeitet werden kann.

Ein weiteres, häufig vernachlässigtes Problem ist das unterschiedliche Nutzungsverhalten auf den verschiedenen Kanälen. Während im Web-Umfeld längere Interaktionen mit einigen wenigen Request-Response-Paaren üblich sind, zeichnet sich das Anfrageverhalten auf mobilen Endgeräten durch sehr viele kleine, dafür aber dicht aufeinanderfolgende Requests aus, deren Responses häufig nur signalisieren, dass keine neuen Daten vorliegen. So klickt man sich zum Beispiel im Web ein bis zweimal täglich durch den Wizard eines Shops, während man parallel auf seinem Smartphone Hunderte von Anfragen an Social-Media-Dienste auslöst.

Diesem feingranularen, Pull-basierten Anfrageverhalten sind viele Web-Architekturen nicht wirklich gewachsen. Besser wäre es daher, bei Use Cases, die lediglich Änderungen an Timelines, Kontoständen, Börsendaten, Wetterdaten etc. darstellen sollen, von dem klassischen Pull-Verfahren auf ein Push-Verfahren zu wechseln, den Client also aktiv zu informieren, wenn neue Daten vorliegen – und nur dann. Technologien wie Server Side Push, WebSockets oder Mobile Push Notifications erlauben eine Realisierung des Verfahrens auf allen denkbaren Kanälen *(siehe Abbildung 3)*.

Die Zurverfügungstellung von Ressourcen statt Views sowie eine sinnvolle Kombination von Push und Pull sind wichtige Schritte auf dem Weg zu einer flexiblen und zukunftssicheren Cross- beziehungsweise Omni-Channel-Architektur, mit der schnell auf neue Anforderungen einzelner Clients beziehungsweise Kanäle reagiert werden kann.

Damit die Evolution der verschiedenen Clients und des Servers aber tatsächlich mehr oder minder autark funktionieren kann, sollte zusätzlich die Service-Schicht aufseiten des Backend entzerrt werden. Im ersten Schritt werden die Service-Endpoints in die Lage versetzt, die angefragten Ressourcen – via RESTful-Calls – zu liefern, anstatt wie bisher Anfragen à la Remote-Procedure-Call zu bearbeiten *(siehe Abbildung 4)*.

Im einem zweiten Schritt sollte man anschließend den gesamten Service-Layer modularisieren, um so nicht nur auf der Client-Seite, sondern auch aufseiten des Servers eine unabhängige Entwicklung und damit einhergehend ein getrenntes Deployment einzelner fachlicher Module zu ermöglichen. Als sinnvoller Schnitt für die einzelnen Module bieten sich die in Domain Driven Design als „Bounded Contexts“ bezeichneten, fachlich in sich geschlossenen Bereiche an *(siehe „http://domainlanguage.com/ddd“* und *Abbildung 5)*.

Dank der bisherigen Refactoring unserer althergebrachten Web-Architektur sind wir in der Lage, relativ schnell auf neue Anforderungen beliebiger Clients zu reagieren sowie die damit verbundenen notwendigen Änderungen an Client und Server getrennt zu entwickeln und einzurichten. Einziger Knackpunkt bleibt die Provisionierung. Änderungen an Datenbanken, neue App-Server-Instanzen, LDAP-Repositories oder andere benötigte Enterprise-Ressourcen stellen nach wie vor einen Flaschenhals dar, wenn es darum geht, in kürzester Zeit neue Features freizugeben.

Eine mögliche Lösung für dieses Problem wäre ein vollständiger oder zumindest teilweiser Wechsel der Infrastruktur in die Cloud sowie die Verwendung von Platform-as-a-Service-Diensten *(siehe Abbildung 6)*. Dieses Vorgehen hätte zusätzlich den Vorteil, dass Laufzeitkosten verursachungsgerecht anfallen und somit Kosten und Nutzen bei Last-Schwankungen oder starkem Wachstum in einer gesunden Relation stehen.

Alles wird gut, oder?

So weit, so gut. Aber ist es wirklich so einfach, seine bestehende, meist historisch gewachsene Web-Architektur in nur wenigen Schritten auf die Herausforderungen einer Cross- oder Omni-Channel-Architektur umzustellen? Zunächst einmal muss gesagt werden, dass es sich bei den oben aufgezeigten Schritten teilweise um recht umfangreiche Refactorings handelt. Je nach bestehender beziehungsweise nicht bestehender fachlicher Trennung und damit verbundener Modularisierung innerhalb der vorliegenden Architektur kann der oben aufgezeigte Weg Änderungen an nahezu dem gesamten Sourcecode nach sich ziehen. Darüber hinaus haben wir durch die fünf dargestellten Schritte ein verteiltes, lose gekoppeltes System geschaffen, mit all den Problemen, die derartige Systeme nun einmal seit Jahr und Tag mit sich bringen.

Bedingt durch die Tatsache, dass die verschiedenen Clients sich unabhängig von den anderen Clients entwickeln können und nur noch über REST-Endpoints mit dem Server kommunizieren, wird es über kurz oder lang zu dem typischen Problem der Schnittstellen-Versionierung kommen. Es gilt, die Schnittstellen beziehungsweise die Austauschformate möglichst flexibel zu gestalten, sodass Änderungen nicht grundsätzlich zu einer Inkompatibilität führen. Ein guter Ansatz ist hierbei Postels Gesetzt der Robustheit *(siehe „http://tools.ietf.org/html/rfc793“)*: „Be conservative in what you do, be liberal in what you accept from others.“

Änderungen an den Schnittstellen sollten also gut überlegt sein. Gleichzeitig sollte der Consumer einer Schnittstelle in der Lage sein, mit abwärtskompatiblen Änderungen klarzukommen. Zusätzliche Daten oder geänderte Strukturen zum Beispiel sollte er problemlos verarbeiten können. In welcher Art die Schnittstelle sich verändert hat, sollte der Server durch Semantic-Versioning 2.0 signalisieren *(siehe „http://semver.org“)*.

Eine weitere Herausforderung ergibt sich für den neuen Architektur-Ansatz im Umfeld der Fehlerbehandlung. Wurden bisher innerhalb des Monolithen (Java-)Exceptions geworfen, die wir fangen und bearbeiten konnten, haben wir es nun im Fehlerfalle mit HTTP-Responses zu tun, auf die wir gezielt reagieren müssen. Zunächst einmal gilt es dabei zu entscheiden, ob es sich wirklich um einen Fehler oder um ein Feature handelt. Die Verwendung von HTTP Status Codes *(siehe „https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html“* und *„https://httpstatuses.com“)* helfen uns dabei, genau diese Frage zu beantworten.

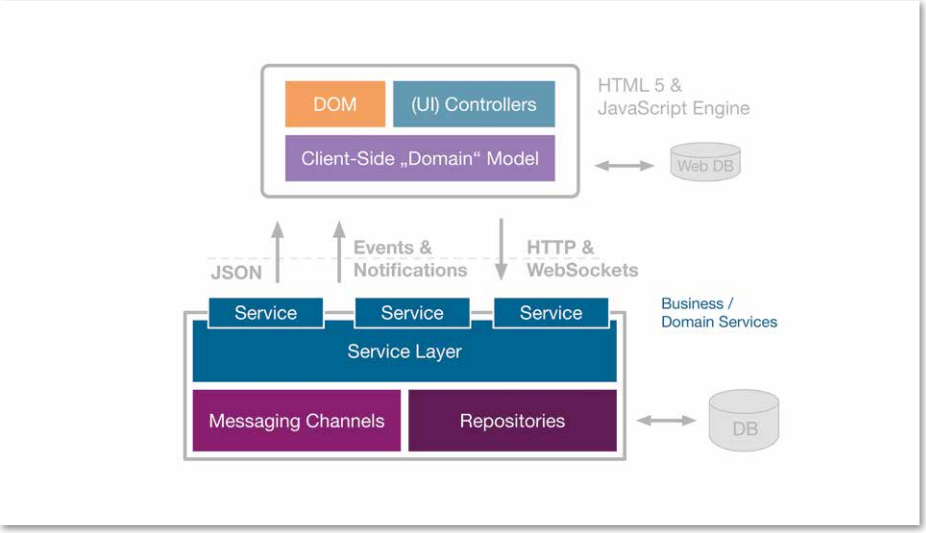


Abbildung 4: RESTfull-ServiceEndpoints

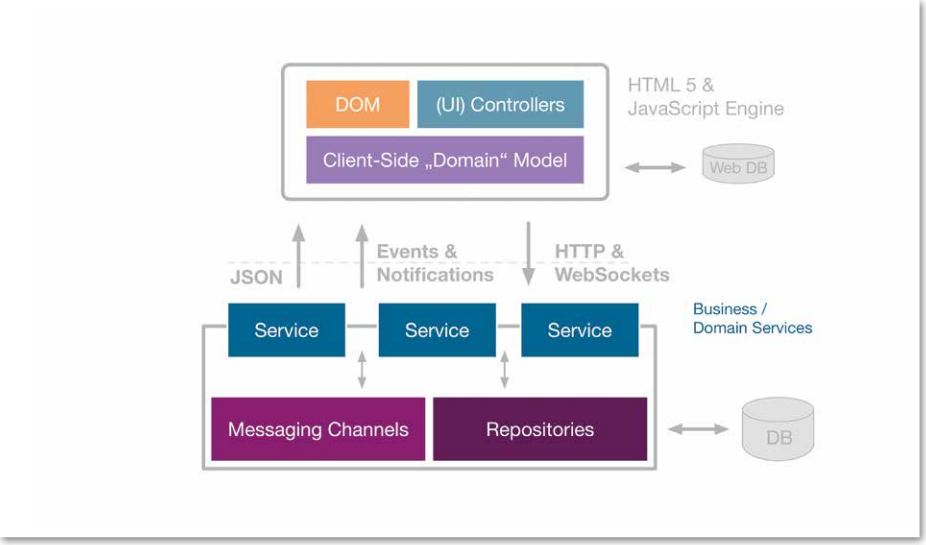


Abbildung 5: Domänen-Services

Liefert zum Beispiel eine Suchanfrage kein Ergebnis zurück, dann kann dies durchaus korrekt sein, da die angegebenen Suchkriterien zu keinem Treffer geführt haben. In diesem Fall würde der Server das korrekte Verhalten durch den Statuscode 204 („No Content“) signalisieren. Lag dagegen aufseiten des Servers ein Fehler vor, weil zum Beispiel die Datenbank nicht erreichbar oder die Suchanfrage syntaktisch nicht korrekt war, würde der Server dies durch einen entsprechenden Fehlercode aus dem Bereich der HTTP-Statuscodes 500 („Serverfehler“) oder 400 („fehlerhafte Anfrage“) beantworten. Zusätzlich kann eine wohldefinierte Error-Payload dem Client die Möglichkeit bieten, zielgerichtet auf den Fehler zu reagieren.

Aber welche Möglichkeit zur Reaktion hat der Client überhaupt? Zunächst einmal könnte er die Anfrage wiederholen. Dies ergibt natürlich nur dann Sinn, wenn der Statuscode signalisiert, dass es sich um ein vorübergehendes Serverproblem handelt. Alternativ könnte der Client auf einen anderen Endpoint zugreifen oder mit gecachten Werten beziehungsweise Fallback-Szenarien arbeiten. Im zweiten Fall spricht man auch von Compensation, also einem fachlichen Plan B. Was sich eventuell ein wenig abstrakt anhört, kann in der Realität absolut sinnvoll sein und zu einer deutlich verbesserten User Experience beitragen.

Nehmen wir einmal als Beispiel die Recommendations, also die personalisierten

Kaufvorschläge, in einem Web-Shop. Sollte der zugehörige Endpoint vorübergehend nicht erreichbar sein, ist es sinnvoll, stattdessen auf gecachte Werte zurückzugreifen. Alternativ könnte ein anderer Service oder ein Content Delivery Network (CDN) eine Liste der allgemeinen Bestseller liefern. Dieses Fallback-Szenario ist aus Sicht des Web-Shop-Nutzers auf jeden Fall besser als eine Fehlermeldung. Die Wahrscheinlichkeit, dass er etwas kauft, ist zwar etwas geringer als bei den personalisierten Recommendations, aber durchaus noch vorhanden.

Ziel sollte es auf jeden Fall sein, die eben beschriebenen Abläufe für Timeouts, Compensation etc. weitestgehend zu automatisieren. Gleiches gilt für die regelmäßigen Health-Checks der Services und eine damit verbundene „Selbstheilung". Ist all dies gegeben, spricht man auch von „Resilient Softwaredesign".

Die beiden aufgezeigten Problemfelder „Versionierung" und „Fehlerbehandlung" zeigen, dass wir durch unseren neuen Architektur-Ansatz zwar viel gewinnen, gleichzeitig aber auch eine Reihe neuer Herausforderungen auf uns zukommen. Neben den bisher genannten gibt es noch eine ganze Menge weiterer. So muss zum Beispiel damit umgegangen werden können, dass sich das (Domänen-)Modell in Teilen sowohl auf den Clients befindet und dort auch verändert werden kann, ohne dass es der Server mitbekommt. Entsprechende Caching- und Synchronisations-Mechanismen müssen konzeptioniert und umgesetzt werden.

Eine weitere Herausforderung ergibt sich für das Thema „Sicherheit". Dies gilt insbesondere für Anwendungen, die bis dato eher intern genutzt wurden und nun auch über den Kanal „Mobile" zur Verfügung gestellt werden sollen. War intern das Thema „Web Security" wahrscheinlich eher untergeordnet, sieht dies bei Anwendungen, die via Hotspot vom Flughafen oder von einem Café aus genutzt werden können, schon ganz anders aus. Einen guten Eindruck davon, welche Probleme auf einen zukommen können, findet man in der Liste der Top 10 des Open Web Application Security Project (OWASP, siehe *„https://www.owasp.org"*).

Last but not least bleibt die Herausforderung, clientseitige Fehler überhaupt mitzubekommen beziehungsweise das Nutzerverhalten zu tracken, um auf Basis dieser Erkenntnisse die verschiedenen Clients zu verbessern. Sowohl bei Single Page Applications als auch bei Native Mobile Apps geben wir automatisch einen Teil der bisher gewohnten Kontrolle über das Nutzerverhalten ab. Wir bekommen zwar noch mit, wenn die Clients RESTful-Calls an den Server absetzen, was aber dazwischen passiert, bleibt uns zunächst einmal verborgen: Warum schließt ein Nutzer den Kaufprozess nicht ab? Ist auf dem Client ein Fehler aufgetreten oder wird einfach nur das von uns ausgedachte UX-Konzept vom Nutzer nicht akzeptiert? Crash Reporting sowie Analytics Tools und Libraries können uns helfen, diese Herausforderung in den Griff zu bekommen und Kanal- beziehungsweise Device-übergreifende Auswertungen zu fahren.

Fazit

In der heutigen Zeit ist es für viele Unternehmen zwingend notwendig, Dienste und Produkte über mehr als nur einen Kanal anzubieten. Mit dieser Anforderung als Treiber sind Multi-Channel oder besser Cross- beziehungsweise Omni-Channel-Architekturen notwendige Voraussetzung dafür, schnell und effizient auf neue Markt-Anforderungen reagieren zu können. Mit einigen wenigen Schritten können bestehende Web-Architekturen in eine dieser Architekturen überführt werden. Allerdings ergeben sich mit dem neuen Architektur-Ansatz auch neue Herausforderungen, wie zum Beispiel Schnittstellen-Versionierung, Modell-Synchronisation, Resilience, Security oder Analytics. Einmal gemeistert, steht am Ende eine hochflexible, skalierbare Architektur, die allen aktuellen, aber auch zukünftigen Ansprüchen genügt und auch offen für neue, heute eventuell noch nicht bekannte oder bespielte Kanäle ist.

Lars Röwekamp
lars.roewekamp@openknowledge.de

Lars Röwekamp, Gründer des IT-Beratungs- und Entwicklungsunternehmens open knowledge GmbH, beschäftigt sich im Rahmen seiner Tätigkeit als „CIO New Technologies" mit der eingehenden Analyse und Bewertung neuer Software- und Technologie-Trends. Ein Schwerpunkt seiner Arbeit liegt derzeit in den Bereichen „Enterprise" und „Mobile Computing", wobei neben Design- und Architekturfragen insbesondere die Real-Life-Aspekte im Fokus seiner Betrachtung stehen. Lars Röwekamp, Autor mehrerer Fachartikel und -bücher, beschäftigt sich seit der Geburtsstunde von Java mit dieser Programmiersprache, wobei er einen Großteil seiner praktischen Erfahrungen im Rahmen großer internationaler Projekte sammeln konnte.

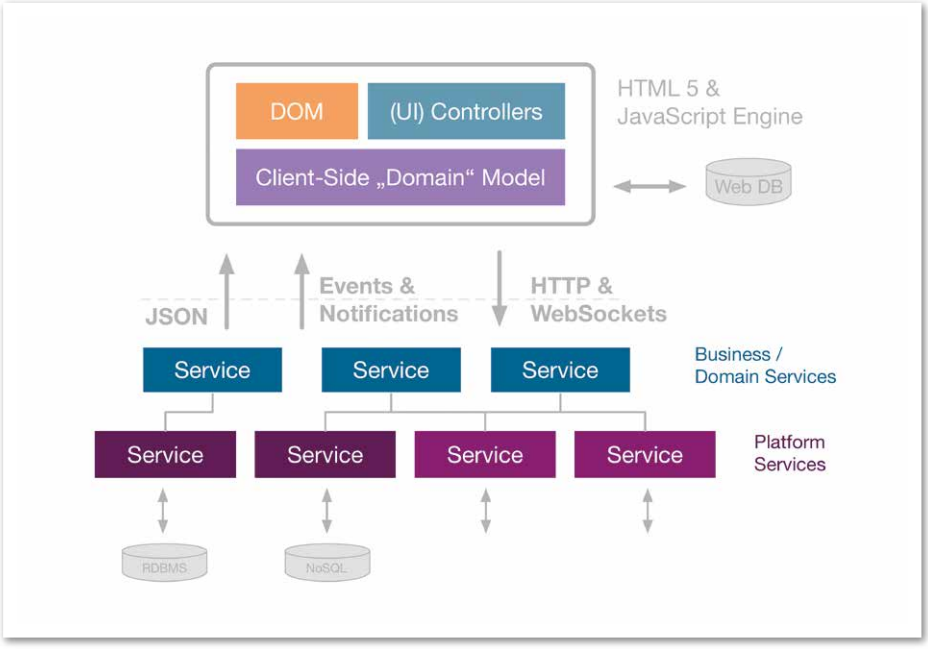
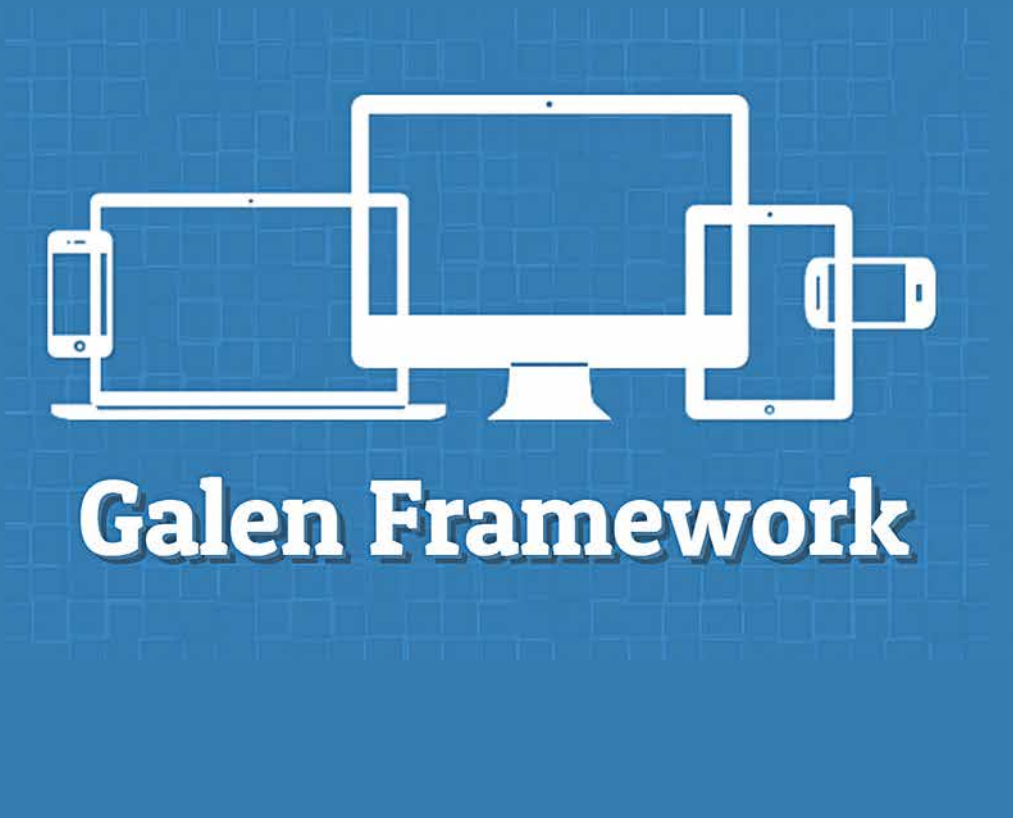


Abbildung 6: Provisionierung via PaaS und Cloud



Galen-tastisch: Layout-Testen leicht-gemacht mit dem Galen-Framework

Jonas Knopf, diva-e Netpioneer GmbH

Mit dem Galen-Framework geschriebene Layout-Tests wirken der Komplexität eines Responsive Webdesign entgegen. Galen hat dabei die Besonderheit, dass nicht das Layout anhand der HTML-Struktur überprüft wird, sondern die relative Positionierung der Elemente zueinander. Dies ermöglicht den Ansatz von Test Driven Development im Frontend-Bereich. Konkret handelt der Artikel davon, wie mit dem Galen-Framework Tests erstellt werden können, die das korrekte Verhalten des Layouts auf unterschiedlichen Displaygrößen überprüfen.

Beim Entwickeln des Frontends einer Webseite reicht es schon lange nicht mehr, das Layout nur für den Desktop-PC zu optimieren. Es muss vielmehr „responsive" sein, sich also an unterschiedliche Bildschirmgrößen und -auflösungen anpassen. Ein in der Praxis weitverbreiteter Ansatz ist es, verschiedene Layouts für den Desktop-PC, Tablets sowie Smartphones zu erstellen. Welches Layout der Benutzer zu sehen bekommt, wird anhand der Bildschirmauflösung des Geräts beziehungsweise des Browsers festgelegt. Je nach Breite des Geräts – und damit des Browsers – bekommt der Benutzer zum Beispiel ein vier- oder ein dreispaltiges Layout zu sehen.

Möchte der Frontend-Entwickler eine Änderung am Layout vornehmen, muss er diese auf allen drei Fenstergrößen des Browsers überprüfen. Sollen diese Änderungen auch zusätzlich auf den Browsern Mozilla Firefox, Google Chrome und Microsoft Edge getestet werden, kommt er so schon auf neun zu überprüfende Kombinationen. Weitere Tests auf unterschiedlichen Devices steigern den Aufwand zusätzlich.

Der Weg zu automatisierten Layout-Tests

Damit ein Testen solcher Änderungen am Layout von Hand überflüssig wird, wurde das Galen-Framework *[1]* entwickelt. Für die automatisierte Überprüfung von Web-

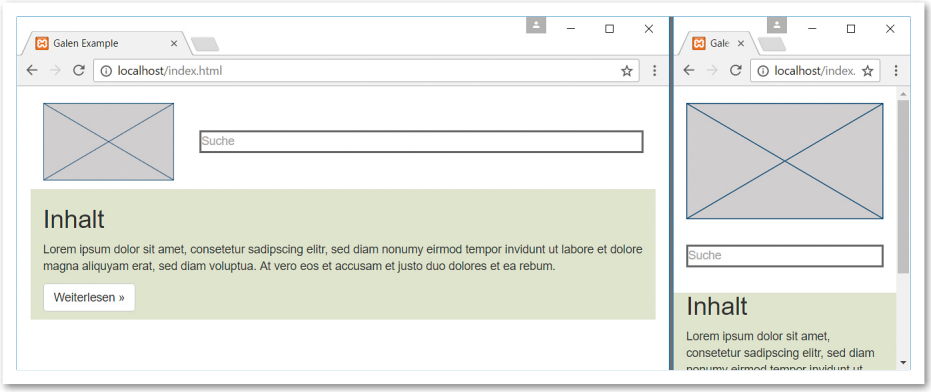


Abbildung 1: Beispiel-Webseite

seiten-Layouts muss zuerst ein Test-Setup aufgestellt werden. Es beschreibt, welche Seite(n) getestet werden sollen und unter welchen Bedingungen ein Test ausgeführt wird. Mögliche Parameter sind zum Beispiel der verwendete Browser sowie dessen Fenstergröße.

Da das Galen-Framework auf Selenium *[2]* aufbaut, können die Tests auf quasi jedem gängigen Browser ausgeführt werden. Da darüber hinaus über Selenium Grid *[3]* auch physikalische Geräte angesprochen werden können, steht einem parallelen Ausführen der Tests auf zum Beispiel einem iPhone und einem Android-Gerät nichts entgegen.

Das Galen-Test-Setup kann über unterschiedliche Wege definiert werden. Zum einen bietet Galen eine eigene DSL an, in der alle relevanten Parameter festgelegt werden können. Darüber hinaus lässt sich das Testsetup auch in Java oder JavaScript schreiben. Dies wird anhand eines Beispiels gezeigt. Diese Webseite verfügt sowohl über eine Desktop- als auch über eine Mobile-Ansicht *(siehe Abbildung 1)*. Die unterschiedlichen Layouts werden mithilfe des Galen-Frameworks getestet.

Sobald die Fenstergröße des Browsers eine Breite von 767 Pixel überschreitet, wird die Desktop-Ansicht dargestellt (links). Dort ist das Logo der Webseite prominent oben links positioniert. Rechts neben dem Logo ist ein Eingabefeld für Suchanfragen. Unterhalb des Logos und des Eingabefeldes ist eine Inhaltsbox. Für Fenstergrößen von 767 Pixel und kleiner erscheint die Mobile-Ansicht. Dort ist das Logo über die volle Fensterbreite gestreckt. Das Eingabefeld ist nun unterhalb des Logos und ebenfalls horizontal gestreckt. Die Inhaltsbox befindet sich unter diesen beiden Elementen.

Layout als JUnit-Test überprüfen

Jetzt wird das Test-Setup in Java als JUnit-Test geschrieben. Dazu verwaltet das Build-Management-Tool Maven die Abhängigkeiten. Diese sind zum einen das Galen-Framework in der Version 2.2.6 *[4]* und zum anderen JUnit in der Version 4.12 *[5]*.

Die JUnit-Testklasse *(siehe Listing 1)* erweitert die Klasse „GalenJUnitTestBase", die über die „galen-java-support"-Abhängigkeit eingebunden ist. Das Galen-Framework erwartet einen parametrisierten JUnit-Test. Aus diesem Grund muss ein Data-Provider implementiert sein. Er besteht aus einer statischen Methode, die mit „@Parameters" annotiert ist und eine Liste von Tupeln zurückliefert. Ein Tupel kann hier als ein Gerätetyp gesehen werden und besteht aus einem internen Namen sowie der Fensterbreite und -höhe des Browsers.

Pro Gerätetyp beziehungsweise Tupel wird bei der Testausführung eine neue Instanz der Testklasse erzeugt. Die einzelnen Werte innerhalb des Tupels werden bei der Instanziierung des Tests den Variablen zugewiesen, die mit „@Parameter" annotiert sind. Im konkreten Fall wird hier ein Test für die Mobile-Ansicht erstellt. Die Fenstergröße des Browsers beträgt 750 x 1000 Pixel. Darüber hinaus muss die „createDriver"-Methode überschrieben werden, die ein „WebDriver"-Objekt zurückliefert. In diesem Fall wird ein WebDriver zur Kommunikation mit dem Firefox-Browser erzeugt.

Nun kann der eigentliche JUnit-Test geschrieben werden *(siehe Listing 2)*. Der Methoden-Aufruf „load" übergibt die URL der zu testenden Webseite sowie die Fensterbreite und -höhe des Browsers an das Galen-Framework. Im Hintergrund werden über Selenium ein Firefox-Browser geöffnet, die übergebene URL aufgerufen und die Fenstergröße angepasst.

```
public class DemoWebsiteTests extends GalenJUnitTestBase {

    @Parameters(name = "{index}: Teste auf {0} ({1}x{2})")
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] {
            {"mobile", 750, 1000}
        });
    }

    @Parameter(0)
    public String deviceName;

    @Parameter(1)
    public int browserWidth;

    @Parameter(2)
    public int browserHeight;

    @Override
    public WebDriver createDriver() {
        return new FirefoxDriver();
    }
}
```

Listing 1: Test-Setup in Java

```
@Test
public void testResponsiveLayout() throws IOException {
    load("http://localhost/index.html", browserWidth, browserHeight);
    checkLayout("localhost.gspec", Arrays.asList(deviceName));
}
```

Listing 2: JUnit-Test

Abbildung 2: Reporting des Galen-Frameworks

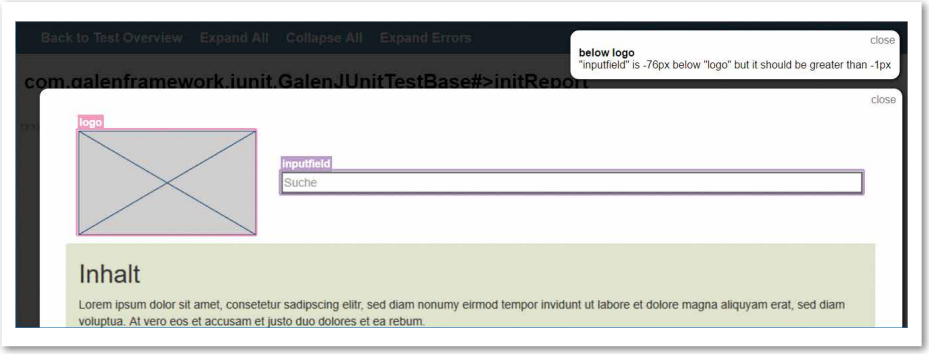


Abbildung 3: Screenshot beim Ausführen des Tests

Damit ist das Test-Setup erfolgreich erstellt. Nun können die eigentlichen Testfälle aufgestellt werden. Sie werden in einer eigenen DSL geschrieben, der Galen Spec Language *[6]*. Mit deren Hilfe wird eine Spezifikation dafür erstellt, wie sich das Layout der Seite verhalten soll. Die Spezifikation der Webseite ist als „.gspec"-Dateien gespeichert.

Über die Methode „checkLayout" wird die Spezifikation geladen und mit der Webseite verglichen. Dazu erhält die Methode als Parameter einen Pfad zur Spezifikationsdatei. Die „localhost.gspec" liegt dabei im Standard-Ordner für Maven-Test-Ressourcen „src/main/resources". Darüber hinaus wird der Name des Gerätetyps in einer Liste gepackt übergeben. Der Sinn hinter diesem Parameter wird zu einem späteren Zeitpunkt deutlich. Vorher wird allerdings die Spezifikation erstellt. Ziel dieses Beispiels ist es, das Verhalten des Eingabefelds zu testen.

Spezifikation der Webseite

Eine Spezifikation in Galen beginnt immer mit der Definition der Elemente, die getestet werden sollen *(siehe Listing 3)*. Unterhalb des Keywords „@Objects" wird für jedes zu testende Element zuerst ein interner Bezeichner vergeben. Im Anschluss wird das Element per ID, CSS-Selektor oder mithilfe eines XPath im DOM adressiert. Im Beispiel werden das Logo und das Eingabefeld per CSS-Selektor im DOM selektiert. Ihnen sind die internen Bezeichner „logo" und „inputfield" zugewiesen.

Geschweifte Klammern oder Ähnliches kennt die Galen Spec Language nicht. Die Semantik wird über Leerzeichen hergestellt. Die einzelnen Element-Definitionen müssen mit bis acht Leerzeichen zum „@Objects" eingerückt sein.

Nun lassen sich einzelne Spezifikationen dazu festlegen, wie sich Elemente auf der Seite zu verhalten haben. Die Spezifikation des jeweiligen Verhaltens wird in Sektionen gruppiert. Diese Sektion wird über einen beliebigen String hergestellt, der von „=" umschlossen ist. Jede Spezifikation des Verhaltens muss sich mindestens in einer Sektion befinden. In *Listing 3* entsteht die Sektion „Header".

Nun kann das Verhalten des Eingabefelds spezifiziert werden. Bisher wurde nur ein Test-Setup für die Mobile-Ansicht erstellt. In diesem Fall soll sich das Eingabefeld unterhalb des Logos befinden. Dies lässt sich mit dem Sprachkonstrukt „below logo" spezifizieren, das sich auf das Element „inputfield" bezieht. Dabei wird auch schon eine Kern-Eigenschaft des Galen-Frameworks sichtbar. Beim Schreiben der Spezifikation werden nicht ein einzelnes Element selektiert und dessen CSS-Eigenschaften überprüft; es wird vielmehr verglichen, wie sich Elemente relativ zueinander verhalten. Wird der JUnit-Test mit der Spezifikation ausgeführt, ist dieser erfolgreich.

