Java aktuell

iJUG Verbund
www.ijug.eu

Polymer
Komponentenbasierte Software-Entwicklung

SonarQube
Statische Codeanalyse

Rancher
Docker Application Stacks

Java ist voll im Trend

Ausgabe 04/2017
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 04

Jetzt **Ticket** sichern

Early-Bird bis
28. September

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

mit Java- & Middleware-Stream

PROGRAMM ONLINE
mit rund 450 Vorträgen

2017.doag.org

Eventpartner: AOUG Austrian Oracle User Group · SOUG Swiss Oracle User Group · iJUG Verbund · ORACLE

Nach der JavaLand ist vor der JavaLand

Die JavaLand schloss am 29. März 2017 ihre Pforten. Das außergewöhnliche Konferenzformat kam super an: Zur vierten Auflage der Veranstaltung im Phantasialand in Brühl waren rund 1.650 Entwickler und Programmierer anwesend, um sich in mehr als hundert Vorträgen und vielen Community-Aktivitäten über die neuesten Trends der Szene zu informieren. Die Stimmung war super, nicht zuletzt aufgrund des herrlichen Wetters.

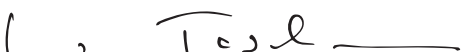
Die Mehrzahl der Artikel in dieser Ausgabe ist aus Vorträgen auf der JavaLand entstanden. Diejenigen, die dabei waren, können somit das Thema nochmals Revue passieren lassen. Alle anderen sehen, was sie auf der JavaLand alles erwartet.

Der Interessenverbund der Java User Groups e.V. (iJUG) hat im Nachgang zur JavaLand 2017 auf seiner Mitgliederversammlung ebenfalls bereits die Weichen für die JavaLand 2018 gestellt. So soll es wieder ein umfangreiches Community-Programm geben, um die Teilnehmer beim Einstieg in die Praxis zu unterstützen. Ein weiteres wichtiges Thema war die Neugestaltung der Zeitschrift Java aktuell, die in diesem Jahr mit fünf Ausgaben auf den Markt kommt.

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. Das bewährte Konzept bleibt im Wesentlichen gleich. Das Organisationsteam bemüht sich noch darum, einen weiteren Vortragsraum zur Verfügung stellen zu können. Auch die neuen Ziele stehen bereits fest: 40 Aussteller und mehr als 1.800 Teilnehmer. Hoffentlich wird Oracle im kommenden Jahr wieder auf der Ausstellung präsent sein, um den Kontakt zur Java-Community zu nutzen. Das hatten viele Teilnehmer in diesem Jahr vermisst.

Ich würde mich freuen, wenn die nächste JavaLand Ihre Erwartungen erfüllt und Sie neue Anregungen für Ihre Projekte mitnehmen können.

Ihr

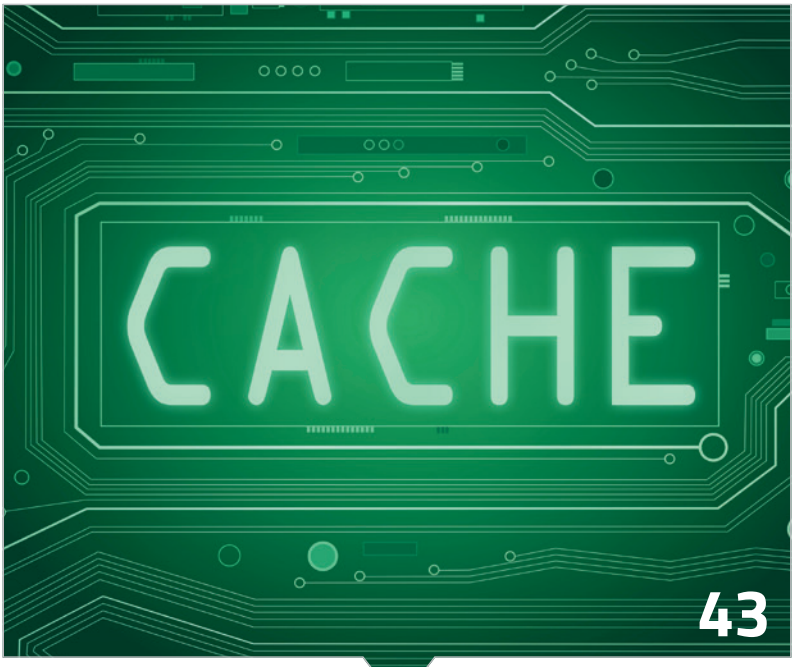


Wolfgang Taschner
Chefredakteur Java aktuell

Inhalt

Eine Container-Management-Plattform, um das Arbeiten mit Docker-Containern so einfach wie möglich zu gestalten

Das Kombinator-Entwurfsmuster kombiniert kleine fachliche Funktionen situationsgerecht zu komplexer Fachlogik



43

Bei jedem Remote-Aufruf stellt sich die Frage, ob Caching helfen kann, lange Wartezeiten zu vermeiden

57

Web Components ist ein Standard, der komponentenbasierte Software-Entwicklung für das Web ermöglicht

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

27. März 2017

JavaLand startet mit den ganz Kleinen

Das JavaLand 2017 öffnet morgen seine Tore, und es ist jetzt schon klar, dass nach 1.220 Besuchern im Vorjahr nun mehr als 1.600 kommen werden. Heute sind allerdings erstmal die Entwicklerinnen und Entwickler von morgen bei „JavaLand4Kids". Dreißig Viertkläss-ler aus St. Augustin machen ihre ersten Schritte in der Programmierung mit Lego Mindstorms und „Jumping Sumo"-Robotern. Der Nao-Roboter kann leider dieses Jahr nicht programmiert werden, aber es gibt auch noch schöne Bastelarbeiten mit Kupferfolie und LEDs zur Abwechslung. Und die Kinder haben eine Menge Spaß.

https://www.javaland.eu/de/javaland-2017/javaland4kids

28. März 2017

JavaLand: Java EE 8 und andere

Das JavaLand 2017 startet mit interessanten Keynotes und gegen Mittag mit einem Status-Panel zu Java EE 8, an dem viele Mitglieder und Leiter von EE 8 Expert Groups teilnehmen. Unter anderem ist David Delabassee dabei, der eine Schlüsselrolle in der Kommunikation zwischen Oracle und der Community eingenommen hat (ich versuche, das Wort „Evangelist" zu vermeiden). Besonders interessant sind die Dinge, die offensichtlich beim Stoppen und Neustarten der Aktivitäten aus Zeitgründen fallen gelassen oder schlichtweg vergessen wurden, so auch Anforderungen von MVC gegenüber JAX-RS (die eigentlich schon spezifiziert und implementiert waren und nur hätten übernommen werden müssen), und die erst in der Diskussion der Experten untereinander im JavaLand ans Tageslicht kommen. Eigentlich hätte ich die Diskussion gerne noch in Richtung Microprofile und die Zukunft von „Umbrella-Spezifikationen" wie EE 9, 10 etc. gelenkt – aber das laufende Release war dann doch wichtiger. Vielleicht sollte man die JavaLand mehrmals im Jahr veranstalten, um die Kommunikation im JCP zu verbessern …

https://programm.javaland.eu/2017/talk.html#talk?talkId=537951

29. März 2017

JavaLand 2917: Neuer Besucherrekord

Tag zwei der JavaLand bringt unter anderem eine „Release Party" für JavaServer Faces 2.3. Specification Lead Ed Burns hat extra eine Torte mit JSF-Logo backen lassen, um den Push der finalen Version in die Maven Repositories unter anderem mit weiteren Mitgliedern der Expertengruppe zu feiern. Auch MVC spielt eine große Rolle in der „Early Adopters' Area". Nebenan stellt die JUG Goldstadt ihren selbst gebastelten Flipper auf Raspberry-Pi-Basis vor, und mitten im Saal sind wieder jede Menge Roboter, neue VR-Projekte etc. im „Innovation Lab" zu bestaunen. Das normale Vortragsprogramm beinhaltet viele Vorträge zu Microservices und Docker, gerne auch in Kombination – der Hype dürfte seinen Höhepunkt erreicht haben. Am Ende des Tages schließt das JavaLand seine Pforten mit einem neuen Besucherrekord: 1.655 Besucher – also mehr als eine Verdopplung gegenüber der ersten JavaLand 2014.

https://www.doag.org/de/home/news/javaland-2017-java-konferenz-im-phantasialand-verdoppelt-teilnehmerzahl/detail

22. April 2017

Entwickler äußern Bedenken gegenüber Jigsaw

Eine Reihe von Entwicklern und Managern von Red Hat, Apache und anderen hat sich zusammengeschlossen, um schwerwiegende Probleme aufzuzeigen, die sie in der Spezifikation von Jigsaw sehen. Die Spezifikation sei geeignet, um die Java-Plattform selbst zu modularisieren, nicht jedoch für die darauf aufbauende Anwendungsentwicklung. Insbesondere werde es entgegen den Zielen die Entwicklung von großen Applikationen und Bibliotheken erschweren und zu weniger Robustheit führen. Auch das Ziel, die Modularisierung für Java EE 9 zu nutzen, sei aufgrund von Einschränkungen in Jigsaw fast unmöglich, da es beispielsweise zu Kompatibilitäts- und Interoperabilitätsproblemen führen würde. Die Gruppe befürworte eine kleine Verschiebung von SE 9, um die größten Probleme zu beheben. Nachdem es im März schon mal Gerüchte gab, dass sich SE 9 weiter verzögert, sieht es aber momentan nicht so aus, als ob Mark Reinhold oder jemand anders bei Oracle noch die Notbremse ziehen würde. Außerdem gibt es andere Kommentare aus der Community dahingehend, dass der Beitrag vielleicht auch nicht zufällig von Organisationen kommt, die eigene Modul-Systeme entwickelt haben, mit denen Jigsaw nun konkurriert – und dass bei einem so massiven Projekt immer erstmal Probleme auftreten. Vermutlich heißt es daher: „Augen zu und durch."

http://red.ht/2oE7l8F

3. Mai 2017

„java.net"-Migration abgeschlossen

Die Abschaltung der alten „Kenai"-Server hinter „java.net" ist vollzogen und die Repositories wurden zu GitHub migriert. Alles zum Thema „Java EE" lässt sich jetzt dort finden oder man springt direkt in die GitHub-eigene Repository-Übersicht unter *„https://github.com/javaee[/javaee-spec]"*. Zu sonstigen Projekten stehen die Informationen unter *„https://javaee.github.io/other-migrated-projects.html"*. Offensichtlich haben nicht alle Informationen die Migration überlebt. MVC etwa ist nicht unter den „other migrated projects" aufgeführt – ein Schelm, wer Böses dabei denkt … Spaß beiseite, das Repository ist natürlich migriert worden, es befindet sich unter *„https://github.com/mvc-spec"* – auch wenn es in der Übersicht fehlt. „java.

net" ist damit aber nicht komplett verschwunden, das OpenJDK und seine Mercurial-Repositories leben beispielsweise auf ihrer eigenen Hardware weiter unter „openjdk.java.net".
https://javaee.github.io

8. Mai 2017

EC-Nachwahlen zu Ratified Seats

Bedingt durch den bereits angekündigten Rückzug von Ericsson und TOTVS aus dem JCP Executive Committee wurden jetzt zwei neue Vertreter von Oracle nominiert, um einen „ratified seat" einzunehmen: JetBrains (unter anderem bekannt durch die IDE IntelliJ IDEA) und der Chip-Hersteller ARM, allen Raspberry-Pi-Nutzern ebenfalls bestens bekannt. Beide wurden mit jeweils nur wenigen Gegenstimmen und mehr als 90 Prozent Ja-Stimmen von den JCP-Mitgliedern angenommen. Wegen der Staffelung von Wahlperioden im Zuge der Zusammenführung der JCP Executive Committee zu einem einzigen für alle Plattformen wird ARM seinen Sitz zunächst bis Herbst 2017 innehaben, während die Wahlperiode für JetBrains bis Herbst 2018 läuft.
https://jcp.org/en/participation/committee

8. Mai 2017

Jigsaw: Public Draft abgelehnt!

Die Spezifikation für „Project Jigsaw" (der offizielle Name im Java Community Process lautet „Java Platform Module System") ist vom Executive Committee des JCP abgelehnt worden. Die Wächter über die Standardisierung von Java waren mehrheitlich der Auffassung, dass die aktuell diskutierten Unzulänglichkeiten der Spezifikation zu groß sind. Das heißt jedoch nicht unbedingt, dass das Abenteuer „Modularisierung" beendet ist oder das JDK-9-Release, in dem Jigsaw das bei weitem umfangreichste Feature ist, nicht mehr im Juli kommen wird. Chef-Architekt Mark Reinhold und sein Team haben zunächst dreißig Tage Zeit, die Spezifikation zu überarbeiten und dem Executive Committee erneut zur Abstimmung vorzulegen. Die meisten der Nein-Sager haben auch ausdrücklich betont, dass die Spezifikation jüngst deutliche Fortschritte gemacht hat und einfach noch ein bisschen mehr Zeit braucht, damit ein Konsens in den noch offenen Punkten gefunden wird. Wenn das EC beim zweiten Mal mehrheitlich zustimmt, könnte es sogar noch mit dem Zeitplan für das Release im Juli klappen. Momentan wird daher mit Hochdruck an den offenen Punkten gearbeitet, etwa an der Berechnung von „Automatic Module Names", wie auf der „Issue Tracking"-Seite unter *„http://openjdk.java.net/projects/jigsaw/spec/issues"* zu sehen ist. Neben den technischen Diskussionen gibt es allerdings auch eine politische Ebene: Oracle und einige der unabhängigen Beobachter vermuten hinter der Ablehnung speziell bei Red Hat und IBM eigene Geschäftsinteressen. Das macht die Angelegenheit natürlich deutlich komplizierter. Aber letztlich ist Jigsaw auf ein Ja-Votum der beiden nicht unbedingt angewiesen. Das erste Votum ging 10:13 aus. Da der JCP keinen Konsens einfordert, würde es schon reichen, zwei andere Vertreter der dreizehn zu überzeugen, etwa die London Java Community, Twitter oder die individuellen Vertreter Werner Keil und Ivar Grimstad – wobei das Vertrauen in Jigsaw und das JDK 9 bei einem knappen Ja wohl leiden würde.
http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2017-May/000695.html

https://www.heise.de/developer/meldung/Modulsystem-fuer-Java-steht-auf-der-Kippe-3706597.html

8. Mai 2017

CDI 2.0 einstimmig angenommen

CDI 2.0 ist fertig. Der JSR wurde bereits vor einer Woche vom Executive Committee des JCP einstimmig angenommen. Jetzt ist auch die finalisierte Spezifikation auf „jcp.org" hochgeladen.
http://www.cdi-spec.org/news

16. Mai 2017

Java EE 8: Fortschritt im April

Noch mehr Neues zu Java: Eine Reihe von JSRs für EE 8 haben den nächsten Meilenstein erreicht, JSF 2.3 ist zum Beispiel durch (Doku finalisiert, Binaries in Maven Repositories gepusht), JSON-P 1.1 so gut wie (die Spezifikation wurde grundsätzlich vom EC angenommen, es fehlen nur noch der letzte Schliff und das „final approval") und JAX-RS ist ins „Public Review" eingestiegen. Java EE Security wurde aufgrund entsprechenden Feedbacks aus der Community jetzt ins EE 8 Web Profile aufgenommen. Das inzwischen als „Community JSR" laufende MVC 1.0 hat nun neben Ivar Grimstad mit Christian Kaltepoth einen Co-Spec-Lead (Christian war auch bereits vorher Mitglied der Expert Group).
https://blogs.oracle.com/theaquarium/java-ee-8-april-recap

30. Mai 2017

Jigsaw-Expertengruppe arbeitet an Lösungen

In den letzten Wochen hat Mark Reinhold eine Reihe von Vorschlägen über die OpenJDK-Mailingliste *(siehe „http://mail.openjdk.java.net")* verbreitet, um die Spezifikation für das neue Plattform-Modulsystem doch noch durch das JCP Executive Committee zu bringen. Auch die Expertengruppe hat sich in der zweiten Monatshälfte gleich drei Mal per Videokonferenz getroffen, um Lösungsvorschläge zu den offenen Punkten zu besprechen. Auf der „Issue List" sind die Punkte entsprechend weitestgehend abgehakt. So wird aus dem in der letzten Tagebuch-Ausgabe erwähnten „--permit-illegal-access" mit einer potenziellen Flut von Warnungen im Log jetzt ein parametrisiertes „--illegal-access" (permit, warn, debug, deny). Einziger Punkt, zu dem kein Konsens erzielt werden konnte, ist die Behandlung von neuen sogenannten „restricted keywords" wie „requires" und „transitive", die für Compiler aufgrund der geforderten Rückwärtskompatibilität schwierig zu behandeln sind. Sie sollen nur im Kontext einer Modul-Dekla-

ration als „keywords" behandelt werden (also genauso wie „if", „class" etc.), ansonsten aber als normale „identifier" („String", „someParameter" etc.). Die Entscheidung hier, offensichtlich nicht mitgetragen vom Eclipse-Vertreter: Die Spezifikation für bessere Code-Lesbarkeit so lassen, wie sie ist, und stattdessen die Compiler-Bauer schwitzen lassen. Ein paar andere Punkte wurden als nicht dringend zu lösen angesehen und stattdessen in eine Wunschliste für ein Folge-Release gepackt, beispielsweise „Multi Module JARs" und „Cyclic Dependencies".
http://openjdk.java.net/projects/jigsaw/spec

7. Juni 2017

Java EE 8: Fortschritt im Mai

Was hat der Mai sonst noch für EE 8 gebracht? Java EE 8 selbst (die Mantelspezifikation, englisch „Umbrella"), Servlet 4.0, JAX-RS 2.1 und Bean Validation haben alle den „Public Review Ballot"-Meilenstein erreicht. Die Mitglieder des Executive Committee stimmen zurzeit ab – innerhalb einer 14-tägigen Frist. Auch die Referenz-Implementierung (GlassFish 5) macht wieder Fortschritte: Laut David Delabassee wurde viel Zeit für den Umzug von „java.net" und Verbesserungen der Build Pipeline verwendet. Nachdem dies abgeschlossen ist, erwartet er eine deutliche Beschleunigung in den noch anstehenden Arbeiten. Das im November gesetzte Ziel „Juli 2017" ist in Reichweite.
http://download.oracle.com/glassfish/5.0/promoted

8. Juni 2017

Java SE 9 Release wird um acht Wochen verschoben

Mark Reinhold hatte die Verschiebung vor einer Woche beantragt, wegen der zusätzlich benötigten Zeit, um durch den JCP-Prozess zu gehen. Da es keinen Widerspruch gab, ist die „General Availability" des JDK 9 damit für den 21. September vorgesehen. Termin für den ersten „Release Candidate" soll aber weiter der 22. Juni bleiben.

26. Juni 2017

Jigsaw geht im zweiten Anlauf über die Hürde

Jetzt ist es geschafft: Im zweiten Anlauf, dem sogenannten „Public Review Reconsideration Ballot", wird der JSR zum Plattform-Modulsystem vom Executive Committee akzeptiert – diesmal sogar einstimmig, nur Red Hat enthielt sich. Freuen wir uns also auf ein JDK 9 am 21. September 2017, nach endlos langem Warten jetzt mit einem Modulsystem, das aber sicher noch die eine oder andere Tücke beinhalten wird. Wobei es natürlich noch ein paar andere, teilweise spannendere Features im JDK 9 gibt, wie Reactive Streams – nicht zu verwechseln mit den JDK 8 Streams, auch wenn das API fast identisch ist. Einen guten Einstieg bietet der Vortrag „Real World Java 9" von Trisha Gee auf der Devoxx UK.
https://www.youtube.com/watch?v=GkP83hvdeMk

Andreas Badelt
stellv. Leiter der DOAG Java Community

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich in der DOAG Deutsche ORACLE-Anwendergruppe e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit dem Jahr 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Update+
Java-Community

From Legacy to modern Web – ein Reisebericht

Mirko Sertic

Das Internet ist zur Standard-Plattform für Anwendungen aller Art geworden. Eine große Herausforderung ist, bestehende Anwendungen an diesen Standard anzupassen und anpassbar zu halten.

Dieser Artikel zeigt am Beispiel einer Testbed-Anwendung, wie Domain-Driven Design und eventbasierte Architekturen als Grundlage für portierbare Anwendungen verwendet werden können. Ein besonderer Schwerpunkt liegt auf der Nutzung von bestehendem Quellcode und vorhandenen Libraries, Präsentations-Frameworks unterschiedlicher Art (JavaFX / HTML5) und den Möglichkeiten von Transpiler-Technologien, um den Spagat zwischen Wiederverwendung und technologischer Erneuerung zu meistern.

Die Reise

Am Anfang jeder Reise steht eine Idee, vielleicht auch ein gewisses Fernweh. Auch am Anfang meiner Reise war das nicht anders. Der Autor hat nicht von Palmen auf einer Südseeinsel geträumt, sondern sich mit Fragen über System- und Software-Architektur beschäftigt: Unsere (technologische) Welt verändert sich sehr schnell. Wir müssen uns ständig fragen, welche Technologie und welches Framework das Beste für den jeweiligen Anwendungsfall ist. Wir müssen uns fragen, welche Erscheinungen auf dem Mark nur ein kurzfristiger Hype sind und welche einen echten Mehrwert darstellen. Nicht zuletzt müssen wir uns über die wirtschaftlichen Aspekte unserer Lösungen Gedanken machen. Ist das wirtschaftlich? Ist das nachhaltig? Ist das wartbar? Funktioniert das wirklich? Aus all diesen Fragen ist der Wunsch für eine Testbed-Anwendung entstanden, um konkretere Anwendungen und auch Testfälle für bestehende und neue Technologien zu bekommen.

Was für ein Testbed notwendig ist

Für alle Programmiersprachen und Frameworks existiert ein klassisches „Hello World"-Beispiel. Damit erkennt man meistens mit wenigen Zeilen Quellcode die wichtigsten Features einer Programmiersprache. Für eine schnelle Übersicht sind diese Beispiele nicht schlecht. Für eine qualifiziertere Beurteilung sind sie allerdings viel zu klein. In einem Testbed braucht man komplexe Domänen-Logik. Man benötigt synchrone und asynchrone Komponenten sowie performancekritische Anforderungen. Es ist eine auf unterschiedliche Plattformen portierbare Anwendung erforderlich. Bei diesen Überlegungen kam dann dem Autor die ursprüngliche Motivation für seine Reise in den Sinn: Er möchte etwas prüfen und dabei lernen. Er möchte Spaß mit Arbeit kombinieren, was gerne unter dem Stichwort „Gamification" zusammengefasst wird. Und genau dieses Wort lieferte die entscheidenden Hinweise.

Ein Computerspiel erfüllt bei genauerer Betrachtung all seine Anforderungen an eine Testbed-Anwendung. Es besitzt komplexe Domänen-Logik bis hin zu Physik-Simulationen. Es besteht aus synchronen und asynchronen Komponenten. Es ist sehr stark ereignisgetrieben. Es stellt sehr hohe Anforderungen an die Performance. Ein Computerspiel ist idealerweise nicht an eine bestimmte Plattform gebunden. Es sollte auf dem PC genauso laufen wie auf einem Android- oder iOS-Handy oder auch auf einem Tablet. Das Testbed sollte also ein Computerspiel werden.

Koffer packen

Vor jeder Reise müssen Vorbereitungen getroffen werden. In fremden Ländern werden natürlich auch fremde Sprachen gesprochen. Es gibt Unterschiede in der Landschaft und auch in den Bräuchen. Auf diese Unterschiede muss sich der Reisende natürlich auch einlassen.

Zum Glück ist das Thema „Sprache" bei dieser Reise ein kleineres Problem, denn als plattformunabhängige Programmiersprache war Java sehr schnell das Mittel der Wahl. Viel problematischer ist die angesprochene Komplexität in der Testbed- Anwendung. Für die Modellierung von genau solchen Systemen gibt es zum Glück schon ein Werkzeug in unserem Werkzeugkasten, das natürlich auch in den Reisekoffer gehört: Domain-Driven Design. Darüber gibt es zwei Bücher, die mit ins Reisegepäck kommen: „Domain-Driven Design" von Eric Evans und „Implementing Domain-Driven Design" von Vaugn Vernon. Über die Entwicklung von Computerspielen gibt es ebenfalls zwei Bücher, die dem Autor besonders gut gefallen und die er deshalb auch mit auf die Reise nehmen möchte: „Real-Time Rendering: Third Edition" von Tomas Akenine-Möller, Eric Haines und Naty Hoffmann sowie „Game Coding Complete: Fourth Edition" von Mike McShaffry und David Graham. Diese Bücher lieferten die Grundlagen für diese Reise.

Fremde Sprachen lernen

Auf dieser Reise wird zum Glück überall Java gesprochen. Aus diesem Grund sollte das Thema „Sprache" eigentlich kein großes Problem darstellen. Trotzdem ist das Sprachthema ein Problem, und zwar ein großes. Wie sich herausstellt, werden hier nämlich zwei Sprachen gesprochen. Java ist die technische Sprache für die Formulierung von Quellcode. Es gibt aber noch eine zweite Sprache, die noch viel wichtiger ist und überall gesprochen wird: die Domänensprache, also die Sprache der Nicht-Techniker.

Wie sich als Vorbereitung für die Reise durch intensives Studium der erwähnten Fachliteratur herausstellte, ist die Domänensprache sehr komplex und umfangreich. Hier tauchen Begriffe auf wie „Game Objects", „Instrances", „Behaviors", „Sprites", „Views", „Events", „Templates" und „Sounds" sowie eine ganze Menge von Ereignissen und Beziehungen, die diese Begriffe auslösen und untereinander haben.

Wie passt nun diese Domänensprache mit der Implementierungssprache zusammen? Die Antwort auf diese Frage liefert uns das Domain-Driven Design. Es kennt den Begriff der „ubiquitous language" – der allgegenwärtigen Sprache. Diese Sprache ist spezifisch für eine Fachdomäne und beschreibt diese möglichst genau und widerspruchsfrei. Im weitesten Sinne ist sie also ein Wörterbuch. In diesem Wörterbuch kann man im Lauf der Reise immer wieder nachschlagen. Natürlich lassen sich auch Ergänzungen vornehmen, falls man etwas Neues entdeckt oder ein Eintrag vielleicht doch nicht ganz richtig ist.

Somit haben wir jetzt sowohl die Implementierungs- als auch die Domänensprache sehr stark normiert – aber wie sieht es mit der erwähnten Komplexität aus? Das Testbed soll natürlich kein aus Spaghetti-Code bestehender Monolith werden. Um dies zu verhindern, muss der Quellcode anders strukturiert sein. Damit diese Strukturierung so effizient wie möglich ist, beginnen wir mit der Strukturierung schon auf der Ebene der Domänensprache. Der Umgang mit Komplexität ist also nicht nur ein Teil der technischen Implementierung, sondern ein

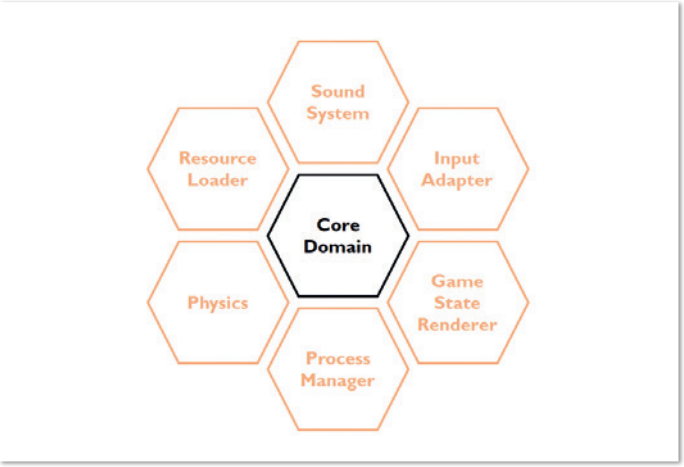


Abbildung 1: Context-Map der Testbed-Anwendung

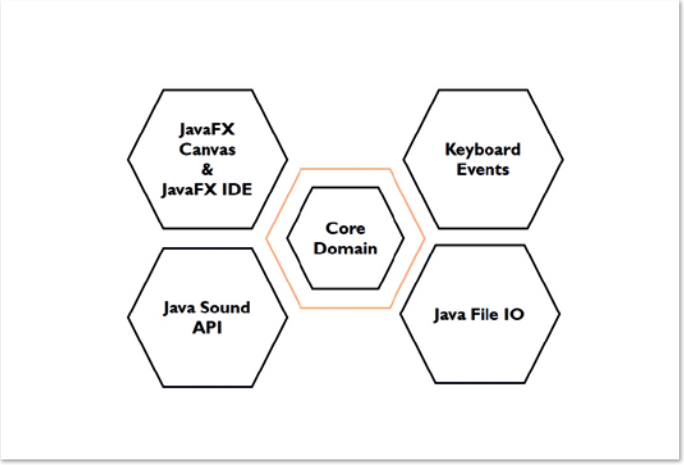


Abbildung 2: Die JavaFX-System-Architektur

wichtiger Teil der gesamten Problem-Modellierung. Domain-Driven Design kennt dafür das Werkzeug der sogenannten „Context-Map". Damit zerlegen wir ein großes System in mehrere Teilsysteme, gewinnen einen besseren Überblick und können damit auch die Komplexität des Gesamtsystems besser im Griff behalten. Die technische Struktur des Quellcodes kann sich dann an der Struktur der Context-Map orientieren beziehungsweise ist sogar identisch damit in Hinblick auf Modul- und Paket-Strukturen sowie deren Abhängigkeiten. *Abbildung 1* zeigt die erarbeitete Context-Map für das Testbed.

In der Mitte steht die Core-Domain. Darin ist der Kern des Systems, also die Essenz, enthalten. Um diese Core-Domain sind Adapter-Domänen angeordnet. Sie stellen entweder Schnittstellen zur Verfügung oder kapseln Funktionalität, die nicht Teil der Core-Domain, aber trotzdem für die Lauffähigkeit der Anwendung notwendig ist. Beispiele für Adapter-Domänen sind das Sound-System oder die Input-Adapter-Domänen.

Durch Anpassung beziehungsweise kompletten Austausch dieser Adapter lässt sich die Core-Domain einfach an andere Umgebungen anpassen. Ein Beispiel für eine eher kapselnde Domäne ist „Physics". Für die Lauffähigkeit des Systems wird in diesem Beispiel zwar eine Physik-Simulation benötigt, der Core-Domain ist es aber egal, welche konkrete Implementierung hier zum Einsatz kommt. Durch dieses Design kann beispielsweise das komplette Physik-Subsystem ausgetauscht werden, falls es zu Laufzeit-Problemen kommen sollte oder wichtige Features fehlen. Die Core-Domain muss dafür nicht angepasst werden.

In der Grafik sind die einzelnen Domänen ganz bewusst als Hexagone dargestellt. Es handelt sich also um eine hexagonale System-Architektur. Diese Architektur-Form bietet die zuvor genannten Vorteile und erleichtert zudem die Modularisierung, damit also auch die Test- und Wartbarkeit des Systems. Ein wesentliches Merkmal ist, dass in der Mitte unserer Hexagon-Struktur ein Rich-Domain-Modell steht, das keine Abhängigkeiten nach außen hat; es stellt lediglich Anforderungen an die umliegenden Adapter in Form von Java-Interfaces zur Verfügung.

Aufgabe der äußeren Schichten ist es nun, diese Schnittstellen zu

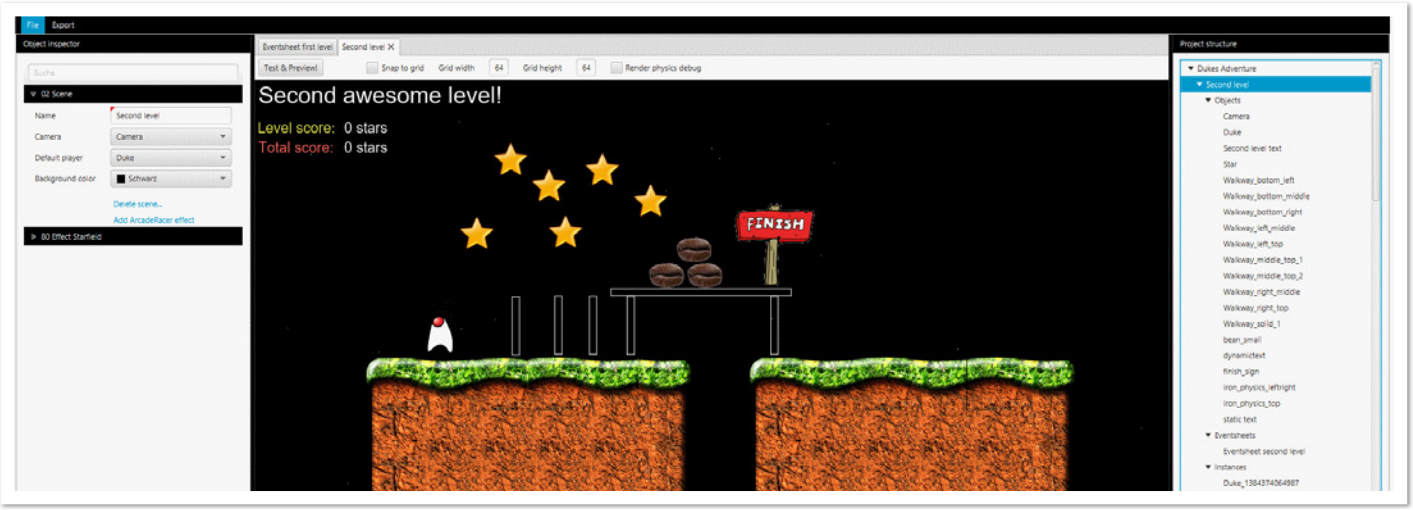


Abbildung 3: JavaFX Editor mit geladenem Level

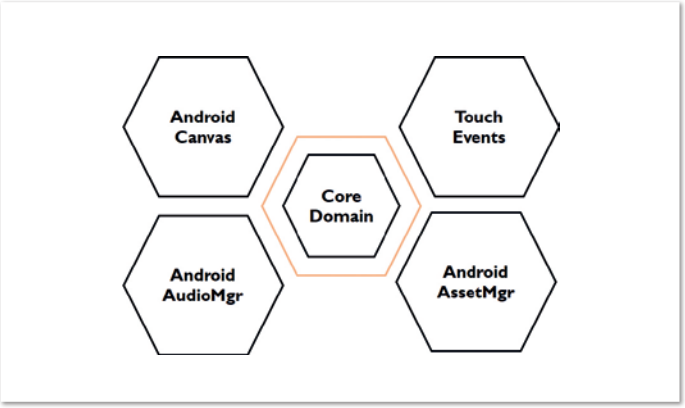


Abbildung 4: Die Android-System-Architektur

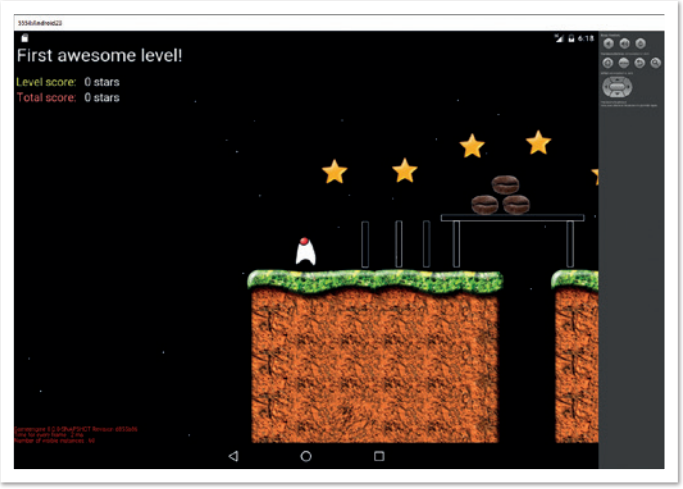


Abbildung 5: Das Spiel auf Android

bedienen. Somit ist die Implementierung der Physik-Simulation durch das Interface zwar abhängig von der Core-Domain, die Core-Domain ist jedoch nicht abhängig von der Implementierung. Beim Starten des Systems werden dann die einzelnen System-Komponenten über Dependency Injection verbunden. Mit dieser System-Architektur machen wir uns nun auf die Reise.

Ankunft im JavaFX-Land

Eine erste Etappe der Reise war ein lauffähiges Spiel, das JavaFX für die Visualisierung nutzt. Für die Eingabe sollte die Tastatur zum Einsatz kommen, die Darstellung erfolgt über eine Canvas-Komponente. Für dieses Reiseziel standen gleich mehrere Probleme an. Es musste natürlich die Core-Domain für alles implementiert werden. Zusätzlich waren aber auch die umliegenden Adapter der hexagonalen Architektur für die JavaFX-Umgebung zu implementieren. Außerdem musste natürlich noch das eigentliche Spiel konzipiert werden, das dann natürlich auch gespielt werden kann. Der Autor musste also noch einen Editor für die Core-Domain bauen. Hier zeigte sich ein großer Vorteil der hexagonalen Architektur, denn die Laufzeitumgebung-unabhängige Core-Domain kann sowohl innerhalb eines Editors als auch im tatsächlichen Spiel verwendet werden. *Abbildung 2* zeigt, wie die hexagonale System-Architektur aussieht.

Natürlich musste auch das Spiel konzipiert werden. Hier entschied sich der Autor für ein einfaches Jump-and-Run-Spiel mit dem Java-Duke. *Abbildung 3* zeigt die Ergebnisse der Arbeit – der Editor mit einem geladenen Level des Spiels. Im Spiel soll der Duke, gesteuert durch den Spieler über die Tastatur, Sterne sammeln, auf Bohnen herumhüpfen und über die Ziel-Flagge ins nächste Level springen.

Weiter nach Android-Land

In der nächsten Etappe der Reise soll das zuvor erstellte Spiel auf eine andere Laufzeit-Umgebung portiert werden, in diesem Fall Android. Wie kann das funktionieren? Zum Glück können Java-Programme auch für Android kompiliert werden, die komplette Core-Domain ist also wiederverwendbar. Aber wie sieht es mit den umliegenden Adaptern aus? Ziel der hexagonalen Architektur ist eine hohe Modularisierung und damit auch Adaptierbarkeit. Für Android waren also neue Adapter erforderlich.

Im Beispiel wird der Android Canvas für die Anzeige, der Android AudioManager für die Soundausgabe und der Android AssetManager für das Laden von Game Assets wie PNGs etc. verwendet. Neu für Android ist ebenfalls, dass es keine Tastatur im eigentlichen Sinne mehr gibt; die komplette Benutzer-Interaktion erfolgt über Touch-Events. Wir brauchen also noch einen Adapter, der diese Touch-Events in ein Format übersetzt, das von der Core-Domain verstanden werden kann. Das Diagramm in *Abbildung 4* zeigt die System-Architektur für Android und in *Abbildung 5* ist ein Screenshot für das Spiel zu sehen, das im Android-Emulator läuft.

Eine Zwischenbilanz

An diesem Punkt der Reise zieht der Autor eine kleine Zwischenbilanz. Wie weit war die Planung erfolgreich? Was hat funktioniert? Was nicht? Möchte er die Reise fortsetzen oder doch lieber an diesem Punkt abbrechen?

Als Fazit lässt sich auf jeden Fall sagen, dass die Wahl eines Computerspiels für ein Testbed eine sehr gute Idee ist. Die Portierungsanforderungen erzwingen geradezu eine Architektur im hexagonalen Stil. Ohne diese wäre eine Migration der Anwendung von JavaFX auf Android nicht so einfach möglich gewesen. Auch Java als plattformunabhängige Programmiersprache ermöglicht natürlich eine schnellere Migration auf andere Geräte. Auch hier war die Wahl aus Sicht des Autors richtig. Domain-Driven Design in Verbindung mit hexagonaler Architektur hat sehr positiven Einfluss auf die Gesamt-Architektur des Systems. Er möchte die Reise auf jeden Fall fortsetzen.

