Ausgabe 06/2019

Java aktuell

JDK 13
Was gibt's Neues?

GitOps
Mit Helm und Kubernetes durchstarten

Microservices
Wie sinnvoll sind sie bei Transaktionen?

Wie sauberer Code Ihr Projekt nachhaltig verbessert

PROJEKT-BOOSTER <CLEAN CODE>

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

WIR GESTALTEN DIE DIGITALE ZUKUNFT DEUTSCHLANDS.

Werde Teil unseres Expertenteams und finde Deinen Einstieg als Developer oder Consultant in den Bereichen:

- Business Intelligence
- Data Analytics
- Middleware
- Infrastructure & Operations

Mehr Informationen: karriere.virtual7.de

Ordentlich ist nicht gleich clean

Diese Ausgabe widmet sich dem Thema „Clean Code", von dem Sie sicherlich schon einmal, wenn auch nur am Rande, etwas mitbekommen haben. Entwickler/innen sind in diesem Bereich sehr verschieden, denn sauberer Code ist ein subjektives Thema. Einige haben den persönlichen Anspruch und die Angewohnheit, diesen nach bestimmten Richtlinien zu schreiben und andere nicht.

Doch ab wann gilt mein Code überhaupt als „sauber" und welche Voraussetzungen muss er dafür erfüllen? Welche Vorteile ergeben sich daraus für die Software und mein Projekt? Diese und weitere Fragen beantwortet Holger Tiemeyer, Pentasys, ab Seite 17 in seinem Artikel „Die Definition of Clean". Darin teilt er wertvolle Tipps, Tricks und Regeln für das Schreiben von sauberem Code und beschreibt, wieso ein Umlernen sich lohnt.

Weiterhin erhalten Sie in dieser Ausgabe wieder Neuigkeiten und Wissenswertes rund um Ihre Lieblingssoftware. Aktuelle Informationen und Geschehnisse der letzten Monate erhalten Sie im Java-Tagebuch und in der Eclipse Corner. Letztere ist diesmal gefolgt von einem Interview mit Jan Supol zum Thema Jersey und Jakarta EE. Bernd Müller, Ostfalia, holt die Klasse „Files" aus seiner SDK-Schatztruhe und Falk Sippach beleuchtet das Java Release 13 im Detail. Ob Microservices und aufeinander aufbauende Abfolgen von Aktionen (Transaktionen) sich vereinen lassen, untersuchen Matthias Koch und Markus Grabert ab Seite 29. Tauchen Sie außerdem in die Welt von Tests und Testautomatisierung ein. Anja Papenfuß-Straub, ING Deutschland, zeigt Ihnen ab Seite 54 wie sinnvoll das Testen von Software ist. Diese und viele weitere spannende Artikel erwarten Sie in dieser Ausgabe.

Wir wünschen Ihnen viel Spaß beim Lesen!

Ihre

Lisa Damerow

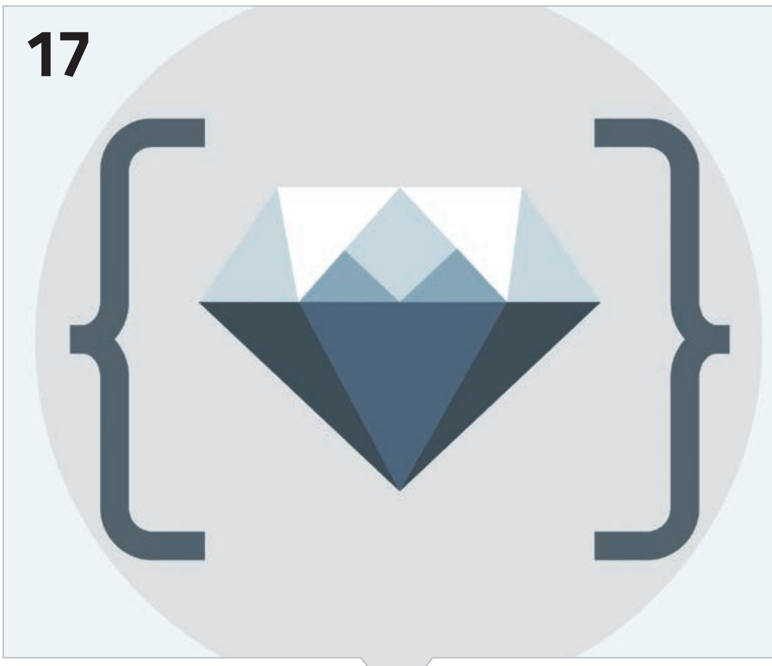
Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

11

Entdecken Sie verborgene Schätze im Java SDK

17



Wieso es sich für jeden Entwickler lohnt, auf sauberen Code zu achten

3 Editorial

6 Java-Tagebuch
Andreas Badelt

8 Markus' Eclipse Corner
Markus Karg

9 Java aktuell im Interview: Jersey und Jakarta EE
Interview mit Jan Supol

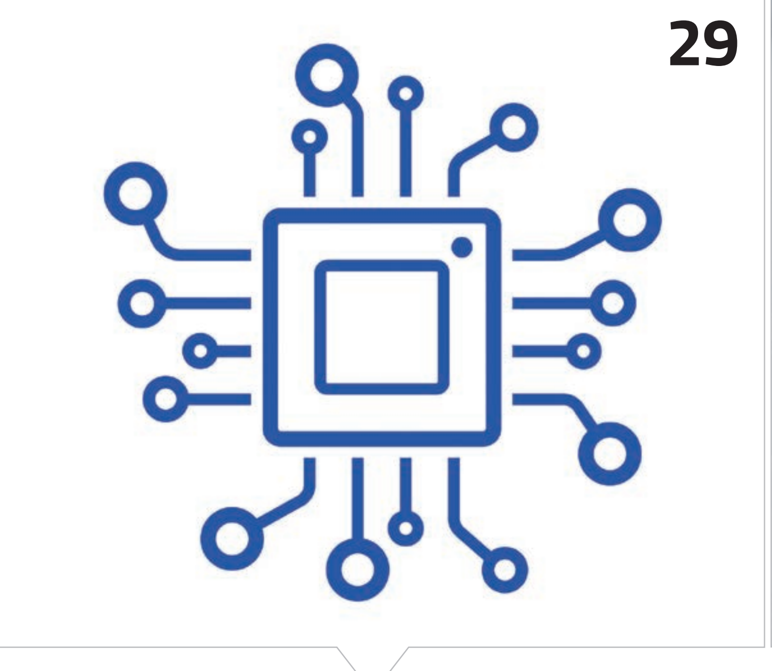
11 Unbekannte Kostbarkeiten des SDK
Heute: Die Klasse „Files"
Bernd Müller

13 Und halbjährlich grüßt das Java Release
Falk Sippach

17 Die „Definition of Clean" – Wie sauberer Code zu besseren Ergebnissen in Software-Entwicklungsprojekten führt
Holger Tiemeyer

21 GitOps mit Helm und Kubernetes
Bernd Stübinger und Florian Heubeck

26 Make Java Hot Again
Heiko Rupp



Microservices und Transaktionen – geht das?

Keine Angst vorm Testen

29 Hallo Microservices, gibt's euch auch transaktional?
Matthias Koch und Markus Grabert

38 Langfristige, effiziente Web-UI-Entwicklung für Unternehmensanwendungen
Björn Müller

44 „So entwickelt sich eine App weiter: Man beobachtet, was die User brauchen und was für sie wichtig ist"
Interview mit Michal Harakal

48 Automatisierung und Durchblick mit (Git-)Hub und (Git-)Lab
Kai Schmidt

54 Edit and P(r)ay – Oder doch lieber testen?
Anja Papenfuß-Straub

59 „Man kann selbst entscheiden, ob man Teil des Wandels sein möchte oder diesen ignoriert"
Interview mit Lars Thomsen

62 Impressum/Inserenten

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

07. Juni 2019

JakartaOne

Es herrscht wieder Aufbruchstimmung rund um Jakarta EE. Am 10. September wird es zum ersten Mal eine JakartaOne-Konferenz geben, erst mal als eintägige, virtuelle Konferenz. Am Programm „basteln" unter anderen Reza Rahman, Adam Bien und (als Projektleiter von EE4J) Ivar Grimstad.Exakt zu diesem Datum soll dann auch Jakarta EE 8 erscheinen. Mal sehen, in den GitHub-Projekten herrscht jedenfalls Betriebsamkeit. Derweil wird aber immer noch über den Umgang mit den Package-Namen diskutiert: „Big Bang" oder inkrementelle Umbenennung. Mitglieder der Working Group analysieren die Möglichkeiten, durch Toolunterstützung Binärkompatibilität zu Java EE zu erhalten.

10. Juni 2019

Dependencies via HTTP Reloaded

Man-in-the-Middle-Attacks auf Libraries, die zum Beispiel von Maven über HTTP geladen werden? Da war doch mal was! Vor ungefähr fünf Jahren hatte sich ein Nutzer beschwert, dass verschlüsselte Verbindungen zu Maven Central nur gegen Extrazahlung möglich sind. Mit dem „Man-in-the-Middle-Proxy Dilettante" (Code auf GitHub) demonstrierte er die Möglichkeiten der Code Injection, die sich dadurch ergaben – nicht nur für Geheimdienste. Sonatype hatte seine Policy daraufhin sofort geändert, und HTTPS war eigentlich durch das geschärfte Bewusstsein auch sonst überall gesetzt. Dachten wir. Ein Gradle-Mitarbeiter hat nun vor derselben Falle bei indirekten Dependencies gewarnt, nachdem er festgestellt hat, dass das Kotlin-Framework Ktor seine Dependencies per HTTP lädt. Eine Suche über öffentliche GitLab-Projekte zeigte schnell, dass das Problem sehr weit verbreitet ist. In seinem Blog *[1]* hat er eine Liste der gefundenen Projekte veröffentlicht. Es ist nicht damit zu rechnen, dass alle Projekte die nötigen Anpassungen in Windeseile durchführen (können). Für sicherheitsbewusste Entwickler hat Spring Source ein Projekt veröffentlicht, mit dem sich betroffene Projekte identifizieren und mit etwas Scripting auch ohne großen manuellen Aufwand fixen lassen *[2]*.

13. Juni 2019

Java auf iOS

Insbesondere auf der GraalVM und deren AOT Compiler basierend, hat Gluon eine Beta-Version des neuen Gluon Client Plug-ins veröffentlicht. Mit dieser kann Java(FX) jetzt auch nativ für iOS kompiliert werden – und sie soll unter bestimmten Bedingungen für Open-Source-Projekte kostenlos nutzbar sein *[3]*. Schon klar, viele werden jetzt wieder sagen, direkte native Programmierung sei das einzig Wahre und Gute. Aber warum ohne Not einen Graben im Projekt schaffen, wenn die Anforderungen es zulassen? ...Und ich dazu in meiner Lieblingsprogrammiersprache entwickeln kann.

18. Juni 2019

JDK 13 auf der Zielgeraden

Das JDK 13 befindet sich in der „Rampdown-Phase", die Funktionalität ist damit vollständig und es werden nur noch Fehler behoben. Fünf Features haben es rechtzeitig auf den Zug geschafft: „Dynamic Class Data Sharing Archives" zum Application Tuning, „Z Garbage Collector: Uncommit Unused Memory", „Reimplement the Legacy Socket API" sowie als Preview „Switch Expressions" und „Text Blocks". Moment – Switch Expressions? Die waren tatsächlich bereits in JDK 12 als Preview enthalten und haben aufgrund des Feedbacks noch eine Änderung erfahren: „To yield a value from a switch expression, the break with value statement is dropped in favor of a yield statement" *[4]*. Auf dieser Grundlage soll dann in einem zukünftigen Release „instanceof pattern matching" in Switch Statements möglich sein *[5]*.

18. Juni 2019

Docker Snafu

Snafu steht für „Situation normal, all fucked up". So etwas in der Art ist denjenigen, die die offiziellen Debian-basierten Docker Images für Java 8 und 11 benutzen, seit Ende März begegnet. Durch ein falsches Tag im OpenJDK-Projekt, das eine „GA"-Version („generally available") suggerierte, die aber noch nicht alle Security Patches enthielt. Sie wurde daraufhin vom Debian-Projekt in offizielle Builds aufgenommen. Entdeckt wurde das wohl schon im Mai. Wer die ganze Geschichte lesen möchte und auch, welche Alternativen es zum Debian Build gibt, steigt am besten hier ein *[6]*.

14. Juli 2019

Visual Studio, Java Version

Microsoft, mit seinem eigenen Entwickler-Ökosystem einstmals „nur" als die große Konkurrenz zur Java-Welt angesehen, ist längst tief in letzterer angekommen. Das demonstriert das Unternehmen nicht nur personell und mit der Beteiligung an Projekten wie Micro-Profile, sondern auch in den „kleineren Dingen" wie einem eigenen Installer für die IDE Visual Studio Code speziell für Java-Entwickler. Auch die Azure SDKs sind kürzlich überarbeitet wurden, damit sie sich besser in die jeweils genutzte Programmiersprache einfügen, wobei dies aber nicht speziell für Java geschehen ist.

15. Juli 2019

Eclipse: Jakarta EE Update

Der Juli-Newsletter der Eclipse Foundation dreht sich wieder um Jakarta EE: Die JakartaOne-Konferenz, das für den 10. September angekündigte Release usw. –alles schon bekannt. Aber das hier

sollte erwähnt werden: Die Eclipse Foundation sucht nach Firmen und Personen, die in der Vergangenheit zu Java-EE-Spezifikationen beigetragen haben, um die Urheberrechte zu übertragen. Keine Angst, ein Kopfgeld wurde bislang nicht ausgesetzt, aber die Unterstützung für Jakarta EE wäre wirklich überaus nett! Ein Blog-Eintrag *[7]* beschreibt das Problem und die konkrete Bitte.

16. Juli 2019

MicroProfile: Läuft...

MicroProfile entwickelt sich zügig weiter. Einiges betrifft die Spezifikationen und deren Unterstützung durch die verschiedenen Hersteller. Letzten Monat wurde MicroProfile 3.0 freigegeben. Vor ein paar Tagen ist Reactive Messaging 1.0 herausgekommen, erst mal als „Standalone Specification". Parallel wird fleißig an einer GraphQL-Spezifikation gearbeitet *[8]*. Wer wissen möchte, welche Implementierungen es für welche Teile beziehungsweise Versionen von MicroProfile gibt, schaut hier *[9]*. Daneben gibt es aber auch Neuerungen bei den Tools rund um MicroProfile. Eine davon ist Boost – ein Maven-Plug-in zum Erzeugen herstellerspezifischer Jar Files (das allerdings noch nicht von allen Herstellern unterstützt wird). Und für den MicroProfile Starter (start.microprofile.io) gibt es jetzt auch ein Command Line Interface.

17. Juli 2019

IBM: Mehr Unterstützung für Kubernetes-Entwickler

IBM, für mich immer noch eher der CloudFoundry-Typ (BlueMix), ist nicht erst seit dem Kauf von Red Hat (mit OpenShift) voll im Kubernetes-Fieber. Drei neue Projekte hat Big Blue jetzt vorgestellt, die zusammen die Entwicklung rund um die ursprünglich von Google entwickelte Container-Plattform erleichtern sollen: Kabanero, Appsody und Codewind. Letzteres ist aus dem Projekt Microclimate entstanden und bietet Plug-ins für die IDEs Eclipse, Eclipse Che und Visual Studio, um das Entwickeln von „cloud native" Apps zu erleichtern. Appsody bietet Templates für die App-Entwicklung für Java (MicroProfile und Spring), JavaScript (Node.js / Express) und Swift. Go wäre wohl zu viel Google. Aber Apples Swift – ist das die Antwort auf Java für iOS?

18. Juli 2019

OpenJDK-Entwicklung künftig unter Git

Die Pläne reifen schon seit Längerem, mit JEP 357 gibt es nun ein offizielles Proposal, um die OpenJDK-Entwicklung von Mercurial auf Git umzustellen – zumindest für alle „Single-Repository-Projekte". Wo sie dann genau liegen sollen, wird erst noch festgelegt.

19. Juli 2019

Don't Go!

Die nächste RedMonk-Umfrage zu Programmiersprachen: TypeScript, das JavaScript für richtige Softwareentwickler (war das zu gemein?), ist jetzt immerhin auf Platz 10 zu finden. JavaScript selbst liegt aber immer noch auf Platz 1, es gibt also noch viel Raum für Verbesserungen. Java ist weiterhin auf Platz 2, Scala auf 13 und Kotlin auf 20 zu finden. Go ist hinter R auf Platz 16 gefallen.

24. Juli 2019

AdoptOpenJDK Quality Assurance

Auch als Antwort auf das geschlossene JCK hat die AdoptOpenJDK Community eine umfassende Suite offener Tests erstellt, mit denen sichergestellt werden soll, dass die Qualität der Releases „den hohen Qualitätsanforderungen von Enterprise-Kunden entsprechen". AdoptOpenJDK Quality Assurance (AQA) soll kontinuierlich unter Zuhilfenahme von Metriken und Nutzer-Feedback weiterentwickelt werden, um auch das Richtige zu testen. Auch Tests von Third-Party-Applikationen (Tomcat, Jenkins, Kafka und andere) sowie MicroProfile TCKs sind Bestandteil der Suite und sorgen für Relevanz. Die AQA werden detailliert in einem Google Doc beschrieben *[10]*.

Referenzen

[1] *https://tinyurl.com/y5a9jycy*
[2] *https://github.com/spring-io/nohttp*
[3] *https://tinyurl.com/y3genjkq*
[4] *https://openjdk.java.net/jeps/354*
[5] *https://openjdk.java.net/jeps/8213076*
[6] *https://tinyurl.com/yyl6499n*
[7] *https://tinyurl.com/y65z4l3w*
[8] *https://github.com/eclipse/microprofile-graphql*
[9] *https://wiki.eclipse.org/MicroProfile/Implementation*
[10] *https://tinyurl.com/y6rgyhz4*

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

Jahrelang haben Verbände und Einzelpersonen lautstark und unnachgiebig gekämpft, um den Untergang von Java EE zu verhindern. Oracles bekannt unglückliches Händchen mit Open-Source-Produkten (man erinnere sich an die Geschichte von Open Office, Open Solaris oder Hudson) in Kombination mit monatelangen Terminverschiebungen brachten den Kessel zum Kochen. Das unter dem Eindruck des ultimativen Streichens von APIs und Features eher als „Not-Release" angesehene Java EE 8 war dann der Höhepunkt der Enterprise-Leidenskarriere und eher der stattfindenden Java One geschuldet als einer messbaren Feature-Vollständigkeit. Vor zwei Jahren dann hat Oracle dem Druck endlich nachgegeben oder sagen wir, die Reißleine gezogen: Java EE wurde in die Obhut der Community gegeben – an die Eclipse Foundation. Inzwischen stellt sich heraus: Es war ein Pyrrhussieg.

Die Eclipse Foundation ist nämlich kein Zusammenschluss enthusiastischer Programmierer. Sie ist vielmehr der Insolvenzverwalter abgehalfterter Produkte im Auftrag ihrer strategischen Mitglieder. Auch wenn ihr charismatischer Präsident, Mike Milinkovich, nach dem damaligen Zusammentreffen von IBMs aufgegebener VM „OpenJ9", IDE „Eclipse" sowie Oracles aufgegebenem Framework „Java EE" auf der EclipseCon Europe 2017 in Ludwigsburg die Stiftung euphorisch als den Hort der Innovation anpries – Realität ist eher: Friedhofsgärtnerei.

