Ausgabe 03/2017

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

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

4 191978 304903 03

Java aktuell

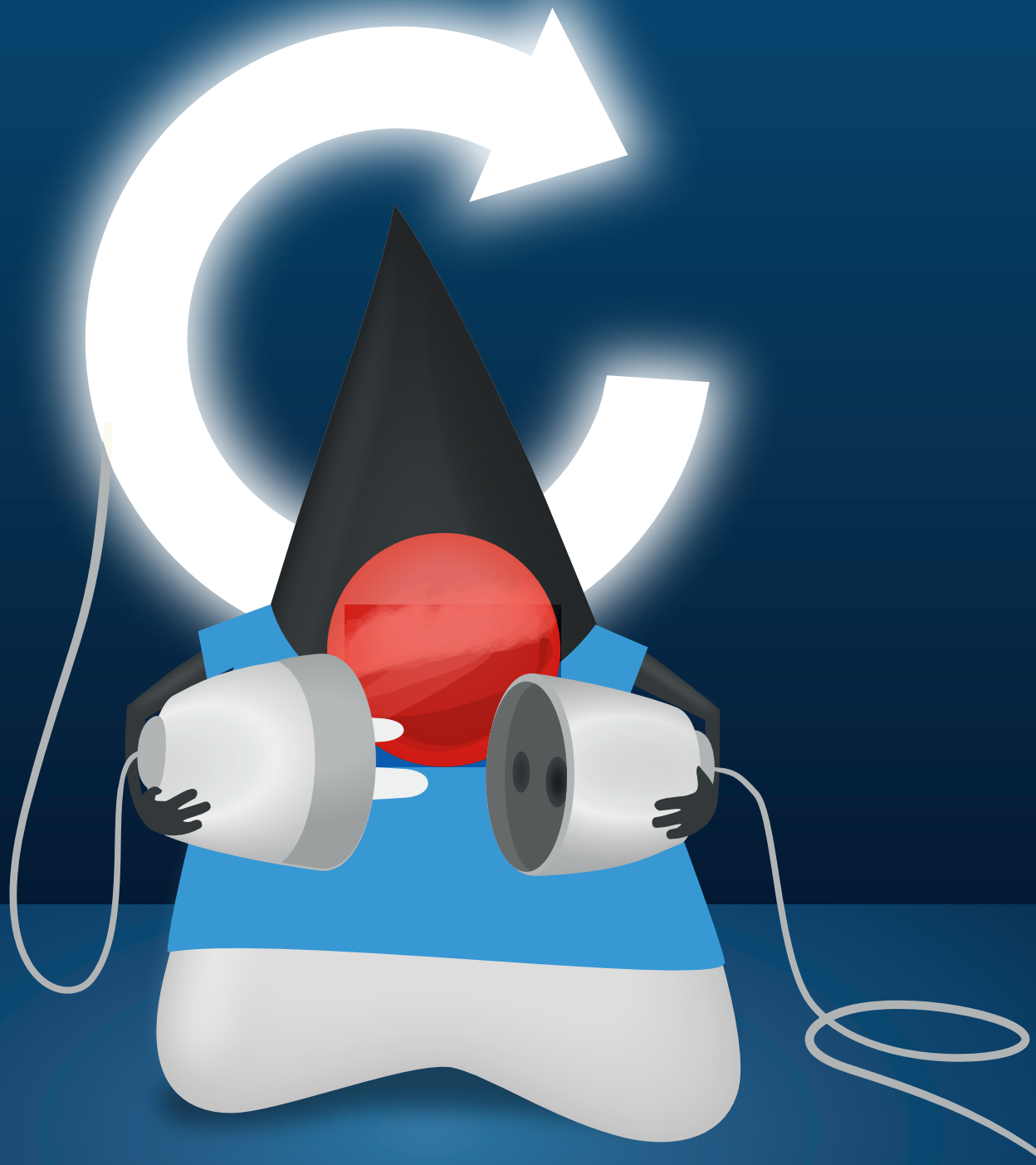
iJUG Verbund
www.ijug.eu

Java 9
Modulare Anwendungen halten Einzug

Domain-Driven Design
Microservices mit hoher Qualität entwickeln

AsciiDoctor
Eine lebendige Dokumentation erstellen

Java aktuell reloaded



> Viele spannende Projekte auf Grundlage moderner Technologien.

> Auch wenn's mal hakt, wird ehrlich und geradeaus kommuniziert, ohne unfair oder persönlich zu werden. So stelle ich mir gute Teamarbeit vor!

Wir gehören schon zum Team. Es gibt viele gute Gründe, weshalb auch du zu uns kommen solltest!

> Es fällt schwer, sich einen besseren Job vorzustellen.

> Super Teamwork – hier hilft jeder jedem.

> Fantastisch ist die Zusammenarbeit mit unglaublich vielen hochtalentierten Leuten.

> Eine der besten Firmen, die ich bisher kennengelernt habe.

> Endlich wieder Spaß an der Arbeit.

An zehn Standorten formt die codecentric AG mit Leidenschaft für Technologien und dem Expertenwissen unserer Mitarbeiter aktiv die IT-Landschaft Deutschlands. Das gelingt uns durch den hohen Qualitätsanspruch an die eigene Arbeit und unsere freundschaftliche Zusammenarbeit auf Augenhöhe. Nicht umsonst nennt uns kununu: "Top Arbeitgeber der IT Branche"!

Du willst auch Teil des Teams sein?

Dein Weg beginnt hier: codecentric.de/karriere-info

codecentric

codecentric AG | Hochstraße 11 | 42697 Solingen

kununu TOP COMPANY – VON MITARBEITERN EMPFOHLEN!

kununu OPEN COMPANY – WIR LEBEN OFFENHEIT!

FAIR company 2017 karriere.de

Die Java aktuell im neuen Gewand

Ich habe es nachgezählt: Dies ist die 27. Ausgabe der Java aktuell, seit Oracle Sun Microsystems übernommen hat. Die erste Java aktuell erschien im September 2010 mit dem Titelslogan „Oracle und die Zukunft von Java". Die Verunsicherung in der Java-Community war groß; keiner wusste, wie es mit dem Java-Biotop weitergehen wird. Sogar Abspaltungen vom Java-Standard waren im Gespräch. Mit dem Satz „The Sun we knew is gone" brachte der Blogger John Gruber die Stimmung, die damals in der Java-Welt herrschte, auf den Punkt.

Vieles hat sich seither verändert. Oracle hat es zunächst geschafft, durch klare Roadmaps und die Veröffentlichung neuer Java-Versionen das Vertrauen der Community zu gewinnen. Jetzt ist der Hersteller auf dem besten Weg, dieses wieder zu verspielen, wie der aktuelle Umgang mit Java EE eindrucksvoll zeigt.

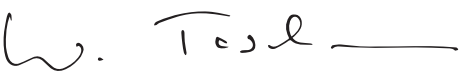
Ich schrieb damals auch, dass sich acht Java User Groups aus Deutschland gemeinsam mit der DOAG Deutsche ORACLE-Anwendergruppe e.V. im Interessenverbund der Java User Groups e.V. (iJUG) vereint haben mit dem Ziel, die gemeinsamen Interessen der Java-Anwendergruppen sowie der Java-Anwender im deutschsprachigen Raum, insbesondere gegenüber Entwicklern, Herstellern, Vertriebsunternehmen sowie der Öffentlichkeit, umfassend zu vertreten. Mittlerweile hat der iJUG 32 User Groups aus Deutschland, Österreich und der Schweiz unter seinem Dach vereint.

Einige dieser User Groups organisieren seit etlichen Jahren bekannte Veranstaltungen – etwa das Java Forum Stuttgart, das Java Forum Nord, die Berlin Expert Days oder das JUG Saxony Camp. Mit der JavaLand, die in Kooperation mit Heise Medien, der DOAG und dem iJUG organisiert wird und die im Jahr 2014 mit 800 Java-Begeisterten ihren Auftakt feierte, ist eine weitere hochkarätige Veranstaltung hinzugekommen. Die vierte Auflage in diesem Jahr brachte 1.655 Java-Entwickler zusammen.

Auch die Java aktuell hat sich weiterentwickelt. In diesem Jahr sind es zum ersten Mal fünf Ausgaben und ab dem Jahr 2018 bringen wir sechs Ausgaben im Jahr heraus. Aus diesem Anlass haben wir auch das Layout überarbeitet.

Ich wünsche Ihnen viel Spaß beim Lesen dieser Zeitschrift und freue mich wie immer auf Ihr Feedback an redaktion@ijug.eu.

Ihr



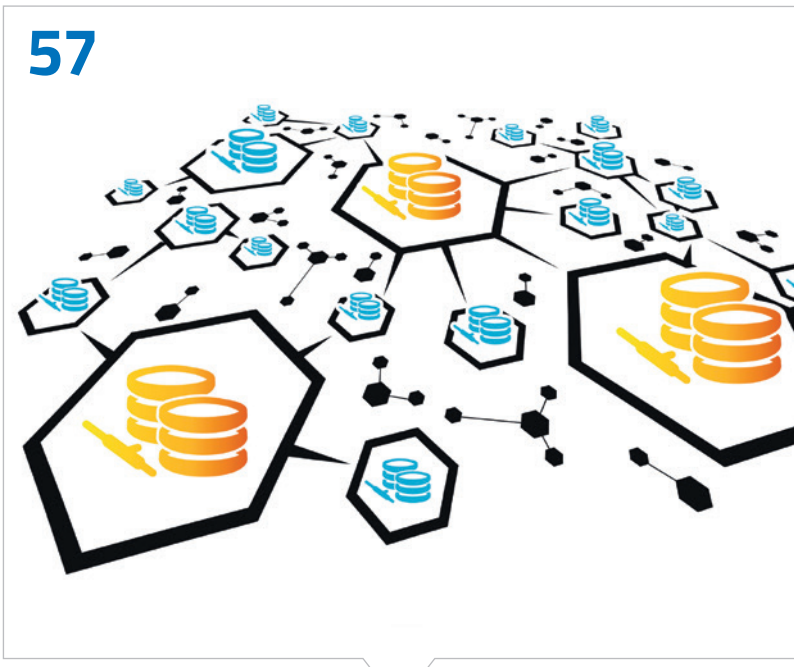
Wolfgang Taschner
Chefredakteur Java aktuell

Inhalt

Die Anwendungslogs der Identitäts- und Zugriffs-Management-systeme einer Web-Anwendung liefern zentrale Informationen über Zugriffsprobleme oder Angriffe

Beim Bau der Module ist zu beachten, an welche anderen Module diese gekoppelt werden und wie die Verbindungsstellen zwischen den Modulen aussehen müssen

„Serverless" heißt keinesfalls, dass Programmcode nicht mehr auf Servern ausgeführt wird, vielmehr geht es um das Management von Servern



Bei Datenbank-Administratoren haben Java-Anwendungen den Ruf schlechter Performance und seltsamer Datenbank-Zugriffe, die viel zu selten behoben werden

Das Java-Tagebuch

Andreas Badelt, stellv. 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 ersten Quartal 2017.

1. Februar 2017

Java EE 8 – aktueller Status

David Delabassee, der sich jetzt um die Kommunikation mit der Community rund um Java EE 8 kümmert, gibt in seinem Blog monatliche Updates zum Status. Im Januar gab es unter anderem große Fortschritte bei JAX-RS (konkrete Vorschläge für das Reactive-Client-API und SSE liegen vor) und Servlet 4.0 (wie Server Push). CDI 2.0, JSF 2.3 und JSON-P 1.1 haben den „Public Draft" veröffentlicht, sie liegen also voll im Zeitplan. Außerdem: Der „Transfer Ballot" für MVC 1.0 ist einstimmig angenommen, der JSR geht also an die Community beziehungsweise an Ivar Grimstad als Spec Lead – wenn die ganzen rechtlichen Details geklärt sind (was bei der großen Oracle-Rechtsabteilung noch etwas dauern kann).

blogs.oracle.com/theaquarium

1. Februar 2017

Kontrolle über JDK Interfaces: Oracle will mehr Transparenz

Mark Reinhold, Java-Plattform-Architekt bei Oracle, hat über die JDK-Mailingliste einen Vorschlag eingereicht, um die Kontrolle über die Java-Interfaces transparenter zu gestalten. „Since the dawn of time (i.e. 1997)" gibt es das CCC, ein internes Gremium bei Sun und jetzt bei Oracle, das alle Änderungen an APIs, Kommandozeilen-Parametern etc. überprüfen und ablehnen kann. Dieses Gremium soll nun durch das neue „Compatibility & Specification Review" (CSR) ersetzt werden, das transparenter arbeiten und gegenüber der OpenJDK-Community rechenschaftspflichtig sein soll. Die Entscheidungen des Gremiums sollen nicht mehr von sich aus verbindlich sein, aber die Leiter jedes OpenJDK-Projekts sollen es dazu freiwillig einbinden können – was Mark Reinhold für sein JDK-Release-Projekt schon mal ankündigt.

http://mail.openjdk.java.net/pipermail/gb-discuss/2017-January/000320.html

14. Februar 2017

Oracle gegen Google, nächste Runde

Oracle hat erwartungsgemäß beim „US Court of Appeals" Berufung gegen das Urteil aus dem Jahr 2016 eingelegt, in dem die Verwendung der Java-APIs in Android durch Google von einer Jury als „fair use" bezeichnet wurde. Die Jury hatte damit Schadenersatzansprüche von Oracle in Milliardenhöhe abgeschmettert. Ob die Berufung überhaupt zur Verhandlung angenommen wird, muss allerdings vom Gericht noch entschieden werden.

https://www.theregister.co.uk/2017/02/11/oracle_refuses_to_let_java_suit_die/

14. Februar 2017

NetBeans und Apache

Oracle-Product-Manager Geertjan Wielenga schreibt in seinem Blog über den Fortschritt der NetBeans-Übergabe an die Apache Software Foundation. Fazit: Es wurden große Fortschritte erzielt, trotzdem sind noch viele Dinge zu erledigen. So prüft Oracle die zu übergebenden Repositories gerade nach Inhalten, die nicht an Apache freigegeben werden können, etwa weil sie Oracle gar nicht gehören (was im Laufe von zwanzig Jahren sicher passieren kann). Ziel ist, mit dem Release von Java 9 auch ein Apache NetBeans 9 freizugeben.

https://blogs.oracle.com/geertjan/entry/welcome_to_me

22. Februar 2017

Java-EE-8-Zeitplan aktualisiert

Linda DeMichiel, Specification Lead für den Java-EE-8-Umbrella, verkündet in der Mailing-Liste des JSR-366, dass EE 8 praktisch zeitgleich mit SE 9 im Juli 2017 freigegeben werden soll. Ein aggressiver Zeitplan, aber der Fortschritt der meisten JSRs scheint es zu ermöglichen. Wobei natürlich festgestellt werden muss, dass EE 8 kein wirklich umfangreiches Release ist und mit dem Stoppen und Neustarten noch weitere Features fallen gelassen wurden. Alle großen Wünsche sind für EE 9 im Jahr 2018 geplant – zumindest bis zum Eintritt der Realität.

https://blogs.oracle.com/theaquarium/entry/javaone_replay_java_ee_8

28. Februar 2017

Java EE 8 – Status im Februar

Ein neuer Monat, ein neues Update von David Delabassee in den Oracle Blogs: So hat Bean Validation 2.0 einen ersten vollständigen Entwurf der Spezifikation veröffentlicht („Early Draft") und bei JAX-RS wurden noch umfangreichere Änderungen am Server-Sent-Events-API vorgenommen.

https://twitter.com/delabassee/status/823431237595435008

7. März 2017

Java-SE-9-Spezifikation zum Review

Die „Public Review Specification" von Java SE 9 ist veröffentlicht worden. Sie umfasst die fertige Spezifikation und die darin umgesetzten JSRs aus der Standardisierung sowie die im OpenJDK-Projekt selber definierten „JDK Enhancement Proposals".

http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html

15. März 2017

Scala hat verbesserten AOT-Compiler

Scala hat einen verbesserten Ahead-of-time-Compiler auf Basis der Compiler-Infrastruktur LLVM erhalten. Die Motivation ist insbesondere eine bessere Einsetzbarkeit auf kleinen Geräten.

https://www.scala-lang.org/blog/2017/03/14/scala-native-0.1-is-here.html

20. März 2017

GlassFish für Java EE 8

Der erste „promoted build" im Jahr 2017 für GlassFish 5, der die annähernd fertigen Spezifikationen zu Java EE 8 abdecken soll, steht zum Download zur Verfügung: GlassFish-5.0-b03. Weitere „promoted builds" sollten etwa wöchentlich erfolgen. Rund sechs Wochen nach EE 8 soll dann auch GlassFish 5 die offizielle Freigabe erhalten – dem Plan nach also Ende August oder im September.

https://glassfish.java.net

22. März 2017

Jigsaw: Unerlaubter Zugriff wird weiter geduldet

Java 9 hat laut Plattform-Architekt Mark Reinhold jetzt einen „big kill switch" erhalten. Eigentlich werden Zugriffe per Reflection auf geschützten Code in anderen Modulen mit dem JDK 9 unterbunden. Genauer: Zugriffe aus dem „Unnamed Module" (also Klassen und „.jar"-Files im Classpath) auf Code in anderen („named") Modulen. Da dies wohl zu Problemen mit einer Vielzahl von Bibliotheken führt (es wird offensichtlich massiv gepfuscht – oder gezaubert, je nach Standpunkt), soll der Schalter „–permit-illegal-access" das Release-Datum retten, ohne die Adoption von Java 9 generell signifikant zu reduzieren. Trotz Setzen des Schalters erzeugen die illegalen Zugriffe jedoch massiv Runtime-Warnungen, um die Entwickler und Betreiber auf das Problem hinzuweisen. Mit Java 10 soll der Schalter nämlich wieder verschwinden – es handelt sich also nur um eine Gnadenfrist.

jaxenter.de/java-9-kill-switch-interview-55850

22. März 2017

IntelliJ IDEA für Java 9

Die Version 2017.1 der Entwicklungsumgebung IntelliJ IDEA ist herausgekommen. Sie unterstützt insbesondere schon das für Juli angekündigte Java 9, inklusive Modularisierung mit Jigsaw.

https://blog.jetbrains.com/idea/2017/03/support-for-java-9-modules-in-intellij-idea-2017-1

26. März 2017

JDK 9: Jigsaw-Projekte in NetBeans

Auch NetBeans unterstützt inzwischen Java 9, allerdings erst im „Development Build" für NetBeans 9. Dieser eher instabile nächtliche Build bietet unter anderem eine JShell-Integration sowie IDE-Unterstützung für Projekte aus mehreren Jigsaw-Modulen, die gemeinsam kompiliert werden sollen. Aber auch hier sind gegebenenfalls die Folgen des erweiterten Schutzes vor illegalen Zugriffen in andere Jigsaw-Module zu spüren, Stichwort: „setAccessible()".

http://wiki.netbeans.org/JDK9Support

10. April 2017

JCP: „Special Election" zum Executive Committee

Wie im Jahr 2016 bereits vom JCP angekündigt, finden im April Nachwahlen zum Executive Committee für zwei „ratified seats" statt – Oracle schlägt vor und die JCP-Member stimmen zu oder lehnen ab. ARM und JetBrains sollen Ericsson und TOTVS ersetzen, die schon im Jahr 2016 von ihrem Sitz zurückgetreten beziehungsweise nicht mehr zu einer neuen Wahl angetreten waren. Am 13. April werden sich die Kandidaten per Telefonkonferenz vorstellen. Ganz unerfahren mit dem JCP sind aber beide nicht, ARM war in der Vergangenheit schon einmal über die frei gewählten Sitze im EC vertreten, JetBrains hatte vor vielen Jahren zumindest schon mal kandidiert.

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

21. April 2017

MVC offiziell übergeben

Jetzt ist es offiziell – soeben hat David Delabassee in seinem Tweet geschrieben: Ivar Grimstad und Oracle haben die rechtlichen Details der Übergabe von MVC 1.0 geklärt (inklusive des „closed-source" Test Compatibility Kit, das vermutlich der wesentliche Knackpunkt war). Der JSR ist damit in seine Hände übergegangen.

https://twitter.com/delabassee?lang=de

Andreas Badelt
stellv. 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.

Taschenspielertricks

Markus Karg, Java User Group Goldstadt

Das englische Wort „Prestidigitation“ (zu Deutsch „Taschenspielertricks“) bezeichnet die feinmotorischen und psychologischen Fähigkeiten eines Gauklers, durch gezielte Manipulation eine glaubhafte Illusion zu erschaffen, also alternative Fakten als Realität zu verkaufen. Anwendbar sind diese Fähigkeiten aber auch in der Unternehmenskommunikation.

Nach einer langen Zeit des „lauten Schweigens", um einen renommierten Oracle-Mitarbeiter und JSF Spec Lead zu zitieren, hat sich Oracle Ende vergangenen Jahres überraschend zu Java EE 8 „committed" und ohne vorherige Konsultation des Java Community Process (JCP) eine stark abgespeckte Roadmap für die genannte Norm präsentiert. In einer nachfolgenden Umfrage wurde die Community gebeten, sich zu diesem Vorhaben zu äußern.

Offenbar war das Ergebnis für den aktuellen Java-Steward derart unerquicklich, dass es überraschend lange dauerte, bis der Community das Ergebnis präsentiert wurde – dafür inklusive obskurer Schlussfolgerungen. Aus der Oracle-Sicht bestätigt das Ergebnis angeblich die vorgestellte Roadmap, obgleich kurz darauf weiter gekürzt wurde.

Die Community steht nicht unbedingt geschlossen hinter dieser Einschätzung und fühlt sich aufgrund der seltsamen Interpretation teilweise veräppelt: Getreu dem Motto „Traue keiner Statistik, die du nicht selbst gefälscht hast" wurde das tatsächliche Umfrage-Ergebnis so stark in Richtung der vorab gewünschten Ziele verbogen, dass die Studie eher der Kategorie „Belletristik" zuzuordnen ist.

Allein das Präsentieren einer Roadmap für einen JCP-Standard am JCP EC vorbei ist bereits eine Farce, denn nur das demokratisch gewählte EC des JCP hat offiziell die Befugnis, über Wohl und Wehe von Java-Normen – so auch von Java EE – zu entscheiden. Oracle hat dieses Faktum offenbar gezielt ignoriert, möglicherweise aus politischen Gründen: Motiv eines solchen Vorgehens könnte sein, die (unter anderem auch dank großer Vereinigungen wie iJUG, SouJava und LJC) zunehmend mächtigere User Community zu spalten (etwa in pro-Oracle vs. Java EE Guardians vs. microprofile.io vs. sonstige), große Konkurrenten wie IBM und SAP kaltzustellen, indem ihr Mitspracherecht ins Leere läuft und sie somit noch stärker zur direkten Verhandlung (auch um Lizenzgebühren) mit Oracle gezwungen werden, oder das EC in seiner Gesamtheit der Lächerlichkeit preiszugeben und somit Java EE von einem internationalen Standard zu einem reinen Oracle-Produkt umzumünzen, um letztendlich die alleinige Kontrolle über dieses Produkt (und vor allem dessen Entwicklungskosten und Umsatz-Chancen) zu haben. Dass die Community das nicht hinnimmt, hat sich die strategische Ebene des Weltkonzerns wohl nicht ausmalen können. Umso panischer erscheinen sowohl die praktisch inhaltsleere Präsentation der Java-EE-Roadmap auf der vergangenen JavaOne als auch die hektisch zusammengestrichenen JSRs von Java EE 8 und die fast schon scheinheilig anmutende, zweite Umfrage Ende des Jahres.

Letztendlich wusste Oracle sehr genau, was die Community in Java EE 8 sehen will. Hierzu wurde schließlich vom gleichen Konzern bereits im Jahr 2014 eine Umfrage gemacht *(siehe „https://blogs.oracle.com/ldemichiel/entry/results_from_the_java_ee")*, in der beispielsweise das MVC-API, aber auch abstrakte Themen wie Cloud-Fähigkeit (u. a. also auch Microservices und Multi-Tenancy) als wichtig begründet wurden. Ebenso hat das JCP EC seit Langem eine klare Vision, wohin der Standard sich entwickeln soll. Daher gab es im Jahr 2014 mit dem JSR 366 eine eindeutig definierte Charta *(siehe „https://jcp.org/en/jsr/detail?id=366#2")* dazu, welcher Inhalt mit Java EE 8 abzuliefern ist.

Machtverteilung im Java-Universum

Man stelle sich vor, per Grundgesetz stellt eine einzige, immer gleiche Partei den gesamten Regierungsapparat, vom kleinen Kommunalbeamten bis hinauf zur Kanzlerin. Wahlrecht besitzt kein einziger Bürger, nur Unternehmen. Doch selbst jene dürfen nur das Parlament wählen, während die Regierung auf Ewigkeit von der immer gleichen Partei gestellt wird. Darüber hinaus wird die Wahl immer von dieser einen Partei finanziert, durchgeführt und ausgewertet. Das Parlament darf diskutieren und Gesetze erlassen. Daran halten muss sich allerdings niemand, denn es gibt keine Justiz, keine Strafen und keine Verpflichtung der Regierung, exakt die vom Parlament beschlossenen Gesetze umzusetzen. In einer Demokratie undenkbar, gerade vor dem Hintergrund der deutschen Historie. In der Wirtschaft hingegen Alltag: Exakt so funktioniert das Java-Universum, und die Partei heißt Oracle!

Im Java-Ökosystem gibt es viele Stakeholder – größere und kleinere. Laut Satzung des JCP sind sie alle nahezu gleich mächtig. Es gibt jedoch Ausnahmen. „Normale" Mitglieder besitzen seit einiger Zeit keinerlei Macht mehr, da sie ihr Wahlrecht verloren haben. Zudem hat Oracle mit seinem Verhalten im Jahr 2016 allen anderen Stakeholdern klar demonstriert, praktisch unbegrenzte Macht zu besitzen – technisch zumindest. Als Inhaber der Marke „Java" und als größter Contributor und Finanzier im Java-Universum konnte Oracle (und kann es auch weiterhin) Fakten schaffen, wie es dem Unternehmen gefällt – solange die Kundschaft mitzieht. Der JCP und vor allem das Executive Committee (EC) kann daran nichts ändern, denn ohne die Einwilligung der Oracle-Anwälte kann niemand anderes auf der Welt das Wort „Java" auf ein Software-Produkt schreiben. Somit ist das JCP ein zahnloser Tiger – dessen Personal zudem weitestgehend auf der Oracle-Payroll steht.

Doch Oracle hatte sich wirtschaftlich und produktpolitisch inzwischen umentschieden und wollte nicht mehr viele Millionen in einen Standard pumpen, den die Anwender zwar goutieren, der aber nicht das Maximum an Profit generiert. Vielmehr zog man augenscheinlich Personal aus den Bereichen „EJB", „JSF", „JAX-RS", „JavaFX" und weiteren ab, um das Geld lieber in Themen wie „JavaScript" (Nashorn, Avatar etc.) zu pumpen – die dann, Ironie der Geschichte (oder absehbar), der Java-Community herzlich egal waren. Wer nämlich JavaScript will, greift lieber zu Node.js und Angular und ist zudem vermutlich eher selten Kunde der Oracle-Java-Abteilung. Von daher ist verwunderlich, dass Oracle auf der einen Seite auf der letzten JavaOne behauptete, hinter Java EE 8 zu stehen und bereits eine klare Vision zu haben, was zu streichen sei (MVC nämlich ebenso wie die Cloud-Fähigkeit), auf der anderen Seite dann jedoch eine Umfrage starten muss, um heraus-

zufinden, was inhaltlich überhaupt Java EE 8 einmal werden soll. Gleichzeitig wird eine nur wenige Monate in der Zukunft liegende Deadline gesetzt, die in höchstem Maße daran zweifeln lässt, dass irgendwelche Änderungen, unter Beachtung der Fristen des JCP-Release-Modells, bis dahin überhaupt noch möglich erscheinen. Die Quintessenz des Commitments liest sich daher eher so, dass Oracle bis zu Java EE 8 das eigene Commitment und den selbst gewählten Inhalt anscheinend in Frage stellte (Wozu sonst ein Commitment verkünden während eines angeblich gut laufenden Projekts?) und bis zur Umfrage dann anscheinend vergessen hatte, was der Inhalt von Java EE 8 sein soll und wie der JCP funktioniert – was sicherlich niemand ernsthaft behaupten will.

Insofern bleibt nur ein Schluss: Die Umfrage Ende 2016 war nur Augenwischerei und sollte irgendwie begründen, wie es zu einer Roadmap kommen kann, die dem JSR 366 nicht entspricht und die dem JCP nicht vor der Präsentation auf der JavaOne vorgelegt wurde. Praktisch „im besten Sinne für die Community" – wie fürsorglich! Schade nur, dass die Community wachsam war und den Taschenspielertrick erkannt hat: Um den Anschein zu wahren, hätte Oracle auf der JavaOne zumindest behaupten müssen, dass die neue Roadmap ein Vorschlag ist und dem JCP selbstverständlich zur Abstimmung vorgelegt wird, da man den mit JSR 366 übernommenen Verpflichtungen absichtlich nicht nachgekommen ist, weil man selbst inzwischen nicht mehr an Java EE glaubt und lieber mit dem Projekt Avatar gespielt hat. Hat Oracle so aber eben nicht gesagt und umso mehr die Glaubwürdigkeit des Commitments infrage gestellt. Daran hat auch die Interpretation des Umfrage-Ergebnisses nichts mehr geändert – im Gegenteil, wie die nachfolgenden Ausführungen zeigen.

Fragwürde Interpretation der Umfrage

Man muss schon eine Weile suchen, um die Oracle-Interpretation der Java EE Survey zu finden. Sie steht unter *„https://java.net/downloads/javaee-spec/Java%20EE%20Survey%20Results%20December%202016.pdf".* Vielleicht ist es ja aber auch Absicht, dass auf der Landing-Page *„java.net"* dieses PDF weder verlinkt noch über die dortige Suchmaschine zu finden ist. Jedenfalls ist es wenig überraschend, dass Oracle in diesem Dokument natürlich seine vorab zusammengestrichene Roadmap ganz klar durch das Umfrage-Ergebnis bestätigt sieht. Kritischen Lesern werden dagegen einige Ungereimtheiten auffallen: Üblicherweise wird ein Wahlergebnis am Folgetag veröffentlicht, und zwar vollständig und nicht interpretiert. Oracle hat dazu – im Zeitalter von Big-Data-Analyse-Werkzeugen – mehrere Wochen benötigt, der Inhalt ist unvollständig und das Dokument ist eher eine subjektive Interpretation als ein nüchternes Ergebnis. Die tatsächlich gestellten Fragen erscheinen erst ab Seite neun, während der Autor jedoch bereits auf Seite fünf seine eigene Interpretation in Form eines „Wichtigkeits-Rankings" aufzeigt, das zudem gar nicht alle Java-EE-Technologien enthält, sondern nur eine vom Autor der Studie nach eigenem Gutdünken erstellte Selektion darstellt. Ein Zusammenhang zu den tatsächlich gestellten Fragen ist unter dem abstrakten Gesichtspunkt „Wichtigkeit" allerdings nicht nachvollziehbar, ebenso wie die Roadmap sich aus der Umfrage ableiten lässt.

Die Seiten sieben und acht zeigen trotzdem bereits „Conclusions", also vom Autor subjektiv gezogene Schlüsse, ohne darzulegen, wie diese auf das tatsächliche Umfrage-Ergebnis passen sollen. Im Gegenteil, es wird sogar zugegeben, dass die Schlüsse zu CDI 2.0, Bean Validation 2.0 und JSF 2.3 gar nicht erst in Zusammenhang mit der Studie stehen. Und bei einigen Punkten wird erst gar nicht der Versuch unternommen, die Oracle-Entscheidungen überhaupt in irgend einer Form zu erklären: OAUTH beispielsweise ist aus der Agenda verschwunden, obwohl die Studie selbst zeigt, dass es im oberen Drittel der Wichtigkeit zu finden ist, und obwohl es fertige Implementierungen von EG-Mitgliedern gibt *(siehe „http://cxf.apache.org/docs/jax-rs-oauth2.html").*

Die Studie behauptet, MVC, JMS und andere hätten „das Ranking verloren". Das stimmt nicht. MVC beispielsweise wurde in der Umfrage von mehr als 30 Prozent der Teilnehmer als „Very Important" angesehen; „Verlieren" sieht sicherlich anders aus! Zudem wurde in der Umfrage nie erwähnt, dass das Ziel ist, Technologien gegenseitig auszuspielen. Unter dieser Überschrift hätten die Teilnehmer möglicherweise ganz anders votiert.

Oracle hat nur nach JSRs gefragt (also nach Überschriften), nicht aber nach Inhalten. Trotzdem wird das Umfrage-Ergebnis als Begründung dafür genommen, die Richtung innerhalb der JSRs nach eigener Interessenlage zu ändern. JAX-RS beispielsweise wird zwar weiterhin in Java EE 8 enthalten sein, da es laut Umfrage eine „wichtige Technologie" ist, doch der Inhalt entspricht nicht ansatzweise dem, was ursprünglich für JAX-RS 2.1 im JCP festgelegt wurde (weder von der Menge noch von der Zielstellung her). Insofern scheint Oracle bei Durchführung der Umfrage nur daran interessiert gewesen zu sein, zu erfahren, ob die Überschrift „JAX RS 2.1" wichtig ist, nicht daran, welche Änderungen an dieser Technologie die Community wirklich benötigt.

Im Dokument wird sinngemäß zugegeben, dass zwar ein Rating von 1 bis 5 Punkten möglich war, es aber keine Ergebnisse kleiner als 3 Punkte gab – also auch keine „unwichtigen" Themen. Von daher ist es vollkommen unklar, wie angeblich zum Beispiel ein Maintenance-Release von JMS explizit als „unwichtig" begründet werden soll. JMS ist (und bleibt) das Standard-API für jede Art von MOM und somit unter anderem auch zur Nutzung von AMQP. Letzteres, wer es nicht kennt, erlebt gerade einen Hype dank der Nutzung in OpenStack *(siehe „https://docs.openstack.org/developer/nova/rpc.html#amqp-and-nova").* Hier bekommt ein neutraler Betrachter die Enden der Argumentationskette beim besten Willen mehr nicht zusammen.

Ähnlich ist es auch bei MVC, das von der Bewertungsskala her zwar im Mittelfeld lag, von dem der Autor der Studie aber seltsamerweise bestätigt sieht, dass es niemand benötigt. Um dies nicht allzu augenfällig werden zu lassen, wurde ein weiterer Taschenspielertrick angewendet: Die Skala der „Wichtigkeitsgrafik" auf Seite 5 beginnt erst bei 3 Punkten (nicht etwa bei 0) und endet bei 5. So sieht es natürlich aus, als wäre in der Community praktisch keinerlei Interesse an JMS und MVC vorhanden. Eine wissenschaftlich korrekte Darstellung hätte diesen Eindruck vermieden. Es ist kaum vorstellbar, dass die Ingeni-

eure bei Oracle diesen Skalierungsfehler versehentlich begangen hatten – einen klassischen Anfängerfehler der beschreibenden Statistik.

Einige Aussagen sind völlig aus der Luft gegriffen: Beispielsweise wird behauptet, dass viel Arbeit von JAX-RS (mit HTTP/2 und JSON-B als Top-Technologien dargestellt) bereits erledigt sei. Angesichts der aktuellen Diskussion auf der Mailingliste von JAX-RS um den kurzfristigen Rückbau wegen Nichtmachbarkeit (!) von NIO beziehungsweise des Flows-API und in Hinsicht auf die wenigen in JIRA sichtbaren Erfolge der letzten Monate ist es sicherlich vermessen, von „already complete" zu sprechen. Und HTTP/2 wird von JAX- RS 2.1 erst gar nicht unterstützt werden, sondern ist nur im Servlet-API enthalten. Servlets spielen aber dank JAX-RS eine immer weniger wichtige Rolle, was die Aussage, HTTP/2 sei bereits zum Zeitpunkt der Umfrage umgesetzt gewesen, doch sehr in Zweifel zieht.

Von den sechs in der Studie genannten Top-Technologien werden laut Roadmap nur drei in Java EE 8 umgesetzt, teilweise, weil man es für zu viel Aufwand hält – obwohl beispielsweise das Configuration-API noch auf der JavaOne großartig angekündigt wurde. Dieser Fakt lässt sich nur schwer in Einklang mit der Idee bringen, dass Oracle die Wünsche der Community wirklich ernst nimmt. So wird beispielsweise über OAUTH ebenfalls gesagt, die Zeit hätte einfach nicht gereicht. Dies entspricht schlicht nicht der Wahrheit: OAUTH war bereits aus dem Rennen, als noch genug Zeit war. Dass OAUTH für Cloud Services ein Must-have darstellt, scheint irrelevant zu sein. Dafür behält Oracle aber JSON-P nur bei, weil es gute Fortschritte gäbe, und ignoriert auch hier den Willen der Community, die JSON-P gar nicht im Spitzenfeld sieht. Es wurde also in Gebiete investiert, die die Community gar nicht als wichtig erachtet, und daher soll die Community nun einfach nehmen, was Oracle halt auf eigene Kappe entwickelt hat.

Über JSF 2.3 behauptet die Studie, es hätte bedeutsamen Fortschritt gegeben. Auch dies darf angezweifelt werden, da bereits die Planung des JSR lediglich die Sammlung der von der Community bereitgestellten kleinen Verbesserungen vorsah. Zwar hat ein Oracle-Mitarbeiter auf der JavaLand medienwirksam das Release verkündet, doch was die Community (weitgehend ohne Oracle) hier aufgefahren hat, war doch eher spärlicher Natur. Ohne die Leistung der Community kritisieren zu wollen: Unter „bedeutsam" stellt man sich doch eher Großes vor, und das wird von JSF 2.3 sicherlich nicht zu erwarten sein. Eine der JSF-Branchengrößen, PrimeTek, schreiben in ihrem Blog denn auch viel zu Angular und so gut wie nichts zu JSF, trotz der auf der JavaLand angeschnittenen Torte.

Über JPA schweigt sich die Studie, und Oracle bis heute, komplett aus. Eine Frage dazu wurde der Community erst gar nicht gestellt; laut GitHub *(siehe „https://github.com/eclipse/javax.persistence/graphs/contributors?from=2016-01-01")* arbeitet derzeit an einem von der Community eigentlich dringend benötigten Maintenance-Release (es haben sich viele kleine Wünsche und Nöte angestaut im JIRA-Tracker) offenbar keine einzige Person aktiv, an der Referenz-Implementierung EclipseLink nur zwei Personen – und auch diese nur sehr sporadisch *(siehe „https://github.com/eclipse/eclipselink.runtime/graphs/contributors?from=2017-01-01")!* Auf Rückfrage hierzu in der JavaLand-Paneldiskussion wurde auf die Arbeit an EclipseLink verwiesen. Dumm nur, dass EclipseLink ein Produkt ist, während die Frage auf JPA als internationale Norm abzielt. Hier zeigt sich, dass Oracle weiterhin Probleme hat, Industrie-Normen von Oracle-Produkten abzugrenzen.