Das Internet-Land betreten

Das Computerspiel läuft bereits auf dem Desktop und auf mobilen Android-Geräten. Nun soll es auch als Webseite zur Verfügung stehen. Hier gibt es die erste grundlegende Änderung. Ein Browser versteht nur HTML, CSS und JavaScript. Ohne Erweiterungen lassen sich im Browser keine Java-Programme starten. Gerade diese Erweiterungen stellen ein Problem dar – sie sind entweder nicht installiert, aus Sicherheitsgründen deaktiviert oder schlichtweg veraltet. Es muss also ein anderer Weg gefunden werden, um das Spiel im Browser auszuführen.

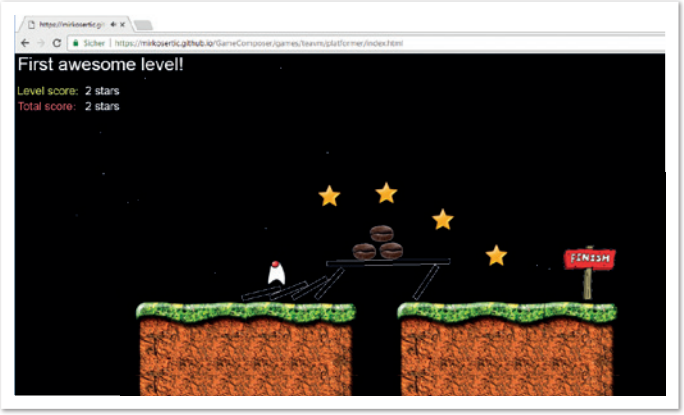


Abbildung 6: Spiel im Browser

Eine Option wäre natürlich, alles in JavaScript komplett neu zu schreiben. Dies ist aus technischer Sicht zwar möglich, aus wirtschaftlicher Sicht allerdings nicht machbar. Es würde redundanter Quellcode entstehen, der getrennt gewartet und weiterentwickelt werden muss, also entsprechender Aufwand. Zudem besteht das Risiko, dass die beiden Codebasen früher oder später auseinanderdriften.

Welche Möglichkeiten gibt es also noch? Hier betreten interessante Technologien die Bühne – sogenannte „Cross-Compiler". Diese können bestehende Programme von einer Sprache in eine andere übersetzen. Wir brauchen also einen Cross-Compiler, der Java nach JavaScript übersetzen kann. Als mögliche Lösungen kann der Autor zwei ganz besonders hervorheben: GWT und TeaVM.

GWT (Google-Web-Toolkit) ist ein Java-zu-JavaScript-Cross-Compiler und kann bestehenden Java-Quellcode nach JavaScript kompilieren. Für unser Computerspiel sind also neue Adapter zu bauen, die HTML5-Canvas für die Darstellung oder das Audio-API für die Soundausgabe nutzen. Diese lassen sich dann mittels GWT-Compiler nach JavaScript übersetzen.

TeaVM ist ein JVM-Bytecode-zu-JavaScript-Cross-Compiler. Anders als GWT ist kein Java-Quellcode für die Übersetzung notwendig, sondern JVM-Bytecode. Mit TeaVM kann man also auch andere Sprachen, die zu Bytecode kompiliert sind, nach JavaScript übersetzen. Genau wie für GWT sind hier die Adapter der hexagonalen Architektur an die HTML5-APIs anzupassen.

Für die weitere Reise entschied sich der Autor nach gründlicher Evaluierung für TeaVM. Gründe dafür waren, dass es für GWT schon länger kein größeres Major-Release mehr gab und dass TeaVM der erste Cross-Compiler war, der auch Java 8 nach JavaScript übersetzen konnte. Ein sehr starkes Argument war auch, dass der TeaVM-Compiler schneller als der GWT-Compiler war, bei teilweise besserem Laufzeitverhalten, Speicherverbrauch und Größe der entstandenen JavaScript-Datei. *Tabelle 1* zeigt die Metriken beim Vergleich der beiden Technologien. Für unsere weitere Reise verwenden wir TeaVM. Als Ergebnis erhalten wir unser Computerspiel lauffähig im Browser *(siehe Abbildung 6)*.

Bei der Evaluation von GWT und TeaVM hat sich eine weitere interessante Erneuerung herausgestellt. Die Implementierung eines HTML5-Canvas-Rendering-Adapters für die hexagonale Architektur war bei beiden Technologien fast identisch. Bei dieser Arbeit hat sich der Autor die Frage gestellt, ob diese Mechanik nicht auch mit WebGL in Verbindung mit GPU-Beschleunigung abbildbar ist. Es wäre natürlich sehr mühsam, die WebGL-Shader und die gesamte Infrastruktur für Textur-Management etc. von Hand zu schreiben, hier gibt es schon fertige Lösungen. Eine davon ist Pixi.js. Damit kann

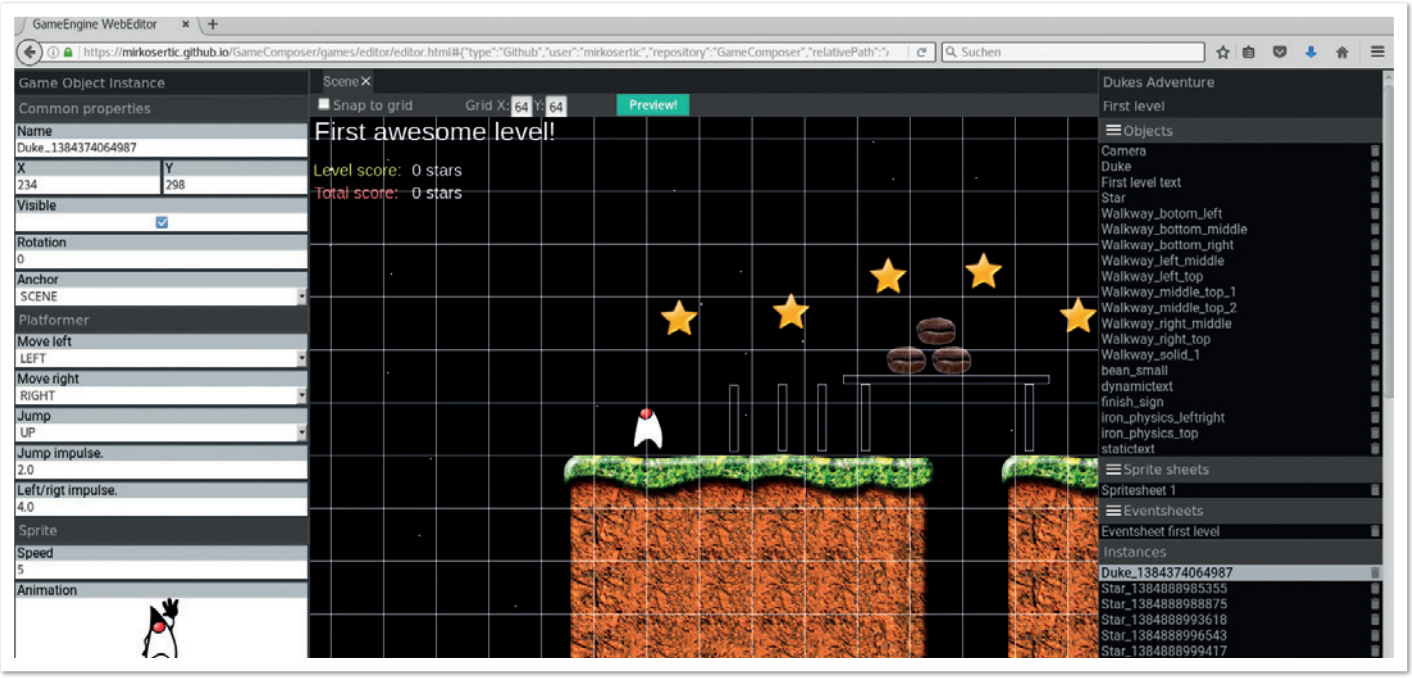


Abbildung 7: Die IDE im Browser

Metrik	GWT 2.8.0	TeaVM 0.4.3
Laufzeit des Compilers	43 Sekunden	12 Sekunden
Größe der JS-Datei	1.540 KB	1.013 KB

Tabelle 1

man einen komplexen Scene-Graphen bauen, der dann entweder WebGL beschleunigt oder über einen Canvas-Fallback ausgegeben wird. Dies ist geradezu ideal für unseren Anwendungsfall. Hier zeigt sich wieder einmal sehr deutlich, wie einfach eine hexagonale System-Architektur die Adaptierung an andere Umgebungen macht.

Fast am Ziel

Auf der bisherigen Reise hat der Autor erlebt und gezeigt, wie eine Java-Anwendung auf dem Desktop, auf mobilen Android-Geräten und als Web-Anwendung zur Verfügung gestellt werden kann, ohne dass die eigentliche Kernlogik neu geschrieben oder angepasst werden musste. Auf dieser Reise ist aber auch ein kleiner Teil leider noch in der alten Welt geblieben, und zwar die JavaFX-basierte IDE für das Spiel. Nun soll auch diese als Web-Anwendung zur Verfügung gestellt werden. Es gibt Projekte wie JPro, mit der JavaFX-Anwendungen ins Web portiert werden können. Diese erfordern aber teilweise Backend-Logik.

In unserem Anwendungsfall möchten wir allerdings eine Offline-Webanwendung bauen, ohne Verbindung zu irgendeiner Backend-Logik. Zum Glück lässt sich der bestehende Quellcode via TeaVM zu JavaScript kompilieren. Dies geht leider nicht für JavaFX-Komponenten, dieser Teil ist also, der hexagonalen Adapter-Philosophie folgend, neu zu schreiben. Hier stellt sich wieder die Frage: Mit welcher Technologie? Da JavaFX bereits komponentenbasiert ist, möchte man auch für die neue HTML-Anwendung ein komponentenbasiertes System einsetzen. Für den weiteren Weg der Reise entschied sich der Autor für die Evaluierung von HTML5-WebComponents zusammen mit dem Polymer-Framework. Nachdem die entsprechenden Adapter neu geschrieben waren, stand auch die IDE als Web-Anwendung bereit *(siehe Abbildung 7)*.

Hinter den Kulissen arbeitet hier ein interessanter Technologie-Stack. Die komplette Anwendung ist in Java geschrieben und wurde mit TeaVM zu JavaScript übersetzt. Die Anzeige ist mit Pixi.js über WebGL geregelt. Die einzelnen Anzeigebereiche sind über teilweise verschachtelte Web-Components gegliedert. Der aktuelle Zustand der Spielebeschreibung ist in der IndexedDB des Browsers gespeichert und kann bei Bedarf in ein GitHub-Repository übertragen werden.

Die GitHub-Integration war ein notwendiger Schritt für die Game-IDE, da der Zustand des Spiels ja irgendwo gespeichert werden muss und ein JavaScript im Browser bedingt durch die JavaScript-Sandbox nicht auf das lokale Dateisystem zugreifen kann. GitHub stellt ein gutes JavaScript-API zur Verfügung. Die Integration war daher sehr einfach möglich und bietet für unseren Anwendungsfall eine echte Bereicherung.

Am Ziel und darüber hinaus

Die bisherige Reise war eine interessante Erfahrung. Seit Beginn der Reise, die schon mehrere Jahre zurückliegt, benutzt der Autor dieses Testbed immer wieder, um neue Technologien zu prüfen, oder einfach nur, um ein wenig zu spielen und Abwechslung vom Alltag zu bekommen. Er hofft, Interesse für Themen wie Domain-Driven Design und hexagonale Architektur wecken zu können. Natürlich hofft er auch, dass am Beispiel seines Testbed die theoretischen Vorteile auch in der Praxis bewiesen werden konnten.

Was gibt es noch zu sagen? Die Reise wird weitergehen. Am Horizont sind schon neue, interessante Technologien zu erkennen, die geprüft und untersucht werden müssen. Neuronale Netzwerke können beispielsweise für Muster- und Gestenerkennung benutzt werden, mit Erweiterung natürlich auch für komplexe Planungsaufgaben. Die KI von Computerspielen stellt hier ein sehr breites Aufgabenspektrum zur Verfügung.

Eine andere Technologie ist WebAssembly. Es ist neu in Form eines MVP in allen aktuellen Browser-Versionen verfügbar und die Weiterentwicklung von asm.js. WebAssembly ist Bytecode für das Web. Der hier aufgezeigte Transpiler TeaVM unterstützt nicht nur die Kompilierung von JVM-Bytecode nach JavaScript, er bietet auch experimentelle Unterstützung für WebAssembly. WebAssembly bietet neben einem kompakteren Format noch den Vorteil einer effizienteren Laufzeitumgebung und kann als Ergänzung zu bestehenden Web-Anwendungen für komplexere Berechnungen wie Physik-Simulationen genutzt werden.

Wo die Reise enden wird, lässt sich im Moment nicht sagen. Der Autor hofft, auch weiterhin viel Spaß mit seinem Technologie-Testbed zu haben und auch zukünftig viele neue Themen damit ausprobieren und verwirklichen zu können. In diesem Sinne: Schöne Reise.

Mirko Sertic
mirko@mirkosertic.de

Mirko Sertic ist Software Craftsman im Web/eCommerce-Umfeld. In Funktionen als Software-Entwickler, Architekt und Consultant in Projekten in Deutschland und der Schweiz sammelte er Erfahrungen mit einer Vielzahl an Frameworks, Technologien und Methoden. Heute arbeitet er als System-Analyst bei der Thalia Bücher GmbH in Münster mit Schwerpunkt auf Java, eCommerce und Integrations-Technologien. Seine Freizeit verbringt er mit seiner Freundin, seiner Familie und hin- und wieder mit Open-Source-Projekten.

Statische Code-Analyse mit SonarQube

Josha von Gizycki, TRIOLOGY GmbH

„Software-Qualität" ist nach wie vor ein Dauerthema, denn sowohl Kunden als auch Entwickler wissen, welche Folgen fehlerhafte Software haben kann. Umso wichtiger ist es, den Quellcode regelmäßig einer entsprechenden Untersuchung zu unterziehen.

Tools zur statischen Code-Analyse können einen wichtigen Beitrag liefern. Die Analyse kann dazu beitragen, Fehler, Sicherheitslücken oder einen ungenügenden Aufbau ausfindig zu machen, ohne dass dazu eine Ausführung notwendig ist. Der Artikel zeigt, wie eine statische Code-Analyse funktioniert, welche Vorteile das Tool SonarQube bietet und warum eine umfassende Interpretation notwendig ist.

Was macht statische Code-Analyse?

Statische Code-Analyse nimmt kompilierten Code oder Quelltext, wendet Metriken darauf an und generiert Zahlen. Damit dies nicht von Hand erfolgen muss, gibt es eine Vielzahl unterschiedlicher Tools, auf die zurückgegriffen werden kann. Die bekanntesten in der Java-Welt sind „PMD", „Checkstyle" und „FindBugs":

- „PMD" sucht nach ineffizientem Code. Darunter fallen beispielsweise leere Codeblöcke, ungenutzte Variablen oder verschwenderisches Nutzen von „Strings" oder „StringBuffern".
- „Checkstyle" prüft den Programmierstil und arbeitet dabei genau wie „PMD" auf Quelltext-Ebene. Dadurch kann die Einhaltung von Programmier-Richtlinien oder Formattern erzwungen werden.
- „FindBugs" arbeitet als einziges der drei genannten Tools auf Bytecode-Ebene. Es sucht dabei nach harten Fehlern wie Problemen in Klassen-Hierarchien, fehlerhaften Array-Behandlungen, unmöglichen Typen-Umwandlungen oder nicht in Paaren überschriebenen „equals"- und „hashcode"-Methoden.

Diese Werkzeuge nutzen entsprechend ihrer Natur und Einsatzgebiete mal mehr, mal weniger Metriken, um ihre jeweiligen Statistiken zu erzeugen. Der Name „Metrik" trägt jedoch wenig Bedeutung von dem in sich, was eine Metrik ausmacht. So hat der Name seinen Ursprung im Lateinischen „ars metrica", was mit „Lehre von den Maßen" übersetzt wird. Fragt man jedoch das Institute of Electrical and Electronics Engineers, erhält man folgende Antwort: „Software quality metric: A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality." „Eine Software-Qualitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet, der als Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit interpretierbar ist." – IEEE-Standard 1061, 1998.

In diesem Fall wird eine Metrik als eine Funktion verstanden, die für beliebige Eingaben Zahlen erzeugt. Diese sind dabei so beschaffen, dass sie untereinander vergleichbar sind, solange sie von derselben Funktion erzeugt wurden. Auf diese Weise können Rückschlüsse auf die Eingabe mit Hinblick auf die Funktion gezogen werden.

Ein Beispiel dafür ist die McCabe-Metrik, auch „zyklomatische Komplexität" genannt. Diese sehr grundlegende Metrik berechnet die Anzahl der unterschiedlichen Pfade durch ein Stück Code. Die For-

```
String nameOfDayInWeek(int nr) {
    switch(nr) {
        case 1: return "Monday";
        case 2: return "Tuesday";
        case 3: return "Wednesday";
        case 4: return "Thursday";
        case 5: return "Friday";
        case 6: return "Saturday";
        case 7: return "Sunday";
    }
    return "";
}
```

Listing 1

```
String nameOfDayInWeek(int nr) {
    String[] names =  new String[] {
        "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday",
        "Sunday"
    };
    if(nr > 0 && nr <= names.length) {
        return names[nr - 1];
    }
    return "";
}
```

Listing 2

mel ist sehr einfach: Es wird die Anzahl von Kontrollstrukturen wie „if", „while", „case" sowie boolescher Operatoren wie „&&" und „||" hochsummiert und 1 addiert. Diese Information soll anhand eines Beispiels verdeutlicht werden *(siehe Listing 1)*.

Diese sehr einfache Methode gibt den Namen eines Wochentages entsprechend seiner 1-indizierten Position innerhalb der Woche zurück. Ihre zyklomatische Komplexität beträgt acht: 1 plus 7 mal „case". Dies ist ein relativ hoher Wert: Ein Maximalwert von 10 gilt als allgemein akzeptiert und ausreichend erprobt. Um also die Komplexität dieser Methode zu verringern, wird sie refaktorisiert *(siehe Listing 2)*.

Die zyklomatische Komplexität dieser Methode beträgt drei: „1 plus 1 mal if plus 1 mal &&". Durch den unterschiedlichen Ansatz wird die Komplexität verringert, jedoch ist es relativ unstrittig, dass die erste Version schneller verstanden werden kann.

Sollen die genannten Werkzeuge nun zusammen benutzt werden, müssen alle entsprechend konfiguriert und ihre Ergebnisse zusammengeführt werden, damit sich ein gemeinsames Bild ergibt. Außerdem kommt es dabei zwangsweise zu Dopplungen in ausgewerteten Metriken oder anderen Kennzahlen. „PMD" beispielsweise besitzt durch seinen relativ vagen Aufgabenbereich Überschneidungen im Hinblick auf Codestil mit „Checkstyle", während es aber auch genauso wie „FindBugs" auf ungenutzten Code achtet. An solchen und weiteren Stellen kann SonarQube Verbesserungen herbeiführen.

SonarQube

SonarQube besteht im Wesentlichen aus drei grob voneinander abtrennbaren Komponenten: einem Scanner, der Code entgegennimmt und analysiert, einer Datenbank, in der Analyse-Ergebnisse gespeichert werden, und einer Web-Komponente, die die gesammelten Ergebnisse aufbereitet anzeigt. Auf diese Weise kann der Scanner aus beliebigen Quellen aufgerufen werden, beispielsweise von einem Maven-Build, einem CI-Server oder aus IDEs heraus. SonarQube steht unter der LGPL v3 und ist damit Open Source.

In Sachen Analyse und Metriken bedient sich SonarQube dabei unter anderem der bereits genannten Tools. Die Analysen aus „PMD" und „Checkstyle" sind fest integriert, „FindBugs" kann über eine Plug-in-Schnittstelle nachinstalliert werden.

Für die Analyse-Ergebnisse von Metriken werden von SonarQube Richtwerte bereitgestellt. Für jeden Verstoß gegen einen solchen Richtwert entsteht ein Issue. Diese werden nach Kategorie und Schwere sortiert. Für die folgenden Screenshots wurde eine Analyse von Apache Log4j in der Version 1.2.18-SNAPSHOT gefahren.

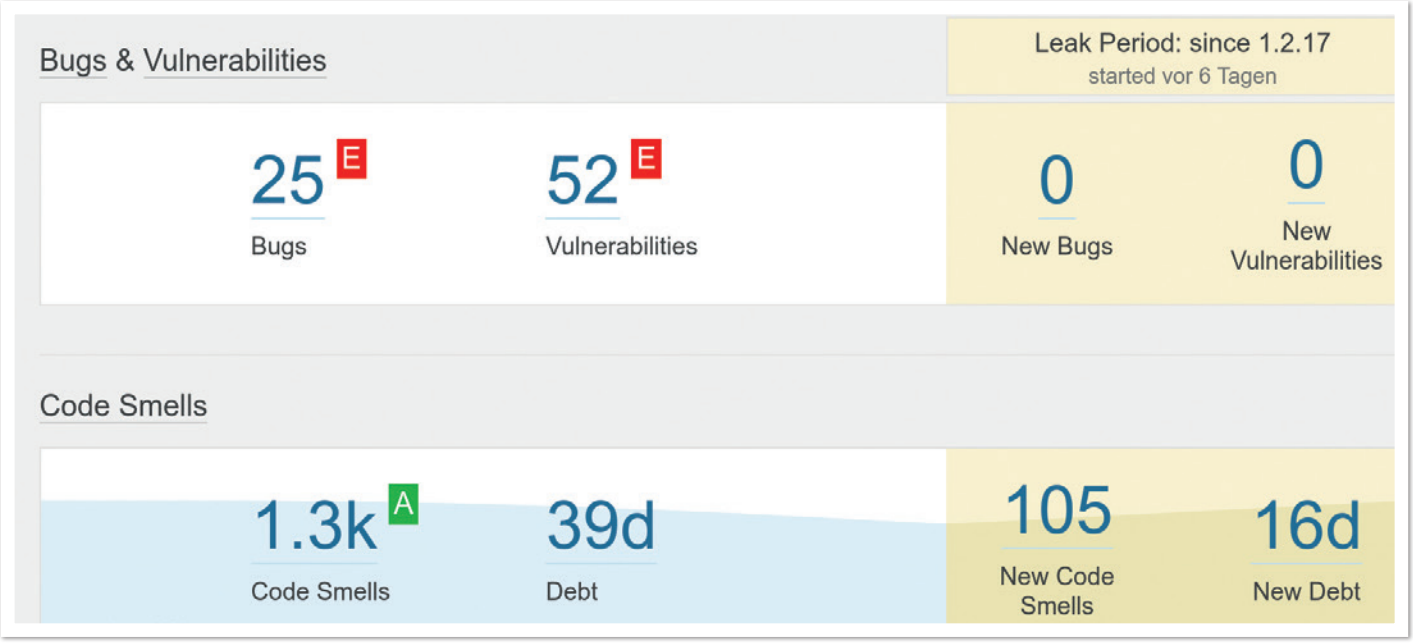


Abbildung 1: Die Kategorien

Kategorien

Bei der Analyse unterteilt SonarQube Metrik-Verstöße, „Issues" genannt, in drei Kategorien *(siehe Abbildung 1):*

- **Code Smells**
  Beispiele sind die zyklomatische Komplexität, als „Deprecated" markierter Code oder unnütze mathematische Funktionen, beispielsweise das Runden von Konstanten. Solche Issues deuten in den meisten Fällen auf tieferliegende Probleme hin. Ist beispielsweise die zyklomatische Komplexität einer Methode zu hoch, deutet das eventuell auf einen Design-Mangel in der Architektur hin.
- **Vulnerability**
  In dieser Kategorie landen Issues, bei denen es um Sicherheit geht. Damit ist nicht nur die Sicherheit in Form von SQL-Injection oder fest einprogrammierten Passwörtern gemeint, sondern auch die innere Sicherheit. Beispielsweise wird „public static fields should be constants" hier eingeordnet.
- **Bug**
  Darüber können klassische Handwerksfehler identifiziert werden, die beispielsweise der Java-Spezifikation widersprechen. So werden hier Issues wie der Vergleich von Klassen über ihren nicht voll qualifizierten Namen, unendliche Schleifen oder das Dereferenzieren von bekannten Null-Variablen verortet.

Schwere

Darüber hinaus werden Issues nach ihrer Schwere unterteilt. Diese reichen vom Entwicklungsstopp bis zum Hinweis am Rande *(siehe Abbildung 2):*

- **Blocker**
  In der schwersten Kategorie befinden sich Issues wie fest hinterlegte Passwörter. Wie der Name schon nahelegt, sollte kein Projekt mit derartigen Issues weiterentwickelt werden, ohne sie zu beseitigen.
- **Critical**
  Issues wie unbehandelte Exceptions oder andere, die Programmabläufe schwer beeinträchtigen können, werden mit „Critical" bewertet.
- **Major**
  Diese Issues gehören bereits in den Bereich „Codestil" oder „Konventionen" wie leere Code-Blöcke ohne erklärenden Kommentar, fehlende „@Override"-Annotationen oder auskommentierter Code.
- **Minor**
  Issues dieser Schwere sind syntaktisch korrekt, ihre Semantik lässt jedoch zu wünschen übrig. Dazu gehören unnötige Typumwandlungen, Duplikationen oder ungenutzte Rückgabewerte.
- **Info**
  Diese Schwere wird Issues zugeordnet, die irgendwann einmal behandelt werden sollten, wie Todo-Kommentare oder das Entfernen von „@Deprecated"-Code.

Automatische Interpretation

SonarQube nutzt ein Verfahren zur groben Bewertung der Code-Basis: die technische Schuld. Jedem Issue wird dabei eine Zeit zugeordnet, die benötigt wird, um ihn zu beheben. Die Summe dieser technischen Schuld wird dann im Verhältnis zum Gesamtaufwand des Projekts gestellt und ergibt das Maintainability Rating: „Main-

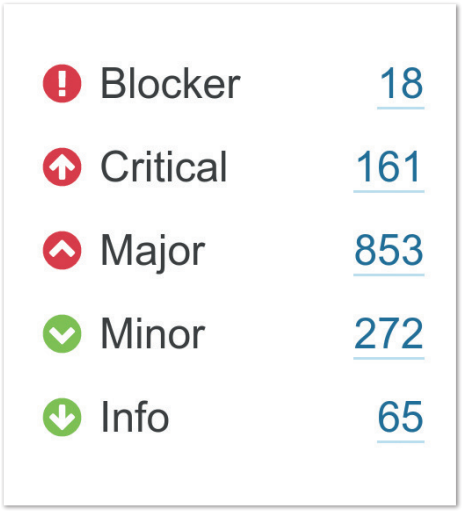


Abbildung 2: Die Schwere

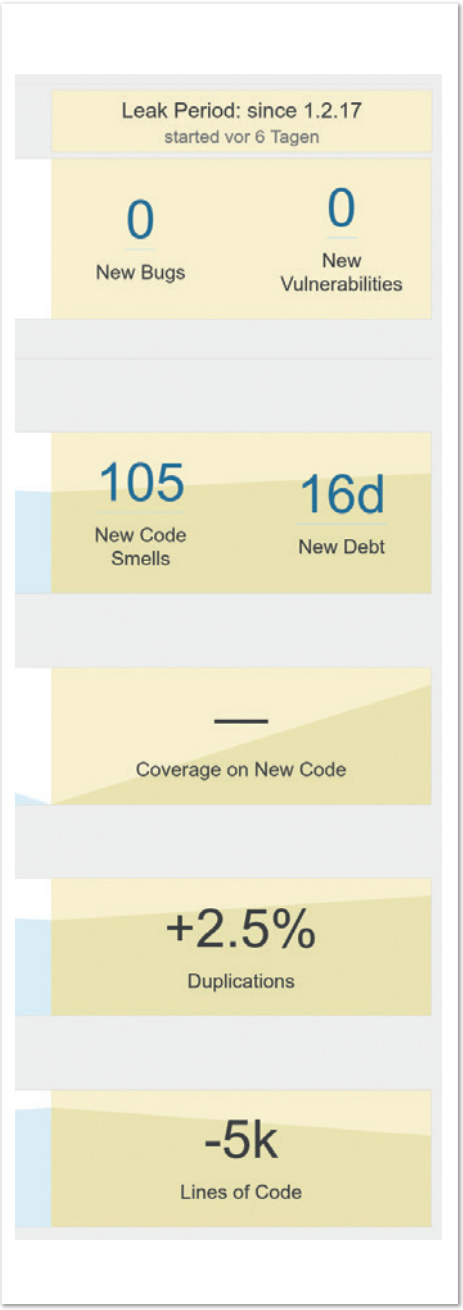


Abbildung 3: Entwicklung Projektgröße zu Duplikationen

tainability Rating = Technical Debt/Development Cost". Je nach Verhältnis dieser beiden Kennziffern fällt demnach das von SonarQube genannte Maintainability Rating aus:

- A: 0 – 0,1
- B: 0,11 – 0,2
- C: 0,21 – 0,5
- D: 0,51 – 1
- E: > 1

Als Mittelwert nimmt SonarQube an, dass pro Codezeile dreißig Minuten Entwicklungszeit benötigt wurden. Dies trifft natürlich lange nicht auf jede Codezeile zu, aber als grober Mittelwert sollte dies für große Projekte über die Zeit zutreffen. Eine Beispielrechnung: Bei einem kleinen Projekt von 2.500 Zeilen produziertem Code und fünfzig Tagen Entwicklungszeit wurde technische Schuld angesammelt. Bei einem typischen Arbeitstag von acht Stunden Länge können sechzehn Zeilen Code geschafft oder 0,0625 Tage pro Zeile benötigt werden. Dies führt zu folgender Rechnung: „50/(0,0625 * 2.500) = 0,32". Laut der obigen Tabelle ergibt das die Note C.

Bei genauerer Betrachtung des Bewertungsmaßstabs wird erkennbar, dass pro Zeiteinheit für die Entwicklung maximal zehn Prozent davon an technischer Schuld generiert werden muss, um schlechter als mit einer A-Note bewertet zu werden. Die Erfahrung lehrt, dass jedes Projekt, das groß genug ist, in der Gesamtbetrachtung genau diese Note erreicht. Das wirkt weniger erstaunlich, wenn man bedenkt, dass große Projekte lange laufen und über genau diesen langen Zeitraum hinweg im Durchschnitt viel durchschnittlich guter Code produziert wurde. Daher ist es interessanter, wenn das Maintainability-Rating nach den bereits bekannten Kategorien berechnet wird: „Bugs", „Vulnerability" und „Code Smells", wie *Abbildung 2* schon zeigt.

Manuelle Interpretation

Im ersten Schritt ergibt sich durch die Analyse eine eher unüberschaubar große Menge von Statistiken und Zahlen. Dabei ist es nicht damit getan, diese Statistiken zu akzeptieren und ihnen schlichtweg zu glauben. Der interessante Teil bei Code-Analyse ist, dieser durch Interpretation Bedeutung zu verleihen. Dabei gilt es, ein paar einfache Regeln zu beachten:

- **Relative Werte schlagen absolute**
  5.000 Issues im Projekt? Diese auf den ersten Blick hohe Zahl relativiert sich, wenn man weiß, dass es nur fünf pro Klasse sind.
- **Änderungen schlagen den Status quo**
  Ganz dem Prinzip des Vorwärtsdenkens entsprechend ist es interessanter, dass 5.000 Issues im letzten Release geschlossen werden konnten und nur 500 hinzukamen, als dass 4.500 noch offen sind. Die positive Entwicklung ist das Wichtige.
- **Metriken verstehen**
  Wenn eine Metrik nicht verstanden wurde, kann sie weder angewandt noch kann ihr Ergebnis interpretiert werden. Beispielsweise ist es schön zu wissen, was zyklomatische Komplexität ist. Wer das allerdings mit Lesbarkeit oder gar Wartbarkeit gleichsetzt, befindet sich auf dem Holzweg.
- **Metriken in Relation zueinander stellen**
  Wie der vorige Punkt schon andeutet, sagt eine Metrik allein wenig aus. Eine hohe zyklomatische Komplexität kann nämlich durchaus bedeuten, dass der betroffene Code schwer zu lesen ist. Allerdings gibt es auch noch Metriken wie die maximale Verschachtelungstiefe oder die Komplexität boolescher Ausdrücke, die da auch noch ein Wörtchen mitzureden haben.

Metriken verstehen

Die beliebteste, zugleich aber auch am häufigsten missbrauchte Metrik ist die der „Lines of Code" (LOC). Dass allerdings LOC nicht gleich LOC ist, zeigt ein kurzer Blick in gängige Nachschlagewerke:

- **Lines of Code (LOC)**
  Alle physisch existierenden Zeilen, also Kommentare, Klammern, Leerzeilen etc.
- **Source Lines of Code (SLOC)**
  Wie LOC, nur ohne Leerzeilen und Kommentare
- **Comment Lines of Code (CLOC)**
  Alle Kommentarzeilen
- **Non-Comment Lines of Code (NCLOC)**
  Alle Zeilen, die keine Leerzeilen, Kommentare, Klammen, Includes etc. sind

Wenn man nun die Größe eines Projekts beziffert, welche Kennziffer sollte herangezogen werden? Oder anders: Wie groß mag der Einfluss von unterschiedlichen Code-Styleguides auf die Anzahl der LOC und auf die Anzahl von NCLOC sein? SonarQube nutzt für seine Angaben über die Anzahl an Codezeilen NCLOC.

Metriken in Relation zueinander stellen

Listing 3 zeigt ein weiteres Mal Beispielcode. Die oben stehende Methode sucht in einer doppelt geschachtelten Liste nach einem Objekt mit einer gegebenen ID. Nach McCabe besitzt sie eine zyklomatische Komplexität von fünf, alles im Rahmen. Allerdings besitzt sie gleichzeitig eine maximale Verschachtelungstiefe von vier. SonarQube definiert ein Maximum von drei, also besitzt diese Methode ihren ersten Issue. Hätten wir nur McCabe betrachtet, wäre sie als grün durchgegangen und im Bericht nicht aufgefallen. Ein Early Return später haben wir das Problem gelöst *(siehe Listing 4)*.

Die Metriken sind zufrieden – wir auch? Es gibt einen Grund, warum sie zur Kategorie „Code Smell" gehören: Hier wurde sehr wahrscheinlich beim Architektur-Design versagt. Warum wurde eine doppelt geschachtelte Liste in die Methode hineingereicht? Arbeitet der Code absichtlich auf dieser geringen Abstraktionsebene? Wurde vielleicht am Domänen-Design vorbei gearbeitet? Warum wurde nicht gleich das passende DTO aus der Datenquelle abgefragt, statt eine Methode mit hoher Laufzeit-Komplexität anzustoßen?

```
DTO search(List<List<DTO>> rawData, int id) {
  if(rawData != null) {
    for(List<DTO> sublist : rawData) {
      for(DTO dto : sublist) {
        if(dto.getId() == id) {
          return dto;
        }
      }
    }
  }
  return null;
}
```

Listing 3

```
DTO search(List<List<DTO>> rawData, int id) {
  if(rawData == null) {
    return null;
  }

  for(List<DTO> sublist : rawData) {
    for(DTO dto : sublist) {
      if(dto.getId() == id) {
        return dto;
      }
    }
  }
  return null;
}
```

Listing 4

Viele sagen auch, dass das DTO-Pattern ein Antipattern ist. Diese Fragen legen den Schluss nahe, dass der eigentliche Fehler nicht in der einzelnen Methode liegt, sondern auf einer höheren Ebene zu suchen ist.

Als weiterer Punkt wurde oben schon die Entwicklung genannt: Änderungen schlagen den Status quo. SonarQube erstellt Historien, die sich auch wunderbar auswerten lassen. Beispielsweise kann man die Entwicklung der Projektgröße mit der Anzahl der Duplikationen vergleichen *(siehe Abbildung 3)*.

Im selben Zeitraum, in dem 5.000 Zeilen Code entfernt wurden, kamen 2,5 Prozent Code-Duplikationen hinzu. Dies kann bedeuten, dass redundanter Code geschaffen oder nicht redundanter Code gelöscht wurde, sodass Duplikationen schwerer wiegen konnten. Wird dann noch in Betracht gezogen, dass das Projekt insgesamt nur 16.000 Zeilen Code besitzt, ist Letzteres wahrscheinlicher.

Erfahrungswerte

Eine ausreichend große Code-Basis vorausgesetzt, erreichen viele Legacy-Projekte ein A-Rating. Wie bereits gesagt, ist das wenig verwunderlich. Die Aufteilung nach Issue-Kategorien hilft an dieser Stelle enorm.

Im Embedded-Bereich gibt es den Grundsatz, dass man pro dreißig Regelverstöße drei kleinere und einen schwerwiegenden Bug erwarten kann. Diese Regel hilft, aus der Anzahl der Issues während der Entwicklung auf die Bugs während des Betriebs zu schließen. Denn meistens fällt es schwer, eine direkte Verbindung zwischen diesen Kennziffern herzustellen: In manchen prognostizierten „NullPointer" laufen Benutzer beispielweise einfach nicht hinein, weil die Applikation gar nicht auf die nötige Weise bedient wird. Allerdings helfen auch hier Mittelwerte wie die oben genannten. Die reinen Zahlenwerte müssen sicherlich fein abgestimmt sein, um auf den jeweiligen Einsatzbereich zu passen, das Muster sollte jedoch stimmen.

Live mit SonarQube zu arbeiten, kann dazu führen, dass Issues entweder sofort behoben werden oder versucht wird, sie gleich im Vorfeld zu vermeiden. Diese durchaus lobenswerte Neigung kann allerdings dazu führen, dass zu komplex gedacht und das eigentliche Ziel aus den Augen verloren wird. Dieses Verhaltensmuster hat einen Namen: „Over Engineering". Empfehlenswerter ist es, gemeinsam zum Sprint-Ende oder an vergleichbaren Terminen zu sichten, wie sich die Codebasis entwickelt.

Fazit

Statische Code-Analyse kann eine gute Grundlage für die Sicherstellung von Code-Qualität sein. Allerdings ist ebenso die Fähigkeit der Interpretation notwendig, damit die Code-Analyse die vollständige Wirkungskraft entfalten kann. So sind Statistiken zwar interessant, jedoch ist eine statische Code-Analyse erst dann wirklich vollständig, wenn ein Mindestmaß an Interpretation hineingeflossen ist.

So liefert Code-Analyse in erster Linie ein Gefühl für die Code-Basis. Erst so können fundierte Aussagen darüber getroffen werden, welche Bereiche des Projekts besonders gefährdet, instabil oder renovierungsbedürftig sind.

Für die Teams können regelmäßige Analysen die Team-Motivation erhöhen. Eine positive Issue-Bilanz am Ende eines Sprints und aufwärtszeigende Historien-Graphen sind gute Treiber für ein Entwickler-Team und Beweis der eigenen Leistung. Darüber hinaus können Analyse-Ergebnisse als Argumentationsgrundlage gegenüber dem Kunden dienen. Mithilfe der Projekt-Historie, die eine Auswahl gut darstellbarer Kennzahlen beinhaltet, kann vor Kunden oder Entscheidern besser über ein eventuell nötiges technisches Release diskutiert werden.