Nun können das Test-Setup und die Spezifikation auch für die Desktop-Ansicht der Webseite erweitert werden. Dazu definiert man zuerst einen neuen Gerätetyp „desktop" *(siehe Listing 4)*. Die Fenstergröße des Browsers beträgt dabei 1024 x 768 Pixel.

```
@objects
    logo            css     header img.main-logo
    inputfield      css     header input.search-field


= Header =
    inputfield:
        below logo
```

Listing 3: Spezifikation der Mobile-Ansicht

```
[...]
    @Parameters(name = "{index}: Teste auf {0} ({1}x{2})")
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] {
            {"mobile", 700, 1000},
            {"desktop", 1024, 768}
        });
    }
[...]
```

Listing 4: Erweiterung des Test-Setups für Desktop-Ansicht

```
= Header =
    inputfield:
        below logo

    inputfield:
        aligned horizontally centered logo 1px
```

Listing 5: Spezifikation der Desktop-Ansicht

```
[...]

= Header =
    @on desktop
        inputfield:
            aligned horizontally centered logo 1px

    @on mobile
        inputfield:
            below logo
```

Listing 6: Spezifikationen nach Gerätetypen

Im Layout der Desktop-Ansicht soll das Eingabefeld nicht mehr unterhalb des Logos, sondern links vom Logo mittig positioniert sein. Dies wird in der Galen-Spezifikation wie folgt getestet: Das Element „inputfield" soll horizontal zum Element „logo" positioniert sein. Darüber hinaus soll „inputfield" mittig zu „logo" liegen. Dies wird mit der Spezifikation „aligned horizontally centered logo" abgebildet *(siehe Listing 5)*. An dieser Stelle muss allerdings ein Offset von 1 Pixel hinzugefügt werden. Wäre das Logo zum Beispiel 21 Pixel hoch, wäre dessen genaue Mitte bei genau 10,5 Pixeln. Da bei Pixeln allerdings nur Ganzzahlen angegeben werden können, kann die Mitte nur bei 10 oder 11 Pixeln liegen.

Die neue Spezifikation wird als erster, trivialer Ansatz direkt unterhalb der Spezifikation für die Mobile-Ansicht geschrieben. Wird im Anschluss der JUnit-Test ausgeführt, wird einmal ein Browser-Fenster mit der Mobile-Konfiguration und einmal eines mit der Desktop-Konfiguration geladen. Dabei wird die komplette Spezifikation überprüft, im Fall der Mobile-Ansicht also auch, ob sich das Eingabefeld neben dem Logo befindet. Dies ist allerdings nicht der Fall, weswegen der Test fehlschlägt.

In diesem Fall ist die Ursache des Fehlschlags offensichtlich. In der Praxis ist dies leider nicht immer gegeben, besonders wenn Tests nächtlich von einem Continuous-Integration-System gestartet werden und plötzlich fehlschlagen. Hier kann das Galen-Framework unterstützen, indem es nach dem Ausführen eines Tests ein Reporting erzeugt. Dieses ist im „target"-Verzeichnis des Maven-Projekts abgespeichert.

Reporting des Galen-Frameworks

Das Galen-Framework liefert eine Übersicht über das Test-Setup, also eine Art Heat-Map dazu, welche Elemente auf der Seite mit dem Test abgedeckt wurden, sowie die einzelnen Spezifikationen inklusive der Information, ob eine einzelne Spezifikation erfolgreich war *(siehe Abbildung 2)*. Außerdem erzeugt das Galen-Framework pro Spezifikation einen Screenshot der Seite *(siehe Abbildung 3)*. Mit dessen Hilfe ist nach einem fehlgeschlagenen Test die Seite so sichtbar, wie sie das Galen-Framework beim Test vorgefunden hat.

Um nun Spezifikationen nur für bestimmte Geräte-Typen zuzulassen, kommt der zweite Parameter der „checkLayout"-Methode aus *Listing 2* zum Tragen. Hier wird eine Liste mit Strings übergeben. Für den Test der Desktop-Ansicht befindet sich in der Liste nur der String „desktop", für die Mobile-Ansicht nur „mobile" *(siehe Listing 6)*.

Mithilfe eines „@on desktop" beziehungsweise „@on mobile" sind Spezifikationen definiert, die nur für bestimmte Gerätetypen ausgeführt werden sollen. Auch hier ist auf die Einrückung zu achten. Nur wenn die einzelnen Spezifikationen zum „@on" eingerückt sind, beziehen sie sich darauf. Wird nun der JUnit-Test ausgeführt, ist er erfolgreich. Damit lässt sich das Verhalten des Eingabefelds zum Logo erfolgreich testen.

Neben dem vorgestellten „below" und „aligned" existieren noch einige weitere Sprachkonstrukte zum Überprüfen der Position von Elementen. Darüber hinaus bietet das Galen-Framework zahlreiche weitere Features, die in den Spezifikationsdateien verwendet werden können: Unterteilen von Elementen in Komponenten zum Testen , Import von Spezifikationsdateien in andere Spezifikationen, Vergleich von DOM-Elementen mit Grafiken, Definition von Variablen, Schleifen und noch viel mehr. Die einzelnen Features werden an dieser Stelle nicht weiter vorgestellt, der Autor verweist auf die ausführliche Dokumentation *[6]* der Galen Spec Language.

Test Driven Development im Frontend-Bereich

Mithilfe des Galen-Frameworks lässt sich auch Test Driven Development im Frontend-Bereich einführen. Dabei werden die Tests vor dem eigentlichen Entwickeln der Komponente geschrieben. Vereinfacht gesagt, wird nach dem Schreiben der Tests die Komponente so lange entwickelt, bis die Tests nicht mehr fehlschlagen.

Damit das Galen-Framework die Webseite mit der Spezifikation vergleichen kann, sind in irgendeiner Form Selektoren für die einzelnen Elemente zu definieren. Ein trivialer Ansatz ist zum Beispiel, dass alle zu testenden Elemente um IDs erweitert werden. Ist dies geschehen, werden das Test-Setup und die Spezifikationen erstellt. Im Anschluss wird das Layout des Frontends entwickelt. Ist ein erster Stand erreicht, werden die Layouttests gestartet und der Entwickler erhält sofort Feedback dazu, ob das Layout korrekt ist.

Fazit

Das Galen-Framework eignet sich hervorragend für das Testen responsiver Layouts. Da es auf Selenium beziehungsweise Selenium Grid aufbaut, sind die Tests beinahe auf jedem Gerät und Browser ausführbar. Das Test-Setup wurde beispielhaft in Java als JUnit-Test umgesetzt. Wie eine Webseite auszusehen und wie sie sich auf den unterschiedlichen Geräteklassen zu verhalten hat, ist in einer Spezifikationsdatei festgelegt. Darin werden zuerst die zu testenden Elemente selektiert und anschließend deren Verhalten spezifiziert. Das Galen-Framework überprüft dabei Elemente nicht losgelöst, sondern immer relativ zu anderen Elementen auf der Webseite. Dabei ist das Feature-Set groß. Besonders das ausführliche Reporting inklusive Screenshots erweist sich als außerordentlich praktisch, besonders wenn die Tests über Nacht von einem Continuous-Integration-System erfolgen. Für die Zukunft ist nur zu wünschen, dass sich die IDE-Unterstützung der Galen Spec Language verbessert.

Grundsätzlich sind automatisierte Layout-Tests besonders bei komplexeren responsiven Webdesigns sinnvoll. Wie aber bei allen Frontend-Tests ist der Aufwand beim Schreiben der Tests besonders hoch, wenn die zu testende Webseite dynamisch ist oder Elemente erst durch Browser-Interaktion sichtbar werden. Das Galen-Framework ist ein sehr praktisches Tool, das den Werkzeugkasten eines Entwicklers erweitert.

Hinweis: Der Quellcode dieses Artikels ist auch auf GitHub zu finden *[7]*.

Weiterführende Informationen

[1] Galen-Framework: http://galenframework.com
[2] Selenium http://www.seleniumhq.org
[3] Selenium Grid: https://github.com/SeleniumHQ/selenium/wiki/Grid2
[4] Galen Java Support: https://mvnrepository.com/artifact/com.galenframework/galen-java-support/2.2.6
[5] JUnit: https://mvnrepository.com/artifact/junit/junit/4.12
[6] Galen Spec Language: http://galenframework.com/docs/reference-galen-spec-language-guide
[7] Quellcode auf GitHub: https://github.com/jonasknopf/galen

Jonas Knopf
jonas.knopf@diva-e.com

Jonas Knopf ist Software-Entwickler bei der diva-e Netpioneer GmbH in Karlsruhe. Neben der klassischen Backend-Entwicklung hat er eine Leidenschaft für die Web-Entwicklung mit HTML, CSS und JavaScript.

Thymeleaf – eine Template-Engine für Entwickler und Designer

Gerrit Meier

In den letzten Jahren wurde mit Single Page Applications auf JavaScript-Basis immer mehr der Schritt weg vom serverseitigen Rendering gemacht. Auch die Aussage von Oracle im Rahmen der JavaOne, dass die meisten Anwendungen in der Cloud gänzlich ohne Oberfläche auskommen werden, gibt diesem Ansatz wenig Rückenwind. Warum aber erfreut sich, ganz gegen diesen Trend, die Template-Engine Thymeleaf immer weiter steigender Nutzerzahlen? Der Artikel zeigt, was Thymeleaf anders macht und warum es gerade bei neuen Projekten eine interessante Alternative sein kann.

Bei der Entscheidung, ob eine Web-Anwendung server- oder clientseitig gerendert werden sollte, gibt es viele möglichen Faktoren, die man berücksichtigen und bewerten muss. Dabei sollte, wie immer, das grundlegende (Kunden-)Problem und dessen Lösung die Hauptrolle spielen. Ist einmal die Entscheidung getroffen, sich nicht auf JavaScript-Frameworks wie ReactJS oder Angular zu verlassen, sondern die HTML-Repräsentation auf dem Server zu generieren, steht man vor der nächsten Frage: Mit welcher Engine soll die Seite verarbeitet werden?

An dieser Stelle kann zu traditionellen Bibliotheken wie JavaServer Pages (JSP), Velocity oder Freemarker gegriffen und das Rendering wie immer gelöst werden. Ein Blick über den Tellerrand beziehungsweise eine Google-Suche nach Alternativen in der Java-Welt bringt einen weiteren Kandidaten zum Vorschein: Thymeleaf *[1]*.

Das Open-Source-Projekt, das sich vor allem durch eine übersichtliche, aber dennoch nicht einschränkend wirkende Syntax auszeichnet, existiert seit dem Jahr 2011. Dazu gibt es eine hervorragende Integration in das Spring-Ökosystem, unter anderem durch einen Spring Boot Starter *[2]*. Auf diesen Weg lassen sich sehr schnell Anwendungen unter Verwendung von Thymeleaf-Templates erstellen und die ersten Schritte mit der Engine unternehmen.

Natural Templating

Vor der Besprechung der Syntax und der Funktionsweise von Thymeleaf wird ein einfaches JSP- mit einem Thymeleaf-Template verglichen *(siehe Listings 1 und 2)*. Beide Dateien erzeugen die gleiche Ausgabe, wenn die Seiten aus einer laufenden Anwendung aufgerufen werden. Das Besondere an Thymeleaf ist, dass das HTML-Template auch statisch geöffnet werden kann und ein gemocktes, grafisch korrektes Ergebnis zu sehen ist *(siehe Abbildung 1)*. Im Vergleich dazu die statisch geöffnete JSP-Quelldatei *(siehe Abbildung 2)*.

Wer den Sourcecode beider Template-Engines genauerer betrachtet, erkennt schnell, dass Thymeleaf im Gegensatz zu JSP keine Custom Tags für die Logik verwendet, sondern mit Pseudo-Attributen arbeitet. Dieser Ansatz, auch als „Natural Templating" bezeichnet, beschreibt vor allem die Nutzung der Quelldateien als Prototyp/Mocks.

Ein großer Vorteil besteht darin, dass die Entwickler diese Quelldateien etwa an einen Grafiker geben können, der diese ohne eine laufende Anwendung auf dem Server weiter stylen kann. Solange die Datei direkt geöffnet und nicht durch die Engine verarbeitet wird, werden die unbekannten Attribute vom Browser verworfen und die gemockten Daten aus den bekannten Attributen und Texten in den HTML-Tags angezeigt. Wird die View durch einen Server ausgeliefert,

ersetzt Thymeleaf die Attribute, für die entsprechende Werte definiert wurden.

Alternativ zur Standard-Syntax können auch Data Attributes verwendet werden (also „data-th-href" statt „th:href"). So lässt sich anstelle mit unbekannten HTML-Attributen, die vom Browser ignoriert werden, mit korrekten HTML5-Attributen arbeiten. Dies betrifft nur den statischen Zustand der Datei und ist aus Sicht des Autors als „nice to have" einzuordnen. Im weiteren Verlauf gilt die Konvention, die Attribute aus dem Thymeleaf-Namespace zu nutzen.

Einfache Syntax

Wie man schon am Beispiel erkennen kann, ist die Syntax auch beim ersten Kontakt mit der Engine leicht zu verstehen. Das liegt vor allem daran, dass sich Thymeleaf lediglich auf zwei Kern-Komponenten im Markup konzentriert: Expressions und Processors. In *Listing 3* sind einige grundlegenden Expressions zu sehen, die nun genauer beschrieben werden. Als Expression bezeichnet man den Ausdruck im Wert des Thymeleaf-HTML-Attributs.

In der ersten Zeile wird die wahrscheinlich bekannteste Form angetroffen, Variablen in einer (Java-)Web-Anwendung auszugeben: das Einschließen der Variablen in „${...}". Mit diesem Ausdruck werden Werte oder Objekte aus unserem Controller ausgegeben beziehungsweise verwendet. Als weiterer Bekannter gehört auch der Ausdruck „#{...}" zum Sprachumfang, der für die Ausgabe von fixen Strings/Messages wie „i18n"-Properties verwendet werden kann.

Bis jetzt wurde für die meist verwendeten Zugriffsarten auf Werte oder Objekte keine neue Syntax eingeführt, sodass hier schon zu erkennen ist, dass die Lernkurve von Thymeleaf erfreulich flach ist. Soll es ein wenig mehr Komfort sein, kann man auf die Expression *{...} zurückgreifen. Wie in den Zeilen drei bis fünf zu sehen ist, wird mit ihr auf ein umschließendes Objekt zurückgegriffen, dessen Attribute werden direkt referenziert. Falls kein äußeres Objekt vorhanden ist, verhält sich dieser Ausdruck wie der genannte Variablen-Zugriff per ${...}.

Sehr nützlich ist die URL-Expression „@{...}". In *Listing 3* sind mehrere Beispiele zu sehen, allen voran die Verwendung von relativen Pfaden in der Anwendung. Hier wird der Context-Pfad der Anwendung automatisch ermittelt und ergänzt, sodass ein komplexes beziehungsweise unschönes Präfixen per Hand nicht notwendig ist. URL-Expressions lassen sich auch parametrisieren. Dies bietet zum Beispiel den Vorteil, dass mit relativ elegantem Code in Schleifen bei jedem Eintrag auf die gleiche URL mit unterschiedlichen Parametern gemappt werden kann. Die Parametrisierung kann dabei mit oder ohne Platzhalter erfolgen. Dies führt im ersten Beispiel zu einem Link mit Query-Parameter und im zweiten zur Ersetzung des Platzhalters.

Soll ein Link doch einmal auf eine andere

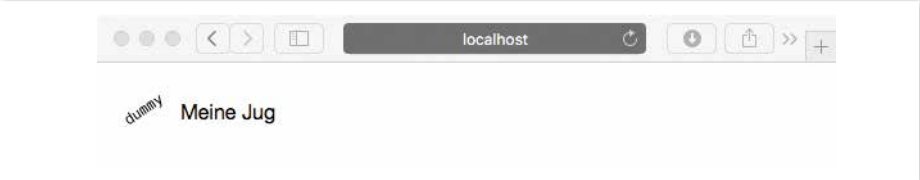


Abbildung 1: Statische Ausgabe von Thymeleaf

Abbildung 2: Statische Ausgabe von JSP

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
    <title></title>
    <link href="${pageContext.request.contextPath}/css/jug_style.css"
rel="stylesheet" type="text/css"/>
</head>
<body>
<table>
<c:forEach items="${jugs}" var="jug">
    <tr class="jug_entry">
        <td><a href="<c:url value="/jug/"/>${jug.id}"><img src="<c:out
value="img/${jug.image}"/>"/></a></td>
        <td><a href="<c:url value="/jug/"/>${jug.id}">${jug.name}</a></td>
    </tr>
</c:forEach>
</table>
</body>
</html>
```

Listing 1

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8"/>
  <title>JUG List</title>
  <link th:href="@{/css/jug_style.css}" href="css/jug_style.css"
rel="stylesheet" type="text/css"/>
</head>
<body>
<table>
  <tr class="jug_entry" th:each="jug : ${jugs}">
    <td><a href="#"><img th:src="@{'/img/' + ${jug.image}}" src="img/dummy.
jpg"/></a></td>
    <td><a href="#" th:text="${jug.name}">Meine Jug</a></td>
  </tr>
</table>
</body>
</html>
```

Listing 2

```
<span th:text="${jug.name}">Platzhalter</span>
<span th:text="#{welcome.message}">Platzhalter</span>

<div th:object="${jug}">
  <span th:text="*{name}">Platzhalter</span>
</div>

<a th:href="@{/jugs}">JUGs</a>
<!-- Parameter ohne Platzhalter: /jug/details?jugId=1 -->
<a th:href="@{/jug/details(jugId=${jug.id})}">JUG Details</a>
<!-- Parameter mit Platzhalter: /jug/1/details -->
<a th:href="@{/jug/{jugId}/details(jugId=${jug.id})}">JUG Details</a>
<a th:href="@{~/otherApplication}">eine andere Anwendung</a>
<a th:href="@{//otherApplication}">ein anderer Server</a>
```

Listing 3

```
<span th:text="${jug.name}">Reintext</span>
<span th:utext="${jug.name}">unescaped Text</span>
<span th:inline="text">Meine JUG: [[${jug.name}]]!</span>
```

Listing 4

```
<tbody>
<tr th:each="jug : ${jugs}" th:object="${jugs}">
  <td th:text="*{name}">erste JUG</td>
</tr>
<tr>
  <td>zweite JUG</td>
</tr>
<tr>
  <td>dritte JUG</td>
</tr>
</tbody>
```

Listing 5

```
<tbody th:remove="all-but-first">
<tr th:each="jug : ${jugs}" th:object="${jugs}">
  <td th:text="*{name}">erste JUG</td>
</tr>
<tr>
  <td>zweite JUG</td>
</tr>
<tr>
  <td>dritte JUG</td>
</tr>
</tbody>
```

Listing 6

```
<table>
<!--/*/ <th:block th:each="jug : ${jugs}"> /*/-->
<tr th:object="${jug}">
  <td th:text="*{name}">erste JUG</td>
</tr>
<tr>
  <td>zweite JUG</td>
</tr>
<tr>
  <td>dritte JUG</td>
</tr>
<!--/*/ </th:block> /*/-->
</table>
```

Listing 7

Anwendung auf dem Server oder ein ganz anderes System zeigen, kann dieses auch in den URL-Expressions mit angegeben werden. Dabei wird der URL ein „~“ (Server relativ) beziehungsweise ein „//“ (Protocol relative) vorangestellt. Im zweiten Fall kann alternativ auch die vollständige URL inklusive Protokoll in der Expression verwendet werden. Dies schafft jedoch eine Bindung an das Protokoll und erzwingt den Einsatz von Environment-Configs, um beispielsweise im Testbetrieb jeglichen Datenverkehr mit anderen Systemen über HTTP anstatt HTTPS laufen zu lassen.

Processors

Unter „Processor“ wird im Thymeleaf-Kontext nicht nur das verwendete HTML-Attribut verstanden, sondern auch seine dahinterliegende Logik und das Resultat bei der Verarbeitung durch die Engine. In Thymeleaf steht eine theoretisch überschaubare Anzahl von Processors zur Verwendung bereit. Theoretisch aus dem Grund, weil es für jedes denkbare Standard-HTML5-Attribut einen Processor gibt. Diese sind intuitiv so benannt, dass man nur vor das Attribut ein „th:“ einfügen muss, um den passenden Processor zu bekommen.

Die wichtigste Funktion einer Template-Engine ist wahrscheinlich die Ausgabe von Informationen, also Text. Um Text in ein HTML-Tag unter der Verwendung von Processors auszugeben, werden die Attribute „th:text“ beziehungsweise „th:utext“ verwendet *(siehe Listing 4)*. Dies ist, wenn es sich als geeigneter erweist, auch durch den Processor „th:inline“ und das Schreiben des auszugebenden Texts in einer Inline-Form möglich. Es ist bei dieser Expression egal, ob die Ausgabe direkt als Kind oder in einer tieferen Ebene erscheinen soll. Dabei sollte beachtet werden, dass in diesem Fall die Ausgabe des Platzhalters beim Öffnen der Quelldatei immer zu sehen ist und nicht durch einen prototypischen Wert ersetzt werden kann.

Schleifen und Beispiel-Daten

Viele Anwendungen basieren auf der Anzeige von Listen, Tabellen oder ähnlichen Darstellungsformen, die meist in einer Schleife ausgegeben werden. Hier bietet Thymeleaf mit „th:each“ *(siehe Listing 5)* genau eine Lösung, um Daten aus einer Sammlung (Iterable, Map, Array etc.) zu durchlaufen. Nun besteht eine der beworbenen Besonderheiten der Engine darin, dass mit gemockten Daten

gearbeitet werden kann.

Wenn das Beispiel in der Form auf dem Server aufgerufen wird, bekommt man nicht nur die Werte geliefert, die unsere Anwendung bereitstellt, sondern am Ende auch noch die Mock-Daten. Um dieses Problem zu umgehen, wurde ein weiterer Processor („th:remove") bereitgestellt *(siehe Listing 6)*. Wie der Name schon erraten lässt, entfernt dieser je nach gewähltem Wert etwa den ganzen Tag („all"), alle Kinder („body") oder alle Kinder bis auf das erste („all-but-first"). Bei Verwendung des letztgenannten Werts kann genau das gewünschte Verhalten erreicht werden: Nach dem Rendern werden nur noch die echten Daten aus der Anwendung angezeigt und die gemockten Elemente sind entfernt.

Bedingungen

Für die Abfrage von Bedingungen gibt es in Thymeleaf wiederum eine überschaubare Anzahl von Processors. Bedingungsabfragen lassen sich durch „th:if" beziehungsweise invers durch „th:unless" realisieren. Bei der Verwendung der Expression ist jedoch etwas Vorsicht geboten, da hier nicht nur pauschal auf einen booleschen Wert im Java-Sinn zu achten ist. Abhängig von der Art des Objekts beziehungsweise seines Werts ergeben sich folgende Möglichkeiten für true: Boolean mit dem Wert „true", eine Zahl oder ein Character ungleich „0" oder ein String, der nicht „no", „false" oder „off" als Wert hat.

Soll es zu einer Fall-Unterscheidung kommen, sollte man anstelle von komplexen „if"-Kaskaden die Processors „th:switch" und „th:case" verwenden. In Thymeleaf gibt es keinen Fallthrough, sondern der erste zutreffende Case wird verwendet. Um den Default-Fall abzufangen, gibt es den speziellen Wert „th:case="*"".

Der Block-Prozessor

Da Ausnahmen die Regel bestätigen, hat auch Thymeleaf seine ganz eigene Ausnahme, den Blockprozessor. Wie in *Listing 7* zu sehen ist, ist dieser Processor kein HTML-Attribut, sondern ein – genau genommen das einzige – HTML-Tag in der Thymeleaf-Welt. Er wird vor allem verwendet, wenn Schleifen realisiert werden, aber kein Eltern-Element verwendet werden kann, das als Iterations-Anfang dient. Das Element wird nach der Verarbeitung nicht an den Browser ausgeliefert, erzeugt jedoch in der statischen Ansicht durch das unbekannte HTML-Tag Fehler im HTML-Source. Wen diese stören, der kann diesen Block mit einem bestimmten Kommentar sowohl vom Browser ignorieren als aber auch von der Engine erkennen lassen.

Templating

Durch die Verwendung von Fragments, einem weiteren Processor, ist es möglich, einfaches Templating mit Thymeleaf vorzunehmen *(siehe Listing 8)*. Die Definition der einzufügenden Inhalte erfolgt dabei in einer ausgelagerten HTML-Datei *(siehe Listing 9)*.

So ergibt sich wieder der Vorteil, dass das zu inkludierende Fragment in ein vollständiges HTML-Dokument eingebettet werden kann, um es statisch zu bearbeiten. Bevor das Einbetten der Fragmente gezeigt wird, sei an dieser Stelle der Hinweis gegeben, dass es bei komplexeren Fragmenten auch

zu einem komplexen Mocking in der Hauptseite kommen kann.

Das Hinzuziehen von HTML-Bestandteilen aus anderen Dateien ist intuitiv mit der Wahl einer von drei unterschiedlichen Einbindungsarten möglich. Zur Auswahl stehen „th:insert" (ab Thymeleaf 3), „th:replace" und „th:include", die dazu führen, dass ein Fragment entweder komplett als Kind eingefügt, das Tag in der Hauptseite ersetzt oder nur der Inhalt des Fragments in die Hauptseite übernommen wird. Dabei bezieht man sich beim Referenzieren immer auf den Datei- und definierten Fragmentnamen.

Fragmente sind darüber hinaus auch parametrisierbar und lassen sich somit sehr flexibel einsetzen. Ein möglicher Anwendungsfall wäre, eine Breadcrumb-Bar in ein Fragment auszulagern und den Besuchs- beziehungsweise Navigationsverlauf über Parameter für die Visualisierung mitzugeben.

```
<div th:include="footer :: footer"></div>
<div th:replace="footer :: footer"></div>
<div th:insert="footer :: footer"></div>
```