Denn wo und was ist denn nun in den letzten zwei Jahren passiert? Wo ist die hohe Pace, die die Stiftungsmitglieder versprochen hatten? Bis auf einzelne Ausnahmen, bei denen Einzelpersonen tatsächlich hervorragende Arbeit geleistet haben, ist faktisch kaum etwas passiert.

Ja, Oracle hat geliefert: Sämtlicher Quellcode ist da, wenngleich ohne Historie. Die TCKs sind da und niemand außerhalb von Oracle traut sich, den Moloch anzufassen. Die Spezifikationen? Pustekuchen! Oracle und Vorgänger Sun hatten versäumt, sich von den Autoren das Recht zur Weitergabe einräumen zu lassen (wir erinnern uns an die Vorlesung „Recht für Informatiker" im Studium). Somit liegt es nun an der Eclipse Foundation, jeden einzelnen Mit-Urheber anzuschreiben und um Erlaubnis zu bitten. Diese Aktion findet derzeit statt. Allerdings ist hier die Rede von Hunderten Personen, deren aktuelle E-Mail-Adressen nach zwei Dekaden Java EE inklusive Arbeitgeber- und Providerwechseln teilweise kaum noch bekannt sein dürften. Das Vorhaben ist vermutlich aussichtslos, wie die Eclipse Foundation selbst einräumt. Und über die gescheiterten Verhandlungen um die Marke „Java" wurde an dieser Stelle schon ausgiebig berichtet.

Das war's dann also? Laut Project Management Committee (PMC) stehen IBM, Red Hat, Payara, Tomitribe und Fujitsu in den Startlöchern. Diese könnten neuen Code schreiben und neue Spezifikationen tippen. Sie würden nur warten, bis Oracle Jakarte EE 8 durch die Tür geschoben hat. Davor wurde übrigens behauptet, sie würden warten, bis der Markenrechtsstreit beigelegt sei. Davor wurde behauptet, sie würden warten, bis die TCKs übertragen seien. Davor… Ihr wisst, was ich sagen will. Hätten diese Unternehmen wirklich vor, sich nennenswert ins Zeug zu legen, wären sie längst kräftig am Rudern. Doch schaut man, was tatsächlich unter deren Leitung passiert, stellt sich wiederum Ernüchterung ein: Man schaue sich das Programm der großspurig in Analogie an vergangene Oracle-Zeiten „Jakarta One" genannten virtuellen Konferenz an. Ist das alles? Ein bisschen Faselei um die wichtigsten APIs? Fortschritt sieht für mich anders aus.

Und nun? Kopf in den Sand stecken? Nein, dafür ist diese Technologie zu wichtig. Im Gegenteil: Wenn es die Eclipse Foundation nicht zu leisten vermag, dann müssen eben andere ran – wir. Auch wenn es die Java User Groups mitsamt Dachverband iJUG bislang nicht geschafft haben, ihre Mitglieder in nennenswerter Zahl zur aktiven Programmier-Mitarbeit an Jakarta EE zu bewegen, ist es doch der einzige Ausweg, der bleibt. Und dieser sieht doch ganz gut aus: Neben zigtausend Java-Fähigen im deutschsprachigen Raum, von denen ein erheblicher Anteil täglichen Nutzen aus Java EE zieht, ist auch hier ein ganz neuer Protagonist an Bord: Microsoft. Der Konzern aus Redmond ist vor dem Hintergrund seiner Azur-Cloud schon seit Längerem als Unterstützer von Java bekannt. Bekannte Java-Größen wie Reza Rahman *[1]* und Ed Burns *[2]* stehen auf seiner Payroll und werkeln bereits fleißig an „Java made by Microsoft". Was dabei herauskommt, wollten die beiden mir aber leider vorab nicht verraten.

Insofern ist das Glas weder halb voll noch halb leer, sondern entsprechend Schrödingers Katze beides auf einmal. Es kommt nur darauf an, wer reinschaut. Ich für meinen Teil werde weiterhin zu den Aktiven zählen, da die direkte Einflussnahme auf Jakarta EE und die implementierenden Produkte mir und meinem Unternehmen einen direkten Marktvorteil verschafft. Daher investieren wir ganz gezielt Arbeitszeit in die genutzten Open-Source-Projekte. Wenn nur ein Bruchteil der Unternehmen, die lautstark den Tod von Java EE beweinen, unserem Beispiel in ähnlicher Weise folgen würden, hätte Jakarta EE eine stabile und prosperierende Zukunft. Die Hand ist ausgestreckt – die Entscheidung fällt jetzt.

Referenzen

[1] *https://github.com/m-reza-rahman*
[2] *https://github.com/edburns*

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.

Java aktuell im Interview: Jersey und Jakarta EE

Jan Supol ist bei Oracle Projektleiter von Eclipse Jersey, der bisherigen Referenzimplementierung von JAX-RS und, neben Weld (CDI), einer wichtigen Kernkomponente des Application-Servers Eclipse GlassFish. Jersey hat im August 2019 als erste Implementierung einen Zertifizierungsantrag für die Kompatibilität mit dem Standard Jakarta RESTful Web Services 2.1 eingereicht. Im Interview mit Markus Karg gibt Jan Supol einen Überblick über das Framework und darüber, was uns in Zukunft erwartet.

Jersey ist die Referenzimplementierung von JAX-RS 2.1. Sie ist in GlassFish enthalten und wird daher vermutlich im Moment von Tausenden von Menschen verwendet, ohne dass sie es wissen. Was macht Jersey besonders und wodurch ist es Wettbewerbern überlegen? Warum sollte ich gerade Jersey anstatt einer anderen JAX-RS-Implementierung nutzen?

Jan Supol: Bei Java EE gingen die Veröffentlichung einer Spezifikation und eines API schon immer Hand in Hand mit dem Release einer Referenzimplementierung. Diese diente dann als Proof-of-Concept. Das gilt auch für Jakarta EE. Mit dem Release des API wird auch eine kompatible Implementierung benötigt.

Jersey ist Teil von Jakarta EE, ebenso wie die Spezifikation zu Jakarta RESTful Web Services. Zwar kann jede kompatible Implementierung verwendet werden, aber Jersey wird eine der ersten Referenzimplementierungen zur Verarbeitung der neuen Spezifikationsfunktionen sein. Das liegt vor allem an den Beiträgen des Teams Jakarta RESTful Web Services zur Implementierung der neuen Jersey-Funktionalität.

Teil von Jakarta EE zu sein, ist ein riesiger Vorteil für Jersey. Jersey war durch seinen reichhaltigen Funktionsumfang schon immer beliebt. Als Teil von Jakarta EE wird das Beisteuern von neuen Ideen und Funktionsanforderungen einfach für die Community sein. Mit der wachsenden Beliebtheit von Jakarta EE wird auch die Anzahl an Beitragenden für das Jersey-Projekt wachsen.

Jersey ist bekannt als Teil von GlassFish und ich nehme an, es ist auch Teil von Helidon. Aber es kann direkt verwendet werden, beispielsweise als Teil einer Java SE Application. Warum sollte ich nicht einfach immer Java SE verwenden?

Jan Supol: Es ist absolut in Ordnung, Jersey in einer Java-SE-Umgebung zu verwenden. Allerdings bieten die Jakarta-EE-Plattform (für GlassFish) oder die Microprofile- Plattform (für Helidon) zusätzliche Features und Funktionen. Jersey ist in diese Plattformen integriert und Benutzer profitieren somit davon.

Die Community möchte sehr gerne über das Produkt ihrer Wahl auf dem Laufenden gehalten werden. Es gab großartige Blogs über Jersey, in den letzten Monaten wurden jedoch nur wenige Berichte veröffentlicht. Was geschieht hinter den Kulissen? Wird die Lage sich verbessern?

Jan Supol: Die Mitwirkung von Java EE bei der Eclipse Foundation ist eine gewaltige Aufgabe gewesen. Jeder Code-Schnipsel und jede Datei musste hinsichtlich möglicher Lizenzprobleme oder Sicherheitsverstöße geprüft werden. Jede Abhängigkeitsbeziehung musste von Architekten und Rechtsabteilungen betrachtet und untersucht werden. Es war ein Kraftakt und erforderte sehr viel Zeit von einer Vielzahl an Leuten. Das erste Release der Eclipse Foundation musste mit Java EE kompatibel sein, wodurch die Anzahl neuer Funktionen in Jersey eingeschränkt wurde.

Nun, da die Produkte stabil sind, planen wir ein neues Release alle drei bis sechs Monate. Diese Releases werden dann mit Blogs, Artikeln und Konferenzvorträgen unterstützt. Wir möchten da eine Menge an Informationen bereitstellen.

Wie ist der derzeitige Stand von Jersey? Die Veröffentlichung der Version 2.29 verlief recht still und leise. Was sind die Top-Eigenschaften?

Jan Supol: Jersey 2.26 war eine gewaltige Verbesserung, da es das erste Jersey Release war, das JAX-RS 2.1 implementierte, und es erhielt sehr viel öffentliche Aufmerksamkeit.

Bei Jersey 2.27 und 2.28 lag der Schwerpunkt auf dem Prozess zum Beitragen. Bei diesen Releases blieb das Jersey API unverändert und, abgesehen von ein paar kleineren Fehlerkorrekturen, war die wesentliche Neuerung die freundliche Eclipse-Lizenz.

Jersey 2.29 ist ein großes Release. Es enthält zahlreiche Korrekturen und Funktionen. Am wichtigsten ist aber, dass es das erste Release ist, das mit JDK 11 arbeitet. Wir haben außerdem Support für die Microprofile-Config-Spezifikation hinzugefügt und die Microprofile-Rest-Client-Spezifikation implementiert.

Was erwartet uns in naher Zukunft und auf der weiteren Roadmap? Kannst du uns die Top-Funktionen jedes geplanten Releases nennen? Und wann wird welches Release veröffentlicht?

Jan Supol: Das Ziel ist die Veröffentlichung eines neuen Releases alle drei bis sechs Monate, idealerweise alle drei bis vier Monate. Die Jakarta-EE-API-Pakete werden gerade von javax in den jakarta-Namensraum migriert und es ist unsere oberste Priorität, dass Jersey diese Dualität der APIs handhaben kann. Die Jakarta RESTful Web Services 2.2 werden bald veröffentlicht und Jersey wird sie so schnell wie möglich unterstützen. Wir müssen uns auch für wesentliche Verbesserungen bei den Jakarta RESTful Web Services 3.0 rüsten. Wir planen den Support von Java Fibers und einigen zusätzlichen Funktionen der Microprofile-Spezifikation, da diese Funktionen auch für Jersey-Benutzer sinnvoll sein können.

Jersey ist ein Open-Source-Projekt. Gibt es viele externe Beitragende und Mitwirkende, da das Projekt jetzt von Oracle zur Eclipse Foundation gewandert ist?

Jan Supol: Die Projekte in Jakarta EE werden der Eclipse-Community vorgestellt und erreichen größere Sichtbarkeit in der gesamten Java-Community; Leute, die Java EE verwenden oder nicht. Die Community kann die Arbeit an Jersey einsehen und wir können sehen, was die Mitwirkenden von Oracle Jersey uns zurückgeben. Wir nehmen ein Interesse von Leuten außerhalb der Java-EE-Community wahr. Sie möchten beispielsweise an Korrekturen von Jersey für Spring Boot mitarbeiten, im Wesentlichen sehen wir jedoch neue Beitragende aus der Jakarta-EE-Community.

Woran sollte jemand, der zu Jersey beitragen möchte, am besten arbeiten?

Jan Supol: Wir freuen uns über jede Art von Hilfe bei der Verbesserung von Jersey. Wir möchten, dass Leute zu Teilen beitragen, mit denen sie Erfahrung oder Probleme haben, oder zu Teilen, die sie verbessern möchten. Bei Jersey gibt es eine Liste von Feature-Requests und Problemen, die sich mit allen möglichen Interessengebieten von Mitwirkenden decken. Wenn jemand Neues etwas beitragen möchte, raten wir dazu, mit einer kleineren Aufgabe anzufangen und sich dann zu den größeren hochzuarbeiten. Es hängt im Endeffekt vom Selbstvertrauen und der Bereitschaft des Mitwirkenden ab.

Vielen Dank, Jan, für diese interessanten Einblicke!

Werden Sie Mitglied im iJUG!

Ab 15,00 EUR im Jahr erhalten Sie

30 % Rabatt auf Tickets der

Jahres-Abonnement der Java aktuell

Mitgliedschaft im Java Community Process

www.ijug.eu

Unbekannte Kostbarkeiten des SDK Heute: Die Klasse „Files"

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.

Mit Java 7 wurde das Package `java.nio.file` eingeführt, das unter anderem die Klasse `Files` enthält. Mit Java 8 wurden dieser Klasse einige neue Methoden hinzugefügt, weitere mit Java 11 und 12. Diesen neuen Methoden gilt die Aufmerksamkeit unserer heutigen unbekannten Kostbarkeiten.

Java IO reloaded, reloaded, reloaded...

Eine Programmiersprache, die in der Praxis eingesetzt werden soll, muss Funktionalitäten zur Ein- und Ausgabe insbesondere auch in Bezug zum Dateisystem bereitstellen. Man möchte fast behaupten: je mehr, desto besser. Leser im Alter des Autors können sich eventuell noch an Pascal *[1]* (nicht Turbo-Pascal) und Modula *[2]* von Niklaus Wirth erinnern. Diese Sprachen waren mit sehr spartanischen I/O-APIs ausgestattet, die wahrscheinlich dazu beigetragen haben, dass sie sich nicht in der Praxis durchsetzen konnten. Java besaß von Anfang an eine reichhaltige Bibliothek von I/O-Klassen, zur Zeit von Java 1 im Package `java.io`. Java entwickelte sich weiter, auch wenn manche das Gegenteil behaupteten. Mit Java 1.4 wurde 2002 Java NIO (New I/O) im Package `java.nio` eingeführt, sogar in einem eigenen JSR, dem JSR 51: *New I/O for the Java Platform*. Diesem API

```
• public static BufferedReader newBufferedReader(Path path)
• public static BufferedWriter newBufferedWriter(Path path,OpenOption... options)
• public static List<String> readAllLines(Path path)
• public static Path write(Path path,Iterable<? extends CharSequence> lines,OpenOption... options)
• public static Stream<Path> list(Path dir)
• public static Stream<Path> walk(Path start,int maxDepth,FileVisitOption... options)
• public static Stream<Path> walk(Path start,FileVisitOption... options)
• public static Stream<Path> find(Path start,int maxDepth,BiPredicate<Path,BasicFileAttributes>matcher,
                                  FileVisitOption... options)
• public static Stream<String> lines(Path path,Charset cs)
• public static Stream<String> lines(Path path)
```

Listing 1

```
public void listFilesWithExtension(String extension) {
  try (Stream<Path> walk =
       Files.walk(Paths.get("").toAbsolutePath())) {
    walk.map(f -> f.toString())
        .filter(f -> f.endsWith(extension))
        .forEach(System.out::println);
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Listing 2

	Java	find	Java GraalVM
real	1,076 s	0,609 s	0,836 s
user	2,344 s	0,261 s	0,290 s
sys	0,732 s	0,344 s	0,541 s

Tabelle 1

```
• public static String readString(Path path)
• public static String readString(Path path, Charset cs)
• public static Path writeString(Path path, CharSequence csq, OpenOption... options)
• public static Path writeString(Path path, CharSequence csq, Charset cs, OpenOption... options)
• public static long mismatch(Path path, Path path2)
```

Listing 3

hat Ron Hitchens ein ganzes Buch gewidmet *[3]*. Es umfasst knapp doppelt so viele Seiten wie die Bücher über Pascal und Modula. Mit Java 7 wurden 2011 abermals mit einem JSR, nun dem JSR 203, *More New I/O APIs for the Java Platform („NIO.2")*, die I/O-Funktionalitäten von Java erweitert. Wir gehen davon aus, dass dem Leser dies mehr oder weniger bekannt ist, und konzentrieren uns darauf, was danach geschah. Mit Java 8 wurden Lambda-Ausdrücke und Streams eingeführt, sodass die zehn neuen Methoden der Klasse `Files` eventuell etwas ins Hintertreffen geraten sind. *Listing 1* zeigt eine Übersicht dieser Methoden.

Wir belassen es bei der Angabe der Signaturen, um den Umfang des Artikels nicht zu sprengen, und entwickeln nur ein kleines Beispiel mit der zweiten `walk()`-Methode. Beide `walk()`-Methoden liefern einen Stream zurück und arbeiten daher "lazy". Das Beispiel, alle Dateien mit einer bestimmten Dateinamenserweiterung auszugeben, kann daher wie in *Listing 2* zu sehen realisiert werden.

Die `walk()`-Methode durchsucht den Verzeichnisbaum depth-first und ist durch die Stream-inherente Lazy-Eigenschaft auch für große Verzeichnisbäume geeignet.

Nach unserer Meinung ist dies eine sehr elegante Lösung. Es stellt sich aber die Frage, was uns diese Eleganz in Bezug auf die Laufzeit kostet. Wir haben dies gemessen, wenngleich nicht unter wirklichen Benchmark-Bedingungen. *Tabelle 1* gibt diese Messung wieder. Die Zeiten wurden mit dem Unix-Befehl `time` ermittelt.

Als Unix-Befehl wurde `find . -name *.java` verwendet. Die Ausgaben der drei Alternativen wurden auf `/dev/null` umgeleitet. Wir sehen, dass Java fast doppelt so lange benötigt wie der „find"-Befehl. Die tatsächliche CPU-Benutzung beträgt fast das Zehnfache. Das Executable, das mit der GraalVM durch den Befehl `native-image` erzeugt wurde, entspricht in etwa dem Linux-„find"-Befehl.

Java 11 und 12

Wie bereits angedeutet, entwickelt sich Java mit hoher Geschwindigkeit weiter. Das Einlesen eines Dateiinhalts in einen String beziehungsweise das Schreiben eines Strings in eine Datei sind seit Java 11 mit einem einzigen Methodenaufruf möglich. Das Suchen nach der (ersten) Stelle, an der sich zwei Dateien unterscheiden, gibt es seit Java 12. Die Signaturen der entsprechenden Methoden zeigt *Listing 3*.

Zusammenfassung

Seit Java 7 gibt es die Klasse `Files`, die im Rahmen von NIO.2 entstanden ist. Mit Java 8, 11 und 12 kamen eine Reihe weiterer Methoden hinzu, die verschiedene Aufgaben mit I/O stark vereinfachen. Insbesondere das Lesen einer Datei in einen String beziehungsweise das Schreiben eines Strings in eine Datei mit einem einzigen Befehl sollte in das Repertoire eines Java-Entwicklers gehören.

Referenzen

[1] Kathleen Jensen, Niklaus Wirth. PASCAL User Manual and Report, Springer-Verlag, 1975.
[2] Niklaus Wirth. Programming in MODULA-2, Springer-Verlag, 1982.
[3] Ron Hitchens. Java NIO, O'Reilly, 2002.

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.

Und halbjährlich grüßt das Java Release

Falk Sippach, Orientation in Objects GmbH

Mittlerweile haben wir Java-Entwickler uns an die kurzen Release-Zyklen gewöhnt. Wie geplant wurde im September 2019 das JDK 13 veröffentlicht. Und auch wenn es keine ganz großen Änderungen gibt, lohnt sich doch ein Blick auf die neue Version.

Seit Java 10 können wir uns zweimal im Jahr – im März und September – über neue Java-Releases freuen. Da natürlich in einem halben Jahr nicht so viel in der Entwicklung passieren kann, sind die jeweiligen Feature-Listen zum Glück überschaubar. Aber immerhin können wir jetzt regelmäßig wenige neue Funktionen ausprobieren, und sie werden nicht alle paar Jahre von einem riesigen Funktionsumfang neuer Features erschlagen. Das Java-Team bei Oracle bekommt zudem schnelleres und wertvolleres Feedback zu den neuen Funktionen, von denen sich einige zunächst nur im Preview-Status befinden. Auf dem Weg zum nächsten LTS (Long Term Support) Release werden sie durch Feedback der Nutzer gegebenenfalls noch angepasst. Letztlich hat man auch beim aktuellen LTS-Release (Java 11) die in Version 9 und 10 begonnenen Arbeiten finalisiert, damit es für die nächsten drei Jahre gut dasteht. Im Jahr 2021 erscheint mit Java 17 dann die nächste Version mit längerer Support-Dauer.

Für den produktiven Einsatz seiner Java-Anwendungen kann man zwischen zwei Varianten wählen. Entweder man bleibt für drei Jahre beim aktuellen LTS-Release oder man aktualisiert zweimal im Jahr auf die jeweils für ein halbes Jahr von Oracle mit Updates versorgte Major-Version des OpenJDK. Letzteres darf man frei einsetzen, muss aber nach sechs Monaten auf die nächste Version aktualisieren, um weiterhin freie Security Patches zu erhalten. Bei der LTS-Variante gibt es sowohl kostenpflichtige Angebote (von Oracle und alternativen Herstellern) als auch freie Installationen, wie zum Beispiel von AdoptOpenJDK, Amazon Corretto, Red Hat und anderen.

Im JDK 13 wurden die folgenden Java Enhancement Proposals umgesetzt *[1]*:

- JEP 350: Dynamic CDS Archives
- JEP 351: ZGC: Uncommit Unused Memory
- JEP 353: Reimplement the Legacy Socket API
- JEP 354: Switch Expressions (Preview)
- JEP 355: Text Blocks (Preview)