Josha von Gizycki
josha.von.gizycki@triology.de

Josha von Gizycki arbeitet als Software-Entwickler bei der TRIOLOGY GmbH in Braunschweig und hat sich auf die Entwicklung von Enterprise-Applikationen im Java- und Oracle- Umfeld spezialisiert. Sein Schwerpunkt liegt dabei in den Bereichen „Wartung" und „System-Design".

Penetrationstest – geschnitten oder am Stück?

Tobias Glemser, secuvera GmbH

Der Wissensstand und entsprechend die Planungen und Anfragen zu Penetrationstests sind sehr unterschiedlich. Der Artikel gibt einen Überblick darüber, was ein Penetrationstest sein kann, welche Fähigkeiten Penetrationstester haben sollten und nach welchen Standards vorgegangen wird.

Die Anfragen für Penetrationstests sind häufig sehr unterschiedlich. Dies hängt direkt mit dem Wissensstand der Fachseite zusammen, die diese Anfragen formuliert. Es ist nicht unüblich, dass die Anfrage alleinig darin besteht zu erfahren, was denn ein Penetrationstest kostet. Unter Penetrationstests kann man sehr viele unterschiedliche Prüfungen subsumieren. Dafür ist auf Seiten des Anfragenden eine gewisse Expertise notwendig. In einigen Fällen ist die Anfrage deutlich zu unspezifisch oder gar widersprüchlich.

Ein Beispiel aus der Praxis: Angefragt wird die Prüfung von Servern und Infrastruktur. In der detaillierten Leistungsbeschreibung sind dann Adressen von Web-Anwendungen genannt. Damit ist der Fokus der Prüfungen nicht klar.

Sofern es wirklich um die Prüfung von Servern und Infrastruktur, wie zum Beispiel die eines Firewall-Systems, gehen soll, ist die Vorgehensweise eine andere als bei der Prüfung von Web-Anwendungen. Es handelt sich in beiden Fällen um Penetrationstests; die Prüfmethodik ist jedoch eine andere. Aus der Prüfmethode ergeben sich auch andere Prüfrisiken.

In anderen Fällen haben sich die Ausschreibenden bereits mit dem Thema beschäftigt, Literatur gelesen und spezifizieren den Penetrationstest sehr genau. Aufgrund mangelnder Praxiserfahrung werden jedoch häufig Anforderungen gestellt, die entweder einen großen Aufwand bei geringem Erkenntnisgewinn verursachen würden oder für das jeweilige Prüfobjekt nicht zielführend sind. Darüber hinaus wird das Risiko, das von den Prüfungen ausgeht, nur selten betrachtet.

Basisprüfungen von Systemen sind vergleichsweise risikoarm. Prüfungen von Anwendungen, insbesondere auf Produktivsystemen, bergen je nach Anwendung große Risiken – beispielsweise die Veränderung von Daten, Integritätsverlust oder auch der Ausfall von Anwendungen, selbst wenn der Ausfall durch sogenannte Denial-of-Service-Angriffe (DoS) nicht explizit provoziert wird.

Definition Penetrationstests

Es gibt sehr viele unterschiedliche Definitionen von Penetrationstests. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) definiert in seiner Studie *(siehe „https://www.bsi.bund.de/DE/Publikationen/Studien/Pentest/index_htm.html")* einen Penetrationstest als kontrollierten Eindringversuch, der vergleichbar zu einem realen Angriff durchgeführt wird. Er ist zeitlich begrenzt und auch nur eingeschränkt standardisierbar. Das National Institute of Standards and Technology (NIST) schreibt im Dokument SP800 *(siehe „http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf"),* dass das Ziel eines Penetrationstests ist, Schwachstellen aufzudecken. Ein Penetrationstest sei eine Resilienzprüfung vor dem Hintergrund von Zeit, Ressourcen und Kenntnissen. Ebenso wie das BSI schreibt das NIST, dass ein Penetrationstest vergleichbar mit einem realen Angriff sein soll. Damit wird klar, dass der angenommene Angreifer selbst zu definieren ist.

Welche Eigenschaften ein Penetrationstester mitbringen soll, beschreiben BSI und NIST ebenso. Ein Penetrationstester benötigt demnach langjährige Erfahrung als System- und Netzwerk-Administrator, er beherrscht Programmiersprachen, kennt sich in Produkten wie Firewalls sowie in Intrusion-Detection-Systemen aus und kennt seine Angriffswerkzeuge aus dem Effeff. Darüber hinaus ist er ein kreativer Geist. Salopp gesagt: Wenn irgendwo ein Administrator oder ein Entwickler fehlt, wie schön, dass ein Penetrationstester das alles kann. Das NIST greift einen weiteren interessanten Punkt auf: Kommunikationsfähigkeit.

Das Ergebnis eines Penetrationstests ist immer die Dokumentation des Tests. Sie muss den Leser fachlich dort abholen, wo er steht. Diese Transferleistung in einem geschriebenen Dokument muss der Penetrationstester erbringen. Insbesondere bei der Zusammenfassung für das Management ist dies enorm wichtig. Der Entscheider muss in der Lage sein, das durch die Prüfungen aufgedeckte Risiko aufgrund der Dokumentation zu bewerten. Sollen alle oder nur einige der identifizierten Schwachstellen behoben werden? Dies liegt nicht zuletzt an der Kommunikationsfähigkeit des Prüfers.

Versucht man alle Definitionen auf eine möglichst allgemeingültige und prägnante Definition zu verkürzen, so kann man formulieren: „Ein Experte führt für einen Zeitraum X einen Angriff gegen Ziel Y nach Vorgehensmodell Z durch." Letztlich wird das Risikoprofil des Angreifers abgebildet, den man simulieren möchte. Wie viel Zeit wird der Angreifer sich mit einem Ziel auseinandersetzen? Welche Expertise hat der Angreifer?

Es macht einen erheblichen Unterschied, ob man beispielsweise für die Prüfung einer Anwendung nur wenige Stunden zur Verfügung hat oder mehrere Tage. In wenigen Stunden ist selbst bei einfachen Anwendungen keine baldige Prüfaussage möglich. Bei systembasierten Prüfungen hingegen, bei denen also je System auf erreichbare Dienste und bekannte Schwachstellen innerhalb dieser Dienste geprüft wird, ist der manuelle Prüfaufwand vergleichsweise gering. Die geprüften Protokolle und Dienste sind stark automatisiert prüfbar. Dennoch gibt es von den Prüfwerkzeugen erkannte Schwachstellen, die aufgrund falscher Indizien identifiziert wurden, sogenannte „False-Positives". Diese muss der Prüfer selbstverständlich durch manuelle Verifikation der Schwachstellen identifizieren. Darüber hinaus gibt es einige wenige rein manuell durchführbare Prüfungen. Bei der Prüfung von Anwendungen oder auch bei der Prüfung von individuellen Lösungen wie etwa speziellen Produkten mit speziellen Produkteigenschaften ist die Anforderung an die Expertise des Prüfers ungleich höher.

Nutzen von Penetrationstests

Ein Penetrationstest ist im Regelfall keine deterministische Vorgehensweise. Insbesondere, da der Faktor Zeit und die Aufteilung der Zeit eine wichtige Rolle spielen. Meist werden Penetrationstests zum Festpreis durchgeführt. Dabei steht die zur Verfügung stehende Zeit im Vorfeld fest. Welchen Anteil der Projektzeit reine Prüfzeit darstellt, ist im Vorfeld für keine Partei abschließend vorhersehbar. Die Prüfzeit besteht aus den eigentlichen Prüfungen, aber auch der Lösung bei Prüfproblemen. So kommt es in der Prüfpraxis regelmäßig vor, dass Systeme und Anwendungen während der Prüfung verlangsamt antworten oder ganz abstürzen. Dies alles geht auf das Projektkontingent.

Ebenso unklar ist im Vorfeld, wie viel Zeit für die Dokumentation der Ergebnisse notwendig sein wird. Je mehr Schwachstellen während eines Tests gefunden werden, desto mehr Zeit benötigt der Prüfer für die Dokumentation. Dieses Mehr an Zeit für die Dokumentation im Falle vieler Schwachstellen geht von der effektiven Prüfzeit innerhalb des Gesamtkontingents ab. So kann es Fälle geben, bei denen durch Testabbrüche und eine Vielzahl von Anwendungen die angestrebte Prüftiefe in einem Projekt nicht erreicht werden kann. Dies ist jedoch klar zu dokumentieren. In einem solchen Fall ist die nicht erreichte Prüftiefe kein Praxisproblem. Es ist offensichtlich, dass das Prüfobjekt erheblich und grundsätzlich verbessert werden muss.

Da kann man im Regelfall von „Time Boxed"-Prüfungen sprechen. Der Tester hat eine gewisse Zeit für Prüfung und Dokumentation zur Verfügung. Der Nutzen eines Penetrationstests hängt also klar von der Expertise des Testers und der Aufbereitung des Ergebnisses ab.

Etablierte Prüfstandards

Eine grundsätzliche Definition für Penetrationstests stellt das Durchführungskonzept des BSI dar. Für spezifische Prüffälle bietet beispielsweise das Open Web Application Security Project, bekannt durch die zehn häufigsten Risiken in Web-Anwendungen (OWASP Top 10, *siehe „https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project"),* mit dem OWASP Testing Guide *(siehe „https://www.owasp.org/index.php/Testing_Guide")* eine hervorragende Möglichkeit, einen granularen Prüfplan und damit auch Prüfnachweis bei der Prüfung von Web-Anwendungen zu pflegen.

Im Rahmen des Angebots muss der Anbieter darstellen, mit welcher Methodik und welchen Werkzeugen er die jeweiligen Prüfungen durchführen möchte. Für die allermeisten Prüffelder gibt es mittlerweile spezialisierte Werkzeuge. Die Transparenz ist dabei ein hohes Gut. Nur wenn Tests nachvollziehbar sind, entfalten sie einen Mehrwert. Sicherheit und auch Penetrationstests sind kein Voodoo.

Anbieterqualifizierung

Es gibt wenige Möglichkeiten, die Qualifikation eines Anbieters zu prüfen. Ausschreibende Stellen sollten sich vorab einige Mitarbeiterprofile des Anbieters geben lassen. Deren Expertise sollte die Anforderungen abbilden. Im Falle, dass andere Personen als die angebotenen eingesetzt werden, sollte der Ausschreibende sich ein Vetorecht einräumen.

Ein wichtiger Aspekt bei der Qualifikation von Prüfern sind die Berufserfahrung, die Erfahrung in der spezifischen Prüfdomäne und personenbezogene Zertifizierungen. So gibt es zum Beispiel in Deutschland durch das BSI zertifizierte IT-Dienstleister für Penetrationstests *(siehe „https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/Stellen/IS_REV_PEN/IS_REV_Dienstleister/IS_REV_Dienstleister_node.html").* Diese beschäftigen entsprechend durch das BSI geprüfte und zertifizierte Penetrationstester *(siehe „https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/Personenzertifizierung/PEN/personen_auditoren_pentester.html").*

Das BSI als Behörde ist wirtschaftlich unabhängig in seinen Feststellungen und führt selbst Penetrationstests für Bundesbehörden durch. Die Wertigkeit des Zertifikats ist entsprechend hoch. Es gibt weitere personenbezogene, kommerzielle Prüfungen, etwa Offensive Security Certified Professional (OSCP, *siehe „https://www.offensive-security.com/information-security-certifications/oscp-offensive-security-certified-professional")* oder der deutlich theoretische Certified Ethical Hacker (CEH, *siehe „https://www.eccouncil.org/programs/certified-ethical-hacker-ceh").*

Darüber hinaus empfiehlt es sich, Referenzen zu prüfen. Es ist zwar schwer, Referenzen in diesem Segment zu erhalten; doch für Unternehmen, die sich lange genug erfolgreich am Markt positioniert haben, sollten Referenznennungen kein Problem darstellen.

Herausforderung spezieller Prüfungen

Es ist nicht unüblich, dass sehr spezielle Umgebungen überprüft werden müssen. Das können zum Beispiel Automatisierungsumgebungen, Radarnetze oder Produktklassen wie Geldspielgeräte sein. Im Gegensatz zu Standardprüfungen, wie der Prüfung von Webshops, gibt es hier im seltensten Fall eine spezialisierte, standardisierte und etablierte Vorgehensweise. Daher empfiehlt es sich, mit wenigen grundsätzlich qualifizierten Anbietern entsprechende Workshops vorab abzuhalten. Dabei werden durch den Ausschreibenden die Prüfziele vorgestellt. Der Anbieter hat dann die Möglichkeit, spezifische Fragen zu stellen und mögliche Prüfungen aus den Angaben abzuleiten. In diesen Gesprächen ist meist relativ klar erkennbar, welche Anbieter für diese Art von Spezialprüfungen besonders geeignet sind.

Oder doch lieber ein Audit?

Es kommt in der Praxis immer wieder vor, dass Organisationen Penetrationstests planen, jedoch das zu betrachtende Risikoprofil eigentlich eher für ein Audit geeignet ist. Bei einem Audit werden zum Beispiel auf Basis der Grundschutzkataloge des BSI Grundsätze der technischen Implementierung und der Prozesse im Unternehmen abgefragt. Für kleinere Unternehmen ist der Cyber-Sicherheits-Check von ISACA und BSI eine sehr effiziente und kostengünstige Möglichkeit, eine Bewertung der Informationssicherheit festzustellen. Noch dieses Jahr wird ein spezieller Cyber-Sicherheits-Check für Industrie 4.0 veröffentlicht. Es stellt sich also nicht die Frage, ob Penetrationstest oder Audit, sondern, in welcher Reihenfolge dies für eine Organisation sinnvoll ist.

Tobias Glemser
tglemser@secuvera.de

Tobias Glemser führt seit dem Jahr 2000 Sicherheitsüberprüfungen sowie Penetrationstests durch und ist Geschäftsführer der secuvera GmbH. Er ist vom Bundesamt für Sicherheit in der Informationstechnik zertifizierter Penetrationstester und Common Criteria Evaluator. Tobias Glemser ist Autor von Fachartikeln und Referent bei Seminaren und Kongressen. Er hat diverse Security Advisories für selbst gefundene Schwachstellen, etwa in Web-Anwendungen, veröffentlicht. Tobias Glemser ist Chapterlead des German Chapter des Open Web Application Security Project (OWASP).

Microservices? Mit Sicherheit!

Claus Straube, Landeshauptstadt München

In jedem produktiven Software-System spielt Sicherheit eine zentrale Rolle. Warum sollte dies in einer Microservice-Architektur anders sein? Allerdings ist es im Vergleich zu einer monolithisch geprägten Architektur in einem verteilten System deutlich komplexer, die Anwendung adäquat abzusichern. Dieser Artikel diskutiert die Themen „Anwendungsdesign", „Authentication" und „Authorization" sowie die Angriffserkennung aus Architektur-Sicht.

Microservices finden immer mehr Verwendung in produktiven Systemen. Das liegt nicht nur an ihren Vorzügen, wie punktgenaue Skalierbarkeit oder – bedingt durch ihre Größe und lose Kopplung – bessere Wartbarkeit, sondern vor allem daran, dass in den letzten Monaten ein Ökosystem von Werkzeugen aus dem Boden geschossen ist, das die Nachteile einer verteilten Anwendungslandschaft zumindest abmildert. Eine dieser Herausforderungen, die produktive IT-Systeme immer mit sich bringen, ist Security. Daran hat sich auch mit dem Microservice-Hype nichts geändert. Nach wie vor will man als Systembetreiber bestimmen, wer was in der Anwendung tun darf.

Die Architektur

Der erste (und einfachste) Schritt in Richtung einer sicheren Microservice-Anwendung ist es, die Angriffsfläche so klein wie möglich zu halten. Im Vergleich zu einer monolithischen Architektur wächst diese bei einer verteilten Anwendungsarchitektur von Natur aus. Man hat viele unabhängige Services, von denen jeder ein API hat, und diese kleinen Anwendungen kommunizieren auch noch über das Netzwerk miteinander. Das erhöht logischerweise die Fläche. Glücklicherweise lässt sich die Angriffsfläche mit ein paar einfachen Mitteln recht gut verkleinern.

Grundsätzlich sollte die Anwendung in einer sicheren Umgebung ausgebracht sein. Der Zugriff von außerhalb ist also beispielsweise durch eine Firewall limitiert. Natürlich sollten sowohl auf der Firewall als auch auf den Containern beziehungsweise virtuellen Maschinen, auf denen die Services laufen, alle Ports geschlossen sein, die nicht unbedingt benötigt werden. Wenn man sich an die Regel hält, jeden Service in einem eigenen Container zu installieren, wird man normalerweise mit einem offenen Port auskommen. Die Kommunikation zwischen den Services beziehungsweise zwischen den Maschinen sollte über „https" verschlüsselt sein. Für den Entwickler ist das dank Werkzeugen wie Spring Boot oder Kubernetes kein großer Aufwand mehr. Die größte Herausforderung liegt tatsächlich in der Verwaltung der Zertifikate.

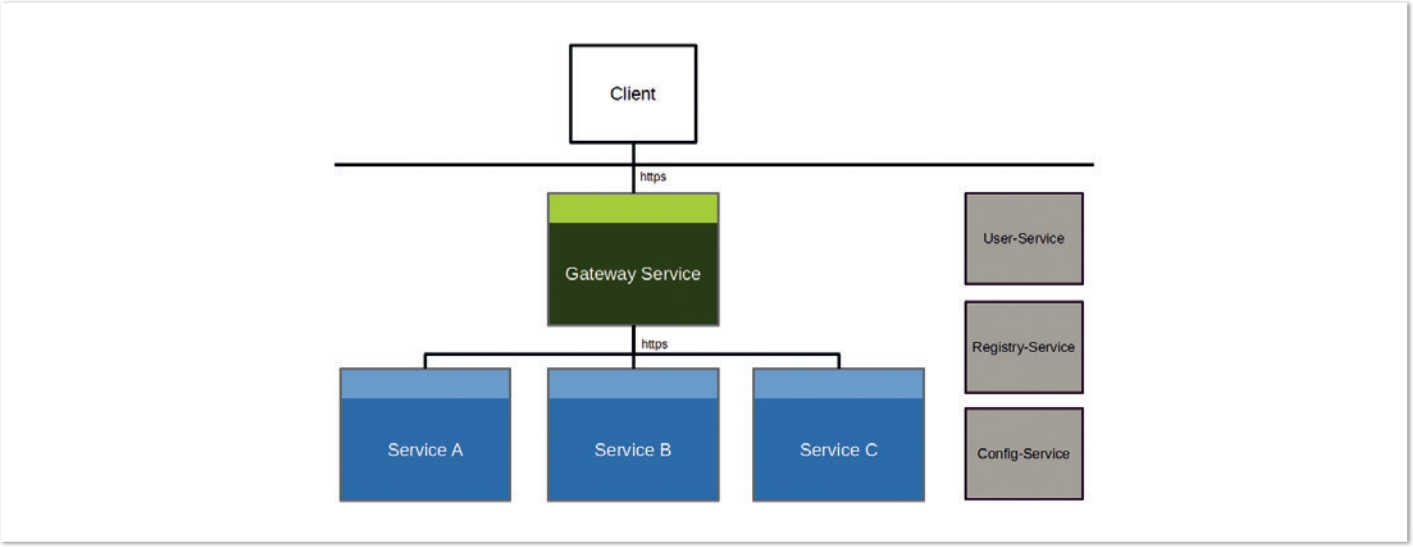


Abbildung 1: Microservice-Architektur mit API-Gateway

Ein weiterer wichtiger Aspekt ist die Verschattung der Schnittstellen. In der Regel wird die Gesamtheit aller Services mehr Schnittstellen zur Verfügung stellen, als von außen verwendet werden. Dies kann zum Beispiel daran liegen, dass bestimmte Operationen ausschließlich von anderen (internen) Services aufgerufen werden. Um auch hier die Angriffsfläche möglichst klein zu halten, ist es notwendig, die öffentlich verfügbaren Schnittstellen an einer zentralen Stelle zu bündeln.

Ein zusätzlicher Effekt durch diese Bündelung ist, dass von außen keine Rückschlüsse auf das Deployment gemacht werden können. Der Client muss nicht „n" Service-Adressen aufrufen, sondern genau eine. Um dieses API-Gateway-Pattern *(siehe „http://microservices.io/patterns/apigateway.html")* anzuwenden, gibt es bei Spring Cloud Netflix *(siehe „https://cloud.spring.io/spring-cloud-netflix")* den Zuul-Service. Dieser ist selbst ein Spring-Boot-Service und wird mit der Annotation „@EnableZuulProxy" in Verbindung mit den entsprechenden Abhängigkeiten im Klassenpfad erstellt.

Im einfachsten Fall kann der Zugriff auf die anderen Services durch URL-Pattern-basierte Routes konfiguriert werden. Wenn man den vollen Funktionsumfang von Zuul nutzen will, dann hat man auch die Möglichkeit, auf die Routen programmatisch zuzugreifen *(siehe Abbildung 1)*.

Bei sehr hohen Anforderungen an die Sicherheit können auch zwei API-Gateways zum Einsatz kommen. Das erste bündelt den Zugriff auf die öffentlichen Services, das zweite den auf die privaten Services. Bei dieser Architektur wird man aber gegebenenfalls beim fachlichen Design der Microservices Zugeständnisse an das Deployment machen müssen. In der ersten Zone werden also zwangsläufig vor allem Services zur Orchestrierung platziert werden. Von einem Schnitt nach der reinen Domain-Driven-Design-Lehre wird man in diesem Fall etwas abweichen müssen. Wichtig ist, dass in beiden Zonen jeweils auch eine separate Authentication verwendet wird. Das heißt, jede Zone hat ihre eigene, übergeordnete Security.

Der Security-Context

In einer monolithischen Anwendungs-Architektur besteht die Anwendung aus einem einzigen Sicherheitskontext. In diesem spielen sich praktisch alle sicherheitsrelevanten Vorgänge ab. Dort werden die Rollen und Rechte verwaltet (sofern man nicht mit einem Identity-Management-System arbeitet) und an jeder Stelle im Verfahren ist klar, wer eine Aktion gerade ausführt.

Dies ist in einer Microservice-Architektur fundamental anders. Hier ist jeder Service autark. Er kümmert sich nach dem „Do One Thing and Do it Well"-Prinzip um seine Aufgabe. Natürlich sind (beziehungsweise sollten) die einzelnen Services abgesichert sein, aber eben jeder für sich. Ein großer gemeinsamer Security-Context wie in einer monolithischen Architektur kommt so nicht zustande. Er sollte vielleicht auch gar nicht so existieren, weil ein Service unter Umständen von mehr als einer Anwendungsdomäne genutzt werden könnte – also gar nicht, wie in der klassischen monolithischen Denkweise, zu einer einzelnen Anwendung gehört. Dieser dezentrale Ansatz macht Security in einer Microservice-Architektur zur Herausforderung.

Authorization

Wo werden die Nutzer, Rollen und Rechte verwaltet? Wenn man sich überlegt, welchen Aufwand es nach sich ziehen würde, die Zuordnungen „Nutzer zu Rolle" und „Rolle zu Recht" in vielleicht fünf unterschiedlichen Services zu pflegen, und dann den Gedanken weiter spinnt, wie das mit hundert Services aussehen würde, kommt man schnell zum Schluss, dass es hierfür eine zentrale Stelle geben muss: einen User-Service nach dem Motto „Do One Thing and Do it Well", der rein für das Mapping „Nutzer auf Rolle" und „Rolle auf Recht" zuständig ist.

Woher der User-Service die Nutzer- und Rollen-Daten bezieht, ist hierbei zweitrangig. Im einfachsten Fall sind die Daten im Speicher vorgehalten oder sie kommen aus einem Lightweight Directory Access Protocol (LDAP) oder Identity-Management. Selbst Mischlösungen sind vorstellbar. Das heißt, Nutzer werden in einem LDAP verwaltet, die Rollen und das Mapping in einer Datenbank des Service. Hier kann man sämtliche Freiheiten, die einem Frameworks wie Spring Security bieten, nutzen, um die Anforderungen bestmöglich abzudecken. Die Rechte kommen jedoch immer aus den einzelnen Microservices, denn das Recht ist dort mit einem Stück Code verbunden, der nur ausgeführt wird, wenn der Nutzer das Recht

besitzt. Damit ist klar, dass es wichtig zu wissen ist, welche Person (oder gegebenenfalls Maschine) hinter einem Aufruf steckt. Nur so können die Rechte im Service aufgelöst werden.

Oftmals wird ein Request nicht von genau einem Service abgearbeitet, sondern er löst eine Kaskade von Anfragen an unterschiedliche Services aus. Die Kommunikation hinter dem ersten Service könnte natürlich durch technische User durchgeführt werden. Ein detailliertes Berechtigungssystem auf allen Service-Ebenen ist dann allerdings nicht mehr möglich. Besser wäre es, wenn die Identität des aufrufenden Nutzers bei jedem Aufruf mitgegeben würde.

Authentication

Hier stellt sich die Frage, wie man die Rechte an den Aufrufer binden kann und – mindestens genauso spannend – wie diese Daten von Service zu Service transportiert werden. Eine Möglichkeit, dies einfach in einer verteilten Service-Wolke einzubauen, bietet hier OAuth2. Zwar ist OAuth2 („the industry-standard protocol for authorization", *siehe „https://oauth.net/2"*) ein Authorization- und kein Authentication-Protokoll – aber es erzeugt für einen Benutzer einen eindeutigen Token. Diesen bekommt der Nutzer nur, wenn er sich vorher auch authentifiziert hat. Der Token kann bei „http"-Aufrufen im Nachrichten-Header weitergegeben werden. Dadurch lässt sich in einer verteilten Anwendung sehr einfach ein Authentication-Mechanismus aufbauen, der die zuvor festgelegten Anforderungen perfekt erfüllt.

Wer sich im Spring-Ökosystem bewegt, kann sich über die Unterstützung freuen, die Spring Cloud Security *(siehe „https://cloud.spring.io/spring-cloud-security")* von Hause aus mitbringt. Mit wenigen Annotationen ist hier ein OAuth-Server (dieser kann natürlich auch von jedem Nicht-Spring-Service genutzt werden) auf Basis einer Spring-Boot-Anwendung erstellt.

Genauso schnell sind die von den Microservices angebotenen Endpunkte als OAuth-Ressourcen mit „@EnableResourceServer" gekennzeichnet. Dadurch wird der gesamte Service automatisch abgesichert. Um in den einzelnen Services nun automatisch bei Aufruf einen Security-Context aufbauen zu können, muss dem OAuth-Server noch ein „User Info Endpoint" hinzugefügt werden. Auf Seiten der Microservices ist dieser User-Info-Endpoint noch mit „security.oauth2.user-info-uri" zu konfigurieren, damit er gefunden werden kann. Wie bei Spring üblich, funktionieren die Dinge mit einer Basiskonfiguration einfach. So auch hier: Bei kaskadierenden Service-Aufrufen sorgt Spring dafür, dass der Token über ein „OAuth2Rest"-Template automatisch weiter propagiert wird *(siehe Abbildung 2)*.

Ein Nachteil dieser Methode ist allerdings, dass jeder Service-Aufruf gleichzeitig einen Request gegen den User- beziehungsweise OAuth-Service erzeugt. Das ist aus zwei Gründen kritisch:

- Die vielen Requests sind natürlich eine Belastung für Netzwerk und User-Service. Je mehr Aufrufe erfolgen und je mehr Services von diesen betroffen sind, desto höher wird die Belastung.

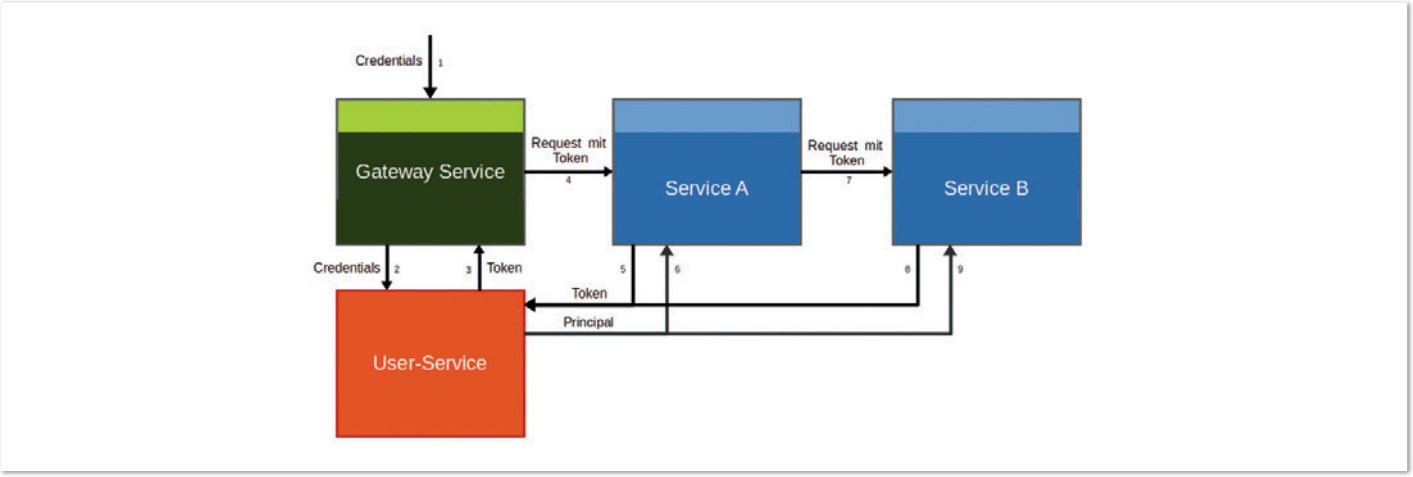


Abbildung 2: Kommunikation mit User-Endpoint pro Request

- Der User-Service ist eine absolut kritische Ressource; ist diese nicht vorhanden, wird potenziell nichts funktionieren.

Die Probleme lassen sich natürlich durch die eine oder andere Maßnahme, wie beispielsweise ein Cache auf Seiten der Ressourcen-Server, abmildern. Ganz eliminieren kann man sie allerdings nicht. Es gibt noch eine weitere Möglichkeit, Autorisierung mit OAuth2 umzusetzen. Die Spezifikation lässt dem Nutzer recht viele Freiheiten bei der Wahl des Tokens. Entsprechend viele Geschmacksrichtungen gibt es hier auch. Die beschriebene Methode funktioniert mit einem einfachen Bearer-Token *(siehe „https://tools.ietf.org/html/rfc6750").* Dieser besteht aus einer zufälligen Zeichenfolge und hat sonst keine semantische Bedeutung.

Es gibt allerdings auch sogenannte „strukturierte Token" wie den JSON-Web-Token (JWT, *siehe „https://tools.ietf.org/html/rfc7519").* Er besteht immer aus drei Teilen, einem Header, der Payload und der Signatur, wobei die drei Teile jeweils durch einen Punkt getrennt sind. Wird der JWT verschickt, dann werden Header und Payload per „Base64" encodiert und der gesamte Token per „HMAC" oder „RSA" signiert. Die Signatur stellt Integrität und Authentizität des Tokens sicher, ohne nochmals beim Server anfragen zu müssen. Interessant am JWT ist, dass man beliebige Dinge in der Payload mitschicken kann, beispielsweise auch Informationen über den Nutzer.

Genau das passiert, wenn man die JWT-Funktionalität von Spring Cloud Security nutzt. Im User-Service wird automatisch ein JWT mit allen notwendigen Informationen zum eingeloggten Nutzer erzeugt. Neben dem Namen des Users werden dort beispielsweise alle „Authorities" gespeichert, also die Rechte, die in der Methoden- oder Klassen-Security verwendet werden, die dem Nutzer zugeordnet werden können. Verwendet man Spring auch auf Seite der OAuth-Ressourcen, dann wird der lokale Security-Context des Service automatisch, auf Basis der Informationen, die im JWT gespeichert wurden, erzeugt. Ein Request Richtung User-Service ist an dieser Stelle nicht mehr notwendig.

Es gibt allerdings eine kleine Einschränkung: In einem JWT können zwar prinzipiell beliebig viele Daten gespeichert werden – da der Token aber im „http"-Header transportiert wird, muss man sich hier den Rahmenbedingungen des empfangenden Servers beugen. Aktuell können die meisten Server standardmäßig um die sieben bis acht KB verarbeiten. Ob das reicht, hängt sehr stark von der Größe der Anwendung und noch viel mehr vom Rollen- und Rechte-Konzept ab.

Grundsätzlich sollte man in der IT versuchen, volatile Dinge möglichst aus dem Code herauszuhalten. Die Zuordnung von Rollen auf Rechte oder von Nutzern auf Rollen ist so etwas Flüchtiges. Um trotzdem möglichst viel Handlungsfreiheit zu haben, kann man an der Granularität der Rechte ansetzen. Diese können beispielsweise auf Klassen- oder sogar auf Methoden-Ebene vergeben werden. Je weniger Code von einem Recht abgesichert wird, desto größer sind später die Freiheitsgrade bei der Konfiguration der Rollen.

Bei Spring heißen diese Rechte „Authorities". Das sind genau die Daten, die über den JWT geliefert werden müssen, um den Security-Context zu erzeugen. Hier zeigt sich das Problem. Hat man in der Anwendung Rechte mit sprechenden Namen wie „contractservice__calculate_insurance_risk" vergeben, was ja durchaus hilfreich bei der Zuordnung zu Rollen ist, dann kann man sich leicht ausrechnen, dass bei 150 bis 250 Rechten je Nutzer ein Problem auftritt. Diese Zahl ist bei kaskadierenden Service-Aufrufen in Verbindung mit einem feingranularen Rechte-Konzept schnell erreicht.Lösen lässt sich dieses Problem, indem man statt Rechten Rollen überträgt. Ein Nutzer wird in der Regel nur wenige Rollen haben. Die Wahrscheinlichkeit, dass man mit dieser Methode auf technische Limitationen stößt, ist relativ gering. Allerdings verlässt man damit bis zu einem gewissen Grade die Infrastruktur, die einem Spring per Konvention bietet. Die Rollen in den Token zu bekommen, ist dabei noch nicht das Problem. Das lässt sich auf Seite des User-Service durch eine einfache Query (sofern eine Datenbank genutzt wird) in einer „AuthenticationConfigurer"-Klasse lösen, die Rollen statt Rechte ausliest.

Im User-Service muss zusätzlich ein Endpunkt geschaffen werden, über den das „Rollen zu Rechte"-Mapping abgerufen werden kann. Auf Service-Seite sind allerdings mehr Dinge anzupassen. Einmal müssen die „Rollen zu Rechte"-Mappings vom User-Service in die Services übertragen und dann im Speicher vorgehalten werden. Dies kann per „Push" immer dann passieren, wenn Änderungen am Mapping vorgenommen werden. Alternativ könnten die Services die Informationen periodisch per „Pull" vom User-Service abrufen. Zudem ist in der Security-Pipeline dafür zu sorgen, dass die im Token mitgelieferten Rollen beim Aufbau des Security-Context in die richtigen Rechte umgewandelt werden *(siehe Abbildung 3).* Hat man diese Bedingun-

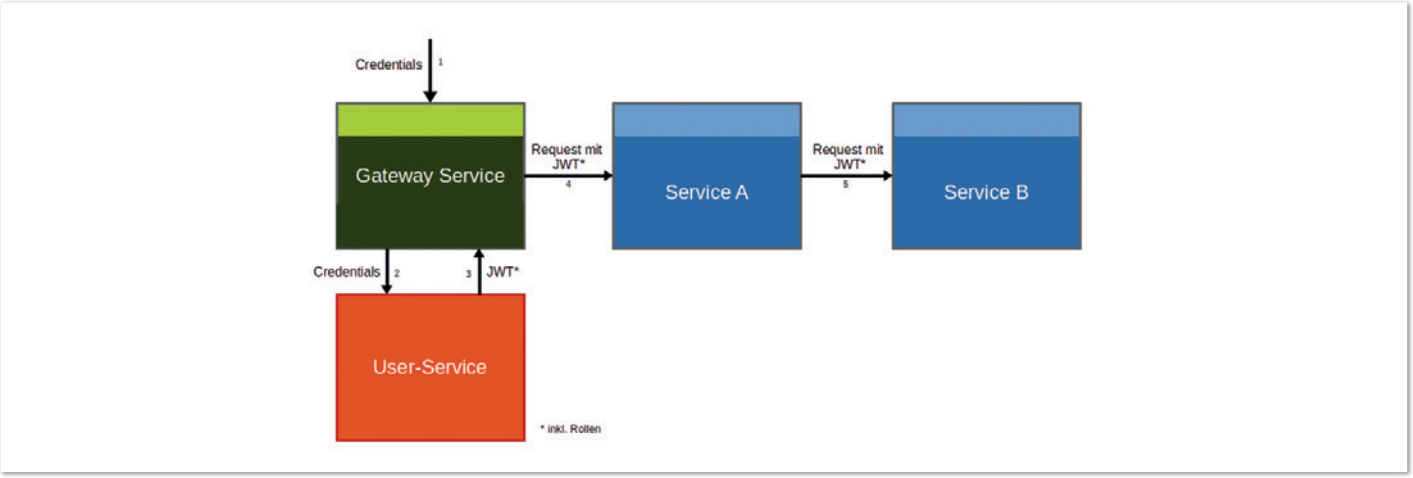


Abbildung 3: Kommunikation mit JWT pro Request

gen erfüllt, kann sich der Nutzer nach der ersten Anmeldung frei im System bewegen, ohne auf einen User-Service angewiesen zu sein.

Angriffserkennung

Neben den beschriebenen, passiven Vorkehrungen wäre es natürlich auch interessant, Angriffe erkennen und aktiv Gegenmaßnahmen ergreifen zu können. Im einfachsten Fall könnte man beispielsweise einen verdächtigen User sperren. Doch wie erkennt man, dass sich ein Nutzer verdächtig verhält? Dafür gibt es „Intrusion Detection"-Systeme wie AppSensor *(siehe „http://appsensor.org")* vom Open Web Application Security Project (OWASP, *siehe „https://www.owasp.org")*. Diese Systeme sammeln aus unterschiedlichsten Quellen Informationen und fügen sie zu einem „Lagebild" zusammen. Auf dessen Basis müssen dann die entsprechenden Maßnahmen zur Abwehr von Angreifern abgeleitet werden.

AppSensor bietet hier nicht nur die Software, sondern auch ein konzeptuelles Basis-Framework mit einer Beschreibung typischer „Detection Points". Das sind Ereignisse, die in einer Web-Anwendung auftreten können und dabei helfen, die Lage einzuschätzen. Auf Entwurfsebene unterscheiden sich die Detection Points einer monolithischen und einer verteilten Anwendung wenig. Auch die Frage, wie man an die gewünschte Information kommt, ist relativ ähnlich. Verwendet man in beiden Architektur-Varianten beispielsweise das Spring-Framework, so wird man viele Daten aus den zahlreichen Spring-Events ziehen können. Der größte Unterschied liegt wohl im Transport der Nachrichten von dem Punkt, am dem ein Ereignis aufgetreten ist, zum zentralen Sensor-Service. In einem monolithischen System können die Ereignisse in der Anwendung gesammelt und zum ID-System geleitet werden. Bewegt man sich in einer verteilten Anwendung, so sind die sicherheitsrelevanten Events in voneinander unabhängigen Klein-Anwendungen verteilt. Jeder dieser Services muss also selbst dafür sorgen, dass sicherheitsrelevante Nachrichten beim Sensor-Service ankommen.

Wie so oft gibt es natürlich für dieses Problem mehrere Lösungen. Verwendet man ein „Security Information and Event Management"-System (SIEM), so hat man hier oft die Möglichkeit, Logfiles aus unterschiedlichen Quellen einlesen und zentral auswerten zu lassen. Einen leichtgewichtigen Ansatz bietet Spring Cloud Stream *(siehe „https://cloud.spring.io/spring-cloud-stream")* – vorausgesetzt, man bewegt sich im Spring-Ökosystem.

