Java aktuell

iJUG Verbund
www.ijug.eu

Coding Continuous Delivery
Hilfreiche Werkzeuge für die Jenkins-Pipeline

Besser Programmieren
Die Entwicklungsumgebung optimal nutzen

Aus der Praxis
Microservices richtig einsetzen

Java und mehr

Ausgabe 03/2018
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 03

DOAG 2018
Logistik + IT

14. Juni 2018 in Köln

logistik.doag.org

Java und mehr

Die Artikel in dieser Ausgabe zeigen einmal mehr, was alles hinter dem Java-Ökosystem steckt. An erster Stelle stehen Erfahrungsberichte über die Implementierung neuer Technologien: Andreas Weigel und Jakob Fels zeigen, wie sie einen Monolithen erfolgreich in Microservices umgewandelt haben. Ebenfalls um Microservices geht es in dem Artikel von Daniel Clasen und Jan Nonnen, die ein Medizinsoftware-Projekt mit Spring Cloud realisierten, in dem die gesamte Architektur als komponiertes System umgesetzt wurde. Er stellt die Erfahrungen der Autoren vor, erzählt von den Herausforderungen und hinterfragt kritisch die Vor- und Nachteile der gewählten Architektur.

„REST – Versprechen und Wirklichkeit" schildert die Erfahrungen von Thomas Bayer, der auf die Nachteile von REST aufmerksam macht und ermutigt, Alternativen wie GraphQL oder JSON RPC auszuprobieren. Dr. Marius Hofmeister rückt neue Sicherheitsrisiken in den Fokus und stellt die OWASP-Top-10 vor, eine weltbekannte Rangliste der wichtigsten Sicherheitsrisiken für Anwendungen im Web, die in regelmäßigen Abständen vom Open Web Application Security Project (OWASP) herausgegeben wird.

Auch die Programmierung selbst entwickelt sich weiter. Wolfgang Nast stellt besseren Java-Code außerhalb der Automatismen vor und der bereits dritte Teil der Jenkins-Serie von Johannes Schnatterer gibt interessante Einblicke in das Coding Continuous Delivery. Thorsten Kohn berichtet über reaktive Programmierung mit Java und Spring; Christian Nockemann präsentiert eine Auswahl an OO-Design-Patterns, erweitert um Lambdas.

Auch das Java-Umfeld kommt nicht zu kurz: Carsten Wiesbaum zeigt einige Entwicklungen der letzten Jahre auf und wägt ab, ob Agilität ein valider Ansatz für die Lösung der daraus entstehenden Herausforderungen ist. In „Lass uns mal skypen!" gibt Steven Schwenke Tipps, wie man Remote Meetings erfolgreich moderiert.

In diesem Sinne wünsche ich euch allen viel Spaß beim Lesen dieser Ausgabe und freue mich wie immer auf euer Feedback an redaktion@ijug,eu.

Ihr

Wolfgang Taschner
Chefredakteur Java aktuell

Inhalt

Die OWASP-Top 10 ist eine weltbekannte Rangliste der wichtigsten Sicherheitsrisiken für Anwendungen im Web

Mit jeder neuen Java-Version kamen auch Verbesserungen in der Art und Weise, wie Code besser geschrieben werden kann.

3 Editorial

6 Das Java-Tagebuch
Andreas Badelt

10 Neue Sicherheitsrisiken im Fokus
Dr. Marius Hofmeister

14 Vom Monolithen zu Microservices – ein Erfahrungsbericht
Andreas Weigel und Jakob Fels

19 Besserer Java-Code – außerhalb der Automatismen
Wolfgang Nast

24 Is Agile Eating up the World?
Carsten Wiesbaum

30 Microservices aus anderen Gründen – ein Erfahrungsbericht
Daniel Clasen und Jan Nonnen

Ein Grund für die weite Verbreitung von REST sind die geringen Hürden für die Nutzung als API

Design Pattern gehören mittlerweile zu den Standardwerkzeugen fast aller Java-Entwickler

35 REST – Versprechen und Wirklichkeit
Thomas Bayer

41 „Lass uns mal skypen!" –
Remote Meetings erfolgreich moderieren
Steven Schwenke

46 Reaktive Programmierung
mit Java und Spring
Torsten Kohn

51 Coding Continuous Delivery –
hilfreiche Werkzeuge für die Jenkins-Pipeline
Johannes Schnatterer

56 OO-Design-Patterns erweitert
um Lambdas – eine Auswahl
Christian Nockemann

66 Impressum/Inserenten

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 ersten Quartal 2018.

4. Januar 2018

Java EE Re-Branding – mal wieder

Oracle hat sich für die Zukunft von Java EE klar positioniert, was die konkrete Namensgebung jenseits des Projektnamens „EE4J" angeht: Die Marke „Java EE" soll bei Oracle bleiben und nicht für zukünftige EE4J genutzt werden; ebenso sollen auch keine neuen Packages mehr unter „javax" erzeugt, bereits bestehende können aber für Updates der jeweiligen APIs weiter genutzt werden. Das hat natürlich in der Community nicht für Jubelschreie gesorgt und schon im Herbst mal wieder die „Java EE Guardians" auf den Plan gerufen. Während das EE4J-Projekt bereits öffentlich nach einem neuen Markennamen sucht, wollen die Guardians mit einem „Joint Community Open Letter on Java EE Naming and Packaging" den Druck auf Oracle erhöhen: Die Umbenennung von J2EE in Java EE stifte nach zehn Jahren immer noch Verwirrung, also bitte keine weitere; noch wichtiger sei es aber, das Package „javax" auch für neue APIs weiterzuverwenden. Als möglichen Lösungsweg schlagen sie vor, beschränkte Nutzungsrechte für „javax.enterprise" an die Eclipse Foundation zu geben. Auch eine fortgesetzte – beschleunigte – Standardisierung innerhalb des JCP, der dann weiterhin die Namen verwaltet, wird als Option gesehen. Mal sehen, wie Oracle reagiert – beziehungsweise ob.

https://github.com/eclipse-ee4j/ee4j/issues/1

16. Januar 2018

Testing Wars

Oje, da hatte sich Antonio Goncalves (unter anderem durch seine Java-EE-Bücher bekannt) mit einem Tweet wohl bei einigen Java-EE-Fans unbeliebt gemacht und fühlte sich zu einer genaueren Erklärung genötigt: „I've reached a point where I can test Spring code in a couple of minutes, and Java EE code in a couple of hours", hieß es am 7. Januar auf Twitter, was heftige Reaktionen provozierte, obwohl es nur um Spring und EE und keine tieferliegenden Konflikte wie zwischen dem HSV und Werder Bremen geht ;-). In einem Blog-Eintrag führt er nun aus, was er genau meint: Dass Unit-Tests in einem „managed environment" wie EE oder Spring so gut wie nutzlos, Integrationstests jedoch entscheidend sind. Und dass Spring von Anfang an mit Blick auf Testbarkeit entwickelt wurde, sodass das Schreiben von Integrationstests extrem einfach ist, verglichen mit proprietären EE-Frameworks wie Arquillian und entsprechenden APIs von TomEE oder Payara. Aus der Perspektive eines Entwicklers, der sich nur innerhalb von Spring bewegt, hat er sicher recht – die Tests sind bei Java EE nicht standardisiert und vielleicht ein bisschen umständlicher. Aber da bei EE(4J) ja sowieso gerade Aufbruchstimmung herrscht, müsste sich doch auch dieses Problem endlich lösen lassen.

https://antoniogoncalves.org/2018/01/16/java-ee-vs-spring-testing

17. Januar 2018

Oracle antwortet den Guardians

Oracle hat tatsächlich binnen zwei Wochen auf den Open Letter der Java EE Guardians reagiert – was bei einem so heiklen, von Anwälten besetzten Thema vermutlich ziemlich schnell ist. Die Antwort hat Will Lyons gestern über die EE4J Community gegeben. Im Wesentlichen sagt er aber nichts, was nicht schon ausgetauscht worden wäre. Er bestätigt noch einmal, dass Oracle unter anderem die TCKs freigeben und auch die Bildung der neuen Marke unterstützen werde. Aber „Java EE" und „javax.*" seien Namen, die die Marke Java einsetzen und damit auf Oracle und von Oracle verwaltete Prozesse verweisen – somit seien sie ein wichtiger „Source Identifier" für die Nutzer. Oracle glaube daher, dass für einen neuen (offeneren und von Oracle unabhängigen) Prozess auch ein neuer Namespace und ein neuer Markenname benötigt werde. Und der Glaube scheint stark zu sein: Die Guardians beziehungsweise diejenigen aus der Community mit gegensätzlichen Überzeugungen werden hier wohl auf Granit beißen. Mark Little, „VP of Engineering" bei Red Hat, versucht dann auch gleich das Thema ein für alle Mal zu beenden: Seine Position sei inhaltlich auch eher die der Guardians, aber er verstehe auch die Oracle-Position und wisse, dass kein noch so hoher Einsatz den Hersteller davon abbringen werde. Viel wichtiger sei der bereits gemachte Schritt hin zur Eclipse Foundation, und jetzt müssten alle an einem Strang ziehen und die Sache zum Erfolg führen.

https://jaxenter.com/ee4j-will-not-inherit-the-java-ee-name-140549.html

17. Januar 2018

EE4J und der Standard

Nachdem wohl alle eingesehen haben, dass weitere Diskussionen um die zukünftige Namensgebung zwecklos sind, wird auch gleich wieder das andere große Thema in der öffentlichen Mailing-Liste diskutiert: Wie sieht es mit einer weiteren Standardisierung aus? Mark Little erinnert sich auf Anfrage von Reza Rahman, dass unter anderem OASIS und das W3C schon in früheren Diskussionen Thema zwischen Oracle, Red Hat, IBM und der Eclipse Foundation waren. Die überwiegende Meinung war aber wohl, dass sie für die eher Entwickler-zentrierten Aktivitäten rund um Java EE / EE4J nicht der richtige Ort wären. Statt Code, Standardisierung und alle zugehörigen Aktivitäten nun über zwei Organisationen zu verteilen und unter anderem Doppel-Mitgliedschaften erforderlich zu machen, wäre es besser, maßgeschneiderte Prozesse innerhalb der Eclipse Foundation dafür zu entwickeln. Klingt vernünftig, aber erstmal muss das ganze Projekt jetzt ins Rollen kommen.

https://dev.eclipse.org/mhonarc/lists/ee4j-community/msg00884.html

19. Januar 2018

Ein Inkubator für das JDK

Das JDK erhält einen Inkubator. JDK Enhancement Proposal 12 – Incubating Language and VM Features – beschreibt die Motivation und die Details: Neue Features sollen zunächst mit spezieller Kennzeichnung in Releases aufgenommen werden können, um Feedback zu sammeln. Je nach Feedback kann ein Feature dann in Folge-Releases noch größere, nicht kompatible Änderungen erfahren oder auch wieder komplett verschwinden. Damit sie also nicht versehentlich benutzt werden und später die Überraschung groß ist, müssen sie über das Flag „--incubating" und die jeweilige Release-Nummer freigeschaltet sein (Letztere muss immer der des benutzten JDK entsprechen, also beispielsweise „java --incubating 11 ..." für das JDK 11). Jeglicher Grundlage entbehren Gerüchte, dass das Java-Modulsystem im Nachhinein noch ein „incubating"-Flag erhält.

openjdk.java.net/jeps/12

19. Januar 2018

Längere Public Updates für Java SE 8

Oracle hat die Frist für kostenlose Updates für Java 8 von September 2018 auf mindestens Januar 2019 verlängert, für private (nichtkommerzielle) Nutzung sogar bis Ende 2020 (im Wesentlichen geht es dabei um Sicherheits-Updates). Das Ganze schließt Java Web Start mit ein – kostenpflichtiger Support ist hier bis mindestens März 2025 zugesichert. Auch Applets sollen weiter bis mindestens März 2019 unterstützt werden, soweit die Browser-Hersteller mitspielen. Zu Java 9 gibt es allerdings nichts Neues – der Support dürfte also wie geplant mit dem Folge-Release im März 2018 enden. Darüber, dass Java 9 und 10 keine „Long Term Support"-Releases sind (auch nicht für zahlende Oracle-Kunden), sondern erst Java 11 im Herbst 2018 wieder langjährigen Support bieten wird, und was das für die Adoption bedeutet, wird ja viel diskutiert. Andere Hersteller haben auch darauf reagiert. Azul zum Beispiel hat angekündigt, für seine an die OpenJDK-Termine angepassten Releases immer überlappenden (kostenpflichtigen) Support zu bieten und in Jahren ohne „LTS Release" ein Release mit zumindest 18 Monaten Support („Medium-Term").

http://www.oracle.com/technetwork/java/javase/eol-135779.html

24. Januar 2018

Mike Milinkovich zum Fortschritt von EE4J

Die EE4J-GitHub-Repositories werden langsam mit Inhalt gefüllt. Nun gibt der Chef der Eclipse Foundation einen Ausblick auf die nächste Phase: Zunächst soll möglichst schnell ein Java-EE-8-kompatibles Release erzeugt werden, um „der Welt zu demonstrieren, dass die Projekte liefern können", und natürlich, um eine erste Grundlage zu haben, auf der alle aufbauen können. Einige hätten ihn zuletzt gefragt, ob sie jetzt anfangen könnten, die APIs direkt in den EE4J-Projekten zu ändern. Seine Antwort: „Bitte noch nicht jetzt." Zunächst geht es um das Festlegen und Fein-Tunen der Prozesse und Regeln und um das erste EE-8-kompatible Release. Aber natürlich stehe es allen frei, die GitHub-Repositories zu „forken", um schon an neuen Prototypen zu arbeiten.

https://jaxenter.de/java-ee-ee4j-66530

2. Februar 2018

Jakarta EE oder Enterprise Profile?

Die öffentliche Suche nach einem neuen Markennamen geht in die zweite Runde. Aus den eingegangenen Vorschlägen hat das Projektteam zwei ausgewählt – „Jakarta EE" und „Enterprise Profile" – und ein Google-Formular erzeugt, mit dem jeder (registrierte Google-Nutzer) bis zum 23. Februar abstimmen kann. Jakarta scheint viel Zustimmung zu finden, da die gleichnamige Hauptstadt Indonesiens auf Java liegt und die alten Kaffeewitze weiter funktionieren. Die Zustimmung von Apache wurde schon eingeholt (die älteren Java-Entwickler werden sich noch an das im Jahr 2011 eingestellte Apache-Jakarta-Projekt erinnern).

https://www.infoq.com/news/2018/02/JavaEENewNameJan18

8. Februar 2018

Java EE: Entwurf für die Gründung der EE.next Working Group vorgelegt

Die Eclipse Foundation hat den Entwurf einer Charta für die Eclipse EE.next Working Group veröffentlicht, die wohl auch die Frage um die zukünftige Standardisierung von Enterprise Java beantworten soll. Die Working Group kann man sich in Mike Milinkovichs eigenen Worten als „Ersatz für den Java Community Process für Java EE vorstellen". Die Charta wäre dann also in etwa das Äquivalent zum JCP Process Document. Die EE.next Working Group ist in mehrere Gremien unterteilt: Das Steering Committee soll sich generell mit dem EE4J-Ökosystem beschäftigen, das Specification Committee nur mit Fragen rund um die Entwicklung und technische Spezifikation. Daneben sind noch ein Marketing Committee und ein Enterprise Requirements Committee geplant. Letzteres soll wichtige „Influencer Members" der Eclipse Foundation, die im Gegensatz zu „Strategic Members" keine eigenen Entwickler beisteuern, mit dem Projekt vernetzen. Das existierende Project Management Committee befasst sich mit dem Tagesgeschäft, wird also durch die eher strategisch arbeitende Working Group nicht ersetzt, sondern ergänzt – die Working Group soll auch kein Weisungsrecht gegenüber dem PMC haben, sondern eher ein Partner sein.

https://www.eclipse.org/org/workinggroups/eclipse_ee_next_charter.php

13. Februar 2018

JDK 10 – erster Release-Kandidat

Der erste Release-Kandidat von Java 10 ist erschienen – der neue „Release Train" ist noch nicht ganz so pünktlich wie die Schweizerischen Bundesbahnen, aber den ICE können sie schon schlagen. Verglichen mit Java 9 ist das Release natürlich klein. 12 Features sollen am 20. März 2018 offiziell freigegeben werden, von denen die meis-

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

ten wohl keine direkten Auswirkungen für die breite Masse der Entwicklerinnen und Entwickler haben. Neben einigem Housekeeping und Performance-Tuning wird es jedoch beispielsweise einen parallelen Full GC für den G1 und Typinferenz für lokale Variablen geben. Darüber hinaus werden wohl einige Methoden verschwinden, die seit Release 9 als „deprecated" mit dem neuen Zusatz „forRemoval" gekennzeichnet sind. Betroffen sind einige Methoden von java.lang.SecurityManager – allesamt seit JDK 1.1 bzw. 1.2 deprecated.
java.lang.Runtime.getLocalizedInputStream/getLocalizedOutputStream

14. Februar 2018

EE4J – Erste Projekte auf GitHub

Die ersten EE4J-Projekte sind seit Mitte Januar im neuen Repository auf GitHub angekommen; nach und nach sollten weitere ehemalige Java-EE-Projekte eintrudeln. So richtig einheitlich sieht es nicht aus: Das JSON-P-Repository enthält API und Referenz-Implementierung. Für JAX-RS ist nur das API unter „/eclipse-ee4j" zu finden, weil die RI schon vorher auf GitHub zu Hause war (siehe „github.com/jersey"). Das JMS-API ist ebenfalls unter „/eclipse-ee4j" zu finden, die Implementierung OpenMQ hat aber gleich zwei Kopien: die neue unter „/eclipse-ee4j" und die schon vorhandene unter „/javaee". Da müsste demnächst mal aufgeräumt werden.
github.com/eclipse-ee4j

19. Februar 2018

NetBeans 9 Beta

Die Beta-Version von NetBeans 9.0 – des ersten NetBeans-Release unter der Regie der Apache Software Foundation – ist da. Es bietet eine vollständige Unterstützung von Java 9, seines Modulsystems (inklusive „Modular Projects") sowie der JShell.
http://wiki.netbeans.org/NetBeans_9

23. Februar 2018

Java EE, Spring und EE4J

Java-EE-Guardian Jean-François James hat einen interessanten Kommentar im Guardians-Blog verfasst. „I have a dream", startet er im Martin–Luther-King-Ton, „that one day the whole Java community would join forces to strengthen its position on server-side applications." Es geht um Java EE und Spring, die schon immer „beste Feinde" waren. Wenn aber der Übergang zu EE4J gelinge, könnte es doch möglich werden, dass die beiden Projekte direkt zusammenarbeiten – was in einer Welt mit immer größeren Alternativen dringend nötig wäre. EE4J soll dazu unabhängige, aber kompatible Spezifikationen „à la carte" liefern und Spring sich direkt an denjenigen beteiligen, die für sie von Interesse sind (anders als bisher, wo Spring einige Spezifikationen einfach übernimmt und im Gegenzug Java EE/der JCP neue Features des agileren Spring in seine Standards aufnimmt – ohne wirkliche Zusammenarbeit). Ein schöner Gedanke, um das Tagebuch für diese Ausgabe abzuschließen.
https://jefrajames.wordpress.com/2018/01/09/where-is-java-ee-going/comment-page-1#

27. Februar 2018

Java EE heißt jetzt Jakarta EE, auch sonst ändert sich ganz viel

Aus Java EE wird Jakarta EE – zumindest nach außen, der interne Projektname „EE4J" wird beibehalten. 64,4 Prozent der knapp 7.000 Entwickler, die sich an der Umfrage beteiligt haben, bevorzugten diesen Namen gegenüber „Enterprise Profile". Auch ein paar andere Namen, die nicht öffentlich ausgeschrieben waren, stehen jetzt fest: Die Referenz-Implementierung GlassFish heißt in Zukunft „Eclipse Glassfish". Was früher der „Java Community Process" für EE erledigt hat, übernimmt in Zukunft die „Jakarta EE Working Group" bei der Eclipse Foundation – insbesondere auch die Standardisierung, so wie es aussieht. Eine der nächsten Aufgaben der Working Group wird jedoch erstmal die Ausarbeitung eines Kompatibilitätsprogramms sein, um festzulegen, welche Produkte sich demnächst mit der Marke „Jakarta EE" schmücken dürfen.
https://mmilinkov.wordpress.com/2018/02/26/and-the-name-is

1. März 2018

Spring Boot 2.0 unterstützt Java SE 9

Spring Boot 2.0 ist offiziell freigegeben und bietet unter anderem Unterstützung für Java 9 (Baseline ist Java 8), Kotlin 1.2.x sowie reaktive Web-Programmierung mit WebFlux beziehungsweise WebFlux.fn.
https://spring.io/blog/2018/03/01/spring-boot-2-0-goes-ga

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich 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").

Donnerstag
5. Juli 2018

Kultur- und Kongresszentrum
Liederhalle Stuttgart

Besucheranmeldung
ab 18. April 2018
www.java-forum-stuttgart.de

JAVA FORUM stuttgart 2018

Donnerstag, 5. Juli 2018
JFS - Java Forum Stuttgart

Mit nun 21 Jahren Tradition ist das JFS zur festen Institution geworden. Das Forum bietet den Teilnehmern die Möglichkeit, sich umfassend über Themen zu Java bzw. im JVM-Umfeld zu informieren. Die Breite wird erreicht durch Grundlagenvorträge, Erfahrungsberichte und Informationen über konkrete Projekte. Produkte werden sowohl in Form von Vorträgen als auch als Demonstrationen an den Ausstellungsständen präsentiert.

Mittwoch, 4. Juli 2018
Workshop „Java für Entscheider“

Eintägige Überblicksveranstaltung für Entscheider aus der IT oder IT-nahen Einsatzfeldern wie Abteilungs- oder Teamleiter, deren Mitarbeiter im Java Umfeld tätig sind oder sein sollen.

40 Aussteller

Dieses Jahr waren wir bereits mit den Ständen Anfang Februar ausgebucht

Möchten Sie beim JFS 2019 einen interessanten Fachvortrag einreichen? Dann melden Sie sich bitte rechtzeitig bei uns!

max. 48 Fachvorträge & zusätzlich einige „Pecha Kucha“ Kurz Vorträge

NOVATEC make IT happen! PRODYNA MIMACOM

Eine Veranstaltung der Java User Group Stuttgart e.V.

JUGS java user group stuttgart
www.jugs.org

Neue Sicherheitsrisiken im Fokus

Dr. Marius Hofmeister, OPITZ CONSULTING Deutschland GmbH

Die OWASP-Top-10 ist eine weltbekannte Rangliste der wichtigsten Sicherheitsrisiken für Anwendungen im Web. Sie wird in regelmäßigen Abständen vom Open Web Application Security Project (OWASP) herausgegeben. Ende 2017 wurde eine neue Version veröffentlicht, die zahlreiche Änderungen enthält. Neue Risiken sind hinzugekommen, alte wurden entfernt. Auch sonst gibt es einiges zu berichten.

Ziel von OWASP ist es, Einzelpersonen und Unternehmen dabei zu unterstützen, sichere Anwendungen zu entwickeln, zu kaufen und zu warten. Security für Software soll sichtbar sein. Die Non-Profit-Organisation legt Wert darauf, kommerzielle Produkte oder Services weder aktiv zu bewerben noch Empfehlungen auszusprechen, um die eigene Neutralität zu wahren *[1]*. Jeder, der Interesse an Security-Themen hat, ist aufgerufen, sich im Projekt einzubringen und einen Teil zu dessen Erfolg zu leisten. Auf den entsprechenden Webseiten finden sich Literatur, Werkzeuge und Standards zur Anwendungssicherheit. Konferenzen und Stammtische finden an zahlreichen Orten statt, regelmäßig auch in deutschen Städten.

Als eines von mehr als 90 aktiven OWASP-Projekten trägt die Top 10 dem übergeordneten Ziel der Schaffung von Awareness für Security seit der ersten Ausgabe 2003 kontinuierlich Rechnung. Die Rangliste der zehn wichtigsten Sicherheitsrisiken für Web-Anwendungen wird in der Regel alle drei bis vier Jahre neu aufgestellt und veröffentlicht.

Der lange Weg zur OWASP-Top-10

Grundlage für die Zusammenstellung der OWASP-Top-10 ist ein Pool von Daten. Im Mai 2016 wurde daher ein Data Call auf der OWASP-Website veröffentlicht. Mehr als 40 Einreichungen von Unternehmen, die auf Anwendungssicherheit spezialisiert sind, kann die neue Top 10 vorweisen. Darin enthalten sind Schwachstellen von Hunderten Organisationen sowie von mehr als 100.000 realen Applikationen und APIs. Um diese zu selektieren und zu priorisieren, finden unterschiedliche Faktoren Beachtung. Dazu gehören:

- Die Ausnutzbarkeit, Verbreitung und Auffindbarkeit einer Schwachstelle
- Die technischen Auswirkungen eines Angriffs *(siehe Tabelle 1)*

Durch die Multiplikation der gemittelten Einflussfaktoren mit den technischen Auswirkungen lässt sich so eine Rangfolge der Risiken ableiten. Keine Beachtung finden dabei die jeweiligen anwendungsspezifischen Bedrohungsquellen oder die geschäftsspezifischen Auswirkungen auf das konkrete Unternehmen. So kommen als Bedrohungsquellen einer Anwendung beispielsweise alle Nutzer des Internets oder alternativ auch nur unternehmensintern freigeschaltete Personen in Betracht. Ferner wird sich die Auswirkung eines erfolgreichen Angriffs auf das Unternehmen dahingehend unterscheiden, ob dabei hochsensible, personenbezogene Daten entwendet wurden oder nur wenige sicherheitsbedürftige Inhalte.

Neben den datengestützt erhobenen Risiken finden sich auch zwei sogenannte „Forward-looking Items" im Ranking. Es handelt sich hierbei um Risiken, die aus Expertensicht als für die Zukunft besonders relevant eingeschätzt werden. Sie wurden online über eine Umfrage unter Industrievertretern ermittelt, an der sich mehr als 500 Personen beteiligten.

Tabelle 2 stellt die OWASP-Top-10 aus dem Jahr 2017 im Vergleich zur Version von 2013 dar. Erkennbar ist, dass drei neue Risiken hinzugefügt und zwei zusammengeführt wurden (gelb markiert). Zwei Risiken aus der bisherigen Rangliste sind gänzlich verschwunden.

Bedrohungsquellen	Schwachstelle Ausnutzbarkeit	Schwachstelle Verbreitung	Schwachstelle Auffindbarkeit	Technische Auswirkungen	Auswirkungen auf das Unternehmen
Anwendungsspezifisch	Einfach	Sehr häufig	Einfach	Schwerwiegend	Anwendungs-/ Geschäftsspezifisch
	Durchschnittlich	Häufig	Durchschnittlich	Mittel	
	Schwierig	Selten	Schwierig	Gering	

Tabelle 1: Bewertungsschema für Risiken der OWASP-Top-10 [2]

OWASP-Top-10 – 2013		OWASP-Top-10 – 2017
A1: Injection	→	A1: Injection
A2: Fehler in Authentifizierung und Session-Management	→	A2: Fehler in Authentifizierung
A3: Cross-Site Scripting (XSS)	↘	A3: Verlust der Vertraulichkeit sensibler Daten
A4: Unsichere direkte Objektreferenzen	U	A4: XML External Entities (XXE) – NEU
A5: Sicherheitsrelevante Fehlkonfiguration	↘	A5: Fehler in Zugriffskontrollen – zusammengeführt
A6: Verlust der Vertraulichkeit sensibler Daten	↗	A6: Sicherheitsrelevante Fehlkonfiguration
A7: Fehlerhafte Autorisierung auf Anwendungsebene	U	A7: Cross-Site Scripting (XSS)
A8: Cross-Site Request Forgery (CSRF)	X	A8: Unsichere Deserialisierung - NEU
A9: Verwendung von Komponenten mit bekannten Schwachstellen	→	A9: Verwendung von Komponenten mit bekannten Schwachstellen
A10: Ungeprüfte Um- und Weiterleitungen	X	A10: Unzureichendes Logging & Monitoring – NEU

Tabelle 2: OWASP-Top-10 im Jahr 2017 im Vergleich zur Version von 2013 [2]

Im Vergleich zur ursprünglichen zeitlichen Planung ist die Top 10 im Jahr 2017 später als erwartet erschienen. Hintergrund waren kontroverse Diskussionen, die sich auf dem OWASP Summit 2017 entzündeten. Der dort diskutierte erste Release-Kandidat der Top 10 fand so wenig Anklang unter den Anwesenden, dass er schlichtweg abgelehnt wurde. Bemängelt wurden die zu geringe Datenbasis auf der einen Seite als auch die konkrete Auswahl der „Forward-looking Items" auf der anderen Seite. Die hier ursprünglich vorgeschlagenen Punkte „Ungenügende Angriffserkennung" und „Ungeschützte APIs" wurden zurückgestellt und später durch die Ergebnisse der Industrieumfrage ersetzt. Zwei der Gründerväter des Projekts, Dave Wichers und Jeff Williams, zogen sich im Laufe dieses Prozesses aus dem Projekt zurück. Die Leitung übernahm Andrew van der Stock mit den Stellvertretern Brian Glas, Neil Smithline und Torsten Gigler. Wer das Projekt in den letzten Jahren aufmerksam verfolgt hat, dem wird aufgefallen sein, dass sich die personellen Veränderungen auch in den neu veröffentlichten Top 10 wiederfinden. Große Teile des Dokuments wurden textuell neu verfasst, und häufiger als zuvor wird auf moderne Technologien und Architekturen wie Microservices und Single-Page-Applikationen Bezug genommen.

Die Risiken an der Spitze

An der Spitze der Top 10 rangieren wie schon im Jahr 2013 die Injection-Angriffe – trotz der Tatsache, dass der Bekanntheitsgrad dieser Schwachstelle größer kaum sein könnte. Offensichtlich ist noch viel Legacy-Code im Einsatz, der für die große Verbreitung der Schwachstelle sorgt. Schadhafte Daten werden ungefiltert an einen Interpreter weitergeleitet, der diese dann ausführt. Dies kann zu unerwünschten Lese- und Schreib-Operationen führen, bis hin zur feindlichen Übernahme eines Systems. Auch wenn die bekannteste Variante, SQL-Injection, meist die volle Aufmerksamkeit auf sich zieht, sollten je nach Anwendung auch Anfälligkeiten für beispielsweise LDAP-, XPath- oder NoSQL-Injection-Angriffe untersucht werden.

Im Gegensatz zu dieser konkreten Angriffsform auf Web-Anwendungen handelt es sich beim zweiten Risiko der Top 10 um eine grundsätzlichere Problematik: Der Punkt „Fehler in Authentifizierung" stellt fehlerhafte und damit anfällige Authentifizierungsmechanismen in den Mittelpunkt. Die Behebung von Schwachstellen in diesem Bereich ist meist aufwendiger, da Angriffe auf unterschiedliche Weise vonstattengehen können. Wie bei allen Risiken bietet OWASP in seinen Top 10 eine Liste von Literatur-Referenzen zum Umgang mit solchen Schwachstellen an; so wird, wenn möglich, der Einsatz von Multi-Faktor-Authentifizierung empfohlen, also die zeitgleiche Verwendung mehrerer unterschiedlicher Authentifizierungstechniken. Dies beugt Brute-Force-Attacken vor und erschwert das Entwenden von Zugangs-Informationen. Ferner ist zu erwarten, dass auch der Einsatz biometrischer Eigenschaften im Rahmen von Authentifizierungsszenarien zukünftig zunehmen wird. In den Top 10 auf Platz drei aufgerückt und damit an Bedeutung gewonnen hat der „Verlust der Vertraulichkeit sensibler Daten". Häufigstes Problem an dieser Stelle ist die nicht vorhandene oder mangelhafte Verschlüsselung sensibler Datenbestände. Zu Beginn steht eine Untersuchung des Schutzes, den Daten in der Übertragung und Speicherung brauchen. Wie sensibel sind die Daten, die erhoben werden? Werden aktuelle Verschlüsselungsalgorithmen eingesetzt? Ist eine Speicherung und Übertragung dieser Daten wirklich notwendig? Dies sind nur einige der Fragen, denen sich Entwickler stellen sollten.

Neue Risiken im Jahr 2017

Drei neue Risiken haben den Weg in die Top 10 gefunden. „XML External Entities" (XXE) wurde datengestützt erhoben, während „Unsichere Deserialisierung" und „Unzureichendes Logging & Monitoring" über die Industrie-Umfrage als „Forward-looking Items" Einzug gehalten haben.

Ursache für XML-basierte Angriffsszenarien ist die Möglichkeit, externe Entitäten anzugeben, die über einen URI referenziert werden. Wenn derartiges XML ungeprüft von Anwendungen entgegengenommen wird, können schützenswerte Daten freigelegt werden, bis hin zur Ausführung von Denial-of-Service-Attacken. Die „Billion Laughs"-Attacke, die über die Expandierung einer Milliarde Entitäten den zur Verfügung stehenden Speicherplatz eines Prozesses überschreitet, ist ein bekanntes Beispiel für einen XML-basierten Angriff. In jedem Fall empfiehlt es sich, wenn XML-Parser im Einsatz sind, diese entsprechend den Hinweisen des XML External Entity (XXE) Prevention Cheat Sheet *[3]* der OWASP zu konfigurieren.

Nächster Punkt sind die über Industrie-Umfragen ermittelten Risiken. Dass „Unsichere Deserialisierung" gewählt wurde, zeigt, wie sehr die Diskussionen der letzten Jahre (besonders auch im Java-

Umfeld) das Bewusstsein für diese Angriffsart geschärft haben. Durch Serialisierung werden strukturierte Daten zur Speicherung oder Übertragung auf eine sequenzielle Form abgebildet. Um einen Endpunkt zur Deserialisierung anzugreifen, müssen sogenannte „Gadgets" vorliegen, also ausnutzbare Klassen auf dem Klassenpfad der angegriffenen Anwendung. So kann es zu einer Remote Code Execution kommen, dem Ausführen von Schadcode auf den Servern der jeweiligen Webanwendung. Umfassende Abhilfe schafft hier einzig die Vermeidung von Deserialisierung aus nicht vertrauenswürdigen Quellen.

Das zweite „Forward-Looking Item" der Industrie-Umfrage, „Unzureichendes Logging & Monitoring", nimmt sich der Problematik an, dass Sicherheitslücken oftmals nicht zeitnah erkannt werden und lange verfügbar bleiben. In sensiblen Anwendungen ist es empfehlenswert, mithilfe von Detection-Points Ereignisse wie fehlerhafte Log-ins oder verdächtige Eingaben zu erkennen und dann bei Überschreitung gewisser Thresholds eine entsprechende Aktivität wie die Sperrung eines Accounts durchzuführen. Im Idealfall reagiert die Anwendung somit selbstständig auf Angriffe.

