Ausgabe 05/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 05

Java aktuell

iJUG Verbund
www.ijug.eu

Microservices
Performancetests, Service Meshes, MicroProfile GraphQL und mehr

Javas Geheimnisse
Weniger bekannte Features und Eigenheiten

Deep Learning
Einblick in das Trend-Thema

Klein aber oho: MICROSERVICES

JavaLand

16. - 18. März 2021
in Brühl bei Köln

Save the Date

Hybride Veranstaltung

Was die JavaLand als Plattform für Wissentransfer und Networking ausmacht, kannst du im Phantasialand oder online erleben. Als Teilnehmer entscheidest du selbst, welche Variante du wählen möchtest.

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Liebe Leser der Java aktuell,

in dieser Ausgabe dreht sich alles um Microservices. Ein absolutes Trend-Thema, das sich besonders in den letzten Jahren stetig steigender Beliebtheit erfreut hat. Doch ist die Zerlegung in viele kleine Services immer eine geeignete Lösung? Keinesfalls. Darum sollte die Umstellung auf eine Microservices-Architektur gut bedacht und begründet sein. Unsere Autoren geben Ihnen praxisnahe Entscheidungshilfen an die Hand. Auch technische Beiträge sind wieder mit an Bord. Zum Beispiel in Form von Performancetests für Microservices, Domain-driven Design mit MicroProfile GraphQL, GraalVM und Saga-Orchestrierung. Weiterhin werfen wir einen detaillierten Blick auf Service Meshes, lüften die Geheimnisse von Java und begeben uns auf die Suche nach unserem ganz persönlichen „Ikigai".

Eine wichtige Information zur Zeitschrift: Die Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) herausgegeben. Verlag ist die DOAG Dienstleistungen GmbH. Das Magazin wird zu einem kleineren Teil über Inserate finanziert, der weitaus größte Teil der Kosten wird aus den Überschüssen der JavaLand-Konferenz getragen. Daneben entstehen noch Einnahmen aus dem Vertrieb über den Zeitschriftenhandel und durch Abonnements.

Die aktuelle Pandemie hat aufgrund des Veranstaltungsverbots erhebliche Auswirkungen auf die Finanzen des Magazins, denn die Überschüsse aus der JavaLand fallen dieses Jahr vollständig weg. Dies zwingt die DOAG Dienstleistungen GmbH als Verlag zu drastischen Kosteneinsparungen, auch bei den Produktionskosten der Magazine. Deswegen musste der Verlag in Abstimmung mit dem iJUG schweren Herzens entscheiden, die Ausgabe 6/20 der Java aktuell nicht zu produzieren.

Die Ausgabe 1/21 wird pünktlich im Dezember dieses Jahres erscheinen. Im Jahr 2021 sind insgesamt nur vier, statt wie gewohnt sechs Ausgaben geplant. Diese vier Ausgaben erscheinen quartalsweise. Ab 2022 sollen dann wieder sechs Ausgaben produziert werden. Wir bedanken uns für Ihr Verständnis für diese Maßnahmen, die zur Zukunftssicherung des iJUGs und seiner Aktivitäten beitragen.

Wir wünschen Ihnen viel Spaß beim Lesen!

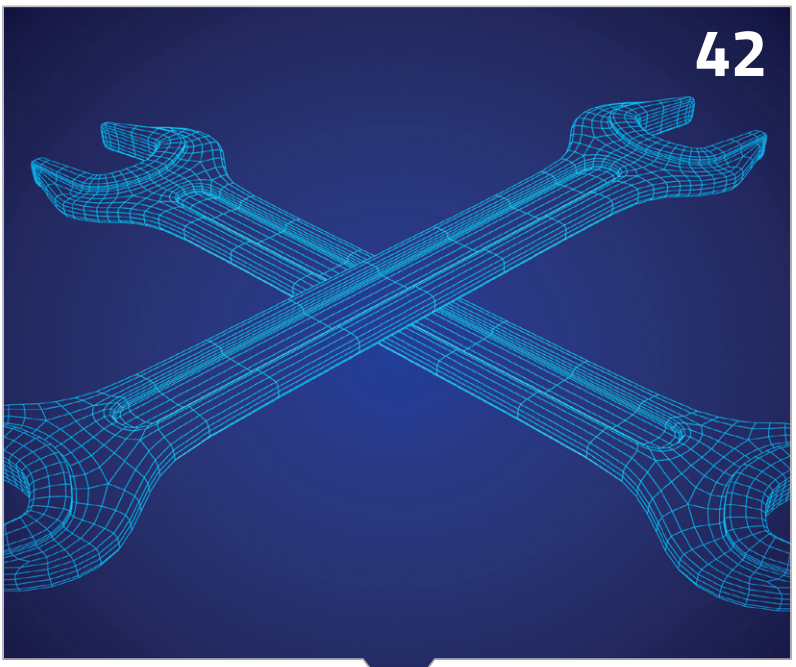
Ihre

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Wann ist eine Microservices-Architektur die richtige Wahl?

Domain-driven Design mit MicroProfile GraphQL



Service Meshes in Microservices-Architekturen

Ein Einstieg in das Thema Deep Learning

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

16. März 2020

Java 8 Support: Verlängerung geht in die Verlängerung

Der kostenpflichtige „Extended Support" für Java 8 wird auf vielfachen Kundenwunsch von 2025 bis 2030 verlängert *[1]*. Java 8 ist immer noch sehr weit verbreitet: Der kürzlich veröffentlichte „Snyk JVM Ecosystem Report" berichtet, dass Ende 2019 immer noch 64 Prozent der 2000 Umfrage-Teilnehmer Java 8 für ihre Haupt-Anwendung nutzten, nur 25 Prozent hingegen Java 11 (alle anderen Versionen sind unter „ferner liefen …" gelistet). Wer weiß, vielleicht wird Java 8 ja mein persönliches COBOL und begleitet mich bis zur Rente. Übrigens ist im Snyk-Report auch zu finden, dass nur neun Prozent überhaupt für kommerziellen Support zahlen – aber das dürfte insgesamt trotzdem noch eine hübsche Summe sein. 55 Prozent davon zahlen an Oracle, während sich Red Hat, IBM und Azul den Rest teilen.

1. April 2020

Eclipse Theia 1.0

Die Eclipse Foundation hat gestern die Version 1.0 von Theia, einer weiteren IDE-Plattform, bekannt gegeben. Theia soll die Grundlage für sowohl browserbasierte als auch Desktop-IDEs für alle möglichen Sprachen sein und wird ganz offiziell als Konkurrenz zu Visual Studio Code positioniert. Es folge „vielen der Design-Entscheidungen" von VS Code und unterstütze sogar dessen Erweiterungen, sei aber modularer aufgebaut, auf die Cloud und den Desktop ausgerichtet und darüber hinaus herstellerneutral. Hinter Theia steckt insbesondere die deutsche Firma TypeFox, aber auch Schwergewichte wie Red Hat und SAP.

20. April 2020

EclipseCon 2020 wird virtuell

Auch die EclipseCon 2020 (19. bis 22. Oktober) wird zu einer virtuellen Veranstaltung. Die virtuellen Konferenzen sind gut und werden sicher technologisch und in der Umsetzung noch mal einen Schub bekommen durch die massive Erfahrung, die gerade gewonnen wird. Aber nur virtuell macht es auf Dauer ja auch keinen Spaß – jedenfalls solange die VR-Technologie mir noch nicht vorgaukeln kann, ich sei mittendrin.

28. April 2020

Quarkus 1.4

Version 1.4 des Quarkus-Frameworks ist fertig. Support für Java 8 ist darin als „deprecated" gelistet und soll mit Quarkus 1.6 verschwinden – also laut Plan bereits im Juni! Moment, wie war das mit meiner Rente? Quarkus ist ja ziemlich hip, aber kann es genügend Entwickler*innen (oder deren Bosse) dazu bewegen, auf Java 11 und höher zu migrieren? Ein paar richtige Features gibt es natürlich auch: HTTP/2-Support zum Beispiel oder das Framework Funqy, mit dem sich Funktionen für AWS-Lambda, Azure Functions oder Knative schreiben lassen.

3. Mai 2020

#JakartaEE

In der neuesten Ausgabe des Jakarta-EE-Newsletters (auf *agilejava.eu*) kündigt Ivar Grimstad – Projektleiter und „Developer Advocate" für Jakarta bei der Eclipse Foundation – die neue Webseite *start.jakarta.ee* an. Es handelt sich nicht um ein Projektgenerierungstool sondern soll die zentrale Anlaufstell für alle werden, die mit Jakarta EE neu loslegen wollen. Interessant ist in diesem Zusammenhang sicher auch der neue Jakarta EE YouTube Channel, parallel zum bereits existierenden „Studio Jakarta EE".

16. Mai 2020

Spring Boot 3.0 mit Docker-Unterstützung

Spring Boot 3.0 ist offiziell freigegeben. Neben Kompatibilität mit Java 14 sowie einigen Upgrades in den Abhängigkeiten zu Spring- und Dritt-Projekten bietet es unter anderem Unterstützung für das Bauen von Docker-Images. Dafür werden Cloud Native Buildpacks genutzt, deren Konzept ursprünglich von Cloud Foundry stammt; sie werden jetzt im Rahmen der herstellerübergreifenden Cloud Native Computing Foundation entwickelt und kommen so auch Docker- beziehungsweise Kubernetes-Nutzern zugute. Einfach mal „mvn spring-boot:build-image" ausprobieren. Ups, jetzt habe ich mich geoutet – aber laut Snyk JVM Ecosystem Report gehöre ich damit zur weiterhin deutlichen Mehrheit. Funktioniert aber mit Gradle genauso.

20. Mai 2020

inside.java

Die Java Platform Group bei Oracle bietet eine neue Website an, die „curated content" (also ausgewählte Inhalte anderer Kanäle) zu Java bieten soll: inside.java.

21. Mai 2020

GraalVM 20.1

Graal VM 20.1 ist da und bietet eine Reihe von Verbesserungen für Java und Kotlin sowie für andere unterstützte (Nicht-JVM-)Programmiersprachen. Außerdem enthält es Verbesserungen bei der Erzeugung nativer Images: Unter anderem soll durch einen neuen „Saturated Type Flow" – in dieser Version noch explizit zu aktivieren – die statische Analyse besser skalieren, die bislang „für einige große Applikationen" nicht möglich gewesen sei.

22. Mai 2020

25 Jahre Java – „Moved by Java“

Java wird 25 Jahre alt, und das geht in der „kontaktlosen“ Zeit ein bisschen unter. Oracle hat die Kampage #MovedbyJava gestartet, unter anderem mit Videos, in denen bekannte Personen der Java-Welt über ihre eigenen Erlebnisse mit der Sprache und der Plattform berichten und erzählen, was für sie Javas Popularität ausmacht. „Relevant“, „sich anpassend an die geänderten Bedürfnisse der Softwareentwicklung“ und entgegen dem Klischee „überhaupt nicht langsam“, sondern der Treiber hinter High-Performance-Anwendungen nicht nur im Finanzsektor – das sind nur drei von vielen Schnipseln, die mir im Gedächtnis geblieben sind. Aber wir alle haben sicher unsere eigenen Geschichten zu Java. In diesem Sinne: Happy Birthday – und auf weitere 25 Jahre mit vielen Erfolgsgeschichten!

25. Mai 2020

JDK Distributions

Die Frage nach einer Übersicht zu JDK-Distributionen kam heute bei einem iJUG-Meeting auf und wurde auch prompt beantwortet: Es gibt bereits eine gute Übersicht *[2]*. Sie scheint nicht immer topaktuell zu sein, aber das kann ja jede und jeder über GitHub Pull Requests selbst ändern („git rebase“ nicht vergessen, das scheint aktuell das Problem zu sein, warum die Verlängerungdes Java-8-Supports noch nicht aufgeführt ist).

29. Mai 2020

Octopus Malware

Das GitHub SIRT (Security Incident Response Team) hat über eine Attacke auf Open-Source-Projekte auf ihrer Plattform berichtet, die auf NetBeans-Projekte spezialisiert ist. Warum ausgerechnet auf NetBeans-Projekte, die ja eine vergleichsweise geringe Verbreitung haben, ist nicht klar – vielleicht ist bislang auch nur die Spitze des Eisbergs sichtbar geworden. Nach Attacken auf Build Tools wie npm oder durch speziell präparierte Versionen von XCode sollte spätestens jetzt klar sein: Überlege gut, welchem Open-Source-Projekt du traust! GitHub war bereits im März auf die „Octopus Scanner“ Malware hingewiesen worden, die sich lokal in NetBeans-Projekte kopiert und unter anderem Jar-Files mit einem „Dropper“ infiziert. Dieser startet letztlich ein „Remote Administration Tool“, das dann Befehle von einem „Command and Control Server“ entgegennimmt. Die Owner der 26 infizierten Repositories waren offensichtlich arglos, ebenso wie diejenigen, die Forks oder Clones erstellt hatten. Die detaillierte Analyse ist online zu finden *[3]*.

30. März 2020

MicroProfile 4.0 verschoben

MicroProfile 4.0 wird sich laut einem Eintrag von Roberto Cortez im offiziellen MicroProfile-Blog vermutlich auf August 2020 verzögern. Grund ist die formelle Struktur in Form einer Working Group, die sich das Projekt geben möchte beziehungsweise muss – so wie Jakarta EE und andere Projekte bei der Eclipse Foundation; die gemeinsame Working Group mit Jakarta EE ist allerdings vom Tisch. Das Vorhaben ist noch nicht abgeschlossen – und das neue Release soll erst im Rahmen dieser neuen Organisation herauskommen. Vor einigen Wochen gab es für 4.0 noch die Diskussion, ob Java 8 unterstützt werden muss oder Java 11 vorausgesetzt werden darf. Aufgrund der Zahlen scheint aber auch hier die (Nicht-)Entscheidung zu sein, weiterhin mit Java 8 zu arbeiten.

1. Juni 2020

Java 15 kurz vor der Zielgeraden

Das für September anvisierte JDK 15 soll am 11. Juni in „Rampdown Phase One“ gehen. Neben den bereits früher im Tagebuch genannten Features soll das „Foreign-Memory Access API“ (JEP 383), bereits also Inkubator-Feature in Java 14, mit einigen Änderungen noch mal als „Second Incubator“ dazukommen, bevor es dann hoffentlich ein stabiles und reguläres Feature wird. Außerdem wird die „RMI Activation“ entfernt (wohlgemerkt nicht RMI als solches, sondern nur die „Remote Object Activation“, die angeblich kaum noch genutzt wird und bereits in Java 8 als optional deklariert wurde).

Referenzen:

[1] *https://www.oracle.com/java/technologies/java-se-support-roadmap.html*

[2] *https://rafael.codes/openjdk/*

[3] *https://securitylab.github.com/research/octopus-scanner-malware-open-source-supply-chain*

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 im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 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“).

Markus' eclipse-Corner

Entsprechend ihrem aktuellen Plan *[1]* hat die Eclipse Foundation am 23. Juni 2020 den ersten Meilenstein von Jakarta EE 9 veröffentlicht – und somit mit nur leichtem Verzug zur ursprünglichen Planung vom 12. Juni. Zeit, um sich genauer anzuschauen, was darin enthalten ist und womit das finale Release am 16. September aufwarten wird. Selbst, wenn der Termin vermutlich nicht auf den Tag genau gehalten wird: Ein Blick auf Jakarta EE 9 lohnt sich in dieser Ausgabe trotzdem.

Jakarta EE 9 ist, wie im letzten Heft bereits beschrieben, aus Anwendersicht nicht sonderlich spektakulär, da es das exakt gleiche API mitbringt, das bereits Jakarta EE 8 respektive dessen unter Oracle-Leitung entwickeltes und durch den JCP zur Norm erhobenes Pendant Java EE 8 anbot. Nur, dass dieses API nun eben in einem neuen Namensraum lebt (jakarta.* statt javax.*), wodurch keine markenrechtlichen Streitigkeiten mit Oracle mehr entstehen können, und dass Jakarta EE nun auf einem JRE der Version 11 (und früher) lauffähig ist. Und Letzteres ist aus zweierlei Hinsicht ein technisch großer und wichtiger Schritt.

Zum einen ist es bekanntlich so, dass Java SE 11 von sich aus einige Technologien nicht mehr enthält, beispielsweise JAXB und SOAP. Application-Server haben sich bislang auf die Existenz dieser Technologien verlassen. Wer als Hersteller Jakarta-EE-9-zertifiziert sein möchte, muss nun selbst die entsprechenden APIs und Implementierungen mitbringen (das muss nichts Schlechtes sein, die eigene Implementierung kann ja durchaus besser sein als die von Oracle). Dies wird nicht den Anwendungsentwicklern aufgelastet – vorerst zumindest, denn dies könnte sich in zukünftigen Releases der Plattform allerdings ändern, wie es jetzt bereits mit Teilen von EJB und JAX-WS der Fall ist, die nur noch „optional" und somit nicht zwingend auf jedem Application-Server verfügbar sind.

Zum anderen aber, und das kann gar nicht genug hervorgehoben werden, dürfen zwar die APIs selbst keine neueren Sprach- und Bibliotheksbestandteilte nutzen, als Java SE 8 sie enthielt, dies gilt aber nicht für die Anwender der APIs. Ganz konkret bedeutet das: Es ist ab Jakarta EE 9 nun offiziell erlaubt, in der eigenen Anwendung all das zu nutzen, was Java SE 11 hergibt (explizit in Jakarta EE 9 aufgezählte Einschränkungen ausgenommen). Und, ja genau: 11, nicht 9! Angefangen von der wesentlich flotteren JVM *[2]*, neuen Sprachkonstrukten wie „var", bis zu neuen Interfaces und Klassen, die es in 8 nicht gab, darf ein Anwendungsentwickler nun aus dem Vollen schöpfen (oder Fast-Vollen, da die Welt sich ja bereits weitergedreht hat und wir eigentlich schon bei Java SE 14 sind – aber Jakarta EE 10 steht ja in der Startlöchern und wer weiß, was man dort dann zusätzlich erlauben wird)!

Auch wenn also die meisten beim Lesen der Feature-Liste von Jakarta EE 9 gedacht haben, erst Jakarta EE 10 würde „der große Wurf" und man könne das Neuner-Release doch eher überspringen wie bereits das vorige: Falsch gedacht! Auf den zweiten Blick bietet die Garantie, dass Java EE 9 unter der Motorhaube Java-SE-11-tauglich ist, genug Pferdestärken, um kräftig an der Leistungsschraube von Bestandsanwendungen zu drehen. Und mit Jakarta EE 9 ist der Weg zum 10er-Release geebnet, das dann auch die APIs selbst kräftig modernisieren wird, unter anderem mit Application-Server-freien Microservices (Java SE Bootstrap API), CDI-zentrischer Architektur und vielem mehr, auf das wir seit Langem warten!

Das Glas ist also endlich wieder halb voll, und ich freue mich schon darauf, meine ersten Anwendungen auf Jakarta EE 9 zu portieren!

Referenzen

[1] Aktueller Jakarta EE 9 Release Plan: *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9#jakarta-ee-9-schedule*

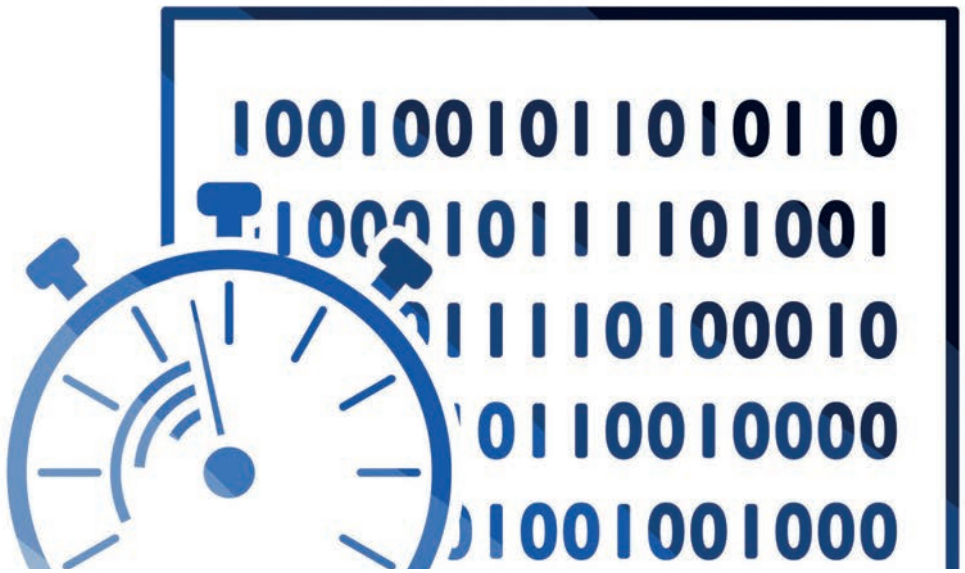
[2] Ursprünglicher Jakarta EE 9 Release Plan: *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9ReleasePlan*

[3] JVM 8/11 Performance-Vergleich: *https://blog.qfotografie.de/2019/01/21/java-8-vs-java-11-benchmark-a-productive-business-application/*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.



Unbekannte Kostbarkeiten des SDK Heute: Temporäre Dateien und Verzeichnisse

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

```
public static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)

public static Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)

public static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)

public static Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)
```

Listing 1

Von Zeit zu Zeit haben wir die Anforderung, Dateien zu erzeugen, beispielsweise um Zwischenergebnisse abzuspeichern oder Dateifunktionen des Betriebssystems verwenden zu können. Häufig sind die Dateien nach ihrer Verwendung wieder zu löschen. Java kann uns das Erzeugen der Dateien zwar nicht abnehmen, aber beim Finden noch nicht verwendeter Namen helfen. Das Löschen der Dateien kann dagegen mehr oder weniger automatisiert werden. Dasselbe gilt für Verzeichnisse und damit sukzessive für Dateien in diesen temporären Verzeichnissen. Den entsprechenden Methoden gilt die Aufmerksamkeit unserer heutigen unbekannten Kostbarkeiten.

Die Klasse Files

Mit Java 7 wurde das Package `java.nio.file` eingeführt, das unter anderem die Klasse `Files` enthält. Mit Java 8, 11 und 12 bekam diese Klasse eine ganze Reihe neuer Methoden spendiert, die wir bereits in den *unbekannten Kostbarkeiten 06/2019* gewürdigt haben. Heute gehen wir noch weiter zurück und schauen uns Methoden an, die bereits in der Ursprungsversion der Klasse `Files` mit Java 7 Einzug hielten; sie beginnen alle mit dem Präfix `createTemp` und sind in *Listing 1* dargestellt. Beginnen wir mit der Methode `createTempFile()`. Die in *Listing 2* gezeigten Code-Zeilen erzeugen eine Datei mit einem zufälligen Namen und registrieren deren Löschung bei Programmende.

Der Dateiname hat das Präfix „`pre-`" und die Extension „`.txt`". Der mittlere und generierte Teil ist eine Ziffernfolge, die als Long-Zufallszahl erzeugt wird. Das Interessante ist das Registrieren einer automatischen Löschung bei Terminierung der JVM mit der Methode `File.deleteOnExit()`. Wir zitieren deren JavaDoc:

„Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates. Files (or directories) are deleted in the reverse order that they are registered. Invoking this method to delete a file or directory that is already registered for deletion has no effect. Deletion will be attempted only for normal termination of the virtual machine, as defined by the Java Language Specification. Once deletion has been requested, it is not possible to cancel the request. This method should therefore be used with care."

Unter einem normalen Ende ist hier das Programmende oder ein CTRL-C auf Betriebssystemebene gemeint. Ein „`kill -9 <pid>`" gehört nicht dazu.

Die anderen Methoden des Listings 1 sind analog zu verwenden. Optional können ein Verzeichnis oder Dateiattribute angegeben werden, die jedoch nichts mit dem temporären Charakter der Erzeugnisse zu tun haben und daher von uns nicht näher erläutert werden.

Abschließend bleibt noch zu klären, wo die temporären Dateien und Verzeichnisse erzeugt werden. Das Elternverzeichnis der

```
Path tempFile = Files.createTempFile("pre-", ".txt");
tempFile.toFile().deleteOnExit();
```

Listing 2

temporären Dateien und Verzeichnisse wird durch das System-Property `java.io.tmpdir` festgelegt. Beim Linux-System des Autors ist dies `/tmp`.

Das automatisierte Löschen von temporären Dateien kann alternativ auch über Shutdown-Hooks realisiert werden. Da Shutdown-Hooks aber ganz allgemein eingesetzt werden können, sind sie eine eigene *unbekannte Kostbarkeit.*

Funktioniert das überall?

Wir haben das Beispiel mit OpenJDK 14, GraalVM 20 Native Image und WildFly 19 getestet und können die Funktionsfähigkeit bestätigen. Lediglich der Wert des System-Property im Native Image ist `/var/tmp`.

Zusammenfassung

Die Klasse `Files` erlaubt das Erzeugen temporärer Dateien, also Dateien, deren Namen generiert werden. Sie ermöglicht es, diese Dateien bei Programmende automatisch zu löschen, indem sie mit einem einfachen Methodenaufruf für diese Löschung registriert werden.

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Das Spring-Ökosystem: genug Material für ganze Regalmeter an Büchern

Rezension von Craig Walls "Spring im Einsatz" gelesen von Stefan Pfeiffer

Craig Walls macht es sich in seinem Buch „Spring im Einsatz" zur Aufgabe, einen Querschnitt durch die aktuelle Welt der Spring-Projekte zu zeigen, ohne wirklich alles erschöpfend behandeln zu wollen. Aber das ist auch nicht notwendig, um die typischen Anforderungen einer Businessanwendung abzudecken. So beleuchtet Walls einzelne Themenbereiche anhand einer Beispielapplikation, die im Laufe des Buchs immer weiterwächst und weitere Technologien, die rund um das Spring Framework existieren, einbindet.

In einem weiten Bogen vom Setup eines Spring-Projekts in den marktführenden IDEs, über die Grundlagen der Dependency Injection, des Application-Context-Konzepts, über Web-Endpunkte, Persistenz, Security, Messaging, Spring Integration, reaktive Programmierung, Spring Cloud bis hin zur Unterstützung im Betrieb durch Spring Boot, wird hier ein roter Faden geboten, dem man angenehm folgen kann. Einzig der knapp 100 Seiten umfassende Teil über reaktive Programmierung, der sicherlich die größte Änderung in der aktuellen Auflage ist, fällt hier ein bisschen aus der Reihe, weil die Motivation zur Einführung in die Beispielapplikation etwas schwach wirkt. Sicherlich kommt man um dieses Thema in einem Buch, das auch Spring 5 abdecken will, nicht herum.

Etwas verwirrend wirkt hingegen der Ansatz, ein relativ komplexes Modell von Entitäten zunächst durch Spring JDBC zu persistieren, um erst dann Spring-Data-Abstraktionen einzuführen. So stechen sicherlich die Vorteile von Spring Data deutlich heraus, man riskiert aber, hier schon erste Leser verloren zu haben, die vor der Komplexität des Handlings von Entitäten mit JDBC kapitulieren.

Der Autor versucht gar nicht erst, sauber zwischen dem Spring Framework und Spring Boot zu trennen. In modernen Spring-Projekten macht diese Trennung auch keinen Sinn mehr, da Spring Boot zum Synonym für Spring im Allgemeinen geworden ist. So können im Buch von Anfang an die Vorteile der Autokonfiguration und der leichten Ergänzung von Abhängigkeiten genutzt werden. Auch die Kapitel zu den Themen Konfigurationseigenschaften und Spring Boot Actuator integrieren sich nahtlos in den Verlauf des Buchs.

Ein heikles Thema ist sicherlich die Übersetzung englischer Bücher ins Deutsche und vor allem, wo diese Übersetzung enden sollte. Viele technologische Begriffe sind auch für deutsche Entwickler in ihrer englischen Form gesetzt und man kennt wahrscheinlich Circuit Breaker, Log Level oder das Deployment, das man nicht am Freitag machen sollte. Begriffe wie Trennschalter, Protokollierungsstufe oder Bereitstellung lesen sich eher holprig und erschweren auch das Finden weiterer Informationen zum Thema in einer Web-Suchmaschine. Eine gute Abwägung in einem deutschsprachigen Buch zu finden ist nicht einfach, manchmal scheint man aber über das Ziel hinausgeschossen zu sein. Zuweilen ärgern kleine Fehler, wenn beispielsweise Methodenparameter auf der nächsten Seite als Annotationen bezeichnet werden und man zurückspringen muss, um diese Verwechselung zu entlarven. Auch der bewusst lockere „amerikanische" Erzählstil gerade zu Beginn neuer Kapitel verliert manchmal seinen Charme und Witz und wirkt etwas fremdartig in einem deutschsprachigen Fachbuch.

Insgesamt bietet das Werk aber eine solide deutschsprachige Einführung in die Entwicklung einer realistischen webbasierten Business-Applikation mit serverseitig erzeugtem HTML unter Ausnutzung der relevanten Spring-Komponenten. Ein Entwickler, der erstmals oder nach längerer Abstinenz wieder mit Spring in Kontakt kommt, zieht sicherlich den größten Nutzen aus dem Buch, aber auch erfahrene Spring-Entwickler können in einzelnen Bereichen ihr Wissen vertiefen und neue Erkenntnisse gewinnen. Der neue Buchteil über reaktive Programmierung ist ebenfalls ein guter Einstieg in das Thema.

Stefan Pfeiffer

Autor: Craig Walls
Titel: Spring im Einsatz
Verlag: Hanser Verlag
Umfang: 559 Seiten
Preis: 54,99 Euro
ISBN 978-3-446-45512-2

Drum prüfe, wer sich bindet – eine Microservice-Checkliste

Jürgen Lampe, S&N Invent GmbH

Microservices sind ein mächtiges, aber ebenso anspruchsvolles wie aufwendiges Architekturkonzept. Einmal begonnen, ist es nur unter erheblichem Aufwand möglich, auf eine monolithische Struktur zurückzuschwenken. Dieser Artikel versammelt in Form einer Checkliste wichtige Fragen, die man rechtzeitig überdenken und beantworten sollte.

Jede Microservices betreffende Diskussion sieht sich mit der Unschärfe dieses Begriffs konfrontiert. Deshalb sei vorausgeschickt, dass es hier um das voll ausgeprägte Konzept mit konsequenter Trennung der Entwicklungsstränge geht. Die dazugehörende DevOps-Kultur mit umfassender Testautomatisierung und häufigen Deployments wird als unstrittig vorausgesetzt. Einige Bemerkungen zu Ansätzen, die das Konzept nur teilweise anwenden, finden sich am Ende. Wenn im folgenden Text von „Service" die Rede ist, so ist damit immer ein einzelner Microservice gemeint.

Eine folgenschwere Entscheidung

Der Entschluss, eine Microservices-Architektur aufzubauen, reicht in seinen Auswirkungen weit über andere Architekturentscheidungen hinaus, weil er die umgebende Organisationsstruktur unweigerlich mit einbeziehen muss. Das bedeutet, große Teile der Organisation sind mehr oder weniger davon betroffen und müssen diese Entscheidung in vollem Umfang unterstützen.

Ebenso wichtig ist es zu bedenken, dass ein solcher Entschluss nur unter erheblichem Aufwand rückgängig gemacht werden kann. Es ist im Zweifel günstiger, eine zunächst sauber modularisierte monolithische Applikation in Microservices zu zerlegen, weil dadurch nur geringe Mehrkosten entstehen. Beim umgekehrten Weg bleiben erhebliche, letztlich verschwendete Aufwände zurück (wenn man den Erfahrungsgewinn einmal ausklammert).

Deshalb ist es enorm wichtig, dass die im Folgenden aufgeworfenen Fragen gründlich bedacht und ehrlich beantwortet werden, bevor man eine so folgenschwere Entscheidung trifft. Natürlich wird es in vielen Fällen keine klare Ja-Nein-Antwort geben und mehr noch als bei anderen IT-Fragen spielt die Einbeziehung geschäftsstrategischer Gesichtspunkte eine wichtige Rolle. Schließlich ist nie zu vergessen, „dass Microservices kein Ansatz sind, um die Kosten der Software-Entwicklung zu minimieren. Es gehe vielmehr darum, die Reaktionszeiten – Stichwort „Time-to-Market" – sowie die Qualität der Systeme deutlich zu verbessern." *[1]*

Warum Microservices?

Diese erste, nur scheinbar triviale Frage sollte nicht zu leicht genommen werden. Der Aufbau einer Microservice-basierten Anwendung ist teuer. Dazu kommen, wie im Weiteren noch dargestellt wird, nicht zu ignorierende Folgekosten. Aber nicht nur deshalb sollte eine entsprechende Architekturentscheidung gut begründet sein, denn die Konsequenzen reichen weit über die IT hinaus und beeinflussen die Strategie über Jahre.

Alle wichtigen Vorteile von Microservices basieren auf drei Punkten:

1. **Schnelligkeit:** schnelle Entwicklung, schnelle Erweiterung der Ressourcen, schnelle Anpassung an den Markt
2. **Flexibilität:** einfacher Austausch von Komponenten und Technologien
3. **Skalierbarkeit:** gute Anpassbarkeit an stark wechselnde Bedürfnisse

Es gibt weitere positive Aspekte, deren Bedeutung jedoch geringer ist.

Die Vorteile bekommt man allerdings nicht geschenkt. Sie haben ihren Preis. Daher muss als Erstes geklärt werden, wie groß und wichtig der jeweils erzielbare Nutzen realistischerweise sein wird und welche Kosten man dafür akzeptieren kann oder will. Dieser Nutzen wird sich nicht immer monetär beziffern lassen, wenn es beispielsweise darum geht, eine Position im Markt zu besetzen oder zu behaupten. Trotzdem lässt sich die grundsätzliche Entscheidung, welchen Einsatz das angestrebte Ziel wert ist, nicht vermeiden, weil es um nennenswerte Beträge geht.

Sollen bestehende IT-Probleme gelöst werden?

Das wäre ein denkbar schlechter Ausgangpunkt, um mit dem Aufbau einer neuen komplexen IT-Struktur zu beginnen, weil in dieser Lage oft fachliche und organisatorische Probleme vorliegen, die besser vorher gelöst werden sollten. Wie bereits gesagt, lösen Microservices keine originären IT-Probleme, sondern können dabei helfen, bestimmten Anforderungen aus dem Geschäftsumfeld zu genügen. Eine Microservices-Architektur verlagert Komplexität, aber vermindert sie nicht. Und sie hilft ebenso wenig bei ungenügender Durchdringung oder Strukturierung des Anwendungsbereichs. Natürlich gibt es Ausnahmen, aber ob diese vorliegen, sollte sehr kritisch geprüft werden.

Ist die Organisation vorbereitet?

Jeder Microservice verkörpert eine Anwendungssäule. Wenn die Vorteile der unabhängigen Entwicklung zum Tragen kommen sollen, muss die Organisationsstruktur entsprechend vorbereitet sein. Conways Gesetz „Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." fordert sein Recht.

Daher ist es gefährlich, wenn versucht wird, Microservices als reines IT-Projekt einzuführen. In den seltensten Fällen wird die Organisationsstruktur schon ausreichend den Erfordernissen entsprechen. Je weiter die bestehende Struktur vom erforderlichen Zielbild entfernt ist, desto aufwendiger und riskanter wird das Projekt. Auch die gleichzeitige Umwandlung der Organisationsstruktur ist wegen der schwer beherrschbaren Komplexität kritisch und nur bei konsequenter Unterstützung durch das gesamte Management überhaupt umsetzbar. Manchmal funktioniert es, einen neuen Organisationszweig quasi parallel aufzubauen, der dann schrittweise Funktionen aus der „alten Welt" übernimmt. Im Vorteil sind in diesem Punkt Startups, die ihr Geschäftsmodell gerade erst entwickeln.

Ist die notwendige Unterstützung des Managements gegeben?

Eine für Microservices geeignete Organisationsstruktur erfordert zwangsläufig auch eine entsprechende Dezentralisation von Verantwortung und Entscheidungen. Das ist vor allem deshalb eine große Herausforderung, weil die gängige Praxis gewöhnlich anders aussieht. Unabhängige Entwicklung führt ganz natürlich zu vielen Redundanzen – das ist das genaue Gegenteil vom „Heben von Synergien". Auf allen Ebenen muss deshalb vorbehaltlos akzeptiert werden, dass jedes Entwicklungsteam frei entscheiden kann, beispielsweise auch über die zu verwendenden Arbeitsmittel. Ebenso müssen die einzelnen Geschäftsbereiche ihre Anforderungen ohne lange Abstimmung an das jeweilige Entwicklungsteam übergeben können.

Sind die übergreifenden Prozesse hinreichend stabil?

Bei aller Dezentralisierung wird es einige übergreifende Geschäftsprozesse geben – sonst wäre es ja nicht *ein* Anwendungsprojekt.

Häufig muss ein zentrales Buchungs- und Abrechnungssystem bedient werden. Es kann sich aber ebenso um eine gemeinsame Weboberfläche handeln. Und in jedem Fall gehört ein umfassendes Sicherheits- und Berechtigungskonzept dazu.

Alles, was bei Änderungen über einen einzelnen Service hinausreicht, ist kritisch. Wegen der dabei notwendigen Synchronisation verursachen solche Modifikationen enormen Aufwand und müssen sorgfältig geplant werden. Ihre Wahrscheinlichkeit sollte daher von vornherein minimiert werden. Aus diesem Grund ist ein möglichst stabiles (weiteres) Umfeld eine nicht zu vernachlässigende Voraussetzung. Das ist nun ein Punkt, an dem Startups eher im Nachteil sind, weil diese Stabilität nur durch Erfahrung erreicht werden kann.

Sind die Geschäftsprozesse ausreichend fehlertolerant?

Niemand sollte sich etwas vormachen: Fehler passieren. Wenn Services häufig und unabhängig produktiv gehen – und das ist ja eines der Kernprinzipien – wird es sich nicht vermeiden lassen, dass es, möglicherweise auch an ganz unerwarteten Stellen, zu Fehlern kommt, weil es unter anderem nicht möglich ist, alle Wechselwirkungen im Vorfeld zu testen. Damit muss man leben können, denn es gibt in diesem Paradigma keine Stabilisierungsphase wie bei herkömmlichen Projekten, an deren Ende alle schweren Fehler gefunden und beseitigt sind. Selbst umfassende Tests werden nie die vollständige Korrektheit des Codes garantieren können.

Die notwendige Fehlertoleranz im Geschäftsbereich ist deshalb ein entscheidendes Kriterium, das Microservices in ihrer konsequenten Umsetzung für bestimmte Anwendungen faktisch ausschließt. Denn während der Schaden, den ein abgebrochener Verkaufsvorgang oder eine Fehllieferung beim Kunden verursacht, durch einen Bonus beispielsweise relativ einfach behoben werden kann, ist das bei juristisch relevanten Vorgängen wie termingemäßen Überweisungen und Erteilungen oder Versagungen von Genehmigungen deutlich schwieriger oder sogar unmöglich.

Sind unsichere Antwortzeiten hinnehmbar?