Die Idee hinter Spring-Cloud-Stream ist, auf Basis einer Messaging Middleware (etwa RabbitMQ, *siehe „https://www.rabbitmq.com")* Annotationen anzubieten, mit denen Event-Nachrichten in einer verteilten Anwendung versendet werden können, ohne dass Sender und Empfänger voneinander wissen müssen. Der Sender versendet also ein Event und ein an den Bus angeschlossener Empfänger nimmt sich die Nachrichten, die interessieren.

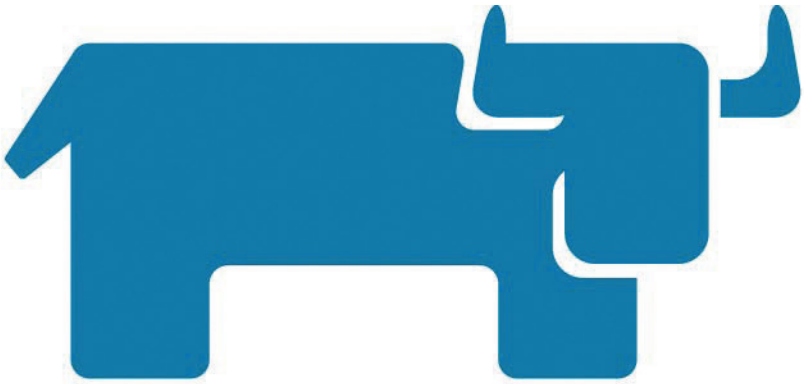
Im Falle von ID würden die Services ihre Detection-Point-Events verschicken, der Sensor-Service würde sie aufnehmen. Der große Vorteil einer Bus- gegenüber einer Logfile-Lösung ist, dass sich die Ergebnisse praktisch in Echtzeit auswerten lassen. Damit kann auch ohne zeitliche Verzögerung automatisiert auf einen Angriff reagiert werden.

Fazit

Wie an den drei Beispielen gezeigt, hat der Microservice-Trend nichts an den grundsätzlichen Security-Design-Prinzipien geändert – sehr wohl aber daran, wie die Dinge umgesetzt werden müssen. Das muss allerdings nicht immer auch viel aufwändiger sein; mit den richtigen Werkzeugen lassen sich hier sehr smarte und alltagstaugliche Lösungen erzielen.

Claus Straube
claus.straube@gmail.com

Claus Straube ist seit dem Jahr 2011 als IT-Architekt für Java-Architektur und Anwendungs-Integration beim internen IT Dienstleister (itM) der Landeshauptstadt München (LHM) angestellt. Zusätzlich berät er freiberuflich Organisationen in Architekturfragen – insbesondere bei Integrationsthemen. Vor seiner Zeit bei der LHM war er als Entwickler und Architekt bei Oracle, Kabel Deutschland und einem Startup beschäftigt.



RANCHER

Docker-Application-Stacks mit Rancher

Patrick Busch, Polyas GmbH

Bei Rancher handelt es sich um eine Container-Management-Plattform, die darauf abzielt, das Arbeiten mit Docker-Containern so einfach wie möglich zu gestalten. Im Vergleich zu Kubernetes, Mesos etc. erfindet Rancher das Rad nicht neu – macht aber einiges anders. Dieser Artikel zeigt, wie schnell und einfach eine Rancher-Installation aufgesetzt ist und wie das Deployment eigener Services funktioniert.

Voraussetzung für einen erfolgreichen Einsatz ist lediglich ein laufender Docker-Daemon auf dem Host-System. Für die Absicherung des Systems ist der Nutzer selbst verantwortlich. Als Basis-Installation in den Beispielen dient ein aktuelles Debian Jessie, auf dem der Docker-Daemon installiert wurde.

Grund-Architektur

Rancher setzt einen Master-Slave-Mechanismus ein. Der Master, auch Server genannt, der zuerst eingerichtet werden muss, ist die Steuerzentrale für alle anderen Hosts. Auf dem Server kann das Rancher-UI bedient werden und er stellt das API zur Verfügung, um programmatisch alle Funktionen bedienen zu können. Die Hosts, auf denen die Container später laufen, sind in Environments unterteilt.

Abbildung 1: Startbildschirm nach erfolgreicher Installation

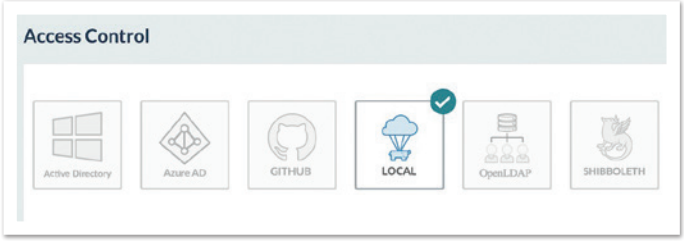


Abbildung 2: Die Möglichkeiten zur Nutzerverwaltung

Ein Environment kann beliebig viele Hosts beinhalten, wobei zwischen den Hosts innerhalb eines Environments ein IPSec-Netzwerk aufgebaut ist. Bestehen mehrere Environments, dann kennen sich diese untereinander nicht. Über das UI ist hier eine feingranulare Rechtevergabe an verschiedene Nutzer möglich.

Erstinstallation

Der Rancher-Server selbst ist ein Docker-Image, das in der einfachsten Variante ohne Konfiguration mit „docker run -d --restart=unless-stopped -p 8080:8080 rancher/server" gestartet werden kann. Ohne weitere Konfiguration wird hier eine im Container laufende Datenbank verwendet. Für einen produktiven Einsatz wird empfohlen, eine extern laufende Datenbank zu nutzen. Der Rancher-Server ist HA-fähig; die Konfigurationsanleitung kann direkt über das Frontend abgerufen werden.

Nach erfolgreichem Start ist Rancher über den Browser erreichbar. Es empfiehlt sich, die Installation nur über Reverse Proxy nach außen verfügbar zu machen. Nach erfolgreicher Einrichtung erscheint die in *Abbildung 1* gezeigte Webseite.

Erster Schritt im Server ist, die Installation gegen unbefugten Zugriff abzusichern. Über den Menüpunkt „Access Control" unterhalb von „Admin" stehen verschiedene Möglichkeiten zur Verfügung *(siehe Abbildung 2)*. Im Beispiel aktivieren wir die lokale Nutzerverwaltung. Dazu ist ein Admin-Account erforderlich. Anschließend sollte das in *Abbildung 3* dargestellte Login-Formular sichtbar sein.

Aufbau eines Environments

Bisher gibt es nur einen Master-Server, aber noch keine Environments, auf denen wir tatsächlich Container starten können. Ein Environment ist eine Gruppe von Hosts, die durch ein IPSec-Netzwerk miteinander verbunden sind. Das Default-Environment, das beim Start des Servers angelegt wird, basiert auf Cattle. Dahinter verbirgt sich der Rancher-eigene Unterbau zur Orchestrierung; er ist vergleichbar mit Kubernetes, Mesos oder Swarm. Diese Alternativen sind in Rancher auch als Unterbau für die Orchestrierung möglich; die Auswahl ist beim Neuanlegen eines Environments zu treffen. *Abbildung 4* zeigt die derzeit möglichen Environment-Templates.

Bei der Auswahl von Mesos oder Kubernetes als Unterbau für ein Environment kümmert sich Rancher um das Deployment der jeweiligen Container zum Management des Clusters. Ebenso kann der Zugriff über die jeweiligen UIs/CLIs erfolgen. Im Artikel bewegen wir uns im bei der Installation angelegten Default-Environment, also auf Cattle.

Ein erster Host kann im Frontend-Menüpunkt „Infrastructure" über den Unterpunkt „Hosts" hinzugefügt werden. Über den entsprechenden Button wird der Prozess gestartet. Vor dem Hinzufügen des ersten Hosts ist die Host-Registration-URL anzugeben *(siehe Abbildung 5)*. Die Host-Registration-URL muss von jedem Host aus erreichbar sein und auf das API des Servers leiten. Per Default ist dies die URL, über die auch das UI erreichbar ist. Nach der einmaligen Eingabe der Host-Registration-URL lässt sich über das Formular der neue Host konfigurieren *(siehe Abbildung 6)*.

Der Screenshot zeigt die Konfiguration für einen selbst aufgesetzten Host. Bei den zur Verfügung stehenden Cloud-Anbietern gestal-

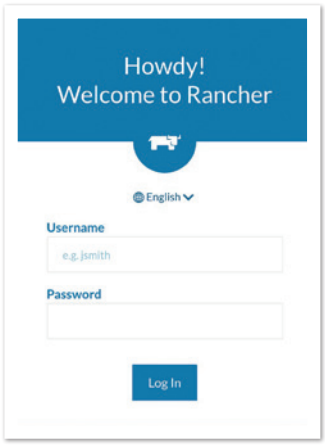


Abbildung 3: Das Login-Formular

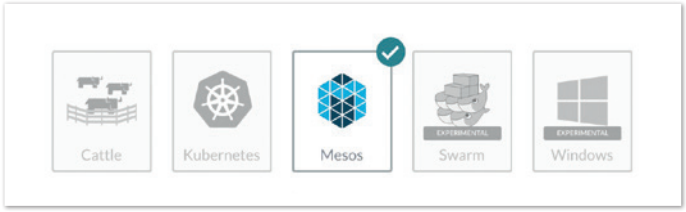


Abbildung 4: Die Environment-Templates

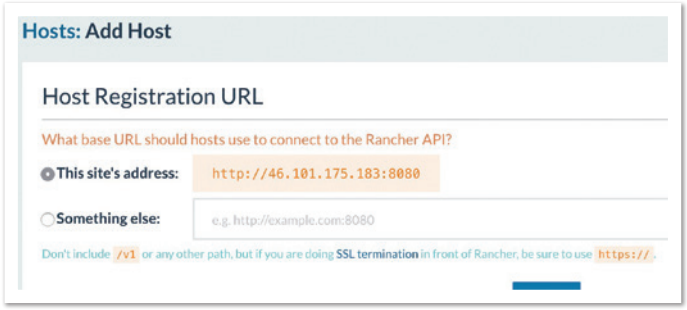


Abbildung 5: Die Eingabe der Host-Registration-URL

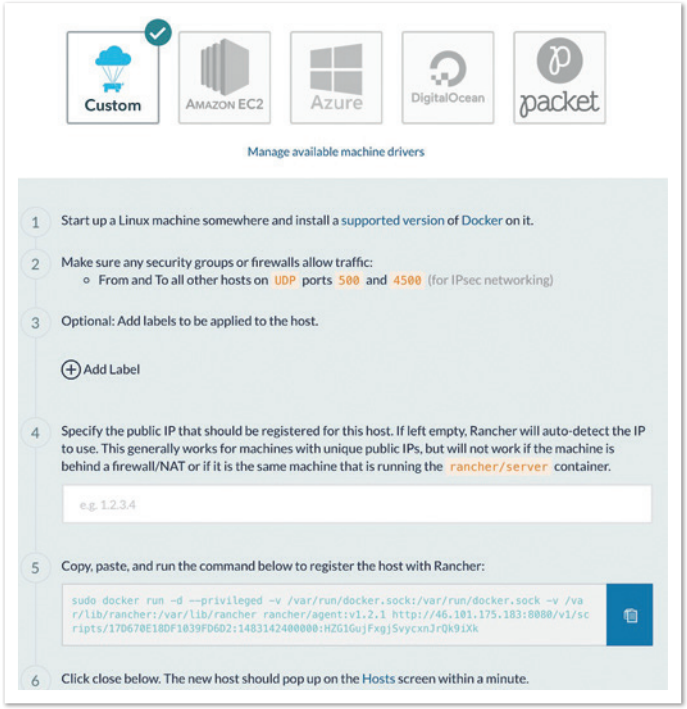


Abbildung 6: Die Konfiguration eines neuen Hosts

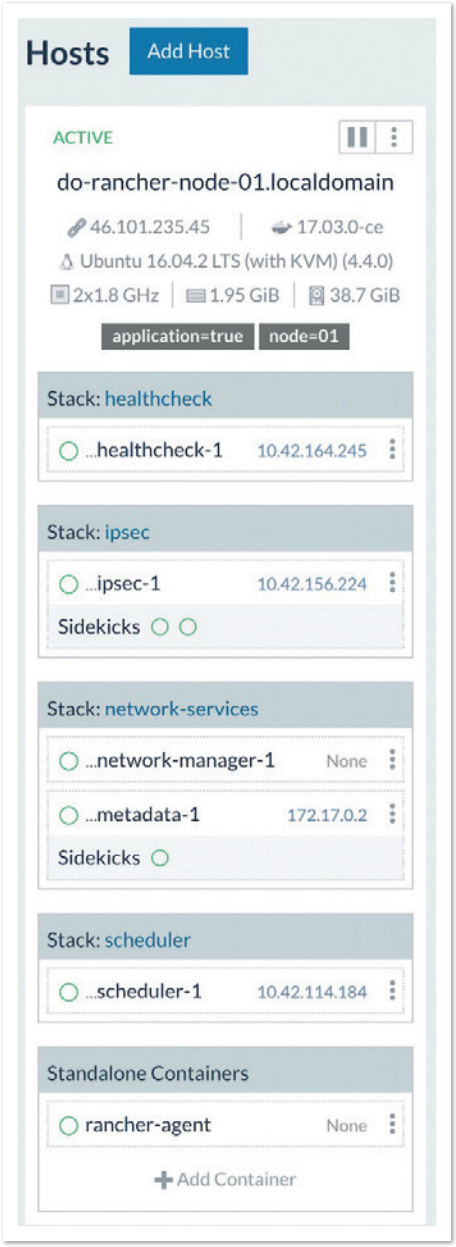


Abbildung 7: Der laufende Host

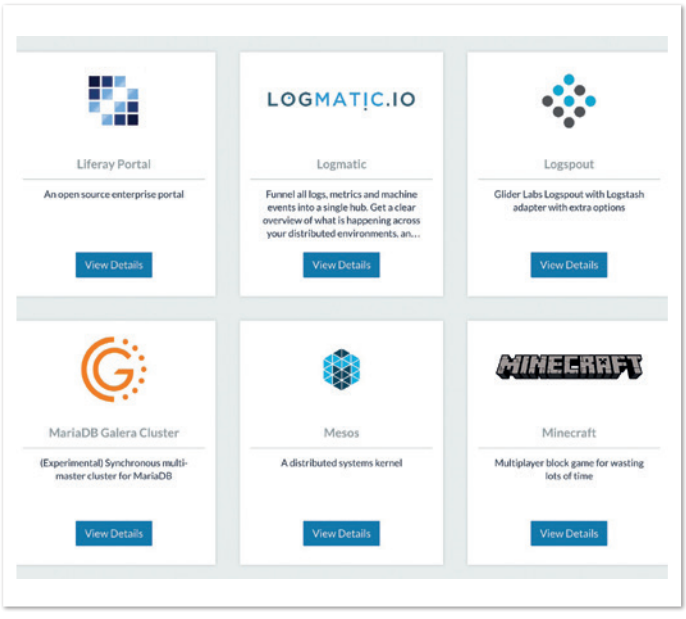


Abbildung 8: Beispiele für Applikationen im Rancher Catalog

tet sich die Konfiguration entsprechend anders. Bei DigitalOcean genügt es beispielsweise, den entsprechenden API-Key anzugeben, um dann die Hosts dort direkt provisionieren zu können.

Dieses Formular hilft beim manuellen Aufsetzen dabei, den Host zu konfigurieren. Voraussetzung ist, wie beim Master-Server, ein Linux-Host mit installiertem Docker-Daemon. Das Hinzufügen eines neuen Hosts ist so einfach wie der Start des Master-Servers; mit einem kleinen Unterschied. Das Formular hilft lediglich, den entsprechenden Docker-Befehl zusammenzustellen, der auf dem neuen Host ausgeführt wird *(siehe Listing 1)*.

Nach dem Ausführen des Befehls sollte der neue Host innerhalb kurzer Zeit im Server erscheinen *(siehe Abbildung 7)*. Das Formular stellt neben zusätzlichen Informationen, unter anderem zu Ports, die Möglichkeit bereit, Hosts mit Labels zu versehen. Über diese Labels lässt sich später das Deployment-Verhalten von Containern bestimmen. Die einfachste Variante lautet, die Hosts durchzunummerieren, wobei die empfohlene Variante lautet, den Hosts fachlich passende Labels zu geben. Es ist beispielsweise denkbar, ein heterogenes Environment mit Hosts, die auf starke I/O-Last ausgelegt sind, und Hosts, die lediglich Datenhaltung betreiben, aufzubauen. Durch entsprechende Labels auf den Hosts kann den Containern dann beim Starten vorgegeben werden, auf welchen Hosts mit dazugehörigen Labels sie starten dürfen. Für das weitere Vorgehen in diesem Artikel wurde ein Environment von drei Hosts aufgebaut.

Rancher Catalog

Der Rancher Catalog ist ein von der Community gepflegter Katalog aus fertig deploybaren Applikationen und die einfachste Möglichkeit, um schnell zu Ergebnissen zu kommen. Eine Applikation heißt Rancher-intern „Stack", wobei ein Stack aus verschiedenen Containern besteht. Die Definition eines Stacks wird als „docker-compose.yml" vorgegeben. Zusätzlich besitzt Rancher ein eigenes „rancher-compose.yml"-Format. Letzteres enthält lediglich zusätzliche Informationen, die beispielsweise die Skalierung der im „docker-compose.yml" definierten Container oder deren Health-Management betreffen. Der Rancher Catalog selbst ist eine auf GitHub gepflegte Ansammlung solcher Service-Definitionen verschiedenster Ausprägungen.

Abbildung 8 zeigt, wie beispielsweise ein MariaDB-Galera-Cluster vollständig und auf Knopfdruck über den Catalog gestartet werden kann. Beim Auswählen der Option erscheinen hierzu applikationsspezifische Konfigurationsmöglichkeiten. Als Beispielapplikation starten wir nun „Rocket.Chat". Dazu suchen wir den entsprechenden Service im Rancher Catalog, wählen eine Version aus und starten ihn. *Abbildung 9* zeigt den darauf startenden Vorgang: Das Deployment und Starten der einzelnen Container. Da in der Konfiguration keine Regeln für das Deployment nach Host-Labels hinterlegt sind, werden die Hosts zufällig ausgewählt. *Abbildung 10* zeigt die vollständig laufende Applikation. Durch einen Klick auf den blau hinterlegten Port können wir direkt zur laufenden Applikation springen; allerdings nur auf diesem einen Host und vorausgesetzt, der Port ist nicht durch die Firewall geblockt.

Load Balancing und SSL

Rancher bietet fertige Services zum Load Balancing an, die auch SSL terminieren können. Sie heißen im UI „Load Balancer". Unter der Haube steckt ein HAProxy-Server. Dementsprechend können über die

Defaults hinausgehende Einstellungen über „HAProxy.cfg"-Dateien vorgenommen werden. Um damit SSL terminieren zu können, müssen zuerst die gewünschten SSL-Zertifikate eingepflegt werden.

Über die Menüpunkte „Infrastructure" und „Certificates" lassen sich einzelne Zertifikate hinzufügen. Für jedes Zertifikat können hier ein Name und eine Beschreibung hinterlegt sein. Außerdem sind mindestens ein Zertifikat und ein Private Key anzugeben. Der Private Key ist nach dem Abschicken des Formulars nicht mehr einsehbar. Um Änderungen an der Zertifikatseinstellung vorzunehmen, muss er daher jedes Mal neu angegeben werden.

Um das Zertifikat zu verwenden, muss ein öffentlich erreichbarer Load Balancer definiert sein. Hierzu sollten ein neuer Stack angelegt und die Konfigurationsoptionen leer gelassen werden. Innerhalb dieses Stacks wird dann über den blauen Dropdown-Button ein Load Balancer angelegt.

Abbildung 11 zeigt, wie ein Load Balancer per Default auf allen Hosts eines Environments ausgerollt wird. Damit die Lastverteilung sauber funktioniert, sollten die Requests von außen auch gleichverteilt auf allen Hosts ankommen. Erreicht werden kann dies beispielsweise durch ein DNS-Round-Robin-Verfahren, bei dem eine Domain auf mehrere IP-Adressen zeigt. Als Best Practice hat es sich bewährt, für jedes Environment solche Domains einzurichten und die Domains, über die die Services erreicht werden sollen, als CNAME auf dieser Environment-spezifischen Domain einzurichten. Das hält den Pflegeaufwand beim Hinzufügen neuer Hosts in Grenzen, da nur eine Domain bearbeitet werden muss.

In der Konfiguration des Load Balancer sollte der Access auf „public" gesetzt sein. Als Protokoll ist HTTPS auf jeden Fall vorzuziehen; dafür muss aber bereits ein entsprechendes SSL-Zertifikat eingerichtet worden sein. Dieses kann im unteren Abschnitt des Formulars konfiguriert werden. „Request Host" sollte die Domain sein, über die der Service erreichbar sein soll. Die Domain sollte darüber hinaus auch mit dem SSL-Zertifikat übereinstimmen. Der Port wird entsprechend zum Protokoll angegeben. Als Target ist der Service definiert, an den die Requests weitergeleitet werden sollen, und als Port gilt der, auf den der Service lauscht. In unserem Fall sind das „rocketchat" und „3000". Dabei ist es unerheblich, ob der Port, auf den weitergeleitet wird, auf eine öffentliche Adresse hört, da wir hier in einem komplett internen IPSec-Netzwerk sind. „Rocket.Chat" ist von hier an unter der eingegebenen Domain und dem konfigurierten Protokoll erreichbar.

Für den Fall, dass wir SSL konfiguriert haben und auch nur darüber erreichbar sein wollen, bietet es sich an, einen Service einzurichten, der alle Requests von Port 80 auf Port 443 weiterleitet *(siehe Listing 2)*. In *Abbildung 12* ist dafür eine beispielhafte Konfiguration dargestellt. Es handelt sich hierbei um einen NginX-Proxy, der über die untenstehende Konfiguration alle einkommenden Requests auf Port 443 umleitet.

Wichtig in der Konfiguration ist, dass auf jedem Host eine Instanz davon läuft, wenn DNS-Round-Robin für das Environment eingerichtet ist. Das vorkonfigurierte Image heißt „pbusch/https-proxy:1.0". Hier wird über die Port-Map der öffentliche Port 80 auf den Container-Port 80 weitergeleitet. Dafür muss Port 80 auf allen Maschinen offen sein. Nach dem Start des Service ist „Rocket.Chat"

```
docker run -d --privileged -v
/var/run/docker.sock:/var/run/docker.sock -v
/var/lib/rancher:/var/lib/rancher rancher/agent:v1.2.1
https://your-host-registration-
url/v1/scripts/4C321E7DD75BAC37052A:1483142400000:OIqF
s2R66XKj36McyFdr66io9Fo
```

Listing 1

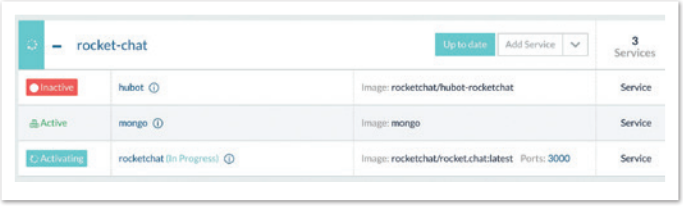


Abbildung 9: Startvorgang eines Applikations-Stacks

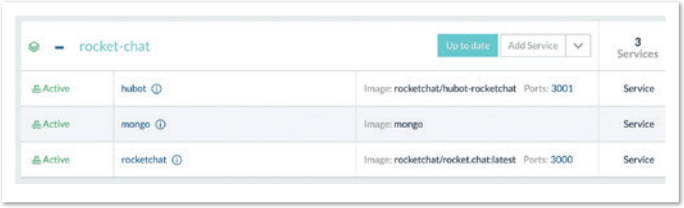


Abbildung 10: „Rocket.Chat"-Stack

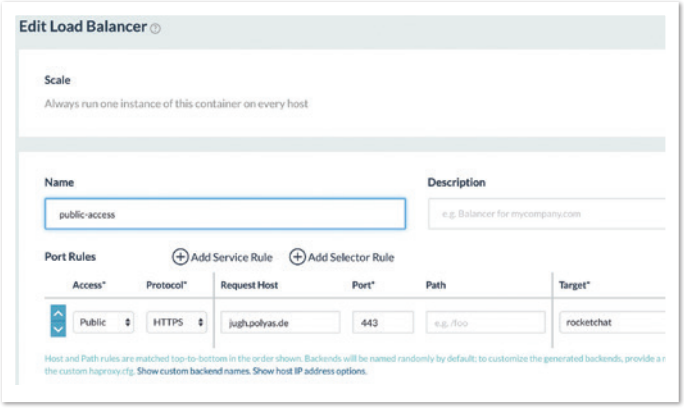


Abbildung 11: Load Balancer mit SSL-Terminierung

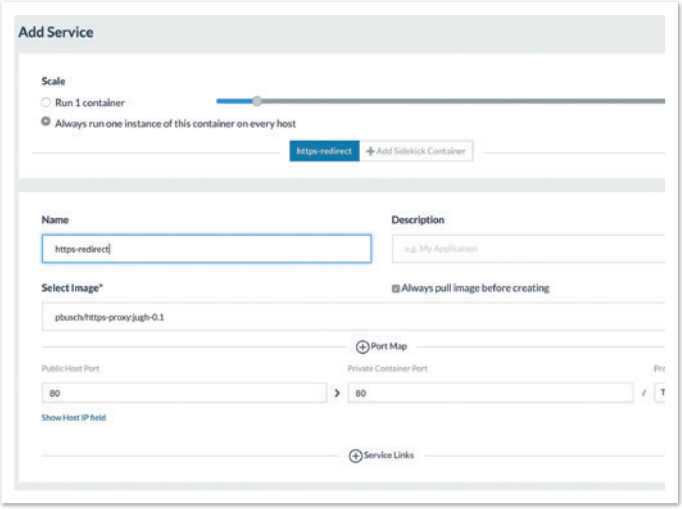


Abbildung 12: Konfiguration für einen Service zur Weiterleitung von HTTP auf HTTPS

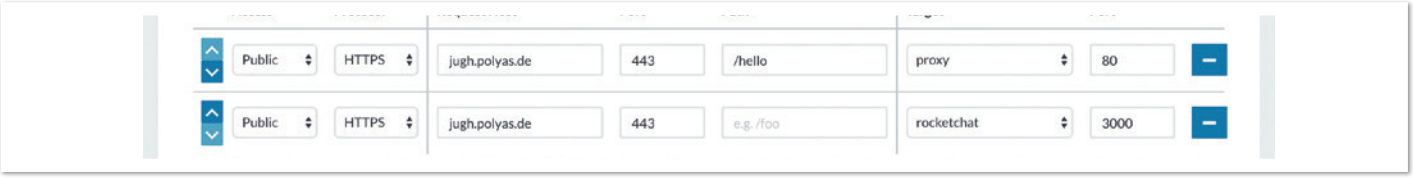


Abbildung 13: Angepasste Konfiguration des Load Balancer mit zusätzlichem Service

nun unter derselben Domain über HTTPS erreichbar und bei dem Versuch, auf HTTP zu verbinden, wird auf HTTPS umgeleitet.

Eigene Applikationen

Mit dem nun angeeigneten Vorwissen und den durchgeführten Vorbereitungen ist das Deployment einer eigenen Applikation relativ einfach. Als Beispiel nehmen wir eine simple, vorkonfigurierte Applikation, bestehend aus einem Webserver, der statische Assets ausliefert, und einem vorgeschalteten Proxy. Dafür legen wir zuerst einen leeren Stack an. Als erster Service wird das Image „pbusch/webpage-proxy:1.0" verwendet. Es sollte unter dem Namen „proxy" laufen.

Als zweiter Service kommt das Image „pbusch/webpage-assets:1.0" zum Einsatz. Dieses sollte unter dem Namen „assets" laufen. Über die entsprechenden Namen können die Services eine Namensauflösung über das Rancher-interne DNS machen. Generell ist jeder Service innerhalb eines Environments unter der Domain „servicename.stackname.rancher.internal" erreichbar. Da „rancher.internal" als Suchdomain vorkonfiguriert ist, reicht „servicename.stackname" als Adresse. Innerhalb eines Stacks genügt sogar nur der Name des Service.

Bei Einhaltung eines Namensschemas ist die Verlinkung der Container innerhalb eines Stacks nur für die Namensauflösung nicht nötig. Natürlich kann man auch im Beispiel beliebige Namen verwenden, dann müsste allerdings der Service mit den Assets unter dem Namen „assets" zum Proxy-Service verlinkt werden.

Da hier gar keine öffentlichen Ports definiert wurden und diese Applikation wenn möglich unter demselben Zertifikat laufen soll, muss nun der Load Balancer angepasst werden. Dafür muss in der Konfiguration des Load Balancer eine neue Service-Regel hinzugefügt werden. Diese sollte mit demselben Protokoll, Host und Port wie die bestehende Regel angelegt sein. „Target" sollte der gerade angelegte Proxy-Service auf Port 80 sein. Zusätzlich wird jetzt ein „Request Path" angegeben: „/hello". Die neue Regel sollte dann über der bestehenden Regel stehen, da der Request Path auf den neuen Service beschränkt ist. Die Regeln des Load Balancer werden von oben nach unten abgearbeitet und es greift für jeden Request die erste Regel, bei der alle Bedingungen erfüllt sind. *Abbildung 13* zeigt die Konfiguration. Unter der konfigurierten Domain und dem Pfad „/hello" sollte nun die Seite wie in *Abbildung 14* erscheinen. Somit ist unsere erste eigene Applikation lauffähig.

```
server {
  listen 80 default_server;
  listen [::]:80 default_server;
  server_name _;
  add_header "Strict-Transport-Security" "max-
age=31536000";
  return 301 https://$host$request_uri;
}
```

Listing 2

Abbildung 14: Erster eigener Service

Fazit

Der typische Application Stack mit Frontend, Backend und Datenbank ist mit Rancher einfach zu managen, solange Frontend- und Backend-Services „stateless" gehalten sind. Die Möglichkeiten zur Skalierung ergeben sich hierbei fast von selbst. Rancher ist schnell aufgesetzt, bietet die Möglichkeit, schnell zu Ergebnissen zu kommen, und bringt dennoch die Robustheit für einen Produktivbetrieb mit, solange die einzelnen Host-Systeme entsprechend gegen Angriffe geschützt sind.

Durch die Möglichkeit, verschiedene Environments betreiben zu können, und über die Erstellung und Einbindung eines eigenen Rancher Catalog ist die Einbindung in die eigene Release-Pipeline sogar empfehlenswert. Allerdings sollte dabei ein Augenmerk darauf gelegt werden, dass die eigene Software über private Docker-Registries angebunden ist.

Patrick Busch
p.busch@polyas.de

Patrick Busch ist Head of SaaS Development bei der Polyas GmbH. Aus Karlsruhe stammend, verbrachte er die ersten Jahre seines Berufslebens als Entwickler bei einem dort ansässigen Hosting-Unternehmen. Seit dem Umzug nach Kassel arbeitet er mit der Polyas GmbH, erst als Entwickler, nun in einer Führungsposition, daran, Online-Wahlen auch in Deutschland erfolgreich auf den Weg zu bringen.

Application Performance Management mit Open-Source-Tooling

Mario Mann, NovaTec Consulting GmbH

Performance ist wichtig! Die Performance der Anwendungen wirkt sich direkt auf die Geschäftsmetriken aus. Schlechte Anwendungs-Performance – also hohe Antwortzeiten, geringer Durchsatz oder übermäßiger Ressourcenverbrauch – haben in der Regel hohe Kosten für ein Unternehmen zur Folge. Oft sind dabei direkt die Nutzer betroffen, was zu Reputationsschäden und im schlimmsten Fall zu Kundenverlust führen kann.

Application Performance Management (APM) stellt die erforderlichen Prozesse und Werkzeuge zur Verfügung, um ein kontinuierliches und aktuelles Abbild relevanter Leistungsmaße der Anwendung zu erhalten. So lassen sich Performance-Probleme und leistungsbezogene Vorfälle frühzeitig erkennen, analysieren und auflösen.

Performance-Probleme, die bei der Entwicklung der Anwendungen entstehen können, haben in der Regel einen Umsatzverlust zur Folge und können sogar dazu führen, dass sich Kunden abwenden. Beispiele für diese Verluste und deren Auswirkungen sind gut dokumentiert. Das Unternehmen Google verliert 20 Prozent Traffic, wenn

die Antwortzeiten 500 Millisekunden langsamer sind als üblich *[1]*. Der Onlinehändler Amazon verliert ein Prozent des Umsatzes für jede Latenz von 100 Millisekunden *[2]*. Laut einer Studie von Mozilla verlassen Nutzer eine Website bereits, sobald diese nicht innerhalb von ein bis fünf Sekunden geladen ist *[3]*.

Als eine Management-Disziplin zielt APM darauf ab, ein angemessenes Leistungsniveau sicherzustellen. Um dies zu erreichen, umfasst APM Methoden, Techniken und Werkzeuge für die kontinuierliche Überwachung des Zustands einer Anwendung und ihrer Nutzung sowie für das Erkennen, Diagnostizieren und Auflösen von Performance-spezifischen Problemen, die in Zusammenhang mit den Daten auftreten.

Der Artikel gibt einen Überblick über die APM-Aktivitäten, Open-Source-Werkzeuge, die unter anderem zur Verfügung stehen, und deren Herausforderungen. Ein kurzer Ausblick rundet das Thema ab.

APM-Aktivitäten

Wenn man über APM spricht, sind die vier folgenden Aktivitäten gemeint, unabhängig von der konkreten Technik. An erster Stelle steht das Datensammeln: Performance-Metriken werden aus den verschiedenen Tiers, Schichten und Standorten einer Systemlandschaft durch eine Kombination verschiedener Techniken und Technologien gesammelt. Hierzu gibt es verschiedene Ansätze – es werden aktive sowie passive Ansätze unterschieden. Das aktive Datensammeln wird durch ein periodisches Sampling von Services oder Ressourcen eines Systems realisiert. Dies umfasst auch das Simulieren von Endbenutzern durch Ausführen synthetischer Anfragen.

Das passive Datensammeln wird realisiert, indem man bestimmte Ereignisse aufzeichnet, die durch Benutzer-Aktionen erfolgen. Diese Daten können gesammelt werden, indem der Sourcecode der Anwendung zur Laufzeit analysiert wird (Byte-Code-Instrumentierung), ein zyklisches Stack-Trace-Sampling durchgeführt wird, die Anwendungs-Logs analysiert werden oder der Netzwerk-Traffic mitgelesen wird.

An zweiter Stelle steht das Speichern und Verarbeiten der Daten: Die gesammelten Daten werden in verschiedenen Datenstrukturen gespeichert und logisch kombiniert, etwa in Time-Series-Graphen oder Execution-Traces. Dazu können proprietäre oder Standard-Technologien (wie Datenbank-Managementsysteme) zum Einsatz kommen. Während Zeitreihen die Statistik (Aufzählungen, Perzentile etc.) im Lauf der Zeit darstellen, liefern Execution-Traces eine detaillierte Darstellung des anwendungsinternen Kontrollflusses, der sich aus einzelnen Anfragen an das System ergibt. Aus diesen Daten lassen sich unter anderem architektonische Informationen extrahieren. APM führt in der Regel zu einer riesigen Menge von Datensätzen, die effizient gehandhabt werden müssen.

Dritter Punkt ist das Darstellen der Daten: Die gesammelte Information soll sinnvoll und nachvollziehbar mit unterschiedlichen zusammenhängenden Ansichten präsentiert werden. Diese Ansichten lassen sich in zwei Dimensionen kategorisieren, den Umfang und die Ebene der Abstraktion. Die Darstellungen können detaillierte Geschäftsinformationen wie den Status von Benutzergeräten, Geo-Lokationen sowie den Status verfügbarer Dienste enthalten. Zudem lassen sich Daten in Form von Zeitreihen und Page-Flows darstellen. Der Status von Servern und die zugrunde liegende Topologie sind ebenfalls darstellbar. Diese Ansichten variieren von abstrakten bis hinzu detaillierten Darstellungen, je nachdem, was erforderlich ist.

An letzter Stelle steht das Interpretieren und Verwenden der Daten: Die gesammelten Daten werden zur manuellen oder automatischen Interpretation genutzt. Somit kann bei erhöhter Last durch zeitgerecht eingeleitete Gegenmaßnahmen die Verfügbarkeit gewährleistet werden, etwa durch Autoscaling von Containern.

Es gibt eine Reihe von kommerziellen APM-Tools (AppDynamics, DynaTrace, NewRelic etc.), die einen großen Funktionsumfang haben und die beschriebenen APM-Aktivitäten in einem Produkt gesammelt anbieten. In einigen Fällen sind kommerzielle Werkzeuge aufgrund von Lizenzkosten, Lieferantensperren oder anderen Gründen jedoch nicht die geeignete Wahl.

APM-Open-Source-Tools

Open-Source-Werkzeuge bieten eine gute Alternative. Hier sind unter anderem Kieker *[4]*, Hawkular *[5]* und inspectIT *[6]* zu nennen. Im Artikel ist der Fokus auf inspectIT gerichtet. inspectIT ist seit dem Jahr 2005 in der Entwicklung; seit August 2015 ist es Open-Source. Als Grundgedanke verfolgt inspectIT das Pareto-Prinzip *[7]*, was nicht heißen soll, dass nur 20 Prozent des Aufwands in die Entwicklung von inspectIT fließen. Vielmehr wird der Fokus der Entwicklung auf die grundlegende Funktionalität gelegt, die ein APM-Tool ausmacht. Die übrigen 20 Prozent der Funktionalität, die meist seltene Anforderungen und Spezialfälle adressieren, lassen sich durch die Kombination mit anderen Open-Source-Werkzeugen in der APM-Kette abdecken.

Die wichtigsten Aktionsfelder von inspectIT liegen in der Art und Weise, wie Metriken der Anwendung gesammelt werden können, und darin, welche Interpretationsmöglichkeiten diese bieten. Das Speichern von Daten sowie die Darstellung in verschiedenen Ebenen der Abstraktion werden von anderen Tools übernommen. So strebt inspectIT die Integration mit anderen Werkzeugen an und ist individuell erweiterbar, sodass alle APM-Aktivitäten vollständig abgedeckt werden können. *Abbildung 1* zeigt eine Übersicht über inspectIT und dessen Komponenten.

Der inspectIT-Server stellt die zentrale Komponente in der inspectIT-Architektur dar. Dieser empfängt die von den Agenten gesammelten Daten, verwaltet sie und stellt Schnittstellen für Abfragen zur Verfügung. Dabei unterscheidet der Server zwischen zwei Arten von Daten:

- Langzeitdaten hinsichtlich der Überwachung (CPU-Auslastung, Antwortzeiten, laufende Threads etc.)
- Detaillierte Daten für die Diagnose (Execution-Traces, Methodenaufruf-Hierarchien etc.)

Die Langzeitdaten sind in einer Zeitreihen-Datenbank gespeichert; in diesem Fall in einer InfluxDB *[8]*. Die detaillierten Daten liegen in einem In-Memory-Ring-Buffer.