Neben diesen drei neu hinzugekommenen Risiken wurden „Unsichere direkte Objekt-Referenzen" und „Fehlerhafte Autorisierung auf Anwendungsebene" zusammengefasst. Dies kam insofern wenig überraschend, als es sich hierbei bereits in früheren Versionen der Top 10 um einen gemeinsamen Punkt handelte. Das Ziel, durch Aufsplittung höhere Aufmerksamkeit für die Thematik zu erreichen, wurde in den Augen vieler erreicht.

Altbekanntes auf wechselnden Plätzen

Weiterhin in den Top 10 vertreten sind die Risiken „Sicherheitsrelevante Fehlkonfiguration", „Cross-Site Scripting (XSS)" sowie die „Verwendung von Komponenten mit bekannten Schwachstellen". „Sicherheitsrelevante Fehlkonfiguration" rückt die fehlerhafte Konfiguration von Anwendungen, Frameworks, Applikations-, Web- und Datenbank-Servern sowie deren Plattformen in den Fokus. Voreinstellungen müssen gewartet und Updates regelmäßig eingespielt werden, damit bekannte Sicherheitslücken nicht allzu leicht ausgenutzt werden können.

„Cross-Site Scripting (XSS)" hat demgegenüber glücklicherweise einen Bedeutungsverlust in den Top 10 zu verzeichnen. Lange Zeit handelte es sich hierbei um die Schwachstelle mit der weitesten Verbreitung in Web-Anwendungen. Ähnlich wie bei den Injection-Angriffen liegt eine Ursache für diese Lücke darin, dass eine Anwendung ungeprüft Daten annimmt und diese dann zusätzlich noch ungeprüft wieder ausgibt. Angriffsziel ist der Browser des Website-Nutzers, in dem dann in der Regel JavaScript-Schadcode ausgeführt wird. Benutzerdaten können so ausgelesen und Sessions übernommen werden.

Das letztgenannte Risiko, „Verwendung von Komponenten mit bekannten Schwachstellen", thematisiert, dass häufig anfällige Komponenten eingesetzt werden, die dringend ausgetauscht oder aktualisiert werden sollten. Ungenutzte Abhängigkeiten sollten regelmäßig entfernt und Scans mit Werkzeugen wie den OWASP Dependency Checks *[4]* durchgeführt werden.

Ganz aus den Top 10 verabschiedet haben sich die alten Bekannten „Cross-Site Request Forgery (CSRF)" sowie „Ungeprüfte Um- und Weiterleitungen". Während Frameworks wie Java Server Faces (JSF) heutzutage standardmäßig Schutzmechanismen gegen CSRF bieten, hat die Ausnutzung ungeprüfter Um- und Weiterleitungen schlichtweg an Bedeutung verloren.

Neue Transparenz in den Top 10 und Durchhaltevermögen in der Umsetzung

Mit dem Jahr 2017 ist die OWASP-Top-10 transparenter geworden denn je. Auf GitHub *[5]* sind große Teile der Projekt-Kommunikation einsehbar. Die Beteiligung an aktuellen Diskussionen ist ohne große Umschweife möglich. Zu betonen ist jedoch, dass auch eine vollständige Beachtung der Rangfolge keine umfassende individuelle und ganzheitliche Analyse der eigenen Anwendung ersetzen kann. Unzählige Sicherheitsrisiken, die individuell gefährlich werden können, existieren über die Top 10 hinaus.

Gefordert ist in jedem Fall Durchhaltevermögen, wenn Sicherheitsthemen angegangen werden. Oft lassen sich Schwachstellen nicht von heute auf morgen beseitigen, geschweige denn erkennen. Aber wer mit Ausdauer dabeibleibt und das Ziel einer „möglichst sicheren" Webanwendung nicht aus den Augen verliert, der wird damit auch langfristig erfolgreich sein.

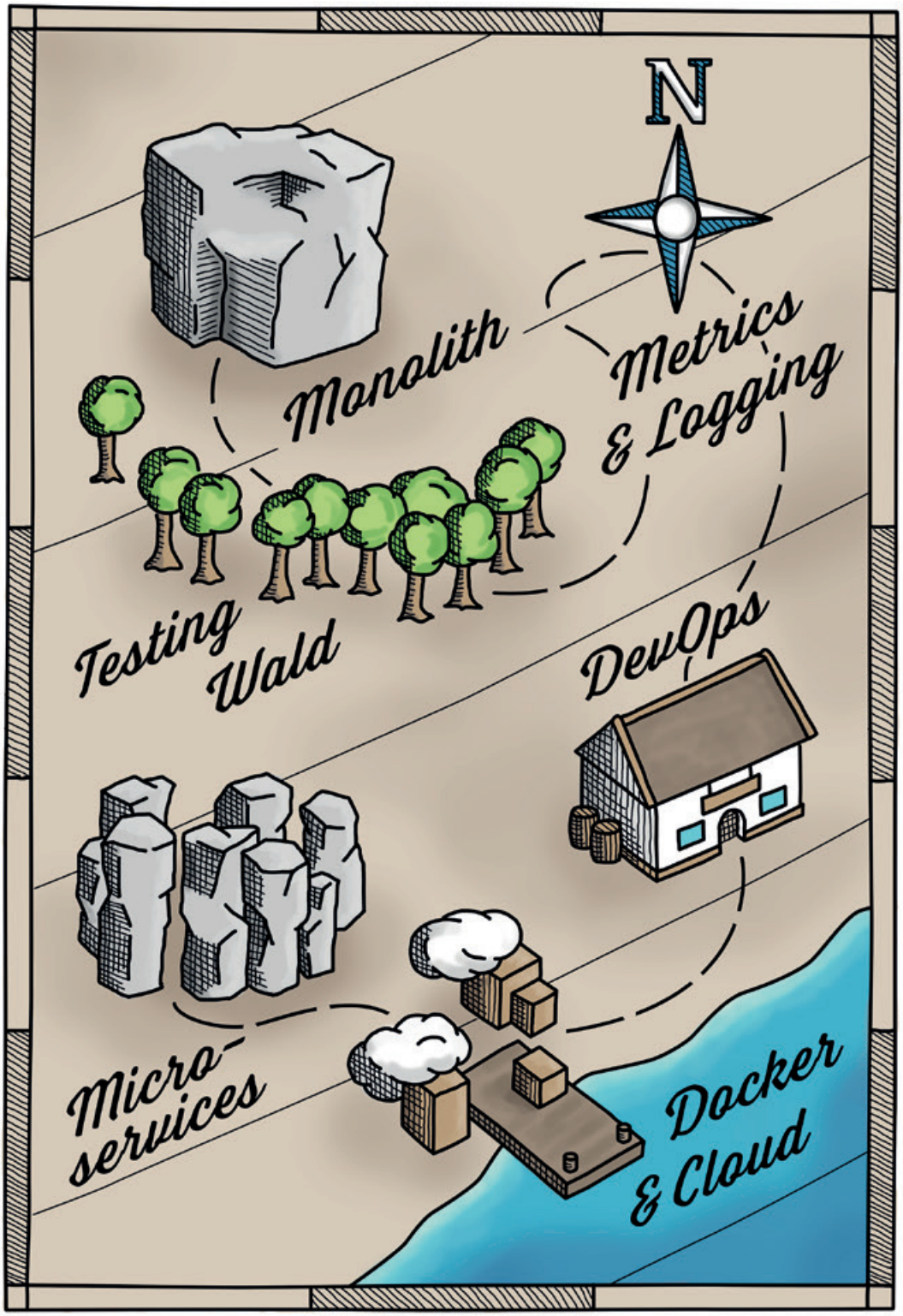
Literatur

[1] OWASP Website: *https://www.owasp.org*
[2] OWASP-Top-10 – 2017: *https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project*
[3] XML External Entity (XXE) Prevention Cheat Sheet: *https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet*
[4] OWASP Dependency Check: *https://www.owasp.org/index.php/OWASP_Dependency_Check*
[5] OWASP Top 10 GitHub Issues: *https://github.com/OWASP/Top10/issues*

Dr. Marius Hofmeister
marius.hofmeister@opitz-consulting.com

Dr. Marius Hofmeister ist als Senior Consultant und Entwicklungsleiter in Kundenprojekten bei OPITZ CONSULTING Deutschland beschäftigt und widmet sich schwerpunktmäßig dem Entwurf und der Implementierung von Web-Anwendungen in Back- und Frontend. Von besonderem Interesse für ihn sind dabei Security-Aspekte der verwendeten Technologien.

Vom Monolithen



zu Microservices

Vom Monolithen zu Microservices – ein Erfahrungsbericht

Andreas Weigel, synyx GmbH & Co. KG, und Jakob Fels, dm-drogerie markt GmbH + Co. KG

„Microservices" steht für ein Architektur-Muster, das den Fokus auf eine fachliche Modularisierung des gesamten Software-Stacks legt. Gestützt durch technischen Fortschritt und organisatorische Entwicklung, lassen sich damit besser nachvollziehbare Architekturen umsetzen, die viele Vorteile mit sich bringen.

Die Komplexität löst sich damit nicht auf, wird jedoch leichter handhabbar. Zudem verschiebt sie sich teilweise in andere Bereiche, zum Beispiel in der Infrastruktur bei der Bereitstellung einer Orchestrierung von Containern oder in der Organisation bei dem richtigen Schnitt der Domäne und den zuständigen Entwicklungsteams. Vorteile können dabei unter anderem Verständnis der Domäne, Geschwindigkeit im Release-Zyklus oder Skalierung der Software beziehungsweise der Entwicklungsteams sein.

Dieser Artikel zeigt, wie im Rahmen der Software-Entwicklung der Coupon-Verwaltung bei dm-drogerie markt eine monolithische Anwendung in eine Microservice-Architektur transformiert wurde, wobei zunächst gezielt mit einem strukturierten Monolithen angefangen wurde.

Neue Möglichkeiten durch Umdenken und Innovation

Herrscht mangelnde Kommunikation zwischen dem Entwicklungsteam und dem Fachbereich, beschränkt sich die Kommunikation weitestgehend auf den Austausch von Last- und Pflichtenheft. Somit ist es unumgänglich, dass der Funktionsumfang vom Fachbereich im Vorfeld vollständig definiert wird, das Entwicklungsteam darauf aufbauend die Hardware-Dimensionierung einplant und die Software-Entwicklung durchführt. Je nach Dauer der Entwicklung und mangels Feedback des Endanwenders ist schließlich unklar, ob das Ergebnis alle Bedürfnisse erfüllt.

Das Resultat ist ein Software-Entwicklungsprozess nach dem Wasserfallmodell. Parallele Entwicklung in der gleichen Codebasis führt zu Abstimmungsaufwänden und aufgeschobenen und damit seltenen Releases. Die Folge ist meist eine monolithische Anwendung mit einem komplexen und aufwendigen Software-Stack für den Betrieb, der nur schwierig zu skalieren ist.

Eine Reaktion darauf in der Software-Entwicklung ist die Realisierung von horizontalen Schnitten (Präsentations-, Logik-, Datenhaltungs-Schicht) zur Definition von Abhängigkeitsbeziehungen und Abstraktion von Komponenten. Um in diesem Rahmen Synergie-Effekte zu erzielen, entstand das Architekturmuster „Service-orientierte Architektur" (SOA), das sich auf die Orchestrierung von Services innerhalb der Logikschicht konzentriert. Das Ergebnis ist eine stetig wachsende Anwendung, die nur im Ganzen deployt werden kann, mit zunehmendem Funktionsumfang schwer überschaubar ist und hohes Potenzial für Seiteneffekte zwischen unabhängigen Domänen aufweist. Wachsende funktionale Anforderungen können nur langsam bedient werden, da viel Zeit in die Wartung und den Betrieb der Anwendung fließt.

Um die Time-to-Market zu minimieren, sind drei wesentliche Punkte zu adressieren. Zum Ersten sind ein Umdenken in der Art und Weise der Kommunikation und das gegenseitige Verständnis zwischen allen Projektbeteiligten zwingend notwendig, unabhängig von ihrer Rolle. Zum Zweiten muss die Domäne verstanden und der gesamte Software-Stack entsprechend geschnitten sein. Damit die aufgeteilten Services performant ausgerollt werden können, braucht es zudem eine automatisierte Provisionierung des Infrastruktur-Stacks.

Ein agiler Software-Entwicklungsprozess fördert einen engeren Austausch zwischen allen Produktbeteiligten. Mit dem Fokus auf die Entwicklung eines Anwendungsfalls schaffen Fachverantwortliche, Produktverantwortliche und Entwickler ein gemeinsames Verständnis für die nächste zu entwickelnde Funktionalität mit dem höchsten Mehrwert für den Kunden. Dieser iterative Entwicklungszyklus bietet den Raum zur stetigen Ausrichtung der Produktvision und stellt sicher, dass die aktuell relevanten Bedürfnisse des Kunden befriedigt werden.

Einen Ansatz für den richtigen Schnitt bietet Eric Evans mit Domain Driven Design *[1]*. Darauf aufbauend existieren verschiedene Methodiken wie das Event-Storming, mit dem sich Domänengrenzen (Bounded Contexts) und zusammenhängende Domänen (Aggregates) eines Anwendungsfalls identifizieren lassen. Die Konzentration auf abgegrenzte Anwendungsfälle erleichtert das Verständnis sowie die Kommunikation bei der Lösungsfindung. Schließlich muss die Aufteilung in Services umgesetzt werden, um von unabhängigen Deployments und flexibler Skalierung zu profitieren. Der technologische Rahmen ist damit neuen Anforderungen ausgesetzt und muss neu gedacht werden.

Die Entwicklung von kleinen Services bedeutet, dass mehr Infrastruktur notwendig ist und diese schnell provisioniert werden muss. Auf der einen Seite kann man für diesen Zweck diverse Cloud-Dienste verwenden, auf der anderen Seite existieren die notwendigen Werkzeuge Open-Source-basiert und als On-Premise-Lösungen. Zum Beispiel kann mit Spring Boot *[2]* eine vollständige und lauf-

fähige Anwendung erstellt werden, die alle notwendigen Abhängigkeiten mit sich bringt.

Mit systemD *[3]* vereinfacht sich die Steuerung einer solchen Anwendung. Container wiederum ermöglichen eine schlanke Isolierung der Betriebssystem-Virtualisierung (etwa mit Docker *[4]*). Schließlich gibt es auch für die Orchestrierung der Container verschiedene Alternativen (wie Kubernetes *[5]* oder Docker Swarm *[6]*). Diese Werkzeuge folgen dem „Everything as Code"- Ansatz und bieten eine Beschreibungssprache zur Konfiguration der Laufzeitumgebung. Die Infrastruktur ist damit versionierbar und reproduzierbar. Es muss zusätzlich die Abwägung getroffen werden, ob die Anwendungskonfiguration mit dem Deployment ausgeliefert wird und damit jederzeit nachvollziehbar, aber starr ist, oder ob ein Konfigurationsserver als weitere Abhängigkeit eine zur Laufzeit flexible Konfiguration erlaubt.

Die Komplexität liegt schließlich im Aufbau der Orchestrierung, jedoch bietet eine solche Infrastruktur die Möglichkeit einer Continuous Deployment Pipeline; also einen automatisierten Prozess, um die Anwendung in Produktion zu nehmen und damit die Time-to-Market zu senken.

Vom Monolithen zu Microservices bei dm-drogerie markt

Um sowohl für die wachsenden fachlichen als auch die technischen Anforderungen vorbereitet zu sein, ist dm-drogerie markt mit der Unterstützung von synyx den Pfad dieser Transformation von einem monolithischen hin zu einem auf Microservices basierenden System gegangen.

Die steigende Zahl der Marketing-Kampagnen bei dm-drogerie markt führt zu einem stetig wachsenden Funktionsumfang im Backend des Coupon-Verwaltungssystems. Das alte System wurde direkt in der bestehenden Integrations-Plattform, basierend auf starren proprietären Komponenten, entwickelt und unterlag den klassischen, oben beschriebenen Strukturen. Das Resultat nach nur wenigen Jahren war ein Zustand, der Erweiterungen kaum zuließ und hohen Aufwand für die Wartung erforderte. Zu diesem Zeitpunkt wurde mit der Entwicklung des Online-Shops begonnen, wobei erkannt wurde, dass eine Anbindung der beiden Systeme nicht ohne Weiteres möglich war. Eine funktionierende Interaktion der Systeme setzte die Ablösung der bestehenden Coupon-Verwaltung durch eine Neuentwicklung voraus.

Der Weg zum Monolithen

Wenn ein Unternehmen einen Größenschwellwert überschreitet, wird versucht, Synergie-Effekte zu erzielen. Analog zum Vorgehen in der Software-Entwicklung werden horizontale Schichten eingeführt und beispielsweise technische Teams wie ein Betriebsteam bereitgestellt, das generische und umfassende Lösungen für die Entwicklerteams anbietet. Aufgrund der hohen Priorität des Online-Shops war das Angebot des Betriebsteams auf dessen technische Bedürfnisse ausgerichtet. Diese Rahmenbedingungen bieten viele Funktionen zum Projektstart, stellen aber gleichzeitig ein Korsett dar, das nur mit großem Kommunikations- und Arbeitsaufwand anpassbar ist.

Es wurde ein Software-Stack angeboten, der viele Funktionen umfasste, wie Active- und Standby-Software-Load-Balancer, Proxy inklusive Varnish und HTTP-Server sowie Web-Container *(siehe Abbildung 1)*. Die Komponenten liegen aufgrund von Ausfallsicherheit jeweils in zweifacher Dimensionierung vor. Außerdem wurden ein Deployment sowie ein Anwendungsmonitoring angeboten. Da zusätzlich zu dieser Ausgangssituation die funktionalen sowie nicht-funktionalen Anforderungen noch nicht feststanden, wurde ein iterativer Entwicklungsprozess gewählt und mit einem fachlich-strukturierten Monolithen angefangen. Darüber hinaus einigte sich das Team – zum einen aus Know-how-Gründen, zum anderen aufgrund der Funktionalität – auf das Spring-Framework und insbesondere Spring Boot.

Als zentrales Backend-System hat die Coupon-Verwaltung zur Aufgabe, viele Schnittstellen anzubieten, die von unterschiedlichen Systemen konsumiert werden. Die Konsumenten spiegelten unterschiedliche Anwendungsfälle wider, benötigten individuelle Logik und Perspektiven auf die Daten. Aus diesem Grund war es intuitiv, die Software anhand der Konsumenten aufzuteilen („Consumer Driven Slices").

Als „Maven Multi Module"-Projekt wurde jedes Modul als eigenständige und lauffähige Anwendung entwickelt. Erst beim Deployment entsteht aus den Modulen ein Monolith. Mit Disziplin und Tool-Unterstützung (wie SonarQube *[7]*) wurde sichergestellt, dass keine zyklischen Abhängigkeiten entstehen. Aufgrund der gerin-

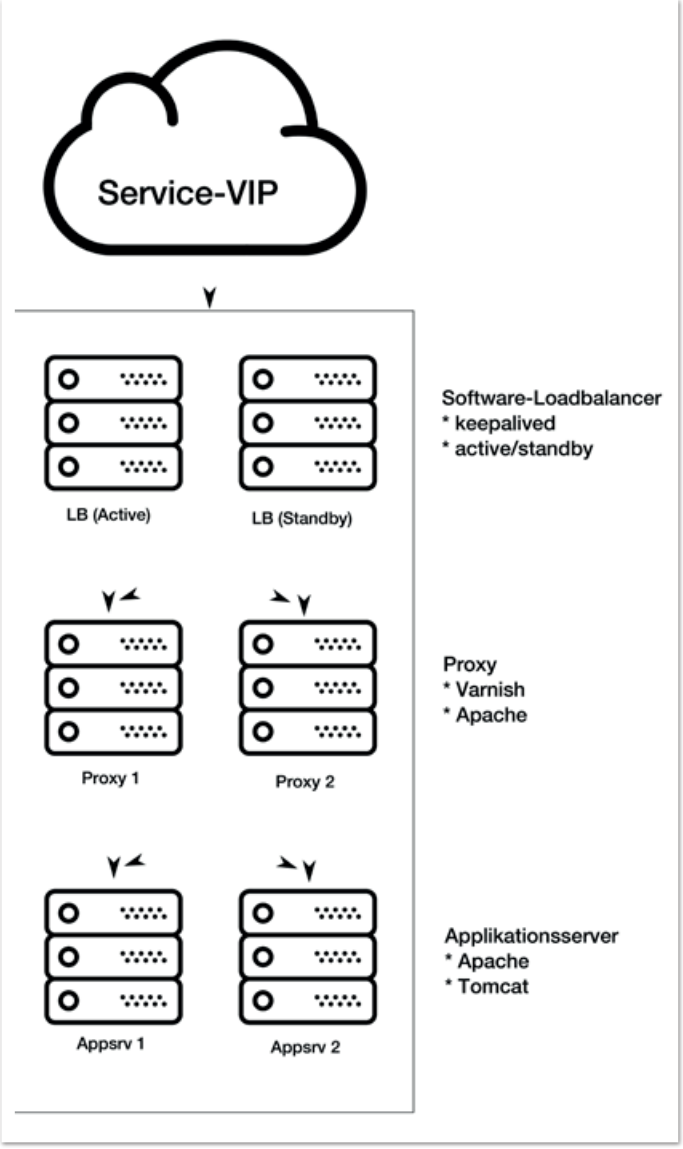


Abbildung 1: Ausgereifter und umfangreicher Software-Stack

gen Anforderungen an das Datenmodell sowie der zeitlich getrennten Schreibzugriffe wurde auf Datenbankebene integriert.

Microservice-Infrastruktur bei dm-drogerie markt

Zur gleichen Zeit wurde mit höchster Priorität der dm-Online-Shop entwickelt. Da das Bedürfnis der zuständigen Entwickler nach Rahmenbedingungen für Microservices zunahm, folgte ein Umdenken an mehreren Stellen. Das Betriebsteam ging mehr auf die Entwicklungsteams ein und reduzierte den Software-Stack für den Betrieb einer Anwendung. Er bestand lediglich aus einem HTTP-Server und einem Web-Container *(siehe Abbildung 2)*.

Für die ersten Microservices des Online-Shops wurde zunächst eine Basis-Infrastruktur bereitgestellt *(siehe Abbildung 3)*. Es kamen verschiedene Komponenten aus dem Netflix-OSS-Stack *[8]* zum Einsatz, wie zum Beispiel eine Service-Discovery (Eureka *[9]*), die eine einfache Skalierung ermöglicht. Zusätzlich wurden dedizierte Load Balancer durch Client Side Load Balancing (Ribbon *[10]*) ersetzt.

Alle Microservices stehen in einer DMZ und Zugriffe werden durch einen Proxy authentifiziert. Die Microservices selber übernehmen letztlich die Autorisierung. Der Proxy dient auch als API-Gateway (Zuul *[11]*) und vereinheitlicht als Reverse-Proxy die Schnittstellen zu den Microservices.

Mit der „dm glückskind"-App, die Coupons abfragen und aktivieren kann, entstand die Anforderung einer Funktion, die einen zentralen Sicherheitsmechanismus benötigte, um Kunden zu identifizieren. Da dieser Mechanismus in der Microservice-Infrastruktur schon bereitstand, war das der richtige Zeitpunkt, mit dem ersten Microservice auf die neue Infrastruktur zu migrieren. Das relevante Modul wurde aus dem Monolithen gelöst und als Microservice deployt.

BDD, Logging, Metrics und Tracing – Voraussetzung für Microservices

Das Ersetzen des Backend für die Coupon-Verwaltung birgt nicht nur technische Herausforderungen. Um den korrekten Betrieb zu gewährleisten, existierten in der Vergangenheit viele manuelle Tests. Es gibt zwei Gründe, aus denen der Fachbereich zu zeitintensiven, manuellen Tests greift, obwohl sie für das Ausrollen einer Funktion hinderlich sind. Entweder es ist technisch nicht möglich, einen automatisierten Test zu implementieren, weil Teilprozesse keine Schnittstelle haben, oder es fehlt an Vertrauen in das Entwicklungsteam. Auf beide Umstände muss das Entwicklungsteam von Projektbeginn an eingehen.

Wo es keine Schnittstellen gab, wurde Zeit in Kommunikation und Entwicklung investiert, um diese herzustellen. Außerdem wurde mit Development, Stage, Performance, Release und Production ein Fünf-Stages-Konzept etabliert, um dem Ziel, Continuous Deployment, näherzukommen.

Das Vertrauen wurde durch eine hohe Priorität auf vollständige Akzeptanz-Tests von Beginn an aufgebaut. Als Teil der Definition-of-Ready und damit noch vor dem Refinement werden in einem sogenannten „Drei-Amigo-Meeting" (Fachverantwortlicher, Produktverantwortlicher und Entwickler) alle Tests für die in der nächsten Story zu implementierende Funktion definiert. Diese Akzeptanz-Tests werden in der Beschreibungssprache gherkin *[12]* (Given - When - Then) für alle Projektbeteiligten verständlich erstellt und mit dem Werkzeug Cucumber *[13]* implementiert. Aus einem dedizierten Projekt heraus werden die unterschiedlichen Stages als Black-Box getestet. Somit konnten die Tests auch nach der Extraktion des Moduls aus dem Monolithen sicherstellen, dass die Funktionalität als Microservice weiterhin gewährleistet ist.

Um einer neuen Infrastruktur vertrauen zu können, muss man sehen und verstehen, wie sich die Anwendung im Live-Betrieb verhält. Dafür gibt es drei Mechanismen:

Mit Logging kann man ereignisbasierte Informationen sammeln. Diese Informationen werden mit den Werkzeugen ElasticSearch *[14]*, Logstash *[15]* und Kibana *[16]* zentral für alle Teams durchsuchbar bereitgestellt.

Metrics sind Laufzeit-Aggregate zur Wahrnehmung der Performance. Die grafische Oberfläche Grafana *[17]* stellt die Performance-Informationen überwiegend aus InfluxDB-Datenbanken *[18]* dar. Mittels Telegraf senden die Microservices dafür täglich etwa zehn Gigabyte an Daten.

Tracing sind auf Anfragen basierende Daten zur Korrelation von Laufzeit-Informationen. Eine Tracing-Infrastruktur mit OpenZipkin *[19]* ist gerade im Aufbau. Mit Spring-Cloud-Sleuth *[20]* kann anhand von Tracing-Ids und Span-Ids schon jetzt in den Logs-Anfragen korreliert werden.

Viele Hürden passiert, aber noch nicht am Ziel

Je nach Kultur, Code-Basis und Rahmenbedingungen kann eine solche Transformation Zeit brauchen. Es sind Experimente notwendig und ein guter Umgang mit Fehlentscheidungen. Als gute Entscheidung hat sich herausgestellt, den Abstand zwischen Betriebs- und Entwicklerteam zu minimieren und damit die DevOps-Kultur zu leben. Dediziert verstärken betriebsnahe Mitarbeiter die Entwicklungsteams für mehrere Tage die Woche und begleiten den Entwicklungsprozess.

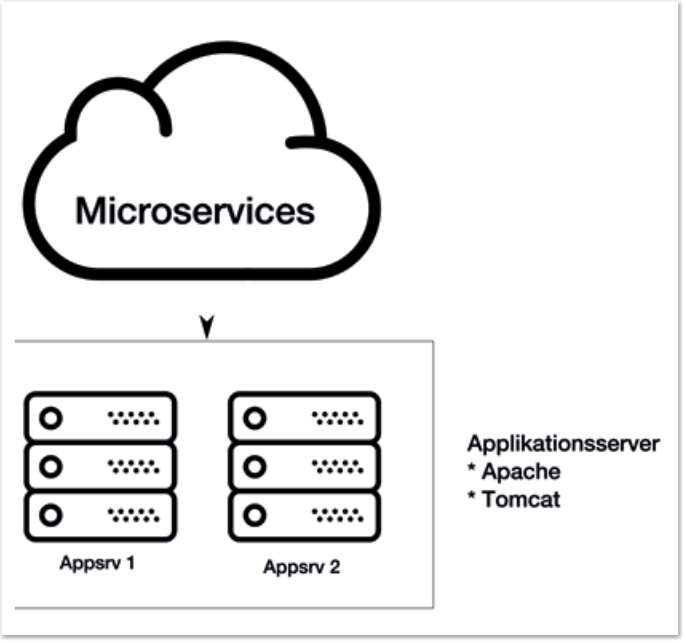


Abbildung 2: Schlanker Software-Stack, bestehend aus HTTP-Server und Web-Container

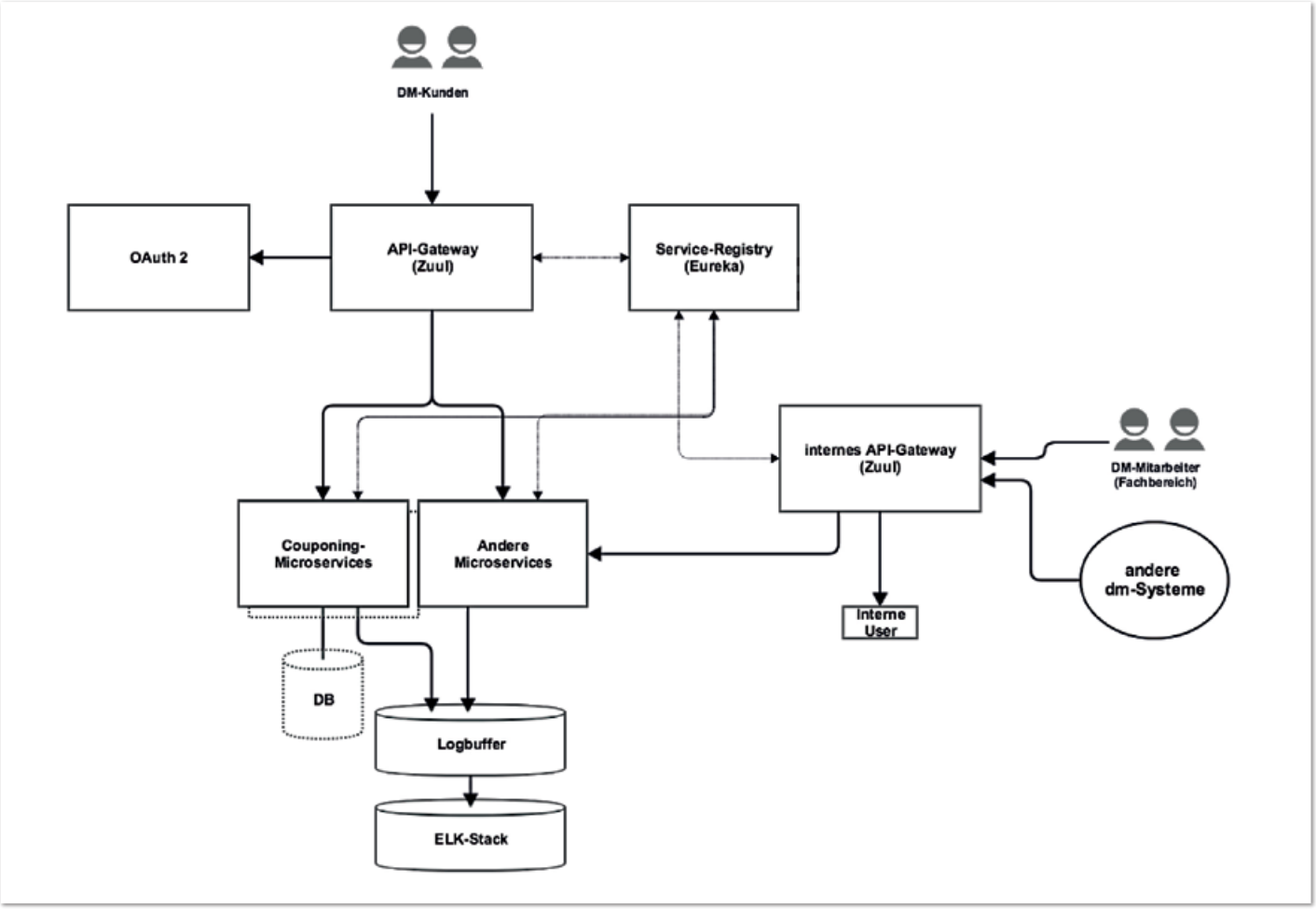


Abbildung 3: Schematischer Überblick über die Microservice-Infrastruktur

Es hat sich als keine so gute Entscheidung erwiesen, zentrale Komponenten zwischen Teams zu teilen, die bei Änderungen enormen Kommunikationsaufwand bedeuten. Die laufende Transformation zeigt allerdings jetzt schon deutliche Vorteile:

- Services können unabhängig voneinander deployt und skaliert werden
- Bessere Fokussierung auf Domäne und einfache Kommunikation
- Keine Seiteneffekte auf fremde Domänen durch Modularisierung
- Schnelle Releases durch Continuous Deployment Pipeline

Auch für dm-drogerie markt ist der Weg noch nicht zu Ende. Die automatisierte Provisionierung der Microservices – und damit eine Optimierung des Deployments mit Containern – sowie die Orchestrierung sind die nächsten spannenden Themen auf dem Pfad zu einer hoch flexiblen und automatisierten Infrastruktur.

Weiterführende Informationen

[1] Eric Evans, Domain-Driven-Design, Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 978-0-321-12521-7
[2] *https://projects.spring.io/spring-boot*
[3] *https://www.freedesktop.org/wiki/Software/systemd*
[4] *https://www.docker.com*
[5] *https://kubernetes.io*
[6] *https://github.com/docker/swarm*
[7] *https://www.sonarqube.org*
[8] *https://netflix.github.io*
[9] *https://github.com/Netflix/eureka*
[10] *https://github.com/Netflix/ribbon*
[11] *https://github.com/Netflix/zuul*
[12] *https://github.com/cucumber/cucumber/wiki/Gherkin*
[13] *https://cucumber.io*
[14] *https://www.elastic.co/de/products/elasticsearch*
[15] *https://www.elastic.co/de/products/logstash*
[16] *https://www.elastic.co/de/products/kibana*
[17] *https://grafana.com*
[18] *https://www.influxdata.com*
[19] *http://zipkin.io*
[20] *https://cloud.spring.io/spring-cloud-sleuth*

Andreas Weigel
weigel@synyx.de

Andreas Weigel ist Software-Entwickler und Berater bei synyx GmbH & Co. KG und unterstützt seit mehreren Jahren verschiedene Kunden bei der Entwicklung von Software-Lösungen im agilen Umfeld. Dabei beschäftigt er sich primär mit Architektur-Entscheidungen, der vollumfänglichen Automatisierung des Entwicklungsprozesses sowie der Umsetzung von individuellen Lösungen.

Besserer Java-Code – außerhalb der Automatismen

Wolfgang Nast, MT AG

Mit jeder neuen Java-Version kamen auch Verbesserungen in der Art und Weise, wie Code besser geschrieben werden kann. Wenn man nützliche Beispiele gesehen hat, unterstützt einen dabei die Entwicklungsumgebung.