Fazit

Bereits wenige Monate nach der Umfrage merkt Oracle nun scheinbar selbst, dass nicht einmal mehr der zusammengestrichene Inhalt von Java EE 8 und 9 zeitlich machbar ist: So hat beispielsweise das JAX-RS-Team im März 2017 damit begonnen, die Unterstützung für das Java-9-Flows-API aus dem NIO-Teil JAX-RS 2.1 zu streichen, obgleich auf der nur wenige Tage vorher stattfindenden Paneldiskussion auf der JavaLand noch behauptet wurde, alles sei auf Kurs – wohlgemerkt, von einem Expert Group Member und auch von zwei Oracle-Mitarbeitern. Nur zwei Tage später wurde von anderen Oracle-Mitarbeitern mitgeteilt: „Manche Ziele, die Java EE hatte, passen in dieser Zeit nicht mehr." und „Die Slides der JavaOne zu Java EE 9 sind nur eine Vision, aber es ist nicht machbar." Das hört sich nicht nach einem Commitment zu Java EE 8 und schon gar nicht zu dessen im JCP verabschiedeten Inhalt an. Zur Klarstellung sei hier auf den Oracle-üblichen Disclaimer verwiesen, dass jeder Mitarbeiter grundsätzlich nur seine persönliche Meinung vertritt und es keine offizielle Aussage des Unternehmens hierzu gibt. Trotzdem sprechen die Fakten letztendlich für sich: Hier hat ein Mitarbeiter zugegeben, dass das, was die Community unter Java EE erwartet, nicht mehr in Einklang zu bringen ist mit dem, was Oracle bereit ist zu bezahlen.

Bleibt abzuwarten, was am prognostizierten Release-Tag von Java EE 8 tatsächlich an Inhalt verfügbar ist und was wiederum nur heiße Luft beziehungsweise eine Buzzword-Sammlung war. Ob Java EE 9 jemals das Licht der Welt erblickt und dann mehr ist als nur eine Sammlung von Bug Fixes, steht jedenfalls weiterhin in den Sternen, trotz Commitments, Roadmaps und Surveys! **Hinweis:** Java aktuell plant für die nächste Ausgabe eine Stellungnahme zu diesem Thema von den Spec Leads der verschiedenen JSRs.

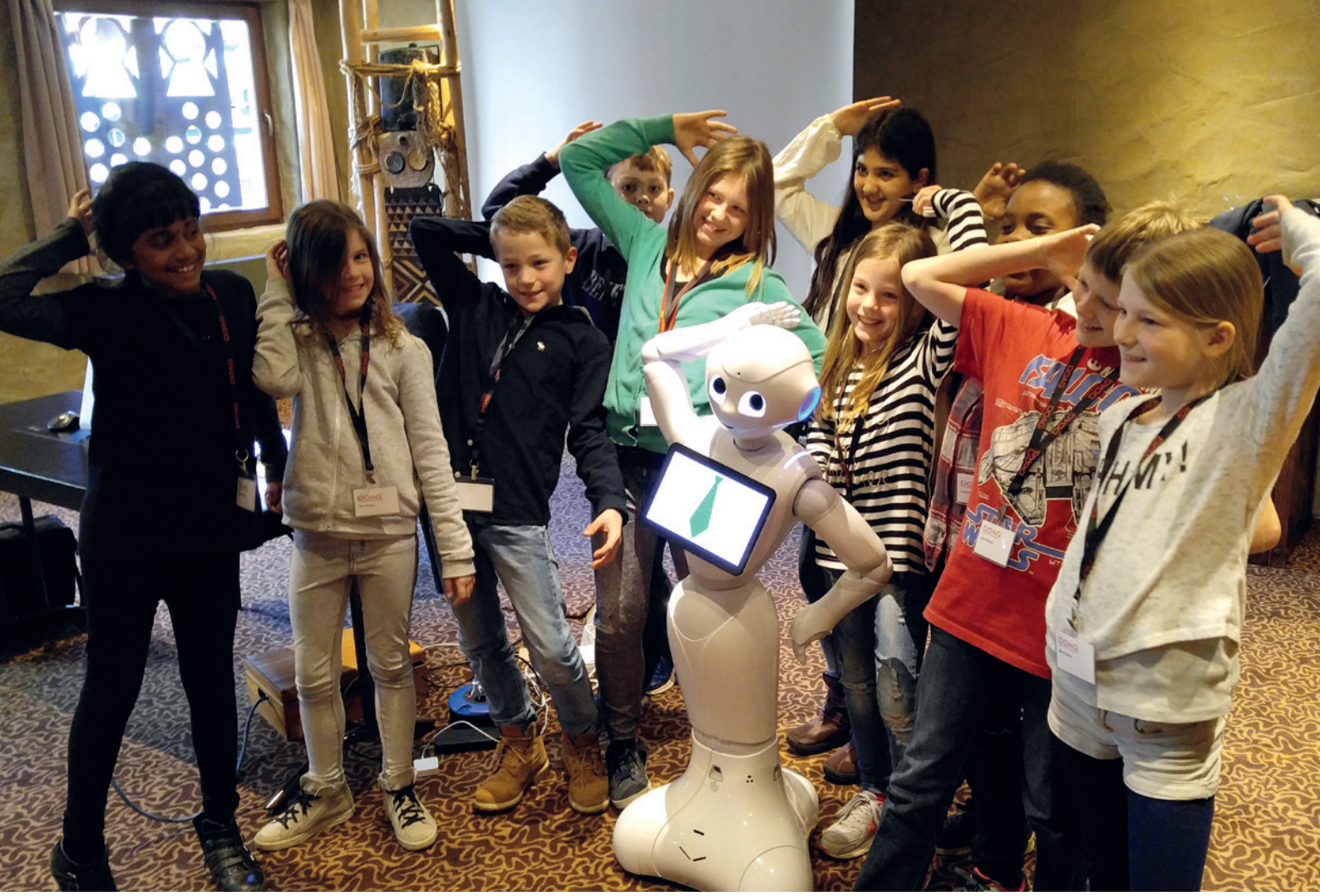
Markus Karg
markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. Der Sprecher der Java User Group Goldstadt verfolgt mit kritischem Blick die aktuellen Entwicklungen im Java-Ökosystem.

Impressionen von der JavaLand 2017

DOAG Online

Die JavaLand schließt am 29. März 2017 ihre Pforten. Das außergewöhnliche Konferenzformat kommt super an: Zur vierten Auflage der Veranstaltung im Phantasialand in Brühl sind 1.655 Entwickler und Programmierer anwesend, um sich in mehr als hundert Vorträgen und vielen Community-Aktivitäten über die neuesten Trends der Szene zu informieren. Auch die begleitende Ausstellung mit mehr als 35 Ausstellern hält wieder spannende Technologien und Networking-Möglichkeiten bereit. „Die JavaLand hat sich seit ihrem Start im Jahr 2014 zu einer der größten und bedeutendsten Veranstaltungen der Java-Community entwickelt. Wir freuen uns, dass unser außergewöhnliches Veranstaltungsformat so gut ankommt, dass wir inzwischen die Teilnehmerzahl verdoppeln konnten", sagt Fried Saacke, Vorstandsvorsitzender des iJUG. Mit zwei Keynotes hat der erste Veranstaltungstag gleich zwei Highlights im Programm: Java-Champion Martin Thompson gibt in seinem Vortrag einen neuen Blickwinkel auf die Entwicklung von Software-Architekturen. Was mit Unternehmen passiert, die die Digitalisierung verschlafen, macht dagegen Jean-Jacques van Oosten, Chief Digital Officer bei der Rewe Group, in seiner Keynote unmissverständlich klar: „Diese Unternehmen können ihr Testament machen." Wie man die Digitalisierung geschickt angeht, erklärt der Online-Stratege am Beispiel der digitalen Transformation der Rewe Group. Das Datum für die fünfte Ausgabe der JavaLand steht bereits fest: Sie findet vom 13. bis zum 15. März 2018 an gewohnter Stätte im Phantasialand statt. Weitere Informationen unter *„https://www.javaland.eu"*.



Kann man Strom zerschneiden?

Uwe Sauerbrei, Kids4IT

Dies war eine der Fragen, die im Verlaufe der dritten JavaLand4Kids beantwortet wurden. Die kleine Konferenz, deren Fokus auf den Workshops mit den Kindern liegt, bot auch in diesem Jahr dreißig Kindern aus zwei vierten Klassen die Möglichkeit, sich einen Tag mit MINT-Themen (Mathematik, Informatik, Naturwissenschaft und Technik) zu beschäftigen.

Vor drei Jahren begann es mit einem Testballon. Mehr oder weniger unter Ausschluss der Öffentlichkeit fand die erste JavaLand4Kids statt. Am ersten Tag der Konferenz begaben wir uns in die Konferenzräume des Hotels Matamba und absolvierten unsere ersten Workshops mit den Kindern. Die Resonanz war durchweg positiv. Einziger Wermutstropfen: Alle Aktivitäten fanden unter Ausschluss der Öffentlichkeit statt und viele potenziell interessierte Mentoren waren zur gleichen Zeit als Speaker unterwegs und somit nicht abkömmlich. Also verschoben wir die zweite Auflage der JavaLand4Kids auf den Montag vor der Konferenz. Dieses Konzept behielten wir auch in diesem Jahr bei. Im Gegensatz zum Jahr 2016 gingen wir mit dreißig Kindern und fast zwanzig Mentoren an den Start. Alle drei Tagungsräume des Hotels wurden für die Workshops (Chibitronics, Jumping Sumo, Lego) hergerichtet.

Chibitronics

Während des Junghackertags auf der 33. Ausgabe des Chaos Communication Congress (33C3) entdeckte Melanie Wallintin das Bastelset eines amerikanischen Start-ups. Ihr gebührt auch der Dank, diesen Workshop mit viel Liebe zum Detail vorbereitet und organisiert zu haben. Inhaltlich handelt sich um eine innovative Kombination aus Papier und Elektrik. Mit Schere und Klebestift können Leiterbahnen in Figuren eingearbeitet und mit LEDs und Batterien versehen werden *(siehe Abbildung 1)*. Am Ende leuchteten nicht nur die Augen der Kinder, sondern auch die der gebastelten Osterhasen aus Papier.

Zuvor hatten wir uns gefragt, ob ein solches Thema für Jungs in der vierten Klasse noch spannend oder gar peinlich wäre. Die Sorge war völlig unbegründet; sowohl Jungs wie auch die Mädchen boten mehrfach an, auf die Pausen zu verzichten, um ihre Exponate weiter

Abbildung 1: Verschiedene Utensilien zum Basteln mit Chibitronics

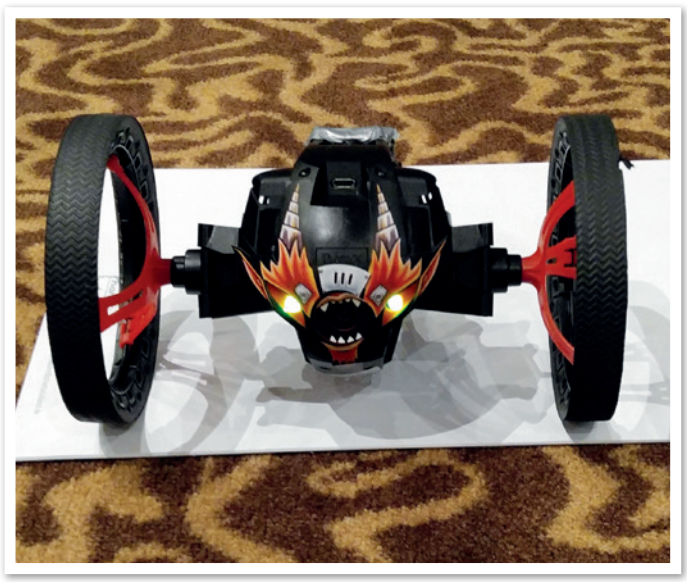


Abbildung 2: Ein wenig martialisch, ansonsten ein netter Zeitgenosse: Jumping Sumo

zu verfeinern. Einziger Nachteil: Das Material ist am Ende des Workshops verbraucht und durch den Import aus Übersee leider auch nicht ganz preiswert. Spaß hat es allen Beteiligten aber auf jeden Fall gemacht und wir haben eine Menge gelernt.

Lego und Pepper Milli

Wie in den beiden Jahren zuvor machte sich Bernhard Löwenstein aus Wien auf den Weg ins Phantasialand. Er hat in Österreich das Institut zur Förderung des IT-Nachwuchses (IFIT) gegründet und veranstaltet im Jahr mehr als hundert Workshops mit Kindern und Jugendlichen. Spezialgebiet sind die Roboter und so hatte er im Gepäck nicht nur die Lego-Sets, sondern auch Pepper Milli, die zwischendurch immer mal wieder für Auflockerung bei den Kindern sorgte.

Jeder, der vor einem Kasten mit Legosteinen sitzt, sei es nun ein Erwachsener oder Kind, wird über kurz oder lang damit beginnen, kreativ zu werden. In solchen Fällen entwickeln die Finger eine ungeahnte Eigendynamik. So legten die Zweier-Teams auch sofort los, ihre fahrbaren Modelle zu konstruieren und einen Parcours zu programmieren. Die Mentoren, jeweils einer für zwei Kinder, mussten nur selten eingreifen, auch wenn sie es gerne öfter getan hätten.

Jumping Sumos

Wenn man so will, ist die JavaLand4Kids eine Kooperation der DOAG (vielen Dank an die tolle Organisation durch Simone Fischer), der Devoxx4Kids Deutschland in Person von Katja Arrasz-Schepanski und ihrem Team sowie von Kids4IT Hamburg. Katja hatte im letzten Jahr auch erstmalig die Jumping Sumos ins Spiel gebracht, die uns in diesem Jahr erneut begleiteten. Zusätzlich konnten wir auf ein durch die Community entwickeltes Scratch-Plug-in zurückgreifen, das die Programmierung noch mal ein ganzes Stück erleichterte *(siehe Abbildung 2)*.

Die Sumos können fahren, springen und übertragen mit einer Kamera alles, was sie sehen, auf eine App oder den Rechner, der sie steuert. Grundsätzlich würden sie sich sehr gut dazu eignen, verlorene Dinge unter dem Sofa zu finden. Bei uns ging es jedoch in erster Linie um ernsthafte Steuerung in einem vorgegebenen Fahr-Sprung-Hindernis-Parcours. Es ist immer wieder erstaunlich, wie schnell sich die Kinder auf eine ihnen bis dato unbekannte Technologie einlassen, sie sofort annehmen und mit eigenen Tests und Experimenten beginnen. Das ist Neugier und Begeisterung pur, die sich auf jeden übertrug, der ihnen bei der Arbeit zuschaute.

Fazit

Unser Resümee auch in diesem Jahr: Aus einer kleinen Veranstaltung, die nur als Experiment gedacht war, ist inzwischen eine Institution geworden, die das Potenzial zu einer eigenen Marke im Umfeld der JavaLand-Konferenz hat. Alle Beteiligten hatten großen Spaß und die abschließende Nutzung der Fahrgeschäfte im Park tat dem keinen Abbruch. Im nächsten Jahr geht es natürlich weiter und wir freuen uns schon jetzt, viele bekannte Gesichter unter den Mentoren und freiwilligen Helfern erneut zu sehen. Es wird toll werden, versprochen.

Uwe Sauerbrei
info@kids4it.de

Uwe Sauerbrei arbeitet als selbständiger IT-Berater in Hamburg. Er organisiert die dortige Java User Group und hat mit Kids4IT eine gemeinnützige User Group gegründet. Kids4IT veranstaltet monatliche Workshops in der Hansestadt, zeichnet für die Devoxx4Kids-HH verantwortlich und unterstützt die JavaLand4Kids.

Named Queries strike back

Mario Vöhl

Named Queries sind nützlich, verursachen aber Overhead. Dieser hat abschreckende Wirkung. Doch er lässt sich drastisch reduzieren.

Wer bereits mit JPA Named Queries gearbeitet hat, kennt Code mit dieser Struktur *(siehe Listing 1)*. Er erzeugt und konfiguriert die Query mit Namen „User.findByName". Im letzten Schritt wird die Query dann ausgeführt, wobei man eine Liste von User-Instanzen als Resultat erwarten darf.

Man erkennt leicht, dass das Listing nicht den kompletten Code zeigt, denn offenbar fehlt die Definition der eigentlichen Query. Der Code setzt voraus, dass die Query-Logik zuvor unter jenem Namen „User.findByName" im System registriert wurde. Nur dann kann eine Instanz der Query durch die Methode „EntityManager.createNamedQuery()" abgerufen werden.

Die Annotation

Typischerweise geschieht die Registrierung von Named Queries beim Start der Anwendung automatisch. Dazu werden verschiedene Quellen nach potenziellen Queries abgesucht: XML-Dateien und populärere Annotationen in Entity-Klassen. In *Listing 2* ist die Query daher über eine Annotation gegeben, die sich etwa so darstellen könnte.

Die Annotation definiert ein Paar aus Namen und Query-Statement. Das Statement darin ist kein SQL, sondern entstammt der Java Persistence Query Language. Dabei müssen Klassen und Felder anstelle von Datenbank-Tabellen und -Spalten verwendet werden. Man erhält also eine am Java-Modell orientierte Abfragesprache.

Den echten Mehrwert stiftet das Statement, da es normalerweise einen Teil des Use Case abbildet. Die Geschäftslogik ist demnach als String in einer Annotation implementiert. Der übrige Code ergibt sich fast automatisch und ist deshalb als Overhead oder Glue Code zu sehen. Es ist beispielsweise klar, dass der Code den im Statement durch einen Doppelpunkt markierten Parameter „theName" belegen muss. Zusätzlich kann man noch ableiten, dass die Query am Ende User-Objekte liefern wird. Allerdings ist zunächst nicht ersichtlich, wie viele Treffer es geben kann. Dazu muss das Zusammenspiel des Statements mit der Entity-Klasse analysiert werden. Ausgehend vom Statement ergibt sich nur, dass ein User ein Feld „name" mit nicht weiter definiertem Datentyp besitzen muss. *Listing 3* zeigt eine mögliche Implementierung der User-Klasse. Dabei ist der Name ein String, der per se nicht eindeutig sein muss. Es ist also sinnig, dass der Glue Code am Ende „getResultList()" aufruft und damit potenziell mehrere Treffer liefert.

Insgesamt ergibt sich ein impliziter Vertrag zwischen den Codestellen. Wird dieser verletzt, kommt es zu Fehlern beim Deployment oder gar erst zur Laufzeit. Kurz gesagt müssen Entity, Statement und Glue Code zueinander passen.

Die String-Hölle

Die Einhaltung des Vertrags erfordert unter anderem, die Strings aus der Query-Annotation an den richtigen Stellen im Glue Code zu wiederholen. Schließlich verwendet dieser Code hochgradig Stringlastige Schnittstellen wie den EntityManager und die Query. Dass der Parameter „theName" sinnvollerweise nur mit einem String belegt werden sollte, kann der Compiler dabei nicht feststellen. Die verwendeten Schnittstellen sind abstrakt und vom konkreten Use Case unabhängig.

Letztendlich wäre für den Java-Entwickler ein einfaches und typsicheres API pro Named Query wünschenswert, etwa „List<User> findByName(String theName)".

Bei der Implementierung dieses API stößt man schließlich aber wieder auf die zu wiederholenden Strings, nämlich den Query-Namen und die Namen der Parameter. Weil die Query-Annotation und der Glue Code typischerweise in verschiedenen Klassen liegen, kommt es häufig zur Verwendung von String-Konstanten, um sicherzustellen, dass die beiden Codestellen auch in jedem Fall synchron sind. Diese Konstanten werden meistens in der Entity-Klasse platziert. Zudem ist es naheliegend, die Konstanten für die Parameter im Statement zu verwenden.

Neben dem erhöhten Aufwand wird das Statement damit schlechter lesbar und die gewünschte automatische „Vertragstreue" wird auch nur bedingt erreicht. Denn String-Konstanten (public static final String) werden zur Compile-Zeit im Bytecode des Verwenders hinterlegt. Es findet keine Laufzeitauswertung des Strings mehr statt. Damit bietet dieses Vorgehen möglicherweise gar nicht die erhoffte Sicherheit. Einen tatsächlichen Zugriff auf die bereits definierten Konstanten kann man mit Zugriffsmethoden implementieren. Die erweiterte Entity könnte wie in *Listing 4* aussehen. Dazu passend verwendet der Glue Code die Getter-Methoden zum Auslesen der Konstanten zur Laufzeit *(siehe Listing 5)*.

```
Query query = entityManager.createNamedQuery("User.
findByName");
query = query.setParameter("theName", theName);
return query.getResultList();
```

Listing 1

```
@NamedQuery(
  name = "User.findByName",
  query = "Select u From User u Where u.name = :the-
Name"
)
```

Listing 2

```
@Entity
@NamedQuery(
  name = "User.findByName",
  query = "Select u From User u Where u.name = :the-
Name" )
public class User  {

  @Id private int id;

  private String name;
  ...
}
```

Listing 3

Der Zugriff auf die Konstanten über Methoden ist eher ein Gedanken-Experiment. Die Entity wird damit enorm aufgebläht. Aber auch ohne die Methoden erfordert das beschriebene Vorgehen mit Verwendung von Konstanten vergleichsweise hohen Aufwand für eine simple Named Query.

Einfacher mit Anqu Method

Wie erwähnt, ergibt sich der Glue Code fast automatisch aus einer vorhandenen Query. Das Anqu-Method-Plug-in für Eclipse kann daher den Implementierungsaufwand merklich verringern. Eine Analyse der Entity sammelt alle Informationen, um das gewünschte Java API pro Query zu generieren. Außerdem gibt es Unterstützung für das Aufsetzen des Statements beziehungsweise der notwendigen Annotation. Mit dem Anqu-Method-Plug-in („@N"amed „Qu"ery) sind verschiedene Funktionen im Kontextmenü verfügbar *(siehe Abbildung 1)*.

Die Implementierung der Beispiel-Query kann damit etwa wie folgt ablaufen: Als Erstes wird per „New Named Query" eine neue Annotation im Quellcode der Entity ergänzt. Sofort danach kann mit der Eingabe des Query-Namens begonnen werden. In diesem Fall „User.findByName", wobei das Präfix „User." standardmäßig bereits vorgeneriert wird. Einzutippen ist also nur noch „findByName".

Aufgrund der in Anqu eingebauten Grammatik kann anschließend das passende beziehungsweise erwartbare Statement aus dem Query-Namen „findByName" automatisch erzeugt werden. Dafür

```
@Entity
@NamedQuery(
  name = User.FIND_BY_NAME,
  query = "Select u From User u Where u.name = :" +
        User.NAME_PARAM
)
public class User  {

  static final String FIND_BY_NAME = "User.findBy-
Name";
  static final String NAME_PARAM = "theName";

  public static String getQueryNameFindByName() {
     return FIND_BY_NAME;
  }

  public static String getQueryParamName() {
     return NAME_PARAM;
  }
  ...
}
```

Listing 4

steht die Funktion „Derive Statement" zur Verfügung. Auch aus komplexeren Query-Namen ist es möglich, das Statement abzuleiten. Die zu verwendende Grammatik ist auf der Anqu-Method-Hilfeseite zu finden, die unter anderem über das Kontextmenü leicht erreichbar ist. Alternativ lässt sich das Statement natürlich auch eintippen.

Nachdem das Statement definiert ist, kann die Java-Methode mit dem Glue Code erzeugt werden. Hier wählt man im allgemeinen Fall den obersten Punkt „Execution" im Kontextmenü aus. Dies generiert den Code, der die Query am Ende tatsächlich ausführt. Ist das Statement ein „Select", liefert die erzeugte Methode eine Liste von Objekten. Bei Updates oder Deletes wird die Anzahl der betroffenen Objekte zurückgegeben.

Ist man sich sicher, dass die Query maximal einen Treffer beim „Select" hat, kann auch der zweite Menüpunkt „Single Result Execution" verwendet werden. Der generierte Code liefert hierbei den ersten Treffer – oder „null", wenn es keine Treffer gab.

Damit ist festgelegt, welche Art von Query-Ausführung gewünscht

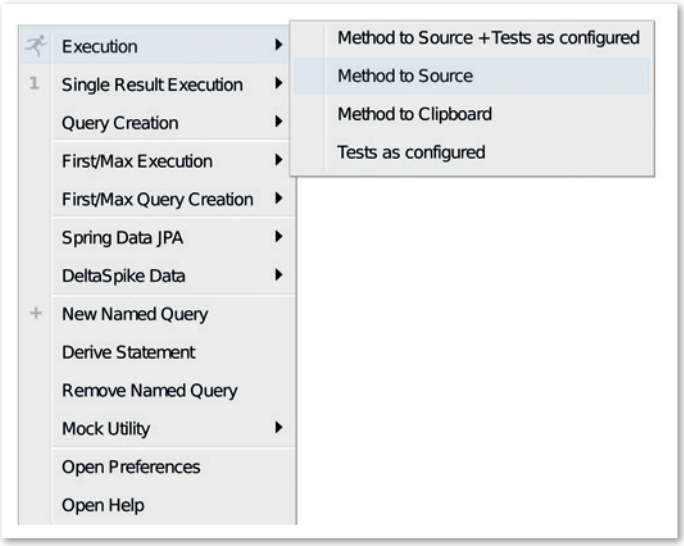


Abbildung 1: Das Kontextmenü von Anqu Method

```
Query query = entityManager.createNamedQuery(
           User.getQueryNameFindByName()
          );
query = query.setParameter(User.getQueryParamName(),
theName);
return query.getResultList();
```

Listing 5

```
@Query(named = "User.findByName")
List<User> findByName(@QueryParam("theName") String
theName);
```

Listing 6

ist: das Lesen einer Liste, das Lesen maximal eines Objekts oder das Updaten/Löschen von Daten. Es fehlt nun noch die Entscheidung, wohin der Glue Code generiert werden soll. Dies geschieht auf der nächsten Ebene des Menüs. Bei einem Design, das mit Data Access Objects (DAO) arbeitet, ist es wahrscheinlich, dass der Code dort angesiedelt werden soll. In diesem Fall kann der Punkt „Method to Clipboard" gewählt werden. Anschließend liegt der Code in der Zwischenablage und kann im DAO eingefügt werden. In diesem einfachen Beispiel wurde daher allein der Query-Name „findByName" eingegeben. Der Rest des Codes konnte über Anqu Method generiert werden.

Wer dies einmal ausführt, wird sich über den generierten Code wundern. Denn die Methode ist statisch und hat als ersten Parameter den EntityManager. Das entspricht nicht der erwarteten Methoden-Signatur in einem DAO. Anqu Method ist standardmäßig auf das Pattern des maximal tiefen Glue Codes eingestellt, der weiter unten erläutert wird. Mit wenigen Klicks kann die Generierung aber für typische DAOs eingerichtet werden. Dazu die Eclipse Preferences für das Plug-in öffnen. Das Anqu-Method-Kontextmenü enthält den entsprechenden Punkt „Open Preferences". *Abbildung 2* zeigt die Konfiguration für eine DAO-Variante.

Das Anqu-Method-Pattern

Der Entwicklung von Anqu liegen drei Prinzipien zugrunde, denen

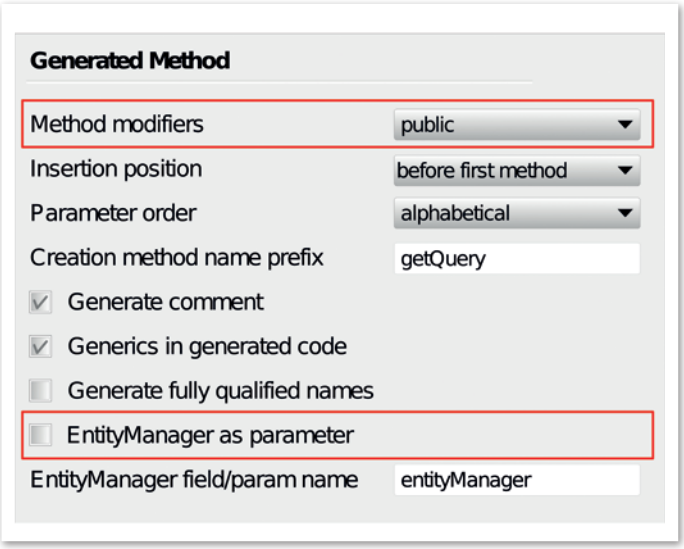


Abbildung 2: Einstellungen für Glue Code im DAO

wahrscheinlich die meisten Softwareentwickler grundsätzlich zustimmen können: Das erste Prinzip besagt, dass der Glue Code aufgrund des schlecht wartbaren, String-basierten Vertrags nur einmal im Anwendungscode vorkommen sollte („einmaliger Glue Code"). Dazu gehört, dass man sich bewusst entscheidet, wo der Glue Code anzusiedeln ist, etwa im entsprechenden DAO. Das zweite Prinzip ist das des „tiefen Glue Codes": Der Code soll möglichst weit unten in der Architektur angesiedelt sein, damit den darüber liegenden Schichten frühzeitig ein Compiler-geprüftes API zur Verfügung gestellt wird.

Drittens ist der Test des Vertrags gefordert. Durch die inhärente Kopplung über Strings und die schwach ausgeprägte Typsicherheit der Grundkonstruktion können die oben beschriebenen Verträge leicht gebrochen werden. Im schlimmsten Fall tritt dies jedoch erst zur Laufzeit auf. Daher sollte die Einhaltung der Verträge zwischen den beteiligten Codestellen durch Tests überwacht werden.

Die drei Prinzipien mit „einmaligem" und „tiefem" Glue Code sowie dem Test des Vertrags bilden das Anqu-Method-Pattern – eine allgemeine Programmier-Empfehlung für Named Queries. Das Pattern ist frei interpretierbar. Sein Wert besteht darin, sich mit den drei Prinzipien gedanklich auseinanderzusetzen und zu eigenen Design-Entscheidungen zu kommen.

Maximal tiefer Glue Code

Der erwähnte „maximal tiefe Glue Code" ist eine Ausgestaltung des Patterns, die erhöhten Wert auf das Verbergen des Glue Codes legt. Konkret bedeutet das, die Methode mit dem Glue Code statisch in der Entity anzusiedeln. Daher muss der EntityManager als Parameter übergeben werden. Dies erklärt die oben beschriebenen Voreinstellungen des Plug-ins.

Aus diesem Vorgehen ergibt sich automatisch die Einmaligkeit des Glue Codes in der Anwendung. Zudem wird alles, was mit der Query zu tun hat, in einer Klasse gekapselt. Nach außen erhält man ein einfaches Java-API. Insbesondere werden die Strings nur intern verwendet.

Man kann sich nun an der Übergabe des EntityManager in der Methoden-Signatur stören oder der Meinung sein, dass eine Entity keine Logik enthalten darf. Am Ende ist es aber eine Option von JPA, die Named Query, die definitiv Logik abbildet, an der Entity zu annotieren. Infolgedessen ist nicht abwegig, auch die statische Ausführung in der Entity zu verankern.

Zudem ist dieser Ansatz hochgradig automatisierbar. Anqu hat dazu die Variante „Method to Source" bei der Auswahl des Ziels für die Generierung. Damit wird die Methode direkt in der Entity erzeugt. Der Umweg über die Zwischenablage entfällt. Ganz allgemein unterstützt das Plug-in die Forderung nach dem Test des Vertrags zwischen den beteiligten Codestellen. Dazu können JUnit-Tests generiert werden, die fehlschlagen, wenn sich Code-Bestandteile wie

das Statement oder der Glue Code ändern. Diese Tests etablieren dabei eine Art Schreibschutz auf den betroffenen Code-Abschnitten. Es soll schlichtweg verhindert werden, dass diese manuell geändert werden. Die JUnit-Tests können pro Query über das Kontextmenü erzeugt werden. Auch hier gibt es Einstellungsmöglichkeiten in den Preferences. Entsprechend heißt der Kontextmenü-Unterpunkt „Tests as configured". Die so generierten Tests benötigen lediglich das weit verbreitete Mockito-Framework im Classpath.

Mit dem Ansatz des Glue Codes in der Entity lassen sich Glue Code und Tests mit einem Klick gemeinsam erzeugen. Man erhält dadurch mit wenig Aufwand eine über Tests abgesicherte Implementierung des Glue Codes sowie ein Java-API pro Query. Der maximal tiefe Glue Code besticht durch seine Automatisierbarkeit, was von dessen Einfachheit zeugt. Einfachheit wiederum kann für Langlebigkeit des Designs sprechen.

Die zuvor beschriebenen generierten Tests zielen allein auf die Überwachung des Vertrags ab. Sie sind also kein inhaltlicher Test des Statements. In vielen Fällen möchte man aber nicht nur die Funktionsfähigkeit der Query, sondern auch den Code testen, der die Query aufruft. Beim maximal tiefem Glue Code erfolgt die Query-Ausführung in der statischen und zustandslosen Methode der Entity. Damit ist das Verhalten so, als ob der Code der Methode inline in der aufrufenden Klasse implementiert wäre.

Will man im Test das Verhalten der Query simulieren, so wird zuerst der EntityManager gemockt. Dieser produziert aber nur das Query-Objekt, das anschließend konfiguriert und ausgeführt wird. Dabei soll letztendlich das im Test benötigte Query-Ergebnis geliefert werden.

Zur Vereinfachung stellt Anqu Method sogenannte „Mock-Utility-Klassen" bereit. Sie übernehmen das Mocken des gesamten Glue Codes für eine ausgewählte Query. Eine Mock-Utility-Klasse lässt sich nachträglich oder direkt beim Anlegen des Glue Codes pro Query erzeugen.

Der maximal tiefe Glue Code legt oberste Priorität auf die Abschirmung der Anwendung von den Implementierungsdetails einer Named Query. Dies wird durch Kapselung des String-basierten technischen API hinter einer ausmodellierten Schnittstelle erreicht. Nicht nur mit Code-Generierung ist dies eine schnell zu implementierende Variante, die insbesondere betrachtet werden sollte, wenn noch keine Design-Entscheidung zu Named Queries gefällt wurde oder die bestehende als zu aufwändig erachtet wird.

Weitere Features

Anqu Method bietet noch weitere Funktionen wie das Entfernen von Query, Glue Code und Tests in einem Schritt. Sollte man eine Änderung an einem bestehenden Statement vornehmen wollen, ohne den Query-Namen zu ändern, kann bei Verwendung des maximal tiefen Glue Codes einfach eine Neugenerierung angestoßen werden. Das Plug-in überschreibt dann alle relevanten Stellen mit der neuen Version.

Die Generierung von Methoden mit Pagination-Parametern wird ebenso unterstützt wie Statements mit abweichenden Ergebnistypen oder Parameter, die Zeitangaben darstellen und daher eine Sonderbehandlung erfordern. Sollte die benötigte Variante zur Ausführung eines Statements nicht per se angeboten sein, lässt sich als Fallback eine Methode erzeugen, die die bereits parametrisierte Query als Objekt liefert. Diese kann dann noch weiter konfiguriert und anschließend ausgeführt werden.

Zusätzlich gibt es Unterstützung für Frameworks mit einem Repository-Ansatz wie Apache DeltaSpike oder Spring Data JPA. Bei einem typischen Repository muss zur Ausführung einer vorhandenen Named Query spezieller Code in einem Interface abgelegt sein (siehe *Listing 6*, Code gemäß DeltaSpike).

Das Framework kann die Query dann anhand der Methoden-Signatur beziehungsweise der dort angebrachten Annotationen ausführen. Man erkennt leicht die Wiederholung von Strings aus der Query-Annotation. Im Prinzip ist dies eine andere Form des bekannten Glue Codes, der von Anqu Method ebenfalls aus einer vorhandenen Query erzeugt und danach im Repository-Interface eingefügt werden kann. Wer gerne mit Named Queries arbeiten möchte, kann auch bei einem Repository-Ansatz noch die Funktionalitäten zum Erzeugen der Annotation und zum Ableiten des Statements aus dem Query-Namen verwenden.

Fazit

Das Anqu-Method-Pattern soll motivieren, sich aktiv um eine Design-Entscheidung bezüglich der Implementierung von Named Queries zu kümmern. Als einfache Variante wurde der maximal tiefe Glue Code besonders beleuchtet.

Mit dem Anqu-Method-Plug-in für Eclipse lassen sich dann in vielen Fällen fast alle Arbeitsschritte der konkreten Implementierung beschleunigen. Teilweise reicht es aus, einen Query-Namen zu vergeben und dem Plug-in den Rest der Arbeit zu überlassen.

Anqu Method ist ein Tool für Sourcecode-Generierung. Der Verwender erhält JPA-konformen Code und damit Klarheit und Kontrolle. Es wird weder eine besondere Laufzeit-Bibliothek noch ein weiteres Framework vorausgesetzt. Das Plug-in steht unter *„www.anqu-method.de"* und kostenlos nutzbar. Die Website bietet weitere Beispiele und auch kurze Video-Tutorials.

Mario Vöhl
contact@anqu-method.de

Mario Vöhl ist Diplom-Informatiker und arbeitet als Software-Entwickler und -Architekt beim Verlag für Standesamtswesen in Frankfurt am Main. Er beschäftigt sich meist mit Java-EE-Backend-Programmierung und ist Entwickler des Anqu-Method-Plug-ins für Eclipse.

Praktischer Einstieg in Neo4j

Darko Križić, PRODYNA AG

Quasi jede Anwendung verwendet eine (oder mehrere) Datenbanken für das Persistieren von Daten. Dabei wird das fachliche Domänen-Modell auf die Datenbank abgebildet. Traditionell kommen dabei relationale Datenbanken zum Einsatz. Es gibt entsprechend viele Erfahrungen damit, welche Stärken und Schwächen sie haben. In den letzten Jahren sind mehr als 200 Datenbanken im NoSQL-Bereich aufgetaucht [1], darunter auch Neo4j, eine Graph-Datenbank. Im Gegensatz zu relationalen Datenbanken, die auf der Mengenlehre basieren, stützt sich Neo4j auf die Graphentheorie. Dieser Artikel zeigt, welche Auswirkungen das auf die Praxis hat.

Auf den ersten Blick ist das Konzept von Neo4j sehr einfach: Es gibt nur Nodes (Knoten) und Relationships (Kanten). Knoten können für sich allein existieren, Kanten müssen immer an einem Knoten anfangen und wieder an einem (auch demselben) enden. Dabei können Knoten ein Label haben, etwa „Person" oder „Group", während Kanten einen Typ haben; letztere werden üblicherweise großgeschrieben. Kanten habe eine explizite Richtung. Beide (Knoten und Kanten) haben Properties (Eigenschaften). Der große Unterschied zur relationalen Datenbank ist, dass die Relationen explizit sind und nicht implizit als Werte in den Tabellenspalten sitzen.

Abbildung 1 zeigt die Tabelle „user", die mit der Spalte „gid" auf die Tabelle „group" verweist. Tabellen vermischen also ihre fachlichen Daten (wie „name") mit technischen. Die Relation ist schwer zu sehen. Das gleiche Beispiel sieht bei Neo4j wie in *Abbildung 2* aus.

Um die Knoten und Kanten zu erzeugen, ist ein Befehl in der Sprache Cypher notwendig *(siehe Listing 1)*. Sowohl Knoten als auch Kanten haben beliebige Properties *(siehe Abbildung 3)*.

Mit diesem Modell lassen sich relationale Daten problemlos in Neo4j importieren und weiterverarbeiten. Häufig gibt es das Missverständnis, dass Neo4j Daten auch nur grafisch repräsentieren kann, was nicht der Realität entspricht. Letztendlich ist Neo4j eine Datenbank und die Werte der Eigenschaften lassen sich ganz normal tabellarisch ausgeben *(siehe Abbildung 4)*. Eine Anwendung, die auf Neo4j basiert, kann für den Anwender ganz normal aussehen, allerdings gibt es natürlich auch die Option, dem Anwender eine grafische Repräsentation der Daten anzubieten.