Die Verwendung von inspectIT in Verbindung mit einer Zeitreihen-Datenbank bietet die Möglichkeit, die Langzeitdaten über Dash-

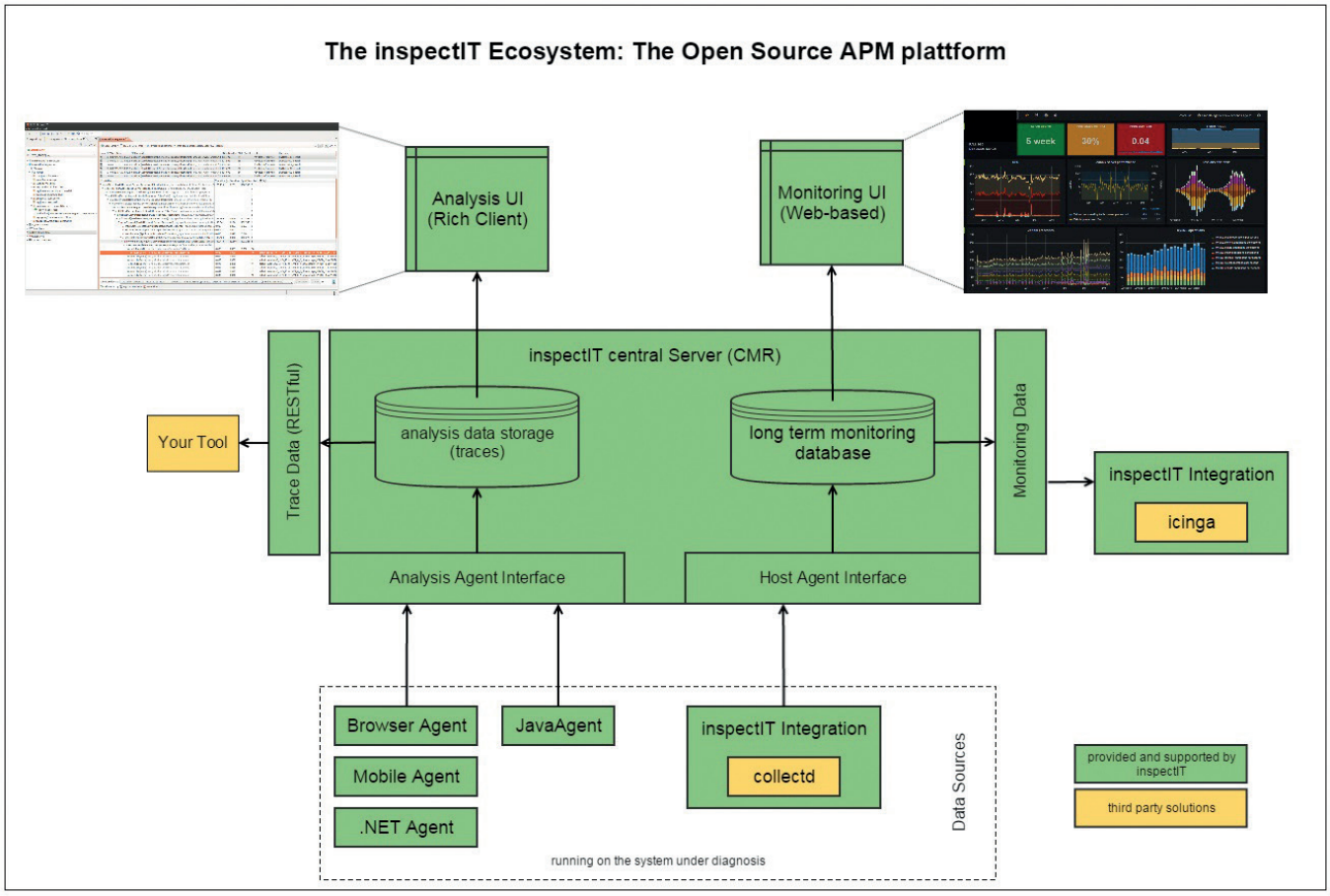


Abbildung 1: Das inspectIT-Big-Picture

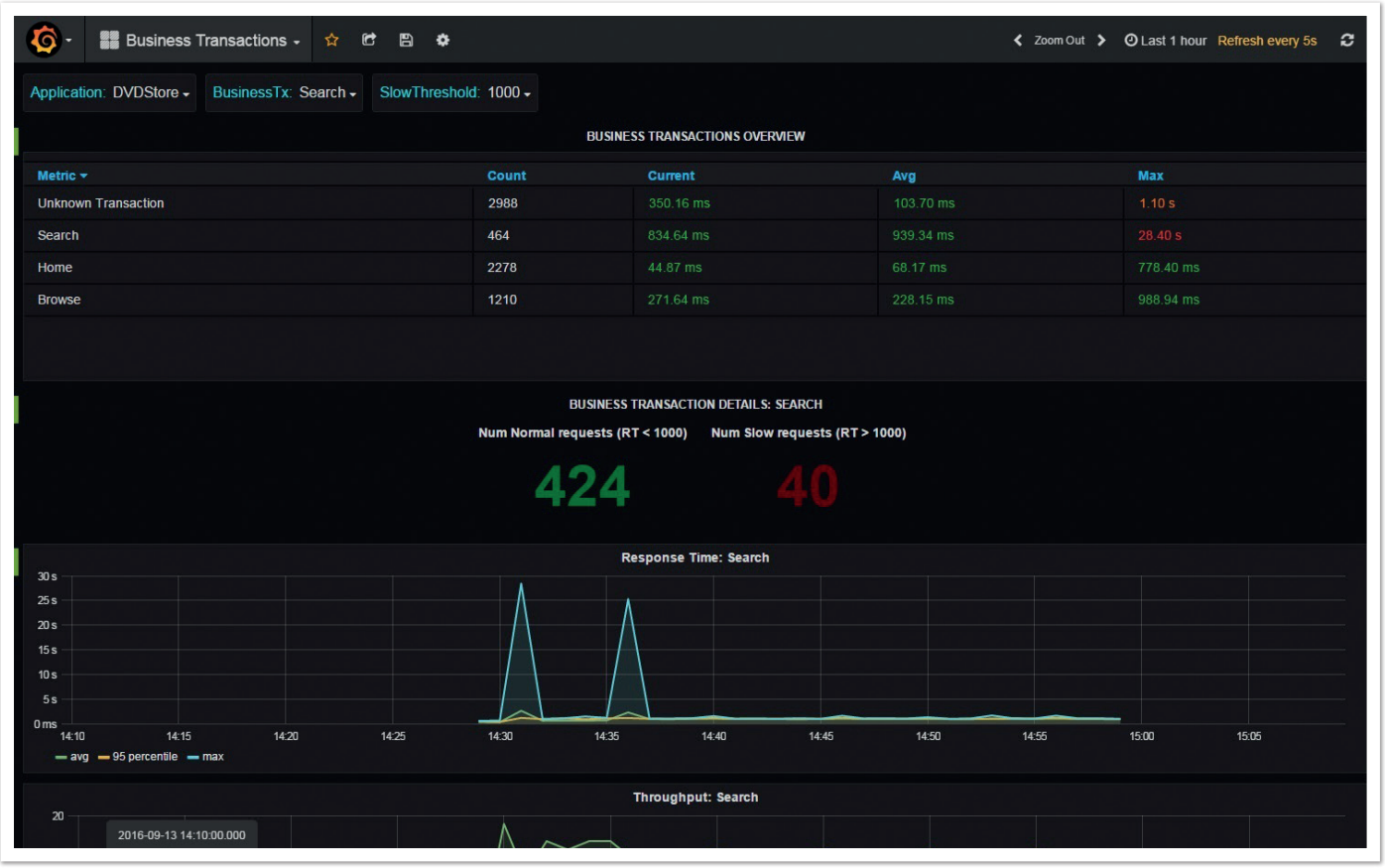


Abbildung 2: Business-Transaction

Abbildung 3: Anomalie-Detection

boards visuell darzustellen. Hier hat sich in den letzten Jahren das Open-Source-Tool Grafana *[9]* bewährt. Es bietet die Möglichkeit, verschiedene Panels zu einem Dashboard zusammenzufügen. Die Panels können dabei unterschiedlichste Daten grafisch darstellen, abhängig von den Bedürfnissen des jeweiligen Nutzers. So lassen sich Graphen, Tabellen, Pie-Charts, Einzelwerte (Singlestats) etc. anzeigen. *Abbildung 2* zeigt das Beispiel eines Dashboards, das auf den Metriken von inspectIT basiert.

Dashboards stellen nicht nur Metriken in Abhängigkeit von der Zeit dar, sie setzen auch verschiedene Metriken in Zusammenhang zueinander. So ist es beispielsweise möglich, den Zusammenhang zwischen der CPU-Auslastung mit der Antwortzeit von Anfragen in einem Graphen zu visualisieren. Damit wäre zum Beispiel die Analyse einer Thread-Contention an Software-Ressourcen (Connection-Pools, Worker-Pools etc.) mit einem Blick möglich.

Weiterhin bieten Dashboards hervorragende Möglichkeiten, die gesammelten Metriken visuell darzustellen. Diese können auf einem großen Bildschirm angezeigt werden, sodass beteiligte Nutzer die Werte der zu überwachenden Anwendung stets im Blick haben. Auch wenn Dashboards durchaus informativ sein können, kann es nicht das Ziel eines Monitoring-Systems sein, dass dessen Anwender 24 Stunden am Tag, 7 Tage die Woche ein Dashboard beobachten, um auf signifikante Änderungen in der Anwendung rechtzeitig reagieren zu können. Es muss eine Benachrichtigungsmöglichkeit zum Zeitpunkt des ersten Auftretens eines Problems geben.

Es gibt mehrere Möglichkeiten, wie solch eine Information beziehungsweise ein Alarm sichtbar werden. Das Tool Grafana bietet die Möglichkeit, fixe Schwellwerte zu definieren und über verschiedene Kanäle Informationen fließen zu lassen, wie E-Mail, PagerDuty, Slack, HipChat oder über webhooks. inspectIT bietet ebenfalls fixe Schwellwerte und Informationen per E-Mail. Der Unterschied zum Alerting von Grafana besteht darin, dass man mit inspectIT einen Business-Context definieren und bei Veränderungen dazu einen Alarm schalten kann.

So ein rudimentäres Alerting ist gut, birgt allerdings auch Nachteile. Zum Beispiel erfolgt eine Alarmierung oft, wenn Schwellwerte in regelmäßigen Abständen über- beziehungsweise unterschritten werden. Eine Benachrichtigungsflut ist jedoch vom Benutzer meistens nicht gewünscht. Es soll vielmehr bei einer Anomalie eine einzige Information gesendet werden. Hier bieten die aktuell verwendeten Open-Source Tools inspectIT, InfluxDB und Grafana noch keine Lösung an. Allerdings bilden auch Machine-Learning-Ansätze brauchbare Optionen, ein dynamisches Alerting zu integrieren.

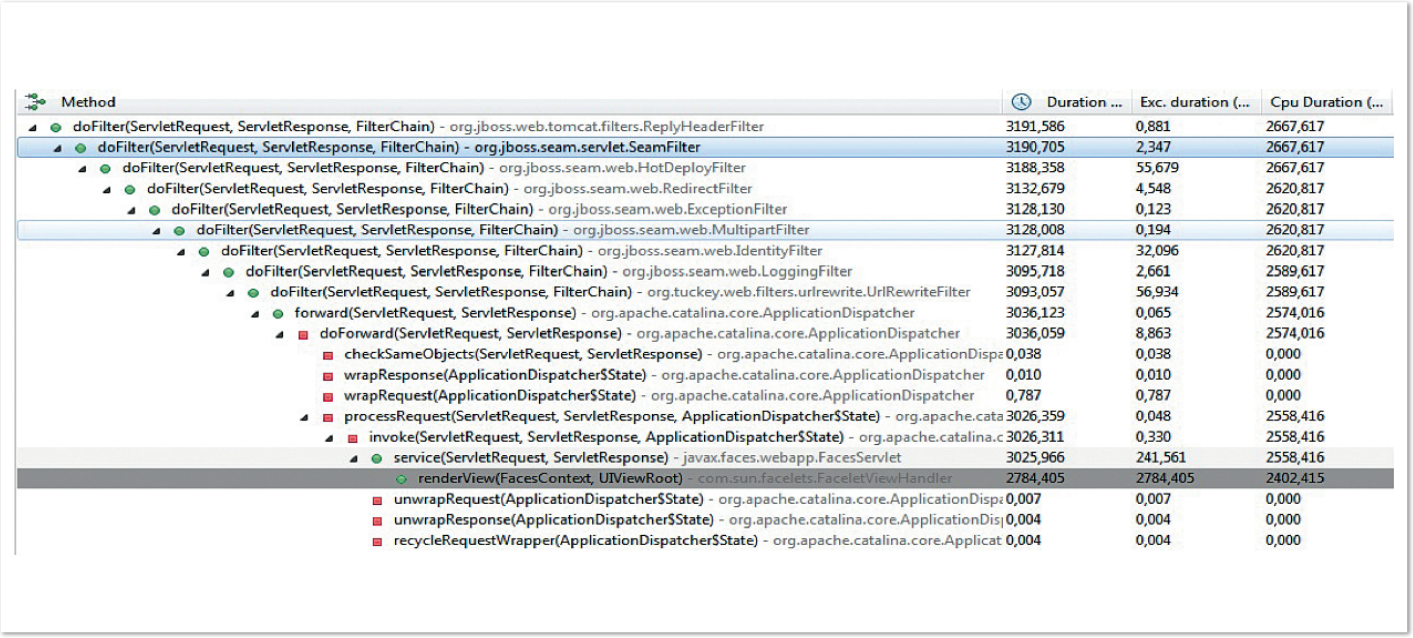
Zukünftig soll auch bei inspectIT eine Anomalie-Erkennung zur Verfügung stehen. Anhand eines dynamischen Bands (in *Abbildung 3* blau gekennzeichnet) sind Ober- und Untergrenzen definiert. Diese werden dynamisch zur Laufzeit aus dem historischen Verhalten des Systems gelernt und bieten so den Ansatz eines dynamischen Alerting.

Wurde ein Problem festgestellt, soll dieses auch genauer analysiert werden sowie die gleichzeitige Diagnose der Ursache erfolgen. Das ist allein mit schönen High-Level-Dashboards nicht möglich. So lassen sich mit dem User-Interface von inspectIT die Execution-Traces hinsichtlich der Ausführungszeiten, Ausführungsanzahl, CPU-Zeiten etc. analysieren *(siehe Abbildung 4)*.

Ausblick

In diesem Artikel sind alle APM-Aktivitäten mit Open-Source-Tools abgedeckt. Das Sammeln von Metriken kann mit inspectIT erfolgen, das Speichern mit einer InfluxDB und das Darstellen mit Grafana. Lediglich die Interpretation der Daten erfolgt bislang manuell; aber auch hier bietet der Open-Source-Markt Abhilfe.

Eine manuelle Analyse von Execution-Traces funktioniert nur dann zufriedenstellend, wenn nur wenige Traces vorhanden sind. Aber was passiert, wenn viele Hunderte Traces zur Analyse zur Verfügung stehen? In diesem Fall ist davon auszugehen, dass signifikante Probleme übersehen werden. Dadurch lassen sich eventuell sogar die Ursachen gar nicht erst finden. Deshalb ist es notwendig, auch



Method	Duration ...	Exc. duration (...	Cpu Duration (...
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.web.tomcat.filters.ReplyHeaderFilter	3191,586	0,881	2667,617
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.servlet.SeamFilter	3190,705	2,347	2667,617
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.web.HotDeployFilter	3188,358	55,679	2667,617
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.web.RedirectFilter	3132,679	4,548	2620,817
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.web.ExceptionFilter	3128,130	0,123	2620,817
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.web.MultipartFilter	3128,008	0,194	2620,817
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.web.IdentityFilter	3127,814	32,096	2620,817
doFilter(ServletRequest, ServletResponse, FilterChain) - org.jboss.seam.web.LoggingFilter	3095,718	2,661	2589,617
doFilter(ServletRequest, ServletResponse, FilterChain) - org.tuckey.web.filters.urlrewrite.UrlRewriteFilter	3093,057	56,934	2589,617
forward(ServletRequest, ServletResponse) - org.apache.catalina.core.ApplicationDispatcher	3036,123	0,065	2574,016
doForward(ServletRequest, ServletResponse) - org.apache.catalina.core.ApplicationDispatcher	3036,059	8,863	2574,016
checkSameObjects(ServletRequest, ServletResponse) - org.apache.catalina.core.ApplicationDispa	0,038	0,038	0,000
wrapResponse(ApplicationDispatcher$State) - org.apache.catalina.core.ApplicationDispatcher	0,010	0,010	0,000
wrapRequest(ApplicationDispatcher$State) - org.apache.catalina.core.ApplicationDispatcher	0,787	0,787	0,000
processRequest(ServletRequest, ServletResponse, ApplicationDispatcher$State) - org.apache.cata	3026,359	0,048	2558,416
invoke(ServletRequest, ServletResponse, ApplicationDispatcher$State) - org.apache.catalina.c	3026,311	0,330	2558,416
service(ServletRequest, ServletResponse) - javax.faces.webapp.FacesServlet	3025,966	241,561	2558,416
renderView(FacesContext, UIViewRoot) - com.sun.facelets.FaceletViewHandler	2784,405	2784,405	2402,415
unwrapRequest(ApplicationDispatcher$State) - org.apache.catalina.core.ApplicationDispa	0,007	0,007	0,000
unwrapResponse(ApplicationDispatcher$State) - org.apache.catalina.core.ApplicationDisp	0,004	0,004	0,000
recycleRequestWrapper(ApplicationDispatcher$State) - org.apache.catalina.core.Applicat	0,004	0,004	0,000

Abbildung 4: Execution-Traces

die Analyse von Problemen automatisiert durchzuführen. Ein Automatismus, der eine Kategorisierung von Problemen vornimmt und gleichzeitig Performance-Anti-Pattern aufzeigt, ist hierfür bestens geeignet.

Neben dem Anzeigen von Problemen wäre es auch wünschenswert, dass Problemlösungen ebenso dargelegt werden. Diese Art von automatisierter Analyse bietet das Open-Source-Tool diagnoseIT *[10]*. Es ist ein Forschungsprojekt, widmet sich genau diesen Aspekten und wird im Herbst 2017 voll in inspectIT integriert sein.

Fazit

Die vorhandenen Open-Source-Tools bieten attraktive Möglichkeiten, die eingangs genannten APM-Aktivitäten zu übernehmen; es muss lediglich eine Kombination dieser erfolgen. Dabei ist es nicht notwendig, auf die in diesem Artikel genannten Open-Source-Tools zurückzugreifen: Anstatt der Zeitreihen-Datenbank InfluxDB kann man auch KairosDB verwenden, anstelle von Grafana kann Kibana beziehungsweise anstatt inspectIT kann Kieker zum Einsatz kommen.

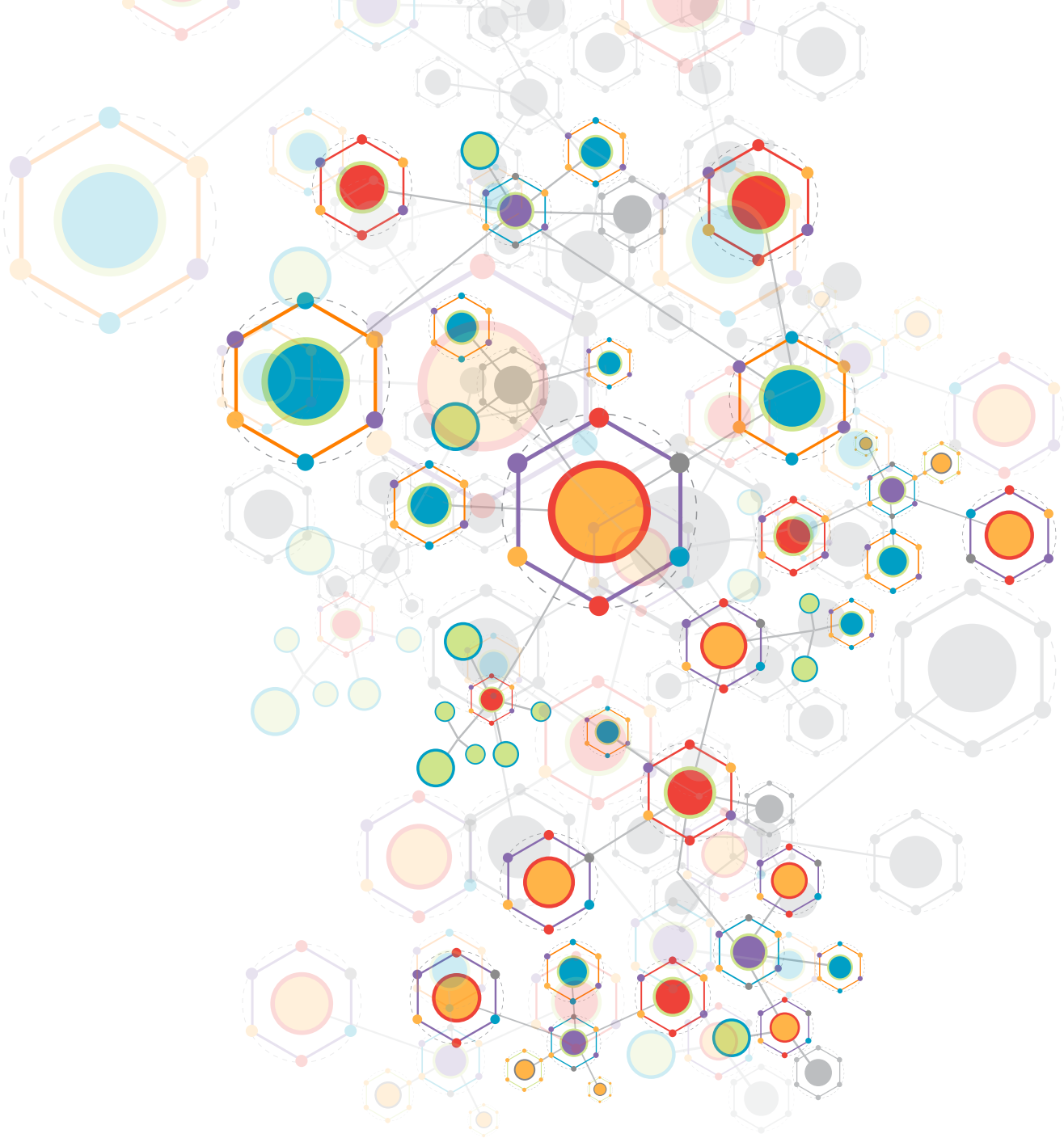
Gerade das ist der Vorteil von Open-Source-Tools: Sie sind sehr gut miteinander kombinierbar, da sie offene Schnittstellen bieten. Allerdings ist dabei zu beachten, dass, wenn zu viele verschiedene Tools im Einsatz sind, schnell der Überblick darüber verloren gehen kann, welches Tool für welche Aufgabe verantwortlich ist. Daher sollte man in regelmäßigen Abständen verifizieren, ob das verwendete Tool einen Mehrwert beziehungsweise den erhofften Nutzen bietet. Sofern das nicht der Fall ist, sollte es aus der Tool-Kette entfernt werden, um die Übersichtlichkeit zu wahren und ein effizientes Monitoring-System für die Anwendung zu gewährleisten.

Weitere Informationen

[1] *http://glinden.blogspot.de/2006/11/marissa-mayer-at-web-20.html*
[2] *http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales*
[3] *https://blog.mozilla.org/metrics/2010/03/31/firefox-page-load-speed-part-i*
[4] *http://kieker-monitoring.net/apm*
[5] *http://www.hawkular.org*
[6] *http://www.inspectit.rocks*
[7] *https://de.wikipedia.org/wiki/Paretoprinzip*
[8] *https://www.influxdata.com*
[9] *https://grafana.com*
[10] *https://diagnoseit.github.io*

Mario Mann
mario.mann@novatec-gmbh.de

Mario Mann ist seit vier Jahren IT-Berater bei der NovaTec Consulting GmbH im Bereich „Application Performance Management". Er ist bestrebt, die Vision von APM zu verbreiten, weshalb er regelmäßig als Sprecher auf Konferenzen auftritt. Seine Erfahrungen aus den APM-Projekten lässt er auch als Entwickler in inspectIT einfließen.



Kombinator als funktionales Entwurfsmuster in Java

Gregor Trefs, Freiberufler

Neben Streams und Optionals bietet Java 8 weitere Möglichkeiten, um Problemstellungen funktional zu lösen. Das Kombinator-Entwurfsmuster kombiniert kleine fachliche Funktionen situationsgerecht zu komplexer Fachlogik. Vorteile sind, neben der klaren Trennung von Verantwortlichkeiten, die Erweiterbarkeit, die Wiederverwendbarkeit und der deklarative beziehungsweise domänengetriebene Ansatz. Dieser Artikel beschreibt anhand praktischer Beispiele, wie das Kombinator-Entwurfsmuster funktioniert und wann der Einsatz sinnvoll ist.

```
int addOne(int i){
    return i + 1;
}
```

Listing 1

```
Function<Integer, Integer> addOne = i -> i + 1;
int compute(Function<Integer, Integer> f, int value){
    return f.apply(value);
}
compute(addOne, 1); // ergibt 2
```

Listing 2

```
Function<Integer, Integer> addOne = i -> i + 1;
Function<Integer, Integer> addTwo;
addTwo = i -> addOne.apply(addOne.apply(i));
Function<Integer, Integer> addThree = addTwo.
compose(addOne);
```

Listing 3

Das Kombinator-Entwurfsmuster, kurz Kombinator, folgt dem „Teile und Herrsche"-Prinzip. Eine Menge von Primitiven beschreibt die simpelsten Elemente einer Domäne und wird mit einer Menge von Kombinatoren zu komplexerer Logik kombiniert. Kombinator ähnelt also dem Kompositum-Entwurfsmuster. Der Unterschied liegt in den Dingen, die miteinander kombiniert werden. Im Kompositum sind es Objekte, die in einer Baumstruktur zusammengeführt werden; beim Kombinator sind es Funktionen, die mit Funktionen zu neuen Funktionen kombiniert werden.

Im Umkehrschluss sollten wir zunächst verstehen, was eine Funktion ist und was Funktionen höherer Ordnung sind. Danach betrachten wir das Kombinator-Entwurfsmuster am Beispiel einer Benutzervalidierung und diskutieren, welcher Rückgabewert in solch einer Domäne sinnvoll ist und wie dieser die Gestaltung unserer Primitiven und Kombinatoren beeinflusst. Zuletzt werden Vorteile, Herausforderungen und das Einsatzgebiet vorgestellt.

Funktionen und Funktionen höherer Ordnung

Java ist keine funktionale Sprache. Jedoch ist es möglich, Programme funktional zu gestalten. Betrachten wir dazu die Methode „addOne" in *Listing 1*. Die Logik ist simpel: Jeder übergebene Wert wird um 1 erhöht und zurückgegeben. Allerdings hat „addOne" zwei interessante Eigenschaften: Zum einen hängt der Rückgabewert ausschließlich vom Übergabeparameter „i" ab. Zum anderen gibt es, außer dem Rückgabewert, keinen anderen von außen wahrnehmbaren Effekt auf die Ausführung des Programms, „addOne" hat keine Seiteneffekte. Eine Methode mit diesen beiden Eigenschaften wird als „pure Funktion" bezeichnet. In diesem Artikel nehmen wir an, dass mit Funktion immer eine pure Funktion gemeint ist. Durch ihre Zustandslosigkeit sind Funktionen unter anderem einfach testbar, parallel ausführbar und ihr Verhalten ist nachvollziehbarer.

Funktionen sind Werte. Sie können anderen Funktionen übergeben und/oder von diesen zurückgegeben werden. Funktionen, die auf anderen Funktionen arbeiten, werden „Funktionen höherer Ordnung" genannt. Um das zu verdeutlichen, sehen wir in *Listing 2* die Funktion „addOne" in Lambda-Syntax und die höhere Funktion „compute".

Die Logik von „addOne" hat sich nicht geändert. Die Funktion „compute" nimmt eine Funktion „f" und einen Wert „value" entgegen und wendet „f" auf „value" an. Das Ergebnis mit den Parametern „addOne" und „1" ist „2".

Funktionen können auch miteinander zu neuen Funktionen verknüpft werden. Dabei beschreiben der Eingabe- und Ausgabetyp einer Funktion eine Schnittstelle und bestimmen die Verknüpfungsmöglichkeiten *(siehe Listing 3)*.

In Funktion „addTwo" wird „addOne" mit sich selbst verknüpft. Zuerst wird „addOne" auf den Parameter „i" angewendet und danach nochmals auf den Rückgabewert. Das Ergebnis ist „i", um „2" erhöht. Diese Art der Komposition ist als „default"-Methode auf der Schnittstelle „Function" verfügbar und wird für die Funktion „addThree" benutzt, die sich aus den Funktionen „addOne" und „addTwo" zusammensetzt.

Das Kombinator-Entwurfsmuster

Wie die meisten Konstrukte in der funktionalen Programmierung sind Primitive und Kombinatoren Namen für Abstraktionen. In Wikipedia sind Primitive in Programmiersprachen wie folgt beschrieben: „[...] Sprach-Primitive sind die simpelsten Elemente, die in einer Programmiersprache vorhanden sind. Ein Primitiv [...] ist ein atomares Element eines Ausdrucks in einer Sprache. Sie sind Einheiten mit Semantik [...]." Gleichermaßen sind Primitive im Kombinator-Entwurfsmuster die simpelsten Elemente innerhalb einer Domäne, etwa Addition und Multiplikation in der Domäne der Ganzzahlen. Kombinatoren beschreiben, wie Primitive und/oder bereits verknüpfte Strukturen miteinander in noch komplexere Strukturen verknüpft werden können, zum Beispiel die Verknüpfung von Multiplikation und Addition.

```
public class User{
   public final String name;
   public final int age;
   public final String email;

   public User(String name, int age, String email){
     this.name = name;
     this.age = age;
     this.email = email;
   }

   public boolean isValid(){
     return nameIsNotEmpty() && mailContainsAtSign();
   }

   private boolean nameIsNotEmpty(){
     return !name.trim().isEmpty();
   }

   private boolean mailContainsAtSign(){
     return email.contains("@");
   }
}

new User("Gregor", 31, "nicemail@gmail.com").is-
Valid(); // true
```

Listing 4

```
interface UserValidation extends Function<User, Boolean>{}

UserValidation nameIsNotEmpty = u -> !u.name.trim().isEmpty();
UserValidation mailContainsAtSign = u -> u.email.contains("@");
User gregor = new User("Gregor", 31, "nicemail@gmail.com");

nameIsNotEmpty.apply(gregor) &&
  mailContainsAtSing.apply(gregor); // true
```

Listing 5

```
interface UserValidation extends Function<User, Boolean>{
    UserValidation nameIsNotEmpty = u ->
       !u.name.trim().isEmpty();
    UserValidation mailContainsAtSign = u ->
       u.email.contains("@");

    default UserValidation and(UserValidation other) {
        return user -> this.apply(user) && other.apply(user);
    }
}

User gregor = new User("Gregor", 30, "nicemail@gmail.com");
nameIsNotEmpty.and(mailContainsAtSign).apply(gregor); // true
```

Listing 6

```
List<User> users = findAllUsers()
    .stream().parallel()
    // Konvertierung von UserValidation zu Predicate
    .filter(nameIsNotEmpty.and(mailContainsAtSign)::
apply)
    .collect(Collectors.toList());
```

Listing 7

Die Validierung mit Kombinatoren

Stellen wir uns vor, wir sind beauftragt, die Benutzer einer Applikation zu validieren. Benutzer sind berechtigt, einen Dienst zu nutzen, wenn der Name nicht leer ist und die E-Mail Adresse ein „@"-Zeichen enthält. Ein direkter Weg, dies zu modellieren, wären Abfragemethoden in der entsprechenden Entität *(siehe Listing 4)*.

Obwohl sich das gut liest, hat diese Lösung ein paar Nachteile. Was passiert, wenn sich die Validierungsregeln ändern? Zum Beispiel kann es sein, dass Benutzer zur Nutzung von bestimmten Diensten älter als 14 sein müssen. In diesem Fall ist die Methode „isValid" anzupassen, was eine Verletzung des Single-Responsibility-Prinzips ist.

Bevor wir mit dem Refactoring beginnen und die Validierung in eine eigene Klasse extrahieren, sollten wir noch einmal darüber nachdenken, was wir eigentlich ausdrücken wollen. Benutzer sind genau dann valide, wenn eine Menge an Regeln erfüllt ist. Eine Regel beschreibt hierbei eine Benutzer-Eigenschaft. Verschiedene Regeln können miteinander zu komplexeren Regeln kombiniert werden. Die Anwendung einer Regel ergibt ein Validierungsergebnis, das den Ausgang der Validierung beschreibt. Zunächst nehmen wir an, dass der boolesche Wert „true" Benutzer als „valide" auszeichnet. *Listing 5* drückt das in Funktionen aus.

Der Code ist etwas langatmig. Warum werden die Regeln „nameIsNotEmpty" und „mailContainsAtSign" manuell erstellt und verknüpft? Sind vier Zeilen Initialisierung für gerade mal eine Anwendungszeile wirklich nötig? Auch wenn der Code etwas wortreich ist, so enthält er doch alle Bausteine für das Kombinator-Entwurfsmuster: „nameIsNotEmpty" und „mailContainsAtSign" sind Primitive und „&&" ist ein Kandidat für einen Kombinator. Seit Java 8 ist es möglich, Primitive und Kombinatoren in den „UserValidation"-Typ zu schieben *(siehe Listing 6)*.

Primitive sind als statische Variablen definiert und Kombinatoren als „default"-Methoden. Beide Primitive „nameIsNotEmpty" und „mailContainsAtSign" sind ein Lambda-Ausdruck mit „UserValidation" als Ziel-Typ. Im Unterschied dazu ist der „and"-Kombinator eine Funktion höherer Ordnung, die zwei „UserValidation" mit dem „&&"-Operator verknüpft. Hinzu kommen zwei Beobachtungen: Zum einen werden „UserValidation" während der Konstruktion einer Benutzervalidierung nicht ausgeführt. Zum anderen haben sie keinen eigenen Zustand. Unter anderem bedeutet das, dass eine

```
interface ValidationResult{
    static ValidationResult valid(){
      // Gibt immer die selbe Instanz zurück
      // mit isValid gleich true
      // und getReason gleich Optional.empty()
      return ValidationSupport.valid();
    }

    static ValidationResult invalid(String reason){
        // Gibt neue Instanz mit
        // isValid gleich false
        // und getReason gleich Optional.of(reason)
        return new Invalid(reason);
    }

    boolean isValid();

    Optional<String> getReason();
}
```

Listing 8

```
interface UserValidation extends Function<User, ValidationResult>{
    UserValidation nameIsNotEmpty =
       user -> !user.name.trim().isEmpty()?valid():invalid("Name is empty.")

    UserValidation mailContainsAtSign =
       user -> user.email.contains("@")?valid():invalid("Missing @-sign.");

    default UserValidation and(UserValidation other) {
        return user -> {
            final ValidationResult r = this.apply(user);
            return r.isValid() ? other.apply(user):r;
        };
    }
}

UserValidation v = nameIsNotEmpty.and(mailContainsAtSign);
User gregor = new User("", 31, "mail@mailinator.com");

ValidationResult r = v.apply(gregor);
r.getReason().ifPresent(System.out::println); // Name is empty.
```

Listing 9

```
UserValidation ext = nameIsNotEmpty.and(user -> … );
```

Listing 10

Benutzer-Validierung parallel auf mehreren Benutzern ausgeführt werden kann *(siehe Listing 7)*.

Aus einer anderen Perspektive betrachtet, ist der Typ „UserValidation" ein eigener Kontext mit Elementen aus der Domäne der Benutzer-Validierung. Der Compiler garantiert, dass bei der Beschreibung einer Benutzer-Validierung nur Elemente aus dem „UserValidation"-Kontext benutzt werden. Dies bedeutet den Aufbau einer eingebetteten domänenspezifischen Sprache. Der Code wird deklarativer und orientiert sich an der Domäne.

Das Validierungsergebnis

„Boolean" ist keine gute Wahl, um Validierungsergebnisse zu repräsentieren. In Java ist es nur schwer möglich, fremden Code anzupassen. Der Typ „Boolean" kann nicht um Abfrage-Methoden erweitert werden, um festzustellen, welche Regel das Ergebnis invalidiert. Auch die Bedeutung eines booleschen Werts ist implizit und kontextspezifisch. Zum Beispiel kann die Annahme, dass „true" im Validierungskontext einen validen Benutzer auszeichnet, in einem anderen Kontext falsch sein. Aus diesen Gründen wird ein neuer Typ „ValidationResult" benötigt, der beschreibt, warum ein Benutzer invalide ist *(siehe Listing 8)*.

Ein „ValidationResult" ist entweder „valid" oder „invalid". Die Gleichheit zweier „ValidationResult" ist bestimmt durch die Gleichheit ihrer Felder. Zum Beispiel sind zwei „Invalid"-Instanzen genau dann gleich, wenn sie den gleichen Invalidierungsgrund haben. Im Gegensatz dazu sind valide Ergebnisse nicht anhand ihres Zustands unterscheidbar; hier reicht eine einzelne anonyme Instanz. Das bedeutet, „valid" und „invalid" sind Wert-Objekte. *Listing 9* zeigt, wie „UserValidation" für die Nutzung von „ValidationResult" angepasst wird.

Die Primitiven „nameIsNotEmpty" und „mailContainsAtSign" wurden entsprechend geändert. Der „and"-Kombinator berechnet das Ergebnis von „other" nur, falls „this" valide ist. Zum Beispiel wird die E-Mail eines Benutzers nur geprüft, falls der Name vorhanden ist. Aus Entwicklerperspektive übermittelt die Änderung am API mehr Informationen als ein einfacher boolescher Wert. Jedoch gibt es Makel am Design. Zum Beispiel wäre es in einem Web-Kontext hilfreich, alle Validierungsregeln für einen ausführlichen Bericht zu evaluieren. Der „and"-Kombinator beendet jedoch die Validierung, sobald eine Regel nicht erfüllt wurde. Die Implementierung eines entsprechenden „all"-Kombinators ist dem Leser überlassen.

Vorteile und Herausforderungen

Die Vorteile des Kombinator-Entwurfsmusters sind der domänenspezifische Ansatz, explizit modellierte Informationen, die Trennung von Verantwortlichkeiten, Erweiterbarkeit und Wiederverwendbarkeit.

- **Domänenspezifischer Ansatz**
  Das Beispiel der Benutzervalidierung hebt den domänenspezifischen Ansatz hervor. Regeln und deren Verknüpfungsmöglichkeiten werden in einem Kontext zusammengefasst, eine eingebettete domänenspezifische Sprache wird aufgebaut.

```
Comparator<User> byAge = Comparator.comparing(u -> u.age);
Comparator<User> byName = Comparator.comparing(u -> u.name);
Comparator<User> byAgeAndName =
                 byAge.thenComparing(byName);
Random r = new Random(12);
IntStream.range(0,100)
  .mapToObj(i -> new User("user: "+i,r.nextInt(i+1), "m"+i))
  .sorted(byAgeAndName);
```

Listing 11

- **Explizit modellierte Informationen**
  Das Ergebnis einer Validierung wird explizit modelliert und trägt mehr Informationen als ein integrierter Datentyp wie beispielsweise „Boolean". Zudem sind die einzelnen Validierungsregeln und Verknüpfungsmöglichkeiten klar benannt.
- **Trennung der Verantwortlichkeiten**
  Die Verantwortlichkeiten zwischen Primitiven und Kombinatoren sind klar aufgeteilt. In der Benutzer-Validierung sind Primitive die simpelsten Regeln, die einen Benutzer beschreiben. Kombinatoren sind die Verknüpfungsmöglichkeiten, wie diese Regeln zu komplexeren Regeln kombiniert werden können.
- **Erweiterbarkeit**
  Durch die Verwendung des Kontexts als Zieltyp kann die Benutzervalidierung einfach um eigene Regeln erweitert werden, die nicht im eigentlichen Kontext definiert sind *(siehe Listing 10)*.
- **Wiederverwendbarkeit**
  Einmal beschrieben, kann eine Funktion vielfach angewendet werden. Zum Beispiel können komplexe Validierungsregeln in verschiedenen Diensten verwendet werden.