Listing 8

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8"/>
  <title>Footer dummy</title>
</head>
<body>
<div th:fragment="footer">
  <span>footer text</span>
</div>
</body>
</html>
```

Listing 9

Kleine Helfer

Damit das Hantieren mit unterschiedlichen Objekt-Typen mehr Komfort bietet, stehen unterschiedliche Helfer-Objekte („Expressions Utility Objects") bereit, um dem Entwickler unter die Arme zu greifen. Da wären zum einen Methoden, die mit Collections (Lists, Set, Maps, Arrays) arbeiten und praktische Funktionalität wie „contains", „isEmpty" oder „sort" anbieten. Hinzu kommt ein guter Support für Datums-, Zahlen- und String-Objekte, der viele üblichen Zugriffe oder Operationen abdeckt. Zwei String-Methoden, die hier als Beispiel des Umfangs erwähnt sind, lauten „listJoin" und „listSplit". Damit lassen sich Listen-Inhalte in einen konkatenierten String verwandeln beziehungsweise anhand eines Trennzeichens eine Liste anlegen.

Gerade an solchen Stellen erkennt man, dass die Entwickler der Engine nicht nur das Nötigste anbieten, sondern auch dem Nutzer helfen, Probleme auf eine elegante Weise zu lösen. Richtig eingesetzt, können die Utility Objects somit sowohl zur Funktionalität als auch zur Lesbarkeit des Codes beitragen.

Ausblick

Dieser Artikel basiert größtenteils auf der Version 2.1 von Thymeleaf, obwohl schon seit Mai letzten Jahres die Version 3 verfügbar ist. Grund dafür ist, dass es bei den angesprochenen Features wenig bis gar keine Änderungen in der Nutzung gibt. Ein weiterer und eigentlich der ausschlaggebende Faktor ist, dass bei der Erstellung dieses Artikels der Support für die aktuelle Version in Spring Boot, mit dem man sehr schnell Thymeleaf-Projekte erstellen kann, noch nicht existierte *(siehe „https://github.com/spring-projects/spring-boot/issues/7450)"*. Man muss jedoch keine Angst davor haben, Templates unter Verwendung der Version 2.1 zu erstellen. Sie lassen sich später auf Version 3 migrieren.

Thymeleaf 3 zeichnet sich vor allem durch ein annähernd komplettes Rewrite der Engine, unter Rücksichtnahme auf die aktuellen Anforderungen, aus. So wurden zum Beispiel die Processor- und Dialect-APIs, die in erster Linie den Erweiterungen der Engine dienen, noch zugänglicher gestaltet. Es existieren sowohl offizielle als auch Community-getriebene Extensions *[3]*, die an dieser Stelle ansetzen. Beispielsweise kann man mit dem thymeleaf-spring-data-dialect *[4]* komfortabel Pagination in den Views ohne viel Eigenaufwand realisieren.

Fazit

Wer sauberen und wartbaren Code in seinen Views haben und zeitgemäße Template-Engines einsetzen möchte, dem sei Thymeleaf ans Herz gelegt. Basiert das aktuelle Projekt auf Spring Boot, bekommt man unter Zuhilfenahme eines Thymeleaf-Starters die passende Konfiguration mitgeliefert. Will man sich nicht an Spring Boot binden, die Engine aber dennoch in einer Spring-Anwendung nutzen, so bietet Thymeleaf Support für die Versionen 3, 4 und 5 (Milestone Releases). Wenn die Entwickler an ihre CSS-Grenzen stoßen, kann die Datei einfach von einem Designer bearbeitet werden, ohne dass Thymeleaf die Views zuerst verarbeitet haben muss. Vor allem im Gegensatz zu dem doch etwas angestaubten JSP bringt Thymeleaf wieder frischen Wind in die Entwicklung von serverseitig gerenderten Web-Anwendungen.

Weiterführende Links

[1] Thymeleaf: http://www.thymeleaf.org
[2] Spring Boot: http://start.spring.io
[3] Thymeleaf Extensions: http://www.thymeleaf.org/ecosystem.html
[4] Spring Data Extension: https://github.com/jpenren/thymeleaf-spring-data-dialect

Gerrit Meier
gerrit.meier@posteo.de

Gerrit Meier beschäftigt sich beruflich hauptsächlich mit Themen aus dem Umfeld der (Java-)Web-Entwicklung. Neue Technologien, Ideen und Ansätze – auch abseits seines technologischen Schwerpunkts – motivieren ihn immer wieder zum Ausprobieren und Verstehen. Da er dies als eine wichtige Kompetenz von Entwicklern sieht, bemüht er sich, dieses Wissen und den Spaß am „Einfach-mal-Machen" weiterzuvermitteln. Gerrit ist Co-Organisator der JUG Ostfalen.

Big-Data-Piloten – ready for Take Off

Dominique Rondé und Alexandra Klimova, Allianz Deutschland AG

Wer sich dem Thema „Big Data" nähert, findet eine Vielzahl von Frameworks, Methoden, Werkzeugen und Internetquellen vor. In diesem Artikel zeigen die Autoren, wie sie ihr Projekt aufgesetzt haben, welche Schritte sie unternommen und welche Erkenntnisse sie gewonnen haben. In Anlehnung an die Luftfahrt ist eine Art Checkliste entstanden, die man bequem im Projektverlauf abarbeiten kann.

Doch zuerst die Frage: Warum dieser Titel? Er umschreibt am besten unsere tägliche Arbeit. Wir sind Technologie-Experten, kennen eine Menge Werkzeuge, Frameworks sowie Programmiersprachen und wissen, wann welches Werkzeug am effizientesten ist. Wir haben Erfahrungen gemacht und sind in den Technologien entsprechend trainiert und zertifiziert.

Allerdings sind wir keine Risiko-Analysten, Produkt-Designer oder Marketing-Experten. Das überlassen wir bewusst jenen Kollegen, die auf diesen Gebieten eine entsprechende Expertise besitzen. Wir steuern so gesehen die Technik, nehmen das richtige Setup vor und stellen sicher, dass die gesamte Verarbeitung entsprechend den Regularien erfolgt; am Zielort angelangt, muss sich unser „Passagier" jedoch selbst beschäftigen. Wir stehen aber weiter beratend zur Verfügung.

Außerdem arbeiten wir im Tagesgeschäft wie eine Cockpit-Crew. Es gilt bei allen signifikanten Eingriffen das Vier-Augen-Prinzip. Wie auch im echten Cockpit gibt es keinen Co-Piloten und Piloten, sondern einen „Pilot Flying", der aktiv an der Tastatur sitzt, und den „Pilot Monitoring", der kritische Befehle aktiv bestätigt und den gesamten Cluster im Auge behält. Jeder ist gleichberechtigt und Entscheidungen werden von beiden Crewmitgliedern getragen.

Before Engine Start

Vor dem Anlassen der Triebwerke werden die Crew vom Piloten gebrieft, die Maschine überprüft und die Navigationsdaten eingetragen. So ist es auch im Projekt. Zu Beginn müssen die Ziele des Vorhabens zumindest grob umrissen sein. Wir haben hierfür eine eigene Kennziffer entworfen. Die Time-to-Data (TTD) wird vor Beginn ermittelt beziehungsweise als eine Zielgröße festgelegt. Sie definiert sich mit „die Zeitspanne, die vergeht, bis ein Stakeholder/Entscheider auf die notwendigen Daten zugreifen kann, um eine fundierte Entscheidung zu treffen". Diese Zeit umfasst insbesondere:

- Auffinden der benötigten Datenquellen
- Aggregation/Korrelation der Daten
- Bereinigen
- Scripts zur Daten-Analyse erstellen
- Ausführen der Skripte und gegebenenfalls feinjustieren
- Visualisierte Ansicht der Daten

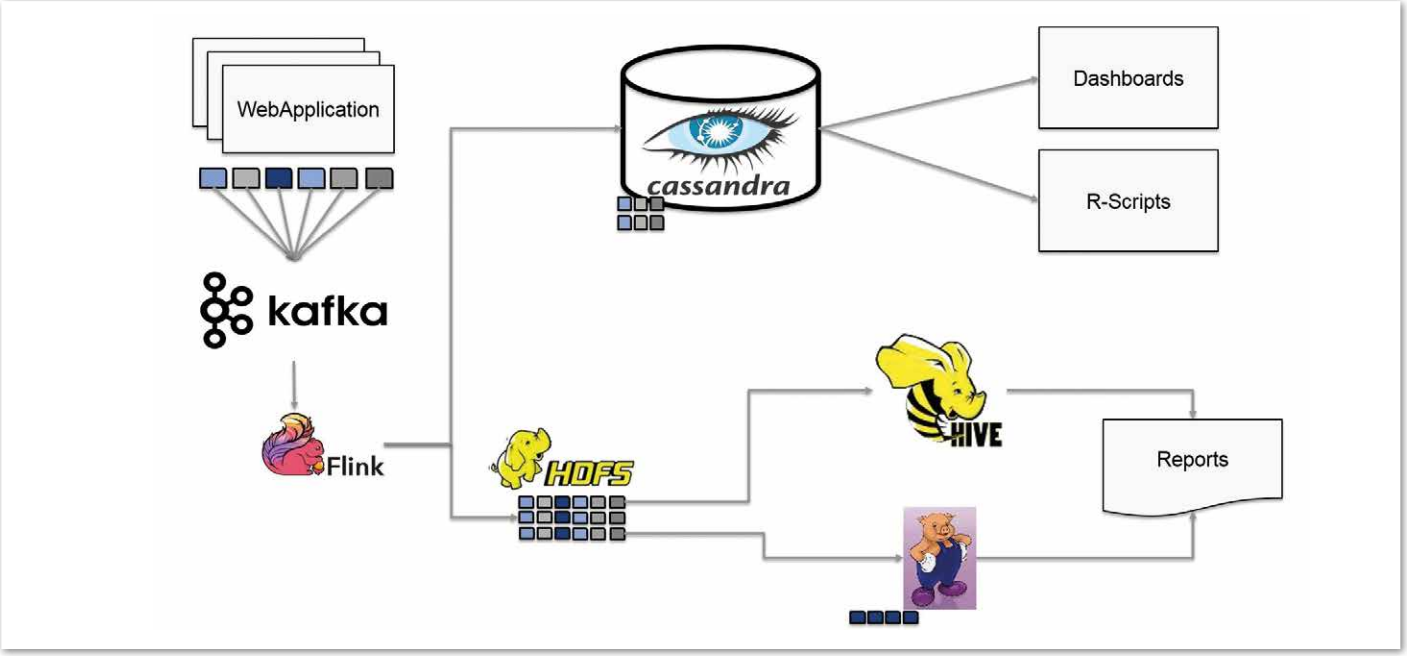


Abbildung 1: Architektur-Skizze

In den meisten Fällen sinkt die TTD nach der ersten Auswertung signifikant, da der jeweilige Anforderer ab der zweiten Auswertung keine weitere Hilfe benötigt, sondern die Daten im Self-Service-Verfahren selbst abrufen kann.

Hinsichtlich der Team-Organisation hat es sich bewährt, die tägliche Arbeit in einer agilen Struktur zu erledigen, um regelmäßige und schnelle Releases zu gewährleisten. Ob es nun Scrum sein muss oder ein Kanban-Ansatz die zielführende Variante ist, muss jeder für sich selbst entscheiden.

Zu Beginn ist ein kurzer Workshop sicherlich keine verschwendete Zeit. Nur wenn die Kommunikation funktioniert und das Team die gleiche Sprache spricht, kann eine Zusammenarbeit funktionieren. Im Team-Briefing können die Organisation, Verfügbarkeiten, Sprachregelungen und Weiteres kurz thematisiert werden.

Before Taxi

Bevor es auf den Weg zum Projektstart geht, liegen einige Themen an, über die sich die „Cockpit Crew" dringend abstimmen muss. Die Analytik sollte keinen Einfluss auf die Technologiewahl oder den Entwicklungszyklus der datenliefernden Systeme haben. Darüber hinaus sollte der zusätzliche Arbeitsaufwand für die Einlieferung von Daten nahezu null sein. Es ist also zu überlegen, wie die Arbeit entsprechend isoliert werden kann. Außerdem ist es wichtig zu klären, ob innerhalb der Aufgabenstellung ReReads von Daten möglich sind und ob die Analytik hiermit umgehen muss.

In der Regel müssen Systeme, die Daten (auch) in Realtime verarbeiten, hochverfügbar sein. Es gilt also, die Skalierbarkeit im Auge zu behalten und zu überlegen, wie man mit Lastspitzen umgeht, etwa „busy seasons" oder eine Marketingaktion, die besser läuft als gedacht.

Zu guter Letzt sollte man mit den Stakeholdern sprechen hinsichtlich etwaiger Anforderungen und rechtlicher Aspekte wie Data Privacy, Data Protection und Data Security sowie Anforderungen, die sich gegebenenfalls aus Compliance-Regelungen ergeben könnten wie zum Beispiel ein Zugriffsaudit.

Before take off

Bis hierher ist schon ein gutes Stück Weg und Zeit vergangen und alle sind startbereit. Es fehlt nun noch eine Architektur, mit der die später eintreffenden Events verarbeitet werden können. Wir folgen einer (fast) klassischen Lamda-Architektur *(siehe Abbildung 1)*.

Die erzeugten Events, hier exemplarisch an einer Webanwendung veranschaulicht, werden an einen Kafka-Cluster gesendet, wobei die Events nach Quelle und Eventtyp in getrennte Topics geschrieben sind. Bibliotheken für Kafka sind in den verschiedensten Programmiersprachen verfügbar und schränken somit die Technologiestacks der Datenlieferanten nicht ein. Zudem bietet der Kafka-Broker eine Art Puffer für kurzzeitige Lastspitzen oder für (un)geplante Downtimes der nachgelagerten Infrastruktur. Somit sind wir in der Lage, jederzeit und ohne Rücksicht auf die Datenlieferanten eine neue Version der Analytik-Jobs, Software- oder Betriebssystem-Updates durchzuführen. Ein Event, das korrekt an einen Kafka-Broker übermittelt wurde, wird sicher verarbeitet werden.

Die erzeugten Events werden entsprechend der Hierarchie der Topics in ein HDFS-System geschrieben und im Rahmen der Verfeinerungen der Modelle in regelmäßigen Abständen ganz oder teilweise neu verarbeitet. Über die Auswahl von Apache Flink und die Abwägung gegen Spark sowie Gründe gegen Apache Storm könnte man sogar einen eigenständigen Artikel verfassen. In der Zusammenfassung bietet Flink im Gegensatz zu Apache Storm eine End-to-End-exactly-Once-Verarbeitung (Kafka -> HDFS und Kafka -> Cassandra (idempotent update)).

Da wir uns in der Hauptsache mit Stream-Processing beschäftigen, war dies auch eines der ausschlaggebenden Argumente gegen Spark, das auf Batches und Microbatches setzt. Darüber hinaus wirkte das API von Flink ebenfalls aufgeräumter. Allerdings sei der Fairness halber erwähnt, dass zum Zeitpunkt der Evaluation die Version 2.0 von Spark noch nicht verfügbar war.

In unserer Cassandra-Datenbank liegen die von den Stakeholdern benötigten Daten in aggregierter Form vor. Auf personen-

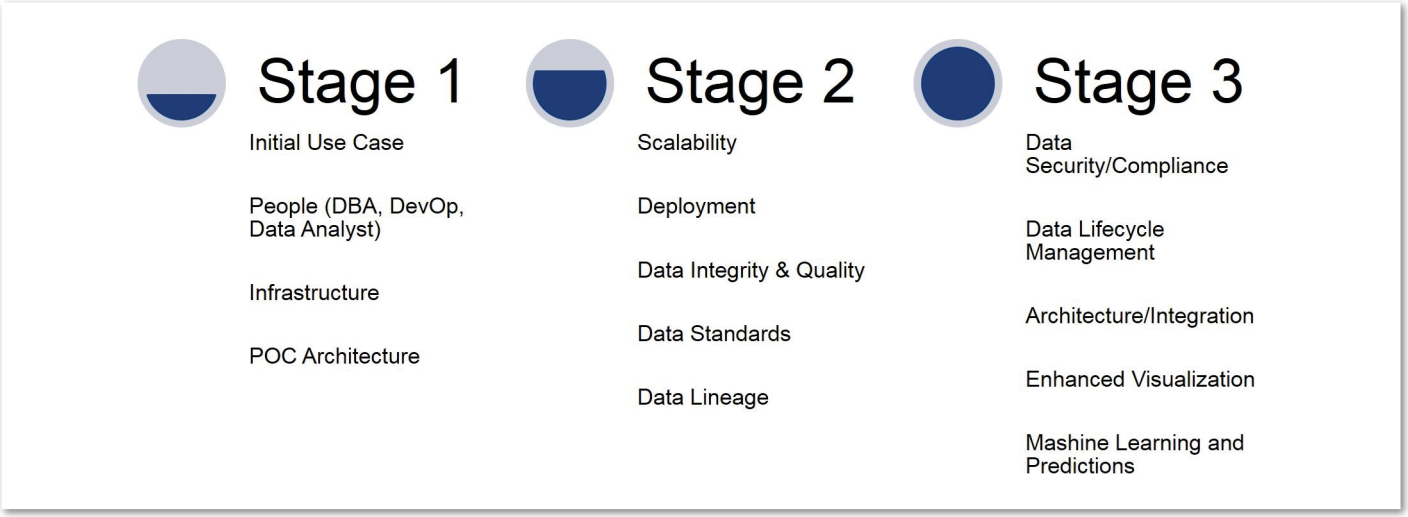


Abbildung 2: Schritte zur Produktionsumgebung

bezogene Daten wird an dieser Stelle sehr bewusst verzichtet. Aus diesem Grund ist es möglich, die vorhandenen Daten auch über Abteilungs- und Funktionsgrenzen hinaus zu betrachten und mit weiteren Daten zu verschränken. Die Auswahl einer Apache-Cassandra-Datenbank für den Speedlayer erfolgte aus Gründen der Skalierung, da die Verarbeitungskapazität nahezu linear mit jedem neuen Node gesteigert werden kann *(siehe „https://www.datastax.com/nosql-databases/benchmarks-cassandra-vs-mongodb-vs-hbase").* Dies ist gerade hinsichtlich möglicher Lastspitzen im kombinierten Betrieb von Analyse und Dateneinlieferung von großem Vorteil. Darüber hinaus macht der masterlose Ansatz das System robust im Fall einer Störung. Fällt im Betrieb ein Knoten aus, kann dies ohne Probleme kompensiert werden. Zwar sind Modelle mit komplexen Relationen schwierig darzustellen, allerdings sorgt hier ab Version 5.0 der Datastax Enterprise Suite die DSE-Graph-Funktion für Abhilfe.

During take off

Bevor es an die Analyse geht, muss der Datentopf erst einmal gefüllt sein. Um den Aufwand in der Entwicklung zu reduzieren, setzen wir stark auf Interceptoren, Filter und Aspekte. So wurden beispielsweise Hooks des OR-Mapper genutzt, um Events bei Änderung einer Entity zu senden. Ebenso verhält es sich mit dem SOAP-Framework, das wir über die Handler-Chain angebunden haben.

Die relevanten Daten aus dem HTTP-Header (Referer, Error-Codes etc.) liefert ein HTTP-Filter, der deklarativ vor jedes zu überwachende Servlet konfiguriert ist.

Felder innerhalb der Angular-Komponenten können per Flag ebenfalls für ein Tracking aktiviert werden, um auf diese Art Events zur Conversion zu erhalten. Hierzu wurden die bereits vorhandenen zentralen Komponenten entsprechend erweitert.

Das meiste Tracking erfolgt nunmehr implizit oder wenigstens deklarativ. Ein explizites Tracking und somit ein manueller Zusatzaufwand erfolgt nur in ausgewählten Fällen, in denen ein Stakeholder im Quelltext einer Anwendung einen Messpunkt benötigt.

Die zentralen Komponenten, wie der Datenproducer oder die fachlichen Event-Implementierungen, werden durch das Analytik-Team geliefert und beispielsweise per Maven eingebunden. Für die Events selbst verwenden wir einfache PoJos, die alle von einem Basis-Event erben. Die fachlichen PoJos sind versioniert; mit jeder Generation können neue Felder hinzugefügt werden. Dies garantiert eine volle Abwärtskompatibilität.

Gerade zu Beginn kann die Finanzierung eines Projekts schwierig werden, da die

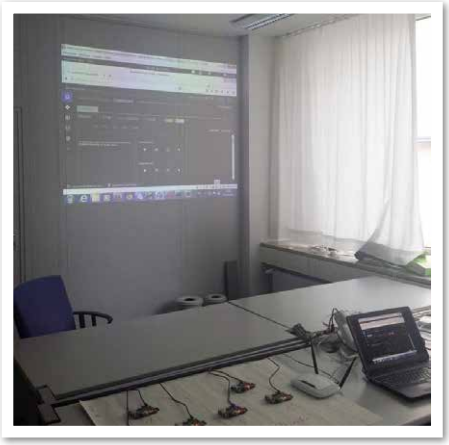


Abbildung 3: Cassandra auf BananaPi-Boards in Aktion

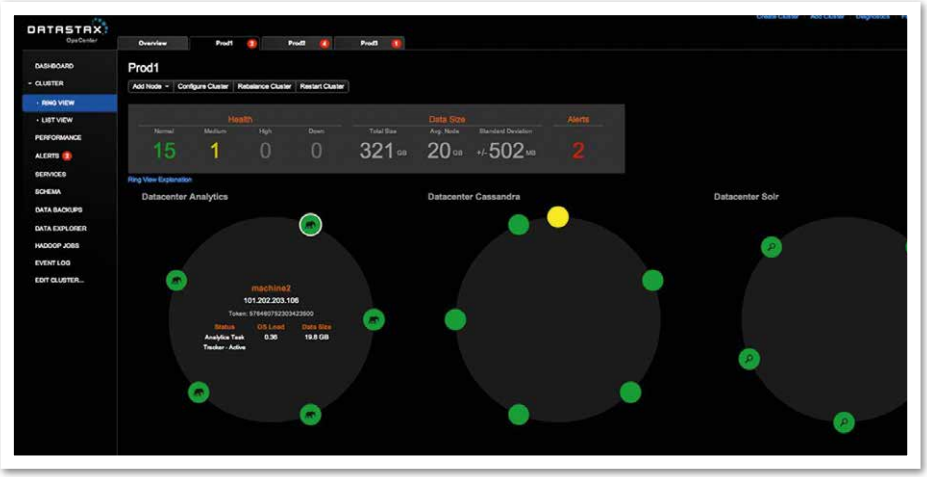


Abbildung 4: DataStax OpsCenter (Quelle: DataStax Enterprise Dokumentation)

Budgetverantwortlichen selbstverständlich erst ein Gefühl für Technologie und Einsatz bekommen möchten. Es ist also nicht ratsam, mit 20 Knoten, 80 CPUs und 2,5 TB Memory beginnen zu wollen.

Unsere ersten Gehversuche haben wir auf einem Cluster, bestehend aus sechs BananaPi-Boards gemacht. Diese Hardware ist mit knapp 45 Dollar pro Einheit inklusive SD-Karte und WLan-Dongle erschwinglich und bietet die volle Flexibilität, alle erdenkbaren Szenarien auszuprobieren, ohne beispielsweise direkt auf Kollegen aus den Rechenzentren zurückgreifen zu müssen *(siehe Abbildung 3)*.

Rechnet man die Kosten für ein Tisch-Hub und USB-Netzteile mit ein, so erhält man für knapp 320 Euro eine voll funktionsfähige Umgebung, mit der Stakeholder, Entscheider und Fachanwender greifbar an das Themenfeld herangeführt werden können. Mit diesem sehr einfach gehaltenen Cluster haben wir fast die gesamte erste Phase des Projekts *(siehe Abbildung 2)* durchgeführt. Er war uns auch für den Entwurf des späteren Architekturbildes, das Sizing der Infrastruktur sowie zur Schulung der Kollegen aus Betrieb und Risikomanagement sehr hilfreich. Wir benutzen ihn auch heute noch zum schnellen Validieren einer These.

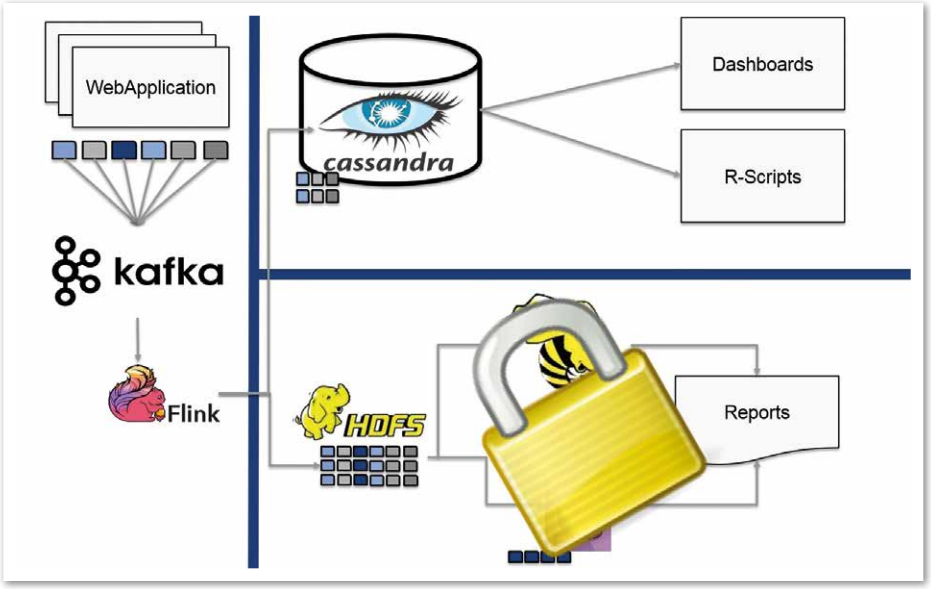


Abbildung 5: Datenschutz

During clime out

Sind die ersten Knoten einmal im Rechenzentrum bereitgestellt, ist es Zeit, sich mit dem Deployment zu beschäftigen, schließlich sollen Änderungen zeitnah ihren Weg auf die produktiven Umgebungen finden. Dabei sind vielleicht bekannte Zyklen von Quartals-, Halbjahres- oder Jahres-Releases eher ungeeignet. Wir setzen in unseren Projekten auf einen kontinuierlichen Build & Deploy, automatisiert auf Anforderung auf unsere Entwicklungs- und Testumgebungen. Das Deployment auf die produktiven Umgebungen geschieht, nach Bereitstellung einer neuen Release-Version, je nach Lastsituation spontan. Hier kommt uns die Architektur mit Kafka zur Zwischenspeicherung der Nachrichten zugute.

Ebenfalls ist in dieser Phase ein (fast) lückenloses Monitoring notwendig. Die meisten Werkzeuge des betrieblichen Managements fokussieren sich jedoch auf die Parameter von Betriebssystem und Hardware. Dies ist für unsere Zwecke allerdings nicht ausreichend. Für die Überwachung der Cassandra-Datenbank setzen wir das DataStax OpsCenter ein *(siehe Abbildung 4)*.

Hier erhalten wir gleichzeitig einen Überblick über die Verteilung der Daten innerhalb des Clusters. Wie auf dem mittleren Ring zu sehen ist, befindet der Cluster sich nicht „in Balance", was auf ein unzureichendes Datenmodell oder eine Störung eines Nodes hinweisen kann. Für die Hadoop-Knoten nutzen wir Ambari Metrics, das einen guten Überblick über die aktuelle Auslastung im Yarn der Data- und Name-Nodes liefert.

At 10.000 Feet

Nun ist es Zeit, ein Thema anzugehen, das für die Techniker zuweilen lästig ist, allerdings eine sehr hohe Wichtigkeit hat. Hinsichtlich des Datenschutzes kann unsere Architektur ihre Stärken voll ausspielen *(siehe Abbildung 4)*.

Währen der Speedlayer rund um unsere Cassandra-Datenbank als eine Art DataHub fungiert *(siehe Abbildung 6)* und nur aggregierte oder anonymisierte Daten enthält, befinden sich innerhalb des HDFS auch relevante,

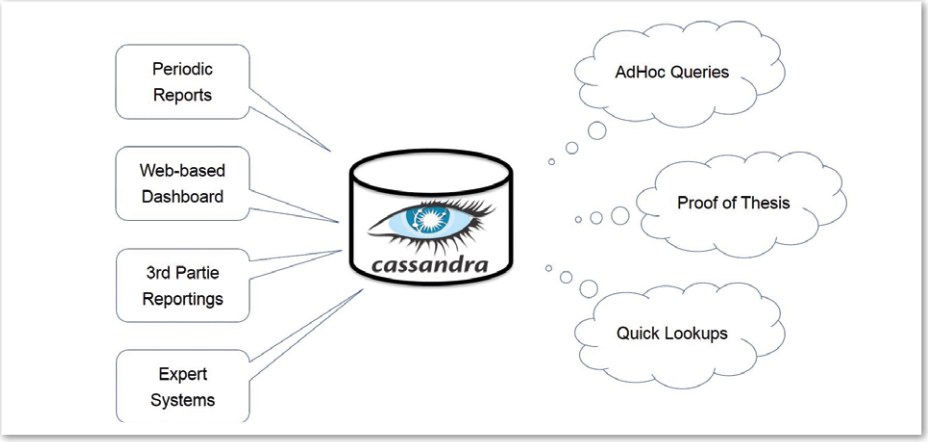


Abbildung 6: DataHub mit Cassandra

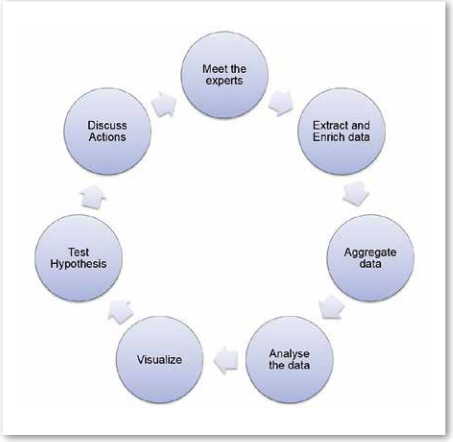


Abbildung 7: Circle of data

```
CREATE ROLE flink;
CREATE ROLE productsales;
CREATE ROLE riskanalyst;

GRANT SELECT ON allianz.solditems TO productsales;
GRANT SELECT ON allianz.riskdata TO riskanalyst;
GRANT MODIFY ON KEYSPACE allianz TO flink;
```

Listing 1: Zugriffsrechte mit Cassandra

```
CREATE TABLE confidentialTable …

with compression_parameters:sstable_compression = 'Encryptor'
...   and compression_parameters:cipher_algorithm = 'AES/ECB/PKCS5Padding'
...   and compression_parameters:secret_key_strength = 128;
```

Listing 2: Encryption mit DataStax Enterprise

personenbezogene Daten, etwa aus der Vertragsanbahnung. Diese Daten werden innerhalb der Aufbewahrungsfristen zum Beispiel zu Revisionsgründen bereitgehalten.

Zudem befinden sich hier auch datenschutzrechtlich anonymisierte, jedoch nicht aggregierte Daten, die zum Füllen des Speedlayer benötigt werden. Der Zugriff auf diesen Datapool ist streng limitiert und jeder Zugriff wird revisionssicher dokumentiert. Ein detailliertes Reporting ist nur auf schriftliche Anordnung einer berechtigten Stelle gestattet. Die Daten selbst liegen innerhalb kryptografischer Container und sind somit etwa bei einem Diebstahl der Hardware nicht nutzbar.

Der Zugriff auf den SpeedLayer erfolgt weniger restriktiv. Interessierte Abteilungen können ihre Absicht vortragen und werden in Folge auf die benötigten Datenquellen freigeschaltet. Dabei werden ausdrücklich nur Leserechte für die entsprechenden Tabellen vergeben. Hiermit wird die Qualität der Daten sichergestellt und versehentliche Änderungen sind ausgeschlossen *(siehe Listing 1)*. Auch wenn es aktuell nicht benötigt wird, besteht die Möglichkeit, die Daten innerhalb des Speedlayer ebenfalls zu verschlüsseln *(siehe Listing 2)*.

Bevor es nun auf die Reiseflughöhe geht, noch einige Worte zur Visualisierung der Daten. Die Auswahl eines Visualisierungstools ist schwer und wir können an dieser Stelle keine klare Empfehlung aussprechen. Im Rahmen unserer Evaluation haben wir Erkenntnisse gewonnen, die wir gerne teilen:

- *Zeppelin*
  Einfache Installation
  Für Entwickler und Data Scientist vollkommen okay
  Für Vorstände, Entscheider, Businessanwender eher weniger geeignet
- *MicroStrategy*
  Unterstützt laut Website nur Cassandra 2.x
  Hat in unseren Testläufen immer versucht, in die Cassandra-Datenbank zu schreiben
  Kann Cassandra über die Spark-SQL-Integration verwenden
- *Tableau*
  Kein eigenes Plug-in für Cassandra
  Kann Cassandra über die Spark-SQL-Integration verwenden
- *D3.js*
  Ein ausgezeichnetes Framework für Visualisierungen auf JavaScript-Basis
  Sehr viele fertige Charts
  Es wird mindestens ein AngularJS-Entwickler benötigt, um einen Report anzupassen
- *R*
  Einfache Visualisierungen sind möglich
  Es sind Kenntnisse in R erforderlich

At cruising altitude

Der Cluster läuft einwandfrei und die Datentöpfe füllen sich. Es besteht kein Grund zum entspannten Zurücklehnen. In Kooperation mit verschiedenen Anforderern, Stakeholdern, Analysten und Entwicklern hat sich der „Circle of Data" bewährt *(siehe Abbildung 7)*. Hier ist ein kollaborativer Arbeitsansatz wichtig. Während des „Meet the Experts" treffen sich alle Beteiligten und diskutieren, welche These bearbeitet werden soll und wie ein Beweis aussehen könnte. Es werden auch die benötigten Datentöpfe identifiziert. In weiteren Schritten werden die Daten extrahiert, angereichert und aggregiert. Anschließend erfolgen die mathematischen Analysen und die entsprechende Visualisierung.

Die gewonnenen Erkenntnisse werden genutzt, um die These zu widerlegen. Es hat sich bewährt, eine These zu widerlegen, da negative Beweise schwieriger zu führen sind als positive Bestätigung. Die Ergebnisse werden im Team erneut diskutiert und bei Bedarf mithilfe weiterer Daten oder anderer Eingangsthesen nochmals bearbeitet.

Dominique Rondé
dominique.ronde@allsecur.de

Dominique Rondé ist Big-Data-Architekt und unterstützt aktuell die Allianz Deutschland AG. Der Fokus seiner Arbeit liegt auf den aktuellen Big-Data-Technologien sowie der Daten-Analytik. Als bekennender Java-Nerd ist er seit dem Jahr 2002 sowohl in der Konzeptions- als auch in der Implementierungsphase beteiligt. Er ist zertifizierter Solution Architect für Apache Cassandra.

Alexandra Klimova
alexandra.klimova@allsecur.de

Alexandra Klimova ist Big-Data-Architektin und unterstützt aktuell die Allianz Deutschland AG. Sie beschäftigt sich schwerpunktmäßig mit dem Hadoop-Ökosystem sowie der Hortonworks-Data-Platform. Aufgrund ihrer technischen Ausbildung ist sie sowohl in der Konzeptions- als auch in der Implementierungsphase beteiligt. Sie ist Trainerin für das gesamte Curriculum aus dem Hause Hortonworks.

Predictive Analytics mit Apache Spark

Dr. Ralph Guderlei, eXXcellent solutions GmbH

In den meisten Unternehmen sind die Kernprozesse über unterschiedliche Informationssysteme abgebildet. Teilweise kommt dazu Standard-Software zum Einsatz. Oft werden die Prozesse allerdings auch über Individual-Lösungen umgesetzt. Diese sind über einen langen Zeitraum im Einsatz und enthalten daher große Mengen an Daten, die jedoch oft nur für die Umsetzung der Prozesse, aber nicht für tiefergehende Analysen genutzt werden. Solche Analysen lassen sich aber für Prozess-Verbesserungen oder neue Features nutzen. In diesem Zusammenhang spricht man häufig von „Predictive Analytics".

Predictive Analytics beschäftigt sich mit Verfahren für die Extraktion von Informationen aus Daten mit dem Ziel, aus den gewonnenen Informationen Trends oder Verhaltensmuster abzuleiten. In der Regel wird der Begriff für Prognosen von zukünftigen Ereignissen verwendet, die Analysen können sich aber auch auf gegenwärtige oder vergangene Ereignisse beziehen. Für Predictive Analytics kommen Verfahren aus dem Data Mining und induktive statistische Methoden zum Einsatz. Typische Verfahren sind Regressionen oder Klassifikationsmethoden.

Ein Beispiel bildet den Rahmen dieses Artikels: Ein Unternehmen betreibt ein Forum für den Kunden-Support. Die Zufriedenheit der Kunden ist im Wesentlichen durch die Antwortzeit bestimmt. Aufgabe ist es nun, aus der vorhandenen Foren-Datenbank Fragen und Antworten zu bestimmen, Antwortzeiten zu messen und die zukünftige Entwicklung der Antwortzeiten zu prognostizieren. Der Artikel zeigt, wie diese Aufgabe mithilfe von Apache Spark umgesetzt werden kann. Die Beispiele sind in Scala geschrieben, Apache Spark kann allerdings genauso mit Java (oder Python) programmiert werden.

Daten laden

Zunächst müssen die Daten für die Analyse aus dem bestehenden System geladen werden. Apache Spark bietet dazu eine Reihe von Möglichkeiten: (CSV-) Dateien, Zugriff auf relationale Datenbanken mit Spark SQL oder diverse Message Queues (als Quelle für Spark Streaming). Für das Szenario der Foren-Datenbank bietet sich der Zugriff mit Spark SQL an. Der Zugriff via SQL erlaubt eine gewisse Vorstrukturierung und Vorfilterung der Daten. Das Code-Beispiel in *Listing 1* zeigt das Vorgehen.

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession
      .builder()
      .appName("message board trends")
      .getOrCreate()

val jdbcDF = spark.read
      .format("jdbc")
      .option("url", "jdbc:...")
      .option("driver", "my.jdbc.Driver")
      .option("user", "user")
      .option("password", "password")
      .option("dbtable", "(select ...)")
      .option("fetchsize", 1000)
      .load()
```

Listing 1

Zunächst ist Spark (in Form einer Spark-SQL-Session) selbst zu initialisieren. Danach wird die Session genutzt, um die Daten zu laden. Spark SQL verwendet intern JDBC. Daher sind für den Zugriff die üblichen Informationen wie JDBC-URL und Treiberklasse sowie Benutzername und Passwort erforderlich. Spark SQL lädt die Daten dann aus einer angegebenen Tabelle oder View (Parameter „dbtable“).

Um Daten aus einer beliebigen SQL-Query zu laden, muss diese als Sub-Select verpackt sein. Deswegen die Klammern um das angedeutete Select-Statement im Beispiel. Die Option „fetchsize“ steuert, wie viele Zeilen pro Zugriff geladen werden sollen (analog zu JDBC). Manche Datenbanken nutzen standardmäßig eine kleine „fetchsize“. Daher kann dieser Parameter eine wichtige Stellschraube für die Performance des Datenbank-Zugriffs sein.

Daten aufbereiten

Nachdem die Rohdaten jetzt in Spark geladen sind, müssen sie aufbereitet und die gewünschten Informationen berechnet werden. In unserem Beispiel sind also Fragen und Antworten zu identifizieren und die resultierenden Antwortzeiten zu berechnen.

In Spark liegen die Daten als Resilient Distributed Dataset (RDD) vor. Dieser bildet eine Abstraktion ähnlich einer Collection (beziehungsweise eines Java-8-Streams) über die (potenziell) auf einem Cluster verteilten Daten. Für den Entwickler stellt es sich also so dar, als ob er mit einer normalen, lokalen Collection arbeiten würde, die Arbeit wird aber automatisch auf den Cluster verteilt. Das ermöglicht einfache und gut lesbare Programme. RDD stellt dem Entwickler die üblichen Operationen wie „filter“, „map“, „fold“, „sort“ etc. zur Verfügung.