```
// Switch-Statement mit break
private static String statementMultilabel(int switchArg){
     String str = "not set";
     switch (switchArg){
         case 1,2:
             str = "one or two";
             break;
         case 3:
             str = "three";
             break;
     };
     return str;
}

// Switch-Expression mit yield
private static String expressionBreakWithValue(int switchArg){
     return switch (switchArg){
         case 1, 2: yield "one or two";
         case 3: yield "three";
         default: yield "smaller than one or bigger than three";
     };
}
```

Listing 1

```
javac --release 13 --enable-preview Examples.java
java --enable-preview Examples
```

Listing 2

Dynamic CDS Archives

Bereits in Java 5 wurde Class Data Sharing (CDS) eingeführt. Bis Java 10 waren die geteilten Archive jedoch nur für den Bootstrap-Classloader zugänglich. Das Ziel von CDS ist die Verkürzung der Startzeiten von Java-Anwendungen, indem bestimmte Informationen über Klassen in sogenannten Class-Data-Sharing-Archiven abgelegt werden. Diese Daten können dann zur Laufzeit geladen und auch von mehreren JVMs benutzt werden. Mit Java 10 wurde CDS um Application-Class-Data-Sharing (AppCDS) erweitert. AppCDS ermöglicht dem eingebauten System- und Plattform-Classloader sowie benutzerdefinierten Classloadern, auf die CDS-Archive zuzugreifen. Zum Anlegen der CDS-Archive werden Klassenlisten benötigt, um die zu ladenden Klassen identifizieren zu können. Bisher mussten diese Klassenlisten durch Probeläufe der Anwendung ermittelt werden, um festzustellen, welche Klassen während der Ausführung tatsächlich geladen werden. Seit Java 12 werden Default-CDS-Archives standardmäßig mit dem JDK ausgeliefert, die auf der Klassenliste des JDK basieren.

Dynamic CDS Archives bauen nun darauf auf. Ziel ist es, sich die zusätzlichen Probeläufe der Anwendung zu sparen. Nach der Ausführung einer Anwendung werden nur noch die neu geladenen Anwendungs- und Bibliotheksklassen, die nicht bereits im Default-/Base-Layer-CDS enthalten sind, archiviert. Aktiviert wird das dynamische Archivieren mit Kommandozeilenbefehlen. In einer zukünftigen Erweiterung könnte es vollständig automatisch und transparent ablaufen. Weitere Details finden sich auf der offiziellen Seite des JEP 350 *[2]*.

ZGC kann ungenutzten Speicher wieder freigeben

Den Z Garbage Collector (ZGC) gibt es seit Java 11. Er wurde von Oracle entwickelt und verspricht sehr kurze Pausen beim Aufräumen von Heap-Speichern mit mehreren Terabytes. Bisher gab er den für die Anwendung reservierten Heap-Speicher jedoch nicht wieder frei. Das führte dazu, dass Anwendungen über ihre Lebensdauer hinweg für gewöhnlich weit mehr Speicher verbrauchen als notwendig. Besonders betroffen sind Anwendungen, die in Ressourcen-armen Umgebungen ausgeführt werden. Andere Garbage-Collectoren wie der G1 und Shanandoah unterstützen bereits das Freigeben von ungenutztem Speicher.

ZGC unterteilt den Heap in Z-Pages. Leere Z-Pages werden nach dem Garbage-Collector-Lauf in einem Cache abgelegt, um sie später für die Neuzuweisung wiederzuverwenden. Das Zuweisen und Freigeben neuer Z-Pages ist nämlich sehr kostenintensiv. Der Cache repräsentiert also eine Liste der bereits beanspruchten, aber derzeit ungenutzten Speicherabschnitte. Mit Java 13 soll eine Z-Page im Cache nun nach einem simplen Timeout wieder freigegeben werden. Der Timeout-Value kann per Kommandozeile überschrieben werden. In zukünftigen Versionen könnten zudem noch weitere verfeinerte Entscheidungsmechanismen eingeführt werden. Für mehr Details siehe JEP 351 *[3]*.

Socket APIs wurden überarbeitet

Die java.net.Socket und java.net.ServerSocket APIs und deren zugrunde liegenden Implementierungen sind noch Überbleibsel aus dem JDK 1.0. Zu großen Teilen bestehen sie aus Legacy-Java- und C-Code. Das erschwert die Wart- und Erweiterbarkeit deutlich. Die NioSocketImpl soll die veraltete PlainSocketImpl nun ablösen. NioSocketImpl ist an die bereits vorhandene New-I/O-Implementierung angelehnt und benutzt deren vorhandene Infrastruktur im JDK. Die bisherige Implementierung ist außerdem nicht kompatibel zu den geplanten Erweiterungen der Sprache im Rahmen des Projekts Loom. Die leichtgewichtigen User-Threads (Fiber) werden durch Concurrency-Probleme des Socket API behindert. Für mehr Details siehe JEP 353 *[4]*.

Switch Expressions wurden leicht angepasst

Die bisher beschriebenen Änderungen betreffen die JVM und die Klassenbibliothek. Es gibt allerdings auch einige Syntax-Erweiterungen. So wurden die in Java 12 als Preview eingeführten Switch Expressions nochmals angepasst.

Die Switch Expression ist eine Alternative zu dem ausschweifenden und für Fehler anfälligen Switch Statement. Eine ausführliche Übersicht über die Verwendung findet sich in diversen Artikeln zu Java 12 (zum Beispiel in Java aktuell 04/2019). Die größte Änderung in Java 13 ist das Ersetzen des Schlüsselwortes „break" in der Switch Expression durch „yield". Der Hintergrund ist die bessere Unterscheidbarkeit zwischen Statement (mit möglichem „break") und den Expressions (mit „yield"). Die „yield"-Anweisung verhält sich dabei wie ein „return", verlässt den Switch und liefert das Ergebnis des aktuellen Zweiges zurück.

Die zweite neue Variante mit der Arrow-Syntax funktioniert übrigens weiterhin wie in Java 12 eingeführt. Die Switch Expressions bleiben jedoch vorerst noch ein Preview-Feature, somit könnte es in den nächsten Java-Versionen weitere Anpassungen geben. In *Listing 1* sind zwei Code-Beispiele aufgeführt. Auf der einen Seite sehen wir ein Statement mit „breaks", das aber bereits mehrere Labels pro Zweig als redundanzfreie Fall-Through-Variante verwendet. Im direkten Vergleich folgt eine Switch Expression mit dem neuen Schlüsselwort „yield".

Beim Kompilieren und zum Starten unter JDK 13 müssen die jeweiligen Preview-Flags angegeben werden *(siehe Listing 2)*. Die Build-Tools (Maven, Gradle etc.) bringen natürlich auch entsprechende Konfigurationsschalter mit. Für mehr Details zu den Änderungen an den Switch Expressions, siehe JEP 354 *[5]*.

Raw String Literals kommen als Text Blocks

Ursprünglich schon für Java 12 geplant, wurde der JEP 326 (Raw String Literals) dann doch kurzfristig zurückgezogen. Die Details kann man in der Mailingliste *[6]* nachlesen. Letztlich hatte man sich aufgrund des Feedbacks und der noch nicht bis ins letzte Detail durchdachten Umsetzung zu diesem Schritt entschlossen. Mit Java 13 hat es nun aber der JEP 355 (Text Blocks) als Preview Feature in das Release geschafft. Hier konzentriert man sich zunächst auf einen Teil der ursprünglichen Raw String Literals, nämlich die mehrzeiligen Texte. In vielen Java-Anwendungen wird Code aus anderen Sprachen wie HTML oder SQL verarbeitet. Bisher waren Strings mit mehreren Zeilen weder gut zu lesen noch einfach aufzuschreiben. Für Zeilenumbrüche müssen beispielsweise extra Steuerbefehle (Escapes mit \n) eingesetzt werden. Andere Sprache wie Groovy oder Kotlin bieten längst die Möglichkeit, mehrzeilige Texte zu definieren.

Um die jetzigen Entscheidungen für Text Blocks nachvollziehen zu können, lohnt sich ein Blick auf die ursprüngliche Implementierungsidee der Raw String Literals (JEP 326). Diese speziellen Zeichenketten konnten sich auch über mehrere Zeilen erstrecken und durften zudem keine Escape-Sequenzen interpretieren. Als Begrenzungszeichen konnte man eine beliebige Anzahl von Backticks (`) verwenden. Dadurch hätte man sogar Backticks im Inhalt des Strings verwenden können, solange die Begrenzer immer mindestens ein Zeichen mehr enthalten. Wenn also im Text einzelne

Du wünschst Dir GitHub-Stars
statt Vollpfosten?

Uns geht's genauso.

Cofinpro berät Deutschlands führende Banken
und Asset Manager.
Wir suchen brillante Sturköpfe für unser Dev-Team.

cofinpro.de/karriere

COFINPRO

```
// Ohne Text Blocks
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, Escapes</p>\n" +
              "    </body>\n" +
              "</html>\n";