Die Herausforderungen des Kombinator-Entwurfsmusters sind das Verständnis über Funktionen und die Bestimmung von Primitiven und Kombinatoren in einer Domäne:

- **Verständnis der funktionalen Programmierung**
  Seiteneffekte wie beispielsweise das Werfen von Ausnahmen sind in der funktionalen Programmierung nicht erlaubt. Die Frage ist also, wie eine funktionale Fehlerbehandlung gestaltet werden kann. Solche Fragestellungen treiben die Entwicklung eines funktionalen API und erfordern daher ein funktionales Verständnis.
- **Bestimmung von Primitiven und Kombinatoren**
  Es gibt verschiedene Herangehensweisen, Primitive und Kombinatoren in einer Domäne zu bestimmen. Eine davon nennt sich „algebraisches Design". Hierbei werden auf einer abstrakteren Ebene Typen, Funktionen und deren Eigenschaften bestimmt beziehungsweise diskutiert, bis sie den Anforderungen entsprechen; dann wird die eigentliche Implementierung geschrieben. Eine weitere Herangehensweise ist die „testgetriebene Entwicklung". Hierbei beschreiben Tests das erwartete Verhalten von Funktionen und helfen bei der Bestimmung von Primitiven und Kombinatoren durch Herausarbeitung der Namen und Eigenschaften.

Wann der Einsatz sinnvoll ist

Das Kombinator-Entwurfsmuster hilft bei der Gestaltung eines modularen API. Der Typ „Comperator" ist beispielsweise seit Java 8 auf diese Art aufgebaut und ermöglicht so eine Vielzahl von Kombinationen. *Listing 11* zeigt, wie Benutzer nach Name und Alter sortiert werden.

Um den gleichen Funktionsumfang ohne Kombinator zu erreichen, müsste jede mögliche Kombination an „Comparator" als eigene Methode definiert werden. Unter Berücksichtigung der Unterstützung von primitiven Datentypen wären das mehr als hundert Funktionen. Mit Kombinator sind es hingegen nur 16.

Allgemein ist das Kombinator-Entwurfsmuster sinnvoll, wenn das grundlegende Konzept eine Funktion ist. Ein Beispiel wäre das Parsing. Ein Parser ist eine Funktion, die Zeichen in eine Struktur überführt. Parser für beliebige Grammatiken werden aus primitiven Parser-Funktionen kombiniert. So erkennt eine Parser-Funktion einen Buchstaben und eine andere erkennt Zahlen. Mit einem Kombinator für ein logisches Oder und einem für Vielfache können beliebige alphanumerische Zeichen erkannt werden.

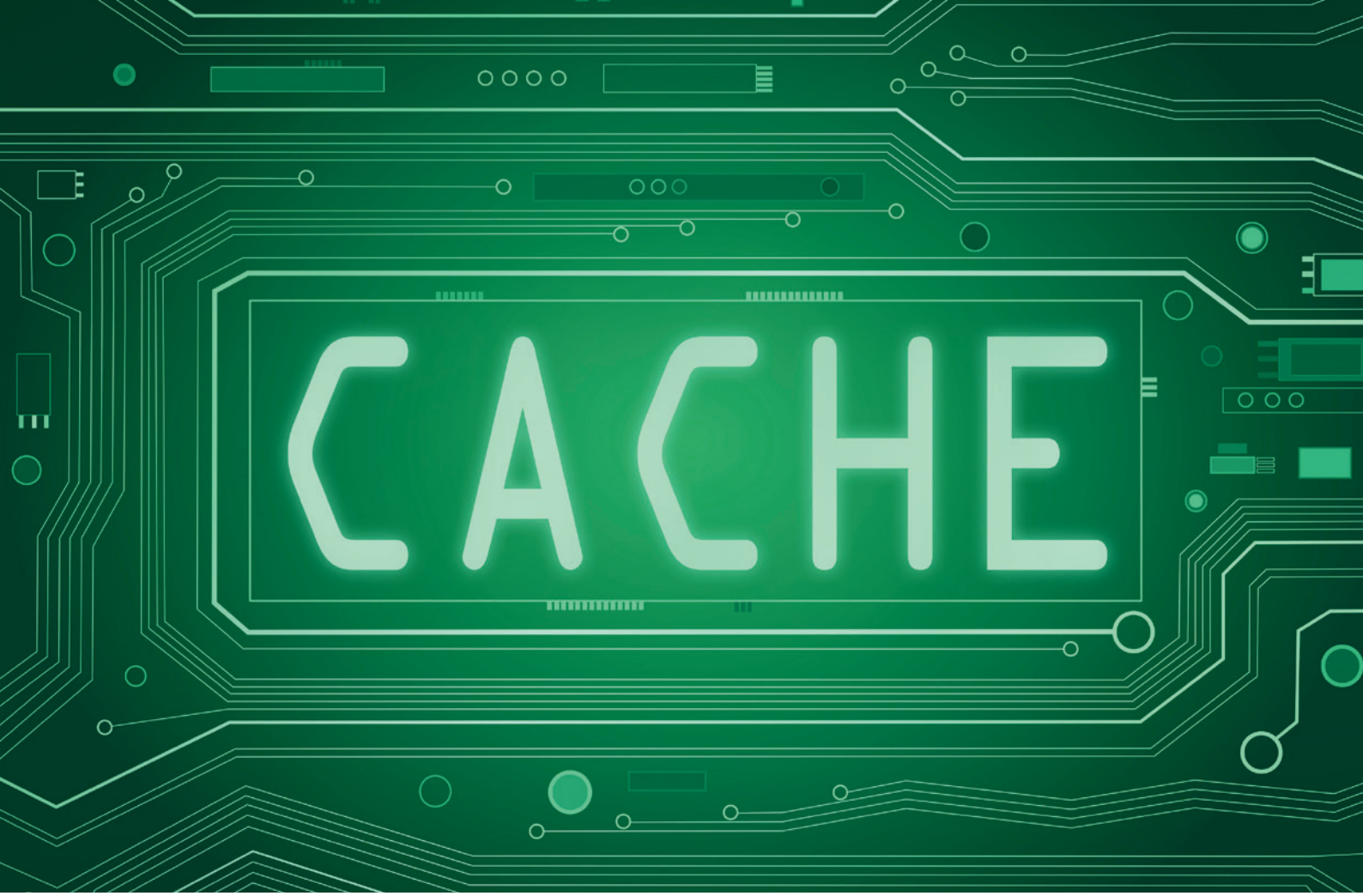
Fazit

Ein gut umgesetztes Kombinator-Entwurfsmuster hat einige Vorteile. Hervorzuheben ist hier die Trennung von Verantwortlichkeiten in einzelne Funktionen, die das Single-Responsibility-Prinzip stärkt. Besonders bei der Gestaltung eines API ist das sinnvoll. Allerdings kann die funktionale Denkweise auch herausfordernd sein. Wie wird beispielsweise Fehlerbehandlung funktional gestaltet? Wie werden die richtigen Primitiven und Kombinatoren bestimmt?

Auch wenn dieser Artikel Lösungsvorschläge für diese Fragestellungen präsentiert hat, liegt es immer noch am Team, wie das Entwurfsmuster eingesetzt wird und ob Alternativen im aktuellen Kontext nicht besser geeignet sind. Dennoch zeigt das Kombinator-Entwurfsmuster, dass es in Java möglich ist, Programme funktional zu gestalten, auch wenn Java selbst keine funktionale Sprache ist. Weitere Informationen unter *„https://github.com/gtrefs/javaaktuell_combinator"*.

Gregor Trefs
gregor.trefs@gmail.com

Gregor Trefs' erstes Programm war ein in BASIC geschriebenes Text-Adventure. Inzwischen ist er Mitorganisator der Java User Group Mannheim (majug) und schreibt regelmäßig auf seinem Blog. Sein persönliches Code-Highlight 2016 war sein Beitrag zu „vavr".



Caching mit Spring über JCache hinaus

Andreas Wirth, Bitech AG

Der Artikel zeigt anhand eines durchgängigen Beispiels die Spring-Cache-Abstraktion analog zu JCache (JSR-107). Zudem wird auf die über JCache hinausgehenden Möglichkeiten des Spring-Cachings eingegangen und demonstriert, wie der Spring-Context innerhalb eines Aspekts verwendet werden kann.

Im Beispiel kommen vor allem die Spring-Framework-Komponenten „spring-context-support" und „spring-test" zum Einsatz. Anschließend sind die Ziele des JSR-107 klar und man kann JCache in einer JEE-Umgebung nutzen. Hinzu kommen einige über JCache hinausgehende Caching-Möglichkeiten, die in der Spring-Cache-Abstraktion geboten werden und in einer Spring-Anwendung verwendet werden können. Die Leser wissen den Spring-Context zielgerichtet innerhalb ihrer eigenen Aspekte zu nutzen, eine über Caching hinausgehende interessante Option.

Caching ist ein altes Thema, aber gerade heutzutage wieder aktuell. In (Micro-) Service-Landschaften ist eine Vielzahl von Remote-Aufrufen notwendig und bei jedem Aufruf stellt sich die Frage, ob hier Caching helfen kann, zu lange Wartezeiten zu vermeiden. Umso erfreulicher, dass mit JCache (JSR-107) das Caching standardisiert ist und Teil von JEE 8 (JSR-366) sein wird. Dabei lohnt sich der standardisierte Ansatz auch für leichtgewichtige Implementierungen.

Definition JCache

JCache definiert Caching so: „In the context of application design it is often used to describe the technique whereby application developers utilize a separate in-memory or low-latency data-structure, a Cache, to temporarily store, or cache, a copy of or reference to information that an application may reuse at some later point in time, this alleviating the cost to re-access or re-create it" *[1]*.

Ebenso wird in der Spezifikation bereits beschrieben, wo JCache eingesetzt werden kann: „Fundamentally any information that is ex-

pensive or time consuming to produce or access can be stored in a cache" *[1]*. Damit spielt Caching sowohl auf Service-Consumer- als auch auf Service-Provider-Seite eine Rolle.

Das 5-Kern-Konzept

JCache basiert auf fünf Kernkomponenten: Über einen CachingProvider wird in der Regel ein CacheManager angefordert, der n Caches verwalten kann. Die jeweiligen Cache-Entries bestehen immer aus einem Key-Value-Paar *(siehe Abbildung 1)*.

Sofern nur ein JCache-konformer CachingProvider im Classpath vorhanden ist, kann dieser über den DefaultClassLoader geladen werden. Der Zugriff auf die anderen vier Kernkomponenten erfolgt entsprechend der Hierarchie *(siehe Listing 1)*.

Die Grundoperationen bezüglich der Cache-Entries können programmatisch oder per Annotation erfolgen *(siehe Tabelle 1)*. Werden Annotations verwendet, muss dafür ein geeigneter Context zur Verfügung stehen. Geeignet sind alle JEE-8-Container (wie GlassFish 5), per Extension können auch CDI-Container-JCache-Annotations berücksichtigt werden *[3]* oder einfach ein Spring-Context – wie später gezeigt. Darüber hinaus definiert JCache weitere Covenient-Methoden *[4]* wie zum Beispiel:

- replace
- containsKey
- unwrap

Weitere Konzepte

Neben den fünf Kernkomponenten definiert JCache einige weitere Standards. Besonders hilfreich findet der Autor folgende:

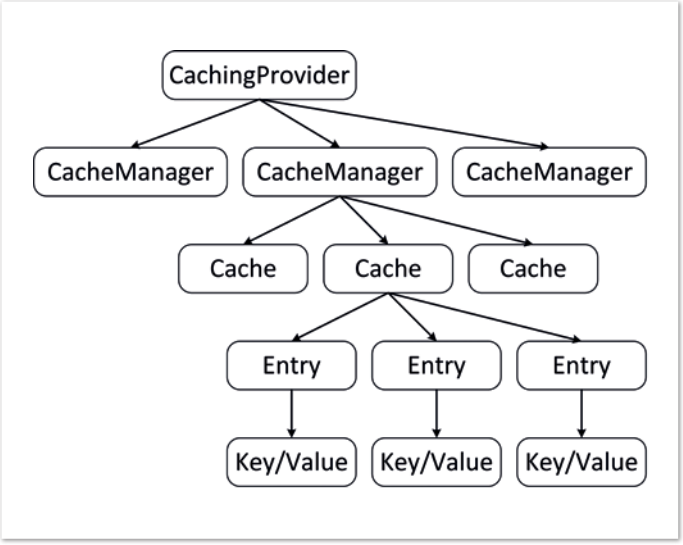


Abbildung 1: JCache Core Concepts [2]

- **Configuration**
  Die wichtigsten Konfigurationen lassen sich standardisiert durchführen. Speziellere implementierungsspezifische Features lassen sich beispielsweise durch Ableitung von „MutableConfiguration" konfigurieren. Wird eine spezielle Konfiguration nur einmalig benötigt, ermöglicht „unwrap" (siehe oben) einen einfachen Zugang zur implementierungsspezifischen Konfiguration.
- **CacheEntryListener**
  Für die „CacheEntry"-Events „create", „expire", „remove" und „update" können Listener definiert werden *(siehe Listing 2)*. Gerade für Logging-Belange ist dies hilfreich.

```
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager cacheManager = cachingProvider.getCacheManager();
MutableConfiguration<Integer, List> conf = new MutableConfiguration<>();
Duration duration = new Duration(TimeUnit.SECONDS, 3L);
conf.setExpiryPolicyFactory(CreatedExpiryPolicy.factoryOf(duration));
Cache<Integer, List> cache = cacheManager.createCache("myCache", conf);
cache.put(key1, val1);
List val2 = cache.get(key2);
```

Listing 1

```
MutableConfiguration<Integer, List> conf = new MutableConfiguration<>();
CacheEntryListenerConfiguration celc = new MutableCacheEntryListenerConfiguration(
  FactoryBuilder.factoryOf(MyCacheEntryListener.class), null, false, false);
conf.addCacheEntryListenerConfiguration(celc);
```

Listing 2

Programmatisch	Annotation	Beschreibung
cache.put	@CachePut	Fügt einen Entry dem Cache hinzu
cache.get	@CacheResult	Falls Key vorhanden, liest einen Entry aus dem Cache; falls Key nicht vorhanden, dann führt „@CacheResult" die Methode aus und fügt das Ergebnis dem Cache hinzu
cache.remove	@CacheRemove	Entfernt einen Entry aus dem Cache
cache.removeAll	@CacheRemoveAll	Entfernt alle Entries aus dem Cache

Tabelle 1: Die Grundoperationen bezüglich der Cache-Entries

JCache	Spring
@CachePut	@CachePut
@CacheResult	@Cacheable
@CacheRemove	@CacheEvict
@CacheRemoveAll	@CacheEvict(allEntries=true)
@CacheDefaults	@CacheConfig

Tabelle 2

- **EntryProcessor**
  EnryProcessor ermöglichen durch Locks, Operationen atomar durchzuführen.

JCache-Implementierungen

Es gibt eine weite Auswahl von JCache-Implementierungen, darunter EhCache3, Infinispan, Hazelcast, Oracle Coherence und Apache Ignite. Alle Listings und einige Features werden in einem durchgängigen Beispiel demonstriert *[5]*. Darin kommen beispielsweise EhCache3 und Infinispan zum Einsatz.

Bevor wir zur Spring-Cache-Abstraktion kommen, ein Blick auf den zeitlichen Ablauf: Der Prozess zu JCache (JSR-107) wurde im Jahr 2001 begonnen und erst im Jahr 2014 abgeschlossen. *Abbildung 2* stellt die Ereignisse von JCache, Java Platform Enterprise Edition (JEE) und Spring-Framework gegenüber.

Spring-Cache-Abstraktion

JCache ist im Jahr 2014 verabschiedet worden und wird im Sommer 2017 in JEE 8 aufgenommen. Mit der Version 3.1 (Ende 2011) wurde im Spring-Framework ein ähnliches Konzept realisiert. Seit der Spring-Version 4.1 (Ende 2015) wird auch JCache in Spring unterstützt. Dabei schlägt „JCacheCacheManager" die Brücke von JCache zur Spring-Cache-Abstraktion *(siehe Listings 3 und 4)*. Mit „<cache:annotation-driven/>" oder „@EnableCaching" lässt sich das Caching per Annotations aktivieren. Analog zu den JCache-Annotations können auch die Spring-Cache-Annotations verwendet werden *(siehe Tabelle 2)*.

Der Mehrwert von Spring

Das Spring-Framework bietet dabei einige über JCache hinausgehende Möglichkeiten *[6]:*

- **synchronized caching „@Cacheable(sync=true)"**
  Auch in Multithreading-Umgebungen ist damit sichergestellt, dass die Methode nur einmalig pro Key ausgeführt wird
- **conditional caching „@Cacheable(condition=SpEL, unless=SpEL)"**
  Je nach Input-Parameter („condition") oder Output-Parameter („unless") kann entschieden werden, ob gecacht werden soll
- **key generator „@Cacheable(keyGenerator=REF)"**
  Keys können programmatisch definiert sein
- Adapter für Nicht-JCache-konforme Implementierungen wie EhCache2, Caffeine etc.

Das ist zu berücksichtigen

Wie in Spring üblich, wird Caching per Proxy realisiert. Dabei ist der Standard-Modus JDK-basiert („<cache:annotation-driven mode="proxy"/"> beziehungsweise „EnableCaching(mode=AdviceMode.PROXY)"). Dementsprechend lassen sich nur „public"-Methoden annotieren. Sofern mehrere Advices an demselben

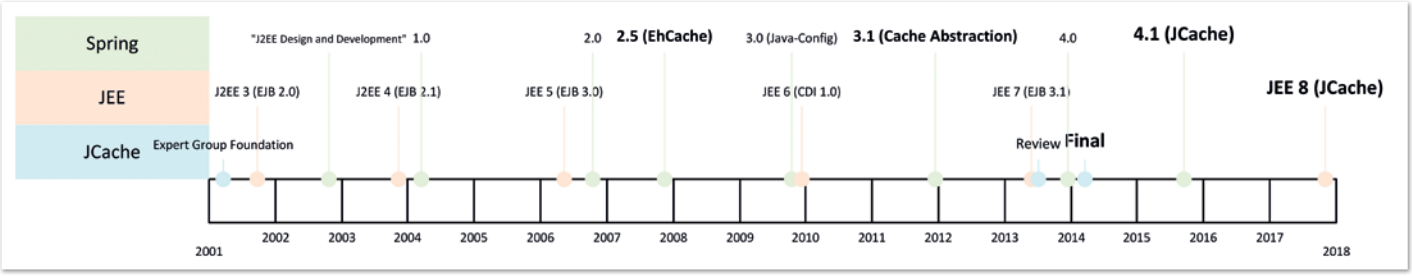


Abbildung 2: JCache-Timeline

```
<bean id="cacheManager" class="org.springframework.cache.jcache.JCacheCacheManager">
  <constructor-arg ref="jCacheManager"/>
</bean>
<bean id="jCacheManager" factory-bean="jCacheFactoryBean" factory-method="buildJCacheCacheManager"/>
```

Listing 3

```
public CacheManager buildJCacheCacheManager() {
  CachingProvider cachingProvider = Caching.getCachingProvider();
  CacheManager ret = cachingProvider.getCacheManager();
  MutableConfiguration<String, List<Integer>> configuration = new MutableConfiguration<>();
  configuration.setExpiryPolicyFactory(AccessedExpiryPolicy.factoryOf(new Duration(TimeUnit.HOURS, 1L)));
  ret.createCache(CACHE_NAME, configuration);
  return ret;
}
```

Listing 4

```
@Autowired
@Qualifier("cacheManager")
private CacheManager cacheManager;

private Cache myCache;

@PostConstruct
public void init() {
  this.myCache = this.cacheManager.getCache("myCache");
}

@Pointcut("execution(* *.*(..))")
public void allExecutions() {
  // NOP
}

  @Pointcut("@annotation(de.bitech.javaday.spring.cache.aspect.MyCache)")
public void myCache() {
  // NOP
}

@Pointcut("allExecutions() && myCache()")
public void execMyCache() {
  // NOP
}

@Around("execMyCache()")
public Object aroundExecMyCache(ProceedingJoinPoint pjp) throws Throwable {
  Object ret = null;
  Object[] params = pjp.getArgs();

  // condition
  int number = (int) params[0];
  if (number <= 1000000) {
     ret = pjp.proceed();
  } else {
     ValueWrapper valueWrapper = this.myCache.get(number);
     if (valueWrapper == null) {
        ret = pjp.proceed();
        this.myCache.put(number, ret);
     } else {
        ret = valueWrapper.get();
     }
  }
  return ret;
}
```

Listing 5

```
@MyCache
private List<Integer> fractionize(int pNumber) {
  [...]
```

Listing 6

```
<bean class="de.bitech.javaday.spring.cache.aspect.CacheAspect" factory-method="aspectOf"/>
```

Listing 7

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/de/bitech/javaday/spring/context-cache.xml")
@DirtiesContext
public class MyTest {
```

Listing 8

Joinpoint definiert sind, muss gegebenenfalls die Ausführ-Reihenfolge definiert sein.

Falls die eingebauten Möglichkeiten („@EnableCaching(order=Ordered.LOWEST_PRECEDENCE)“) nicht ausreichen, kann der Proxy-Modus auf AspectJ geändert werden („@EnableCaching(mode=AdviceMode.ASPECTJ)“). Nun können auch „private“-Methoden annotiert sein, sodass die Aufruf-Reihenfolge in Form der Reihenfolge der Methoden festgelegt ist. Als positiver Nebeneffekt wird der Quellcode damit auch leichter lesbar. Allerdings muss „aop“-typische Instrumentation aktiviert, also „spring-instrument“ als Dependency dem Classpath hinzugefügt und per „<context:load-time-weaver aspectj-weaving="on"/>“ AspectJ-Weaver aktiviert sein.

Eigener Caching-Aspekt

Für Annotations an privaten Methoden benötigt man Bytecode-Manipulation, etwa durch AspectJ. Anstelle von Load-Time-Weaving hätte der Autor lieber Compile-Time-Weaving, damit entfällt Instrumentation. Also kann man einen eigenen Aspekt schreiben, der innerhalb seines Advice das Caching programmatisch ausführt und die Joinpoints per Annotation deklariert *(siehe Listings 5 und 6)*.
Der Aspekt lebt in der „aspectj-runtime“, aber wie kann er auf den Spring-Context zugreifen? Dazu deklariert man mit der „factory-method aspectOf“ den Aspekt als „factory-bean“ und Dependency Injection à la Spring funktioniert im Aspekt ebenfalls *(siehe Listing 7)*.

Ein letztes Problem muss noch gelöst werden: Tests, die den Spring-Context benötigen, erzeugen per Default nur einmalig den Spring-Context. AspectJ erzeugt aber die Aspekte für jeden Durchlauf neu. Mit „@DirtiesContext“ kann man auch Spring dazu veranlassen, in jedem Durchlauf einen frischen Context zu erzeugen *(siehe Listing 8)*.

Fazit

JCache bietet mit seinen fünf Kern-Konzepten ein einheitliches und einfaches Caching-API sowohl programmatisch als auch per Annotation an. Ebenso vereinheitlicht ist die Basiskonfiguration der JCache-Manager. Speziellere Konfigurationen können mit den herstellerspezifischen APIs umgesetzt werden und trotzdem kann der Spring „JCacheCacheManager“ verwendet werden.

Die Caching-Strategie ist nicht Bestandteil von JCache und in jedem Fall zu berücksichtigen. Abhängig von Anzahl und Größe der zu cachenden Objekte ist zu entscheiden, ob der Cache „embedded“ oder „standalone“ realisiert wird – je nach Applikation, also je nachdem, ob der Cache verteilt oder repliziert ausgeprägt sein sollte.

Die Spring-Cache-Abstraktion kann zu 100 Prozent JCache-konform eingesetzt werden. Über die Spring-spezifischen Annotations kommen einige kleine, aber feine zusätzliche Funktionen hinzu. Insbesondere „synchronized“ und „conditional“ Caching bieten in der Praxis einen Mehrwert. Auch die feingranularere Steuerung über AOP-Proxies kann hilfreich sein.

Die meisten Möglichkeiten bietet ein eigener Aspekt. Dieser Artikel zeigt, wie man den Spring-Context einfach innerhalb eines Aspekts verwenden kann, sodass der Mehraufwand dieser Lösung überschaubar bleibt. Gerade auch in Szenarien mit „distributed“ oder „standalone“ Caches bietet ein eigener Aspekt ein einheitliches API innerhalb der Applikation und eine zentrale Stelle für die Caching-Strategie.

Weitere Informationen

[1] *Greg Luck, Brian Oliver: JSR107FinalSpecification 1.2: https://jcp.org/en/jsr/detail?id=107*
[2] *Abhishek Gupta: https://dzone.com/refcardz/java-caching*
[3] *Jonathan Gallimore: http://www.tomitribe.com/blog/2015/06/using-jcache-with-cdi*
[4] *http://static.javadoc.io/javax.cache/cache-api/1.0.0/index.html*
[5] *https://github.com/awi72/spring-jcache*
[6] *Spring Framework Reference Documentation (4.3.6.Release): http://docs.spring.io/spring/docs/current/spring-framework-reference*
[7] *http://hazelcast.com/content/Using-JCache-with-Hazelcast-Slides.pdf*
[8] *http://www.ehcache.org/documentation/3.0/107.html*
[9] *http://www.ehcache.org/blog/2016/05/18/ehcache3_jsr107_spring.html*
[10] *https://www.slideshare.net/SpringCentral/spring-one2gx-caching-with-spring?qid=8fd5ecbc-f5e7-4c84-b984-ce4aa5a9fd4e&v*

Andreas Wirth
andreas.wirth@koeln.bitech.de

Andreas Wirth arbeitet als IT-Berater bei der Bitech AG in Köln. Er unterstützt seine Kunden als Entwickler, Architekt oder Coach in der Java-Enterprise-Entwicklung. Darüber hinaus setzt er sich in seinen Projekten für „clean code“ ein.

Wir bauen uns eine Monade – Railway Oriented Programming statt Exception-Handling

Stefan Macke, ALTE OLDENBURGER Krankenversicherung AG

Dieser Artikel zeigt an praktischen Beispielen den inzwischen alltäglichen Einsatz von Monaden in Java 8. Dieses Konzept aus der funktionalen Programmierung erleichtert den Umgang mit teilweise komplexen Problemen wie der asynchronen Programmierung. Zusätzlich wird eine eigene Monade entwickelt, die eine Alternative zur Verwendung von Exceptions zur Fehlerbehandlung bietet.

Monaden sind vielen objektorientierten Softwareentwicklern nur als kryptisches Konstrukt der funktionalen Programmierung bekannt. Dabei arbeiten auch Java-Entwickler inzwischen fast täglich damit, wenn sie die neuen Funktionen in Java 8 einsetzen. So zählen beispielsweise „Optional" und „Stream" zur Gruppe der monadischen Datenstrukturen. Sie sind also alles andere als exotisch, sondern helfen uns bei der täglichen Arbeit.

Darüber hinaus ist es auch gar nicht so schwierig, sich selbst eigene Monaden zu programmieren. Dies werden wir in diesem Artikel tun. Analog zum „Optional", das die Behandlung von „null"-Werten verhindern soll, wird eine Monade programmiert, die die Fehlerbehandlung mit Exceptions ersetzen kann. Der resultierende Code wird ein wenig funktionaler, einfacher zu testen und vielleicht sogar etwas verständlicher. Die Idee hinter diesem Ansatz heißt „Railway Oriented Programming". Doch zunächst eine kurze Einführung in Monaden.

Monaden im Java-Alltag

Wer schon einmal Code wie den in *Listing 1* geschrieben hat, verwendet bereits zwei Monaden: „Optional" und „Stream". Im Beispiel werden Benutzer anhand ihrer Benutzernamen aus einer Datenbank gelesen. Dabei wird geprüft, ob die jeweiligen Benutzer über-

haupt vorhanden sind. Zuletzt werden dann die Namen der vorhandenen Benutzer auf der Konsole ausgegeben. Interessant an diesem Code ist, dass weder eine explizite Iteration zu sehen ist, noch „null" als Repräsentation eines nicht vorhandenen Benutzers verwendet wird. Stattdessen wird das Durchlaufen der Liste dem „Stream" überlassen und „Optional" als Rückgabewert des Repository macht deutlich, dass es gegebenenfalls zum angegebenen Benutzernamen keinen Benutzer gibt.

Die beiden Datenstrukturen repräsentieren zwei allgemeine Konzepte: das Vorhandensein mehrerer Werte, über die iteriert werden kann, und das Nicht-Vorhandensein eines Werts. Zusätzlich stellen sie Funktionen bereit, die den Entwicklern die Arbeit mit beiden Konzepten einfacher machen: Es ist keine explizite Iteration mehr nötig und „null"-Checks sind überflüssig.

Java 8 bringt gleich mehrere solcher Datenstrukturen mit. Sie umschließen jeweils einen zugrunde liegenden Datentyp (im Beispiel „User") mit einem operationalen Kontext, dessen teilweise recht hohe Komplexität dadurch vor dem Aufrufer verborgen bleibt. Stattdessen kann die einheitliche Schnittstelle dieser Monaden – zum Beispiel „map()"und „filter()" – genutzt werden, um unterschiedliche Konzepte im Code sehr ähnlich zu verwenden. Folgende Liste zeigt drei bekannte Monaden in Java 8 und beschreibt, welches allgemeine Konzept sie repräsentieren;

- **Optional**
  Repräsentiert das Nicht-/Vorhandensein eines Werts und vermeidet „null"-Checks
- **Stream**
  Repräsentiert potenziell mehrere Werte und vermeidet explizite Iterationen
- **CompletableFuture**
  Repräsentiert das Ergebnis einer asynchronen Operation und vermeidet Thread-Programmierung

Monaden im Java-Alltag

Untersucht man die vorgestellten Datenstrukturen etwas genauer, stellt man fest, dass sie sich gewisse Eigenschaften teilen. Alle Monaden haben eine gemeinsame Basis an Funktionen, die sie anbieten müssen:

- Einen Typ-Konstruktor, der die Monade mit einem Datentyp parametrisiert. Im Beispiel sind das die generischen Klassen „Optional<T>" beziehungsweise „Stream<T>".
- Eine Funktion, die aus einem normalen Datentyp eine Monade dieses Datentyps macht, ihn also quasi in der Monade verpackt. Diese Funktion heißt in funktionalen Programmiersprachen wie Haskell häufig „unit()". Die Methode „Optional.of("Mein Text")" macht beispielsweise aus dem „String" mit dem Wert „Mein Text" ein „Optional<String>".
- Eine Funktion, die eine andere Funktion als Parameter bekommt, die aus dem inneren Datentyp der Monade eine neue Monade eines beliebigen Datentyps erzeugt und diese neue Monade zurückgibt. In funktionalen Sprachen heißt diese Funktion beispielsweise „bind()", in Java „flatMap()".

Zum dritten Punkt ein Beispiel: „Optional.of("Mein Text"). flatMap(text –> Optional.of(text.length()))" liefert als Ergebnis ein

```
List<String> usernames = new ArrayList<>();
usernames.add("user1");
usernames.add("user2");
usernames.add("user3");

UserRepository repo = new UserRepository();

usernames.stream()                    // Stream<String>
    .map(repo::findUserByUsername)    //
Stream<Optional<User>>
    .filter(Optional::isPresent)
    .map(Optional::get)               // Stream<User>
    .map(User::getName)               // Stream<String>
    .forEach(System.out::println);
```

Listing 1: Einsatz der Monaden „Optional" und „Stream" im Java-Code

„Optional<Integer>". Die an „flatMap()" übergebene Funktion (der Lambda-Ausdruck) bekommt einen „String" als Parameter und liefert ein „Optional<Integer>" zurück. „flatMap()"nimmt also den im „Optional" enthaltenen „String" mit dem Wert „Mein Text", wendet die übergebene Funktion „Optional.of(text.length())" darauf an, erhält als Ergebnis ein „Optional<Integer>", das die Länge des Strings („9") enthält, und gibt dieses dann zurück.

Das Vorhandensein dieser Funktionen allein reicht jedoch noch nicht aus, um von einer echten Monade sprechen zu können. Die Funktionen müssen zusätzlich folgende Gesetze einhalten:

- Die „unit()"-Funktion muss das linksneutrale Element der „bind()"-Funktion sein.
- Die „unit()"-Funktion muss das rechtsneutrale Element der „bind()"-Funktion sein.
- Die „bind()"-Funktion muss assoziativ sein.

Diese recht mathematischen Definitionen sind für die Theorie hinter den Monaden sicherlich sehr wichtig, für ihren Praxiseinsatz muss man allerdings nicht unbedingt jedes Detail verstehen. Da dieser Artikel keine wissenschaftliche Einführung in die mathematische Kategorien-Theorie ist, in der die Monaden ihren Ursprung haben, sondern ihre Vorteile in der Programmierpraxis zeigen soll, sei daher an dieser Stelle auf eine weitere Vertiefung der Monadengesetze verzichtet. Wer Interesse an den mathematischen Hintergründen hat, dem ist der entsprechende Wikipedia-Artikel *(siehe „https://en.wikipedia.org/wiki/Monad_(functional_programming)")* zum Einstieg empfohlen.

Wichtig ist an dieser Stelle noch der Hinweis, dass die hier vorgestellten Datenstrukturen eventuell keine echten Monaden im funktionalen Sinne sind. Insbesondere „Optional" bricht mindestens eines der drei Gesetze. Eine gute Erklärung liefert Marcello La Rocca in seinem Artikel „How Optional Breaks the Monad Laws and Why It Matters" *(siehe „https://www.sitepoint.com/how-optional-breaks-the-monad-laws-and-why-it-matters/").* Dort sind auch die Monadengesetze noch einmal im Detail erklärt.

Monaden statt Exceptions

Bereits im Jahr 2003 schrieb Joel Spolsky, der Gründer von StackOverflow, in einem Blog-Artikel *(siehe „https://www.joelonsoftware.com/2003/10/13/13/"),* dass er selbst niemals Exceptions verwen-

```
usernames.stream()
    .map(repo::findUserByUsername)  // Unhandled exception type DatabaseException
    .map(User::getName)
    .forEach(System.out::println);
```

Listing 2: Exceptions und Streams vertragen sich nicht

den würde. Sie seien im Quelltext unsichtbar, da man einer Methode nicht ansehen könne, ob sie eine Exception werfe (in Java gilt das natürlich nur für Unchecked Exceptions). Daher sei es für Leser des Codes schwierig zu erkennen, ob ein Fehler auftreten kann oder nicht. Außerdem ähnelten sie dem berüchtigten „goto"-Statement, da nicht offensichtlich sei, zu welcher Stelle im Code nach dem Auftreten einer Exception gesprungen würde.

Als Alternative schlägt Spolsky die Verwendung von Rückgabewerten vor, die auf einen potenziellen Fehler hinweisen. Diesen Vorschlag greifen wir nun auf und entwickeln eine monadische Datenstruktur, die den potenziell fehlerhaften Ausgang einer Funktion repräsentiert. Wir können die obige Liste somit um folgenden Punkt erweitern:

- **Result**
  Repräsentiert das potenziell fehlerhafte Ergebnis einer Operation und vermeidet Exception-Handling

Der Code in *Listing 2* greift das obige Beispiel wieder auf und erweitert es um eine Checked Exception in „UserRepository.findUserByUsername()". Da diese Methode auf eine Datenbank zugreift, die ausfallen kann, wirft sie nun eine „DatabaseException". Das macht ihre direkte Verwendung im Stream leider unmöglich, da „map()" keine Methoden mit Checked Exceptions erlaubt.

Listing 3 zeigt eine mögliche Lösung des Problems: Die Checked Exception wird gefangen und in eine „RuntimeException" verpackt. Der Code wird dadurch allerdings nicht gerade verständlicher. Die Schachtelungstiefe steigt und ein komplexer Lambda-Ausdruck wird programmiert, wo vorher nur eine einzige Zeile Code nötig war.

Eine Alternative zu obigem Vorgehen steht in *Listing 4*. Statt eine Checked Exception zu werfen, gibt „findUserByUsername()" nun ein „Result<User>" zurück. Diese noch recht rudimentäre Implementierung der Idee lässt sich analog zu „Optional" im ersten Beispiel verwenden und verhindert bereits die umständliche Fehlerbehandlung

```
usernames.stream()
    .map(username -> {
        try
        {
            return repo.findUserByUsername(username);
        }
        catch (DatabaseException e)
        {
            throw new RuntimeException(e);
        }
    })
    .map(User::getName)
    .forEach(System.out::println);
```

Listing 3: Eine mögliche, aber unschöne Lösung

aus *Listing 3*. Allerdings ist als Fehlertyp nur ein „String" möglich und auch die Monaden-Funktionen wie „flatMap()" fehlen noch. Außerdem wird keine interne Fehlerbehandlung durchgeführt und „null" als Repräsentation fehlender Werte ist auch nicht die beste Idee.

Railway Oriented Programming

Man kann und sollte die Klasse „Result" noch um die fehlenden Methoden und einen weiteren Typ-Parameter für den Fehlerfall ergänzen. Da dies jedoch den Rahmen dieses Artikels sprengen würde, sei auf das Projekt „ao-railway" bei GitHub *(siehe „https://github.com/StefanMacke/ao-railway")* verwiesen. Dort stellt der Autor eine komplette Implementierung des Konzepts bereit, die zusätzliche Methoden enthält, die den Umgang mit Fehlern weiter vereinfachen. *Listing 5* zeigt als Beispiel das Ändern eines Benutzer-Passworts. In wenigen Zeilen sprechenden Codes (dank des Einsatzes eines Fluent-API *(siehe „http://martinfowler.com/bliki/FluentInterface.html")* wird die komplette Geschäftslogik inklusive Fehlerbehandlung implementiert.

Der Name des Projekts „ao-railway" ist eine Referenz auf die Idee, den Ablauf eines Programms analog zu Bahnschienen in einen Erfolgs- („onSuccess()") und einen Fehlerweg („onFailure()") einzu-

```
usernames.stream()                  // Stream<String>
  .map(repo::findUserByUsername)    //
Stream<Result<User>>
  .filter(Result::isSuccess)
  .map(Result::getValue)            // Stream<User>
  .map(User::getName)               // Stream<String>
  .forEach(System.out::println);

public class Result<T> {
  private T value;
  private String error;

  private Result(T value, String error) {
     this.value = value;
     this.error = error;
  }

  public static <T> Result<T> withValue(T value) {
     return new Result<T>(value, null);
  }

  public static <T> Result<T> withError(String error)
{
     return new Result<T>(null, error);
  }

  public boolean isSuccess() {
     return error == null;
  }

  public T getValue() {
     return value;
  }
}
```

Listing 4: „Result" als Rückgabewert statt Exception