Die SQL-Abfrage liefert ein RDD von Tupeln. Diese enthalten die einzelnen Werte des Eintrags im ResultSet. Tupel in Scala

```
responseTimes = jdbcDF.rdd
  .map((_._1, Instant.parse(_._2), _._3))
  .groupByKey()
  .filter(entry => entry._2.length == 2)
  .mapValues(thread => {
    val sorted = tread.toList.sortBy(_._2)
    val responseTime = ChronoUnit.HOURS.between(sorted(0)._2, sorted(1)._2)
    (sorted(0)._2, responseTime)
  })
  .filter( entry => entry._2._2 > 0)
```

Listing 2

sind eine bequeme Methode, um dynamisch mehrwertige Daten zu beschreiben und mit diesen zu arbeiten. Gerade bei der Analyse von Daten kommt es oft vor, dass sich die Struktur der Daten während der Bearbeitung ändert. Wenn jeder Zwischenschritt statisch typisiert werden müsste, wäre die Bearbeitung sehr schwerfällig; mit Tupeln entfällt die Notwendigkeit der statischen Typisierung. Dabei geht jedoch das Wissen über die genaue Zusammensetzung der Daten leicht verloren und bei komplexen Vorgängen kann man schnell den Überblick verlieren. In unserem Fall gehen wir davon aus, dass das Tupel drei Werte besitzt: die ID des Thread, das Datum des Posts (als String) und den Nachrichtentext. Das Code-Fragment in *Listing 2* berechnet die Antwortzeit zu einer Frage.

Im ersten Schritt wird der Datums-String in einen Instant gewandelt, um später einfacher rechnen zu können, im zweiten Schritt werden die Datensätze nach Thread-ID gruppiert. Danach lassen sich die Nachrichten pro Thread betrachten. Die Funktion „groupByKey“ liefert ein zweiwertiges Tupel: den Schlüssel, nach dem gruppiert wurde, und die Liste aller gruppierten Einträge.

Der Einfachheit halber wird für die Berechnung der Antwortzeiten angenommen, dass ein Thread mit einer beantworteten Frage genau zwei Einträge enthält: Frage und Antwort. Ohne diese Annahme stellt sich das Problem der Klassifikation der Nachrichten in Fragen und Antworten. Fragen lassen sich einigermaßen zuverlässig (zumindest in den dem Beispiel zugrunde liegenden, realen Daten) durch das Vorhandensein eines „?“ im Text erkennen. Die darauffolgende Nachricht ohne „?“ wird dann als Antwort gewertet. Auch diese Klassifikation erscheint zu einfach zu sein, eine stichprobenartige Prüfung der Daten hat aber auch hier eine hinreichende Genauigkeit gezeigt.

Zunächst werden alle Threads entfernt, die nicht genau zwei Einträge enthalten. Unter den zugegebenermaßen stark vereinfachenden Annahmen ist die Berechnung der Antwortzeiten dann simpel: Sie ergibt sich als Differenz der Datumsangaben der beiden Einträge. Die Funktion „mapValues“ führt die Berechnung pro Thread durch. Für die weitere Analyse liefert die Funktion jedoch nicht nur die Antwortzeit, sondern ein Tupel aus Datumsangabe der Frage und der berechneten Antwortzeit. Deswegen findet auch zunächst eine Sortierung der Einträge statt (die Frage ist dann der erste Eintrag). Für die reine Berechnung wäre diese nicht notwendig.

Prognose erstellen

Die Antwortzeiten sind nun bekannt. Diese Informationen waren im System bisher nicht explizit vorhanden, konnten aus vorhandenen Daten aber (zumindest unter gewissen Annahmen) einfach berechnet werden. *Abbildung 1* zeigt die Verteilung der Antwortzeiten.

Dort ist erkennbar, dass mehr als 80 Prozent der Fragen innerhalb von fünf Tagen beantwortet wurden. 50 Prozent der Fragen wurden sogar innerhalb eines Tages beantwortet. Die durchschnittliche Antwortzeit betrug 3,5 Tage. Dieses Wissen ist zwar interessant, es ermöglicht allerdings noch keine Prognose der zukünftigen Entwicklung.

Für die Prognose der Antwortzeiten kommt eine lineare Regression der Antwortzeiten der letzten zehn Tage zum Einsatz. Dazu wird die von Spark in der MLLib bereitgestellte Funktion verwendet. Die Machine Learning Library (MLLib) umfasst Implementierungen der wichtigsten Algorithmen für Machine Learning. Neben den bereits erwähnten Regressionsverfahren gibt es beispielsweise Algorithmen für Klassifikationen oder Clustering *(siehe Listing 3)*.

Zur Vorbereitung der Klassifikation sind zunächst die berechneten Daten nochmals zu adaptieren. Aus dem Datum eines Eintrags werden die Differenz in Tagen zum Beginn des Beobachtungszeitraums errechnet und dann alle Einträge mit Differenz kleiner 0 (also vor dem Beobachtungsbeginn) und größer 10 Tage entfernt.

Die letzten zwei Schritte in der Vorbereitung dienen dazu, die Daten in die für die lineare Regression notwendige Form zu bringen. Der Algorithmus erwartet ein sogenanntes „DataFrame" aus zwei Spalten, eine mit dem Namen „label", die andere mit dem Namen „features". Die erste Spalte enthält die Zielgröße (also in unserem Fall die berechneten Antwortzeiten), die andere einen Vektor von Einflussgrößen (in unserem Fall die Anzahl der Tage seit dem Beginn des Beobachtungszeitraums).

Nach der Datenaufbereitung werden die Parameter des linearen Modells geschätzt. Im Endeffekt beschreiben sie eine Trendgerade. Die Steigung der Gerade ist der einzige Eintrag im Vektor „coefficients" des Modells. Ist dieser größer als null, kann von steigenden Antwortzeiten ausgegangen werden.

Fazit

In den Daten vorhandener Informationssysteme schlummert einiges Potenzial für neue Erkenntnisse. Mithilfe von Apache Spark ist es einfach möglich, die Daten zu extrahieren und für weitere Analysen aufzubereiten. Die von Apache Spark bereitgestellten Abstraktionen und Funktionalitäten, insbesondere die Algorithmen in der MLLib, sind ein mächtiges Werkzeug für Entwickler zur Umsetzung von Aufgaben im Predictive-Analytics-Umfeld.

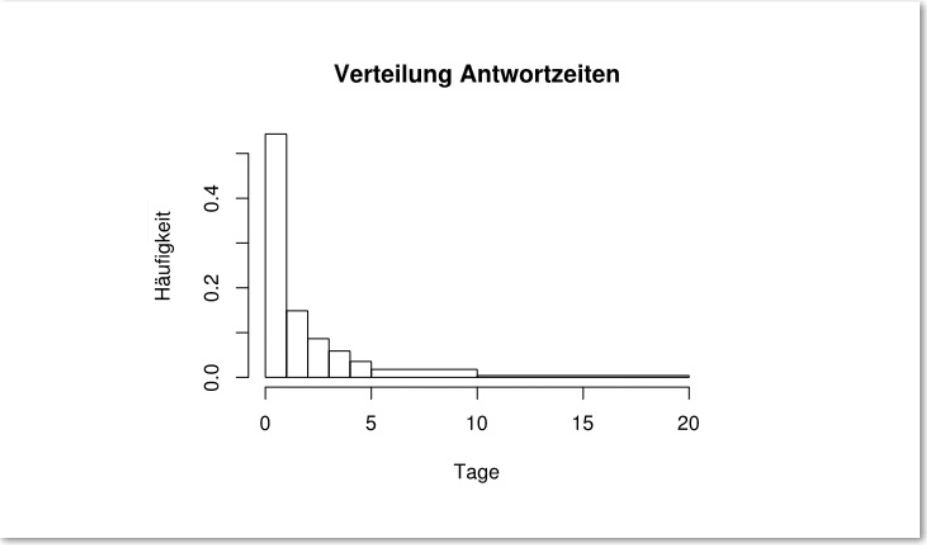


Abbildung 1: Die Antwortzeiten

```
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.linalg.{Vector, Vectors}
val startDate = Instant.now().minus(10, ChronoUnit.DAYS)
val trainig = responseTimes.map((k,v) => (ChronoUnit.DAYS.
between(startDate, v._1),v._2)
  .filter(entry => entry._1 >=0 && entry._1 < 10)
  .map((_._2, Vectors.dense(_._1)))
  .toDF("label", "features")
val model = new LinearRegression().fit(training)
println(s"Coefficients: ${model.coefficients} Intercept: ${model.inter-
cept}")
```

Listing 3

Dr. Ralph Guderlei
ralph.guderlei@exxcellent.de

Dr. Ralph Guderlei ist Technology Advisor bei der eXXcellent solutions GmbH in Ulm. Neben der Arbeit als Architekt/Projektleiter in unterschiedlichen Kundenprojekten berät er Teams in technologischen und methodischen Fragestellungen. Seine Schwerpunkte dabei sind zukunftsfähige Software-Architekturen, Web-Entwicklung und alternative JVM-Sprachen.

Lagom: Einmal Microservices mit allem, bitte!

Lutz Hühnken, Freiberufler

Microservices sind das Architektur-Muster der Stunde und an entsprechenden Frameworks herrscht kein Mangel. Mit Lagom betritt ein weiterer Player die Arena, in der sich schon Spring Boot, Dropwizard und andere tummeln. Wodurch unterscheidet es sich von den anderen?

Man kann davon ausgehen, dass den Programmierern von Lightbend (manchen noch bekannt unter dem vormaligen Namen „Typesafe") auch bewusst war, dass es schon andere Microservice-Frameworks gibt – zumal sie selbst schon zwei Open-Source-Frameworks unter ihren Fittichen haben, nämlich Play! *(siehe „http://www.playframework.com")* und Akka HTTP *(siehe „http://github.com/akka/akka-http")*, die durchaus geeignet sind, um Java-Microservices zu entwickeln. Trotzdem hielten sie es für eine gute Idee, mit Lagom (schwedisch für „gerade richtig", „nicht zu viel, nicht zu wenig") noch ein weiteres auf den Markt zu bringen. Die wesentlichen Design-Entscheidungen dabei waren:

- Bei Microservices geht es nie um einen einzelnen Service, daher sollte auch das Framework sich nicht auf den einzelnen Service fokussieren, sondern auf das (aus Microservices bestehende) System. Kommunikation zwischen den Services und ein Entwicklungs-Setup, das es erlaubt, effektiv an mehreren Services gleichzeitig zu arbeiten, sollten zum Framework gehören.
- Lagom will den Entwicklern Entscheidungen abnehmen. Statt nur eine HTTP/REST-Library vorzugeben und es den Entwicklern zu überlassen, sich für alles Weitere aus dem Pool von Netflix-OSS und anderen zu bedienen, gibt es ein Standard-Setup für den gesamten Stack (HTTP, JSON, Datenbank, Message-Bus).
- Entwickler sollen an die Hand genommen werden, Microservices skalierbar und resilient (also ausfallsicher, fehlertolerant) zu entwickeln. Jeder einzelne Service soll als Cluster mit variabler Anzahl von Nodes laufen können.
- Die Features von Java 8 sollen konsequent genutzt werden, eine Verwendung mit älteren Java-Versionen ist nicht vorgesehen.

Java 8 ein Muss

Der letzte Punkt fällt jedem, der sich Lagom-Beispielcode ansieht, sofort ins Auge. Wer mit „Lambdas", „Optional" und „CompletionStage" noch auf Kriegsfuß steht, sollte einen Crash-Kurs belegen, denn in Lagom wimmelt es nur so davon *(siehe Listing 1)*. Hauptgrund dafür ist die asynchrone Natur des Frameworks. Lagom folgt nicht dem „Thread-per-Request"-Modell, wie es das Servlet-API ursprünglich vorgesehen hat.

In Web-Frameworks, die auf dem Servlet-API beruhen, ist dem eingehenden HTTP-Request ein Thread fest zugeordnet, der erst wieder freigegeben wird, wenn die HTTP-Response gesendet ist. Das setzt der Skalierbarkeit natürliche Grenzen – auch mit der schnellsten Hardware ist die Anzahl der nativen Threads, die das Betriebssystem parallel verwalten kann, begrenzt.

Lagom kommt mit wenigen Threads aus, da der ausführende Thread bei jeglicher Kommunikation – sei es mit der Außenwelt oder zwischen den Komponenten des Service – immer wieder freigegeben wird. Kaum etwas in Lagom ist ein einfacher Methodenaufruf – stattdessen ist der Standard-Rückgabewert eine „CompletionStage". Dieser können dann wiederum ein (oder

```
@Override
public ServiceCall<User, NotUsed> createUser() {
  return request -> {
    return friendEntityRef(request.userId)
         .ask(new CreateUser(request))
         .thenApply(ack -> NotUsed.getInstance());
  };
}
```

Listing 1: Lambdas, überall Lambdas

mehrere) Callback(s) übergeben werden, die aufgerufen werden, wenn die asynchrone Ausführung abgeschlossen ist.

Es wird schnell deutlich, dass Java ein bisschen „syntactic sugar" für diese asynchrone, Callback-basierte Programmierung fehlt, sodass der Code nicht immer ästhetisch ansprechend ist. Das darunter liegende Konzept überzeugt jedoch – um die parallele Rechenleistung moderner Multicore-Prozessoren auszunutzen, führt an der asynchronen Programmierung kein Weg vorbei.

Eine wichtige Begleiterscheinung dieser Art der Programmierung ist die Notwendigkeit, mit unveränderlichen (immutable) Objekten zu arbeiten. Das ist nicht weiter kompliziert, für viele allerdings sicher eher ungewohnt. Das typische Objekt in der Objektorientierung zeichnet sich ja durch die Eigenschaften aus, eine Identität und einen veränderlichen Zustand zu besitzen. Lagom erfordert ein Umdenken mehr in Richtung funktionaler Programmierung: Ich übergebe einer Funktion einen Wert (oder in Lagom auch häufig: sende eine Nachricht), der dann auch konstant bleibt. Es ist keine Zauberei, solche unveränderlichen Klassen in Java zu schreiben, aber es kommt doch etwas an Boilerplate-Code zusammen, denn es müssen beispielsweise „equals" und „hashCode" jedes Mal mit einer wertebasierten Variante überschrieben werden. Glücklicherweise ist die Arbeit mit solchen „value objects" oder „data objects" auch in der restlichen Java-Welt inzwischen recht gängig, und etwa die „@Value"-Annotation aus Projekt Lombok *(siehe „https://projectlombok.org")* hilft, Tipparbeit zu sparen. Was Java dann noch fehlt, sind passende Klassen für unveränderliche Collections. Hier kann zum Beispiel „pCollections" *(siehe „http://pcollections.org")* Abhilfe schaffen. Was asynchrone Programmierung und unveränderliche Daten angeht, vertritt Lagom einen klaren Standpunkt: So muss es programmiert sein. Es werden keine veränderlichen Daten geteilt, es gibt keine blockierenden Aufrufe. Das ist vielleicht für einige noch ungewohnt, aber dieser Trend wird sich sicher auch außerhalb von Lagom durchsetzen.

Mit im Paket: Event Sourcing, CQRS …

Ist das erste „Hello World" geschrieben, kommt als zweiter Schritt meist: Daten in der Datenbank speichern, eine einfach CRUD-Anwendung schreiben; also eine Entity-Klasse definieren und mit einem ORM schnell auf eine Tabelle in der relationalen Datenbank abbilden. Wer das erwartet, unterschätzt die Ambitionen der Lagom-Entwickler deutlich.

Lagom sieht als den Standardfall für Persistenz das „Event Sourcing" vor. Für den langjährigen Java-EE-Entwickler erfordert das etwas Umdenken, hier zeigt das Framework nach Meinung des Autors jedoch seine größte Stärke (und seine Existenzberechtigung).

In einer CRUD-Anwendung speichern wir in einer relationalen Datenbank den aktuellen Zustand unserer Anwendung. Wenn wir nachverfolgen wollen, wie dieser zustande gekommen ist, greifen wir zu zusätzlichen Kniffen wie Audit-Tabellen, die diese Änderungen festhalten. Event Sourcing *(siehe „https://martinfowler.com/eaaDev/EventSourcing.html")* dreht dies einfach um – statt des Zustands sind alle Ereignisse gespeichert, die zu diesem Zustand geführt haben. Vereinfacht kann man es sich vielleicht so vorstellen: Das, was bisher in Datenbank-Tabellen war, ist nun nur noch im Speicher. In der Datenbank wird lediglich eine Art „commit log" geführt, in dem alle Einträge dauerhaft bestehen bleiben. Um weiterhin die Möglichkeit zu haben, Abfragen über den gesamten Datenbestand auszuführen, dient eine separate „Lese-Seite" (Command Query Responsibility Segregation (CQRS), siehe „https://martinfowler.com/bliki/CQRS.html").

Eine ernsthafte Einführung in Event Sourcing und CQRS würde den Rahmen dieses Artikels sprengen, daher ist auf die angeführten Links verwiesen. Es sei aber gesagt: Dieser Ansatz hat enorme Vorteile. Wer einmal eine relationale Datenbank mit Sharding über mehrere Server verteilen musste, wird Event Sourcing zu schätzen wissen. Da das relationale Modell auf der „Schreiben-Seite" wegfällt und nur noch ein einfaches „Event Journal" geführt wird, kann zum Speichern ohne Weiteres eine NoSQL-Datenbank wie Apache Cassandra eingesetzt werden, und der Skalierung sind quasi keine Grenzen gesetzt. Lesen und Schreiben lassen sich zudem getrennt voneinander skalieren, für die „Lese-Seite" können verschiedene Technologien eingesetzt werden, was unzählige Möglichkeiten eröffnet.

Was hat das nun mit Lagom zu tun? Lagom beinhaltet für die dauerhafte Speicherung das Modul „Lagom Persistence", das im Grunde ein Event-Sourcing-Framework ist. Lagom gibt dabei eine relativ starre Struktur vor und kommt mit allen notwendigen Voreinstellungen. Für den Entwickler heißt es: Ich muss nur die Klasse „PersistentEntity" beerben, die Events definieren, die meine Daten verändern dürfen, und die Event Handler implementieren, die diese Änderungen in meinem im Hauptspeicher gehaltenen Objekt vornehmen. Das war's. Lagom kümmert sich um alles andere: Dass die Events gespeichert werden (per Default in Cassandra, was beim Autor auf Anhieb funktionierte) und dass benötigte Objekte aus den bisherigen Events wiederhergestellt werden (und nach längerer Nichtnutzung auch wieder entfernt).

Ohne weiteres Zutun können die Instanzen eines Lagom-Service, der auf verschiedenen Knoten läuft, ein Cluster bilden. Dann kümmert sich das Framework auch darum, die im Speicher gehaltenen Objekte über die Cluster-Knoten zu verteilen (sogenanntes

```
$ mvn archetype:generate \
    -DarchetypeGroupId=com.lightbend.lagom \
    -DarchetypeArtifactId=maven-archetype-lagom-java \
    -DarchetypeVersion=1.2.0
```

Listing 2

```
$ cd hello-world
$ mvn lagom:runAll
```

Listing 3

„Cluster Sharding"). Dabei können zur Laufzeit auch Knoten wegfallen oder hinzugefügt werden; Lagom kümmert sich um die benötigte Neuverteilung. Das Ganze wirkt sehr ausgereift und bis ins Detail durchdacht, sodass sich damit wirklich gut arbeiten lässt.

Frei nach dem Motto „Batterien sind enthalten, aber auswechselbar" kann man in der Standard-Einstellung mit Cassandra und Event Sourcing sofort loslegen – man kann allerdings auch andere Datenbanken verwenden und auch Services ganz ohne Event Sourcing implementieren. Es empfiehlt sich jedoch, Lagom Persistence auf jeden Fall anzusehen und sich vielleicht darauf einzulassen. Wer sich ohnehin schon einmal mit Event Sourcing und CQRS beschäftigen wollte, findet mit Lagom einen einfachen Einstieg.

… und ein Message-Bus

Wie erwähnt, kommt Lagom als Komplettpaket, und wenn man die Vorgabe Apache Cassandra nutzen möchte, kann man mit der Persistenz sofort loslegen. Das Gleiche gilt auch für asynchrones Messaging – Lagom bringt die Unterstützung für Apache Kafka direkt mit.

Asynchrones Messaging wird von anderen Frameworks etwas stiefmütterlich behandelt. Als Normalfall für die Kommunikation zwischen Services werden oft HTTP-Requests angenommen. Dies ist natürlich auch in Lagom möglich. Eine Besonderheit ist hier die Möglichkeit, sogenannte „Service Descriptor" zu definieren. Ein Service Descriptor kompiliert zu einer kleinen Client-Library, sodass ein typisiertes Interface für die Remote-Aufrufe zur Verfügung steht. Dieses kann per Guice dort injiziert werden, wo der Service verwendet werden soll – aus Entwicklersicht ist dieser dann eine einfache Java-Klasse, mit Typ-Check durch den Compiler (und Methoden-Vorschlägen beziehungsweise -Vervollständigung durch die IDE). Ein nettes Gimmick: Dies funktioniert auch mit WebSockets. Ein WebSocket präsentiert sich als „Akka Stream".

HTTP, REST und JSON in allen Ehren, birgt dieser Ansatz grundsätzlich die Gefahr, dass ein Request zu einer HTTP-Aufrufkette führt, die zusammenbricht, wenn ein Service nicht verfügbar ist. In anderen Worten: Ein nicht verfügbarer Service führt zu einer Fehler-Kaskade und setzt alle direkt oder indirekt von ihm abhängigen Services außer Gefecht. Lagom versieht solche Aufrufe schon standardmäßig mit einem „Circuit Breaker". Diese sorgen zwar für mehr Robustheit, lösen jedoch nicht das grundsätzliche (Architektur-)Problem.

Die Alternative zu diesem „Pull"-Ansatz (Daten bei Bedarf abrufen) ist der „Push" von Daten. Änderungen in einem Service führen dabei zu Domain Events, die über einen Message-Bus an andere Services publiziert werden. Die Services, die die entsprechenden Events abonnieren, sind dann in der Lage, eine eigene Datenbasis lokal vorzuhalten und zu aktualisieren. Es gibt natürlich noch weitere Anwendungsfälle für asynchrone Nachrichten zwischen Microservices. Die durch den Event ausgelöste Aktion kann beliebig sein und muss sich nicht auf die Aktualisierung lokaler Daten beschränken. Ähnlich wie bei der Persistenz kann man bei Lagom mit dem asynchronen Messaging sofort loslegen, ein vorkonfigurierter Apacha-Kafka-Message-Bus ist integriert.

Mit Lagom entwickeln

Der „full stack"-Ansatz von Lagom nährt vielleicht die Befürchtung, dass für die Benutzung erst einmal eine eintägige Installations-Sitzung notwendig ist, um die Voraussetzungen (Cassandra, Kafka etc.) zu schaffen. Die Entwickler haben jedoch besonderes Augenmerk darauf gelegt, dies zu vermeiden. Lagom kommt mit einem eigenen Maven-Plug-in (alternativ wird auch „sbt" als Build-Tool unterstützt). Neben allen Lagom-Microservices, die man als Unterprojekte definiert hat, startet dieses auch die komplette benötigte Infrastruktur – einen integrierten Service-Lookup, einen Gateway-Service sowie die Embedded-Varianten von Cassandra und Kafka.

In der Praxis ist das wirklich recht beeindruckend: Ein Lagom-Demoprojekt auschecken, „mvn lagom:runAll" ausführen und das Terminal füllt sich mit „… started"-Meldungen. Nach einer Weile läuft das System, ohne dass eine einzige Konfiguration oder Installation vorgenommen werden musste. Der schnellste Weg zu einer laufenden Lagom-Anwendung führt über einen Maven-Archetype *(siehe Listing 2)*.

In Folge werden ein paar Namenswünsche abgefragt. Das Verzeichnis, das erzeugt wird, trägt die gewählte „artefactId" als Namen, etwa „hello-world" *(siehe Listing 3)* – und schon hat man eine laufende Lagom-Anwendung. Eine besondere IDE-Unterstützung bringt Lagom nicht mit, die gängige Maven-Integration von IDEA oder Eclipse reichen aber völlig aus: Lagom-Projekte lassen sich ohne Weiteres in diese Entwicklungsumgebungen laden. Sehr angenehm beim Programmieren ist das „hot reloading" – eine Änderung im Source-Code eines Lagom-Service führt dazu, dass beim nächsten Request ohne Neustart des Service neu kompiliert wird und die Änderungen sofort sichtbar sind.

Fazit

Obwohl es noch recht jung ist, wirkt Lagom in den meisten Bereichen gut durchdacht. Für zukünftige Versionen erwartet der Autor vor allem, dass der Austausch von Infrastruktur-Komponenten noch einfacher wird. Es ist jetzt schon möglich, etwa Cassandra durch eine relationale Datenbank zu ersetzen. Man könnte sich vorstellen, dass sich in zukünftigen Versionen beispielsweise auch Kafka leicht durch etwas anderes auswechseln lässt. Sicher wird es auch Verbesserungen geben bezüglich Monitoring und der Integration mit Orchestrierungs-Plattformen wie Kubernetes.

Lagom beeindruckt mit seiner Radikalität und Leistungsfähigkeit. Natürlich löst es nicht alle Probleme und auch mit Lagom schreiben sich die Microservices nicht von allein. Was dem Autor vor allem gefällt, ist, dass es inspiriert. Er hat selten ein Framework kennengelernt, das so sehr zum Nachdenken über Software-Architektur anregt wie Lagom.

Wer sich selbst inspirieren lassen möchte: Beispiel-Applikationen und die komplette Dokumentation stehen unter *„http://www.lagomframework.com"*.

Lutz Hühnken
lutz.huehnken@reactivesystems.eu

Lutz Hühnken ist freiberuflicher Solutions Architect und Trainer. Aktuell beschäftigt er sich mit der Entwicklung von Microservices mit Scala, Akka und Lagom. Er tweetet als „@lutzhuehnken" und bloggt unter „https://huehnken.de".

IntelliJ-IDEA-Trickkiste – ein Entwickler packt aus

Yann Cébron, JetBrains GmbH

„Die Leistungsfähigkeit moderner IDEs ist Segen und Fluch zugleich." So lautet der Untertitel des gleichnamigen Vortrags [1], der mit vielen Live-Demos von Features und effizienten Workflows gespickt ist. Solch eine Darstellung ist in einer Zeitschrift naturgemäß nicht möglich – dieser Artikel beleuchtet zumindest die wichtigsten Punkte und Konzepte.

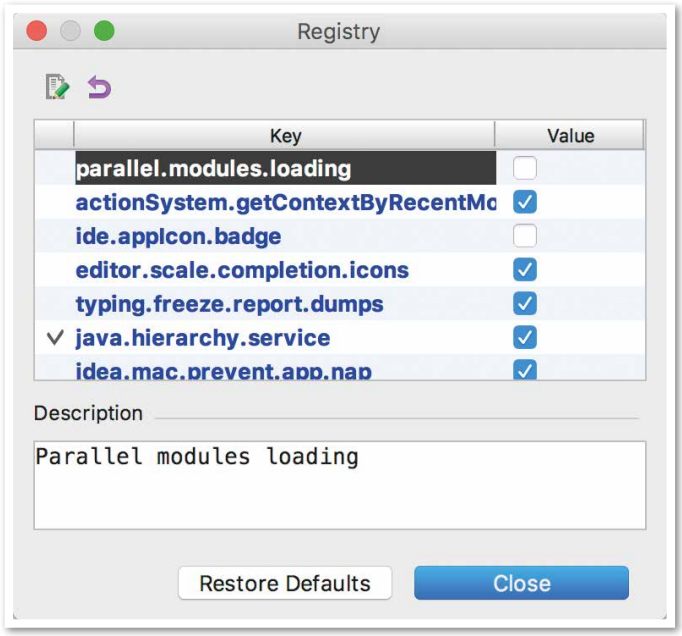


Abbildung 1: Bei der Registry mit versteckten Optionen ist Vorsicht geboten

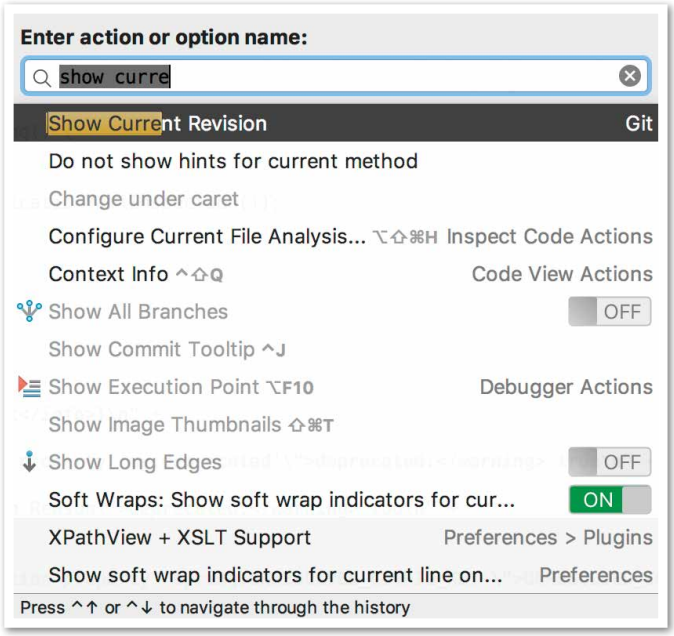


Abbildung 2: „Find Action" durchsucht Befehle und IDE-Einstellungen

Vorweg: Die Tipps gelten nicht nur für IntelliJ IDEA, sondern auch für alle anderen auf der IntelliJ-Plattform basierenden Entwicklungsumgebungen (Android Studio, WebStorm, PyCharm etc.). Menü-Bezeichner sind in *kursiv*, Tastatur-Kürzel in Anführungszeichen gesetzt. Beginnen wir mit dem „Fluch-Thema" Nummer 1: die richtige Konfiguration und das Tuning der Entwicklungsumgebung.

Reine Einstellungssache

Die Vielfalt eingesetzter Hardware und die zunehmende Komplexität von Projekten führen leider oft dazu, dass die Performance der IDE als unzureichend empfunden wird. Meist wird dann fälschlicherweise zur Erhöhung des maximalen Speichers oder Optimierung der anderen vorgegebenen VM-Parameter (keine erfolgversprechende Idee) gegriffen. Zunächst einmal sollte allerdings sichergestellt sein, dass der Import beziehungsweise das manuelle Aufsetzen des Projekts korrekt durchgeführt wurde. Nicht benötigte Plug-ins werden deaktiviert, dies reduziert sowohl Speicherbedarf als auch Startup-Zeit.

Eine Binsenweisheit, aber allzu häufig nicht befolgt: Updates der jeweiligen Entwicklungsumgebung enthalten fast immer deutliche Verbesserungen im Bereich „Performance und Speicherverbrauch". Externe Faktoren wie Anti-Virus-Software, Projekt-Ordner auf Netzwerk-Laufwerk etc. sind regelmäßige Problem-Verursacher. Erfahrungsgemäß reichen 1 – 1,5 GB Speicher-Zuweisung selbst für sehr große Projekte (mehrere Millionen LOC); ansonsten ist genaue Ursachenforschung angebracht.

Grundsätzlich gilt bei der Wahl der Hardware „viel hilft viel". Insbesondere Multicore-Systeme werden von der IntelliJ-Plattform schon seit langer Zeit optimal unterstützt (Stichwort: Parallelisierung von Highlighting im Editor). Übrigens: Bei älteren Rechnern kann schon ein kostengünstiges Upgrade auf eine leistungsfähige SSD Wunder bewirken.

Besonders für Notebook-Nutzer interessant: Mit *File -> Power Save Mode* lässt sich der pure Syntax-Highlighting-Modus im Editor aktivieren; das resultiert in niedrigerer CPU-Belastung und schont damit den Akku.

Werden Referenzen im Code ohne Änderung am Projekt nicht mehr aufgelöst oder hängt das Highlighting im Editor? Dann kann ein Neuaufbau der Indizes helfen (*File -> Invalidate Caches / Restart*). Ein Update oder Deaktivieren von Drittanbieter-Plug-ins kann hier ebenfalls oft Lösung sein.

Über *Registry* ist eine Reihe von versteckten Einstellungen zugänglich *(siehe Abbildung 1)*. Auch wenn diverse Geheimtipps mit angeblichen Verbesserungen im Internet zu finden sind: Alle diese Werte bitte nur nach Anleitung von „offizieller Seite" verändern, da sie potenziell die Funktionalität massiv beeinträchtigen können.

Mein bester Freund, das Keyboard

Der mit Abstand wichtigste Tipp zur Effizienzsteigerung ist, auf die Maus soweit als möglich – am besten komplett – zu verzichten. Zum Abtrainieren empfiehlt es sich, diese nach jeder Benutzung „auf den Rücken gedreht" zurückzulegen. Jeder (instinktive) Griff ist nun mit einem aufwändigen Umdrehen verbunden und erinnert an den Vorsatz, nur noch die Tastatur zu benutzen.