Im Netz ist nichts garantiert. Wenn Services miteinander kommunizieren, dauert das nicht nur wesentlich länger als bei internen Prozeduraufrufen, es kann jederzeit auch zu Verbindungsabbrüchen oder extrem langen Wartezeiten kommen. Vorgegebene Antwortzeiten lassen sich also nur mit einer gewissen – und von vielen Einflüssen abhängigen – Wahrscheinlichkeit gewährleisten. Für eine Verkaufsanwendung wird das im Allgemeinen kein Problem sein. Ein Wertpapier-Hochfrequenzhändler sieht das aber möglicherweise anders.

Ist die Zielstellung angemessen?

Die gute Parallelisierbarkeit der Entwicklung auszunutzen, ist nicht billig. Praktisch verursacht jeder Service für sich den kompletten Verwaltungsaufwand eines (kleinen) Projekts. Es dauert darüber hinaus in der Regel einige Zeit, bis sich die Teams gefunden haben und voll leistungsfähig werden. Die danach beginnende produktive Phase sollte nicht zu kurz sein, das heißt, jeder Service muss genügend Entwicklungspotenzial für einen längeren Zeitraum aufweisen, um den hohen initialen Aufwand zu rechtfertigen. In Ausnahmefällen ist es allerdings denkbar, dass ein extern verursachter Zeitdruck (Time-to-Market) Kostenfragen zweitrangig erscheinen lässt.

Es ist auch zu bedenken, dass sich die Vorteile der häufigen Deployments nur dann wirklich nutzen lassen, wenn die Anwendung in sinnvoller Weise bereits mit einem kleinen Teil der vorgesehenen Funktionen produktiv genommen und in der Folge über einen längeren Zeitraum schrittweise erweitert werden kann.

Ist die fachliche Domäne gut genug verstanden?

Für die Microservices-Architektur muss die Anwendung in möglichst unabhängige *fachliche* Säulen oder Silos zerlegt werden. Das

gelingt nur dann erfolgreich, wenn die Fachdomäne ausreichend gut verstanden ist, weil die Abgrenzung der Säulen die logische Voraussetzung für die Zerlegung ist. Die Schnittstellen gehören natürlicherweise zur das Gesamtprojekt umfassenden Struktur. Damit gilt für sie ganz besonders die bereits getroffene Aussage, dass spätere Änderungen teuer und daher möglichst zu vermeiden sind. Das heißt, dass sie bereits frühzeitig klar gefasst werden müssen. Um qualitativ hochwertige Schnittstellen formulieren zu können, sind jedoch umfassende Kenntnisse nötig, und zwar mit Sicht über den aktuellen Stand der Verhältnisse hinaus – denn die Welt bleibt nicht stehen.

Diese Schnittstellen definieren auf der einen Seite die Architektur und geben der Anwendung ihre Struktur. Auf längere Sicht kann daraus aber auch ein Korsett erwachsen, das die weitere Entwicklung behindert. Da die Zukunft nicht vorhersehbar ist, lässt sich dieses Manko zwar nicht völlig vermeiden, aber ein möglichst tiefes Wissen hilft, den kritischen Zeitpunkt weiter in die Zukunft zu verschieben.

Nach diesen eher geschäftspolitischen Kriterien geht es im Folgenden vorrangig um technische Fragestellungen, die natürlich zumindest über die Kosten mit den ersteren verbunden sind.

Sind die technischen Voraussetzungen erfüllbar?

Microservices verlagern einen Teil der Komplexität aus dem Anwendungscode in die Struktur des Systems, wo sie durchaus nicht leichter zu beherrschen ist. Daraus resultieren entsprechend hohe Anforderungen beim Aufsetzen und bei der Verwaltung der technischen Infrastruktur:

- Heterogenität durch die (mögliche) Vielzahl der eingesetzten Technologien
- Eigene Backup- und Recovery-Prozesse für jeden Microservice
- Monitoring und Logging aller Services
- Skalierung und Resilienz (Konfigurationsautomatisierung)
- Sicherheit, zum Beispiel Zertifikatsverwaltung (siehe Sicherheitsanforderungen)

Diese Aufgaben sind nicht nur zeit- und kostenintensiv, sie erfordern auch spezielle Fähigkeiten und Erfahrungen. Zudem muss ein erheblicher Teil davon, zum Beispiel für Updates, Erweiterungen und Migrationen, über die gesamte Lebenszeit der Anwendung erbracht werden. Insbesondere können neu entdeckte Schwachstellen (Exploits) jederzeit ungeplante Aufwände verursachen. Die nicht unerheblichen Kosten der benötigten Verwaltungswerkzeuge dürfen ebenfalls nicht vergessen werden, denn die gratis angebotenen Funktionen sind für den produktiven Betrieb selten ausreichend. Hinzu kommt, dass Fachleute mit den benötigten Kenntnissen begehrt und rar sind.

Kann die zusätzliche Komplexität beherrscht werden?

Verteilte Programme machen auch den eigentlichen Anwendungscode komplizierter. Im Gegensatz zu einem einfachen Prozeduraufruf kann beim Zugriff auf einen Service einiges schiefgehen, was sich nicht allein auf der technischen Ebene abfangen lässt. Wie die Reaktion auf einen Timeout oder einen unberechtigten Zugriffsversuch aussehen muss, ist oft eine fachliche Frage. Und je nachdem, ob der Zugriff synchron oder asynchron erfolgt, muss die Anwendungslogik anders aufgebaut werden. Die Berücksichtigung dieser Kommunikationsanforderungen macht die Programmierung schwieriger.

Können die Sicherheitsanforderungen gewährleistet werden?

Einen einzelnen Computer kann man einschließen – eine Microservices-Anwendung nicht. Jeder Service steht im Netz und stellt damit einen potenziellen Angriffspunkt dar. Auch eine eventuelle Begrenzung aufs Intranet reduziert das Risiko nur graduell. Ein Teil der Sicherheitsanforderungen kann bereits auf der Ebene der technischen Infrastruktur gewährleistet werden, aber es bleiben Fragen, die rechtzeitig geklärt werden müssen:

1. Welche Rechte und Rollen brauchen die Services und wie müssen diese unter Umständen abgestuft werden? Wo werden diese Rechte und Rollen verwaltet?
2. Wie werden die Berechtigungen durch die Services geprüft? Wie erfolgt die Weitergabe von Rechten?
3. Wie kritisch sind die übertragenen Daten? Ist eventuell eine zusätzliche Verschlüsselung erforderlich?
4. Wie wird gegebenenfalls verhindert, dass bei der Arbeit für einen Mandanten im Namen eines anderen auf Services zugegriffen werden kann?
5. Ist es möglich, allein durch Analyse der Aufruffolgen, Nachrichtenlängen oder Ähnlichem sensible Informationen zu gewinnen? Muss das verhindert werden?

Die Liste ist nicht vollständig, sie soll nur zeigen, dass die Anforderungen an Berechtigungs- und Sicherheitskonzepte nicht unterschätzt werden dürfen.

Als übergreifender Gesichtspunkt muss ein Sicherheits- und Berechtigungskonzept mit hoher Priorität und so früh wie möglich erstellt werden, um rechtzeitig prüfen zu können, ob die zu gewährleistende Sicherheit mit vertretbaren Mitteln überhaupt erreicht werden kann beziehungsweise welche Punkte bei Verträgen mit externen Dienstleistern zu regeln sind.

Sind die Leistungsanforderungen realisierbar?

Microservices skalieren zwar gut mit der Anzahl der Anfragen, aber es wird leicht übersehen, dass ein einzelner Service für sich genommen fast immer längere Antwortzeiten aufweist, als das bei einer monolithischen Anwendung der Fall wäre. Dafür gibt es zwei Gründe. Der erste ist die bereits erwähnte Netzwerklatenz beim internen Aufruf weiterer Services. Dazu kommen jedoch auch noch zusätzlich notwendige Arbeitsschritte vor beziehungsweise nach jedem Serviceaufruf:

- Datenkonvertierung in oder aus einem meist textbasierten Format (JSON, XML etc.) beim Empfangen und Senden
- Ver- und Entschlüsselung, falls notwendig
- Überprüfung der Gültigkeit der Anfrage und der übermittelten Rechte
- Validierung der erhaltenen Daten

Bereits daraus ergeben sich Grenzen für die Antwortzeit, die selbst bei optimaler Netzwerkleistung nicht unterschritten werden können. Und

natürlich schlagen sich diese zusätzlichen Aufgaben auch im Hauptspeicherbedarf und der beanspruchten Prozessorleistung nieder.

Ist die mehrfache Realisierung von Funktionen akzeptabel?

Bei der parallelen und unabhängigen Entwicklung der Services besteht immer das Risiko, dass einzelne Funktionen nicht rechtzeitig oder nicht in der notwendigen Qualität fertig werden. Wenn Funktionen für die Gesamtanwendung essenziell sind, kann es sinnvoll sein, diese mehrfach implementieren zu lassen.

Eine ähnliche Situation ergibt sich, falls eingesetzte Fremdsoftware oder eine Programmiersprache zukünftig nicht mehr verwendet werden können oder sollen. Dieser Mehraufwand muss toleriert werden. Gleichzeitig ist es Aufgabe einer geschickten Personalführung, dafür zu sorgen, dass durch das Ersetzen von Services deren Entwickler nicht demotiviert werden.

Ist der notwendige Testumfang realisierbar?

Microservices benötigen umfangreichere Tests als monolithische Anwendungen. Das liegt einmal daran, dass jeder Service unabhängig geprüft werden muss. Da es keine umfassenden Integrationstests gibt, entfällt das implizite „Mitprüfen" von Teilfunktionen. Jeder Service braucht seinen eigenen, möglichst gut abdeckenden Test.

Zum anderen müssen alle im Zusammenhang mit der Kommunikation stehenden Funktionen abgedeckt werden. Dazu gehören die sicherheitsrelevanten Aufgaben ebenso wie Datenkonvertierungen und -verschlüsselungen. Außerdem muss die korrekte Reaktion auf Verbindungsfehler und -störungen getestet werden. Automatisierte Tests dieser zweiten Gruppe sind meist nicht trivial zu erstellen und verursachen erheblichen Entwicklungsaufwand.

Im dritten Abschnitt geht es nun um die entwicklerbezogenen Voraussetzungen, die ebenfalls nicht leichtgenommen werden sollten.

Sind die benötigten Entwickler in überschaubarer Zeit verfügbar?

Software wird von Menschen gemacht. Um schnell parallel entwickeln zu können, werden ausreichend Entwickler gebraucht. Dabei ist zu berücksichtigen, dass für diese Arbeit einige besondere Voraussetzungen erfüllt sein müssen. Gerade in kleinen und selbstständig arbeitenden Teams spielen Persönlichkeitseigenschaften und Erfahrungen eine große Rolle. Einerseits sollten nicht zu viele Neulinge (mit häufig unrealistischen Vorstellungen) vertreten sein, andererseits darf die Flexibilität und Freude am Neuen nicht völlig durch gewohnten Trott verschüttet werden. Insbesondere letztere Gefahr macht es bisweilen schwierig, die benötigten Entwickler einfach durch Umsetzung aus einer hierarchisch organisierten Projektorganisation, die Eigeninitiativen nur bedingt fordert, zu gewinnen.

Sind die Entwickler bereit, als Full-Stack-Entwickler zu arbeiten?

Microservices brauchen Full-Stack-Entwickler, die von der Erfassung und Formulierung der Anforderungen bis zur Sicherstellung des Betriebs alle Tätigkeiten beherrschen. Vor allem die Auseinandersetzung mit der Fachlichkeit und den fachlichen Auftraggebern kann sich als schwierig erweisen. Dabei sind gute kommunikative

Fähigkeiten gefragt. Darüber hinaus gilt es, sprachliche Hürden zu überwinden, nicht nur bei verschiedenen Muttersprachen, sondern auch im Hinblick auf das Verständnis des Anwendungsgebiets. Je komplexer das Anwendungsgebiet ist, desto langwieriger wird die notwendige fachliche Einarbeitung und damit wächst möglicherweise die Schwierigkeit, Entwickler für diese Spezialisierung zu motivieren. Dazu kommt als eher organisatorische Aufgabe die Sicherung einer effektiven Zusammenarbeit mit den jeweils betroffenen Fachbereichen.

Kann ein ausreichend motivierendes Umfeld gesichert werden?

Die Freiheiten bei der Entwicklung und schnelle Rückmeldungen zu den erreichten Ergebnissen wirken für sich bereits stark motivierend. Trotzdem sind auf Dauer weitere Maßnahmen zur Sicherung der Motivation nötig, denn man darf nicht vergessen, dass den Entwicklern auch einiges abverlangt wird. Neben der notwendigen Vertrautheit mit der Fachlichkeit muss der produktive Betrieb durch Bereitschaftsdienste oder Ähnliches abgesichert werden. Solche Fragen müssen durch angepasste und flexible Vergütungs- und Arbeitszeitmodelle angemessen gelöst werden. Eine übermäßige Fluktuation wirkt sich in den meist kleinen Teams besonders nachteilig aus.

Zum Abschluss noch eine Betrachtung, die auch eine Warnung an alle ist, die denken, dass das doch auch mit viel weniger Aufwand realisierbar sei. Natürlich lässt sich einiges einsparen – aber nicht ohne Folgen.

Microservices „im Kleinen“

Die genannten Herausforderungen sind selbstverständlich nicht unbemerkt geblieben. Deshalb gibt es viele Versuche, den Aufwand zu verkleinern, indem auf Redundanzen verzichtet wird. Das ist jedoch nur in beschränktem Umfang sinnvoll möglich. In vielen Situationen sind Abstriche an der freien Wahl von

- Programmiersprachen,
- Versionsverwaltung,
- Build-Werkzeug oder
- Datenbank-Software akzeptabel.

Schwieriger wird es, wenn in größerem Ausmaß standardisierte Bibliotheken eingesetzt werden, weil dadurch abgestimmte Versionswechsel notwendig werden. Es gibt weitere Möglichkeiten, die Unabhängigkeit der Service-Entwicklungen zu verringern. Allen gemeinsam ist, dass dabei die eingangs genannten Vorteile von Microservices zunehmend verlorengehen, und zwar ohne den Aufwand in vergleichbarem Umfang zu senken. Denn der größte Kostenblock, der durch die in die Struktur verlagerte Komplexität verursacht wird, bleibt davon unberührt. Alles, was mit dem Netzwerk zusammenhängt, muss in gleicher Weise eingerichtet und verwaltet werden. Allein deshalb ist es ein kostspieliger Ansatz, Microservices nur zu verwenden, um eine saubere Modularisierung zu erzwingen.

Das spricht nicht grundsätzlich gegen kleine Microservices-Anwendungen. Vor allem zum Erfahrungsgewinn sind sie nützlich. Der wirkliche Nutzen entsteht jedoch erst bei konsequenter Umsetzung, und das bedingt fast immer einen größeren Rahmen.

Fazit

Die aufgeführten Punkte stellen eine sicher unvollständige Liste von Kriterien dar, die bei einer Entscheidung für Microservices berücksichtigt werden sollten. Es geht nicht darum, diese Technologie als solche zu bewerten. Vielmehr soll diese Zusammenstellung helfen, die Chancen und Risiken des Einsatzes in konkreten Situationen realistisch einzuschätzen. Ein beeindruckendes Beispiel dafür, wie man es (unter den richtigen Voraussetzungen) richtig macht, wird auf GitHub *[2]* vorgestellt.

Quellen

[1] Röwekamp, Lars (2015): Microservices: (noch) keine einheitliche Definition. *https://jaxenter.de/microservices-keine-einheitliche-definition-16683*

[2] Gauder, Sebastian (2019): 5 years of food retail e-commerce. A microservice success story. Präsentation DEVelopers Köln Meetup, *https://github.com/devk-insurance/devk-meetups/blob/master/microservices-bright-vs-dark/2019-11-20-devk-meetup-microsevices-brightside.pdf*

Jürgen Lampe

S&N Invent GmbH Eschborn

juergen.lampe@sn-invent.de

Dr. Jürgen Lampe ist IT-Berater bei der S&N Invent GmbH. Er ist promovierter Mathematiker und war unter anderem als Hochschuldozent tätig. Seit mehr als 20 Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen, hauptsächlich im Bankenumfeld. Sein spezielles Interesse gilt effizienten anwenderorientierten Softwarearchitekturen und domänenspezifischen Sprachen. Neben seiner Tätigkeit als Senior-Berater schreibt er Artikel für Fachzeitschriften und spricht auf Konferenzen. Er ist Autor des Buches „Clean Code für Dummies“.

Eine kurze Geschichte über Microservices

Dominik Galler, esentri AG

Diese fiktive Geschichte handelt von Carolin. Carolin ist eine aufstrebende, junge Informatikerin, die soeben ihre neue Stelle als Projektkoordinatorin für die Entwicklung einer Microservices-Anwendung angetreten ist. Der alte Projektleiter musste die Firma verlassen, denn das von ihm gegründete Microservices-Projekt fährt aktuell mit Vollgas gegen die Wand. In der folgenden Geschichte begleiten Sie Carolin bei der Problem- und Lösungsfindung in ihren ersten Monaten.

Der Status quo

Bereits vor Antritt der Stelle wurde Carolin die Vision des Unternehmens nähergebracht. Mit Beginn des Microservices-Projekts wurden mehrere Projektteams gegründet, die vollständig cross-funktional aufgestellt sind und agil nach Scrum arbeiten. Das bedeutet, dass die Teams Anforderungserhebung, Abbildung der Fachlichkeit, Wissen über die Programmierung und den Betrieb in allen Teams gleichmäßig – man möchte fast sagen optimal – verteilt haben. Somit ist jedes Team in der Lage, seine eigenen Microservices zu entwickeln und zu betreiben.

Um die Eigenständigkeit der Teams weiter zu erhöhen, hat sich das Unternehmen von einem projektgetriebenen bereits zu einem produktgetriebenen Entwicklungsansatz entwickelt. Damit ist gemeint, dass die einzelnen Microservice-Entwicklungsteams ihre eigenen Produkte, in der Regel ihre Microservices, den anderen Teams und schließlich dem gesamten Unternehmen zur Verfügung stellen. Um zukunftsfähig zu sein, setzt das Unternehmen auf eine Cloud-Strategie. Die Microservices werden von den Entwicklerteams in der Cloud-Umgebung betrieben. Die Ausführungsmodelle der Microservices reichen von virtuellen Maschinen über Container bis hin zu Serverless Computing.

Um ein inkrementell-iteratives Vorgehen zu gewährleisten und den Kunden in schnellen Zyklen neue Funktionen zur Verfügung zu stellen, setzt das Unternehmen auf Continuous Delivery mit drei Stages: Auf der Entwicklungsstage können die Entwickler den Programmcode jederzeit deployen. Auf der Abnahme-Stage werden Abnah-

men und Tests durchgeführt. Die finale Stage ist die Produktiv-Stage. Diese wird auch von den Kunden genutzt.

Nach einigen Monaten anfänglicher Euphorie haben einige Teams immer größere Probleme in der Entwicklung. Carolin begibt sich daher in ihren ersten Projektmonaten auf Fehlersuche.

Von Komplexität, die nicht einfach verschwunden ist

Basierend auf dem (vielleicht nicht ganz so) alten Spruch, aber dafür viel älteren Prinzip des „divide et impera" (lateinisch für „Teile und Herrsche") sollen Microservices komplexe Probleme durch das Herunterbrechen auf einfache (Teil-) Probleme schnell und verständlich für kleine Entwicklerteams lösbar gestalten. Die Annahme, die Komplexität, die häufig für Herausforderungen in der Entwicklung sorgt, sei hiermit verschwunden, stellt sich bei genauerer Betrachtung jedoch als falsch heraus. Sie wurde lediglich in die Netzwerkschicht verschoben und ist im reinen Programmcode nur noch indirekt zu finden. Auf der Netzwerkebene prägt sich die vormals auf Code-Basis vorliegende Komplexität nun in ebenso komplexen Kommunikationschoreografien aus *[1]*.

Dies schafft neue Herausforderungen für die Entwickler. Konnten diese in einem monolithischen System noch die komplette Anwendung (in der Regel sogar lokal, unterstützt durch die IDE) ausführen und debuggen, so ist das heute mit vielen kleinen Services entweder logistisch oder technisch (aufgrund von Ressourcen) nicht mehr möglich. Bei der Fehlersuche müssen die Entwickler die Kommunikationsketten, die sich über viele Microservices spannen können, nachvollziehen. Dafür benötigen sie eine zentrale Anlaufstelle. An dieser können die Log-Dateien (häufig das wichtigste Artefakt bei der Fehlersuche) aggregiert zur Verfügung gestellt werden.

Leider wurde eine solche Schnittstelle in Carolins Großprojekt nicht geschaffen. Alle Microservices schreiben ihre Log-Ausgaben in ihren Standard-Output und dort werden diese auch belassen. Infolgedessen müssen die mit der Fehlersuche betreuten Entwickler die Logs mühselig bei verschiedenen Teams einsammeln. Durch den zunächst gut gemeinten Gedanken, dass jedes Entwicklerteam komplett frei über die Technologieauswahl entscheiden darf, haben die Teams zudem völlig unterschiedliche Logging-Formate gewählt. Carolin zählt nach. Es gibt zwölf verschiedene (stellenweise komplett disjunkte) Attributmengen und fünf verschiedene Ausgabeformate für die Logs – ein Albtraum.

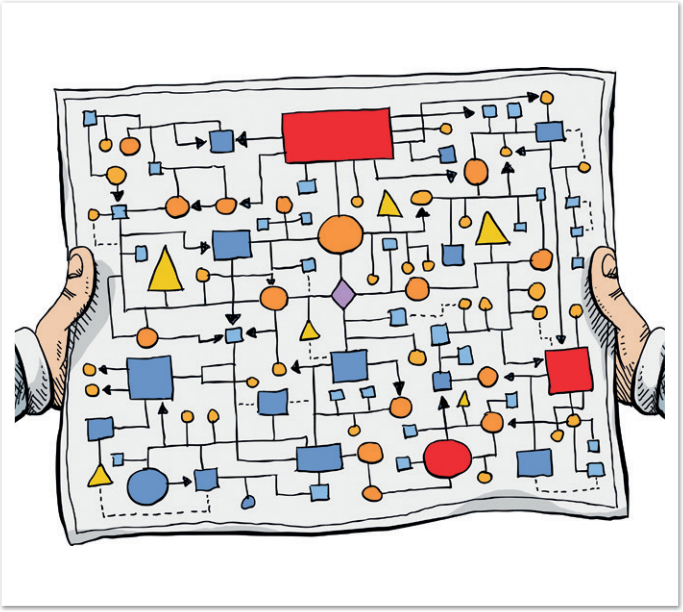
Diese Probleme können durch Vorgabe von JSON als Logging-Format, eine einheitliche Attributmenge für die Logausgaben sowie eine zentrale Log-Aggregierung mit komfortabler Suchfunktion (wie beispielsweise Kibana) gelöst werden.

Für die Entwickler sollten diese Schritte bereits eine große Hilfe sein, doch die Fehlersuche dauert leider immer noch viel zu lange. Ein Blick auf die aggregierten Logs lässt Carolin sofort erkennen, wo das Problem liegt. Die Nachrichten können nicht über die Grenzen einer Anwendung heraus zugeordnet und verfolgt werden. Dieses Verfolgen der Nachrichten wird für gewöhnlich Tracing genannt und erfolgt in der Regel über eindeutige Bezeichner für eine Nachricht, die sogenannte Tracing ID *[2]*. Diese ID wird einer Nachricht – bei HTTP beispielsweise über den Header – mitgegeben. Bei allen folgenden Nachrichten, die dieser Nachricht entsprungen sind, wird dann dieselbe ID weitergegeben. Damit lassen sich Nachrichtenketten viel einfacher verfolgen. Die Tracing ID lässt sich den Nachrichten zum Beispiel in einem Loadbalancer hinzufügen. Nun können Entwickler – da sie in diesem Zuge direkt auch noch Logausgaben bei den Ein- und Austrittspunkten ihrer Services einpflegen – die Anfragen nachvollziehen und durch die zentrale Anlaufstelle effektiv nach Fehlern suchen.

Komplexität entsteht in Microservices-Anwendungen noch an einer ganz anderen Stelle, und zwar auf Ebene der Infrastruktur. Wie wir bereits herausgefunden haben, findet ein entscheidender Teil der Problemlösung bei Microservices-Anwendungen auf Kommunikationsebene der einzelnen Services statt. Durch diese Kommunikationen kommt der Infrastruktur eine hohe Bedeutsamkeit zu. Diese wird noch erhöht, wenn Autoscaling, Load-Balancing und dynamische Ressourcen-Allokation automatisiert werden sollen, um entsprechende Lastspitzen autonom abzufangen (etwa am Wochenende). Darüber hinaus soll erst gar nicht verschwiegen werden, dass jeder Microservice mit einem eigenen bunten Strauß an Infrastrukturkomponenten, unter anderem auch für dieselben Aufgaben, daherkommt. Das gilt ebenso für Carolins Projekt.

Damit gibt es für die (Dev)Ops-Teammitglieder eine mannigfaltige Aufgabenwelt, in der viel Spezialwissen vonnöten ist, um die oftmals kleinen, aber feinen Unterschiede zu behandeln. Hierbei führt ein im Betrieb von Infrastruktur weit bekanntes Phänomen zu besonderen Problemen: der Configuration-Drift. Dabei schlägt sich das Spezialwissen Einzelner in immer mehr kleinen Konfigurationsänderungen am laufenden System nieder *[3]*. Das zeigt sich auch in diesem Projekt. Durch eine Elternzeit und zwei Positionswechsel ist es den Teams von drei Kernprojekten aktuell nahezu unmöglich, die Anwendung weiterzuentwickeln und auf Probleme im Betrieb zu reagieren. Aus Angst, etwas zu zerstören und anschließend nicht reparieren zu können, verzichten die drei Teams bereits seit vier Wochen auf ein Deployment. Es weiß einfach niemand, wie das im Detail funktioniert.

Um dies zukünftig zu verhindern, führt Carolin mit sofortiger Wirkung „Everything-as-Code" ein. Everything-as-Code erfordert, dass



jede Infrastrukturkomponente durch Code beschrieben wird. Es sollte keine dynamische CLI-Interaktion durch Menschen erfolgen und damit auch keine manuellen Aufgaben (abgesehen vielleicht vom Aufbauen und Verkabeln von Servern). Somit sollte niemand mehr – außer der Build- und Deployment-Server – in direkter Verbindung mit der Infrastruktur interagieren müssen. Darüber hinaus sollte der Code für die Infrastruktur demselben Qualitätsprozess unterzogen werden wie der Programmcode. Dazu zählen insbesondere Code-Reviews, Coding-Rules, Namenskonventionen und Versionskontrolle für die Dateien, welche die Infrastruktur beschreiben. Damit ist das Wissen auf viele Köpfe verteilt und durch den Quellcode selbst ohne Dokumentation nachvollziehbar. Die Änderung der Infrastruktur kann anhand des Versionsverwaltungssystems nachvollzogen werden und ist somit transparent. Da Carolins Arbeitgeber nun um die Tragik von Silowissen weiß, wird außerdem bei jedem Produktivgang ein anderer Entwickler das Deployment durchführen und überwachen. Somit haben alle Entwickler Erfahrung in der Interaktion mit der Infrastruktur.

Von den Auswirkungen der Skalierbarkeit

Warum wurde überhaupt entschieden, eine Microservices-Anwendung zu bauen? Carolin hofft inständig, dass der ehemalige Projektleiter die Anforderung erfüllen wollte, dynamisch zu skalieren oder wenigstens Entkopplung und Parallelisierung des Entwicklungsprozesses im Sinn hatte. Sonst ließe es sich nur schwer vertreten, nicht mit einfacher zu handhabenden, verteilten Systemen oder Monolithen zu arbeiten.

In diesem Projekt müssen spontan zu jeder Zeit auftretende Lastspitzen in einzelnen Systembereichen durch dynamische Skalierung abgefangen werden. Dafür ist es wichtig, eines in Microservices-Anwendungen nie zu vergessen: Was passiert, wenn es mehr als eine Instanz eines Service gibt? Diese Frage schreibt Carolin auf ein großes Plakat und zeigt es den Entwicklerteams. Es offenbart sich ein potenzieller weiterer Grund für die Probleme im Projekt. Die Antwort auf diese Frage kann nicht jeder geben. Auf den Rechnern der Entwickler und auch auf der Entwicklungs- und Pre-Produktiv-Stage gibt es zur Kosteneinsparung von jedem Service nur eine Instanz. „Das ist ja auch kein Produktivsystem und wir müssen sparen", hatte der ehemalige Projektleiter bei der Übergabe gesagt.

Da die Entwickler also keine direkt zugängliche Umgebung mit mehreren Instanzen pro Service haben, haben viele Entwickler der Einfachheit halber Zustände und manche Teams sogar Sessions eingebaut. Und schon hat Carolin den nächsten Baustein im lahmenden Projekt gefunden. Die Gründe, weswegen man weder Zustände noch Sessions mit Microservices abbilden sollte, sind Carolin natürlich bewusst. Sie selbst hat diese Erfahrungen ebenfalls schmerzlich machen müssen, als sie noch als Entwicklerin tätig war.

Welcher konkrete Service eine Anfrage bekommt, ist nämlich unklar, wenn es mehr als eine Instanz pro Service gibt und zwischen diesen Instanzen durch Loadbalancing vermittelt wird. Welche Instanz die folgende – auf der ersten Anfrage aufbauende – Nachricht bekommt, ebenso. Bei asynchronen Aufrufketten ist außerdem unklar, wer die Antwort(en) erhält. Carolin führt deshalb eine weitere Grundlage ein: keine Zustände und auch keine Sessions in Services. Ausnahmen müssen begründet, dokumentiert und in der Architekturdokumentation festgehalten werden.

Von Vertrauen in das System

Obwohl die Fehlersuche im System bereits erheblich erleichtert wurde und die Infrastruktur nun deutlich leichter zu beherrschen ist, muss immer noch eine beachtliche Anzahl an Entwicklern am Wochenende auf Bereitschaft sein. Das betrifft die meisten Teams, da immer wieder wichtige Services Probleme haben und abrauchen. Meistens sind Bugs oder Konzeptionsfehler dafür verantwortlich und machen einen Neustart oder sogar ein Zurückrollen der Datenbank nötig, um den Fehlzustand zu korrigieren. Diese hohen Ressourcenkosten belasten das Unternehmen, aber auch die Entwicklerteams selbst. Auch hier weiß Carolin Rat und schlägt vor, Chaos-Engineering in den Entwicklungsprozess einzubetten.

Was aber ist Chaos-Engineering *[4]* und warum sollte man es einsetzen? Die Entwickler sollten ihren Code in Microservices-Projekten auch selbst betreiben (vergleiche DevOps). Dafür müssen sie aber folgende Fragen zuverlässig beantworten können: Kann ich über das Wochenende in die Berge fahren, ohne Internet und Telefon und ohne, dass das Produktionssystem stirbt und nicht repariert werden kann? Und wenn es kaputt geht, was muss ich tun, damit ich es reparieren kann? Die Antworten auf diese Fragen lassen sich mit Chaos-Engineering sicher nicht abschließend klären, aber gewisse Tendenzen lassen sich schon zu Gewissheiten verdichten.

Wie kann Chaos-Engineering aber nun durchgeführt werden und was macht man mit den einzelnen Elementen? Zunächst muss Carolin die Frage beantworten, wo Chaos-Engineering eingesetzt werden soll. Bei Netflix beispielsweise wird Chaos-Engineering im Produktivsystem eingesetzt *[5]* (Anmerkung: Das liegt natürlich auch an den fehlenden Stages bei Netflix, da Netflix Canary-Releases einsetzt *[6]*). Allerdings hat Netflix eine gewisse Vorreiterrolle im Umgang mit Microservices und darüber hinaus ist Netflix ein weltweit agierendes Tech-Unternehmen mit rund 20 Milliarden US-Dollar Umsatz.

Das Unternehmen von Carolin ist nicht so groß und eher etwas vorsichtiger. Daher entschließt sie sich, Chaos-Engineering auf der letzten Stage vor Produktiv einzusetzen, da diese nach den bereits gewonnenen Erkenntnissen zur Skalierbarkeit nun der Produktiv-Stage weitestgehend entspricht. Durch die mittlerweile hohe Vertrautheit mit der heterogenen Softwarelandschaft braucht Carolin einige Tools und Frameworks, um alle Bereiche mit Chaos-Engineering abzudecken. Die Grundlage bildet der Chaos Monkey von Netflix *[7]*. Der Chaos Monkey war früher Teil der Simian Army im Netflix-OSS, ist mittlerweile aber herausgelöst, da die Simian Army von Netflix nicht weiterentwickelt wird. Es gibt ebenfalls eine Portierung auf Spring beziehungsweise Spring Boot *[8]*. Der Chaos Monkey terminiert dabei nach festgelegten Rahmenbedingungen laufende Instanzen in der Microservices-Umgebung. Dadurch kann überprüft werden, wie diese auf derartige Ausfälle reagiert.

Neben Spring Boot werden auch viele Docker-Container in Carolins Projektumfeld eingesetzt. Daher entschließt sie sich, neben dem Chaos Monkey Pumba *[9]* einzusetzen. Pumba kann man sich als Chaos Monkey für Docker-Container vorstellen. Damit lassen sich Container herunterfahren, Netzwerkfehler herbeiführen und die Container-Ressourcen an die Grenzen bringen (Speicher, CPU, Dateisystem, etc.) Durch den Einsatz von Pumba können zudem alle Services in Containern ins Chaos-Engineering eingebettet werden. Was aber ist mit den Stellen im System, die nur durch die Netzwerkkommunikation eingebunden

werden können, wie beispielsweise Drittanbieter-APIs? Hierfür eignet sich Toxyproxy von Shopify. Durch Toxyproxy können die Latenzen im Netzwerk manipuliert und Totalausfälle simuliert werden *[10]*.

Von der Fehlertoleranz in Microservices-Projekten

Nachdem das Chaos-Engineering einige Wochen eingeführt wurde und sich die Teams daran gewöhnt haben, lässt sich feststellen, dass viele Stellen der Anwendung nicht wirklich robust sind. Die fehlende Resilienz in der Architektur sowie im Code der einzelnen Servicekomponenten und des Gesamtwerks macht sich an vielen Stellen deutlich bemerkbar. Aber die Teams kennen nun diese Ausnahmefälle, die Probleme verursachen; gemeinsam können sie diese Probleme in den Griff bekommen und ihre Auswirkungen abmildern. Offenbar war der größte Irrglaube der Teams, dass die Services korrekt genutzt würden. Clients verwenden die Services gelegentlich mit falschen Eingaben.

Diese Fehlnutzung kann von den Entwicklern nicht vermieden werden. Allerdings können die Entwicklerteams verhindern, dass ihre Services bei einer Fehlnutzung sofort in nicht behebbare Fehlzustände navigieren. Dafür stehen ihnen, neben dem bunten Baukasten der vernünftigen Architektur- und Softwareentwicklung, auch einige Entwurfsmuster zur Verfügung, die sich in Microservices-Umgebungen besonders bewährt haben (beispielsweise Circuit-Breaker oder Retry-Backoff). Durch das Schaffen eines Bewusstseins für die Probleme in der Anwendung und deren Lösung durch geschicktes Softwaredesign kann letztendlich die Resilienz eines jeden Microservice und damit der gesamten Anwendungslandschaft merklich erhöht werden. Das Projekt ist damit bereits deutlich stabiler geworden und Carolin kann sich daran machen, das Leben der Entwickler weiter zu vereinfachen.

Von sprechenden Namen und Code

Dass Microservices besondere Sorgfalt bei der Entwicklung benötigen, haben die Entwicklerteams bereits erkannt. Da diese schon genug Komplexität in ihrem Arbeitsalltag haben, möchte Carolin dafür Sorge tragen, dass so viele andere Bereiche wie möglich simpel gehalten werden und einem einfachen Muster folgen. Dafür führt sie zunächst einheitliche Nomenklaturen für alle APIs im Projekt mit einer entsprechenden Namenskonvention ein. Bei REST sollen dafür nach Möglichkeit die HTTP-Verben verwendet sowie Ressourcen und Operationen einheitlich und sinnvoll benannt werden. Ebenfalls durch die Namenskonvention abgedeckt sind eine saubere Struktur für Übergabeparameter und Konventionen für den Austausch von asynchronen Nachrichten, die das Einlesen und Ausgeben für jedes Team einheitlich gestalten.

Da es sich bei einer Microservices-Anwendung letztendlich auch um ein Stück Software handelt, setzt Carolin zu guter Letzt zusammen mit allen Teams eine Code-Guideline auf, die zwar nicht jedes feine Detail jeder genutzten Technologie abdeckt, aber grobe Richtlinien aufzeigt. Hierzu zählt beispielsweise die Regelung, die Errungenschaften der letzten 30 Jahre in der Softwareentwicklung nicht komplett mit Füßen zu treten und auch Microservices-Code zu testen. Dies wird von den Teams nun mit einem Sonar-Server automatisch im Build-Prozess eines jeden Service überwacht.

Damit hat das Projekt von Carolin, gestartet mit den besten organisatorischen und kulturellen Voraussetzungen, nun auch die technischen Voraussetzungen, um ein wahrer Erfolg zu werden und den Vorsprung des Unternehmens auf Jahre abzusichern. (Anmerkung: Die Abstinenz der aufgeführten Fehler bedeutet natürlich nicht, dass das Projekt funktionieren muss, das Vorhandensein einiger der aufgeführten Probleme kann aber durchaus auf technische Defekte schließen lassen, die häufig recht pragmatisch behoben werden können).