Abbildung 1: Implizite Beziehungen in einer relationalen Datenbank

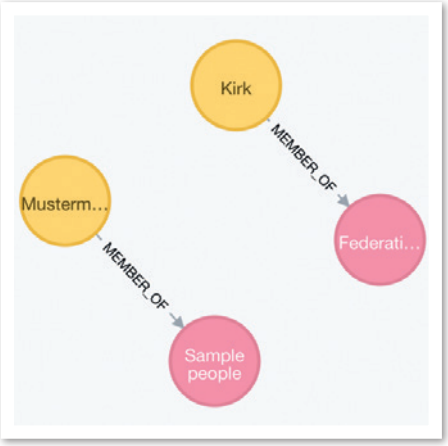


Abbildung 2: Explizite Relationen (Relationships in Neo4j)

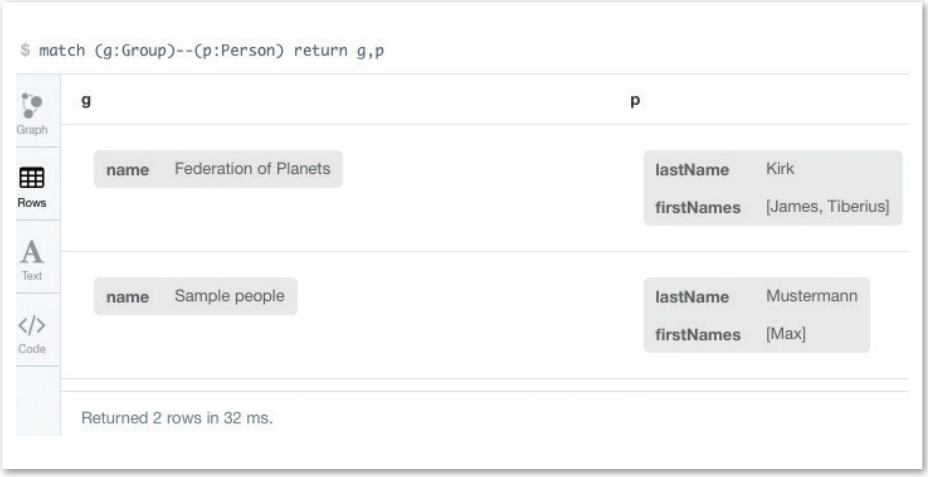


Abbildung 3: Properties von Knoten

Darüber hinaus gibt es noch wesentlich komplexere Abbildungsmöglichkeiten, die für die Modellierung eine ausschlaggebende Rolle spielen:

- Ein Knoten kann beliebig viele Labels enthalten, eine Person kann also zum Beispiel auch gleichzeitig Mitarbeiter sein.
- Auch Kanten können beliebige Properties haben, etwa „since", um darzustellen, ab wann jemand Mitglied ist.
- Kanten habe zwangsweise eine Richtung, A->B ist etwas anderes als A<-B. Die Richtung spielt für die Abfrage und die Modellierung eine große Rolle, hat aber auf die Antwortzeiten keine Auswirkung.
- Zwischen zwei Knoten können beliebig viele Kanten existieren und jede Kante kann ein beliebiger Typ sein.
- Die Werte von Properties können einzeln existieren oder ganze Listen sein.

Dadurch ergeben sich völlig neue Möglichkeiten der Modellierung, die weit über das hinausgehen, was mit einer relationalen Datenbank möglich ist. In der Praxis ist es daher sinnvoll, dass die Modellierung an einem Flipchart startet, und zwar nicht auf Entitäten-, sondern auf Objekt-Ebene. Dabei sollte man auf Werkzeuge wie UML verzichten, da diese die Möglichkeiten erst gar nicht abbilden können.

Wenn die ersten Daten in Neo4j abgebildet sind, kann der fachliche Mitarbeiter sie sofort sehen, verstehen und bestätigen, ob das fachliche Modell richtig abgebildet ist. Durch die freie Wahl der Richtung von Kanten kann das Graphenmodell den Vorgaben des fachlichen Modells direkt folgen; bei einer relationalen Datenbank sind die Richtungen von Fremdschlüsseln bei einer „1:N"-Relation immer von „N" nach „1".

Installation

Neo4j ist vollständig in Java entwickelt und benötigt nur eine aktuelle Java-Runtime für die Ausführung. Es existieren fertige Pakete für Unix/Linux, macOS, Windows, aber auch für fertige Docker-Container. Zusätzlich gibt es für Debian ein eigenes APT-Repository *[2]*, sodass ein „apt-get install neo4j" für die Installation ausreicht. So hat man zusätzlich den Vorteil, dass auch gleich Start/Stopp-Skripte installiert und die Installation einfach aktuell gehalten werden kann. Gerade für Produktionsumgebungen ist dieses Verfahren von Vorteil. Nach der Installation kann man mit dem Internet-Browser auf *„http://localhost:7474"* gehen und den Neo4j-Browser sehen *(siehe Abbildung 5)*.

In diesem Browser lassen sich nun Abfragen mit Cypher machen und die Daten sowohl grafisch als auch tabellarisch darstellen. Der Browser ist im Übrigen keine Oberfläche für Endanwender – dafür ist die Abfragesprache Cypher für Endanwender in der Regel zu kompliziert –, aber er ist ein sehr mächtiges Werkzeug für Entwickler, etwa um Abfragen zu testen. Standardmäßig lauscht Neo4j auf den folgenden Ports:

- **7474**
  Zugriff per REST/HTTP. Hier steht ein REST-Interface zur Verfügung, um auf die Datenbank aus diversen Programmiersprachen zuzugreifen und eigene REST-Erweiterungen bereitzustellen (später mehr dazu). Zusätzlich steht hier der genannte Browser zur Verfügung.
- **1337**
  Der Port, auf den Kommandozeilen-Befehle zugreifen, etwa „neo4j-shell" oder „neo4j-import". Dieser Port ist standardmäßig deaktiviert und sollte nur für Zugriffe von der lokalen Maschine aktiviert sein.

```
create (g:Group {name:"Sample people"})
create (g:Group {name:"Federation of Planets"})
create (p:Person {firstNames:["James","Tiberius"],last
Name:"Kirk"})
create (p:Person {firstNames:["Max"],lastName:"Muster
mann"})
match (p:Person {lastName:"Kirk"}),(g:Group
{name:"Federation of Planets"}) create (p)-[:MEMBER_
OF]->(g)
match (p:Person {lastName:"Mustermann"}),(g:Group
{name:"Sample People"}) create (p)-[:MEMBER_OF]->(g)
```

Listing 1

```
id,name,member_of
management,Management
sales,Sales,management
marketing,Marketing,management
development,Development,management
operations,Operations,management
rnd,Research and Development,development
```

Listing 2

Abbildung 4: Tabellarisches Ergebnis

- **7687**
  Zugriff per „BOLT". Dies ist in der Version 3 hinzugekommen und die präferierte Methode für den Zugriff auf die Datenbank, da „BOLT" im Gegensatz zu „REST" binär ist und stehende Verbindungen hat – dadurch ist „BOLT" schneller. „REST" steht aber weiterhin zur Verfügung, da es quasi von jeder Technologie verwendbar ist.

Community und Enterprise

Neo4j existiert sowohl in einer lizenzfreien Community als auch in einer lizenzpflichtigen Enterprise-Version. Letztere bietet zusätzlich unter anderem die folgenden Eigenschaften:

- **Clustering**
  Betrieb auf mehreren Knoten für erhöhte Ausfallsicherheit und Lastverteilung
- **Hot Backups**
  Backup der Daten im Betrieb, ohne die Datenbank zu stoppen

Für den Betrieb einer professionellen Lösung sind die genannten Punkte essenziell, daher sollte man Budget für entsprechende Lizenzen einplanen. Projekte lassen sich problemlos mit der Community-Version starten, ein Umstieg zu einem späteren Zeitpunkt ist erfahrungsgemäß unproblematisch. Beide Versionen sind immer auf dem gleichen Stand.

Die Neo4j Community Edition ist Open Source und die Quelltexte stehen auf GitHub *[5]* zur Verfügung. Jeder hat dadurch die Möglichkeit, die Quelltexte zu forken, die Funktionsweise zu verstehen, aber auch selbst Änderungen vorzunehmen und über einen Pull Request sogar die eigenen Änderungen in das Produkt zu übernehmen. So sieht ein vorbildliches kommerzielles Open-Source-Projekt aus.

```
id,name,departments,headquarter
ny,New York,management;sales,true
sf,San Francisco,marketing;development,false
la,Los Angeles,operations;development;sales,false
ho,Honolulu,rnd,false
```

Listing 3

Import von Daten

Zur Verdeutlichung dienen beispielhaft die folgenden beiden CSV-Dateien. „department.csv" beschreibt Abteilungen in einer Hierarchie *(siehe Listing 2)*, während „subsidary.csv" die Niederlassungen erläutert, die jeweils ein oder mehrere Departments haben können *(siehe Listing 3)*.

Die CSV-Dateien enthalten jeweils eine Kopfzeile mit den Spalten-Namen. Um diese zu importieren, können sie einfach in das „import"-Verzeichnis von Neo4j kopiert werden. Von dort aus werden sie mit einem Befehl geladen *(siehe Listing 4)*.

Die CSV-Datei wird geladen, die erste Zeile wird als Header verwendet und gibt die Namen vor. Jede Zeile wird in die Variable „line" geladen;

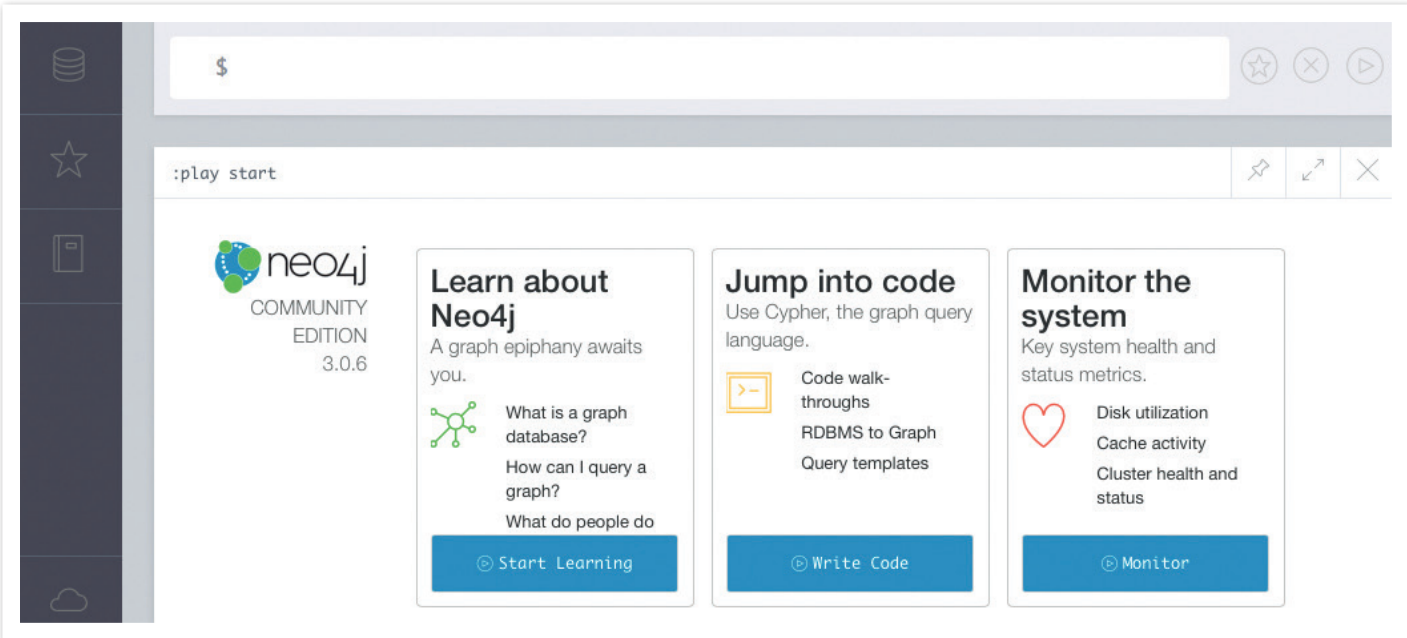


Abbildung 5: Der Neo4j-Browser

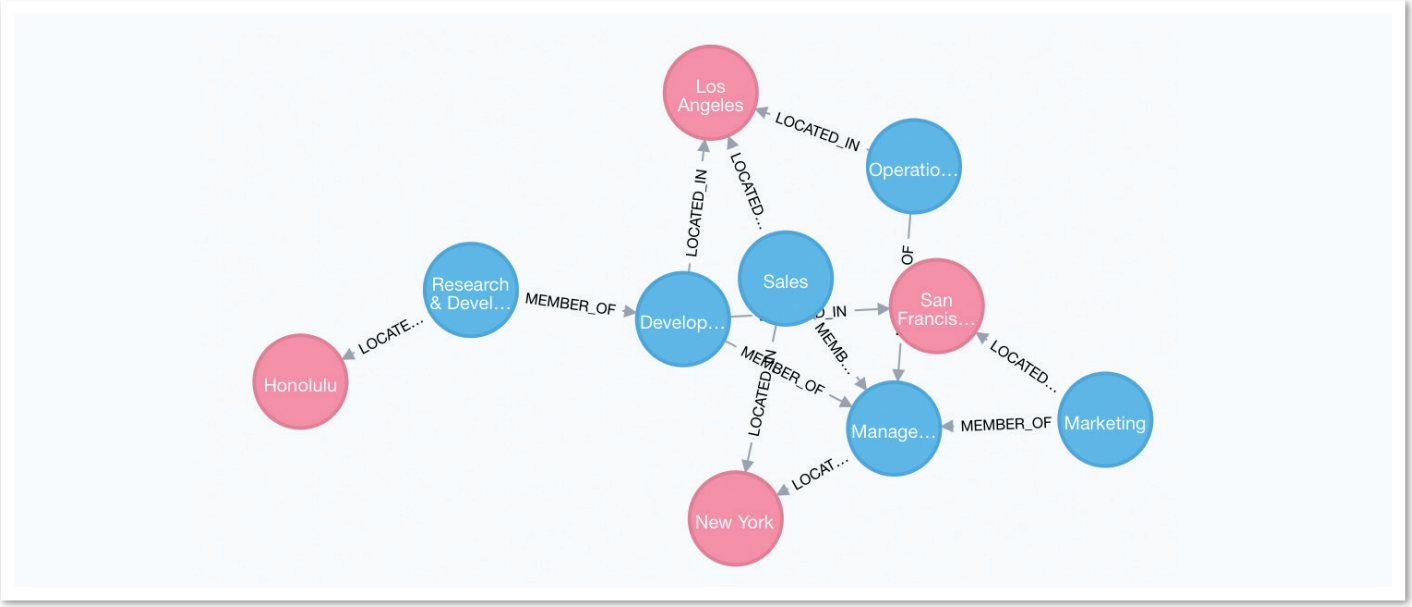


Abbildung 6: Der vollständige Beispiel-Graph

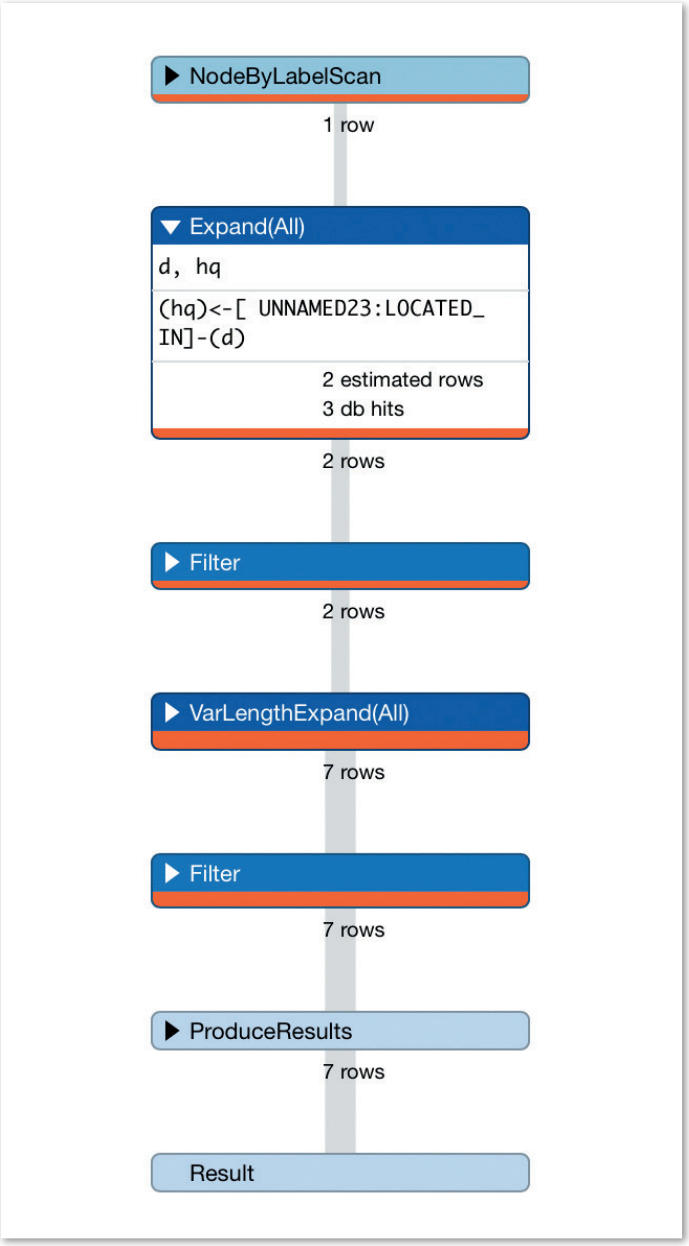


Abbildung 7: Ergebnis eines „profile" in Neo4j

über einen „merge" werden die Knoten angelegt oder über die „id" aktualisiert. Mit dem Befehl „match (p:Department),(c:Department) where p.id = c.member_of create (c)-[:MEMBER_OF]->(p);" werden die Departments von Kind zu Vater als „MEMBER_OF" verknüpft. Den Headquarter können wir mit einem zusätzlichen Label auf dem betreffenden Knoten abbilden: „match (s:Subsidary {headquarter:"true"}) set s:Headquarter;".

Die Properties „MEMBER_OF", „Departments" und „Headquarter" benötigen wir nicht mehr. Da sie als explizite Kanten abgebildet sind, entfernen wir sie mit „match (d:Department) remove d.member_of;" und „match (s:Subsidary) remove s.headquarter, s.departments;".

Die Niederlassungen importieren wir mit „load csv with headers from "file:/subsidary.csv" as line merge (s:Subsidary {id:line.id}) set s.name = line.name, s.departments = split(line.departments,";")". Die Spalte „Departments" wird mit einem Split gleich in mehrere Werte zerteilt. Jetzt verknüpfen wir „Departments" mit „Subsidary" als „LOCATED_IN": „match (d:Department) match (s:Subsidary) where d.id in s.departments create (d)-[:LOCATED_IN]->(s);". *Abbildung 6* zeigt das Ergebnis.

Testweise können wir jetzt die folgenden Abfragen verwenden: „match (d:Department)-[:LOCATED_IN]->(s:Subsidary) return d.name,collect(s.name)" zeigt die Liste aller Abteilungen und die Niederlassungen, an denen sie vorkommen. Eine Stärke von Cypher ist, dass die Pfade beliebig lang sein dürfen. Welche Abteilungen stehen direkt oder indirekt unter dem Headquarter? „match (hq:Headquarter)<-[:LOCATED_IN]-(d:Department)<-[:MEMBER_OF*0..2]-(d2:Department) return hq,d,d2" liefert das Ergebnis.

Der Quantifier „*0..2" erlaubt die Entfernung von null bis zwei Knoten; Neo4j ist also in der Lage, die Pfade selbst zu finden. Damit sind Traversierungen über hierarchische Datenstrukturen sehr einfach. Cypher-Statements lassen sich damit wiederverwenden, unabhängig davon, von welcher Ebene aus man startet. Wenn man dann komplexe Abfragen erstellt, die über große Datenmen-

gen arbeiten, kann man sich durch ein Voranstellen von „profile“ und „explain“ entsprechende Details darstellen lassen *(siehe Abbildung 7)*. Mit diesen Informationen kann man die Zugriffs-Logik und -Reihenfolge nachvollziehen, um Performance-Probleme zu lösen, etwa über Indizes.

Erweiterbarkeit

Es gibt mehrere Möglichkeiten, den Funktionsumfang von Neo4j zu erweitern – etwa Procedures, die sich mit „call“ aufrufen lassen. Der Befehl „call dbms.procedures()“ liefert eine Liste der aktuell verfügbaren Procedures. Einige sind bereits in der Grundinstallation enthalten. Beispielsweise gibt „call db.labels()“

eine Liste der Labels aller Nodes in der Datenbank aus, erwartungsgemäß Subsidary, Department und Headquarter. Diese Procedures können selbst entwickelt werden. Zuvor sollte man sich Awesome Procedures for Neo4j 3 (APOC) *[3]* anschauen, wo bereits etliche gut wiederverwendbare Procedures stehen. Nach dem Download packt man das JAR-File in das Plug-in-Verzeichnis und startet Neo4j neu. Mit „apoc.index.create“ lässt sich ein Volltext-Index auf die Namen der Knoten erzeugen: „CALL apoc.index.addAllNodes('name',{ Department: ["name"], Subsidary: ["name"] })“. Dabei werden auch die bereits vorhandenen Knoten indiziert.

Mit „call apoc.index.search("name","*lo*")“ findet man alle Knoten, deren Name „lo“ beinhaltet. Außerdem erhält man erwartungsgemäß das Subsidary „Los Angeles“ und die beiden Departments, in denen Development vorkommt. Bei dieser Antwort sieht man übrigens eine Stärke von Cypher; Antworten können beliebige Knoten und Kanten gemischt enthalten.

APOC bietet sehr viele nützliche Funktionen an, so lassen sich mit „match (s:Subsidary) call apoc.spatial.geocodeOnce(s.name) yield location set s.longitude = location.longitude, s.latitude = location.latitude;“ die Subsidaries geokodieren. Dabei erhalten wir die Geo-Koordinaten als „longitude“ und „latitude“ und speichern diese als weitere Properties. Nun können wir diese Information nutzen, um die Entfernungen zwischen zwei Niederlassungen derselben Abteilung zu berechnen *(siehe Listing 5* und *Abbildung 8)*.

Für diese Berechnung kommen die neuen Funktionen „point“ und „distance“ zum Einsatz, die ab Version 3.0 verfügbar sind. APOC bieten neben Volltextsuche und Geokodierung etliche weitere Prozeduren an, unter anderem das Laden von Daten aus einer JDBC-Datenquelle oder eine Integration mit anderen Datenbank-Technologien wie ElasticSearch, MongoDB, Couchbase oder Cassandra.

Integration in Anwendungen

Nachdem Neo4j installiert ist und die Daten zur Verfügung stehen, folgt die Integration in die Anwendungen. Prinzipiell kann jede Programmiersprache auf Neo4j zugreifen, die in der Lage ist, REST aufzurufen. Neo4j selbst bietet fertige Treiber für Java, JavaScript, .Net und Python an. Diese vereinfachen den Zugriff auf die Datenbank. Uns interessiert im Kontext dieser Zeitschrift der Treiber für Java.

Alle Artefakte, die wir benötigen, stehen direkt im Maven Central Repository *[6]* zur Verfügung und können daher direkt in Maven oder Gradle verwendet werden.

Neo4j kann übrigens „standalone" laufen oder „embedded" innerhalb der Anwendung. Letztere ist keine eingeschränkte Version, sondern vollständig; sie eignet sich daher für Frameworks wie Spring Boot und natürlich auch für das Testen mit JUnit. Für produktive Anwendungen ergibt es allerdings mehr Sinn, Neo4j „standalone" zu starten, idealerweise auf einer dedizierten Maschine.

Für eine optimale Integration in Java steht Neo4j-OGM (Object Graph Mapper) *[7]* zur Verfügung, um das Mapping zwischen Java-Objekten und Knoten/Kanten in der Datenbank zu vereinfachen. Ähnlich wie JPA werden dafür entsprechende Annotationen, und zwar „@NoteEntity" für Knoten und „@RelationshipEntity" für Kanten, verwendet. Zusätzlich existiert mit Neo4j-Harness eine Erweiterung für JUnit, die es sehr einfach ermöglicht, Neo4j-Instanzen „embedded" über die in JUnit bereitgestellten „@Rule"-Annotationen zu integrieren und damit das Testen zu vereinfachen. Als Erstes benötigen wir ein paar Maven-Dependencies. Die Code-Beispiele sind unter *[8]* zu finden. Im Einzelnen sind das:

- **neo4j-ogm-core**
  Der Hauptteil von Neo4j-OGM
- **neo4j-ogm-bolt-driver**
  Zugriff auf die Datenbank mittels BOLT. Einige Abhängigkeiten auf „org.neo4j" müssen ausgeschlossen werden, da sie aktuell noch auf eine veraltete Neo4j-Version zeigen.
- **neo4j-ogm-http-driver**
  Wird nicht benutzt; Neo4j-OGM beschwert sich allerdings, wenn er fehlt.
- **neo4j-harness**
  Das Neo4j-Harness-Test-Framework, das eine passende „@Rule" liefert, bringt auch transitiv die zurzeit aktuelle Neo4j 3.0.6 selbst als Abhängigkeit mit.
- **Junit**
  Das JUnit-Test-Framework: Für die beiden Entitäten müssen wir Department und Subsidary als DTOs anlegen und mit @NodeEntity annotieren. Listing 6 zeigt beispielsweise „Department.java".

Es handelt sich um eine ganz normale DTO-Klasse. „@GraphId" definiert das Field, das die interne, von Neo4j automatisch zugewiesene ID enthält. Diese wird für die interne Verwaltung gebraucht. Die Verbindung zu Subsidary vom Typ „LOCATED_IN" wird mit einem „Set<Subsidary>" und der Annotation „@Relationship(type="LOCATED_IN")" abgebildet.

Die Richtung kann mit „direction" angegeben werden, wobei „OUTGOING" der Standardwert und daher nicht angegeben ist. Ausgehende Kanten und die dazugehörigen Knoten werden von OGM automatisch geladen, sie sind daher implizit „EAGER". Die Kante an sich erhält man hier nicht. Um die Eigenschaften der Kante zu erhalten, kann man eine weitere DTO hinzufügen und diese mit „@RelationshipEntity" annotieren. Die Geschäftslogik wird entsprechend in eine Klasse „DepartmentService" gepackt *[8]*.

„DepartmentService" erhält im Constructor eine Session. In Frameworks, die Dependency Injection unterstützen, wie Java EE 6 (mittels CDI) oder Spring, kann man sich dieses Objekt injizieren lassen. Session ähnelt dem JPA EntityManager, enthält die passenden Methoden zum Finden/Laden und Speichern von Objekten und arbeitet mit Entitäten, die ,wie oben gezeigt, mit „@NodeEntity" oder „@RelationshipEntity" annotiert sind.

Die Methode „findByName" zeigt, wie „session.query" verwendet wird, um eine Cypher-Query ausführen zu lassen, dabei werden Variablen wie „{name}" durch die Inhalte der übergebenen Map ersetzt. Das Ergebnis ist ein Department. Das Mapping zu diesem Objekt wird von OGM erledigt. Man sieht hier weder die internen Datenstrukturen noch die Kommunikation mittels REST oder BOLT. In *[8]* ist der Test zu sehen.

Die „@Rule" Neo4jRule startet Neo4j als Embedded-Version, die Authentifizierung wird deaktiviert. Standardmäßig sind HTTP und BOLT aktiviert und BOLT lauscht in diesem Fall abweichend auf Port 5001. Mit „copyFrom" wird ein bereits vorhandenes Datenbank-Verzeichnis geladen. Dazu wurden die „standalone"-Version von Neo4j gestoppt und der Inhalt von „$NEO4J_HOME/data/databases/graph.db" in das Projekt nach „src/main/resources" kopiert. Es ist sehr wichtig, dass Neo4j nicht läuft, während man die Daten wegkopiert. Neo4rule kopiert nun bei jedem Starten den Inhalt dieses Verzeichnisses in ein temporäres Verzeichnis und löscht dieses nach dem Beenden. Dadurch bleibt der Inhalt der Datenbank unverändert. Es handelt sich dabei um eine Best Practice, da es sehr einfach ist, ein bestimmtes komplexes Datenbank-Szenario als Grundlage für die Tests zu erzeugen.

```
load csv with headers from "file:/subsidary.csv" as
line merge (s:Subsidary {id:line.id}) set s.name
= line.name, s.departments = split(line.depart-
ments,";"), s.headquarter = line.headquarter;
```

Listing 4

```
match (s1:Subsidary)--(d:Department)--(s2:Subsidary)
where id(s1) < id(s2) with d as d, s1 as s1, s2 as s2,
ceil(distance(point({longitude: s1.longitude, lati-
tude: s1.latitude}),point({longitude: s2.longitude,
latitude: s2.latitude})) / 1000) as kilometers return
d.name,s1.name,s2.name,kilometers
```

Listing 5

d.name	s1.name	s2.name	kilometers
Development	San Francisco	Los Angeles	560
Sales	New York	Los Angeles	3942

Abbildung 8: Tabellarisches Ergebnis der Entfernung einzelner Abteilungen

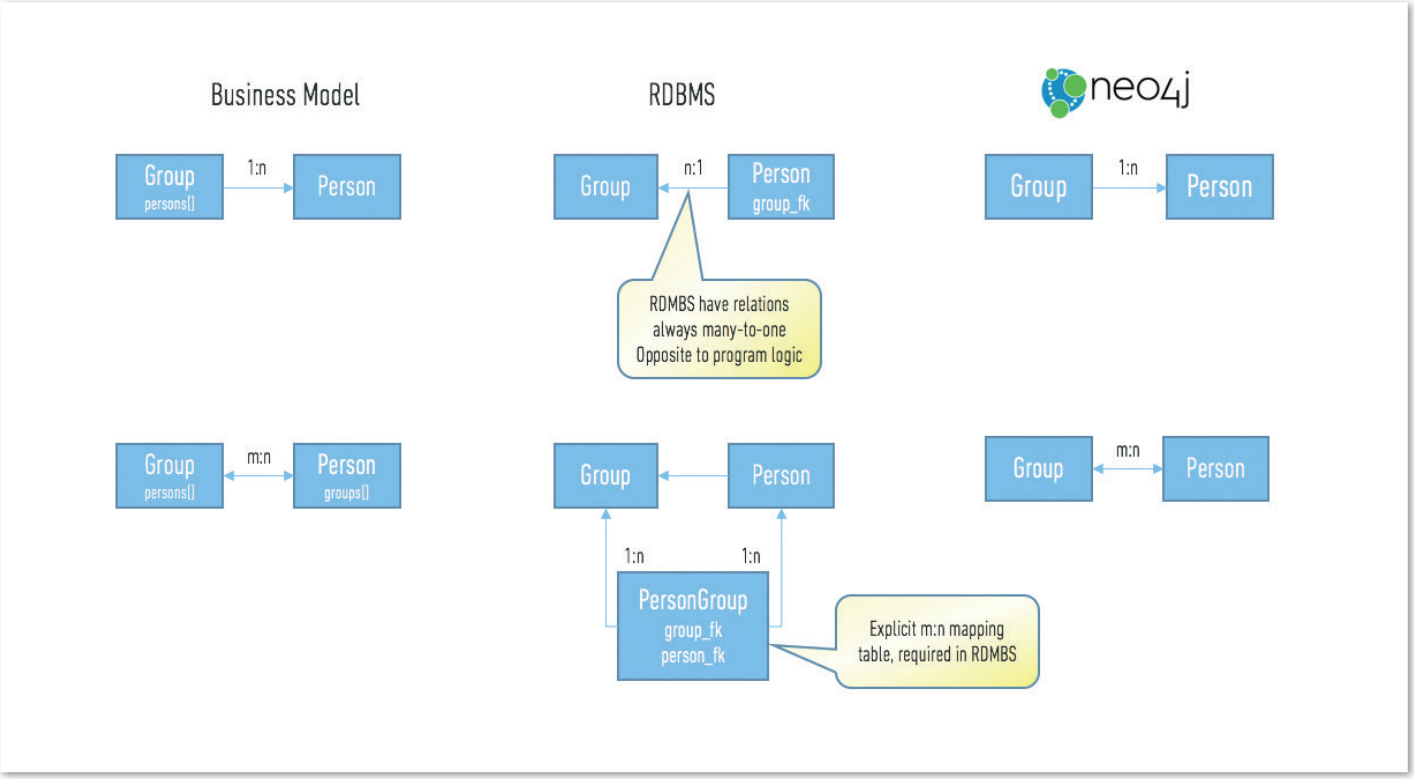


Abbildung 9: Neo4j kann fachliche Beziehungen direkt als Relationship abbilden

In unserem Fall ist der Graph aus den Beispielen enthalten. Die Methode „createSession" erzeugt eine Session für OGM, die „DepartmentService" benötigt. „SessionFactory" erhält in diesem Fall den Parameter „com.prodyna.neo4j.sample", das Basis-Package, in dem unsere Entitäten zu finden sind. Neo4j-OGM scannt alle Klassen in diesem Package, um die Annotationen zu ermitteln. Es werden der BOLT-Driver verwendet und auf Port 5001 (den BOLT Port) zugegriffen. Zusätzlich wird mit „setEncryptionLevel("NONE")" jede Art von Security und Zertifikatsprüfung ausgeschaltet.

Die Testmethode „readDepartmentWithSubsidaries" holt sich mit „createSession" eine Session, erzeugt damit eine Instanz von „DepartmentService" und kann sie dann verwenden. In diesem Fall wird geprüft, dass sechs Departments gefunden werden und das Department Sales sowohl in Los Angeles als auch in New York zu finden ist.

Es existiert noch Spring Data Neo4j (SDN), das auf Neo4j-OGM aufbaut und damit Neo4j in das Konzept der Repositories von Spring Data integriert, wo grundsätzlich keine Implementierungen notwendig sind, da alles deklarativ ist. Die Klasse „DepartmentService" könnte man als „DepartmentRepository" dann wie in *Listing 7* implementieren.

SDN empfehlt sich, wenn man sowieso auf der Spring-Plattform entwickelt. Dann kann man insbesondere mit Spring Boot und einer „embedded" Neo4j schnell und einfach auszurollende und startende Anwendungen bauen, die sich auch sehr einfach in Docker-Container integrieren lassen. Dies ist ideal für Microservices.

Sowohl Neo4j-OGM als auch SDN erfordern ein Scannen der Klassen. In Umgebungen, in denen Java keinen direkten Zugriff auf die Klassen und JAR-Dateien hat, wie OSGi oder Java EE, hat sich das teilweise als sehr kompliziert ergeben. Während SDN damit noch relativ gut umgehen kann, ist die Implementierung von Neo4j-OGM etwas holprig und verursacht teilweise große Probleme mit der Auswirkung, dass es die Entitäten nicht findet. Um jedes Scannen zu vermeiden, könnte man die annotierten Klassen bei der Instanziierung übergeben.

Fazit

Neo4j ist eine hochinteressante Datenbank, die ihre Popularität mit Recht erhält. Die Tatsache, dass alle Artefakte bereits im Maven Central Repository und die Quelltexte im GitHub verfügbar sind, erleichtert die Entwicklung enorm. Auch die Verfügbarkeit einer Community-Edition sorgt dafür, dass ein Projekt schnell starten kann und eine kommerzielle Lizenz sowie ein Umstieg auf die Enterprise Version später einfach möglich sind.

Die verfügbaren REST-Schnittstellen lassen eine Integration in beliebige Technologien und Programmiersprachen zu, wobei Treiber für gängige Plattformen, insbesondere Java, die Entwicklung von Anwendungen erleichtern. Einzig das unglückliche Scan-Verhalten von Neo4j-OGM (siehe oben) sorgt für Probleme in diversen Umgebungen.

```
@NodeEntity
public class Department {
    @GraphId
    private Long graphId;
    private String id;
    private String name;
    @Relationship(type="LOCATED_IN")
    private Set<Subsidary> subsidaries = new
HashSet<Subsidary>();
    // getter und setter, hashCode und equals und
evtl. toString
}
```

Listing 6

Die wahre Stärke von Neo4j ist natürlich ihr Graphenmodell und die Tatsache, dass sie vollständig transaktional ist – sie folgt, im Gegensatz zu vielen NoSQL-Datenbanken, dem ACID-Konzept. Daher gibt es mit Transaktionalität keine Probleme, eine Anforderung, die im Geschäftsumfeld häufig benötigt wird. Relationale Modelle lassen sich problemlos auf das Graphenmodell abbilden, aber mit den genannten Eigenschaften, wie mehrere Labels pro Knoten, lassen sich wesentlich komplexere Modelle abbilden, die am Ende immer noch beherrschbar sind.

Auch viele unvorhersehbare Anforderungen lassen sich meist ohne viel Aufwand integrieren. Ein typisches Beispiel wäre: „Eine Person kann Mitglied mehrerer Gruppen sein". Ein typisches Problem, bei dem eine „1:N"-Relation zu einer „M:N"-Relation umgebaut werden muss. Bei relationalen Datenbanken ist dafür eine technische Hilfstabelle notwendig, die eine „M:N"-Relation in zwei „1:N"-Relationen abbildet *(siehe Abbildung 9)*.

Bei Neo4j gibt es dieses Problem nicht, es können beliebig viele Kanten zwischen zwei Knoten existieren, es gab also vorher eher eine künstliche Beschränkung, die aufgehoben werden kann. Bei relationalen Datenbanken gibt es zusätzlich die Eigenschaft, dass bei „1:N"-Relationen die Relation immer von der „N"-Seite auf die „1"-Seite zeigen muss (Fremdschlüssel). Bei einer Graph-Datenbank kann der Entwickler die Richtung selbst festlegen; es spielt für die Datenbank keine Rolle. Daher kann das Graphenmodell wesentlich näher an das fachlichen Modell angelehnt sein, ohne auf Hilfskonstruktionen zurückgreifen zu müssen.

Die Abfragesprache Cypher sieht auf den ersten Blick kompliziert aus, nach etwas Übung schreibt man jedoch komplexe Abfragen, ohne nachzudenken. Es existieren viele Funktionen, die auch sehr komplexe Abfragen erlauben. Zusätzlich kann der Umfang durch fremde oder eigene Procedures beliebig erweitert werden.

Die Möglichkeit, selbst Pfade zu finden, erlaubt es beispielsweise, problemlos mit hierarchischen Strukturen oder mit Ontologien zu arbeiten. Bei relationalen Datenbanken sind teilweise viele SQL-Abfragen notwendig, die zusätzlich mit Java-Logik integriert sein müssen. Bei Cypher reicht meist ein einziges Statement aus.

Eine Eigenschaft relationaler Datenbanken ist das Problem, dass Antworten selbst immer eine Tabelle sein müssen; bei komplexen Abfragen ist also die Tabelle denormalisiert und enthält Redundanzen, die im Programm dann wieder verarbeitet sein müssen. Bei Cypher sind die Ergebnisse von Abfragen auf Wunsch auch tabellarisch, vorzugsweise sind es jedoch Aufzählungen von Knoten und Kanten, mit dem Verweis darauf, wie die Knoten untereinander verbunden sind.

Wie fast alle NoSQL-Datenbanken ist die Technologie proprietär. Selbst ein Umstieg auf eine andere Implementierung würde einen hohen Aufwand bedeuten, da etwa die Abfragesprache Cypher nicht genormt ist. Neo Technologies (der Hersteller von Neo4j) möchte dem entgegenwirken und hat die Sprache Cypher als OpenCypher *[9]* zur Verfügung gestellt mit dem Ziel, dass auch andere Hersteller von Graph-Datenbanken Cypher (wenn auch zusätzlich zur vorhandenen) als Abfragesprache verwenden. Damit wäre eine Migration zwischen verschiedenen Herstellern wesentlich leichter und auch Frameworks wie SDN könnten besser wiederverwendet werden.

```
public interface DepartmentRepository extends
GraphRepository<Department> {
    @Query("match (d:Department {name:{name}}) return
d")
    Department findDepartmentByName(@Param("name")
String name );
}
```

Listing 7

Neo4j ist clusterfähig, es laufen also mehrere Instanzen. Man muss sich bewusst sein, dass Neo4j kein Sharding unterstützt – jede Neo4j-Instanz hält eine Kopie aller Daten. Der Grund ist einfach: Bei Graphen gibt es keine gute Logik dafür, wie die Knoten auf verschiedene Instanzen aufgeteilt werden könnten. Die Clusterfähigkeit ist aber essentiell für eine hochverfügbare Umgebung und steht auch nur in der Enterprise-Version zur Verfügung.