Wer (erstmal) nur einen Keyboard-Shortcut auswendig lernen will, ist mit *Help -> Find Action* („Ctrl/Cmd + Shift + A") gut bedient *(siehe Abbildung 2)*. Damit lassen sich alle Aktionen und IDE-Einstellungen mittels Volltextsuche finden und aufrufen. Praktisch, wenn man den Shortcut einer Aktion

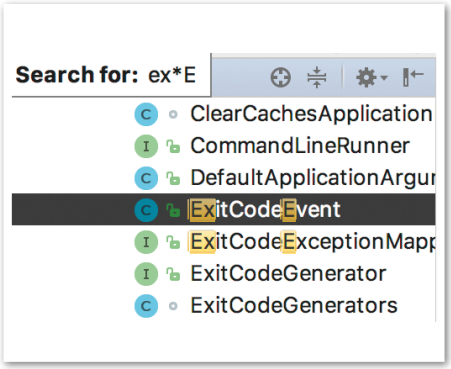


Abbildung 3: Direkte Suche ist überall in der Benutzeroberfläche verfügbar, hier mit CamelCase und Platzhalter-Notation

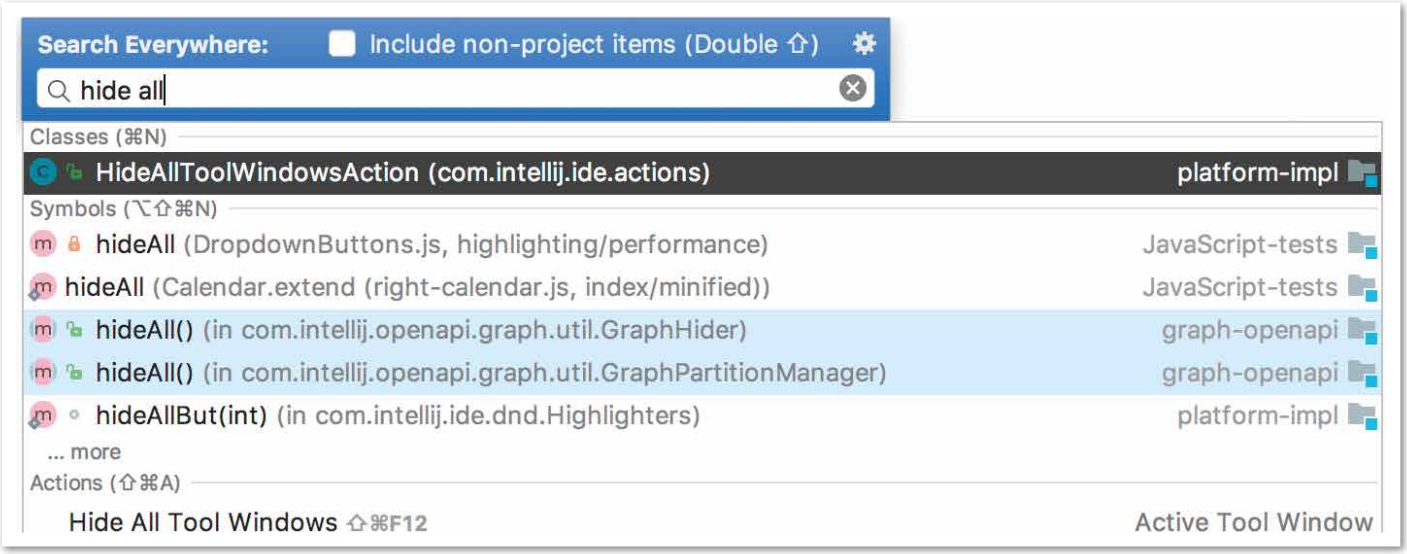


Abbildung 4: „Search Everywhere" durchsucht Dateien, Klassen, Symbole und mehr

vergessen hat oder in komplexen Menüs gerade den Wald vor lauter Bäumen nicht mehr sieht. Wer es lieber physikalisch mag: Die wichtigsten Shortcuts sind als PDF über *Help -> Keymap Reference* zum Ausdrucken verfügbar.

Es gibt eine Reihe von verwandten Aktionen, was sich auch in den Tastatur-Kürzeln widerspiegelt (und damit das Lernen deutlich erleichtert): Durch Hinzufügen von Modifier-Tasten (wie „Shift", „Alt") können Varianten leicht erraten werden (etwa *Navigate -> Goto Class | File | Symbol*). Zweifaches Drücken desselben Shortcut kann optionale Dialoge mit zusätzlichen Optionen aktivieren (etwa *Refactor -> Rename*). Allein diese durchgehende Systematik sollte für Umsteiger von anderen IDEs Anstoß sein, baldmöglichst die „Original"-Keymap zu benutzen – nicht zuletzt weil diese in allen Tutorials als Referenz dient.

Aber nicht nur für das Ausführen von Aktionen gilt, „alles ist mit der Tastatur machbar". So lassen sich sämtliche Elemente der Benutzeroberfläche wie Listen, Trees etc. durch einfaches Lostippen durchsuchen beziehungsweise filtern. (Teil-)Treffer werden hierbei farblich unterlegt; die Verwendung von CamelCase-Notation („ICE" findet „*I*ntelli*J* *C*ommunity *E*dition") und Platzhaltern („*") im Suchbegriff spart Tipparbeit *(siehe Abbildung 3)*. Kontextsensitive Aktionen (wie *View -> Quick Definition*) werden dabei immer für das selektierte Element ausgeführt; dies gilt ebenso für Pop-ups wie die Autovervollständigung.

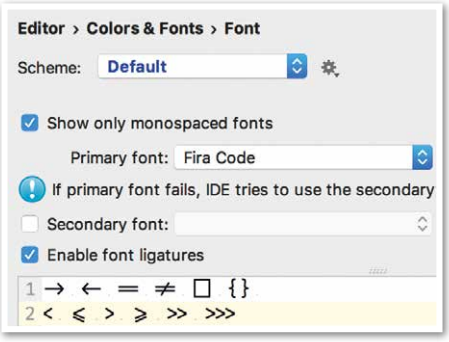


Abbildung 5: Ligaturen optimieren die Darstellung von Zeichenfolgen (hier mit Fira-Code)

Navigation – mehr als nur ein „Goto"

Wer die genannten Varianten von *Navigate -> Goto* ... zum schnellen Auffinden nicht unterscheiden möchte, bekommt mit *Search Everywhere* („2 x Shift") alle Ergebnisse (inklusive der von *Find Action*) auf einen Schlag geliefert *(siehe Abbildung 4)*. Ein Sprung zwischen den Kategorien ist dabei mit „(Shift-) TAB" schnell möglich.

Eine Vielzahl von Icons am linken Rand des Editors stellt zusätzliche Informationen dar und erlaubt die Navigation zu referenzierten Elementen, wie Superklasse oder Implementierungen. Viele Frameworks fügen hier noch weitere Kategorien hinzu (etwa Spring Beans); in den Einstellungen lassen sich diese unter *Editor | General | Gutter Icons* beliebig ein- und ausblenden. Die sprachenübergreifende Aktion *Navigate -> Related Symbol* bietet zudem eine kontextabhängige Navigation zwischen verwandten Elementen an, etwa zwischen Controller und dazugehörigen Views bei Web-Frameworks.

Das aktuell selektierte Element lässt sich mit *Navigate -> Select In* direkt in einem der relevanten Tool-Windows, in der Projekt-Struktur oder sogar im Dateisystem-Browser öffnen. Die mit *View -> Navigation Bar* aktivierbare Breadcrumbs-Leiste oberhalb des Editors erlaubt parallel dazu eine jederzeit sichtbare und schnelle Navigation der Dateistruktur innerhalb des Projekts. Erwartungsgemäß funktionieren hier alle bekannten Tastaturkürzel wie *New -> File* auf einem Ordner. Im *Navigate*-Menü finden sich noch viele weitere nützliche Navigationshilfen wie Bookmarks, Hin- und Herspringen zwischen letzten Cursor-Positionen/Änderungen oder Fehlern im aktuellen Editor etc.

Der Kern des Ganzen: der Editor

Die wohl am häufigsten gebrauchte Funktion zum schnellen Wechsel zwischen den zuletzt geöffneten Dateien ist *Recent Files* („Ctrl/Cmd + E"). Sie erlaubt auch die schnelle Aktivierung der aktiven Tool-Windows. Wie gewohnt filtern Tastatur-Eingaben auch hier direkt die Ergebnisse. Hingegen zeigt *Recently Changes Files* („Ctrl/Cmd + Shift + E", man beachte wieder den verwandten Keyboard-Shortcut) nur die tatsächlich bearbeiteten an. So gerüstet, kann man nun getrost die Editor-Tabs komplett ausschalten und gewinnt dadurch wertvollen Platz auf dem Bildschirm (Einstellung „Placement" auf „None" in *Editor | General | Editor Tabs*).

Apropos Optimierung: Wer sich voll auf den Editor-Inhalt konzentrieren möchte,

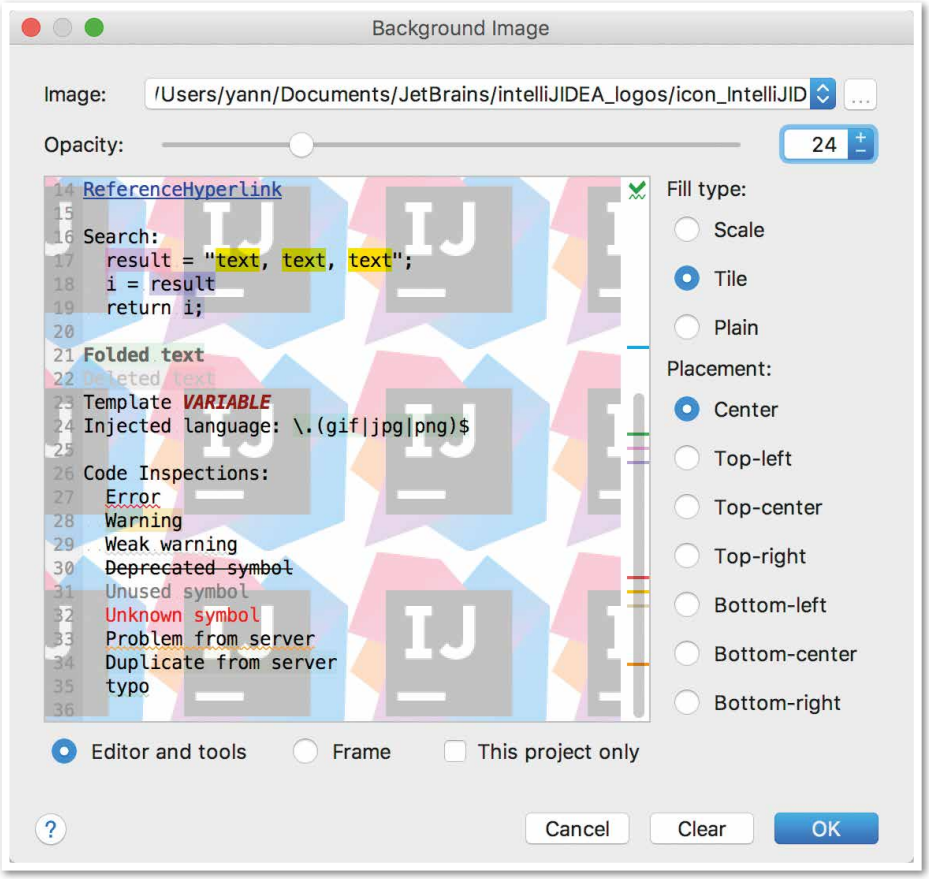


Abbildung 6: Hintergrund-Bilder in Aktion – darf es für den Editor ein wenig bunter sein?

kann alles Drumherum mit *View -> Enter/Exit Distraction Free Mode* mit einem Griff aus- und wieder einblenden. Der *Presentation Mode* ist übrigens nicht nur für den Einsatz am Beamer, sondern auch für Pair-Programming sehr gut geeignet.

Eine der auffälligsten Neuerungen ist die Unterstützung von Schriftarten mit Ligaturen (wie der mitgelieferte Fira-Code). Darunter versteht man die Darstellung mehrerer aufeinander folgender Schriftzeichen zu einer neuen, optimierten Glyphe *(siehe Abbildung 5)*. Für an Monospace gewohnte Programmierer ist das definitiv eine Umstellung – daher muss die verbesserte Darstellung mit „Enable font ligatures" im Reiter *Editor | Colors & Fonts | Font* der Einstellungen explizit aktiviert werden.

Die Selektion von Text sollte man generell mit *Edit -> Extend/Shrink Selection* vornehmen. Dies erweitert/reduziert den markierten Text in semantischen Schritten (abhängig von der aktuellen Programmiersprache) und ist für die meisten Operationen daher deutlich schneller als eine manuelle Auswahl. Ebenso sollte man die syntaktische Korrektur der aktuellen Zeile der IDE überlassen: *Edit -> Complete Current Statement* korrigiert fehlende Klammern und Ähnliches erstaunlich smart. Beim Zusammenfügen von Multiline-Ausdrücken wie String-Konstanten ist *Edit -> Join Lines* eine tatkräftige Hilfe.

Was wäre eine Entwicklungsumgebung ohne automatische Code-Formatierung? Allerdings erschlägt einen oft die Vielzahl der Einstellungen, aber auch hier gibt es einen Kniff. Text markieren, mit „Alt + Enter" das Intention-Pop-up aufrufen – und *Adjust code style settings* zeigt nur noch die für die aktuelle Selektion relevanten Optionen an.

Zum Schluss noch ein zunächst exotisch anmutendes Feature: Mit *Set Background Image* kann ein beliebiges Bild als IDE- oder Editor-Hintergrund eingestellt werden *(siehe Abbildung 6)*. Dieses lässt sich zusätzlich auf das aktuelle Projekt beschränken. So kann man beispielsweise beim Wechsel vom Hauptprojekt zur im Release-Branch ausgecheckten Kopie in dieser dann ein Warnbild zur visuellen Markierung nutzen.

War es das?

Die Liste der Tipps und Tricks ist natürlich noch viel länger – viele weitere finden sich im IntelliJ-IDEA-Blog *[3]* nebst Vorstellung von Neuigkeiten zur aktuellen Entwicklungsversion. Demos und Screencasts zu speziellen Themen oder Features auf YouTube *[4]* sind insbesondere für Ein- und Umsteiger gut geeignet.

Bitte nicht verzagen – der Autor lernt selbst nach fünfzehn Jahren intensiver Nutzung noch regelmäßig neue Kniffe.

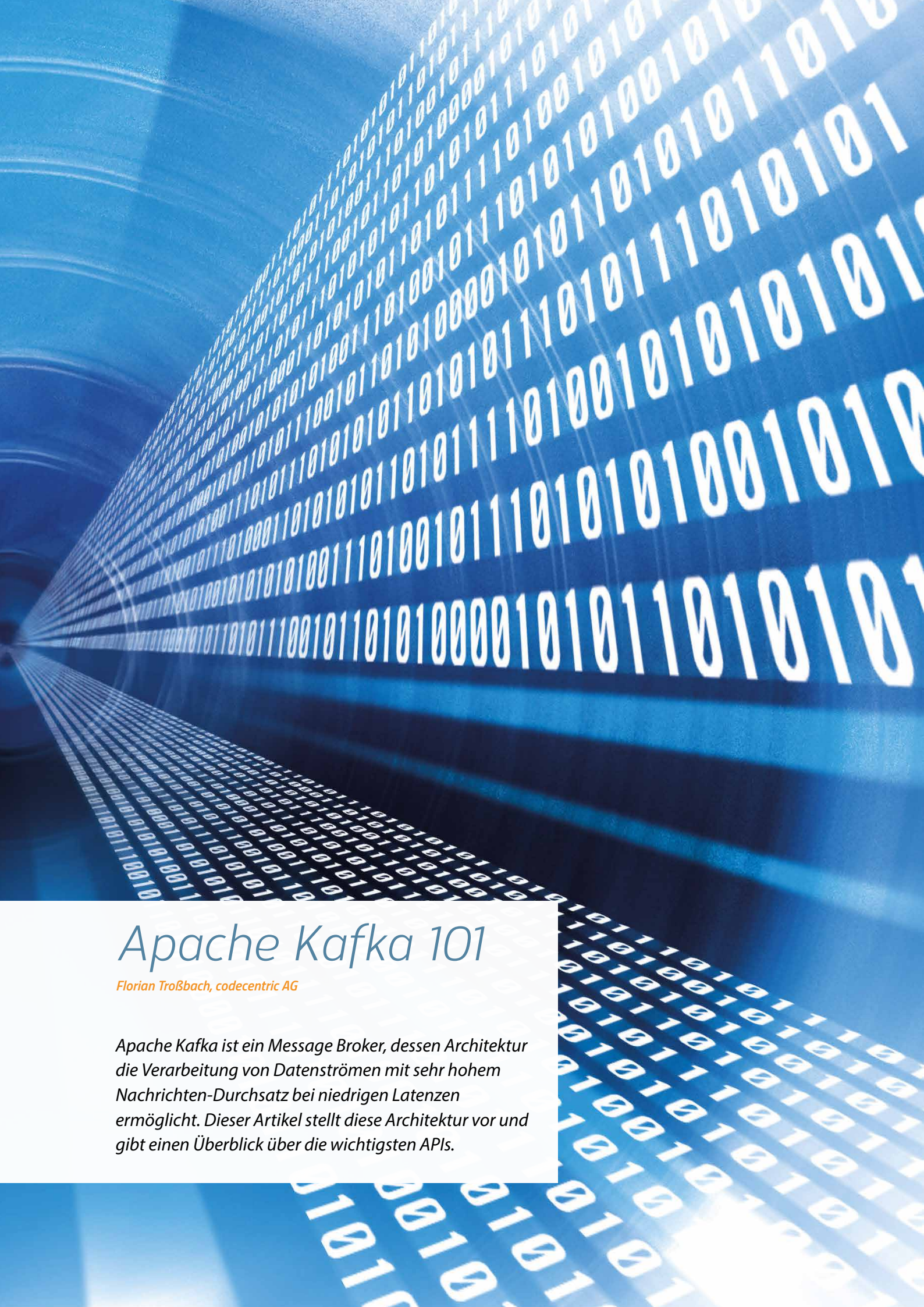
PS: Im Mai gibt es den Vortrag „live" bei der JUG Darmstadt *[2]*.

Links

[1] Folien zum Vortrag: http://www.slideshare.net/yanncebron/intellij-idea-trickkiste
[2] Vortrag bei JUG Darmstadt: http://www.jug-da.de/2017/05/IntelliJ-Tricks/
[3] IntelliJ-IDEA-Blog: https://blog.jetbrains.com/idea
[4] IntelliJ-IDEA-YouTube-Kanal: https://www.youtube.com/c/intellijidea

Yann Cébron
yann.cebron@gmail.com

Yann Cébron (@yanncebron) ist nicht nur begeisterter User von IntelliJ IDEA, er entwickelte im Laufe der Zeit auch eine Reihe von Plug-ins dafür. Seit fünf Jahren ist er als Entwickler für den Spring-Framework-Support sowie das Plug-in SDK verantwortlich. Zu beiden Themen hält er Vorträge bei JUGs und Konferenzen im In- und Ausland. In der JUG Hannover und beim Java Forum Nord ist er von Anfang an als Mitorganisator aktiv.



Apache Kafka 101

Florian Troßbach, codecentric AG

Apache Kafka ist ein Message Broker, dessen Architektur die Verarbeitung von Datenströmen mit sehr hohem Nachrichten-Durchsatz bei niedrigen Latenzen ermöglicht. Dieser Artikel stellt diese Architektur vor und gibt einen Überblick über die wichtigsten APIs.

Rasant steigende Datenmengen und der Anspruch, diese nach der jeweils eigenen Definition von „Echtzeit" zu verarbeiten, stellen in vielen Unternehmen neue Anforderungen an die Fähigkeiten von Messaging-Systemen. Dies hat bei LinkedIn zur Entwicklung eines Message Brokers geführt, der ambitionierte Eigenschaften aufweisen sollte: lineare horizontale Skalierbarkeit, Hochverfügbarkeit sowie niedrige Schreib- und Lese-Latenzen. Das Projekt war erfolgreich und das Ergebnis wurde im Jahr 2011 an die Apache Software Foundation (ASF) übergeben, wo es den Inkubator ein Jahr später erfolgreich als Top-Level-Projekt „Apache Kafka" verließ.

Dieser Artikel stellt zunächst vor, wie Kafka Nachrichten intern verwaltet. Dann wird erläutert, welche Konzepte hinter der Produktion und Konsumierung von Nachrichten stehen und wie Kafka Hochverfügbarkeit und Datenkonsistenz balanciert. Zum Abschluss werden die Kern-APIs anhand von Beispielen vorgestellt.

Anatomie eines Kafka-Topics

An der Oberfläche sieht Kafka aus wie ein gewöhnlicher Publish-Subscribe Message Broker. Kafka wird im Regelfall in einem Cluster aus n Brokern betrieben, die sich über ein Quorum von Apache Zookeeper koordinieren – der Betrieb eines Zookeeper-Clusters ist zwingende Vorbedingung für den Einsatz von Kafka.

In einem Kafka-Cluster gibt es eine beliebige Zahl an Topics, „Producer" veröffentlichen Nachrichten, „Consumer" lesen sie. Ein Topic besteht aus einer Menge von Partitionen. Diese werden auf die Broker verteilt, sodass mehrere Broker für ein Topic zuständig sind. Hiermit wird eine Lastverteilung sowohl für schreibende als auch für lesende Operationen erreicht. Da die Anzahl der Partitionen in der Regel die Anzahl der Broker übersteigt, ist ein Broker für n Partitionen desselben Topics zuständig *(siehe Abbildung 1)*.

Auf Partitionsebene werden die Nachrichten in Form eines fortlaufenden Commit Logs vorgehalten. Jede neue Nachricht zu einer Partition wird hinten an das Log angehängt und durch eine pro Partition eindeutige Nummer identifiziert. Die erste Nachricht, die in eine Partition geschrieben wird, erhält das sogenannte „Offset" mit dem Index 0. Die zweite Nachricht erhält dann das Offset 1. In einer anderen Partition desselben Topics erhält die erste Nachricht – im Normalfall mit völlig anderem Inhalt – ebenfalls das Offset 0.

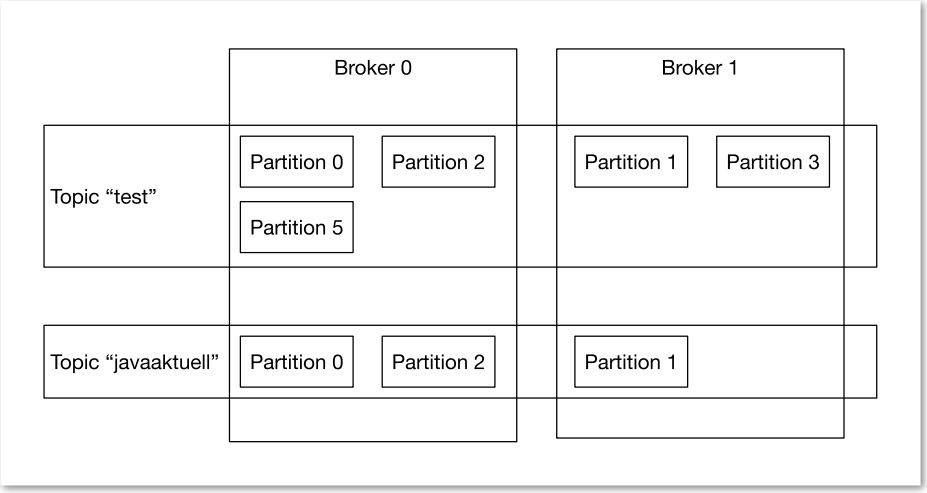


Abbildung 1: Partitionen von Topics werden auf Broker verteilt

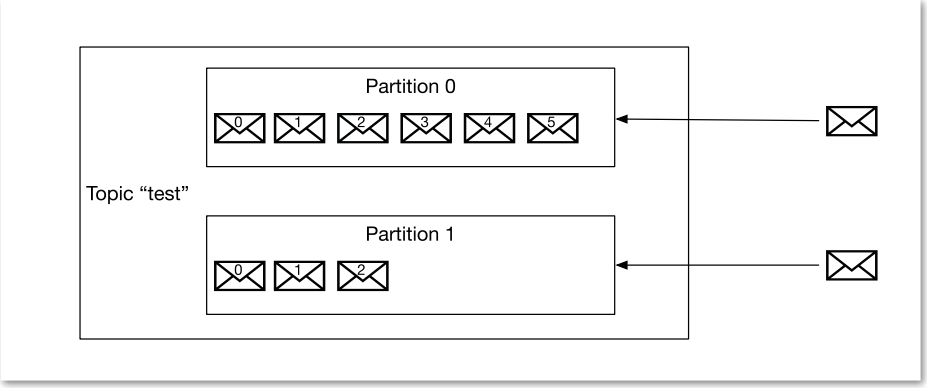


Abbildung 2: Nachrichten werden hinten an das Log angehängt. Jede Nachricht erhält ein pro Partition eindeutiges Offset.

```
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
00000000000015339062.index
00000000000015339062.log
00000000000015339062.timeindex
```

Listing 1

Eine Nachricht in Kafka kann über das Tripel (Topic, Partition, Offset) eindeutig identifiziert werden. *Abbildung 2* stellt dies grafisch dar.

Persistenz

Die Details der physischen Datenspeicherung sind sehr interessant. Kafka wurde für Commodity-Hardware konzipiert und setzt auf normale Festplatten oder SSDs. Diese Speichermedien eignen sich nur bedingt für performante beliebige Lese- und Schreib-Operationen; durch die Struktur einer Partition als Commit Log sind diese jedoch gar nicht nötig. Neue Nachrichten werden immer nur hinten an das Log angehängt und sind von diesem Moment an unveränderlich. Gelesen werden die Daten im Normalfall ebenfalls pro Partition linear ab einem definierten Offset.

Page Cache und Zero-copy

Lineares Lesen und Schreiben auf Standard-Festplatten ist schön und gut, aber wie sieht es mit parallelen Lese- und Schreib-Zugriffen aus? Hierbei enthüllt sich ein cleveres Detail der Architektur. Die JVM eines Kafka-Brokers selbst benötigt vergleichsweise wenig Hauptspeicher. Empfohlen werden fünf bis sechs GB auf Maschinen mit 32 bis 64 GB RAM. Der Rest des Hauptspeichers wird vom Betriebssystem für das Puffern des Dateisystems, den sogenannten „Page Cache", genutzt. Kürzlich gelesene oder geschriebene Daten werden vom Betriebssystem so

lange dort vorgehalten, bis ein anderer Prozess diesen Speicher benötigt oder der Page Cache selbst die Daten überschreibt. Da es in einem eingeschwungenen Kafka-System in der Regel so ist, dass die Consumer nur die neuesten Nachrichten lesen, wird auf ältere Daten selten zugegriffen.

In vielen Kafka-Clustern werden mehr als 90 Prozent der Anfragen aus dem Page Cache beantwortet – ein großer Geschwindigkeitsschub. Eine weitere klug genutzte Betriebssystem-Funktion ist „Zero-copy", mit der Daten direkt aus dem Page Cache auf Netzwerk-Sockets geschrieben werden können. Dies erklärt auch den relativ geringen Hauptspeicherbedarf des Brokers selbst. Leider gilt dies nicht, wenn die von Kafka angebotene Transport-Verschlüsselung genutzt wird, da die Daten hierbei innerhalb der JVM verschlüsselt werden müssen.

Die Abhängigkeit von der Verfügbarkeit des Hauptspeichers für den Page Cache für eine gute Performance führt zu der Empfehlung, einen Kafka-Host nicht zusätzlich mit anderen Anwendungen zu bestücken.

Auf dem Dateisystem

Die Ablage der Daten im Dateisystem wurde natürlich auch optimiert. Im Speicherverzeichnis für die Commit Logs wird für jede Partition, für die ein Broker zuständig ist, ein Unterverzeichnis angelegt. Ist ein Broker für die Partitionen 0 und 2 eines Topics namens „topic" zuständig, heißen diese Verzeichnisse „topic-0" und „topic-2". *Listing 1* zeigt einen Blick in eines dieser Unterverzeichnisse und offenbart den Aufbau des Logs auf Partitionsebene.

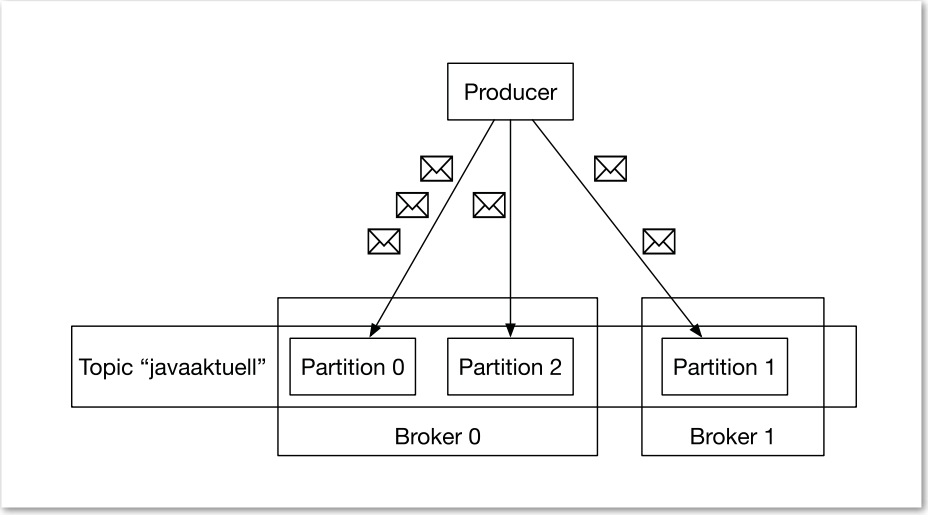


Abbildung 3: Der Producer schickt Nachrichten direkt an den verantwortlichen Broker

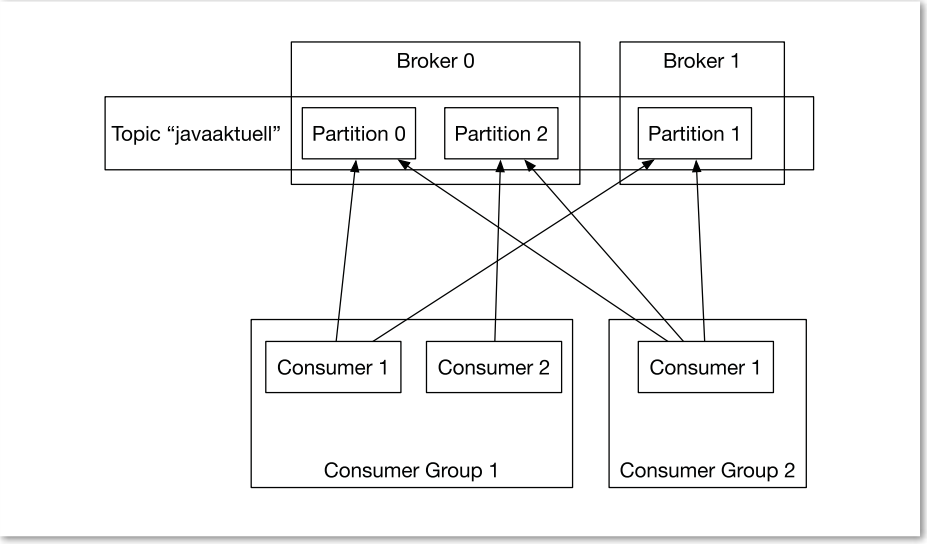


Abbildung 4: Consumer 1 von Consumer Group 1 liest die Partitionen 0 und 1, Consumer 2 die Partition 2. Der einzelne Consumer von Consumer Group 2 liest alle Partitionen.

Das Listing zeigt sechs Dateien mit unhandlichen Namen. Was hat es damit auf sich? Das Commit Log ist zwar logisch gesehen eine Datei, aber um effizienteren Zugriff gewährleisten zu können, gibt es ein paar Kniffe, zum einen den Rotations-Mechanismus. Es gibt pro Partition immer genau eine Datei, an die aktuell Nachrichten angehängt werden. Hat diese Datei eine konfigurierbare Größe erreicht oder ist eine definierte Zeitspanne abgelaufen, wird die Datei geschlossen und eine neue angelegt. Der Name leitet sich aus dem ersten enthaltenen Offset ab. Im Beispiel kann der Broker bei der Suche nach Offset 20000000 also schon am Namen ableiten, dass die Nachricht in der Datei „00000000000015339062.log" zu finden ist. Um schneller Offsets innerhalb einer Log-Datei finden zu können, gibt es für jedes Log-Segment eine deutlich kleinere „.index"-Datei. In dieser sind Offsets und Byte-Positionen festgehalten. Seit Version 0.10.1 gibt es zusätzlich einen zeitbasierten Index, über den man Offsets und Zeitpunkte korrelieren kann.

Produktion von Nachrichten

Mittlerweile wissen wir, wie Kafka-Nachrichten intern gespeichert sind. Im Folgenden sprechen wir darüber, wie Nachrichten in das System gelangen. Eine Kafka-Nachricht ist als Schlüssel-Wert-Paar definiert. Sowohl im Schlüssel als auch im Wert stehen beliebige Byte-Arrays. Kafka macht keinerlei Vorschriften über Formate oder Inhalte. Jeder Nachrichtenproduzent ist selbst dafür verantwortlich, welches Format er wählt. Seine optimale Performance erreicht Kafka bei Nachrichten im niedrigen Kilobyte-Bereich.