```
public Result<User> changePassword(
    Username username, Password oldPasswort, Password newPassword) {
    return Result.combine(
        Result.of(username, "Username cannot be empty"),
        Result.of(oldPassword, "Old password cannot be empty"),
        Result.of(newPassword, "New password cannot be empty"))
    .onSuccess(() -> userRepo.find(username))
    .ensure(user -> user.isCorrectPassword(oldPassword), "Invalid password")
    .onSuccess(user -> user.changePassword(newPassword))
    .onSuccess(user -> userRepo.update(user))
    .onFailure(() -> logger.error("Password could not be changed"));
}
```

Listing 5: „ao-railway" im Praxiseinsatz

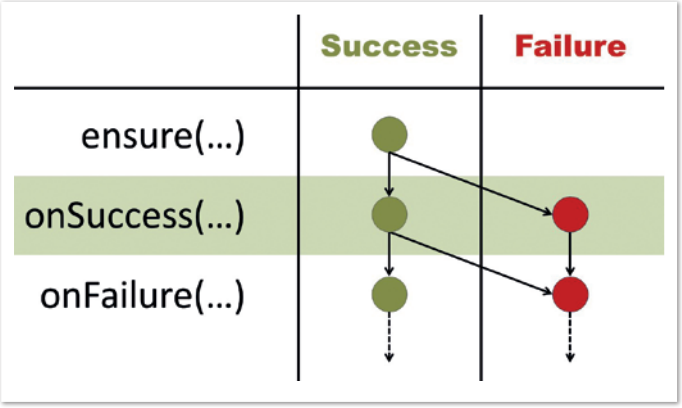


Abbildung 1: Programmfluss beim Railway Oriented Programming

teilen. Sie stammt von Scott Wlaschin, der sie bei der funktionalen Softwareentwicklung mit F# einsetzt. In einem Video zeigt er ausführlich ihren Praxiseinsatz *(siehe „https://vimeo.com/97344498")*.

Sobald an irgendeiner Stelle der Aufrufkette vom erwarteten Verhalten abgewichen wird (etwa weil das alte Passwort nicht gültig oder ein Datenbank-Fehler aufgetreten ist), soll der restliche Code nicht mehr durchlaufen werden. Der Fehlerfall führt das Programm also quasi wie bei einer Weiche auf eine andere Bahnschiene, die parallel zum normalen Programmfluss verläuft, aber eben komplett daran vorbeiführt und bis zum Ende der Befehlskette nicht wieder verlassen werden kann *(siehe Abbildung 1)*.

Die Klasse „Result" übernimmt dabei die gesamte Steuerung des Programmflusses, indem sie je nach internem Zustand (Erfolg oder Fehler) die ihr übergebenen Operationen ausführt oder nicht. Damit muss sich der Aufrufer nicht mehr um die Fehlerbehandlung kümmern, sondern nur noch entscheiden, welche Aktionen in den beiden Zuständen ausgeführt werden sollen.

Fazit

Dieser Artikel zeigt, dass Monaden nicht nur abstrakte Gebilde aus der funktionalen Programmierung sind, sondern auch bei der alltäglichen Arbeit in Java eingesetzt werden können und dabei auch konkrete Vorteile für den Entwickler mitbringen. Sie abstrahieren von teilweise sehr komplexen Problemen wie der asynchronen Programmierung mit Threads und machen den Code oftmals kompakter und verständlicher für den Leser.

Java 8 bringt bereits eine Menge monadischer Datenstrukturen mit, auch wenn diese gegebenenfalls nicht alle Monadengesetze einhalten. Dennoch erleichtern sie uns an vielen Stellen die Programmierung und sind aus modernem Java-Code kaum mehr wegzudenken. Auch das Erstellen einer eigenen Monade ist kein Hexenwerk. Das Beispiel der Klasse „Result" zeigt, wie einfach der Einstieg in die eher funktionale Problemlösung sein kann. Die komplette Klasse *(siehe „https://github.com/StefanMacke/ao-railway/blob/master/src/main/java/net/aokv/railway/result/Result.java")* inklusive zusätzlicher Methoden für das Fluent-API im Sinne des Railway Oriented Programming ist lediglich etwa hundert Zeilen lang (ohne Kommentare, Umbrüche etc.) und für erfahrene Java-Entwickler gut zu verstehen.

Viele objektorientierte Programmiersprachen lassen funktionale Konzepte in ihre neuen Versionen einfließen. Zuletzt wurden in Java 8 Lambda-Ausdrücke und Streams hinzugefügt sowie Funktionen zu Bürgern erster Klasse erhoben. Das sollte für alle Java-Entwickler ein Anreiz sein, sich intensiver mit den Ideen der funktionalen Programmierung auseinanderzusetzen. Dazu ist kein abgeschlossenes Mathematikstudium nötig, sondern nur das Interesse für neue Wege der Problemlösung. Monaden bilden dabei eine wichtige Grundlage. Vielleicht regt dieser Artikel dazu an, die altbewährten Exceptions abzulösen oder ihren Einsatz zumindest zu überdenken. Schließlich verwenden wir ja auch schon lange keine „goto"-Statements mehr.

Stefan Macke
mail@anwendungsentwicklerpodcast.de

Stefan Macke ist Software-Entwickler und -Architekt bei der ALTE OLDENBURGER Krankenversicherung AG aus Vechta. Seit dem Jahr 2007 ist er dort außerdem Ausbilder für Anwendungsentwickler und seit dem Jahr 2009 IHK-Prüfer in Oldenburg. Seine Erfahrungen in der Ausbildung kommuniziert er in seinem Podcast unter „http://anwendungsentwicklerpodcast.de". In seinen aktuellen Projekten beschäftigt er sich vor allem mit der Modernisierung von Alt-Anwendungen auf Basis einer serviceorientierten Architektur mithilfe von Java.

Hysterie in verteilten Systemen – Hystrix im Einsatz

Benjamin Wilms, codecentric AG

Ob man sich nun in einer modernen Microservice-Architektur bewegt oder an bestehende Legacy-Backends binden muss, in beiden Fällen ist mit dem Verhalten und den Fehlern dieser Systeme zu leben und damit umzugehen. Das Hystrix-Framework erfreut sich immer größerer Beliebtheit. Service-Aufrufe werden isoliert, um die eigene Anwendung vor den Fehlern ihrer Abhängigkeiten zu schützen. Timeouts und Exceptions, die zum Absturz der eigenen Anwendung führen können, werden verhindert.

Das Thema „Resilient Software" findet immer mehr Aufmerksamkeit und wird dabei von mehreren Global Playern wie zum Beispiel Netflix stetig vorangetrieben. Der Artikel geht unter anderem der Frage nach, was eine resiliente Software ist und wie man diese erreichen kann.

Eines der obersten Ziele ist es, eine fehlerfreie und vor allem immer korrekt arbeitende Applikation zu entwickeln. Der Anwender sollte von technischen Fehlern nichts mitbekommen, im schlimmsten Fall kann ein sehr aufmerksamer Anwender dies nur an einem langsam reduzierten Service-Level merken. Dabei spricht man von „graceful degradation", das System hält den Betrieb so weit als möglich aufrecht und reduziert nur die Funktionalität. Dies hört sich im ersten Moment einfach an, bedeutet aber auch, dass technische Fehler Teil der Fachlichkeit werden, was eine starke Zusammenarbeit mit den Verantwortlichen für das fachliche Verhalten der Applikation erfordert. So sind Fragen nach möglichen Fallbacks, statischen Defaults und/oder finalen Fehlerseiten zu klären.

Um die genannte Stabilität zu erreichen, werden wir uns einiger Projekte aus dem Hause Netflix bedienen und anhand derer eine Beispielanwendung erstellen. Die in diesem Artikel beschriebene Anwendung ist auf GitHub unter *„https://github.com/MrBW/resilient-transport-service"* verfügbar und kann gerne für erste Erfahrungen mit Hystrix und Co. verwendet werden. Alle notwendigen Informationen zum Aufsetzen der Demo-Anwendung sind dort dokumentiert.

Transport my Package

Die genannte Beispiel-Anwendung ist in der Lage, einen Transportauftrag entgegenzunehmen und einen sogenannten „Booking-Request" zu verarbeiten. Anhand einer Kundennummer wird ermittelt,

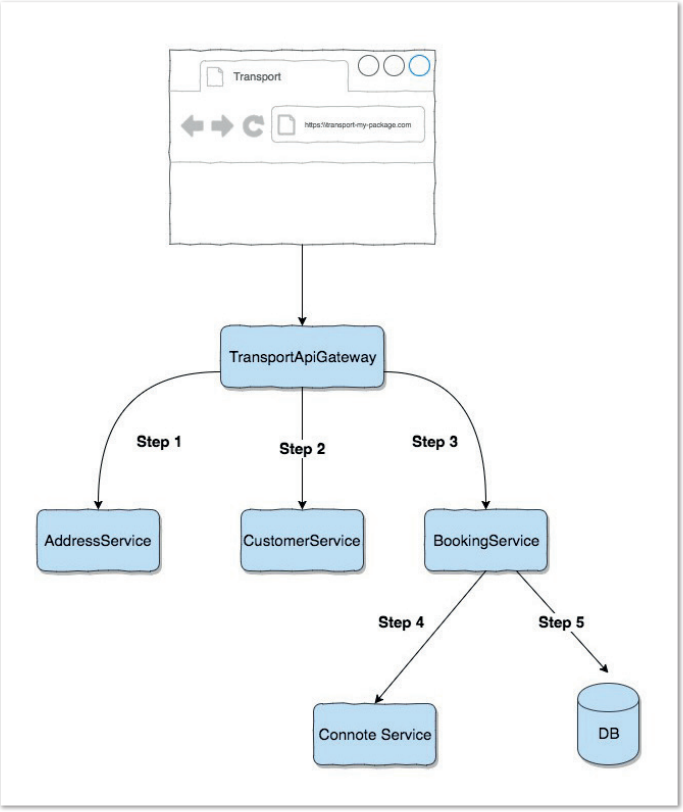


Abbildung 1: Architektur-Diagramm

ob es diesen Kunden gibt, ob die Abhol- und Zustelladresse valide ist und ob diese von unserem fiktiven Unternehmen „Transport my Package" angefahren werden kann. Im Anschluss an diese Schritte wird eine Frachtbriefnummer („Connote") generiert und eine finale Buchung ausgelöst *(siehe Abbildung 1)*.

Gehen wir zunächst die notwendigen Schritte eines Booking Request durch. Schritt 1 ist die Validierung der Abhol- und Zustelladresse und Schritt 2 die Überprüfung des angemeldeten Kunden. Beide Schritte werden parallel ausgeführt und liefern die benötigten Daten, um mit Schritt 3 fortzufahren. Nur bei einem Erfolg beider Schritte kann die eigentliche Buchung durchgeführt werden.

Im Schritt 3 übernimmt der Booking Service die zuvor in den Schritten 1 und 2 ausgelesenen und validierten Daten und bereitet eine Buchung vor. Schritt 4 ist die Erzeugung einer eindeutigen Frachtbriefnummer, unter der die Sendung geführt wird. Im Schritt 5 erfolgt das Fortschreiben der Buchung, wenn eine Frachtbriefnummer erzeugt werden konnte. Im Fehlerfall wird eine entsprechende Meldung zurückgegeben.

Architektur

Jede Komponente ist dabei eine schlanke Spring-Boot-Applikation, die in einem eigenen Docker-Container betrieben wird. Dadurch ist es möglich, die Skalierung als eine mögliche Fallback-Strategie bereits während der Entwicklung zu verwenden. Hystrix versetzt einen in die Lage, für jeden Service-Call einen entsprechenden Fallback bereitzustellen und den eigentlichen Service-Aufruf von Hystrix absichern zu lassen.

In diesem Zusammenhang wird auch immer gerne von „Bulkheads" (Schotts) gesprochen, in die man eine Applikation aufteilen kann, um kaskadierende Fehler zu verhindern. Diese Aufgabe übernimmt Hystrix indirekt, da man auf technische Fehler und definierte Timeouts reagieren kann. Somit sind diese mit Hystrix abgesicherten Bereiche isoliert, was die Stabilität des Gesamtsystems deutlich erhöht.

Reicht Hystrix für die Absicherung und das Definieren von Bulkheads aus? Die Antwort lautet ganz klar – nein! Bulkheads sind eines der elementarsten Resilience-Pattern, wenn nicht gar das elementarste.

Die meiste Zeit bei der Entwicklung eines neuen Systems wird dafür verwendet, die fachlichen Verantwortlichkeiten zu ermitteln und zu verstehen. Erst mit diesem Wissen ist es möglich, die einzelnen Bausteine (Module) so zu schneiden, dass keine langen Aufrufketten entstehen und die Module möglichst unabhängig voneinander arbeiten können. Nun ein Blick auf die beteiligten Akteure zur Absicherung der Beispielanwendung.

Hystrix

Hystrix wurde von Netflix entwickelt und als eines von etwa dreißig Open-Source-Projekten veröffentlicht; es erfährt in den letzten Jahren eine immer größere Beliebtheit. Wann und wofür man Hystrix verwenden soll, ist eine der ersten Fragen, die es zu beantworten gilt.

Hystrix kann mit Java, Java EE und Spring eingesetzt werden und wird als Dependency im Projekt eingebunden. Egal in welcher Galaxie des Java-Universums man sich aufhält, es gibt immer die drei Möglichkeiten, Hystrix im Code zu verankern.

Als erste Variante kommt das Command-Pattern *(siehe „https://github.com/Netflix/Hystrix/wiki/How-To-Use")* zum Einsatz, wobei man für jeden externen Service-Aufruf ein eigenes Command erstellt und dort seinen Service-Aufruf mit passendem Fallback bereitstellt. Die Fallbacks sind die hohe Kunst des Ganzen und müssen mit der Fachlichkeit zusammen entwickelt werden. Hier trifft die Fachlichkeit auf technische Fehler, die ihr vom Entwickler erklärt werden und die zu einem definierten Fallback führen müssen.

Ein Entwickler hat ebenso die Möglichkeit, Hystrix-Javanica *(siehe „https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica")* zu nutzen und somit anstatt mit einer eigenen Klasse mit einer einfachen „@HystrixCommand"-Annotation zu arbeiten. In der Spring-Galaxie gibt es, für Spring typisch, auch die Möglichkeit, mit einer einfachen „@HystrixCommand"-Annotation zu arbeiten. Spring baut dabei auf der Hystrix-Javanica-Implementierung auf. Zu erwähnen ist, dass man seine Fallback-Methode als einfachen String in der Annotation angibt und somit erst zur Compile-Zeit einen möglichen Tippfehler im Methodennamen erkennt.

Das Schöne an Hystrix ist, dass Netflix eine Unmenge an Konfigurationsmöglichkeiten vorgesehen hat und für jeden Parameter einen passenden Default bereitstellt. Somit ist es möglich, Hystrix aus dem Stand heraus zu nutzen und mit den Defaults erstmal loszulegen. In der Beispiel-Anwendung verwenden wir Hystrix immer dann, wenn wir uns an einen Remote-Service binden müssen und diesen nicht asynchron aufrufen können *(siehe Abbildung 2)*.

Archaius

Archaius stammt ebenfalls aus dem Hause Netflix und steht auch als

Open-Source-Projekt zur Verfügung. Hystrix baut intern auf Archaius auf und bietet damit die Möglichkeit, eine dynamische Konfiguration zur Laufzeit bereitzustellen. Dies ist ein klarer Vorteil bei den ersten Schritten mit Hystrix und versetzt einen in die Lage, Einfluss auf die definierten Timeouts zu nehmen oder den Circuit Breaker im laufenden Betrieb vom Status „geschlossen" auf „offen" zu setzen. Warum sollten wir das tun? In einer Integrationsumgebung können wir so unsere Fallback-Strategie überprüfen und auf ihre Standfestigkeit hin testen. Ebenso können wir dafür sorgen, dass unsere Fallbacks von Hystrix nicht mehr im Fehlerfall verwendet werden, und testen, ob unser Gesamtsystem damit umgehen kann. Keiner möchte in einer modernen, verteilten Servicelandschaft dreißig oder fünfzig Services neu einrichten, nur weil man mit zwei Sekunden anstatt mit einer Sekunde als Timeout arbeiten möchte.

In der gezeigten Demo befindet sich ein CoreOS-etcd-Server *(siehe „https://coreos.com/etcd")*, in dem die Hystrix-Konfiguration abgelegt ist, die so bequem über das REST-API des etcd-Servers geändert werden kann. Der etcd-Server kann in einem Cluster betrieben werden und bietet ein paar Gimmicks, mit denen man zum Beispiel ein Feature-Toggle implementieren kann. Hier wird eine Property mit dem neuen Wert „true" via REST-Call gesetzt und dies zeitlich auf fünf Minuten begrenzt. Danach nimmt die Property wieder ihren Ursprungswert „false" an. Dank Archaius können wir darauf via Callback reagieren und das neue Feature aktivieren beziehungsweise deaktivieren.

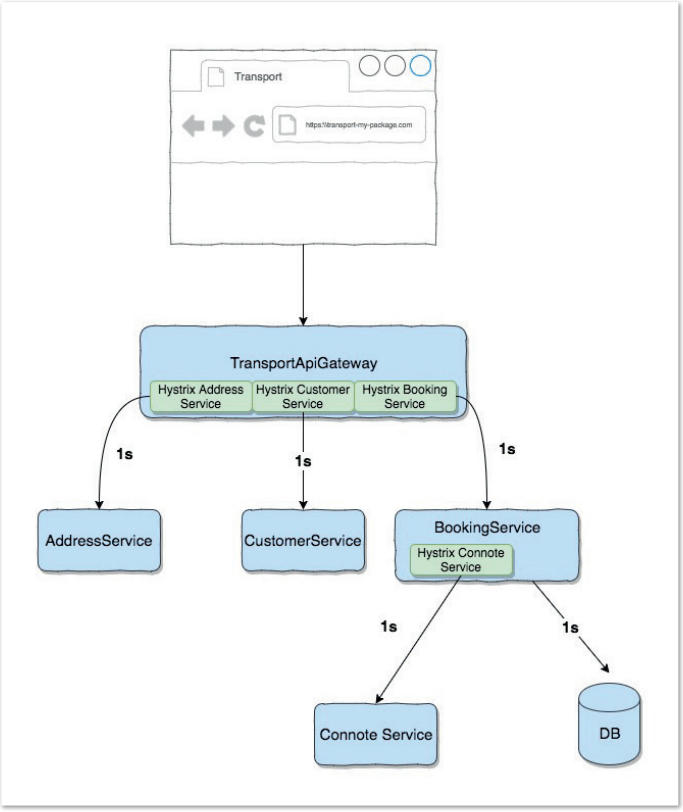


Abbildung 2: Architektur-Diagramm Hystrix

Eureka – Service Discovery

Wie finden sich nun unsere einzelnen Module untereinander und wie verwalten wir deren Zustand und auch die Anzahl der verfügbaren Instanzen? Das Mittel der Wahl ist eine Service Discovery, bei der sich jeder Service registriert. Ein anderer Service kann sich alle aktuellen Instanzen bei der Service Discovery abholen und für den Aufruf verwenden. Damit ist es allerdings noch nicht getan – in unserem Beispiel verwenden wir Eureka *(siehe „https://github.com/netflix/eureka")* aus dem Hause Netflix.

Eureka führt für uns auch einen Health Check der registrierten Services durch, nimmt automatisch nicht mehr antwortende oder fehlerhafte Services aus der Liste und benachrichtigt alle registrierten Consumer darüber. Eureka ist so aufgebaut, dass die Consumer eine lokale Kopie der registrierten Services vorhalten und diese periodisch aktualisieren. Somit sind die Services nach dem Start auch ohne eine laufende Instanz eines Eureka Servers in der Lage weiterzuarbeiten.

Ribbon

Wie könnte es auch anders sein, auch Ribbon *(siehe „https://github.com/Netflix/ribbon")* kommt aus dem Hause Netflix und ermöglicht im Zusammenspiel mit Eureka unter anderem ein Client-Side-Load-Balancing. Die verfügbaren Instanzen sind über Eureka bekannt und werden im einfachen Round-Robin-Verfahren angesprochen. Sollte Eureka feststellen, dass eine Instanz nicht mehr korrekt antwortet, wird diese vom Eureka-Client nicht weiter verwendet.

Dank Spring ist es möglich, mit einem „LoadBalanced Rest"-Template *(siehe „https://spring.io/guides/gs/client-side-load-balancing")* zu arbeiten, um das Client-Side-Load-Balancing durchzuführen. Als Endpoint wird dabei der Name des in Eureka registrierten Service

```
...
@LoadBalanced
  @Bean
  RestTemplate restTemplate(){
    return new RestTemplate();
  }

  @Autowired
  RestTemplate restTemplate;

@RequestMapping("/hi")
  public String hi(@RequestParam(value="name", defaultValue="You") String name) {
    String greeting = this.restTemplate.getForObject("http://greeting-service/greeting", String.class);
    return String.format("%s, %s!", greeting, name);
  }
...
```

Listing 1

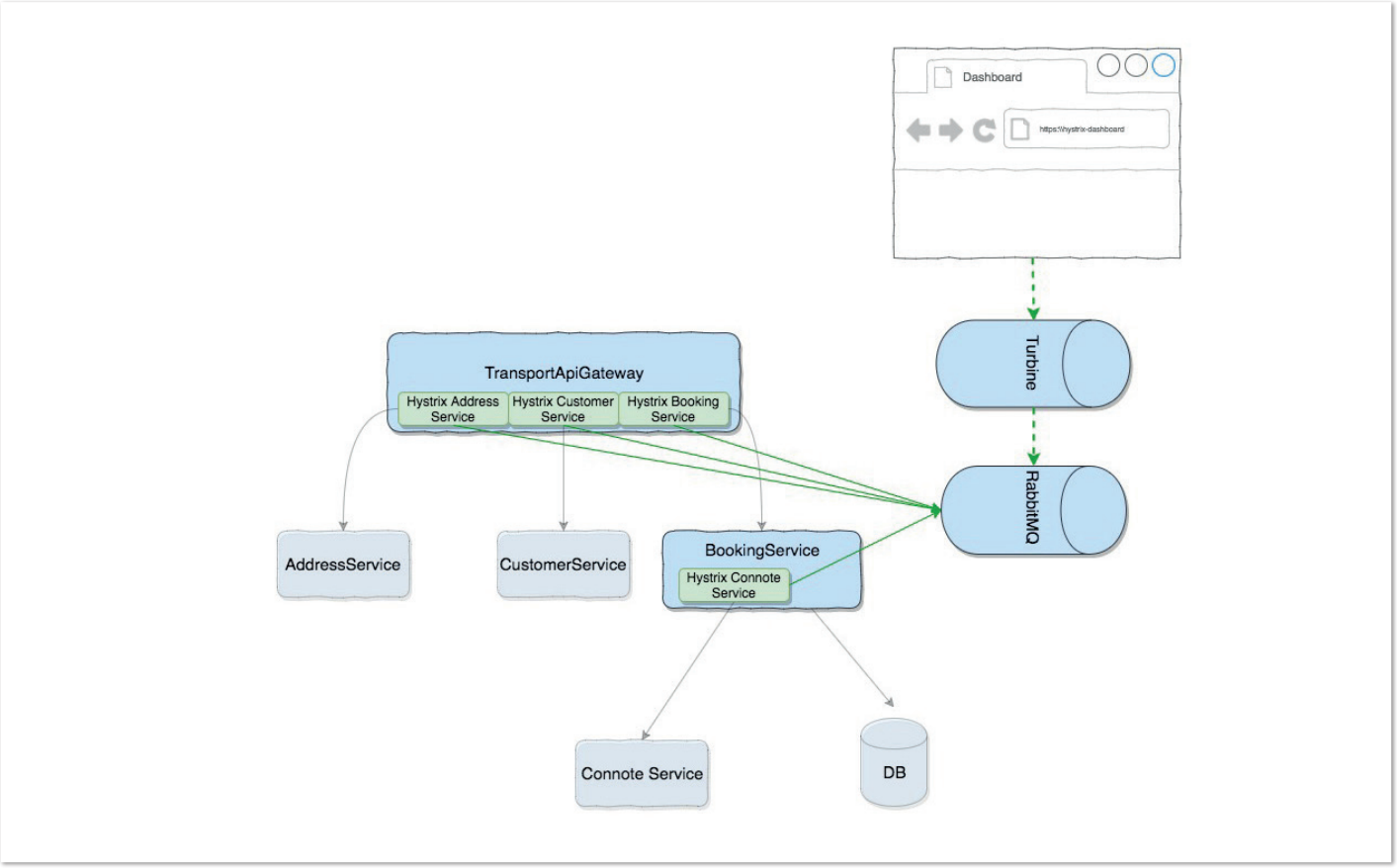


Abbildung 3: Architektur-Diagramm-Metriken via RabbitMQ

verwendet, dieser wird vom RestTemplate mit der entsprechenden URL und dem zugeteilten Port ersetzt. Somit ist der Entwickler völlig frei von der Umgebung, in der der Service ausgeführt wird, und muss sich nicht mit passenden Ports beziehungsweise URLs herumschlagen *(siehe Listing 1)*. Wie in dem Beispiel der Spring-Kollegen *(siehe „https://spring.io/guides/gs/client-side-load-balancing")* zu erkennen ist, verwendet das RestTemplate den Service-Namen „greeting-service", der zuvor in Eureka registriert worden ist.

AOP Chaos Monkey

So langsam wächst unsere kleine Beispielanwendung und es wird Zeit für ein wenig mehr Leben. Es ist doch sehr ernüchternd und langweilig für einen Entwickler, eine neue Anwendung zu gestalten und nie zu erleben, wie die Anwendung mit zufälligem Chaos umgeht. Ein ungutes Bauchgefühl bleibt, sollte man die entwickelte Anwendung nie unter Stress und im Zusammenspiel gesehen haben. Den erfahrenen Entwickler erkennt man dann am spontanen Fernbleiben zur Release-Einführung. Viel besser wäre es doch, die Anwendung durch einen kleinen Helfer und ein paar seiner bösen Freunde in einen instabilen Zustand zu versetzen, um zu sehen, ob die unabhängigen Module und die daraus entstehenden Bulkheads, der Einsatz von Hystrix, Eureka und Ribbon auch wirklich funktionieren.

Bei Netflix hat dies zur Simian Army geführt, mit der sich jeder Entwickler messen lassen und deren Angriffen jedes entwickelte System standhalten muss. Einer der bekanntesten Protagonisten ist ein Kollege mit dem passenden Namen „Chaos Monkey". Bei Netflix hat er die Aufgabe, AWS-Instanzen im laufenden Betrieb abzuschießen und so sicherzustellen, dass die Entwickler den aufrufenden Service entsprechend robust, also resilient gebaut haben. Diesen Ansatz findet der Autor sehr interessant und hat sich daher einen eigenen Chaos Monkey für Spring geschrieben. Mit einer einfachen Annotation „@EnableChaosMonkey" kann er in seiner Spring-Boot-Applikation einen Chaos Monkey bereitstellen, den er mithilfe von Archaius dynamisch zur Laufzeit aktivieren kann. Sollte der Chaos Monkey aktiviert sein, wird er per Zufall darüber entscheiden, wie er uns das Leben zur Hölle machen möchte. Folgende Optionen bieten sich hier:

- RuntimeException
- Timeout
- nichts

Im besten Fall haben wir also nichts zu befürchten und wir können auf eine funktionierende Anwendung hoffen. Dies wird mit den aktuellen Einstellungen, mit denen der Chaos Monkey läuft, jedoch nicht allzu oft passieren. Er wird sich an jede Komponente der Beispielanwendung hängen, die einen Service bereitstellt, und dort jede „public"-Methode beeinflussen. Nur so ist es möglich, Hystrix im tatsächlichen Einsatz zu sehen und zu der Erkenntnis zu gelangen, ob sich dessen Einsatz wirklich lohnt.

Hystrix-Metriken

Ein weiteres sehr nützliches Feature von Hystrix sind die Metriken, die wir frei Haus geliefert bekommen. Hystrix muss für die korrekte Zustandsberechnung des Circuit Breaker die Aufrufe überwachen und für einen definierten Zeitraum vorhalten. Diesen Zeitraum können wir, wie von Hystrix gewohnt, an unsere Bedürfnisse anpassen. Dank unseres Chaos Monkey haben wir richtig Leben in der Bude und sehen Hystrix in Action. Dies wird sich auch in den Metriken von

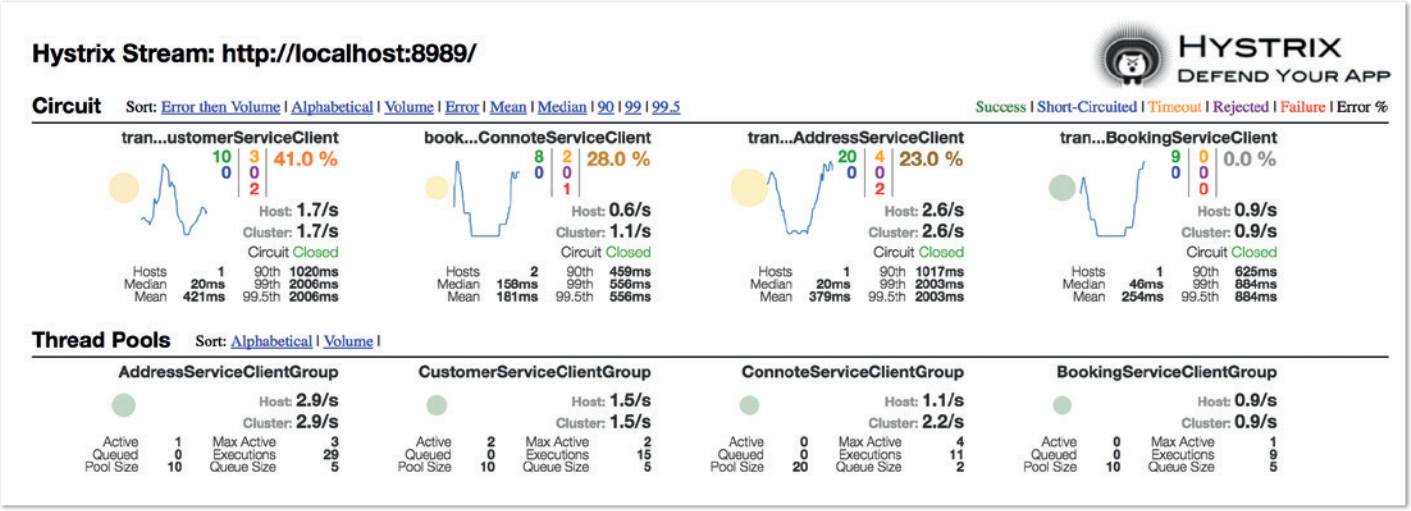


Abbildung 4: Das Hystrix-Dashboard

Hystrix widerspiegeln. Wie können wir uns diese Metriken nun zunutze machen und vor allem: Wie kommen wir an diese sehr wichtigen Daten?

Jede Komponente, in der wir Hystrix einsetzen, kann uns einen HTTP-Stream mit allen Metriken liefern. Dies wäre einer der Wege, die wir gehen können. Da wir aber eine verteilte Anwendung haben und nicht immer wissen können, unter welcher IP und mit welchem Port die Anwendung läuft, wäre es doch viel schöner, wenn die Anwendung uns die Daten über einen definierten Kanal liefert.

In der Beispielanwendung hat sich der Autor für einen Messaging-Weg entschieden und RabbitMQ eingesetzt. Dies hat den Vorteil, dass alle Komponenten die Daten in eine Queue ablegen, an der sich die Hystrix-Turbine *(siehe „https://github.com/Netflix/Turbine/wiki")* als Consumer registriert und die Daten aggregiert zurückliefert *(siehe Abbildung 3)*.

Die Turbine hat dabei die Aufgabe, alle eintreffenden Metriken zu einem zentralen Stream zu aggregieren, der dann vom Hystrix-Dashboard visualisiert oder von anderen Consumern gespeichert, aufbereitet und ausgewertet werden kann. Somit können wir in den Metriken auch erkennen, wie viele Instanzen unseres Service laufen und wie viele Requests wir über alle Instanzen eines Service hinweg haben.

Hystrix-Dashboard

Eines der ersten und vor allem schnell in Betrieb zu nehmenden Monitoring-Tools ist das Hystrix-Dashboard. Damit ist es sehr einfach und schnell möglich, einen umfangreichen Überblick über den Zustand unserer Anwendung zu erhalten. Ein Gesamtbild der verteilten Anwendung erhalten wir aber nur, wenn wir uns als Datenquelle den Turbine-Stream mit allen aggregierten Streams unserer verteilten Anwendung nehmen *(siehe Abbildung 4)*.

Fazit

Der Artikel gibt einen guten und vor allem praxisnahen Überblick zum Thema „Hystrix" und zu den sehr nützlichen Projekten aus dem Hause Netflix. Hystrix ist kein Allheilmittel und kann auch zu einem unerwünschten Verhalten führen, wenn es nicht korrekt konfiguriert wurde. So ist es zum Beispiel möglich, dass sehr viele Requests aufgrund einer falschen Hystrix-Konfiguration in einem Fallback landen. Um dies zu verhindern, müssen die Metriken herangezogen werden, und es erfordert auch viel Zeit und Vorbereitungen, die richtigen Werte der Konfiguration zu ermitteln.

Einige der in der Beispielanwendung verwendeten Tools sind in diesem Artikel leider nicht erläutert, so kommen zum Beispiel auch der Spring-Boot-Admin zur Administration von Spring-Boot-Anwendungen oder das Open-Tracing-Tool ZipKin zum Einsatz, die nicht unerwähnt bleiben sollen. Diese beiden Tools helfen ungemein dabei, die beteiligten Akteure einer verteilten Spring-Boot-Anwendung übersichtlich im Auge zu behalten. ZipKin ermöglicht es, die Datenströme und die bei einem Request beteiligten Instanzen eines Service visualisiert zu bekommen. Dank ZipKin ist es möglich zu sehen, welchen Weg ein Request genommen hat und welcher Service geantwortet hat. Das Thema „Open Tracing" in einer verteilten Spring-Boot-Anwendung ist auf jeden Fall einen Blick Wert und erleichtert die Fehlersuche ungemein.

Benjamin Wilms
benjamin.wilms@codecentric.de

Benjamin Wilms arbeitet als Senior IT Consultant und Developer bei der codecentric AG. Seine aktuellen Schwerpunktthemen sind Skalierbarkeit und Resilience in verteilten Anwendungen. Er teilt und diskutiert seine Ideen regelmäßig auf Konferenzen sowie als Autor von Artikeln und Blog-Posts.

Web Components mit Polymer

Marcus Fihlon, CSS Versicherung

Web Components ist ein Standard, der komponentenbasierte Software-Entwicklung für das Web ermöglicht. Der Artikel zeigt, wie man mittels Web Components und Polymer wiederverwendbare Komponenten entwickelt, die in sich gekapselt sind und sich kombinieren sowie erweitern lassen.

Die Zeiten, in denen man als Java-Entwickler ausschließlich Java-Technologien für Benutzeroberflächen eingesetzt hat, sind schon lange vorbei. Über die Jahre hinweg hat HTML immer mehr Einzug gehalten und kann nun als fester Bestandteil des Java-Ökosystems betrachtet werden. Doch HTML ist nicht gleich HTML – seit der ersten Version, die vor nunmehr fünfundzwanzig Jahren erschienen ist, hat sich sehr viel getan. Eher still und leise hat sich ein Standard für „Web Components" fast schon eingeschlichen, der aus vier verschiedenen Spezifikationen besteht.

Der Standard spezifiziert, wie wiederverwendbare Widgets und Komponenten für Web-Dokumente und Web-Applikationen erstellt und genutzt werden können. Die Idee dahinter ist, die komponentenbasierte Software-Entwicklung in das World Wide Web zu bringen. Das Komponentenmodell erlaubt die Kapselung von HTML-Elementen und deren Interoperabilität. Von vielen Web-Entwicklern bisher kaum wahrgenommen, bieten Web Components enormes Potenzial, um auf einfache und schnelle Weise moderne und modulare Benutzeroberflächen mit HTML zu erstellen.

Bücherwurm

In diesem Artikel entwickeln wir gemeinsam eine kleine Web-Applikation auf Basis von Web Components und des Polymer-Frameworks. Wir werden dabei sowohl bereits vorhandene Komponenten einsetzen als auch eigene Komponenten entwickeln. Der Business-Case ist sehr simpel: Mithilfe eines Eingabefelds sollen eine Buchsuche durchgeführt und die Resultate übersichtlich dargestellt werden. Doch was sich so simpel anhört, hat es durchaus in sich: Wir teilen die Funktionalitäten auf einzelne Komponenten auf, die miteinander interagieren werden.

Zuerst müssen wir unser Projekt einrichten. Wir beginnen mit einem leeren Verzeichnis mit dem Namen „bookworm". Um uns das Management externer Abhängigkeiten in unserem Projekt einfacher zu machen, nutzen wir Bower *(siehe „https://bower.io")*. Nach der Installation von Bower initialisieren wir unser Projekt im Projektverzeichnis. Alle Fragen können mit dem Standardwert quittiert werden. Nach einer abschließenden Sicherheitsfrage wird automatisch die neue Datei „bower.json" angelegt *(siehe Abbildung 1)*.

```
bower install --save Polymer/polymer PolymerElements/iron-ajax PolymerElements/paper-card PolymerElements/paper-input PolymerElements/paper-material PolymerElements/paper-styles
```

Listing 1

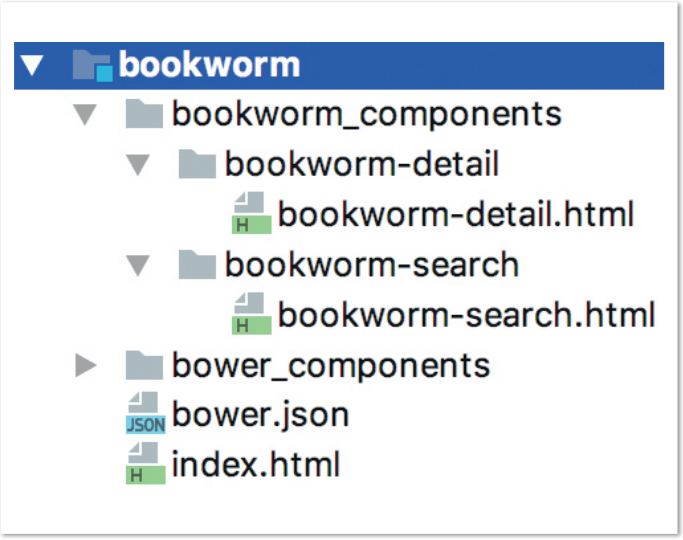


Abbildung 1: Die Verzeichnisstruktur

Fremdkomponenten einbinden

Im weiteren Verlauf dieses Tutorials nutzen wir einige Komponenten aus dem Polymer Project *(siehe „https://www.polymer-project.org“)*, die wir jetzt einbinden *(siehe Listing 1)*. Eine Erläuterung der einzelnen Komponenten folgt später.

Es entsteht automatisch ein neues Verzeichnis mit dem Namen „bower_components“ und zusätzlich zu unseren gewünschten Komponenten werden noch etliche Abhängigkeiten heruntergeladen. Alles landet in diesem neuen Verzeichnis, es ist also wunderbar von unserem eigenen Quelltext separiert. Bevor wir mit der eigentlichen Entwicklung starten können, fehlen noch ein Browser und ein Editor. Die Wahl des Browsers ist an dieser Stelle sehr wichtig, denn einige Browser benötigen zum Laden von Web Components eine HTTP-Verbindung.

Während Safari auch mit Web Components über „file://“-URLs keinerlei Probleme hat, funktionieren Web Components im Chrome ausschließlich über eine HTTP(S)-Verbindung. Daher empfiehlt sich der Einsatz eines lokalen Webservers oder eines Editors, der einen integrierten Webserver mitbringt, beispielsweise den kostenlosen Editor Atom *(siehe „https://atom.io“)* mit dem Live-Server-Package *(siehe „https://atom.io/packages/atom-live-server“)*.

Die erste eigene Komponente

Unsere kleine App realisieren wir als eigenständige Komponente, damit sie einfach und schnell auf jeder Webseite eingebunden werden kann. Für unsere eigenen Komponenten erstellen wir im Projektverzeichnis das Unterverzeichnis „bookworm_components“, dort sammeln wir alle unsere eigenen Komponenten für diese Anwendung.

Es hat sich als Best Practice eingebürgert, für jede Komponente ein eigenes Unterverzeichnis zu erstellen und dort alle Dateien dieser Komponente abzulegen. Das Unterverzeichnis sollte den Namen der Komponente tragen. Für jede Komponente wird uns ein Tag zur Verfügung gestellt, mit dem wir die Komponente in den HTML-Quelltext einbauen können. Dafür wird der Name der Komponente genutzt, der zwingend einen Bindestrich enthalten muss. Da normale HTML-Tags keinen Bindestrich enthalten, kann der Browser so erkennen, dass es sich um eine Komponente handelt.