Die wichtigste Unterstützung beim Codieren sind Entwicklungsumgebungen (IDE) wie Eclipse, IntelliJ und NetBeans. Dabei sollte in einem Projekt eine einheitliche Vorgabe beim Formatieren und Überprüfen eingestellt sein.

Verbreitete Tools zum Finden und Vermeiden von unschönem Code sind Checkstyle, FindBugs und SonarQube. Diese können sowohl im automatischen Build mit Maven oder Gradle eingebunden sein als auch in die IDEs selbst, damit man schon beim Codieren unterstützt wird. Nun sind die üblichen Automatismen ausreichend beschrieben und wir kommen zum Teil der Verbesserungen.

Es geht um Neuerungen von Java 5 bis 9, die meist zusammen vorgestellt werden. Mit dabei sind Themen wie Exception Handling, Generics, Collections mit Sortierungen, Streams mit Primitiven und optionale Werte (null). Im Artikel sind die Verbesserungen in zwei bis drei Blöcken beschrieben. Dabei ist im ersten Block der Code, der bisher üblich war, danach kommt der verbesserte Code. Im optionalen dritten Block ist der praktische Hilfscode beschrieben.

Beginnen wir mit den Ressourcen (wie „IOStreams"), die nach der Verwendung wieder geschlossen werden müssen. Bis Java 7 war folgender Code üblich *(siehe Listing 1)*. Hier lässt sich mit „try with resources" viel Code sparen *(siehe Listing 2)*.

Dabei entfällt der rot markierte „finally"-Block und die Fehlerbehandlung geschieht an einer Stelle. Die Variable „inStream" ist nur im Block gültig. Dies kann ab Java 9 wieder erweitert werden, wenn diese implizit final ist und vor dem Block angelegt wird. Bei dem Code in *Listing 3* möchte man gerne von der vorgestellten Ressourcen-Verwendung profitieren. Auf den „finally"-Block kann verzichtet werden, wenn die Lock-Verwendung zu einer Ressource wird *(siehe Listing 4)*. Hier steht ein „try"-Block ohne „catch" und ohne „finally", aber mit Ressource. Die Hilfsklasse „LockBlock" ist in *Listing 5* beschrieben.

Der blau markierte Code ist der Teil, der von der Ressource ausgeführt wird; der grüne Code ist von der Ressource vorgegeben. Der „close"-Teil wird beim Verlassen des Blocks aufgerufen. Zu beachten ist, dass der „throws Exception"-Teil, der vom Interface „Closeable" vorgegeben wird, weglassen wurde.

Im Konstruktor ist der Teil des Holens der Ressource übernommen. Die Hilfsklasse hat zwei Konstruktoren; der zweite dient dazu, den Lock mit der Methode „lockInterruptibly" zu holen, der auch eine „InterruptedException" werfen kann. Der zweite Parameter „inter" wird nicht ausgewertet, da er nur dazu dient, eine andere Signatur zu haben, weil der „throws"-Teil nicht zur Unterscheidung dient, obwohl dieser beim Aufruf wichtig ist. Es folgt das Beispiel mit der Verwendung der „InterruptedException" *(siehe Listing 6)*. Die ursprüngliche Collection-Klasse „Vector" wurde wie in *Listing 7* verwendet.

Hier ist in Blau der Teil mit der Initialisierung und in Orange der Teil mit dem Ermitteln der grünen Werte dargestellt. Es lässt sich übersichtlicher codieren *(siehe Listing 8)*. Die Initialisierung in Blau kommt mit Java 9. Der rote Code sollte anstelle des Fragezeichens den verwendeten Typ angeben. Die ursprüngliche Hashtable-Klasse zum Mappen von Werten sah wie in *Listing 9* aus. Auch hier ist die Initialisierung blau und der Teil zum Ermitteln der grünen Werte orange. Auch die Map lässt sich übersichtlicher codieren *(siehe Listing 10)*.

```
InputStream inStream = null;
try {
     inStream = new FileInputStream (Datei);
//Logic
  } catch (IOException e) {
     //Fehlerbehandlung 1
} finally {
  try {
     if (inStream != null) {
        instream.close();
     }
  }catch (IOException e) {
     //Fehlerbehandlung 2
  }
}
```

Listing 1

```
  try (InputStream inStream = new
FileInputstream(Datei)) {
     //Logic
  } catch (IOException e){
     //Fehlerbehandlung 1 und 2
  }
```

Listing 2

```
  Lock lock = new ReentrantLock();
  try {
     lock.lock();
     //Logic;
  } finally{
     lock.unlock();
  }
```

Listing 3

```
  Lock lock = new ReentrantLock();
  try (LockRes bl = new LockBlock(lock)){
     //Logik;
     return Ergebnis;
  }
```

Listing 4

```
public class LockRes() implements Closeable{
  private Lock lock;
  public LockRes(Lock lock) {
  this.lock = lock;
     lock.lock();
  }
  public LockRes(Lock lock, boolean inter) throws
           InterruptedException {
     this.lock = lock;
     lock.lockInterruptibly();
  }
  @Override
  public void close() {
     lock.unlock();
  }
 }
```

Listing 5

```
  Lock lock = new ReentrantLock();
  try (LockRes bl = new LockBlock(lock, true)){
     //Logic
  } catch(InterruptedException e) {
     //Unterbrochen
  }
```

Listing 6

Bei der Initialisierung der Werte ist zu beachten, dass bis zu zehn Werte-Paare nacheinander angegeben werden können. Bei elf oder mehr Werten ist die Methode „Map.ofEntries(Map.Entry<K,V>...entries)" zu verwenden. Bei Streams von „int"-Werten ist recht häufig der Code aus *Listing 11* zu finden.

Im blauen Teil wird der Objekt-Datentyp von „int" verwendet, was zu Autoboxing und Autounboxing führt. Der rote Teil wird dabei gerne weggelassen, weil es auch ohne die „Null"-Tests meistens gut geht. Die Methode „mapToInt" mit dem Identity-Aufruf „x -> x" funktioniert dank Autounboxing. Besser ist hier *Listing 12*. Doch es geht auch ohne Autoboxing *(siehe Listing 13)*.

Es gibt nur für die primitiven Datentypen „int", „long" und „double" passende Streams. Bei „float" ist auf „double" zu erweitern, „byte", „char" sowie „short" auf „int". Beim Verwenden von optionalen Werten wird meist „null" als „nicht vorhanden" gekennzeichnet. Damit sind viele „null"-Tests verbunden, wie *Listing 14* zeigt. Hier eignet sich die Klasse „Optional" besser *(siehe Listing 15)*.

Dabei wird durch „map" zweimal der „null"-Test ausgeführt und ein neues „Optional" angelegt. Mit „ifPresent" wird nur der „Logic"-Block aufgerufen, wenn „Optional" gesetzt ist (nicht „null"). Optional gibt es auch die primitiven Datentypen „int", „long" und „double". Sie eignen sich besser, um optionale Werte darzustellen, als die Objekt-Entsprechungen. Wie die Objekt-Entsprechungen ist „Optional" ein konstanter Wert; bei dessen Änderungen ist eine neue Instanz anzulegen. In Java gibt es keinen „call by reference", deshalb wird für primitive Datentypen und konstante Werte gerne der Ansatz aus *Listing 16* verwendet.

Rot markiert sind hier die Änderungen, die durch die Verwendung des Arrays mit einem Element entstanden sind. Dies ist bei interner Nutzung ausreichend. Bei der Verwendung als API muss geprüft werden, ob das Array nicht „null" ist und es nur die Größe „eins" hat. Bei der Verwendung einer lokalen Variablen in der „For"-Schleife ist es direkt möglich *(siehe Listing 17)*. *Listing 18* zeigt, wie man die „For"-Schleife auf eine Lambda-Expression in der Methode „forEach" umstellt.

Da alle lokalen Variablen in der Lambda-Expression „final" sein müssen, ist die Änderung in Rot notwendig. Es kann zwar das „final" weggelassen werden, der Compiler setzt es jedoch implizit. Meistens lassen sich die Lambda-Blöcke soweit optimieren, dass keine

```
Vector werte = new Vector();
  werte.addElement(wert1);
  werte.addElement(wert2);
for(int i = 0; i <werte.size(); ++i) {
  Object wert = werte.get(i);
  //Logic
}
```

Listing 7

```
Collection<?> werte = List.of(wert1, wert2);
werte.forEach(wert -> {
  //Logic
});
```

Listing 8

```
Hashtable map = new Hashtable();
map.put(key1, wert1);
map.put(key2, wert2);
for(Enumeration keyEnum = map.keys(); keyEnum.hasMo-
reElements(); ) {
  Object key = keyEnum.nextElement();
  Object wert = map.get(key);
  //Logik
}
```

Listing 9

```
Map<?, ?> map = Map.of(key1, wert1,
  key2, wert2);
map.forEach((key, wert) -> {
  //Logik
});
```

Listing 10

```
Collection<Integer> zahlen = List.of(1,3,6);
zahlen.stream().filter(x -> x != null)
  .mapToInt(x -> x)
  .forEach(System.out::println);
```

Listing 11

```
Collection<Integer> zahlen = List.of(1,3,6);
zahlen.stream().filter(Objects::nonNull)
  .mapToInt(Integer::intValue)
  .forEach(System.out::println);
```

Listing 12

```
IntStream zahlenSt = IntStream.of(1,3,6);
zahlenSt.forEach(System.out::println);
```

Listing 13

```
if (wert != null) {
  if (wert.getText() != null) {
     if (wert.getText().getTeile() != null) {
        //Logic
     }
  }
}
```

Listing 14

```
Optional.ofNullable(wert)
  .map(Wert::getText)
  .map(Text::getTeile)
  .ifPresent(x -> {
     //Logic
});
```

Listing 15

```
public void aufruf(int[] ref){
  ref[0]++;
}
int zahlRef[] = {0};
aufruf(zahlRef);
```

Listing 16

```
boolean gefunden = false;
for(Wert wert : wertList) {
  //Logic
  gefunden = true;
}
```

Listing 17

```
final boolean gefunden[] = {false};
wertList.forEach(wert -> {
  //Logic
  gefunden[0] = true;
});
```

Listing 18

```
public static Comparator<WerteKlasse> comp =
  Comparator.comparingLong(WerteKlasse::getId)
  .thenComparing(WerteKlasse::getText);
```

Listing 19

```
public static Comparator<WerteKlasse> compNull =
  Comparator.nullsFirst(comp);
```

Listing 20

```
@Override
public int compareTo(WerteKlasse o) {
  return comp.compare(this, o);
}
```

Listing 21

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
     return true;
  }
  if (obj != null && !getClass().equals(obj.get-
Class())) {
     return false;
  }
  return compareTo((WerteKlasse)obj) == 0;
}
```

Listing 22

```
@Override
public int hashCode() {
  return Objects.hash(id, text);
}
```

Listing 23

```
public void aufruf(Wert wert) {
  if (wert == null) {
     throw new NullPointerException(msg);
  }
  //Logic
}
```

Listing 24

lokalen Variablen verwendet werden müssen. Bei der Verwendung von eigenen Klassen in Collections kommt es vor, dass diese auch sortiert werden sollen; dafür gibt es den „Comparator". Dieser lässt sich mittlerweile wie in *Listing 19* realisieren.

Hier wird in der Reihenfolge der gewünschten Sortierung der Getter der Werte angegeben. Sollen auch „null"-Werte mit sortiert werden, lässt sich dieser zweite Comparator wie in *Listing 20* angegeben. Alternativ geht auch „nullsLast". Für die natürliche Sortierung der Klasse ist das Interface „Compareable<Klasse>" zu implementieren. Dies kann recht leicht mit den Comparator geschehen *(siehe Listing 21)*.

Leider wird häufig vergessen, dass auch „equal" und „hasCode" zu überschreiben sind, um eine natürliche Sortierung richtig umzusetzen. *Listing 22* zeigt den Code für „equals".

Der rote Code ist notwendig, um auf den Parameter „Object" richtig zu reagieren. Wichtig ist, dass die Klasse gleich sein muss, damit „a.equals(b) == b.equals(a)" für alle „a" und „b" gilt. Der Test „instanceof" ist nicht ausreichend. *Listing 23* zeigt den Code für „hasCode".

Die Methode „Objects.hash(Object ... objs)" beachtet auch „null"-Werte beim Berechnen des Hash-Werts. Bei Parametern in Schnitt-

```
public void aufruf(Wert wert) {
  Objects.requireNonNull(wert, msg);
     //Logic
}
```

Listing 25

```
public Object update(Updateable daten){
  try {
     return em.update(daten);
  }
  return null;
}
daten1 = (Daten1)update(daten1);
daten2 = (Daten2)update(daten2);
```

Listing 26

```
public <T extends Updateable>
  T update(T daten){
  try {
     return em.update(daten);
  }
  return null;
}
daten1 = update(daten1);
daten2 = update(daten2);
```

Listing 27

```
@SuppressWarnings("unchecked")
public <T> T aufruf(int pos){
  return (T)daten[pos];
}
Daten1 daten1 = aufruf(1);
Daten2 daten2 = aufruf(2);
```

Listing 28

stellen sind meistens „null"-Tests notwendig. *Listing 24* zeigt die übliche Umsetzung. Dabei ist der zu testende Wert grün markiert. Hier bietet die Klasse „Objects" eine kürzere Schreibweise *(siehe Listing 25)*.

Der „Null"-Test und das „Throw"-Statement sind in der Methode „requireNonNull" umgesetzt, es muss noch die Message übergeben werden, wenn der Wert „null" ist. Beim Aufruf von Speicher-Methoden wird meist das gespeicherte Objekt wieder zurückgegeben, was zu einem Cast bei der Verwendung führt *(siehe Listing 26)*. Mit einem generischen Parameter kann auf den roten Cast verzichtet werden, damit ist der Code wie in *Listing 27*.

```
public class Werte {
  private final long id;
  private final String name;
  private final String vorname;
  public Werte(long id, String name, String vorname) {
     this.id = id;
     this.name = name;
     this.vorname = vorname;
  }
  public Werte(long id, String name) {
     this(id, name, null);
  }
  public long getId() {
     return id;
  }
  public String getName() {
     return name;
  }
  public String getVorname() {
     return vorname;
  }
}
```

Listing 29

```
public class WerteBuilder {
  private long id = 0;
  private String name;
  private String vorname;
  public WerteBuilder() {
  }
  public WerteBuilder withId(long id) {
     this.id = id;
     return this;
  }
  public WerteBuilder withName(String name) {
     this.name = name;
     return this;
  }
  public WerteBuilder withVorname(String vorname) {
     this.vorname = vorname;
     return this;
  }
  public Werte build() {
     return new Werte(id, name, vorname);
  }
}
```

Listing 30