Quellen

[1] J. Thönes (2015): „Microservices" in IEEE Software, vol. 32, no. 1, pp. 116-116, p. 114

[2] M. Santana, A. Sampaio, M. Andrade und N. Rosa (2019): „Transparent Tracing of Microservice-Based Applications" in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, p. 1252-1259, Association for Computing Machinery, New York, NY, USA

[3] J. Carnell (2017): „Spring Microserices in Action", Manning Publications Co. Shelter Island, NY, USA

[4] *http://principlesofchaos.org/?lang=ENcontent*

[5] A. Basiri, A. Blohowiak, L. Hochstein, N. Jones, C.Rosenthal, H. Tucker (2017): ChAP: Chaos Automation Platform, Netflix Technology Blog, *https://netflixtechblog.com/chap-chaos-automation-platform-53e6d528371f*

[6] M. Graff, C. Sanden (2018), Automated Canary Analysis at Netflix with Kayenta, Netflix Technology Blog *https://netflixtechblog.com/automated-canary-analysis-at-netflix-with-kayenta-3260bc7acc69*

[7] L. Hochstein, C. Rosenthal (2016) Netflix Chaos Monkey Upgraded, Netflix Technology Blog *https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d*

[8] *https://codecentric.github.io/chaos-monkey-spring-boot/*

[9] *https://github.com/alexei-led/pumba*

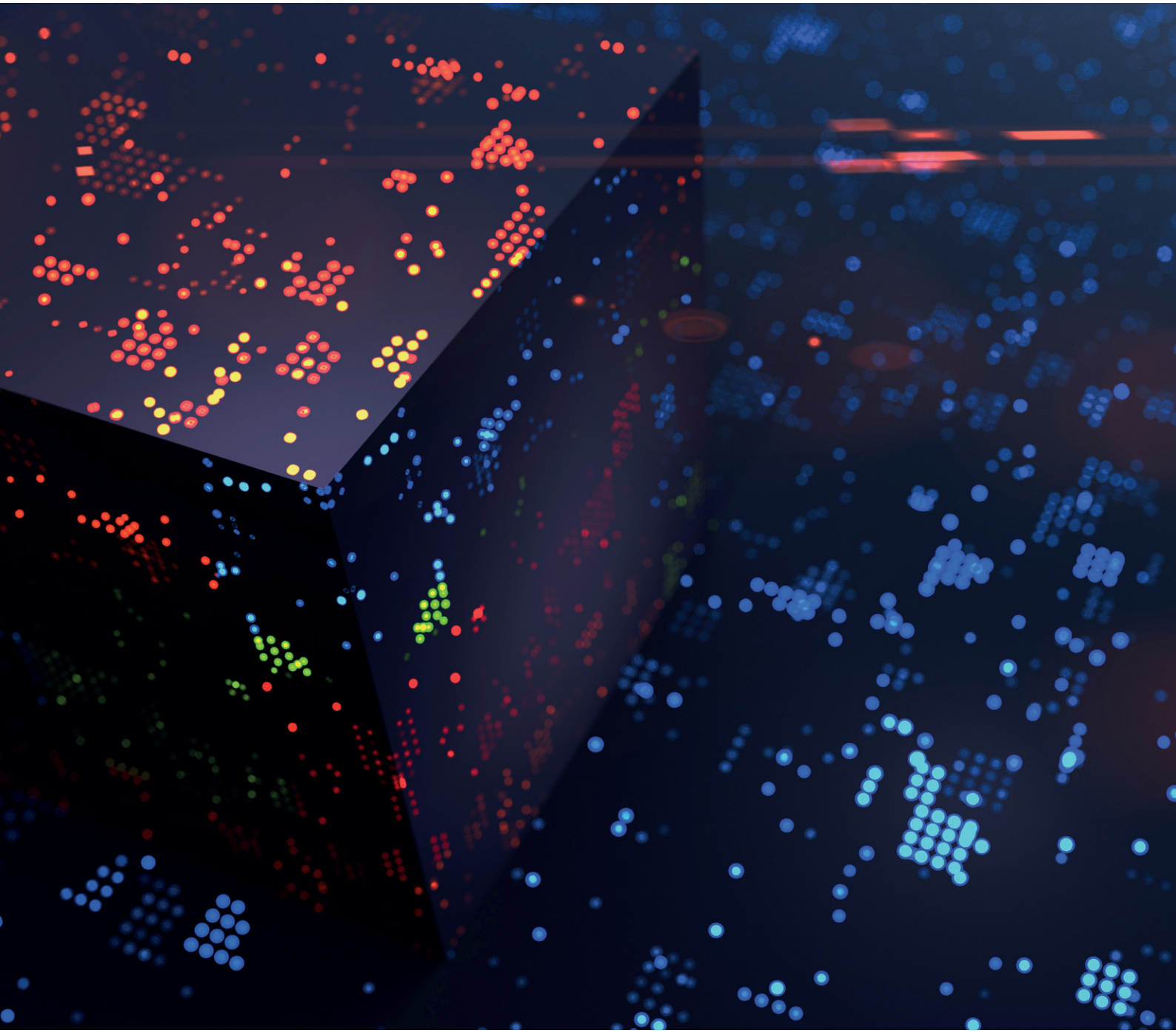
[10] *https://github.com/Shopify/toxiproxy*

Dominik Galler

esentri AG

dominik.galler@esentri.com

Dominik Galler ist als Architekt und Berater bei der esentri AG in der Entwicklung und Konzeption verteilter Anwendungen tätig. Er hat immer Freude daran, neue Techniken auszuprobieren und für den Enterprise-Einsatz zu bewerten. Mit seinem breiten Wissen über Problemlösungsstrategien und Architektur setzt er Anforderungen der Systemlandschaft unter Berücksichtigung der Unternehmenskultur und -struktur um.



Performancetests von Microservices

René Schwietzke, Xceptance GmbH

Eine Softwarearchitektur, die aus kleinen und unabhängigen Teilen zusammengesetzt ist, ist einfacher zu erstellen und zu warten. Microservices sind dadurch zum dominierenden Thema in der modernen Softwareentwicklung geworden. Während die Komplexität auf der Entwicklungsseite sinkt, steigt sie in Bezug auf die Themen Architektur, Performance und Zuverlässigkeit. Dieser Artikel diskutiert die Anforderungen an die Performance und die Planung von Performancetests von Microservices.

Microservices definieren sich laut Jaxcenter *[1]* wie folgt: „Die wesentliche Eigenschaft von Microservices ist das unabhängige Deployment." Laut Red Hat *[2]* hat eine Microservice-Architektur die folgenden Vorteile: schnellere Markteinführung, hochgradig skalierbar, Robustheit, einfache Implementierung, besserer Zugriff, mehr Offenheit." Die Punkte Skalierbarkeit und Robustheit lassen sich allerdings nur über umfangreiche Tests nachweisen. Diese Tests sind gleichzeitig auch Bedingung für eine schnelle und verlässliche Markteinführung. Skalierung und Robustheit sind hochkomplexe Themen, die im Design und in der Implementierung logisch und beherrschbar erscheinen, deren tatsächliches Verhalten jedoch meist von den Erwartungen abweicht.

In den folgenden Kapiteln wird meist „Services" als Synonym für Microservices verwendet, um die textuelle Darstellung zu verkürzen und zu verdeutlichen, dass Tests von größeren Services ähnlich zu Microservices erfolgen.

Testansätze

Services lassen sich auf vier Arten testen: isoliert, direkt, indirekt oder im Rahmen des Gesamtsystems.

Isoliert: Die Kommunikation zu anderen Services wird durch geeignete Mocks oder Simulatoren ersetzt und die Testlast erreicht den Service direkt über sein Interface.

Direkt: Beim direkten Test wird die Kommunikation zu weiteren Services nicht ersetzt (gemockt). Damit lassen sich Probleme mit der Service-Interkommunikation sowie durch Feedback hervorgerufenes Verhalten beobachten und beurteilen.

Indirekt: Wenn ein Service als konsumierter Service getestet wird, dann wird die Last nicht direkt auf den Service angewendet, sondern indirekt durch den Test anderer Anwendungsbereiche.

Gesamtsystem: Der Test im Gesamtzusammenhang ist am realistischsten. Er hilft falsche Annahmen auszuschließen, zeigt die Abhängigkeiten und verdeutlicht den Einfluss einzelner Komponenten auf das Gesamtsystem.

Es gibt kein allgemeines Rezept dafür, welcher Ansatz zum Test eines Service am besten geeignet ist. Diese Fragestellung muss individuell beantwortet werden. Empfohlen sind zu Beginn das isolierte Testvorgehen und ein Gesamtsystemtest auf der zukünftigen Hard- und Softwareplattform.

Trotz gegenteiliger Empfehlungen verwenden Services oft gemeinsame Komponenten. Das sind häufig Datenbanken und Eventbusse sowie generische Dienste, wie Storage, Security, Log-Aggregatoren und Monitoring. Bei der Planung von Tests ist der Einfluss dieser Drittsysteme zu beachten. Häufig lastet zum Beispiel ein Dienst die Datenbank aus und verringert dadurch die Performance anderer Dienste.

Anforderungen

Zuerst werden allgemeine, nicht funktionale Anforderungen wie Skalierbarkeit, Robustheit und das schnelle Deployment betrachtet.

Das Thema Performancetests sollte nicht in seiner oft klassischen Definition gesehen werden – als reine Messung der Performance unter einer bestimmten Arbeitslast –, sondern als ausführliches Vorgehen, um das gesamte Verhalten der Anwendung unter wechselnden Lasten und Zuständen zu beurteilen. „Performance testing… It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage *[3]*." Denn aus dem späteren Einsatzzweck der Services ergeben sich weitergehende Anforderungen, zum Beispiel eine vorgegebene Verfügbarkeit oder garantierte Antwortzeiten. Auch das Datenwachstum zustandsbehafteter Services kann die Performance beeinflussen.

Durchsatz

Typischerweise steht meist der gewünschte Durchsatz im Vordergrund und selten die erwarteten Antwortzeiten. Beide stehen allerdings in einem engen Zusammenhang und sollten auf keinen Fall losgelöst voneinander getestet werden, denn der Durchsatz definiert sich als die Anzahl von Anfragen, die ein Service in einem Zeitraum unter Einhaltung der Kriterien Antwortzeiten und Stabilität beantworten kann.

Man sollte auf keinen Fall den Durchsatz als alleinige Größe nutzen, denn damit sind weder Antwortzeiten noch Fehlerraten ein Teil des Gesamtbildes. Es würde also nur eine quantitative Aussage getroffen, keine qualitative.

Antwortzeiten/Latenz

Die Zielgrößen für die Antwortzeiten sollten mindestens ein Perzentil mit einer Maximalzeit für den Großteil der Antworten sowie die maximal tolerierbare Antwortzeit für einzelne Requests umfassen, zum Beispiel 100 ms im Perzentil P99, aber nie mehr als 500 ms.

Ein P99 von 100 ms bedeutet, dass 99 Prozent aller Anfragen in 100 ms oder weniger beantwortet werden müssen. Nur in einem Prozent der Fälle darf die Antwortzeit darüber liegen *[4]*. Ein P99 definiert kein oberes Limit.

Die Nutzung von Durchschnitten sollte vermieden werden, da diese Spitzen nicht abbilden und oft langsam auf Änderungen reagieren. Durchschnitte können jedoch zum Vergleich zweier Tests zusätzlich zu den PXX-Werten herangezogen werden.

Für Services mit hohen Anforderungen an den Durchsatz sollte das P99.9 genutzt werden, da ein P99 es einem Prozent der Antwortzeiten erlaubt, über dem Zielkriterium zu sein. Das sind bei 100 Requests pro Sekunde bereits 3.600 Anfragen pro Stunde, die nicht innerhalb der Zielerwartung liegen. Auch die Kombination von P95, P99, P99.9 und einer Maximalzeit kann hilfreich sein, um die Erwartungshaltung genau zu definieren und damit auch das spätere Produktionsverhalten mit einer Garantie ausstatten zu können.

Bei der Definition der Ziele sind die spätere Nutzung und natürlich die Aufgabe des Service einzubeziehen. Die Antwortzeiterwartung sollte bereits zu Beginn der Implementierung feststehen, da diese einen entscheidenden Einfluss auf die Umsetzung haben kann.

Bei der Betrachtung der Antwortzeiten kann man entweder von einem komplett identischen Verhalten zu jedem Zeitpunkt ausgehen oder die Erwartungen abstufen, zum Beispiel in Normalzustand

und Ausnahmezustand. Ausnahmezustände können Deployments, Komponentenausfall oder Überlastsituationen sein.

Stabilität

Im Idealzustand sollte ein Service natürlich fehlerfrei laufen. „Fehlerfrei" ist hier durch das erwartete Antwortverhalten definiert. Fehler können rein technischer Natur sein, wie Connection-Fehler oder Timeouts, aber auch funktionaler Natur, verursacht beispielsweise durch inkorrekte Implementierung von Logik, die unter Belastung zu unerwarteten Daten oder Betriebszuständen führt.

Zusätzlich kann die Stabilität wieder für den normalen Betriebszustand und für Ausnahmen definiert werden. Für normale Betriebszustände sollte man eine Null-Fehler-Policy definieren. Das sollte auch für die Skalierung gelten, da dies kein besonderer Betriebszustand ist. Ausnahmesituationen sollten auftretende technische Fehler selten sein, zum Beispiel ein Prozent der Anfragen betreffen, und das Verhalten sollte definiert sein, also sind beispielsweise 502/503 Proxyfehler-Meldungen in Ordnung, Connection Resets oder Timeouts jedoch nicht. Funktionale Fehler sollten in keinem Fall auftreten.

Skalierung

Der Begriff Skalierung ist im Zusammenhang mit Microservices oft leichtsinnig im Gebrauch: „Hochgradig skalierbar – wenn der Bedarf für bestimmte Services steigt, können diese über mehrere Server und Infrastrukturen hinweg flexibel implementiert werden *[2]*."

Ein Microservice oder Service skaliert nicht per se, sondern bringt die Grundlagen mit, im Vergleich zu einer traditionelleren Architektur einfacher skalierbar zu sein.

Systeme lassen sich auf zwei Arten skalieren *[5]*. „Scale Out" – mehr zusätzliche Komponenten der gleichen Art (logische Einheit), also zum Beispiel mehr Serviceinstanzen durch mehr Maschinen – und „Scale Up" – mehr Ressourcen pro Serviceinstanz, zum Beispiel mehr CPUs oder Speicher. In beiden Fällen ist die Erwartung, dass eine Verdoppelung der Ressourcen auch die Leistung verdoppelt, also mehr Durchsatz bei gleichen Antwortzeiten. In den wenigsten Fällen dürfte diese Erwartung eintreten, denn die Aufgaben des Service und sein Zustand müssten dafür linear skalierbar sein.

In fast allen Fällen muss Skalierung erprobt und untersucht werden. Liefern mehr Ressourcen mehr Leistung und wenn ja, bis zu welchem Punkt und mit welchem Faktor? Welche unerwarteten Skalierungshemmnisse treten dabei auf und lassen sich diese beseitigen? Diese Tests können auch zu einem Teil der betriebswirtschaftlichen Abschätzung der Kostenbasis eines Service dienen.

Ausfallsicherheit

Ausfallsicherheit wird in modernen Service-Landschaften häufig durch Chaos Testing *[6]* ausprobiert und herausgefordert. Nicht jede Organisation hat den Mut dazu und es scheint auch nicht ratsam, ohne Basistests mit dieser Methode zu beginnen, da zunächst Gewissheit bestehen sollte, dass der Service prinzipiell den Anforderungen an Ausfallsicherheit genügt und nicht sofort in undefinierte Zustände gerät.

Abseits von der Erprobung genereller Ausfallsicherheit und der Reaktionen darauf innerhalb des Systems ist vor allem das Verhalten der Anwendung bei einem Teilausfall interessant. Zu den Anforderungen an das Verhalten bei einem Ausfall zählen: Wie hoch ist der Verlust der Performance? Wie viele Fehler welcher Art treten auf und wie lange dauert der Übergang zu einem stabilen System?

Deployment

Ein Betriebszustand, der häufig bei Performancetests nicht betrachtet wird, sind die für Services essenziellen kontinuierlichen Deployments. Aus Sicht der Anforderungen stellen sich die folgenden Fragen:

- Sind Fehler während des Deployments erlaubt und wenn ja, welche Fehler und mit welcher Rate?
- Welchen Einfluss auf die Performance des Gesamtsystems darf ein Service-Deployment haben?
- Wie lange darf ein System beeinträchtigt sein, bevor es in den Normalzustand zurückkehrt?

Selbstverständlich muss man beachten, dass Services eigentlich unabhängig sind, deshalb kann es auch dazu kommen, dass mehrere Deployments gleichzeitig laufen. Hier kann es zu kaskadierenden Problemen kommen, die bis zum Ausfall führen können.

Daten

Daten treten als zwei Dimensionen auf. Daten, die als Bewegungsdaten gesehen werden können, die also empfangen, verarbeitet und zurückgesandt werden. Weiterhin bilden Daten auch einen Zustand, wenn der Service diese Daten persistiert.

Kundenservice

Ein Service, der Kundendaten speichert und es erlaubt, diese abzufragen, zu aktualisieren und gegebenenfalls zu löschen, hat zwei Grundanforderungen: die Anzahl der Kundendatensätze im System und die Anzahl der Anfragen. Für die Operationen „Anlegen", „Abfragen", „Aktualisieren" und „Löschen" lassen sich einzelne Anforderungen definieren, die es zu testen gilt. Wichtig ist auch, dass im späteren Betrieb die Operationen gemischt auftreten und damit auch so getestet werden müssen. Es kann hilfreich sein, Messungen zunächst auf einzelnen Dimensionen durchzuführen, um Informationen über die Performance der Basisoperationen zu gewinnen. Bei den Messungen sind auch Einflüsse der Datenlokalität zu beachten, also welche Teilmengen der Kunden werden angefragt, zum Beispiel alle zehn Millionen gleichmäßig verteilt oder mit einer Fokussierung auf 100.000 und in seltenen Fällen die verbleibenden Kunden.

Bilderservice

Ein Service, der Bilder skaliert, ist theoretisch zustandslos. Zu beachten ist, dass Caches jedoch auch einen Zustand darstellen und in der Test- und Anforderungsbetrachtung berücksichtigt werden müssen. Daten für diesen Dienst sind nicht persistent, also sind die reine Anfragegröße der Bilder, die auszuführende Operation sowie die Rückgabegröße wichtig. Dabei kann man zunächst drei Betriebszustände betrachten: die Maximalwerte, den Betriebsdurchschnitt oder die kleinste Operation.

Beispielhafte Anforderung

Dieses Beispiel soll die Überlegungen zu Anforderungen an einen Service oder eine servicebasierte Anwendung illustrieren.

Normalbetrieb

- Durchsatz: 100 Requests pro Sekunde
- Fehlerrate: Keine technischen oder funktionalen Fehler
- Antwortzeiten: P99 150 ms, P99.9 500 ms, maximal 2.000 ms

Der Service soll diese Werte mit der Hälfte der maximal möglichen logischen Einheiten (siehe Kapitel Skalierung) erreichen.

Skalierung

- Maximal: 32 logische Einheiten (16 CPUs pro Einheit, 32 GB Speicher)
- Eine zusätzliche Instanz verbessert die Gesamtleistung um die Leistungsfähigkeit einer einzelnen logischen Einheit
- Fehlerrate: Keine Fehler bei der Skalierung in beide Richtungen
- Dauer: zwei Minuten vom Erkennen des Bedarfs bis Normalbetrieb
- Antwortzeiten während der Skalierung: Antwortzeit P99 500 ms und P99.9 2.000 ms, maximal 4.000 ms

Deployment

- Dauer: maximal zehn Minuten mit der maximalen Anzahl von logischen Einheiten
- Der erreichbare Durchsatz darf sich nicht ändern (100 Requests pro Sekunde)
- Fehlerrate: keine Fehler
- Antwortzeiten: Es gelten die Werte für die Skalierung

Ausfallverhalten

Fällt eine Instanz aus, dann dürfen für eine Minute genau 1/X * Y (X = Anzahl der Instanzen, Y = Gesamtanzahl der Anfragen) fehlschlagen und die Antwortzeiten dürfen für zwei Minuten 25 Prozent höher sein als in der allgemeinen Anforderung beschrieben. Für die nächsten zwei Minuten, um den ausgefallenen Service zu ersetzen, gelten die Anforderungen für die Skalierung. Es dürfen nie mehr als 50 Prozent der maximalen Anzahl logischer Einheiten ausfallen.

Beispiel eines Testvorgehens

Die vorherigen Kapitel diskutieren die Ideen für Anforderungen als Motivation und welche Gesichtspunkte in die Testplanung einfließen sollten. Im Rahmen dieses Artikels kann die Ausführung nicht gründlich diskutiert werden, deshalb sei diese nur als Vorgehensvorschlag erwähnt.

Basisperformance:

Zuerst sollte die Basisperformance bei minimalem Ausbau und ohne Skalierung geprüft werden. Minimaler Ausbau heißt in den meisten Fällen zwei logische Serviceeinheiten, um Redundanz zu gewährleisten. Zunächst wird ohne Parallelität gemessen, um die Baseline (Grundperformance) zu erfassen. Sollten diese bereits über den Zielkriterien liegen, dann kann der Test hier abgebrochen werden.

Basisausbau und Durchsatz:

Im nächsten Schritt sollte man den erreichbaren Durchsatz des Minimalausbaus bei Einhaltung der Zielkriterien ermitteln. Hieraus ergibt sich bereits eine theoretische Hochrechnung, ob die Skalierung es überhaupt ermöglichen kann, Durchsatz und Performancekriterien zu erreichen.

Daten:

Der Einfluss der Datenmenge und -größe sollte vermessen werden, um die Grenzen zu kennen, in denen die Basisperformance gültig ist.

Mindeststabilität:

Immer noch auf der Mindestgröße sollten jetzt erste längere Tests stattfinden, um sicherzustellen, dass Durchsatz und Antwortzeiten stabil sind und keine Fehler auftreten.

Skalierung:

Drei Fragen sollte der Skalierungstest beantworten helfen.

1. Wie skaliert der Service? Was gewinnt man mit mehr Ressourcen?

2. Wann skaliert der Service? Welche Trigger benötigt man und sind diese korrekt?
3. Wie verhält sich der Service, wenn er skaliert, beziehungsweise wie verhält er sich in der kurzen Hochlastphase?

Zieldurchsatz:
Funktioniert die Skalierung, dann kann man prüfen, ob sich der geplante Durchsatz erreichen lässt und ob dabei die geforderten Antwortzeiten erreicht werden.

Stabilität und Robustheit:
Sind die Basisziele erreichbar, dann kann die Langzeitstabilität und das Verhalten bei Störungen geprüft werden. Hier können Methoden des Chaos Testing zum Einsatz kommen.

Überlastverhalten:
Jedes System hat Grenzen, deshalb ist es wichtig zu verstehen, wie sich das System an diesen Grenzen und bei Überschreitung verhält. In den meisten Fällen reicht es aus, wenn das Verhalten vorhersagbar und definiert ist sowie das System von selbst wieder in den regulären Betriebszustand zurückgekehrt.

Deployment:
Essenziell für Microservices sind schnelle und kontinuierliche Deployments. Mit dem Test dieser Deployments unter Last sind das daraus resultierende Verhalten und Auswirkungen auf das Gesamtsystem zu evaluieren.

Performancetests sollten die vorhandenen Logging- und Monitoringtools nutzen, um deren Einsetzbarkeit zu prüfen, und natürlich, um die richtigen Schlussfolgerungen für Verbesserungen zu ziehen. Performancetests ermöglichen auch hier einen ersten Probelauf für den späteren Produktivbetrieb.

Zusammenfassung

Während ein Service selbst oft einfach zu bauen und zu beherrschen ist, ist das Zusammenwirken vieler Services eine komplexe Fragestellung und Herausforderung. Ohne grundlegende Tests der Grundannahmen wie Skalierbarkeit, Zuverlässigkeit und erreichbare Performance können kritische Systeme nicht produktiv genommen werden.

Nachtrag

Microservices und moderne Softwareentwicklung postulieren in vielen Fällen, dass auf Produktion getestet werden soll und gerade die Fähigkeit zu fortlaufenden Deployments hier eine schnelle Fehlerbehebung verspricht. Das schließt einen klassischen Softwareperformancetest vor der Inbetriebnahme oder nach größeren Änderungen allerdings nicht aus. Speziell dann, wenn:

- die Organisation unerfahren mit Services ist,
- neue Technologien zur Anwendung kommen,
- kritische Prozesse ersetzt oder erneuert werden,
- hohe Anforderungen an Durchsatz, Antwortzeiten und Stabilität existieren.

Ein Performancetest kann nicht alle Anwendungsfälle abdecken, sollte aber in der Lage sein, ein Grundvertrauen in den Service oder die Servicearchitektur aufzubauen und damit geschäftliche Risiken zu minimieren.

Dieser Artikel erwähnt einige Testideen, wie das Testen des Einflusses von Daten auf die Performance, nur am Rande, um den Rahmen des Artikels nicht zu sprengen, beziehungsweise lässt Themen wie Sicherheit, Quotas und Multi Tenancy außenvor.

Quellen

[1] „Was sind eigentlich Microservices?" Jaxenter *https://jaxenter.de/was-sind-microservices-40571*

[2] „Was sind Microservices?" Red Hat *https://www.redhat.com/de/topics/microservices/what-are-microservices*

[3] „Software Performance Testing", *https://en.wikipedia.org/wiki/Software_performance_testing*

[4] „Percentile", *https://en.wikipedia.org/wiki/Percentile*

[5] „Was ist Skalierbarkeit?", Autor/Redakteur: Otto Geißler/Ulrike Ostler, 15. August 2019 *https://www.datacenter-insider.de/was-ist-skalierbarkeit-a-852037/*

[6] „What is Chaos Testing?", Ben E. C. Boyter *https://boyter.org/2016/07/chaos-testing-engineering/*

René Schwietzke
Xceptance GmbH
r.schwietzke@xceptance.com

René Schwietzke ist Mitgründer und Geschäftsführer der Xceptance GmbH. Er arbeitet seit ca. 20 Jahren im Bereich Last- und Performancetest und hat viele internationale Kunden und Softwareanbieter unterstützt. Er ist Product Owner für das Xceptance Lasttest-Tool XLT, dass seit 15 Jahren sowohl bei Xceptance als auch bei Kunden erfolgreich im Einsatz ist. Seit 2020 ist XLT Open Source unter der Apache-Lizenz 2.0. Des Weiteren beschäftigt sich René mit Qualitäts- und Performancethemen rund um Programmierung und die Java Virtual Machine.

DDD mit MicroProfile GraphQL

Rüdiger zu Dohna, codecentric AG

Überraschenderweise kann man auch mit Java leicht einen GraphQL-Service schreiben, der sich nach echtem Java anfühlt (also mit typsicheren Plain Old Java Objects arbeitet) und trotzdem die Anforderungen an die Dynamik von GraphQL erfüllt. Mit MicroProfile GraphQL haben wir jetzt einen Standard an der Hand, mit dem man das bequem und zukunftssicher machen kann. Aber wozu sollten wir überhaupt etwas anderes als REST machen? Beispielsweise bei einem Public API sind die Clients separate Bounded Contexts, das heißt, zur fachlichen Entkopplung muss ein REST-Service entweder alle Daten auf einmal oder mit vielen verschachtelten Requests liefern; beides kann, gerade in mobilen Anwendungen, zu langsam sein. Und genau da kann GraphQL helfen!

In den 90er Jahren haben wir in jedem neuen Projekt immer als Allererstes ein ER-Diagramm gemalt: Welche Entitys mit welchen Attributen brauchen wir in der Datenbank und welche Beziehungen (Relationships) gibt es dazwischen? Das war eine Best Practice, das hat man eben so gemacht. Nachdem wir dieses Schema dann in der Datenbank umgesetzt haben, bastelten wir die entsprechenden Eingabemasken dazu. Und die waren dann natürlich Mist: Beispielsweise werden komplexere Datenobjekte oft in mehreren Schritten erfasst, vielleicht sogar von verschiedenen Personen. Wenn es allerdings nur eine einzige Eingabemaske für ein komplexes Datenobjekt mit mehreren Tabs oder langem Scrollbalken gibt, dann muss man sich merken, wo die verschiedenen Felder genau sind – ein flüssiges Arbeiten ist so kaum möglich. Für eine gute User Experience (UX) braucht man für jeden Arbeitsschritt unterschiedliche Eingabemasken mit den jeweils zu erfassenden Feldern. So etwas haben wir jedoch nur auf spezielle Anforderung hin beachtet – es waren Ausnahmefälle, die Extrakosten nach sich gezogen haben. Am Computer arbeiten zu müssen war nun mal Arbeit und kein Vergnügen.

Um die Jahrtausendwende herum kamen dann solche Dinge wie Extreme Programming auf. Da haben Leute wie Kent Beck viele Sachen einfach andersherum gemacht. Beispielsweise haben sie erst Tests geschrieben und dann die Implementierung. Das nannten sie TDD – Test-driven Development. Crazy! Sie haben auch als Allererstes Prototypen von der UI auf Papier gemalt und mit echten Endkunden besprochen, bevor sie überhaupt über Datenbanken nachgedacht haben. Das hat aber komischerweise zu sehr zufriedenen Benutzern geführt, was sich wiederum bei Webanwendungen schnell als lohnenswerter Ansatz herausgestellt hat: Kunden kaufen tatsächlich mehr, wenn ein Shop bequem zu bedienen ist! Diese Denk- und Vorgehensweise hat sich also nicht deshalb durchgesetzt, weil es mehr Spaß macht, sondern weil es sich rentiert. Auch bei Handy-Apps haben wir das dann so gemacht, weil es sich bezahlt gemacht hat, Kunden glücklich zu machen. Langsam passiert vieles davon auch in Business-Anwendungen, denn auch hier ist eine gute UX ein Erfolgsfaktor. Da moderne Benutzer auch in ihrer Freizeit IT-Erfahrung sammeln und von den Handy-Apps und Webanwendungen regelrecht verwöhnt werden, leiden sie umso mehr unter schlechter UX von Business-Anwendungen. Vor allem kryptische Fehlermeldungen oder magische Workarounds schmerzen und verlangsamen die Einarbeitung und die tägliche Bedienung. Daher wird auch bei Kaufentscheidungen von Business-Software mehr und mehr darauf geachtet, dass sie eine brauchbare UX hat – zu Recht, denn es geht dabei nicht um „hübsch", sondern um effiziente Bedienbarkeit.

Nachdem eine gewisse Routine in diese neue Arbeitsweise eingekehrt ist, bemerken wir jedoch auch ernstzunehmende Nachteile: Unsere Datenmodelle entwickeln sich nicht von selbst in eine saubere Struktur. Daten werden an verschiedenen Stellen benötigt und müssen irgendwie über Systemgrenzen hinweg synchronisiert werden. Dann müssen vielleicht die Kapselung aufgebrochen oder Konsistenzbedingungen fachlich evaluiert werden etc. Was in relativ einfachen Web-Shops gut funktioniert, führt in Enterprise-Anwendungen zu einer Komplexitätsexplosion – und Komplexität ist schließlich die größte Herausforderung bei der Entwicklung und dem Betrieb von Softwarelandschaften, die von vielen, vielen Teams gemeinsam und trotzdem autonom entwickelt werden.

Wenn man mal die üblicherweise dafür eingesetzten Technologien beiseitelässt und sich ganz auf den fachlichen Mehrwert fokussiert, dann sind SOA und Microservices im Wesentlichen nur neuere Formen der Modularisierung, mit der man Komplexität schon lange effektiv bekämpft. Die neuen Technologie-Stacks bringen allerdings auch ganz neue technische Probleme mit, die die Komplexität wiederum erhöhen. Wenn das also für die Entwicklungsgeschwindigkeit kein Nullsummenspiel sein soll (geschweige denn die Komplexität sogar noch weiter erhöhen soll), dann muss man sich nochmal ganz genau auf die Fachlichkeit konzentrieren: die saubere, fachliche Aufteilung und Modellierung der Domänen, also der Daten und der Operationen, die darauf angewendet werden. Ein sehr solider Ansatz dazu, der in den letzten Jahren eine wahre Renaissance erlebt, ist das Domain-driven Design.

Domain-driven Design ultra-knapp

Sich auf die Modellierung der Domäne zu fokussieren bedeutet konkret, dass wir Entwickler mit den Domänenexperten viel mehr reden müssen. Dadurch lernen wir ihre Sprache und fangen bewusst an, diese auch selbst zu verwenden, sogar in unserem Code: Wir nennen unsere Klassen, Methoden etc. genauso, wie es die Fachleute tun würden. Dazu müssen wir aber auch mit den Fachleuten zusammen daran arbeiten, ihre Sprache so zu präzisieren und zu standardisieren, dass sie auch in Software noch funktioniert. Wir müssen vor allem alle Mehrdeutigkeiten schärfen, die Fachleute normalerweise weniger stören als uns Software-Entwickler; die Schärfung hilft aber auch den Fachleuten, Missverständnisse zu vermeiden. Alle arbeiten und denken dann in einer gemeinsamen Sprache: in der bei DDD sogenannten *Ubiquitous Language* – und die muss gar nicht unbedingt englisch sein. Wenn beispielsweise auf absehbare Zeit alle Fachleute und Techies sowieso deutsch sprechen, darf Software durchaus auch auf Deutsch sein.

Bei der Arbeit an der Ubiquitous Language werden wir feststellen, dass es manchmal unter verschiedenen Gruppen von Fachleuten sehr unterschiedliche Sichtweisen gibt. Beispielsweise redet man sowohl im Billing als auch bei der Lieferung immer einfach von einer Bestellung, jedoch aus sehr unterschiedlichen Perspektiven: Nicht nur Liefer- und Rechnungsadresse sind zwei getrennte Dinge, sehr viele Felder (zum Beispiel Artikelgewicht oder Preis) und teilweise sogar ganze Objekte (zum Beispiel Rabatt-Positionen) sind nur für die einen oder nur für die anderen relevant. Mit steigender Komplexität wird es immer anstrengender, sich auf eine Ubiquitous Language zu einigen, weil sich jeder eigentlich nur für einen Teil davon interessiert und die anderen Dinge als irrelevant und störend empfindet. Man kann allerdings an genau solchen Stellen gut eine fachliche Trennung einführen, das bedeutet, man schneidet die Systeme so, dass man sich immer auf jeweils einen dieser Teilaspekte konzentrieren kann. Zwischen allen geteilt werden nur Kernattribute wie die Bestellnummer und das Bestelldatum. Einen so abgetrennten Teil nennt man bei DDD einen *Bounded Context*. Diese Trennung sollte man dann auch bis in die Teamstruktur durchziehen, also beispielsweise jeweils ein Team für Billing und Lieferung aufstellen (siehe auch Conway's Law [1]).

DDD beschreibt noch viele weitere, interessante Konzepte; diese beiden sind aber wohl der Kern.

Dabei darf man den hohen Stellenwert der UX nicht vernachlässigen. Man muss eine Balance finden zwischen dem Inside-Out-Ansatz, bei dem man die Domäne zuerst betrachtet und die UX daraus ableitet, und dem Outside-In-Ansatz, bei dem man das andersherum macht. Am besten versucht man nicht beides auf einmal, sondern wechselt regelmäßig zwischen beiden Ansätzen hin und her. In beiden Denkrichtungen gibt es auch weiterhin interessante Neuerungen. Beispielsweise sind Consumer-driven Contracts *[2]* eine Fortführung des TDD: Man geht dabei von den Anforderungen der Clients outside-in zu den Services. Andererseits ist Event Storming *[3]* ein neues Workshop-Format, bei dem man in einer Gruppe aus Fachleuten und Techies gemeinsam inside-out arbeitet, um Erkenntnisse zum Domänenmodell herauszuarbeiten.

Torten

Wenn man Bounded Contexts zur fachlich-organisatorischen Trennung von Systemen und Teams verwendet, dann nennt man das auch vertikale Schnitte; während horizontale Schnitte eine Trennung anhand von Technologien bedeutet, also beispielsweise zwischen Frontend und Backend. Bei vertikal geschnittenen Teams liegt die Verantwortung, etwa für den Warenkorb, in einem einzigen Team. Es hat die volle Hoheit über die Entwicklung und den Betrieb des kompletten Stacks: von der fachlichen Ausgestaltung über die UX bis hin zu den Services und Datenbanken. Wenn alles aus einer Hand kommt, dann gibt es keine Interessenskonflikte mehr. Ein Team gewinnt oder verliert immer als Ganzes. Es ist egal, ob ein Fehler in der fachlichen Anforderung, im Datenbank-Design oder in einem Service gemacht wurde, er muss nun mal behoben statt reassigned werden. Ich habe Bugs erlebt, die über 50 Mal an andere Teams (auch in Zyklen) assigned wurden, bevor sie endlich jemand behoben hat. Die Analogie mit einer Torte veranschaulicht das sehr schön: Jedes Stück einer Torte hat die optimale Mischung aus allen Schichten.

Das zieht sich bis ins Frontend durch – auch der entsprechende Teil des Zuckergusses gehört zu jedem Stück Torte. Die Backends sauber in fachliche Teilsysteme zu zergliedern, dann aber ein monolithisches Frontend draufzusetzen, macht den ganzen Ansatz zunichte. Das Frontend soll ja die Entwicklung des Backend treiben, muss sich also genauestens mit der Fachlichkeit von der UX bis hin zum Domänenmodell auskennen. Über die ganze Bandbreite aller Bounded Contexts hinweg ist das sehr schnell nicht mehr zu leisten und die Qualität leidet. Stattdessen brauchen die Microservices im Backend auch dazugehörige Micro-Frontends *[4]*. Bei Webanwendungen lässt sich das relativ leicht lösen, wenn man jeden Bounded Context auf seinen eigenen Seiten anzeigen kann, die dann jeweils von verschiedenen Teams autonom entwickelt werden können. Wenn ein Team für Lieferadressen und Versandoptionen verantwortlich ist und ein anderes Team für Rechnungsadressen und Bezahloptionen, dann kann man diese auf separaten Seiten anbieten, die von verschiedenen Systemen geliefert werden, die wiederum unabhängig voneinander entwickelt und deployt werden können. Diese fachliche Aufteilung ist auch für Kunden nachvollziehbar und hilfreich.

Native Handy-Apps?

Manchmal werden auch mobile Apps in mehrere, separat entwickelte Apps aufgeteilt, zwischen denen man über Deep Links trotzdem hin- und hernavigieren kann. So können die Teams ihre jeweiligen Apps selbstständig releasen. Es irritiert jedoch bei der Verwendung, wenn das zu häufig passiert und der Mehrwert nicht ersichtlich ist; es ergibt also hauptsächlich dann Sinn, wenn manche Kunden tatsächlich nur einzelne Apps benötigen.

Wenn das also nicht infrage kommt, muss man notgedrungen mit einem App-Monolithen leben. Den baut man zwar selbstverständlich aus getrennten Modulen für die verschiedenen Teams auf, aber spätestens die Release-Prozesse müssen teamübergreifend koordiniert werden. Das ist anstrengend, auch wenn man Apps nicht so häufig releasen will wie Webanwendungen, die man ja idealerweise bei jedem Commit ausrollt *[5]*.

Alternativ kann man Handy-Apps eher wie einen Spezial-Browser entwickeln. Das heißt, dass die Anwendungslogik komplett im Backend liegt und die App sie nur anzeigt. Neben der Vertrautheit der Teams mit Webtechnologien aus ihren Webanwendungen ist das ein wesentlicher Grund, warum Webframeworks auch für die Handy-Entwicklung so populär sind, obwohl native Apps meist doch eine bessere UX bieten.

Schwieriger sind gemischte Seiten. Beispielsweise wird der Warenkorb ja nicht nur auf einer eigenen Bestellübersicht angezeigt; auch auf den Katalogseiten und anderswo soll meistens ein kleines Widget angezeigt werden, in dem die Anzahl der ausgewählten Artikel und die Gesamtsumme angezeigt werden. Moderne Frontendtechnologien wie WebComponents *[6]* erlauben jedoch auch innerhalb einer einzigen Seite, Komponenten dynamisch aus mehreren Backends anzuzeigen. Das erhöht zwar etwas die technische Komplexität, reduziert aber gleichzeitig die fachliche Komplexität stark. Über solche Ansätze kann man auch andere als Domänen-Aspekte der UX lösen: So sollen etwa der Header mit Navigationsmenü und der Footer mit den Links auf das Impressum über alle Seiten hinweg immer einheitlich sein, ohne alle Services gleichzeitig koordiniert ausrollen zu müssen, nur weil sich da etwas geändert hat und die neuen Komponenten in allen Services neu eingebunden werden müssen.