Der Schlüssel selbst impliziert keine Eindeutigkeit. Producer können beliebig viele Nachrichten zu einem Schlüssel verschicken. Wofür wird er dann benötigt? Die Antwort ist relativ einfach: Der Producer allein entscheidet, an welche Partition eines Topics er eine Nachricht schickt *(siehe Abbildung 3)*.

In der Standard-Konfiguration des Java-API wird ein Hash auf dem Schlüssel berechnet und darüber die Partition definiert. Somit ist sichergestellt, dass Nachrichten mit demselben Schlüssel auch an dieselbe Partition eines Topics geschickt werden. Die Verwendung eines Schlüssels ist in vielen Anwendungsfällen sinnvoll. Ein Beispiel ist die Erfassung von Click-Events auf Webseiten. Um Statistiken für alle Clicks auf Seiten derselben Domäne leichter aggregieren zu können, ist es hilfreich, wenn alle Nachrichten dieser Domäne in derselben Kafka-Par-

tition liegen. Der Nachteil ist natürlich, dass sich hierdurch Hot-Spots bilden können – eine gleichmäßige Verteilung von Nachrichten auf Partitionen ist nicht inhärent gewährleistet. Wird kein Schlüssel vergeben, greift ein Round-Robin-Verfahren.

Ein weiterer wichtiger Aspekt der Nachrichten-Produktion ist die Quittierung von Nachrichten. Der Producer entscheidet, welches Quittierungsniveau er verlangt. Hier gibt es drei Möglichkeiten: Der Producer kann deklarieren, dass er keine Quittungen benötigt. In diesem Fall gibt es keine Garantien. Häufiger ist der Fall, dass der für die Partition verantwortliche Broker die Nachricht quittieren muss. Die dritte Möglichkeit umfasst eine Quittung durch den verantwortlichen Broker und – sofern konfiguriert – durch die Replikas der Partition. Das Thema „Replikation" wird im späteren Verlauf des Artikels noch näher beleuchtet.

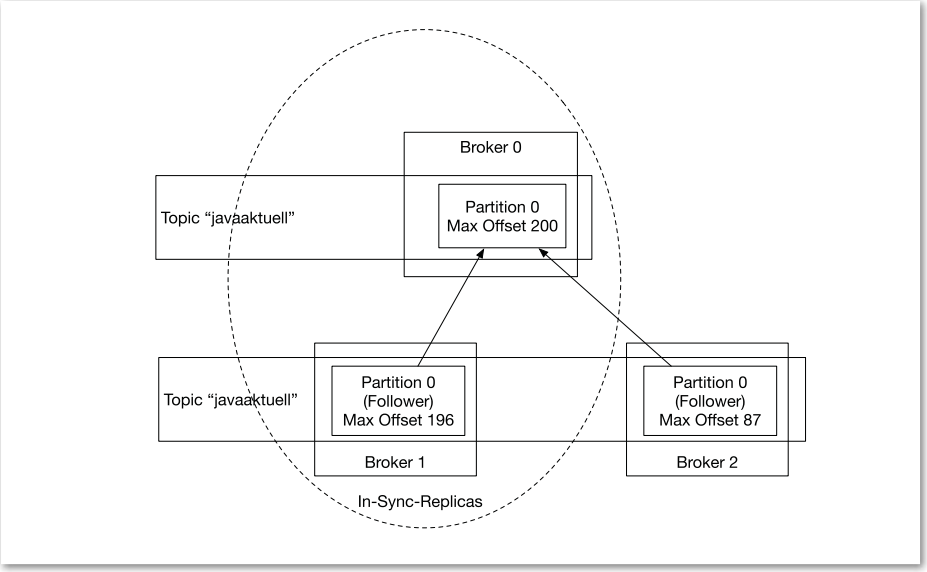


Abbildung 5: Broker 0 ist Leader von Partition 0 des Topics „javaaktuell". Die Replikation durch Broker 1 ist sehr nahe am Leader, daher ist Broker 1 eine ISR. Broker 2 hängt zurück und kann somit nicht zum Leader gewählt werden.

Nachrichten-Verarbeitung

Beim Abholen der Nachrichten durch Consumer offenbart sich der Hauptvorteil der Partitionierung der Topics. Consumer werden in sogenannten „Consumer-Groups" zusammengefasst. Das klingt komplizierter, als es ist – die Consumer-Group ist ein Konfigurationsparameter der Consumer und kann eine beliebige Zeichenkette enthalten. Consumer können Topics abonnieren. Gibt es nur einen Consumer in einer Consumer-Group, weist das Cluster alle Partitionen dieses Topics diesem Consumer zu. Gibt es allerdings zwei oder mehr Consumer in einer Consumer-Group, werden die Partitionen des abonnierten Topics auf diese Consumer aufgeteilt. *Abbildung 4* zeigt dies anhand eines Beispiels.

Die Verteilung der Partitionen erfolgt automatisch und reagiert dynamisch. Kommt ein neuer Consumer zu einer Gruppe hinzu, werden Partitionen neu verteilt (vorausgesetzt, es gibt noch Consumer, die auf mehr als einer Partition arbeiten). Wird ein Consumer geplant gestoppt, werden dessen Partitionen direkt umverteilt. Bei einem ungeplanten Stopp eines Consumers erfolgt die Umverteilung nach Ablauf eines Timeouts.

Jede Partition ist zu jedem Zeitpunkt genau einem Consumer innerhalb einer Consumer-Group zugeordnet. Im Kontext des Ausfalls eines Consumers sowie der Umverteilung von Partitionen sind die Offsets von besonderer Bedeutung. Woher weiß ein Consumer, der eine Partition neu zugewiesen bekommen hat, ab welchem Offset er weiterlesen muss? Hierzu sieht Kafka das Konzept des „Offset-Commit" vor.

Consumer halten ihren Lesefortschritt in dedizierten Kafka-Topics fest. Ein Consumer kann somit abfragen, welche Nachrichten eines Topics bereits gelesen wurden. Im Standardfall erfolgt dies automatisch und periodisch. Entwickler können jedoch auch manuell Commits durchführen. Dies passiert in der Praxis besonders bei Anwendungsfällen, die „Exactly Once"-Verarbeitung erfordern. Diese Fähigkeit ist zwar in Arbeit, wird aber zum aktuellen Zeitpunkt noch nicht standardmäßig unterstützt.

Während der Entwicklung von Kafka stellten die Entwickler sich die Frage, ob Consumer aktiv die Daten bei den Brokern abfragen („Pull"-Ansatz) oder ob die Broker die Daten an die registrierten Consumer schicken sollten („Push"-Ansatz). Die Entwickler haben sich aus zwei Hauptgründen für den „Pull"-Ansatz entschieden: Zum einen reduziert er die Komplexität der Broker, zum anderen wird dadurch, dass die Consumer aktiv Nachrichten abholen, implizit ihre Überlastung vermieden – ein Consumer holt nur dann Daten ab, wenn er sie auch verarbeiten kann.

Replikation und Hochverfügbarkeit

Um Ausfallsicherheit und Hochverfügbarkeit zu gewährleisten, bietet Kafka Replikation auf Topic-Ebene an. Jeder Partition eines replizierten Kafka-Topics sind ein Broker als „Leader" und eine definierbare Zahl „Follower" zugeordnet. Der Leader einer Partition ist der einzige Broker, der für diese Partition Nachrichten entgegennehmen und an Consumer verteilen darf. Fällt ein Leader aus, findet eine Neuwahl statt. Mögliche neue Leader sind diejenigen Follower, die zur Menge der sogenannten „In-Sync-Replicas" (ISR) gehören. Dies sind Follower, die auf einem konfigurierbar aktuellen Stand sind *(siehe Abbildung 5)*.

Dieser Mechanismus hat einige Implikationen. Zum einen kann der Fall eintreten, dass keiner der Follower zu den ISRs einer Partition gehört. In diesem Fall erhalten Producer, die das maximale Quittungslevel anfordern, bereits dann eine positive Quittung, sobald der Leader die Nachricht verarbeitet hat. Zum anderen kann es durch die inhärente Verzögerung zwischen Leader und Followern zu Nachrichtenverlusten kommen.

Kafka legt allerdings Wert auf Konsistenz. Eine Nachricht wird erst dann an Consumer ausgeliefert, wenn sie von allen Followern (nicht nur ISRs) quittiert wurde. Damit kann zwar kein Datenverlust vermieden werden, es entsteht jedoch auch keine Situation, bei der manche Consumer einer Partition eine Nachricht erhalten und andere nicht. Ebenso besteht die Möglichkeit, durch Konfigurationsoptionen Datenkonsistenz auf Kosten der Hochverfügbarkeit von Partitionen sicherzustellen – in dem Fall, dass eine Partition nicht hinreichend repliziert ist, würde die Quittierung einer neuen Nachricht dann verweigert.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
for(int i = 0; i < 100; i++)
    producer.send(new ProducerRecord<String, String>("test", Integer.toString(i), Integer.toString(i)));

producer.close();
```

Listing 2

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", “javaaktuell-consumer-group“);
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("test"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());
}
```

Listing 3

Client-APIs

Java-basierte Consumer- und Producer-APIs sind Teil des offiziellen Apache-Projekts. Diese APIs sind stabil und in ihrem Aufbau konsistent. *Listing 2* zeigt einen einfachen Producer, der die Zahlen von 1 bis 100 in ein Topic schreibt. Zu Beginn wird ein Konfigurationsobjekt erzeugt. Zunächst muss man dem Producer natürlich mitteilen, wie das Kafka-Cluster erreichbar ist. Es ist mindestens ein Broker anzugeben, dieser gibt die Adressen der weiteren Broker dann an den Producer weiter. Die nächsten zwei Einstellungen sind die vollqualifizierten Klassen-Namen von Serialisierern. Diese übersetzen die gesendeten Schlüssel und Werte in Byte-Arrays. Das Consumer-API ist sehr ähnlich aufgebaut, wie *Listing 3* zeigt.

Auch in diesem Beispiel muss wieder mindestens ein Broker als Erstkontakt genannt sein. Analog zum Producer werden die in Kafka gespeicherten Byte-Arrays nun wieder in Strings deserialisiert. Die Consumer-Group wird als „group.id” definiert. Nachdem der Consumer initialisiert ist und ein Topic abonniert hat, muss er durch den Pull-Ansatz regelmäßig bei den Leadern der ihm zugeordneten Partitionen nachfragen, ob es neue Nachrichten gibt. Diese werden in diesem simplen Beispiel auf die Standard-Ausgabe geschrieben. Neben den Java-APIs werden von Confluent und der Community noch weitere Sprachen unterstützt, zum Beispiel Python und Go. Zwei weitere Java-basierte APIs könnten eigene Artikel füllen, sollen aber zumindest nicht unerwähnt bleiben.

„Kafka Connect” ist eine Laufzeitumgebung und ein API, mit dem die Konnektoren für die Integration von Kafka und anderen Datenquellen entwickelt und hochverfügbar betrieben werden können. Beispiele sind Konnektoren für Apache Cassandra oder JDBC-Datenbanken.

„Kafka Streams” ist eine Neuerung von Kafka 0.10. Hiermit tritt Kafka in gewissen Anwendungsfällen in Konkurrenz zu Streaming-Frameworks wie Apache Spark oder Apache Flink. Kern des Streams-API sind zum einen Ströme zwischen zwei Kafka-Topics, auf denen man die mittlerweile bekannten Funktionen wie „map“, „flatMap“ oder „filter” aufrufen kann. Zusätzlich zu diesen zustandslosen Operationen können auch zustandsbehaftete Funktionen wie „count“ oder „groupBy“ ausgeführt werden.

Fazit

Hiermit endet die kurze Einführung in Apache Kafka. In diesem Artikel wurde vorgestellt, wie die Architektur von Kafka hohe Durchsätze, niedrige Latenzen und horizontale Skalierbarkeit ermöglich. Ebenso wurden die Abwägungen zwischen Verfügbarkeit und Konsistenz beschrieben, die Architekten und Entwickler beim Einsatz von Kafka bedenken müssen. Kafka ist sehr stabil und wird bereits von vielen Unternehmen produktiv eingesetzt. Der Autor legt jedem, der sich mit Stream-Processing beschäftigt, ans Herz, sich bei der Wahl von Messaging-Systemen Kafka genauer anzuschauen.

Florian Troßbach
florian.trossbach@codecentric.de

Florian Troßbach arbeitet als IT-Consultant bei der codecentric AG in Karlsruhe. Seine Wurzeln hat er in der klassischen Java-Enterprise-Entwicklung; mittlerweile gilt sein Augenmerk den Themen "Fast Data" und "Stream Processing".

Hallo, ich bin Redis

Mark Paluch, Pivotal

Redis ist eine vielseitige und leichtgewichtige NoSQL-Datenbank, die uns in etlichen Integrationen begegnet. Ein genauer Blick zeigt, dass es sich um viel mehr als nur einen Key-Value-Store handelt. Mit seinen grundlegenden Datenstrukturen bietet der In-Memory-Key-Value-Store vielseitige Einsatzmöglichkeiten in hochverfügbaren Konfigurationen.

Redis ist die Abkürzung für „Remote Dictionary Server" und wird seit dem Jahr 2006 von Salvatore Sanfilippo („@antirez") als Open-Source-Projekt in ANSI C entwickelt. Der Key-Value-Store folgt einem ganz eigenen Pfad, indem dieser mit atomaren Befehlen auf Datenstrukturen operiert. Die Datentypen orientieren sich an fundamentalen Datenstrukturen wie Listen, Sets und Hashes.

Eine weitere Besonderheit ist der Single-Threaded-Betrieb, der eine Reihe interessanter Eigenschaften mit sich bringt. Obwohl User-Operationen nur auf einem Thread ausgeführt werden, erreichen die meisten Operationen Latenzen unterhalb einer Millisekunde. Je nach Datenmenge und Art der Operation kann ein einzelner Server 100.000 bis 1.000.000 Operationen pro Sekunde ausführen. Die Gründe dafür sind einerseits die In-Memory-Natur und andererseits die effiziente Implementierung.

Installation

Redis ist als Quellcode-Paket auf dem GitHub-Repository „antirez/redis" veröffentlicht *[1]*. Daher ist es nicht verwunderlich, dass nur wenige vorkompilierte Binärpakete erhältlich sind; meist kommen die Binär-Distributionen über Linux-Pakete. Eine Ausnahme zu dieser Regel gibt es: Microsoft OpenTech pflegt einen Windows-Build. Nach der Installation *(siehe Abbildung 1)* lässt sich Redis direkt von der Konsole aus starten *(siehe Abbildung 2)*.

Der Server ist gestartet und nimmt am Port 6379 Verbindungen entgegen. Der Prozess lässt sich auch mit einer Konfigurationsdatei starten, um Konfigurationsparameter einzustellen. Seit Version 3.2.0 aktiviert Redis den Protected Mode, sofern der Server mit Standard-Parametern gestartet wird (kein Passwort gesetzt, Port-Binding an allen Netzwerk-Karten). Der Protected Mode führt dazu, dass Befehle aller Verbindungen außer vom Loopback-Interface („localhost") abgelehnt werden. Remote-Verbindungen werden mit einem Fehler quittiert, bis entweder ein Passwort oder das Port-Binding konfiguriert sind.

Je nach Installationsmethode ist „redis-cli" gleich mitgeliefert *(siehe Abbildung 3)*. Es

```
# Linux
$ apt-get install redis-server

# macOS via Homebrew
$ brew install redis

# Source-Installation
$ wget http://download.redis.io/releases/
redis-3.2.4.tar.gz | tar xzf -
$ cd redis-3.2.4
$ make
$ src/redis-server

# Docker
$ docker run --name redis -d redis:alpine
```

Abbildung 1: Redis installieren

```
$ redis-server
85072:C 01 Jan 17:33:16.449 # Warning: no config file specified, using the default config. In
order to specify a config file use redis-server /path/to/redis.conf
85072:M 01 Jan 17:33:16.451 * Increased maximum number of open files to 10032 (it was original
ly set to 8192).
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 3.2.4 (070d0471/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 85072
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

85072:M 01 Jan 17:33:16.452 # Server started, Redis version 3.2.4
85072:M 01 Jan 17:33:16.452 * The server is now ready to accept connections on port 6379
```

Abbildung 2: Redis starten

handelt sich um einen Kommandozeilen-Client, der Redis-Befehle ausführt. Daten lassen sich mit einfachen Befehlen ablegen und abfragen:

- *SET*
  Key setzen
- *GET*
  Key abfragen
- *TYPE*
  Datentyp für einen Key ausgeben
- *EXPIRE*
  Time-to-Live festlegen
- *TTL*
  Verbleibende TTL ausgeben
- *EXISTS*
  Prüfen, ob ein Key existiert
- *KEYS*
  Keys (mit Filterangabe) auflisten
  (nicht bei großen Keyspaces einsetzen)

Die Treiber

Treiber werden, genau wie Redis selbst, durch die Community entwickelt. Es sind rund 146 gelistete Treiber *[2]* für 48 Programmiersprachen (Stand Ende 2016) verfügbar. Die Programmiersprachen C, C#, JavaScript (via node.js), Java und PHP genießen das reichhaltigste Angebot an Treibern mit sehr aktuellen Features. Dies ist jedoch ein entscheidender Punkt für die Client-Auswahl, denn Client ist nicht gleich Client.

Ein Großteil der Treiber wurde durch Privatpersonen entwickelt, jedoch ist mehr als die Hälfte der Projekte nicht mehr aktiv. Oftmals reichte den Autoren ein gewisses Maß an Funktionalität und so ist nur ein Ausschnitt an Features implementiert. Neuere Features sind in vielen Treibern nicht enthalten. Einige Treiber sind effizienter als andere implementiert und eine Treiber-Community ist stärker als eine andere. Treiber für Java bieten mitunter die beste Feature-Abdeckung; es gibt vier aktive Treiber-Projekte:

```
<dependency>
  <groupId>biz.paluch.redis</groupId>
  <artifactId>lettuce</artifactId>
  <version>4.3.0.Final</version>
</dependency>
```

Listing 1

```
RedisClient redisClient = RedisClient.create(„redis://localhost:6379“);
StatefulRedisConnection<String, String> connection = redisClient.connect();
// SET 42 107.6
connection.sync().set(„42“, „107.6“);
// GET 42
String response = connection.sync().get(„42“);
```

Listing 2

- Jedipus *[3]*
- Jedis *[4]*
- Lettuce *[5]*
- Redisson *[6]*

In diesem Artikel kommt der Lettuce-Treiber zum Einsatz. Er basiert auf Netty und ist ein asynchroner sowie Non-Blocking Client *(siehe Listing 1)*.

Lettuce setzt auf Java 8 und unterstützt Stand-alone, Master/Slave, Sentinel, Redis-Cluster und TLS. Er bietet synchrone, asynchrone („Future“) und reaktive APIs („RxJava Observable“), sodass Client-Applikationen vom Non-Blocking-Verhalten direkt profitieren *(siehe Listing 2)*.

Der Redis-Betrieb

Redis kann in unterschiedlichen Konfigurationen betrieben werden. Der Betriebsmodus wirkt sich erheblich auf die Wahl des Clients und die zur Verfügung stehenden Befehle aus. Nicht jeder Client unterstützt

```
$ redis-cli
127.0.0.1:6379> set fookey value
OK
127.0.0.1:6379> keys *
1) "fookey"
127.0.0.1:6379> type fookey
string
127.0.0.1:6379> get fookey
"value"
127.0.0.1:6379> expire fookey 10
(integer) 1
127.0.0.1:6379> ttl fookey
(integer) 7
127.0.0.1:6379> ttl fookey
(integer) 6
127.0.0.1:6379> exists fookey
(integer) 0
127.0.0.1:6379> keys *
(empty list or set)
```

Abbildung 3: Redis-Kommandozeile via „redis-cli“

jede Konfiguration. Wird der Server gestartet, so läuft Redis im Stand-alone-Betrieb und ein Server verwaltet alle Daten auf einem Knoten. Im Stand-alone-Betrieb lassen sich zusätzliche Server hinzuschalten, um Daten vom primären Knoten zu synchronisieren. So wird aus Redis-Stand-alone ein Master/Slave-Verbund. Daten werden auf

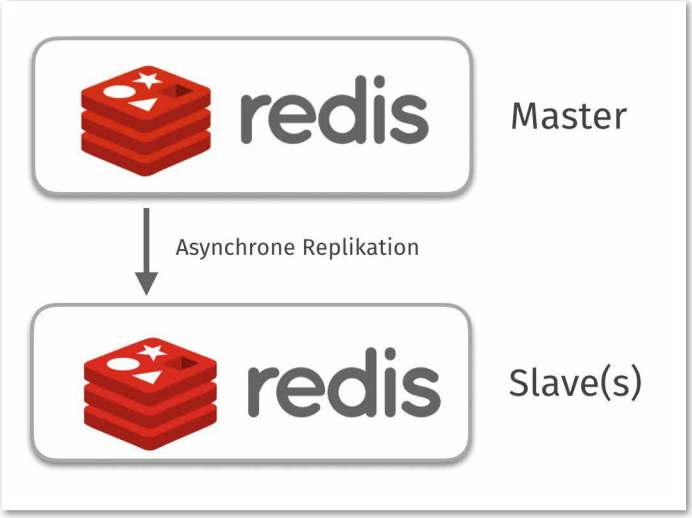


Abbildung 4: Master/Slave

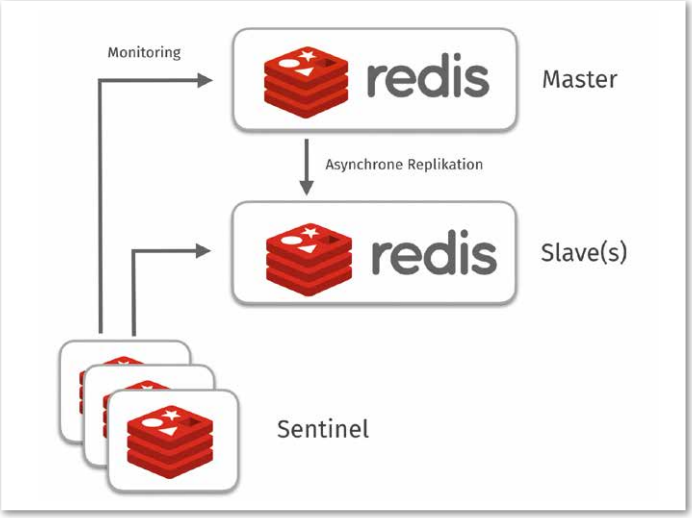


Abbildung 5: Sentinel

den Master-Knoten geschrieben und asynchron auf einen oder mehrere Slave-Knoten synchronisiert. Jeder Knoten beinhaltet die Gesamtheit der Daten, daher spricht man bei Master/Slave von Replikation.

Bricht im Fehlerfall der Master-Knoten weg, können Daten immer noch von den Slave-Knoten gelesen, jedoch nicht geschrieben werden: Es gibt bei Master/Slave keine Instanz, die einen Slave-Knoten zu einem Master umkonfiguriert *(siehe Abbildung 4)*. Auch ist die Konfiguration statisch gehalten, sodass die Clients nur beschränkt Topologie-Informationen besitzen.

Redis Sentinel

Sobald es um Hochverfügbarkeit geht, wird Redis Sentinel interessant. Es bietet Monitoring und Konfigurations-Management. Sentinel-Server werden als zusätzliche, externe Prozesse gestartet und bilden einen Verbund von Management-Servern. In einem Sentinel-Verbund sind Master- und Slave-Server registriert, die ab dem Zeitpunkt der Registrierung überwacht werden. Fällt ein Master-Knoten aus, so wählen Sentinel-Knoten einen neuen Master und aktualisieren die Konfigurationen der einzelnen Prozesse. Redis Sentinel stellt Topologie-Informationen zu Master- und Slave-Knoten bereit *(siehe Abbildung 5)*.

Redis Cluster

Ging es bei Master/Slave und Sentinel primär um Redundanz, so erweitert Redis Cluster den Betrieb um Datenpartitionierung („Sharding"). Redis Cluster verteilt Daten auf mehrere Master-Knoten. Ein Master ist exklusiv für Schreibzugriffe einer Datenpartition verantwortlich. Gelesen werden kann entweder von Master- oder Slave-Knoten. Ein Master-Knoten lässt sich mit Slave-Knoten für Redundanz absichern.

Fällt nun ein Master aus, so können Daten von anderen Master-Knoten gelesen und geschrieben werden, sofern nicht die ausgefallene Datenpartition betroffen ist. Bei Redis Cluster kommt eine erhebliche Einschränkung zum Tragen: Aufgrund der Partitionierung liegen Daten verteilt auf mehreren Knoten. Dies führt dazu, dass Befehle, die mehrere Keys betreffen (Multi-Key-Commands), nur eingeschränkt nutzbar sind.

Für das Sharding wird ein Hash aus dem Key errechnet und daraus ein Slot-Hash ermittelt. Über diesen Mechanismus werden Keys zu Slots zugeordnet. Slots sind wiederum einzelnen Master-Nodes zugewiesen. Multi-Key-Commands können daher nur auf demselben Slot operieren. Befehle, die mit Daten operieren, die auf mehrere Knoten verteilt sind, sind faktisch nicht realisierbar und müssten innerhalb der Anwendung selbst nachempfunden werden *(siehe Abbildung 6)*.

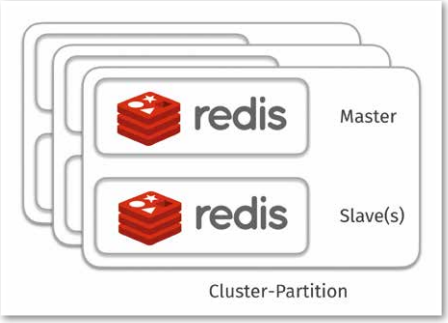


Abbildung 6: Redis Cluster

Redis in der Cloud

Redis kann lokal, in Containern oder im eigenen Rechenzentrum betrieben werden. Es gibt darüber hinaus auch einige Cloud-Angebote:

- AWS ElastiCache
- Azure Redis
- RedisLabs Redis
- RedisToGo

Bei einigen dieser Services (insbesondere AWS ElastiCache) ist zu beachten, dass Provider eigene HA/Clustering-Varianten implementieren, die von den standardmäßigen Redis-Mechanismen abweichen. So sind beispielsweise bei AWS-ElastiCache-HA-Implementierungen die Topologie-Informationen von Master/Slave nicht nutzbar, da dort private IP-Adressen angegeben werden, die nicht erreichbar sind. Oftmals werden auch DNS-Namen verwenden, um auf den aktuellen Master-Knoten zu zeigen, was wiederum bedeutet, dass der clientseitige DNS-Cache darauf ausgerichtet sein muss.

Persistenz

Redis ist ein In-Memory Key-Value-Store, was jedoch nicht bedeutet, dass Daten nicht persistent sind. Es stehen zwei Persistenz-Optionen zur Verfügung:

- Append-Only-File (AOF)
- Redis Database Snapshots (RDB, standardmäßig aktiv)

Bei AOF werden alle empfangenen Befehle hintereinander weggeschrieben. Ist das Journal zu groß, so kann die AOF-Datei

reorganisiert werden und ab dem aktuellen Stand aufsetzen. Nach einem Neustart werden alle AOF-Einträge als Redis-Befehle erneut abgespielt, um den Zustand und Daten wieder zu laden.

RDB Snapshots erstellt Point-in-Time-Snapshots des gesamten Datenbestands auf dem Server. Der Vorteil von RDB sind abgeschlossene und konsistente Snapshots, die schneller geladen werden können als AOF. Beide Methoden arbeiten asynchron. Die Schreibintervalle können durch Einstellungen konfiguriert werden.

Die Redis-Sicherheit

Die Sicherheit im Kontext des NoSQL-Servers tangiert mehrere Aspekte:

- Kommunikation
- Zugriff (Authentifizierung/Autorisierung)
- Datenablage

Redis bringt von sich aus nur wenig an sicherheitsrelevanter Unterstützung mit. Das Redis-Protokoll (RESP, Redis Server Protocol) ist ein Klartextprotokoll ohne zusätzliche Verschlüsselungsmöglichkeiten. Redis bietet von sich aus keine TLS-Unterstützung. Hier bietet es sich an, TLS-Tunnel zu verwenden (beispielsweise „stunnel"), um auf Transportebene eine verschlüsselte Übertragung zu gewährleisten.

Insbesondere bei der Verschlüsselung ist zu beachten, dass Redis in einem Master/Slave-Verbund/Cluster betrieben werden kann, der Details zur Topologie bereitstellt. Redis selbst hat kein Wissen über TLS-Tunnel oder umgeleitete Ports. Daher ist bei TLS-Betrieb ein Client erforderlich, der mit TLS und eventuellem Port-Remapping umgehen kann.

Der Zugriff auf den Server kann eingeschränkt werden. Redis unterstützt authentifizierte Verbindungen, die mit einem Passwort abgesichert werden können. Dabei handelt es sich um einen Alles-oder-nichts-Zugriff: Jeder Client, der dieses Passwort kennt, kann auf alle Daten innerhalb des Servers zugreifen. Die Authentifizierung erfolgt genauso schnell wie bei den übrigen Operationen: Es sind mehrere Hunderttausend Authentifizierungsversuche pro Sekunde möglich. Daher empfiehlt es sich, ein möglichst komplexes Passwort zu verwenden und den Server hinter einer Firewall zu betreiben, um zumindest die möglichen Zugriffsquellen einzuschränken.

Daten werden nicht weiter verschlüsselt. Die primäre Datenablage im Hauptspeicher enthält alle Daten im Klartext und die eingestellte Persistenz (Append-Only-File, RDB Snapshots) liegt ebenfalls im Klartext vor. Sicherheitskritische Daten sollten daher nur in verschlüsselter Form abgelegt werden.

Use-Cases

Redis eignet sich aufgrund seiner Datentyp-Vielfalt und seiner Single-Threaded-Eigenschaften für eine ganze Reihe von Anwendungsfällen:

- Als Key-Value-Store
- Caching
- Queues
- Zum Puffern von Daten-Importen und Real-Time-Analysen
- Als Session-Store für verteilte Web-Sessions
- Implementierung von verteilten Sperren (Distributed Locking)
- Geo-Indexing

Primär wird Redis für Caching und als Key-Value-Store verwendet. Der immense Durchsatz auf In-Memory-Daten in einem eigenen Prozess (Off-Heap-Storage) ist ein überzeugendes Argument dafür. Keys können mit einer Time-To-Live (TTL) versehen und bei deren Erreichen freigegeben werden. Speicher-Richtlinien (Eviction Policies) können eingestellt werden, um Speicher freizugeben, sobald eine eingestellte Speichergrenze erreicht ist.

Eine ganz andere Kategorie von Anwendungsfällen ermöglicht das Single-Threaded-Verhalten. Alle Zugriffe werden serialisiert und von einem Thread bearbeitet, was garantiert, dass immer nur ein Prozess auf Daten zugreift (lesend, schreibend). Damit ist auch sichergestellt, dass Nebenläufigkeits-Erscheinungen nicht auftreten können. Versuchen zwei Clients gleichzeitig, einen Zähler zu inkrementieren, so werden die Befehle serialisiert und nacheinander ausgeführt. Mit diesem Verhalten löst Redis Synchronisations-Herausforderungen auf eine natürliche Art. Daraus gehen Nutzungsmöglichkeiten wie atomare Zähler, Queues und Sperren hervor.