```
Wert wert1 = new Wert(12, "Müller", "Karl");
Wert wert1 = new WerteBuilder()
  .withId(12)
  .withVorname("Karl")
  .withName("Müller").build();
```

Listing 31

Hier bestimmt der Typ des Parameters den Typ des Rückgabewerts. Der „Updateable"-Typ, der für den Aufruf notwendig ist, steht jetzt im generischen Parameter nach dem „Extends". Es ist auch möglich, den Cast des Rückgabewerts in die Methode zu verlagern, damit er nicht nach dem Aufruf notwendig wird *(siehe Listing 28)*. Hier sollte die Warnung des Compilers an der orangenen Stelle mit der Annotation „SuppressWarnings()" auch in Orange unterdrückt werden.

Gerne werden konstante Container-Klassen wie in *Listing 29* codiert. Dabei sind die angebotenen Konstruktoren immer unpraktischer, je mehr Werte übergeben werden müssen und wenn mehrere Werte optional sind, wie im Beispiel mit Vorname. Hier eignen sich „Builder"; diese kommen nicht durch Neuerungen in Java, sondern aus der Verbesserung von APIs *(siehe Listing 30)*.

Der Builder ist keine Vereinfachung des zu implementierenden Codes, da er komplett zusätzlich zur Klasse kommt. Er ist sogar umfangreicher als die eigentliche Klasse, da alle „Setter(with)" sich „Selbst(this)" zurückgeben, und er eine Build-Methode zum eigentlichen Anlegen der Instanz in Grün hat. In Ocker sind die Erweiterung des Builders und die Konstruktoren angegeben. Erst beim Aufruf zeigt der Builder seinen Vorteil *(siehe Listing 31)*.

Hier ist beim Builder über die blauen „with"-Methoden immer klar, welcher Wert gesetzt wird, auch wenn sich die Anzahl der Parameter mal erweitern sollte. Der Aufrufer bestimmt die Reihenfolge. Beim Konstruktor muss man bei vielen Parametern vom gleichen Typ aufpassen (Name oder Vorname zuerst).

Fazit

Es sind viele Verbesserungen mit den neuen Versionen von Java gekommen, die auch einfacheren Code ermöglichen. Streams, Optional und Builder werden gerne im Code-Chaining verwendet. Das heißt, das Ergebnis wird für den nächsten Aufruf verwendet und nicht mehr in einer Variablen zwischengespeichert. Man hat damit eine Kette von Aufrufen.

Wolfgang Nast
wolfgang.nast@mt-ag.com

Wolfgang Nast (Dipl. Ing.) ist Senior Berater bei der MT AG und seit dem Jahr 1998 mit Java SE sowie seit dem Jahr 2006 mit Java EE in den Bereichen „Schnittstellen" und „Architektur" beratend und umsetzend tätig.

Is Agile Eating up the World?

Carsten Wiesbaum, esentri AG

Wir versuchen über Jahrzehnte, etablierte Strukturen aufzubrechen, die uns in der heutigen beruflichen und sozialen Welt schaden. Agile Software-Entwicklung, Microservices und DevOps predigen seit Jahren, dass ein kultureller Wandel nötig ist. Mittlerweile greift dieser Gedanke auf weitere Bereiche unseres Lebens über. Dieser Artikel zeigt einige Entwicklungen der letzten Jahre auf und wägt ab, ob Agilität ein valider Ansatz für die Lösung der daraus entstehenden Herausforderungen ist.

In der IT ist agile Entwicklung bereits ein alter Hut: gestartet im Jahr 2001 mit der Definition des agilen Manifests und seiner zwölf Prinzipien. Nach dem steilen Aufstieg bis zum Gipfel der überzogenen Erwartungen befinden wir uns aktuell irgendwo im Bereich des Tals der Enttäuschung und des Pfads der Erleuchtung. Auf der einen Seite sagen Experten, dass Agilität bereits längst in Unternehmen angekommen ist. Auf der anderen Seite gibt es noch immer viele Firmen, die sich mit der Einführung einer agileren und iterativen Vorgehensweise beschäftigen und teilweise sehr schwertun.

Viele derer, die angeblich die Transition zur agilen Vorgehensweise innerhalb kürzester Zeit vollzogen haben wollen, sind nach Ansicht des Autors alles andere als agil und haben die Prinzipien nicht wirklich verstanden. Eine weitere Beobachtung, die man aktuell oft machen kann, ist, dass Personen aus IT-fernen Disziplinen immer häufiger über „agil" sprechen. So kann sich der Autor unter anderem mit seiner Frau, die eine klassische BWL-Karriere verfolgt, über das Thema unterhalten.

Der unaufhaltsame Wandel

Zunächst einmal geht der Autor auf die aktuelle Situation ein. In den letzten Jahren hat er drei grundlegende Beobachtungen gemacht:

- Fast alle Produkte sind Software oder basieren zu einem großen Teil auf Software
- Kunden-Erwartungen sind stark gestiegen, die Toleranz aber gleichzeitig gesunken
- Der Optimierungs- und Produktivitätssteigerungs-Wahn in der Arbeitswelt schränkt Mitarbeiter ein und lässt sie verdummen

Die Definition eines Produkts hat sich in den letzten Jahrzehnten stark verändert. Früher war ein Produkt, das in den Verkauf ging, primär ein abgeschlossenes System. Es wurde geplant, produziert und für einen kalkulierten Preis an den Endkunden ausgeliefert. Erweiterungen und neue Funktionen hinzuzufügen, war kein primäres

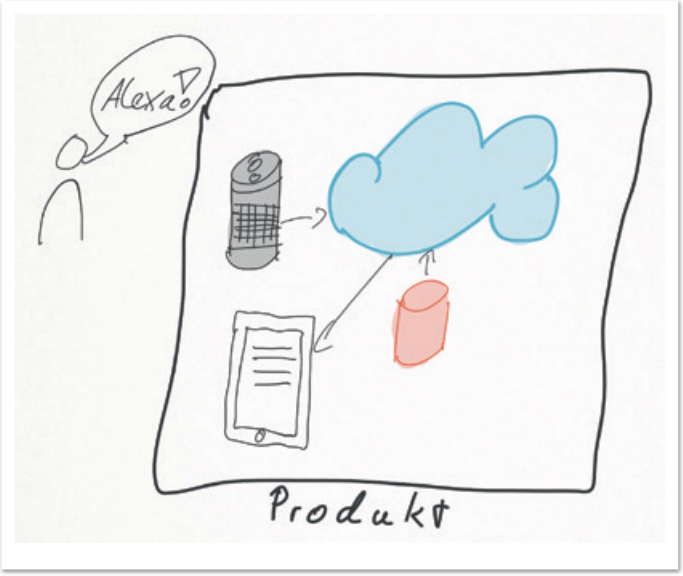


Abbildung 1: Produkte im Zeitalter der Digitalisierung

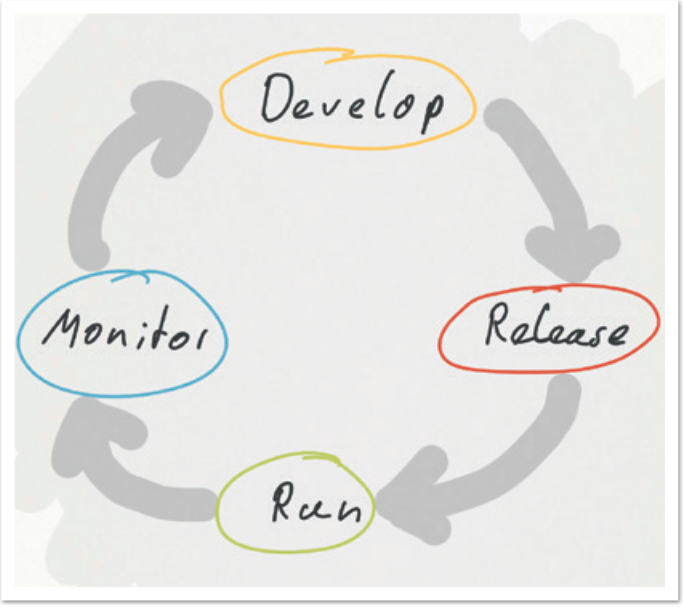


Abbildung 2: Produkte im ständigen Wandel

Entwicklungsziel; die nächste Version des Produkts lieferte diese nach, der Kunde musste bei Bedarf die neue Version zum vollen Preis kaufen.

Produkte im Zeitalter der Digitalisierung sind jedoch in der Regel Teil des Internets der Dinge und damit keine abgeschlossenen Systeme mehr *(siehe Abbildung 1)*. Neben der eigentlichen Funktion gibt es zusätzliche Services, die genutzt werden können oder sogar müssen. Für die Bereitstellung dieser Services bezahlt der Kunde oft eine zu-

Abbildung 3: Realität der Arbeitswelt

sätzliche Gebühr. Der Vorteil für den Kunden ist, dass die Produkte durch die Vernetzung stetig verbessert, Fehler korrigiert und neue Funktionen hinzugefügt werden können. Das „Ding", das man kauft, ist damit nur noch als Benutzerschnittstelle zu sehen, das Produkt als solches ist die Sammlung dieser Schnittstelle und aller damit verbundenen Services.

Neben der Vernetzung von Dingen werden auch immer mehr Produkte digital zur Verfügung gestellt, etwa Fahr- oder Bonuskarten. Der international renommierte Unternehmensberater und Buchautor Karl-Heinz Land spricht hierbei von der Dematerialisierung. Zudem gibt es auch komplett neue, servicebasierte Geschäftsmodelle. Plattform-Anbieter wie Uber oder Airbnb besitzen häufig kein eigenes physisches Produkt, sie bieten über ihre Software-Plattform Services an.

Im Kern kann man sagen, dass jedes Produkt mittlerweile einen hohen Software-Anteil hat und die Vernetzung immer weiter voranschreitet. Die Vorteile für den Kunden sind dabei stetige die Verbesserung und Erweiterung der Services. Das Unternehmen seinerseits generiert durch neue Servicegeschäftsmodelle einen konstanten Geldfluss und kann viel mehr Daten über seine Kunden zur Weiterentwicklung seiner Produkte sammeln.

Mit der Veränderung der Produkte weg vom einfachen, isolierten hin zum vernetzten, intelligenten und sich stetig verändernden Ding haben sich auch die Erwartungen der Kunden und damit die Art und Weise der Produkt-Entwicklung geändert. Der Kunde erwartet von modernen Produkten, dass sie verständlich sind und seinen Erwartungen entsprechen. Geschult durch das Smartphone ist er es gewohnt, einige Apps auszuprobieren und die für ihn passendste zu nutzen. Gefällt diese irgendwann nicht mehr, wird einfach gewechselt.

Regelmäßige Updates bieten in kurzen Intervallen Verbesserungen und neue Funktionen. Dabei verändern sich die Trends und Technologien so rasant, dass man nie sicher sein kann, wie lange ein Produkt relevant für den Markt ist. Dabei stehen Hersteller stets unter Druck, bei jedem neuen Trend mit dabei zu sein und ihre Produkte stetig weiterzuentwickeln. Selbst in Branchen wie der Leuchtmittel-Herstellung sind die Lebens- und Entwicklungszyklen von fünf bis fünfundzwanzig Jahren und wenigen Produkten auf ein großes Portfolio mit drei bis sechs Monatszyklen gestaucht worden, Smart-Home und der LED sei Dank *(siehe Abbildung 2)*. Karl-Heinz Land beschreibt dies in seinem Buch als digitalen Darwinismus, bei dem der Langsamere, hinsichtlich Geschwindigkeit der Releasezyklen und Adaption von neuen Trends, auf lange Sicht auf der Strecke bleibt und ausstirbt.

Bei diesen Rahmenbedingungen ist schnell klar, dass ein klassisches Projekt-Management mit detaillierter Design- und Planungsphase vorab sowie striktem Controlling nach dem Wasserfall-Modell in so kurzen Entwicklungszyklen nur schwer umzusetzen ist. Generell müssen die Trends und geforderten Funktionen des Marktes durch ständigen Kontakt mit dem Endkunden und die durch vernetzte Produkte gesammelten Daten frühzeitig erkannt und direkt mit der Umsetzung begonnen werden. In diese Kerbe schlug auch LinkedIn-Gründer Reid Hoffman mit seiner Aussage „If you are not embarrassed by the first version or your product, you've launched too late". Wenn man zu lange mit der Umsetzung einer neuen Funktion wartet und zu perfekt sein möchte, wird man vom Markt überrollt und verliert das Rennen ums Überleben.

Wer an klassischen Entwicklungs- und Produktzyklen festhält, wird es durch den existierenden Wandel schwierig haben, in der heutigen Produktwelt zu bestehen. Damit geht auch einher, dass sich die klassische Arbeitswelt zunehmend wandelt. Die in den letzten Jahrzehnten gelebte Arbeitsweise strebte oft in Richtung gesteigerter Produktivität und Optimierung. Die Ursprünge dieses Wahns liegen im Zeitalter der Industrialisierung und erfüllten für Fließbandarbeiten sicherlich ihren Zweck.

Es wird schon lange nicht nur im Bereich der Produktion, sondern auch in anderen Bereichen wie der Verwaltung, dem Personal-

Abbildung 4: „Steht aber genauso im Buch!"

management oder im Verkauf optimiert, durch Software unterstützt oder automatisiert. Wer hat sich nicht schon einmal bei einem Termin bei einer Bank oder Versicherung gefragt, wieso man sich die Mühe macht, einen Berater aufzusuchen, wenn er auch nichts anderes macht, als die Daten abzufragen, in eine Maske einzugeben und anschließend die gleichen Angebote vorzuschlagen, die auf Vergleichs- und Webauftritten einsehbar sind? Dem Mitarbeiter wird damit jegliche Möglichkeit, Entscheidungen zu treffen und kreative Lösungen zu finden, genommen. Als Resultat schalten viele Mitarbeiter während der Arbeit ab. Im Grunde wird man für seine Anwesenheit bezahlt und ist nicht mehr als die Arbeitssteuerung und das User Interface der Software, die die eigentliche Arbeit für den Endkunden macht. Diese Art von Jobs ist so lange sicher, bis entschieden wird, auch diesen Bereich zu digitalisieren und zu automatisieren oder in Länder zu verlagern, in denen die stupide ausführende Arbeitskraft viel günstiger zu haben ist *(siehe Abbildung 3)*. Aber auch in eigentlich kreativen Bereichen wie der Software-Entwicklung wird einem häufig die Möglichkeit, kreative Lösungen zu finden, von Elfenbein-Architekten oder Pflichtenheft- und Lastenheft-Schreibern genommen.

Der Aufschwung der Agilität

Durch die sich ändernden Begebenheiten erkennen immer mehr Menschen aus unterschiedlichen Bereichen, dass sich etwas in der Produkt-Entwicklung und Arbeitswelt ändern muss. Mit dem klassischen Vorgehen kann man die aktuellen Herausforderungen nur schwierig bis gar nicht meistern. Dabei stellen sich häufig die folgenden Fragen:

- Wie können wir mit den verkürzten Release-Zyklen mithalten?
- Wie können wir schnell auf die Launen unserer Kunden reagieren?
- Wie können wir innovative Lösungen entwickeln und den Markt anführen?

Ein Trend, der sich dabei abzeichnet, ist die Adaption von agilen Methoden in IT-fremden Bereichen. So wird schon seit einiger Zeit in vielen Bereichen versucht, zum Beispiel Scrum als Entwicklungs-/Designprozess einzuführen. Ziel ist dabei häufig die Beschleunigung des gesamten Prozesses.

Eine andere Maßnahme, die oft angegangen wird, ist, die Art und Weise zu verändern, wie Projekte, Teams und Unternehmen geführt werden. Dabei kommt dann häufig Agile Leadership ins Spiel, wobei nach flachen Hierarchien und bevollmächtigten Mitarbeitern (Empowerment) gestrebt wird. Was oft in den Firmen und im Management vergessen wird, ist jedoch, dass es mit der einfachen Einführung eines Prozesses, wie er in der Theorie beschrieben ist, nicht getan ist. Genauso wenig wird man Erfolge in der Führung von Mitarbeitern erreichen, wenn man die Hierarchie-Ebenen zusammenstreicht und sich duzt, aber generell so weiter macht wie bis-

her. Ein häufiger Fehler, der in Sachen „Agilität" bereits seit Jahren in der IT vorkommt, wird nun nahtlos in andere Bereiche übernommen *(siehe Abbildung 4)*.

Es wird sich auf die einfach umzusetzenden Aspekte der Agilität wie „1:1"-Einführung eines Prozesses beschränkt und der dafür nötige kulturelle Wandel völlig außen vor gelassen. Gerade der kulturelle Wandel und die damit verbundene Art und Weise, wie wir zusammenarbeiten, sind jedoch die wesentlichen Aspekte der Agilität und helfen in jeglicher Art von Prozessen, selbst im klassischen Wasserfall.

Is Agile Eating up the World?

Im Titel dieses Artikels wird die Frage gestellt, ob Agilität die „Welt isst". Dies ist eine abgewandelte Version der Aussage „Software is Eating up the World" von Marc Andreessen, einem der Mitbegründer von Netscape. Im Jahr 2011 erklärte er schon, dass Software zentraler Bestandteil aller Produkte und des täglichen Lebens ist. Die Frage ist nun, ob der vielfach gewählte Ansatz, ein agileres Vorgehen in vielen Bereichen des täglichen Lebens einzuführen, die daraus entstehenden Herausforderungen zu lösen vermag.

Zunächst ist hier die Verkürzung von Release-Zyklen. Im Bereich der Software-Entwicklung ist dies ein Ziel, das oft mit der Einführung von agilen Methoden erreicht werden soll. Dabei wird die Entwicklungsgeschwindigkeit nicht unbedingt gesteigert, allerdings bereits sehr früh damit begonnen, erste Anforderungen umzusetzen und diese in einer hohen Qualität an den Endkunden oder den Auftraggeber auszuliefern. Wenn es sich um ein reines Softwareprodukt handelt, ist dies einfach umzusetzen. Aber auch bei Hardware und einer Kombination aus Hard- und Software sollte dies möglich sein, wird aber oft als unmöglich angesehen. So könnte man zum Beispiel erste Modelle der Hardware mit Virtual- oder Augmented-Reality-Technologien dem Kunden zugänglich machen. Wenn eine Software Sensordaten von Hardware verarbeitet und gegebenenfalls anzeigt, kann mit der Entwicklung der Software bereits begonnen werden, ohne die reale Hardware fertig entwickelt zu haben. Dafür muss lediglich ein erster Wurf der benötigten Schnittstellen und Ereignisse definiert sein.

Durch die verkürzten Release-Zyklen und regelmäßiges Feedback des Kunden oder Auftraggebers wird auch die zweite Herausforderung angegangen, dem Kunden das zu liefern, was er wirklich benötigt. Man wird bei der Entwicklung sehr viel flexibler und sammelt deutlich mehr Feedback. Als Resultat ist ein kurzfristiger Kurswechsel überhaupt erst möglich.

Auch die letzte Herausforderung der Innovationssteigerung wird im Kern von den agilen Prinzipien adressiert. Klar ist, dass Mitarbeiter zur Entwicklung von Innovationen ein entsprechendes Umfeld benötigen. Ein klassisches, Top-down verwaltetes Unternehmen hemmt jedoch durch viele starre Prozesse und Regeln die Mitarbeiter bei der Entwicklung neuer Ideen. Die existierenden Vorgaben engen dabei ein und Mitarbeiter werden eher dahin gedrängt, das Gehirn am Eingang abzugeben und Dienst nach Vorschrift zu machen. Ein Kern von Agilität ist jedoch auch die Bevollmächtigung (Empowerment) des Teams. Es soll eine Aufgabe gestellt bekommen und hierfür eine Lösung ohne eng definierte Rahmen-Parameter entwickeln. Dadurch, dass ein Team und die Mitarbeiter eigenverantwortlich Entscheidungen treffen können, haben sie einerseits viel mehr Optionen und eine innovative Lösung von Problemen wird gefördert. Andererseits hat jedes Team und jeder Mitarbeiter auch eine ganz andere persönliche Bindung zu seinen Arbeitsergebnissen und wird versuchen, sein Bestes zu geben und eine erfolgreiche Lösung abzuliefern. Immerhin hat man sich selbst für diesen Weg entschieden.

Fazit

Nach Meinung des Autors ist Agilität eine Möglichkeit, dem Wandel der Produkt- und Arbeitswelt zu begegnen. Er begrüßt damit die Adaption von agilen Prinzipien in anderen Disziplinen des täglichen Lebens und geht davon aus, dass sie auch noch in weiteren Bereichen Einzug halten wird.

Wie in der Software-Entwicklung stehen wir hier jedoch auch vor der Herausforderung des kulturellen Wandels. Agilität ist eben nicht nur ein weiterer Entwicklungs- oder Designprozess. Es ist nicht nur die Elimination von Hierarchiestufen und das Ablegen der Krawatte. Es geht in erster Linie darum, die Bedeutung des agilen Manifestes und seiner zwölf Prinzipien zu verstehen sowie einen schmerzhaften und langwierigen kulturellen Wandel zu vollziehen. Auf dem Weg werden sich einige so gut entwickeln, wie man es nie für möglich gehalten hätte, weil sie ihre neu gewonnene Freiheit effektiv nutzen können. Andere werden auf der Strecke bleiben oder sich nach neuen Chancen umsehen. In jedem Fall ist dies jedoch nötig, um mit Agilität die bevorstehenden Herausforderungen zu meistern.

Referenzen

- Karl-Heinz Land, Digitaler Darwinismus, der stille Angriff auf Ihr Geschäftsmodell und Ihre Marke, das Think!Book, 2016
- Karl-Heinz Land, Dematerialisierung – Die Neuverteilung der Welt in Zeiten des digitalen Darwinismus, 2015
- Marc Andreessen, Why Software Is Eating the World, The Wall Street Journal, 2011

Carsten Wiesbaum
carsten.wiesbaum@esentri.com

Carsten Wiesbaum betreut als Software-Architekt und Berater für agile Software-Entwicklung die Kunden der esentri AG bei der Wandlung von klassischen Unternehmensstrukturen sowie Software-Architekturen hin zu modernen digitalisierten Unternehmen. Zur Durchführung eines erfolgreichen Wandels verbindet er dabei moderne Cloud-Technologien und Architekturen mit tiefem Methodenwissen im Bereich der Agilität und des kultureller Wandels. Neben der Projekt-Arbeiter engagiert er sich als Oracle-ACE in den Bereichen „Oracle Fusion Middleware" sowie „Oracle Cloud" und ist Themenverantwortlicher für Microservices bei der DOAG e.V.

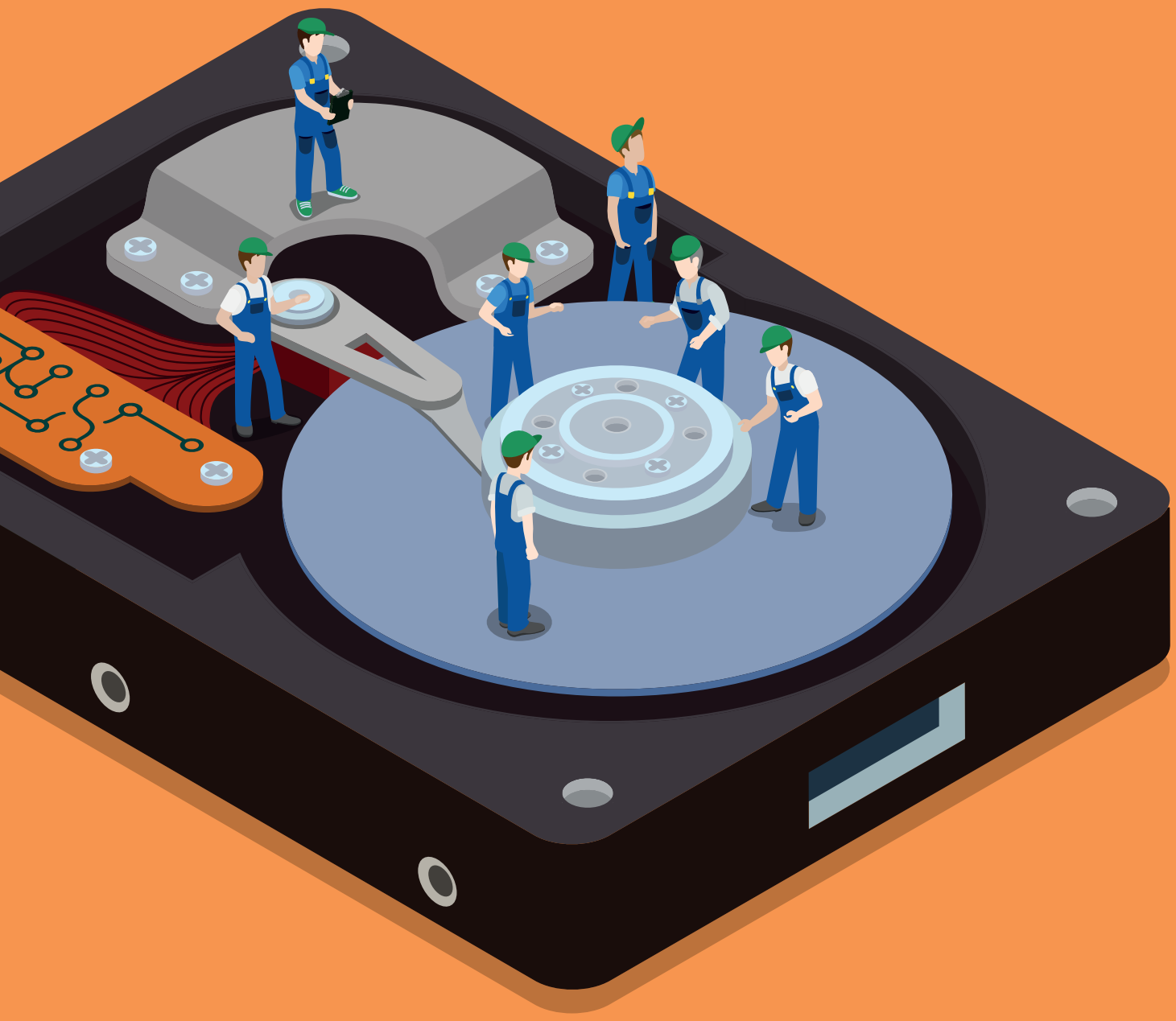
QAWARE
SOFTWARE ENGINEERING

IT-Probleme lösen.
Digitale Zukunft gestalten.

Mit Erfindergeist
und Handwerksstolz.

GREAT PLACE TO WORK®
2018
Beste Arbeitgeber™
Information & Telekommunikation

Von Mitarbeitenden empfohlen:
kununu.com/qaware
qaware.de/karriere



Microservices aus anderen Gründen – ein Erfahrungsbericht

Daniel Clasen und Jan Nonnen, Viaboxx GmbH

Dieser Artikel stellt ein Medizinsoftware-Projekt mit Spring Cloud vor, in dem die gesamte Architektur als komponiertes System realisiert wurde. Er zeigt die Erfahrungen der Autoren, erzählt von den Herausforderungen und hinterfragt kritisch die Vor- und Nachteile der gewählten Architektur. Bei Softwareprojekten in der Medizinbranche sind auch fachliche, rechtliche und prozessbedingte Herausforderungen zu beachten, die Architektur und Design maßgeblich formen und prägen.

Dank Spring Cloud *[1]* ist es heutzutage einfach, mit dem Spring Framework und Java eine Microservice-Architektur aufzubauen. Auch wenn man überall von Microservices hört, sollte man stets überlegen, ob je nach Anforderungen und Projekt so eine Architektur verwendet werden sollte oder nicht. Die Autoren präsentieren ihre Erfahrungen mit dem Einsatz solch einer Architektur in einem Medizinsoftware-Projekt nach dem GAMP-5-Regelwerk *[2]*, das sie mit Spring Cloud Netflix und Angular JS realisiert haben.

In dem Projekt waren Eigenschaften wie Datensicherheit, isolierte Testmöglichkeit und unabhängige Entwicklung wichtiger als die Größe und Hochverfügbarkeit der Komponenten. Durch diese Priorisierung hat sich im Laufe der Zeit die Sicht der Autoren als Software-Entwickler auf Microservices geändert. In anderen Projekten hatten sie bereits Microservices eingesetzt, insbesondere wegen ihrer Unabhängigkeit und dezentralen sowie horizontalen Skalierbarkeit.

Das Medizinprojekt war ein Greenfield-Projekt, das in erster Linie als Informationssystem von einer ganzen Medizinfirma genutzt werden sollte. Bereits im Rahmen der Anforderungserhebung stellte sich jedoch heraus, dass die verschiedenen Abteilungen das frühere Informationssystem komplett unterschiedlich nutzten und verwendete Begriffe anders verstanden. Durch diese Erkenntnis wurde die Idee bestärkt, keinen hochverfügbaren Monolithen zu entwickeln, sondern das Projekt als einen verteilten Workflow zu realisieren und mit einer Microservice-Architektur auch die Einzelbausteine einfacher anpassbar und skalierbar zu gestalten.

Von Microservices zu Modulen

Da man als agiles Unternehmen auch in diesem Projekt agil und iterativ arbeiten wollte, gab es zuerst eine grobe Anforderungserhebung mit Domain Driven Design, um die Begriffe und Abgrenzungen zwischen den Abteilungen zu erkennen und fachliche Schnittstellen zwischen den Komponenten zu definieren. Vor dem ersten Implementierungssprint fanden Verfeinerungssprints statt, um die für die Implementierung benötigten Spezifikationen und Dokumente gemeinsamen mit dem Kunden zu erstellen und aus den fachlichen Schnittstellen technische Komponentenschnittstellen zu entwerfen.

Im Laufe der Zeit entstanden, basierend auf den verschiedenen Abteilungen, fachliche Module, die jeweils ein eigenständiges Frontend besitzen und größere unabhängige Komponenten darstellen. Bei diesem Ansatz gibt es immer den Konflikt zwischen den Software-Entwicklern, die gerne mit möglichst vielen unabhängigen, kleinen, wiederverwendbaren Services arbeiten, und dem Qualitätsmanagement (QM), das sich möglichst wenig installierte Komponenten wünscht, da jede Installation und Änderung jede Menge Dokumentation, Validierung und Aufwand für die Administratoren bedeutet. Im Nachhinein ist das Ergebnis daher eine Mischung aus Microservices und einer dezentralen, serviceorientierten Architektur, die nach den Erfahrung der Autoren gut zusammenspielen und insgesamt eine in ihren Augen moderne, serviceorientierte Architektur darstellen.

Durch den iterativen Projektansatz konnten Anpassungswünsche des Kunden in späteren Iterationen übernommen und es mussten jeweils nur einzelne Services neu herausgegeben und installiert werden. Insbesondere für den Datenschutz stellte sich der Ansatz als sehr gut heraus, so ließen sich kritische Dinge wie Patientendaten in speziellen Services wegkapseln und speichern.

Außerhalb der kritischen, personenbezogenen Services sind diese Daten nur als Pseudonym referenziert. Zudem können Module mehrfach installiert und auch physikalisch getrennt werden. So ist es beispielsweise möglich, je ein Patienten-Modul pro Land zu betreiben, in dem Patientendaten erhoben werden. Das entschärft viele Risiken, die mit personenbezogenen Daten in einer Medizinsoftware einhergehen, und stellt die Einhaltung gesetzlicher Datenschutz-Richtlinien sicher. Nachteilig bei dieser Kapselung von Daten ist, dass Zusammenhänge, die über mehrere Module hinweg aggregiert werden müssen, Kommunikation und somit Netzwerklast zwischen den Modulen verursachen.

Ein weiteres Problem bei der gewählten Architektur und dem agilen Projektansatz ist die Einheitlichkeit beziehungsweise Konsistenz zwischen den Modulen. Da Module teils in unterschiedlichen Feature-Versionen produktiv betrieben werden, muss seitens der Spezifikation und Entwicklung ein besonderes Augenmerk auf

Kompatibilität und Nachziehen von Verbesserungen in anderen Modulen gerichtet und beachtet werden. Ein Feature muss langsam durch die produktiven Module durchsickern, bis es überall einheitlich vorhanden ist, was nachfolgend genauer beleuchtet wird.

Segregation of the UI

Aufgrund der Anforderungen des QM sowie der fachlich relevanten Anforderungen hinsichtlich der Modularität und Unabhängigkeit aller Module fiel die Entscheidung, wie eingangs erwähnt, die Frontends mittels Angular als modulare, eigenständige Web-Apps zu entwickeln. Dass dies einfacher gesagt als getan ist, war bereits vor der Entwicklung klar, wenn auch nicht im vollen Umfang. Die Unterteilung in diverse Frontends bringt zwar den Vorteil mit, dass diese sich später iterativ weiterentwickeln lassen und folglich auch separat getestet werden können, jedoch darf beim Benutzer nicht der Eindruck verschiedener Anwendungen entstehen.

Das Look and Feel sollte also über alle Module hinweg einheitlich sein, sich aber gleichzeitig unabhängig voneinander weiterentwickeln können. Diesen besonderen Anforderungen an die Entwicklung konnte man zum Großteil über die Auslagerung von Frontend-Komponenten wie Eingabefelder, Tabellen-Ansichten, Header, Footer, Layout etc. gerecht werden. Eine Gratwanderung, da durch eine maximale Wiederverwendbarkeit auch ein hoher Abstraktionsgrad entsteht, der ausgiebig getestet werden möchte.

Die ausgelagerte Frontend-Bibliothek wurde in einem separaten „git"-Repository entwickelt, mit „npm" versionisiert und auf einem privaten „npm"-Repository gehostet. Somit können Änderungen an den gängigen Komponenten an einer Stelle implementiert werden, ohne dass dies direkt eine Auswirkung auf alle Frontends hat, denn in jedem Frontend wird eine bestimmte Version der Komponenten-Bibliothek verwendet. Durch die Vermeidung von dupliziertem Code, den man oft in Microservice-Architekturen findet, lässt sich auch der manuelle Testaufwand verringern, da die Komponenten wiederverwendet anstatt neu geschrieben werden.

Wie bei den meisten Angular-Web-Apps findet die Kommunikation mit dem Backend mittels JSON über HTTPS statt. Explizit handelt es sich hier um JSON+HAL, also beschreibende Metadaten und Links auf relevante Ressourcen, die durch Spring HATEOAS *[3]* bei der Serialisierung und Deserialisierung der Kommunikations-Objekte aufbereitet werden. Das REST-API jedes einzelnen Moduls ist zur Wahrung der Kompatibilität bei zukünftigen Änderungen sowohl im Quellcode als auch in der Schnittstelle per URL-Parameter versionisiert. Ein typischer Endpunkt zum Abfragen der Modul-Eigenschaften sieht zum Beispiel so aus: „GET /api/v1/info".

Das REST-API jedes einzelnen Moduls ist im Java-Backend jeweils in ein eigenes JAR ausgelagert, das nur die Ressourcen-Klassen, die Interfaces mit den annotierten Endpunkten und entsprechende Feign-Clients enthält. Dies hat den Vorteil, dass die Definition des REST-API eines Moduls als Dependency in einem anderen Modul genutzt und mithilfe des Feign-Clients direkt angefragt werden kann.

Doch wie fügt sich das nun alles zu einer Anwendung zusammen und woher weiß der Feign-Client, an welchen Host und welche URL die HTTP-Anfragen an ein anderes Modul gehen sollen?

Spring Cloud to the Rescue!

Neben dem Feign-Client kommen auch Zuul als API-Gateway, Eureka als Service-Discovery, Hystrix als Circuit-Breaker und Ribbon als Client-Side-Loadbalancer aus dem Spring-Cloud-Netflix-Stack *[1]* zum Einsatz. Jedes fachliche und technische Modul besitzt einen Typ- („serviceID") und einen Standort-Bezeichner („location"), mit denen er sich an der Service-Discovery anmeldet. Aufgrund dieser beiden Eigenschaften wird dann immer die richtige Microservice-Instanz bei REST-Anfragen angesprochen, was unter anderem Zuul mit einer eigenen Routen-Konfiguration möglich macht. Wird also versucht, Eigenschaften von einem Patienten-Modul in Deutschland abzufragen, sieht der Request so aus: „GET https://domain.com/patients/de/api/v1/info". Dabei wertet das API-Gateway Zuul direkt den angefragten Pfad aus. Aufgrund der enthaltenen Parameter fragt es die Service-Discovery (beziehungsweise einen lokalen Cache) nach einer Liste aller laufenden Instanzen von Modulen mit den Parametern an. Zuul wählt nun mithilfe des Ribbon-Load-Balancers eine der Instanzen aus, an die der Request ohne den Routing-Präfix weitergeleitet wird. Dabei enthält jede Instanz-Information auch das jeweilige Protokoll, Host und Port. Diese Weiterleitung ist kein HTTP-Redirect, sondern wird direkt von Zuul als eine Art Proxy durchgeführt und sieht beispielsweise wie folgt aus: „GET http://prod-01-fra-de.domain.com:8001/api/v1/info". Dabei läuft auf dem Ziel-Host „prod-01-fra-de" das Patienten-Modul auf dem Port 8001. Dies hat wiederum den Vorteil, dass das API-Gateway das einzige Modul ist, das durch die Benutzer erreichbar sein muss. Danach finden alle Anfragen intern statt und eine etwaige Firewall muss nur sicherstellen, dass sich alle beteiligten Hosts unter den verwendeten HTTP(s)-Ports erreichen können.

Auch alle Web-Ressourcen des Frontends werden über diesen Mechanismus durch Zuul als Proxy und letztlich von den einzelnen Services ausgeliefert. Im Browser ist daher das gesamte System unter einer einzigen URL erreichbar, was den angenehmen Vorteil bringt, dass man sich weder im Frontend noch im Backend um CORS kümmern musste.

Segregation of Data?

Die nächste Herausforderung war, die richtige Instanz eines Service zur Abfrage von verknüpften Ressourcen zur Laufzeit zu ermitteln. Wie eingangs erwähnt, gab es regulatorische und datenschutzrechtliche Gründe, gewisse Daten in getrennten Datenbanken zu speichern. Somit entschied man sich, jedem Service die Hoheit über die Verwaltung seiner Daten zu geben, und zwar in Form einer eigenen Datenbank. Diese Anforderung steht jedoch in Konflikt mit der Art und Weise, wie der Kunde in seinem Prozess die Informationen verknüpfen muss.

Dazu ein Beispiel: Ein Befund zu einem medizinischen Fall enthält Daten, die aus so ziemlich allen Services aggregiert werden müssen. Schließlich geht es am Ende um das Wohl eines Patienten, der einer Krankenkasse angehörig ist und einen behandelnden Arzt hat, der den Befund mitteilt und gegebenenfalls eine Medikation passend zur Anamnese veranlasst. Könnten alle diese Informationen nicht mehr aufgelöst und in Relation gebracht werden, da sie sich aus rechtlichen Gründen in diversen Datenbanken befinden, wäre der Befund wertlos. Der Arzt würde den Befund nicht erhalten, der Patient keine Medikation und die Krankenkasse keine Rechnung.

Zur Laufzeit wird ein Lokalisierungs-Service angefragt, um den Standort einer Ressource im gesamten System zu ermitteln. Darauffolgende Requests werden mit dieser Information angereichert, und das gleichermaßen im Frontend wie im Backend.

Diese Trennung der Daten war eine elegante Möglichkeit, den nichttechnischen Anforderungen gerecht zu werden, jedoch stellen sich die Auswirkungen nach nun über zwei Jahren und einigen Iterationen der Entwicklung aus technischer Sicht zunehmend als problematisch dar. Das Sortieren beziehungsweise Suchen nach extern referenzierten und folglich aggregierten Daten kann nicht wie bei einem Monolithen über die Datenbank erfolgen, da sich der dargestellte Datenbestand gar nicht in nur einer Datenbank befindet. Zudem ist bei der Darstellung von Informationen zu beachten, dass der aktuelle Benutzer gegebenenfalls keine Berechtigung hat, die aggregierten Informationen oder Teile davon zu sehen. Mit zunehmender Granularität an Berechtigungen wird es jedoch immer komplizierter, alle Kombinationen zu erkennen und zu testen. Außerdem war es Wunsch der Autoren, aufgrund der allgemeinen SOA-ähnlichen Architektur zu vermeiden, dass sich ein Service um das berechtigungsabhängige Verhalten eines anderen Service kümmern muss.

Segregation of Permissions

Ähnlich wie bei der Trennung der Benutzer-Oberflächen, der Datenhoheit und des Fokus der Services entschieden sich die Autoren, auch die Berechtigungen völlig unabhängig unter den Modulen aufzuteilen. Als Identity-Management-Service mit Active-Directory-Integration und JSON-Web-Token-Provider (JWT) setzen sie in diesem Projekt auf Keycloak. Jeder Request an das Backend jedes einzelnen Moduls muss also ein signiertes Token enthalten, mit dem der agierende Benutzer identifiziert werden kann. Dies hat zur Folge, dass keine herkömmlichen Sessions im Backend verwaltet und keine Informationen des Benutzers, wie seine E-Mail-Adresse, gespeichert werden können. Die E-Mail-Adresse könnte zwar im Token enthalten sein, ist jedoch immer nur transient, da das Active Directory die Datenhoheit über diese Informationen innehat und sich die Daten zu jedem Zeitpunkt ändern können.

Soll nun beispielsweise ein Benutzer benachrichtigt werden, wenn eine bestimmte Aktion durch einen anderen Benutzer ausgeführt wird, muss immer das Identity-Management nach der aktuellen E-Mail-Adresse des Benutzers gefragt werden. Bei den Berechtigungen sieht es ähnlich aus. Zwar sind diese in der Implementierung im Token enthalten, allerdings wertet jedes Modul nur die für sich relevanten Rollen und Rechte aus.

Gibt es nun eine Funktion, die Auswirkungen auf Daten zweier Module hat, muss zunächst immer der andere Service angefragt werden, um zu prüfen, ob der aktuelle Benutzer auf diese Daten zugreifen beziehungsweise die Aktion ausführen darf. Ein Beispiel: Ein medizinischer Fall gilt als abgeschlossen, wenn der daraus resultierende Befund an den behandelnden Arzt versendet wurde. Die

Versenden-Aktion befindet sich im Service, der sich um die Befunde kümmert. Sie hat allerdings auch Auswirkungen auf den Service, der die Fälle verwaltet, und auf den Benachrichtigungs-Service, der die E-Mails oder Faxe versendet.

Die Herangehensweise an die Entwicklung dieser Art von Software war mit dem stark QM-lastigen Prozess, bedingt durch den medizinischen Kontext, an manchen Stellen sehr ungewohnt für das Team. Dies hat sich auch auf die Code-Basis ausgewirkt, schließlich musste man sich Gedanken um Fälle und Ausnahmen machen, die sonst in der Entwicklung à la „Wenn das passiert, steht die Welt eh in Flammen" abgewunken werden. Jedoch entstand nach den ersten Sprints im Team schnell das Verständnis dafür, nicht fragen zu dürfen, ob dieser Edge-Case eintritt, sondern wann und welche Auswirkungen dies am Ende für einen Patienten haben kann.

Zugegeben ist dieses geschaffene Bewusstsein der Verdienst des QM-Teams und dessen – bildlich gesprochenen – vehementen „Auf-die-Finger-Hauens". Das Durchhaltevermögen, diese Prämisse bis an die letzte Stelle im Code zu verfolgen, um eine solche Architektur stemmen zu können, ist jedoch letztendlich genau das, was dieses Projekt ausgemacht hat. All diese harten Schnitte und Trennungen in der Architektur waren aufwendig und haben auf Seiten der Implementierung und Konzeption viel Zeit in Anspruch genommen.

Allerdings zahlt sich der Trade-off inzwischen aus. Obgleich der Kunde ursprünglich gar kein zu 99,99 Prozent verfügbares System haben wollte, will er einen Ausfall durch Updates von nur wenigen Sekunden inzwischen nicht mehr in Kauf nehmen und wünscht sich für die Zukunft einen Blue-Green-Deployment-Prozess *[4]*. Mit Spring Cloud ist das kein Problem und die verteilte Architektur macht es uns noch einfacher. Da die Autoren bereits von Beginn an in der Entwicklung den Fall behandelt haben, dass ein API inkompatibel werden kann oder dass manchmal ein Service nicht schnell genug oder erst gar nicht antwortet, sind alle Module und speziell die Stellen, bei denen es Abhängigkeiten zwischen den Services gibt, auf ein solches Verhalten vorbereitet.

Fazit

Bei vielen der Herausforderungen wie dem Umgang mit den Aggregationen von Datensichten fehlt auch jetzt noch die perfekte Lösung, um sie zu meistern. Überwiegend waren es Kompromisse, die umgesetzt wurden. Diese musste der Kunde tragen und verstehen. Letztlich war das Ziel aller, den Aufwand an Dokumentation, manuellen Tests und externen Validierungen durch CE-Kennzeichnung, TÜV, U.S. Food and Drug Administration (FDA) etc. für die initiale und weiterführende Entwicklung so gering wie möglich zu halten. Das dazu nötige beidseitige Verständnis von Anforderungen, Risiken und Problemstellungen sowie die daraus folgenden Lösungen und Kompromissen wären ohne eine sehr enge Zusammenarbeit mit dem Kunden nicht möglich gewesen. Daher gab es auch auf Kundenseite einige Personen, die aktiv an der Entwicklung und Gestaltung der Software beteiligt waren. So fanden nach jedem Sprint Tests und Feedback-Runden mit Key-Usern statt. Mehrere Stakeholder, wie zum Beispiel der Product-Owner und diverse Abteilungsleiter, gaben bei den Sprint-Planungen essenzielles Fachwissen mit und steuerten die Entwicklung durch Priorisierung der Aufgaben.

Am Ende ist es gelungen, eine Software zu erstellen, die auch – aber eben nicht nur – den QM-relevanten Aspekten gerecht wird und zugleich eine extrem hohe Benutzer-Akzeptanz beim Kunden findet. Dabei ist auch am Ende eine Microservice-orientierte Architektur gewachsen, die geholfen hat, einige der Prozess-inhärenten Probleme in dem Projekt zu lösen, um dafür andere, neue Herausforderungen für die weitere Entwicklung zu schaffen.

Literatur

[1] *https://cloud.spring.io/spring-cloud-netflix*
[2] GAMP 5, ein risikobasierter Ansatz für konforme GxP-computergestützte Systeme, ISPE, 2008
[3] *http://projects.spring.io/spring-hateoas/ & https://tools.ietf.org/id/draft-kelly-json-hal-01.html*
[4] *https://martinfowler.com/bliki/BlueGreenDeployment.html*

Daniel Clasen
daniel.clasen@viaboxx.de

Daniel Clasen ist Software-Entwickler und IT-Nerd mit Herz und Seele. Als Full-Stack-Entwickler – spezialisiert auf moderne UI-Entwicklung und Java-basierte Web-Anwendungen – liebt er jede Art von Herausforderungen, seien sie Software-, Hardware- oder Netzwerk-bezogen. Neben seinem Job als Entwickler und Team-Lead bei der Viaboxx GmbH beteiligt er sich an Open-Source-Projekten, schreibt Blog- sowie Fachartikel und hält Vorträge auf Konferenzen.

Jan Nonnen
jan.nonnen@viaboxx.de

Jan Nonnen ist als Entwickler und Team-Lead schon einige Jahre bei der Viaboxx GmbH aktiv. Daneben ist er bekennender „Software Craftsman" sowie seit dem Jahr 2011 Mitbegründer und Organisator des Bonn Agile Meetups. Er spricht auf Konferenzen und veröffentlicht Fachartikel. Ihn begeistern insbesondere testgetriebene Entwicklung, Clean Code und Software-Design.

REST – Versprechen und Wirklichkeit

Thomas Bayer, predic8

REST verspricht eine einfache und leistungsfähige Kommunikation zwischen Anwendungen. Es kommt bei zahlreichen öffentlichen und zunehmend organisationsinternen Schnittstellen zum Einsatz. Ein Grund für die weite Verbreitung sind die geringen Hürden für die Nutzung eines REST-API. Für das Testen einfacher Aufrufe genügt ein Browser und Clients lassen sich in jeder Programmiersprache mit wenigen Zeilen Code realisieren; ein Compiler oder Build-Werkzeug ist nicht notwendig. Auf den zweiten Blick hat REST Nachteile für nicht öffentliche Schnittstellen. In der Projekt-Wirklichkeit verursacht es Mehraufwände, wirft Fragen auf und bringt Komplexität mit sich. Dieser Artikel macht auf die Nachteile von REST aufmerksam und ermutigt, Alternativen wie GraphQL oder JSON RPC auszuprobieren.

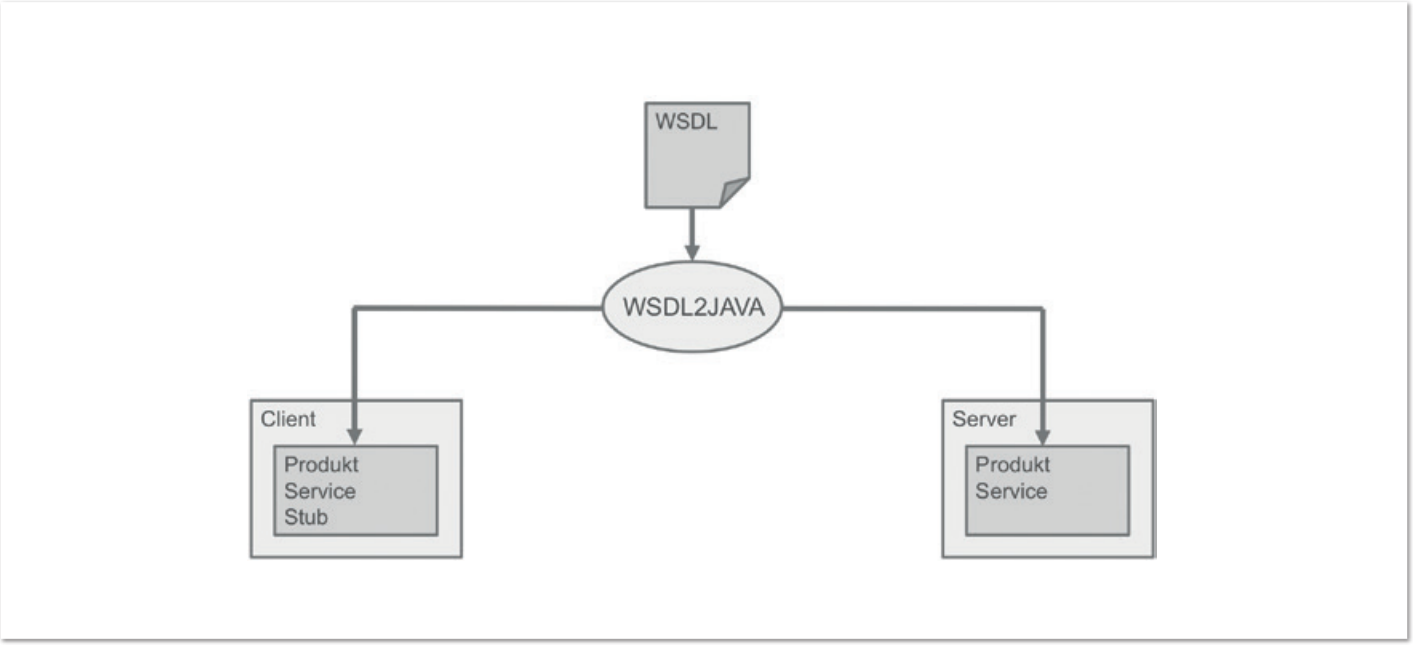


Abbildung 1: Ein Generator erzeugt Code für Client und Server

REST ist sehr technisch, was anhand eines Vergleichs mit Remote Procedure Calls ersichtlich ist. Als Beispiel für den Vergleich dienen die SOAP-basierten Web-Services. SOAP ist veraltet und die „WS-*" -Standards sind ein Paradebeispiel für unnötig komplexe Abläufe. SOAP soll hier nur als Beispiel dienen, da es von den RPC-Technologien derzeit noch am bekanntesten ist; man könnte für den Vergleich genauso gut Google RPC verwenden.

Zentral bei vielen RPC-Technologien ist eine Schnittstellen-Beschreibung. Im Beispiel in *Abbildung 1* ist dies ein WSDL-Dokument, aus dem ein Generator den Code für Client und Server erzeugt. Die für den Client erzeugte Bibliothek kümmert sich um alles Technische wie die Serialisierung zwischen Java und XML sowie die Kommunikation über HTTP.

Funktionen des Servers können mithilfe der Client-Bibliothek einfach aufgerufen werden. Der Code des Web-Service-Aufrufs in *Abbildung 2* unterscheidet sich nicht von einem lokalen Aufruf einer Funktion.

Das Erzeugen des Codes sowie das Kompilieren und Einbinden sind meist mithilfe eines Build-Werkzeugs automatisiert. Das Aufsetzen des Builds beispielsweise mit Maven ist der komplizierteste Arbeitsschritt bei RPC. Das eigentliche Programmieren hingegen ist einfach. Der Entwickler benötigt für die Arbeit mit SOAP weder XML- noch HTTP-Kenntnisse.

Die Tatsache, dass die Kommunikation über das Netz erfolgt, wird bei RPC in der Client-Bibliothek „weggekapselt"; so werden beispielsweise Netzwerk- und Anwendungsfehler in Exceptions der jeweiligen Programmiersprache überführt. Im Gegensatz zu RPC werden bei REST die Eigenschaften von HTTP ganz bewusst für die Entwicklung auf der Anwendungsebene genutzt:

- Die Adresse eines Objekts ist gleichzeitig seine ID (URI)
- Über HTTP-Header werden die Formate der Payload ausgehandelt
- Der Client interpretiert HTTP-Status-Codes und fängt keine Exceptions ab

Listing 1 zeigt typischen REST-Client-Code. Im Gegensatz zum RPC-Einzeiler ist der Aufruf aufwendiger und technischer. Der Client-Entwickler benötigt selbst bei einfacheren Client-Bibliotheken wie Unirest folgende Kenntnisse:

- Das HTTP-Protokoll
- Das Format der Repräsentation wie JSON
- Die REST-Prinzipien

REST ist die Technologie der Wahl für einfache öffentliche Schnittstellen. Der Client-Entwickler muss weder Compiler noch Code-Generator verwenden. Für komplexere Schnittstellen innerhalb eines Unternehmens spielt der Build-Aufwand einer RPC-Technologie eine untergeordnete Rolle. Dafür kann diese eine einfachere Um-