Weitere Informationen

[1] *http://nosql-database.org*
[2] *http://debian.neo4j.org*
[3] *https://github.com/neo4j-contrib/neo4j-apoc-procedures*
[5] *https://github.com/neo4j/neo4j*
[6] *http://search.maven.org*
[7] *https://neo4j.com/docs/ogm-manual/current/*
[8] *https://github.com/dkrizic/neo4j-sample*
[9] *http://www.opencypher.org*

Darko Križić
darko.krizic@prodyna.com

Darko Križić ist Gründungsmitglied von PRODYNA und in der Funktion des Chief Technical Officers tätig. Damit bildet er die Brücke von der Software-Entwicklung über den hochverfügbaren Betrieb kritischer Anwendungen bis zur strategischen Management-Beratung. In seiner Freizeit entspannt er sich als Privatpilot bei einem Rundflug oder ist auf dem Mountainbike unterwegs.

Analyse von IAM-Anwendungslogs

Tobias Hülsken, Roland H. Steinegger, Dr. Sebastian Abeck, Karlsruher Institut für Technologie

Florian Röser, Dr. Nadina Hintz, iC Consult Gesellschaft für Systemintegration und Kommunikation mbH

Die Anwendungslogs der Identitäts- und Zugriffs-Managementsysteme (engl. Identity and Access Management, IAM) einer Web-Anwendung können zentrale Informationen über Zugriffsprobleme oder Angriffe liefern. Der Artikel zeigt die Konfiguration von ElasticSearch, Logstash und Kibana für ein großes Automobil-Unternehmen. Zudem ist dargestellt, wie Logs mit Watcher live überwacht werden können und wie Timelion zur grafischen Aufbereitung und Auswertung genutzt werden kann. Abschließend sind die Erfahrungen mit den Werkzeugen sowie Lösungen für aufgetretene Probleme zusammengefasst.

Wer kennt die Situation nicht? Der Kunde meldet, dass er sich nicht an der Web-Anwendung anmelden kann oder dass ihm der Zugriff verwehrt wird. Vermutlich liegt ein Problem in den Identitäts- und Zugriffs-Managementsystemen vor. Jetzt heißt es, den Fehler möglichst schnell zu lokalisieren und zu beheben.

Die Identifikation der Problemstelle in einer verteilten Web-Anwendung kann sich als schwieriges Unterfangen erweisen. Meist sind Anwendungslogs die Hauptquelle für Informationen über die vergangenen Anfragen. Im schlimmsten Fall sind diese nicht zentral gespeichert und der Fehler bezieht sich nicht nur auf ein einzelnes System. Im besten Fall ruft der Kunde gar nicht erst an, weil der Fehler automatisch erkannt und bereits behoben wurde.

Anhand eines Szenarios wird das Zusammenführen von Logs verteilter Anwendungen an einer zentralen Stelle gezeigt und ebenso, wie durch eine Bedrohungsanalyse die Auswertungsgrafiken angepasst werden können. In diesem Beispiel kommen die bekannten Open-Source-Vertreter ElasticSearch, Logstash und Kibana, der ELK-Stack, zum Einsatz. Sie sind ergänzt um Watcher, um automatisch Probleme der Web-Anwendung zu erkennen, sowie um das Kibana-Plug-in Timelion, um das Auffinden von Problemen durch den Vergleich von Zeitreihen zu erleichtern.

ElasticSearch, Logstash und Kibana

Der ELK-Stack besteht im Grundaufbau aus den Tools ElasticSearch, Logstash und Kibana. Diese drei Tools kommunizieren jeweils über eine Web-Schnittstelle miteinander. Damit sind sie austauschbar und mit Eigen- oder Fremd-Entwicklungen kombinierbar. Zudem ist die Einbindung von Plug-ins zur Erweiterung der Funktionalitäten möglich *(siehe Abbildung 1)*. Ihr Aufgabenbereich beginnt beim Einlesen der Log-Dateien und endet mit der visuellen Darstellung.

Die Verarbeitungspipeline beginnt mit Logstash *[1]*. Dessen Zielsetzung ist die Vorbereitung der Logdaten für die Analyse. Dazu ist das Tool in eine dreistufige Pipeline eingeteilt. Die erste Stufe ist „Input". Auf ihr werden die Daten zunächst mithilfe von Filebeat (einer leichtgewichtigen Logstash-Variante) zentralisiert. Unterstützte Eingabe-Quellen sind beispielsweise Sensoren, IoT-Komponenten, Logs und Datenstreams. Zudem sind bereits grundlegende Funktionen wie das Zusammenfügen von mehrzeiligen Logs möglich.

Die zweite Stufe ist „Filter". Dieser Bereich ist für die Überführung der Log-Dateien in das einheitliche und normalisierte Datenaustausch-Format JSON gedacht. Um dies zu bewerkstelligen, gibt es zahlreiche Logstash-Plug-ins *[2]*. Es existieren unter anderem Plug-ins zum Anwenden von Regex-Ausdrücken, Parsen von Zeitstempeln oder auch zum Zuordnen von geografischen Lagen anhand der IP-Adresse.

Die letzte Stufe ist „Output". Auf dieser wird das zuvor erzeugte JSON-Dokument weitergeleitet. Hier lassen sich neben der reinen Weiterleitung noch Konfigurationen für die Übertragungsart (UDP, TCP), das Ziel (ElasticSearch, Datei) oder auch für die nachstehenden Dienste, wie die Anzahl der Worker-Threads von ElasticSearch, einstellen. Um noch nicht umgesetzte Funktionalitäten einbinden zu können, ist es möglich, eigene Plug-ins mittels Ruby zu entwickeln und über den Logstash-Plug-in-Manager einzubinden.

Die nächste Stufe der Verarbeitungspipeline ist der Datenspeicher ElasticSearch *[3]*. Er basiert auf Apache Lucene und bietet als NoSQL-Datenbank nahezu Echtzeit-Funktionalitäten. Zu den unterstützten Funktionen zählen das Aggregieren, Suchen und Verwalten der Daten. ElasticSearch ist dezentral aufgebaut. Hierfür besitzt jedes Cluster einen Namen als Kennung. Über diesen können sich neue Knoten selbstständig anschließen. Die Daten haben jeweils

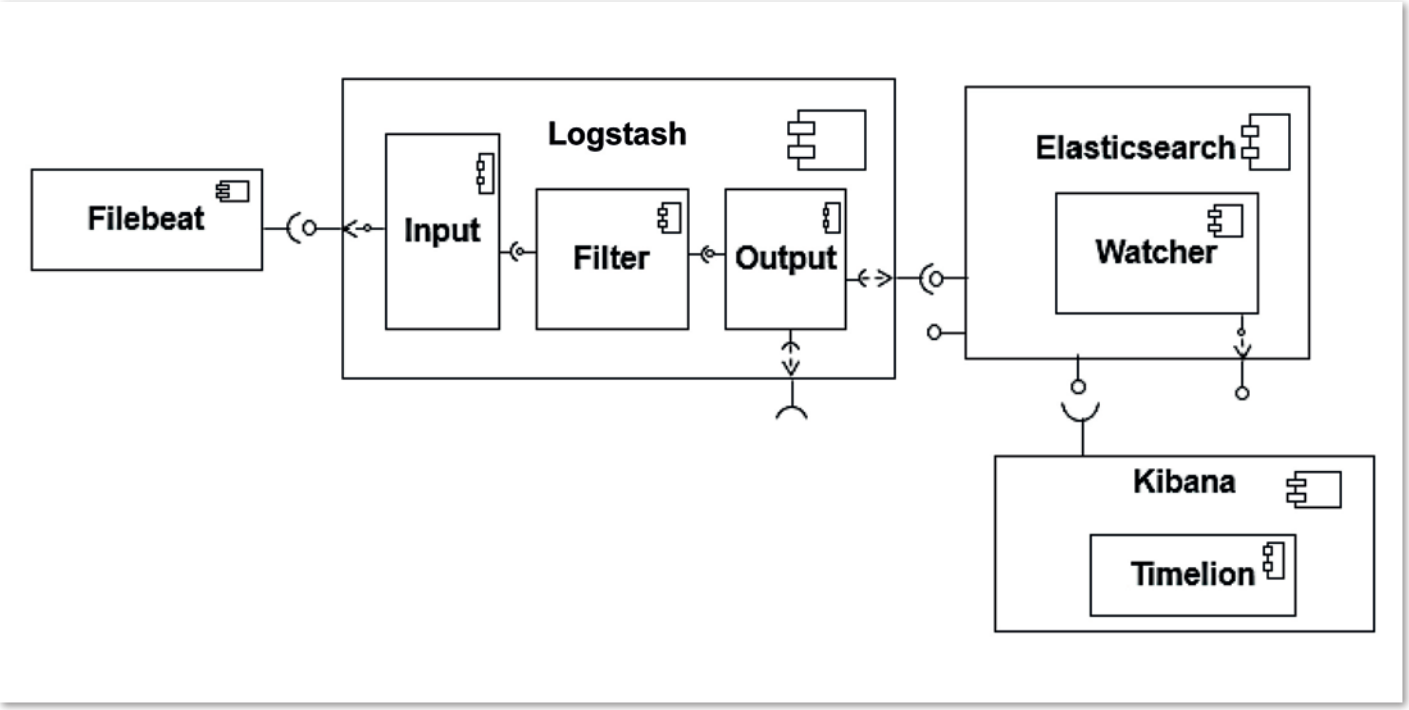


Abbildung 1: Komponenten-Diagramm des ELK-Stacks mit vorgeschaltetem Filebeat zur Logdaten-Sammlung und den Plug-ins Watcher und Timelion

einen Index, der ihre Zugehörigkeit definiert. Zur Verteilung und Replizierung von Daten verwendet ElasticSearch für jeden Index sogenannte „Shards" und „Replikas". Shards sind Teile, auf die die Daten eines Index aufgeteilt sind. Replikas sind Kopien dieser Shards. Shards und Replikas werden dann entsprechend der Konfiguration über der Gesamtanzahl an Knoten dezentral angelegt. Dies erhöht sowohl die Ausfallsicherheit als auch die Geschwindigkeit.

Die letzte Stufe der Verarbeitungspipeline ist das Anzeigetool Kibana *[4]*. Es besitzt eine Web-Oberfläche zum Analysieren und Visualisieren von ElasticSearch-Indizes. Die Oberfläche ist in die drei Bereiche „Discover", „Visualize" und „Dashboard" aufgeteilt. Mithilfe der Discover-Ansicht lassen sich einzelne Einträge anzeigen und filtern. Zudem werden die JSON-Einträge anhand ihrer Schlüssel vorausgewertet, um einen Überblick über die häufigsten Werte zu liefern.
In der Visualize-Ansicht lassen sich Diagramme anfertigen – Schritt für Schritt durch die GUI geleitet. Die Datenbankanfragen können durch das Interface spezifiziert oder manuell mit einer Anfrage in der Apache-Lucene-Query-Syntax erstellt werden.

Dashboard, die letzte Ansicht, dient zur Zusammenstellung mehrerer, in der Visualize-Ansicht erstellter Diagramme. Dies erleichtert den Vergleich verschiedener Daten und das Erkennen von Zusammenhängen.

Watcher

Watcher *[5]* ist ein Plug-in für ElasticSearch, um Anomalien zu erkennen. Diese werden durch sogenannte „Watcher" identifiert. Sie bestehen aus den vier Bestandteilen „Trigger", „Input", „Condition" und „Action". Die Conditions werden periodisch oder nach bestimmten Ereignissen für die aus dem Input stammenden Daten überprüft. Sind die Conditions erfüllt, werden die Actions ausgeführt. Dies kann beispielsweise das Senden einer E-Mail oder auch das Ausführen eines Web-Hook sein, eine auf HTTP aufbauende Server-Kommunikation. Generell lassen sich auch Skripte einbinden. Unterstützte Formate sind unter anderem Groovy oder über weitere Plug-ins auch JavaScript und Python.

Timelion

Timelion *[6]* ist ein Anzeige-Plug-in für Kibana, das im Gegensatz zu den meisten Plug-ins nicht nur neue Diagramm-Typen einführt, sondern eine eigenständige App innerhalb von Kibana ist. Es kann dazu genutzt werden, verschiedene Datenquellen in einem Diagramm zusammenzuführen. Ergänzend zu den bordeigenen Mitteln ist es mit Timelion möglich, mathematische Operationen auf verschiedene Datensätze anzuwenden. So sind einfache Operationen, wie die Addition von Diagramm-Verläufen oder das Hinzufügen von zeitlichen Offsets, möglich. Daten lassen sich auf diese Weise einfacher vergleichen, da ihre Beziehungen und Relationen in einem Diagramm sichtbar sind. So kann beispielsweise das Login-Verhalten bei einem Server wochenweise verglichen werden.

Der ELK-Stack zur Log-Aufarbeitung

Werden die Logs dezentral erzeugt, müssen sie an eine zentrale Logstash-Instanz weitergeschickt oder abgeholt werden, um den späteren Verwaltungsaufwand, beispielsweise das Anpassen der Konfigurationen, zu minimieren. Um dies zu realisieren, kann Filebeat benutzt werden, wobei es ausreicht, den Speicherort der Logs und die Logstash-Instanz anzupassen. In *Listing 1* sind die zu ändernden Stellen (Zeile 2 und 5) als Auszug der Filebeat-Konfiguration aufgelistet.

```
paths:
    %PathToLogfiles%/*.log
output:
  logstash:
    hosts: ["127.0.0.1:5044"]
```

Listing 1

```
input {
  beats {
    port => 5044
    codec => multiline {    #For Multiline Log Entrys
      pattern => "^%{TIMESTAMP_ISO8601} "
      negate => true
      what => "previous"
    }
  }
}
```

Listing 2

```
filter {
  if [message] =~ /^[\s]*$/   #removing empty lines
  { drop {} }
  grok {
    match => { "message" => "%{TIMESTAMP_
ISO8601:timestamp} %{GREEDYDATA:logmessage}" }
  }
  grok {
    patterns_dir => "./custom-grok-patterns"
    match => { "logmessage" =>"\
AR{%{REQUESTID:[producer][requestId]}?}
T{%{TRACKINGID:[producer][trackingId]}
IP{%{IP:clientIP}?}- %{GREEDYDATA:logmessage}" }
    overwrite => ['logmessage']
  }
  yamlgrok {
    patterns_dir => "./custom-grok-patterns"
    path => "./yamlgrok-files/backend.yml"
  }
  mutate {  #Converts the Thread Number and adds the
appname
    convert => { "[producer][thread]" => "integer" }
    add_field => { "[producer][appname]" => "backend"}
  }
}
```

Listing 3

Zudem muss Logstash heruntergeladen und konfiguriert werden. Es reicht aus, das Programm zu entpacken; eine Installation ist nicht nötig. Für die Konfiguration empfiehlt es sich, dieses in drei Bereiche „Input", „Filter" und „Output" aufzuspalten, um die jeweiligen Konfigurationen für andere Projekte oder Indizes wiederverwenden zu können.

Für den ersten Bereich „Input" muss angegeben werden, aus welcher Quelle die Logs stammen (Filebeat, Datei etc.). Als Beispiel wird eine Input-Konfiguration für Filebeat herangezogen, die Daten vom Port 5044 empfängt. Optional können mehrzeilige Einträge in diesem Bereich anhand von Zeitstempeln zusammengefasst werden (siehe *Listing 2*, Zeilen 4 bis 8). Danach kann der Log-Eintrag im Bereich „Filter" geparst werden. *Listing 3* zeigt Auszüge aus einer Filter-Konfiguration.

Zu Beginn werden Leerzeilen entfernt und die Log-Nachricht vom Zeitstempel getrennt. Danach wird die Log-Nachricht in ein hierarchisches Format geparst. Die dafür notwendigen Muster sind der Übersicht und Wiederverwendbarkeit wegen ausgelagert. Danach werden mithilfe von Yamlgrok, einem Logstash-Plug-in zum Identifizieren von String-Bausteinen anhand der Yet Another Markup Language (YAML), erst ein String- und dann ein Regex-Vergleich durchgeführt. Dies erlaubt die Volltextsuche durch Grok *[7]*, einem Logstash-Plug-in, um Daten anhand eines Musters zu strukturieren, zu vermeiden und somit Rechenaufwand zu minimieren. In den Zeilen 20 und 21 wird dem Feld „Thread" der Datentyp „Integer" zugewiesen und das Feld „Appname" erhält den Wert „Backend"; beide sind Attribute des Producer-Objekts, das seinerseits eine Sammlung von Informationen zu der Maschine enthält, die die Logs erstellt. Die Konvertierung zum Datentyp „Integer" spart zum einen Speicherplatz in ElasticSearch, zum anderen ermöglicht es weitere Operationen während der Auswertung in Kibana. Die Zuweisung des Wertes „Backend" als Applikationsname bietet bei der Auswertung das Zuordnen der verschiedenen Log-Quellen zu den einzelnen Applikationen.

Zuletzt ist der Output zu konfigurieren. Im Beispiel in *Listing 4* wird als Zielsystem eine ElasticSearch-Instanz auf dem „localhost" eingestellt. Zudem wird der Index „ciam-access-<Datum>" gesetzt. Die Wahl eines guten Index ist wichtig, da nur über diesen performant aggregiert werden kann.

Der nächste Schritt ist das Installieren und Konfigurieren von ElasticSearch. Dieses ist nach dem Download bereits voll funktionsfähig. Zur Laufzeit können mittels REST-API weitere Einstellungen wie die Anzahl der Shards oder das Löschen von alten Indizes getroffen werden. Als Beispiel soll das Benutzen einer Vorlage dienen, die zur Konfiguration der Datenstruktur und der Feld-Eigenschaften in ElasticSearch zum Einsatz kommt. Damit wird zum einen Typsicherheit garantiert und zum anderen Speicherplatz gespart.

Die Vorlage wird per HTTP-PUT an ElasticSearch geschickt. Sie definiert das Datenformat der Felder des jeweiligen Index und ob diese Felder analysiert werden. Das Analysieren ermöglicht später eine schnelle Volltextsuche nach einzelnen Begriffen innerhalb des jeweiligen Feldes, benötigt jedoch ein Vielfaches an Speicher. Ein nicht analysiertes Feld erlaubt nur noch einen Volltextvergleich *(siehe Listing 5)*.

Um mit Kibana fortzufahren, muss ElasticSearch gestartet sein. Es empfiehlt sich, erste Log-Einträge mit Logstash in ElasticSearch eingespielt zu haben, um die Kibana-Funktionalitäten bereits nutzen zu können. Danach ist Kibana standardmäßig unter dem Port 5601 im Webbrowser verfügbar.

Auf der Startseite von Kibana, etwa „localhost:5601", ist zunächst ein Index anzugeben. Dieser besteht aus einer Zeichenkette und optional einem „*" (Asterisk), um verschiedene Indizes zu vereinen. Die Benutzung ist intuitiv und einfach gehalten.

Im Bereich „Discover" sind links die verschiedenen Felder des Index aufgelistet. Zu jedem Feld lässt sich eine Statistik der fünf häufigsten Werte anzeigen. Zudem werden die einzelnen Einträge aufgelistet, die expandiert und somit übersichtlicher dargestellt werden können.

```
output {
  elasticsearch {
    action => "index"
    hosts => "localhost"
    index => "ciam-backend-%{+YYYY.MM.dd}"
  }
}
```

Listing 4

```
PUT /_template/template_1
{
  "template" : "ciam-backend*",
  "mappings" : {
    "_default_" : {
      "properties" : {
        "clientip" : {"type": "string", "index" :
"not_analyzed" },
      }
    }
  }
}
```

Listing 5

Im „Visualize"-Bereich lassen sich nun diverse Diagramme erstellen. Hierzu müssen als Erstes der Diagramm-Typ (wie Balken- oder Kuchendiagramm) und das darzustellende Feld einer Datenquelle ausgewählt werden. Für das gewählte Feld muss zudem die Aggregationsmethode (etwa als Histogramm) eingestellt sein. Zusätzliche Anpassungen sind der anzuzeigende Wertebereich oder das Ausblenden bestimmter Werte gemäß einem Muster. Das erstellte Diagramm kann daraufhin gespeichert und in der Dashboard-Ansicht genutzt werden. Sowohl in der Visualize- als auch in der Dashboard-Ansicht lassen sich die Zeit-Intervalle der dargestellten Daten einstellen.

Überwachung der Logs mit Watcher

Watcher ist ein kommerzielles Plug-in für ElasticSearch. Dementsprechend ist eine Lizenz zur Installation erforderlich. Nach der Installation wird Watcher beim Ausführen von ElasticSearch automatisch gestartet. Die Watches lassen sich mit dem REST-API oder übersichtlicher mit dem Kibana-Plug-in Sense verwalten.

„Trigger" bestimmt, in welchen Intervallen eine Watch ausgeführt wird. „Inputs" können sowohl HTTP-Anfragen auf einen Webserver als auch Anfrage-Ergebnisse von ElasticSearch sein. Diese lassen sich verketten und aggregieren. „Condition" überprüft, ob die durch „Actions" definierten Aktionen ausgeführt werden oder nicht. Das können einfache numerische Operationen wie „größer oder gleich", textuelle Vergleiche oder komplexere Überprüfungen sein, die als Skript definiert sind. „Actions" beschreiben schließlich die auszuführenden Aktionen. Das können beispielsweise das Senden einer Mail, das Schreiben in eine Log-Datei oder auch das Senden einer Nachricht an einen Webhook sein.

Manuelle Anomalie-Erkennung mit Timelion

Um Timelion zu benutzen, müssen bereits Daten in ElasticSearch abgelegt sein. Zur Installation reichen der Befehl „%Kibana_DIR%/bin/kibana plugin -i elastic/timelion" und der anschließende Neustart von Kibana aus. Timelion wird nun in der Rubrik „Apps" in der Webansicht von Kibana aufgeführt. Die Ansicht in Timelion ist in ein

Feld für Anfragen sowie in einen Bereich aufgeteilt, der maximal 16 x 16 Diagramme (Visualisierungen von Einträgen) enthalten kann.

Um mit einem Diagramm zu arbeiten, muss dieses erst mit „add Chart" erstellt und anschließend ausgewählt werden. Nun lässt sich das Diagramm konfigurieren, indem man die Anfrage inkrementell aufbaut. Die Anfrage beginnt mit dem Index, der mathematische Operationen wie das Dividieren oder Addieren von Anfragen oder auch das Anzeigen mehrerer Graphen innerhalb eines Diagramms erlaubt. Eine der interessantesten Anwendungen ist es, Daten derselben Quelle zu verschiedenen Zeiten, beispielsweise im Abstand von einer Woche, miteinander zu vergleichen.

Die Anfragen in *Listing 6* zeigen die Funktionsweise. In der ersten Anfrage wird die Anzahl der eintreffenden Logs zum aktuellen Zeitpunkt und vor einem Monat in einem Diagramm angezeigt. In der zweiten Anfrage wird die Differenz gebildet und visualisiert. Anhand dessen lassen sich Visualisierungen erzeugen, die eine schnelle und übersichtliche manuelle Auswertung großer Daten ermöglichen.

Erweiterung der Konfiguration

Durch den Umfang des ELK-Stacks und dessen Plug-ins sind zahlreiche Erweiterungen für die Zukunft denkbar und vor dem Produktiv-Einsatz auch erforderlich. Zum einen war es bereits notwendig, eine portable Entwicklungsumgebung zu schaffen, da der Aufwand zum Projekteinstieg und zur korrekten Konfiguration stetig steigt. Dafür wurde auf eine Kombination aus Vagrant und Puppet gesetzt, um die Entwicklungsumgebung auf verschiedenen Systemen ressourcenschonend aufzusetzen und zu verwenden.

Bevor das System für den Produktiv-Einsatz bereit ist, sind noch zahlreiche Konfigurationsänderungen nötig. Zum einen muss die ElasticSearch-Konfiguration aufgrund der Datenmenge angepasst werden, sodass diese auf mehrere Shards und Replikas verteilt sind. Zudem müssen alte Daten aus dem System als Aggregationen gespeichert und ihre detaillierten Informationen gelöscht werden, um das System auf Dauer stabil und performant halten zu können.

Ein anderer Aspekt ist die Sicherheit. Momentan sind alle Verbindungen und Daten frei zugänglich. Hier ist ein Ansatz der Einsatz von Shield, um die Kommunikation zu verschlüsseln und die Daten mit einem Passwort zu schützen.

Fazit

Der ELK-Stack bietet eine gute Grundlage zur Erfassung, Verarbeitung und Darstellung von Logdaten. Durch seine Architektur ist er skalierbar und auf allen verbreiteten Systemen ausführbar. In seiner reinen Form ist es möglich, einfache Überwachungsaufgaben und Spitzen oder Anomalien zu erkennen. Mithilfe von Timelion wird die Analyse durch den Vergleich von Daten erleichtert. Dies bietet beispielsweise die manuelle Auswertung großer Datenmengen. Watcher ergänzt die manuelle Analyse um die Überwachung, basierend auf Regeln in Echtzeit.

Um diese Werkzeuge effizient einsetzen zu können, sind jedoch ein tiefgreifendes Verständnis der Software und ein strukturiertes und durchdachtes Logkonzept nötig. Für die Überwachung und Korrelation mehrerer Applikationen ist es ratsam weitere Werkzeuge wie Drools Fusion *[8]*, ein auf Java basierendes Eclipse-Plug-in, das Log-Ereignisse korreliert und zu einem Event zusammenfasst, zu benutzen.

Weitere Informationen

[1] *https://www.elastic.co/guide/en/logstash/current/introduction.html*
[2] *https://www.elastic.co/guide/en/logstash/current/filter-plugins.html*
[3] *https://www.elastic.co/de/products/elasticsearch*
[4] *https://www.elastic.co/guide/en/kibana/current/introduction.html*
[5] *https://www.elastic.co/guide/en/watcher/current/introduction.html*
[6] *https://www.elastic.co/de/blog/timelion-timeline*
[7] *https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html*
[8] *http://www.drools.org*

Tobias Hülsken
tobias.huelsken@student.kit.edu

Roland H. Steinegger
roland.steinegger@kit.edu

Florian Röser
florian.roeser@ic-consult.de

Dr. Nadina Hintz
nadina.hintz@ic-consult.de

Dr. Sebastian Abeck
sebastian.abeck@kit.edu

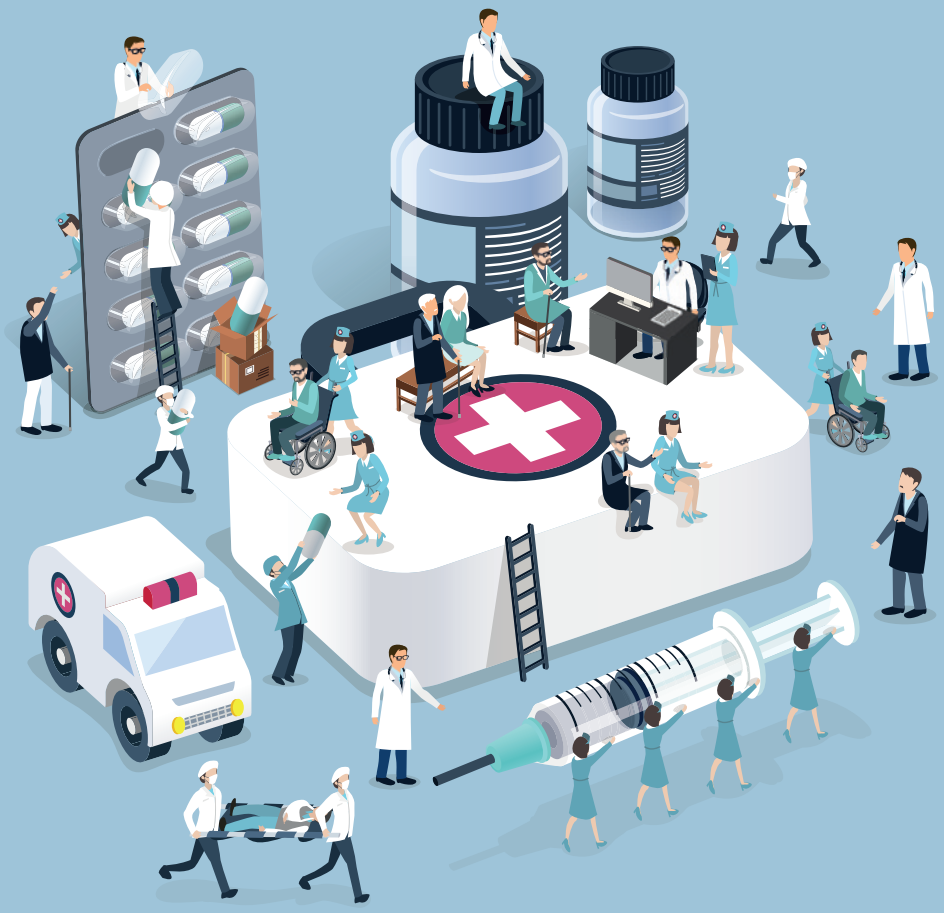
Tobias Hülsken ist Hauptautor des Artikels. Er ist Student mit Schwerpunkten der IT-Sicherheit und Rechnerstrukturen am KIT. Im Rahmen seiner Tätigkeit ist er für die Sicherheit und Instandhaltung der Industrie-4.0-Software-Landschaft zuständig.

Roland H. Steinegger forscht seit vier Jahren im Bereich des Identity und Access Management. Schwerpunkt seiner Arbeit ist der Einsatz von Sicherheitsmustern beim domänenorientierten Entwurf von (Micro-) Serviceorientierten Web-Anwendungen.

Florian Röser ist als Consultant und Software-Engineer bei iC Consult für einen deutschen Automobilhersteller im Bereich Identity und Access Management tätig.

Dr. Nadina Hintz ist als Key-Account Manager für einen Kunden der Automobilbranche bei der IC Consult, einem systemunabhängigen Integrator im Bereich Identity und Access Management tätig. Promoviert hat sie im Bereich IT-Security an der Universität Erlangen-Nürnberg.

Dr. Sebastian Abeck leitet am Karlsruher Institut für Technologie die Forschungsgruppe Cooperation & Management, mit der er in den Bereichen der serviceorientierten und mobilen Web-Anwendungen, dem Identity and Access Management und dem Internet of Things Forschungs- und Projektarbeiten durchführt.



Lebendige Dokumentation mit AsciiDoctor

Markus Schlichting, Canoo Engineering AG

Die Dokumentation von Software-Systemen ist immer wieder eine Herausforderung. Beim Versuch, sie aktuell zu halten und zu jedem Release die passende Version bereitzustellen, haben sich Office-Suiten und Wikis wiederholt als unpraktisch erwiesen. Dieser Artikel zeigt, dass mit AsciiDoctor zusammen mit bekannten Werkzeugen aus der Software-Entwicklung eine gute Alternative verfügbar ist.

Eine Dokumentation zu Software-Projekten zu erstellen und pflegen, ist immer wieder eine Herausforderung. Man muss nicht nur um verfügbare Zeit kämpfen, sondern auch dafür Sorge tragen, dass Implementierung und Dokumentation auf dem gleichen Stand sind. Die Dokumentation mit Tools zu erstellen, die relativ weit von denen der normalen Toolbox der Software-Entwicklung entfernt sind, erweist sich immer wieder als mühsam und beschwerlich. Die Folge ist ein Auseinanderdriften des Standes der Dokumentation und dem implementierten System.

Ein Dokumentations-Format und -Workflow nahe an den bei der täglichen Entwicklung der Software benutzten Werkzeugen hilft, die Lücke kleiner werden zu lassen. Die Hemmschwelle, bei einer Änderung auch gleich die zugehörige Dokumentation anzupassen, wird herabgesetzt.

AsciiDoctor

Ein Format, das sich hier anbietet, ist „AsciiDoc“. Es ist als Format älter, als zumeist vermutet wird, und wurde bereits im Jahr 2002 veröffentlicht. Die dazugehörige Tool-Chain wurde mit Python entwickelt. Die Weiterentwicklung kam jedoch zum Erliegen und das Paket geriet technologisch ins Hintertreffen. Dieser Umstand ist auch daran beteiligt, dass das jüngere und weniger mächtige Format Markdown zu einer wesentlich größeren Popularität gelangt ist.

Bei AsciiDoctor handelt es sich um eine Neu-Implementierung in Ruby. Diese ermöglicht neben einer höheren Wartbarkeit auch die

```
= Beispieldokument
Autor Schreiber <author@schreiber.org>
:icons: font

_Ein Beispiel_ um http://www.asciidoc.org[AsciiDoc] als `plain text`
Format zu *demonstrieren*.

[TIP]
Für die bestmöglichste _AsciiDoc_ Erfahrung footnote:[Wobei man
natürlich die Ausgabeformatierung nicht ausser Acht lassen darf!] ist
die Benutzung von http://www.asciidoctor.org[AsciiDoctor] wärmstens
empfohlen.
Auf icon:twitter[] hört man gerne von Erfahrungen.

== Abschnitt

[square]
* Erster Punkt
* Zweiter Punkt
* der Aufzählung

[source,java]
----
System.out.println("Hello, JavaAktuell !");
```


Abbildung 1: Beispiel-AsciiDoc und erzeugte Ausgabe

unkomplizierte Verwendung auf der JVM, dank JRuby. Die fertig gebündelten Pakete werden mithilfe der Bibliothek AsciiDoctorJ *[1]* bereitgestellt *(siehe Abbildung 1)*.

Während des Schreibens von Dokumentationen mit AsciiDoctor hat sich die QuickReference *[2]* als sehr wertvoll erwiesen. Weitergehende Fragen kann zumeist die detaillierte Dokumentation *[3]* beantworten.

Diagramme

Bilder sagen mehr als tausend Worte, sagt man. Nicht zuletzt im Kontext von Dokumentationen bewahrheitet sich dieses Sprichwort immer wieder aufs Neue. Umso besser die Nachricht, dass AsciiDoctor eine Vielzahl von Dialekten unterstützt, mit denen unterschiedliche Diagramm-Typen aus Ascii-Syntax erzeugt werden können.

Ein Beispiel zeigt das Diagramm in *Abbildung 2*, das aus dem folgenden Ascii-Code erzeugt wurde. Eine Übersicht liefert die Homepage der AsciiDoctor Diagram Extensions *[4]*. Ein nützliches Tool für die Arbeit mit Ditaa-Diagrammen ist *„http://asciiflow.com“*, das die Bearbeitung der Diagramme deutlich vereinfacht *(siehe Listing 1)*.

Helferlein

Wie beim Programmieren ist es auch beim Dokumentieren sehr angenehm, wenn man etwas bei der Arbeit unterstützt wird. Für AsciiDoc bieten mittlerweile alle populären Java-IDEs Plug-ins, mit denen das Arbeiten durch Syntax-Highlighting und eine integrierte Vorschau vereinfacht wird. Wenn man zum Dokumentieren nicht extra die IDE starten möchte, steht mit AsciiDocFX *[5]* ein Featurereicher Editor bereit, der bei Bedarf auch direkt in verschiedene Ausgabeformate exportieren kann.

```
+--------+   +-------+    +--------+
|        | --+ some  +--> |        |
|initial |   +-------+    |simple  |
|    Idea|   |!magic!|    | diagram|
|     {d}|   |       |    |        |
+---+----+   +-------+    +--------+
    :                         ^
    |       Lots of work      |
    +-------------------------+
```

Listing 1: Quelle des Ditaa-Diagramm-Beispiels

Integration: Gradle

Gemäß der eingangs aufgezeigten Motivation soll die Trennung zwischen Dokumentation und Entwicklung möglichst weit reduziert werden, um Aktualisierungen in den natürlichen Workflow einzubinden. Gradle etabliert sich zunehmend als das Build-System für Projekte im Java-Umfeld und bietet sich auch für unsere Zwecke dank seiner großen Flexibilität freundlich an.

Ausgangspunkt ist das AsciiDoctor-Gradle-Plug-in *[6]*, mit dem sich die Erzeugung der Dokumentation im gewünschten Ausgabeformat durch Gradle automatisieren lässt, wie im Beispiel in *Listing 2* gezeigt.

Neben den eingebundenen Plug-ins ist der „asciidoctor“-Block interessant. Spezifiziert man keinen anderen Wert, erwartet das Plug-in die AsciiDoc-Dateien unter „src/docs/asciidoc“. Das Einbinden des „asciidoctorj-pdf“-Plug-ins und die Angabe von PDF und HTML5 für „backends“ erzeugen Dokumente in beiden Formaten. Die Attribute in der Liste „attributes“ sprechen für sich, eine vollständige Übersicht der verfügbaren Attribute findet sich in der AsciiDoc-Dokumentation unter *[7]*. Ein umfangreiches Repository mit weiteren Beispielen steht unter *[8]* bereit.

Die Verknüpfung mit dem Build-System und -Prozess des zu dokumentierenden Systems ermöglicht es, immer die passende Dokumentation zu einem Entwicklungsstand zu liefern, da diese im gleichen Prozess wie das System-Artefakt selbst erzeugt wird.

Continuous Build

Das erste Feature, das die Nützlichkeit der Kombination mit Gradle zeigt, ist Continuous Build *[9]*. Startet man die „asciidoctor“-Task mit „-t“, reagiert Gradle auf eine Änderung an den AsciiDoc-Quelldateien mit der Ausführung des Builds. So werden die Dokumente ständig aktuell gehalten und man kann den Fortschritt der Arbeit im Browser oder PDF-Betrachter ohne den manuellen Umweg über die Kommandozeile direkt kontrollieren.