Es gibt auch etliche Use-Cases, die sich nicht für Redis eignen. Daten sollten trotz Persistenz-Optionen als semi-persistent angesehen werden. Daten, die nicht wiederbeschafft werden können, sollten man möglichst nicht in Redis ablegt. Einzelne Schlüssel und auch der gesamte Datenbestand lassen sich sehr schnell löschen;

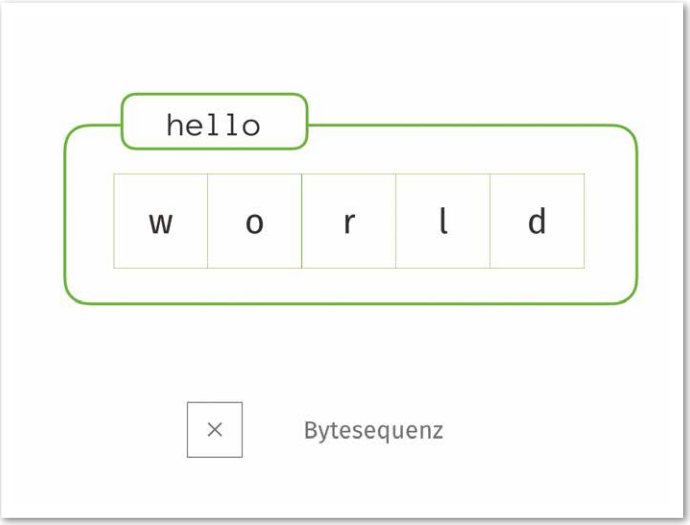


Abbildung 7: Ein Redis String

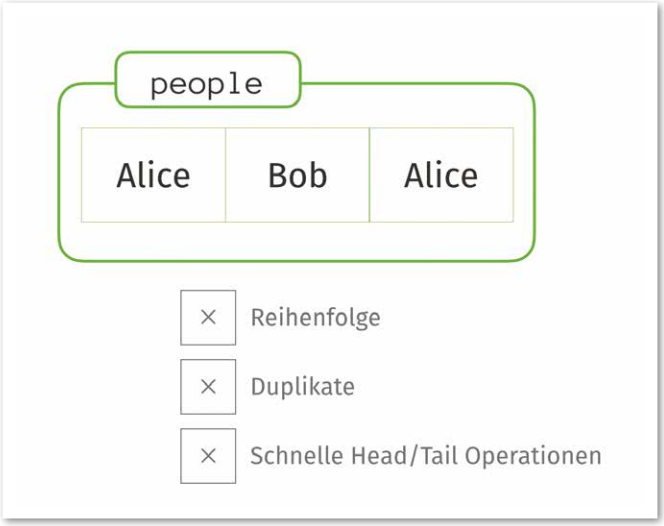


Abbildung 8: Liste

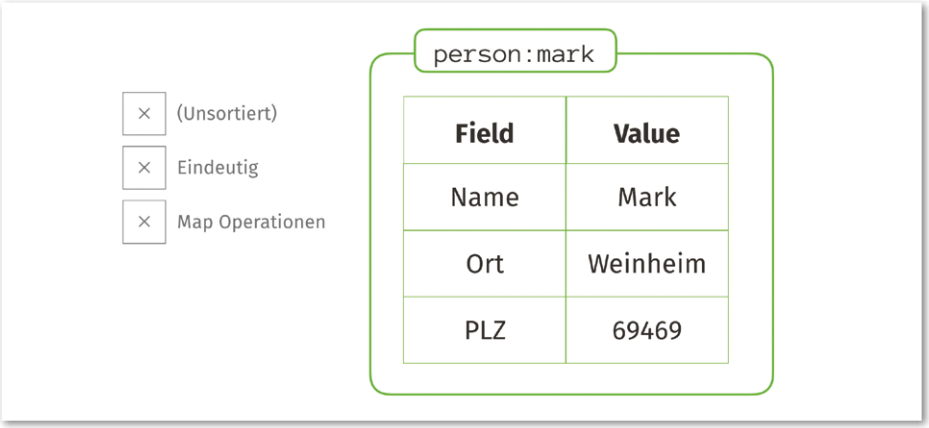


Abbildung 9: Hash

diese Löschung ist je nach Persistenz-Option dann auch festgeschrieben.

Datentypen

Redis stellt als Key-Value-Store eine ganze Reihe fundamentaler Datentypen bereit:

- Key-Value-Tupel („String")
- Liste
- Set
- Hash
- Sortiertes Set
- HyperLogLog (Erweiterung von „String" mit eigener Semantik)
- Geo-Index (Erweiterung eines sortierten Sets mit eigener Semantik)

Neben den unterstützten Datentypen unterstützt Redis Transaktionen, Lua-Scripting und Publish/Subscribe Messaging. In jedem Datentyp können jeweils ein oder mehrere Werte abgelegt sein, jedoch können Datentypen nicht geschachtelt werden (so sind Listen von Hashes nicht möglich). Jeder Datentyp benötigt sein individuelles API, da eine Liste anders bedient wird als ein Hash oder ein Geo-Index. Daher ist es nicht verwunderlich, dass rund 170 Befehle verfügbar sind, die mit ihren Unterbefehlen auf etwa 650 ausführbare Befehle kommen. Natürlich lässt diese Menge jegliches Client-API explodieren. Das ist auch der Grund dafür, warum manche Clients Befehle in API-Gruppen unterteilen und mithilfe gruppierter APIs bereitstellen. Wenn es um Datentypen-Beschränkungen geht, so sind zwei Größen relevant:

- String-Limit: Ein String kann bis zu 512 MB (2 hoch 29 - 1 Bytes) umfassen. Strings werden an unterschiedlichen Stellen verwendet, entweder direkt als Tupel-Wert oder als Element einer höherwertigen Datenstruktur (Liste, Hash, Set etc.)
- Anzahl der Elemente in einer Datenstruktur: Listen, Hashes, Sets und andere Datenstrukturen können bis zu 4,29 TB (2 hoch 32 - 1 Bytes) untergeordnete Elemente beinhalten.

Elemente innerhalb von Datenstrukturen sind zumeist einfache Strings ohne weitere Semantik. Referenzen zwischen Daten werden üblicherweise als Strings modelliert, bei denen die Anwendung weitere Lookups durchführt.

Key-Value-Tupel („String")

Ein Redis-String ist eine unstrukturierte Folge von Bytes und „binary-safe", was bedeutet, dass die Bytes in ihrer ursprünglichen Form gespeichert und ausgegeben werden (ohne zusätzliche ASCII-, UTF-8-Kodierung etc.) – Redis speichert Bytes so, wie sie übertragen wurden. Redis kennt innerhalb eines Strings keine weitergehende Strukturierung oder Semantik. Ein String kann, direkt unter einem Schlüssel (Key) abgelegt (SET), gelesen (GET) werden. Jede Datenstruktur, die über einen Key identifiziert wird, kann gelöscht (DEL) oder mit einem TTL (EXPIRE) versehen werden. Strings sind die zugrunde liegenden Datenstrukturen für Bit- und HyperLogLog-Operationen. Diese Befehle verleihen Strings eine eigene Semantik und so werden, je nach Dokumentation, Bit und HyperLogLog als eigene Datenstrukturen geführt *(siehe Abbildung 7)*.

Liste

Listen enthalten mehrere Werte in einer geordneten Reihenfolge und erlauben Duplikate. Listen sind auf schnelle Anfangs- und End-Operationen (Head/Tail) ausgelegt, sodass Lesen (LRANGE), Einfügen (LADD) und Entfernen (LREM) von Elementen in konstanter Zeit (O(1)) ausgeführt werden *(siehe Abbildung 8)*.

Eine leere Liste (Liste ohne Elemente) wird als nicht vorhanden angesehen. Dieses Verhalten ist bei allen Datentypen außer dem String konsistent. Ein leeres Set oder ein leerer Hash werden im Keyspace nicht als Key geführt.

Hash

Ein Hash (Hashtabelle oder assoziatives Array) ist eine Zuordnung von Key-Value-Paaren, die wiederum als Ganzes mit einem Key identifiziert wird. Ein Hash ist vergleichbar mit einer Java HashMap. Hash-Elemen-

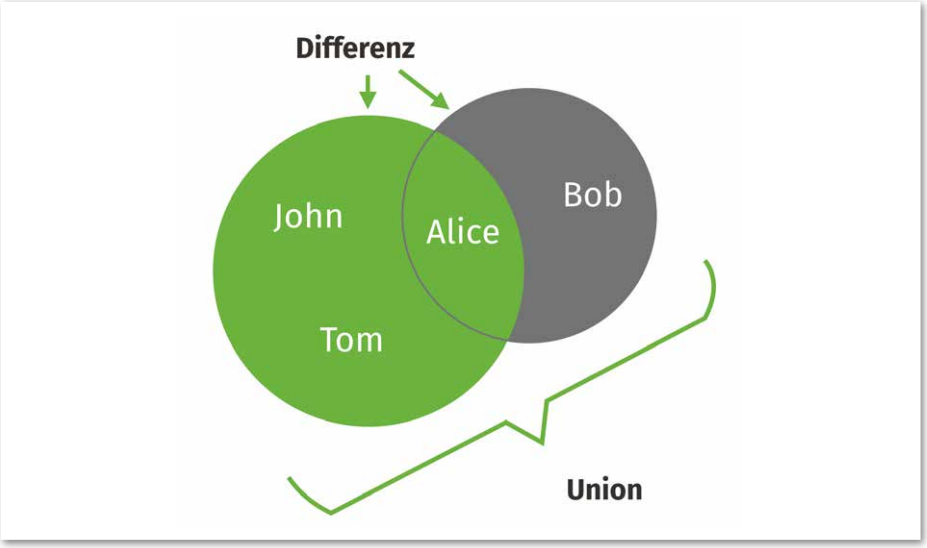


Abbildung 10: Set

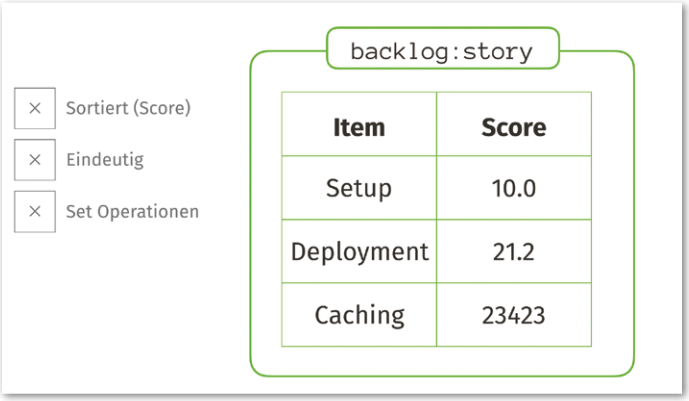


Abbildung 11: Sortiertes Set

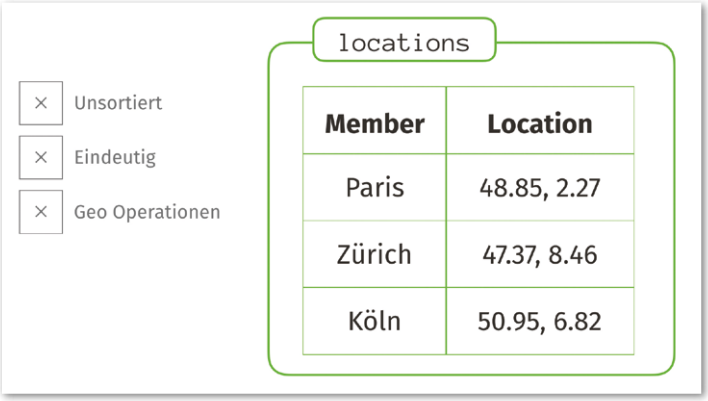


Abbildung 12: Geo-Index

te bestehen aus einem Feldnamen und einem Wert. Feldnamen und Werte sind Redis Strings und somit Byte-genau. Hashes eignen sich, um strukturierte Daten abzubilden *(siehe Abbildung 9)*.

Set

Ein Set enthält, ähnlich wie eine Liste, mehrere Werte, jedoch ohne Reihenfolge. Sets lassen Duplikate nicht zu. Hinzufügen/Entfernen von Set-Elementen erfolgt in konstanter Zeit (O(1)). Mit Sets lassen sich serverseitig Differenz- (SDIFF), Vereinheitlichungs- (SUNION) und Überschneidungs-Operationen (SINTER) ausführen und optional in einem neuen Set speichern (wie SINTERSTORE). Diese Operationen gehören zur Kategorie der Multi-Key Commands und bedürfen besonderer Aufmerksamkeit im Redis-Cluster-Betrieb *(siehe Abbildung 10)*.

Sortiertes Set

Sortierte Sets verhalten sich wie eine Mischung aus Set und Hash. In einem sortierten Set ist jedem Element ein Score zugeordnet (Fließkomma-Zahl). Anhand des Scores und der lexikographischen Ordnung wird die Sortierung festgelegt. So können in einem sortierten Set die ersten/letzten Elemente abgerufen werden oder Bereiche nach Score oder lexikographischer Ordnung gelesen werden *(siehe Abbildung 11)*.

Basierend auf sortierten Sets sind Geo-Indizes realisiert. Bei Geo-Indexing wird in einem Geo-Set ein Set von Elementen mit zugeordneter Geo-Position abgebildet. Die Geo-Position wird vom Geo-API als WGS84 Breiten- und Längengrad entgegengenommen und in einen Geohash *[7]* kodiert. Dieser entspricht einem 52-Bit-Integer und wird als Score im sortierten Set abgelegt. Elemente werden über das Geo-API zum Geo-Set hinzugefügt (GEOADD), jedoch gibt es keinen Geo-API-Befehl, um Elemente wieder zu entfernen, daher muss der Befehl ZREM verwendet werden, mit dem Elemente aus einem Sortieren-Set entfernt werden *(siehe Abbildung 12)*.

Publish/Subscribe

Publish/Subscribe (Pub/Sub) ist im eigentlichen Sinne keine Datenstruktur, da Messages transient sind. Dabei handelt es sich um ein leichtgewichtiges Messaging, bei dem ein Publisher eine Nachricht (Message) an einen Kanal (Channel) publiziert. Interessierte Clients (Subscriber) können sich für Kanäle registrieren, um Nachrichten von einem oder mehreren Kanälen zu erhalten, oder sich für Kanal-Muster (Pattern) registrieren. Pattern-Subscriptions eignen sich, um Nachrichten mehrerer Kanäle zu erhalten, die mit einer bestimmten Zeichenfolge beginnen oder enden.

Nachrichten werden nicht persistiert; ist ein Client zum Sende-Zeitpunkt nicht verfügbar, so verfällt die Nachricht und wird nicht zugestellt. Redis verwendet Pub/Sub auch intern, um Keyspace-Ereignisse (Schlüssel angelegt/Timeout abgelaufen/etc.) zu publizieren. Sentinel nutzt Pub/Sub für die Kommunikation von Konfigurations-Ereignissen, wie dass ein Master-Knoten nicht verfügbar ist.

Fazit

Die aktuelle Redis-Version 3.2.6 wurde Anfang Dezember 2016 veröffentlicht. Die Arbeiten an der nächsten Major Version, 4.0 haben bereits Mitte 2016 begonnen. Mit Redis 4.0 wird Redis Cluster stabil und erhält zusätzliche Optionen für Port-Remapping, was insbesondere im Docker-Betrieb erforderlich ist. Zudem erhält Redis ein Modul-System, um Funktionalität durch Plug-in-Module zu erweitern. Erste Module wurden bereits durch Salvatore, RedisLabs und die Community entwickelt. Zu den beachtenswertesten Modulen gehört auch das Neural Redis Modul, das ein neutrales Netzwerk bereitstellt.

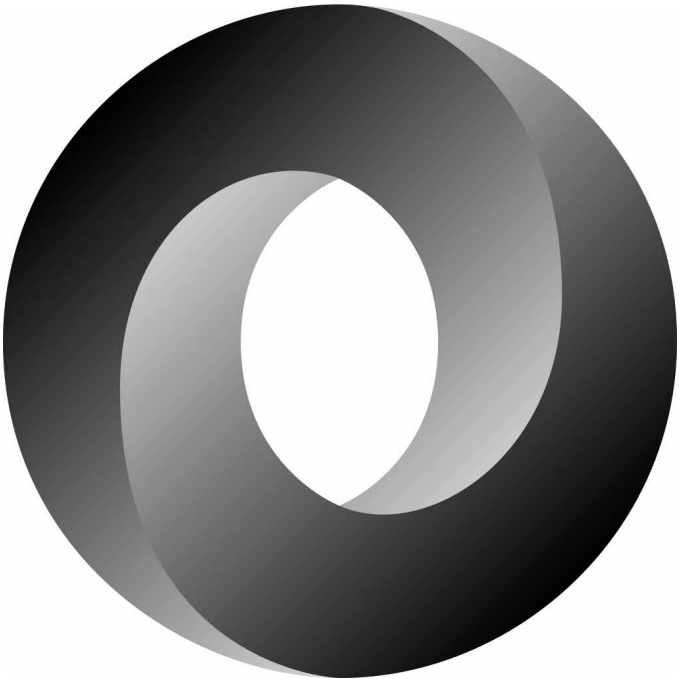
Module können bestehende Datenstrukturen durch Befehle erweitern oder neue Datenstrukturen mitbringen. Hier sind auch Client Maintainer gefragt, um einen möglichst wiederverwendbaren Ansatz für Module bereitzustellen.

Links

[1] http://github.com/antirez/redis
[2] http://redis.io/clients
[3] https://github.com/jamespedwards42/jedipus
[4] https://github.com/xetorthio/jedis
[5] https://github.com/mp911de/lettuce
[6] https://github.com/redisson/redisson
[7] https://en.wikipedia.org/wiki/Geohash

Mark Paluch
mpaluch@pivotal.io

Mark Paluch ist Software Craftsman und arbeitet als Spring Data Engineer bei Pivotal. Er fing schon als Jugendlicher an, Software zu entwickeln, und arbeitete fünfzehn Jahre in Software-Projekten rund um Java-Server-, Frontend- und Web-Technologien, bevor er seinen Fokus auf Open-Source-Software verlegte. Mark leitet mehrere Open-Source-Projekte, zu denen auch der Lettuce-Redis-Treiber gehört. Als Sprecher trägt er regelmäßig auf internationalen Konferenzen vor und ist Autor zahlreicher Artikel.



JSON Schema-Driven Development in Java mit JSSD

Andreas W. Bartels, Disy Informationssysteme GmbH

Der Wandel von Desktop-Anwendungen hin zu Web-Anwendungen in Browsern sorgte auch für einen Wandel bei den Daten-Austauschformaten. Dies lässt sich deutlich daran erkennen, dass Services, die in den letzten Jahren entstanden sind, nicht mehr oder nicht ausschließlich XML-basiert arbeiten, sondern eher auf JSON als Austauschformat setzen.

Entwickler, die Sprachen wie Java verwenden, die Typensicherheit bieten, und Frameworks nutzen, die auch für JSON auf XSD-Schemas setzen, haben einige Probleme zu lösen. Bei Geo-Informationssystemen spielen da besonders die JSON-Ausprägungen von Geometry-Objekten eine Rolle. Um diese Probleme zu lösen, ist die Sprache JSSD *[1]* entstanden, die vom Autor in Privatprojekten und im Unternehmen eingesetzt wird.

Die vom Open Geospatial Consortium (OGC) *[2]* definierten Webservices wie der Web Feature Service (WFS) dominieren seit Anfang dieses Jahrtausends die Geodatendienste. Deren Daten-Austauschformate benutzen durchgängig XML. Daher hat sich im GIS-Bereich neben der textuellen Beschreibung des Formats die XML Schema Definition (XSD) als Beschreibungssprache durchgesetzt. Einer der Vorteile dieser formalen und bedingt durch Menschen lesbaren Sprache ist, dass sich daraus Klassen generieren lassen, die den Entwicklern den Umgang auch mit komplexen Datenstrukturen erleichtern.

In den letzten fünf Jahren tauchen aber auch im GIS-Bereich immer mehr Services auf, die ausschließlich JSON *[5]* verwenden. Beispiel sind die Geo-Datendienste ArcGIS REST Server von ESRI *[4]* sowie die Cloud-Dienste Mapbox *[5]*, Carto *[6]* und Mapnik *[7]*. Aber auch die gerade aktuell werdenden Open-Data-Dienste, die CKAN *[8]* beziehungsweise OGD-Metadata (Open Government Platform) *[9]* nutzen, sind entsprechende Beispiele.

Durch die explizite Entscheidung dieser Dienste für JSON können Datenstrukturen entstehen, die sich mit XSD nicht mehr abbilden lassen. Dadurch entfällt auch die Nutzung von Frameworks wie beispiels-

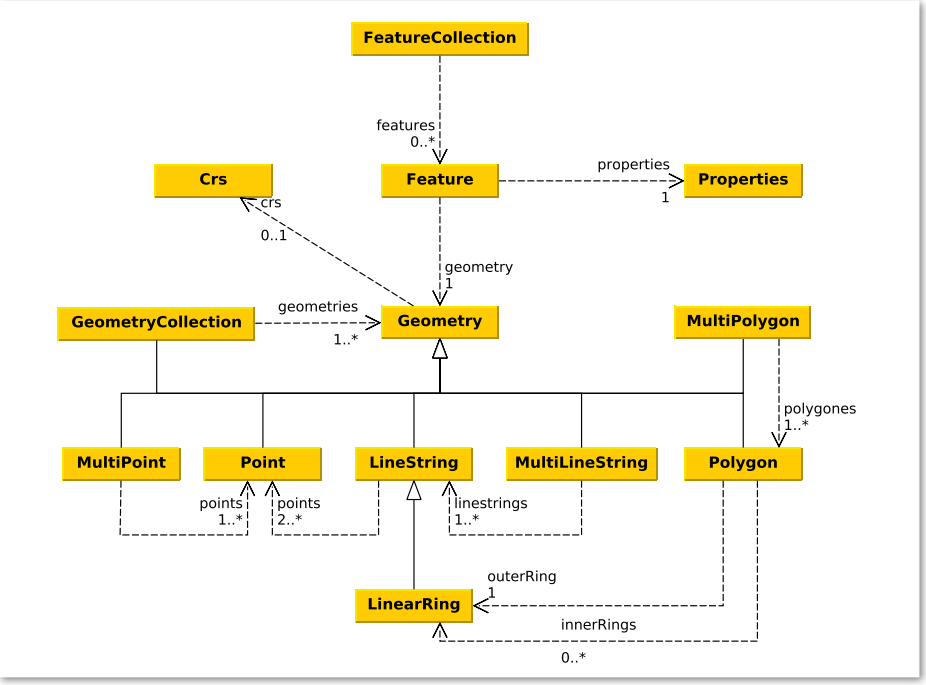


Abbildung 1: UML-Diagramm „Simple Feature“

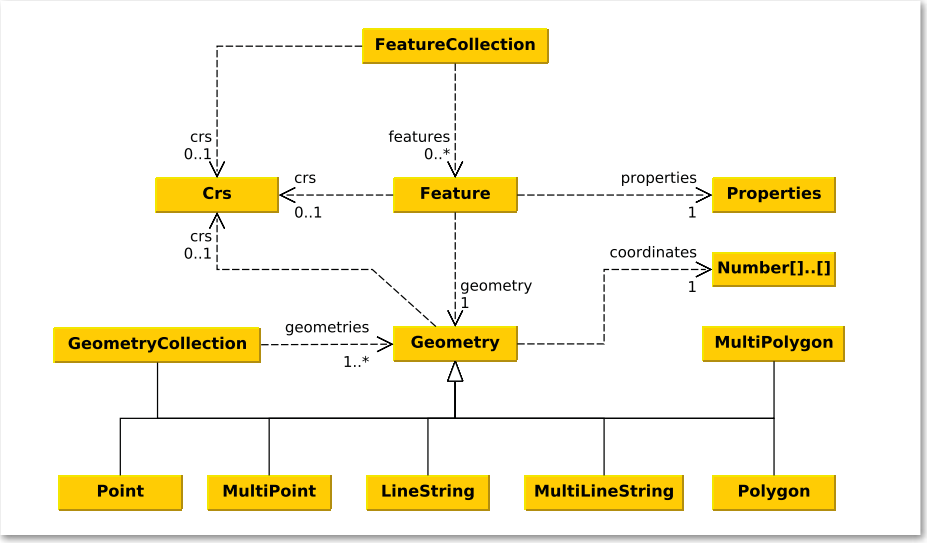
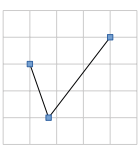


Abbildung 2: UML-Diagramm „GeoJSON“

```
{ "type": "Point",
  "coordinates": [3.5, 2]
}
```



```
{ "type": "LineString",
  "coordinates": [
     [2, 6], [3.5, 2], [8, 8]
  ]
}
```



```
{ "type": "Polygon",
  "coordinates": [
     [[1, 1], [9, 1], [9, 9], [1, 9], [1, 1]],
     [[2, 2], [2, 8], [6, 2], [2, 2]],
     [[8, 4], [4, 8], [8, 8], [8, 4]]
  ]
```

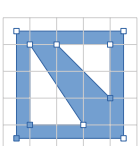


Abbildung 3: JSON-Ausprägung „Point“, „LineString“ und „Polygon“ mit Beispiel-Grafik

weise JAXB mit Jackson, die es für entsprechend designte Strukturen erlauben, mit XML oder JSON zu arbeiten. Möchte man die Vorteile von XSD nicht verlieren, bedarf es einer Sprache, die die Aufgaben von XSD für JSON übernimmt. Ein Projekt, das diese Rolle übernehmen möchte, ist JSON-Schema *[10]*. Um diese Sprache herum sind in den letzten Jahren einige vielversprechende Projekte entstanden. Eins davon ist „SON-Schema2pojo *[11]*. Dieses Werkzeug ermöglicht das Generieren von Java-Klassen aus JSON-Schemadateien. Im Internet ist es das Projekt dieser Art, das die meisten Referenzen aufweist. Es unterstützt die meisten Features des JSON-Schemas und ist für viele Projekte daher sicher ausreichend.

Bei Geo-Datendiensten hat sich GeoJSON als maßgebliche Datenstruktur durchgesetzt. Die GeoJSON-Datenstruktur lässt sich zwar mit JSON-Schema beschreiben, aber es gibt noch keinen Java-Code-Generator, der daraus eine zufriedenstellende Struktur erzeugt. Dieser Sachverhalt und die folgenden Anforderungen an eine Sprache, zur Generierung von Java-Klassen für die Serialisierung von JSON, haben zur Entwicklung von JSSD geführt:

- Schnelles Definieren von Datentypen anhand von JSON-Ausprägungen
- Berücksichtigung der Eigenheiten von Geo-Objekten
- Beschreibende Erläuterungen der Datentypen innerhalb der Schema-Dateien
- Dokumentation des Verhaltens innerhalb der Schema-Dateien
- Übersichtlichkeit und leichte Verständlichkeit der Schema-Dateien

GeoJSON

GeoJSON ist ein JSON-Format, um geografische Objekte abzubilden. Die erste Format-Spezifikation ist im Juni 2008 *[12]* erschienen. Seit August 2016 ist GeoJSON durch die Internet Engineering Task Force (IEFT) als Standard Track Document RFC7946 *[13]* veröffentlicht. Zudem können die in den beiden Spezifikationen beschriebenen Strukturen als JSON-Umsetzung der gängigsten Geometrie-Typen der Simple Feature Access, einer Spezifikation des Open Geospatial Consortium (OGC), und des ISO-Standards 19125 betrachtet werden.

Ein geografisches Objekt setzt sich aus Sachdaten einer Geometrie und dem zugrunde liegenden Koordinaten-Referenzsystem zusammen. Der Datentyp, der

einem geografischen Objekt entspricht, heißt „Feature", eine Liste davon „FeatureCollection", die Sachdaten nennen sich „Properties" und die Geometrie „Geometry". Optional kann ein Feature noch einen eindeutigen Identifier beinhalten. Die Pre-RFC-Spezifikation erlaubt die Angabe eines Koordinaten-Referenzsystems entweder in der FeatureCollection, dem Feature oder der Geometry *(siehe Abbildung 1)*.

Das RFC schränkt GeoJSON auf das vom GPS verwendete geografische Referenzsystem WGS84 ein. Koordinaten-Referenzsysteme beschreiben den Bezug der Geometrie zur Erdoberfläche. Zusätzlich kann ein Koordinaten-Referenzsystem auch die Abbildung der Geometrie in einer Karte definieren. Das durch das RFC vorgegebene System beschreibt nur den Bezug der Geometrien zur Erde. Als Geometrietypen unterstützt GeoJSON Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon und GeometryCollection *(siehe Abbildung 2)*.

Der deutlichste Unterschied zwischen einer Simple-Feature-Implementierung und GeoJSON ist, wie beim Vergleich zwischen *Abbildung 1 und 2* zu sehen, die Abbildung der Geometriestruktur in der Datenstruktur. Während die Simple-Feature-Definition komplexere Geometrien durch die Verwendung einer oder mehrerer Geometrien der nächst weniger komplexen Struktur abbildet, benutzt GeoJSON ein multidimensionales Number Array, um die Struktur abzubilden *(siehe Abbildung 3)*.

Dieses multidimensionale Array ist die Struktur, die sich mit XSD nicht abbilden lässt. XSD unterstützt nur eindimensionale Arrays. Mit JSON-Schema lässt sich die Struktur zwar beschreiben, aber bisher nicht in zufriedenstellende Java-Klassen konvertieren. GeoJSON-Schemata sind unter *[14]* beziehungsweise *[15]* zu finden.

JSSD

Ausgehend von der Anforderung, dass mit geringem Aufwand aus Beispiel-JSON-Ausprägungen Datentypen beschrieben und generiert werden sollen, basiert JSSD auf der JSON-Grammatik *[7]*. Daher entspricht der JSSD-Code für ein Objekt auch dem eines JSON-Objekts *(siehe Listing 1)*. Um Java-Klassen mit typisierten Fields generieren zu können, wurde die JSON-Grammatik für Member um eine Type-Angabe erweitert *(siehe Listing 2)*.

Als Datentypen können JSSD-Grammatiken oder Java-Klassen angegeben sein. Um die multidimensionalen Koordinaten-Arrays der Geo-Objekte zu unterstützen, lassen sich hier auch entsprechende Arrays als Typ definieren.

```
--   net/anwiba/jssd/example/bar.jssd  ---------------------
{
  "title": <string> "bar",
  "foo": <foo[][]> null
}
--  net/anwiba/jssd/example/Bar.java  ----------------------
package net.anwiba.jssd.example;
import com.fasterxml.jackson.annotation.JsonProperty;
public class Bar {
    private String type = "bar";
    private Foo[][] foo = null;
    @JsonProperty("type")
    public void setType(final String type) {
        this.type = type;
    }
    @JsonProperty("type")
    public String getType() {
        return this.type;
    }
    @JsonProperty("foo")
    public void setFoo(final Foo[][] foo) {
        this.foo = foo;
    }
    @JsonProperty("foo")
    public Foo[][] getFoo() {
        return this.foo;
    }
}
```

Listing 1: JSSD-Objekt und Java-Ergebnis

```
--  net/anwiba/gis/geo/json/geometry.jssd  -----------------
@JssdFactory
{ "type" : <String> "Geometry",
    "crs": <crs> null,
    "bbox": <double[]> null
}
--  net/anwiba/gis/geo/json/Geometry.java  -----------------
package net.anwiba.gis.geo.json;
 :
import net.anwiba.commons.utilities.string.StringUtilities;
public class Geometry {
  private String type = "Geometry";
  private Crs crs = null;
  private double[] bbox = null;
   :
  @JsonCreator
  public static Geometry create(
      @JsonProperty("type") String type
  )
    if (StringUtilities.isNullOrTrimmedEmpty(type)) {
      return new Geometry();
    }
    Class<? extends Geometry>clazz=_createClass(type);
    if (clazz!= null) {
      return _createBean(clazz);
    }
    return new Geometry();
  }
  :
}
--  net/anwiba/gis/geo/json/point.jssd  --------------------
@JssdExtends(type="geometry"
{ "type": <String> "Point",
  "coordinates": <Number[]> null
}
```

Listing 2: JSSD-Member und Java-Ergebnis

Vererbung

Um die unterschiedlichen Dimensionen der Geometrien beschreiben zu können, sind aber noch zwei weitere Fähigkeiten nötig:

```
--  net/anwiba/jssd/example/bar.jssd  ----------------------
/*
* Beispiel Struktur
*/
{ // type bar
  "type": <String> "Bar",
  // array of foos
  "foos": <foo[]> null
}
```

Listing 3: JSSD-Extends und -Factory mit Java-Ergebnis

die Vererbung sowie die Factories, die anhand der Daten die entsprechende Klasse instanziieren können. Zur Umsetzung dieser Fähigkeiten benutzt JSSD Annotationen, die der Java-Syntax entsprechen *(siehe Listing 3)*.

JSSD unterstützt drei Factory-Strategien. Die einfachste ist, dass der Member „type" verwendet wird, um richtige Klassen zu ermitteln. Es kann aber auch eine Factory-Klasse in der Factory-Annotation definiert oder während der Laufzeit über Injection gesetzt werden.

Verhalten

Die im oberen Absatz beschriebenen Factory-Strategien sind bereits eine Definition des Verhaltens der generierten Klassen. Es hat sich während der Implementierung von JSSD gezeigt, dass sich Best-Case-Implementierungen in JavaScript und Java durchaus voneinander unterscheiden. Ein Beispiel dafür sind wieder Arrays beziehungsweise Listen: Während Java-Entwickler erwarten, dass diese „nicht null" sein können, ist es in JavaScript üblich, den Member in die JSON-Struktur erst gar nicht einzufügen. Um hier Missverständnissen aus dem Weg zu gehen, können Members über eine entsprechende Annotation als NotNullable deklariert werden.

Übersichtlichkeit

JSSD unterstützt Kommentare entsprechend der Java-Syntax. Die Kommentare werden zurzeit nicht in den generierten Java-Code übernommen *(siehe Listing 4)*. Um die noch fehlende Anforderung nach Übersichtlichkeit zu erfüllen, unterbindet JSSD die verschachtelte Definition von Objekten, erlaubt aber die Definition mehrerer Datentypen in einer Datei *(siehe Listing 5)*.

Für die Integration der Java-Codegenerierung in ein Build-System beziehungsweise in eine IDE gibt es ein Maven-Plug-in und eine Ant-Task-Implementierung *(siehe Listing 6)*. Aus den JSSD-Schema-Dateien werden Java-Klassen mit Jackson-JSON-Annotation generiert. Damit ist die Serialisierung beziehungsweise Deserialisierung in Java oder als Request- beziehungsweise Response-Body mit Spring MVC mit wenigen Zeilen Code möglich *(siehe Listing 7)*.