Autonome Mischung

Sowohl bei der Aufteilung auf verschiedene Seiten, erst recht aber bei der Mischung von Komponenten innerhalb einer Seite muss man

aufpassen, dass diese auch optisch zusammenpassen, auch wenn Amazon auf seinen Webseiten zeigt, dass das nicht unbedingt perfekt sein muss – gerade bei AWS sind die Oberflächen der verschiedenen Dienste komplett unterschiedlich. Normalerweise will man allerdings, dass die Gesamtanwendung nach außen wie eine Einheit wirkt. Das geht eigentlich nur über eine Koordinierung über Teamgrenzen hinweg. Praktisch jede Firma hat ihre CI (Corporate Identity) festgelegt, also welche Fonts, Farben etc. überall in der Außendarstellung verwendet werden sollen. Das passiert in der Regel zentral; bei der Entwicklung in autonomen Teams tauchen hingegen immer wieder Detailfragen auf, beispielsweise wie im Date-Picker die Kalenderwoche angezeigt werden kann. Diese Fragen zentral in einem Gremium zu beantworten ist einfach viel zu langsam. Wenn die Experten in den Teams ihr Handwerk beherrschen, dann kann man ihnen auch die Hoheit überlassen, solche Fragen erst mal selbst zu beantworten und erst später in die anderen Teams zu tragen, etwa über eine gemeinsam entwickelte Dokumentation oder – noch besser – Komponentenbibliothek. Das kann in einem Gremium vorgestellt werden, aber auch sofort, ohne formelle Freigabe eingesetzt werden. Vorab eine kurze Absprache mit einer Kollegin oder einem Kollegen aus einem anderen Team ist ja trotzdem möglich. Wichtig ist jedoch, dass schon bei der Sprintplanung beachtet wird, dass diese Koordination zusätzlichen Kommunikationsaufwand bedeutet, der mit eingeplant werden muss – unter Zeitdruck wird das zu leicht vernachlässigt.

Webanwendungen haben manchmal auch Anforderungen, die von denen für Handy-Apps abweichen. Da auf einem Handy weniger Daten auf einmal angezeigt werden können, braucht man andere Navigationsstrukturen als im Web. So könnte man die Gewichtsangabe zu einer Bestellposition im Web gleich in der Übersichtstabelle anzeigen, während man sie auf dem Handy erst anzeigt, wenn man in die Detailansicht springt. Es kann dann sinnvoll sein, diese Anforderungen mit separaten Backendschnittstellen zu erfüllen, die für den jeweiligen Client spezifische Datenstrukturen liefert. Diese nennt man auch „Backend for Frontend" (BFF), was ein weiterer Outside-in-Ansatz ist.

Wie weit kann man vertikal schneiden?

Ein Team, das native Handy-Apps für iOS und Android sowie ein Web-Frontend, die jeweils dazugehörigen BFFs, Domänenservices und Datenbanken entwirft, testet, entwickelt und betreibt, müsste sehr groß sein – schon allein, um diese technologische Bandbreite abdecken zu können, selbst wenn die fachlichen Anforderungen relativ schmal sind und zum Beispiel nur den Warenkorb betreffen. Mit weiteren Team-Rollen und einer gewissen Ausfallsicherheit *[7]* kommt man schnell auf Teamgrößen von zwölf bis über zwanzig, während ein ideales Team eher sechs Mitglieder hat *[8]*. Die Domäne noch kleiner zu schneiden ergibt irgendwann keinen Sinn mehr.

Manchmal braucht man also doch auch horizontale Schnitte – auch dann, wenn man ein Public API anbieten will, also ein API, das von anderen Teams oder Firmen mit ihren eigenen Systemen angesprochen werden soll, damit sie unsere Systeme in ihre eigenen Arbeitsabläufe nahtlos integrieren können. Dadurch wählt man bewusst einen horizontalen Schnitt: Sie entwickeln eigene Anwendungen in ihrem eigenen Bounded Context, von denen die Backend-Systeme gar nichts wissen sollen. Dieses „gar nichts wissen" ist dabei sehr wichtig, damit das gut funktioniert. Es ist also ein bisschen gemogelt, wenn man mehrere Schnittstellen anbieten will, die möglichst generische Use Cases abbilden sollen, denn dann weiß man ja doch ein bisschen von ebendiesen.

Dahinter stecken jedoch unter Umständen Annahmen, die falsch sein können – oder falsch werden können, sobald ein neuer Client ganz andere Einsatzgebiete findet. Wenn man daher nur wenige oder eine einzige, einheitliche Schnittstelle anbieten will, dann gibt es zwei grundsätzliche Vorgehensweisen: Entweder man liefert immer alle Daten, die im Backend zu einem Einstiegspunkt vorhanden sind (und die der Client sehen darf – später dazu mehr), also beispielsweise zu einer Bestellung immer alle Adressen und alle Bestellpositionen, mit den Preisen, Rabatten und Artikeln mit allen Artikeleigenschaften. Daraus werden leicht sehr große und entsprechend unübersichtliche Dokumente; es kann aber auch zu erheblicher Last in den Backends führen, wenn manche Daten mit separaten Selects aus der Datenbank oder sogar per Remote-Call von weiteren Backend-Systemen geholt werden müssen. Durch geschicktes Caching kann man solche Probleme oft einigermaßen eindämmen, das erhöht allerdings wiederum die Komplexität noch weiter, weil diese Caches ja auch invalidiert werden müssen oder man mit der eventuellen Inkonsistenz fachlich umgehen können muss. Für einen Client in einem mobilen Netzwerk kann auch die reine Datenmenge ein Problem sein; und wenn ein Client diese ganzen Daten gar nicht braucht, dann ist dieses sogenannte Overfetching sowohl im Client als auch im Service nicht nur unnötig, sondern schädlich.

Alternativ kann man auch nur Referenzen auf weiterführende Daten liefern. Diese Referenzen können einfache IDs sein, die der Client verstehen und in neue Anfragen zusammenbauen muss; bei wirklichen RESTful Services sind das jedoch direkt URLs, die der Client anfragen kann, um sich die dahinter liegenden Daten zu holen. In beiden Fällen muss man unter Umständen sehr viele Requests machen, etwa wenn man eine Liste von Bestellpositionen und deren Artikel braucht. Oft kann man diese Requests nicht parallel ausführen, weil die Antwort des ersten für den zweiten gebraucht wird. Auch hier kann man durch geschicktes Caching die Performance einigermaßen in Schach halten, erhöht dadurch aber auch wieder die Komplexität. Dieses sogenannte Underfetching führt also ebenfalls zu Performance-Problemen, wobei es in diesem Fall für Clients in einem mobilen Netzwerk nicht mehr um die Datenmenge, sondern um die Anzahl von Requests geht.

GraphQL

In genau dieser Situation war Facebook schon 2012: Vor allem in mobilen Netzen wurden ihre Anwendungen immer langsamer und die Entwicklung immer komplexer. Auf der Suche nach einer grundsätzlichen Lösung fingen sie an, intern eine flexiblere Alternative zu REST zu entwickeln, die sie GraphQL nannten – eine generische Query Language wie SQL, aber für komplexe Datengraphen ausgelegt. Im Jahr 2015 veröffentlichten sie es unter einer Open-Source-Lizenz und übergaben es später an die Linux Foundation, um den Befürchtungen entgegenzutreten, dass Facebook zu viel Kontrolle über die Technologie haben könnte. Die Linux Foundation entwickelt es seitdem aktiv und offen weiter.

GraphQL wurde zuerst hauptsächlich in Single-Page-Webanwendungen eingesetzt – meist mit Frontends in React *[9]* und Backends in Node *[10]*, also in einer reinen JavaScript-Welt. Bald entstanden

```
@GraphQLApi
public class SuperHeroesBoundary {
    @Query public List<SuperHero> heroes() {
        return ...;
    }

    @Query public SuperHero hero(@NonNull String name) {
        return ...;
    }

    @Mutation public SuperHero createHero(SuperHero hero) {
        return ...;
    }
}
```

Listing 1: Beispielanwendung mit MicroProfile GraphQL

jedoch auch immer mehr Frameworks für andere Programmiersprachen und Laufzeitumgebungen. Eine der jüngsten ist MicroProfile GraphQL, das im Februar 2020 in Version 1.0 released wurde.

MicroProfile

MicroProfile wurde als Initiative gestartet, um Spezifikationen zu schaffen, mit denen Java-Anwendungen optimal auch in Cloud-Umgebungen laufen können. Dazu wurden aus Java EE die notwendigsten Standards ausgewählt (CDI, JAX-RS, JSON-P und bald JSON-B) und dafür um die besonders für den Cloud-Einsatz dringend fehlenden Features ergänzt: Config, Fault Tolerance, Metrics und Health.

Das war eine Reaktion auf die Verlangsamung der zwingend notwendigen Weiterentwicklung der Java-EE-Standards vor allem durch den schleichenden Rückzug von Oracle in den Jahren zuvor. Mittlerweile sind MicroProfile und Java EE (das jetzt Jakarta EE heißt) Schwesterprojekte in der Eclipse Foundation und alle wichtigen Jakarta-EE-Implementierungen unterstützen direkt auch die Hauptstandards von MicroProfile. GraphQL zählt noch nicht dazu (später dazu mehr).

MicroProfile GraphQL bietet bisher offiziell nur ein Server-API an (zum Client-API später mehr). Ich stelle hier nur ein paar Beispiele für die Basics vor – eine vollständigere Einführung ist im Internet zu finden *[11]*. Die Einstiegspunkte in einen Service annotiert man wie in *Listing 1* zu sehen: lesende Operationen als `@Query` (das Äquivalent zu JAX-RS `@GET`) und schreibende Operationen als `@Mutation` (`@POST`/`@PUT`/`@DELETE`).

Das sieht sehr ähnlich wie JAX-RS aus, nur dass es vor allem keine `@Path`-Annotationen gibt. Das liegt daran, dass GraphQL mit einem einzigen HTTP-Endpunkt arbeitet und es somit also keine Unterpfade gibt. Stattdessen werden der Methodenname und die Parameter exponiert, das heißt, der Client fragt genau diesen ab (mehr dazu im Folgenden). Meistens verwendet man bei GraphQL auch immer `POST`-Requests, egal ob lesend oder schreibend – `GET` ist auch möglich, aber eher optional. Tatsächlich kann man in einfachen Fällen die gleiche Schnittstelle auch als REST-Schnittstelle anbieten, indem man nur die JAX-RS-Annotationen hinzufügt.

Das Ganze kann man nicht nur aufrufen, MicroProfile GraphQL baut daraus auch ein Schema, das unter anderem als Dokumentation

```
type SuperHero {
  name: String
  realName: String
}

input SuperHeroInput {
  name: String
  realName: String
}

type Query {
  heroes: [SuperHero]
  hero(name: String!): SuperHero
}

type Mutation {
  createHero(newHero: SuperHeroInput): SuperHero
}
```

Listing 2: Beispiel GraphQL-Schema

dienen kann – GraphQL hat also sozusagen ein eingebautes OpenAPI *[12]* (das vorher als Swagger bekannt geworden ist *[13]*). Das Schema zu dem oben definierten API sieht aus wie in *Listing 2*.

Der `type SuperHero` bedarf eigentlich keiner weiteren Erklärung. Der `input SuperHeroInput` ist eine Variation davon, die MicroProfile GraphQL bei Bedarf automatisch für die `@Mutation`-Methode `createHero` generiert, in der es aber keine Resolver-Felder gibt. Der `name`-Parameter von `hero` ist vom Typ `String!` – das Ausrufezeichen bedeutet, dass er angegeben werden muss. Das Ergebnis von `heroes` ist `[SuperHero]` – die eckigen Klammern zeigen an, dass es sich dabei um eine Liste handelt.

Das ist alles ziemlich eingängig; überraschend ist allerdings, dass auch die Einstiegspunkte als Typen auftauchen: `Query` und `Mutation`, in denen die Methoden als Felder definiert sind. Dass Felder und Methoden das Gleiche sind, mutet auf den ersten Blick etwas sonderbar an, eröffnet jedoch interessante Möglichkeiten. So kann jedes beliebige Feld ebenfalls Parameter haben, zum Beispiel könnte der `SuperHero` ein Feld `superPowers(legal: Boolean): [SuperPower]` haben, mit dem man die Superkräfte filtern kann, je nachdem ob sie legal oder illegal sind oder egal, wenn man nichts angibt.

Das sind alles interessante Unterschiede, aber der wirklich wesentliche Unterschied zu JAX-RS ist, dass man bei GraphQL zusätzlich zu Attributen auch sogenannte Resolver definieren kann *(siehe Listing 3)*.

Dadurch erweitert man den Typ `SuperHero` um ein Feld `realName`, das in der POJO-Klasse (Plain Old Java Object) gar nicht enthalten ist. Das ist vor allem dann wichtig, wenn es teuer ist, den Wert zu ermitteln, etwa durch einen Serviceaufruf. Ein Resolver wird allerdings nur aufgerufen, wenn der Client das Feld auch anfragt. Für den Client bleibt der Unterschied zu einem normalen Feld hingegen verborgen. In einem JAX-RS-API würde man dazu wohl einen Query-Parameter einführen, was aber dieses Implementierungsdetail im Client sichtbar macht. Auch im Service erzeugt es zusätzliche Komplexität, weil es auch bei einem zweiten Endpunkt eingebaut werden müsste, der die Super-Teams mit ihren Mitgliedern zurückliefert.

Da MicroProfile GraphQL, wie bereits erwähnt, noch nicht Teil der Haupt-APIs von MicroProfile ist, muss man eine Dependency auf eine Implementierung, zum Beispiel `io.smallrye:smallrye-graphql-servlet` hinzufügen, die wiederum die Dependency auf das MicroProfile-GraphQL-API anzieht. Mehr muss man jedoch auch gar nicht tun – der ganze Rest geht automatisch. Man kann den Service und das Schema einfach mit Tools with GraphiQL (man beachte das „i") *[14]* inspizieren und direkt ausprobieren.

Security

GraphQL bietet auch beim Thema Security Vorteile. Wenn manche Clients die Lagerbestände und Einkaufskonditionen von Artikeln nicht sehen sollen, dann wird das bei RESTful Services meist über die URLs geregelt, manchmal sogar in einer separaten Infrastruktur wie einem API-Gateway oder Service Mesh. Wenn andererseits ein BFF tief verschachtelte Dokumente liefert, kann es leicht passieren, dass mit einer neuen URL auch Daten mitkommen, die in einer anderen URL explizit für einen Client ausgeschlossen wurden, da diese Prüfung nicht in der Domänenlogik erfolgt. Der Abstand der Berechtigungsprüfungen von der Domäne ist schädlich, denn das ist kein technischer, sondern ein fachlicher Aspekt. Bei GraphQL kann man das direkt im Resolver prüfen, also auf der Domänenebene. Das wirkt dann automatisch für alle Einstiegspunkte.

Andererseits gibt man mit GraphQL den Clients gegebenenfalls große Macht an die Hand. Die Anfragebäume können sehr tief sein und hohe Last auf den Servern erzeugen. Um sich davor zu schützen, gibt es einige Konzepte *[15]*, die zwar noch nicht Teil von MicroProfile GraphQL sind, die ich hier aber ganz kurz vorstelle, um die Entwicklungsrichtung zu zeigen. Das einfachste ist es, ein Timeout zu setzen; das ist für die Clients jedoch unangenehm, weil es abhängig von der Last auf dem Server sporadisch passieren kann. Außerdem wird die Anfrage trotzdem durchgeführt, erzeugt also tatsächliche Last, nur dass der Client nichts mehr davon hat. Man kann aber auch schon die Anfragen vor der Ausführung analysieren und die maximale Tiefe begrenzen. Noch ausgefeilter ist es, jedem Feld (inklusive Resolver) eine Komplexität zuzuordnen und dann die Anfragen über die Summe der Komplexitätswerte zu begrenzen. Alle diese Limitierungen kann man pro einzelnem Request machen oder einem Client eine bestimmte Quote pro Zeiteinheit erlauben, zum Beispiel 1.000 Komplexität pro Minute.

Wenn die Clients und ihre Anfragen bekannt und einigermaßen stabil sind, dann kann man auch mit Persisted Queries arbeiten *[16]*. Dann überträgt der Client nicht den vollen Query-String, sondern nur einen Hash davon. Das verbessert einerseits die Performance, andererseits kann der Service auch nur diese manuell geprüften Queries zulassen und darüber zu teure oder unsichere Anfragen verhindern. Wenn man das möchte, kann man auch die Bereitstellung des Schemas, die sogenannte Introspection, ausschalten. Das ist jedoch für Angreifer keine wirkliche Hürde, denn fachlich korrekte Bezeichnungen kann man auch leicht erraten. Nur eine saubere Rechteprüfung hilft da wirklich.

```
public String realName(@Source SuperHero hero) {
    return ...;
}
```

Listing 3: MicroProfile-GraphQL-Resolver-Methode

```
@GraphQlClientApi
public interface SuperHeroesApi {
    public List<SuperHero> heroes();
}

public class SuperHero {
    private String name;
}

@Inject SuperHeroesApi api;

List<SuperHero> all = api.heroes();
```

Listing 4: Beispiel MicroProfile GraphQL Client

MicroProfile GraphQL Client

Aktuell wird bei MicroProfile auch an einem API für GraphQL Clients gearbeitet. Dieses soll aus einem dynamischen API und einem darauf aufsetzenden typsicheren API bestehen. Letzteres ist seit 1.0.1 bereits Teil von SmallRye GraphQL und kann über eine Dependency `io.smallrye:smallrye-graphql-client` angezogen werden. Diese orientiert sich stark am MicroProfile REST Client *[17]*, das heißt, man spezifiziert seine DTOs und das API direkt im Client genau so, wie man es braucht. *Listing 4* zeigt einen Beispiel-Client in einer CDI-Umgebung.

Das API muss ein Interface und als `@GraphQlClientApi` annotiert sein, während eine `@Query`-Annotation an den Methoden optional ist. Da wir im POJO `SuperHero` nur das Feld `name` deklarieren, wird das Feld `realName` auch nicht angefragt.

Die Namen sowohl der Felder als auch der Methoden und Parameter können von denen im Schema des Service abweichen. Vielleicht möchte man im Client beispielsweise lieber mit `secretIdentiy` statt `realName` arbeiten. Dazu kann man das jeweilige Element mit `@Name` annotieren *(siehe Listing 5)*. Auf diesem Weg kann man auch die gleiche Query mehrfach mit unterschiedlichen Rückgabetypen in demselben API deklarieren, um je nach Anforderung Helden mit ihrer Geheimidentität anzufragen oder ohne.

Es hat einige Vorteile, wenn man den Client mit komplett eigenen Klassen aufbaut: Dadurch fragt man nur genau die Felder an, die man auch tatsächlich benötigt und kann alles ganz einfach gegen Mocks getestet entwickeln. Man denkt in der eigenen Domäne und den eigenen Anforderungen, anstatt sich zu sehr vom Modell des Service leiten zu lassen. Das ist unabhängig davon, ob man inside-out oder outside-in arbeitet. Das ist praktiziertes Domain-driven Design (DDD), auch wenn es etwas mehr Arbeit bedeuten kann. Eine Alternative ist es, Client-Code aus GraphQL Query Strings zu generieren, und auch daran wird zurzeit gearbeitet.

```
@GraphQlClientApi
public interface SuperHeroesApi {
    public List<SuperHero> heroes();

    @Name("heroes")
    List<SuperHeroWithSecretIdentity> heroesWithIdentity();
}

public class SuperHero {
    private String name;
}

public class SuperHeroWithSecretIdentity {
    private String name;
    @Name("realName")
    private String secretIdentity;
}
```

Listing 5: Verwendung der @Name-Annotation

Fazit

Es ist aktuell nicht sinnvoll, geschweige denn nötig, alle REST-Schnittstellen auf GraphQL umzubauen. Wenn die Systeme und die dafür verantwortlichen Teams fachlich vertikal geschnitten werden können, dann sind BFFs vollkommen ausreichend. Ein Service kann genau die Daten liefern, die sein Client braucht, ohne zwischen mehreren Teams koordinieren zu müssen. Auch eine echte RESTful-Architektur kann sinnvoll sein, wenn ein großer Teil der Anfragen gegebenenfalls auch im Client gecacht werden kann, sich Daten also nicht so oft ändern oder Änderungen nicht so dringend sind.

Gerade für Public APIs kommen diese Ansätze aber mit steigender Komplexität an ihre Grenzen. Um in dieser Situation die Clients von den Services fachlich zu entkoppeln, ist GraphQL eine vielversprechende Lösung. Die Technologie ist ausgereift und auch in der Java-Welt praktikabel einsetzbar. Zukünftige Entwicklungen von GraphQL sind nur Ergänzungen, um es auch in weiteren Szenarien einsetzen zu können, wie etwa Subscriptions, mit denen man über Änderungen aktiv informiert wird. Die Standardisierung in Java mit MicroProfile ist in vollem Gange und spannend zu beobachten.

Die größte Hürde für den praktischen Einsatz ist sicherlich das meist fehlende Know-how, auch wenn die Lernkurve tatsächlich deutlich flacher ist, als sie es bei REST-Services war. Mit dem wachsenden Know-how wird man allerdings auch immer mehr Fälle finden, in denen einem die Konzepte von GraphQL die tägliche Arbeit erleichtern können.

Manche Leute behaupten, GraphQL sei das neue REST und würde schon bald alle REST-Services ersetzen. Ganz so enthusiastisch bin ich da nicht, denn in vielen Einsatzszenarien ist GraphQL nur ein kleines bisschen besser – für einen Umstieg lohnt sich da der Aufwand nicht. Teams, die nur wenig dringenden Bedarf für GraphQL haben, werden noch lange auch für Neuentwicklungen auf REST setzen. Wichtig ist nur, dass sie die Augen offenhalten und erkennen, wenn sie in eine Situation geraten, bei der ein Umstieg große Vorteile bringen würde. Auf lange Sicht kann es jedoch auch sehr wohl sein, dass immer mehr Teams mit ausreichender GraphQL-Erfahrung beschließen, all ihre Services umzustellen. Es bleibt spannend!

Quellen

[1] *https://de.wikipedia.org/wiki/Gesetz_von_Conway*
[2] *https://martinfowler.com/articles/consumerDrivenContracts.html*
[3] *https://www.eventstorming.com*
[4] *https://martinfowler.com/articles/micro-frontends.html*
[5] Forsgren, Humble, Kim (2018): Accelerate – The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press, Portland.
[6] *https://www.webcomponents.org*
[7] *https://en.wikipedia.org/wiki/Bus_factor*
[8] *https://www.totalteambuilding.com.au/ideal-team-size/*
[9] *https://reactjs.org*
[10] *https://nodejs.org*
[11] *https://www.phillip-kruger.com/post/microprofile_graphql_introduction*
[12] *https://www.openapis.org*
[13] *https://swagger.io*
[14] *https://github.com/graphql/graphiql/blob/master/packages/graphiql/README.md*
[15] *https://www.howtographql.com/advanced/4-security*
[16] *https://blog.codecentric.de/en/2020/05/how-to-secure-a-graphql-service-using-persisted-queries*
[17] *https://download.eclipse.org/microprofile/microprofile-rest-client-1.4.1/microprofile-rest-client-1.4.1.html*

Rüdiger zu Dohna
codecentric AG
ruediger.dohna@codecentric.de

Für Rüdiger ist Agilität keine Methode, sondern ein Wertekanon, der auf uneingeschränkter Kundenorientierung basiert, was auch das Aussprechen unangenehmer Wahrheiten beinhaltet. Das praktiziert Rüdiger schon deutlich länger, als es das Agile Manifest gibt, in sehr unterschiedlichen Rollen, Branchen, Firmenkulturen und Technologiestacks. Seit Januar 2018 ist er Codecentricer in Karlsruhe.

Microservices mit GraalVM

Wolfgang Nast, MT AG

Zur Entwicklung von Microservices in Java steht eine Vielzahl von Frameworks bereit. Mehrere dieser Frameworks unterstützen die neueste virtuelle Maschine (VM) von Oracle, die GraalVM [1]. Das interessanteste Feature der GraalVM ist, dass umgebungsoptimierte Applikationen erstellt werden können, sogenannte Native Images. Dabei sind noch ein paar Besonderheiten zu beachten, um damit kleine und schnelle Microservices zu erstellen.

Allgemeines zu GraalVM

GraalVM unterstützt die Java-Versionen 8 und 11. Die aktuellen Versionen sind 19.3 (LTS) und 20.0. Es gibt zwei Editionen der GraalVM, die CE (Community Edition) *[2]* und die EE (Enterprise Edition). Da die EE keine Docker-Unterstützung mitbringt, liegt unser Fokus doch eher auf der CE. Es werden noch andere Sprachen wie JavaScript, Python, R und Ruby unterstützt. Diese werden hier nicht betrachtet, im Vordergrund stehen Java und andere JVM-Sprachen wie Kotlin, Clojure, Scala und Groovy. Als Betriebssysteme werden Linux, MacOS und Windows unterstützt. Für Microservices ist Linux die beste Umgebung, da die meisten Microservices in Docker betrieben werden. Die wichtigste Eigenschaft ist das Native Image unter Linux. Aktuell werben Helidon SE *[3]*, Micronaut *[4]* und Quarkus *[5]* mit der Unterstützung von GraalVM und den Native Images.

Native-Image-Vorbereitung

Es empfiehlt sich, unter Linux die ersten Tests mit der GralVM und Native Image zu unternehmen. Gerne kann man dafür auch Container verwenden. Zu beachten ist, dass auch eine C/C++ Build-Umgebung benötigt wird. Eine gute Beschreibung dazu ist im Internet *[6]* *[7]* zu finden. Für Linux sollten die Pakete passend zu den Distributionen installiert werden *(siehe Listing 1)*.

Bei Docker Images ist darauf zu achten, dass diese Pakete auch installiert sind. Nützliche Images sind:

1. `quay.io/quarkus/centos-quarkus-maven:20.0.0-java8`
2. `oracle/graalvm-ce:20.0.0-java8`

```
# dnf (rpm-based)
sudo dnf install gcc glibc-devel zlib-devel libstdc++-static
# Debian-based distributions:
sudo apt-get install build-essential libz-dev zlib1g-dev
```

Listing 1: Die Linux-Pakete

```
FROM oracle/graalvm-ce:20.0.0
ARG MAVEN_VERSION=3.6.3
ARG USER_HOME_DIR="/root"
ARG SHA=c35a1803a6e70a126e80b2b3ae33eed961f83ed74d18fcd16909b2d44d7dada3203f1ffe726c17ef8dcca2dcaa9fca676987befeadc9b9f759967a8cb77181c0
ARG BASE_URL=https://apache.osuosl.org/maven/maven-3/${MAVEN_VERSION}/binaries

RUN mkdir -p /usr/share/maven /usr/share/maven/ref \
  && curl -fsSL -o /tmp/apache-maven.tar.gz ${BASE_URL}/apache-maven-${MAVEN_VERSION}-bin.tar.gz \
  && echo "${SHA}  /tmp/apache-maven.tar.gz" | sha512sum -c - \
  && tar -xzf /tmp/apache-maven.tar.gz -C /usr/share/maven --strip-components=1 \
  && rm -f /tmp/apache-maven.tar.gz \
  && ln -s /usr/share/maven/bin/mvn /usr/bin/mvn

ENV MAVEN_HOME /usr/share/maven
ENV MAVEN_CONFIG "$USER_HOME_DIR/.m2"
RUN gu install native-image
```

Listing 2: Das Dockerfile zum Erweitern der GraalVM um `native-image` und Maven

Beim ersten Image sind schon Maven *[8]* und die GraalVM-Erweiterung `native-image` installiert. Beim zweiten Image fehlen noch Maven und die GraalVM-Erweiterung, dafür ist es das offizielle Image. Von beiden Images gibt es auch noch die Java-11-Implementierung. Dabei ist nur der Teil `java8` durch `java11` zu ersetzen.

Um das offizielle Image zu erweitern, habe ich den in *Listing 2* gezeigten `dockerfile` verwendet. Zum Erstellen des Docker Image ist der Befehl `docker build -t graalvm-ce-native-mvn:20.0.0` zu verwenden. Damit sind die Vorbereitungen für `native-image` abgeschlossen.

Erstellen von Native Images

Der Aufruf, um aus einem Java-Programm ein natives Programm zu erstellen, ist einfach. Man muss nur im Java-Aufruf das Programm `java` durch `native-image` ersetzen. Heißt also der normale Aufruf `java -jar microservice.jar`, so wird daraus `native-image -jar microservice.jar` oder mit dem Docker Image `docker run -it -v`

```
[
  {
    "name" : "java.lang.Class",
    "allDeclaredConstructors" : true,
    "allPublicConstructors" : true,
    "allDeclaredMethods" : true,
    "allPublicMethods" : true,
    "allDeclaredClasses" : true,
    "allPublicClasses" : true
  },
  {
    "name" : "java.lang.String",
    "fields" : [
      { "name" : "value", "allowWrite" : true },
      { "name" : "hash" }
    ],
    "methods" : [
      { "name" : "<init>", "parameterTypes" : [] },
      { "name" : "<init>", "parameterTypes" : ["char[]"] },
      { "name" : "charAt" },
      { "name" : "format", "parameterTypes" : ["java.lang.String", "java.lang.Object[]"] }
    ]
  },
  {
    "name" : "java.lang.String$CaseInsensitiveComparator",
    "methods" : [
      { "name" : "compare" }
    ]
  }
]
```

Listing 3

```
@AutomaticFeature
class RuntimeReflectionRegistrationFeature implements Feature {
  public void beforeAnalysis(BeforeAnalysisAccess access) {
    try {
      RuntimeReflection.register(String.class);
      RuntimeReflection.register(/* finalIsWritable: */ true, String.class.getDeclaredField("value"));
      RuntimeReflection.register(String.class.getDeclaredField("hash"));
      RuntimeReflection.register(String.class.getDeclaredConstructor(char[].class));
      RuntimeReflection.register(String.class.getDeclaredMethod("charAt", int.class));
      RuntimeReflection.register(String.class.getDeclaredMethod("format", String.class, Object[].class));
      RuntimeReflection.register(String.CaseInsensitiveComparator.class);
      RuntimeReflection.register(String.CaseInsensitiveComparator.class.getDeclaredMethod("compare", String.class,
String.class));
    } catch (NoSuchMethodException | NoSuchFieldException e) { ... }
  }
}
```

Listing 4

`projectDir/target:/opt/prj graalvm-ce-native-mvn:20.0.0 native-image -jar microservice.jar`.

Jetzt wird es spannend. Leider kann noch nicht alles an Java-Programmen in native Programme übersetzt werden. Bei der Verwendung von Reflections gibt es eine Warnung, dass in den Mixed Mode gewechselt wird. Das bedeutet, dass für die Ausführung des nativen Programms zusätzlich noch die JVM benötigt wird. Das ist nicht, was wir erreichen wollten. Deshalb sollte noch der Parameter `--no-fallback` übergeben werden. Für die Verwendung von HTTP und HTTPS gibt es die Parameter `--enable-http` beziehungsweise `--enable-https`.

Da das Native Image möglichst klein sein und keinen toten Code verwenden soll, wird über den normalen Aufrufweg, die zu verwendende Main-Klasse, gegangen. Dabei wird dann eine statische Codeanalyse vorgenommen. Wenn bei der Analyse festgestellt wird, dass ein dynamischer Code-Teil aufgerufen wird, wie beispielsweise Reflections, so führt dies zu einem Problem. Hier ist nicht klar, welche weiteren Klassen aufgerufen werden. Deshalb wird ohne `-no-fallback` der „Fallback" auf die JVM und die Java-Klassen vorgenommen. Da der dynamische Teil der Klassen nicht berücksichtigt wird, kann es vorkommen, dass die native Anwendung unvollständig ist. Um dies zu prüfen, ist das Native Image zu testen. Eine Beschreibung der problematischen Code-Implementierungen ist im Web zu finden *[9]*. JVM-Sprachen und Frameworks, die häufig Probleme machen, sind Groovy, Scala, Spring und Spring Boot. Für Spring Boot gibt es ein Projekt, das hier eine Lösung umsetzt *[10]*.

Es ist auch möglich, explizit anzugeben, welche Klassen, Methoden und Felder noch berücksichtigt werden sollen. Dazu verwenden wir `-H:ReflectionConfigurationFiles=/path/to/reflectconfig`. Ein Beispiel dafür, wie diese Informationen angegeben werden können, ist in *Listing 3* zu sehen. Alternativ kann auch die Konfiguration in einer Klasse vorgenommen werden *(siehe Listing 4)*.

Eine Beschreibung der zusätzlichen Abhängigkeiten ist ebenfalls im Internet zu finden *[11]*. Damit ist es möglich, auch Java Code, der nicht direkt unterstützt wird, in Native Images umzuwandeln.

Zusammenfassung

Es ist mit unterschiedlichem Aufwand möglich, Java-Applikationen in native Applikationen zu verwandeln. Diese Applikation ist dann deutlich kleiner und schneller und kommt ohne JVM aus. Benötigt werden hier andere Docker Images. Da dynamischer Code Probleme macht, sollte man bei der Auswahl des Microservices-Frameworks darauf achten, dass Native Images unterstützt werden.

Quellen

[1] Das GraalVM-Projekt: *https://www.graalvm.org/*

[2] Die GraalVM-CE-Releases: *https://github.com/graalvm/graalvm-ce-builds/releases*

[3] Das Oracle-Helidon-Projekt: *https://helidon.io/*

[4] Das Quarkus-Projekt: *https://quarkus.io/*

[5] Das Micronaut-Projekt: *https://micronaut.io/*

[6] Die Beschreibung für native Images mit der GraalVM und Quarkus: *https://quarkus.io/guides/building-native-image*

[7] Die Beschreibung für native Images mit der GraalVM: *https://www.graalvm.org/docs/reference-manual/native-image/*

[8] Das Apache-Maven-Projekt: *https://maven.apache.org/*

[9] Die Beschränkungen von Native Image mit der GraalVM: *https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md*

[10] Das Projekt, das Spring und die GraalVM zusammenbringt: *https://github.com/spring-projects-experimental/spring-graal-native*

[11] Die Beschreibung, wie dynamische Abhängigkeiten konfiguriert werden können in der GraalVM: *https://github.com/oracle/graal/blob/master/substratevm/REFLECTION.md*

Wolfgang Nast

MT AG

Wolfgang.nast@mt-ag.com

Wolfgang Nast arbeitet seit der Version 1.0 mit Java und hat die unterschiedlichsten Unternehmen bei der Einführung und Umsetzung mit Java begleitet. Aktuell unterstützt er sie bei der Einführung von Microservices und CI/CD-Pipelines.

Saga-Orchestrierung mit Imixs-Microservices

Ralph Soika, Imixs GmbH

Jedes moderne Projekt setzt heute auf eine Microservices-Architektur. Damit lassen sich einzelne Funktionen besser kapseln und schneller in kleinen Teams umsetzen und betreiben. Jeder, der in einem größeren Projekt arbeitet, merkt aber auch hier, dass die Komplexität nicht sinkt. Gerade wenn unterschiedliche Services in einem zusammenhängenden Geschäftsprozess koordiniert werden müssen, kann es schnell unübersichtlich werden. Saga-Orchestratoren bieten hierfür eine interessante Lösung.

Eine Kernidee der Microservices-Architektur ist es, die fachlichen Anforderungen auf einzelne „Micro"-Services zu verteilen. Über die Vorteile wurde bereits viel geschrieben. Ein Aspekt, der bei diesem neuen Architekturstil immer wieder auftaucht, ist die Verteilung der Verantwortlichkeiten. Dies ergibt durchaus Sinn, da dadurch auch die Komplexität von großen Datenstrukturen aufgelöst werden kann. An dieser Stelle kommt das Konzept des *„Bounded Context"* zum Einsatz.

Domain-driven Design

Die Idee hinter dem Begriff *„Bounded Context"* stammt ursprünglich aus dem *Domain-driven Design (DDD)* und wurde bereits im Jahr 2003 entwickelt. Die Idee hier ist es, die Fachlogik möglichst Technologie-neutral in einem Domänen-Modell zu beschreiben, um so die Auf-

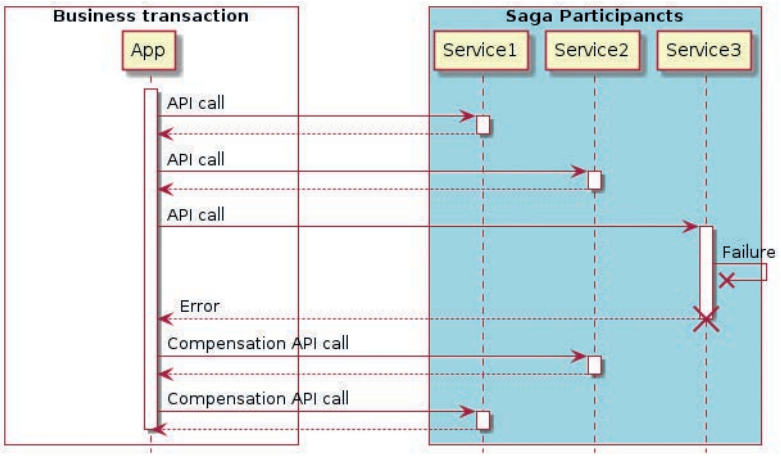


Abbildung 1: Das Saga-Pattern (© [Imxis-Workflow])

teilung in kleinere Blöcke besser zu unterstützen. Konkret bedeutet das, dass sich die einzelnen Teams zusammen mit den Fachbereichen Gedanken darüber machen, welche fachlichen Daten in einen Microservice gehören und welche nicht. Der einzelne Microservice trägt damit gleichzeitig die Verantwortung für seine Daten. Diese werden dazu oft in einer „lokalen" Datenbank unter Verwendung des Transaktionsprinzips verwaltet.

Verteilte Geschäftsprozesse

Problematisch wird die Sache mit der Verantwortung erst dann, wenn Daten zwischen einzelnen Services ausgetauscht werden müssen. Der Grund für den Austausch der Daten und die damit verbundene Verknüpfung der einzelnen Microservices untereinander sind hier nicht die Daten, sondern der dahinter liegende Geschäftsprozess. Und dieser lässt sich so gar nicht in eine Datenbank-orientierte Sichtweise zwängen. Geschäftsprozesse beschreiben den eigentlichen Zweck einer Anwendung aus Business-Sicht. Es geht hier weniger um die Daten, sondern vielmehr darum zu beschreiben, wie ein bestimmtes betriebswirtschaftliches Ziel erreicht werden kann. Sei es der Verkauf von Produkten aus einem Onlineshop, die Abwicklung von Serviceleistungen auf einer Internetplattform oder die Steuerung von Produktionsabläufen in technischen Systemen. In jedem Fall müssen hierzu Daten zwischen den einzelnen Services ausgetauscht werden.

Nun stellt sich schnell die Frage, wer in solch einem verteilten System die Verantwortung für die Datenkonsistenz über die Grenzen einzelner Microservices hinweg behält. Beispielsweise muss bei einem Onlineshop die Aktualisierung des Lagerbestandes mit der Logistik und der Zahlungsabwicklung synchronisiert werden, um nicht Gefahr zu laufen, am Ende inkonsistente Daten in einzelnen Systemen zu haben. Das sogenannte „Saga-Pattern" *[1]* ist hier ein gängiges Entwurfsmuster, um solche Abhängigkeiten zu beschreiben und zu implementieren.