// Mit Text Blocks
String html = """
              <html>
                  <body>
                      <p>Hello, Text Blocks</p>
                  </body>
              </html>""";
```

Listing 3

Backticks enthalten sind, hätte man als Begrenzer mindestens doppelte Backticks verwenden müssen. Dieser Ansatz wurde jedoch kritisiert. Es könnte bei vielen Entwicklern zu Verwirrung führen, so die Befürchtung.

Bei den Text Blocks hat man sich darum für dreifache Anführungszeichen als einzige Art von Begrenzungszeichen entschlossen. Die öffnenden müssen im Gegensatz zu den schließenden Anführungszeichen in einer eigenen Zeile stehen. Nichtsdestotrotz beginnt der eigentliche Inhalt erst mit der zweiten Zeile, also ohne die scheinbare Leerzeile mit Zeilenumbruch. Das erhöht die Lesbarkeit des Quellcodes, da so die Einrückung der ersten Zeile korrekt im Quelltext dargestellt wird.

Textblöcke können überall da benutzt werden, wo auch reguläre Strings erlaubt sind. Ein einfaches Beispiel zeigt die Unterschiede zwischen traditioneller und neuer Syntax *(siehe Listing 3)*.

Die Textblöcke werden übrigens bereits zur Compile-Zeit ersetzt. Zunächst erfolgt die Umwandlung der Umbrüche in Betriebssystem-unabhängige Line-Feeds. Dann werden unnötige Whitespaces, die aufgrund der Code-Formatierung entstanden sind, entfernt. Zuletzt werden noch vorhandene Escape-Sequenzen aufgelöst. Nach dem Kompilieren kann man darum nicht mehr herausfinden, wie der String ursprünglich definiert war. Für mehr Details siehe JEP 355 *[7]*.

Ausblick

Die in diesem Artikel angesprochenen Änderungen waren durch JEPs (Java Enhancement Proposals) beschrieben. Weitere Anpassungen direkt an der Klassenbibliothek kann man sich entweder über den JDK API Diff Report Generator *[8]* oder den Java Almanac *[9]* auflisten lassen. Dort entdeckt man zum Beispiel, dass das alte Doclet API unter com.sun.javadoc entfernt wurde *[10]*. Sonst gab es im Gegensatz zu den letzten Releases keine weiteren nennenswerten Änderungen.

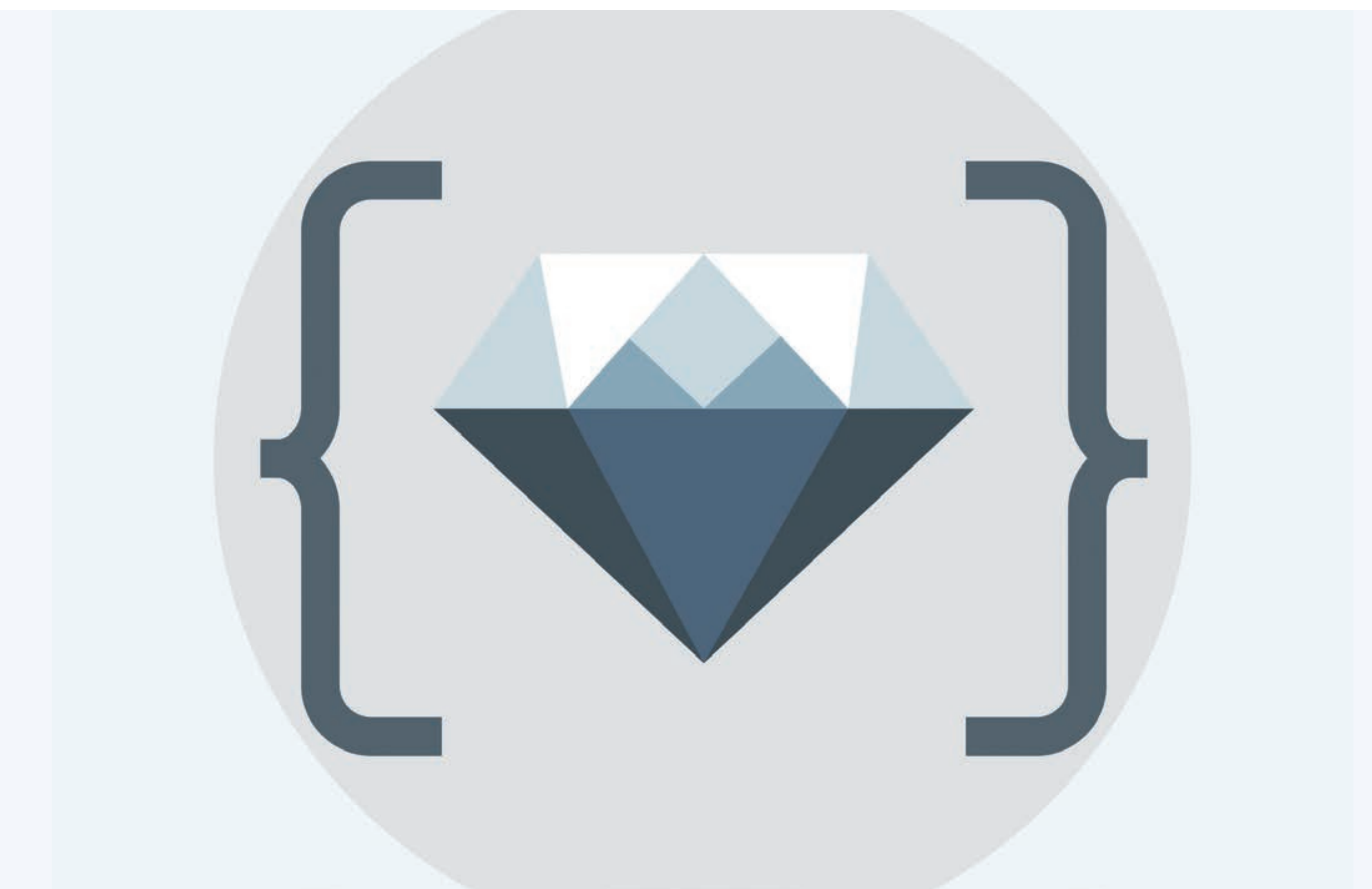
Während Java 13 gerade final erschienen ist und zum Ausprobieren heruntergeladen werden kann *[11]*, laufen auch schon die Arbeiten an der nächsten Version *[12]*. Zum Zeitpunkt der Entstehung dieses Artikels war nur der JEP 352 (Non-Volatile Mapped Byte Buffers) eingeplant. Gegebenenfalls werden die Previews von Switch Expressions und Text Blocks finalisiert. Außerdem könnte im Rahmen des JEP 343 ein neues Tool zum Verpacken von in sich abgeschlossenen Java-Anwendungen in Erscheinung treten. So ungewiss der Blick in die Glaskugel im Moment noch scheinen mag, so schnell wird das nächste Release letztlich da sein. Denn schon im März 2020 grüßt uns Java 14. Wir dürfen also gespannt sein und freuen uns auf die nächsten Neuerungen.

Referenzen:

[1] JDK 13 Projektseite: *https://openjdk.java.net/projects/jdk/13/*
[2] JEP 350: *https://openjdk.java.net/jeps/350*
[3] JEP 351: *https://openjdk.java.net/jeps/351*
[4] JEP 353: *https://openjdk.java.net/jeps/353*
[5] JEP 354: *https://openjdk.java.net/jeps/354*
[6] *https://mail.openjdk.java.net/pipermail/jdk-dev/2018-December/002402.html*
[7] JEP 355: *https://openjdk.java.net/jeps/355*
[8] API-Änderungen: *https://github.com/AdoptOpenJDK/jdk-api-diff*
[9] Java Almanac: *http://download.eclipselab.org/jdkdiff/V12/V13/index.html*
[10] *https://bugs.openjdk.java.net/browse/JDK-8215584*
[11] JDK 13 Download: *http://jdk.java.net/13/*
[12] JDK 14 Projektseite: *https://openjdk.java.net/projects/jdk/14/*

Falk Sippach
Orientation in Objects GmbH
falk.sippach@oio.de

Falk Sippach hat zwanzig Jahre Erfahrung mit Java und ist bei der Mannheimer Firma OIO Orientation in Objects GmbH als Trainer, Softwareentwickler und -architekt tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit anderen die örtliche Java User Group. Falk twittert unter @sippsack.



Die „Definition of Clean" – Wie sauberer Code zu besseren Ergebnissen in Software-Entwicklungsprojekten führt

Holger Tiemeyer, PENTASYS AG

Ob bei der Übernahme und Weiterentwicklung von Projekten, bei Wechseln zwischen den Entwicklerteams oder bei einfachen bis komplexen Wartungsarbeiten: Wenn der Softwarecode oder auch die inneren Strukturen unsauber sind, stehen die Projektbeteiligten vor einer frustrierenden sowie zeit- und kräfteraubenden Aufgabe. Denn schlecht geschriebener Code quält uns; ganz einfach auch deshalb, weil es eines hohen Maßes an technischer Lösungskompetenz, fachlicher Expertise und letztendlich Konzentration des Einzelnen bedarf, sich in schlecht geschriebene Code-Basen einzulesen, um die Absicht ihrer Schöpfer nachzuvollziehen. An bestimmten Punkten des Projekts kann es dabei durchaus vorkommen, dass die Projektbeteiligten dann genauso viel oder sogar mehr ihrer Lösungskompetenz dafür einsetzen müssen, die vorhandene Software zu rekonstruieren, anstatt neue Programmteile oder Features zu entwickeln.

Sauberer Code ist daher ein wichtiges Thema – sowohl aus fachlichem, technischem als auch aus geschäftlichem Blickwinkel. Der folgende Artikel beschreibt, aus welchen Gründen die Entwicklerszene seit Jahren zunehmend über sauberen Code spricht. Er legt dar, welche Faktoren in der Praxis dazu führen können, dass unsauberer Code entsteht, wie dem begegnet werden kann und welche grundlegenden Regeln dabei helfen, die Entwicklung von sauberem Code in einen nachvollziehbaren Prozess zu überführen.

Was bedeutet „sauberer Code"?

Was steckt nun genau hinter der Bezeichnung „sauberer Code"? In seinem Grundlagenwerk *Clean Code: A Handbook of Agile Software Craftsmanship* aus dem Jahr 2009 hat Robert C. Martin drei grundlegende Kriterien hierfür eingeführt:

Code ist dann sauber geschrieben, wenn er:

1. verständlich,
2. änderbar und
3. testbar ist.

Dies ist weniger eine formale als eine funktionale Definition. Denn bei Clean Code geht es nicht darum, besonders penibel oder akkurat zu arbeiten, sondern nachvollziehbare und einfach zu wartende Software herzustellen.

Erfüllt der Quelltext ebendiese Kriterien, hat das viele Vorteile: Die Verständlichkeit erleichtert den Einstieg in die Funktionsweise eines Programms bei einer neuen Teambesetzung und fördert den Austausch von Wissen. Die Änderbarkeit unterstützt das Refactoring und die fachliche Erweiterung der Software. Und schließlich ist Testbarkeit eine wichtige Voraussetzung für die (Ausfall-)Sicherheit und funktionale Güte der Software.

Warum ist sauberer Code wichtig?

Sauberer Code erhöht die Wartbarkeit eines Softwaresystems. Sind die oben genannten Kriterien für sauberen Code erfüllt, so lassen sich einzelne Teile eines Softwaresystems mit geringerem Aufwand austauschen oder neue Teile effektiv integrieren.

Es lässt sich häufig beobachten, dass der Code in Open-Source-Projekten sauberer ist als in kommerzieller Software. Aber warum ist das so? Eine mögliche Erklärung sind die unterschiedlichen Rahmenbedingungen, unter denen die Entwickler arbeiten, sowie die zugrunde liegenden Wertesysteme beider Welten.

In der Open-Source-Szene erfahren die beteiligten Programmierer eine hohe individuelle Sichtbarkeit und Transparenz: Wer mitmacht, möchte vor der „ganzen Welt" gut dastehen und sich möglichst keine Blöße geben. Zudem haben viele Außenstehende Einblick in den Code, wodurch bereits vorhandener Code hohe Standards erfüllt. Und: Die Qualität des Sourcecodes wird in Open-Source-Projekten strikt überwacht. So wird schlecht geschriebener Code von der engagierten Community zum Teil durch sauberen ersetzt.

In kommerziellen Projekten fehlen durch hierarchische und kulturelle Instanzen oftmals diejenigen Vorbilder und Kontrollmechanismen, die im Open-Source-Bereich die Community bereitstellt. Ganz im Sinne der „Broken-Windows-Theorie" ziehen sich vorhandene Aspekte (Fehler, schnelle Lösungen, „das wurde ja schon immer so gemacht" etc.) weiter durch die gesamte Software oder verschlimmern sich sogar im Lauf der Zeit. Projektverantwortliche legen ihren Fokus auch eher auf den äußeren Faktoren wie Budget, Zeit, Umfang, Funktionsfähigkeit oder Features als auf der inneren Qualität der Software.

In Projekten, die eine besonders niedrige Codequalität aufweisen, kommen meist noch weitere Probleme hinzu. Unter anderem: Kommunikationsprobleme, schlechte Infrastruktur, Workarounds statt tatsächlicher Behebung von Mängeln, Verdopplungen im Code, überkomplizierte Implementierung, Spaghetticode (also verworrene Strukturen), Vorhandensein von totem Code, ungeschickte Namensgebung und so weiter. Die Reihe ließe sich noch um einiges fortsetzen. Die Gründe, warum in Projekten unsauberer Code auftritt, sind jedenfalls vielfältig. Durch eine veränderte Kultur und Praxis sowie Selbstdisziplin lässt sich jedoch gegensteuern.

Letztendlich wirken sich all diese Faktoren auf die Kosten der Softwareentwicklung aus. Die Kosten für ein Softwaresystem setzen sich unter anderem aus den Kosten der initialen Entwicklung des Systems und den Kosten für die Wartung des Systems zusammen, wobei gilt: „The cost of maintenance is usually much higher than the cost of initial development" (Kent Beck in: Implementation Patterns, Addison Wesley, 2007, Kapitel 4).

Clean Code ist die Basis für geringere technische Schulden

In einem Softwareprojekt kann die Erzeugung von schlechtem Code bewusst oder unbewusst in Kauf genommen werden. Fällt zu Beginn des Projekts beispielsweise die Entscheidung, so schnell wie möglich lieferfähig zu sein, können im späteren Verlauf Folgeprobleme auftreten: Bugs, Ausfall von Funktionalität, ein hoher Bedarf an Refakturierung und so weiter. Dabei findet eine Anhäufung von Aufwänden statt (zum Beispiel „wir kümmern uns später darum"), damit die Nachhaltigkeit und Wartbarkeit im späteren Verlauf sichergestellt werden kann. Diese Anhäufung wird auch als das Sammeln von „technischen Schulden" bezeichnet. Diese Metapher, angelehnt an das Sammeln ökonomischer Schulden, beschreibt die technische und geschäftliche Nachhaltigkeit in einem Softwareprojekt. Die Kosten für ihre Behebungen werden, um in der Metapher zu bleiben, „Zinszahlungen" genannt.

Im Regelfall ist davon auszugehen, dass die Zinszahlungen ab einer bestimmten Höhe der Schulden die Mehrwerte der Lösung übersteigen. Die technischen Schulden sollten daher also so gering wie möglich bleiben. Es gibt einige wenige Fälle, bei denen es sich lohnt, technische Schulden bewusst in Kauf zu nehmen: Zum Beispiel, wenn klar ist, dass eine Softwarelösung ab einem bestimmten Zeitpunkt nicht mehr gebraucht oder ersetzt wird. Dennoch sollte auch solch eine Lösung dazu herangezogen werden, die Regeln von Clean Code zu trainieren. Denn Clean Code gilt als eine Möglichkeit, die „technischen Schulden" eines Projektes zu verringern oder zu vermeiden.

Wenn es speziell um Softwareentwicklung geht, sind die technischen Schulden für schlechten Code in den „Währungen" Aufmerksamkeit und Zeit hinterlegt. Unsauber geschriebener Code kostet die betei-

ligten Softwarenentwickler viele Ressourcen, die sie für Wieder- und Umlernen („Relearning") des vorhandenen Quelltextes aufwenden müssen. Üblicherweise „scannen" sie diesen zunächst auf der Oberfläche, um seine Struktur zu erfassen – analog zum Lesen von Texten in natürlicher Sprache. Anschließend können sie in den Abschnitt „springen", der für die momentane Aufgabe relevant ist.

Ist eine solche Struktur hingegen nicht vorhanden oder viel zu unübersichtlich, braucht es viel Zeit und Konzentration, um sie zu rekonstruieren. Dem Product Owner oder Auftraggeber sind solche (scheinbar) unproduktiven Phasen und Mehrkosten jedoch nur schwer zu vermitteln. Eine Alternative ist, dass das Entwicklerteam eben „auf Sicht fährt" und ohne eine klare Übersicht weiteragiert. Daraus resultiert allerdings die Gefahr, dass sich nicht mehr genau abschätzen lässt, welche Auswirkungen eine Änderung an einer Stelle auf andere Bereiche und Funktionalitäten des Programms hat. Treten dann Fehler auf, schießt der technische Zins schnell in die Höhe.

Mindset ist für die Softwarequalität entscheidend

Clean Code ist nicht einfach ein starres Regelwerk, sondern eine Bewegung. Diese ist völlig unabhängig von einzelnen Plattformen und Programmiersprachen – es geht nämlich um das Wertesystem hinter der Programmierung. Laut der Initiative „Clean Code Developer" *[1]* sind die folgenden vier Werte dafür konstituierend:

- Evolvierbarkeit (Änderbarkeit)
- Korrektheit
- Produktionseffizienz (Verständlichkeit)
- Kontinuierliche Verbesserung

Entwicklerteams sollten es zu ihrer inneren Haltung machen, sich dieser Faktoren bewusst zu werden und ihre Umsetzung im Code anstreben. Das Wertesystem stellt eine Orientierung bereit, die unabhängig ist vom Druck aus dem Projekt, Vorgaben der Kunden oder praktischen Einschränkungen.

Neben den Werten lautet ein weiteres Stichwort „Empathie". Auch hier geht es um ein generelles Umdenken beim Programmieren. Zumeist sind Entwickler damit beschäftigt, Code so zu schreiben, dass er die gewünschten Funktionen bereitstellt. Allerdings wird nicht berücksichtigt, ob der Code für einen menschlichen Akteur noch verständlich ist. Hier kann sich eine problematische „Betriebsblindheit" einstellen, welche die Produktion schlechten Codes begünstigt.

Für Entwickler ergibt sich daraus die Herausforderung, ihr Programm aus dem Blickwinkel einer nicht „eingeweihten" Person zu betrachten und zu überprüfen. Ein solches Hin- und Herschalten zwischen den Perspektiven mag zunächst nicht einfach sein – letztendlich geht es bei Clean Code aber vor allem darum, anderen Beteiligten das Verständnis so leicht wie möglich zu machen.

Pfadfinderregel

Eine Pfadfinderregel besagt, dass der Campingplatz immer sauberer verlassen werden sollte, als er vorgefunden wurde. Diese Regel lässt sich auf die Entwicklung von Sourcecode übertragen. Jedes Mal, wenn ein Softwareentwickler unsauberen Code während seiner Arbeit entdeckt, sollte er darüber nachdenken, wie er ihn aufräumen könnte. Diese Regel ist jedoch mit Vorsicht anzuwenden. Allzu oft kann eine einzelne Zeile Code, die eventuell falsch platziert ist, dazu führen, dass ein vollständiges Modul, das aus mehreren Tausend Zeilen Code besteht, vollkommen restrukturiert werden muss. Daher sollte eine solche Aufräumaktion immer mit den Kollegen besprochen werden.

Wichtige Regeln für sauberen Code

Das Clean-Code-Paradigma umfasst den gesamten Software-Lebenszyklus – vom Design, über die Entwicklung und das Testen bis hin zur Implementierung der Softwarelösung. Unter anderem im Buch von Robert C. Martin oder auch in diversen Clean Code Cheat Sheets *[2]* finden sich für jeden Aspekt der Entwicklung entsprechende Regeln. Insofern wird auch deutlich, dass die – insgesamt sicher nicht leichte – Einführung von Clean-Code-Regeln und -Prinzipien ein absolutes Grundlagenthema ist, das jeden Entwickler betrifft.

Für jede Spezialisierung und auch für die Besonderheiten mancher Programmiersprachen finden sich spezifische Regeln. Hier wurde daher eine Auswahl getroffen, die sich als Grundlage für die Einführung von Clean Code in fast allen Software-Projekten heranziehen lässt.

Auf übergeordneter Ebene sollten zunächst fünf Entwurfsprinzipien zum Einsatz kommen, die hinter dem Akronym „SOLID" stehen und die für ein reibungsloses Zusammenspiel der verschiedenen Code-Einheiten stets Anwendung finden sollten:

1. **Single Responsible Principle (SRP)** – Das Prinzip der eindeutigen Verantwortlichkeit: Es darf nur genau einen Grund geben, eine Klasse zu ändern.
2. **Open Closed Principle (OCP)** – Das Prinzip der gleichzeitigen Offenheit und Verschlossenheit: Module sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein.
3. **Liskov Substitution Principle (LSP)** – Das Liskovsche Substitutionsprinzip beziehungsweise Ersetzungsprinzip: „Sei q(x) eine beweisbare Eigenschaft von Objekten x des Typs T. Dann soll q(y) für Objekte y des Typs S wahr sein, wobei S ein Untertyp von T ist."
4. **Interface Segregation Principle (ISP)** – Das Prinzip der Schnittstellenaufteilung: Eine Schnittstelle soll in mehrere Schnittstellen aufgeteilt werden, falls es notwendig ist, dass implementierende Klassen unnötige Methoden haben.
5. **Dependency Inversion Principle (DIP)** – Das Abhängigkeits-Umkehr-Prinzip: Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

Diese Design-Regeln sind sicherlich sehr abstrakt, doch ihre konsequente Anwendung erleichtert die Übersicht der strukturellen Beschaffenheit eines Softwareprogramms sehr. Ebenso können sie vor unangenehmen „Überraschungen" logischer Natur schützen, wenn einzelne Module beziehungsweise Code-Einheiten geändert werden und sich plötzlich unerwünschte Nebeneffekte einstellen.

Unter anderem sind die folgenden Regeln für die tägliche Entwicklungsarbeit im Code zu beachten:

- Namensgebung – Variablen, Funktionen und Klassen sind mit selbsterklärenden oder zumindest aussagekräftigen Namen zu versehen. Wichtig ist auch, dass die Namen in der natürlichen Sprache aussprechbar sind, also nicht aus Buchstaben- oder Zahlenkürzeln bestehen. Somit ist es einfacher, sie sich zu merken und über sie zu sprechen.
- Wiederholungen – Es gilt die Devise „DRY – Don't repeat yourself". Code sollte stets auf Wiederholungen geprüft und diese gegebenenfalls gelöscht werden.
- Umfang von Funktionen – Funktionen sollten so klein wie nur möglich sein, also nur eine „Sache machen". Robert C. Martin: „Functions should do one thing, they should do it well, they should do it only." *[3]*
- Kommentare – Es ist ein Fehlschluss zu glauben, dass Kommentare in hoher Quantität automatisch zu mehr Verständlichkeit führen. Oft ist sogar das Gegenteil der Fall. Kommentare sind spärlich einzusetzen, denn nach den Maßgaben des Clean Code sollte alles Wesentliche bereits aus dem Code selbst hervorgehen.
- Einfachheit – Zu guter Letzt lautet eine der wohl wichtigsten Regeln: „Keep it Simple, Stupid!" (KISS). Dies bedeutet unter anderem, Funktionalitäten stets eng auf den vorliegenden Anwendungsfall bezogen zu entwickeln, anstatt den Code mit generischen oder schlichtweg überflüssigen Features zu überfrachten.

Fazit: Ein Umlernen lohnt sich!

Zweifelsohne bedarf es bei der Einführung des Clean-Code-Paradigmas zumindest zu Beginn einer größeren Anstrengung. Ebenso mag es verführerisch wirken, dem Ansatz „Quick and Dirty" nachzugeben. Denn dieser spart zu Beginn tatsächlich einiges an Zeit und Nerven, die sonst für den Kulturwandel und die Einhaltung der Clean-Code-Regeln anfallen würden. Auf lange Sicht gesehen lohnt sich eine disziplinierte Herangehensweise, um sauberen Code zu schreiben, so gut wie immer. Was „Quick and Dirty" anfangs einspart, schlägt im Verlauf des Projekts mit stetig steigenden technischen Zinsen zu Buche.

Nicht zu vergessen sind auch die qualitativen und menschlichen Variablen: Mit einem sauberen Code können Entwicklerteams am Ende auch einfach besser arbeiten. Sie sind seltener mit vermeidbaren Problemen konfrontiert. Sie haben seltener das Gefühl, als „Löschtrupp" anrücken zu müssen, weil es mal wieder – vermeidbar – brennt. Sie werden seltener bei ihrer Arbeit unterbrochen und können sich konzentrierter und mit mehr Zeit ihren eigentlichen Aufgaben widmen. Nicht zuletzt finden sie sich seltener in einer frustrierten, dafür öfter in einer motivierten Stimmung wieder.

Viele Gründe also, das Thema Clean Code heute noch in den Programmieralltag einzuführen. Dabei gilt: Sauberer Code, der sich einfach lesen lässt, ist unter Umständen sehr herausfordernd zu schreiben.

Referenzen

[1] *https://clean-code-developer.de/das-wertesystem/*
[2] Siehe z.B. *https://www.bbv.ch/images/bbv/pdf/downloads/V2_Clean_Code_V3.pdf*
[3] Clean Code - A Handbook of Agile Software Craftmanship, Robert C. Martin, 2009, Prentice Hall, Kapitel 3, Seite 35

Holger Tiemeyer
PENTASYS AG
holger.tiemeyer@pentasys.de

Holger Tiemeyer hat an der Universität Hamburg Informatik mit Nebenfach Psychologie studiert. Er realisiert in der Rolle als Senior Software Architekt unterschiedliche Projekte in verschiedenen Enterprise-Kontexten bei der PENTASYS AG. Seine Hauptaufgabe liegt in dem Entwurf und der Umsetzung von komplexen Softwarearchitekturen. Er ist Mitglied im iSAQB e.V. und leitet dort die Arbeitsgruppe Hochschulen.

GitOps mit Helm und Kubernetes

Bernd Stübinger und Florian Heubeck, Java User Group Ingolstadt e.V.

DevOps, also das Entwickeln und Betreiben von Software als Teamaufgabe, verlangt entweder nach Spezialisten aus beiden Bereichen oder interdisziplinären Experten, die beides abdecken. Da dies in der Realität schwer zu erreichen ist, erfordert echtes DevOps Werkzeuge, die einen weitestgehend automatischen Betrieb ermöglichen und die Laufzeitumgebung einem Entwicklungsteam zugänglich machen. Hier hilft das Konzept GitOps, dessen Einsatz wir auf Basis von Helm und Kubernetes erläutern wollen.

DevOps – ein Begriff, der sich inzwischen bis in die letzten Winkel des IT-Managements verbreitet hat. Doch was bedeutet DevOps für uns Entwickler? Entwicklung und Betrieb aus einer Hand! Um aktuellen Trends zu folgen, läuft Software heutzutage natürlich in der Cloud, und somit landet man schlussendlich bei einem Team, das eine Menge unterschiedlicher Dinge in einen Kubernetes-Cluster schiebt und auf das Beste hofft.

Wer es ein wenig organisierter mag, fängt an, seine Software in Helm-Charts zu bündeln. Damit hat man dann zumindest eine Menge zusammengehöriger Dinge, die versioniert und einfacher überblickt werden können. Helm allein ermöglicht allerdings auch noch kein kontinuierliches und zuverlässiges Ausspielen konkreter Softwarestände. Was tun, wenn nur bestimmte Ressourcen erfolgreich

aktualisiert werden konnten? Was, wenn sich die Konfiguration ändert? Und wie stellt man überhaupt fest, welche Version und Konfiguration aktuell läuft? Was typischerweise auch weniger gut funktioniert, ist das Wiederaufsetzen nach einem größeren Ausfall. Oder: Wie lange dauert es, bis ein leerer Cluster wieder produktiv wird?

CI != CD

Aus der klassischen Softwareentwicklung sind wir es gewohnt, Build-Pipelines zu instrumentalisieren. Diese lösen schließlich bereits das Problem der Continuous Integration. Integriert man dort auch noch Delivery und Deployment, besteht die Gefahr, dass diese Pipeline aufgrund diverser Anforderungen und verschiedener Unwägbarkeiten schnell eine Komplexität erreicht, die weder überschau- noch wartbar ist.

Das Bauen eines Artefaktes ist ein geradliniger Ablauf, mit zwei möglichen Ergebnissen: einer neuen Version des zu bauenden Artefaktes oder eben einem Fehlschlag. Viele Anbieter stellen Build Services zur Verfügung, die unkompliziert an das eigene Git Repository angebunden werden können und diese Aufgabe zuverlässig erledigen.

Die Komplexität von Continuous Delivery ist jedoch eine ganz andere als die des Builds. Fehler müssen behandelt, Zustände gesichert und gegebenenfalls wiederhergestellt werden. Hinzu kommt, dass sich Deployments womöglich je nach Stage unterschiedlich verhalten müssen. Die Entwicklungsumgebung kann anders behandelt werden als die Produktion, und jede weitere Ausprägung ist denkbar. Zusätzlich wollen noch eventuelle Zugangsdaten hinterlegt werden, und man will oft lieber nicht so genau darüber nachdenken, was eigentlich zu tun wäre, falls sich einmal der komplette Cluster spontan verabschiedet.