```
GetMethod method = new GetMethod("http://api.predic8.de/shop/products/65");

int sc = client.executeMethod(method);

if (sc != HttpStatus.SC_OK) {
    …
}

byte[] body = method.getResponseBody();
```

Listing 1: Client mit HTTP-Bibliothek

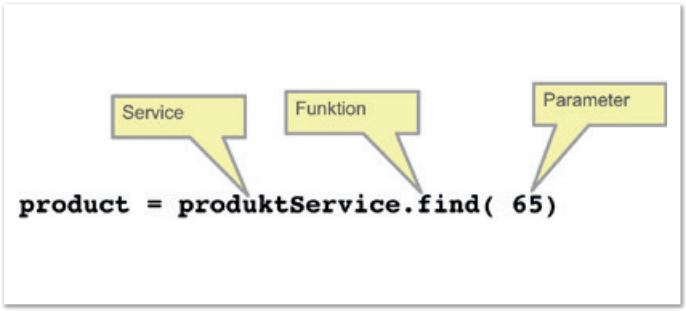


Abbildung 2: RPC-Aufruf am Beispiel von SOAP

setzung von Client und Server besonders bei Services mit vielen Parametern oder komplexen Datenstrukturen ermöglichen. Wird REST verwendet, muss jeder Anwendungsentwickler und Business-Analyst zusätzlich HTTP- und REST-Experte sein.

Fehlende Standards

Für REST gibt es keinen Standard. Wer wissen möchte, wie ein API zu gestalten ist, der schaut in die Dissertation von Fielding, in die HTTP-Spezifikation und auf Stack-Overflow nach. Diese Quellen werfen oft mehr Fragen auf, als sie beantworten. Daher gibt es unzählige API-Style-Guides von Adidas oder dem Weißen Haus, die diese Lücken füllen.

Roy Fielding beschreibt in seiner Arbeit „Architectural Styles and the Design of Network-based Software Architectures" aus dem Jahr 2000 in Kapitel 5 einen Architektur-Stil und nennt diesen „Representational State Transfer". Konkrete Hinweise darauf, wie ein REST-API zu gestalten ist, fehlen dort. Fieldings Arbeit ist nicht als Standard oder Handbuch geeignet, vielmehr als Hintergrund für ein Anwendungsprotokoll auf Basis des HTTP-Standards.

APIs sollten den HTTP-Standard einhalten. Zum Beispiel beschreibt die HTTP-Spezifikation, dass ein Server auf eine POST-Anfrage mit dem Status-Code „201 Created" antworten muss, falls eine neue Ressource angelegt wurde. HTTP beschreibt die Kommunikation zwischen Browser, Proxy und Web Server. Die Besonderheiten einer Kommunikation zwischen einem Anwendungsclient und einem Web-API sind dort nicht beschrieben. Verfolgt man auf Stack-Overflow Diskussionen zum REST-Design, kommen Fragen wie:

- Soll POST oder PUT für das Anlegen von neuen Ressourcen verwendet werden?
- PUT oder PATCH für das Ändern von Ressourcen?
- Darf ein GET-Request einen Body enthalten?
- Welchen Status-Code verwendet man für …

Es wird deutlich, dass es bei REST viel Spielraum für Interpretationen gibt. Häufig werden Fielding oder die HTTP-Spezifikation zitiert und die Bedeutungen ausgelegt:

- Laut Fielding …
- Ich verstehe die RFC 2762 anders …
- Dann ignorierst du die Empfehlung in HTTP 1/1 Sektion 4.3 …

Abbildung 3 zeigt eine typische Diskussion auf Stack-Overflow, die mehr als eine Million Aufrufe erzielen kann. Die Auslegung der HTTP-Standards und das Philosophieren über REST machen bisweilen Spaß. Aber bringt das ein Projekt wirklich weiter? Die Zeit zum Finden einer REST-konformen Lösung könnte zielführender für die Arbeit am Problem des Kunden eingesetzt werden.

Für REST-Puristen relativiert der Autor seine Aussage: Verwendet der Client Hypermedia und stellen die Repräsentationen ausschließlich den Vertrag zwischen Client und Server dar, so genügt Fieldings Arbeit in Verbindung mit der HTTP-Spezifikation. Würden REST-Prinzipien besser umgesetzt, gäbe es diese Kritik nicht. Bisher wurde so argumentiert: REST ist perfekt, es wird nur nicht richtig umgesetzt. Man kann auch anders argumentieren: REST ist so abgehoben und komplex, dass selbst 17 Jahre nach Fieldings Arbeit der Anteil an REST-konformen APIs verschwindend gering ist.

Das API-Design ist kompliziert

Ein API zu entwerfen, ist um einiges schwieriger, als eines aufzurufen. Beim Entwurf einer Schnittstelle sind zahlreiche Entscheidungen zu treffen: Werden Daten mit POST, PUT oder PATCH geändert? Kommt an das Ende von Container-Ressourcen ein Schrägstrich oder nicht, also „/produkte/" oder „/produkte"? Wie werden Relationen abgebildet? Nehmen wir an, ein Hersteller kann mehrere Produkte liefern. Wie wird dann eine Produkt-Ressource erzeugt? Etwa mit einem Post an die Container-Ressource *(siehe Listing 2)*? Oder mit einem POST an die Liste der Produkte des Herstellers durch „POST /hersteller/32/produkte/"?

Viel Spielraum gibt es auch bei der Abbildung von Relationen. Als Referenzen auf andere Geschäftsobjekte werden in den Repräsen-

Roy Fielding's comment about including a body with a GET request.

Yes. In other words, any HTTP request message is allowed to contain a message body, and thus must parse messages with that in mind. Server semantics for GET, however, are restricted such that a body, if any, has no semantic meaning to the request. The requirements on parsing are separate from the requirements on method semantics.

So, yes, you can send a body with GET, and no, it is never useful to do so.

This is part of the layered design of HTTP/1.1 that will become clear again once the spec is partitioned (work in progress).

....Roy

Yes, you can send a request body with GET but it should not have any meaning. If you give it meaning by parsing it on the server and *changing your response based on its contents*, then you are ignoring this recommendation in the HTTP/1.1 spec, section 4.3:

[...] if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request.

And the description of the GET method in the HTTP/1.1 spec, section 9.3:

The GET method means retrieve whatever information ([...]) is identified by the Request-URI.

which states that the request-body is not part of the identification of the resource in a GET request, only the request URI.

share improve this answer

edited Apr 18 '16 at 18:18 Jarl 2,443 • 3 • 16 • 29

answered Jun 11 '09 at 20:27 Paul Morgan 16.5k • 3 • 18 • 26

Abbildung 3: Beitrag auf Stack-Overflow zu „HTTP GET mit Request Body"

```
POST /produkte/

{
  "hersteller": 342
  ...
}
```

Listing 2

```
{
  “vendor“: 342
  ...
}
```

Listing 3

```
{
  “vendor_url“: “https://api.predic8.de/shop/vendors/32“
}
```

Listing 4

```
PATCH /bestellung/74

{
  “status“: “storniert“
}
```

Listing 5

```
/rechnungen/
/rechnungen/{rid}
/positionen/
/positionen/{pid}
```

Listing 6

```
/rechnungen/
/rechnungen/{rid}
/rechnungen/{rid}/positionen/
/rechnungen/{rid}/positionen/{pid}
```

Listing 7

tationen oft Schlüssel verwendet *(siehe Listing 3)*. REST basiert auf Hypermedia. Anstatt eines Primärschlüssels in der Repräsentation sollte eine URI verwendet werden *(siehe Listing 4)*.

Bei der Verwendung von Verweisen stellen sich weitere Fragen:

- Sollten relative oder absolute URIs verwendet werden?
- Soll man ein eigenes Format für die Repräsentation einsetzen oder einen Standard wie die Hypertext Application Language?

In diesem Absatz wurden nur einige Fragestellungen exemplarisch aufgezählt. Je tiefer man sich mit dem API-Design beschäftigt, desto mehr tauchen davon auf.

REST begünstigt das CRUD-Antipattern

Die Eigenschaften der REST-Architektur führen zwangsläufig zu datengetriebenen Schnittstellen. Die wenigen HTTP-Methoden müssen für die Abbildung aller Funktionalitäten einer Schnittstelle ausreichen. Meist werden nur die CRUD-Methoden POST, GET, PUT und DELETE verwendet, die den Datenbankoperationen CREATE, READ, UPDATE und DELETE entsprechen. Die Kunst besteht darin, aussagekräftige Hauptwörter, also URIs, zu finden, auf die die wenigen HTTP-Methoden angewendet werden. Dazu ein Beispiel: Der Funktionsaufruf einer Nicht-REST-Schnittstelle dient zum Stornieren von Bestellungen: „storniereBestellung(74)“. Eine Möglichkeit, diesen Funktionsaufruf mit REST abzubilden, wäre ein DELETE-Aufruf an die zu stornierende Ressource mit „DELETE /bestellung/74“.

Für einfache Anwendungen ist das eine praktikable Lösung. Bei einem Storno wird einfach die Ressource gelöscht. Was aber, wenn die Daten später noch benötigt werden? Anstatt eine Bestellung zu löschen, könnte alternativ ein Storno angelegt werden: „PUT /bestellung/74/stornos“. Eine dritte Variante wäre das Überschreiben

Abbildung 4: Verlinkte Repräsentationen

der Bestellung mit einer neuen Repräsentation, bei der der Status aktualisiert würde *(siehe Listing 5)*.

Alle drei Varianten bilden den Aufruf „storniereBestellung()“ über eine Manipulation der Daten ab. REST-Schnittstellen bilden oft die Datenschicht ab, ohne zu abstrahieren und Geschäftslogik zu kapseln. Die Logik wandert dann in den Client und es entsteht eine Zweischichten-Architektur. Ein API ist keine Abstraktion für den Datenbank-Zugriff. Für die Manipulation von Daten gibt es SQL.

Eine weitere Variante des Antipattern ist eine flache CRUD-Schnittstelle, bei der jede Tabelle auf zwei URI-Templates abgebildet ist: eines für einzelne Objekte und eines für Listen. Das Beispiel in *Listing 6* zeigt die URIs für Rechnungen und die zugehörigen Positionen.

Ein solches Design ist „CRUD über HTTP“! Mit Unter-Ressourcen lässt sich die Fachlichkeit passender abbilden. Eine Position ist immer einer Rechnung zugeordnet. Eine Position existiert nie ohne eine Rechnung. Mit Sub-Ressourcen könnten abhängige Objekte zuordnet werden *(siehe Listing 7)*.

Eine neue Position ließe sich dann über die zugehörige Rechnung mit „POST /rechnungen/{rid}/positionen/“ erzeugen. Beim Löschen einer Rechnung sollten alle zugehörigen Positionen gelöscht werden. Das Löschen der Positionen könnte die Implementierung übernehmen. Das API übernimmt Verantwortung und kapselt die Geschäftslogik. Diese Realisierung ist besser, aber immer noch datenorientiert.

Wie das Beispiel zeigt, können Schnittstellen mit REST modelliert werden, die Abstraktionen bieten und Verhalten kapseln. Die Verwendung von Hypermedia verringert die Gefahr des CRUD-Antipattern. Dennoch neigt das REST-API-Design zu datenorientierten Schnittstellen mit der Gefahr der Verlagerung von Geschäftslogik vom Server zum Client.

Noch ein weiteres Beispiel: Wie startet man mit REST eine virtuelle Maschine oder einen Container? Ein Aufruf wie „POST /container/73/start“ verwendet ein Verb in der URI und ist somit nicht REST-konform. Wie sieht eine REST-konforme Lösung aus, die nicht datengetrieben ist?

Hypermedia wird nicht genutzt

Hypermedia ist bei REST nicht optional. *Abbildung 4* zeigt ein Beispiel mit drei verlinkten Repräsentationen. Der Aufbau der URIs spielt keine Rolle. Anstatt „/shop/vendors/672“ könnte eine URI auch „/foo“ oder „/3141“ heißen. Die Adresse für den Einstieg in das API ist die einzige URI, die dem Client bekannt sein muss. Danach sollte ein RESTful-Client die Repräsentationen interpretieren und die enthaltenen Links verfolgen. Im Client sollte keine weitere URI oder ein Template für URIs wie „/vendor/{vid}“ hinterlegt sein.

Bestimmte Fragen würden sich bei einem RESTful-Design erst gar nicht stellen; beispielsweise wäre das Design von URIs unnötig. Über Hypermedia wird viel geredet, es wird jedoch leider nur selten eingesetzt. Einige APIs verwenden Links, aber kaum ein Client nutzt diese zur Navigation zwischen Ressourcen. *Listing 8* zeigt einen REST-Client, der mit der Traverson-JavaScript-Bibliothek einen Link von einer Artikelbeschreibung zum Hersteller verfolgt.

```
const traverson = require('traverson');

traverson
  .from('https://api.predic8.de/shop/products/33')
  .json()
  .follow('vendor_url')
  .getResource((err, json) => {
     console.log(json.name)
});
```

Listing 8: Client mit HATEOAS-Unterstützung

```
{
  "products": [
        {
            "name": "Bananas",
            "product_url": "/shop/products/3"
        },
        {
            "name": "Oranges",
            "product_url": "/shop/products/10"
        },
        {
            "name": "Pineapples",
            "product_url": "/shop/products/33"
        }
    ]
}
```

Listing 9: Repräsentation mit Produktname und Link

```
ProductsApi api = new ProductsApi();
  for(ProductEntry pe : api.getShopProducts().getProd-
ucts()) {
    System.out.println(pe.getName());
  }
}
```

Listing 10

Vielleicht werden zukünftige Client-Programme mit Hypermedia-basierten Bibliotheken realisiert. Momentan wird Hypermedia im Client meist ignoriert. Möglicherweise sind Hypermedia und HATEOAS zu komplex oder deren Nutzen wird unterschätzt? Ohne Hypermedia fehlt REST ein wesentliches Konzept und die Vorteile wie lose Kopplung und State Transfer sind nicht nutzbar.

Swagger tötet Hypermedia

Mit Swagger beziehungsweise Open-API lassen sich REST-APIs beschreiben. Aus einer Swagger-Beschreibung kann über einen Code-Generator eine Client-Bibliothek erzeugt werden. Wird aus einem Swagger-Dokument eine Bibliothek erzeugt, dann folgt der Client, der diese verwendet, dem RPC-Paradigma. Ein Beispiel soll dies erläutern. Angenommen, ein GET-Aufruf gegen eine Ressource für Produkte liefert die Repräsentation in *Listing 9*.

Listing 10 zeigt einen auf einer mit Swagger erzeugten Bibliothek basierenden Java-Client, der diese Liste abruft und eine Liste mit den Produktnamen ausgibt.

Die Bibliothek schirmt den Entwickler vollkommen von HTTP ab; im Code ist nichts vom HTTP-Protokoll oder von JSON erkennbar.