```
<template>
  It works!
</template>
```

Listing 2

```
<link rel="import" href="../../bower_components/polymer/polymer.html" />
```

Listing 3

```
<script>
  Polymer({
    is: "bookworm-search"
  });
</script>
```

Listing 4

Unsere erste Komponente heißt „bookworm-search“, also erstellen wir im soeben erzeugten Verzeichnis noch das Unterverzeichnis „bookworm-search“ und legen darin eine leere Datei mit dem Namen „bookworm-search.html“ an. Das ist noch so eine Best Practice: Die Hauptdatei der Komponente heißt wie die Komponente selbst. Beim späteren Import von Komponenten erleichtert uns diese Konvention in der Benennung von Verzeichnissen und Dateien, die richtige Datei für den Import zu finden.

In der soeben erstellten HTML-Datei beginnen wir nicht mit der üblichen HTML5-Struktur, denn wir erstellen keine Webseite, sondern eine Komponente. Mit dem Tag „<dom-module>“ definieren wir unsere Komponente und vergeben eine „id“, die unsere Komponente benennt und auch als Tag für sie fungiert, also „<dom-module id="bookworm-search">“.

Nun erstellen wir den Inhalt unserer Komponente. Der Einfachheit halber benutzen wir erstmal einen statischen Text. Dazu fügen wir innerhalb des „<dom-module>“-Tag ein „<template>“-Tag ein *(siehe Listing 2)*.

Bevor wir unsere Komponente nutzen können, ist sie noch bei Polymer zu registrieren. Dazu sind zwei Schritte notwendig. Zuerst müssen wir Polymer importieren, damit es uns zur Verfügung steht.

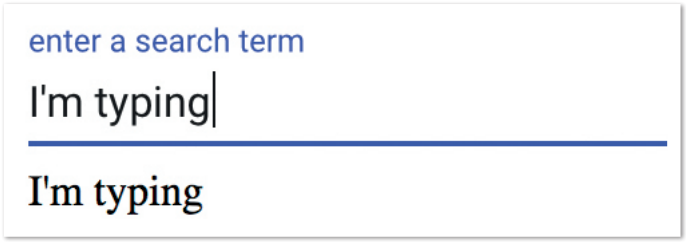


Abbildung 2: Das Eingabefeld in Aktion

Anschließend können wir unsere Komponente mit einem kleinen JavaScript registrieren lassen. Bei einer Komponente gehört der Import, ähnlich einer Java-Klasse, an den Anfang. Fügen wir nun also folgenden Import ganz am Anfang unserer Datei vor dem „<dom-module>"-Tag hinzu *(siehe Listing 3)*.

Dabei ist der Pfad zu beachten: Bei Imports mit relativen Pfad-Angaben ist immer die Position der aktuellen Komponente ausschlaggebend, nicht die der Webseite, in der unsere Komponente verwendet wird. Daher müssen wir erst zwei Verzeichnisebenen zurück. Nun ergänzen wir die Registrierung unserer Komponente bei Polymer. Wir übergeben dabei den Tag für unsere Komponente und Polymer kümmert sich darum, den Tag und unsere Komponente dem Browser bekannt zu machen. Dieser Code-Schnipsel gehört zwischen das schließende „</template>"- und das schließende „</dom-module>"-Tag *(siehe Listing 4)*. Mit nur elf Zeilen Code haben wir unsere allererste eigene Komponente auf Basis von Web Components und Polymer entwickelt.

Die Komponente einbauen

Jetzt wäre es toll, die erste eigene Komponente in Aktion sehen zu können. Nichts ist leichter als das – im Projektverzeichnis erstellen wir ein einfaches HTML5-Dokument mit dem Dateinamen „index.html" *(siehe Listing 5)*.

Soweit handelt es sich um ein klassisches HTML5-Grundgerüst. Wenn wir nun Web Components nutzen möchten, ergibt sich ein kleines Problem: Dieser Standard wird noch nicht von jedem Browser vollumfänglich unterstützt (Chrome und Opera unterstützen ihn in aktuellen Versionen komplett, alle anderen nur teilweise). Doch dafür gibt es eine einfache Lösung: Ein Polyfill ist ein in JavaScript geschriebener Code-Baustein, der in Browsern noch nicht unterstützte Funktionen mithilfe eines Workarounds nachrüstet. Einen solchen Polyfill für Web Components binden wir nun mit „<script src="bower_components/webcomponentsjs/webcomponents-lite.js"></script>" im „<head>"-Bereich ein.

Nun können wir damit starten, unsere Komponente einzubauen. Zuerst müssen wir unsere Komponente importieren, damit der Browser sie kennt. Den Import fügen wir im „<head>"-Bereich mit „<link rel="import" href="bookworm_components/bookworm-search/bookworm-search.html" />" hinzu. Einer Nutzung unserer Komponente steht nun nichts mehr im Weg, wir binden sie einfach durch „<bookworm-search></bookworm-search>" mit ihrem eigenen Tag im „<body>"-Bereich ein. Fertig! Wir rufen nun unsere Webseite auf und schauen uns das Ergebnis an: „It works!".

Beim Parsen unserer Webseite stößt der Browser auf den Import unserer Komponente. Ähnlich einem Stylesheet wird diese nachgeladen und bringt dem Browser unser Tag „<bookworm-search>" bei. Später stößt der Browser dann auf unser Tag und fügt an dieser Stelle unsere Komponente ein, die nichts anderes macht, als „It works!" auszugeben. So einfach funktionieren Web Components mit Polymer.

Ein Eingabefeld in hübsch

Auch wenn wir auf unseren statischen Text stolz sein können, schaut unser heutiges Ziel etwas anders aus. Wir benötigen ein Eingabefeld. Da wir den Umgang mit Komponenten lernen möchten, nutzen wir nicht das HTML-Input-Element, sondern eine Komponente. In diesem Fall eine von Polymer, die dem Material Design entspricht und etwas hübscher aussieht. Diese Komponente heißt „paper-input" und wir haben sie bereits am Anfang des Tutorials heruntergeladen.

Erweitern wir also nun unsere eigene Komponente „bookworm-search" um die Fremdkomponente „paper-input", indem wir diese mit „<link rel="import" href="../../bower_components/paper-input/paper-input.html" />" importieren. Nun kennt der Browser diese Fremdkomponente und wir können durch „<paper-input label="enter a search term"></paper-input>" unseren statischen Text durch das neue Eingabefeld ersetzen.

Diese beiden Schritte reichen aus. Nach dem Import und dem Einbau des Tags funktioniert diese Komponente schon. Wenn wir nun mit der Maus in das Eingabefeld klicken, sehen wir eine Animation, und wenn wir mit der Eingabe beginnen, eine weitere. Diese sind Bestandteil der neuen Komponente, die wir bei uns eingebunden haben.

Data Binding

Damit wir später eine Suche mit dem eingegebenen Begriff durchführen können, müssen wir an den Wert des Eingabefelds gelangen. Das ist mit Data Binding von Polymer ziemlich einfach. Wir müssen das Eingabefeld nur auffordern, den Wert in eine Variable zu schreiben. Dazu geben wir den Variablennamen in doppelten geschweiften Klammern mittels „<paper-input label="enter a search term" value="{{searchterm}}"></paper-input>" im „value"-Attribut an.

Um zu überprüfen, ob das Data Binding funktioniert, können wir den Inhalt der Variablen ausgeben lassen. Dazu geben wir die Variable innerhalb des Templates wieder mit doppelten geschweiften Klammern an – „{{searchterm}}". Wenn wir nun in das Eingabefeld tippen, erscheint der Suchtext auch darunter als Ausgabe *(siehe Abbildung 2)*. Nachdem das Data Binding überprüft ist, können wir die testweise Ausgabe des Suchbegriffs entfernen.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Bookworm</title>
  </head>
  <body>
  </body>
</html>
```

Listing 5

Name	Status	Type	Initiator	Size	Time
volumes?q=T	200	xhr	iron-request.html:304	4.7KB	1.03s
volumes?q=Te	200	xhr	iron-request.html:304	4.0KB	805ms
volumes?q=Tes	200	xhr	iron-request.html:304	4.0KB	561ms
volumes?q=Test	200	xhr	iron-request.html:304	6.0KB	529ms

Abbildung 3: Jeder Tastendruck löst einen Request aus

AJAX mal einfach

Nun führen wir eine Buchsuche per AJAX durch. Glücklicherweise müssen wir uns nicht mehr selbst mit dem „XMLHttpRequest" herumschlagen, das nimmt uns die Komponente „iron-ajax" ab, die wir bereits am Anfang des Tutorials heruntergeladen haben. Fügen wir nun den entsprechenden Import „<link rel="import" href="../../bower_components/iron-ajax/iron-ajax.html" />" hinzu, bauen die AJAX-Komponente ein und verwenden in der URL an der richtigen Stelle einfach die Variable „<iron-ajax url="https://www.googleapis.com/books/v1/volumes?q={{searchterm}}"></iron-ajax>" für das Data Binding mit dem Eingabefeld.

Wenn wir jetzt etwas in das Eingabefeld tippen, schreibt es die „paper-input"-Komponente in die Variable „searchterm". Da sich die Variable ändert, aktualisiert Polymer automatisch die URL der „iron-ajax"-Komponente. Wir können nun die AJAX-Komponente auffordern, bei jeder Änderung der URL automatisch einen AJAX-Request abzusetzen. Dazu ergänzen wir einfach mit „<iron-ajax auto url="https://...q={{searchterm}}"></iron-ajax>" den Parameter „auto". Wenn wir unsere Webseite im Browser neu laden und jetzt etwas in das Eingabefeld eintippen, können wir in der Netzwerk-Ansicht unseres Browsers sehen, dass nach jedem Tastendruck ein neuer Request abgeschickt wird *(siehe Abbildung 3)*.

Wir können die Netzwerk- und Serverlast reduzieren, indem wir den Request nicht sofort nach jedem Tastendruck, sondern mit einer Verzögerung auslösen lassen, wenn über einen Zeitraum von 500 Millisekunden keine weitere Eingabe erfolgte. Diese Zeitspanne können wir durch „<iron-ajax auto debounce-duration="500" url="https://...q={{searchterm}}"></iron-ajax>" über den Parameter „debounce-duration" steuern.

Damit wir die Antwort auf unseren Request später weiterverarbeiten können, speichern wir sie. Dazu geben wir an, in welchem Format wir die Antwort erwarten und dass wir den Inhalt der letzten (aktuellsten) Antwort in einer Variablen speichern möchten. Um dies zu erreichen, ergänzen wir im „<iron-ajax>"-Tag zwei Attribute *(siehe Listing 6)*.

Über „handle-as" teilen wir mit, dass wir eine Antwort im JSON-Format erwarten und mittels „last-response" und Data Binding lassen wir uns die letzte Antwort in die Variable „searchresult" schreiben. So einfach können wir einen AJAX-Request durchführen.

Verschachtelte Templates

Die Suchergebnisse warten jetzt darauf, angezeigt zu werden.

```
handle-as="json"
last-response="{{searchresult}}"
```

Listing 6

Hierzu kombinieren wir das Data Binding von Polymer mit einem weiteren Template, das die Daten eines Buches anzeigt und so oft wiederholt eingebunden wird, wie es Bücher im Suchergebnis gibt. Innerhalb des Templates unserer Komponente fügen wir dafür ein neues Template hinzu *(siehe Listing 7)*.

Das Attribut „is" mit dem Wert „dom-repeat" sorgt dafür, dass das Template selbst so oft eingebunden wird, wie es Einträge in dem Array gibt, das dem Attribut „items" zugewiesen wurde. Der jeweilige Eintrag aus dem Array steht automatisch in der Variablen „item" zur Verfügung. Zum Test geben wir innerhalb des Templates den Titel des Buches aus. Wenn wir nun einen Test im Browser durchführen, sollte es wie im Screenshot in *Abbildung 4* aussehen.

Interoperabilität

Innerhalb unserer eigenen Komponente könnten wir nun in der inneren Schleife die Ausgabe der Suchergebnisse vervollständigen und wären fertig. Doch das wäre zu einfach – und um die Interoperabilität zwischen eigenen Komponenten zu demonstrieren, brauchen wir mindestens zwei eigene Komponenten. Daher lagern wir die Anzeige eines Suchergebnisses in eine eigene Komponente aus. Daraus ergibt sich die interessante Frage, wie nun die Daten eines Buches von der einen Komponente in die andere gelangen, um dort angezeigt zu werden. Doch dazu später mehr, wir beginnen ein paar Schritte vorher und legen erst eine neue Komponente an.

Die neue Komponente soll „bookworm-detail" heißen. Entsprechend legen wir im Verzeichnis „bookworm_components" das Unterverzeichnis „bookworm-detail" an und darin eine neue Datei mit dem Namen „bookworm-detail.html". Das Grundgerüst unserer neuen Komponente entspricht dem Grundgerüst unserer ersten Komponente *(siehe Listing 8)*. Alle Komponenten haben in etwa das gleiche

```
<template is="dom-repeat" items="{{searchresult.
items}}">
  <p>{{item.volumeInfo.title}}</p>
</template>
```

Listing 7

Abbildung 4: Unsere Suchergebnisse werden angezeigt

Grundgerüst. Unserer zweiten Komponente, die die Details eines Buches anzeigen soll, müssen wir die Daten eines Buches übergeben können. Dazu erweitern wir den JavaScript-Code am Ende der Komponente um ein Property-Objekt *(siehe Listing 9)*.

Innerhalb des Property-Objekts definieren wir ein „bookdata"-Objekt vom JavaScript-Typ „Object". Die Property-Objekte werden über die Attribute des Tags unserer Komponente gesetzt, so können wir die Daten eines Buches von einer Komponente an die nächste übergeben. Bevor wir unsere neue Komponente einbauen, ergänzen wir die Ausgabe des Buchtitels im Template *(siehe Listing 10)*.

Wir können auf unser definiertes Property zugreifen wie auf eine Variable. Bauen wir nun unsere zweite Komponente in die erste Komponente ein. Dazu müssen wir in der Datei „bookworm-search.html" mit „<link rel="import" href="../bookworm-detail/bookworm-detail.html" />" einen Import hinzufügen. Dabei ist zu beachten, dass wir, um zur Komponente „bookworm-detail" zu gelangen, nur eine Verzeichnisebene nach oben wechseln müssen. Ersetzen wir nun die direkte Ausgabe des Buchtitels durch den Einbau unserer neuen Komponente und übergeben dabei gleich mit „<bookworm-detail bookdata="{{item}}"></bookworm-detail>" die Daten des Buches. Wenn wir uns das Ganze nun erneut im Browser anschauen, sollte es immer noch wie im vorherigen Screenshot aussehen, jedoch haben wir technisch gesehen eine schönere Kapselung erreicht.

Material Design

Wenn wir genau hinschauen, entspricht unsere Applikation noch nicht den Regeln für Material Design. Wir können nun mit sehr wenig Aufwand dafür sorgen, dass unsere App einen schönen Rahmen erhält und überall die gleiche Schriftart verwendet wird. Dazu ergänzen wir in unserer Such-Komponente zwei Imports *(siehe Listing 11)*. Dann umschließen wir den kompletten Inhalt des Templates mit der soeben importierten „paper-material"-Komponente *(siehe*

```
<link rel="import" href="../../bower_components/
polymer/polymer.html" />
<dom-module id="bookworm-detail">
  <template>
  </template>
  <script>
    Polymer({
      is: "bookworm-detail"
    });
  </script>
</dom-module>
```

Listing 8

```
<script>
  Polymer({
    is: "bookworm-detail",
    properties: {
      bookdata: {
        type: Object
      }
    }
  });
</script>
```

Listing 9

```
<template>
  <p>{{bookdata.volumeInfo.title}}</p>
</template>
```

Listing 10

```
<link rel="import" href="../../bower_components/paper-
material/paper-material.html" />
<link rel="import" href="../../bower_components/paper-
styles/typography.html" />
```

Listing 11

```
<template>
  <paper-material>
    …
  </paper-material>
</template>
```

Listing 12

```
<template>
  <style>
    * {
      @apply(--paper-font-common-base);
    }
    paper-material {
      padding: 1em;
    }
  </style>
  <paper-material>
    …
  </paper-material>
</template>
```

Listing 13

```
<template>
  <paper-card heading="{{bookdata.volumeInfo.title}}" image="{{bookdata.volumeInfo.imageLinks.thumbnail}}">
    <div id="subtitle">{{bookdata.volumeInfo.subtitle}}</div>
    <div id="details">{{bookdata.volumeInfo.pageCount}} Pages ({{bookdata.volumeInfo.publishedDate}})</div>
  </paper-card>
</template>
```

Listing 14

```
<style>
  paper-card {
    margin: 1em;
    padding: 0.5em;
    width: 20em;
  }
  #subtitle {
    font-size: 1.2em;
    font-weight: bold;
  }
  #details {
    font-style: italic;
  }
</style>
```

Listing 15

```
paper-card {
  margin: 1em;
  padding: 0.5em;
  width: 20em;
  --paper-card-header-image-text: {
    color: white;
    font-weight: bold;
    text-shadow:  1px  1px 1px black,
                  1px -1px 1px black,
                 -1px  1px 1px black,
                 -1px -1px 1px black;
  }
}
```

Listing 16

Listing 12). Jetzt noch schnell etwas eigenen Style hinzugefügt, und unsere Buchsuche sieht schon viel gefälliger aus. Eigene Styles müssen Bestandteil des Templates sein, also innerhalb des Templates definiert werden *(siehe Listing 13)*.

Wir übernehmen für alle HTML-Elemente die von Polymer für uns vordefinierte Schrift und setzen für die „paper-material"-Komponente einen Abstand zum enthaltenen Inhalt. Nun verwendet unsere App überall die gleiche Schriftart und hat einen schönen Rahmen.

Responsive Design

Bisher zeigen wir das Suchergebnis untereinander an. Es funktioniert, ist jedoch nicht wirklich responsiv, da der zur Verfügung stehende Platz nicht optimal genutzt wird. Daher werden wir nun die Komponente für die Detail-Ansicht erweitern. Dazu importieren wir mit „<link rel="import" href="../../bower_components/paper-card/paper-card.html" />" die Fremdkomponente „paper-card" und ersetzen im Template die Ausgabe des Titels als Absatz durch die soeben importierte „paper-card", wobei wir gleich ein paar weitere Angaben ausgeben *(siehe Listing 14)*.

Bei „paper-card" definieren wir zwei Attribute, eines zur Ausgabe des Buchtitels und eines zur Anzeige des Buchcovers. Innerhalb von „paper-card" geben wir in einzelnen „div"-Elementen neu auch den Sub-Titel, die Seitenzahl und das Erscheinungsdatum aus. Wenn wir das nun im Browser betrachten, sehen wir die entsprechenden Angaben und auch das Buchcover wird angezeigt. Leider alles andere als schön. Daher fügen wir am Anfang des Templates noch ein Stylesheet hinzu *(siehe Listing 15)*.

Wir definieren damit für „paper-card" einige Abstände und beschränken deren Breite. Auch der Sub-Titel und die Anzeige von Seitenzahl und Erscheinungsdatum sind nun schon etwas gefälliger. Wir haben noch etwas erreicht: Durch die Beschränkung der Breite von „paper-card" werden diese nun nebeneinander angezeigt, je nach zur Verfügung stehendem Platz. Unsere kleine Applikation reagiert jetzt bereits responsiv.

Fremd-Komponenten stylen

Der Buchtitel wird von „paper-card" über dem Buchcover angezeigt. Das wäre nicht weiter schlimm, wenn der Titel jetzt nicht schlecht zu lesen wäre. Je nach Buchcover ist der Titel gar nicht zu entziffern. Das müssen wir ändern. Doch wie?

Jede Komponente hat ihr eigenes kleines DOM, „Shadow DOM" genannt. Wir können HTML, CSS und JavaScript schreiben, ohne Angst haben zu müssen, dass dies unerwünschte Nebeneffekte auf andere Komponenten hat. Allerdings haben wir nun ein Problem: Wir können in unserer Komponente so viel CSS definieren, wie wir möchten, „paper-card" sieht es schlicht nicht, da sie ihr eigenes DOM hat.

CSS unterstützt Variablen und sogenannte „Mix-ins". Wenn der Autor einer Komponente vorgesehen hat, von extern Styles verändern zu dürfen, so stellt er dafür entsprechende Variablen bereit. Welche das sind und was man damit jeweils stylen kann, muss der Autor der Komponente dokumentieren. Google hat das glücklicherweise für alle Komponenten getan, auch für „paper-card" *(siehe „https://www.webcomponents.org/element/PolymerElements/paper-card/paper-card#styling").* Gemäß Dokumentation müssen wir zum Stylen des Textes über dem Bild die Variable „--paper-card-header-image-text" setzen. Diese Variable ist ein Mix-in, sie enthält also nicht nur einen Wert, sondern eine komplette CSS-Definition (Name-Wert-Paare). Ergänzen wir nun das CSS für die „paper-card"-Komponente *(siehe Listing 16)*.

Wir setzen damit die Schriftfarbe auf weiß, machen den Text fett und fügen noch einen Schatten hinzu, sodass die Schrift schwarz umrandet wird. So ist sie nun auch über dem Buchcover sehr gut zu lesen – und wir haben über ein CSS-Mix-in in das Styling einer anderen Komponente eingegriffen.

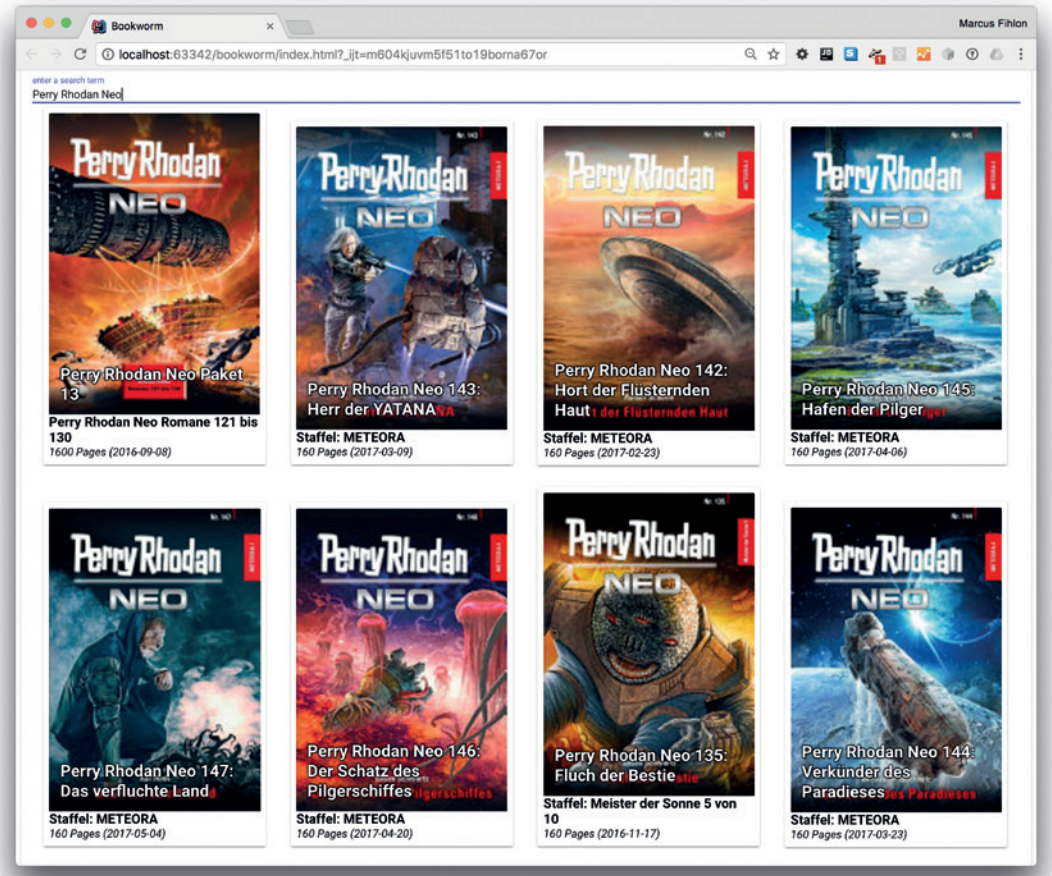


Abbildung 5: Die fertige Anwendung

Fazit

Für unsere erste Applikation auf Basis von Web Components und Polymer haben wir einiges geleistet. Wir mussten zwei eigene Komponenten entwickeln und haben auf Fremd-Komponenten zurückgegriffen. Durch ein Polyfill haben wir dafür gesorgt, dass alle aktuellen Browser mit unserer App funktionieren, auch wenn sie noch keine vollständige Web-Components-Unterstützung implementiert haben.

Wir haben mehrere Komponenten über Data Binding interagieren lassen und so auch gelernt, wie Daten von einer Komponente an eine andere weitergegeben werden können. Den AJAX-Request zur Suche nach Büchern konnten wir komplett ohne JavaScript realisieren. Schließlich haben wir noch auf das Styling einer Komponente Einfluss genommen und es unseren Wünschen angepasst. Das Ergebnis kann sich sehen lassen *(siehe Abbildung 5)*.

Mittels Web Components und Polymer haben wir wiederverwendbare Komponenten entwickelt, die in sich gekapselt sind und die sich kombinieren und erweitern lassen. Dabei haben wir immer „in Komponenten" gedacht und auf effiziente Weise übersichtlichen, leicht verständlichen Quelltext geschrieben. Web Components – die neue, standardisierte Art, Anwendungen für das Web zu entwickeln.

Hinweis: Der Quelltext der Beispielanwendung dieses Tutorials steht auf GitHub unter der AGPL zur Verfügung *(siehe „https://bit.ly/wcpoltut")*.

Marcus Fihlon
marcus@fihlon.ch

Marcus Fihlon arbeitet als Agile Coach und Software-Entwickler bei der CSS Versicherung in Luzern und als Dozent für Web Engineering an der TEKO Schweizerische Fachschule in Olten. Seit Anfang 2017 ist er in der Zentralschweiz für die Organisation und Durchführung von Veranstaltungen der Java User Group Switzerland zuständig. Er organisiert den monatlichen Hackergarten in Luzern und gibt sein Wissen gerne mit Vorträgen und Workshops bei User Groups und Konferenzen weiter.

„Wir sind jetzt unabhängig vom Zeitplan für Java EE 8 …“

MVC 1.0 sollte eine Ergänzung für Java EE neben dem existierenden Web-Framework Java Server Faces werden. Bei der Neuausrichtung von Java EE 8 hat Oracle den JSR über Bord geworfen. Starkes Feedback aus der Community hat den Konzern aber dazu bewegt, ihn zumindest außerhalb von Java EE 8 weiterzuführen – mit Ivar Grimstad als einem von Oracle (und den anderen großen Herstellern) unabhängigen Specification Lead: Sozusagen als „Community JSR“. Andreas Badelt, stellvertretender Leiter der DOAG Java Community, wollte wissen, wie es um den JSR steht.

Oracle hat den MVC 1.0 JSR an die Community übergeben. Du bist jetzt gemeinsam mit Ivar Grimstad Specification Lead. Was ist dabei deine Rolle?

Die Aufgaben des Spec Lead sind im Java Community Process (JCP) sehr genau definiert. Er ist verantwortlich dafür, dass der JSR im zeitlich vorgegebenen Rahmen die entsprechenden Ergebnisse liefert, beispielsweise die ersten Entwürfe veröffentlicht bis hin zum Release der finalen Spezifikation. Rechtlich gesehen trifft er alle Entscheidungen und könnte sich sogar über die Meinung der Expert Group hinwegsetzen. In vielen JSRs wird der Spec Lead als der inhaltliche Treiber gesehen und die Expert Group hat lediglich eine beratende Funktion. Unser Ziel für JSR 371 ist es, die Expert Group in einer möglichst demokratischen Art zu führen und die Spezifikation gemeinsam zu erarbeiten. Dazu gehört beispielsweise, dass wir viele Abstimmungen abhalten, die dann nach Mehrheit entschieden werden. Als Spec Leads versuchen wir, den notwendigen Rahmen zu liefern, damit die Expert Group gemeinschaftlich zum bestmöglichen Ergebnis kommen kann.

Die Übergabe des MVC durch Oracle hat länger gedauert als geplant. Was war der Hintergrund und kannst du Tipps für mögliche zukünftige Community-JSRs geben?

Die besondere Schwierigkeit für MVC war sicherlich, dass die Übergabe des JSR mitten im Prozess stattfand. Dadurch wurde sie aus rechtlicher Sicht sehr viel komplizierter. Im JCP hat der Spec Lead auch immer alle Rechte am geistigen Eigentum der Spezifikation. Bei einem Wechsel des Spec Lead müssen diese Rechte auf den neuen Leiter übertragen werden. Wie man sich vorstellen kann, ist das ein komplexer und zeitaufwändiger Prozess. Glücklicherweise handelte es sich bei dieser Übergabe um einen Sonderfall, der gerade deshalb sehr kompliziert war. Ich denke, dass sich die Situation für zukünftige Community JSRs deutlich einfacher gestaltet, wenn der entsprechende JSR von Anfang an von der Community geleitet wird. In diesem Fall ist die rechtliche Situation von Beginn an klar geregelt und die Expert Group kann sich auf die eigentliche Arbeit konzentrieren.

Wie sind jetzt die rechtlichen Rahmenbedingungen bezüglich des Quellcodes und insbesondere des TCK?

Nach Übernahme des JSR haben wir uns schnell für eine Änderung der Lizenz entschieden. Sowohl die Spezifikation als auch die Referenz-Implementierung und das TCK stehen inzwischen unter der freien Apache-Lizenz. Wir sind der Meinung, dass eine freie Lizenz ganz wesentlich für den Erfolg des JSR ist. Nur sie erlaubt Herstellern und Endnutzern einen sehr flexiblen Umgang mit der Spezifikation und öffnet die Tür für Beiträge aus der Community. Bei der Entscheidung bezüglich der Lizenzierung waren besonders CDI und Bean Validation unsere Vorbilder, die von Red Hat seit jeher in sehr offenerer Art geführt wurden.

MVC ist aus dem Java-EE-8-Umbrella herausgenommen worden und läuft jetzt als unabhängiger JSR. Wie bewertest du die Konsequenzen?

Nachdem bekannt wurde, dass MVC 1.0 nicht Bestandteil von Java EE 8 sein wird, ist die Spezifikation aus vielen Köpfen verschwunden. Viele haben diese Entscheidung als Todesurteil für MVC interpretiert. Faktisch heißt das aber lediglich, dass Java-EE-8-Applikationsserver nicht zwingend MVC unterstützen müssen. Aber natürlich dürfen sie mehr Funktionalität bieten, als es die Java-EE-8-Spezifikation vorschreibt. Unterstützt der Applikationsserver MVC trotzdem nicht, muss man in diesem Fall lediglich eine MVC-Implementierung mit in seinem WAR-Archiv verpacken. Auch wenn die MVC-1.0-Spezifikation nicht in Java EE enthalten ist, bleibt sie natürlich trotzdem ein Standard des Java Community Process – mit allen positiven Effekten wie beispielsweise der Langlebigkeit und Stabilität des API. Außerdem ist die aktuelle Situation natürlich noch mehr ein Ansporn, besonderen Wert darauf zu legen, dass die Referenz-Implementierung auch in allen Applikationsservern und in Java SE funktioniert, etwa in einem Apache Tomcat oder Jetty. Natürlich sind wir jetzt unabhängig vom Zeitplan für Java EE 8, der aufgrund der genannten Verzögerungen für MVC sowieso kaum einzuhalten gewesen wäre.

War es im Nachhinein die richtige Entscheidung, auf der JAX-RS-Spezifikation als Basis für MVC aufzusetzen?

Aus meiner Sicht war es definitiv eine gute Idee. JAX-RS bietet bereits viel Funktionalität, die auch für MVC in ähnlicher Form benötigt wird. Wäre JAX-RS nicht Basis gewesen, hätten viele Konzepte dupliziert werden müssen, was für die Einheitlichkeit der Plattform sicherlich nicht von Vorteil gewesen wäre. Außerdem ist das JAX-RS-API unter Entwicklern gut bekannt. Der Einstieg in MVC 1.0 fällt somit tendenziell leichter. Trotzdem war die Entscheidung in der Expert Group nicht unumstritten. Der Nachteil ist sicherlich, dass wir uns in gewisser Weise abhängig von JAX-RS und somit von dessen Funktionsumfang und Entwicklungsgeschwindigkeit gemacht haben.

In der Java-EE-Panel-Diskussion im JavaLand kamen ein paar Abhängigkeiten gegenüber JAX-RS zutage, die im Rahmen des Java-EE-8-Prozesses ignoriert worden waren. Wie ist da der aktuelle Stand?

Es war in der Tat so, dass die JAX-RS-Spezifikation einige Lücken aufwies, die für uns zum Problem wurden. Die JavaLand-Konferenz bot eine sehr gute Gelegenheit, diese Thematik im persönlichen Gespräch zu erörtern und nach Lösungswegen zu suchen. Glücklicherweise hat die JAX-RS-Expert-Group inzwischen reagiert und die Spezifikation entsprechend erweitert – trotz des Termindrucks hinter EE 8. Die genannten Änderungen werden Teil von JAX-RS 2.1 sein, sodass wir auf der neuen Funktionalität aufbauen können.

Warum braucht das Java-Ökosystem MVC?

Die Frage nach dem besten Web-Framework wird seit jeher kontrovers diskutiert. Es gibt verschiedene technische Ansätze, die je nach Anwendungsfall mehr oder weniger gut funktionieren. Mit Java Server Faces ist in Java EE seit mehr als zehn Jahren ein komponentenorientiertes Web-Framework enthalten. In vielen Fällen sind aktionsbasierte Frameworks besser geeignet, da sie leichtgewichtiger sind und üblicherweise mehr Kontrolle über technische Details bieten. Für Java EE soll MVC 1.0 diese Lücke schließen. Natürlich ist beispielsweise Spring MVC ein sehr populärer Vertreter dieser Kategorie und in vielerlei Hinsicht auch Vorbild für MVC 1.0. Trotzdem wäre es in den meisten Projekten nicht sinnvoll, Spring einzig und allein für das Web-Framework einzubinden, obwohl man eigentlich auf Java EE setzen möchte. Daher denke ich, dass MVC 1.0 ein wichtiger Standard für die Plattform ist.

Wie ist der aktuelle Stand der Spezifikation und wie sieht der Zeitplan für das erste Release aus?

Leider haben die Übergabe der Spezifikation und die Aktualisierung der Lizenz sehr viel Zeit gekostet. Zusätzlich hat Oracle die *„java.net"*-Plattform eingestellt, auf der bisher unsere komplette Infrastruktur bereitstand. Somit mussten wir zu allem Überfluss auch noch ein neues Zuhause für die Spezifikation finden. Das hat dafür gesorgt, dass wir bezüglich des Zeitplans umdenken mussten. Die Spezifikation selbst ist bereits auf einem sehr guten Stand. Trotzdem wollen wir in den kommenden Monaten alle Aspekte noch einmal durchgehen und genau prüfen. Es gibt auch durchaus einige Punkte, die noch komplett offen sind. Dazu gehört beispielsweise, dass der Mechanismus für das Data Binding die Sprache korrekt berücksichtigen muss, damit die Verarbeitung von Dezimalzahlen und Datumsangaben korrekt funktioniert. Aktuell planen wir, im Herbst in das „Public Review" zu gehen. Danach folgt dann Anfang 2018 der „Proposed Final Draft" und – wenn alles gut geht – Mitte 2018 das „Final Release".

Was sind die nächsten Schritte und wie kann euch die Community dabei am besten unterstützen?

Wie bereits erwähnt, planen wir, die Spezifikation in den nächsten Monaten zu stabilisieren. Wir müssen uns dann bei jedem Aspekt fragen: Ist die Formulierung genau genug? Sind alle Spezialfälle abgedeckt? Gibt es Widersprüche? Ist noch unerwünschter Interpretationsspielraum vorhanden? Deckt die Spezifikation die Anforderungen realer Projekte ab? Besonders der letzte Punkt ist natürlich ganz wesentlich. Die Expert Group kann stets nur aufgrund der eigenen Erfahrung urteilen. Vielleicht hat der normale Entwickler ja eine ganz andere Sichtweise. Ist es aktuell einfach genug, bestimmte Standardfälle zu implementieren? Diese Fragen können uns natürlich am besten Entwickler beantworten, die sich im täglichen Job mit Web-Applikationen beschäftigen. Jeder ist eingeladen, an den Diskussionen auf der Mailingliste teilzunehmen.

Links

[1] *Homepage: https://www.mvc-spec.org*
[2] *Mailing List: https://groups.google.com/forum/#!forum/jsr371-users*
[3] *Issue Tracker: https://github.com/mvc-spec/mvc-spec/issues*
[4] *Reference-Implementation: https://www.mvc-spec.org/ozark*
[5] *Twitter: https://twitter.com/mvc_spec*

Hinweis: Die DOAG Java Community möchte den MVC JSR im Rahmen des „Adopt a JSR"-Programms unterstützen. Hierzu wird es in der zweiten Jahreshälfte 2017 Veranstaltungen geben, die unter anderem über Twitter („@DOAGJava" beziehungsweise „#DJC") und im iJUG-Newsletter angekündigt werden.

Zur Person:
Christian Kaltepoth

Christian Kaltepoth *(„christian@kaltepoth.de")* arbeitet als Senior Developer bei der ingenit GmbH & Co. KG in Dortmund. Sein Schwerpunkt ist die Entwicklung von webbasierten Unternehmensanwendungen auf Basis von Java EE. Seit Mai 2017 ist er zusammen mit Ivar Grimstad MVC 1.0 Specification Lead. Twitter: „@chkal"

iJUG Verbund

Alle Mitglieder auf einen Blick

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

www.ijug.eu

Impressum

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

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

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

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

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

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

Fotonachweis:
Titel: © Elena Rankova/ DOAG Diestleistungen GmbH
Grafik S. 9: © Sergey Nivens/123RF
Grafik S. 20: © kran77/Fotolia
Grafik S. 23: © Chris Titze Imaging/Fotolia
Grafik S. 33: © convisum/123RF
Grafik S. 38: ©Oleksandr Omelchenko/123RF
Grafik S. 43: © Samantha Craddock/123RF
Foto S. 48: © hxdyl/123RF
Grafik S. 52: © Alexander Kirch/123RF

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

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

Inserentenverzeichnis

iJUG Verbund

Werden Sie Mitglied im iJUG!

20% Rabatt auf JavaLand -Tickets

Ab 10,- EUR im Jahr erhalten Sie ein Jahres-Abonnement der Java aktuell

Mitglied im **J**ava **C**ommunity **P**rocess

www.ijug.eu


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