Das Saga-Pattern

Eine Saga beschreibt eine Folge lokaler Transaktionen innerhalb einer Microservices-Architektur mit dem Ziel, Daten über mehrere Services hinweg zu aktualisieren. Die Saga beschreibt zum einen die Reihenfolge der Aktualisierungen und zum anderen, wie in einem Fehlerfall bereits festgeschriebene Daten nachträglich korrigiert und damit die Datenkonsistenz des gesamten Systems sichergestellt werden kann.

Konkret bedeutet das, dass, sobald innerhalb eines Geschäftsvorgangs der Aufruf eines einzelnen Microservice fehlschlägt, die Saga die bereits festgeschriebenen Änderungen einzelner Microservices explizit rückgängig machen muss. Dabei spielt es keine Rolle, ob der Fehler technisch oder fachlich bedingt ist. Dies wird in *Abbildung 1* veranschaulicht. Der Geschäftsvorgang besteht aus einer Reihe verschiedener Service-Calls. Schlägt der letzte Aufruf fehl, müssen die vorangegangenen Änderungen der Service-Calls 1 und 2 wieder korrigiert werden. Dies wird als Kompensation von Transaktionen bezeichnet und ist ein wichtiges Konzept, um die Datenkonsistenz innerhalb eines lang laufenden Geschäftsvorgangs aufrechtzuerhalten.

Choreografie oder Orchestrierung?

Eine Saga-Implementierung besteht also aus einer logischen Abfolge einzelner Service-Calls. Doch wer koordiniert diese Aufrufe? Hier gibt es zwei mögliche Lösungsansätze: die Choreografie und die Orchestrierung.

Choreografie

Bei der Verwendung von Choreografie erfolgt die Koordinationslogik über Events. Dazu abonniert ein Saga-Teilnehmer die Events der jeweils anderen Teilnehmer und veröffentlicht daraufhin selbst eigene Events. Für die Kommunikation können diese dann entweder direkt an den nächsten Microservice versendet werden oder über einen zentralen Nachrichtenbroker – beispielsweise eine Kafka-Queue – gesammelt und asynchron verteilt werden. Auf den ersten Blick scheint diese Art der Koordinationslogik einfach realisierbar. In der Praxis ist sie jedoch meist schwer zu verstehen. Der Grund ist, dass es im Code keine einzelne Stelle gibt, die die Saga vollständig beschreibt. Vielmehr verteilt sich die Umsetzung der Saga auf die einzelnen Services und deren Implementierung von Ereignissen. Ändert sich der dahinter liegende Geschäftsprozess und somit die Saga, müssen unter Umständen mehrere Microservices gleichzeitig

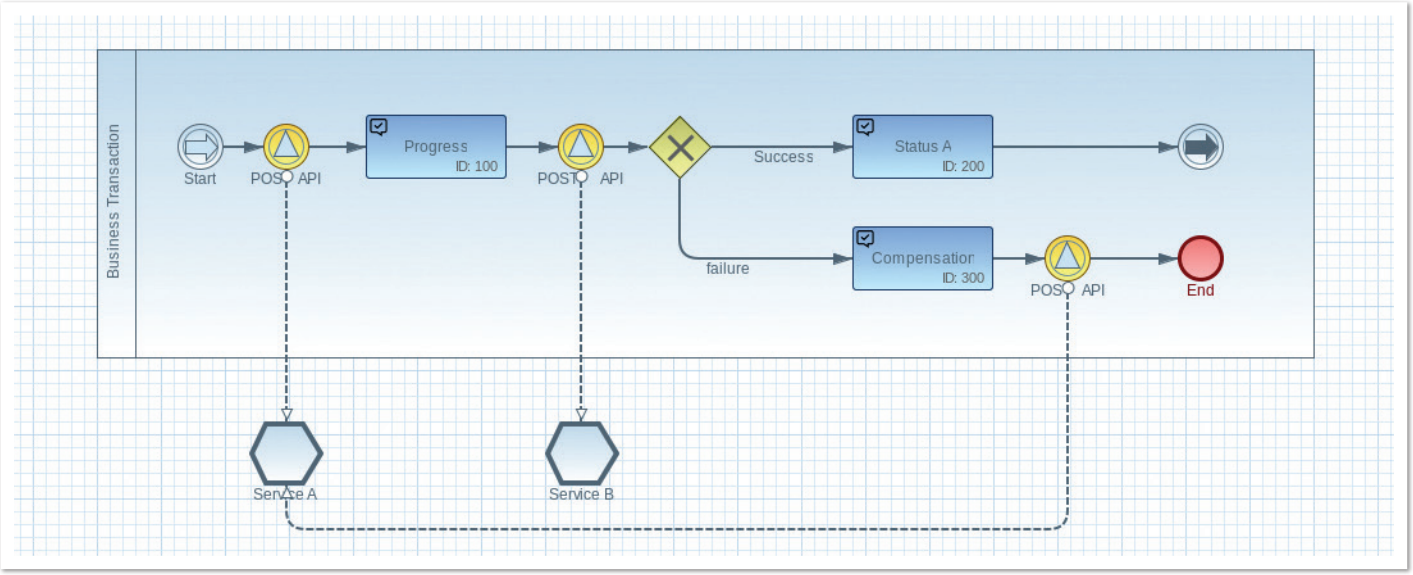


Abbildung 2: BPMN-Modell (© [Imixs-Workflow])

angepasst werden. Ein weiteres Problem sind zyklische Abhängigkeiten, die bei komplexer werdenden Geschäftsprozessen schnell entstehen können.

Orchestrierung

Die Orchestrierung stellt eine alternative Möglichkeit dar, eine Saga zu implementieren. Bei der Verwendung von Orchestrierung ist es die alleinige Verantwortung eines Saga-Orchestrators, die Saga-Teilnehmer darüber zu informieren, was zu tun ist. Der Saga-Orchestrator tritt hierbei selbst als Microservice auf, der mit den Teilnehmern wahlweise synchron oder asynchron kommuniziert. Bei jedem neuen Prozessschritt ruft dieser einen bestimmten Teilnehmer auf. Der Vorteil: Der Saga-Orchestrator ruft die Teilnehmer direkt über ein API auf und muss nicht über Ereignisse von außen informiert werden. Infolgedessen ist der Orchestrator von den Teilnehmern abhängig und nicht umgekehrt. Die Teilnehmer haben selbst keine Kenntnis von der Saga. Es gibt weniger Kopplung und auch keine zyklischen Abhängigkeiten.

Model-driven Design mithilfe von BPMN

Wie kann nun das Saga-Pattern innerhalb einer Microservices-Architektur konkret mit einem Saga-Orchestrator umgesetzt werden? Da es sich hier letztendlich um die Beschreibung eines Geschäftsprozesses handelt, ist ein modellbasierter Ansatz hier naheliegend.

Die Business Process Modelling Notation (BPMN) *[2]* hat sich für die Beschreibung von Geschäftsprozessen mittlerweile als Defacto-Standard etabliert. Mithilfe von BPMN lassen sich zum einen fachliche Abläufe beschreiben, zum anderen bietet BPMN aber auch die Möglichkeit, technische Umsetzungsdetails in einem Modell zu hinterlegen. Dadurch wird das fachliche Modell für ein Softwaresystem ausführbar und ist damit sehr gut für die Umsetzung des Saga-Patterns geeignet.

Imixs-Workflow

Das Open-Source-Projekt Imixs-Workflow *[3]* bietet mit seiner Microservices-Implementierung eine einfache Lösung, um das Saga-Pattern innerhalb einer Microservices-Architektur umzusetzen.

Die Imixs-Workflow-Engine kann selbst als ein Microservice über ein RESTful API angesprochen werden. Das Unterprojekt „Imixs-Microservice" *[4]* stellt dazu einen Docker-Container bereit, der mit Kubernetes oder Docker-Swarm schnell in eine entsprechende Architektur aufgenommen werden kann.

Nach dem erfolgreichen Setup des Service kann mit der Modellierung einer Saga mithilfe des BPMN-Modeler „Imixs-BPMN" begonnen werden.

Die Modellierung

Imixs-Workflow ist eine Event-orientierte Workflow-Engine. Das heißt, die einzelnen Zustände der Saga werden als Tasks beschrieben. Die Übergänge von einem Zustand zum nächsten erfolgen mithilfe von Events. Durch die Modellierung entsprechender Gateways kann nun der Ablauf im Erfolg oder Fehlerfall entsprechend modelliert werden. *Abbildung 2* zeigt ein vereinfachtes Beispiel für eine Onlinebestellung.

Sobald eine neue Order erstellt wurde, wird über das Event „Update Stock" zunächst der Lagerbestand durch einen API-Call am Stock-Service aktualisiert. Anschließend befindet sich die Saga im Status „Payment". Hier wird ein weiterer API-Call durchgeführt, um den Bezahlvorgang über den Payment-Service abzuschließen. Schlägt dieser fehl, wird über ein Gateway eine „Compensation"-Task eingeleitet. Diese leitet über einen erneuten API-Call gegen den Stock-Service die Korrektur des zuvor gebuchten Lagerbestandes ein.

Dieses stark vereinfachte Beispiel zeigt, wie die Koordination des eigentlichen Geschäftsprozesses mit einem Model-driven Design beschrieben werden kann. Ein weiterer Vorteil von Imixs-Workflow als Saga-Orchestrator ist die Tatsache, dass der jeweilige Zustand innerhalb der Saga sowie sämtliche bereits erfolgten Schritte persistiert werden. Es kann also immer auch rückwirkend der genaue Ablauf eines Geschäftsprozesses nachvollzogen werden. Gerade in Hinblick auf Revisionssicherheit und Compliance-Richtlinien ist diese Funktionalität bei komplexen Geschäftsabläufen eine zwingende Voraussetzung.

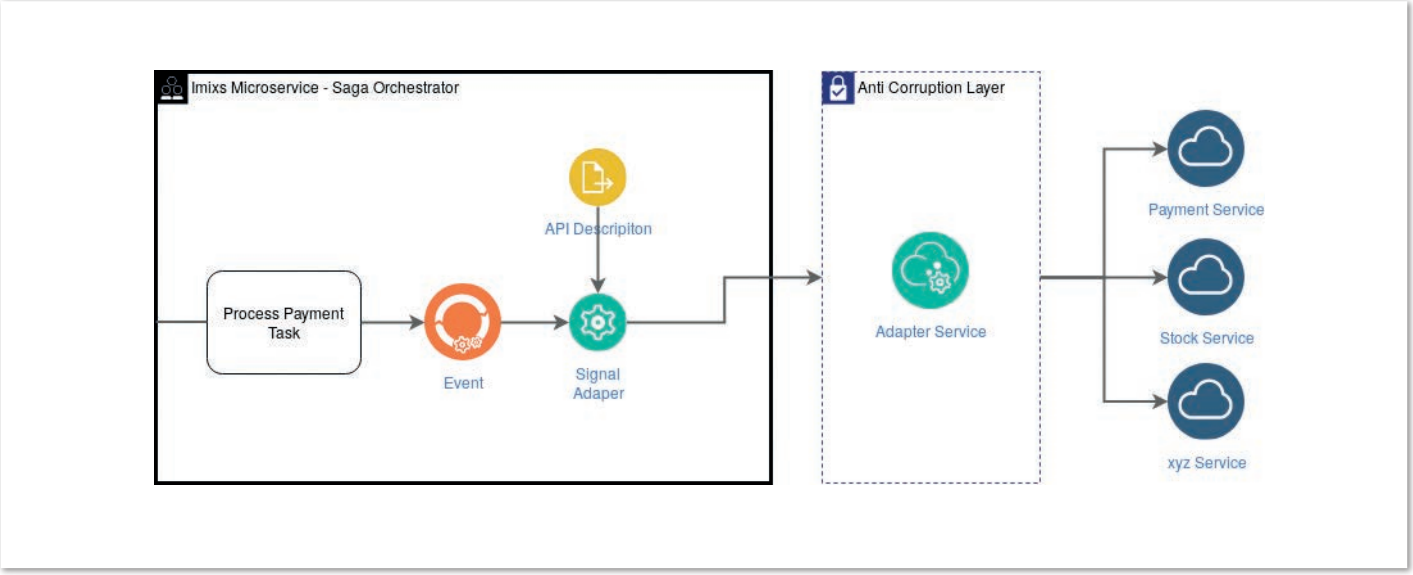


Abbildung 3: Anti-Corruption Layer (© [Imixs-Workflow])

Plug-ins und Adapter

Imixs-Workflow bietet über Plug-ins und Adapter die Möglichkeit, zusätzliche Funktionen und Dienste bereitzustellen. Diese helfen, den Implementierungsaufwand deutlich zu reduzieren.

Das Imixs-Workflow-Adapter-API *[5]* ist ein Erweiterungsmechanismus, um den Verarbeitungslebenszyklus eines BPMN-Ereignisses anzupassen. Eine Adapterklasse kann so an bestimmten Stellen innerhalb der Saga einen externen Microservice aufrufen und Daten senden oder empfangen. Das Ergebnis solch eines Aufrufs kann direkt für die weitere Prozessverarbeitung ausgewertet werden.

Listing 1 zeigt die Implementierung eines Adapters anhand eines Beispiel. Der Adapter erhält durch die Workflow-Engine Informationen zum aktuellen Vorgang sowie zum aufgerufenen Event. Diese Daten können verwendet werden, um den eigentlichen API-Call an den externen Service auszuführen und das Ergebnis entsprechend zu adaptieren. Nachfolgende Calls können so auf diese Daten zugreifen – beispielsweise auf eine Transaktions-ID.

Schlägt der Aufruf fehl oder kann dieser fachlich nicht ausgeführt werden, wird eine AdapterException ausgelöst. Diese kann nun in der weiteren Verarbeitung über das BPMN-Modell abgefangen und entsprechende Kompensationen können eingeleitet werden.

Das Anti-Corruption-Layer-Pattern

Auch bei dem hier gezeigten Saga-Orchestrator entstehen letztendlich Abhängigkeiten zu anderen Services, wenn deren APIs direkt über eine Adapter-Klasse implementiert werden. Um diese Abhängigkeit aufzulösen, kann das sogenannte *Anti-Corruption-Layer-Pattern* eingesetzt werden. Es stellt eine Zwischenschicht zwischen dem Saga-Orchestrator und dem eigentlichen API-Call dar und übersetzt die Daten zwischen zwei Systemen *(siehe Abbildung 3)*.

Der eigentliche API-Aufruf wird damit in einen eigenen Microservice gekapselt. Dadurch trennt man die beiden Schichten und der Saga-Orchestrator kann nun unabhängig von weiteren Services entwickelt werden. Sollte sich der Geschäftsprozess dahinter ändern oder

```
public class StockAdapter implements SignalAdapter {
  public ItemCollection execute(
     ItemCollection doc,
     ItemCollection event) throws AdapterException {

  Client client = ClientBuilder.newClient();
  try {
    result=client
      .target(REST_URI)
      .path(doc.getItemValue("productId"))
      .request(MediaType.APPLICATION_XML)
      .post(Order.classEntity.entity(order,
        MediaType.APPLICATION_XML));
     // adapt result
     ...
  } catch (ResponseProcessingException e) {
    throw new AdapterException(
      StockAdapter.class.getSimpleName(),
      ERROR_API_COMMUNICATION,
      "Failed to call rest api!");
  }
}
```

Listing 1: Adapter-Beispiel

kommen weitere APIs dazu, können diese Änderungen nun direkt im BPMN-Modell vorgenommen werden, ohne dass der Orchestrator-Service selbst neu deployt oder angepasst werden muss.

Die eigentliche Konfiguration des API-Aufrufs im Modell erfolgt dazu über ein API-Description-Objekt. Dieses kann eine XML- oder JSON-Struktur mit den entsprechenden Informationen aufweisen und ist Teil der BPMN-Modellierung.

Fazit

Die Komplexität innerhalb einer Microservices-Architektur kann schnell erheblich anwachsen, wenn im Zuge eines Geschäftsprozesses eine Vielzahl von Services mit unterschiedlichen Daten (Bounded Context) koordiniert werden müssen. Das Saga-Pattern bietet hier eine elegante Möglichkeit, die Zusammenhänge und Abläufe zu beschreiben.

Das Open-Source-Projekt Imixs-Workflow stellt mit dem Imixs-Microservice eine elegante Möglichkeit bereit, einen Saga-Orchest-

rator selbst als Microservice zu betreiben. Dabei können komplexe Sagas mithilfe von BPMN modelliert werden. Das Plug-in- und Adapter-API von Imixs-Workflow erlaubt die Einbindung beliebiger Services und eine Umsetzung auch komplexer Abläufe über Gateways und Business Rules.

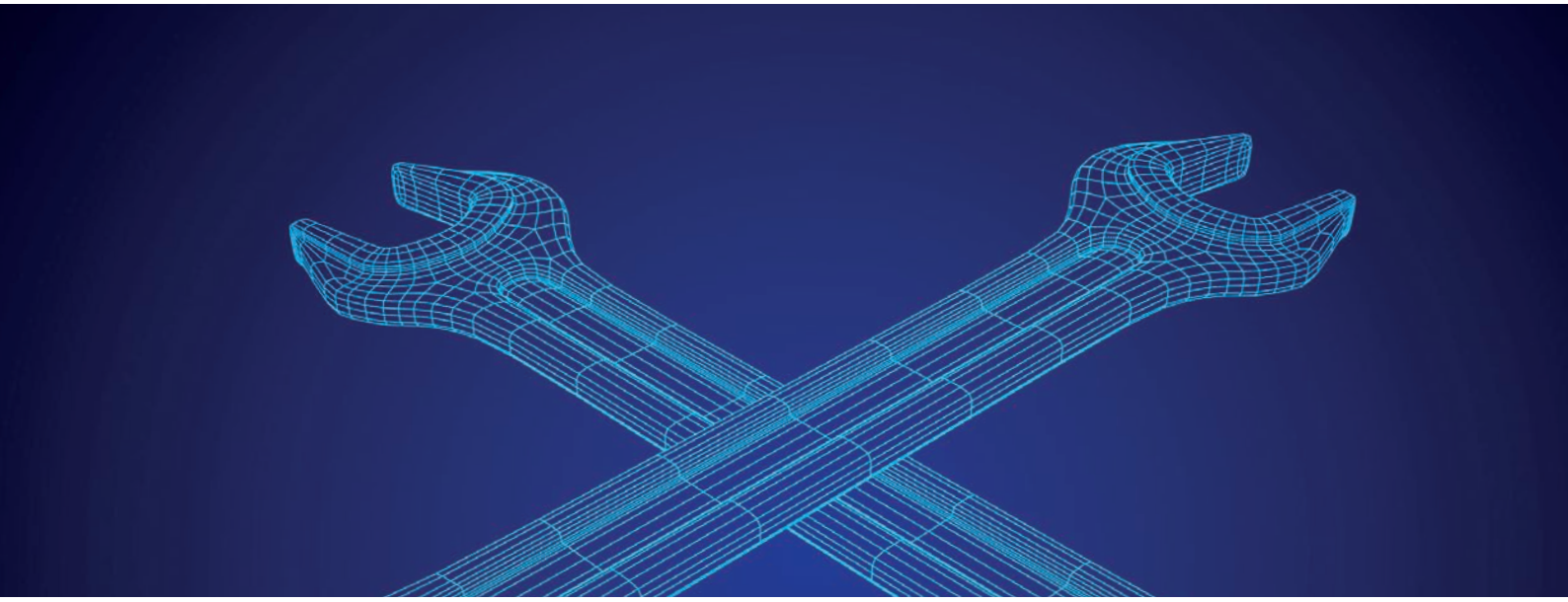
Durch den Einsatz eines Anti-Corruption-Layers kann der Saga-Orchestrator vollständig von den einzelnen Services entkoppelt werden. Dadurch werden feste Abhängigkeiten zwischen den einzelnen Services vermieden und Geschäftsprozesse lassen sich ohne Anpassung der Implementierung anpassen.

Quellen

[1] *https://microservices.io/patterns/data/saga.html*
[2] *https://www.omg.org/spec/BPMN/2.0/*
[3] *https://www.imixs.org*
[4] *https://github.com/imixs/imixs-microservice*
[5] *https://www.imixs.org/doc/core/adapter-api.html*

Ralph Soika
Imixs GmbH
ralph.soika@imixs.com

Ralph Soika ist Projectlead im Open-Source-Projekt Imixs-Workflow und Geschäftsführer der Imixs GmbH. Er berät seit mehr als 20 Jahren Unternehmen bei der Einführung von Geschäftsprozess-Management-Lösungen und Workflow-Architekturen. Ralph Soika entwickelt moderne Microservices-Architekturen und ist aktives Mitglied im Open-Source-Projekt Eclipse BPMN2 Modeler.



Service Mesh: eine Infrastruktur für Microservices

Jörg Müller, INNOQ

Service Meshes lösten als Technologie in den vergangen ein bis zwei Jahren ein hohes Interesse aus. Der Hauptgrund dafür sind die gestiegenen Anforderungen an die Infrastruktur durch den Trend zu Microservices-Architekturen. Können diese Anforderungen durch diese neue Infrastruktur gelöst werden?

Architekturen, die auf Microservices basieren, versprechen viele Vorteile. Die Services für sich selbst sind leichter zu verstehen. Einzelne Services lassen sich leichter ersetzen. Die Schnittstellen sind klar definiert. Skalierung lässt sich individueller gestalten. Vor allem erlauben sie den Einsatz der jeweils am besten passenden Technologie für den jeweiligen Service.

Diese Vorteile haben allerdings ihren Preis. Eine Microservices-Architektur ist eine verteilte Architektur. Verteilte Systeme haben neue Herausforderungen. Das Netzwerk zwischen ihnen kann ausfallen. Es entsteht eine höhere Latenz. Die Übersichtlichkeit über die Services und ihre Beziehungen untereinander leidet. Außerdem ist das Deployment neuer Funktionalitäten komplexer als einfach nur eine neue Version einer einzelnen Anwendung auszurollen.

Viele Probleme im Betrieb werden durch die Containerisierung der Anwendung und die Nutzung von Clustermanagementsystemen, wie Kubernetes, gelöst. Es bleiben aber noch eine Reihe von Herausforderungen. Statt diese individuell in der Anwendung zu lösen, ist die Grundidee von Service Meshes, dies in die Infrastruktur zu verlagern.

Dazu übernimmt ein Service Mesh die Kontrolle über den Netzwerkverkehr zwischen den einzelnen Services. Ein Service Mesh besteht aus zwei Ebenen, auf die wir später noch detaillierter eingehen. Auf der einen Ebene wird jeder Service-Instanz ein Proxy zur Seite gestellt, der sowohl eingehenden als auch ausgehenden Netzwerkverkehr überwacht und beeinflusst. Alle Proxys zusammen werden als Data Plane bezeichnet.

Die zweite Ebene bildet eine zentrale Komponente namens Control Plane. Diese steuert und überwacht die Data Plane beziehungsweise die in ihr enthaltenen Proxys. Das ist sinnvoll, da umfangreiche Anwendungen oft aus sehr vielen Services mit jeweils mehreren Instanzen bestehen. Die Kombination dieser beiden Ebenen gibt dem Service Mesh seine umfangreichen Möglichkeiten.

Features eines Service Mesh

Die Lösungen, die ein Service Mesh bietet, lassen sich in vier Bereiche einteilen:

- Observability
- Resilience
- Routing
- Security

Die **Observability** ermöglicht es, einen besseren Überblick über die Services innerhalb einer Anwendung zu erhalten. Da ist zum einen das Monitoring individueller Services bezogen auf den Netzwerkverkehr. Wie oft wird der Service aufgerufen? Welche Antwortzeiten gibt es? Wie oft treten Fehler auf? Wichtig ist, dass diese Überwachung nur auf der Ebene der Netzwerkkommunikation stattfindet. Der Service selbst ist für das Mesh eine Black-Box.

Gleichzeitig wird auch beobachtet, wie die Services untereinander kommunizieren. Welcher Service spricht mit welchen anderen Services? Welche Services sind an der Beantwortung einer bestimmten Anfrage an das Gesamtsystem beteiligt? Welcher Service könnte der Flaschenhals bei der Beantwortung sein? Diese Funktionalität ist auch als Tracing bekannt.

Service Meshes können auch dabei unterstützen, eine Anwendung widerstandsfähiger gegen den Ausfall einzelner Services zu machen, also die **Resilience** zu erhöhen. Dazu implementieren sie bekannte Methoden wie Timeouts, Wiederholungen und Circuit Breaker. Letzteres ist ein Mechanismus, der den Netzwerkverkehr zu einzelnen Services temporär unterbricht, um Kaskaden von Ausfällen zu vermeiden.

Da Service Meshes den Netzwerkverkehr steuern, sind sie natürlich auch in der Lage, das **Routing** zwischen Services oder zwischen Services und der Außenwelt zu beeinflussen. Das fängt mit einfachen Anforderungen wie dem Load Balancing an. Hier bieten Service Meshes oft schon deutlich intelligentere Funktionalitäten als die darunter liegende Infrastruktur. Neben dem Load Balancing sind auch komplexere Szenarien des Routings abbildbar. Ein typisches Beispiel ist das Canary Release. Hier wird eine neue Version zuerst für einen sehr kleinen Teil der Nutzer ausgerollt, um sie zu testen. Mithilfe eines Service Mesh kann festgelegt werden, wie hoch dieser Anteil ist, und das Mesh kümmert sich dann um die korrekte Verteilung der Aufrufe an alte und neue Versionen des Service. Als Kriterium für diese Verteilung können alle Merkmale eines HTTP-Aufrufes, wie die URL oder der Header, verwendet werden. Auch andere gängige Release-Verfahren, wie zum Beispiel Blue-Green-Releases, können so mithilfe eines Service Mesh abgebildet werden.

Der letzte Bereich, in dem ein Service Mesh Lösungen bietet, ist die Implementierung von Maßnahmen der **Security**. Dazu zählt erstens die Verschlüsselung der Kommunikation der Services untereinander. Durch die zentrale Steuerung über die Control Plane ist es recht einfach möglich, mTLS (Mutual TLS) zu nutzen. TLS ist als Verschlüsselung im Internet bekannt. Dabei wird über Zertifikate sichergestellt, dass der Client tatsächlich mit dem Server redet, den er erreichen möchte. Bei mTLS findet dies in beide Richtungen statt. Auch der Server kann über Zertifikate die Identität des Clients prüfen. Neben der Verschlüsselung können einige Service Meshes auch steuern, welche Services miteinander kommunizieren dürfen. Von manchen werden ebenfalls Authentifizierungen der Endnutzer unterstützt.

Wie ein Service Mesh funktioniert

Um diese detaillierte Kontrolle über den Netzwerkverkehr zu übernehmen, nutzt ein Service Mesh in der Data Plane einen Proxy. In den meisten Implementierungen existiert genau ein Proxy pro Service-Instanz. Setzt das Service Mesh auf Kubernetes auf, kommt dazu das Sidecar-Pattern zum Einsatz. Kubernetes kennt neben dem Container noch eine weitere Gruppierung, den Pod. Dieser kann mehrere Container beinhalten, die sich dann ein gemeinsames Filesystem und das Netzwerk teilen. Damit ist es möglich, dass der Proxy quasi wie ein Beiwagen eines Motorrads bei jedem relevanten Pod mit ausgerollt wird. Es ist durch einen Injektions-Mechanismus sogar möglich, dieses Sidecar automatisch zu jedem Pod hinzuzufügen, der gestartet wird. Es muss also kein gesonderter Aufwand getrieben werden, damit ein Service Teil des Mesh wird.

Die Nützlichkeit des Sidecar-Patterns kombiniert mit den Möglichkeiten der Injektion ist sicher ein wesentlicher Grund, warum die meisten Service Meshes auf Kubernetes aufsetzen. Viele Implementierungen setzen Kubernetes sogar voraus.

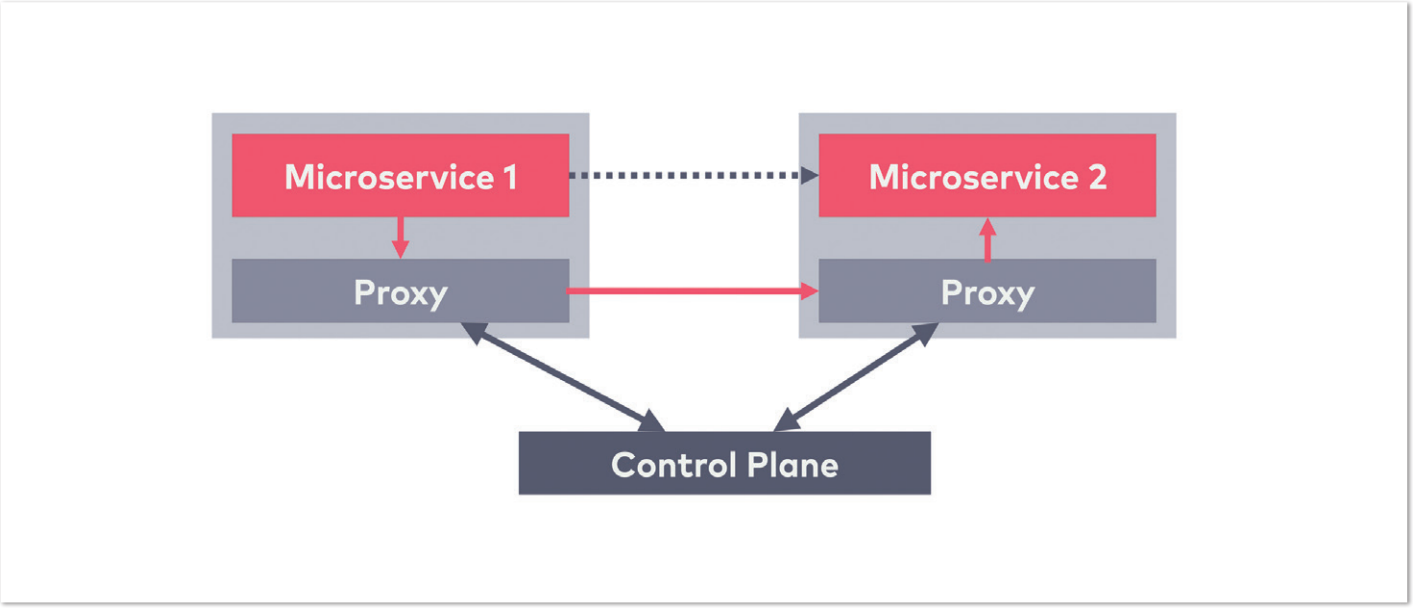


Abbildung 1: Grundlegender Aufbau eines Service Mesh

Die Control Plane steuert all diese Proxys. Dabei erfüllt sie eine Reihe von Aufgaben. Sie kommuniziert mit der zugrunde liegenden Infrastruktur, um Informationen über die installierten Services zu erhalten. Sie konfiguriert die Proxys der Data Plane mit Informationen zum Routing und zu erlaubter Kommunikation. Dazu zählt die Verwaltung der Zertifikate für eine sichere Kommunikation zwischen den Services. Schließlich sammelt die Control Plane die Daten der Services und stellt sie Monitoring-Systemen bereit beziehungsweise bietet eigene Dashboards. Eine Benutzeroberfläche für die Überwachung und die Konfiguration des Service Mesh ist ebenfalls typischer Bestandteil der Control Plane.

Service-Mesh-Implementierungen

Entsprechend der hohen Popularität des Konzepts „Service Mesh" gibt es inzwischen einige Implementierungen. Die bekanntesten sind Istio und Linkerd.

Istio *[1]* ist die Fusion von zwei Projekten bei Google und IBM. Ähnlich wie bei Kubernetes sollten die internen Erfahrungen in ein Open-Source-Projekt übergehen. Bei dem Service-Proxy fiel die Wahl auf Envoy von der Firma Lyft, der auch bei vielen anderen Service Meshes zum Einsatz kommt. Erst der Hype um Istio hat wohl zu der Entwicklung vieler anderer Service Meshes geführt. Istio ist sicher das Service Mesh mit den meisten Features. Das hat über die Zeit auch zu einer hohen Komplexität geführt. Mit dem Release von Istio 1.5 Anfang März 2020 wurde die Implementierung allerdings wieder drastisch vereinfacht, was sehr zu begrüßen ist.

Linkerd 2 *[2]* und sein Vorgänger Linkerd werden von Buoyant entwickelt, einer Firma von ehemaligen Twitter-Entwicklern. Während die erste Version von Linkerd noch völlig unabhängig von Container-Infrastruktur entwickelt wurde, setzt die zweite Version konsequent auf Kubernetes als Basis. Der Fokus von Linkerd 2 liegt vor allem auf der guten Benutzbarkeit und Konfigurierbarkeit. Außerdem erlaubt Linkerd 2 eine feingranularere Steuerung und Überwachung der Services auf Endpunkt-Ebene. Linkerd 2 ist außerdem das einzige Service Mesh im Portfolio der CNCF (Cloud Native Computing Foundation).

Weitere Implementierungen kommen oft aus dem Umfeld bereits vorhandener Produkte und ergänzen diese um Service-Mesh-Funktionalitäten. So bietet die unter anderem für seine Service-Discovery-Lösung Consul bekannte Firma Hashicorp mit **Consul Connect** ein Mesh, das sich dank Consul auch außerhalb der Welt von Kubernetes sehr wohl fühlt. AWS bietet mit **AWS App Mesh** eine Lösung, die sehr gut in die vorhandene AWS-Infrastruktur passt. Auch dieses Service Mesh setzt auf Envoy als Proxy. Die Firma Containous bietet mit **Maesh** eine Implementierung, die den ebenfalls von ihnen stammenden Proxy Traefik nutzt. Schließlich gibt es mit **Kuma** auch eine Implementierung des durch sein API Gateway bekannt gewordenen Anbieters Kong.

Eine gute Übersicht über die vorhandenen Implementierungen bietet die Website *Servicemesh.es [3]*. Dort werden auch die Features der einzelnen Implementierungen detailliert verglichen.

Service Mesh, ja oder nein?

Nachdem nun erläutert wurde, wie ein Service Mesh funktioniert, welche Features es bietet und welche Implementierungen existieren, stellt sich die Frage, wie entschieden werden kann, ob ein Service Mesh in der eigenen Architektur sinnvoll ist. Als Hilfe zu dieser Entscheidung können eine Reihe von Fragen genutzt werden:

- Wie viele Services sind im Einsatz?
- Wie technisch heterogen sind die Services implementiert?
- Erfolgt die Kommunikation zwischen Services synchron oder asynchron?
- Ist Kubernetes bereits im Einsatz?
- Wie oft ändern sich die Services?

Es gibt keine absolute Zahl an Services, ab der der Einsatz eines Service Mesh sinnvoll ist. Wenn weniger als fünf verschiedene Services vorliegen, sollten allerdings triftige Gründe vorliegen, um ein Service Mesh zu rechtfertigen. Dies könnte zum Beispiel der Einsatz des oben bereits erwähnten mTLS sein. Dieses aufzusetzen ist generell ein hoher Aufwand, wodurch sich das Service Mesh bereits bei einer geringeren Zahl an Services lohnen kann.

Die nächste Frage beschäftigt sich mit der technischen Heterogenität der Services. Basieren sie alle auf einer gemeinsamen technischen Basis, wie zum Beispiel Java und Spring Boot, so lohnt sich möglicherweise der Einsatz von Bibliotheken oder Frameworks, die die oben genannten Problemfelder lösen. Resilience, Service Discovery oder Monitoring können so ebenfalls umgesetzt werden. Service Meshes spielen ihre Stärke da aus, wo unterschiedliche Tech-Stacks miteinander interagieren. Dort müsste sonst für jeden Stack eine eigene Lösung gesucht werden.

Ein weiterer zentraler Punkt ist die Art der Kommunikation zwischen Services. Service Meshes sind besonders nützlich, wenn die Kommunikation synchron erfolgt. Bei asynchroner Kommunikation, die zum Beispiel über Messaging-Systeme erfolgt, ist der Mehrwert eines Service Mesh geringer.

Sehr viele Service-Mesh-Implementierungen setzen auf Kubernetes auf. Dementsprechend wird die Einführung aufwendiger, wenn Kubernetes nicht im Einsatz ist. Gleichzeitig ist vermutlich der Einsatz von Containerisierung und Kubernetes auch ein guter Hinweis darauf, dass die eigene Servicelandschaft einen Komplexitätsgrad erreicht hat, der Service Meshes sinnvoll macht.

Letztlich ist auch entscheidend, wie oft sich Services tatsächlich ändern. Dies betrifft sowohl den Rollout neuer Versionen als auch die Änderung der Skalierung, also wie viele Instanzen eines Service jeweils im Einsatz sind. Eine Servicelandschaft, in der neue Versionen nur alle paar Wochen ausgerollt werden und in der von jedem Service eine feste Anzahl von Instanzen existiert, benötigt vermutlich nicht den Automatisierungsgrad, den Service Meshes bieten. Eine manuelle Konfiguration, einfache Rollout-Skripte und statisch konfiguriertes Monitoring können dann vollkommen ausreichend sein. Das soll natürlich kein Plädoyer für seltene Rollouts sein. Im Gegenteil: Eine höhere Release-Frequenz ist für fast alle Projekte eine sinnvolle Entscheidung. Die Einführung eines Service Mesh ist allerdings selten ein guter erster Schritt in diese Richtung.

Fazit

Eine Microservices-Architektur steht vor anderen Herausforderungen als eine monolithische Architektur. Diese Herausforderungen basieren auf der Tatsache, dass Microservices verteilte Systeme sind.

Ein Service Mesh bietet Lösungen für diese Probleme auf der Ebene der Infrastruktur, sodass sich die Services auf ihre eigentliche Domäne konzentrieren können. Das kann in komplexen Microservices-Architekturen den Arbeitsaufwand deutlich reduzieren.

Gleichzeitig ist ein Service Mesh auch sehr komplex. Seine Installation und Wartung bedeuten einen nicht zu unterschätzenden Aufwand. Es sollte also genau überlegt werden, ob der Aufwand den Nutzen rechtfertigt. Wenn dies aber so ist, dann ist ein Service Mesh ein mächtiges Werkzeug, um vor allem nicht-funktionale Anforderungen umzusetzen.

Eine empfehlenswerte, weiterführende Einleitung zum Thema Service Meshes ist im "Service Mesh Primer" E-Book von Eberhard Wolff und Hanna Prinz zu finden *[4]*.

Ein herzlicher Dank für das Review dieses Artikels geht an meine Kollegin Hanna Prinz.

Quellen und weiterführende Informationen

[1] Istio: *https://istio.io/*
[2] Linkerd: *https://linkerd.io/*
[3] Service-Mesh-Vergleich: *https://servicemesh.es*
[4] Service Mesh Primer: *https://leanpub.com/service-mesh-primer*

Jörg Müller
INNOQ
joerg.mueller@innoq.com

Jörg Müller ist Principal Consultant bei INNOQ. Seit mehr als zwanzig Jahren arbeitet er in verschiedenen Rollen in der IT-Beratung und Softwareentwicklung. In den letzten Jahren beschäftigt er sich schwerpunktmäßig mit der Architektur und dem Betrieb von Software-as-a-Service. Aktuelle Themen sind Microservices, Continuous Delivery und Kubernetes. In der Community ist er als Autor aktiv, hält Vorträge und ist beteiligt an der Organisation der JUG Berlin-Brandenburg sowie mehrerer Konferenzen.