Fazit

Der Autor setzt JSSD in seinem privaten Projekt JGISShell *[16]* ein, um ArcGIS-REST- und Mapbox-Dienste zu nutzen, zum Lesen und Schreiben von GeoJSON und zum Zugriff auf den unter Linux laufenden GPS-Server gpsd. In Cadenza *[17]* nutzt er bei Disy mit JSSD generierte Klassen wie in JSGIShell für den Zugriff auf die Geo-Datendienste ArcGIS REST und Mapbox sowie Carto. Darüber hinaus werden mit JSSD generierte Klassen auch für den Datenaustauch zwischen Cadenza Mobile und einer Cadenza-Server-Implementierung verwendet.

Die besondere Stärke von JSSD ergibt sich durch Ähnlichkeit zu den JSON-Ausprägungen, die Reduzierung auf die wesentlichen Informationen und die Verwendung von Entwicklern vertrauten Mechanismen aus Java, wie beispielsweise die Annotationen. Dies hat sich besonders durch eine im Vergleich zur Anzahl benötigter Klassen schnelle Erstellung und leichte Erweiterbarkeit der beiden ArcGIS-REST-Clients gezeigt. Die ArcGIS-REST-Anbindung in JGISShell verwendet 42 JSSD-Schemadateien. Dabei unterstützt die Implementierung nur ein kleines Spektrum der vom ArcGIS REST Server angebotenen Dienste. Wünschenswert wären folgende Weiterentwicklungen:

- Übernahme der Kommentare aus den Schema-Dateien in den Java-Code
- Ein Tool, das anhand von JSON-Ausprägungen JSSD-Schema-Dateien generiert

Bleibt noch eine offene Frage: Wofür steht JSSD eigentlich? Die Grundidee war es, etwas Ähnliches zu erschaffen, wie es die Kombination von XSD und JAXB für XML bietet. Daher steht JSSD für JSON Schema Definition – den Anspruch, der dahinter steht, kann und soll JSSD allerdings nicht erfüllen. Die maßgebende Zielsetzung war das Generieren von Java-Klassen. Eine Verwendung wie bei JSON-Schema zum Validieren oder Transformieren von Daten ist nicht vorgesehen.

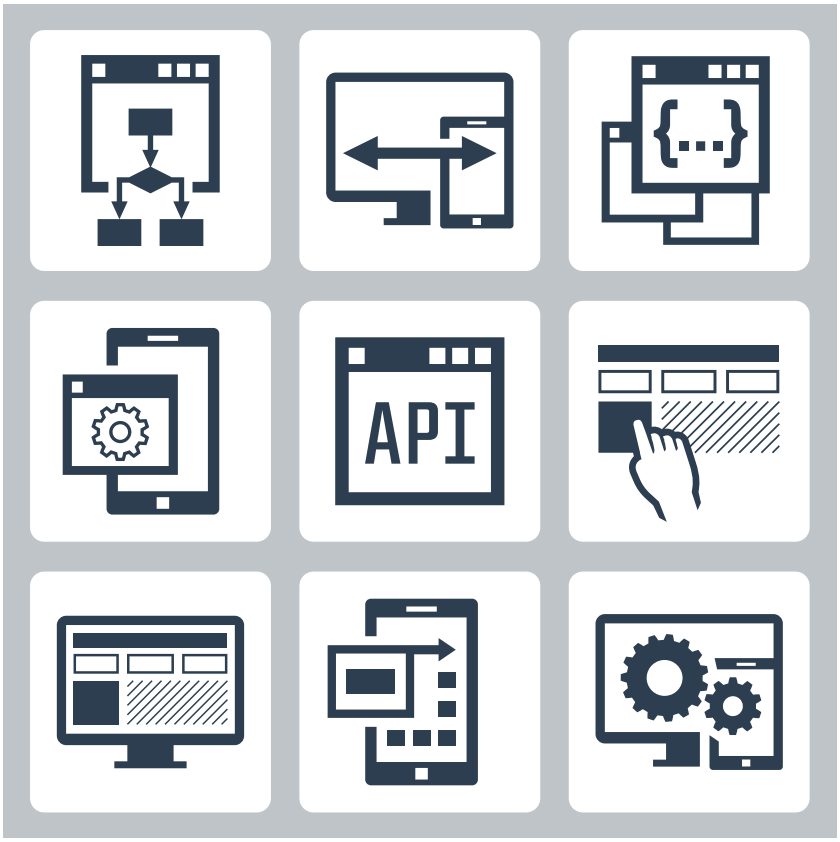
Referenzen

[1] https://github.com/AndreasWBartels/libraries/wiki/JSSD
[2] http://www.opengeospatial.org
[3] http://www.json.org
[4] http://resources.arcgis.com/en/help/rest/apiref
[5] https://www.mapbox.com/developers
[6] https://carto.com/docs
[7] https://mapzen.com/documentation
[8] http://docs.ckan.org/en/latest
[9] https://github.com/GovDataOfficial/OGD-1.1
[10] http://json-schema.org
[11] http://www.jsonschema2pojo.org
[12] http://geojson.org/geojson-spec.html
[13] https://tools.ietf.org/html/rfc7946
[14] http://www.jsonschemavalidator.net
[15] https://github.com/fge/sample-json-schemas/tree/master/geojson
[16] https://github.com/AndreasWBartels/JGISShell
[17] http://www.disy.net/produkte/cadenza.html

Hinweis: Die Listings 4 bis 7 finden Sie unter *www.doag.org/go/java_aktuell/201702/listings*

Andreas W. Bartels
andreas.bartels@disy.net

Andreas Bartels ist Software-Architekt bei der Disy Informationssysteme GmbH mit den Schwerpunkten "GIS" und "Anbindung von Geodatenquellen". Er hat an der Universität-Gesamthochschule Paderborn Angewandte Informatik mit Nebenfach Geographie studiert und ist Vermessungstechniker. Seit etwa dreißig Jahren arbeitet er im GIS-Umfeld als Anwender, Anwendungsbetreuer, Fachberater, Software-Entwickler und -Architekt.



Application Programming Interfaces – auf welche Entwurfsziele man achten sollte

Kai Spichale, SAP SE

Application Programming Interface (API) ist nur der allgemeine Oberbegriff für viele unterschiedliche Spielarten, die uns beispielsweise im Web in Form von RESTful HTTP oder JSON-RPC begegnen. Eine weitere Kategorie bilden die objektorientierten Programmiersprachen-APIs, wie man sie mit Java entwerfen und implementieren kann. Ungeachtet dieser Unterscheidung verfolgen erfahrende API-Designer eine Reihe wiederkehrender Ziele, die in diesem Artikel vorgestellt sind.

Ein objektorientiertes Programm besteht typischerweise aus einer Vielzahl von Objekten, die durch den Austausch von Nachrichten miteinander kommunizieren und auf diese Weise ein gewünschtes Informationssystem simulieren. Zwischen diesen Objekten bestehen Abhängigkeiten, die einen Graphen bilden *(siehe Abbildung 1)*. Weil Software-Entwicklung nur in seltensten Ausnahmefällen ohne Wiederverwendung auskommt, besteht auch das gezeigte Beispiel-Programm aus eigenen und fremden (wiederverwendeten) Objekten, die beispielsweise aus der Java-Standardklassen-Bibliothek stammen.

Die Einhaltung der Zugriffs-Modifizierer wird von der JVM sichergestellt, doch zwischen eigenen und fremden Klassen oder eigenen und fremden Packages wird nicht unterschieden, weil der Bytecode im Klassenpfad gleichbehandelt wird. Deswegen ist das Modell nicht von klaren Kanten durchzogen, die die Objekte in Schichten einteilen würden. Nichtsdestotrotz ist der eigene Code, der Client-Code, der direkt und indirekt von der wiederverwendeten Bibliothek abhängt, zur Veranschaulichung farblich oran-

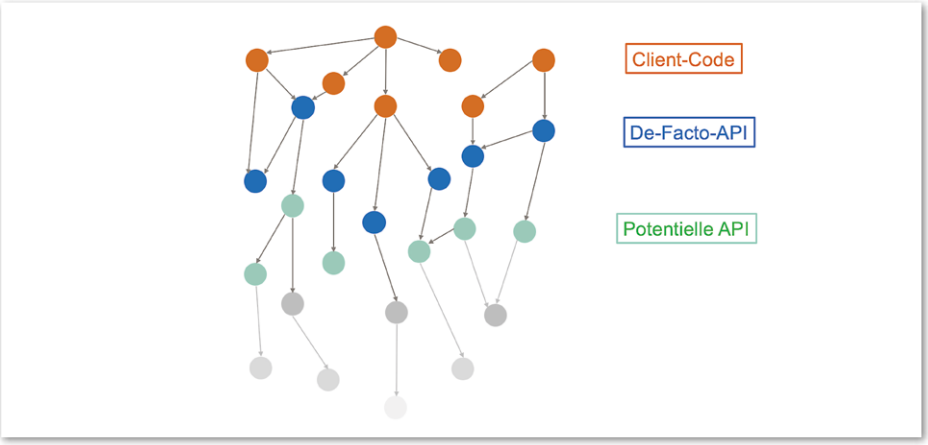


Abbildung 1: Die Objekte eines Programms bilden einen gerichteten Graphen. Die Wiederverwendung von existierendem Code erfolgt über ein De-facto-API, das nicht immer für diesen Zweck optimiert ist, aber sein sollte.

ge hervorgehoben. Alle anderen Objekte des Modells stammen aus der wiederverwendeten Bibliothek, die über ein API angeschlossen ist. In der Abbildung ist dieses API als „De-facto-API" bezeichnet, weil diese Objekte nicht explizit für diesen Zweck entworfen sein könnten. Denn jedes von außen sichtbare Objekt könnte Teil des API der Bibliothek sein. Folglich zeigt die Abbildung weitere Objekte, die als potenzielles API hervorgehoben sind. Schließlich können wir feststellen, dass jede Bibliothek – und an dieser Stelle können wir verallgemeinern und von Modulen sprechen – ein API hat, auch wenn dieses nicht explizit entworfen wurde. Denn durch das Schneiden einer Codebasis in Module entstehen Schnitte, die mit den Mitteln des API-Designs zu gestalten sind.

Funktionen eines API

Ein API hat mehrere unterschiedliche Funktionen, deren man sich beim Entwurf bewusst sein sollte:

- Es definiert die Operationen eines Moduls durch seine Operationen mit Ein- und Ausgabewerten. Aus diesem Grund kann man ein API auch als Vertrag bezeichnen. Der Vertrag umfasst das Verhalten, das eine Implementierung ihren Clients anbietet und garantiert.
- Durch ein API wird ein Client von der Implementierung eines Moduls entkoppelt. Der zuvor erwähnte Vertrag wird mit dem API geschlossen, nicht mit seiner Implementierung, dem Modul.
- Es sollte Software-Wiederverwendung und Integration unterstützen. Darauf kommen wir gleich zurück.
- APIs und Modularisierung sind eng miteinander verknüpft: Um Module in einer Codebasis identifizieren zu können, bedarf es guter Modul-Schnittstellen, die die Module im Sinne des Geheimnis-Prinzips und der Datenkapselung voneinander entkoppeln.

Die genannten Funktionen sind den meisten Entwicklern bekannt, doch es gibt eine weitere Funktion, die häufig vergessen oder zumindest vernachlässigt wird: die Kommunikation. Wann immer eine Software-Entwicklerin oder ein Software-Entwickler Code entwickelt, der von einer anderen Person wiederverwendet werden soll, entsteht ein Kommunikationsproblem. Denn nur in seltenen Fällen kann der Person persönlich erklärt werden, wie der Code aufzurufen ist, um ein bestimmtes Problem zu lösen.

In einem großen Unternehmen oder bei der Entwicklung von Open-Source-Software gibt es unzählige unbekannte Benutzer. Aus diesem Grund muss das API Teil der Lösung dieses Kommunikationsproblems sein. Es sollte als Kommunikationskanal betrachtet werden, denn APIs werden für Menschen geschrieben und die Perspektive der späteren Benutzer ist beim API-Design entscheidend.

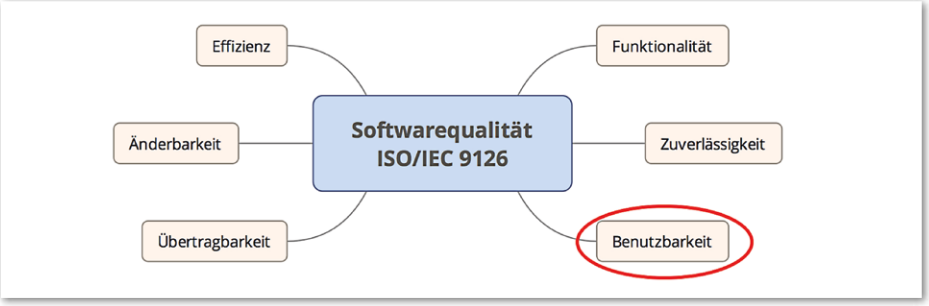


Abbildung 2: Qualitätsmerkmale von Softwareprodukten

Qualitätsziele eines API

Die Qualitätskriterien für Software-Produkte sind in der Norm ISO/IEC 9126 umfänglich beschrieben. Software-Qualität umfasst demnach die Aspekte „Effizienz", „Änderbarkeit", „Übertragbarkeit", „Funktionalität", „Zuverlässigkeit" und „Benutzbarkeit" *(siehe Abbildung 2)*. Was für Software-Produkte im Allgemeinen gilt, gilt auch für APIs. Doch ein Aspekt, die Benutzbarkeit von Software, spielt für API-Designer eine besondere Rolle – vorausgesetzt das API erfüllt eine nützliche Funktion korrekt.

Das Qualitätsmerkmal „Benutzbarkeit" stimmt überein mit der zuvor geforderten Benutzer-Perspektive beim API-Design. Um Entwickler bei der Umsetzung von immer komplexeren Systemen zu unterstützen, bedarf es APIs, die intuitiv verstanden, gelernt und benutzt werden können. Darüber hinaus sollte ein Programmiersprachen-API dabei helfen, sauberen und verständlichen Code zu schreiben.

Benutzbarkeit

Benutzbarkeit ist ein zentrales Qualitätsziel beim API-Design. Ein API sollte aus diesem Grund nicht weniger als konsistent, intuitiv verständlich, dokumentiert, leicht zu lernen und erweiterbar sein. Ein gutes API sollte zusätzlich auch noch lesbaren Code fördern. Ob aber ein API beispielsweise konsistent ist, lässt sich nicht pauschal mit „Ja" oder „Nein" beantworten. Stattdessen sind die genannten Attribute graduelle Größen.

Wie viel Aufwand man in diese Attribute investiert, ist eine Frage der Priorität. Wer beabsichtigt, ein API für viele unbekannte Benutzer anzubieten, wie dies beispielsweise Twitter, PayPal und GitHub tun, dann sollte er in dieses API mehr investieren als in ein inter-

```
java.util.zip.GZIPInputStream
java.util.zip.ZipOutputStream
```

Listing 1

```
java.awt.TextField.setText();
java.awt.Label.setText()
javax.swing.AbstractButton.
setText();
java.awt.Button.setLabel();
java.awt.Frame.setTitle();
```

Listing 2

nes API, das nur wenige Entwickler benutzen. Dennoch ist ein verständliches API auch für Teammitglieder wertvoll, denn es erleichtert die Wiederverwendung existierenden Codes oder die Integration anderer Systeme.

Konsistent

Konzeptionelle Integrität setzt ein kohärentes Design mit der Handschrift eines Architekten voraus. Das heißt nicht, dass es tatsächlich einen Architekten geben muss, der im gesamten Projekt alle Entwurfs-Entscheidungen trifft. Eine Architekten-Rolle ist beispielsweise in Scrum gar nicht definiert, aber das Entwicklungsteam sollte dennoch darauf achten, dass die in dem API verwendeten Begriffe und Konzepte einheitlich Anwendung finden. In der Standardklassenbibliothek von Java findet man beispielsweise folgende Klassennamen *(siehe Listing 1)*. Diese Abweichungen sind sicherlich ärgerlich, aber nicht kritisch. Was ist hingegen mit den Methodennamen in *Listing 2*?

Die API-Designer hätten es uns Benutzern etwas einfacher machen können, wenn sie sich auf den Methodenamen „setText" beschränkt hätten, denn ein API mit weniger Bezeichnungen ist kleiner und einfacher erlernbar. Außerdem stellt sich nun den Benutzern die Frage, ob sich das Verhalten von „setText()", „setLabel" und „setTitle()" unterscheidet. Der Name einer Methode sollte immer möglichst genau und eindeutig das Verhalten der Methode beschreiben. Die unterschiedlichen Methodennamen sind in diesem Fall nicht notwendig. Das bringt uns auch zum Thema „Semantik", denn man muss neben Konsistenz auch auf semantische Genauigkeit achten *(siehe Listing 3)*.

Ruby bietet beispielsweise die Methoden „length", „size" und „count". Man könnte argumentieren, dass dieses API inkonsistent oder zumindest unnötig groß ist, weil man auf zwei der drei Methodennamen verzichten könnte. Es gibt hier jedoch einen semantischen Unterschied und auch bei den Java-Klassen der AWT- und Swing-Packages könnte man vielleicht so argumentieren.

Die Methode „length" gibt die Länge einer Collection oder einer Zeichenkette zurück. Die Antwort erfolgt in konstanter Zeit und ist nicht von der Länge der Collection oder der Zeichenkette abhängig. Die Methode „size" hat die gleiche Funktion und ist tatsächlich nur ein Synonym. Der Methodenname „length" passt jedoch besser zu einer Zeichenkette und „size" besser zu einer Collection. Die Methode „count" verdient ebenfalls einen eigenen Namen, weil hier Objekte traversiert werden. Das Ergebnis ist die Anzahl der Matches. Die unterschiedlichen Namen können demzufolge lesbaren Code fördern, weil die Absicht des Entwicklers besser zum Ausdruck gebracht werden kann.

Intuitiv verständlich

Intuition basiert auf Vorwissen. Dieses Wissen sollte man beim API-Design ausnutzen, um es ihren Benutzern so einfach wie möglich zu machen. Bereits existierende Konventionen, Namen und Konzepte, die ihren Benutzern vermutlich bekannt sind, gilt es wiederzuverwenden. Ein Java-Entwickler würde vermutlich für Datei-Operationen nach „create" und „delete" als Erstes suchen, bei Persistenz-Operationen hingegen nach „insert" und „remove". Namen wie „produce" und „erase" sind für Benutzer vielleicht irritierend. Im Idealfall entwirft man ein API so, dass dessen Client-Code ohne Dokumentation verständlich ist. Man etabliert, falls notwendig, neue starke Begriffe und wendet diese konsequent an. Auch beim Domain-Driven Design (DDD) wird eine ubiquitäre Sprache erarbeitet, die zur Fachlichkeit der Domäne passt.

Dokumentiert

Gute Dokumentation ist unverzichtbar, wenn ein API von möglichst vielen anderen Entwicklern verstanden und erfolgreich benutzt werden können soll. Wer beispielsweise den Konstruktor einer Klasse „private" macht, weil Instanzen der Klasse nur mit einem Builder erzeugt werden sollen, der sollte dies auch dokumentieren.

Angenommen, ein Benutzer möchte die Methode „createBankAccount(AccountInfo info)" eines API aufrufen. Der Benutzer muss für den Aufruf ein Objekt vom Typ „AccountInfo" erzeugen, das sagt ihm die Signatur der Methode. Doch woher bekommt der Benutzer ein Objekt dieses Typs, falls es sich hierbei um ein Interface handelt oder um eine Klasse mit privatem Konstruktor?

In einem Benutzerhandbuch lässt sich mit Beispielen der Gebrauch des API beschreiben. Man kann aber auch direkt mit JavaDoc erklären, wie Objekte dieses Typs erzeugt werden. Am besten versetzt man sich in die Situation des späteren Benutzers dieser Methode. Wie kann der Benutzer ausgehend von dieser Methode alle notwendigen Informationen finden?

Das Ziel „leicht zu lernen" hängt von vielen anderen Faktoren direkt oder indirekt ab. Ein kleines, konsistentes und gut dokumentiertes API ist sicherlich leichter zu lernen als ein großes, inkonsistentes API mit nicht dokumentierten Features. Auch das Vorwissen der Benutzer hat einen großen Einfluss. Deswegen ist es auch so wichtig, möglichst viele Konzepte von anderen bekannten APIs wiederzuverwenden.

Erweiterbar

Erweiterbarkeit wird häufig vom API gefordert, aber welche Änderungen wird es in Zukunft geben? Wer die späteren Änderungen bereits kennt, kann diese Informationen für den Entwurf des API von heute berücksichtigen, doch falls diese Änderungen nicht folgen, waren der Aufwand und das unnötig komplexe Design vergeblich.

Bei Erweiterungen gilt es, auf Abwärtskompatibilität zu achten, sodass neue Versionen auch von existierenden Clients genutzt werden können, ohne dass diese ihren Code anpassen müssen. Falls die Änderung doch inkompatibel ist, sollte der Anpassungsaufwand für existierende Clients minimal sein.

```
arr = [1, 2, 3]
arr.length     # => 3
arr.size       # => 3
arr.count      # => 3
```

Listing 3

```
EntityManager em = ...;
List<Order> list = new
JPAQuery(em)
  .from(QOrder.order)
  .where(order.positions.size().
gt(1))
  .list(order).getResults();
```

Listing 4

Es empfiehlt sich, für ein API mit inkompatiblen Änderungen eine neue API-Version zu vergeben, jedoch mit neuen Versionen sparsam umzugehen. Eine neue Version sollte eine Ausnahme sein, falls abwärtskompatible Änderungen nicht möglich sind. Falls sich sowohl der Code des API als auch der Code der Clients ändern, kann man auf API-Versionierung verzichten und den Code regelmäßig durch Refactorings erweitern und verbessern.

Lesbaren Code fördern

Ein Vorteil leicht lesbaren Codes ist Qualität, denn die Absicht der Entwickler und deren eventuelle Fehler sind in leicht lesbarem Code besser erkennbar. Generell wird Code viel häufiger gelesen als geschrieben. QueryDSL bietet zum Beispiel ein gelungenes API, mit dem typsichere Datenbank-Abfragen formuliert werden können *(siehe Listing 4)*. Diese Beschreibung trifft auch auf das Criteria-API des Java Persistence API (JPA) zu, allerdings ist dessen Client-Code schwerer lesbar. Lesbarkeit ist jedoch nicht alles entscheidend, denn falls scheinbar intuitiv verständlicher Code sich anders verhält, sollte man das API überarbeiten. Ein passendes Beispiel ist das Date-API der Java-Standardklassen-Bibliothek, das Entwicklern viele Jahre Ärger bereitet haben dürfte, weil unter anderem Monate beginnend mit „0" und Tage beginnend mit „1" angegeben werden.

Fazit

Entwickler greifen bei ihrer Arbeit auf eine Vielzahl existierender Bibliotheken, Frameworks und Komponenten zurück. Die Integration und Wiederverwendung erfolgt über APIs, die für diesen Zweck optimiert werden. APIs sind deswegen aus der Perspektive ihrer späteren Benutzer zu beurteilen. Konsistenz, intuitive Verständlichkeit und Dokumentation wurden unter anderem als wichtige Qualitätsziele vorgestellt und Benutzbarkeit wurde für die Qualität von APIs hervorgehoben. Nichtsdestotrotz nützt auch das eleganteste API nichts, falls es nicht hilfreich ist und ausreichend schnell und ausreichend korrekt funktioniert.

Kai Spichale
kai.spichale@sap.com

Kai Spichale beschäftigt sich seit mehr als zehn Jahren leidenschaftlich mit Software-Architekturen von verteilten Systemen und sauberem Code. Nach seinem Studium am Hasso-Plattner-Institut war er unter anderem als Software-Architekt für die adesso AG und als IT-Berater für die innoQ Deutschland GmbH tätig. Als IT Solution Architect arbeitet er heute für SAP SE. Sein technologischer Schwerpunkt liegt auf modernen Architektur-Ansätzen, API-Design und Datenbank-Technologien. Er lebt mit seiner Familie in Berlin.

„Die Community ist ein wichtiger Vorteil von Java …"

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 Java aktuell, sprach darüber mit Michael Simons von der EuregJUG Maas-Rhine.

Wie ist die EuregJUG Maas-Rhine organisiert?

Michael Simons: Die EuregJUG Maas-Rhine wurde von Dr. Stefan Pfeiffer und mir im Frühjahr 2015 gegründet. Im ersten Jahr haben wir gemeinsam Vorträge organisiert, im vergangenen Jahr habe ich mich um Speaker, Räumlichkeiten und Pressearbeit gekümmert. Unsere JUG umfasst mittlerweile 14 feste Mitglieder und ist mit einer Gruppenmitgliedschaft Teil des iJUG e.V. Die engagierten Mitglieder und deren Firmen ermöglichen uns, Vorträge in wechselnden Örtlichkeiten mit Catering anbieten zu können. Wir betreiben unsere eigene Plattform mit Blog, Kalender und RSVP-System. Unser Kalender wird automatisch in den iJUG-Newsletter aufgenommen. Termine und Veranstaltungen kommunizieren wir über unseren Blog, Twitter und einen Newsletter.

Was sind die Ziele der EuregJUG Maas-Rhine?

Michael Simons: Die EuregJUG ist im Dreiländereck Niederlande, Belgien und Deutschland als grenzüberschreitende Java-User-Gruppe konzipiert. Wir möchten mit überwiegend englischsprachigen Vorträgen nicht nur Besucher aus Aachen, sondern auch aus den angrenzenden Städten erreichen. In den vergangenen eineinhalb Jahren ist es gelungen, eine gut vernetzte Gruppe lokaler Java-Enthusiasten zusammenzubringen. Java-Enthusiast bedeutet für uns nicht, dass wir alle „Rockstar-Entwickler" sind, sondern normale Entwickler, die sich auch außerhalb ihres Arbeitsumfeldes engagieren möchten. Sich während eines JUG-Events in eine Runde von Experten zu stellen, die „Champions" oder „Rockstars" sind und mit der gleichen Sicherheit wie die Experten mit den Begrifflichkeiten zu jonglieren, erfordert manchmal Mut. Wir hoffen, mit unseren Events einen Raum dafür bieten zu können, diese Kluft zu schließen.

Wie viele Veranstaltungen gibt es pro Jahr?

Michael Simons: Wir konnten unser ursprüngliches Ziel, einen Vortrag pro Quartal anzubieten, einhalten. Die Vorträge wurden ausnahmslos gut angenommen und wir hatten im Schnitt zwanzig Besucher an wechselnden Orten. Bemerkenswert waren die Peaks von ungefähr jeweils fünfzig Besuchern bei nicht-technischen Vorträgen zu Themen wie „Technische Schulden" oder zur systematischen Verbesserung von Software.

Zur Person:
Michael Simons

Michael Simons ist Software-Architekt (CPSA-F) bei ENERKO Informatik in Aachen und entwickelt dort GIS-, EDM- und Vertriebsmanagement-Systeme für Stromnetz-Betreiber und Energie-Lieferanten. Er ist Mitglied des NetBeans-Dream-Teams und Gründer der Euregio JUG; er schreibt in seinem Blog „info.michael-simons.eu" über Java, Spring und Software-Architektur.

Was bedeutet Java für euch?

Michael Simons: Java ist im Jahr 2015 zwanzig Jahre alt geworden, in unserer Branche mittlerweile so etwas wie ein Dinosaurier. Java wird behutsam weiterentwickelt, große Neuerungen finden sicherlich in anderen Sprachen statt. Allerdings ermöglichen die in Java 8 umgesetzten Erweiterungen und Neuerungen vielen Entwicklern einen einfacheren Zugang zu Konzepten wie funktionaler Programmierung. Konzepte wie reaktive Programmierung und Aktoren sind neben Sprach-Features ein weiterer wichtiger Evolutionsschritt für die Software-Entwicklung. Auch für sie gibt es Lösungen im Java-Umfeld. Mit Java kann eine Vielzahl von Aufgaben ordentlich erledigt, offen und Plattform-unabhängig bearbeitet werden. Die Plattform ist etabliert und weder reiner Hype noch Legacy.

Welchen Stellenwert besitzt die Java-Community für euch?

Michael Simons: Die Java-Community ist für uns ein wichtiger Vorteil von Java: Wir nehmen die Java-Community als offene und freundliche Gemeinschaft wahr. Durch Austausch über Technik und Fachlichkeit wird ein Blick über den Tellerrand ermöglicht. Zudem nimmt sich die Community der Themen an, die von Oracle zurückhaltend behandelt werden, und bietet Lösungen an, die (noch) nicht Teil eines Standards sind. Dabei ist die Bereitschaft, Wissen und Ergebnisse in Form von Software zu teilen und oftmals frei verfüg- und nutzbar zu machen, außergewöhnlich und Teil des Erfolges der Plattform.

Wie sollte sich Java weiterentwickeln?

Michael Simons: Java als Sprache sollte weiterhin mit Bedacht weiterentwickelt werden. Einige Altlasten können sicherlich zurückgelassen werden, Abwärtskompatibilität als Erfolgsfaktor ist dennoch wichtig. Vereinfachungen, wie zum Beispiel durch das Project Coin oder auch Type Inference, sind sicherlich Dinge, die – von außen betrachtet – Java das Stigma „schwerfällig" oder „legacy" nehmen könnten.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?

Michael Simons: Oracle sollte auf der einen Seite seine kommerziellen Interessen klar benennen und bekennen, nicht nur implizit. Dabei gleichzeitig auch die Wichtigkeit der Sprache Java und der Plattform erkennen und diese sowie die damit verbundenen Prozesse offener gestalten. Unter der Annahme, dass Oracle Strategien bezüglich Java, der Plattform und assoziierter Produkte hat, sollte diese kommuniziert werden. Unsicherheit und mangelnde Kommunikation ist zu oft ein Grund, warum Projekte scheitern.

Wie sollte sich die Community gegenüber Oracle verhalten?

Michael Simons: Ruhe bewahren sowie die Weiterentwicklung der Sprache und Plattform unterstützen sind genauso wichtig wie das Einfordern obiger Transparenz. Ob offene Briefe eine Auswirkung haben, kann ich nicht sagen.

Michael Simons
info@euregjug.eu
http://www.euregjug.eu

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e.V.
www.doag.org

Java User Group Stuttgart e.V. (JUGS)
www.jugs.de

Java User Group Köln
www.jugcologne.eu

Java User Group Darmstadt
http://jug-da.de

Java User Group München (JUGM)
www.jugm.de

Java User Group Nürnberg
www.meetup.com/de-DE/JUG-Nurnberg

Java User Group Ostfalen
www.jug-ostfalen.de

Java User Group Saxony
www.jugsaxony.org

Sun User Group Deutschland e.V.
www.sugd.de

Swiss Oracle User Group (SOUG)
www.soug.ch

Berlin Expert Days e.V.
www.bed-con.org

Java Student User Group Wien
www.jsug.at

Java User Group Karlsruhe
http://jug-karlsruhe.mixxt.de

Java User Group Hannover
www.jug-h.de

Java User Group Augsburg
www.jug-augsburg.de

Java User Group Bremen
www.jugbremen.de

Java User Group Münster
www.jug-muenster.de

Java User Group Hessen
www.jugh.de

Java User Group Dortmund
www.jugdo.de

Java User Group Hamburg
www.jughh.de

Java User Group Berlin-Brandenburg
www.jug-berlin-brandenburg.de

Java User Group Kaiserslautern
www.jug-kl.de

Java User Group Switzerland
www.jug.ch

Java User Group Euregio Maas-Rhine
www.euregjug.eu

Java User Group Görlitz
www.jug-gr.de

Java User Group Mannheim
www.majug.de

Lightweight Java User Group München
www.meetup.com/de/lightweight-java-user-group-munchen

Java User Group Düsseldorf rheinjug
www.rheinjug.de

Java User Group Goldstadt
https://gitlab.com/groups/jugpf

Java User Group Bielefeld
www.meetup.com/de-DE/Java-User-Group-Bielefeld

Java User Group Paderborn
http://jug-pb.gitlab.io/

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.

iJUG Verbund www.ijug.eu

Impressum

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

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

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

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

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

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

Fotonachweis:
Titel: © BillionPhotos.com/Fotolia
Foto S. 08 © Andriy Popov/123RF
Foto S. 11 © Swift/https://swift.org
Foto S. 16 © everythingpossible/123RF
Foto S. 21 © Galen Framework/
http://galenframework.com
Foto S. 26 © Thymeleaf/www.thymeleaf.org
Foto S. 31 © syaraku/123RF
Foto S. 36 © aimage/123RF
Foto S. 39 © Kirill Makarov/123RF
Foto S. 42 © Sergey Nivens/123RF
Foto S. 46 © Sean Gladwell/Fotolia
Foto S. 51 © Redis/https://redis.io
Foto S. 57 © Douglas Crockford/
http://clipartist.info
Foto S. 61 © greyjj/123RF
Foto S. 64 © venimo/123RF

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

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

Inserentenverzeichnis


Save the Date
2017
DOAG
Konferenz + Ausstellung
21. - 24. November in Nürnberg