```
ProductsApi api = new ProductsApi();
for(ProductEntry pe : api.getShopProducts().getProducts()) {
  Matcher matcher = Pattern.compile(".*\\/(.*)$").matcher(pe.getProductUrl());
  if(matcher.matches()) {
    int id = Integer.valueOf(matcher.group(1));
    System.out.printf("URL: %s Id: %d\n", pe.getProductUrl(), id);
    Product product = api.getShopProductsId(id);
    System.out.printf("%s : %s\n\n", product.getName(), product.getPrice());
  }
}
```

Listing 11: Parsen von URIs mit regulären Ausdrücken

Angenommen, der Client soll zusätzlich zum Produktnamen auch den Preis anzeigen. Um an den Preis zu gelangen, sind die „product_url"-Verweise im *Listing 11* zu verfolgen. Der Code zeigt den erweiterten Client.

Um die Repräsentation eines einzelnen Produkts abzurufen, ist ein Schlüssel im Integer-Format mit „Product product = api.getShopProductsId(id);" an die Methode „getShopProductsId" zu übergeben. Die Repräsentation der Produktliste enthält jedoch keine Integer-Schlüssel, sondern REST-konforme URIs, die auf die Produkt-Ressourcen verweisen. Dem Client-Entwickler bleibt nichts anderes übrig, als eine URI beispielsweise mit einem regulären Ausdruck zu parsen, um die ID zu erhalten.

Über URI-Templates und Swagger wird im RPC-Stil mit einer REST-konformen Schnittstelle kommuniziert. Warum wird für das REST-Design so viel Aufwand betrieben, wenn dies auf dem Client nicht genutzt wird?

API-Beschreibungen wie Swagger sind nützlich und erleichtern die Arbeit. Schnittstellen-Beschreibungen sind allerdings nicht im Sinne von REST. Ist der Einsatz einer Schnittstellen-Beschreibungssprache gewünscht, sollte über den Einsatz einer RPC-Technologie mit Beschreibungssprache etwa über Google RPC nachgedacht werden. Wird REST richtig eingesetzt, ergeben sich diese Vorteile:

- Lose Kopplung zwischen Client und Server: URIs sind nicht Teil des Vertrags und können umbenannt werden
- Es ist keine Dokumentation notwendig
- Unterstützung für dynamische Clients
- Ohne eine Anpassung der Clients können Schnittstellen evolutionär verändert werden
- Caching kann ohne Änderungen von Client und Server nachträglich eingeführt werden

Mit URI-Design und starren URIs sind diese Vorteile nicht nutzbar. Wenn die REST-Vorteile nur mit Hypermedia zu erzielen sind, aber niemand Hypermedia nutzt, ist es dann sinnvoll, REST zu verwenden?

Fazit

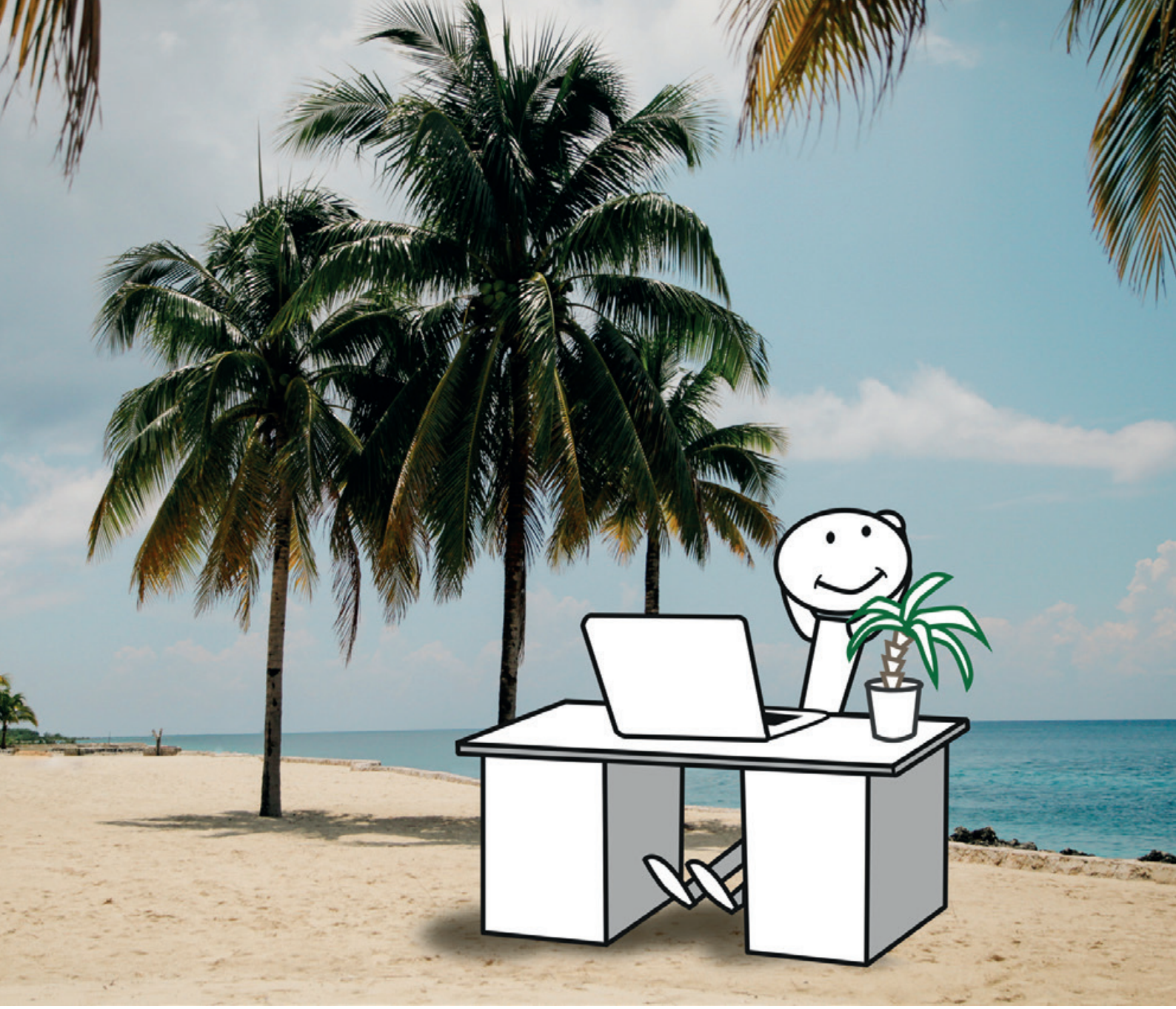
Für einfache öffentliche APIs ist REST die Technologie der Wahl. Bei nicht öffentlichen Schnittstellen lohnt es sich, die Verwendung von REST zu hinterfragen und dem Hype nicht blind zu folgen. Vielleicht ist GRPC, GraphQL oder JSON RPC für bestimmte Aufgaben die passendere Technologie.

Quellen

- Roy Thomas Fielding; Architectural Styles and the Design of Network-based Software Architectures: *https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*
- Representational state transfer (REST) and Simple Object Access Protocol (SOAP): *https://stackoverflow.com/questions/209905/representational-state-transfer-rest-and-simple-object-access-protocol-soap*
- HTTP GET with request body: *https://stackoverflow.com/questions/978061/http-get-with-request-body*
- White House Web API Standards: *https://github.com/WhiteHouse/api-standards*
- Adidas-API-Guidelines: *https://github.com/adidas-group/api-guidelines*

Thomas Bayer
bayer@predic8.de

Thomas Bayer ist Mitgründer der predic8 in Bonn und der OIO in Mannheim. Als Berater und Trainer ist er mit den Themen „Microservices" und „REST" unterwegs. In seiner Freizeit praktiziert er Yoga, fotografiert und baut Quadkopter.



„Lass uns mal skypen!“ – Remote Meetings erfolgreich moderieren

Steven Schwenke, msg DAVID

Zum verteilten Arbeiten gehört eine funktionierende Kommunikations-Infrastruktur, das ist allen klar. Oft wird jedoch nur beispielsweise Skype für alle Kollegen eingerichtet und damit soll dieser Punkt erledigt sein. Doch zur Organisation und Moderation produktiver Meetings gehört mehr – nämlich Schulung und Erfahrung. In diesem Artikel, der eine Fortsetzung des Artikels „Coden von der Dachterrasse – in Rumänien“ aus der letzten Ausgabe ist, werden Aspekte und Ratschläge für verteilte Meetings gegeben, die erfolgreich sind und allen Beteiligten Spaß machen.

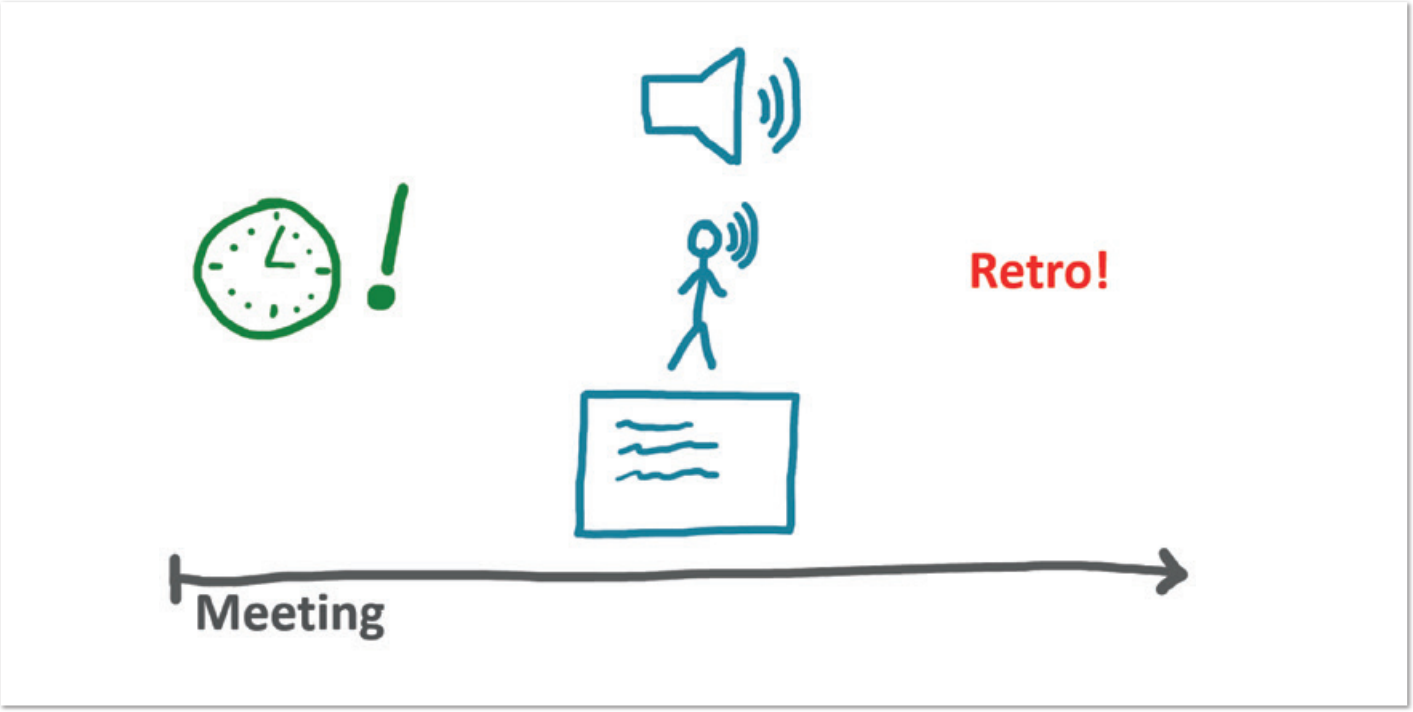


Abbildung 1: Phasen jedes Meetings

Doch von vorne: Was ist eigentlich ein virtuelles Meeting? Sobald sich mindestens zwei Personen mithilfe einer technischen Infrastruktur über mindestens den Audio-Kanal unterhalten, kann man von einem „Meeting" sprechen. Zusätzlich kommen dann noch Video-Übertragungen, Screen-Sharing und der Chat hinzu. Treffen sich mehr als nur zwei Personen, gelten ganz neue Regeln, um dieses Ereignis sinnvoll zu gestalten. Einen guten Eindruck über eine wenig sinnvolle Gestaltung gibt das YouTube-Video „A conference call in real life" *[1]*. Hier werden häufige Fehler wie zum Beispiel Hintergrundgeräusche, schlechte Moderation oder technische Schwierigkeiten dargestellt.

Phasen jedes Meetings

Obwohl in der IT täglich Meetings abgehalten werden, ist vielen die grundsätzliche Struktur dieser Treffen entweder unbekannt oder egal. Dabei sollte wenigstens der Moderator darüber Bescheid wissen. Ach, Sie haben oft gar keinen Moderator? Dann wissen Sie ja, wo Sie anfangen können *(siehe Abbildung 1)*.

Die folgenden Ratschläge gelten sowohl für normale Besprechungen, bei denen alle an einem Tisch sitzen, als auch für verteilte Meetings. Der Moderator leitet jedes Meeting ein. Noch vor dem offiziellen Start prüft er nebenbei, ob alle eingeladenen Personen anwesend sind, und stellt durch Smalltalk eine angenehme Atmosphäre für alle her.

Sind alle Voraussetzungen geschaffen, eröffnet er die Veranstaltung durch die Vorstellung der Ziele, Struktur und Regeln. Er sorgt dafür, dass besprochene Inhalte für alle verständlich bleiben, am besten durch das Anregen von Visualisierungen auf Flipcharts, Whiteboards oder PowerPoint. Sind alle Punkte besprochen, fasst der Moderator die Ergebnisse zusammen und schließt das Treffen ab.

Im Nachgang sorgt er für die Verteilung aller Dokumente, zum Beispiel des Protokolls. Am Treffen verhinderte Teilnehmer sollten das Protokoll ebenfalls bekommen, eventuell wird es zusätzlich in einem Dokumenten-Management-System abgelegt. Eine kurze Retrospektive ist ebenfalls empfehlenswert: Was ist gut gelaufen, was nicht? Was möchte ich nächstes Mal als Moderator besser machen?

Virtuelle Meetings: Vor dem Treffen

Das oben Beschriebene gilt für alle Meetings. Wird sich virtuell verabredet, ist also mindestens eine Person nicht im Raum, gelten zusätzliche Regeln *(siehe Abbildung 2)*. Aufgrund der zusätzlichen technischen Komponente sollte der Moderator eines virtuellen Meetings fünfzehn Minuten vor Beginn mit der Vorbereitung anfangen. Das umfasst das Verbinden in den Skype-Besprechungsraum oder die Telefonkonferenz. Sollte es zu grundsätzlichen Problemen kommen (doppelte Belegung des Raums, keine Verbindung möglich, Server überlastet), bleibt genügend Zeit für das Beschaffen einer Alternative und das Informieren der restlichen Teilnehmer. Geht alles gut, kann die Zeit mit stummgeschaltetem Mikrofon zum normalen Arbeiten genutzt werden.

Leider werden On-Site-Meetings, also Treffen an einem gemeinsamen Ort, oft „auf Kante genäht". So endet ein Termin genau dann, wenn der nächste schon beginnt. Je nach Raumsituation ist es vielleicht noch möglich, mit nur zwei Minuten Verspätung über den Flur zu hechten und quasi von einem Besprechungsraum zum nächsten zu springen. Online kann das zu größeren Verzögerungen führen. Deshalb sollte jeder Teilnehmer fünf Minuten vorher eingewählt sein.

An dieser Stelle sei darauf hingewiesen, dass die umgekehrte Argumentation natürlich ebenfalls richtig ist. In großen Gebäude-Komplexen ist allein das Finden eines Besprechungsraums eine Herausforderung und das schnelle Wechseln zwischen zwei Räumen kann zur sportlichen Herausforderung werden. Solche physische Aktivität entfällt bei virtuellen Meetings. Läuft alles gut,

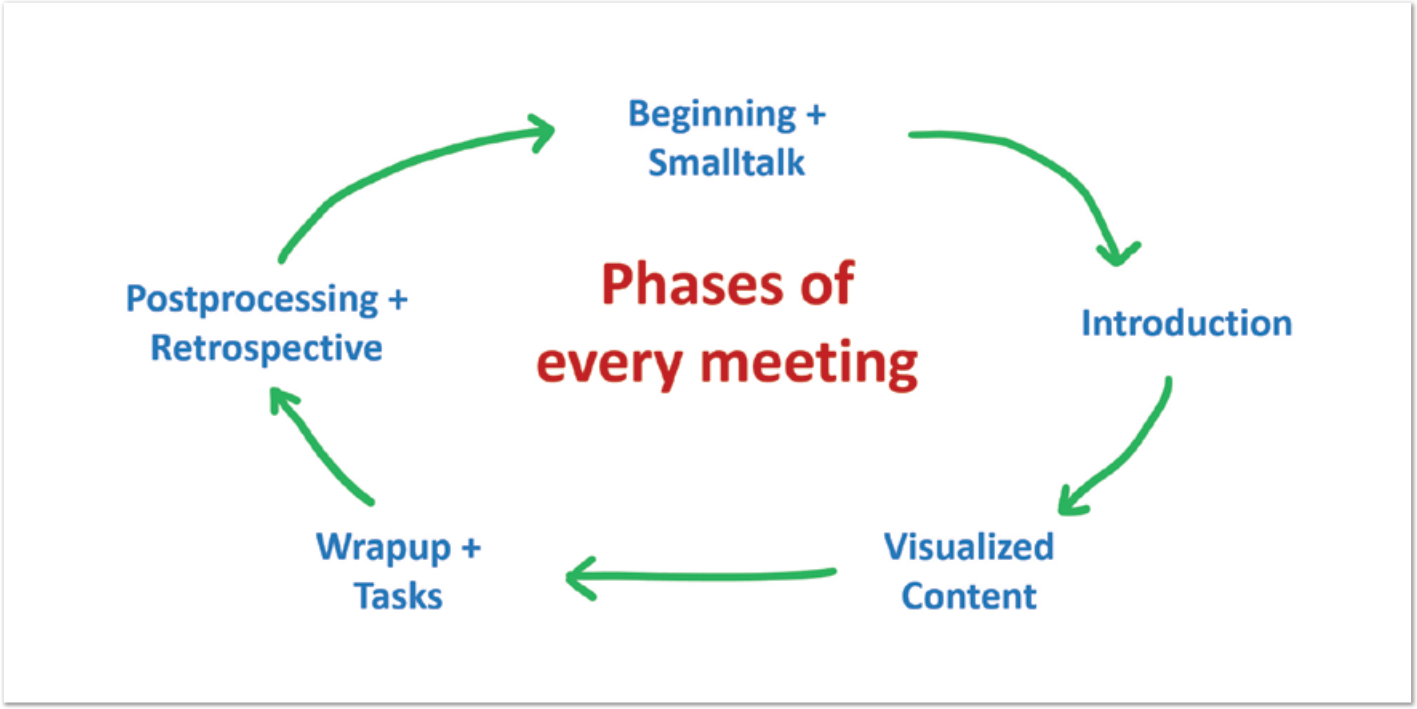


Abbildung 2: Zusätzliche Phasen virtueller Meetings

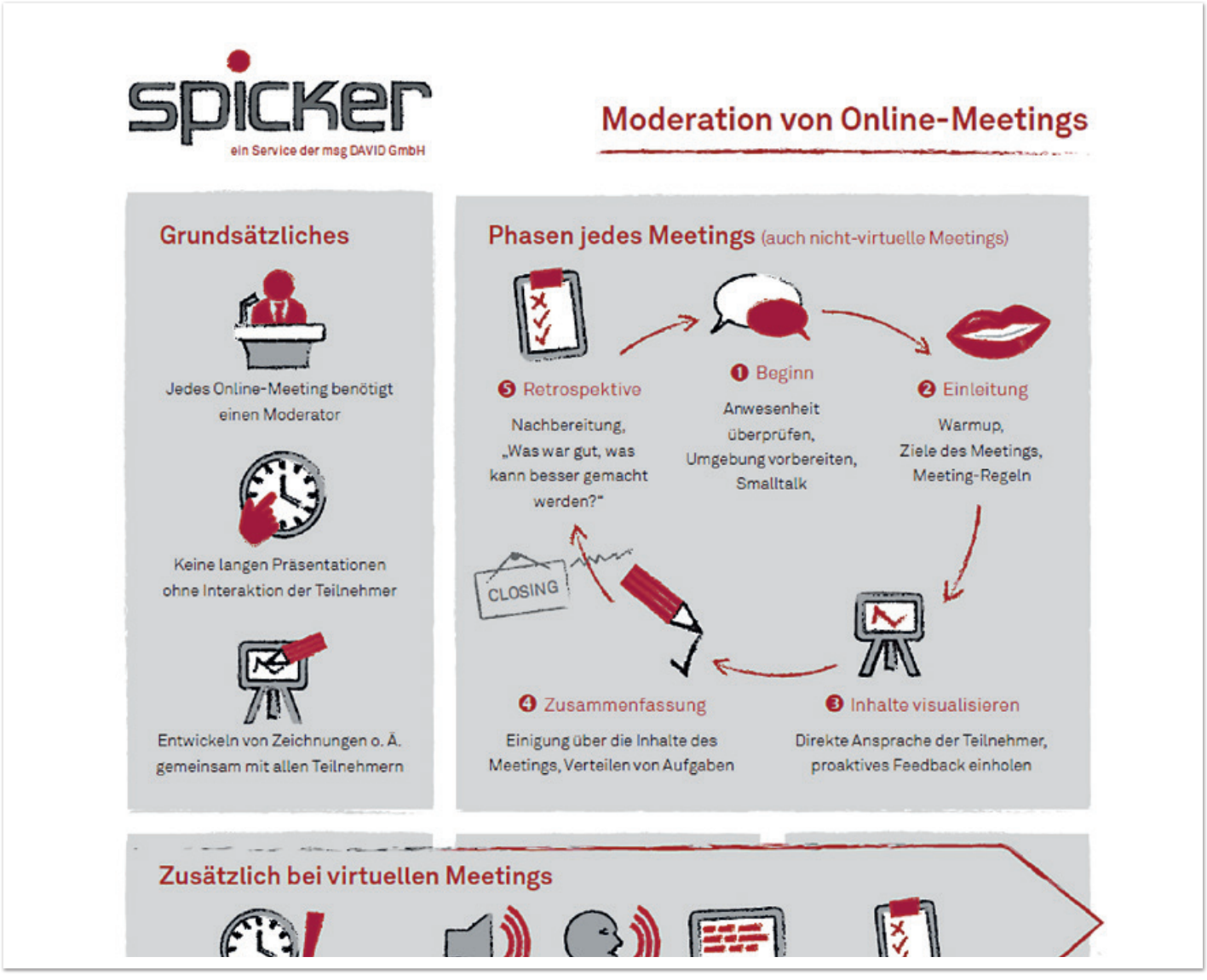


Abbildung 3: Das Cheat Sheet der Firma msg David

ist man innerhalb einer Minute im Meeting. Leider zeigt die Erfahrung, dass eben nicht immer alles glattläuft und man deshalb vorsorgen sollte.

Virtuelle Meetings: Im Treffen

Der Beginn einer virtuellen Besprechung ist besonders wichtig, da Abwesenheiten und technische Probleme wesentlich schwieriger zu erkennen sind als bei On-Site-Meetings. Eine einfache Lösung bietet die Frage „Kann mich jeder hören?", die sich per gesprochenem „Ja" oder einem nach oben gestreckten, in die Kamera gehaltenen Daumen beantworten lässt. Die verbale Bestätigung bietet den Vorteil, dass der Moderator gleich mitbekommt, ob das Mikrofon des Teilnehmers funktioniert. Bei mehr als fünf Teilnehmern wird das jedoch unübersichtlich, hier bietet sich die Daumen-Methode an. Dabei wird der Moderator auch gleich daran erinnert zu prüfen, ob jeder Teilnehmer seine Webcam aktiviert hat.

Auf jeden Fall sollte man sich verbal äußern, wenn die Teilnehmer in dieser Konstellation das erste Mal zusammentreffen. Es zeigt sich, dass in solchen Situationen eher zurückhaltend reagiert und wenig gesprochen wird. Eine produktive Diskussion kommt manchmal gar nicht erst zustande. In diesem Fall ist es die Aufgabe des Moderators, zu Beginn jeden Teilnehmer kurz zum Reden zu bringen. „Hatte jeder ein tolles Wochenende?", „Was ist eigentlich eure Lieblingsfarbe / euer -Lieblingsessen / euer Lieblingsurlaubsort?" sind einfache, von jedem schnell zu beantwortende Fragen. Mit der gesprochenen Antwort wird die psychische Hemmschwelle abgebaut, sich während des weiteren Verlaufs zu äußern. Wichtig: Politische, zu persönliche oder zu weitschweifenden Antworten einladende Fragen sollten vermieden werden.

Während der laufenden Besprechung dient der Chat als zweiter Kommunikationskanal, den der Moderator stets im Auge behalten sollte. Muss ein Teilnehmer die Besprechung verlassen oder gibt es technische Probleme, können Teilnehmer diese im Chat kommunizieren, sodass der Moderator darauf reagieren kann. Ein aus technischen Gründen plötzlich verstummter Teilnehmer wird ohne Chat oft einfach nicht bemerkt.

Örtliche Verteilung der Teilnehmer

Der schlechte Ruf virtueller Meetings kommt oft aus der Erfahrung einer schlecht gestalteten Verteilung der Teilnehmer. Auch wenn Teilnehmer aus einem Standort am Meeting teilnehmen, sollte jeder vor seiner eigenen Webcam sitzen und ein eigenes Mikrofon benutzen. Einen Besprechungsraum zu buchen und sich andere Teilnehmer dazuzuholen, baut immer eine Barriere auf, da die Kommunikation im Besprechungsraum naturgemäß besser läuft als jede Skype-Session. So ergibt sich eine Zwei-Klassen-Situation: Die dazugeschalteten Teilnehmer werden durch Diskussionen und Kommentare inhaltlich abgehängt, da diese oft nur im Besprechungsraum selbst zu hören sind. Wird auf die Nachfrage „Was war das? Ich habe das nicht verstanden" geantwortet: „Ach, das war nichts, schon gut", bleibt immer das negative Gefühl, etwas verpasst zu haben.

In einigen Situationen sind diese hybriden Meetings nicht zu vermeiden, sollten dann jedoch besonders sorgfältig moderiert werden. Die dazugeschalteten Teilnehmer müssen zu Beginn des Meetings sämtliche Namen und, falls unbekannt, Rolle und Aufgabe jedes Teilnehmers im Besprechungsraum genannt bekommen. Selbst wenn der ganze Besprechungsraum mit allen Teilnehmern im Video-Stream sichtbar ist, könnten ja noch Personen hinter oder neben der Kamera sitzen. Ist das der Fall, ist unbedingt explizit darauf hinzuweisen.

Neu hinzukommende oder das Meeting verlassende Personen müssen sofort vom Moderator kommentiert werden. Betritt plötzlich ein Vertreter des Kunden eine bisher interne Besprechung, ohne dass der Kollege vom anderen Standort dies mitbekommt, kann es zu erklärungsbedürftigen Aussagen kommen. Die Ausrede „Das war doch eigentlich nur für die Kollegen bestimmt" hilft da nur begrenzt weiter.

Die beschriebenen Nebengespräche im Besprechungsraum im Falle eines hybriden Meetings muss der Moderator rigoros und ohne Rücksicht auf Position und Rolle unterbinden. Die naturgemäß schlechtere Position der Remote-Teilnehmer lässt sich durch einfache Kommunikation verbessern, indem der Moderator dafür sorgt, dass diese Teilnehmer sich bei jeder Frage zuerst äußern dürfen. Das erhöht den wahrgenommenen Wert in der Situation deutlich und sorgt dafür, dass nicht geistig abgeschaltet wird, weil man nicht folgen kann.

Die Präsenz entfernt sitzender Teilnehmer kann durch die wenig bekannte Ambassador-Rule besonders in größeren Meetings stark verbessert werden. Der physisch im Besprechungsraum sitzende Botschafter hat die Aufgabe, die Präsenz entfernt sitzender Kollegen durch seine eigene Anwesenheit zu verstärken. Er stellt selbstständig Verständnisfragen und äußert sich proaktiv im Sinne der Kollegen, die er vertritt. Hierfür beobachtet er den Chat aktiv und lässt sich auch per Sprache Aufträge erteilen: „Marc, kannst du bitte dafür sorgen, dass wir das eben besprochene Thema X nochmal in den Kontext gerückt bekommen? Wir konnten eben nicht folgen und verstehen die Zusammenhänge nicht."

Teilnehmer virtueller Meetings

Auch jeder Teilnehmer virtueller Meetings muss sich auf die Situation einlassen, Selbstdisziplin üben und die Regeln befolgen. Vermutlich am gewöhnungsbedürftigsten ist es, bei besonders großen Meetings alle seine Äußerungen mit dem eigenen Namen zu beginnen: „Hier Stefan. Zu dem eben Gesagten möchte ich noch hinzufügen …"

Eine Verhaltensweise, die der Autor unbemerkt besonders stark aus Meeting-Situationen in den Alltag übernommen hat, ist das Over-Communicating. Übertriebene Mimik und Gestik macht Sprache oft überflüssig. Wird in einem Treffen mit sechs Teilnehmern eine heikle Fragestellung besprochen, stellen drei schüttelnde Köpfe und ein Facepalm eine klare Aussage dar. Over-Communicating bezieht sich jedoch auch auf Gesprochenes. „Stefan nochmal. Habe ich dich richtig verstanden, dass du für X, aber gegen Y bist? Ich möchte nur sicherstellen, dich verstanden zu haben." Solche Sätze lösen zahlreiche Konflikte, bevor sie überhaupt entstehen können.

ELMO

Ein relativ unbekanntes und vermutlich nur in gut eingespielten Teams einsetzbares Konzept ist „ELMO". Jeder Teilnehmer einer

Besprechung hat jederzeit das Recht, laut „ELMO“ zu sagen. Diese Abkürzung für „Enough, let’s move on!“ signalisiert, dass über den aktuellen Punkt genug gesprochen wurde und man nun weitermachen sollte. Das funktioniert auch im Chat sehr gut.

Hinweis: Viele vorgestellte Gedanken finden sich, übersichtlich und schnell erfassbar, auf einem Cheat Sheet der Firma msg David wieder *(siehe Abbildung 3)*. Es steht kostenfrei auf dem GitHub-Account des Unternehmens *[2]* zur Verfügung.

Fazit

Virtuelle Meetings sind das Ereignis, das am meisten mit verteilter Arbeit assoziiert wird. Aufgrund der Häufigkeit solcher Meetings ist es sehr wichtig, sie optimal zu gestalten und die Motivation aller Teilnehmer dauerhaft hochzuhalten. Werkzeuge dazu umfassen die sehr wichtige Rolle des Moderators, der alle Phasen einer Besprechung kennt und sie sinnvoll ausfüllen kann, die sinnvolle örtliche Verteilung der Teilnehmer und auch die Schulung aller Mitarbeiter. Die in diesem Artikel vorgestellten Regeln, Maßnahmen und Ratschläge stellen bereits einen hohen Qualitätsstandard bei virtuellen Treffen sicher. Jedoch kann nichts die Erfahrung eines durch viele Projekte und spannende Situationen gereisten Moderators ersetzen. Je mehr virtuelle Arbeit gelebt wird, desto besser funktioniert sie in aller Regel auch.

Weitere Informationen

[1] *https://www.youtube.com/watch?v=DYu_bGbZiiQ*
[2] *https://github.com/msg-DAVID-GmbH/RemoteWorking/blob/develop/handouts/OnlineMeetingCheatSheet_GER.pdf*

Steven Schwenke
steven.schwenke@msg-david.de

Steven Schwenke ist Software Craftsman und liebt, was er tut. Als Technical Teamlead leitet er ein verteiltes Team und führt regelmäßig Inhouse-Workshops durch. Zusätzlich organisiert er Veranstaltungen wie den HackTalk und hält Vorträge bei Konferenzen. Er teilt seine Erfahrung gern mit anderen und freut sich auf spannende Gespräche.

Reaktive Programmierung mit Java und Spring

Torsten Kohn, Comsysto Reply GmbH

Reaktive Programmierung gewinnt in der Entwicklung von Backend-Systemen immer mehr an Interesse. Der Artikel bietet einen Einblick und zeigt, wie man mit Java und Spring eine reaktive Anwendung schreibt.

Bei der Entwicklung von Software gibt es unterschiedliche Vorgehensweisen und Schwerpunkte. So kann man synchron und blockierend entwickeln *(siehe Abbildung 1)*. Bei diesem Beispiel führt der „main"-Thread eine lang andauernde I/O-Operation aus, was zur Folge hat, dass die Anwendung während der Ausführung der I/O-Operation blockiert ist und auf deren Ergebnis wartet. Der Code dazu dürfte für einen Entwickler einfach umzusetzen sein. Eine Verbesserung ist die Auslagerung der Operationen in neue Threads *(siehe Abbildung 2)*.

In diesem Fall werden neue Threads für die Operationen erzeugt und die Ergebnisse am Ende zusammengeführt. Das ist effektiver, wenn mehrere Operationen notwendig sind, um das Ergebnis zu erzeugen. Schlussendlich wird der „main"-Thread blockiert, um die Ergebnisse zu erhalten. Das Erzeugen dieser Threads ist kostenintensiv, das Warten sowie Zusammenführen kompliziert.

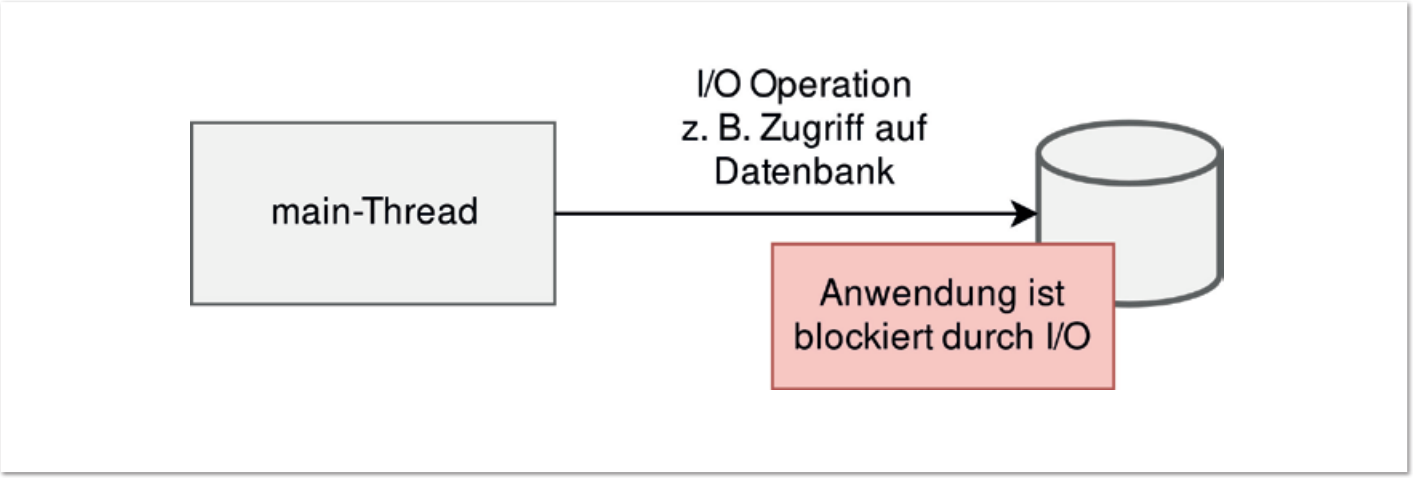


Abbildung 1: „main"-Thread wird durch I/O blockiert

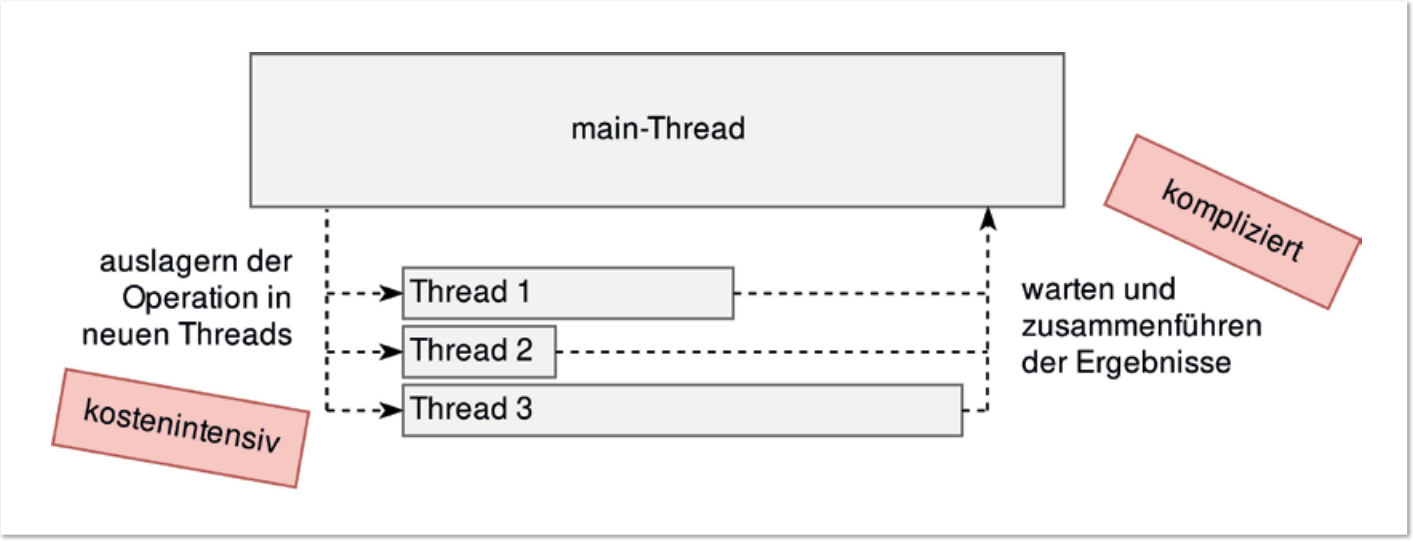


Abbildung 2: Operationen in neue Threads auslagern: asynchron, aber blockierend

Beide gezeigten Ansätze blockieren die Anwendung. Was man hingegen erreichen möchte, ist asynchrone, nicht blockierende Verarbeitung. Daher folgt der dritte Ansatz mit einer Ereignisschleife (Event-Loop, *siehe Abbildung 3*).

Dabei arbeitet ein Thread mehrere Anfragen ab und kann die Verarbeitung dieser Anfragen unterbrechen. Dies führt zu einer verbesserten Ressourcen-Nutzung, da kein Thread im Hintergrund auf Ergebnisse wartet. Es werden somit nur so viele Threads verwendet, wie auch benötigt werden. Implementieren kann man diesen Ansatz in Java unterschiedlich, etwa mit Callbacks, was schnell zu einer Callback-Hölle führt, sodass man den Überblick über die aufgerufenen Funktionen verliert.

Darüber hinaus kann man in Java „Future" (mit Java 8 „CompletableFuture") verwenden, damit die Anwendung asynchron agiert, ohne Callbacks zu verwenden. Ein großer Nachteil ist, dass dies Ergebnis-basiert funktioniert. Das bedeutet, dass beim Erzeugen eines „Compl etableFuture<List<String>>" die gesamte Liste aufgebaut wird (kein Streaming möglich) und man dadurch den Vorteil der asynchronen Verarbeitung verliert. Hier kommt nun reaktive Programmierung ins Spiel, um die Anwendung asynchron und nicht blockierend zu programmieren; und das alles mit Unterstützung des Streaming-API.

Reaktive Programmierung

Reaktive Programmierung (Reactive Programming) ist ein Programmier-Paradigma. Es fokussiert sich auf Datenströme und das Propagieren von Änderungen. Ziel ist es, asynchrone, eventbasierte und nicht blockierende Operationen/Komponenten zu entwickeln. Dieses Vorgehen verhindert Ressourcen-Verschwendung durch auf I/O-Operationen wartende Threads.

Ein reaktives Beispiel-Programm ist Excel: Ändert man den Wert in einer Zelle, dann ändert sich auch der Wert in der Zelle mit der Berechnung. Microsoft hat mit der Reactive-Extentions-Bibliothek (Rx) im .NET-Ökosystem den Ansatz der reaktiven Programmierung gestartet. Die Bibliothek „RxJava" hat diesen Ansatz auf die JVM gebracht, wodurch reaktive Programmierung Einzug in Java hielt. Zwischenzeitlich hat sich eine Initiative gegründet, die eine Spezifikation bereitstellt, um einen Standard für das Verarbeiten von asynchronen Datenströmen mit nicht blockierendem Code anzubieten.

Spezifikation und Manifest

Entwickler von Netflix, Pivotal, Twitter und weiteren Unternehmen arbeiten zusammen an einer Spezifikation für reaktive Programmierung, den sogenannten „Reactive Streams" *[1]*. Das Projekt stellt neben der Spezifikation ein Java-API, ein Technology-Compatibility-

Kit (TCK, zur Validierung der eigenen Implementierung gegen die Spezifikation) sowie Beispiel-Implementierungen bereit. Das Projekt ist unter der Creative-Commons-Zero-Lizenz *[2]* veröffentlicht. Dabei ist im Einzelnen zu prüfen, ob diese Lizenz im Projekt eingesetzt werden darf.

Interessant ist zudem das „reaktive Manifest" *[3]* von Jonas Bonér, Dave Farley, Roland Kuhn und Martin Thompson, das kurz und prägnant die Eigenschaften von reaktiven Systemen darstellt. Die Autoren definieren dazu vier reaktive Qualitäten, die in der Architektur der Softwarekomponente beachtet werden sollen, um reaktive Systeme/Komponenten kombinierbar zu machen:

- Das System ist „responsive" (antwortbereit) und antwortet zeitgerecht. Hier sollen konsistente Antwortzeiten eingehalten werden, damit man Fehler durch Ausbleiben von Antworten erkennen und behandeln kann.
- Bei Ausfällen bleibt das System „resilient" (widerstandsfähig) und ist responsive. Diese Widerstandsfähigkeit wird durch das Replizieren von Funktionalität, Abschirmung der Komponenten, Eindämmung von Fehlern und das Delegieren von Verantwortung erreicht.
- Durch die Widerstandsfähigkeit (siehe oben) bleibt die Komponente „elastic" (elastisch) und kann durch das Replizieren auf eine sich ändernde Last reagieren, weitere Ressourcen beanspruchen oder diese freigeben.
- Zur Sicherstellung der Entkopplung sowie Isolation werden zwischen den Komponenten asynchron Nachrichten übermittelt. Dies wird mit der vierten Qualität „message-driven" (nachrichtenorientiert) beschrieben. Durch die Reactive-Streams-Spezifikation und das reaktive Manifest können reaktive Systeme einheitlich designt werden, ohne das Rad immer wieder neu erfinden zu müssen. Dies fördert Kompatibilität und ermöglicht, Komponenten in Projekten wiederzuverwenden.

Interfaces

Die Spezifikation definiert vier Interfaces, siehe dazu *Listing 1*. Das Publisher-Interface ist die Quelle für die zu verarbeiten Elemente. Damit man diese Daten verarbeiten kann, muss man sich mit einem Subscriber registrieren. *Abbildung 4* zeigt den Zusammenhang

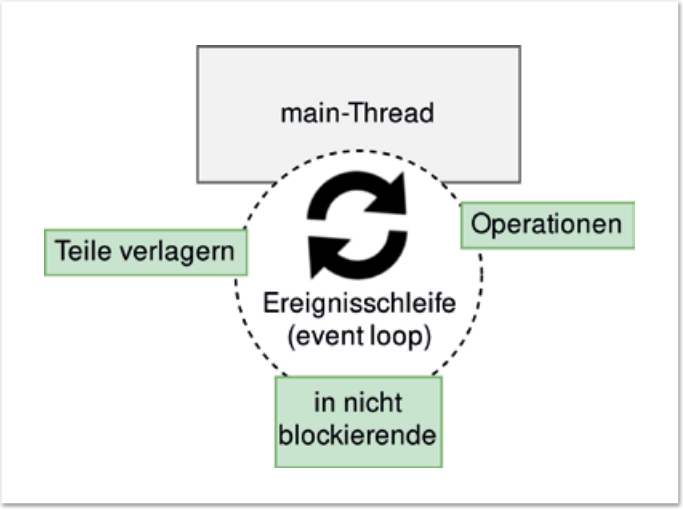


Abbildung 3: Verwendung einer Ereignisschleife zur Abarbeitung der Operationen (Operationen in nicht blockierende Teile verlagern)

zwischen den beiden Interfaces. Damit der Publisher Daten bereitstellt, muss sich mindestens ein Subscriber registrieren. Passiert dies nicht, werden keine Daten vom Publisher verarbeitet. Sobald ein Subscriber sich registriert hat, benachrichtigt der Publisher ihn über vorhandene Daten und sendet diese an den Subscriber.

Zwischen Publisher und Subscriber sitzt ein Subscription-Objekt *(siehe Listing 1)*. Es steuert den Datenfluss. Erhält der Subscriber zu viele Daten vom Publisher, kann er durch die Subscription die Menge reduzieren. Genauso kann der Subscriber die Menge der Daten erhöhen. Dies wird in diesem Zusammenhang „Backpressure" genannt und soll den Subscriber vor Überlastung schützen *(siehe Abbildung 4)*. Zusätzlich gibt es das Processor-Interface, bei dem sich die Implementierung zugleich als Publisher und Subscriber verhält.

Implementierungen für Java-Entwickler

RxJava wurde in der Einführung schon erwähnt als sogenannte „Reactive Extensions" für die JVM. Seit Version 2 verwendet RxJava das API der Reactive Streams und hält sich an die Spezifikation. Eine weitere Implementierung der Reactive-Streams-Interfaces bietet das Project „Reactor" *[4]* von Pivotal.

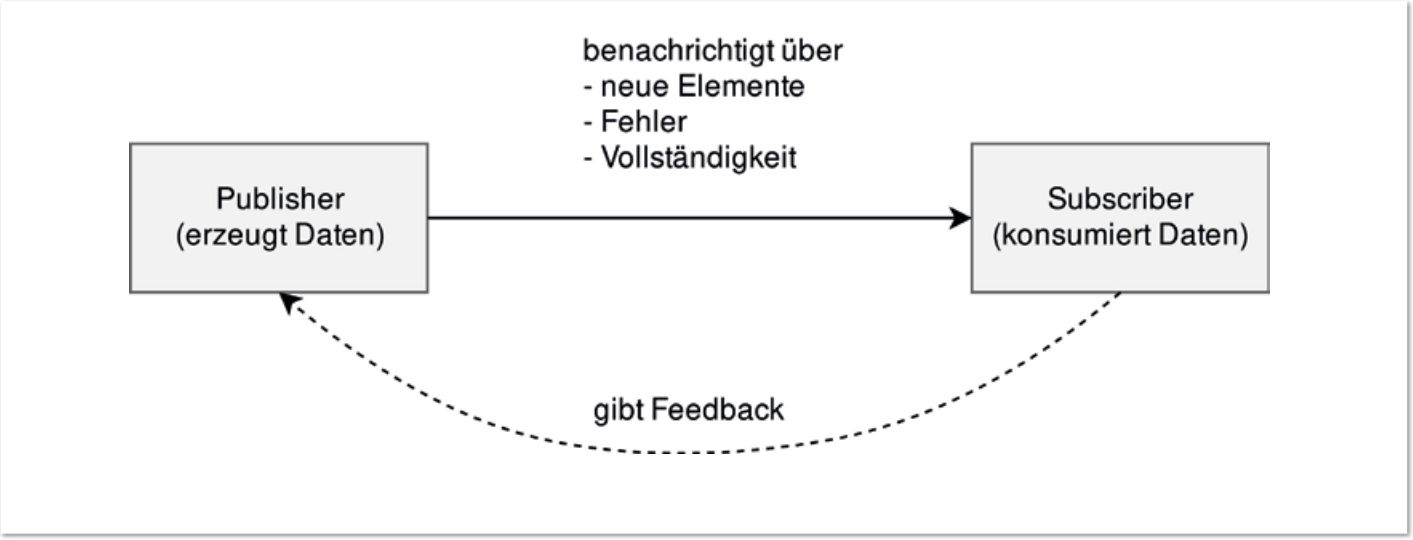


Abbildung 4: Zusammenhang zwischen Publisher und Subscriber

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}

public interface Subscription {
    public void request(long n);
    public void cancel();
}

public interface Processor<T, R> extends
Subscriber<T>, Publisher<R> {
}
```

Listing 1: Interfaces aus der Reactive-Streams-Spezifikation

```
// Imports, Konstruktoren sowie GETTER und SETTER fehlen
zur Übersicht
@Document
public class Todo {
    @Id
    private String id;
    private String title;
    private boolean completed;
}
```

Listing 2: Modell Todo

```
public interface TodoRepository extends
ReactiveMongoRepository<Todo, String> {
}
```

Listing 3: Repository für Todo-Objekte

Beide Projekte bieten vorgefertigte Subscriber- und Publisher-Implementierungen, um die Entwicklung von reaktiven Applikationen zu erleichtern. Mit Java 9 sind die Interfaces in das Flow-API übernommen worden. Hier muss man als Entwickler die Interfaces selbst implementieren und sich um die Einhaltung der Spezifikation kümmern, da es keine vorgefertigten Implementierungen gibt.

Reaktiv mit Spring

Nachdem das Grundgerüst für reaktive Programmierung geschaffen ist, nun das Beispiel-Projekt mit Spring Boot. Spring bietet über die Webseite *„https://start.spring.io“* einen einfachen Projekt-Generator für Spring-Boot-Applikationen *(siehe Abbildung 5)*. Die Abhängigkeit „Reactive Web“ beinhaltet das Projekt „Reactor“, das als Implementierung für die reaktive Programmierung dient.

Das generierte Projekt ist der Einstieg für die nächsten Beispiele. Wichtig ist, dass man Version 2 oder höher bei Spring Boot verwendet sowie die Abhängigkeit zu Reactive Web und Reactive MongoDB hinzufügt, um in die Vorzüge der reaktiven Programmierung zu kommen.

Als Beispiel-Anwendung dient ein Todo-Webservice, der über REST angesprochen wird und die Daten in einer MongoDB speichert. Das Projekt zeigt auf, was Spring an Implementierungen für eine reaktive Anwendung bereitstellt und welche für das Beispiel genutzt werden. Das Projekt ist auf GitLab veröffentlicht *[5]* und im Artikel nur in Ausschnitten gezeigt. *Listing 2* zeigt das Modell „Todo“, ein einfaches POJO. Die Klasse verwendet keine speziellen Typen für die reaktive Programmierung und dient nur zur Datenhaltung. Die beiden Annotationen „@Document“ und „@Id“ dienen dem Datenbank-Zugriff.

Methoden wie „findAll(), save(S entity)“ oder „findById(ID id)“ werden vom Interface geerbt; in diesem Beispiel stehen das Generic „S“ für „Todo“ sowie „ID“ für „String“. Der Namenszusatz „Reactive“ lässt erahnen, dass sich das Repository anders verhalten wird. Man kann als Alternative vom Interface „MongoRepository“ erben, das dann keinen reaktiven Support bietet. Im Beispielprojekt wird das Repository durch einen Service gekapselt. Da keine Logik im Service vorhanden ist, wird der Service hier nicht vorgestellt.

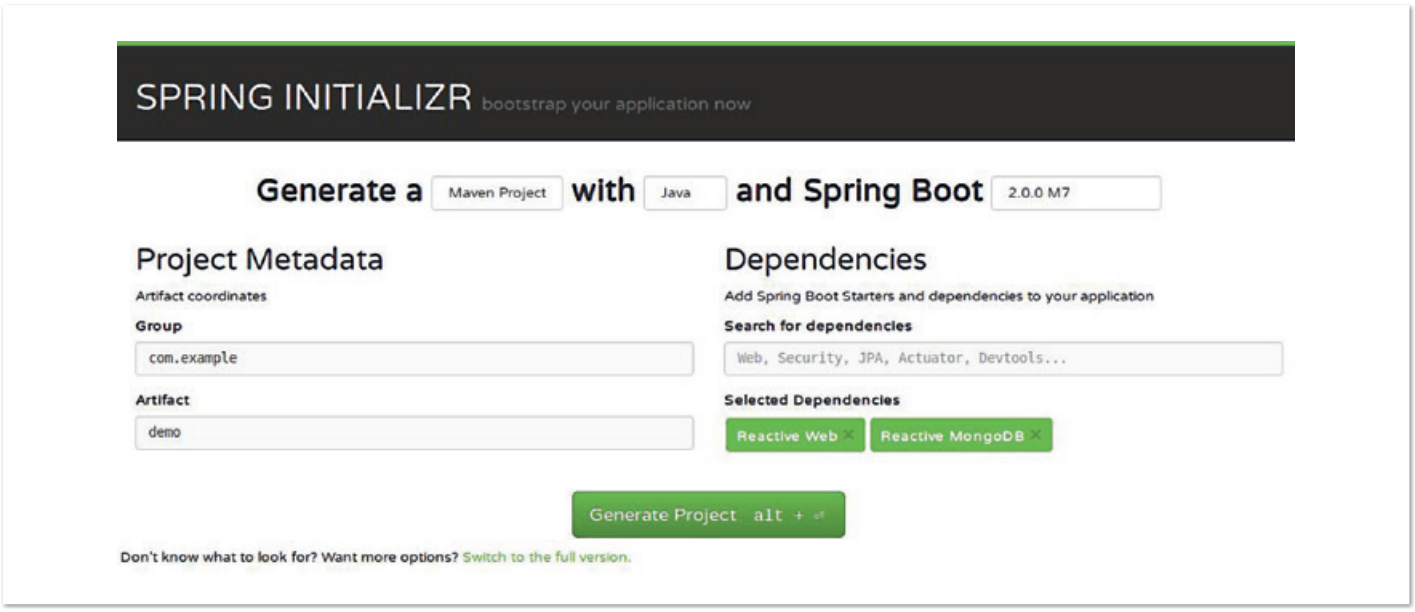


Abbildung 5: Spring-Boot-Projekt-Generator (siehe „https://start.spring.io“)

Listing 4 zeigt einen Ausschnitt des REST-Controllers mit den „GET"-Methoden. Die Annotationen „@RestController", „@RequestMapping" und „@GetMapping" dienen dazu, die Klasse zu markieren, damit Spring die Klasse als Controller erkennt und zusätzlich das Mapping für die verschiedenen REST-Methoden auflösen kann.

Die Annotationen sind Entwicklern bekannt, die mit Spring arbeiten. Neu hingegen sind die Rückgabewerte „Flux<Todo>" und „Mono<Todo>". Hierbei handelt es sich um Publisher-Implementierungen aus dem Project Reactor. Vom REST-Controller bis zum Repository werden die Typen „Flux" und „Mono" durchgängig verwendet. Flux ist ein Publisher, der 0 bis n Elemente zurückliefert, Mono hingegen ist ein Publisher, der nur 0 bis 1 Element liefert. Beide Klassen implementieren das Publisher-Interface von Reactive Streams. Durch Nutzung des „ReactiveMongoRepository<T, ID>"-Interface sowie die durchgängige Nutzung von Flux und Mono erhält der Entwickler eine reaktive Anwendung ohne großen Mehraufwand.

Testen des Controllers

Das Spring-Framework sowie Project Reactor bieten Test-Bibliotheken an, um reaktive Programmierung zu testen; *Listing 5* zeigt die zwei Abhängigkeiten, *Listing 6* einen Test des REST-Controllers mit dem „WebTestClient". Die Klasse ist speziell zum Testen von reaktiven Programmen, da es ein nicht blockierender reaktiver Client ist. Der Test überprüft „MediaType", HTTP-Statuscode und „RequestBody".