Secrets of Java

Elisabeth Schulz, Codecentric AG

In diesem Artikel beleuchten wir einige der weniger bekannten Features und Eigenheiten von Java. An einer Reihe von „unmöglichen" Codebeispielen wagen wir einen Blick unter die Motorhaube, der sonst oft erst nach nächtelangem Debuggen gelingt.

Fast jede Programmiersprache ist an einigen Stellen erstaunlich, wenig intuitiv oder schlichtweg sich selbst widersprechend. So auch Java. Allerdings hat Java als „alte Dame" mit immerhin 24 Jahren Geschichte noch einen weiteren Faktor, der diese Eigenheiten verstärkt: Java ist bemerkenswert auf Abwärtskompatibilität ausgerichtet (die Anzahl der wirklich inkompatiblen Änderungen ist auf wenige Schlüsselwörter begrenzt). Fast jedes gültige Java-1.0-Programm kann auch heute mit dem JDK 14 noch erfolgreich übersetzt und ausgeführt werden.

Dies ist für viele Entwickler ein Segen, da dadurch selten aufwendige Migrationen auf neuere Releases notwendig sind. Das bestehende Programm kann sogar auf dem früheren Sprachlevel übersetzt und 1:1 weiterentwickelt werden. Es ist aber gleichzeitig auch ein Fluch dahingehend, dass keine Entscheidung revidiert und kein einmal eingeführtes Verhalten wieder zurückgenommen werden kann. Im Folgenden zeigen wir einige der daraus resultierenden „dunklen Ecken" der Sprache auf.

Das JVM-Ökosystem ist groß. Zum Beispiel mit Agenten, die Klassen beim Laden umschreiben, sowie Instrumentierungen und Frameworks, die freizügig neue Klassen zur Laufzeit definieren, kann noch jede Menge weiterer Schabernack getrieben werden – aber in diesem Artikel geht es erst einmal um Kernfunktionen der Sprache, die fast jedem Java-Entwickler schon einmal begegnet sein dürften.

Checked Exceptions existieren nicht

In jedem Java-Lehrbuch wird das Konzept von Checked und Unchecked Exceptions erklärt. Nun mag man im Rückblick feststellen, dass Checked Exceptions ein Fehler waren. Man mag auch die Meinung vertreten, dass sie ein gutes Konzept sind, das nur oft falsch verwendet wurde. Unabhängig davon sind sie in Java (auch ohne Bytecode-Manipulationen) leider nicht konsequent umgesetzt –

jede Methode kann beliebige Ausnahmen auslösen, auch ohne sie zu deklarieren.

Werfen wir einen Blick auf *Listing 1*; es scheint klar, dass das Programm keine `IOException` produzieren darf. Die Methode `main` deklariert kein `throws` und darf daher maximal eine `RuntimeException` auslösen. Tatsächlich ist das allerdings abhängig von der genauen Definition der Methode `_throw()`. Wenn wir den „Kniff" im *Listing 2* verwenden, ist es trotz des fehlenden `throw` möglich.

Was ist hier passiert? Eine Kollision zwischen zwei Prinzipien. Als generische Typen in Java 5 eingeführt wurden, war ein zentrales Anliegen, dass nicht-generischer Code genauso weiter funktionieren sollte wie davor. Deshalb wurde (im Gegensatz zu Sprachen wie C++) der Ansatz der *Type Erasure* gewählt und das Konzept des *Raw Type* eingeführt. Gleichzeitig wurde das Prinzip der *Checked Exceptions* nicht geändert.

In der gewählten Umsetzung finden (grob vereinfacht) generische Typen auf Bytecode-Ebene nicht statt – alle Varianten eines generischen Typs verwenden ein und dieselbe Klassendefinition. Der Bytecode unterscheidet sich unabhängig von den gewählten Parametern nicht. Alle nötigen Prüfungen werden stattdessen durch den Aufrufer durchgeführt. Innerhalb der generischen Klasse ist ein Typ-Parameter gleichbedeutend mit seiner oberen Schranke, im Beispiel also `Throwable`. Dazu kommt noch, dass ein generischer Typ auch *Raw* verwendet werden kann. So ein Typ kann dann **jedem** generischen Typ zugewiesen werden.

Das Ergebnis ist so einfach wie wenig intuitiv. Durch den *Raw Type* ist das `Holder`-Objekt gezwungen zu vergessen, welcher Typ von `Exception` in ihm gespeichert ist. Die darauffolgende Konvertierung zum `Holder<RuntimeException>` hebelt die `throw`-Deklaration aus und ermöglicht es, die `Exception` ohne Deklaration zu werfen. Zu keinem Zeitpunkt wird der Typ noch einmal geprüft, sodass auch der aufrufende Code keinen Typcheck mehr vorsieht – `throw` wird ausgeführt, auch wenn es das „eigentlich" nicht mehr sollte.

Dass die JVM Exceptions anders behandelt als die Sprachspezifikation es vorsieht, war schon immer der Fall. Neu mit Java 5 ist lediglich, dass dieses unterschiedliche Verhalten nun auch auf Sprachebene

```
public static void main(String[] args) {
    _throw(new IOException());
}
```

Listing 1

sichtbar wird. Vorher war dies nur mit JNI oder Bytecode-Manipulationen möglich. Kurioserweise ist das Ergebnis dann übrigens eine Ausnahme, die nur auf Ebene von *Exception* oder *Throwable* gefangen werden kann, da ein *catch*-Block sich nur auf Ausnahmen beziehen kann, die auch innerhalb des Blocks deklariert wurden.

Objekte aus dem Nichts

Der Lebenszyklus eines Java-Objekts scheint auf den ersten Blick einfach – es wird mittels `new` (oder `Class.newInstance(…)` etc.) erzeugt, durchläuft seinen Konstruktor, lebt, solange es Referenzen auf dieses Objekt gibt, und wird schließlich vom Garbage Collector entfernt.

Auch in *Listing 3* scheint das Programmverhalten auf den ersten Blick klar. Das Programm kann entweder mit einer Exception abbrechen oder aber `null` ausgeben. Es wird allerdings schon ein kleiner, aber entscheidender Hinweis gegeben: `Serializable`. Dieses Interface (eingeführt in Java 1.1) wird von relativ vielen Klassen im JDK implementiert und oft als „Noise" gesehen. Aber es bietet eine Hintertür, die es ermöglicht, Objekte ohne Verwendung ihres Konstruktors zu erzeugen. In *Listing 4* ist dies beispielhaft gezeigt. Die Idee der Serialisierung ist so einfach wie mächtig: Serialisierung soll den Zustand eines Objektes einfrieren, unabhängig davon, wie es diesen Zustand erreicht hat. Beim Entwurf der Serialisierung wurde daher entschieden, an den normalen Sprachregeln vorbei eine neue Instanz zu erzeugen und direkt im Speicher zu manipulieren. Ziel war eine möglichst originalgetreue Kopie. Alternativen, wie die Verwendung von Properties, waren zu diesem Zeitpunkt noch nicht implementiert und hätten nach wie vor die Frage aufgeworfen, wie das Objekt instanziiert wird. Daher wurde der scheinbar einfache Ansatz gewählt und auf den Konstruktor komplett verzichtet.

Konstruktoren sollten normalerweise keine Seiteneffekte haben – haben sie aber leider doch gelegentlich. Insbesondere die Prüfung von Parametern wird oft über Exceptions im Konstruktor implemen-

```
class Loophole {
    private static class Holder<E extends Throwable>  {
        private final E exception;

        public Holder(E exception) {
            this.exception = exception;
        }

        void doThrow() throws E {
            throw exception;
        }
    }

    static <E extends Throwable> void _throw(E exception) {
        Holder raw = new Holder<>(exception);
        Holder<RuntimeException> sneaky = raw;
        sneaky.doThrow();
    }
}
```

Listing 2

```
public class Impossible implements Serializable {
    private static final long serialVersionUID = 1L;

    private Impossible() {
        throw new AssertionError();
    }

    public static void main(String[] args) throws Exception {
        System.out.println(getImpossible());
    }
}
```

Listing 3

```
static Impossible getImpossible() throws Exception {
    String encoded = "rO0A...";

    var bytes = Base64.getDecoder().decode(encoded);
    var str = new ObjectInputStream(new ByteArrayInputStream(bytes));
    return (Impossible) str.readObject();
}
```

Listing 4

tiert. Um das Objekt korrekt zu rekonstruieren, muss der Autor der Klasse diese Seiteneffekte in der nur wenig bekannten `readObject`-Methode *[1]* ein zweites Mal implementieren.

Besonders tückisch ist dabei, dass `Serializable` ein Marker-Interface ist und somit vererbt wird. Eine Klasse kann somit den Mechanismus erben, ohne dass dem Autor dies zwangsläufig bewusst ist.

Leider ist das genannte Beispiel nur teilweise konstruiert – gerade bei Objekten, die länger serialisiert bleiben, weil sie beispielsweise in Datenbanken abgelegt wurden, kann es dazu kommen, dass die Applikation sich weiterentwickelt und neue Invarianten auf diesen Objekten gesetzt werden. Wenn der Applikationsentwickler dieses Verhalten nicht explizit im Auge behält, gibt es ein reales Risiko, dass solch ein Problem über Wochen und Monate nicht bemerkt wird. Die Folgen sind dann bestenfalls mysteriöse Fehler.

Generell ist Java-Serialisierung ein Mechanismus, der so weit wie möglich vermieden werden sollte, auch wenn er auf den ersten Blick sehr bequem wirkt. Neben den genannten Problemen im Objekt-Lebenszyklus ist es auch möglich, in Sicherheitsprobleme zu laufen (siehe beispielsweise *[2]*). Daher empfiehlt es sich, ein beschränkteres Datenaustausch-Format zu nutzen.

Selbst-wiederbelebende Objekte

Der Lebenszyklus eines Java-Objekts erscheint auf den ersten Blick immer noch relativ einfach. Es wird erzeugt (auf welche Art auch immer) und hat eine oder mehrere Referenzen, ist also erreichbar. Irgendwann wird es unerreichbar und der Garbage Collector gibt den Speicher, den das Objekt beansprucht hat, wieder frei.

Werfen wir einen Blick auf *Listing 5*. Intuitiv erwarten wir, dass diese Schleife einmal durchlaufen wird. Auch hier trügt die Intuition uns. Zum Lebenszyklus eines Objektes gehört nämlich ein weiterer Punkt: `finalize`. Diese Methode bildet ein Gegenstück zum Destruktor eines C++-Objektes. Der Gedanke war, mit dieser Methode das Ziel zu erreichen, das erheblich später mit dem *try-with-resources*-Konstrukt umgesetzt wurde: externe Ressourcen, wie zum Beispiel Netzwerk-Sockets, sicher wieder freigeben, sobald sie nicht mehr benötigt werden.

Leider hat `finalize` einen Haken: Innerhalb der Methode kann beliebiger Code ausgeführt werden. Somit ist der Weg frei für den Code aus *Listing 6*. Was hier passiert, ist einfach: Das `Immortal`-Objekt hat keine Referenz mehr, sobald die Schleife einmal startet. Leider gehört ihm das `Resurrector`-Objekt, das finalisiert werden muss. Dabei wird `Immortal` wieder erreichbar. Noch schlimmer, es wird ein neuer `Resurrector` erzeugt, sodass das gleiche Szenario sich immer wieder wiederholen wird.

Dies ist aus mehreren Gründen problematisch. Zum einen ist es ein quasi nicht zu entdeckendes Speicherleck. Dadurch, dass immer wieder neue Objekte erzeugt werden, wird der ganze vorherige Objektgraph am Leben gehalten. Der Garbage Collector kann also die

```
static Immortal immortal = new Immortal();

public static void main(String[] args) throws Exception {
    do {
        immortal = null;
        System.gc();
        Thread.sleep(500);
    } while (immortal != null);
}
```

Listing 5

```
public class Immortal {

    private class Resurrector {
        @Override
        protected void finalize() {
            Resurrection.immortal = Immortal.this;
            resurrector = new Resurrector();
        }
    }

    private Resurrector resurrector = new Resurrector();
}
```

Listing 6

```
class LazyExpensive {
    private Expensive i = null;

    public Expensive getExpensive() {
        if (i == null) {
            i = new Expensive();
        }
        return i;
    }

    @Override
    protected void finalize() throws Throwable {
        if (i != null) {
            i.release();
            i = null;
        }
        doExtraCleanup(this);
    }
}
```

Listing 7

Objekte nicht einfach löschen, sobald sie unerreichbar sind, sondern muss zurück in den Benutzercode springen, dort Aktionen ausführen und das Objekt vorläufig behalten. Gerade wenn es viele solcher Objekte gibt, kann dies zu zusätzlichen Pausen führen.

Dazu kommt, dass für jede Instanz die `finalize`-Methode nur einmal aufgerufen wird. In unserem Beispiel ist das egal, da jedes Mal neue, frische Instanzen generiert werden. Wenn aber diese Methode zum Management von Ressourcen verwendet wird, können Szenarien wie in *Listing 7* entstehen. Hier wird – sollte in `doExtraCleanup` noch einmal das Objekt benutzt werden – ein Aufruf von `release` verloren gehen.

Generell sollte `finalize` nicht mehr verwendet werden und die Methode ist zum Glück seit einiger Zeit auch *deprecated*. Es gibt auch eigentlich keinen Grund mehr, sie zu verwenden. Das eingangs erwähnte `try-with-resources`-Pattern bildet quasi alle übliche Anwendungsfälle ab. Wenn das Abwickeln des Objekts aber absolut kritisch ist und ein „vergessenes" Freigeben inakzeptabel, können Lösungen mittels `ReferenceQueue` umgesetzt werden. Diese wenig bekannte Klasse kann verwendet werden, um explizit vom Garbage Collector darüber informiert zu werden, wenn ein Objekt freigegeben wurde. Wichtig hierbei ist die Vergangenheitsform – das Objekt wurde bereits freigegeben und kann nicht wieder erreichbar gemacht werden.

Es gibt natürlich noch einiges mehr, das in Java zu entdecken ist: So ist `final` an Feldern keine Garantie dafür, dass sich der Inhalt des Feldes selbst nach der Initialisierung nicht ändert, dass `Integer`-Instanzen manchmal doch mit == verglichen werden können und dass Sichtbarkeit teilweise in kurioser Art und Weise entweder keine Rolle spielt oder doch eingehalten werden muss.

Lauffähige Beispiele für die hier gezeigten Marotten und ein wenig mehr sind auf GitHub *[3]* verfügbar.

Quellen

[1] *https://howtodoinjava.com/java/serialization/custom-serialization-readobject-writeobject/*

[2] *https://www.synopsys.com/content/dam/synopsys/sig-assets/whitepapers/exploiting-the-java-deserialization-vulnerability.pdf*

[3] *https://github.com/Norwae/oddities*

Elisabeth Schulz

Codecentric AG

elisabeth.schulz@codecentric.de

Elisabeth Schulz ist eine Allrounderin mit inzwischen 15 Jahren Erfahrung in der Softwareentwicklung. Sie legt Wert auf tiefes Wissen und Einblick in Details. Neben Java gehören auch die jüngeren JVM-Sprachen Scala und Kotlin zu ihrem Schwerpunkt. Neben JVM-Sprachen interessiert sie sich für IT-Sicherheit sowie funktionale und parallele Programmierung.

React als Template-Engine für Spring Boot

David Tanzer

React erlaubt es, eine Single-Page-Application auch serverseitig zu rendern. Wie funktioniert das, wenn man den Server in Kotlin – mit Spring Boot – schreiben will? In diesem Artikel zeige ich, was man tun muss, um eine React-Anwendung in einem Spring Boot Controller, der in einer GraalVM läuft, zu rendern. Dabei verwende ich direkt das Webpack-optimierte Paket der Anwendung. Somit läuft genau derselbe UI-Code am Server und im Browser der Benutzer.

Im ersten Teil dieses Artikels möchte ich zeigen, was notwendig ist, um die in TypeScript geschriebene Single-Page-Application überhaupt auf dem Server laufen zu lassen. Ich werde sie in einem Spring-Boot-Controller ausführen, um die HTML-Seiten der Anwendung am Server zu generieren. Sobald diese Seite im Browser geladen ist, wird React sie auch clientseitig übernehmen und die Komponenten, die bereits am Server gerendert wurden, als initialen Zustand der Single-Page-Application verwenden. Dafür werde ich den optimierten „Production Build" der Anwendung verwenden.

Im zweiten Teil, der in der nächsten Ausgabe der Java aktuell erscheinen wird, widmen wir uns Routing, Navigation, Styling und dem Laden von Daten.

Die leere React-Anwendung

React ist eine Bibliothek, die es uns erlaubt, komponentenbasierte „Single Page Applications" (SPAs) zu entwickeln – also Anwendungen, die mit JavaScript, HTML und CSS entwickelt wurden und komplett im Browser des Anwenders laufen. Die Darstellung der Anwendung wird immer komplett im Browser gerendert; mit dem Server werden lediglich Daten – meistens in Form von JSON – ausgetauscht.

Die React-Anwendung, die wir als Beispiel verwenden, habe ich mit `create-react-app` erstellt *(siehe Listing 1)*. Dieses Programm erzeugt eine leere Anwendung, installiert alle Abhängigkeiten und konfiguriert verschiedene Skripte: Einerseits kann mit `npm start` ein Entwicklungsserver gestartet werden, der „Hot Reload" ermöglicht (jede Änderung am Quellcode wird sofort im Browser sichtbar). Andererseits kann mit `npm run build` ein mit Webpack optimiertes Paket für den Produktionsbetrieb erstellt werden.

Der gesamte Quellcode dieser Beispielanwendung ist auf GitHub *[1]* zu finden.

Als Frontend für Spring Boot

Diese React-Anwendung möchte ich nun als Frontend für eine Spring-Boot-Anwendung verwenden. Dabei möchte ich auch die Möglichkeit nutzen, die React-Komponenten schon am Server zu rendern und somit eine „Isomorphic Webapp" zu betreiben.

Die Benutzer sollen schon beim Laden der Anwendung den korrekten HTML-Code ausgeliefert bekommen und React rendert im Browser dann nur noch die Ergebnisse aller weiteren Benutzerinteraktionen. Somit würde die Anwendung beim ersten Aufruf schneller die richtige Seite rendern, da nicht vorher der gesamte JavaScript-Code geladen und ausgeführt werden muss. Für Search-Engine-Optimierung sollte dieses Vorgehen ebenfalls vorteilhaft sein – auch wenn Suchmaschinenbots heute JavaScript ausführen können.

Dabei ist es mir wichtig, dass das Webpack-optimierte Paket der React-Anwendung direkt, ohne nachträgliche Änderung, am Server verwendet werden kann. Außerdem möchte ich die React-Anwendung auch weiterhin im Development-Modus verwenden können. Dafür muss die Anwendung auch mit dem Server kommunizieren können, wenn sie nicht über genau diesen Server geladen wurde: Im Development-Modus startet ein eigener Entwicklungsserver der Anwendung.

Wenn die Anwendung am Server gerendert wird, sollen allerdings keine REST-Aufrufe zu ebendiesem Server stattfinden. In diesem Fall sollen aus dem JavaScript-Code heraus direkt die richtigen Kotlin-Methoden aufgerufen werden.

Die React-App am Server

Die generierte React-Anwendung enthält vorerst nur eine Komponente *(siehe Listing 2)*. Diese Komponente bekommt keine Parameter von außen („props") und hat keinen Zustand – sie erzeugt nur eine immergleiche Überschrift. Baut man nun die Client-Anwendung mit `npm run build`, wird ein optimiertes Paket im Unterordner `build` abgelegt. Um diese Ausgabedateien in der Spring-Boot-Anwendung

```
$ npx create-react-app client --template typescript
```

Listing 1: Erstellen einer leeren React-Anwendung

```
function App() {
  return (
    <h1>
      I'm a React Component
    </h1>
  );
}
```

Listing 2: Die Hauptkomponente der generierten Anwendung

```
#!/bin/sh

rm -rf ../server/public
rm -rf ../server/src/main/resources/reactapp
mkdir -p ../server/src/main/resources/reactapp/js

npm run build
mv build ../server/public
mv ../server/public/index.html \
    ../server/src/main/resources/reactapp
cp ../server/public/static/js/2.*.chunk.js \
    ../server/src/main/resources/reactapp/js/2.chunk.js
cp ../server/public/static/js/main.*.chunk.js \
    ../server/src/main/resources/reactapp/js/main.chunk.js
cp ../server/public/static/js/runtime-main.*.js \
    ../server/src/main/resources/reactapp/js/runtime-main.js
```

Listing 3: Verschieben der generierten Webanwendung

```
@Controller
class HtmlController {
  val indexHtml by lazy {
    HtmlController::class.java
      .getResource("/reactapp/index.html").readText()
  }
  @GetMapping("/")
  @ResponseBody
  fun blog(): String {
    return indexHtml
  }
}
```

Listing 4: Ausliefern der Datei „index.html"

verwenden zu können, werden sie mithilfe eines Skripts „`build-client.sh`" verschoben *(siehe Listing 3)*.

Die statischen Dateien der Webanwendung werden von Spring Boot aus dem Ordner `public` geladen. Allerdings werden die Dateien `2.chunk.js`, `main.chunk.js` und `runtime-main.js` auch vom Kotlin-Code der Anwendung gelesen, daher kopiert das Skript diese Dateien nach `src/main/resources`. Und `index.html` soll sogar ausschließlich von einem Spring Boot Controller bereitgestellt werden (und nie als statische Datei), daher wird diese Datei nach `src/main/resources` verschoben.

Ein Controller kann nun die Datei `index.html` vom Classpath lesen und ausliefern *(siehe Listing 4)*. Aber das ist nur die halbe Miete: Diese Datei ist (fast) leer. Sie muss erst mit Inhalt befüllt werden, und dieser Inhalt muss am Server von React gerendert werden.

GraalVM und Initialisierung des JavaScript-Codes

Um den generierten JavaScript-Code am Server auszuführen, verwende ich die `ScriptEngine` von GraalVM, da die ältere Nashorn-Engine mittlerweile ein Auslaufmodell ist. Dafür lasse ich die Anwendung auch in der GraalVM – und nicht in der „normalen" JVM – laufen.

Die `ScriptEngine` wird im Controller einmalig initialisiert und führt nach der Initialisierung auch gleich die drei statischen JavaScript-Dateien aus *(siehe Listing 5)*. Außerdem müssen noch die Objekte `window` und `navigator` hinzugefügt werden, damit diese vom JavaScript-Code gefunden werden. Der Wert `window.isServer` wird am Ende auf `true` gesetzt – damit wird später im JavaScript-Code entschieden, ob serverseitiges Rendern verwendet werden soll oder ob sich die React-Anwendung normal verhält.

Der JavaScript-Code `initJs` wird in *Listing 5* direkt aus der Datei `index.html` gelesen und auch ausgeführt. Es handelt sich hierbei um Code, den Webpack direkt in `index.html` generiert und der auch

```
@Controller
class HtmlController {
  val runtimeMainJs by lazy {
  HtmlController::class.java
    .getResource("/reactapp/js/runtime-main.js").readText()
  }
  val initJs by lazy(::readInitJs)
  val engine by lazy(::initializeEngine)

  private fun readInitJs(): String {
    val startIndex = indexHtml.indexOf(
      "<script>")+"<script>".length
    val endIndex = indexHtml.indexOf("</script>", startIndex)

    return indexHtml.substring(startIndex, endIndex)
  }

  private fun initializeEngine(): GraalJSScriptEngine {
    val engine = GraalJSScriptEngine.create(null,
        Context.newBuilder("js")
          .allowHostAccess(HostAccess.ALL)
          .allowHostClassLookup({ s -> true }))

    engine.eval("window = {location:{hostname:'localhost'}}")
    engine.eval("navigator = {}")
    engine.eval(runtimeMainJs)
    engine.eval(mainJs)
    engine.eval(secondJs)
    engine.eval(initJs)
    engine.eval("window.isServer = true")

    return engine
  }
}
```

Listing 5: Initialisierung der ScriptEngine

```
ReactDOM.render(<App />, document.getElementById('root'))
```

Listing 6: index.tsx – Default-Code zum Rendern der Anwendung

```
const anyWindow: any = window
anyWindow.renderApp = () => {
  ReactDOM.hydrate(<App />, document.getElementById('root'))
}
anyWindow.renderAppOnServer = () => {
  return ReactDOMServer.renderToString(<App />)
}
anyWindow.isServer = false
```

Listing 7: index.tsx – unterschiedliche Initialisierung im Browser und am Server

im Browser nach dem Laden der anderen JavaScript-Dateien ausgeführt wird. Der Code kann erkannt und geladen werden, da es der einzige `script`-Block ohne weitere Parameter in dieser Datei ist.

Rendern der React-Anwendung bei jedem Aufruf

Jetzt muss noch bei jedem Laden der Seite – und somit bei jeder Ausführung der Controller-Methode `blog()` aus *Listing 4* – die React-Anwendung serverseitig gerendert werden.

Der Code zum Rendern der React-Anwendung befindet sich allerdings im Moment direkt in der Datei `client/src/index.tsx` *(siehe Listing 6)*. Genau das ist nun problematisch: Dieser Code ist für das Rendern der Anwendung im Browser zuständig, jetzt würde er allerdings sofort beim Initialisieren der `ScriptEngine` am Server *(siehe Listing 5)* ausgeführt.

Derzeit gibt es hier keine Möglichkeit, zwischen serverseitigem Rendern und Rendern im Browser zu unterscheiden. Anders gesagt, die Codezeile `engine.eval("window.isServer = true")` aus *Listing 5* hat noch keine Auswirkung.

Dieser Code muss geändert werden, sodass…

1. er nicht sofort ausgeführt wird, sondern am Server kontrolliert, zu einem bestimmten Zeitpunkt, ausgeführt werden kann,
2. eine Unterscheidung zwischen Server und Browser anhand von `window.isServer` möglich ist,
3. `ReactDOM.hydrate` anstatt von `ReactDOM.render` verwendet wird, damit gegebenenfalls serverseitig gerenderte Komponenten von React im Browser wiederverwendet werden.

```
<script defer type="module">
    if(window.isServer) {
        window.renderAppOnServer()
    } else {
        window.renderApp()
    }
</script>
```

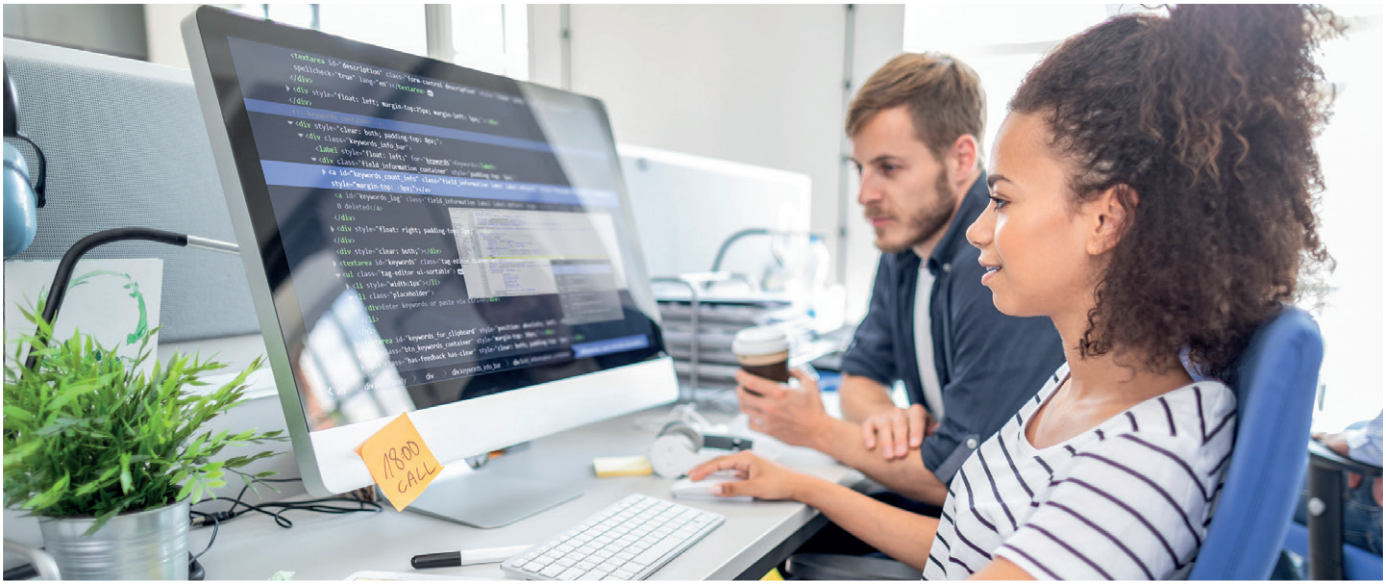
Listing 8: Auswahl der Render-Funktion in index.html

In *Listing 7* wurde `client/src/index.tsx` nun so verändert, dass der Code zum Rendern der Anwendung nicht mehr sofort ausgeführt wird. Außerdem gibt es eine weitere Funktion zum serverseitigen Rendern. Aber `window.isServer` wird standardmäßig auf `false` gesetzt, somit kommt per Default der Code zum Rendern im Browser zum Einsatz.

Während der Initialisierung der `GraalJSScriptEngine` am Server *(siehe Listing 5)* wird dieser Wert dann auf `true` gesetzt.

Jetzt muss noch die jeweils richtige der beiden Funktionen aufgerufen werden. Dazu habe ich in der Datei `client/public/index.html` einen `script`-Block gleich nach dem `root-div` eingefügt *(siehe Listing 8)*.

Dieser `script`-Block kann im Kotlin-Code wieder ausgelesen werden: Es ist der einzige `script`-Block mit `defer` und `type="module"`. Danach kann dieses Skript bei jedem Aufruf der Controller-Methode `blog()` ausgeführt werden *(siehe Listing 9)*. Das „Render"-Skript wird hier also einmalig aus dem HTML-Code von `index.html` gelesen.



```
val renderJs by lazy(::readRenderJs)

//...

private fun readRenderJs(): String {
  val start = "<script defer=\"defer\" type=\"module\">"
  val startIndex = indexHtml.indexOf(start)+ start.length
  val endIndex = indexHtml.indexOf("</script>", startIndex)

  return indexHtml.substring(startIndex, endIndex)
}

@GetMapping("/")
@ResponseBody
fun blog(): String {
  val html = engine.eval(renderJs)
  return indexHtml.replace("<div id=\"root\"></div>",
    "<div id=\"root\">$html</div>")
}
```

Listing 9: Ausführen des Render-Skripts am Server

Bei jedem Aufruf von `blog()` wird dieses Skript ausgeführt und das Ergebnis in den `div` mit der ID `root` eingefügt.

Somit wird die React-Anwendung bereits am Server gerendert und es wird der richtige HTML-Code an den Browser gesendet. Im Browser setzt React durch `ReactDOM.hydrate` genau auf dem Ergebnis des Servers auf.

Fazit und Ausblick

In diesem Artikel habe ich Ihnen gezeigt, wie Sie eine leere React-Anwendung am Server, in einem Spring Boot Controller, rendern können.

Dafür wird direkt die durch Webpack optimierte Anwendung (erzeugt mit dem Befehl `npm run build`) am Server verwendet. Diesen Befehl führe ich in einem eigenen Skript, `build-client.sh` aus, das die generierten Dateien dann auch in die richtigen Verzeichnisse am Server verschiebt.

Am Server werden die generierten JavaScript-Dateien dann gelesen und in einer GraalVM `GraalJSScriptEngine` ausgeführt. Bei jedem Request wird die React-Anwendung gerendert und das Ergebnis an der richtigen Stelle in `index.html` eingefügt.

Dazu musste ich den Code zum Rendern der Anwendung etwas verändern: Die React-Anwendung wird nicht mehr direkt beim Laden der JavaScript-Dateien gerendert, sondern es gibt einen `script`-Block in `index.html`, der entweder das serverseitige Rendern oder das Rendern im Browser ausführt. Dieser `script`-Block kann vom Spring Boot Controller gelesen und bei jedem Request in der `GraalJSScriptEngine` ausgeführt werden.

In einem zweiten Teil in der kommenden Ausgabe werde ich Ihnen dann zeigen, wie Sie im React-Code Daten verwenden können, die vom Server geladen werden. Diese Daten werden im Browser über REST geladen – wird die Anwendung jedoch am Server ausgeführt, wird lediglich eine Methode innerhalb derselben JVM aufgerufen.

Außerdem soll die Navigation innerhalb der Anwendung clientseitig genauso funktionieren wie am Server und es soll ein einheitliches Styling der Komponenten möglich sein.

Zusätzlich zu dem Webpack-optimierten Paket, das in der Spring-Boot-Anwendung verwendet wird, ist es auch weiterhin möglich, die Anwendung mit `npm start` im Development-Modus zu starten. Auch das soll weiterhin möglich bleiben, wenn ich im Teil 2 Navigation, Styling und das Nachladen von Daten implementiere.

Referenzen

[1] *https://github.com/dtanzer/react-graalvm-springboot*

David Tanzer

*business@davidtanzer.net

David Tanzer hilft seinen Kunden als Trainer, Berater und Coach, besser im „Agile Engineering" zu werden. Er unterstützt Entwicklerteams dabei, Entwicklungspraktiken wie Evolutionäre Architektur, Test-Driven Development, Agile Acceptance Testing, Pair Programming, ... einzuführen und zu perfektionieren – und so in der Entwicklung effizienter und effektiver zu werden. Außerdem hilft er bei der Einführung von neuen Technologien durch Schulungen, Coaching und Mitarbeit im Projekt.

JUG SAXONY DAY

25.09.

JUG SAXONY DAY 2020

Online-Konferenz + Meet&Watch Events vor Ort bei Sponsoren

PROGRAMM
2 Keynotes und 10 Sessions
jug-saxony-day.org

TICKETBUCHUNG
Tickets und Freitickets für Studierende
unter backoffice.jugsaxony.org

Eine Veranstaltung des JUG Saxony e. V.

Kontakt
team@jugsaxony.org
jugsaxony.org

Folgt uns auf

Grüne Inseln im Schlamm – Mit Side-by-Side Refactoring allzeit lieferbereit, Teil 2

Georg Berky, Valtech Mobility

Dieser Artikel zeigt anhand von Emily Baches „Gilded Rose"-Kata eine fortgeschrittene Art des Refactoring von Legacy Code, mit der auch unter schwierigen Bedingungen neuer Code testgetrieben in komplizierte Legacy-Anwendungen integriert werden kann. Wir bleiben wie im vorigen Teil 1 dabei allzeit lieferbereit.

Legacy Code ist eine große Herausforderung für Entwickler. Das ursprüngliche Team, das ihn entwickelt hat, ist nicht mehr da. Der Code hat nur schlechte oder keine Dokumentation und schon gar keine Tests. Ändert man an einer Stelle etwas im Code, tritt in einer komplett unerwarteten Klasse ein seltsamer Fehler auf. Man spricht von Brownfield oder Legacy Code – als würde man tief im Schlamm eines Feldes waten, wo jeder Schritt viel zu viel Kraft kostet. Es braucht Zeit, Nerven und Ressourcen, solchen Code wieder in einen wartbaren Zustand zu bringen.

Im vorigen Teil habe ich das Refactoring „Wrap Class" gezeigt, mit dem man gut anämische Datenklassen, die man nicht verändern darf, verbessern und besser handhabbar machen kann. Auch dieses Mal habe ich eine weitere Technik mitgebracht, die die IDE nicht automatisch ausführen kann, die aber dafür an schwierigen Stellen oft umso wirkungsvoller ist.

Die Techniken sind so einsetzbar, dass man sie kleinschrittig umsetzen kann und der Code dabei ständig lieferbar bleibt. Ich benutze als Beispiel wieder die „Gilded Rose Legacy Code Kata" von Emily Bache aus dem letzten Teil *[1]*.

Wir fangen wieder am Ausgangspunkt der Kata an. Lediglich die Tests habe ich aus dem letzten Kapitel übernommen. Der Code dazu findet sich im Branch „sprouting" meines Git Repository. Im Branch „master" findet sich der Ausgangszustand nur mit den Tests *[2]*.

Sprouting

Sprouting kann man verwenden, wenn man in wenig Zeit ein neues Feature hinzufügen muss, ohne dabei auf testgetriebene Entwicklung zu verzichten. Sprouts kommen in mehreren Formen, „Sprout Method" und „Sprout Class", vor. Beide Varianten werden an einem Seam in den vorhandenen Code eingesetzt.

Seams

Das Seam-Modell kommt aus Michael Feathers Buch „Working Effectively with Legacy Code" (WELC): „A place where you can alter behavior (...) without editing in that place" (WELC, p.31, *[3]*).

An Seams kann man das Verhalten des vorhandenen Codes verändern, ohne an diesen Stellen editieren zu müssen. Je nach Programmiersprache und deren Sprachmitteln gibt es mehrere Typen von Seam. Da unser Beispiel in Java gehalten ist, zähle ich hier ein paar Seams auf, die man in Java gut benutzen kann. Jeder Seam hat einen Enabling Point, an dem entschieden wird, welches Verhalten aktiviert wird.

Override Seam

Durch Überschreiben der Methode einer Klasse in einer Unterklasse kann man Verhalten der abgeleiteten Klasse ersetzen. Enabling Point ist die Instanziierung der Klasse und ihre Zuweisung zu einer Variablen.

Ausgangssituation sei die Klasse `Service`. Stellen wir uns vor, sie sei unheimlich unhandlich, was ihre Geschäftslogik angeht, aber das Laden aus der Datenbank konnten wir wenigstens in eine eigene Methode extrahieren *(siehe Listing 1)*. Wir wollen die Daten jetzt aus einem anderen Datenbanksystem laden. Dafür machen wir die Methode zunächst abstrakt und erstellen dann zwei Unterklassen, eine mit der Legacy-Datenbank und eine mit der neuen Datenbanktechnologie *(siehe Listing 2)*.

Man könnte in Versuchung kommen, die Klasse direkt abzuleiten und nur die eine Methode zu überschreiben. Das ist jedoch nicht ratsam. Jeder, der schon einmal Klassenhierarchien mit mehr als zwei Vererbungsstufen bearbeiten musste, wird bestätigen, dass es besser ist, direkt zu sehen, wo die Unterklasse verwendet wird.

GeePaw Hill beschreibt die Gründe genauer in „Avoid Implementation Inheritance" *[4]* oder in seinem Tweet dazu *[5]*.

Methodenparameter Seam

Durch Verwendung eines anderen Methodenparameters kann neues Verhalten in die benutzende Methode injiziert werden, wenn auf dem Parameter Methoden aufgerufen werden. Enabling Point ist der Methodenaufruf.