```
helm repo add fluxcd https://fluxcd.github.io/flux
helm upgrade -i flux --namespace flux \
  --set helmOperator.create=true \
  --set helmOperator.createCRD=true \
  --set syncGarbageCollection.enabled=true \
  --set git.url=<GitOps Repo URI> \
  --set git.branch=master fluxcd/flux
```

Listing 1: Installation von Flux via Helm

Kommen wir zum Punkt: Um einen vernünftigen Betrieb gewährleisten zu können, bedarf es einer zuverlässigen Lieferkette und einfacher Konfigurationsmöglichkeiten aus Sicht des Entwicklungsteams.

Die Lösung für diese Problemstellung heißt GitOps. Ein Begriff, der zuerst von Weaveworks geprägt wurde *[1]* und im Prinzip die konsequente Fortführung des Konzepts von „Infrastructure as Code" darstellt, das sich besonders im Cloud-Umfeld etabliert hat: Der Entwickler beschreibt den gewünschten Zustand der Infrastruktur deklarativ und spezialisierte Software, wie zum Beispiel Ansible oder Terraform, kümmert sich um die zuverlässige Bereitstellung. Warum sollte dieser Ansatz nicht auch für die auf dieser Infrastruktur laufende Software möglich sein – insbesondere wenn Kubernetes ohnehin bereits alle Ressourcen deklarativ beschreibt.

Mit GitOps werden neben dem Sourcecode nun auch die Laufzeitkonfiguration und der gewünschte Zustand des Clusters in Git hinterlegt und damit transparent, automatisch versioniert und vor allem nachvollziehbar. Nachvollziehbar sowohl für Menschen als auch für Tools, die darauf aufbauend automatisch den Soll- mit dem Ist-Zustand abgleichen und gegebenenfalls korrigierend eingreifen können.

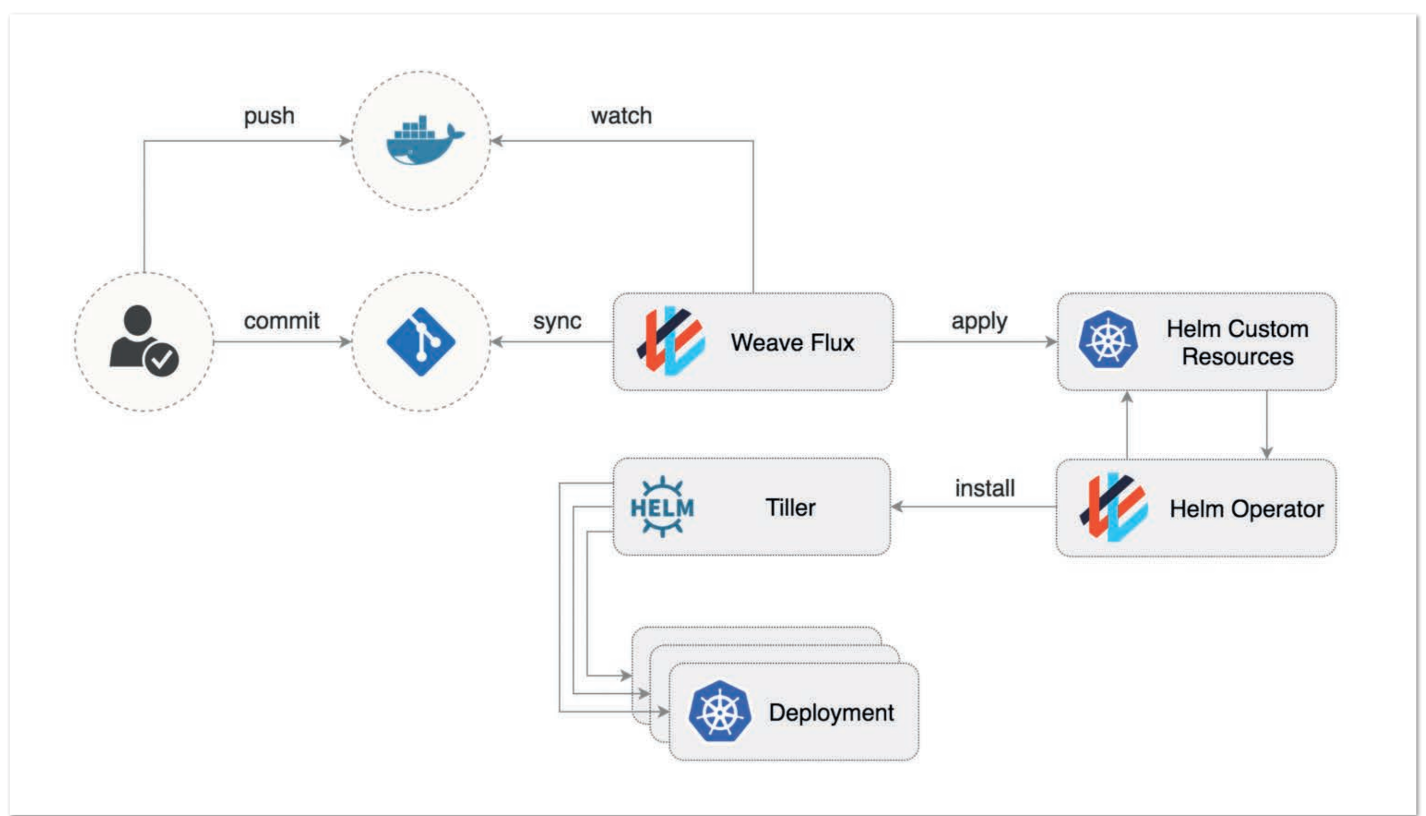


Abbildung 1: Flux mit Helm-Operator (© Weaveworks [2])

```
apiVersion: flux.weave.works/v1beta1
kind: HelmRelease
metadata:
  name: redis
  annotations:
    flux.weave.works/automated: "false"
spec:
  releaseName: redis
  chart:
    repository: https://kubernetes-charts.storage.googleapis.com/
    name: redis-ha
    version: 3.6.1
  values:
    exporter:
      enabled: true
    redis:
      masterGroupName: cacheMaster
```

Listing 2: Beispiel eines statischen HelmRelease

```
apiVersion: flux.weave.works/v1beta1
kind: HelmRelease
metadata:
  name: demo-service
  annotations:
    flux.weave.works/tag.chart-image: semver:~1.2
    flux.weave.works/automated: "true"
spec:
  values:
    image:
      repository: eu.gcr.io/project/demo-service
      tag: 1.2.0
    configuration:
      config_map_key_1: "config_map_value_1"
```

Listing 3: Automatisches Upgrade anhand semantischer Versionierung

Kubernetes arbeitet deklarativ

Kubernetes an sich ist ein großartiges System, jedoch ist es nicht einfach, den Zustand und die Version aller Ressourcen im Blick zu behalten. Verschiedene Stages mit unterschiedlichen Versionen und Konfigurationen der Fachanwendungen erschweren den Überblick weiter. Helm ist bereits eine große Hilfe, indem alle Ressourcen einer Anwendung zu einem sogenannten Chart geschnürt und gemeinsam verwaltet werden. Die Manifeste der Kubernetes-Ressourcen sind statisch. Helm fügt dank seiner Templating-Engine die Möglichkeit dynamischer Konfiguration hinzu.

Eine große Anzahl vom Helm-Projekt gepflegter Charts befinden sich im dortigen Git Repository und sofern eigene Charts existieren, werden diese vermutlich bereits ebenfalls in Git verwaltet. Wäre es nicht traumhaft, wenn das in Git Beschriebene auch dem tatsächlichen Zustand des Systems entspräche und Änderungen am System in Git ersichtlich würden? Das ist GitOps!

Flux: Der Delivery-Operator

Die Firma Weaveworks pflegt den Open Source Kubernetes Controller „Flux“, der als Continuous Delivery Operator dient. Flux synchronisiert ein Git Repository mit dem eigenen Kubernetes Cluster. Ein Zugriff auf den Cluster von außen ist somit nicht nötig, was dessen Absicherung erleichtert. Die Installation von Flux ist dank des bereitgestellten Helm Chart denkbar einfach *(siehe Listing 1)*. Für eine Installation via Terraform eignet sich das Terraform HelmRelease gleichermaßen.

Flux erzeugt beim ersten Start ein Schlüsselpaar, mithilfe dessen man ihm den Zugriff auf das Git Repository gewähren kann. Den dafür nötigen öffentlichen Schlüssel erhält man entweder aus den Logs oder nach Installation des zugehörigen Kommandozeilen-Tools per „fluxctl identity“. Durch regelmäßiges Polling (der Standard sind fünf Minuten) überwacht Flux das konfigurierte Repository und wendet alle gefundenen Kubernetes-Manifeste im Cluster an. Neben den zur Fachanwendung gehörenden können dies auch weitere allgemeine Ressourcen-Manifeste wie Istio Konfiguration oder Grafana Dashboards sein. Zur einfachen Nachvollziehbarkeit markiert Flux den zuletzt angewandten Commit mit einem Tag. Sollte der tatsächliche Stand in Kubernetes zum Beispiel durch manuelle Eingriffe abweichen, würde Flux diesen bei der nächsten Synchronisation wieder überschreiben.

Integration mit Helm

Richtig mächtig wird das Konstrukt, wenn man Helm hinzunimmt *(siehe Abbildung 1)*. Flux bietet einen optionalen Helm Operator an (bereitgestellt und konfiguriert durch die `helmOperator.*`-Values bei der Installation), der Hand in Hand mit dem Flux Operator zusammenarbeitet. Die durch Flux installierte Custom Resource Definition (CRD) „HelmRelease“ ermöglicht die deklarative Installation eines Helm Chart. Referenzierte Charts können im GitOps Repository oder jedem anderen Git oder Helm Repository liegen. Ein HelmRelease wird, wie die anderen Kubernetes-Manifeste des GitOps Repository, durch Flux angelegt. Der Helm Operator installiert dann das Helm Chart. Values, die ein Chart konfigurieren, sind Teil des HelmRelease, womit die komplette Konfiguration der Anwendung an einer Stelle in Git gebündelt *(siehe Listing 2)* und nicht auf mehrere Tools verteilt ist.

Deployment-Automatisierung

Gehen wir auf eines der nützlichsten Features von Flux ein, das automatische Update von Helm Charts. Interessant sind dabei die Annotationen aus *Listing 3*:

- flux.weave.works/tag.chart-image: semver:~1.2
- flux.weave.works/automated: "true"

Erstere beschreibt ein Pattern für Image Tags, in unserem Fall möchten wir alle 1.2.x-Releases gemäß semantischer Versionierung verwenden. Die zweite Annotation weist Flux an, das Deploy-

```
bb2c260 (origin/master) Auto-release eu.gcr.io/project/demo-service:1.2.1
diff --git a/demo-service.yaml b/demo-service.yaml
index 8f32a81..be28faf 100644
--- a/demo-service.yaml
+++ b/demo-service.yaml
@@ -16,7 +16,7 @@ spec:
   values:
     image:
       repository: eu.gcr.io/project/demo-service
-      tag: 1.2.0
+      tag: 1.2.1
     container:
       replicaCount: 1
```

Listing 4: Von Flux erzeugter Git-Commit für das Update von 1.2.0 auf 1.2.1

ment zu automatisieren und neue, das heißt passende Versionen selbstständig auszuspielen. Alternativ zur semantischen Versionierung versteht Flux auch Glob-Muster *[3]* und reguläre Ausdrücke.

Diverse Varianten für die Angabe von Docker Image und Tag, die in vielen Helm Charts genutzt werden, erkennt Flux per Konvention (siehe *[4]*) und überwacht dann ebenfalls die Docker Registry. Wird dort ein neues Image mit entsprechendem Tag gefunden, erstellt Flux einen Commit, der das Image Tag im Manifest des HelmRelease ändert *(siehe Listing 4)*, und aktualisiert das HelmRelease im Kubernetes Cluster. Der Helm Operator führt daraufhin ein Upgrade des Helm Chart durch: Continuous Delivery.

Ein beispielhaftes Helm Chart für eine Fachanwendung samt zugehöriger Konfiguration könnte nun wie im *Listing 5* aussehen und vollständig durch das HelmRelease konfiguriert werden. Die ConfigMap sowie das SealedSecret (siehe weiter unten) werden komplett aus den Values des Chart *(siehe Listing 6)* gefüllt und sind im Manifest des Flux HelmRelease hinterlegt. Änderungen an der Konfiguration werden per Pull Request im Git Repository durchgeführt und unterliegen damit den gleichen Review-Prinzipien wie der Quellcode.

Geheimnisse in Git

Unsere gesamte Laufzeitumgebung ist nun in Git beschrieben. Etwas fehlt jedoch noch für ein rundes Konzept. Einige Konfigurationselemente wie Zugangsdaten und Passwörter können nicht im Klartext veröffentlicht werden. Hier eilt ein weiterer Kubernetes Controller zu Hilfe: SealedSecrets *[5] (siehe Abbildung 2)*.

Das SealedSecret ist ebenfalls eine CRD und enthält die Daten eines normalen Kubernetes Secret, jedoch asynchron verschlüsselt. Dazu erzeugt der SealedSecrets Controller ebenfalls ein Schlüsselpaar, dessen öffentlicher Teil Secrets mit dem eigenen CLI „kubeseal" verschlüsselt *(siehe Listing 7)*. Der private Teil verbleibt im Cluster und ist auch nur dort bekannt. Dadurch kann das SealedSecret bedenkenlos eingecheckt werden, die Entschlüsselung in ein normales Secret ist nur dem Controller im Kubernetes-Cluster möglich. Der private Teil des Schlüssels ist so das einzige außerhalb von Git zu hütende Geheimnis.

Deployment-Update bei Änderung der Konfiguration

Ein umfassendes Helm Chart wie im obigen Beispiel mit Anwendungskonfiguration in einer ConfigMap, bietet ein Stolperfalle: Helm wendet bei einem Upgrade nur die geänderten Ressourcen an, also

```
demo-service
+-- Chart.yaml
+-- templates
|    +-- ConfigMap.yaml
|    +-- Deployment.yaml
|    +-- HorizontalPodAutoscaler.yaml
|    +-- PodDisruptionBudget.yaml
|    +-- SealedSecret.yaml
|    +-- Service.yaml
+-- values.yaml
```

Listing 5: Struktur eines Helm-Charts für eine Fachanwendung

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: "{{ .Chart.Name }}-env"
    namespace: {{ .Values.namespace }}
data:
    {{- toYaml .Values.configuration | nindent 4 }}
```

Listing 6: Kubernetes ConfigMap-Template

```
kubectl create secret generic demo-secret \
  --from-literal=password=root -o yaml \
  --dry-run > secret.yaml
kubeseal < secret.yaml > sealed-secret.yaml
```

Listing 7: Erzeugung eines neuen SealedSecret

bei Konfigurationsänderungen auch nur die ConfigMap. Um der Anwendung neue Konfigurationen mitzuteilen, ist im einfachsten Fall ein Neustart der Pods notwendig. Hier hilft als kleiner Trick *[6]*, die Checksumme der Konfigurationselemente in das Deployment zu schreiben, um diesen Neustart zu erwirken *(siehe Listing 8)*.

Deployment-Monitoring

Ein kleiner Wermutstropfen ist augenblicklich noch die Überwachung der GitOps Tools selbst. Flux und SealedSecrets bringen CLIs mit, die Einblicke in Zustand und Fehler erlauben. Darauf ein Monitoring aufzubauen, dürfte sich jedoch schwierig gestalten. Eine Möglichkeit ist es, die Log-Ausgaben der Controller auszuwerten, im Falle der Google-Cloud zum Beispiel mittels Stackdriver. Eine homogene Lösung lässt sich mithilfe eines weiteren Tools aus dem Hause Weaveworks realisieren: Kubediff *[7]* gleicht die tatsächliche Konfiguration in Kubernetes mit dem gewünschten Zustand im GitOps Repository ab und bietet darüber hinaus einen Prometheus-Export zur Integration in bestehendes Monitoring.