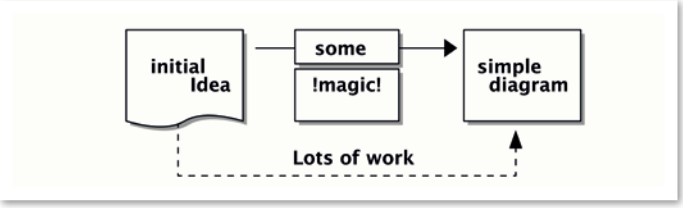


Abbildung 2: Ditaa-Diagramm-Beispiel

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath 'org.asciidoctor:asciidoctor-gradle-
plugin:1.5.3'
        classpath 'org.asciidoctor:asciidoctorj-
pdf:1.5.0-alpha.11'
    }
}

apply plugin: 'org.asciidoctor.convert'

repositories {
    jcenter()
}

asciidoctorj {
    version = '1.6.0-alpha.3'
}

asciidoctor {
    backends = ['html5', 'pdf']

    attributes 'source-highlighter'    : 'coderay',
               'coderay-linenums-mode' : 'table',
               toc                     : 'left',
               icons                   : 'font',
               encoding                : 'utf-8'
}
```

Listing 2

```
attributes  'source-highlighter': 'coderay',
     ...,
     'product-version' : project.version
```

Listing 3

```
= Documentation with AsciiDoc
Markus Schlichting <markus.schlichting@canoo.com>
:revnumber: {product-version}

== Documentation for Version {product-version}
...
```

Listing 4

```
package demo;
class Business {
   // tag::sampleTag[]
   public int calculate(){ return 42; }
   // end::sampleTag[]
}
```

Listing 5

Variablen

Ein weiterer Vorteil, die Erzeugung mit dem Build zu kombinieren, ergibt sich daraus, die beim Build verwendeten Parameter automatisiert in die erzeugte Dokumentation zu übernehmen. So erhält man die gewünschte Konsistenz zwischen der Dokumentation und dem dokumentierten System. Als Beispiel dient die Versionsnummer. Zu den „attributes" im „asciidoctor"-Block im Gradle-Build-Skript kann man diese als eigenes Attribut hinzufügen *(siehe Listing 3)*.

Im Beispiel wird das „product-version"-Attribut angefügt und mit dem Wert der Projekt-Version, einem Standard-Attribut aus dem Kontext des Gradle-Builds, befüllt. Den Wert dieses Attributs kann man entsprechend den Anforderungen im eigenen Projekt praktisch frei setzen. Es kommt unter anderem auch bei der Erzeugung der Projekt-Artefakte und der Publikation selbiger in einem Artefakt-Repository zum Einsatz *(siehe Listing 4)*.

Im Beispiel-Dokument verwenden wir das neue Attribut nun zum einen, um die AsciiDoctor-eigene „revnumber" zu befüllen. Diese wird verwendet, um die Metadaten des Dokuments im Ausgabeformat zu befüllen, etwa bei PDF, oder für bestimmte Felder im Template. Im Standard-HTML-Template wird die „revnumber" im Header des Dokuments verwendet *(siehe Abbildung 3)*. Zum anderen verwenden wir direkt die „product-version"-Variable im Text, eingefasst von geschweiften Klammern.

Source-Code: Schnipsel und Beispiele

Ein häufiges Problem bei der Dokumentation von Software sind in der Dokumentation verwendete Beispiele oder Code-Schnipsel, die bei Änderungen nicht nachgezogen werden oder wegen kleiner Syntaxfehler nicht funktional sind und so beim Leser zu Frustration führen.

AsciiDoctor bietet die Möglichkeit, über „include"-Direktiven Blöcke aus anderen Textdateien einzubinden. Um Verweise auf Quellcode in der Dokumentation zu vereinfachen, definiert man beispielsweise eine Dokument-Variable, die den relativen Pfad zum entsprechenden Verzeichnis beinhaltet: „:sourcedir: ../main/java". Sie kann dann beim eigentlichen Include verwendet werden: „include::{sourcedir}/demo/Business.java[lines=8..10]".

In dem hier gezeigten einfachen Beispiel werden die einzubindenden Zeilen direkt über die Zeilennummer bestimmt. Zwar muss hierzu die Quellcodedatei nicht angepasst werden, die Fehleranfälligkeit dieser Variante ist andererseits offensichtlich: Bei einer kleinen Änderung am Quellcode, auch wenn die referenzierte Stelle gar nicht betroffen ist, etwa bei einem „optimize imports", dem Aktualisieren des Copyright-Kommentarblocks oder dem Einfügen einer weiteren Funktion oberhalb, bricht die Referenz.

Für diese Offensichtlichkeit gibt es eine Lösung: das Einbinden mithilfe von „tags". Die AsciiDoctor-Syntax mit einem „sampleTag" dazu lautet: „include::{sourcedir}/demo/Business.java[tags=sampleTag]". Damit AsciiDoctor den gewünschten Block jedoch finden kann, muss in dieser Variante auch der Quellcode angepasst werden *(siehe Listing 5)*.

Diese Notwendigkeit kann in erster Reaktion Widerstand im Entwicklerteam hervorrufen. Bei genauerer Betrachtung bringt es aber

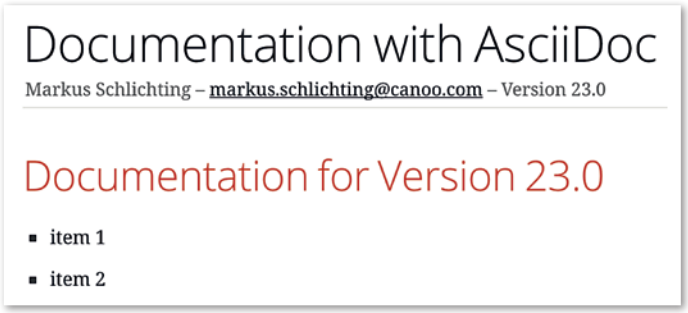


Abbildung 3: Variablen in AsciiDoc

den Vorteil, dass man anhand des Quellcodes erkennen kann, ob er in der Dokumentation referenziert wird. Eine Änderung an einem so markierten Block gibt also mindestens Anlass für ein Review der Dokumentation.

Wenn die in der Dokumentation verwendeten Code-Beispiele auf diese Art eingebunden werden, können (und sollten) sie zuvor auch durch Unit- oder Integrations-Tests überprüft werden. Um diese Abhängigkeit im Build abzubilden, definiert man den „asciidoctor"-Task in Gradle als von den Tests abhängig: „asciidoctor.dependsOn test". So erhält man deutlich zuverlässigere Beispiele in der Dokumentation, als es mit „Copy&Paste"-Blöcken möglich ist. Durch die Verknüpfung von bewiesenermaßen lauffähigem Quellcode mit der Dokumentation hat man bereits einen Teilerfolg auf dem Weg zur lebendigen Dokumentation erzielt.

Integration: Versionskontrolle mit git

Zusätzliches Potenzial kann man entfesseln, wenn man die Kombination von AsciiDoctor und Gradle mit einem zeitgemäßen Versionskontrollsystem wie „git" verknüpft. Dank der Tatsache, dass die AsciiDoc-Dokumente in Plain-Text vorliegen, sind sie dankbare Inhalte für ein „git"-Repository. So können Änderungen gut nachvollzogen und auch zeitgleiche Erweiterungen an der Dokumentation gut zusammengeführt werden.

Um diese Möglichkeiten weitestgehend auszuschöpfen, lohnt sich ein Blick in den „AsciiDoc Writer's Guide" *[10]* und die „AsciiDoc Recommended Practices" *[11]*, die wertvolle Tipps enthalten. Insbesondere das Aufteilen in viele Teildokumente sowie die Anwendung des „One sentence per line"-Prinzips haben sich beim kollaborativen Bearbeiten von AsciiDoc-Dokumenten bewährt.

„One sentence per line" erleichtert das Vergleichen von Änderungen in einer Dokument-Historie. Viele Teildokumente haben neben einer einfacheren Navigation den Vorteil, dass in verschiedenen Dokumentationen benötigte Abschnitte (wie Definitionen) einfach per „include" wiederverwendet werden können.

Features, Branches und der Review-Prozess

In vielen Teams haben sich mittlerweile die von GitHub bekannten und durch GitLab oder Bitbucket adaptierten Merge- oder Pull-Requests für einen Code-Review-Prozess etabliert. Die Verwaltung der Dokumentation im gleichen Repository wie dem Quellcode hat neben den zuvor beschriebenen Möglichkeiten zur Einbettung den Charme, dass die zur Implementierung eines Features notwendigen Änderungen an der Dokumentation direkt auf dem gleichen Feature-Branch erfolgen können und damit Teil des etablierten Review-Prozesses werden.

Zusätzlich stellt man so auch sicher, dass die Änderung der Dokumentation auch immer zum jeweiligen Stand des Systems passt. Dies ist insbesondere bei der Notwendigkeit, verschiedene Software-Stände betreuen zu müssen, sehr wertvoll. Die Möglichkeit, Änderungen in Pull/Merge-Requests zu diskutieren und so auch eine später noch nachvollziehbare Dokumentation der Entscheidungsfindung zu erhalten, hat sich auch für Textdokumente als nützlich und im Besonderen im Vergleich zu Anmerkungen in Office-Dokumenten als vorteilhaft erwiesen.

Die Kür hin zur lebendigen Dokumentation ist die Umsetzung des in *Abbildung 4* dargestellten und von Gojko Adzic im Buch „Specification by Example" beschriebenen Prinzips der Verknüpfung von Spezifikation mit Tests und Dokumentation. Ausgehend von der Spezifikation und den darin verwendeten Werten werden das implementierte System getestet und die Ergebnisse wiederum in die Dokumentation übernommen.

Dies wird am besten in einem Beispiel deutlich: Den Anfang macht die Spezifikation. Kernpunkt ist für uns eine Wertetabelle, die Eingabewerte sowie die erwarteten Ergebnisse spezifiziert. Die Tabelle

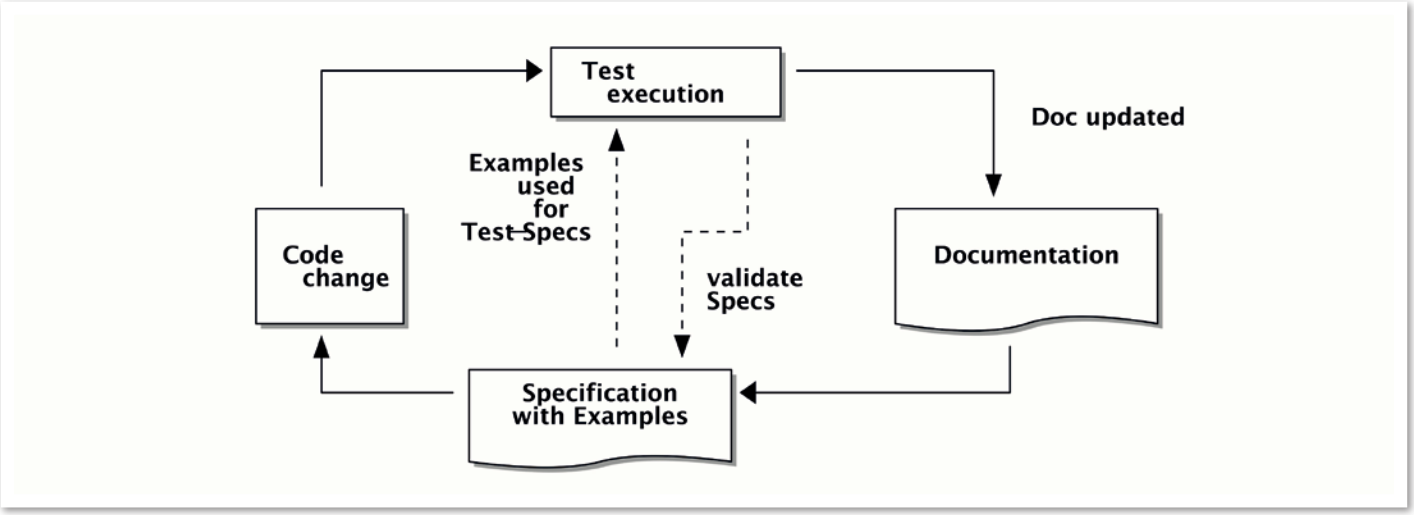


Abbildung 4: Workflow Specification by Example

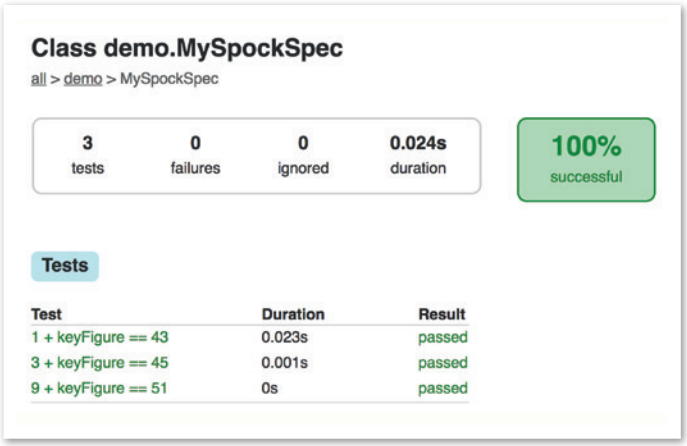


Abbildung 5: Spock-Testergebnis-Report

liegt im CSV-Format vor und kann so gut mit einem Tabellenkalkulationsprogramm bearbeitet, sehr einfach in AsciiDoc eingebunden und auch für Tests verwendet werden.

Für letzteren Aspekt betritt noch ein weiterer Charakter die Bühne: Spock, das Test- und Spezifikations-Framework *[12]*. Es bietet unter anderem wertvolle Features für Daten-getriebenes Testen, auf die für das folgende Beispiel zurückgegriffen wird. Hat man aus der Spezifikation die *Tabelle 1*, kann man diese Werte mit einer Spock-Spezifikation, wie in *Listing 6* gezeigt, gegen die Methode „Business().calculate()" aus *Listing 5* testen.

Wesentlich ist die „@Unroll"-Annotation, die bewirkt, dass die Testmethode für jedes Wertepaar, das über den „where"-Block geliefert wird, ausgeführt wird. Die im Namen der Testmethode mit „#" eingeleiteten Namen werden im Ergebnisreport durch die genutzten Werte ersetzt. Als Ergebnis erhält man einen Report, wie in *Abbildung 5* gezeigt. Diesen kann man zusammen mit der Spezifikation und Dokumentation paketieren und so ein vollständiges Bild zum Software-Projekt liefern.

Als Ausbaustufe ist es möglich, durch die Spock-Reports-Extension *[13]* die Reports mit einem passenden Template direkt in AsciiDoc erstellen zu lassen und diese nahtlos in die Dokumentation zu integrieren. So erhält man die Möglichkeit, mit jedem Build eine validierte Dokumentation zum erzeugten Artefakt zu liefern.

Startklar?

Einen einfachen Einstieg in die vorgestellte Tool-Chain findet man über das Lazybones-Template „asciidoctor-gradle", mit dessen Hilfe man sich mit dem folgenden Befehl eine Basis-Projektstruktur im Verzeichnis „projektName" erzeugen lassen kann: „lazybones create asciidoctor-gradle projektName".

Hat man bisher noch keinen Kontakt mit Lazybones gehabt, lohnt sich ein Blick darauf. Es bietet Möglichkeiten für „project"-Templates, die an Maven Archetypes erinnern, diesen aber in einigen Aspekten überlegen sind. Lazybones lässt sich bequem über SDKman *[14]* installieren, was ebenso eine Bereicherung für die Toolbox ist.

```
@Unroll
def "#firstNum + keyFigure == #secondNum"() {
  expect:
  def result = Integer.valueOf(firstNum as String) +
new Business().calculate()
  def expected = Integer.valueOf(secondNum as String)
  expected == result

  where:
     [firstNum, secondNum] <<
getResourceAsStream("test.csv").readLines()*.
split(",")

}
```

Listing 6

input	expected
1	43
3	45
9	51

Tabelle 1: „test.csv"

Weiterführende Links

[1] *https://github.com/asciidoctor/asciidoctorj*
[2] *http://asciidoctor.org/docs/asciidoc-syntax-quick-reference*
[3] *http://asciidoctor.org/docs/user-manual*
[4] *http://asciidoctor.org/docs/asciidoctor-diagram*
[5] *http://www.asciidocfx.com*
[6] *http://asciidoctor.org/docs/asciidoctor-gradle-plugin*
[7] *http://asciidoctor.org/docs/user-manual/#attribute-catalog*
[8] *https://github.com/asciidoctor/asciidoctor-gradle-examples*
[9] *https://docs.gradle.org/current/userguide/continuous_build.html*
[10] *http://asciidoctor.org/docs/asciidoc-writers-guide/*
[11] *http://asciidoctor.org/docs/asciidoc-recommended-practices*
[12] *http://spockframework.org*
[13] *https://github.com/renatoathaydes/spock-reports*
[14] *http://sdkman.io*

Markus Schlichting
markus.schlichting@canoo.com

Markus Schlichting ist Senior Software Engineer bei Canoo Engineering in Basel. Software Engineering und Delivery, agile Methoden und Open-Source-Projekte zählen zu seinen Leidenschaften. Neben dem täglichen Mix aus Architektur und Implementierung findet man ihn als Speaker auf Entwickler-Konferenzen. Außerdem organisiert er monatlich den Hackergarten in Basel. Er freut sich sehr darüber, von den Erfahrungen der Leser mit lebendiger Dokumentation zu erfahren.

Eine Groovy-DSL zum Erzeugen von Testdaten über JPA

Daniel Behrwind, TRIOLOGY GmbH

Beim automatisierten, integrativen Testen von Software, die mit einem komplexen JPA-Datenmodell arbeitet, steht man unweigerlich früher oder später vor der Frage, wie sich semantisch sinnvolle Testdaten ohne großen Aufwand erzeugen lassen. Dieser Artikel zeigt, wie man mit Groovy eine Domain Specific Language (DSL) definieren kann, die es erlaubt, Testdaten leicht lesbar, modular und getrennt vom eigentlichen Test-Code festzulegen.

Automatisierte Software-Tests gehen von bestimmten Annahmen aus, bevor sie sicherstellen, dass das getestete Feature unter den gegebenen Bedingungen korrekt funktioniert. Testet man auf integrativer Ebene, manifestieren sich diese Annahmen oft in einem bestimmten Datenstand einer relationalen Datenbank. Vor Durchführung eines Tests sind also entsprechende Testdaten zu erzeugen.

Häufig widersprechen sich die für die einzelnen Testfälle benötigten Daten aber gegenseitig, sodass es nicht möglich ist, auf einem einmalig definierten Stand aufzusetzen. Vielmehr müssen die Daten programmatisch pro Testfall definiert und in die unterliegende Datenbank eingefügt werden. Dabei wird der Testcode schnell unübersichtlich, wenn große Hierarchien oder große Datenmengen erforderlich sind.

Es liegt also nahe, die Erzeugung der Testdaten auszulagern, sodass sich der eigentliche Test auf das Wesentliche konzentrieren kann. Dazu drängen sich zunächst zwei Alternativen auf: native SQL-Skripte oder die Test-Bibliothek DBUnit. Beide orientieren sich allerdings an Tabellen und Spalten statt an Java-Objekten. Eine im Produktiv-Code durch JPA gewonnene Abstraktion der relationalen Datenbank würde damit im Test wieder verloren gehen. Insbesondere beim Abbilden von Referenzen durch Fremdschlüssel macht sich das schmerzlich bemerkbar. Die Generierung der Daten lässt sich zwar vom Testcode trennen, in puncto Übersichtlichkeit ist aber nichts gewonnen. Der Fokus sonstiger Alternativen liegt eher darin, große Mengen an Zufallsdaten zu erzeugen. Sie zielen auf Performance-Tests ab, sind jedoch für einen fachlichen Test ungeeignet.

Anforderungen

So sähe eine Lösung aus, die dem Datenbank-scheuen Java-Entwickler zusagt:

- Die Lösung müsste nahtlos aus dem Java-Test-Code heraus aufrufbar sein
- Testdaten sollten wiederverwendbar und modular definierbar sein
- Die Lösung sollte vollständig objektorientiert arbeiten
- Gespeicherte Entities sollten dem aufrufenden Code zugänglich gemacht werden
- Testdaten sollten in einer gut lesbaren Form komfortabel definiert werden können

Das Grails-Fixtures-Plug-in *(siehe „http://www.grails.org/plugin/fixtures")* erfüllt überwiegend die genannten Anforderungen bereits mit einer Domain Specific Language (DSL). Es ist allerdings eng mit dem Grails-Framework verwoben. Im Folgenden wird gezeigt, wie sich die Anforderungen mit wenig Aufwand in einer Groovy-DSL umsetzen lassen, die direkt in herkömmlichen Java-Projekten nutzbar ist. Der vollständige Code steht unter *„https://github.com/triologygmbh/test-data-loader"* zur Verfügung.

Groovy to the Rescue

Die JVM-Sprache Groovy bietet mit ihrer dynamischen Natur und vielen syntaktischen Möglichkeiten beste Voraussetzungen, auf einfache Weise eine eigene DSL zu definieren. Unter *„http://www.*

```
create User, 'Peter', {
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 1

```
class EntityBuilder {
  void buildEntities(String entityDefinitionFile) {
    DelegatingScript script = createExecutableScriptFr
omEntityDefinition(entityDefinitionFile)
    script.setDelegate(this)
    script.run()
  }
  // …
}
```

Listing 2

groovy-lang.org/documentation.html" stehen weiterführende Informationen zu Groovy bereit. Sprach-Features, die in der vorgestellten Lösung Verwendung finden, sind im Fall des Einsatzes kurz erklärt.

Die größte Herausforderung beim Entwickeln der angedachten DSL ist es, eine möglichst einfache Syntax für die Definition der Testdaten zu finden und diese Definition dann in tatsächliche JPA-Entities umzuwandeln. Um dies ein wenig plastischer zu machen, zunächst ein Beispiel: Ziel ist es, einen JPA-Entity-User durch folgendes Snippet zu instanziieren, die Felder entsprechend zu initialisieren, das Entity in der Datenbank zu speichern und den Test-Code unter dem Namen „Peter" zur Verfügung zu stellen *(siehe Listing 1)*.

Groovy-Skripte einlesen und ausführen

Bevor wir uns anschauen, wie aus der Definition ein initialisiertes Entity wird, werden zunächst Lösungen für die übrigen Anforderungen diskutiert. Groovy wird in Java-Byte-Code übersetzt. Daher lässt sich Groovy-Code direkt aus Java-Code heraus aufrufen, als wäre er in Java geschrieben. Die nahtlose Java-Integration bekommen wir also schon einmal geschenkt.

Zusätzlich ist es möglich, Groovy als Skript-Sprache einzusetzen. Dabei kann man relativ einfach Skript-Dateien mit Groovy-Bordmitteln programmatisch einlesen und ausführen. Diese Möglichkeit eignet sich ideal für unsere Zwecke: Wir können so unsere Testdaten-Definitionen in beliebige „.groovy"-Dateien auslagern und je nach Testfall die benötigten Dateien laden. Dazu wird die Klasse „EntityBuilder" eingeführt, die einen Dateinamen entgegennimmt und die in der Datei definierten Entities erzeugt *(siehe Listing 2)*. Der vollständige Code steht im eingangs erwähnten GitHub-Repository zur Verfügung.

„buildEntities" nimmt den Dateinamen eines Entity-Definitions-Skripts entgegen. Diese Definition wird eingelesen und in eine ausführbare Skript-Instanz umgewandelt. Dabei wird zur Laufzeit eine Klasse erstellt, deren „run"-Methode den Inhalt des Skripts enthält. Diese „run"-Methode können wir nun wie jede andere Methode aufrufen und so das Skript ausführen. Hierbei ist zu beachten, dass Groovy Referenzen, die im Skript nicht eindeutig sind, zur Laufzeit auflösen kann. Die eingesetzte Skript-Subklasse „DelegatingScript" erlaubt es, ein Delegate zu setzen, gegen das nicht eindeutige Referenzen aufgelöst werden. Der EntityBuilder setzt sich an dieser Stelle selbst als Delegate des Skripts. Wozu das notwendig ist, wird später deutlich.

Mit dieser Umsetzung können wir nun Testdaten modular in beliebigen Skriptdateien definieren und diese nach Bedarf laden. Gehen wir zunächst davon aus, dass tatsächlich die in den Skripten definierten Entities instanziiert und initialisiert werden, so stellt sich die Frage, wie ihre Daten in die Datenbank gelangen.

Der Glue-Code

Um die Persistierung der Entities von ihrer Erzeugung zu entkoppeln, bietet der EntityBuilder die Möglichkeit, „EntityCreatedListener" zu registrieren. So können wir im Glue-Code in der Klasse „TestDataLoader" *(siehe „https://github.com/triologygmbh/test-data-loader")* einen Listener einsetzen, der sich darum kümmert, die Daten zu speichern. „TestDataLoader" erwartet als Konstruktor-Parameter einen fertig initialisierten JPA-EntityManager. Seine Methode „loadTestData" kann dann vom Client mit Entity-Definitions-Dateien aufgerufen werden. Nach Initialisierung des Listeners reicht er die Definitionen an den EntityBuilder weiter. Dieser erzeugt die definierten Entities und übergibt sie über das Listener-Interface an den EntityPersister zum Speichern.

Definition der eigentlichen DSL

Wir sind jetzt in der Lage, Entities in beliebigen Skripten zu definieren, die Definitionen einzulesen und die erzeugten Entities zu speichern. Beim eigentlichen Erzeugen der Entities kommen gleich mehrere Groovy-Features zum Tragen: Groovy erlaubt es, beim Aufruf von Methoden die Parameter-umschließenden Klammern wegzulassen. Zusätzlich ist es möglich, mit geschweiften Klammern Closures zu definieren, also ausführbare Code-Abschnitte, ähnlich Java-8-Lambdas, die wie gewöhnliche Objekte über Variablen referenziert und übergeben werden können. Closures lassen sich dann an beliebiger Stelle ausführen.

Damit wird klar, dass der Ausdruck „create User, 'Peter', { }" nichts weiter ist als ein Aufruf der statischen Methode „create" im EntityBuilder („static import") mit drei Parametern (in Groovy kann eine Klasse einfach über ihre Namen referenziert werden; der Ausdruck „User" ist daher äquivalent zu „User.class."). *Listing 3* zeigt, was beim Aufruf von „create" aus der DSL heraus passiert.

Wir stellen fest, dass die statische „create"-Methode lediglich an „createEntity" der Singleton-Instanz des EntityBuilder delegiert. Hier wird zunächst eine neue Instanz der übergebenen Entity-Klasse erzeugt und unter dem ebenfalls übergebenen Namen in der „java.util.Map" unter „entitiesByName" registriert. Hinweis: „entitiesByName[entityName] = entity" ist analog zu „entitiesByName.put(entityName, entity)". Nachdem das neue Entity in der Methode „executeEntityDataDefinition" mit Daten initialisiert wird, werden schlussendlich die registrierten Listener informiert.

Die eigentliche Magie passiert in den zwei Zeilen der Methode „executeEntityDataDefinition". Ihr werden das neu instanziierte Entity sowie das aus dem Skript stammende Closure übergeben, in dem die Daten für das Entity definiert sind. Um zu verstehen, was passiert, müssen wir allerdings etwas weiter ausholen. Schauen wir zunächst noch einmal auf das Closure, das im Skript als letzter Pa-

rameter an die „create"-Methode übergeben wird *(siehe Listing 4)*. Es werden augenscheinlich Variablen Werte zugewiesen. Diese Variablen sind allerdings nicht deklariert, sodass hier ein weiteres Groovy-Feature zum Tragen kommt. Der Ausdruck „myObject.someProperty = 'value'" entspricht dem Aufruf eines Setters „myObject.setSomeProperty('value')". Es werden im Closure also zwei Setter aufgerufen, die Frage ist nur: Worauf? Da die gesetzten Felder zufällig genau den Properties des zu erzeugenden User-Entitys entsprechen, wäre es doch praktisch, sie würden direkt auf dem Entity aufgerufen? Die beiden Zeilen in „executeEntityDataDefinition" bewirken genau das.

Ähnlich wie bei Skripten ist Groovy in der Lage, Methoden-Aufrufe innerhalb eines Closure zur Laufzeit dynamisch aufzulösen. Dazu kann man auch für das Closure ein Delegate definieren. Die in „executeEntityDataDefinition" aufgerufene „rehydrate"-Methode erstellt eine Kopie des Closure und setzt den ersten Parameter als Delegate, in unseren Fall also das zuvor instanziierte Entity. Wird das Closure nun mit „entityDataDefinition.call()" ausgeführt, werden die Setter im Beispiel tatsächlich auf dem User-Entity aufgerufen, sodass dieses mit den entsprechenden Daten initialisiert wird.

Es ist an der Zeit, das bisher Erreichte einmal auszuprobieren, bevor wir die DSL noch weiter verfeinern. Definieren wir uns also zunächst einen User *(siehe Listing 5)*.

Die Klasse „Demo.java" führt vor, wie der TestDataLoader aus Java-Code heraus benutzbar ist und dass die erzeugten Entities tatsächlich persistiert werden. Die DSL-Snippets stammen aus der Datei „testData.groovy" *(siehe „https://github.com/triologygmbh/test-data-loader")*.

Verschachtelte Entities

So weit, so gut – wir haben den Roundtrip von der DSL über die Datenbank bis hin zum Testcode geschafft. Bleibt noch offen, wie ein komplexes Datenmodell zu bedienen ist, wie wir also mit Referenzen zwischen den Entities umgehen. Nehmen wir hierzu an, dass ein Benutzer einer Abteilung zugeordnet werden kann. Der User erhält eine „@ManyToOne"-Beziehung zum Department. In der DSL können wir die Erzeugung von Entities ganz einfach schachteln *(siehe Listing 6)*.

Was aber, wenn ein zweiter User derselben Abteilung angehört? Das Department mit der „create"-Methode ein zweites Mal anzulegen, ist offensichtlich keine Option. Wir müssen also eine Möglichkeit schaffen, bereits angelegte Entities aus der DSL heraus zu referenzieren. Da wir uns in ganz normalem Groovy-Code bewegen, wäre es möglich, sich ein von der „create"-Methode erzeugtes Entity in einer Variablen zu merken. Mit ein wenig Unterstützung vom EntityBuilder geht das aber auch einfacher *(siehe Listing 7)*.

Was geht hier nun vor sich? Wie kann die Zuweisung „department = lostBoys" funktionieren, ohne dass „lostBoys" initialisiert wurde? Hier greifen wieder verschiedene Groovy-Eigenschaften ineinander: „lostBoys" ist zunächst ein Bezeichner, der nicht aufgelöst werden kann. Groovy ruft in diesem Fall einen Getter für ein Property mit dem Namen des Bezeichners auf – ähnlich wie den Setter bei der Zuweisung von Werten zu Variablen. Analog entspricht der Ausdruck „myObject.someProperty" dem Aufruf „myObject.getSomeProperty()". Auch hier stellt sich wieder die Frage, worauf der Getter aufgerufen wird. Das

```
class EntityBuilder {
  static <T> T create(Class<T> entityClass, String
entityName, Closure entityData) {
    return instance().createEntity(entityClass, enti-
tyName, entityData);
  }

  private <T> T createEntity(Class<T> entityClass,
String entityName, Closure entityData) {
    T entity = createEntityInstance(entityName, enti-
tyClass)
    executeEntityDataDefinition(entityData, entity)
    notifyEntityCreatedListeners(entity)
    return entity
  }

  private <T> T createEntityInstance(String entity-
Name, Class<T> entityClass) {
    ensureNameHasNotYetBeenAssigned(entityName, enti-
tyClass)
    T entity = entityClass.newInstance()
    entitiesByName[entityName] = entity;
    return entity
  }

  private void executeEntityDataDefinition(Closure
entityDataDefinition, Object entity) {
    entityDataDefinition = entityDataDefinition.
rehydrate(entity, this, this)
    entityDataDefinition.call()
  }

  // …
}
```

Listing 3

```
{
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 4

```
create User, 'Peter', {
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 5

```
create User, 'Peter', {
  firstName = 'Peter'
  lastName = 'Pan'
  department = create Department, 'lostBoys', {
    name = 'The Lost Boys'
  }
}
```

Listing 6

```
create User, 'Tinker', {
    firstName = 'Tinker'
    lastName = 'Bell'
    department = lostBoys
}
```

Listing 7

Delegate des Closure kann es nicht sein. Das ist in diesem Fall eine User-Instanz, die sicherlich keine Methode „getLostBoys()" bietet.

An dieser Stelle kommt das beschriebene Delegate des Skripts ins Spiel. Erinnern wir uns, der EntityBuilder setzt sich vor dem Ausführen des Skripts selbst als Delegate. Zwar hat auch der EntityBuilder keine Methode „getLostBoys()", er implementiert jedoch die Methode „propertyMissing", die von Groovy beim Zugriff auf ein nicht existierendes Property mit dessen Namen aufgerufen wird. Auf diese Weise können wir nun das zuvor angelegte Department unter dem Namen „lostBoys" auffinden und zurückgeben, sodass es im Skript als Wert gesetzt wird *(siehe Listing 8)*. Das Ganze lässt sich sogar so weit treiben, dass wir „Peter" als Department-Head setzen, während wir das Department anlegen und ihm zuordnen *(siehe Listing 9)*.

Code Completion

Damit haben wir alles, was wir brauchen. Es geht aber noch ein bisschen komfortabler. Bislang sind wir bei der Definition der eigentlichen Entity-Daten ziemlich auf uns gestellt. Innerhalb der DSL gibt es keine Möglichkeit herauszufinden, welche Properties ein Entity hat und von welchem Typ sie sind. Code-Completion durch die IDE ist somit auch nicht möglich. Wir können der IDE allerdings mitteilen, was das Delegate des Closure sein wird. Alle Informationen sind im Aufruf der statischen „create"-Methode des EntityBuilder vorhanden. Das Delegate des Closure ist immer eine Instanz der gleichzeitig übergebenen Klasse. Ergänzen wir daher die „create"-Methode um zwei Annotationen, um diese Information bekannt zu machen *(siehe Listing 10)*. Im Resultat weiß die IDE (hier IntelliJ IDEA), dass in unserem Beispiel Aufrufe innerhalb des Closure an eine User-Instanz delegiert werden, und bietet entsprechend die Properties des Users an.

Geschafft! Unsere DSL erlaubt es, Testdaten leicht lesbar, modular und getrennt vom eigentlichen Testcode zu definieren. Hierarchien können verschachtelt und objektorientiert statt über Fremdschlüssel abgebildet werden. Dabei müssen wir die Java-Welt nicht verlassen und keinen gedanklichen Bruch hin zum relationalen Modell der Datenbank hinnehmen. Da wir uns innerhalb der DSL in Groovy-Code bewegen, genießen wir alle Freiheiten, die eine Programmiersprache so mit sich bringt. Ein denkbares Szenario wäre zum Beispiel, große Datenmengen über Schleifen zu generieren.

```
private def propertyMissing(String name) {
  if (entitiesByName[name]) {
    return entitiesByName[name]
  }
  // handle missing reference
}
```

Listing 8

```
create User, 'Peter', {
  department = create Department, 'lostBoys', {
      name = 'The Lost Boys'
      head = Peter
  }
}
```

Listing 9

```
static <T> T create(@DelegatesTo.Target Class<T> entityClass, String entityName, @DelegatesTo(strategy = Closure.DELEGATE_FIRST, genericTypeIndex = 0) Closure entityData) {
  return instance().createEntity(entityClass, entityName, entityData);
}
```