Ausgangssituation ist eine Klasse ähnlich wie oben, nur lädt sie die Datenbankwerte, indem sie eine Methode auf dem Parameter `loader` aufruft *(siehe Listing 3)*. Hier erhalten wir einen Seam, indem wir ein Interface aus dem Loader extrahieren und eine neue Implementierung schreiben, die mit der neuen Datenbank arbeitet *(siehe Listing 4)*.

```
class Service {
    public Result largeUseCase(UseCaseParameters input) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    loadFromDatabase(Keys keys) {
        //loads from legacy database
    }
}
```

Listing 1: Override Seam – davor

```
abstract class Service {
    public Result largeUseCase(UseCaseParameters input) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    abstract loadFromDatabase(Keys keys);
}

class ServiceWithLegacyDatabase extends Service {
    @Override
    loadFromDatabase(Keys keys) {
        //loads from legacy database
    }
}

class ServiceWithNewDatabase extends Service {
    @Override
    loadFromDatabase(Keys keys) {
        //loads from new database
    }
}
```

Listing 2: Override Seam – danach

```
class DatabaseLoader { ... }

class Service {
    public Result largeUseCase(UseCaseParameters input, DatabaseLoader loader) {
        var values = loader.loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }
}
```

Listing 3: Methodenparameter Seam – davor

```
interface DatabaseLoader {
    Values loadFromDatabase(Keys keys);
}
class LegacyDatabaseLoader implements DatabaseLoader {
    public Values loadFromDatabase(Keys keys) {
        //legacy database access
    }
}
class ShinyNewDatabaseLoader implements DatabaseLoader {
    public Values loadFromDatabase(Keys keys) {
        //new database access
    }
}
```

Listing 4: Methodenparameter Seam – danach

Konstruktorparameter Seam

Analog zu den Methodenparametern können wir Kollaborateure auch im Konstruktor übergeben. Auch hier können wir ein Interface extrahieren und die neue Funktionalität durch eine weitere Implementierung des Interface in die Anwendung bringen. Enabling Point ist der Konstruktor-Aufruf.

Make non-static and override

Eine statische Methode kann nichtstatisch gemacht und überschrieben werden. Enabling Point ist die Instanziierung der Klasse *(siehe Listing 5)*. Auch hier verzichten wir darauf, die Methode direkt zu überschreiben, sondern behalten eine abstrakte Oberklasse und zwei konkrete Unterklassen *(siehe Listing 6)*.

Unklarheiten in den Anforderungen

Unser Auftrag für die „Gilded Rose" Kata lautete, dass wir verzauberte „conjured" Gegenstände unterstützen sollen, die doppelt so schnell wie normale Gegenstände ihre Qualität verlieren sollen. In guter Tradition vieler Online-Rollenspiele interpretieren wir das auf diese Weise:

Jeder Gegenstand kann verzaubert werden. Sein Name bekommt dann „Conjured" als Präfix vorangesetzt. Trotzdem bleiben ein paar Unklarheiten: Aus den Anforderungen geht etwa nicht hervor, ob man einen „Aged Brie" auch verzaubern kann und ein „Conjured Aged Brie" dann doppelt so schnell an Qualität gewinnt oder die Qualität stattdessen konstant bleibt. Gibt es „Conjured Backstage Passes"? „Conjured Sulfuras" sollte keine Auswirkungen auf den Qualitätsverlust haben. Verlieren verzauberte reguläre Gegenstände viermal so schnell Qualität, wenn ihr Ablaufdatum erreicht ist?

Fragen über Fragen, die wir am besten Allison stellen, wenn wir sie in der Mittagspause sehen. Bis dahin fangen wir mit verzauberten regulären Gegenständen an, die keine weiteren Eigenschaften haben.

```
class Service {
    public Result largeUseCase(UseCaseParameters input, DatabaseLoader loader) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    static Values loadFromDatabase(Keys keys) {
        //load from legacy database
    }
}
```

Listing 5: Make non-static and override – davor

```
abstract class Service {
    public Result largeUseCase(UseCaseParameters input, DatabaseLoader loader) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    abstract Values loadFromDatabase(Keys keys);
}

class ServiceWithLegacyDatabase extends Service { ... }
class ServiceWithNewDatabase extends Service { ... }
```

Listing 6: Make non-static and override – danach

Sprout Method

Wir wollen den vorhandenen Code nicht noch schlimmer machen. Deswegen verwenden wir eine Sprout Method, um die neue Funktionalität vom vorhandenen Chaos zu trennen. Ein Sprout ist ein Code-Sprössling, den wir testgetrieben neben den vorhandenen Code „pflanzen", um das Risiko zu vermeiden, dort etwas kaputt zu machen.

Wir beginnen mit einigen vorbereitenden Schritten und führen nach jedem Refactoring unsere Tests aus dem vorigen Teil aus, damit wir sicher sind, nichts kaputt gemacht zu haben.

Vorbereitungen: Den Seam finden und erzeugen

Die Funktionalität der Methode `updateQuality()` besteht darin, über alle vorhandenen Instanzen von `Item` zu iterieren und ihre Attribute zu modifizieren. Um die alte Funktionalität zu isolieren, extrahieren wir zunächst das aktuelle `Item` in eine Variable: `Item itemToUpdate = items[i];`.

Danach können wir das aktuelle Verhalten in eine Methode `private void updateOtherItem(Item item)` extrahieren. Unsere Methode `updateQuality()` sieht jetzt so aus, wie in *Listing 7* gezeigt.

Jetzt könnten wir mit dem Sprouting anfangen. Um die Signatur der Sprout Method zu definieren, schreiben wir sie in den vorhandenen Code, als ob wir uns wünschen können, dass sie da wäre. Als Enabling Point fügen wir noch eine Unterscheidung in verzauberte und andere Gegenstände ein *(siehe Listing 8)*.

Unsere IDE kann uns die gewünschte Methode `void updateConjured(Item item)` jetzt anlegen. Damit wollen wir es aber mit den Modifikationen des Produktionscodes belassen. Wir haben zwar aus dem vorigen Teil noch ein Sicherheitsnetz für die bestehende Funktionalität, für verzauberte Gegenstände haben wir allerdings noch keine Tests. Generell ist es auch bei Legacy Code ein guter Ansatz, keinen Produktionscode ohne Tests dafür zu schreiben. Manchmal müssen wir jedoch – wie beim Sprouting – ein paar Modifikationen vornehmen, um überhaupt die Möglichkeit zu haben, neuen Code testgetrieben einzuführen.

```
public void updateQuality() {
    for (int i = 0; i < items.length; i++) {
        Item itemToUpdate = items[i];
        updateOtherItem(itemToUpdate);
    }
}
```

Listing 7: Vorbereitung zum Sprouting

```
public void updateQuality() {
    for (int i = 0; i < items.length; i++) {
        Item itemToUpdate = items[i];
        if (itemToUpdate.name.startsWith("Conjured ")) {
            updateConjured(itemToUpdate);
        } else {
            //here be dragons
            updateOtherItem(itemToUpdate);
        }
    }
}
```

Listing 8: Frisch gepflanzter Sprössling: Sprout Method

```
void updateConjured(Item item) {
    //TODO: implementieren
}
```

Listing 9: Noch leere Sprout Method

Um den Code lieferbar zu halten, kommentieren wir den neuen Methodenaufruf zunächst aus. Wenn wir jetzt liefern müssen, können wir ohne Einschränkungen auf die bestehende Funktionalität zurückgreifen.

Das Pflänzchen einsetzen

Jetzt haben wir ein Methodenskelett, das wir testgetrieben mit Leben füllen können *(siehe Listing 9)*. Beginnen wir mit den Tests für einen regulären verzauberten Gegenstand. Wir nehmen, wie oben angesprochen, an, dass er pro Tag zwei Qualitätspunkte verliert,

```
@ParameterizedTest
@ValueSource(ints = {10, 9, 8})
public void conjuredItem_sellInDateNotPassed_degradesQualityByTwo(int initialQuality) {
    Item item = new Item("Conjured Regular Item", notPastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isEqualTo(initialQuality - 2);
}
```

Listing 10: Testgetriebener Sprössling – erste Tests

```
void updateConjured(Item item) {
    item.quality -= 2;
}
```

Listing 11: Produktionscode zu den ersten Tests (aus Listing 10)

```
@ParameterizedTest
@ValueSource(ints = {10, 9, 8})
public void conjuredItem_sellInDatePassed_degradesQualityByFour(int initialQuality) {
    Item item = new Item("Conjured Regular Item", pastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isEqualTo(initialQuality - 4);
}
```

Listing 12: Zweites Szenario – Tests

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(item.sellIn < 0) {
        degeneration *= 2;
    }

    item.quality -= degeneration;
}
```

Listing 13: Zweites Szenario – Produktionscode

```
private boolean pastSellInDate(Item item) {
    return item.sellIn < 0;
}
```

Listing 14: Extrahierte Methode: Ablaufdatum erreicht

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(pastSellInDate(item)) {
        degeneration *= 2;
    }

    item.quality -= degeneration;
}
```

Listing 15: Sprout Method nach Extraktion der Methode

wenn er sein Ablaufdatum noch nicht erreicht hat und danach doppelt so viele, also vier pro Tag. Wir testen zunächst direkt die Sprout Method. Im Folgenden zeige ich der gebotenen Kürze wegen direkt parametrisierte Tests. Normalerweise extrahiere ich diese aus den zuerst von Hand geschriebenen Tests für die aktuelle Anforderung *(siehe Listing 10)*.

Wir starten mit einem Test, der nur den Anfangswert `10` für die Qualität betrachtet, und fügen dann schrittweise Beispiele für den aktuellen Testfall hinzu. Zuerst geben wir eine Konstante zurück und kommen zum Schluss durch Refactoring auf eine generalisierte Lösung (Triangulation, *siehe Listing 11*).

Zeit für das nächste Szenario, dessen Tests wir in *Listing 12* sehen. Wir brauchen jetzt eine Prüfung, ob der Gegenstand sein Haltbarkeitsdatum schon erreicht hat. In diesem Fall verdoppeln wir den Qualitätsverlust *(siehe Listing 13)*.

Jetzt sehen wir, dass der Prüfung eigentlich ein fachlicher Name fehlt und extrahieren die Methode *(siehe Listing 14)*. Unsere IDE hilft uns, indem sie bemerkt, dass die gleiche Prüfung auch im alten Code verwendet wird, und ersetzt sie auch dort durch den Methodenaufruf. Unsere Sprout Method sieht jetzt so aus, wie in *Listing 15* gezeigt.

Jetzt müssen wir sicherstellen, dass die Qualität nicht negativ wird, sowohl bei Gegenständen vor dem Ablaufdatum als auch danach.

Die Tests dafür sehen wir in *Listing 16*. Das treibt unseren Produktionscode in den Zustand in *Listing 17*.

Zum Schluss müssen wir noch implementieren, dass sich bei verzauberten Gegenständen das Ablaufdatum verringert – und zwar auch bis hin zu negativen Werten. Um dies zu erreichen, schreiben wir die in *Listing 18* gezeigten Tests. Wir brauchen lediglich eine zusätzliche letzte Zeile im Produktionscode, um diese Tests zu bestehen. Die gesamte Methode ist in *Listing 19* zu sehen.

```
@ParameterizedTest
@ValueSource(ints = {1, 0})
public void conjuredItem_sellInDateNotPassed_doesNotDegradeBelowZero(int initialQuality) {
    Item item = new Item("Conjured Regular Item", notPastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isZero();
}

@ParameterizedTest
@ValueSource(ints = {3, 2, 1, 0})
public void conjuredItem_pastSellInDate_doesNotDegradeBelowZero(int initialQuality) {
    Item item = new Item("Conjured Regular Item", pastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isZero();
}
```

Listing 16: Szenario „Qualität wird nicht negativ" – Tests

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(pastSellInDate(item)) {
        degeneration *= 2;
    }

    item.quality = Math.max(0, item.quality - degeneration);
}
```

Listing 17: Szenario „Qualität wird nicht negativ" – Produktionscode

```
@ParameterizedTest
@ValueSource(ints = {2, 1, 0, -1, -2})
public void conjuredItem_decreasesSellInDateByOne_perDay(int initialSellInDays) {
    Item item = new Item("Conjured Regular Item", initialSellInDays, anyQuality());

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.sellIn).isEqualTo(initialSellInDays - 1);
}
```

Listing 18: Szenario „Ablaufdatum verringert sich" – Tests

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(pastSellInDate(item)) {
        degeneration *= 2;
    }

    item.quality = Math.max(0, item.quality - degeneration);

    item.sellIn--;
}
```

Listing 19: Szenario „Ablaufdatum verringert sich" – Produktionscode

Sprout Class

Das Vorgehen bei Sprout Class wäre ähnlich gewesen. Wir hätten denselben Seam und Enabling Point verwenden können, aber anstatt eine neue Methode aufzurufen, hätten wir eine Klasse instanziiert, die intern die Daten des Gegenstands manipuliert – ähnlich wie wir es im vorigen Kapitel mit Wrap Class gemacht haben.

Testing, Staging, Livegang

Die neue Funktionalität können wir liveschalten, indem wir den vorher noch auskommentierten Methodenaufruf wieder aktivieren. Das ist auch möglich, solange wir noch nicht alle Anforderungen an die verzauberten Gegenstände erhalten und umgesetzt haben, etwa jetzt, wo wir mit regulären verzauberten Gegenständen umgehen können. Gibt es ein Staging-Konzept, können wir den Aufruf etwa nur durchführen, wenn ein Feature Switch aktiviert wurde. Dadurch können je nach Projektsituation Tester schon anfangen, einen Teil der neuen Funktionalität zu prüfen, und wir erhalten für die nächste Iteration zusätzlichen Regressionsschutz durch bereits vorhandene Tests. Haben wir die Tests im Voraus als Akzeptanztest formuliert, sollte ein Teil davon jetzt grün sein, nämlich der für die regulären verzauberten Gegenstände.

Auf lange Sicht: Das Strangler Pattern

Über mehrere Iterationen verteilt und auf lange Sicht lassen sich mit Sprouts schrittweise geordnetere Strukturen in den Code bringen. Im vorigen Beispiel können wir weitere Sprout Methods für bestehende Gegenstände erstellen, sobald wir an deren Funktionalität etwas ändern müssen. Neue Gegenstände bekommen sofort ihre eigene Methode. Das bestehende Brownfield, das wir in die Methode `updateOtherItem()` verbannt haben, wird dadurch immer kleiner und verschwindet zum Schluss komplett.

Sobald wir Ähnlichkeiten unter den Sprouts entdecken, können wir anfangen, gemeinsame Funktionalität zu extrahieren. Wenn wir mit Sprout Classes arbeiten, könnte sich beispielsweise auch eine Klassenhierarchie bilden und die Anwendung arbeitet nur noch mit einem Interface. Das ist jedoch alles Spekulation, in der wir uns nicht verfangen sollten.

An dieser Stelle sei deswegen davor gewarnt, sich vorschnell ein neues Design auszudenken, das alle unsere Probleme lösen wird, und dann in einem einzigen großen Umbau die gesamte Applikation „ordentlich zu machen". Das Risiko, dass das Vorhaben fehlschlägt, ist viel größer als bei kleinen, schrittweisen Umbauten. Große Umbauten blockieren auch die Weiterentwicklung zu lange, sodass dringend nötige Features nicht rechtzeitig eingesetzt werden können. Kleine Schritte geben uns die Möglichkeit zu reflektieren, ob das langfristige Ziel noch zu den aktuellen Anforderungen passt.

Martin Fowler beschreibt das Strangler Pattern in seinem Artikel „Strangler Fig Application" *[6]*.

Fazit

Mit Sprouts lässt sich auch in sehr komplexe Anwendungen testgetrieben neue Funktionalität einbauen. Oft liegt die Kunst darin, den richtigen Ort zu finden, an dem sich die neue oder geänderte Funktionalität am besten einsetzen lässt. Hierbei hat uns das Seam Model geholfen. Durch diese Techniken bleiben wir auch bei längeren Umbauten allzeit lieferbereit.

Quellen

[1] Emily Bache (2011): Gilded Rose Kata. *https://github.com/emilybache/GildedRose-Refactoring-Kata*

[2] *https://bitbucket.org/georgberky/gilded-rose-kintsugi/*

[3] Michael C. Feathers (2004): Working Effectively with Legact Code. Prentice Hall, Upper Saddle River, NJ, United States

[4] GeePaw Hill (2018): Avoid Implementation Inheritance. *https://www.geepawhill.org/2018/03/03/avoid-implementation-inheritance-geepaw-goes-all-geek-y/*

[5] GeePawHill (2020): A common mid-sized refactoring: „Replace Implementation Inheritance" *https://twitter.com/GeePawHill/status/1258005102167838723*

[6] Martin Fowler (2004): Strangler Fig Application *https://martinfowler.com/bliki/StranglerFigApplication.html*

Georg Berky

Valtech Mobility

georg.berky@valtch-mobility.com

Georg entwickelt hauptberuflich Connected-Car-Dienste. Handwerk und Leidenschaft ist für ihn die Programmierung, meistens in JVM-Sprachen wie Java, Groovy oder Clojure. Zum Handwerk gehören für ihn auch Themen wie die Pflege von Legacy Code, Automatisierung von Builds und Deployments oder Agilität im Team. Seit einigen Jahren ist er Co-Organisator der Software-Craftsmanship-Communities im Ruhrgebiet und in Düsseldorf. Wenn er mal nicht programmiert, spielt er Trompete oder praktiziert Aikido. Er twittert als @georgberky.

Erfolgreich remote arbeiten

Sven Peters, MongoDB

Seit einigen Wochen arbeiten viele von uns im Home-Office. Die meisten haben sicherlich schon vorher ab und zu oder sogar regelmäßig an ein bis zwei Tagen in der Woche von zuhause aus gearbeitet. Remote-Arbeit bedeutet allerdings, dass man oft wochenlang physisch keine Kollegen trifft. Wie schafft man es, sich täglich neu zu motivieren? Wie kann man die soziale Distanz verkürzen? Welche Tools können helfen, die Remote-Arbeit zu verbessern? Ich arbeite seit neun Jahren mit weltweit verteilten Teams von meinem Home-Office in Kiel.

Warum remote?

Technologieunternehmen investieren viel in ihre Büroarbeitsatmosphäre: großzügige Arbeitsbereiche, kostenlose Getränke und Snacks, Tischfußball- und Yoga-Angebote, gemütliche Besprechungsnischen und vieles mehr. Die Mitarbeiter sollen sich im Büro wohlfühlen und gleichzeitig soll die Zusammenarbeit gefördert werden: neue Ideen beim Zubereiten eines Espresso besprechen, sich schnell eine Couch schnappen und über den Projektstand informieren oder einfach Kollegen aus einer anderen Abteilung bei einem Tischtennisspiel kennenlernen. Manche Unternehmen gehen sogar so weit, keinen Kaffee im Büro anzubieten, damit die Mitarbeiter in kleinen Gruppen zur Kaffeebar gegenüber pilgern und sich auf diese Weise die sozialen Kontakte festigen.

Warum sollte man also, wenn wir uns nicht gerade in einer Pandemie befinden, überhaupt remote arbeiten wollen? Die häufigsten Gründe hierfür sind, dass das Unternehmen keinen Sitz in der Nähe des Wohnortes hat oder es die Familiensituation nicht zulässt, täglich ins Büro zu gehen. Hier kommen auch schon die Vorteile von Remote-Arbeit zum Tragen: flexiblere Arbeitszeiten, extrem kurzer Arbeitsweg und mehr Kontrolle darüber, auch mal für längere Zeit ungestört und konzentriert zu arbeiten. Allerdings klagen viele Remote-Mitarbeiter über mangelnde Information, soziale Isolation, lange Arbeitstage und nicht klar definierte Ziele. Aber bevor es um diese eher weichen Faktoren geht, lasst uns mit der Basis – der physischen Ausstattung – beginnen.

Der Arbeitsplatz

Als Mitarbeiter von Technologieunternehmen brauchen wir zum Glück nicht viel Equipment: einen Laptop, ein bis zwei Monitore, Tastatur, Maus und eine schnelle Internetverbindung. Los gehts. Das ist schon ein guter Anfang, allerdings benutze ich noch ein paar mehr Dinge, die mir die tägliche Arbeit im Home-Office erleichtern:

- Gute Kopfhörer (am besten mit Geräuschunterdrückung), damit die Familie oder Mitbewohner ihren Alltag nicht an meine Arbeitszeiten anpassen müssen.
- Normale, kabelbasierte Kopfhörer sind meine erste Wahl für Videokonferenzen. Ich war schon in zu vielen Meetings, bei denen meine Gesprächspartner aufgrund von nicht erkannten Bluetooth-Kopfhörern nichts hören konnten.
- Eine Lampe, die ich für Videokonferenzen auf mein Gesicht ausrichte, damit meine non-verbalen Reaktionen besser gesehen werden können. Es ist absolut wichtig, dass alle Gesprächsteilnehmer direkt erkennen können, ob ich lächle, grimmig dreinschaue oder zweifle.
- Ein Whiteboard, um Ideen schnell festzuhalten.
- Einen Sessel im Arbeitszimmer (falls vorhanden), damit ich mal die Position und Perspektive wechseln kann. Acht Stunden am Schreibtisch sitzen kann ganz schön eintönig werden.

Man braucht allerdings nicht zwingend eine 4K-Webcam, ein Ring-Light um die Kamera herum, ein professionelles Podcast-Mikrofon oder einen Green-Screen. Damit eure Kollegen eure Arbeitsumgebung besser kennenlernen, solltet ihr auf einer Wiki-Seite oder einem Trello-Board Fotos von euren Arbeitsplätzen teilen. Oft kennt man nur die Webcam-Ansicht mit einer langweiligen Wand oder dem immer gleichen Bild im Hintergrund. Wenn man das Remote-Setup seiner Team-Kollegen besser kennt, hilft das beim Aufbau einer gemeinsamen Remote-Identität.

Der Remote-Alltag

Wenn man allein von zuhause aus arbeitet, kann ein Tag schon einmal sehr lang werden. Deshalb sollte man den Arbeitstag strukturieren, insbesondere wenn man in einem weltweit verteilten Team arbeitet, da Meetings auch schonmal aufgrund der Zeitverschiebung früh morgens oder spät abends stattfinden können. Nehmt euch Zeit für Pausen zwischendurch, gerne auch mal länger. Es kann allerdings dauern, bis man sich daran gewöhnt hat. Meistens liegt es daran, dass man meint, immer per Chat, Videokonferenz oder Telefon sofort erreichbar sein zu müssen. Die Kollegen sollen ja nicht auf die Idee kommen, man arbeite gerade nicht. Allerdings ist genau das auch völlig in Ordnung. Ihr müsst daheim nicht erreichbarer sein als im Büro. Es hilft aber deinem Team, wenn es weiß, ob du gerade am Schreibtisch sitzt oder nicht. Ich benutze mein Chat-Tool dazu, um meinen momentanen Status anzuzeigen, wie beispielsweise: Bin beim Laufen, esse gerade zu Mittag oder begleite meine Tochter zum Fußballtraining. In manchen Teams sollte man diese freien Zeiten allerdings abstimmen.

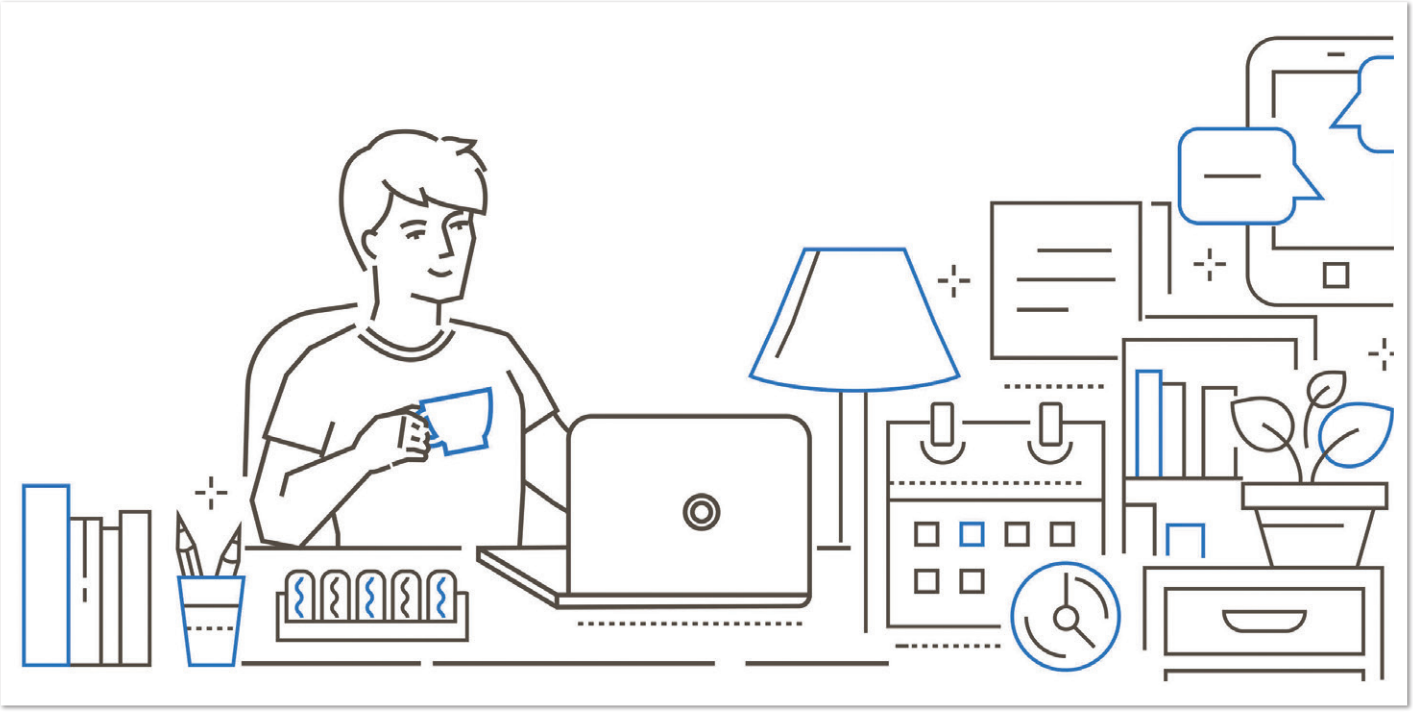
Auch sollte der Tag einen klaren Rahmen haben. Lasst es nicht zur Regel werden, am Abend auf der Couch noch eine wichtige E-Mail zu verfassen oder ein Konzept fertig zu schreiben. Da die Arbeit mit dem Zuhause physisch verbunden ist, fällt es Remote-Arbeitern schwerer, sich mental von der Arbeit zu trennen und wirklich Feierabend zu machen. Speziell, wenn man mit weltweit verteilten Teams zusammenarbeitet, gibt es Tage, an denen man noch spät ein Meeting hat oder man unbedingt noch eine Chat-Nachricht verschicken muss, damit andere weiterarbeiten können. Dies sollte aber die Ausnahme bleiben. Bitte.

In Verbindung bleiben

Im Büro arbeiten bedeutet, dass man seine Kollegen fast jeden Tag trifft und mit ihnen kurz sprechen kann. Man sieht, ob ein Kollege gerade im Stress oder offen für ein kurzes Gespräch ist. Wenn man remote arbeitet, läuft Kommunikation meist viel geplanter: Meetings einberufen, Wiki-Seiten oder Dokumente teilen oder eine Chat-Nachricht verfassen.

Videokonferenzen sind wohl das, was einem persönlichen Gespräch am nächsten kommt. Per Video kommunizieren ist allerdings anders, als physisch mit Kollegen im Meetingraum zu sitzen. Hier ein paar Regeln, die mein Team aufgestellt hat:

- **Everyone from their own laptop.** So ist gewährleistet, dass jeder Teilnehmer die gleichen Voraussetzungen hat, an Gesprächen teilzunehmen. Jeder macht die gleiche Erfahrung mit verzögerter Übertragung und Aussetzern bei Bild und Ton. Für Remote-Arbeiter sind Videokonferenzen oftmals schwierig, bei denen die Mehrheit der Teilnehmer physisch in einem Meetingraum sitzt und sich unterhält. Ich habe schon oft in solchen Sitzungen die Gesprächslücke verpasst, um mich in die Diskussion einzubringen, oder die emotionale Dynamik im Raum falsch interpretiert.
- **Who's the moderator?** Oft lassen sich nicht alle darauf ein, vom eigenen Laptop aus ein Meeting zu machen, wenn das Gros der Teilnehmer im Büro anwesend ist. Man sollte dann einen Moderator bestimmen, der die Remote-Mitarbeiter explizit mit einbezieht.



Dabei spricht dieser die Kollegen im Home-Office aktiv an, um sie in das Gespräch einzubeziehen und ihnen den Raum zu geben, den die anderen schon physisch haben.

- **Mute yourself.** Wenn sich mehr als drei Teilnehmer in einer Videokonferenz befinden, sollte nur derjenige das Mikrofon eingeschaltet haben, der auch gerade spricht. Alle anderen schalten das Mikrofon stumm, damit Nebengeräusche anderer Teilnehmer nicht den momentanen Hauptredner überlagern.
- **Camera on.** In einem physischen Meeting kann man sich auch nicht unsichtbar machen. Es ist einfacher, die Emotionen und direkten Reaktionen zu erkennen, wenn man seine Gesprächspartner gut sehen kann. Ich bevorzuge die Galerieansicht, da mich die Gesichtsausdrücke aller in der Videokonferenz befindlichen Personen interessieren und ich nicht nur die aktuelle Sprecherin sehen möchte.

Wie gesagt, Videokonferenzen werden meist im Voraus geplant und terminiert. Aber was ist, wenn ich spontan jemandem nur kurz eine Frage stellen oder ein schnelles Statusupdate geben möchte? Chat-Tools wie Microsoft Teams, Mattermost und Slack helfen bei der Adhoc-1:1-Kommunikation, Chat-Räume helfen bei der Abstimmung im Team oder bei Anfragen an interne Dienstleistungsabteilungen wie IT, Design, HR oder Recht. Wichtig für Remote-Teams sind auch soziale Chat-Räume. Remote-Mitarbeiter treffen sich halt nicht zufällig an der Kaffeemaschine, um die neuesten Technologietrends oder Unternehmensneuigkeiten zu besprechen. Solche Diskussionen kann man aber trotzdem führen: in speziell eingerichteten Watercooler- oder Küchenchaträumen. Auch solltet ihr in Chat-Tools als Avatar ein richtiges Foto von euch wählen. Zum einen, damit ihr erkannt werdet, wenn ihr persönlich aufeinandertrefft, und zum anderen vermittelt einem dies eher das Gefühl eines sozialen Kontaktes. Wer fühlt sich schon gut dabei, mit Donald Duck die Roadmap für das neue Produkt abzustimmen?

Für Diskussionen sind Chat-Tools ein sehr geeignetes Medium. Wenn man allerdings die Diskussionsergebnisse festhalten oder teilen möchte, sollte man Plattformen wie Confluence, die Google Suite oder Notion einsetzen. Diese Tools helfen Remote-Arbeitern dabei, Ideen, Pläne und Entscheidungen mit anderen zu teilen oder Feedback einzuholen. Wichtig hierbei ist, dass **jedem** im Unternehmen **alle** Informationen zur Verfügung stehen. Unternehmensweit haben wir hierzu zwei einfache Regeln aufgestellt:

- Die Standardeinstellung ist „offen für alle": Wer eine Seite oder ein Dokument erstellt, sollte es für jeden sichtbar und auffindbar machen.
- Wenns nicht im Wiki steht, ist es nie passiert: Alle Informationen und Entscheidungen müssen im Wiki festgehalten werden oder sie existieren nicht.

Das ist nicht immer einfach durchzusetzen. Gerne werden Entscheidungen im Büro einfach bei einer Unterhaltung am Schreibtisch getroffen. Damit aber Remote-Mitarbeiter darüber informiert und beteiligt sind, müssen die Entscheidungen im Wiki abgelegt und geteilt werden. Das ist ein längerer Lernprozess speziell für die Kollegen im Büro. Wenn man aber diese beiden einfachen Regeln in die Unternehmenskultur einbindet und konsequent lebt, hilft es enorm, dass sich die Mitarbeiter im Home-Office nicht von Informationen abgeschnitten fühlen.

Sichtbar bleiben

Doch Kollaborationstools können noch mehr: Sie helfen, dass Remote-Mitarbeiter nicht vergessen werden. Lieber einmal zu häufig kommunizieren als zu wenig. Man kann einfach auf einen internen Blog-Post oder in einer Diskussion mit einem „Like" reagieren oder einen Kommentar schreiben, schon das allein fördert die Sichtbarkeit bei Kollegen.

Eine weitere Möglichkeit, um von den Office-Mitarbeitern nicht vergessen zu werden, hat sich ein ehemaliger Kollege von mir ausgedacht: Adam ist vom Büro in Sydney nach Amsterdam gezogen. Durch den Zeitzonenunterschied war es ihm nicht mehr möglich, an

allen Meetings teilzunehmen und natürlich schon gar nicht an sozialen Team-Events. Um aber im Arbeitsalltag der Australier nicht vergessen zu werden, hat er sich einen lebensgroßen Adam-Pappaufsteller anfertigen lassen und nach Sydney geschickt. Das Ergebnis: Adam ist weiterhin sichtbar, da man ihn an einem zentralen Ort im Büro aufgestellt hat und das Team auch den Pappaufsteller zu jedem Meeting und Team-Event mitnimmt.

Soziale Beziehungen stärken

Als Mitarbeiter im Home-Office trifft man oft die immer gleichen Teammitglieder. Ein Treffen mit Kollegen aus anderen Abteilungen außerhalb von Projekten findet kaum statt. Dabei sind diese Beziehungen wichtig, um andere Perspektiven kennenzulernen und das eigene Arbeitsnetzwerk zu erweitern. Es ist immer gut, Kollegen aus anderen Bereichen zu kennen. Im Büro hat man die Möglichkeit, sich zufällig beim Mittagessen oder bei der Yoga-Stunde zu treffen. Wie schafft man das remote? Einige Unternehmen mit einem höheren Anteil an räumlich verteilten Mitarbeitern organisieren wöchentliche virtuelle Treffen. Dabei werden zwei Mitarbeiter unterschiedlicher Abteilungen zufällig ausgewählt und zu einem 15-minütigen Meeting eingeladen. Es ist den Gesprächsteilnehmern überlassen, über was sie sich unterhalten wollen: Arbeit, Freizeit, Familie oder Technologietrends. Das persönliche Unternehmensnetzwerk wird damit auf einfachste Weise erweitert.

Tatsächlich ist es aber auch extrem wichtig, sich persönlich zu treffen. Vollständig remote-geführte Unternehmen wie Trello oder GitLab organisieren jährlich ein physisches Zusammenkommen aller Mitarbeiter an einem Ort. Dabei geht es aber nicht primär darum, die Strategie für das nächste Jahr festzulegen, gemeinsame Ziele zu besprechen oder Projekte abzugleichen. Das Ziel dieser Events ist es, soziale Bindungen zwischen den Mitarbeitern zu vertiefen. Es geht darum, gemeinsam etwas zu erleben, sich persönlich besser kennenzulernen und in einer relaxten Atmosphäre zusammen Zeit zu verbringen. Je größer eine Organisation, desto schwieriger wird es allerdings, ein solches Meeting zu organisieren. Um nicht ganz auf die Vorteile des physischen Zusammenkommens verzichten zu müssen, können solche Treffen für einzelne Organisationseinheiten stattfinden. Zusätzlich sollte sich ein Remote-Team aber mindestens dreimal im Jahr persönlich treffen, um gemeinsam an neuen Ideen und Strategien zu arbeiten.

Es ist eine Umstellung

Auch wenn uns Technologien wie Videokonferenzen, Chat, gemeinsam bearbeitete Dokumente und so weiter zur Verfügung stehen, unterscheidet sich das Arbeiten im Home-Office doch sehr von dem im Büro. Bei mir hat es zirka zwei Jahre gedauert, bis ich mich an die Situation gewöhnt habe, mir meinen Tag einzuteilen, regelmäßige Pausen einzulegen, mich an die häusliche Stille zu gewöhnen und mich jeden Tag selbst zu motivieren. Nicht mehr die Möglichkeit zu haben, schnell mal beim Kollegen vorbeizuschauen, um ein Statusupdate zu bekommen und nebenbei noch zu fragen, wie es gestern im Fußballstadion war, war eine große Umstellung. Auch folgende Erkenntnis habe ich über die Jahre gemacht: Nicht jeder kann über einen längeren Zeitraum vollständig remote arbeiten. Einige von uns brauchen einfach die regelmäßigen direkten Kontakte.

Unternehmen ohne zentrale Büros bleiben wohl auch auf längere Zeit die Ausnahme. Dabei bietet Remote-Arbeit nicht nur den Mitarbeitern, sondern auch den Unternehmen viele neue Chancen: Mehr Flexibilität bedeutet größere Attraktivität und öffnet die Reichweite bei der Suche nach neuen Mitarbeitern. Die besten Kandidaten wohnen oft nicht in derselben Stadt, in der das Unternehmen einen Bürositz hat.

Softwareprojekte werden zunehmend komplexer und auch die Verschmelzung der verschiedenen Disziplinen wie Programmieren, Design und Operations erfordert oft das Arbeiten in räumlich verteilten Teams. Videokonferenzen werden zunehmend Besprechungen mit trockenen Keksen ablösen und Chat-Nachrichten werden Telefonanrufe zu unpassenden Zeiten ersetzen. Dabei sollten Unternehmen Remote-Arbeit nicht als zu akzeptierendes Übel betrachten. Eine Remote-First-Strategie fördert die Transparenz und Flexibilität für alle, ob im Home-Office oder im Büro. Remote is here to stay.

Sven Peters
MongoDB
sven.peters @mongodb.com

Sven Peters, Developer Advocate bei MongoDB, beschäftigt sich seit mehr als 15 Jahren mit Trends in der Softwareentwicklung. Sven hilft Unternehmen dabei, kulturelle Werte zu erkennen, sodass Teams ihr volles Innovationspotenzial ausschöpfen können. Er arbeitet seit 2011 im Home-Office mit weltweit verteilten Teams für Unternehmen wie Atlassian, K15t und MongoDB.

Ikigai – die japanische Formel für Zufriedenheit

Veit Richter, Emendare GmbH & Co KG

Was verbirgt sich hinter dem Wort „Ikigai"? Wie funktioniert die Formel für Zufriedenheit und wie kann es uns helfen, uns mit der eigenen Motivation tiefer auseinanderzusetzen? Diese Fragen werden im folgenden Artikel näher beleuchtet. Neben der Erklärung dafür, was Ikigai ist und woraus es sich zusammensetzt, werden im Folgenden verschiedene Übungen mit auf den Weg gegeben, wie man sich selbst mit dem Thema auseinandersetzen kann und seinen eigenen Weg somit reflektiert.

Ikigai lässt sich aus dem Japanischen mit iki (生き = Leben) und gai (がい = Wert) übersetzen. Jedoch geht es nur bedingt darum, den „Sinn des Lebens" zu entdecken oder eine tiefere spirituelle Erfahrung zu sammeln, sondern vielmehr um die eigene Motivation und das Gefühl, die richtigen Dinge zu tun. Eine schönere Umschreibung der Thematik ist für mich diese: „Der Grund, morgens aufzustehen." Wer kennt es nicht, der Wecker klingelt (für einige schon um fünf Uhr in der Früh) und man denkt sich: „Ach, ich könnte auch genauso gut noch liegen bleiben… Auf X (beliebige Tätigkeit hier einsetzen) und vor allem Y habe ich heute überhaupt keine Lust." Ikigai versucht hier, diese Motivation tiefer zu hinterfragen. Nach dieser Weisheit hat man den Wert seines Lebens erkannt (beziehungsweise entsprechend geschaffen), wenn man morgens mit voller Energie aus dem Bett springt und sich auf den Tag freut.