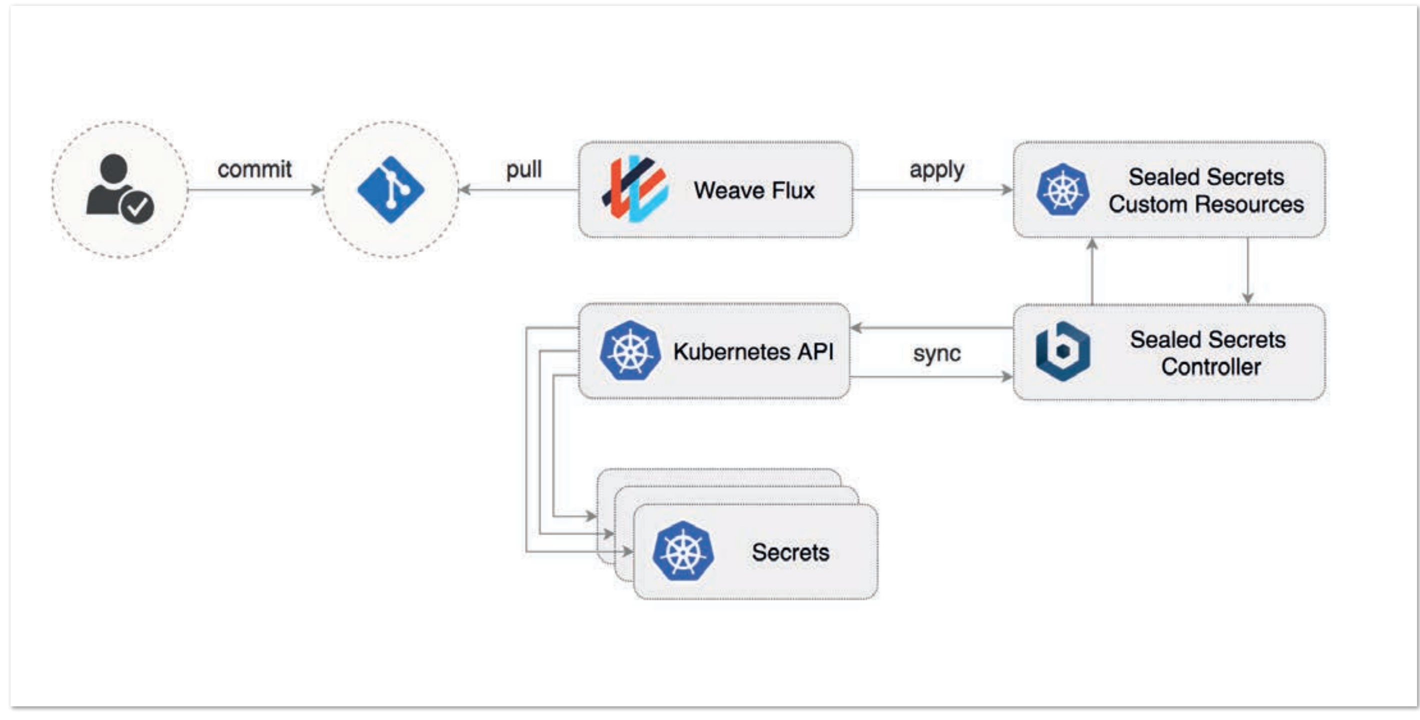


Abbildung 2: Integration mit SealedSecrets (© Weaveworks [2])

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    metadata:
      annotations:
        checksum/config: \
{{ include (print $.Template.BasePath "/ConfigMap.yaml") . | sha256sum }}
        checksum/secret: \
{{ include (print $.Template.BasePath "/SealedSecret.yaml") . | sha256sum }}
```

Listing 8: Deployment-Template mit Checksummen für Konfigurationselemente

Fazit

Durch GitOps lässt sich mit nur wenig Konfiguration eine flexible und zuverlässige Continuous-Delivery-Lösung gestalten. Dank der Feedback-Schleife des Delivery Operators wird Git zur alleinigen Quelle der Wahrheit und zum zentralen Element der Entwicklungs- und Betriebsprozesse. Etablierte Review- und Freigabemechanismen können gleichermaßen angewandt werden, wodurch auch Konfigurationsänderungen nachvollziehbar dokumentiert sind.

Quellen

[1] Alexis Richardson (2017): GitOps - Operations by Pull Request. *https://www.weave.works/blog/gitops-operations-by-pull-request*

[2] Weaveworks: Managing Helm releases the GitOps way. *https://www.weave.works/blog/managing-helm-releases-the-gitops-way*

[3] Wikipedia: glob. *https://en.wikipedia.org/wiki/Glob_(programming)*

[4] Flux Reference: Upgrading images in a HelmRelease using Flux. *https://docs.fluxcd.io/en/latest/references/helm-operator-integration.html#upgrading-images-in-a-helmrelease-using-flux*

[5] SealedSecrets. *https://github.com/bitnami-labs/sealed-secrets*

[6] Chart Development Tips and Tricks. *https://github.com/helm/helm/blob/master/docs/charts_tips_and_tricks.md*

[7] Weaveworks: Kubediff. *https://github.com/weaveworks/kubediff*

Bernd Stübinger, Florian Heubeck

Java User Group Ingolstadt e.V.

info@jug-in.bayern

Bernd und Florian sind Mitgründer und Organisatoren der JUG Ingolstadt. Für ihren Arbeitgeber MediaMarktSaturn erstellen sie ganzheitliche Softwarelösungen und geben ihre Erfahrungen mit neuen Tools und Konzepten gern weiter.

Make Java Hot Again

Heiko Rupp, Red Hat

Das Java-Ökosystem ist immer noch groß und stark. Es wird allerdings von mehreren Seiten angeknabbert. Speziell im Umfeld von Cloud und Serverless schauen sich Anwender nach Alternativen um. Quarkus nutzt hier einige interessante Ansätze, um dem entgegenzutreten.

Nachdem Java längere Zeit das Mittel der Wahl für viele Aufgaben war, hat die Sprache und speziell die Laufzeitumgebung der JVM einige Kritik abbekommen. Auf der einen Seite stehen die Skriptsprachen wie Ruby oder Python (aber auch JavaScript via NodeJS), die zwar interpretiert werden, in der Entwicklung jedoch sehr schnell sind. Der Entwickler ändert eine Codestelle und kann die Änderung direkt ausprobieren. In Java steht hier meist ein Compilier- und Deploy-Lauf dazwischen.

Auf der anderen Seite hat mit Golang eine Sprache und Laufzeitumgebung die Bühne betreten, die relativ schlanke, statische Binaries erzeugt, die auch sehr schnell starten. Dies ist im Cloud-Umfeld wichtig. Hier laufen viele Prozesse in Containern nur für wenige Minuten oder gar Sekunden – ganz anders als die klassischen Enterprise-Anwendungen, bei denen die JVM mit ihrem Hotspot-Compiler ihre ganze Stärke ausspielen kann.

„Supersonic Subatomic Java"

Der Zwischentitel ist die Tagline von Quarkus *(https://quarkus.io)* und versucht auszudrücken, dass hier beide der obigen Problemstellen angegangen wurden. Während der Entwicklung gibt es einen Modus, der einen schnellen Code-Ausprobier-Zyklus gestattet. Wenn man mit dem Ergebnis zufrieden ist, erlaubt Quarkus mithilfe des GraalVM-Projekts ein natives Binary zu erzeugen, das dann keine (klassische) JVM mehr benötigt und damit klein und schnell beim Start ist.

Schauen wir uns die Entwicklung einmal an und starten mit einem neuen Projekt. Das Tooling erlaubt es, mit einer Maven- oder Gradle-Kommandozeile ein neues Projekt zu starten *(siehe Listing 1)*. Fehlende Parameter (zum Beispiel die artifact-Id) werden durch Prompts abgefragt. Im Listing wird die Version von Quarkus (0.20.0) verwendet, die zum Einsendeschluss die aktuelle war.

Nachdem das Projekt erzeugt wurde, steht in `src/main/java/de/bsd/jm/Greeter.java` ein JAX-RS-Endpunkt zur Verfügung. Wir können nun mit `./mvnw compile quarkus:dev` die Anwendung übersetzen und starten. Nach dem Start können wir in unserem Browser *http://localhost:8080/hallo* aufrufen und sehen die Ausgabe „hello". Da dies eine deutschsprachige Publikation ist, gefällt uns dies natürlich nicht. Ohne die laufende Anwendung zu stoppen, ändern wir den Text zum Beispiel auf „Hallo Java Aktuell". Wenn wir nun im Browser die Seite neu laden, werden wir nach kurzer Zeit die Änderung in Aktion sehen. Auf der Konsole sieht dies dann wie in *Listing 1* aus.

In *Listing 2* sieht man auch schön, dass aktuell nur CDI und RestEasy als Erweiterung installiert sind (dies wird auch in der `pom.xml` reflektiert). Wenn wir nun mit einer Datenbank arbeiten wollen, können wir weitere entsprechende Erweiterungen installieren. Mit `./mvnw quarkus:list-extensions` kann die Liste der verfügbaren Erweiterungen angezeigt werden und mittels `./mvnw quarkus:add-extension -Dextensions=„quarkus-hibernate-`

```
mvn io.quarkus:quarkus-maven-plugin:0.20.0:create \
  -DprojectGroupId=de.bsd \
  -DclassName="de.bsd.jm.Greeter" \
  -Dpath=„/hallo"
```

Listing 1: Neues Projekt starten

orm-panache,quarkus-jdbc-postgresql" könnten wir beispielsweise Hibernate-Panache und den Postgres-JDBC-Treiber installieren. Diese Erweiterungen werden in erster Linie in `pom.xml` reflektiert und stehen dann für die Entwicklung zur Verfügung.

Warum Erweiterungen?

Man fragt sich nun, wozu Erweiterungen notwendig sind und speziell solche, die nicht den normalen bekannten Paketen und Maven-Koordinaten entsprechen.

Quarkus ist kein Application-Server, sondern ein Framework, das aus einem Kern, dem Tooling und eben den Erweiterungen besteht. Auf diese Weise kann man nur diese Erweiterungen (in *Listing 2* auch Features genannt) installieren, die man benötigt. Quarkus setzt dabei auf bekannte Projekte wie SmallRye (Implementierung von MicroProfile *[5]*), Vert.x oder Hibernate auf.

Der schnelle Start von Quarkus-Anwendungen kommt unter anderem daher, dass manche Vorgänge, die normalerweise beim Start der Anwendung ausgeführt werden, bereits beim Übersetzen stattfinden. Ein Beispiel wäre das Parsen von `hibernate.xml`: Diese Datei ändert sich üblicherweise nicht, weswegen es nicht notwendig ist, sie bei jedem Start aufwendig zu parsen. Stattdessen kann beim Übersetzen der Anwendung einmalig diese Datei geparst und Java-Code erzeugt werden. Auf diese Weise kann man auch einige Beschränkungen von Graal umgehen. Quarkus-Erweiterungen können selbst entwickelt werden und bestehen aus zwei Teilen.

- Laufzeit-Komponente: Diese stellt die eigentliche Funktionalität bereit.
- Deployment-Komponente: Diese stellt die Konfiguration bereit. Im obigen Beispiel würde hier `hibernate.xml` geparst und das Ergebnis bereitgestellt.

Adam Bien hat dies sehr anschaulich beschrieben *[3]*.

Klassischer Start und Tests

Der gezeigte Entwicklungsmodus nutzt die klassische JVM. Wir können also auch hergehen und die Anwendung auf klassische Art und Weise mit `mvn package` bauen und starten. Das erste Bauen wird fehlschlagen, da wir den Text, den der JAX-RS-Endpunkt zurückgibt, geändert haben, dies aber nicht im ebenfalls durch das Tooling erzeugten Test nachgezogen haben. Ist dies geschehen und der Test erfolgreich, können wir die Anwendung dann starten, wie in *Listing 3* gezeigt.

Wir geben hier nur das `-runner.jar` auf der Kommandozeile mit. Im Hintergrund hat das Bauen der Anwendung weitere Bibliotheken in `target/lib` abgelegt, die dann von diesem Runner geladen werden.

Auf der Suche nach dem heiligen Graal

Wie kommen wir nun von unserer Java-Anwendung zu einem nativen Binary? Wie wahrscheinlich bekannt ist, führt die JVM Bytecode auf einer abstrakten Stack-Machine aus. Wir benötigen also nun einen Compiler, der den Bytecode für eine konkrete Prozessorarchitektur übersetzt. Genau dies leistet die SubstrateVM des Graal-Projekts. Dies geschieht, anders als auf der JVM, nicht zur Laufzeit (just-in-time, JIT), sondern vor der Ausführung des Codes (ahead-of-time, AOT). Der ausgeführte Code ist dann nicht mehr in der Lage, zur Laufzeit weiteren Code zu übersetzen, weswegen es einige Einschränkungen bei der Verwendung des AOT-Compilers gibt (eine vollständige Liste gibt es unter *[4]*):

- Klassen können nicht nachgeladen werden
- Es gibt keinen SecurityManager
- Es gibt kein `finalize()` – Hurra!
- Reflection ist nur eingeschränkt möglich (und benötigt zusätzliche Konfiguration)
- Der Garbage Collector ist nicht so gut wie bei der klassischen VM

Viele dieser Einschränkungen sind der sogenannten *Closed-World-Assumption* geschuldet: Beim Kompilieren wird davon ausgegangen, dass jeder Code-Pfad erreichbar ist und alle Klassen bekannt sind. Damit kann der Compiler optimieren und auch Code eliminieren, der gar nicht erreicht werden kann.

```
2019-07-31 11:05:16,329 INFO  [io.qua.dev] (executor-thread-1) Changed source files detected, recompiling
[/Users/hrupp/tmp/quarkus/src/main/java/de/bsd/jm/Greeter.java]
2019-07-31 11:05:16,863 INFO  [io.quarkus] (executor-thread-1) Quarkus stopped in 0.021s
2019-07-31 11:05:16,865 INFO  [io.qua.dep.QuarkusAugmentor] (executor-thread-1) Beginning quarkus augmentation
2019-07-31 11:05:17,109 INFO  [io.qua.dep.QuarkusAugmentor] (executor-thread-1) Quarkus augmentation completed in 244ms
2019-07-31 11:05:17,147 INFO  [io.quarkus] (executor-thread-1) Quarkus 0.20.0 started in 0.284s.
Listening on: http://[::]:8080
2019-07-31 11:05:17,148 INFO  [io.quarkus] (executor-thread-1) Installed features: [cdi, resteasy]
2019-07-31 11:05:17,148 INFO  [io.qua.dev] (executor-thread-1) Hot replace total time: 0.821s
```

Listing 2: Konsolenausgabe von Quarkus, wenn eine Änderung in den Quellen erkannt wurde.

```
$ java -jar target/ja-demo-1.0-SNAPSHOT-runner.jar
2019-07-31 11:28:42,703 INFO  [io.quarkus] (main) Quarkus 0.19.1 started in 1.008s. Listening on: http://[::]:8080
2019-07-31 11:28:42,713 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
```

Listing 3: Start der Anwendung

```
$ target/ja-demo-1.0-SNAPSHOT-runner
2019-07-31 11:54:56,751 INFO  [io.quarkus] (main) Quarkus 0.20.0 started in 0.018s. Listening on: http://[::]:8080
2019-07-31 11:54:56,752 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]