Listing 10

```
create User named 'Peter' with {
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 11

In diesem Artikel unbehandelt bleibt allerdings die Frage, wie die Datenbank nach einem Testfall bereinigt werden soll. In den vorgeführten Beispielen („Demo.java") machen wir es uns einfach und rollen nach jedem Test die Transaktion zurück. Für einen wirklich integrativen Test ist das natürlich keine Option. In echten Projekten haben wir bislang mittels Datenbank-Skript und „TRUNCATE TABLE" alle Tabellen geleert. Bei großen Schemata hat sich das allerdings erheblich auf die Ausführungszeit der Tests ausgewirkt. Es wäre interessant auszuprobieren, ob es effizienter wäre, die Datenbank programmatisch zu bereinigen. Über einen Stack sollte es relativ einfach möglich sein, die angelegten Entities in umgekehrter Reihenfolge wieder zu löschen.

Noch ein kleiner Wermutstropfen: Die DSL könnte sich mit einem Fluent-API noch schöner gestalten *(siehe Listing 11)*. Da sich die Definition hier jedoch über drei Methoden-Aufrufe verteilt („create", „named" und „with"), habe ich keine Möglichkeit gefunden, das Delegate des Closure über Annotationen bekannt zu machen. Grund ist, dass die Klasse, an die delegiert wird, an eine andere Methode übergeben wird als das Closure selbst. Zugunsten der Code-Completion hat sich der Autor daher für eine etwas unschönere DSL-Syntax entschieden. Vielleicht fällt ja jemandem ein, wie man beides haben kann? Pull-Requests mit Verbesserungen sind in jedem Fall willkommen.

Daniel Behrwind
d.behrwind@gmail.com

Daniel Behrwind ist als Software-Entwickler bei der TRILOGY GmbH tätig. Dort befasst er sich überwiegend mit der Entwicklung von Individual-Software, wobei er schwerpunktmäßig an Java-Webprojekten arbeitet. Als leidenschaftlicher Clean-Code-Verfechter ist er begeistert von kreativen Lösungen für komplexe Probleme, die so offensichtlich aussehen, als seien sie von selbst entstanden.

Modulare Anwendungen mit Java 9

Guido Oelmann, Freelancer

Mit Java 9 hält die Modularität Einzug auf der Plattform. Der Artikel geht der Frage nach, was unter „Modularität" zu verstehen ist und warum diese so wichtig ist für das Bauen von wartbaren und langlebigen Architekturen. Es wird gezeigt, wie Java-9-Anwendungen auf Basis von Modulen entworfen werden und wie dabei moderne Entwicklungsansätze wie Microservices berücksichtigt werden können.

Beispiele für Module finden sich überall. Der handelsübliche Fernseher wird aus einer Vielzahl von Modulen im Baukastenprinzip zusammengebaut und Ähnliches passiert auch in der Automobilfertigung. Verschiedene Autotypen des gleichen Herstellers teilen sich eine Reihe von Modulen, die, unterschiedlich kombiniert und um weitere Module ergänzt, zu entsprechenden Automodellen zusammengesetzt werden.

Ein weiteres Beispiel für den modularisierten Bau ist die internationale Raumstation ISS, die als größtes künstliches Objekt den Erdorbit bewohnt. Dieses gemeinsame Projekt von fünf Raumfahrt-Agenturen und zehn zusätzlichen Ländern ist geradezu ein Paradebeispiel dafür, was durch Modularisierung im Großen geleistet werden kann. Die Raumstation besteht aus 34 Modulen, die in den nächsten Jahren noch um sechs weitere ergänzt werden sollen. Die Module sind in unterschiedlichen Ländern von unterschiedlichen Teams gebaut worden und wurden nach erfolgreichen Tests auf dem Boden mit Trägerraketen und Raumfähren in die Erdumlaufbahn gebracht, um dort an die Station angedockt zu werden.

Das Innenleben der Module ist also unabhängig von den anderen Modulen gebaut worden und auf die gewünschte Funktionalität hin testbar. Beachtet werden muss beim Bau der Module, an welche anderen Module diese gekoppelt werden, ob Funktionen anderer Module genutzt werden sollen oder welche eigenen Funktionalitäten anderen Modulen zu Verfügung zu stellen sind und wie die Verbindungsstellen zwischen den Modulen aussehen müssen. Übertragen auf die Software-Entwicklung ergeben sich daraus drei wesentliche Aspekte:

- Starke Kapselung
- Wohldefinierte Schnittstellen
- Explizite Abhängigkeiten

„Starke Kapselung" bedeutet, dass der Zugriff auf die Implementierungen des Moduls nur über Schnittstellen erfolgt und die eigentlichen Implementierungsdetails nach außen hin verborgen blei-

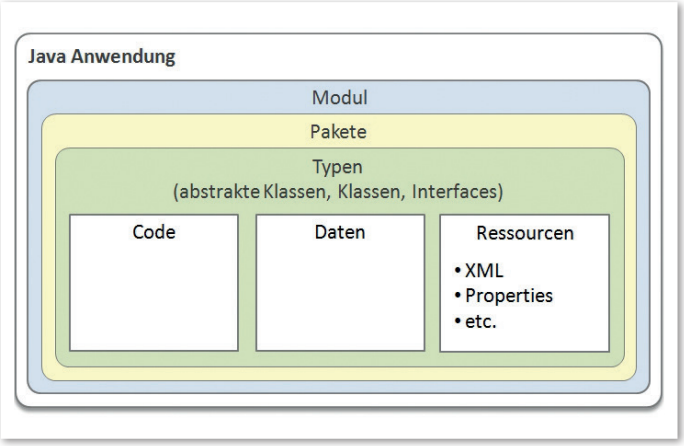


Abbildung 1: Aufbau einer modularen Anwendung

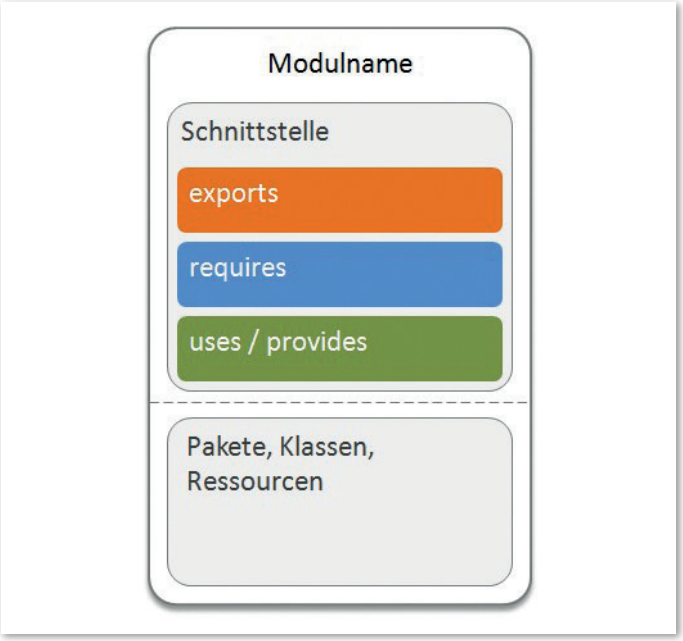


Abbildung 2: Aufbau eines Java-Moduls

ben. Bei Letzterem wird auch vom „Geheimnisprinzip" gesprochen. Schnittstellen müssen wohldefiniert sein, damit andere Module wissen, wie sie zugreifen können und was für Funktionalitäten das Modul überhaupt zur Verfügung stellt. Die expliziten Abhängigkeiten beschreiben, von welchen Modulen ein anderes Modul abhängt. Ohne diese explizite Definition wäre ein Wissen darüber, wie die Module untereinander abhängen und kommunizieren, nur schwer möglich. Aber was macht die Modularisierung so besonders? Warum sollte Modularisierung angewendet werden?

Modularisierung ist ein machtvolles Instrument

Eines der wichtigsten Prinzipien der Softwaretechnik, wenn nicht sogar das wichtigste, ist die Modularisierung. Wie im Beispiel der internationalen Raumstation zu sehen war, half die Aufteilung der Raumstation in einzelne Module, die Komplexität erheblich zu verringern. Dadurch konnten die Module getrennt voneinander entwickelt, aber auch getrennt voneinander betrachtet und verstanden werden. Genau das ist es, was die Wartbarkeit und damit letztlich auch die Langlebigkeit von Systemen erheblich erhöht. Zudem vereinfachen die Schnittstellen-Definitionen die Erweiterbarkeit des Systems und auch die Rekombination von Modulen.

Modularisierung unterstützt direkt die agile Software-Entwicklung, können doch leicht verschiedene Teams für unterschiedliche Module verantwortlich sein. Zudem lassen sich Module ähnlich schneiden, wie es der Microservice-Ansatz fordert. Wie später noch zu sehen ist, können dadurch neue und vorteilhafte Ansätze bei der Software-Entwicklung entstehen.

Modularisierung ist im Grunde nichts Neues und findet sich in allen modernen Softwareprojekten wieder, sei es in Form von Klassen mit ihren Schnittstellen oder bei Komponenten. Was aber häufig sträflich vernachlässigt wird, ist die Modularisierung auf einer höheren Ebene, und genau da fangen dann die Probleme mit der Wartbarkeit an.

In vielen Unternehmen wird der Ansatz verfolgt, ein Softwareprojekt in unterschiedliche Teilprojekte zu untergliedern, die zu einzelnen JARs kompiliert am Ende zu der gesamten Anwendung zusammengesetzt werden. Dies ist eine gute und sinnvolle Gliederung, aber auch nicht mehr. Normale JARs sind nichts weiter als „.zip"-Archive, die mit Modulen nichts zu tun haben. JARs haben keine Schnittstellen, kapseln nicht ihren Inhalt und es gibt auch keine Information darüber, ob und von welchen anderen JARs sie abhängen. An dieser Stelle wird sich eine echte Modularisierung bezogen auf die Lebensdauer des Systems immer als machtvolles Instrument zur Erstellung wartbarer Systeme herausstellen. Mit Java 9 wird es eine direkte Unterstützung für das Bauen modularer Anwendungen geben.

Das Java-Modulsystem

Mit Java 9 wird das JDK ebenfalls modularisiert vorliegen, was es ermöglichen wird, eigene zusammengestellte Java-Laufzeitumgebungen zu erzeugen, die dann nur jene Plattform-Module enthalten, die von der ausgelieferten Java-Anwendung wirklich benötigt werden. Daneben steht dem Entwickler ein Java-Modulsystem zur Verfügung; das erlaubt die direkte Erstellung von Modulen. Bis einschließlich Java 8 bestand eine Anwendung aus Klassen, Daten und Ressourcen, die auf verschiedene Pakete aufgeteilt wurden. *Abbildung 1* zeigt, dass Module oberhalb der Pakete liegen und diese kapseln.

Ein Java-Modul besteht aus seinem Modulnamen, einer Schnittstelle und den Paketen mit den eigentlichen Implementierungen beziehungsweise Daten und Ressourcen. Dabei wird über die Modul-Schnittstelle deklariert, was an Paketen nach außen zur Verfügung steht und was für Abhängigkeiten zu anderen Modulen existieren. *Abbildung 2* zeigt den grundsätzlichen Aufbau eines Java-Moduls.

Die Modul-Schnittstelle deklariert drei primäre Dinge:

- **Exports**
  Die Deklaration der nach außen zur Verfügung gestellten Pakete
- **Requires**
  Angabe darüber, welche Module importiert werden sollen, und damit die Darstellung der Modulabhängigkeiten
- **Uses/Provides**
  Auflistung der Dienste, die ein Modul zur Laufzeit zur Verfügung stellt oder selber benötigt

Die Modul-Schnittstelle wird in Form einer Datei, dem Modul-Deskriptor, angelegt und oberhalb der Paketstruktur unter dem Na-

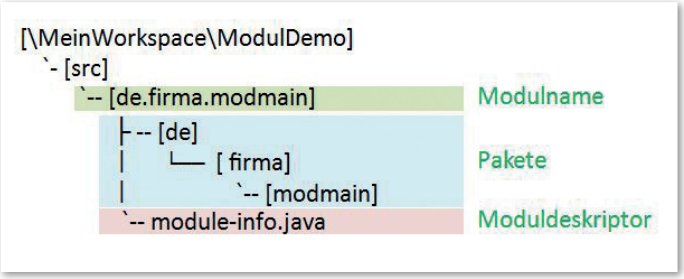


Abbildung 3: Projektstruktur

men „module-info.java" abgelegt. *Listing 1* zeigt die Struktur des Inhalts. Zur eindeutigen Benennung von Modulen wird die Vorgehensweise nach der „Reverse domain name notation" empfohlen, sodass ein einfaches Modul den Modul-Deskriptor aus *Listing 2* haben könnte.

Abbildung 3 zeigt die sich ergebene Projektstruktur. Darin ist das Projektverzeichnis „ModulDemo" mit dem Source-Verzeichnis „src" angelegt. Innerhalb dieses Verzeichnisses liegen das eigentliche Modul „de.firma.modmain" und darunter der Modul-Deskriptor sowie die Pakete. Dies ist bereits ausreichend, um ein funktionsfähiges Java-Modul zu erstellen.

Um eine Modulabhängigkeit zu deklarieren, wird zu dem obigen Modul ein weiteres mit dem Namen „de.firma.moda" erstellt. *Abbildung 4* zeigt die gewünschten Abhängigkeiten. Das Modul „de.firma.modmain" soll das Modul „de.firma.moda" importieren beziehungsweise lesend auf dieses zugreifen und das Modul „de.firma.moda" soll das Paket „de.firma.moda" nach außen freigeben und damit zugreifbar machen. *Listing 3* zeigt die sich ergebenen Modul-Deskriptoren.

Nur auf das, was über die Schnittstelle explizit nach außen freigegeben wird, kann zugegriffen werden, wobei auch ein Zugriff per Reflection verwehrt bleibt. Um diese Art des Zugriffs dennoch zuzulassen, kann das Schlüsselwort „opens" verwendet oder direkt das gesamte Modul für Reflection-Zugriffe freigegeben werden. Dafür gibt es neben anderen Modularten die sogenannten „Open Modules". Die verschiedenen Modularten und die Möglichkeit der Formulierung transitiver Abhängigkeiten soll an dieser Stelle jedoch nicht weiter ausgeführt werden.

```
module <Modulname> {
  requires <importiertes Modul>;
  exports <exportiertes Paket>;
}
```

Listing 1

```
module de.firma.modmain {
}
```

Listing 2

```
module de.firma.modmain {
      requires de.firma.moda;
}
module de.firma.moda {
     exports de.firma.moda;
}
```

Listing 3

Goodbye Klassenpfad – willkommen Modulpfad

Mit den Java-Modulen wird der Modulpfad eingeführt, der neben dem Klassenpfad existiert und diesen auf lange Sicht ablösen soll. Konnte bisher beim Starten einer Anwendung alles im Klassenpfad Liegende geladen werden, verhält es sich beim Modulpfad anders. Beim Starten einer modularisierten Anwendung wird der Name des initialen Moduls angeben, also des Moduls, das die „main()"-Methode enthält.

Die Java-Plattform lädt dieses Modul, schaut im Modul-Deskriptor nach, welche abhängigen Module erforderlich sind, und lädt diese ebenfalls. Diese zusätzlichen Module werden wiederum auf Abhängigkeiten untersucht etc., bis sich schließlich der komplette Abhängigkeitsgraph, der sogenannte „Modulgraph", ergibt. Die Kenntnis des Modulpfads ist wichtig für das Verständnis eines weiteren Mechanismus, des Service.

Services

Seit Java SE 6 existiert in der Spezifikation ein Service-Provider-

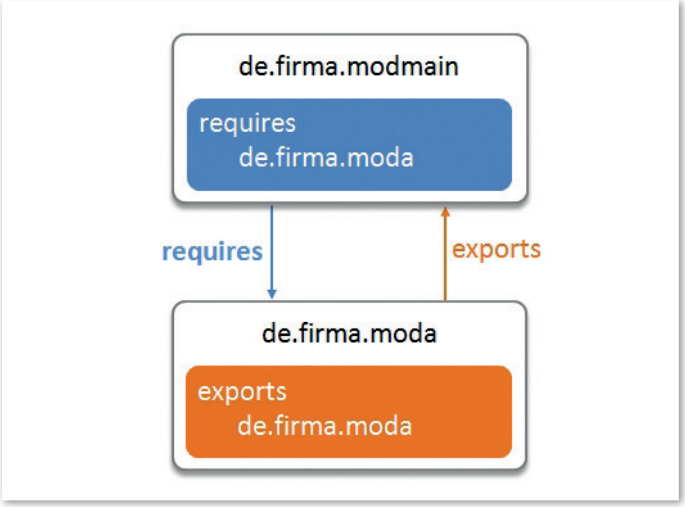


Abbildung 4: Modulabhängigkeiten

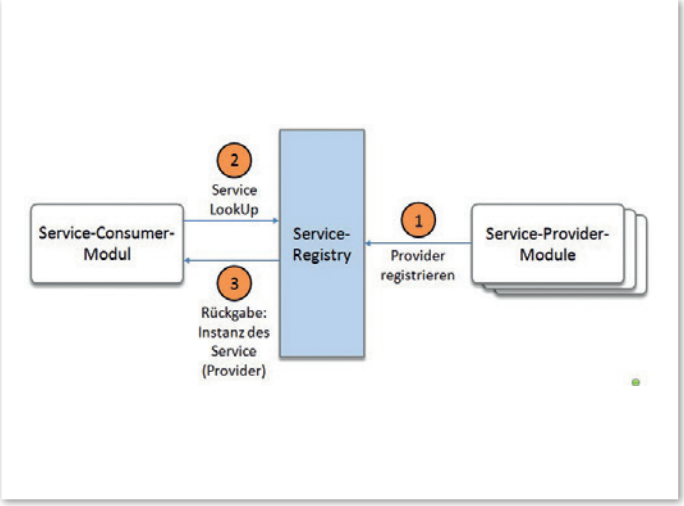


Abbildung 5: Das Service-Provider-Konzept

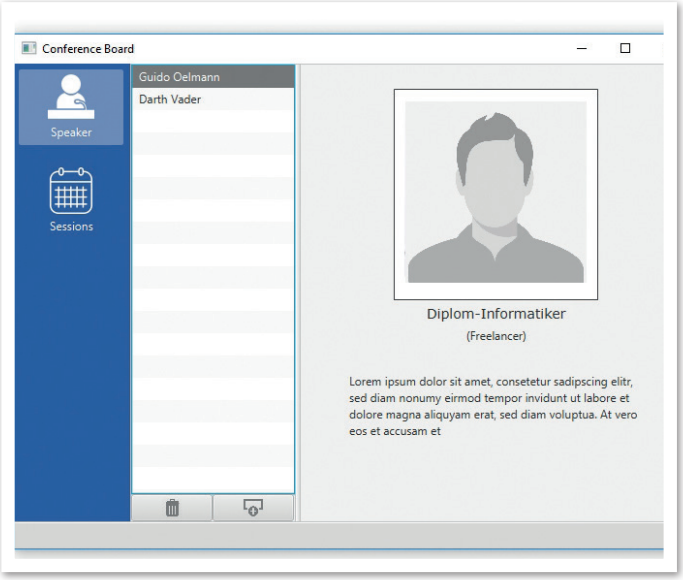


Abbildung 6: Modulare Anwendung

```
module de.firma.api {
        exports de.firma.api;
}
module de.firma.gui {
    requires de.firma.api;
    uses de.firma.api.ConferenceService;
}
module de.firma.sessions {
    requires de.firma.api;
    provides de.firma.api.ConferenceService
    with de.firma.sessions.SessionService;
}
module de.firma.speaker {
    requires de.firma.api;
    provides de.firma.api.ConferenceService
    with de.firma.speaker.SessionService;
}
```

Listing 4

```
package de.firma.api;
public interface ConferenceService<T> {
       public Collection<T> getAll();
       public Optional<?> get(String id);
       public void update(T item);
       public Optional<?> persist(T item);
       public void remove(T item);
}
```

Listing 5

```
package de.firma.sessions.service;
@Sessions
public class SessionService implements
ConferenceService<Session> {
  protected SessionDAO sessionDAO;
  public SessionService() {
     this.sessionDAO = new SessionDAO();
  }
  public Collection<Session> getAll() {
     return sessionDAO.getEntities();
  }
     …
}
```

Listing 6

Mechanismus, der für das Java-Modulsystem entsprechend erweitert wurde und eine zusätzliche Entkopplung zwischen Modulen ermöglicht. *Abbildung 5* zeigt das grundsätzliche Konzept.

Der Modul-Deskriptor erlaubt die Definition von Service-Provider und Service-Consumer. Module, die als Consumer fungieren, können dann zur Laufzeit das Service-Provider-Modul selber wählen, ohne dass zuvor eine explizite Abhängigkeit zwischen diesen Modulen deklariert wurde.

Zur Laufzeit werden die Provider-Module an zentraler Stelle registriert, was automatisch durch die Plattform erfolgt. Wenn dann ein Consumer-Modul Zugriff auf einen bestimmten Provider-Module-Typ haben möchte, der durch ein Service-Provider-Interface definiert ist, werden ein Look-up auf der Registrierungsstelle ausgeführt und eine passende Implementierung des Provider-Modul-Typs zurückgeliefert. Von diesem Mechanismus wird im folgenden Beispiel Gebrauch gemacht.

Eine modulare Anwendung

Die in *Abbildung 6* dargestellte Anwendung soll entworfen werden. Dabei handelt es sich um ein Tool zur Verwaltung von Sessions und Speakern für eine Konferenz. Ausgehend von dem klassischen Ansatz einer Drei-Schichten-Architektur, bei der die oberste Schicht die GUI enthält, die mittlere Schicht die eigentliche Logik und die unterste für die Datenhaltung zuständig ist, wäre die Abbildung dieser Schichten auf Module naheliegend. Diese resultierenden drei Module würden bei einer komplexen Anwendung das Behältnis für Untermodule darstellen.

Bei dieser Art von Modulen wird auch von Aggregator-Modulen gesprochen. Bei großen Anwendungen wäre hier die Kunst die Aufteilung des Systems innerhalb der Aggregator-Module in die einzelnen Untermodule und deren Zusammenspiel untereinander. Hier ist neben vielen anderen zu beachtenden Dingen wichtig, wie groß etwa die Schnittstellen sind, wie hoch die Anzahl der Abhängigkeiten von Modulen zu anderen Modulen ist und wie viele Module insgesamt existieren oder wiederum in Untermodule aufgeteilt sind. Bei genügend komplexen Systemen ist dies keine triviale Aufgabe und es muss darauf geachtet werden, letztlich nicht in einer schwer zu beherrschenden Modul-Hölle zu landen.

Die Microservice-Philosophie verfolgend, kann aber auch ein anderer Weg eingeschlagen werden. Wieder ausgehend von den drei Schichten, lässt sich eine Aufteilung in die einzelnen Fachdomänen gegenüberstellen und auf Basis dessen Module entwerfen. In diesem Beispiel wären die beiden Fachdomänen die Verwaltung der Speaker und die Verwaltung der Sessions. Die Module sind also nicht mehr den Schichten folgend horizontal, sondern den Fachdomänen folgend vertikal geschnitten. *Abbildung 7* zeigt den für das Beispiel verfolgten Ansatz.

Die GUI ist in ein separates Modul ausgelagert und somit für die Darstellung der Inhalte beider Fachdomänen verantwortlich. Im Gegensatz zu diesem horizontal geschnittenen Modul sind die weiteren den Fachdomänen zugeordneten Teile auf den Schichten in ein jeweils vertikal geschnittenes Modul aufgeteilt. Zudem findet hier der Service-Provider-Mechanismus seine Anwendung, indem das initiale GUI-Modul als Consumer handelt und die beiden anderen

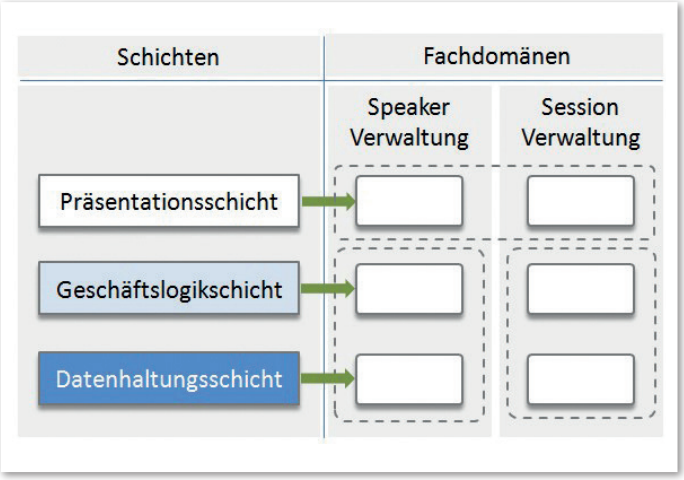


Abbildung 7: Modularer Entwurf

Module die Provider repräsentieren. Damit ist eine maximale Entkopplung zwischen dem GUI-Modul und den anderen beiden Modulen erfolgt, die untereinander ohnehin keine Anhängigkeit voneinander haben. Diese drei Module ließen sich sehr komfortabel auf drei verschiedene Teams zur Entwicklung aufteilen.

Wie im Abschnitt über die Services bereits angesprochen, sind die verschiedenen Provider über ein gemeinsames Interface spezifiziert. Dieses Interface wird im Java-Modulsystem ebenfalls in ein Modul ausgelagert, sodass sich insgesamt die in *Abbildung 8* dargestellten vier Module ergeben. Die entsprechenden Modul-Deskriptoren sehen wie in *Listing 4* aus.

Das GUI-Modul deklariert in seinem Modul-Deskriptor mit dem Schlüsselwort „uses", was für einen Provider-Typ es benötigt. Die Provider-Module geben mit „provides" an, für welchen Provider-Typ sie eine Implementierung liefern, die wiederum hinter dem Schlüsselwort „with" angegeben wird. Das Interface wird im API-Modul angelegt und sieht wie in *Listing 5* aus.

Damit das GUI-Modul die gelieferten Provider zur Laufzeit auch auseinanderhalten kann, führt der Weg über Annotations. Dazu wurden die beiden Annotations „@Sessions" und „@Speaker" erzeugt, die die jeweilige Provider-Klasse markieren. *Listing 6* zeigt die Implementierung des SessionService. Das GUI-Modul nutzt zum Auffinden der Provider den mit Java 6 eingeführten ServiceLocator *(siehe Listing 7)*.

Der Methode „getServiceByAnnotation" wird die Annotation des gewünschten Providers übergeben und innerhalb der Methode mit „ServiceLoader.load(ConferenceService.class)" ein Iterator über alle

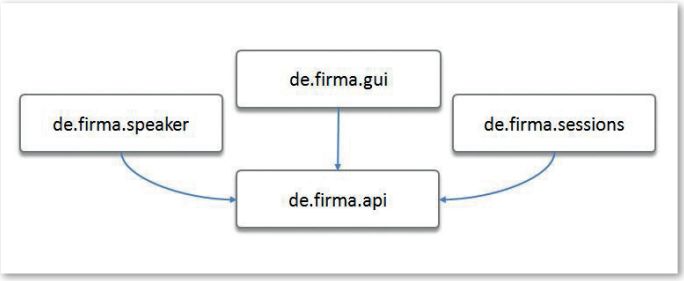


Abbildung 8: Module der Anwendung

```
package de.firma.gui;
import java.util.ServiceLoader;
public class ServiceFactory {
     private ServiceLoader<ConferenceService> services
=

ServiceLoader.load( ConferenceService.class);
     private ConferenceService<?>
getServiceByAnnotation(Class clazz) {
          return services.stream().filter(provider ->
provider.type().isAnnotationPresent(clazz))
  .map(ServiceLoader.Provider::get).findAny().get();
     }
}
```

Listing 7

Implementierungen des „ConferenceService"-Interfaces geholt. Es wird ein Stream erzeugt und dieser bezüglich der Annotation gefiltert, um schließlich den mit der Annotation markierten Provider zu erhalten und zurückzuliefern.

Fazit

Mit Java 9 ist der Fokus wieder mehr auf den Modularisierungsprozess gelenkt, der ein sehr wichtiger Teil für den Bau wartbarer und langlebiger Architekturen ist. Die Modularisierung des JDK und die Integration eines Modulsystems in die Java-Plattform unterstützen den Prozess der agilen Software-Entwicklung und sorgen dafür, dass Java als Sprache auch weiterhin eine große Rolle spielen wird.

Guido Oelmann
guido.oelmann@javaakademie.de

Dipl.-Inform. Guido Oelmann arbeitet als freiberuflicher Software-Architekt, Berater und Trainer. Zu seinen Schwerpunkten gehört neben agilen Entwicklungsmethoden und Software-Architekturen der Einsatz von Java-/Java-EE-Technologien in verteilten Systemen. Er unterstützt Unternehmen durch die Mitarbeit in Entwicklungsprojekten und verfügt über viele Jahre Erfahrung beim Entwurf und der Entwicklung großer IT-Systeme in unterschiedlichen Branchen. Darüber hinaus ist er Autor des Buchs „Modularisierung mit Java 9", das im dpunkt.verlag erscheinen wird.

Domain-Driven Design und Microservices

Philipp Buchholz

Inzwischen gehören die Begriffe „Microservice“ und „Microservice-Architektur“ zum alltäglichen Sprachgebrauch jedes Architekten oder Software-Entwicklers. Trotzdem existieren unterschiedlichste Verständnisse und Meinungen dazu, was genau ein Microservice ist und welche Qualitätseigenschaften er aufweisen sollte. Auch in Bezug auf Microservice-Architekturen gibt es differierende Standpunkte.

Dieser Artikel geht auf einige zentrale Qualitätsmerkmale der Microservices ein. Zusätzlich wird aufgezeigt, wie ausgewählte Grundprinzipien und Methodiken des Domain-Driven Design dabei helfen können, Microservices zu entwickeln und mit passenden Qualitätseigenschaften auszustatten.

Einen Microservice kann man als einen Service beschreiben, der eine abgegrenzte fachliche Aufgabe realisiert und über eindeutig definierte Schnittstellen mit anderen Services interagiert. Er beinhaltet alle technischen Aspekte und Komponenten, um die ihm zugeordnete fachliche Domäne zu realisieren. Hierzu gehört eine eigene Persistenz und damit verbunden, je nach gewählter Persistenz-Technologie, eine eigene Datenbank.

Die Größe von Microservices und Conways-Law

Ein Microservice sollte so geschnitten sein, dass er in überschaubarer Zeit ersetzt werden kann. Im Idealfall reichen für eine komplette Ersetzung wenige Wochen aus. Da viele Variablen eine Rolle spielen, lässt sich in Bezug auf dieses Vorhaben keine absolute Zeitspanne nennen. Organisatorische Rahmenbedingungen wie die Organisation der Entwicklung und des Betriebs, die Zusammenarbeit zwischen Fachbereichen und Entwicklung, die Zusammenstellung des Teams und die Fähigkeiten der Mitarbeiter spielen hier eine große Rolle.

Erfahrungsgemäß sollte man einen Microservice in drei bis maximal vier Wochen komplett neu entwickeln können. Ist das nicht möglich oder nicht realistisch erreichbar, sollte überlegt werden, ob eine andere Architektur besser wäre. Das Conways-Law *(siehe „https://de.wikipedia.org/wiki/Gesetz_von_Conway“),* das die Beziehung zwischen der Struktur einer Software und der Struktur der Organisation beschreibt, sollte auch hier beachtet werden.

Anpassungsfähigkeit an geänderte Bedingungen

Werden sauber abgegrenzte und fein geschnittene Microservices geschaffen, kann sich das resultierende System viel einfacher an geänderte Bedingungen der Unternehmensumwelt anpassen. Entstehen neue Technologien, können einzelne Services diese unabhängig evaluieren. Eine komplette Neuentwicklung einzelner Services auf Basis neuer Technologien wird in überschaubarer Zeit möglich.

Gleichzeitig ist das inhärente Fehlerrisiko bei Änderungen auf die Grenzen des betroffenen Service beschränkt. Bei monolithischen Systemen ist hier erheblicher Testaufwand nötig, um sicherzustellen, dass das komplette System weiterhin korrekt funktioniert. Gleiches gilt bei neuen oder geänderten Anforderungen.

Vertikale Skalierung

Ein weiterer Vorteil ist die vertikale Skalierung, die mit einer solchen Architekturform möglich ist. Bei einem richtigen Schnitt lassen sich gezielt einzelne Microservices und damit bestimmte Fachlichkeiten skalieren, um ein optimiertes Laufzeitverhalten zu erreichen. Das ermöglicht gleichzeitig eine gezielte Reaktion auf geänderte Rahmenbedingungen und die effektive Beseitigung von Bottlenecks in Bezug auf die Performance bestimmter Services. Im Vergleich dazu lässt sich in einem monolithischen System oftmals nur eine horizontale Skalierung durch das Hinzufügen von Rechnerleistung realisieren.

In den meisten dieser Systeme wird das komplette System hochskaliert. Vertikale Skalierung ist, wenn überhaupt, nur in engen Grenzen möglich. Um die beschriebenen Vorteile von Microservices zu erreichen, müssen die nachfolgend beschriebenen Prinzipien eingehalten werden.

Lose Kopplung

Ein Microservice bildet mit vielen anderen Microservices und Komponenten ein System of Systems. Zu diesen anderen Microservices darf nur eine lose Kopplung (LowCoupling, LC) bestehen. Bei einer zu engen Kopplung gehen die genannten Vorteile verloren. Die Kopplung zwischen Microservices wird durch nachfolgend beschriebene Punkte erhöht und negativ beeinflusst.

Veröffentlichung von internen Datenstrukturen

Werden interne Datenstrukturen eines Service von einem anderen verwendet, erhöht sich die Kopplung der beiden Services enorm. Änderungen an einem Service sind dann nicht mehr unabhängig möglich. Lange Regressionstests und das Verlorengehen autonomer, unabhängiger Releases sind die Folge. Darunter fällt die direkte Verwendung der Datenbank oder eines anderen Data-Stores, um Daten auszutauschen („SharedDatabase“).

Auch die Verwendung gemeinsamer Datentypen durch das Teilen von Bibliotheken („SharedLibrary“) erhöht die Kopplung. Um eine solche Kopplung zu verhindern, darf mit einem Microservice nur durch dafür vorgesehene Schnittstellen kommuniziert werden. Diese müssen die internen Realisierungsdetails sauber kapseln. Eine Möglichkeit sind REST-Services oder auch Messaging-Schnittstellen. Das heißt nicht, dass keine Bibliotheken für die Verwendung gemeinsamer Datentypen eingesetzt werden dürfen. Vor der Verwendung muss man sich aber das Risiko einer erhöhten Kopplung bewusst machen.

Ein weiterer Nachteil von geteilten Bibliotheken ist die erhöhte Komplexität durch das notwendige Versionsmanagement von „SharedLibraries“. Oft ist es notwendig, dass einsetzende Services in einer kompatiblen, wenn nicht sogar der gleichen, Technologie oder Programmiersprache entwickelt werden wie die Bibliotheken. Damit wird die Möglichkeit einer technologischen Diversifikation stark eingeschränkt.

Don't Repeat Yourself zwischen Services reduzieren

Oftmals werden geteilte Bibliotheken und Typen eingesetzt, um dem DRY-Prinzip („Don´t Repeat Yourself“) gerecht zu werden. Um sich nicht zu wiederholen und notwendige Änderungen an mehreren Stellen durchführen zu müssen, versucht man, diesen Code in geteilte Bibliotheken auszulagern. Dieses Prinzip sollte innerhalb einer Microservice-Architektur bewusst reduziert werden.

Encapsulation beziehungsweise „Information Hiding“

Das Prinzip der Encapsulation beziehungsweise des „Information Hiding“, das wir aus der Objektorientierung kennen, muss auf Ebene eines Microservice streng beachtet werden; interne Realisierungsdetails sind sauber zu kapseln. Das ist besonders in Bezug auf die Definition von Schnittstellen wichtig.

Hohe Kohäsion

Innerhalb der Objektorientierung beschreibt Kohäsion, wie gut eine Einheit, beispielsweise ein Objekt, eine logische Aufgabe abbildet. Jedes Objekt sollte genau eine wohldefinierte Aufgabe erledigen. Wird mehr als eine Aufgabe abgebildet oder werden für die Erledigung einer Aufgabe mehrere andere Objekte benötigt, ist die Kohäsion niedrig.

Auf Microservices angewandt bedeutet das, dass ein Service eine abgeschlossene, wohldefinierte Fachlichkeit abdecken soll. Mit wohldefinierter Fachlichkeit ist hier eine genau abgegrenzte Domäne beziehungsweise Modell zu verstehen. Sind die zugehörigen Verantwortlichkeiten und Aufgaben über mehrere Services verteilt, so verringert sich die Kohäsion. Sie verringert sich auch, wenn sich innerhalb eines Service fachliche Verantwortlichkeiten befinden, die eigentlich in einem anderen Service abgebildet werden.

Domain-Driven Design konzentriert sich auf das Herausstellen des fachlichen Modells innerhalb der Software. Die Patterns und Design-Methoden des Domain-Driven Design eignen sich aus diesem Grund hervorragend für die Umsetzung von fachlich getriebenen Microservices. Zahlreiche Patterns wie „Aggregate“, „BoundedContext“ und „ContextMap“ oder auch Vorgehensweisen wie das Isolieren der fachlichen Domäne erleichtern das korrekte Modellieren eines Microservice und werden deshalb ausführlich vorgestellt.

Isolation der Domäne

Ein Ziel von Domain-Driven Design ist die Isolierung der fachlichen Domäne von technischen Aspekten. Um dieses Ziel im Kontext von Microservices zu erreichen, muss jeder Microservice intern sauber strukturiert werden. Das Schichtenmodell in *Abbildung 1* stellt nur eine Möglichkeit dar. Die Verantwortlichkeiten der Schichten sind nachfolgend erklärt. Schicht und Layer werden hierbei äquivalent verwendet.

Die Verantwortlichkeiten der Schichten sind:

- **User-Interface-Layer**
  Enthält die Funktionalität für die Anzeige der Benutzeroberfläche. Je nach verwendeter UI-Technologie befindet sich diese Funktionalität Server- oder Client-seitig. Innerhalb dieser Schicht kann jede UI-Technologie verwendet werden, die für den abgebildeten Usecase sinnvoll ist.

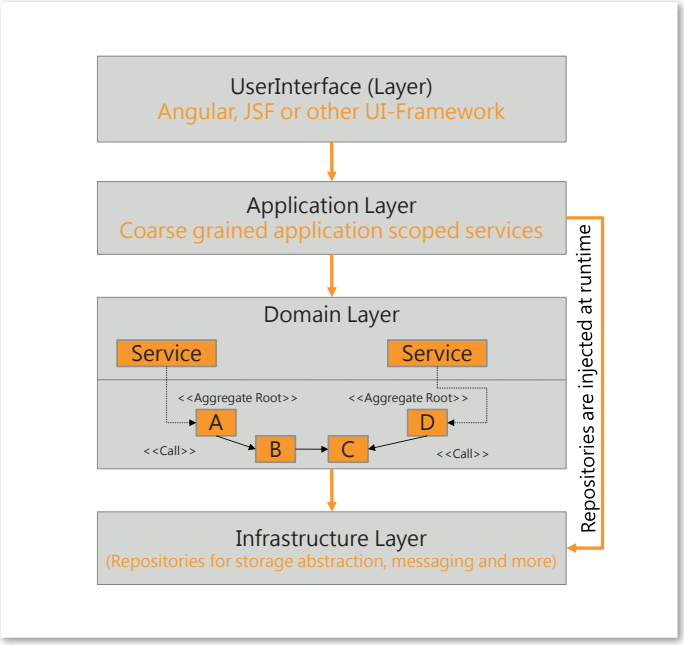


Abbildung 1: Schichtenmodell eines Microservice nach Domain-Driven Design

- **Application-Layer**
  Der Application-Layer bietet auf die UI zugeschnittene Dienste an. Im Vergleich zu Services aus dem Domain-Layer sind diese gröber granuliert („coarse grained"). Ein Service aus dem Application-Layer kann mehrere Services oder fachliche Objekte aus dem Domain-Layer verwenden.
- **Domain-Layer**
  Der Domain-Layer implementiert die Fachlichkeit. Wichtig ist, dass diese sauber von technischen Aspekten wie der Persistenz isoliert ist.
- **Infrastructure-Layer**
  Die Infrastruktur-Schicht enthält technische Verantwortlichkeiten wie Persistenz von Domänen-Objekten und Integrationsaspekte wie Messaging etc.

Die in *Abbildung 1* innerhalb des Domain-Layers gezeigten Begriffe wie „Service", „AggregateRoot" und „Repository" sind zentrale Patterns des domänengetriebenen Designs von Software.

Aggregate

Ein Aggregate fasst mehrere Objekte des Modells innerhalb definierter Grenzen zusammen. Das fachliche Modell enthält immer mehrere Aggregates. Ein Aggregate ist durch eine „AggregateRoot" repräsentiert. Möchte ein Objekt mit den Objekten des Aggregate interagieren, muss es das Aggregate immer über seine AggregateRoot ansprechen und kann über deren Assoziationen auf weitere Objekte des Aggregate zugreifen. Die dadurch entstehenden Grenzen innerhalb des Modells werden für das Sicherstellen der Konsistenz von komplexen Objektbeziehungen verwendet. Die AggregateRoot muss das Einhalten aller Invarianten und Konsistenzbedingungen der enthaltenen Objekte sicherstellen *(siehe Abbildung 2)*.

Transaktionen können innerhalb dieser Konsistenzgrenzen für das Sicherstellen von konsistenten Objekten und deren Beziehungen eingesetzt werden. Da nur durch eine AggregateRoot auf die Objekte eines Aggregate zugegriffen werden kann, ist es logisch, für jede dieser AggregateRoots ein Repository anzubieten, das die Suche nach dieser erlaubt. Objekte, die zu einem Aggregate gehören und keine Root-Objekte sind, dürfen nicht direkt durch ein Repository erreicht werden. Zu diesen kann nur über die AggregateRoot navigiert werden.

Repositories

Repositories abstrahieren von der jeweiligen Persistenz-Technologie und bieten den Objekten innerhalb der Domäne ein Collection-ähnliches Interface an, um Domänen-Objekte zu suchen. In den meisten Fällen sind Methoden für das Suchen von einzelnen oder mehreren Objekten nach eindeutigen Merkmalen vorhanden. Im Domain-Layer befindet sich normalerweise nur das Interface, das die Such-Operationen definiert. Die Technologie-abhängige Implementierung kann zur Laufzeit durch einen Dependency-Injection-Mechanismus zur Verfügung gestellt werden. Damit wird die Domäne frei von Aspekten der Persistenz gehalten.

Services und Domänen-Objekte innerhalb des Domain-Layers

Die Grundprinzipien der Objektorientierung, also die objektorientierte Zerlegung in Klassen und deren Instanzen, Objekte, das Prinzip der Encapsulation und des Information-Hiding, Polymorphie, Abstraktion und Vererbung sind bei der Erstellung eines Domänen-Modells zentral. Es soll zum einen die Struktur, aber auch das Verhalten der Fachlichkeit abbilden. Domänen-Objekte, die nur zum Transport der Daten aus und in die Persistenz dienen, sind zu vermeiden

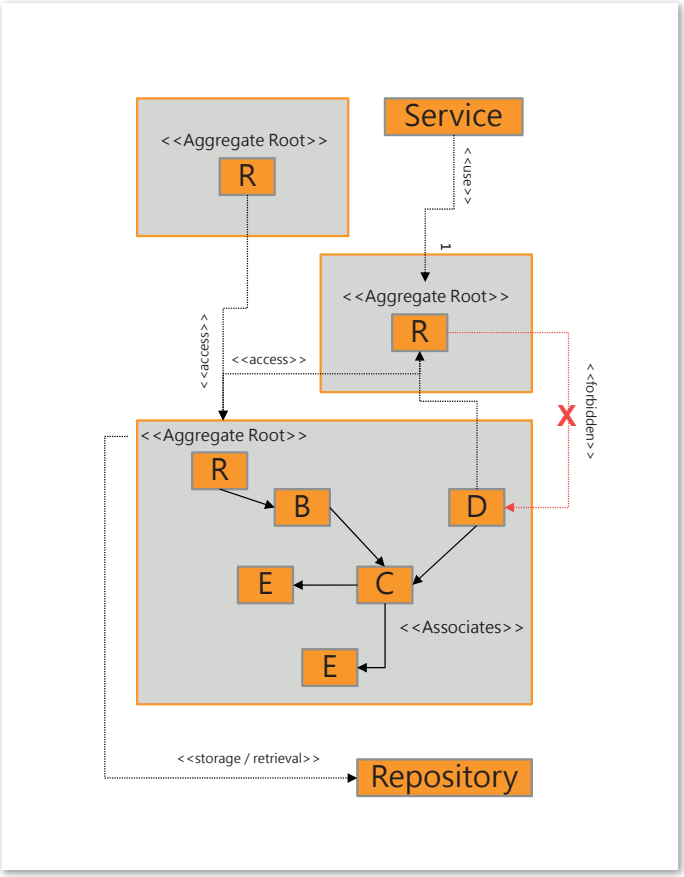


Abbildung 2: Aggregates und AggregateRoots mit Repositories und Services

(siehe „https://martinfowler.com/bliki/AnemicDomainModel.html“). Die genannten Prinzipien der Objektorientierung unterstützen dieses Vorhaben ideal.

Wie beschrieben, ist einer der Vorteile von Microservices die Möglichkeit, diese in kurzer Zeit ersetzen zu können. Dabei ist die Versuchung groß, eine ordentliche Strukturierung und ein ordentliches Design zu vernachlässigen. Aber auch Microservices sollten intern sauber und ordentlich strukturiert sein; Wartbarkeit, Erweiterbarkeit und Verständlichkeit sind auch hier wichtige Qualitäts-Eigenschaften. Deshalb empfiehlt der Autor auch hier die Anwendung eines sauberen, objektorientierten Designs.

Teilweise ergibt sich aber die Situation, dass bestimmte fachliche Logik nicht zu einem bestehenden Domänen-Objekt passt. In diesem Fall können innerhalb des Domain-Layers fachlich orientierte Services angeboten werden. Diese Services enthalten die fachliche Logik und bedienen sich, wenn notwendig, des Verhaltens der anderen Domain-Objekte. Auch fachlich getriebene Services müssen von technischen Aspekten isoliert und sauber gehalten werden.

BoundedContext

Innerhalb von Projekten, die umfangreiche fachliche Anforderungen abdecken sollen, entstehen oft Modelle, die sehr groß und umfangreich sind. Es ist nicht realistisch, ein einziges einheitliches und widerspruchsfreies Modell in solchen Projekten zu etablieren. Das scheitert an der organisatorischen Komplexität – mehrere Entwicklungsteams, die unterschiedliche Systemteile entwickeln – und an der Komplexität der fachlichen Anforderungen.

Jedes Entwicklungsteam benötigt einen gewissen Freiraum für den Einsatz von Kreativität und zur Findung von adäquaten Lösungen. Dieser Freiraum wird durch den organisatorischen Overhead, der für die Aufrechterhaltung eines einzigen einheitlichen Modells entsteht, signifikant eingeschränkt. Auf der anderen Seite müssen die entwickelten Modelle, die die fachlichen Anforderungen abbilden, in sich einheitlich und geschlossen, also kohäsiv sein.

Domain-Driven Design begegnet diesen Problemen mit dem Pattern des „BoundedContext“. Dabei entstehen mehrere Modelle, die in sogenannten „BoundedContexts“ gruppiert sind. Jeder BoundedContext enthält ein wohldefiniertes und einheitliches Modell, das mit anderen BoundedContexts über klar definierte Beziehungen kommuniziert. Eine Vereinheitlichung der Modell-Begriffe und -Objekte über die Grenzen von BoundedContexts hinweg wird nicht angestrebt. Ganz im Gegenteil: Teammitgliedern, die sich außerhalb eines BoundedContext und damit innerhalb eines anderen BoundedContext befinden, wird die Freiheit eingeräumt, ein eigenes Modell und einen eigenen Dialekt der Ubiquitous-Language zu entwickeln und anzuwenden.

Innerhalb eines BoundedContext kommen zur Bildung von Konsistenzgrenzen und zur Strukturierung die bekannten Aggregates und deren AggregateRoots zum Einsatz *(siehe Abbildung 3)*.

ContextMap

Durch die Definition von BoundedContext und damit von mehreren sauber gekapselten Modellen ist die Freiheit geschaffen, diese Modelle unabhängig zu entwickeln und zu definieren. Trotzdem

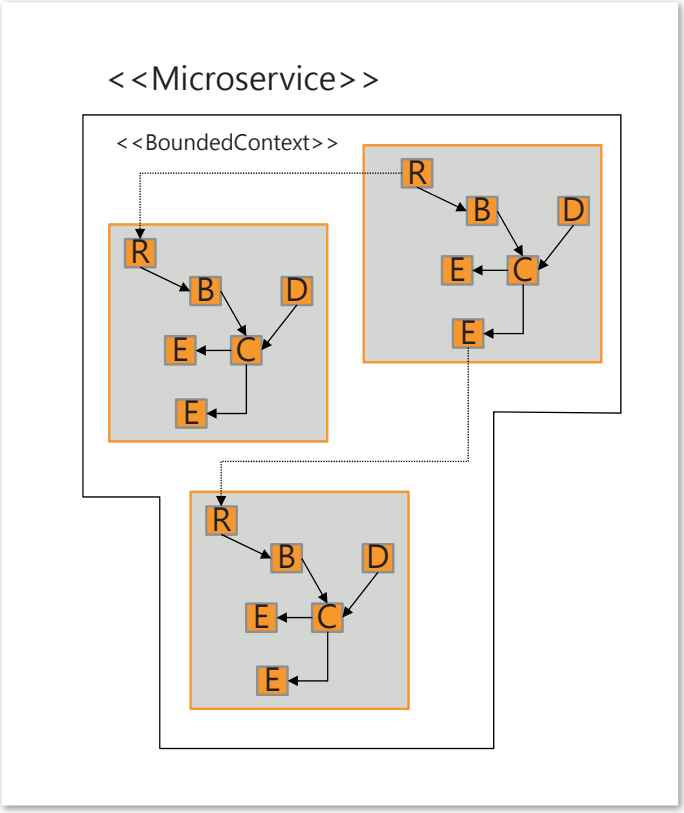


Abbildung 3: BoundedContext mit seinen AggregateRoots und deren Entitäten beziehungsweise ValueObjects

bestehen weiterhin Beziehungen zwischen diesen Modellen. Um Verwirrungen auf technischer und/oder organisatorischer Ebene zu verhindern, müssen diese uni- oder bidirektionalen Beziehungen und die Teilnehmer an diesen genau bekannt sein. Dafür können „ContextMaps“ verwendet werden. Sie zeigen, welche Verantwortlichkeit, welches Modell, in welchem BoundedContext abgebildet wird, und verdeutlichen die Beziehungen zwischen diesen Contexts *(siehe Abbildung 4)*. Das Beispiel zeigt eine ContextMap mit zwei BoundedContexts. Die AggregateRoots dieser BoundedContexts, die für andere interessant sind, sind als geteilt hervorgehoben.

Korrelation zwischen BoundedContext, ContextMap und Microservices

Einige der gewünschten Eigenschaften von Microservices, wie lose Kopplung auf organisatorischer sowie technischer Ebene, lassen sich sehr gut über die Konzepte von BoundedContexts abbilden. Wenn man innerhalb eines Microservice einen BoundedContext umsetzt, wird eine hohe Kohäsion erreicht. Die Freiheit bei der Modell-Definition und die Möglichkeit, eine eigene Ausprägung der Ubiquitous-Language und damit einen eigenen organisatorischen Rahmen zu schaffen, erhöhen die lose Kopplung auf technischer und organisatorischer Ebene.

Nach den grundlegenden Eigenschaften und einigen wichtigen Patterns des Domain-Driven Design noch einige Tipps für die Realisierung von Microservices.

Verfrühte Teilung in einzelne Services vermeiden

Ist der Beschluss gefasst, eine Microservice-Architektur zu erstellen, muss unbedingt realistisch festgestellt werden, in welchem Maße die Software-Entwickler und -Architekten mit der fachlichen

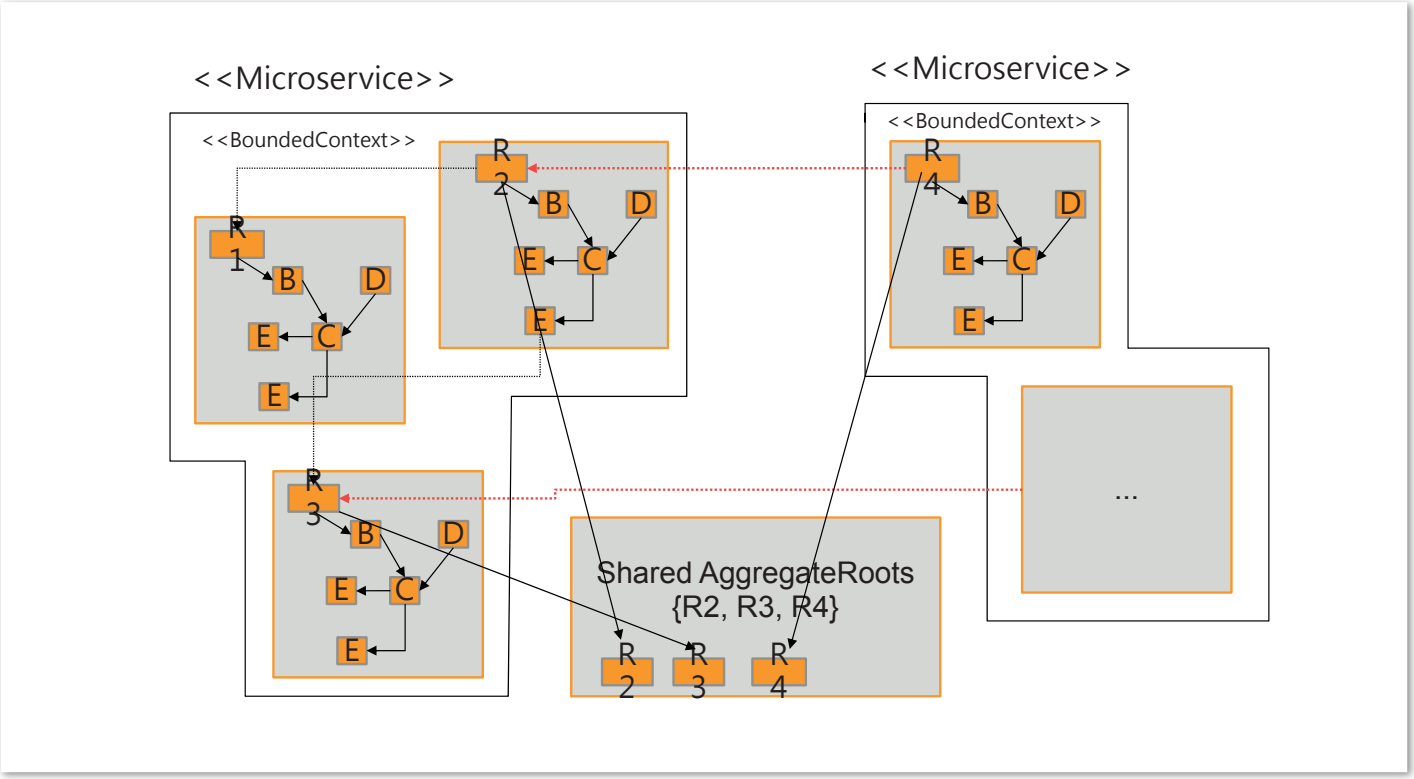


Abbildung 4: ContextMap mit mehreren BoundedContexts und deren geteilte Objekte

Domäne vertraut sind. Ist die Kenntnis der Fachlichkeit gering, lohnt es sich, mit einer monolithischen Architektur zu starten und Erfahrungen zu sammeln. Diese Erfahrungen sind substanziell, um die Services fachlich sauber zu schneiden und damit eine hohe Kohäsion zu erreichen. Das heißt nicht, dass der Monolith bis zum Ende eines Projekts bestehen muss. Vielmehr rät der Autor zu einer evolutionären Vorgehensweise, bei der der Monolith genau dann zerschnitten wird, wenn die fachlichen Schnittpunkte herausstechen. Die Fähigkeit, diese zu erkennen, kann nur mit der Kenntnis der Fachlichkeit erlangt werden.

Die Architektur innerhalb eines Projekts muss dementsprechend ebenfalls evolutionär gestaltet sein und sich weiterentwickeln. Die Entscheidung, welche Services vorhanden sein sollen und welche Verantwortlichkeiten diese wahrnehmen, kann nicht nach einer bestimmten Phase endgültig entschieden werden. Es empfiehlt sich, den Monolith nicht zu groß werden zu lassen und auf ein sauber strukturiertes Inneres zu achten.

Verteilte Transaktionen vermeiden

Werden verteilte Transaktionen eingesetzt, entsteht eine nicht gewollte Kopplung zwischen den Services, die an einer Transaktion teilnehmen. Das widerspricht der losen Kopplung zwischen den Services, die wir erreichen wollen. Es ist ratsam, mit CompensationOperations oder einer Art AuditLog zu arbeiten, um beispielsweise die Unterstützung des Command-Query-Responsibility-Segregation-Prinzips (CQRS) realisieren zu können.

Fazit

Microservice-Architekturen können eine entscheidende Unterstützung bei der Bewältigung der Herausforderungen unserer sich rasch wandelnden Umgebung sein. Die schnelle Adaption von neuen Technologien und die Möglichkeit der vertikalen Skalierung sind hierbei entscheidend. Mindestens genauso wichtig ist allerdings zu beachten, dass auch die Organisation bereit sein muss, diesen sehr flexiblen, aber auch komplexen Ansatz zu unterstützen.

Werden die beschriebenen Prinzipien beachtet und ein evolutionäres Vorgehen bei der Architekturarbeit gewählt, kann ein entscheidender Schritt in Richtung einer flexiblen und wandelbaren Architektur – weg vom klassischen Monolith, der alle Verantwortlichkeiten vereint – getan werden.

Philipp Buchholz
philipp.buchholz@gmx.net

Philipp Buchholz ist aktuell als Senior Software Engineer in Stuttgart und Umgebung tätig. Zurzeit wirkt er beim Aufbau einer neuen Microservice-Architektur im Gesundheitswesen mit. Hier arbeitet er mit den Prinzipien und Methoden des Domain-Driven Design, um einen fachlichen Schnitt von Microservices zu erreichen. Als Entwicklungs-Plattform kommen der Spring-Stack und Java EE 7 zum Einsatz.

„Serverless" Java

Niko Köbler, Freelancer, www.n-k.de

„Serverless Computing" ist als Trend-Thema derzeit in aller Munde. Kaum eine Konferenz, kaum ein Magazin, kaum ein Online-Portal, das sich derzeit nicht mit dem neuesten Cloud-Service beschäftigt. Der Artikel wirft einen Blick auf dieses Thema – speziell unter dem Java-Gesichtspunkt.