Fallstricke bei reaktiver Programmierung

Eine Anwendung besitzt meist eine Datenbank-Schnittstelle und hier bestehen Defizite zu den vorhandenen reaktiven Datenbank-Treibern. Zurzeit gibt es offiziell nur Implementierungen für MongoDB, Redis und Casandra. Möchte man dagegen JDBC oder JPA einsetzen, hat man blockierenden Zugriff auf die Datenbank. Ob es für JDBC und JPA eine reaktive Implementierung geben wird, wird sich zeigen.

Fazit

Der Artikel gibt eine Einführung in die reaktive Programmierung und zeigt, dass es mit wenig Aufwand möglich ist, die Vorteile zu nutzen. Die reaktive Programmierung ist allerdings nicht das Allheilmittel; es ist im Projekt zu überprüfen, ob die Rahmenbedingungen erfüllt sind – dies fängt bei der Auswahl des Datenbank-Systems an und endet beim Entwickler-Wissen.

Vorteile bietet die reaktive Programmierung durch die Zusammensetzbarkeit der Komponenten (siehe dazu das reaktive Manifest) und den effektiven Ressourcen-Einsatz. Die nächsten Monate werden zeigen, wie die Java-Community reaktive Programmierung aufnimmt und vor allem wie das Flow-API aus Java 9 eingesetzt wird.

Quellen

[1] *http://www.reactive-streams.org*
[2] *https://creativecommons.org/publicdomain/zero/1.0/deed.de*
[3] *https://www.reactivemanifesto.org/de*
[4] *https://projectreactor.io*
[5] *https://gitlab.com/torstenkohn/todo-list-backend*


```
@RestController
@RequestMapping("/api/todos")
public class TodoRestController {
    @Autowired
    private final TodoService service;
    @GetMapping
    public Flux<Todo> all() {
        return this.service.all();
    }

    @GetMapping("/{todoId}")
    public Mono<Todo> byId(@PathVariable String to-
doId) {
        return this.service.byId(todoId);
    }
}
```

Listing 4: REST-Controller

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

Listing 5: Abhängigkeiten für Testing

```
@Test
public void testTodosForId() {
    WebTestClient client = WebTestClient
            .bindToController(new
TodoRestController(service))
            .build();

    this.client.get().uri("/api/todos/t_1")
            .accept(MediaType.APPLICATION_JSON_UTF8)
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$.title", "Wohnung putzen").
exists()
            .jsonPath("$.completed", "true").exists();
}
```

Listing 6: Test-Methode für den REST-Controller

Torsten Kohn
t.kohn@reply.de

Torsten Kohn ist Software Engineer bei der Comsysto Reply GmbH in München. Sein Schwerpunkt liegt in der Entwicklung von Software-Lösungen mithilfe von Spring und Java. Er ist stets auf der Suche nach neuen Konzepten und Vorgehen in der Entwicklung von Software mit Java und häufig auf Meetups rund um Software-Entwicklung im Raum München anzutreffen.

Jenkins

Coding Continuous Delivery – hilfreiche Werkzeuge für die Jenkins-Pipeline

Johannes Schnatterer, Triology GmbH

Nachdem in den letzten beiden Ausgaben Grundlagen und Performance von Jenkins-Pipelines thematisiert wurden, beschreibt dieser Artikel nützliche Werkzeuge und Methoden: Mit Shared Libraries lassen sich die Wiederverwendung über verschiedene Jobs hinweg und Unit Testing des Pipeline-Codes realisieren. Außerdem bietet der Einsatz von Containern mittels Docker auch hier seine Vorzüge.

Nachfolgend sind die Pipeline-Beispiele aus den ersten beiden Artikeln sukzessive erweitert, um die Features der Pipeline zu zeigen. Dabei werden die Änderungen jeweils in „declarative"- und in „scripted"-Syntax realisiert. Den aktuellen Stand jeder Erweiterung kann man bei GitHub *[1]* nachverfolgen und ausprobieren. Hier gibt es für jeden Abschnitt unter der in der Überschrift genannten Nummer jeweils für „declarative" und „scripted" einen Branch, der das vollständige Beispiel umfasst. Das Ergebnis der Builds jedes Branch lässt sich außerdem direkt auf der Jenkins-Instanz *[2]* einsehen.

Wie in den ersten beiden Teilen werden auch in diesem die Features des Jenkins-Pipeline-Plug-ins anhand eines typischen

Java-Projekts gezeigt. Als Beispiel dient damit auch hier der „kitchensink"-Quickstart von WildFly. Da dieser Artikel auf den Beispielen aus dem ersten Artikel aufbaut, setzt sich die Nummerierung aus dem ersten Teil fort. Dort wurden mit simpler Pipeline, eigenen Steps, Stages, Fehlerbehandlung und Properties/Archivierung, Parallelisierung und Nightly Builds bereits sieben Beispiele gezeigt.

Shared Libraries

In den Beispielen aus dieser Artikelserie gibt es bereits einige selbst geschriebene Steps wie beispielsweise „mvn()" oder „maillfStatusChanged()". Sie sind nicht projektspezifisch und könnten aus dem „Jenkinsfile" ausgelagert und damit auch in anderen Projekten wiederverwendet werden. Für das Auslagern gibt es bei Jenkins-Pipelines derzeit zwei Möglichkeiten:

- **„load"-Step**
  Lädt eine Groovy-Skript-Datei aus dem Jenkins-Workspace (also dem gleichen Repository) und evaluiert sie; damit lassen sich dynamisch weitere Steps nachladen
- **Shared Libraries**
  Erlauben Einbinden von externen Groovy-Skripten und -Klassen

Der „load"-Step unterliegt gewissen Einschränkungen:

- Es können nur Groovy-Skripte und keine Klassen geladen werden *[3]*. Dadurch lassen sich in diesen Skripten beispielsweise nicht ohne Weiteres zusätzliche Klassen laden und Vererbung ist nicht möglich. Damit die Skripte in der Pipeline einsetzbar sind, muss jedes Skript mit „return this;" enden.
- Man kann nur Dateien aus dem Workspace verwenden. Dadurch ist keine Wiederverwendung über Projekte möglich.
- Die über diesen Step geladenen Skripte werden in dem im ersten Artikel beschriebenen „Replay"-Feature nicht angezeigt. Dadurch sind sie schwerer zu entwickeln und zu debuggen.

Shared Libraries unterliegen diesen drei Einschränkungen nicht, wodurch sie wesentlich flexibler sind. Deshalb ist ihre Verwendung nachfolgend näher beschrieben: Aktuell muss eine Shared Library aus einem eigenen Repository geladen werden. Das Laden aus dem zu bauenden Repository ist momentan noch nicht möglich, wird es aber wahrscheinlich in Zukunft sein *[4]*. Dadurch ist es zukünftig möglich, das „Jenkinsfile" auf verschiedene Klassen/Skripte aufzuteilen. Dies erhöht die Wartbarkeit und schafft die Möglichkeit für das Schreiben von Unit-Tests. Außerdem ist dies für die Entwicklung von Shared Libraries hilfreich, weil diese in ihrem eigenen „Jenkinsfile" verwendet werden können. Das Repository jeder Shared Library muss eine bestimmte Verzeichnisstruktur aufweisen:

- „src" enthält Groovy-Klassen
- „vars" enthält Groovy-Skripte und Dokumentation
- „resources" enthält weitere Dateien

Zudem sind ein „test"-Verzeichnis für Unit-Tests und ein eigener Build empfehlenswert. Um die Komplexität des „Jenkinsfile" aus den Beispielen zu reduzieren und die Funktionalität für andere Projekte wiederverwendbar zu machen, wird im Folgenden exemplarisch ein Step in eine Shared Library ausgelagert. Für den Step „mvn" legt man beispielsweise im Repository der Shared Library im Verzeichnis „vars" eine Datei „mvn.groovy" an *(siehe Listing 1)*. Diese enthält die aus dem ersten Teil dieser Artikelserie bekannte Methode.

Im Groovy-Skript in *Listing 1* wird diese Methode allerdings nach Groovy-Konvention „call()" genannt. Technisch gesehen legt Jenkins für alle „.groovy"-Dateien im Verzeichnis „vars" eine globale Variable an und benennt sie entsprechend dem Dateinamen. Ruft man diese Variable jetzt mit dem Call-Operator „()" auf, wird implizit deren Methode „call()" aufgerufen *[5]*. Da bei Groovy die Klammern beim Aufruf optional sind, bleibt der Aufruf der Steps in „scripted"- und „declarative"-Syntax wie bisher, beispielsweise „mvn 'test'".

Um die Shared Library in der Pipeline zu verwenden, gibt es mehrere Möglichkeiten. Zunächst müssen die Shared Libraries in Jenkins bekannt gemacht sein. Dazu gibt es folgende Möglichkeiten der Definition von Shared Libraries:

- **Global**
  Muss durch einen Jenkins-Administrator in der Jenkins-Konfiguration eingestellt sein. Dort definierte Shared Libraries sind in allen Projekten verfügbar und werden als vertrauenswürdig behandelt; sie dürfen also alle Groovy-Methoden, interne Jenkins-APIs etc. ausführen. Hier ist also Vorsicht geboten. Man kann dies allerdings auch nutzen, um beispielsweise die unter Nightly Builds beschriebenen Aufrufe zu kapseln, für die sonst Script Approval notwendig wäre.
- **Folder/Multibranch**
  Kann von entsprechend berechtigten Projekt-Mitgliedern für eine Gruppe von Build-Jobs eingestellt werden. Dort definierte Shared Libraries gelten nur für zugehörige Build-Jobs und werden nicht als vertrauenswürdig behandelt. Das heißt, sie laufen in der Groovy-Sandbox – wie normale Pipelines auch.
- **Automatisch**
  Plug-ins wie das Pipeline-GitHub-Library-Plug-in *[6]* erlauben das automatische Definieren von Libraries innerhalb von Pipelines, die in einem GitHub Organization Folder definiert sind. Dadurch können Shared Libraries ohne vorherige Definition im Jenkins direkt in „Jenkinsfile" verwendet werden. Auch diese Shared Libraries laufen in der Groovy-Sandbox.

```
def call(def args) {
    def mvnHome = tool 'M3'
    def javaHome = tool 'JDK8'
    withEnv(["JAVA_HOME=${javaHome}", "PATH+MAVEN=${mvnHome}/bin:${env.JAVA_HOME}/bin"]) {
        sh "${mvnHome}/bin/mvn ${args} --batch-mode -V -U -e -Dsurefire.useFile=false"
    }
}
```

Listing 1

```
@Test
void mvn() {
    def shParams = ""
    helper.registerAllowedMethod("tool", [String.class], { paramString -> paramString })
    helper.registerAllowedMethod("sh", [String.class], { paramString -> shParams = paramString })
    helper.registerAllowedMethod("withEnv", [List.class, Closure.class], { paramList, closure ->
        closure.call()
    })
    def script = loadScript('vars/mvn.groovy')
    script.env = new Object() {
        String JAVA_HOME = "javaHome"
    }
    script.call('clean install')
    assert shParams.contains('clean install')
}
```

Listing 2

In unserem Beispiel bietet sich die Verwendung des GitHub-Branch-Source-Plug-ins an, da es bei GitHub verfügbar und deshalb keine weitere Konfiguration in Jenkins notwendig ist. Sowohl in den „scripted"- als auch in den „declarative"-Syntax-Beispielen können die ausgelagerten Steps (beispielsweise „mvn") durch das Einbinden der Shared Library in der ersten Zeile des Skripts mit „@Library('github.com/triologygmbh/jenkinsfile@e00bbf0') _" definiert werden. Dabei ist „github.com/triologygmbh/jenkinsfile" der Name der Shared Library und nach dem „@" steht die Version, in diesem Fall ein Commit Hash. Hier könnte man auch einen Branch- oder Tag-Namen nehmen.

Es empfiehlt sich, einen definierten Stand (Tag oder Commit anstelle von Branches) zu verwenden, um deterministisches Verhalten zu gewährleisten. Da die Shared Library in jedem Build neu aus dem Repository abgerufen wird, besteht sonst die Gefahr, dass eine Änderung daran ohne Änderung des eigentlichen Pipeline-Skripts oder -Codes Auswirkung auf den nächsten Build hat. Dies kann zu unerwarteten Ergebnissen führen, deren Ursache schwer zu finden ist. Alternativ kann man Libraries dynamisch (mit dem „library"-Step) laden. Diese können dann erst nach dem Aufruf des Steps verwendet werden.

Wie erwähnt, kann man in Shared Libraries außer Skripten auch Klassen anlegen (im „src"-Ordner). Liegen diese in Packages, können sie über Import-Statements nach der „@Library"-Annotation angegeben werden. Diese Klassen können in „scripted"-Syntax überall in der Pipeline instanziiert werden, in „declarative"-Syntax nur innerhalb des „script"-Steps. Ein Beispiel dafür ist die Shared Library des Cloudogu EcoSystem *[7]*.

Außerdem bieten Shared Libraries die Möglichkeit, Unit-Tests zu schreiben. Für Klassen ist dies häufig mit Groovy-Bordmitteln möglich *[7]*. Für Skripte bietet sich die Verwendung von „JenkinsPipelineUnit" *[8]* an. Mit diesem Framework kann man Skripte laden und einfach Mocks der eingebauten Pipeline-Steps definieren. *Listing 2* zeigt, wie ein Test für den in *Listing 1* beschriebenen Step aussehen könnte.

Dort wird überprüft, ob der übergebene Parameter korrekt an den „sh"-Step weitergereicht wird. Die Variable „helper" wird dabei der Test-Klasse durch das Framework über Vererbung bereitgestellt. Wie man in *Listing 2* sieht, kommt hier viel Mocking zum Einsatz: etwa die „tool"- und „withEnv"-Steps sowie die globale Variable „env". Daran zeigt sich schon, dass der Unit-Test nur die grundlegende Logik prüft und natürlich nicht den Test in einer echten Jenkins-Umgebung ersetzt.

Diese Integrationstests kann man derzeit noch nicht automatisiert ausführen. Für die Entwicklung von Shared Libraries bietet sich das im ersten Artikel beschriebene „Replay"-Feature an: Neben dem „Jenkinsfile" kann man hier auch temporär die Shared Library verändern und ausführen. Dadurch vermeidet man viele unnötige Commits ins Repository der Shared Library. Dieser Tipp ist auch in der umfangreichen Dokumentation zu Shared Libraries beschrieben *[9]*. Zusätzlich zum Auslagern von Steps kann man ganze Pipelines in Shared Libraries definieren *[10]* und so beispielsweise seine Stages standardisieren. Zum Abschluss des Themas noch einige Open Source Shared Libraries:

- Offizielle Beispiele mit Shared Library und Jenkinsfile *[11]*; enthält Klassen und Skripte
- Shared Library, die von Docker Inc. für die Entwicklung verwendet wird *[12]*; enthält Klassen und Skripte
- Shared Library, die vom Firefox Test Engineering verwendet wird *[13]*; enthält Skripte mit Unit Tests und Groovy Build
- Shared Library des Cloudogu EcoSystem *[7]*; enthält Klassen und Skripte mit Unit-Tests und Maven Build

Docker

Durch den Einsatz von Docker in Jenkins-Builds lassen sich Build- und Test-Umgebung vereinheitlichen und Anwendungen deployen. Außerdem können, wie bereits im ersten Artikel dieser Serie erwähnt, durch die Isolierung Port-Konflikte bei parallelen Builds verhindert werden. Ein weiterer Vorteil ist, dass weniger Konfiguration in Jenkins notwendig ist. Auf Jenkins muss nur Docker bereitgestellt werden. Die Pipelines können dann benötigte Tools (Java, Maven, Node.js, PaaS-CLIs etc.) einfach per Docker-Image beziehen.

Damit man in Pipelines Docker nutzen kann, muss natürlich ein Docker-Host verfügbar sein. Dies ist ein Infrastruktur-Thema, das außerhalb von Jenkins zu lösen ist. Auch unabhängig von Docker ist es empfehlenswert, in Produktion den Build-Executor getrennt vom Jenkins-Master zu betreiben, um die Last zu verteilen und Reaktionszeiten der Jenkins-Web-Anwendung nicht durch Builds zu verlangsamen. Das gilt auch bei der Bereitstellung von Docker auf den Build-Executors: Der Docker-Host des Masters (falls vorhanden) sollte getrennt vom Docker-Host der Build-Executors sein. Auch

```
pipeline {
    agent {
        docker {
            image 'maven:3.5.0-jdk-8'
            label 'docker'
        }
    }
    //...
}
```

Listing 3

```
node('docker') {
    // ...
    docker.image('maven:3.5.0-jdk-8').inside {
            // ...
     }
}
```

Listing 4

dies stellt sicher, dass die Jenkins-Web-Anwendung unabhängig von den Builds reaktionsfreudig bleibt. Außerdem bietet die Trennung der Hosts zusätzliche Sicherheit, da im Falle von Container-Breakouts *[14]* kein Zugriff auf den Jenkins-Host möglich ist.

Wenn man einen speziellen Build-Executor mit Docker aufsetzt, ist es empfehlenswert, darin auch direkt den Docker-Client zu installieren und im „PATH" verfügbar zu machen. Alternativ kann man den Docker-Client auch als Tool in Jenkins installieren. Dieses Tool muss dann (wie Maven und JDK in den Beispielen im ersten Artikel dieser Serie) explizit in der Pipeline-Syntax angegeben sein. Das ist derzeit jedoch nur in „scripted"- und nicht mit „declarative"-Syntax möglich *[15]*.

Sobald Docker eingerichtet ist, bietet die „declarative"-Syntax die Möglichkeit, entweder die gesamte Pipeline oder einzelne Stages innerhalb eines Docker-Containers auszuführen. Das dem Container zugrunde liegende Image kann entweder aus einer Registry gezogen *(siehe Listing 3)* oder aus einem „Dockerfile" gebaut werden.

Durch die Verwendung des „docker"-Parameters in der „agent"-Section wird die gesamte Pipeline innerhalb eines Containers ausgeführt, der aus dem angegebenen Image erstellt wird. Das in *Listing 3* verwendeten Image sorgt dafür, dass die Executables von Maven und des JDK im „PATH" bereitgestellt werden. Man kann damit hier ohne weitere Konfiguration von Tools in Jenkins (wie Maven und JDK in den Beispielen im ersten Artikel dieser Serie) beispielsweise den Step „sh 'mvn test'" ausführen.

Das in *Listing 3* gesetzte Label bezieht sich in diesem Fall auf den Build-Executor. Dies bewirkt, dass die Pipelines nur auf Build-Executors ausgeführt werden, bei denen ein Label „docker" gesetzt ist. Insbesondere, wenn man verschiedene Build-Executors hat, ist diese Best-Practice hilfreich. Wird diese Pipeline auf einem Build-Executor ausgeführt, auf dem kein Docker-Client im „PATH" verfügbar ist, scheitert dieser. Ist jedoch kein Build-Executor mit dem Label verfügbar, bleibt der Build in der Queue.

Ein weiterer Punkt, den man bei in Containern ausgeführten Builds oder Steps bedenken muss, ist das Speichern von Daten außerhalb der Container. Da jeder Build in einem neuen Container ausgeführt wird, sind die darin enthaltenen Daten bei der nächsten Ausführung nicht mehr verfügbar. Jenkins sorgt zwar dafür, dass der Workspace als Working-Directory in den Container gemountet wird. Dies geschieht jedoch beispielsweise nicht für das lokale Maven-Repository. Während der in den Beispielen bisher verwendete „mvn"-Step (basierend auf den Jenkins-Tools) das Maven-Repository des Build-Executor verwendet, legt der Docker-Container ein Maven-Repository im Workspace jedes Builds an. Das kostet zwar etwas mehr Speicher und der jeweils erste Build wird langsamer. Dafür schließt man unerwünschte Seiteneffekte aus, etwa wenn zwei gleichzeitig laufende Builds eines Maven-Multi-Module-Projekts sich gegenseitig Snapshots im gleichen lokalen Repository überschreiben.

Wenn trotzdem das Repository des Build-Executor verwendet werden soll, sind einige Anpassungen am Docker-Image notwendig *[16]*. Was man eher vermeiden sollte, ist das Ablegen des lokalen Maven-Repository im Container. Dies würde dazu führen, dass alle Dependencies bei jedem Build neu aus dem Internet geladen werden, was wiederum die Dauer jedes Builds deutlich verlängern würde. Das in *Listing 3* in „declarative"-Syntax beschriebene Verhalten lässt sich auch in „scripted"-Syntax realisieren *(siehe Listing 4)*.

Wie in „declarative"-Syntax *(Listing 3)* lassen sich auch in „scripted"-Syntax Build-Executors durch Label auswählen. Dort *(Listing 4)* erfolgt dies als Parameter des „node"-Steps. Hier wird Docker über eine globale Variable angesprochen *[17]*. Diese Variable bietet noch weitere Features, unter anderem:

- Man kann bestimmte Docker-Registries verwenden (unter anderem für Continuous Delivery auf Kubernetes hilfreich, was im nächsten Teil dieser Serie beschrieben wird)
- Man kann einen bestimmten Docker-Client (definiert als Jenkins-Tool, wie oben beschrieben) verwenden
- Man kann Images bauen, mit Tags versehen und in eine Registry schieben
- Man kann Container starten und stoppen

Die „docker"-Variable unterstützt nicht immer die neuesten Docker-Features; beispielsweise fehlt das Bauen von Multi-Stage-Docker-

```
def call(def args) {
    docker.image('maven:3.5.0-jdk-8').inside {
        sh "mvn ${args} --batch-mode -V -U -e -Dsurefire.useFile=false"
    }
}
```

Listing 5

Images *[18]*. In diesem Fall kann man auf den CLI-Client von Docker zurückgreifen, beispielsweise mit „sh 'docker build …'".

Beim Vergleich der Listings 3 und 4 zeigt sich deutlich der Unterschied zwischen beschreibender („declarative") und imperativer („scripted") Syntax. Statt deklarativ am Anfang anzugeben, welcher Container zu verwenden ist, wird imperativ festgelegt, ab wo etwas in diesem Container auszuführen ist. Damit ist man auch flexibler: Während man in „declarative"-Syntax darauf beschränkt ist, die ganze Pipeline oder einzelne Stages in Containern auszuführen, kann man in „scripted"-Syntax beliebige Abschnitte in Containern ausführen.

Wie schon mehrfach erwähnt, kann man in „declarative"-Syntax allerdings auch innerhalb des „script"-Steps „scripted"-Syntax ausführen oder eigene Steps aufrufen, die in „scripted"-Syntax geschrieben sind. Das Auslagern wird nachfolgend genutzt, um den „mvn"-Step in der Shared Library *(Listing 1)* von Jenkins-Tools auf Docker umzustellen *(siehe Listing 5)*. Nach der Aktualisierung der Shared Library läuft jeder „mvn"-Step in den „scripted"- und „declarative"-Pipeline-Beispielen dann ohne weitere Änderung in einem Docker-Container. Zum Abschluss noch ein fortgeschrittenes Docker-Thema. Die „scripted"-Pipeline-Syntax lädt fast dazu ein, Docker-Container zu schachteln, also „Docker in Docker" auszuführen. Dies ist nicht ohne Weiteres möglich, da in einem Docker-Container zunächst kein Docker-Client verfügbar ist. Allerdings lassen sich mit „docker.withRun()" *[19]* mehrere Container gleichzeitig ausführen.

Es gibt jedoch auch Builds, die Docker-Container starten, beispielsweise mit dem Docker-Maven-Plug-in *[20]*. Damit können unter anderem Test-Umgebungen hochgefahren oder UI-Builds ausgeführt werden. Für diese Builds muss man tatsächlich „Docker in Docker" verfügbar machen. Dafür ist es jedoch nicht naheliegend, einen weiteren Docker-Host in einem Docker-Container zu starten, auch wenn das möglich wäre *[21]*. Stattdessen kann man den Docker-Socket des Build-Executor in den Docker-Container des Builds mounten.

Auch bei diesem Vorgehen sollte man sich jedoch gewisser sicherheitsrelevanter Einschränkungen bewusst sein *[22]*. Hier ist die oben erwähnte Trennung des Docker-Hosts des Masters von den Docker-Hosts der Build-Executors noch wichtiger. Damit der Zugriff auf den Socket möglich ist, sind außerdem einige Anpassungen am Docker-Image notwendig. So muss der User, mit dem der Container gestartet wird, in der „docker"-Gruppe sein, um Zugriff auf den Socket zu bekommen. Dazu müssen im Image User und Gruppe erzeugt werden *[16]*.

Fazit und Ausblick

Dieser Artikel beschreibt zum einen, wie man die Wartbarkeit der Pipeline durch Auslagerung von Code in eine Shared Library verbessern kann. Dieser Code ist dann auch wiederverwendbar und seine Qualität lässt sich durch Unit-Tests prüfen. Außerdem wird Docker als Werkzeug vorgestellt, mit dem Pipelines in einheitlicher Umgebung isolierter und unabhängiger von der Konfiguration der jeweiligen Jenkins-Instanz ausgeführt werden können. Diese nützlichen Werkzeuge schaffen die Grundlage für den Artikel in der nächsten Ausgabe, mit dem die Continuous-Delivery-Pipeline vollendet wird.

Weitere Informationen

[1] Jenkinsfile Repository GitHub: *https://github.com/triologygmbh/jenkinsfile*
[2] Triology Open Source Jenkins: *https://opensource.triology.de/jenkins/job/triologygmbh-github/job/jenkinsfile*
[3] Groovy Scripts vs. Classes: *http://docs.groovy-lang.org/latest/html/documentation/index.html#_scripts_versus_classes*
[4] cps-global-lib-plugin Pull Request 37: *https://github.com/jenkinsci/workflow-cps-global-lib-plugin/pull/37*
[5] Groovy Call Operator: *http://docs.groovy-lang.org/latest/html/documentation/#_call_operator*
[6] Pipeline GitHub Library Plug-in: *https://wiki.jenkins.io/display/JENKINS/Pipeline+GitHub+Library+Plugin*
[7] Cloudogu ces-build-lib: *https://github.com/cloudogu/ces-build-lib*
[8] JenkinsPipelineUnit: *https://github.com/jenkinsci/JenkinsPipelineUnit*
[9] Jenkins Shared-Libraries: *https://jenkins.io/doc/book/pipeline/shared-libraries*
[10] Standard-Build-Beispiel: *https://github.com/jenkinsci/pipeline-examples/blob/master/global-library-examples/global-function/standardBuild.groovy*
[11] Shared Library Demo: *https://github.com/jenkinsci/workflow-aggregator-plugin/tree/master/demo*
[12] Shared Library Docker: *https://github.com/docker/jenkins-pipeline-scripts*
[13] Shared Library Firefox: *https://github.com/mozilla/fxtest-jenkins-pipeline*
[14] Security Concerns when using Docker: *https://www.oreilly.com/ideas/five-security-concerns-when-using-docker*
[15] Pipeline-Syntax-Tools: *https://jenkins.io/doc/book/pipeline/syntax/#tools*
[16] „Cloudogu-ces-build-lib"-Docker: *https://github.com/cloudogu/ces-build-lib/blob/develop/src/com/cloudogu/ces/cesbuildlib/Docker.groovy*
[17] Global Variable Reference Docker: *https://opensource.triology.de/jenkins/pipeline-syntax/globals#docker*
[18] Jenkins Issue 44609 : *https://issues.jenkins-ci.org/browse/JENKINS-44609*
[19] Pipeline Docker: *https://jenkins.io/doc/book/pipeline/docker*
[20] Docker-Maven-Plug-in: *https://github.com/fabric8io/docker-maven-plugin*
[21] Do Not Use Docker In Docker for CI: *http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci*
[22] Never Expose Docker Socket: *https://dzone.com/articles/never-expose-docker-sockets-period*

Johannes Schnatterer
johannes.schnatterer@triology.de

Johannes Schnatterer ist Solution Architect bei der TRIOLOGY GmbH in Braunschweig. Technologisch ist er dort in den Bereichen „Java EE" und „Web" tätig und versucht, mit besonderem Fokus auf Qualität, Open-Source-Enthusiasmus, einem Hauch von Pedantismus und der Pfadfinderregel die IT-Welt jeden Tag ein bisschen besser zu machen. Derzeit arbeitet er am Cloudogu Ecosystem.

OO-Design-Patterns erweitert um Lambdas – eine Auswahl

Christian Nockemann, viadee Unternehmensberatung GmbH

Die Entwurfsmuster der „Gang-of-Four" haben sich als nützliche Werkzeuge für die Lösung gängiger Herausforderungen in der objektorientierten Programmierung bewährt. Wie profitiert diese Lösung von einem Einsatz der mit Java 8 eingeführten Lambda-Ausdrücke?

Hinweis: Alle der hier vorgestellten Code-Beispiele sind frei auf GitHub unter *https://github.com/cnockemann/lambda-enhanced-patterns* verfügbar.

„Design Patterns", das Buch der Autoren Gamma, Helm, Johnson und Vlissides (Gang-of-Four) ist ein wegweisendes Werk der objektorientierten Software-Entwicklung. Entwurfsmuster wie Beobachter, Strategie oder Adapter gehören inzwischen zu den Standardwerkzeugen von nahezu allen Java-Software-Entwicklern – teilweise ohne dass ihnen deren Ursprung bewusst ist, da viele weitverbreitete Java-Frameworks auf ihnen basieren. Sie bieten standardisierte Lösungsansätze für wiederkehrende Probleme und Herausforderungen im Software-Design.

In ihrer ursprünglichen Form leiten sich die Entwurfsmuster der GoF her aus den grundlegenden Paradigmen der Objektorientierung: Polymorphie, Vererbung und Kapselung. Zusätzlich hat mit den Lambda-Ausdrücken seit Java 8 die funktionale Programmierung Einzug in die objektorientierte Welt gehalten. Daher ist es an der Zeit, die Entwurfsmuster daraufhin zu prüfen, ob sie durch den Einsatz von Lambda-Ausdrücken vereinfacht oder sinnvoll erweitert werden können.

Dieser Artikel stellt eine Auswahl von Entwurfsmustern der GoF sowie jeweils eine um Lambda-Ausdrücke erweiterte entsprechende Variante vor. Die ausgewählten Entwurfsmuster bringen bei Design und Implementierung von Software-Komponenten erfahrungsgemäß den größten Nutzen und erscheinen beim Einsatz von Lambda-Ausdrücken überhaupt sinnvoll.

Warum überhaupt Entwurfsmuster?

Zunächst ein paar Worte zur Notwendigkeit des Einsatzes der Entwurfsmuster beziehungsweise die Beantwortung der grundsätzlicheren Frage „Wozu braucht man so etwas?". Die Software-Entwicklung ist im Vergleich zu vielen anderen Disziplinen sehr jung. Im Jahr 1958 wurde der Begriff „Software" das erste Mal in seinem heute gebräuchlichen Sinn verwendet *[1]*. Gegenwärtig ist also erst die dritte Generation von Software-Entwicklern tätig. Tatsächlich ist diese Disziplin so jung, dass derzeit noch kaum übergreifende Standards bestehen. Damit sind nicht die unzähligen IEEE-Normen, ISOC-RFCs oder Java-JSRs gemeint, sondern allgemein anerkannte, verbindliche Vorgaben für das Erstellen einer aus vielen Komponenten bestehenden Anwendung.

Wie bereits gesagt, Software-Entwickler arbeiten in einer jungen Disziplin und sie haben sich – bildlich gesprochen – bereits von improvisierten Holzverschlägen zu Konstruktionen vorgearbeitet, die gemäß einer reflektiert-übergeordneten Architektur gestaltet sind. Es setzt sich das Bewusstsein durch, dass nicht jedes Software-Projekt gleichsam das Rad neu erfinden muss; es gibt Standard-Lösungen für Standard-Probleme. An dieser Stelle setzen Entwurfsmuster an. Bei ihnen handelt es sich um Werkzeuge, die Software-Entwickler nutzen können, um wiederkehrende Probleme auf eine bewährte Art und Weise zu lösen. Qualitätskriterien wie Wiederverwendbarkeit, Änderbarkeit und Verständlichkeit stehen dabei im Vordergrund *[2]*. Sie sind quasi der Versuch, eine Baustatik für objektorientierte Software zu etablieren.

Erzeugungsmuster

Die GoF unterteilt die Entwurfsmuster in drei Kategorien: Erzeugungs-, Struktur- und Verhaltensmuster. Erzeugungsmuster beschäftigen sich mit der Konstruktion von Objekten. Sie verstecken die Erzeugungsmechanismen von Objekten hinter abstrakten Fassaden und erlauben so eine Nutzung, die entkoppelt ist von konkreten Implementierungen.

Hinter einer „abstrakten Fabrik" steht die Idee, konkrete Erzeugungs-Mechanismen (als „Fabriken" bezeichnet) für Objekte, die sich in der Struktur gleichen, also das gleiche Interface implementieren oder eine gemeinsame Oberklasse besitzen, zusammenzufassen und über eine einheitliche Schnittstelle verfügbar zu machen. Dabei wissen Aufrufende der abstrakten Fabrik nicht, welche konkrete Fabrik letztlich genutzt wird, um ein Objekt zu erzeugen. Darüber hinaus hat die abstrakte Fabrik nur Informationen zur allgemeinen Struktur des erstellten Objekts (Interface oder Oberklasse). Es wird gewissermaßen ein Fahrzeug ausgeliefert, ohne dass angegeben wird, ob es sich um einen Fabia, A8 oder eine E-Klasse handelt.

Ein wesentlicher Vorteil dieses Entwurfsmusters ist die Möglichkeit, konkrete Fabriken zu ergänzen oder auszuwechseln, ohne andere Komponenten der Software dafür anfassen zu müssen. Dadurch wird dem unschätzbar wertvollen Open-Closed-Prinzip Rechnung getragen, das besagt, dass „eine Klasse offen für Erweiterungen sein muss, jedoch geschlossen gegenüber Modifikationen" *[3]*. Anders ausgedrückt: Wenn einer Komponente Verhalten hinzugefügt werden soll, darf es dazu nicht nötig sein, bereits vorhandenes Verhalten zu ändern.

Wer schon einmal in der Situation war, ein kompliziertes (vielleicht auch noch ungetestetes) Stück Code ändern zu müssen, das vor langer Zeit geschrieben wurde, kennt die Gefahr, dass die Änderung das Kartenhaus der Gesamt-Software zusammenbrechen lässt. Eine konsequente Anwendung des Open-Closed-Prinzips reduziert diese Gefahr deutlich. Dieses Prinzip wird uns im Verlauf des Artikels noch wiederholt begegnen. Im Folgenden werden konkrete Implementierungen mit und ohne Lambda-Ausdrücke vorgestellt.

Klassisch

Listing 1 zeigt eine Beispiel-Implementierung für eine abstrakte Autofabrik. In der aufrufenden Komponente wird lediglich angegeben, welches Modell gebaut werden soll. Die Entscheidung darüber, welche konkrete Fabrik für diese Aufgabe gewählt wird, übernimmt die abstrakte Fabrik. Das Ergebnis ist ein allgemeines Auto, das erst im Rahmen der weiteren Verwendung (Log-Ausgabe der konkreten Attribute) die inneren Eigenschaften preisgibt.