$ ls -ls !$
ls -ls target/ja-demo-1.0-SNAPSHOT-runner
   21 -rwxr-xr-x  1 hrupp  staff  21434688 30 Jul 11:54 target/ja-demo-1.0-SNAPSHOT-runner
```

Listing 4: Schneller Start des kleinen nativen Image in 18ms

Natives Image bauen

Nachdem wir hinsichtlich Entwicklung und Tests mit der JVM zufrieden sind, möchten wir nun ein natives Image bauen. Das geht ganz einfach, indem der Maven-Kommandozeile `-Pnative` mitgegeben wird. Allerdings ist dies nicht alles, da wir hierzu auch die SubstrateVM benötigen. Die Community-Edition können wir von *https://github.com/oracle/graal/* beziehen. Nach der Installation müssen wir noch zwei weitere Schritte erledigen:

1. Den Pfad zu Graal setzen: `export GRAALVM_HOME=/Library/Java/JavaVirtualMachines/graalvm-ce-19.1.1/Contents/Home/` (auf OS/X)
2. Das Native Image Binary installieren: `$GRAALVM_HOME/bin/gu install native-image`

Ist beides erledigt, können wir `mvn clean package -Pnative` aufrufen und uns entspannt zurücklehnen. Wobei dies noch untertrieben ist und wir durchaus Zeit genug haben, uns einen Kaffee zu holen (der geneigte Leser denkt hier sicher an *https://xkcd.com/303/*). Leider (oder zum Glück?) schlägt das Erstellen des nativen Image manchmal fehl, obwohl alle Tests mit der JVM erfolgreich sind. Dies liegt oft daran, dass eine Annahme der SubstrateVM nicht erfüllt ist. Ich habe es auch schon gesehen, dass Casts von zum Beispiel *Long* nach *Int* der Grund waren; dies ist ein Fall, den die JVM meist verzeiht (für Werte, die klein genug sind), bei dem der Compiler jedoch sieht, dass es hier nicht passt. Der Build wird auch scheitern, wenn wir nur die obigen Erweiterungen in das Projekt übernehmen, sie aber nicht nutzen.

Nach circa zwei bis drei Minuten ist das Image fertig und wir können es starten, wie in *Listing 4* gezeigt.

Der Autor hat das Image unter OS/X gebaut. Damit ist es nicht für den Einsatz in Linux-Containern geeignet. Dies ist jedoch kein Problem, da wir beim Erstellen ein weiteres Flag `-Dnative-image.docker-build=true` mitgeben können, das den Build in einem laufenden Linux-Container ausführt und damit ein natives Linux-Binary erzeugt. Falls dieser Build sehr viel länger benötigt als ohne Docker oder gar abbricht, benötigt Docker mehr Speicher.

In der Praxis wird man diese nativen Binaries nicht (regelmäßig) auf dem Laptop durchführen, sondern einfach in der CI/CD-Pipeline bauen. Hier ist es auch egal, ob ein Build etwas länger dauert.

Fazit

Quarkus ist ein faszinierendes Stück Technologie, das Vorteile sowohl bei der Entwicklung als auch im Betrieb bringt. Mit Quarkus kann man Jahre an Erfahrung weiter nutzen und muss nicht beispielsweise zu Golang wechseln, um in den Genuss von geringerem Speicherverbrauch oder schnellem Startup zu kommen.

Allerdings ist die SubstrateVM noch nicht auf der Höhe der klassischen VM, was Garbage Collection oder die Ausführungsgeschwindigkeit von JIT-Code angeht.

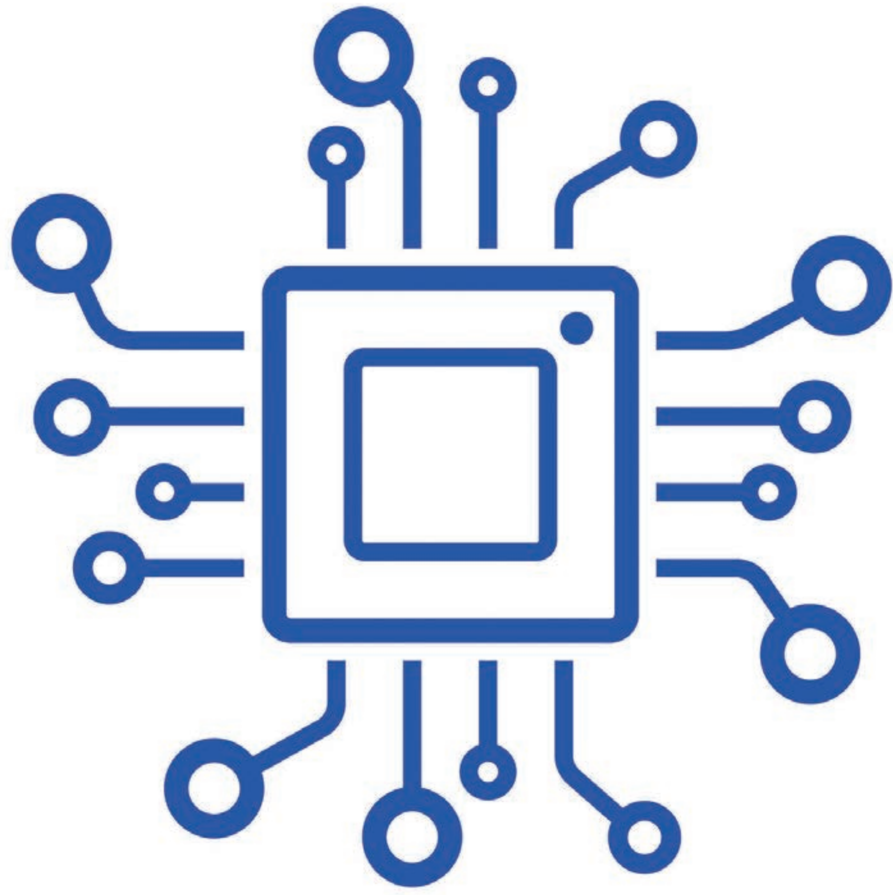
Einfach eine monolithische Legacy-Anwendung mit Quarkus neu zu übersetzen, wird in den wenigsten Fällen ein zufriedenstellendes Ergebnis erzielen. Die Anwendungsfälle für Quarkus sind ganz eindeutig Cloud-native Services, die eher kurzlebig sind und von einem schnellen Start sowie geringerem Speicherverbrauch profitieren.

Quellen

[1] Quarkus Homepage, *https://quarkus.io/D*
[2] Quarkus Cheat Sheet, *https://lordofthejars.github.io/quarkus-cheat-sheet/*
[3] *http://adambien.blog/roller/abien/entry/simplest_possible_quarkus_extension*
[4] *https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md*
[5] *https://microprofile.io*

Heiko Rupp
Red Hat
hrupp@redhat.com

Heiko ist seit langer Zeit ein Open-Source-Enthusiast. Er arbeitet seit mehr als einem Jahrzehnt bei Red Hat im Bereich Middleware-Monitoring und -Management und beschäftigt sich aktuell mit Observability im ServiceMesh und Kiali. Zusätzlich hilft er, die nächste Generation von Java Microservices mit seiner Arbeit in Eclipse MicroProfile zu definieren. Heiko hat das erste deutsche Buch zu JBossAS und eines der ersten deutschen Bücher zu EJB3 geschrieben. Er lebt mit seiner Familie in Stuttgart.



Hallo Microservices, gibt's euch auch transaktional?

Matthias Koch und Markus Grabert, sidion

Es ist 2019. Microservices sind kein Novum mehr und längst in viele Bereiche durchgedrungen. Allerdings hat der Ansatz, viele verteilte kleine Services zu verwenden, auch seine architektonischen Herausforderungen. In den letzten Jahren haben wir gelernt, mit Consumer Driven Contracts, Service Discovery und API-Gateways umzugehen. Für all diese Probleme gibt es mittlerweile fertige Lösungen aus der Werkzeugkiste. Kommt hingegen Datenkonsistenz (Eventual Consistency) über Servicegrenzen hinweg zur Sprache, dann sieht es anders aus. Zuallererst wird überlegt, ob Transaktionen in diesem Kontext überhaupt notwendig sind und nicht ein Indiz für zu harte Kopplung darstellen. Aber: Transaktionen wird es weiterhin geben und das rein fachlich bedingt.

In diesem Artikel wollen wir auf einen alternativen Weg hinweisen, der es ermöglicht, logische Transaktionen in Microservice-Architekturen abzubilden: das Saga-Pattern, das mithilfe von kompensatorischen Operationen eine Art Rollback nachbildet.

Abschließend soll ein Ausblick auf das Proposal der Spezifikation „Long Running Actions for MicroProfile" gegeben werden. Deren API verspricht die Koordination von Services zu vereinheitlichen und damit die Implementierung zu vereinfachen. Hierbei wird ein global konsistenter Zustand sichergestellt, ohne dass Locks auf Daten benötigt werden.

Was sind Transaktionen?

Transaktionen können als Folge von Aktionen beschrieben werden, die als logische Einheit betrachtet werden müssen. Das heißt, entweder alle Aktionen werden in ihrer Gesamtheit umgesetzt oder keine der Aktionen. Auf diese Weise soll in jedem Fall ein konsistenter Zustand sichergestellt werden. Üblicherweise werden Transaktionen in Prozessen eingesetzt, bei denen Daten- oder

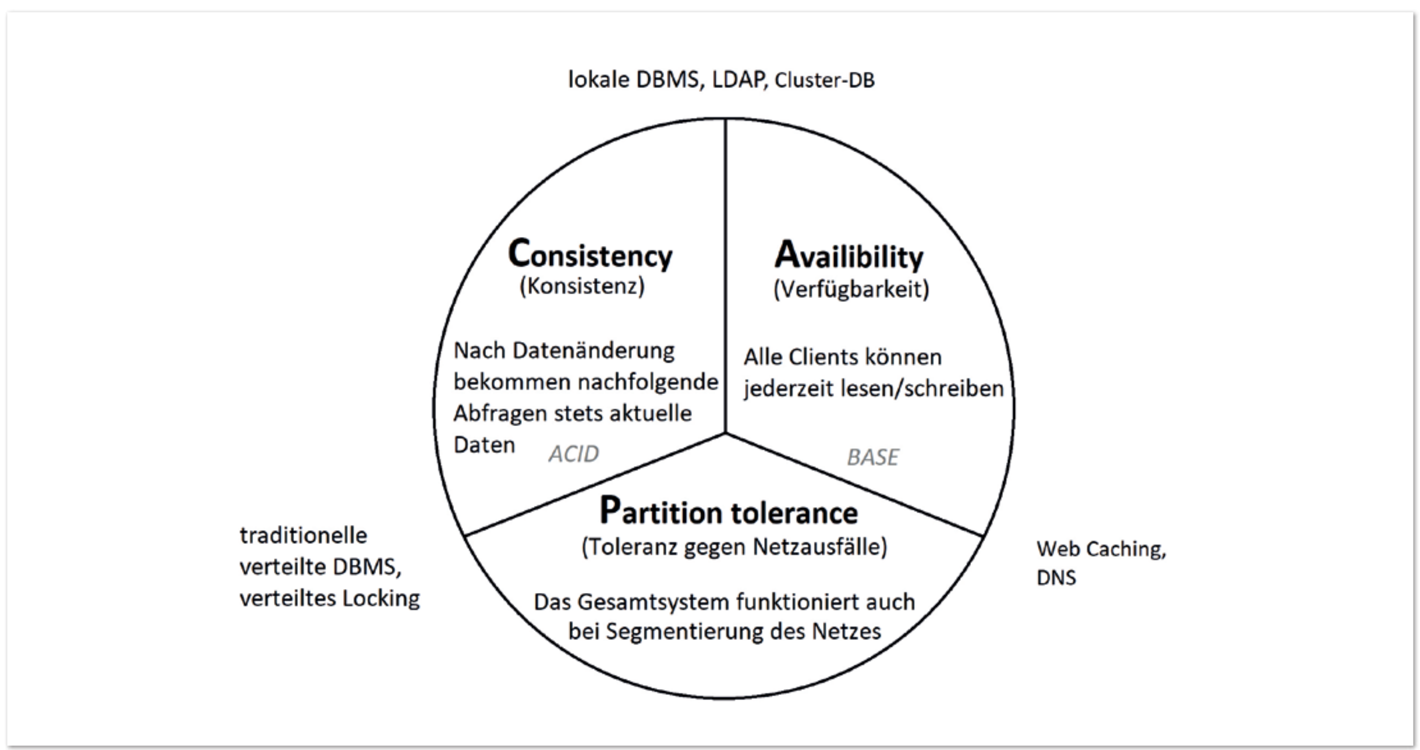


Abbildung 1: Das CAP-Theorem (Quelle: Matthias Koch)

Zustandsinkonsistenzen – verursacht durch eine partielle Umsetzung der Aktionen – große Risiken bergen würden. Das Spektrum der möglichen Risiken reicht von irreversiblen Datenverlusten, Blockierung von Folgeprozessen und Services bis hin zu eingehenden finanziellen Verlusten und so weiter.

Gibt es keine Probleme, wenn nur ein paar und nicht alle Aktionen umgesetzt werden, dann liegt auch keine Transaktion im Sinne der Definition vor. Am Beispiel Bürgerbüro wird dies deutlich: Wer dort ein Ticket zieht, aber nicht zum Schalter geht, wenn die eigene Nummer aufgerufen wird, hat in der Regel nicht mit nennenswerten Auswirkungen zu rechnen. Eine Transaktion ist für diesen Prozess also nicht nötig. Anders ist es bei Banküberweisungen: Wenn das Geld dem Empfängerkonto gutgeschrieben wird, dann sollte immer gleichzeitig das Senderkonto belastet werden. Auch bei Online-Flugbuchungen sollte sichergestellt sein, einen Sitz reserviert zu bekommen, wenn dafür bezahlt wurde. Beides sind Prozesse, bei denen Transaktionen sinnvoll zum Einsatz kommen.

Wie werden Transaktionen nun in IT-Systemen umgesetzt?

Aktionen einer Transaktion gehen immer mit einer Daten- oder Zustandsänderung einher. Diese Änderungen müssen wiederum sofort atomar und konsistent umgesetzt werden, damit die Folge-Aktionen darauf aufbauen können. Bei Monolithen ist es relativ einfach, einen konsistenten Zustand zu garantieren. Dies unterstützen zum Beispiel die traditionellen relationalen DBMS auch schon seit langer Zeit; das Stichwort heißt „ACID“: Atomicity, Consistency, Isolation und Durability. Bei verteilten Systemen, insbesondere Microservices, gibt es jedoch zusätzliche Herausforderungen: Wie kann garantiert werden, dass innerhalb einer Transaktion alle Aktionen auf den beteiligten Systemen wirklich ausgeführt werden? Schließlich sind die Systeme nicht immer zuverlässig erreichbar (Netzwerkprobleme, Überlastung).

Das CAP-Theorem

Mit der Problematik „Datenverarbeitung in verteilten Systemen“ befasste sich 2000 auch Dr. Eric Brewer und stellte sein CAP-Theorem *(siehe Abbildung 1)* in einer Keynote vor *[1]*. Später wurde das CAP-Theorem nochmals formalisiert und bewiesen *[2]*.

Das Theorem in kurz: Bei Systemen, die Daten mit anderen Prozessen teilen, gibt es drei wesentliche, beeinflussende Faktoren: Konsistenz (C für Consistency), Verfügbarkeit (A für Availibility) und Toleranz gegen Netzwerk-Segmentierung (P für Partition Tolerance). Zwischen Konsistenz und Verfügbarkeit muss jedes Mal erneut entschieden werden. Dies wird im Folgenden ausgeführt.

Die Qual der Wahl

Leider gibt es bei verteilten Systemen in Netzwerken immer die Möglichkeit von Netzwerkstörungen. So kann es passieren, dass Teile des Netzwerks abgetrennt werden und ein verteiltes System plötzlich Kommunikationsprobleme hat.

Dann gibt es nur noch zwei Optionen: Die erste ist, auf Konsistenz zu setzen und nur noch mit einem ausgewählten Teil des Systems und der Daten weiterzuarbeiten. Der Rest des Systems wird in diesem Fall deaktiviert, bis die Netzwerkprobleme wieder behoben sind. Abschließend wird alles synchronisiert.

Alternativ ist das verteilte System auf Verfügbarkeit ausgelegt und die unterschiedlichen, abgetrennten Bereiche arbeiten erst einmal unabhängig voneinander weiter. Dies führt längerfristig unweigerlich zu Dateninkonsistenzen innerhalb des Gesamtsystems, die zu einem späteren Zeitpunkt behoben werden müssen. Dies ist oft nicht automatisiert, sondern nur manuell möglich. Die bisherigen Ansätze, verteilte Datenspeicherung zu implementieren, waren daher meist basierend darauf, wie die ACID-Konsistenz-Garantie auf mehrere verteilte DBMS ausgebreitet werden kann.

XA-Transaktionen

X/Open XA (eXtended Architecture) ist wohl das meistverbreitete Protokoll, um verteilte Transaktionen zu implementieren. Es wurde schon 1991 von X/Open, jetzt OpenGroup, veröffentlicht *[3]*, also lange vor SOA und Microservices. Bei Ausführung von verteilten Datenbank-Transaktionen wird immer noch das bereits erwähnte ACID-Prinzip gewährleistet. Wie funktioniert das? XA verwendet dazu das 2-Phase Commit Protocol (2PC). Vereinfacht gesagt werden in der ersten Phase die Datenänderungen bei allen beteiligten DBMS vorbereitet. Wenn alle ein „Okay" melden, folgt die Phase zwei, in der die Datenänderungen wirklich durchgeführt werden.

Geeignet für Microservices?

Leider ist der Prozess ab dem „Okay" blockierend, was es bei Microservices unbedingt zu vermeiden gilt, da dies mit Einschränkungen in der parallelen Verarbeitung verbunden ist. Außerdem führt die 2PC-Kommunikation zu höheren Latenzen. Dazu kommt, dass Ressourcen in verteilten Systemen nicht immer verfügbar sein müssen, was mit einer Fehlerbearbeitung einhergeht, die sehr komplex ist. Dies macht die Fehleranalyse wiederum teuer, und das Ganze wird schwer testbar. Schließlich werden die Systeme eng gekoppelt, was dem Prinzip der Isolation und losen Kopplung von Microservices entgegensteht.

Geeignet für Transaktionen?

Dem 2PC geschuldet, gibt es bei XA-Transaktionen eine komplizierte Fehlerbehandlung, da es verschiedene Fehlerquellen und -ursachen geben kann. In einigen Fehlerfällen muss sogar manuell eingegriffen werden, um die Daten zu korrigieren. Eine atomare Konsistenz ist also nicht immer garantiert. Die unterschiedliche Verfügbarkeit der Services und das heterogene Laufzeitverhalten der Systeme sind weitere Faktoren, die einen möglichen Einsatz einschränken können, vor allem bei Transaktionen mit Anforderungen an starke Konsistenz oder Echtzeit, wie zum Beispiel beim Aktienhandel an der Börse.

Zusammengefasst sind XA-Transaktionen für Microservices nicht besonders geeignet. Letztendlich wurden sie auch rund 20 Jahre vor der Einführung von Microservices konzipiert und eher auf monolithische Systeme zugeschnitten.

Idealerweise bräuchte es also ein Framework, das besser zum Microservice-Ansatz und zur Microservice-Architektur passt. Der nachfolgende Abschnitt gibt einen Überblick darüber, was dafür konkret benötigt wird.

Anforderungen eines Transaktions-Frameworks für Microservices

- Zunächst sollen keine Datensätze gelockt werden. Das würde nicht nur die parallele Verarbeitung behindern, sondern auch potenziell das Risiko mit sich bringen, dass Daten gegebenenfalls für immer gelockt bleiben und damit gar nicht mehr verfügbar sind.
- Auch die lose Kopplung der Services soll nicht aufgegeben werden. Das System soll weiterhin einfach änderbar und erweiterbar sein.
- Da in verteilten Systemen die Zustellung einer Nachricht nie garantiert werden kann, braucht es auch dafür Mechanismen, mit denen sich das leicht handhaben lässt.
- Den Autoren schwebt ein generischer Ansatz vor, der die Möglichkeit bietet, die Datenkonsistenz zu erhalten, und der auch bei der nächsten gebauten Anwendung wieder gleichartig funktioniert. Ziel ist es, das sprichwörtliche Rad nicht bei jeder Anwendung wieder neu erfinden zu müssen.

Zeit sich umzuorientieren?

Bevor diese Frage beantwortet werden kann und geschildert wird, ob und wie das geht, erst einmal zurück zum CAP-Theorem. Der bisher gewählte Ansatz, Transaktionen strikt nach dem ACID-Prinzip durchzuführen und die atomare Datenkonsistenz zu priorisieren, geht mit einigen Nachteilen einher, die im Microservices-Umfeld nicht gewünscht sind.

Wer sich einmal etwas vom ACID-Prinzip entfernt, stößt früher oder später auf den Begriff „BASE". Er stellt quasi den Gegenpol zu ACID dar. Die Abkürzung setzt sich wie folgt zusammen:

- **B**asically **A**vailable
- **S**oft-State
- **E**ventual Consistency

„Eventual Consistency" klingt auf den ersten Blick nach einem Zustand, der in IT-Systemen nicht erstrebenswert ist. Schließlich sollen eine Vorhersagbarkeit und Garantien erreicht werden. Bei tieferer Recherche wird jedoch deutlich, dass sich das Prinzip der „Eventual Consistency" in der IT-Welt schon verbreitet hat: Viele NoSQL-Datenbanken, Elasticsearch Index und nicht zuletzt das DNS-System basieren und vertrauen darauf. Es stellt sich die Frage, ob es denn überhaupt immer notwendig ist, dass verteilte Transaktionen sofort – und atomar als Einheit – umgesetzt werden. Genügt es nicht bisweilen sicherzustellen, dass (innerhalb eines Zeitraumes) verteilte Transaktionen wirklich als Gesamtheit durchgeführt wurden?

Sagas

Es gibt tatsächlich eine Beschreibung, wie eine verteilte Transaktion auf so eine Art abgebildet werden kann: das Saga-Pattern! Zuerst soll ein Blick auf die Ursprünge des Saga-Patterns geworfen werden. Der Begriff „Saga" wurde in einer wissenschaftlichen Abhandlung von Héctor García-Molina und Kenneth Salem bereits 1987 als Alternative zu Long Lived Transactions (LLT) in der Datenbank-Programmierung vorgeschlagen *[4]*. Das Problem dabei war, dass LLT-Datenbank-Tabellen blockierten und dadurch andere (parallele) Prozesse behinderten. Die Idee: Die LLT sollte zur „Saga" werden, indem sie in (möglichst) separate Teil-Transaktionen getrennt wird, die sequenziell oder auch parallel verarbeitet werden können. Diese Datenveränderungen dürfen aber nur dauerhaft in der DB bestehen, wenn sie in ihrer Gesamtheit umgesetzt wurden.

Der Vorteil, der sich durch Aufteilung in einzelne Teil-Transaktionen ergibt, ist die Möglichkeit für andere Anwendungen, auf Datenbank-Tabellen zuzugreifen, die ansonsten durch eine LLT längere Zeit blockiert wären.

Der Nachteil besteht in der Komplexität. Falls Teil-Transaktionen einer Saga fehlschlagen, müssen kompensatorische Transaktionen für alle anderen bisher umgesetzten Teile dieser Saga stattfinden, um den Originalzustand wiederherzustellen.

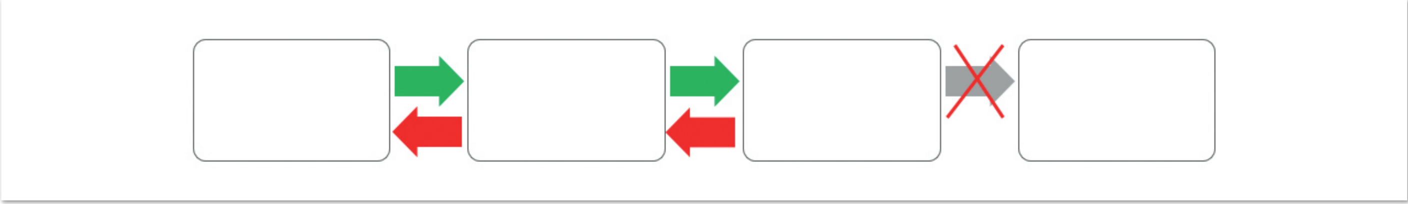


Abbildung 2: Saga Interaction Pattern (Quelle: Matthias Koch)

Saga-Pattern

In seinem Buch „SOA Patterns" von 2012 hatte Arnon Rotem-Gal-Oz schließlich dieses Prinzip auf verteilte Transaktionen angewendet und als „Saga-Pattern" *[5]* vorgestellt. Er beschreibt, dass das Saga-Pattern eigentlich keine neue Erfindung sei, sondern nur eine Anwendung verschiedener schon bestehender „Interaction Patterns". Ab 2013 wurden über das Saga-Pattern und dessen Implementation im SOA- und Microservices-Umfeld vermehrt Diskussionen geführt *[6]*. Es wurden unterschiedliche Vorgehensweisen (Orchestrierung, Choreografie) vorgestellt, wie Sagas implementiert werden können. Darauf wird im späteren Verlauf des Artikels noch eingegangen. 2015 gab es außerdem die Abhandlung „Verteilte Sagas" von McCaffrey, Kingsbury und Dr. Narula *[7]*, bei der eine orchestrierte Saga auf ein verteiltes IT-System angewendet wird.

Welche Ideen stecken nun hinter dem Saga-Pattern?

Beim Saga-Pattern werden naturgemäß kompensatorische Transaktionen durchgeführt, die zu jeder Aktion beziehungsweise jedem Command registriert werden. Im Falle eines Fehlers werden dann alle Transaktionen in inverser Reihenfolge ausgeführt *(siehe Abbildung 2)*. Es soll sichergestellt sein, dass kein möglicher Zustand vergessen wird, betrachtet zu werden. Das würde unweigerlich irgendwann zu einem Fehlverhalten der Anwendung führen.

Angenommen es liegt eine einfache Transaktion vor, die aus vier Einzelaktionen besteht, die sequenziell abgearbeitet werden. Jede Einzelaktion wird immer noch nach dem ACID-Prinzip ausgeführt oder könnte wiederum eine Transaktion sein.
Wie wird eine Transaktion dieser Art mit Sagas implementiert? Und wie orchestriert/choreografiert man so etwas mit Sagas?

Orchestrierung und Choreografie

Es gibt zwei unterschiedliche Vorgehensweisen, um Sagas zu implementieren: Orchestrierung *(siehe Abbildung 3)* und Choreografie *(siehe Abbildung 4)*. Orchestrierung einer Saga bedeutet, dass es eine zentrale Stelle gibt, welche die Fortschritte der einzelnen Aktionen verfolgt und vor allem steuert. Alle Prozesse, die diese Aktionen durchführen, müssen also mit dieser zentralen Stelle in Kontakt stehen und wissen unter Umständen nichts von den anderen beteiligten Aktionen. Bei der Choreografie hingegen müssen die Aktionen selbstständig dafür Sorge tragen, dass sie in der richtigen Reihenfolge ausgeführt werden. Kommunikation zwischen den Prozessen, die die Aktionen durchführen, ist also unbedingt notwendig.

Implementierungen

Derzeit gibt es leider noch nicht viel Auswahl an Frameworks und Werkzeugen, die einen bei der Implementierung des Saga-Patterns unterstützen. Hier sei nur auf das **Eventuate Tram Saga** API *[8]* und das **Axon**-Framework *[9]* verwiesen. Auf beides wollen wir wegen des anspruchsvollen Programmiermodells und der schlechten Kapselung nicht eingehen.

Schließlich gibt es noch den Eclipse Microprofile Standard. Deren Arbeit an Sagas befindet sich noch in der Spezifikationsphase und sieht eine Implementierung mittels sogenannter **Long-Running Actions** (LRA) vor *[10]*. Für Eclipse Microprofile spricht, dass es von

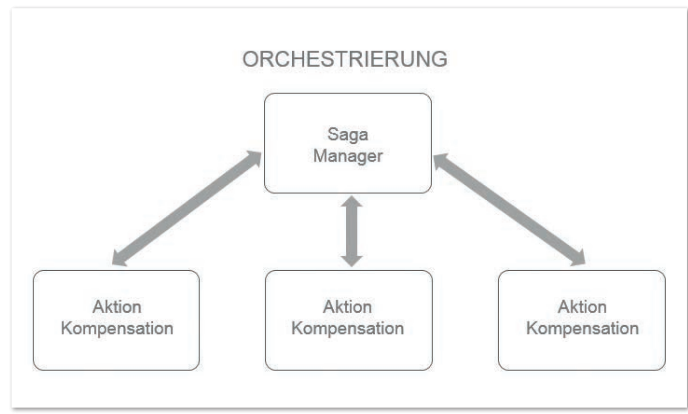


Abbildung 3: Orchestrierung (Quelle: Matthias Koch)

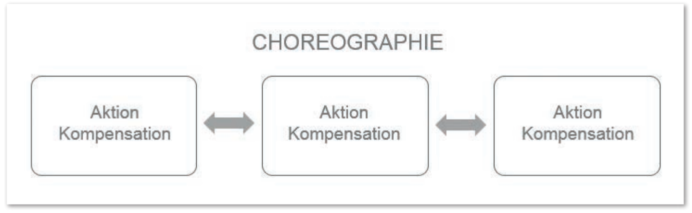


Abbildung 4: Choreografie (Quelle: Matthias Koch)

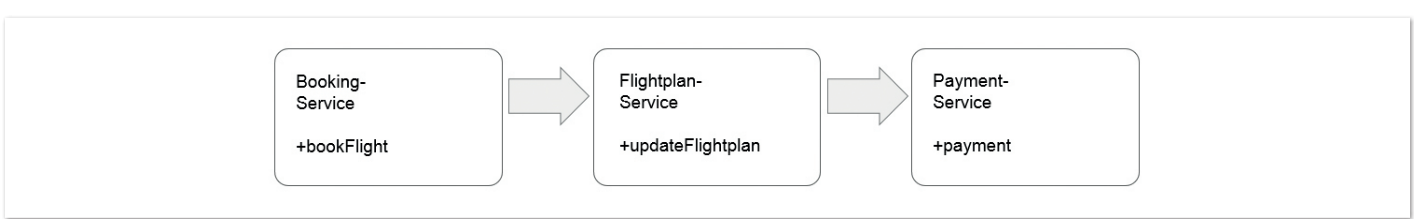


Abbildung 5: Beispiel Flugbuchung (Quelle: Matthias Koch)

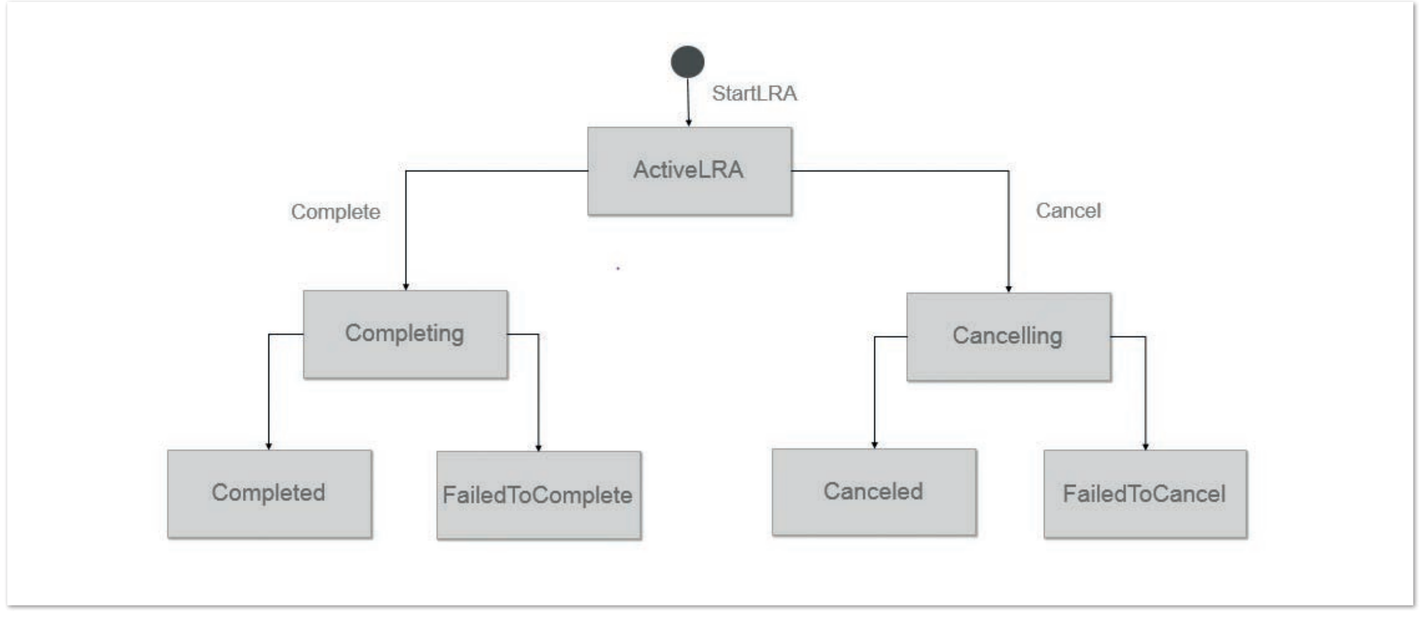


Abbildung 6: LRA-Zustandsmodell (Quelle: Matthias Koch)

vielen namhaften Firmen unterstützt wird und es viele Implementierungen für Microservices (Thorntail, Open Liberty, Payara Micro) dafür gibt. Wenn also die LRA-Spezifikation in den Eclipse Microprofile Standard aufgenommen wird, wird es eine große Auswahl an Implementierungen des Saga-Patterns geben.

Abgrenzung der Zuständigkeiten LRA-Framework vs. Applikation

Da das LRA-Framework die Fachlichkeit der Anwendung nicht kennen kann, muss die Applikation sich weiterhin um einige Dinge selbst kümmern. Daher ist diese Abgrenzung von zentraler Bedeutung. Das LRA-Framework definiert die Zustände und Trigger, die benötigt werden, um eine Transaktion durchzuführen. Die Abfolge der Trigger ist genau definiert. Es wird garantiert, dass die Trigger nicht mehrfach ausgeführt werden, denn das hätte in einem transaktionalen System gravierende Folgen. Das Framework macht keine zeitlichen Zusicherungen, wann die Trigger ausgelöst werden.

Die Applikation hingegen muss für jeden Command (Action) innerhalb einer LRA sicherstellen, dass dieses invertiert werden kann. Bei der Definition des API muss die Kompensierbarkeit bereits mitberücksichtigt und natürlich muss dies auch implementiert werden. In allen Zuständen hat die Anwendung das Verhalten zu definieren. Außerdem hat die Applikation für die Persistenz derjenigen Daten zu sorgen, die für die Kompensation notwendig sind. Dabei muss sichergestellt werden, dass diese Daten auch über einen Neustart des Service hinaus (gegebenenfalls auch durch einen JVM-Crash) verfügbar sind.

Beispiel Flugbuchung

Anhand eines einfachen Beispiels *(siehe Abbildung 5)* soll nun die Funktionsweise einer LRA erklärt werden. Die Beispielanwendung bucht Flüge, aktualisiert die Passagierlisten und stellt schlussendlich die Bezahlung sicher. Jeder Command interagiert mit einem eigenständigen Drittsystem. Daher wurde entschieden, jeden in einen eigenen „bounded context" zu verschieben. Das Payment soll am Ende der Kette stehen, um die Anzahl der stornierten Bezahlungen zu minimieren.

LRA-Zustandsmodell

Das Zustandsmodell einer LRA *(siehe Abbildung 6)* berücksichtigt alle möglichen Fälle und sieht auf den ersten Blick bezüglich der Anzahl der Zustände recht komplex aus. Deswegen werden wir versuchen, dies für die Praxis etwas zu vereinfachen.

Nachdem die LRA begonnen wurde, wird irgendwann im Verlauf der Verarbeitung das Ende eingeleitet. Das kann entweder „complete" oder „cancel" sein. Prinzipiell sollte die Anwendung so geschrieben werden, dass beim „complete" keine Commands mehr ausgeführt werden – somit sollte der Zustand „FailedToComplete" gar nicht eintreten. Keineswegs sollte an dieser Stelle versucht werden, eine XA-Transaktion nachzubilden. Wenn etwas beim „cancel" schiefgehen sollte, müsste in jedem Fall korrigierend eingegriffen und nachbearbeitet werden. Eine jederzeit verfügbare Möglichkeit ist es, die Daten hierzu in eine BackOutQueue zu schreiben. Um den Fokus auf Transaktionen zu halten, wird darauf jedoch nicht weiter eingegangen.

Ähnlich sieht es für einen Participant aus, also einen Service, der an einer LRA teilnimmt *(siehe Abbildung 7)*. Hier liegt die Konzentration ebenfalls auf „complete" und „compensate".

LRA-Annotationen

Das LRA-Framework verwendet für die Implementierung der Sagas Annotationen. Hier nur die wichtigsten:

- **@LRA:** Annotiert die Arbeitsmethode einer LRA
- **@Complete:** Callback für den Fall des Erfolgs einer LRA
- **@Compensate:** Callback im Fehlerfall

Identifiziert wird eine LRA über `LRA_HTTP_CONTEXT_HEADER`. Nur die mit LRA annotierten Services nehmen daran auch teil. Jeder Downstream-Service (vom Request her betrachtet) hat aber potenziell die Möglichkeit, einer LRA beizutreten. Irgendwann terminiert die LRA; mit einem Statuscode 200 wird die LRA erfolgreich beendet. Ein 5xx-Statuscode leitet dann die Kompensation ein. Exceptions werden ebenfalls in einen 5xx verpackt.

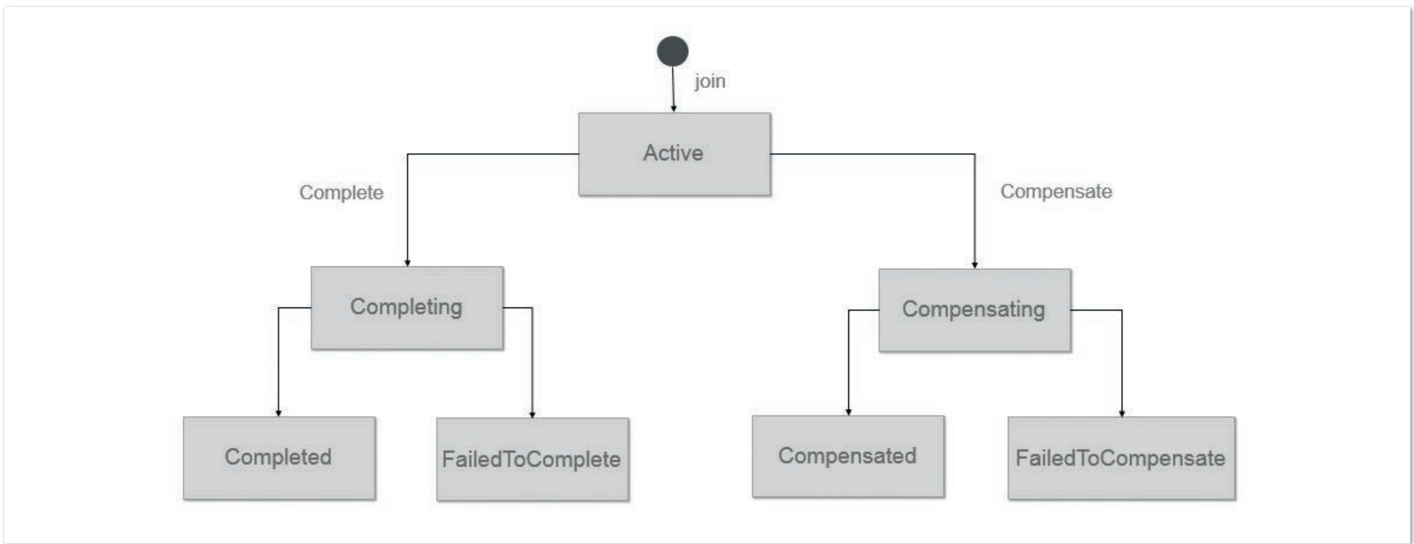


Abbildung 7: Zustandsmodell eines LRA-Participant (Quelle: Matthias Koch)

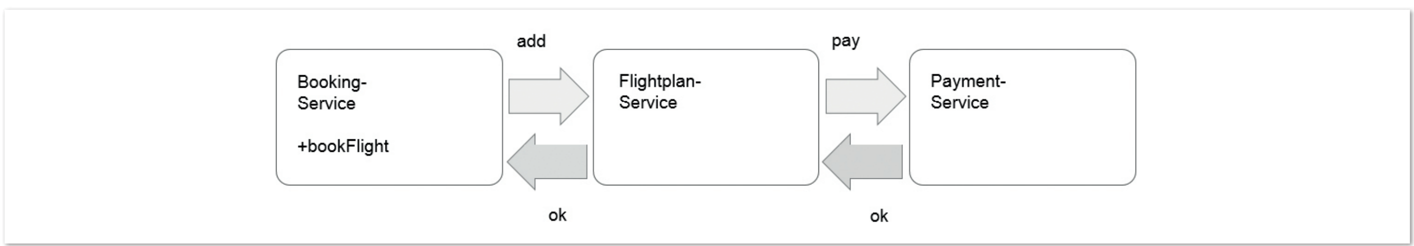


Abbildung 8: Flugbuchung im Erfolgsfall (Quelle: Matthias Koch)