```
function countChars(input) {
  return input.length;
}
```

Listing 1

```
public class SimpleJavaFunction {
  public int chountChars(String input) {
    return input.length();
  }
}
```

Listing 2

Der Name „Serverless" ist im Grunde sehr irreführend. Denn „Serverless" heißt eben nicht, dass Programmcode nicht mehr auf Servern ausgeführt wird. Vielmehr geht es um das Management von Servern – und das kann bekanntlich sehr aufwändig sein. „Serverless" möchte eben dieses Server-Management, also die Bereitstellung und Aktualität von (virtuellen) Maschinen, Containern und Ablaufumgebungen, die richtige Skalierung und Verfügbarkeit etc., von den Entwicklern fernhalten, sodass diese sich auf ihren Anwendungs- und Funktionscode sowie die bestmögliche Nutzung der Ressourcen (wie einer Ablaufumgebung) konzentrieren können. Um alles andere kümmert sich der Cloud-Service-Anbieter.

Function as a Service

Neben dem Namen „Serverless" hat sich auch der Begriff „Function as a Service" (FaaS) etabliert. Dieser beschreibt schon etwas besser, um was es sich – zumindest im Computing-Segment von „Serverless" – dreht: Funktionen als die zu verwaltende Einheit. Der Code, der zur Ausführung gebracht wird, soll in der Größe beziehungsweise der Form einer Funktion geschrieben sein. Die Ausführung der Funktion wird von einem Cloud-Anbieter als Dienst (Service) zur Verfügung gestellt –eben „Function as a Service".

Eine Funktion hat klassischerweise ein oder mehrere Eingabe-Parameter und einen Rückgabewert. Zusätzlich fokussiert sich der Code einer Funktion gemeinhin auf eine einzige bestimmte Aufgabe. Sehr einfach ist das beispielsweise mit JavaScript zu verdeutlichen *(siehe Listing 1)*. Der Aufruf der Funktion „getChars()" mit dem Wert „Hello, World!" würde den Rückgabewert „13" liefern.

In der Tat ist der Einsatz von Skript-basierten Sprachen wie JavaScript (mit Node.js) und Python im „Serverless"-Umfeld sehr populär. Hier geht es jetzt darum, was mit Java möglich ist. In Java ist zur Ausführung der Funktion aus dem Beispiel außen herum noch eine Klasse erforderlich, denn nur über eine Klasse ist Java-Code aufruf- und ausführbar. Eine Klasse enthält typischerweise eine oder mehrere Methoden, aber keine Funktion. *Listing 2* zeigt den Funktionscode in Java.

Packen wir nun diese Klasse in eine „.jar"-Datei und laden sie zu einem FaaS-Cloud-Anbieter hoch, haben wir bereits unsere erste „Serverless"-Java-Funktion. Mittlerweile gibt es mehrere Cloud-Anbieter, die Serverless-Computing als einen Service mit verschiedenen unterstützten Programmiersprachen anbieten:

- **Amazon Web Services mit AWS Lambda [1]**
  Java, JavaScript, Python, C#
- **Microsoft mit Azure Functions [2]**
  Java, JavaScript, Python, C#, PHP
- **Google mit Cloud Functions [3]**
  JavaScript
- **IBM Bluemix mit OpenWhisk [4]**
  JavaScript, Python, Swift

In diesem Artikel konzentrieren wir uns auf die Arbeit mit AWS Lambda *[5]* als „Serverless"-Plattform.

Skalierung, Caching und State

Der FaaS-Anbieter kümmert sich nicht nur darum, dass der Funktions-Code ausgeführt wird, sondern auch darum, dass dieser mit den steigenden oder sinkenden Requests auch perfekt skaliert. Wird also die Funktion hundert Mal zur gleichen Zeit aufgerufen, so wird sie auch hundert Mal parallel ausgeführt. Ist nur eine einzige Instanz-Ausführung notwendig, existiert auch nur eine Instanz der Funktion. Benötigt man gar keine Instanz, da keine Anfragen an die Funktion gerichtet werden, so existiert auch keine Instanz und letztendlich zahlt man auch nur für die Ressourcen, die während der Ausführung der Funktion in Anspruch genommen werden. Keine Requests gleich keine Ausführung gleich keine Kosten. Das ist echtes „Pay per use" – die reine Bereitstellung oder das Deployment des Funktionscodes kosten also nichts.

Damit diese automatische Skalierung funktioniert, sind noch ein paar weitere Besonderheiten zu berücksichtigen. Da man durch die automatische Skalierung nicht weiß, auf wie vielen und welchen Maschinen die Funktion letztendlich läuft, dürfen im Code auch keine Umgebungs-spezifischen Zugriffe für die Ausführung implementiert sein. Zudem sind die Container, in denen die Funktionen ausgeführt werden, flüchtig und werden unter Umständen nach der Ausführung verworfen. Die Synchronisation eines Zustandes („State") über mehrere (parallele) Funktionen hinweg oder das Zwischenspeichern („Cache") von Ergebnissen für eine erneute Funktionsausführung darf nicht (allein) im Ausführungscontainer geschehen, sondern muss zwingend in einem anderen permanenten Speichermedium wie einer relationalen Datenbank, Amazon DynamoDB, Redis oder Amazon S3 erfolgen.

AWS Lambda speichert zwar die Funktions-Container für eine mögliche Wiederverwendung zwischen, jedoch ist zu keinem Zeitpunkt garantiert, dass der gleiche Container bei einem erneuten Funktionsaufruf erneut verwendet wird. Somit kann man Ressourcen zwar innerhalb eines Containers cachen, um bei einem erneuten Aufruf möglicherweise diese Ressourcen wiederzuverwenden und Zeit bei der Instanziierung der Objekte zu sparen, man darf sich aber nicht darauf verlassen. Verlassen darf man sich hingegen darauf, dass es zum Zeitpunkt der Funktionsausführung nur einen einzigen gleichzeitigen Zugriff auf den Code gibt. Aufwand hinsichtlich Synchronisation und Thread-Sicherheit innerhalb des eigenen Codes kann man sich damit sparen.

Event-driven

AWS-Lambda-Funktionen werden typischerweise nicht mit einem simplen Datentyp wie einem String oder einem Integer aufgerufen, wie er im Beispiel definiert wurde, sondern mit einem (JSON-)Objekt, das eine Map-Struktur repräsentiert „{ "key": "value" }". Diese

Objekte stellen Events von verschiedenen AWS-Diensten dar, da Lambda-Funktionen Ereignis-getrieben aufgerufen und ausgeführt werden.

Das typische „Hello World"-Beispiel für Serverless ist der Usecase, dass ein Bild im Originalformat zu Amazon S3 hochgeladen wird. Dieser Upload erzeugt dann ein S3-Event, auf dem eine Lambda-Funktion registriert ist und ausgeführt wird, sobald dieses Event von S3 erzeugt wurde. Das Event enthält Informationen über die neu gespeicherte Datei (wie S3-Bucket-Name und Pfad/Dateiname). Die Lambda-Funktion kann dann mithilfe dieses Events die Datei aus S3 herunterladen, aus dem Originalformat ein Thumbnail berechnen und dieses dann an einer anderen Stelle wieder hochladen, sodass es zum Beispiel von einer Website zur Anzeige verwendet werden kann.

Java-API

Damit man den Umgang mit den verschiedenen AWS-Events nicht umständlich per Hand über ein Map-Objekt implementieren muss und auch einen Rahmen für die Erstellung einer sogenannten „Handler"-Klasse (die Klasse, die die Handler-Funktion beinhaltet) bekommt, hat AWS zwei wichtige Java-Bibliotheken als API zur Verfügung gestellt (hier als Maven-Dependency, siehe *Listing 3*).

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-lambda-java-core</artifactId>
   <version>1.1.0</version>
</dependency>
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-lambda-java-events</artifactId>
  <version>1.3.0</version>
</dependency>
```

Listing 3

Die Bibliothek „aws-lambda-java-core" bietet hauptsächlich Basis-Interfaces an, mit denen unsere Java-Funktion arbeitet. So beispielsweise das „RequestHandler"-Interface, das die Handler-Klasse implementiert, und ein „Context"-Objekt, das Zugriff auf die Lambda-Umgebung (wie einen Logger oder Ähnliches) gibt. Die Bibliothek „aws-lambda-java-events" beinhaltet, wie der Name bereits vermuten lässt, vordefinierte Objekte für verschiedene AWS-Events, darunter auch das genannte „S3Event". Mit diesen beiden Bibliotheken sieht die Thumbnail-Lambda-Funktion dann wie in *Listing 4* aus.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.Request-
Handler;
import com.amazonaws.services.lambda.runtime.events.
S3Event;

public class ThumbnailHandler implements
RequestHandler<S3Event, Void> {
  public Void handleRequest(S3Event input, Context
context) {
    context.getLogger().log("Received event: " + in-
put.toJson());
    //...
  }
}
```

Listing 4

Das „RequestHandler"-Interface definiert als Typ-Parameter die Ein- und Ausgabe-Typen der Funktion, die sich dann in der zu implementierenden „handleRequest()"-Methode wiederfinden. Das „S3Event" ist, wie bereits beschrieben, das Eingabe-Objekt für die Funktion und enthält alle Informationen zum gerade gespeicherten Objekt. Da die Funktion keinen expliziten Rückgabewert benötigt (sie lädt ja nur ein neues Bild zu S3 hoch), ist „Void" als Rückgabewert definiert.

Die genannte und weitere Bibliotheken, die für den Funktionscode notwendig sind (etwa für die Thumbnail-Berechnung), müssen gemeinsam mit dem eigenen Code in ein sogenanntes „Fat-JAR" gepackt und zu AWS Lambda hochgeladen werden. Das „Fat-JAR" enthält Abhängigkeiten, die der eigene Code benötigt, in einer einzigen Datei, der „.jar"-Datei. Weitere JARs sind dann zur Laufzeit nicht mehr nötig. Bei Maven ist das „Maven-Shade-Plug-in" *[6]* ein mögliches Werkzeug, um ein solches Fat-JAR zu erstellen *(siehe Listing 5)*. Die damit erzeugte „.jar"-Datei kann dann zu AWS Lambda hochgeladen werden.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>3.0.0</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Listing 5

Serverless goes HTTP

Interessant wird es, wenn man AWS-Lambda-Funktionen nicht nur innerhalb der Cloud-Services verwendet, sondern auch von extern aufrufen kann. Sozusagen die Funktion als konsequente Evolution des Microservice hin zu einem Nanoservice. Mit dem API-Gateway *[7]* von Amazon wird dies sehr einfach möglich, da es eingehende Requests zu einem Event transformieren und damit eine Lambda-Funktion aufrufen kann. Die Ausgabe, also das Ergebnis der Funktion, wird dann wieder vom API-Gateway in eine Response gewandelt und zum Aufrufer zurückgeschickt. Das geht alles über HTTPS sowie mit optionaler Authentifizierung und Autorisierung (über Amazon IAM und Cognito), sodass das API nicht einfach von jedem Dritten genutzt werden kann und Daten nicht öffentlich einsehbar sind. Nicht nur für Mobile-Anwendungen kann damit schnell und ohne großen Aufwand ein leistungsfähiges Backend erstellt werden.

Ein durch das API-Gateway erstelltes Request-Event enthält alle notwendigen Daten des HTTP-Aufrufs, wie der aufgerufene Pfad, Query-Parameter, die mitgesendeten Header-Informationen, gegebenenfalls Autorisierung und den Body-Payload (bei „POST"- und „PUT"-Requests). Bei der Konfiguration des API-Gateways

```
{
  "body": "{\"test\":\"body\"}",
  "resource": "/{proxy+}",
  "requestContext": {...},
  "queryStringParameters": {
    "foo": "bar"
  },
  "headers": {...},
  "pathParameters": {
    "proxy": "path/to/resource"
  },
  "httpMethod": "POST",
  "stageVariables": {
    "baz": "qux"
  },
  "path": "/path/to/resource"
}
```

Listing 6

kann man entscheiden, ob auf einem bestimmten URL-Pfad und einer bestimmten HTTP-Methode („GET", „POST" etc.) eine dedizierte Lambda-Funktion aufgerufen oder ob über ein sogenanntes „Proxy"-Template alle URLs ab diesem Pfad-Parameter mit allen HTTP-Methoden („=ANY") an die Lambda-Funktion weitergeleitet werden sollen. Die Entscheidung, was bei den unterschiedlichen Pfad-Parametern und Methoden passiert, obliegt im letzteren Falle dann der aufgerufenen Lambda-Funktion *(siehe Listing 6)*.

Das Rückgabe-Objekt muss auch wieder eine Map mit den für eine HTTP-Response relevanten Feldern sein, wie Status-Code, Header und Body. Das Body-Feld ist vom Typ „String" und enthält das eigentliche Ergebnisobjekt. Der Standard-Content-Type für die Response ist „application/json". Falls das Ergebnisobjekt in einem anderen Format gesendet wird, muss das entsprechend in einem Header-Feld angegeben sein. Eine Beispiel-Response kann wie in *Listing 7* aussehen.

Für das API-Gateway-Request- und Response-Event gibt es in der „aws-lambda-java-events"-Bibliothek kein vordefiniertes Event. Hier müssen momentan die einzelnen Elemente noch manuell aus der Input-Map gelesen und in die Output-Map geschrieben werden (oder man schreibt sich alternativ zwei entsprechende Hilfsklassen). Allerdings gibt es aus den AWS-Labs ein Projekt namens „Serverless Java container" *[8]*, das über verschiedene Dependencies nicht nur ein Request- („AWSProxyRequest") und Response-Objekt („AWSProxyResponse") zur Verfügung stellt, sondern auch experimentelle Integrationen von AWS Lambda mit JAX-RS (Jersey), Spring und dem Spark Framework anbietet. Ein interessanter Ansatz, der sich allerdings noch in einem sehr frühen Stadium befindet.

Fazit

Dieser Artikel umreißt nur einen sehr kleinen Teil von „Serverless Computing" oder FaaS und konzentriert sich auf den Umgang von AWS-Lambda-Funktionen mit Java. Dabei führen mehrere Wege zum Ziel – je nachdem, welches man verfolgt. Gerade die Integration mit dem API-Gateway machen Lambda-Funktionen zu einem sehr mächtigen Werkzeug in der Erstellung von kleinen, isolierten Services.

Insgesamt steht das ganze Thema „Serverless" noch sehr am Anfang, auch was das Tooling angeht, hier ist noch ein enormes Entwicklungspotential vorhanden. Lassen wir uns treiben und gestalten die Evolution maßgeblich mit. Das funktioniert sehr gut, da die Cloud-Anbieter hier nicht einfach ihre Ideen und Sichtweisen in Form von Services wiedergeben, sondern recht intensiv auf das hören, was von den Anwendern und der Community gefordert und zurückgemeldet wird.

```
{
  "body": "{\"foo\":\"bar\"}",
  "headers": {
    "Content-Type": "application/json"
  },
  "statusCode": 200
}
```

Listing 7

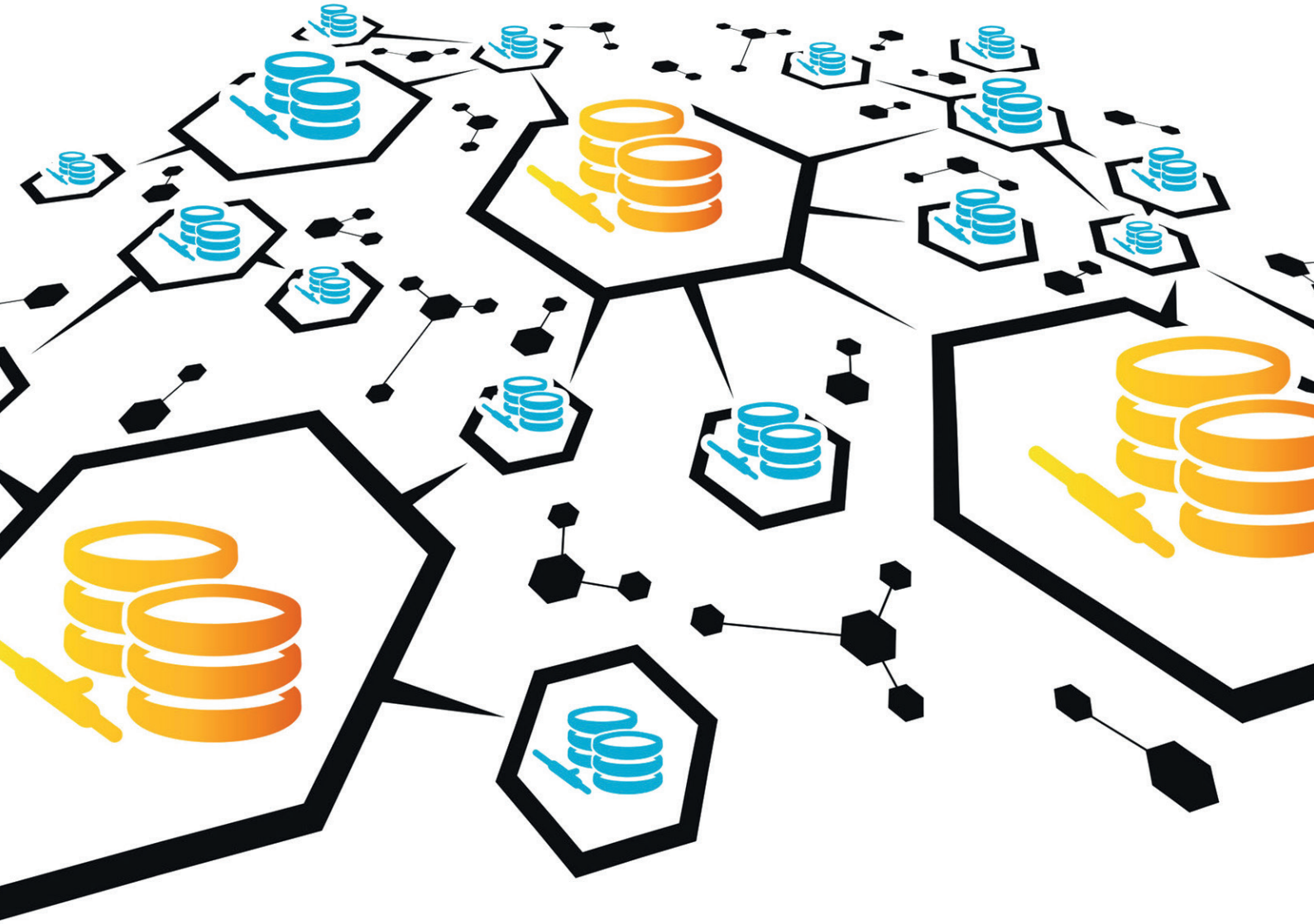
Wer mehr über „Serverless" sowie dessen Anwendungsfälle und Einsatzszenarien wissen möchte, dem ist das im Herbst erscheinende Buch des Autors mit dem Titel „Serverless Computing" *(siehe „http://serverlessbuch.de")* empfohlen, das ebenfalls seinen Schwerpunkt auf AWS-Services legt und tiefer auf Details in verschiedenen Bereichen der „Serverless"-Welt eingeht.

Weiterführende Links

[1] *https://aws.amazon.com/serverless*
[2] *https://azure.microsoft.com/services/functions*
[3] *https://cloud.google.com/functions*
[4] *https://developer.ibm.com/openwhisk*
[5] *https://aws.amazon.com/lambda*
[6] *https://maven.apache.org/plugins/maven-shade-plugin*
[7] *https://aws.amazon.com/api-gateway*
[8] *https://github.com/awslabs/aws-serverless-java-container*

Niko Köbler
niko@n-k.de

Niko Köbler macht irgendwas mit Computern, oft „Serverless" in der Cloud, meist (funktional) auf der JVM. Er ist Co-Lead der JUG Darmstadt, Autor und Sprecher auf internationalen Konferenzen. Niko tweetet unter „@dasniko".



Performance von Datenbank-Zugriffen mit JPA

Thomas Bröll, Trivadis GmbH

Bei Datenbank-Administratoren haben Java-Anwendungen den Ruf schlechter Performance und seltsamer Datenbank-Zugriffe. Schlimmstenfalls ist die Antwortzeit der Anwendung nur manchmal schlecht, aber meist akzeptabel, weshalb die Probleme auch nicht die notwendige Aufmerksamkeit bekommen.

Die Anwender beschweren sich nur selten und mit den Testdaten funktioniert es bestens. Eine der Ursachen, aber auch die Vorzüge des Java-Persistence-API (JPA) liegen darin, dass es eine sehr gute Abstraktion von relationalen Datenbanken darstellt. Spätestens bei der Betrachtung der Performance müssen der relationale und der objektorientierte Ansatz jedoch unbedingt gemeinsam betrachtet werden. Dieser Artikel zeigt durch bewährte Lösungen für häufig auftretende Probleme Wege zu besserer Performance auf.

JPA wurde als JSR 220 schon im Jahr 2006 *[1]* veröffentlicht und feierte damit bereits sein 10-jähriges Jubiläum. Schon deutlich früher konnte man Berührung mit heute gängigen Implementierungen wie Hibernate haben, heute ist die Technologie aus Projekten mit SQL-Datenbanken nicht mehr wegzudenken. JPA geht dabei als Framework für objektrelationales Mapping von zwei wesentlichen Annahmen aus: Die Denkweise des Entwicklers liegt in der objektorientierten Welt. Objekte, Attribute und Relationen bestimmen das Denken, nicht SQL-Befehle, Cursor und Ausführungspläne. Außerdem befindet sich die Anwendung in der Regel auf einem physisch getrennten System, dem Anwendungsserver.

Die Wurzel des Übels

Genau diese Trennung von Datenbank und Anwendungsserver ist die Ursache für den Großteil der Probleme. Im Gegensatz beispielsweise zu Stored Procedures kostet selbst ein einfacher Zugriff auf die Datenbank messbare Zeit, dadurch wird die Anzahl von Interaktionen mit der Datenbank plötzlich relevant. Außerdem ist häufig die Anzahl der Zugriffe aus Sicht des Entwicklers nicht deterministisch, weil sie zur Laufzeit von den konkreten Daten abhängig ist. Es spielt eine Rolle, ob für zehn Postleitzahlen nur eine oder zehn Städte geladen werden müssen. Beides zusammen führt zu Performance-Problemen – schlimmstenfalls nur manchmal.

Abhängig vom konkreten Daten-Szenario funktioniert ein Anwendungsfall, ist einfach nur langsam oder läuft auf Timeouts und ist nicht nutzbar. Wesentliches Merkmal dabei ist die Latenz im Netzwerk („Ping Roundtrip“) beziehungsweise die Laufzeit einer einfachen SQL-Abfrage mit leerer Ergebnismenge für die Java-Anwendung. Selbst wenn die Latenz im lokalen Netzwerk auf den ersten Blick mit unter einer Millisekunde gut aussieht, ist diese unter Last nicht garantiert und verschlechtert die Zeit für eine Interaktion mit der Datenbank.