Die konkreten Fabriken sind dabei Implementierungen eines allgemeinen Auto-Fabrik-Interface, die in der abstrakten Fabrik regist-

```
public static void main(String[] args) {
     AbstractFactoryClassic classicFactory = new AbstractFactoryClassic();
     Car fabia = classicFactory.getCarFactoryByModel(Model.FABIA).assemble();
     System.out.println(fabia.toJson());
     // Assembling Fabia...
     // {"brand":"Skoda","model":"Fabia","ps":54}

     Car a8 = classicFactory.getCarFactoryByModel(Model.A8).assemble();
     System.out.println(a8.toJson());
     // Assembling A8...
     // {"brand":"Audi","model":"A8","ps":190}

     Car eKlasse = classicFactory.getCarFactoryByModel(Model.EKLASSE).assemble();
     System.out.println(eKlasse.toJson());
     // Assembling E-Klasse...
     // {"brand":"Mercedes","model":"E-Klasse","ps":110}
  }
```

Listing 1: Nutzung der abstrakten Fabrik

```
public AbstractFactoryClassic() {
     carFactoryRegistry.put(Model.FABIA, new FabiaFactory());
     carFactoryRegistry.put(Model.A8, new A8Factory());
     carFactoryRegistry.put(Model.EKLASSE, new EKlasseFactory());
  }
```

Listing 2 : Registrierung von konkreten Fabriken

riert werden (in *Listing 2* als Beispiel unter Verwendung einer Map). *Listing 3* zeigt das allgemeine Interface und ein Beispiel für eine konkrete Implementierung.

Man benötigt also für jede Fabrik eine eigene Implementierungsklasse *(siehe Listing 4)*. Nun ist zu klären, ob Lambdas die Umsetzung des Entwurfsmusters „Abstrakte Fabrik" vereinfachen und/oder verbessern.

Lambda

Konkrete Fabriken können, statt mit eigenen Klassen, als Lambda-Ausdrücke umgesetzt werden. Dazu ist zunächst ein Functional-Interface zu definieren, das die allgemeine Struktur der Erzeugung vorgibt *(siehe Listing 5)*.

Per Definition ist jedes Interface, das lediglich eine Methode definiert, ein Functional-Interface (die Annotation „@FunctionalInterface" ist optional). Das in *Listing 3* dargestellte Interface erfüllte diese Voraussetzung bereits, allerdings ermöglicht die Extendierung des Supplier-Interface die Nutzung im Rahmen einiger Standard-JDK8-APIs. Da die „get()"-Methode des Supplier-Interface in *Listing 6* in Form einer Default-Methode überschrieben wird, ist dadurch die Functional-Interface-Voraussetzung nicht verletzt. So muss Code, in dem das „CarFactory"-Interface bereits genutzt wird, nicht geändert werden. Es wird lediglich die Möglichkeit ergänzt, es auch als Supplier zu verwenden – ganz im Sinne des Open-Closed-Prinzips. Nun können Lambda-Factories in der abstrakten Fabrik registriert werden.

Die Nutzung der „AbstractFactoryLambda" ist identisch zur klassischen, in *Listing 1* dargestellten Art und Weise. Neben schlankerem Code ergeben sich dadurch noch weitere Vorteile, die *Listing 7* zeigt.Durch die abstrakte Fabrik bereitgestellte Lambda-Fabriken lassen sich im JDK8-Stream-API nutzen, um auf einfache Art und Weise mehrere Autos zu produzieren. Daneben ist beispielsweise auch eine Verwendung als Fallback-Lösung beim Einsatz von Optionals sinnvoll.

In der „orElseGet()"-Methode lässt sich mithilfe der abstrakten Fabrik eine Alternative im Falle der Abwesenheit eines Autos generieren. Dabei wird einer der großen Vorteile von Lambdas genutzt: Die Objekt-Repräsentation des Lambda-Ausdrucks wird erst zur Laufzeit (mithilfe der mit Java 7 eingeführten ByteCode-Instruktion „invokedynamic") instanziiert und zwar nur dann, wenn sie wirklich notwendig ist (der optionale Wert also nicht vorhanden ist). Dadurch entsteht kein Aufwand für die Erzeugung des Fallback-Objekts im Falle des Vorhandenseins des Optional-Inhalts.

Die Variante, die ohne Lambdas arbeitet („orElse()"), führt den darin übergebenen Code hingegen immer sofort aus – unabhängig davon, ob das Ergebnis tatsächlich benötigt wird. Abgesehen von der Einsparung von Implementierungsklassen profitiert die Nutzung von Lambdas im Kontext der abstrakten Fabrik also von Synergie-Effekten mit JDK8-APIs.

```
public interface CarFactoryClassic<T extends Car> {
     T assemble();
  }
```

Listing 3 : Interface für konkrete Fabriken

```
  public class EKlasseFactory implements
  CarFactoryClassic<EKlasse> {
     @Override
     public EKlasse assemble() {
        return new EKlasse("Mercedes", "E-Klasse",
110);
     }
  }
```

Listing 4 : Beispiel-Implementierung einer Autofabrik: klassisch

```
@FunctionalInterface
  public interface CarFactoryLambda<T extends Car>
extends Supplier<T> {

     T assemble();

     @Override
     default T get() {
        return assemble();
     }
  }
```

Listing 5 : Functional-Interface für konkrete Fabriken

```
public AbstractFactoryLambda() {
     carFactoryRegistry.put(Model.FABIA, () -> new Fabia("Skoda", "Fabia", 52));
     carFactoryRegistry.put(Model.A8, () -> new A8("Audi", "A8", 190));
     carFactoryRegistry.put(Model.EKLASSE, () -> new EKlasse("Mercedes", "E-Klasse", 110));

  }
```

Listing 6 : Registrierung von Lambda-Factories

```
System.out.println("\nAssemble 3 Fabias in a row:");
  Stream.generate(abstractLambdaFactory.getCarFactoryByModel(Model.FABIA))
        .limit(3)
        .map(Car::toJson)
        .forEach(System.out::println);
  // Assemble 3 Fabias in a row:
  // Assembling Fabia...
  // {"brand":"Skoda","model":"Fabia","ps":52}
  // Assembling Fabia...
  // {"brand":"Skoda","model":"Fabia","ps":52}
  // Assembling Fabia...
  // {"brand":"Skoda","model":"Fabia","ps":52}

  Car absent = null;
  Car present = Optional.ofNullable(absent)
        .orElseGet(abstractLambdaFactory.getCarFactoryByModel(Model.A8));
  System.out.println(present.toJson());
  // Assembling A8...
  // {"brand":"Audi","model":"A8","ps":190}
```

Listing 7 : Nutzung einer Lambda-Factory in der JDK8-API

Erbauer

Abstrakte Fabriken sind nützlich in Szenarien, in denen es eine überschaubare Anzahl möglicher Ausprägungen von konkreten Objekten gibt. Wenn jedoch kaum gemeinsame Eigenschaften von Objektgruppen gefunden werden können oder jedes Objekt in seiner Beschaffenheit sogar einzigartig ist, ist es nicht sinnvoll beziehungsweise sogar unmöglich, ihre Erzeugung in Fabriken zu vereinheitlichen. In diesem Fall kann man zum Beispiel auf die Erzeugung per Konstruktor zurückgreifen. Wenn nun aber die Anzahl der Attribute eines Objektes sehr groß ist und davon einige optional sind beziehungsweise nur in Kombination mit bestimmten anderen benötigt werden, können die Konstruktoren beziehungsweise deren Anzahl (mit jeweils unterschiedlichen Signaturen) sehr unübersichtlich werden.

In solch einem Fall kommt das Erbauer-Entwurfsmusters zur Anwendung. Dabei werden die Parameter, die bei der Nutzung von Konstruktoren gleichzeitig übergeben werden, sequenziell eingefordert. Häufig wird es auch genutzt, um ganze Objekt-Bäume zu erzeugen.

Die grundlegende Funktionalität des Erbauer-Entwurfsmusters kann überdies durch die Nutzung eines Fluent-API um Semantik angereichert werden. Ein SQL-Ausdruck hat beispielsweise immer eine bestimmte sinnvolle Reihenfolge von Befehlen („select" – „from" – „where"). Ein Standard-Erbauer definiert jedoch keine Einschränkungen bezüglich der Reihenfolge der übergebenen Parameter. Hier können Fluent-Interfaces genutzt werden, um einen spezifischen Fluss (inklusive verschiedener Flussarme) von Parametern einzufordern und so die Idee eines Fluent-API umzusetzen. Sie schaffen somit die Voraussetzung für die Erstellung von Domänen-spezifischen Sprachen (Domain-Specific-Language, DSL) *[4]*. Dieser Umstand wird im folgenden Beispiel genutzt, um eine Pizzeria-Sprache zu definieren.

Klassisch

Listing 8 und *Listing 9* zeigen die Nutzung und Implementierung eines Pizza-Erbauers. Bis zu dem Punkt, an dem das zusammenzusetzende Objekt fertiggestellt ist und zurückgegeben wird, fügen die einzelnen Methoden diesem jeweils einen neuen Parameter hinzu und geben den Erbauer selbst zurück, damit daraufhin die nächste Methode zum Hinzufügen eines Parameters aufgerufen werden kann. So wird Stück für Stück das Objekt fertiggestellt.

Um einen festen Fluss von Parametern zu gewährleisten, muss der Pizza-Erbauer verschiedene (Fluent-)Interfaces implementieren. Diese sind im *Listing 10* dargestellt. Um dabei eine bestimmte Reihenfolge an Methoden-Aufrufen zu gewährleisten, müssen weitere Implementierungen hintereinandergeschaltet werden. Dies ist beispielhaft an dem „OnGoingPizzaBuilder" in *Listing 9* zu sehen.

Das Bauen einer Pizza kann nur mit der „withDough()"-Methode des „PizzaBuilder" beginnen, da dieser keine anderen Methoden besitzt. Der zurückgegebene „OnGoingPizzaBuilder" gibt daraufhin die weiteren Möglichkeiten zum Zusammensetzen der Pizza vor. Möchte man diese Möglichkeiten noch weiter einschränken (etwa wenn nach dem Teig die Wahl einer Käse-Art erzwungen werden soll), so sind weitere „OnGoingPizzaBuilder"-Implementierungen notwendig.

Lambda

Die Anzahl und Reihenfolge dieser Implementierungen kann sehr unübersichtlich sein. Hier können Lambdas die Komplexität deutlich reduzieren, denn sie bieten die Nutzung eines gängigen Prinzips der funktionalen Programmierung: das „Currying" (benannt nach dem Mathematiker Haskell Curry). Es bezeichnet die

```
public static void main(String[] args) {
     Pizza pizza = ClassicPizzeria
            .makePizza()
            .withDough(wheat())
            .andCheese(mozzarella())
            .andFirstTopping(broccoli())
            .andSecondTopping(mushrooms());

     System.out.println(pizza.toJson());
     // {"dough":wheat,"toppings":[broccoli,
mushrooms],"cheese":mozzarella}
  }

  public static PizzaBuilder makePizza() {
     return new PizzaBuilder();
  }
```

Listing 8 : Nutzung Pizza-Erbauer: klassisch

```
public class PizzaBuilder {

     Pizza pizza;

     public PizzaBuilder() {
        this.pizza = new Pizza();
     }

     public OnGoingPizzaBuilder withDough(Dough dough) {
        this.pizza.setDough(dough);
        return new OnGoingPizzaBuilder(pizza);
     }

     public class OnGoingPizzaBuilder implements CheeseBuilder, FirstToppingBuilder, SecondToppingBuilder {

        private Pizza pizza;

        public OnGoingPizzaBuilder(Pizza pizza) {
           this.pizza = pizza;
        }

        @Override
        public FirstToppingBuilder andCheese(Cheese cheese) {
           this.pizza.setCheese(cheese);
           return this;
        }

        @Override
        public Pizza andSecondTopping(Topping topping) {
           this.pizza.getToppings().add(topping);
           return pizza;
        }

        @Override
        public SecondToppingBuilder andFirstTopping(Topping topping) {
           this.pizza.getToppings().add(topping);
           return this;
        }
     }

  }
```

Listing 9 : Implementierung des Pizza-Erbauers: klassisch

```
public interface CheeseBuilder {
     FirstToppingBuilder andCheese(Cheese cheese);
  }

  public interface FirstToppingBuilder {
     SecondToppingBuilder andFirstTopping(Topping topping);
  }

  public interface SecondToppingBuilder {
     Pizza andSecondTopping(Topping topping);
  }
```

Listing 10 : Fluent-Interfaces

Umwandlung einer Funktion mit mehreren Parametern in eine Sequenz von Funktionen mit jeweils einem Parameter.

Voraussetzung für die Nutzung von Lambdas ist wiederum der Einsatz von Functional-Interfaces. Glücklicherweise erfüllen die bereits genutzten Interfaces *(siehe Listing 10)* dafür die Voraussetzungen, da sie jeweils lediglich eine Methode definieren. So ist weiter nur ein zusätzliches Interface zur Wahl des Teiges erforderlich *(siehe Listing 11)*.

Um den Pizza-Erbauer zu verwirklichen, sind keine zusätzlichen Implementierungsklassen mehr notwendig. Die Umsetzung des Erbauer-Entwurfsmusters vereinfacht sich so im Vergleich zu den vielen nötigen Implementierungsklassen für die Fluent-Interfaces in der klassischen Variante drastisch. Darüber hinaus wird eine klare Reihenfolge von Methodenaufrufen erzwungen: Jegliche Änderung (etwa der Aufruf von „andFirstTopping()" vor „andCheese()", was in der klassischen Variante möglich war) würde zu einem Compiler-Fehler führen.

Zusammenfassend lässt sich sagen, dass der Einsatz von Lambdas im Kontext des Erbauer-Entwurfsmusters zum einen Implementierungsklassen spart und zum anderen mithilfe des Currying das Erzwingen einer bestimmten Reihenfolge von Erbauer-Methoden vereinfacht. Das kommt der Verständlichkeit zugute und behebt so einen der Schwachpunkte der Implementierung des Erbauer-Entwurfsmusters.

```
public static void main(String[] args) {

     Pizza pizza = LambdaPizzeria
            .makePizza()
            .withDough(wheat())
            .andCheese(mozzarella())
            .andFirstTopping(broccoli())
            .andSecondTopping(mushrooms());

     System.out.println(pizza.toJson());
     // {"dough":wheat,"toppings":[broccoli, mushrooms],"cheese":mozzarella}

  }

  public static PizzaBuilderLambda makePizza() {
     // Currying:
     return dough -> cheese -> topping1 -> topping2 -> new Pizza(dough, cheese, topping1, topping2);
  }

  public interface PizzaBuilderLambda {
     CheeseBuilder withDough(Dough dough);
  }
```

Listing 11 : Implementierung und Nutzung des Pizza-Erbauers: Lambda

Verhaltensmuster

Verhaltensmuster befassen sich (wie der Name vermuten lässt) mit dem Verhalten von Objekten, beispielsweise der Interaktion zwischen Objekten sowie der Änder- und Austauschbarkeit von Verhalten zur Design- und Laufzeit. Da Verhalten über Funktionen abgebildet wird, bietet sich hier in besonderem Maße die Nutzung von Lambdas an. Einige Verhaltensmuster können dadurch sogar überflüssig werden.

Dies ließe sich etwa beim Strategie-Entwurfsmuster argumentieren. Die Idee hinter einer Strategie ist die Kapselung von Algorithmen in eine einheitliche Form (etwa über ein Interface), um sie dadurch (auch zur Laufzeit) austauschbar zu machen. Damit ist auch eine der Kern-Eigenschaften von Lambda-Ausdrücken beschrieben: dynamisch austauschbares Verhalten, gekapselt durch Functional-Interfaces.

Listing 12 zeigt die Anwendung des Strategie-Entwurfsmusters ohne Lambdas am Beispiel des Versuchs, einen Safe mithilfe verschiedener Hilfsmittel (Strategien) zu knacken. Die beiden Implementierungen unterscheiden sich also darin, dass die erste (Lockpick) erfolgreich ist, wenn der Safe ein Schloss hat, und die zweite (Dynamite), wenn die Stabilität des Safes einen gewissen Wert unterschreitet. Die Strategien können nun im Zuge des Raubs (englisch „heist") je nach Bedarf genutzt und ausgetauscht werden.

Wie *Listing 13* zeigt, lässt sich nach Bedarf jeweils von außen ein anderer Algorithmus übergeben, ohne dass dafür die innere Funktionsweise des Nutzers der Strategie verändert werden muss. Hier wird wieder dem Open-Closed-Prinzip Rechnung getragen, da so Verhalten ausgetauscht werden kann, ohne den umschließenden Code manipulieren zu müssen. Leider führt keine der beiden Strategien zum gewünschten Erfolg: Der Safe bleibt verschlossen.

Lambda

Wie bereits gesagt, handelt es sich bei diesem Szenario um ein Paradebeispiel für die Anwendung von Lambdas. Da die „SafeCrackingStrategy" wiederum ein Functional-Interface ist, sind keine

```
public interface SafeCrackingStrategy {
     void crackSafe(Safe safe);
  }

  public class Lockpick implements SafeCrackingStrategy {
     @Override
     public void crackSafe(Safe safe) {
        System.out.println("Trying to pick the lock...");
        if (safe.hasLock()) {
           safe.setOpen(true);
        }
     }
  }

  public class Dynamite implements SafeCrackingStrategy {
     @Override
     public void crackSafe(Safe safe) {
        System.out.println("\nBOOM!");
        if (safe.getSturdiness() <= 9000) {
           safe.setOpen(true);
        }
     }
  }
```

Listing 12 : Interface und Implementierungen für die Safe-Knacker-Strategie

weiteren Anpassungen nötig, es können direkt Lambda-Ausdrücke statt Instanzen der Implementierungsklassen übergeben werden *(siehe Listing 14)*.

Die Definition der „SafeCrackingStrategy" als Functional-Interface gewährleistet also auch eine Austauschbarkeit von klassischen Implementierungen und Lambda-Ausdrücken. Wenn stattdessen ein vom JDK mitgeliefertes Standard-Functional-Interface (wie „Function" oder „Consumer") verwendet wird, ist neben den einzelnen Implementierungen auch das „SafeCrackingStrategy"-Interface nicht erforderlich. Bei der Nutzung von Lambdas ist die explizite Anwendung des Strategie-Entwurfsmusters also weitgehend überflüssig, da es sich per Definition bei jedem Lambda-Ausdruck um eine austauschbare Strategie handelt.

Schablonen-Methode

In einer Situation, in der mehrere Varianten eines Algorithmus sich

```
public class SafeHeist {

     private SafeCrackingStrategy strategy;

     public void chooseStrategy(SafeCrackingStrategy
strategy) {
        this.strategy = strategy;
     }

     public void performHeist(Safe safe) {
        strategy.crackSafe(safe);
        System.out.println("Safe is open = " + safe.
isOpen());
     }

     public static void main(String[] args) {
        SafeHeist heist = new SafeHeist();
        Safe safeToBeCracked = new Safe();

        heist.chooseStrategy(new Lockpick());
        heist.performHeist(safeToBeCracked);

        heist.chooseStrategy(new Dynamite());
        heist.performHeist(safeToBeCracked);

        // Trying to pick the lock...
        // Safe is open = false
        //
        // BOOM!
        // Safe is open = false
     }
  }
```

Listing 13 : Nutzung der Strategien: klassisch

```
System.out.println("\nLet's make this easy!");
  heist.chooseStrategy(safe -> safe.setOpen(true));
  heist.performHeist(safeToBeCracked);

  // Let's make this easy!
  // Safe is open = true
```

Listing 14 : Strategy mit Lambda-Ausdruck

```
public abstract class Musician {

     public void performRehearsal() {
        arrive();
        prepare();
        rehearse();
        complain();
        rehearse();
        depart();
     }

     private void complain() {
        System.out.println("complaining about bass
player...");
     }

     private void depart() {
        System.out.println("driving home...");
        System.out.println("\n");
     }

     protected abstract void rehearse();

     protected abstract void prepare();

     private void arrive() {
        System.out.println("driving to rehearsal...");
     }
  }
```

Listing 15 : Abstrakte Oberklasse "Musiker"

```
public class GuitarPlayer extends Musician {
     @Override
     protected void rehearse() {
        System.out.println("playing guitar...");
     }

     @Override
     protected void prepare() {
        System.out.println("connecting guitar to
amp...");
     }
  }

  public class Singer extends Musician {
     @Override
     protected void rehearse() {
        System.out.println("singing...");
     }

     @Override
     protected void prepare() {
        System.out.println("turning on mic...");
     }
  }
```

Listing 16 : Gitarrist und Sänger

```
public static void main(String[] args) {
     List<Musician> band = Arrays.asList(new Guitar-
Player(), new Singer());

     for (Musician musician : band) {
        musician.performRehearsal();
     }

     // driving to rehearsal...
     // connecting guitar to amp...
     // playing guitar...
     // complaining about bass player...
     // playing guitar...
     // driving home...
     //
     //
     // driving to rehearsal...
     // turning on mic...
     // singing...
     // complaining about bass player...
     // singing...
     // driving home...
  }
```

Listing 17 : Bandprobe: klassisch

```
public class Musician {

     Runnable preparation;

     Runnable rehearsal;

     public Musician(Runnable preparation, Runnable
rehearsal) {
        this.preparation = preparation;
        this.rehearsal = rehearsal;
     }

     public void performRehearsal() {
        arrive();
        preparation.run();
        rehearsal.run();
        complain();
        rehearsal.run();
        depart();
     }
     // ...
  }
```

Listing 18 : Musiker mit Functional-Interface (Runnable)

```
public static void main(String[] args) {
     Musician guitarPlayer =
           new Musician(
                 () -> System.out.println("connecting guitar to amp..."),
                 () -> System.out.println("playing guitar..."));
     Musician singer =
           new Musician(
                 () -> System.out.println("turning on mic..."),
                 () -> System.out.println("singing..."));

     Stream.of(guitarPlayer, singer).forEach(Musician::performRehearsal);
     // driving to rehearsal...
     // connecting guitar to amp...
     // playing guitar...
     // complaining about bass player...
     // playing guitar...
     // driving home...
     //
     //
     // driving to rehearsal...
     // turning on mic...
     // singing...
     // complaining about bass player...
     // singing...
     // driving home...
  }
```

Listing 19 : Bandprobe: Lambda

im Ablauf sehr stark ähneln, kann das Entwurfsmuster „Schablonen-Methode" nützlich sein. Es ergänzt einen Algorithmus um variable Teilschritte, die nur in erbenden Klassen ausprogrammiert werden müssen, und ermöglicht so die konsequente Anwendung der DRY-Regel: Don't Repeat Yourself *[5]*.

Der gemeinschaftlich genutzte Teil des Algorithmus wird in einer abstrakten Oberklasse definiert. Die variablen Teilschritte werden über abstrakte Methoden abgebildet, die dann in den erbenden Klassen implementiert sind. Als Beispiel für „klassisch" dient hier eine Bandprobe. Sie ist für alle beteiligten Musiker ähnlich, aber nicht identisch:

1. Anfahrt
2. Vorbereitung
3. Probe
4. Beschwerde über den Bassisten
5. Fortführung der Probe
6. Rückfahrt

Schritt 1, 4 und 6 sind für alle Musiker (mit Ausnahme natürlich des Bassisten) gleich. Für diese Punkte lässt sich also eine abstrakte Oberklasse definieren *(siehe Listing 15)*.

Bei der Vorbereitung und der Durchführung der Probe unterscheiden sich die Tätigkeiten der Musiker jedoch. Anstatt dafür die gemeinschaftlichen Teile zu kopieren, müssen die konkreten Musiker lediglich von der Oberklasse erben und die abstrakten Methoden implementieren *(siehe Listing 16)*. So entsteht eine größtmögliche Wiederverwendung von Code *(siehe Listing 17)*.

Dieser Vorteil wird allerdings dadurch erkauft, dass die Struktur der Anwendung unübersichtlicher ist. So nützlich abstrakte Klassen und Methoden sind, so schwer verständlich sind sie oft auch (besonders für Personen, die den Code nicht selbst erstellt haben). Aus gutem Grund hat sich in den letzten Jahren die Grobregel „Delegation statt Vererbung" durchgesetzt *[6]*. Sie besagt, dass Aufgaben, die erbende Klassen übernehmen, stattdessen bevorzugt an referenzierte Komponenten delegiert werden sollten.

Lambda

Jetzt kommen wieder Lambda-Ausdrücke ins Spiel. Diese ermöglichen nämlich für das betrachtete Szenario den Verzicht auf Vererbung, ohne dabei die Flexibilität und Schlankheit der Schablonenmethode zu verlieren. Dafür delegiert der Musiker die variablen Teile der Probe an Functional-Interfaces beziehungsweise deren Implementierung in Form von Lambda-Ausdrücken *(siehe Listing 18)*.

Im Beispiel kann das „Runnable"-Interface genutzt werden, das keine Parameter entgegennimmt und nichts zurückgibt *(siehe Listing 19)*. Das gemeinsame Verhalten wird also weiterhin im Musiker gekapselt und die variablen Anteile als Lambda-Ausdrücke von außen übergeben. Die Vererbung und damit auch die Unterklassen von Musiker sind damit überflüssig und die Struktur des Codes vereinfacht.

Beobachter

Die Idee hinter dem Beobachter-Entwurfsmuster ist, dass Zustandsveränderungen in Form von Events bekanntgemacht werden. Um über das Eintreten eines Events benachrichtigt zu werden, lassen sich Beobachter in einem zentralen Verteiler (auch „Subjekt" genannt) registrieren. Sobald das Event ausgelöst wird, können die Beobachter darauf mit individuellem Verhalten reagieren. Dies gewährleistet eine lose Kopplung von Event zu Beobachter und eine leichte Austauschbarkeit von Verhalten.

Eingangs wurde bereits erwähnt, dass einige Entwurfsmuster die Grundlage ganzer Java-Frameworks sind und somit unbewusst von vielen Entwicklern genutzt werden. Dies trifft insbesondere auf das Beobachter-Entwurfsmuster zu, das eines der Stützpfeiler des Reactive-Programming-Paradigmas ist und damit maßgeblichen Einfluss auf RxJava und ähnliche Frameworks hat.

```
public class Operator {

     private List<CallCenterAgentListener> agents = new ArrayList<>();

     public void distributeCall(Call call) {

        for (CallCenterAgentListener agent : agents) {
           agent.acceptCall(call);
        }

     }

     public void registerAgent(CallCenterAgentListener agent) {
        this.agents.add(agent);
     }
  }
```

Listing 20 : Callcenter-Operator (Subjekt)

Betrachten wir zunächst wieder die klassische Variante. Als Beispiel dient hier ein fiktives Callcenter *(siehe Listing 20)*. Das Subjekt ist ein Operator, der Telefonanrufe verteilt. Beim ihm können sich Callcenter-Agenten (als Beobachter) registrieren, die dann benachrichtigt werden, wenn das Event „Anruf geht ein" stattfindet. Wie die einzelnen Agenten auf einen Anruf reagieren, ist diesen selbst überlassen. Um als Beobachter vom Operator anerkannt zu werden und sich registrieren zu können, müssen Callcenter-Agenten ein bestimmtes Interface implementieren *(siehe Listing 21)*.

Damit sie über einen Anruf informiert werden, müssen sich Kündigungsmanagement, Kunden-Support und Upselling beim Operator registrieren. Sobald ein Anruf eintrifft, wird dieser vom Operator verteilt (via „distributeCall()") und so die Reaktion der Agenten darauf ausgelöst *(siehe Listing 22)*.

In diesem Beispiel suggerieren die Callcenter-Agenten, bei Beobachtern handele es sich in erster Linie um Zustände (in Form von Objekten), die durch eine Aufruf-Kette gereicht werden. In Wirklichkeit steht das Weiterreichen von Verhalten im Vordergrund. Die Implementierungsklassen um das Verhalten herum sind eine Notwendigkeit aus der Zeit, in der Java noch keine Übergabe von Funktionen als Parameter (also Lambdas) erlaubte.

Lambda

Beim Einsatz von Lambda-Ausdrücken kann nun auf die Implementierungsklassen verzichtet werden. Hier ist wieder der Umstand zu nutzen, dass es sich bei „CallCenterAgentListener" um ein Functional-Interface handelt, sodass dem Operator direkt Lambda-Ausdrücke übergeben werden können *(siehe Listing 23)*.

Durch den Verzicht auf Implementierungsklassen muss erst zum Zeitpunkt der Registrierung entschieden werden, welches Verhalten das beobachtete Event auslösen soll. Ein weiterer Vorteil ergibt sich aus einem Szenario, in dem das Beobachter-Entwurfsmuster oft eingesetzt wird: mehrere nebenläufige Threads. Darin ist jeder Zustand aufgrund der konkurrierenden Zugriffe ein Risiko und sollte daher vermieden werden. Hier haben Lambdas gegenüber anonymen inneren und implementierenden Klassen die Nase vorn, da die Zustände (sprich „Felder") von Letzteren über ihren gesamten Lebenszyklus hinweg geändert, während die Felder in Lambda-Ausdrücken nur während der Deklaration manipuliert werden können.

```
public interface CallCenterAgentListener {
     void acceptCall(Call call);
  }

  public class CustomerRetention implements CallCenterAgentListener {
     @Override
     public void acceptCall(Call call) {
        System.out.println("\"Please stay with us!\"");
     }
  }

  public class CustomerSupport implements CallCenterAgentListener {
     @Override
     public void acceptCall(Call call) {
        System.out.println("\"Have you tried turning it off and on again?\"");
     }
  }

  public class Upselling implements CallCenterAgentListener {
     @Override
     public void acceptCall(Call call) {
        System.out.println("\"Try this new feature for only 10$ a month!\"");
     }
  }
```

Listing 21 : Callcenter-Agenten

```
public static void main(String[] args) {

     Operator operator = new Operator();

     operator.registerAgent(call -> System.out.println("\"Try this new feature for only 10$ a month!\""));
     operator.registerAgent(call -> System.out.println("\"Have you tried turning it off and on again?\""));
     operator.registerAgent(call -> System.out.println("\"Please stay with us!\""));

     operator.distributeCall(new Call());
     // "Try this new feature for only 10$ a month!"
     // "Have you tried turning it off and on again?"
     // "Please stay with us!"

  }
```

Listing 22 : Beobachter: klassisch

```
public static void main(String[] args) {
     Operator operator = new Operator();

     operator.registerAgent(new Upselling());
     operator.registerAgent(new CustomerSupport());
     operator.registerAgent(new CustomerRetention());

     operator.distributeCall(new Call());
     // "Try this new feature for only 10$ a month!"
     // "Have you tried turning it off and on again?"
     // "Please stay with us!"
  }
```

Listing 23 : Beobachter: Lambda

Beiden Varianten ist gemein, dass sie Zustände des umgebenden Kontexts nur manipulieren können, wenn es sich um effektiv finale Variablen handelt, was allerdings nur erzwungen werden kann, wenn sie immutable sind.

Fazit

Es gibt noch viele andere Entwurfsmuster der Gang-of-Four, die von Lambda-Ausdrücken profitieren würden. Dies gilt in besonderem Maße für Verhaltensmuster, die Probleme lösen sollen, die seit der Einführung von Lambda-Ausdrücken in Java viel leichter zu beseitigen oder gar nicht mehr vorhanden sind. Für diesen Artikel musste verständlicherweise eine Auswahl getroffen werden.

Es ist vermutlich aufgefallen, dass eine ganze Kategorie von Entwurfsmustern aus dem GoF-Basismodell fehlt: die Strukturmuster. Der Grund dafür liegt in ihrer Natur: Sie basieren mehr noch als die anderen Kategorien auf den eingangs erwähnten Grundparadigmen der Objekt-Orientierung (Polymorphie, Vererbung und Kapselung). Daher findet sich für Lambda-Ausdrücke und ihre funktionale Natur nur schwer eine gewinnbringende Anwendungsmöglichkeit.

Wie dieser Artikel verdeutlicht, haben die Entwurfsmuster der Gang-of-Four auch in Zeiten der Einführung von funktionaler in die objektorientierte Programmierung nicht an Relevanz und Nützlichkeit verloren. Der Einsatz von Lambda-Ausdrücken macht einen wesentlichen Anteil der Entwurfsmuster schlanker, effizienter und eröffnet weitere neue Anwendungsmöglichkeiten. Darüber hinaus sind die klassischen Implementierungen der Entwurfsmuster mit den jeweiligen Lambda-Varianten kompatibel und austauschbar, wenn Functional-Interfaces sinnvoll definiert und genutzt werden. Dadurch sind bereits vorhandene Umsetzungen der einzelnen Entwurfsmuster nicht komplett zu überarbeiten.

Vielen Dank an Tobias Voss („@tobiaslvoss"), der beim Erstellen dieses Artikels als Sparringspartner fungiert hat.

Quellen

[1] John W. Tukey, The Teaching of Concrete Mathematics, The American Mathematical Monthly, Vol. 65, no. 1 (Januar 1958), Seite 2
[2] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides, Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software, Addison Wesley, 2004, Vorwort
[3] Bertrand Meyer, Object Oriented Software Construction, Prentice Hall, 1988, Seiten 57–61
[4] Martin Fowler: *https://martinfowler.com/bliki/FluentInterface.html*
[5] Andrew Hunt, David Thomas, The Pragmatic Programmer, The Pragmatic Bookshelf, 1999, Seite 27
[6] Robert C. Martin: *http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/2001/cexp1908/martin/martin.htm*

Christian Nockemann
christian.nockemann@viadee.de

Christian Nockemann ist IT-Berater und Software-Entwickler bei der viadee IT-Unternehmensberatung. Sein Fokus liegt auf Design und Umsetzung von individuellen Enterprise-Anwendungen auf Basis von Java/Spring mit einem besonderen Schwerpunkt auf der Verbesserung von Code-Qualität (Testbarkeit, Clean-Code, Nutzung von Entwurfsmustern etc.).

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: © panimoni/Fotolia
S. 10: © Ivan Trifonenko/123RF
S. 19: © Denis Ismagilov/123RF
S. 24: © Cathy Yeulet/123RF
S. 30: © inueng/123RF
S. 35: © yarruta/123RF
S. 46: © convisum/123RF
S. 56: © Tamara Kulikova/123RF

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

Mediadaten und Preise unter:
www.doag.org/go/mediadaten

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

Inserentenverzeichnis

Deutsche Welle	S. 45
Diconium	S. 33
DOAG e.V.	U 2, U 3, U 4
Java Forum Stuttgart	S. 9
QAware	S. 29
Zertificon	S. 27

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

Save the Date

20. - 23. Nov 2018

in Nürnberg

2018.doag.org

Eventpartner: AOUG AUSTRIAN ORACLE USER GROUP · SOUG swiss oracle user group · iJUG Verbund · ORACLE