Ikigai – die vier Kreise

Ikigai findet man in der Regel in Form eines Venn-Diagramms erklärt. Ikigai setzt sich aus vier verschiedenen Werten, die jeweils miteinander Schnittmengen bilden, zusammen. Die vier grundlegenden Bereiche, die es zu beachten gilt, sind:

1. **Wofür ich brenne** – Morgens voller Elan aus dem Bett zu kommen und sich auf das Tagewerk zu freuen, setzt voraus, eine Tätigkeit auszuüben, die man gerne tut und die einem Spaß macht.
2. **Worin ich gut bin** – Bestimmte Dinge werden uns von Geburt an mitgegeben. Sowohl psychologische als auch physische Grundlagen erlernen wir in jüngster Kindheit. Aber auch Schule, Studium und Hobbies können unsere Fähigkeiten in bestimmten Bereichen fördern. Trotzdem kann es Tätigkeiten geben, bei denen man das Gefühl hat, einfach nicht besser zu werden, egal wie viel Aufwand man investiert.
3. **Was die Welt benötigt** – „I want to put a ding in the universe." Der Innovator, der Pixar und Apple zum Erfolg führte, prägte diesen Satz. Auch Ikigai beschäftigt sich mit der Frage, wie viel man der Welt hinterlassen möchte. Berühmtheit muss nicht das Ziel einer jeden Person sein, dennoch darf man sich die Frage stellen, ob das eigene Leben anderen Menschen, der Welt oder Umwelt etwas Gutes tut und welchen Wert man über das eigene Dasein stiftet.

4. **Womit ich Geld verdiene** – Selbst, wenn ich etwas gut beherrsche, es auch gern tue und es die Welt zu einer besseren macht, hilft es alles nichts, wenn ich auf kurz oder lang damit auf der Straße lande. Eine erfüllende Tätigkeit muss auch finanziell ausreichend sein, damit man davon leben kann.

Schaut man sich die Schnittmengen näher an, entstehen hier neue Begriffe:

1. Die Schnittmenge aus „Wofür ich brenne" und „Worin ich gut bin" bezeichnet Ikigai als **Leidenschaft**. Oft sind diese beiden Dinge nah miteinander verwandt. Auf Vorträgen kamen vermehrt Personen auf mich zu, die sagten, dass sie früher liebend gern programmierten und auch richtig gut darin waren. Mit der Zeit stellten die einen aber fest, dass sie immer weniger Spaß daran hatten, Konferenzen zu besuchen oder sich fortzubilden. Andere bemerkten, dass junge neue Mitarbeiter neuere Technologien kannten und diese besser einzusetzen wussten. Der jeweils andere Wert schwand parallel.
2. Eine **Mission** hat jemand, der etwas tut, wofür er brennt und was gut ist für die Welt. Also mit dem eigenen Feuer andere anstecken und die Welt verändern möchte. Nicht umsonst hört man immer mal wieder von Menschen, die für einen guten Zweck ein Sabbatical einlegen, unbezahlt Unterstützung leisten oder eigene Freiheiten aufgeben, um in Armenhäusern ehrenamtlich zu unterstützen.
3. Tue ich etwas, was die Welt benötigt und womit man Geld verdienen kann, ist dies nach Ikigai die **Berufung**. Hat man jedoch eine starke Berufung, ohne die anderen zwei Gebiete von Ikigai auszufüllen, verliert man wohl auch schnell die Motivation. Zwar erhält man Geld und sieht auch glückliche Menschen; man fühlt sich dennoch nie richtig zufrieden und hat keinen Spaß an dem täglichen Wirken.
4. Zu guter Letzt vereint die **Profession** Dinge, in denen ich gut bin und womit ich Geld verdienen kann. Man kennt die Geschichte: Ein erfolgreicher Börsenmakler wirft auf einmal hin, kauft sich einen Bauernhof und fängt an, dort Schafe zu hüten. Nur weil man gut in etwas ist und sich damit gut seinen Lebensunterhalt verdienen kann, hat man noch lange nicht sein Ikigai erreicht.

Der Wert des Lebens kann für jeden Menschen anders aussehen. Füllt man diese Kreise mit seinen persönlichen Kernelementen aus, sieht keines dieser Diagramme für verschiedene Individuen gleich aus. Eine motivierende Tätigkeit erlangt man also, wenn man eine Aufgabe findet, die einem Spaß macht, in der man gut ist, die wertvoll für die Welt ist und mit der man gleichzeitig Geld zum Leben verdienen kann.

Motivation im Arbeitsumfeld

Eine nette Gruppenübung, die ich in der Vorbereitung zu meinem Ikigai-Vortrag zusammen mit Yvonne Luppold (Scrum Masterin bei Heine in Karlsruhe) erfand, führt Teilnehmer in die Welt von Sinn und Signifikanz der eigenen Arbeit ein. In dieser Übung sammeln Teilnehmer zuerst verschiedenste Ideen, wie man Teams und Individuen motivieren kann. Häufig höre ich hier Sachen wie: „Gehalt, herausfordernde Aufgaben und Gummibärchen". Die Übung besteht darin, einen Stapel Münzen zu einem Turm zu errichten und diesen dann Münze für Münze auf den Kopf zu stapeln und wieder zurück… 15 Minuten lang. Die Reaktionen und Blicke der

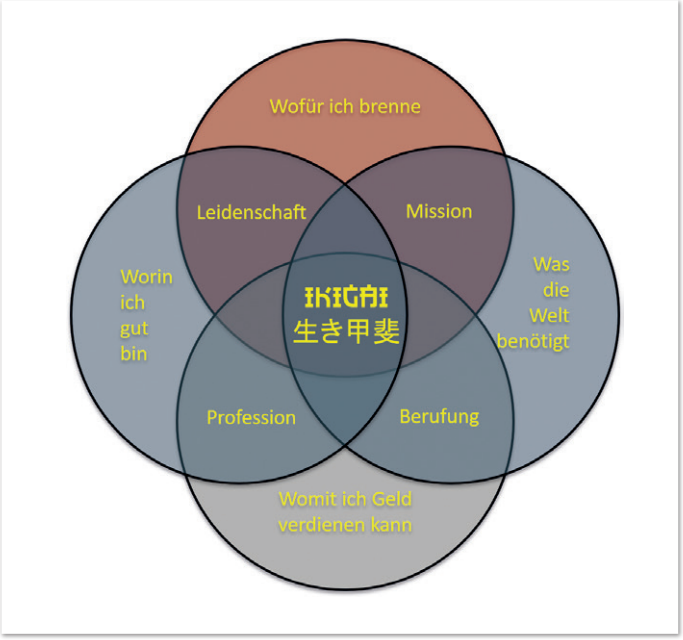


Abbildung 1: Ikigai als Venn-Diagramm (© Veit Richter, Emendare)

„Arbeitenden" ist geradezu magisch. Während einige fleißig anfangen, ihre Münztürmchen zu bauen, resigniert die Hälfte bereits nach spätestens 30 Sekunden. Von verständnislosen „Der Redner möchte uns wohl veräppeln" bis hin zu lächelnden Erkenntnisgewinnern ist hier alles dabei.

Wie wir nun von Ikigai wissen, zahlen wohl alle drei Motivatoren auf ein Gebiet von Ikigai ein. Während Gehalt und Gummibärchen in Form von Belohnung und Verdienst wahrgenommen werden, zielt eine herausfordernde Arbeit auf die Leidenschaft, die Schnittmenge von dem, worin ich gut bin, und dem, was mir Spaß macht, ab. Auch die anderen Punkte für sich scheinen nicht die Motivation auszu-

machen, um diese Übung des Münzenstapelns für länger als drei Minuten durchzuziehen. Und doch gibt es immer wieder Arbeiten in unserem Alltag, die sich überhaupt nicht so sehr von dieser Sisyphos-Aufgabe unterscheiden.

Meine Verbindung zu Ikigai

Während meines Studiums der Unternehmensinformatik stolperte ich das erste Mal über eine Problematik, die ich erst heute im Nachgang mit Ikigai gut beschreiben kann. Im Praxissemester durfte ich sechs Monate lang erleben, wie es ist, als „Code Monkey" in China zu arbeiten.

Statt als Software-Ingenieur Kundenlösungen zu entwickeln, wurden vorgekaute Lösungen runtergeschrieben und über Zäune geworfen. Die allgemeine Motivation und Zufriedenheit waren gering und abends suchten Mitarbeiter nach neuen Jobs, in denen sie die gleiche Aufgabe hatten – allerdings mehr Geld dafür bekamen. Zu diesem Zeitpunkt wusste ich, dass ich Spaß am Programmieren hatte. Feedback von Kollegen und Noten der Universität spiegelten mir zudem ein sehr gutes Skill-Level wider. An der Leidenschaft für Softwareentwicklung haperte es also nicht. Verschiedene Formate berichteten damals vom Fachkräftemangel und guten Gehältern. Dennoch öffnete mir mein Auslandspraktikum die Augen. Obwohl drei der Kreise für mich gut gefüllt waren, fühlte es sich so an, als ob es noch etwas anderes geben müsste. Etwas, womit ich Alltag und Motivation von Entwicklern beeinflussen könnte. Dies war mein Startschuss, den vierten Kreis „etwas Gutes für die Welt tun" anzusteuern. Seit damals änderte ich meine Ausrichtung hin zu selbstorganisierten, kundenfokussierten Teams, die ihre Motivation nicht nur rein aus dem Gehalt beziehen. Seither unterstütze ich Teams mit agilen Methoden und kann somit eine (für mich!) weitaus erfüllendere Tätigkeit ausüben.

Persönliche Selbstreflektion

Eine solche Reise selbst zu erleben und sich mit Ikigai zu beschäftigen kann helfen, sein persönliches Ikigai zu finden. Außerdem könnte es sein, dass man bereits die perfekten Voraussetzungen für Ikigai im Alltag hat. Auch hier kann es wertvoll sein, sich dies noch einmal vor Augen zu führen, um die persönliche Zufriedenheit zu unterstützen.

Um diese Selbstreflektion zu unterstützen, möchte ich euch herzlich zu einem Experiment einladen: Sucht in eurem Kalender einen freien Zeitraum von ein bis zwei Stunden, sucht euch ein ruhiges Zimmer und schafft ein Umfeld, in dem ihr euch gut auf euch selbst konzentrieren könnt. Für viele heißt das, sich diese Anleitung auszudrucken und mit Stift und Papier bewaffnet in sich zu gehen. Im nächsten Schritt fokussiert ihr euch auf die folgenden Fragen und versucht, sie für euch zu beantworten. Am besten gebt ihr euch selbst für jede Frage einen etwas größeren Zeitrahmen, um noch ein wenig mehr herauszukitzeln:

- Wofür ich brenne
  - Welche Facetten meines Alltags liebe ich?
  - Warum und seit wann?

- Worin ich gut bin
  - Worin bin ich gut?
  - War ich schon immer gut darin? Wenn nein, was war der Auslöser?

- Was die Welt benötigt
  - Was tue ich Gutes für die Welt oder Umwelt?
  - Was möchte ich einmal hinterlassen?

- Womit ich Geld verdiene
  - Womit verdiene ich meinen Lebensunterhalt?
  - Ist es genug? Und was bedeutet „genug" für mich?

Im nächsten Schritt gilt es, sich den Schnittmengen zu widmen. Nehmt hierfür zum Beispiel vier verschiedenfarbige Stifte zur Hand und unterstreicht die Dinge, die ihr liebt und in denen ihr gut seid mit einer Farbe. Dinge, mit denen ihr Geld verdient und die ihr gut könnt mit einer anderen Farbe. Wiederholt dies für eure „Mission" und eure „Berufung". Vielleicht ergeben sich hier schon Muster und neue Erkenntnisse.

Im vorletzten Schritt gilt es nun noch einmal zu kondensieren, welche der gefundenen Tätigkeiten und Dinge alle vier Gebiete des Ikigai anschneiden. Nehmt euch nun idealerweise eine kleine Auszeit. Macht zum Beispiel einen kurzen Spaziergang, trinkt einen Kaffee oder tut etwas, um euch abzulenken, bevor ihr auf euren Ikigai zurückkommt. Versucht dann, die letzte Frage für zu reflektieren: Was fehlt?

Ich wünsche euch auf dieser Reise viele Erkenntnisse und entstehende positive Energie. Sollten dennoch Fragen offen sein, oder möchtet ihr noch mehr erfahren, lade ich euch herzlich ein, mit mir in Verbindung zu treten. Solltet ihr mich einmal zum Beispiel auf einer Konferenz (vielleicht ja sogar zum Thema Ikigai) treffen, sprecht mich gern an. Bestimmt habe ich auch dann für euch eine Münze aus dem obigen Beispiel, die euch auf eurem Weg des Ikigai unterstützen kann.

Ich wünsche euch, dass ihr euren Grund findet, morgens aufzustehen.

Veit Richter
Emendare GmbH & Co. KG
veit.richter@emendare.de

Schon während seines Unternehmensinformatikstudiums durfte Veit kennenlernen, wie es ist, in einem auf bloßes Coding fokussierten Team zu arbeiten. Damals arbeitete er in China und lernte dort Taylorismus und die damit entstehende Mitarbeiterdemotivation direkt kennen. Dieser Moment änderte seine Ausrichtung weg von der reinen Softwareentwicklung hin zu Scrum und Agilität. Seitdem entwickelt Veit Teams Führungskräfte und sich selbst weiter. Hin zu einem agilen Mindset, ausgerichtet darauf, durch Spaß und intrinsische Motivation von Teams ganz natürlich den Output zu verbessern.

Der Hype um Deep Learning: Geschichte und Einführung in das Thema

Denis Stalz-John und Mark Keinhörster, codecentric AG

Seit 2013 sind Begriffe wie Deep Learning oder künstliche Intelligenz in aller Munde. Sie haben sich in Wirtschaft und Industrie etabliert und ersetzen Schritt für Schritt statische Regelwerke und starre Algorithmen. Der Artikel erklärt, was Deep Learning genau ist, wie es sich von dem Oberbegriff „künstliche Intelligenz" abhebt und wo Vor-, aber auch Nachteile der Technologie liegen.

Bereits im Jahr 1958 entstand mit dem Rosenblatt-Perzeptron das erste Konzept zur Modellierung neuronaler Netze. Damals als einfaches künstliches Neuron entwickelt, bildet es auch heute noch die Grundlage für komplexe Modellarchitekturen. Der wahre Hype um die Begriffe künstliche Intelligenz (KI), Machine (ML) und Deep Learning (DL) entstand jedoch erst 50 Jahre später mit AlexNet, einer Implementierung eines tiefen neuronalen Netzes, die 2012 mit großem Abstand die ImageNet Large Scale Visual Recognition Challenge (ILSVRC) gewann, ein Wettbewerb zur Bildklassifikation und Objekterkennung. Der zugrunde liegende Datensatz ist ein Subset des ImageNet-Datensatzes, eine Bilddatenbank mit über 14 Millionen Bildern in mehr als 21.000 Kategorien. Ausschlaggebend für den Hype waren aber nicht nur die überragenden Ergebnisse, sondern auch der Fakt, dass das Modell vollständig auf einer Grafikkarte und damit in vertretbarer Zeit trainiert wurde. Damit leitete AlexNet den Durchbruch für neuronale Netze als Teilbereich des Machine Learning ein und weckte das Forschungsinteresse für immer komplexere und tiefere Architekturen, die unter dem Begriff Deep Learning zusammengefasst werden. Mit dem steigenden Interesse an KI, ML und DL *(siehe Abbildung 1)* hielten auch moderne Frameworks zur Entwicklung von ML-Modellen, wie zum Beispiel TensorFlow, PyTorch oder DeepLearning4J, Einzug.

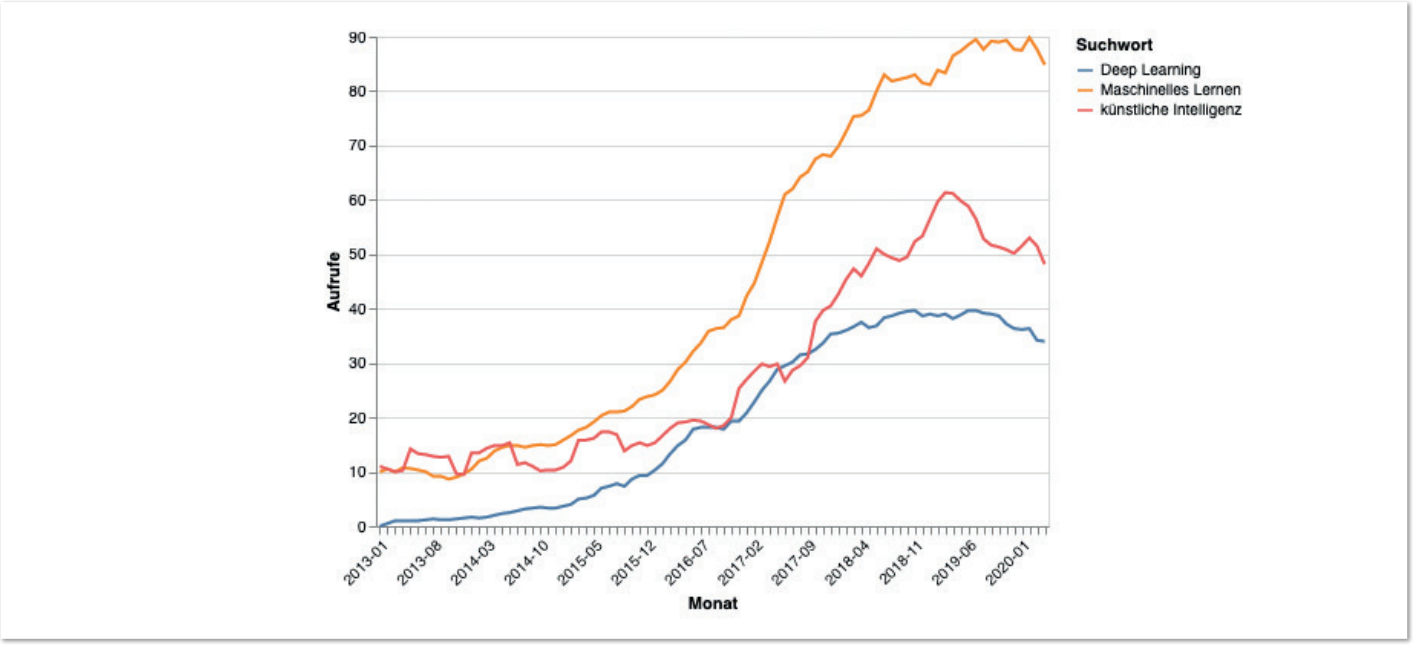


Abbildung 1: Google-Suchvolumen aufgeteilt nach KI, ML und DL (Google Trends)

Deep Learning, Machine Learning, künstliche Intelligenz

AlexNet ermöglichte den Durchbruch von Machine- und Deep-Learning-Verfahren, die sich mittlerweile auch in der Wirtschaft und Industrie fest etabliert haben. Aber was unterscheidet eigentlich Deep Learning von Machine Learning und wie lassen sich die beiden Begriffe unter den Oberbegriff der künstlichen Intelligenz einordnen? *Abbildung 2* gibt einen Überblick über die Zusammenhänge der einzelnen Begrifflichkeiten.

Künstliche Intelligenz ist ein sehr unscharf definierter Oberbegriff für jedes „intelligent" erscheinende Verhalten von Maschinen. Dieses Verhalten ist unabhängig von der eingesetzten Technologie oder den eingesetzten Verfahren. Ein fest einprogrammiertes Regelwerk in einem Schachcomputer, dass für jede Situation einen schlauen Zug parat hat, würde man von außen betrachtet und ohne weitere Information bereits intelligent bezeichnen – auch wenn nur eine riesige Sammlung von Regeln dahintersteckt. Der Begriff bezieht sich damit nur auf äußerliches Verhalten und hat damit auch immer eine subjektive Komponente. Ob ein Computer, Programm oder Algorithmus intelligent ist, liegt im Auge des Betrachters.

Ein Teilbereich der künstlichen Intelligenz ist das Machine Learning. Beim Machine Learning lernt ein Modell aus einer Reihe von Beispielen, dem Datensatz, bestimmte Muster, anhand derer allgemeine Regeln abgeleitet werden, die zu einem erwarteten Ergebnis führen. Es existiert eine Reihe grundverschiedener Modelle, die dazu in der Lage sind, und es kommen täglich neue dazu. Vom einfachen Entscheidungsbaum, der Regelabfolgen lernt, bis hin zum künstlichen neuronalen Netz (KNN), das in seiner grundsätzlichen Form dem menschlichen Gehirn nachempfunden ist. Es verknüpft beliebig viele künstliche Neuronen miteinander, die in Schichten organisiert sind. Die Anzahl der Schichten wird auch als Tiefe bezeichnet.

Sobald ein neuronales Netz besonders viele Zwischenschichten (Layer) verwendet, wird oftmals schon von Deep Learning (tiefes Lernen) gesprochen. Zusätzlich zeichnet sich Deep Learning aber durch einen weiteren Punkt aus. Während einfache KNNs versuchen, Muster direkt in den Eingangsdaten zu finden und damit auf eine möglichst durchdachte Vorverarbeitung angewiesen sind, versuchen Deep-Learning-Modelle zunächst, abstraktere Merkmale aus den Daten zu extrahieren, die in den Folgeschichten weiter abstrahiert und am Ende wie im einfachen KNN verarbeitet werden. Dadurch haben Deep-Learning-Modelle nicht nur besonders viele Layer, es fallen auch viele Schritte in der Datenvorverarbeitung weg, die das DL-Modell im Trainingsprozess übernimmt.

Deep Learning im Alltag

Nach den ersten Forschungsergebnissen wurde schnell deutlich, welches große Potenzial in Deep-Learning-Verfahren schlummert. So fingen Industrieunternehmen bald an, diese Technologie in ihre Produkte und Prozesse zu integrieren. Heutzutage kommen wir mit vielen Produkten, die Deep Learning enthalten, in Berührung. Angefangen vom Smartphone bis hin zum Online-Shopping.

Ein prominentes Beispiel ist hierbei die Spracherkennung beziehungsweise das Sprachverständnis. Als Vergleich der Entwicklung kann hierzu die noch vor einigen Jahren vorhandene Sprachsteuerung bei Automobilen herangezogen werden. Diese ermöglichte dem Nutzer, eine Auswahl von Befehlen per Sprache zu tätigen. So konnte man damit zum nächsten Lied springen oder die Lautstärke erhöhen. Bevor ein Sprachkommando aufgenommen und verarbeitet wurde, musste für gewöhnlich ein Knopf, zum Beispiel am Lenkrad, getätigt werden. Des Weiteren gab es vor der Benutzung des

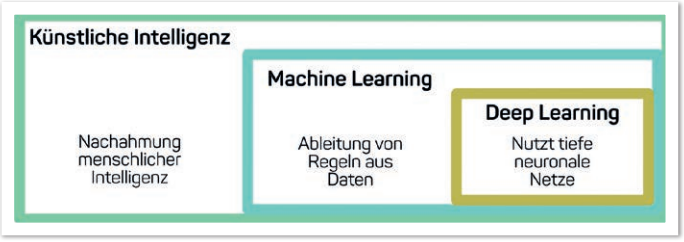


Abbildung 2: Einordnung von KI, ML und DL

Systems eine sogenannte Anlernphase. Hier sprach der Nutzer die Kommandos ein, damit sie zur Benutzung verglichen werden konnten. Es handelte sich also mehr um ein System, das Audioschnipsel verglich, als um eine wirkliche Erkennung. Der aktuelle Stand hat sich in vielerlei Hinsicht verbessert. Zuallererst kommen die digitalen Assistenten mit einer automatischen Erkennung von Schlüsselwörtern daher (beispielsweise „Alexa", „Ok, Google", „Hey, Siri"). Die darauffolgenden Sätze werden automatisch erkannt und verarbeitet. Dabei muss der Nutzer weder die Wörter vorher eingesprochen haben noch sich in der Komplexität begrenzen. Google geht sogar noch einen Schritt weiter. Mit ihrem Duplex-System bieten sie die Möglichkeit, für einen Nutzer telefonbasiert Termine zu vereinbaren. So kann dieser beispielsweise angeben, in welchem Zeitraum er einen Friseurtermin möchte, daraufhin erledigt die Google-Duplex-Software automatisch den Anruf und informiert den Nutzer über den Verlauf des Telefonats.

Ein weiteres Gebiet, in dem Deep Learning in unseren Alltag vordringt, ist das autonome Fahren beziehungsweise Fahrerassistenzsysteme. Obwohl Ersteres noch weit von der Serienreife entfernt ist, sind Teilsysteme mit integrierten ML und DL schon in einer Vielzahl von Ober- bis Mittelklasseautos der letzten Jahre enthalten. Ein Beispiel ist die Erkennung von Verkehrsschildern: Hierbei werden Bilder von einer Kamera aufgenommen, die für gewöhnlich hinter der Frontscheibe in der Höhe des Innenspiegels verbaut ist. Die aufgenommenen Bilder werden von einem Algorithmus, der DL oder klassisches ML enthält, ausgewertet. Bei einem erkannten Schild kann dieses zum Beispiel als Symbol in der Konsole des Fahrers angezeigt werden. Um ein autonomes Fahrzeug serienreif zu entwickeln, werden sehr viele unterschiedliche Algorithmen benötigt. Hervorzuheben ist hierbei, dass dies nicht nur ML-Verfahren sind, sondern von ihnen immer nur bestimmte Teilaufgaben übernommen werden.

Daten und neuronale Netze

Das einfachste neuronale Netz nennt man Multilayer Perceptron (MLP, siehe *Abbildung 3*). Es besteht aus mehreren Schichten und jede davon wiederum aus mehreren Neuronen. Die Neuronen sind zwischen den Schichten miteinander verbunden. Die erste Schicht nennt man Eingabe- und die letzte Ausgabeschicht. Jede Verbindung, der sogenannte Kanten, besitzt einen skalaren Wert, das Gewicht. Um die Berechnung zu veranschaulichen, nehmen wir an, dass wir einen Bilddatensatz verwenden, auf dessen Bildern entweder eine Birne oder ein Apfel zu sehen ist. Das Ziel des neuronalen Netzes ist es, zu bestimmen, welche dieser beiden Früchte sich auf einem Bild befindet.

Für die Auswertung eines Bilds mit dem MLP werden zuerst die Werte des Datenpunkts in die Neuronen der Eingabeschicht eingetragen. Im Falle eines Bildes ist dies für jeden Kanal eines Pixels ein Neuron. Daraufhin werden die Berechnungen von der ersten bis zur letzten Schicht durchgeführt, bis die Ausgabeschicht das Ergebnis zurückgibt.

In der Praxis wäre die Verwendung eines MLP für die Erkennung von Bildern deutlich zu rechenaufwendig, daher wird in solchen Fällen auf Convolutional Neural Networks (CNNs) gesetzt. Aufgrund ihrer Architektur besitzen sie deutlich weniger Gewichte und nutzen dadurch außerdem die Struktur eines Bildes aus.

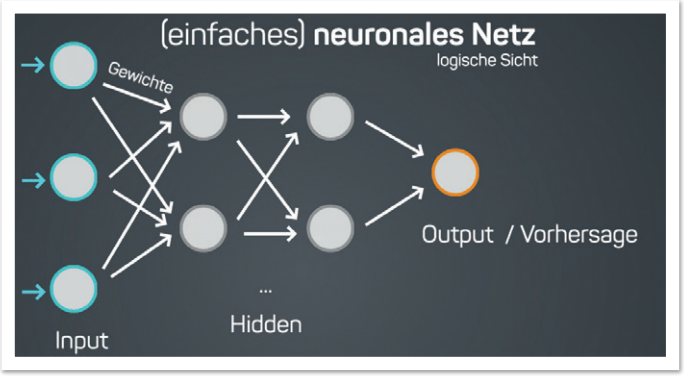


Abbildung 3: Darstellung eines Multilayer Perceptron

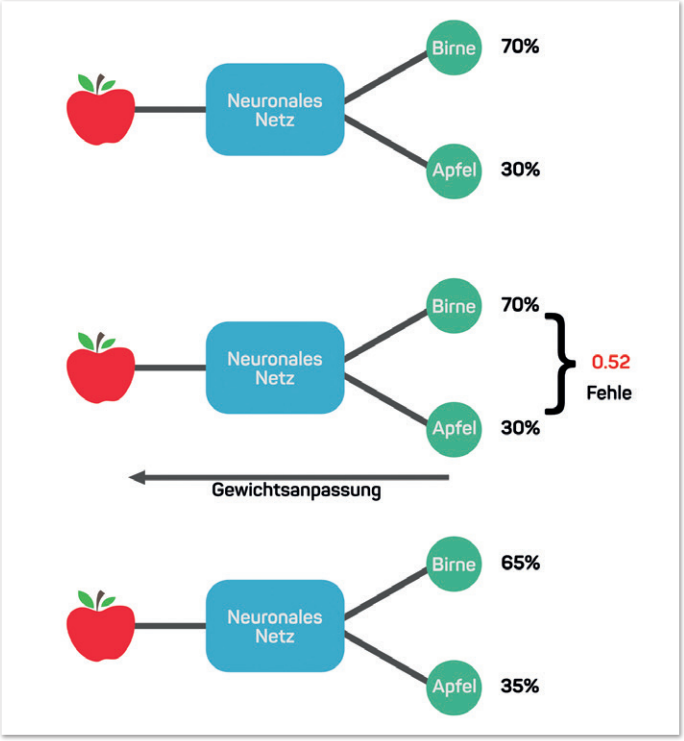


Abbildung 4: Visualisierung eines Trainingsschrittes bei einem neuronalen Netz

Bei der Unterscheidung von zwei oder mehreren Klassen enthält die Ausgabeschicht genauso viele Neuronen wie die Anzahl der Klassen, die unterschieden werden sollen. Jedes Neuron steht dabei für eine Klasse. Deren Ausgaben werden dann zu einer Wahrscheinlichkeitsverteilung umgerechnet. Die Berechnung des neuronalen Netzes wird für jedes Neuron einzeln durchgeführt. Zuerst werden die einkommenden Werte mit den entsprechenden Kantengewichten multipliziert und anschließend aufsummiert. Auf das Ergebnis wird eine nicht-lineare Funktion angewandt und an alle ausgehenden Kanten weitergegeben. Durch diese Berechnungsweise ist die Ausgabe des Netzes von dessen Architektur und den Gewichten abhängig.

Für gewöhnlich wird der Aufbau des Netzes manuell festgelegt, die passenden Gewichte für das gewünschte Verhalten müssen jedoch systematisch gefunden werden. Dieses Vorgehen bezeichnet man auch als Trainingsphase. Dazu wird ein Datensatz benötigt, bei dem die gewünschte Ausgabe bekannt ist. Das bedeutet, in den oben genannten Beispielen muss gespeichert sein, auf welchem Bild sich ein Apfel und auf welchem sich eine Birne befindet. Zuerst werden

nun alle Gewichte des Netzes zufällig initialisiert und liefern damit auch zufällige Ausgaben. Nun werden die Gewichte anhand der Daten optimiert. Dazu werden mit dem Netz aufeinanderfolgend die Bilder verarbeitet und die berechnete Ausgabe mit der gewünschten Ausgabe verglichen *(siehe Abbildung 4)*. In Abhängigkeit davon, wie weit diese Werte auseinanderliegen, werden die Gewichte minimal angepasst, sodass die Ausgabe etwas besser zu dem gegebenen Datenpunkt passt. Wiederholt man den Vorgang mehrfach für alle Bilder aus seinem Datensatz, nähert sich die Ausgabe des Netzes der gewünschten Ausgabe an.

Ist alles ganz einfach?

Das erste neuronale Netz ist also schnell an einem Beispieldatensatz trainiert und liefert auch sehr gute Ergebnisse. Darüber hinaus gibt es allerdings eine Vielzahl von Herausforderungen, die die Resultate negativ beeinflussen können. Um einige Beispiele zu nennen: unpassende Netzarchitektur, falsche Hyperparameterwahl beim Training, zu wenig Daten oder schlechte Datenqualität. Des Weiteren ist eine besondere Eigenschaft maschinellen Lernens das sogenannte Overfitting. Dabei handelt es sich um die Anpassung des Netzwerks auf die gezeigten Trainingsbeispiele ohne eine adäquate Übertragung auf andere Daten.

Das bedeutet, das KNN lernt lediglich die Trainingsdaten auswendig, kann aber auf neue Daten nicht generalisieren. In dem hier verwendeten Beispieldatensatz würde dies bedeuten, dass nach dem Training alle Äpfel und Birnen richtig erkannt wurden. Sobald aber externe Bilder in das Netz gegeben werden, sinkt die Erkennungsrate deutlich. Zu einem gewissen Teil geschieht dieses Auswendiglernen der Trainingsdaten bei allen DL-Algorithmen. Das heißt, in der Regel sind die Ergebnisse auf dem Trainingsdatensatz immer besser als bei neu getesteten Bildern. Um zu messen, wie stark dieser Effekt auftritt, wird der ursprüngliche Datensatz in zwei unterschiedliche Kategorien, Test und Training, aufgeteilt. Dabei wird der Testdatensatz beim Training ausgespart und kann somit als komplett neu für das neuronale Netz betrachtet werden. Mit dem Vergleich der Resultate aus dem Trainings- und Testdatensatz lässt sich abschätzen, wie viel Overfitting auftritt und wie gut die Generalisierungsfähigkeit des Netzes ist. Es gibt verschiedenste Methoden, diesem Effekt zu begegnen. Die beiden einfachsten sind, die Anzahl der Datenpunkte im Trainingsdatensatz zu erhöhen oder die Anzahl der Gewichte im neuronalen Netz zu verringern. Beides führt zu einer Erschwerung des Auswendiglernens der Daten und dadurch auch zu geringerem Overfitting.

Fazit

Deep Learning ist längst nicht mehr nur in der Forschung vertreten, sondern mittlerweile auch ein fester Bestandteil in Industrie und Wirtschaft. Gerade in den Bereichen der Sprachverarbeitung und des maschinellen Sehens haben ML- und insbesondere DL-Modelle die klassischen Verfahren und Algorithmen abgehängt. Trotzdem gilt auch hier die Redewendung „There is no free lunch". Machine-Learning-Methoden bringen einmal mehr neue Komplexität in die Softwareentwicklung, die beherrscht werden muss. Vom qualitativ hochwertigen Datensatz über die richtige Netzarchitektur bis hin zum systematischen Reduzieren des Overfitting gilt es, viele Punkte zu beachten. Aus diesem Grund sollte jedes Problem und dessen Lösung individuell betrachtet werden, bevor mit der großen ML-Kanone auf vermeintlich kleine Softwarespatzen geschossen wird.

Quellen

[1] Alom, Md Zahangir, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S. Awwal, und Vijayan K. Asari. 2018. „The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". *http://arxiv.org/abs/1803.01164*.

[2] Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, und L. Fei-Fei. 2009. „ImageNet: A Large-Scale Hierarchical Image Database". In *CVPR09*.

[3] Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, u. a. 2015. „ImageNet Large Scale Visual Recognition Challenge". *International Journal of Computer Vision (IJCV)* 115 (3): 211–52. *https://doi.org/ 10.1007/s11263-015-0816-y*.

Denis Stalz-John

codecentrig AG

denis.stalz-john@codecentric.de

Denis spezialisiert sich auf die Bereiche Computer-Vision, semantische Segmentierung, Object Detection und Deep Learning. Sein beruflicher Einstieg erfolgte beim Corporate Research der Robert Bosch GmbH im Bereich Fahrer-Assistenzsysteme und autonomes Fahren. Seit 2018 arbeitet er als Data Scientist bei der codecentric AG.

Mark Keinhörster

codecentrig AG

mark.keinhoerster@codecentric.de

Mark ist im Big-Data-Zoo zu Hause und bringt Erfahrungen mit Hadoop und Apache Spark mit. Außerdem beschäftigt er sich mit Docker, Cloud-Technologien und Machine Learning.

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen

iJUG Verbund

www.ijug.eu

Impressum

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

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

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

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Mylène Diacquenod
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

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

Fotonachweis:
Titel: Bild © Quardia | *https://www.shutterstock.com*
S. 9: Bild © Pavel Konovalov | *https://de.123rf.com*
S. 12 + 13: Bild © Siarhei | *https://stock.adobe.com*
S. 15: Bild © Aleksey Telnov | *https://de.123rf.com*
S. 19: Bild © kantver | *https://de.123rf.com*
S. 20: Bild © Brett Lamb | *https://de.123rf.com*
S. 23: Bild © Petr Ciz | *https://stock.adobe.com*
S. 25: Bild © Lukas Kurka | *https://de.123rf.com*
S. 28: Bild © Ruslan Gilyazov | *https://de.123rf.com*
S. 30: Bild © aurielaki | *https://de.123rf.com*
S. 35: Bild © Anton Chervov | *https://de.123rf.com*
S. 38: Bild © Sergey Nivens | *https://de.123rf.com*
S. 42: Bild © bonumopus | *https://de.123rf.com*
S. 46: Bild © rawpixel | *https://de.123rf.com*
S. 50: Bild © Monsit Jangariyawong | *https://de.123rf.com*
S. 53: Bild © REDPIXEL | *https://stock.adobe.com*
S. 56: Bild © maximmmmum | *https://de.123rf.com*
S. 59: Bild © Evgenii Naumov | *https://de.123rf.com*
S. 63: Bild © Ales Utouka | *https://de.123rf.com*
S. 65: Bild © Boiko Ilia | *https://de.123rf.com*
S. 67 + 68: Bild © Veit Richter
S. 70: Bild © rfsrn | *https://de.123rf.com*

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

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

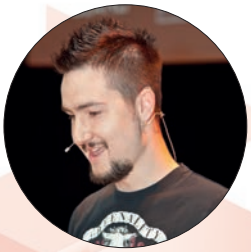
Druck:
WIRmachenDRUCK GmbH,
www.wir-machen-druck.de

Inserentenverzeichnis

2. + 3. DEZEMBER 2020

BERLINER EXPERTENSEMINAR

DOAG



NICOLAI PARLOG

Java after eight

KURSÜBERBLICK

In diesem Kurs werden die Java-Versionen 9 bis 15 beleuchtet. Dabei liegt der Fokus auf neuen Sprachfeatures wie Sealed Classes, Records, Text Blocks, switchExpressions und var, aber auch neue und erweiterte APIs sowie JVM- und Performance-Verbesserungen werden theoretisch vorgestellt und mit praktischen Übungen untermauert. Darüber hinaus werden der Migrationspfad von Java 8 zu 11 bzw. 15, der neue Release-Zyklus und die aktuelle Situation um Distributionen und Support besprochen.

ZIELGRUPPE

Erfahrene Java-Entwickler, die sich theoretisch und praktisch auf die neuen Java-Versionen vorbereiten möchten.

VORAUSSETZUNGEN

Einige Jahren praktische Erfahrung mit Java-Entwicklung und sicherere Java-8-Kenntnisse.

www.doag.org/go/expertenseminar_parlog