```
@Entity
public class City {
  @Id @GeneratedValue long id;
  String name;
  @ManyToOne State state;
@OneToMany(mappedBy="city")
Collection<PostalCode> postalCodes;
… getter und setter …
}
```

Listing 1

```
State state=…
…
City city=new City();
city.setName(name);
city.setState(state);
entityManager.persist(city);
```

Listing 2

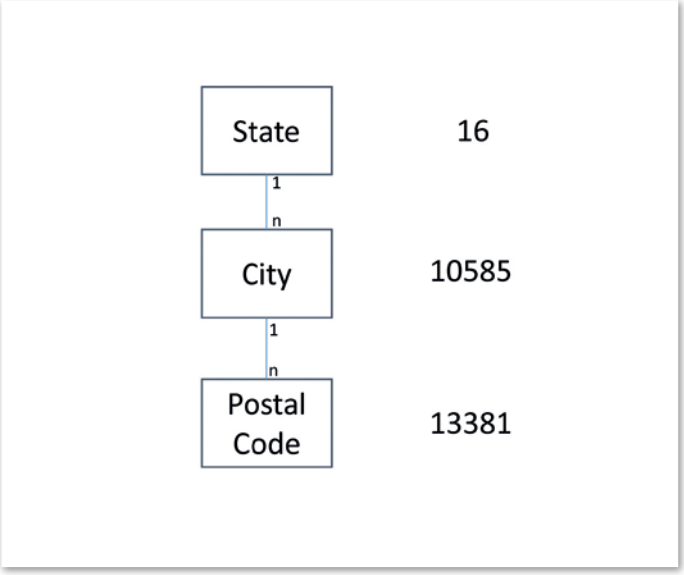


Abbildung 1: Hierarchie aus drei Objekten

Probleme analysieren

Trifft man auf ein solches Problem, gilt es zuerst, es nachstellbar zu machen und auf den betroffenen Systemen zu analysieren. Sollte sich hier eine Ursache nachweisen lassen, sind die jeweiligen Ansprechpartner mit einer Analyse und Problemlösung zu betrauen. Sie werden etwa einen fehlenden Index in der Datenbank anlegen oder den Java-Code optimieren. In diesem Beispiel gehen wir jedoch von folgendem Szenario aus: Die Performance ist nicht ausreichend, obwohl sowohl Datenbank- als auch Applikationsserver keine nennenswerte Belastung zeigen. Daher sind folgende zusätzliche Analysen interessant:

- **Transaktionen und SQLs**
  Die Anzahl der Transaktionen insgesamt sowie die jeweils in einer Transaktion ausgeführte Anzahl der Statements
- **Top-10-SQL-Statements**
  Getrennt nach Anzahl der Ausführungen sowie maximaler und kumulativer Laufzeit
- **Thread-Analyse der Java VM**
  Mit welchen Methoden wird wie viel Zeit verbracht? Im einfachsten Fall kann geprüft werden, wo die Ausführung bei in kurzen Abständen wiederholten Thread-Dumps steht, etwa mit den Werkzeugen „jstack“ oder „jvisualvm“
- **Statistiken des „PersistenceContext“**
  Etwa mithilfe der Hibernate-Statistiken einen Blick auf die Anzahl und Laufzeiten aus Sicht des Anwendungsservers ermöglichen

Im Szenario ergeben die genannten Analysen folgendes Ergebnis: Die Datenbank-Analyse und die Hibernate-Statistiken zeigen viele Statements pro Transaktion, hierbei sind die Statements allerdings schnell außerordentlich häufig. Die Java VM verbringt die Zeit normalerweise in einem Methoden-Aufruf, der versucht, Daten aus der JDBC-Verbindung zu lesen. Beide Ergebnisse zusammen beweisen,

dass die Zeit durch die Kommunikation auf dem Netzwerk-Medium verbraucht wird, obwohl dieses ebenfalls nicht ausgelastet ist. Lokale Kommunikation beschleunigt den Code extrem, hohe Bandbreiten im Netzwerk lösen jedoch leider nicht das Problem der Latenz und grundsätzlicher Grenzen von TCP.

Daten schreiben

Aus einer Klasse ist schnell eine JPA-Entität erstellt: Es genügt ein POJO mit wenigen Attributen, um Annotationen zu erweitern, und schon können Daten in die Datenbank geschrieben werden. Im folgenden Beispiel sollen Geodaten des Projekts OpenStreetMap aus einer CSV-Datei in eine Hierarchie von drei Objekten (Bundesland, Stadt und Postleitzahl) mit den dargestellten Häufigkeiten geschrieben werden *(siehe Abbildung 1)*. Insgesamt handelt es sich dabei um ca. 24.000 Objekte, die den Weg in die Datenbank antreten. Am Beispiel der Entität „City" sieht das in einer ersten Version wie in *Listing 1* aus.

Die Daten liegen im CSV-Format als kartesisches Produkt vor, zum Beispiel vervielfachen sich die 16 Bundesländer auf alle Zeilen der im CSV enthaltenen 13.381 Postleitzahlen. Jedes fachliche Objekt wird beim ersten Vorkommen in einer Zeile des CSV erzeugt, persistiert und für folgende Zeilen weiter als verknüpftes Objekt genutzt, hier am Beispiel einer Stadt mit dem zugehörigen Bundesland *(siehe Listing 2)*.

Diese erste Version lief mit einer lokalen Datenbank (Java DB) ca. 17 Sekunden inklusive Einlesen des CSV. Die oben angesprochene Thread-Dump-Analyse zeigte in 7 von 10 Fällen als aktive Zeile den folgenden Thread-Dump *(siehe Listing 3)*.

Die Standard-Strategie zur automatischen Erzeugung einer ID für das Objekt verursachte die meiste Laufzeit. Die Statistiken von Hibernate zeigten fast 72.000 SQL-Statements – mit einer über LAN direkt erreichbaren MySQL-Datenbank lag die Laufzeit in diesem Szenario bei deutlich mehr als vier Minuten. Im LAN ist der Zugriff auf ein physisches Netzwerk-Medium im Spiel, das im Fall der lokal laufenden Datenbank völlig ausgeblendet wurde.

Die Optimierung des ID-Generators durch Erzeugen mehrerer IDs in einem Datenbank-Zugriff sieht im Quelltext wie in *Listing 4* aus und reduziert sowohl die Statement-Anzahl als auch die Laufzeit auf etwa ein Drittel.

Hier wurde der Tabellen-Multi-ID-Generator gewählt, da dieser auf allen Datenbanken funktioniert. Der Sequenz-Generator hat prinzipiell das gleiche Problem vieler Datenbank-Zugriffe, kommt aber insgesamt mit deutlich weniger SQL-Statements aus, da keine getrennte Transaktion erforderlich ist. Noch besser können beispielsweise UUID-Generatoren sein, die gänzlich ohne Datenbank-Zugriff auskommen. Hier ist für die Datenbank zu prüfen, wie diese mit VARCHAR-Spalten als Primärschlüssel umgeht und ob numerische Schlüssel nicht besser indizierbar sind.

Weitere Optimierung: Batch-Sortierung

Führt man nach dieser ersten Optimierung den Code erneut aus und analysiert das Laufzeitverhalten erneut mittels Thread-Dumps, so sieht man einen weiteren Hauptverursacher *(siehe Listing 5)*. Das abwechselnde Schreiben von Städten und Postleitzahlen, die nahezu eine „1:1"-Beziehung aufweisen, nimmt JPA die Möglichkeit, die Zugriffe per JDBC-Batch zu optimieren. Im Thread-Dump sieht man das sowohl an der „NonBatchingBatch"-Implementierung von Hibernate als auch auf der Datenbank anhand der Anzahl von Ausführungen der Insert-Statements.

Werden die Objekte gemäß ihrer Hierarchie persistiert – also zuerst alle Bundesländer, dann alle Städte und zum Schluss die Postleitzahlen – kann mithilfe des Parameters „hibernate.jdbc.batch_size" die Anzahl der Datensätze je Interaktion mit der Datenbank erhöht werden. Durch diese zwei geringfügigen Änderungen lässt sich die Performance des Anwendungsfalls insgesamt um den Faktor 10 auf unter zwei Sekunden verbessern.

Abbildung 2 zeigt, wie sich die Laufzeit in Millisekunden und die Anzahl der Statements getrennt über der Allocation- beziehungsweise Batch-Size verhalten. Die Kurven werden relativ schnell flach, obwohl auf der horizontalen Achse mit jedem Schritt verdoppelt wurde. Der Effekt der einzelnen Veränderung der „AllocationSize" wie auch der „BatchSize" ist für Werte oberhalb von 20 noch messbar, aber nicht mehr signifikant. Ab dieser Größe beginnen andere Faktoren im Netzwerk zu wirken, die diese Optimierungen begrenzen.

Lesen von Daten in der objektrelationalen Welt

Die geschriebenen Daten werden zu einer späteren Zeit wieder aus der Datenbank gelesen. Hier tritt häufig das sogenannte „1+n-Problem" auf: Wird eine Instanz wie eine Stadt gelesen, so wird von JPA die Relation zum zugehörigen Bundesland („State") hergestellt. Allgemein formuliert, werden für alle Objekte, die die eigentliche und

```
…
at java.net.SocketInputStream.socketRead0(Native
Method)
…
at org.hibernate.id.enhanced.TableGenerator$1.
getNextValue(TableGenerator.java:530)
…
```

Listing 3

```
@Id
@TableGenerator(name = "city", initialValue = 1000,
allocationSize = 1000, table = "sequences", pkColumn-
Name = "name", valueColumnName = "value")
@GeneratedValue(generator="city")
long id;
```

Listing 4

```
"main" #1 prio=5 os_prio=0 tid=0x00000000017be800
nid=0x29fc runnable [0x000000000315c000]
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Native
Method)
…
at com.mysql.cj.core.io.FullReadInputStream.
readFully(FullReadInputStream.java:58)
…
at org.hibernate.engine.jdbc.batch.internal.NonBatch-
ingBatch.addToBatch(NonBatchingBatch.java:45)
…
```

Listing 5

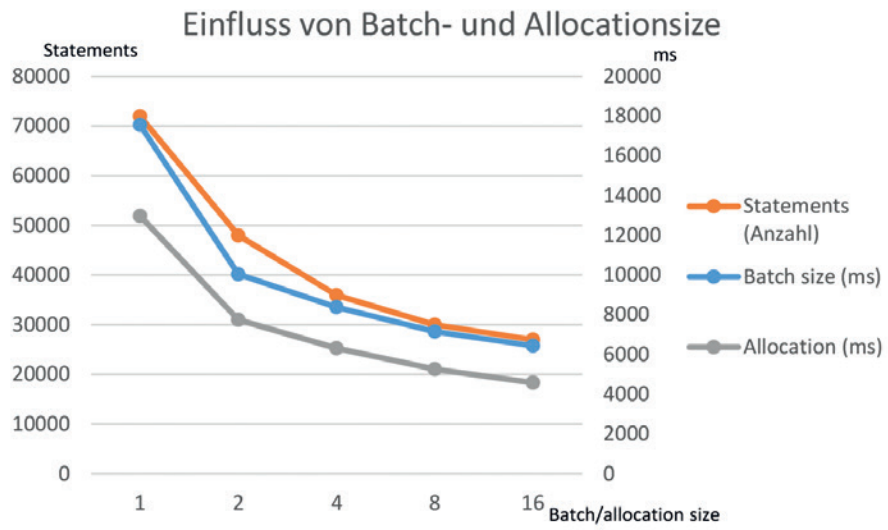


Abbildung 2: Die Laufzeit in Millisekunden

erste („1") Abfrage gefunden hat, alle über „...ToOne"-Relationen verbundenen Objekte nachgeladen. Dazu werden viele einzelne Abfragen auf deren Primärschlüssel ausgeführt – insgesamt „n"-mal, daher „1+n".

Die konkrete Anzahl der Abfragen ist abhängig von der Struktur des Datenmodells, aber auch vom Inhalt der Daten – und damit unberechenbar. Mögliche Lösungswege aus dem „1+n-Problem" sind:

- „Lazy"-markieren der „ManyToOne"-Relation, wenn die Daten meist nicht benötigt werden
- Markieren als „eager", wenn die Daten immer relevant sind
- „Lazy"-markieren, aber in der Abfrage je Anwendungsfall angeben, welche Daten per JOIN geladen werden sollen, dabei sind „NamedQueries" von ihren Möglichkeiten her identisch zum „Criteria"-API
- Caching statischer Daten – bevorzugt bei statischen Daten

„Lazy"-Loading ist für den Persistenz-Provider nur ein Hinweis, „Eager"-Loading kann in Abfragen nicht immer berücksichtigt werden. Daher ist mittels Log-Analyse zu prüfen, wie die ausgeführten Statements aufgebaut sind. Einige JPA-Implementierungen benötigen dafür auch instrumentierten Bytecode, der die Zugriffe auf die Objekte und Attribute verändert sowie Proxy-Objekte möglich macht. Für das „Eager"-Loading führen viele „FETCH JOINs" durch das resultierende kartesische Produkt zu langen und auch breiten „ResultSets" auf der Datenbank. Dies ist sowohl für die Datenbank als auch die Java VM eine unnötig große zu übertragende Datenmenge. Sollten die Standard-Einstellungen nicht ausreichen, kann eine weitere kontrollierte Abfrage helfen, die Daten zu laden.

Prefetch Query

Für einen Warenversand sollen eine Menge Postleitzahlen mit allen bezogenen Daten geladen werden, dazu dient im ersten Versuch die Abfrage „SELECT e FROM PostalCode e WHERE e.code IN (:codes)" für den Zugriff. Funktional ist das völlig ausreichend, führt aber mit fünf wahlfrei ausgewählten Postleitzahlen zu insgesamt sieben SQL-Abfragen. Eines für alle Postleitzahlen und sechs einzelne Statements für die nachzuladenden Städte und Bundesländer. Je nach Daten können es zwischen drei und elf Statements sein, was die Laufzeit unvorhersehbar macht.

Mit dem vorgestellten „Prefetch Query" werden zuerst die über Relationen verbundenen Objekte mit der gleichen Einschränkung wie die eigentlich benötigten Objekte in den Entity-Manager geladen und dort zwischengespeichert. Trifft dieser dann später auf die Relation, so ist ihm das Objekt für den Fremdschlüssel schon bekannt und es ist kein weiteres SQL-Statement notwendig. Der „PersistenceContext" fungiert so als Cache erster Ebene, man muss nur vorab eine weitere kontrollierte Abfrage ausführen.

Zugriffe im Muster von „1+n" sind so nicht mehr notwendig, im folgenden Beispiel wird „1+n" zu „1+1". Die Menge von Postleitzahlen wird immer mit nur zwei Statements geladen, völlig unabhängig davon, ob die Postleitzahlen in der gleichen Stadt in einem Bundesland oder in verschiedenen Städten in verschiedenen Bundesländern liegen *(siehe Listing 6)*.

In der ersten Abfrage werden die benötigten Bundesländer geladen, wobei der EntityManager die Ergebnisobjekte nur in seinem Cache speichert. In der zweiten Query werden bewusst ein JOIN über die „n:1"-Beziehung gemacht und gleichzeitig Städte und Postleitzahlen geladen. Damit duplizieren sich zwar Städte in der Ergebnismenge, das reale Verhältnis der Objekte zueinander ist jedoch nur „1,3:1", weshalb das unproblematisch ist. Drei einzelne Abfragen wären an dieser Stelle auch denkbar. Der gezeigte Code führt im Ergebnis zu einer deterministischen Anzahl von Statements (exakt zwei) und damit zu sehr stabilen Ausführungszeiten.

Problematische „...ToMany"-Relationen

Im Gegensatz zu „...ToOne"- sind „...ToMany"-Beziehungen schon im Standard „lazy", die mit der Entität gelieferte Collection enthält also im ersten Schritt keine Daten und führt erst beim Zugriff auf die Elemente oder die Größe zu einem zusätzlichen SQL-Statement. „Eager"-Loading solcher Relationen führt ohnehin aufgrund der kartesischen Produkte zu großen Ergebnismengen und ist daher nur zu empfehlen, wenn sich die Daten-Konstellation im statistischen Mittel „1:1" annähert; sonst hilft hier eventuell auch wieder ein „Prefetch" der Daten als gesonderte Abfrage.

Weitere Lese-Optimierungen

Sind die Lesezugriffe auf diese Art optimiert, gibt es noch weitere Wege, die Zugriffe zu beschleunigen:

- Das Lesen aus der Datenbank mithilfe der Aufrufe „query.setFirstResult(...)" sowie „query.setMaxResults(...)" ermöglicht die Reduktion der Gesamtmenge und auch das Lesen eines beliebigen Teils eines größeren Ergebnisses. Oft werden nicht alle Daten benötigt und die Datenbank dadurch entlastet.
- Für Lesezugriffe auf Entitäten mit vielen Attributen kann man auch kleiner geschnittene DTO-Objekte mit selektiv ausgewählten Spalten als Inhalt erhalten. JPQL bietet dazu die „constructor_expression" [2] in der „SELECT_CLAUSE". Die geringere Daten-Anforderung an die Datenbank kann beispielsweise mit Indizes auch hinsichtlich I/O optimiert werden.
- Auch die Anbindung von Stored-Procedures hat sich seit JPA 2.1 verbessert und ermöglicht es, komplexe Funktionen in der Datenbank unterzubringen sowie das Ergebnis in wenigen Interaktionen zu lesen, siehe „NamedStoredProcedureQuery" [3].

Updates und Löschungen

Bei Updates hat man weniger Kontrolle über die Interaktion mit der Datenbank, da der Entity-Manager die bekannten Objekte am Ende der Transaktion selbst synchronisiert. Batch-Optimierung wird dabei schon genutzt, sofern der Parameter dafür gesetzt ist.

Bei einer sehr großen Anzahl von Objekten kann das Erkennen veränderter Objekte allerdings auch eine teure Operation sein. Die Standard-Strategie ist oft ein Vergleich mit dem Zustand zum Lade-Zeitpunkt. Bei vielen Objekten ist dies Speicher- und Laufzeit-intensiv und tritt an verschiedenen Stellen einer Transaktion auf, in jedem Fall aber am Ende der Transaktion. Der Code, der hier ausgeführt wird, wird zur Laufzeit von einem entsprechenden Methoden-Interceptor hinzugefügt und ist zur Entwicklungszeit nicht ersichtlich. Daher ist das Erkennen und Zuordnen dieser Laufzeit nicht immer einfach. Mit implementierungsspezifischen Bytecode-Instrumentierungen kann diese Überprüfung der Objekte auf Veränderungen jedoch optimiert werden, etwa zu einer „DirtyFlag"-Strategie. Damit können in der Transaktion geladene Objekte performant aktualisiert werden, oft erhält man allerdings ein verändertes Objekt von der Oberfläche zurück und muss dieses wieder mit der Datenbank zusammenführen.

Merge

Ein Merge vereinfacht die Aufgabe der Aktualisierung von Datenbank-Inhalten, wenn die Objekte außerhalb der Transaktion verändert wurden und anschließend gespeichert werden sollen *(siehe Listing 7)*.

```
SELECT e.city.state
       FROM PostalCode e WHERE e.code IN (:codes)
SELECT e FROM PostalCode e
       JOIN FETCH e.city WHERE e.code IN (:codes)
```

Listing 6

```
City cityFromUI = …
City mergedCity = em.merge(cityFromUI);
```

Listing 7

```
City city = em.getReference(City.class, 4711L);
postalCode.setCity(city);
```

Listing 8

Sofern ein Objekt in der Transaktion noch nicht geladen wurde, führt der Merge-Aufruf ein „SELECT"-Statement auf den Primärschlüssel aus und kopiert alle Werte der übergebenen Instanz in das neu geladene Objekt. Man erhält das zukünftig vom Entity-Manager beachtete Objekt zurück. Weitere Änderungen müssen auf diesem Ergebnis-Objekt durchgeführt werden. Trotzdem führt die Aktualisierung auf diese Art immer zu mehreren Zugriffen, genauso wie im manuellen Fall. Durch das Laden des Datensatzes tritt auch in diesem Szenario das „1+n"-Problem auf, wenn das Objekt geladen werden muss. Im Code der Anwendung zeigt sich ein vergleichbares Problem, wenn eine Relation verändert werden soll; man benötigt eine Instanz des neuen bezogenen Objekts. Ist der technische Schlüssel bekannt, so gibt es hier eine Abkürzung *(siehe Listing 8)*.

Die hier erzeugte Instanz der City ist ein leerer Proxy, der nur den Primär-Schlüssel kennt. Für das Herstellen der Relation genügt dies, weil man sich hier auf die referenzielle Integrität der Datenbank verlassen kann. Greift man aber auf die Attribute dieses Objekts zu, so wird es im Hintergrund nachgeladen. JPQL liefert zusätzlich die direkte Möglichkeit eines Update-Statements, was die maximale Optimierung des Zugriffs zur Aktualisierung ist, wenn man das Objekt aus der Datenbank selbst nicht unbedingt benötigt.

Mit der „CASCADE"-Option kann man sich auf den ersten Blick Code und Zeit sparen, weil der Entity-Manager den Objekt-Graphen durchläuft. Leider bricht das damit einhergehende Wechseln der Entitäten auch eine mögliche „Batch-Sortierung". Zudem besteht die Gefahr, dass man irgendwann nicht mehr überblicken kann, wie groß die Menge der zu betrachtenden Objekte wirklich ist. Schlimmstenfalls werden diese für den „CASCADE" auch nachgeladen. Daher kann hier nur der vorsichtige und eingeschränkte Einsatz angeraten werden.

Caching

Caching ist eine Möglichkeit, Zugriffe auf die Datenbank größtenteils zu vermeiden. Leider ist dies in einem Cluster keine einfache Aufgabe, daher müssen die betroffenen Daten folgende Kriterien erfüllen:

- Sie werden häufig gelesen
- Sie werden selten bis nie verändert

- Veraltete Daten sind für die Anwendung akzeptabel
- Der Speicherverbrauch ist kein Problem

Sind diese Kriterien erfüllt, ist es gut und einfach einsetzbar. Für Daten, die sich häufig verändern, bedeutet Caching allerdings erhöhte Komplexität, die im Projektteam bewältigt werden muss. Daher sollten die bisher genannten Optimierungen ausgeschöpft worden sein, bevor man über Caching-Strategien nachdenkt.

Ab in die Wolke

Cloud-Provider bieten zu guten Bedingungen ganze Plattformen für Anwendungen an, auch Datenbank-Instanzen können mit wenigen Mausklicks bereitgestellt werden. Aus den genannten Gründen der Latenz-Problematik ist eine alleinige Verlagerung der Datenbank in die Cloud keine gute Idee. Die Anwendung sollte ebenfalls umgezogen werden. In den Rechenzentren beziehungsweise Regionen werden Latenz-Zeiten erreicht, die für normale Anwendungsfälle mehr als ausreichend sind. Messungen des Autors ergaben beispielsweise ein bis zwei Millisekunden für einen Datenbank-Roundtrip innerhalb einer Region. Allerdings ist dann meist die Ausfallsicherheit nicht mehr gegeben.

Das zweite Datacenter sollte man betrieblich getrennt, aber trotzdem in der Nähe wählen. Da Licht in Glas nur noch 200.000 km/s schnell ist, benötigt ein Ping-Roundtrip zwischen Amerika und Europa mindestens 80 Millisekunden. Der zu Anfang auf unter zwei Sekunden Laufzeit optimierte Anwendungsfall läuft in einer Region mit ein bis zwei Millisekunden Latenz etwa sechzehn Sekunden. Mit rund hundert Millisekunden Latenz bei der Kommunikation zwischen Amerika und Europa wurden bis zu 44 Minuten Laufzeit gemessen.

Ohne Optimierung sind die Laufzeiten in jedem dieser Infrastruktur-Szenarien inakzeptabel hoch. Die Begründung dafür liegt darin, dass die Latenz nicht nur eine einmalige Auswirkung auf die Netzwerk-Performance hat, sondern auch die Bandbreite der Übertragung mindert. Vor allem im Kontext der Cloud gibt man die Verantwortung, aber auch die Kontrolle über die Infrastruktur größtenteils ab. Daher ist darauf zu achten, dass das Code-Design den Anforderungen und Randbedingungen der Infrastruktur entspricht.

Fazit

JPA ist für Java-Anwendungen ein essenzielles Framework und es funktioniert auch größtenteils erwartungsgemäß, problemlos und äußerst elegant. Viele Statements für eine Transaktion müssen kein Problem darstellen, wenn deren Anzahl stabil bleibt. Kritische Transaktionen und Anwendungen, die viele Statements erzeugen, kann man im Nachgang mit wenig Aufwand optimieren.

Zusätzlicher Code mit expliziten Anweisungen ermöglicht dedizierte Verbesserungen an kritischen Stellen und ist daher wartungsfreundlicher als generische Lösungen, die kleine Veränderungen schwierig umsetzbar machen. Jegliches Tuning sollte jedoch nach der funktionalen Umsetzung erfolgen, da schon kleine fachliche Änderungen das Szenario der Zugriffe grundlegend verändern können.

Den Entwicklern sollten die konkrete Infrastruktur und die notwendigen Randbedingungen immer bewusst sein, damit die richtigen Code-Bausteine optimiert werden können. Eine einfache Maßnahme wie die Aktivierung der Konsolenausgabe der laufenden SQL-Befehle in der Entwicklungsumgebung kann hier schon Wunder bewirken.

Kenntnisse über die tatsächlichen Verhältnisse der Relationen helfen bei der Optimierung der JOINs und damit der Ergebnismengen. Nicht ganz am Ende steht natürlich die Optimierung der Datenbank für die optimierten Zugriffe, damit diese nicht doch an deren Performance scheitern.

Referenzen

[1] *https://de.wikipedia.org/wiki/Java_Persistence_API*
[2] *http://docs.oracle.com/html/E13946_04/ejb3_langref.html#ejb3_langref_select_clause*
[3] *https://docs.oracle.com/javaee/7/api/javax/persistence/NamedStoredProcedureQuery.html*

Thomas Bröll
thomas.broell@trivadis.com

Thomas Bröll ist Principal Consultant für das IT-Dienstleistungsunternehmen Trivadis am Standort Stuttgart. Er arbeitet seit mehr als zwanzig Jahren als Software-Entwickler, -Berater und -Architekt mit Java. Sein Fokus liegt dabei auf der Konzeption und Implementierung von Web-Applikationen und Business-Anwendungen auf Basis von JEE. Bei der Arbeit in Kundenprojekten im Mittelstand, aber auch in großen Organisationen sammelte er Erfahrungen mit dem Einsatz von Java Enterprise, zugehörigen Frameworks und Architekturen. Er ist darüber hinaus für die Trivadis als Referent und Trainer tätig.

Praxishandbuch BPMN

gelesen von Bennet Schulz

Sie haben es wieder getan! Acht Jahre nach der ersten Auflage ist das Praxishandbuch BPMN von Jakob Freund und Bernd Rücker nun bereits in der fünften Auflage erschienen.

Seit der vierten Auflage hat sich einiges verändert, schreiben die Autoren selbst in ihrem Vorwort. Das nicht ohne Grund, denn neben der Aufnahme der BPMN als ISO-Standard sind zwei weitere Standards namens „Case Management Model and Notation" (CMMN) sowie „Decision Model and Notation" (DMN) hinzugekommen und direkt in die neue Auflage eingearbeitet worden.

Begonnen wird in dem Buch allerdings nicht mit den Notationen selbst, sondern mit einem einführenden Kapitel in Business Process Management (BPM), das neben dem Einsatzzweck BPM auch die Brücke zu den neu hinzugekommenen DMN und CMMN schlägt und das Zusammenspiel der drei Standards anhand eines zusammenhängenden Beispiels erläutert.

Darauffolgend stellen die Autoren ein Methoden-Framework für den erfolgreichen Praxiseinsatz vor und räumen einige Missverständnisse beim Vorgehen in BPMN-Projekten aus dem Weg, ehe es im zweiten Kapitel schon direkt an die Notationselemente der BPMN, CMMN und DMN geht.

Über rund 150 Seiten gehen sie sehr detailliert auf die einzelnen Elemente der drei Notationen ein. Besonders positiv fallen dabei die zahlreichen Hinweise und Praxistipps ins Auge, die das Buch zu einem echt wertvollen Nachschlagewerk für den Praxiseinsatz machen. Abgerundet wird das Buch durch Kapitel zur Automatisierung über die Einführung der BPMN im Unternehmen.

Fazit: Der Titel hält definitiv, was er verspricht. Bei dem Buch handelt es sich um ein angenehm zu lesendes Arbeitsbuch für den Praxiseinsatz. Es widmet sich neben der BPMN auch noch der CMMN, DMN sowie Vorgehensmodellen und der Einführung im Unternehmen.

Bennet Schulz
mail@bennet-schulz.de

Autor: Jakob Freund, Bernd Rücker

Titel: Praxishandbuch BPMN

Verlag: Hanser München

Umfang: 301 Seiten

Preis: 38 Euro

eBook (downloadbar) im Preis enthalten

ISBN: 978-3446450547

„Eine basisdemokratische Entwicklung der Java-Plattform ermöglichen …“

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

Du hast kürzlich die JUG Thüringen gegründet. Was war deine Motivation?

Benjamin Nothdurft: Ich bin mittlerweile seit mehr als zwölf Jahren mit Java unterwegs, also mit der Sprache und deren Ökosystem vertraut. Einige Jahre davon habe ich mich damit begnügt, Leser des Java Magazin zu sein und unzählige Events der Java User Groups zu besuchen, wie in Stuttgart, Karlsruhe, München, Nürnberg, Düsseldorf, Dresden, Berlin und Hamburg, oder die JavaLand im Phantasialand bei Köln. Nachdem ich im Jahr 2014 nach Thüringen gezogen bin, habe ich zwar das Glück, projektbezogen nach Hamburg pendeln zu können, verbringe jedoch immer noch einen Großteil meiner Zeit in Jena. Durch die rund fünfzehn Talks und Workshops auf diversen IT-Konferenzen im letzten Jahr reifte zunehmend der Gedanke, auch mehr fachspezifisches Know-how in meine neue Wahlheimat zu tragen und den themenfokussierten Austausch im Java-Umfeld anzuregen.

Wie ist die JUG Thüringen organisiert?

Benjamin Nothdurft: Da ich als Software-Engineering-Dozent an der Ernst-Abbe-Hochschule Jena (EAH) bereits Java-Grundlagen unterrichte sowie die Softwerkskammer Jena, die größte Technologie-unabhängige Software-Engineering-Fachgruppe Thüringens, mit Programmierabenden und Workshop-basierten Wochenend-Veranstaltungen sowie dem Schwerpunkt auf IT-Kooperationen und fachlichem Austausch von und mit lokalen Wirtschaftsunternehmen vor eineinhalb Jahren gegründet habe, besteht bereits ein gutes IT-Netzwerk, auf das wir zurückgreifen können. Mit Prof. Dr. Andrej Werner, Professor für E-Commerce und Wirtschaftsingenieurwesen sowie Business Information Systems an der EAH, haben wir unseren ersten Schirmherren für den Standort Jena gewinnen können. Der Standort Weimar wird tatkräftig organisiert durch unseren Sprecher Jonas Hecht, Senior Engineer und Java-Architekt bei codecentric und ebenfalls IT-Dozent an der Bauhaus-Universität Weimar. Auch hier können wir auf Unterstützung aus dem akademischen Umfeld bauen. Prof. Dr. Norbert Siegmund, unser Schirmherr in Weimar, begleitet aktuell als einer der jüngsten und vielfach ausgezeichneten Professoren den Bereich „Intelligente Softwaresysteme“, wobei hier auch Java-Technologien, IoT und Deep Learning eine Rolle spielen. An den Standorten Erfurt und Gera stehen wir bereits in Kontakt mit mehreren anderen Professoren und Hochschulen, aber auch in Jena werden wir schon bald zudem auf die Friedrich-Schiller-Universität Jena zählen können.

Was sind die Ziele der JUG Thüringen?

Benjamin Nothdurft: Wir möchten das lokale IT-Netzwerk stärken und Expertenzirkel aufbauen sowie die Attraktivität des Landes Thüringens für Java-Entwickler von außerhalb erhöhen. Hinzu kommt die Verzahnung von Wirtschaft und Wissenschaft durch die vielfältigen Kooperationen im akademischen Sektor und den Wissensaustausch auf neutralem Boden.

Wie viele Veranstaltungen gibt es pro Jahr?

Benjamin Nothdurft: Aktuell sind zwölf Veranstaltungen für dieses Kalenderjahr in Planung. Da wir schon jetzt bereits mehr als fünfzehn Zusagen von Top-Speakern in unserem Backlog haben und zudem bis zu fünf verschiedene akademische Einrichtungen bespielen könnten, ist die Möglichkeit zur standortunabhängigen Skalierung und durchaus das Potenzial zu deutlich mehr Vortragsabenden gegeben. Wir sollten allerdings zunächst realistisch bleiben und auch auf das Feedback der ersten Besucher eingehen, die vornehmlich Entwickler und Projektmanager aus der Wirtschaft sowie Studierende im IT-Themenspektrum sein werden.

Was bedeutet Java für euch?

Benjamin Nothdurft: Für Jonas und mich ist Java seit vielen Jahren unser tägliches Brot – wir machen Beratung, Coaching und Implementierung. Wir besuchen Konferenzen, halten Talks, veröffentlichen Artikel und schreiben Blog-Posts. Ohne Java, dessen vielfältige

Plattform und dem wunderbaren Tooling rundherum können wir uns den Entwickleralltag nicht mehr vorstellen; kaum eine andere Enterprise-Technologie wusste in den letzten Jahren so stark zu überzeugen. Auch wenn die .Net-Welt in einigen Features früher dran war, so steckt dahinter doch eher der Closed-Source-Gedanke und die kommerziellen Interessen stehen im Gegensatz zur Open-Source-Philosophie von Java im Vordergrund. Auch diese frühzeitig verankerten Gedanken, die einen ungehinderten Zugang und Wissensaustausch auch für benachteiligte Personengruppen ermöglichen, tragen maßgeblich zum Erfolg, der hohen Popularität, weiten Verbreitung und auch unserem fast uneingeschränkten Commitment zu diesem Technologie-Stack bei.

Welchen Stellenwert besitzt die Java-Community für euch?

Benjamin Nothdurft: Diese Technologie-spezifische Community ist weltweit einzigartig. Kein anderes fachliches IT-Netzwerk ist so vielfältig und bietet so viel Rückhalt und globalen Erfahrungsaustausch. Nicht nur für angehende Entwickler, auch für Java-Experten und Enterprise-Architekten sowie emeritierte alte Hasen bietet die Community eine ausgezeichnete Anlaufstelle – begleitend zur beruflichen und akademischen Ausbildung, beim Berufseinstieg und über die gesamte IT-Laufbahn hinweg und manchmal auch noch im Ruhestand.

Wie sollte sich Java weiterentwickeln?

Benjamin Nothdurft: Generell halten wir den Java Community Process (JCP) für eine geeignete Möglichkeit, eine basisdemokratische Entwicklung der Java-Plattform zu ermöglichen.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?

Benjamin Nothdurft: Oracle sollte wieder aktiver als Mitglied am JCP mitwirken und auch darauf hinwirken, Standards wie Java EE in Abstimmung mit der Community offener voranzutreiben, sodass auch Entwicklungspotenziale genutzt werden können und dadurch die Lücke zu alternativen De-facto-Standards wie Spring minimiert sind. Auch sollten Verzögerungen wie beim JDK 9 frühzeitiger adressiert und kommuniziert werden. Die Einführung neuer Major-Versionen sollte mit öffentlichkeitswirksamer Einbindung der Community durch geeignete Maßnahmen realisiert werden, wie kleine parallele Launch-Partys rund um den Globus. Dadurch lassen sich eine hohe Bekanntheit der neuen Features und damit auch eine schnelle Marktdurchdringung erzielen und nachfolgende Innovation breitflächiger ins Auge fassen. Auch helfen solche Aktionen dabei, das angestaubte Image und die Fürsorge von Oracle für Java wieder aufzupolieren.

Wie sollte sich die Community gegenüber Oracle verhalten?

Benjamin Nothdurft: Die Community sollte sich im Gegenzug so stark wie möglich am JCP einbringen und auch mit gewichtigen Interessenverbänden wie dem iJUG oder der LJC im JCP Executive Committee (EC) per se als neutrale Instanzen vertreten sein – frei von den Interessen einzelner Global-Player-Unternehmen, die teils das Bild zu ihren Gunsten prägen.

Zur Person:
Benjamin Nothdurft

Benjamin Nothdurft arbeitet als Software-Entwickler mit Java-Enterprise-Microservices an der E-Commerce-Plattform der nächsten Generation von ePages an den Standorten Jena und Hamburg. Zudem ist er als Dozent für Software Engineering in drei IT-Studiengängen an der Ernst-Abbe-Hochschule tätig. Er ist Gründer und Sprecher der größten Fachgruppe für Software Engineering in Thüringen, der Softwerkskammer Jena. In seiner Freizeit unterstützt, moderiert und/oder organisiert er Konferenzen wie die microXchg, Coderetreats mit TDD-Vorträgen und Katas/Praxisübungen, Docker-Partys und Docker-Mentorenwochen oder Open-Data-Hackathons. Nebenbei hält er Fachvorträge auf Konferenzen zu Microservices, NoSQL-Technologien, Continuous Delivery und Infrastructure as Code oder erklärt in zahlreichen Workshops disruptive Technologien wie Docker oder auch Serverless Functions und treibt deren Weiterentwicklung voran, indem er diversen Software-Systemen aktuell das Sprechen beibringt.

iJUG Verbund

Alle Mitglieder auf einen Blick

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

www.ijug.eu

Impressum

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

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

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

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

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

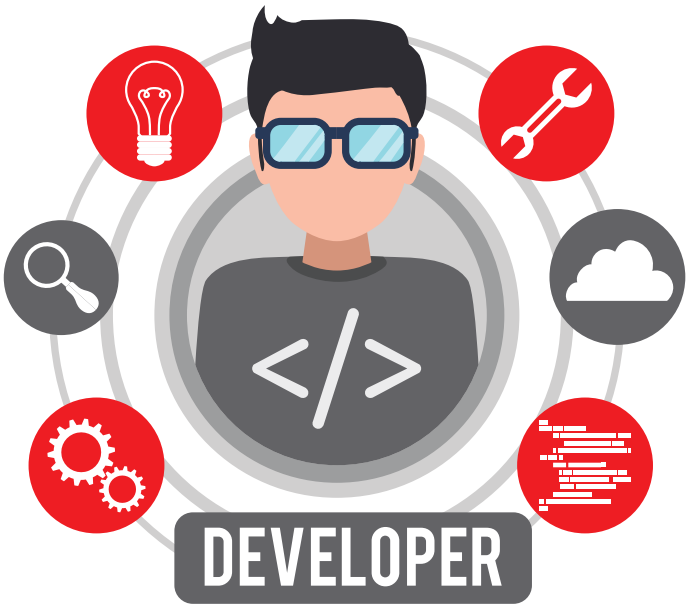
Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titelmotiv: © Elena Rankova/
DOAG Dienstleistungen GmbH
Foto S. 8: © dolgachov/123RF
Foto S. 12-13: © DOAG e.V.
Foto S. 14-15: © Uwe Sauerbrei
Grafik S. 16: © ojogabonitoo/123RF
Grafik S. 29+34: © kchung/123RF
Grafik S. 39: © Zorak1103/Wikipedia
Grafik S. 43: © destinacigdem/123RF
Grafik S. 48: © sentavio/123RF
Grafik S. 53: © THEERAVAT BOONNUANG/123RF
Grafik S. 57: © madpixblue/123RF

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

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

Inserentenverzeichnis



DOAG
Stellenausschreibung

Erfahrungs- und Wissensaustausch sind die Eckpfeiler unserer Philosophie. Um den Austausch zwischen Oracle-Anwendern zu fördern, hat die DOAG eine attraktive, vielschichtige Informationsplattform aufgebaut, die ihren Platz sowohl bei Veranstaltungen, als auch in Print- und Online-Medien einnimmt. Inzwischen begleitet die DOAG mehr als 100 Fachveranstaltungen pro Jahr mit bis zu 2000 Teilnehmern. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. ist in Deutschland die einzige Interessenvertretung der Anwender von Oracle-Produkten. Die DOAG Dienstleistungen GmbH ist für das operative Geschäft des Vereins verantwortlich und führt die Geschäftsstelle.

Zur Verstärkung unseres Teams suchen wir in Voll- oder Teilzeit eine/n

Web-Entwickler (m/w)

Sie sind in einem kleinen Team verantwortlich für die Weiterentwicklung unserer IT-Systeme. Daneben unterstützen Sie uns auf unseren größeren Veranstaltungen in ganz Deutschland vor Ort. Sie verfügen über einen Hoch- oder Fachschulabschluss oder vergleichbare Qualifikationen und haben Berufserfahrung in vergleichbaren Positionen. Sie haben gute Kenntnisse im Bereich Softwaremanagement. Die Trends bei neuen Medien haben Sie im Blick. Zusätzlich bringen Sie Fachkenntnisse aktueller Webtechnologien mit. Ihre Arbeitsweise ist geprägt von strukturierten und methodischen Vorgehen. Sie verfügen über Vermittlungskompetenz zwischen den Fachanforderungen und der EDV-technischen Umsetzung. Sie bringen ausgeprägte Teamfähigkeit und Kommunikationskompetenz mit und arbeiten mit hoher Kundenorientierung.

Ihre Qualifikation:
- technisches Studium oder vergleichbare Abschlüsse
- sehr gute Kenntnisse Typo3, PHP, HTML, CSS, JavaScript, jQuery
- mehrjährige Berufserfahrung
- englische Sprachkenntnisse
- Spaß und Freude an der Arbeit in kleinen Teams

Wir bieten Ihnen einen vielseitigen, zukunftssicheren Arbeitsplatz in einem kreativen jungen Team, eine attraktive Vergütung und einen modernen, ansprechenden Arbeitsplatz in der Hauptstadt Berlin.

Ihre Bewerbung senden Sie bitte per E-Mail unter Angabe Ihres möglichen Eintrittstermins und Ihrer Gehaltsvorstellungen an bewerbung@doag.org. Für Rückfragen steht Ihnen der IT-Verantwortliche Herr Jürgen Pittorf (0175 6649499) gerne zur Verfügung.

JUG SAXONY DAY

29.09.

KEYNOTE
Softwarearchitektur für alle!?
Stefan Zörner (embarc)

PROGRAMM
jug-saxony-day.org

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

TICKETS
Early Bird bis 15. Juli sichern

Eine Veranstaltung des **JUG Saxony e.V.**

Kontakt
team@jugsaxony.org
jugsaxony.org

Mit freundlicher Unterstützung durch

iJUG Verbund

Javaaktuell