```
@Path("/")
@ApplicationScoped
public class BookingService {

  @LRA(value = LRA.Type.REQUIRES_NEW, timeLimit = 30, timeUnit = ChronoUnit.SECONDS)
  @Path("/book")
  @PUT
  public Response bookFlight(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId, BookInfo bookInfo) {

    if (book(bookInfo.getFlightInfo())) {

      // Call downstream service.
      addFlightlist(bookInfo);
    }
    return Response.accepted().build();
  }
  ...
```

Listing 1: Start einer LRA (Quelle: Matthias Koch)

```
@Path("/")
@ApplicationScoped
public class FlightplanService {

  @LRA(value = LRA.Type.REQUIRED, timeLimit = 30, timeUnit = ChronoUnit.SECONDS)
  @Path("/add")
  @PUT
  public Response addFlightlist(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId, BookInfo bookInfo) {

    if (addPassenger(bookInfo.getPassengerInfo())) {

      // Call downstream service.
      payFlight(bookInfo.getPaymentInfo());

      // OK(200) would terminate the LRA. So use ACCEPTED(202) here.
      return Response.accepted().build();
    } else {

      // Finish LRA early.
      return Response.serverError().build();
    }

  }
  ...
```

Listing 2: Downstream-Service innerhalb einer LRA (Quelle: Matthias Koch)

```java
@Path("/")
@ApplicationScoped
public class PaymentService {


  @LRA(value = LRA.Type.REQUIRED)
  @Path("/pay")
  @PUT
  public Response payFlight(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId, PaymentInfo paymentInfo) {

    if (pay(paymentInfo)) {

      //Initiate completion of the LRA:
      return Response.ok().build();
    } else {

      //Initiate compensation of the LRA.
      return Response.serverError().build();
    }
  }
  ...
```

Listing 3: Das Ende der LRA wird eingeleitet (Quelle: Matthias Koch)

```java
@Complete
@Path("/complete")
@PUT
public Response completeFlightlist(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Resources can be released here.
  return Response.accepted().build();
}
```

Listing 4: Complete eines Participant (Quelle: Matthias Koch)

```java
@Complete
@Path("/complete")
@PUT
public Response completeBooking(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Resources can be released here.
  return Response.accepted().build();
}
```

Listing 5: Positiver Abschluss der LRA (Quelle: Matthias Koch)

Abbildung 8 zeigt den Erfolgsfall unseres Beispiels. Der Flug wird gebucht, ein Passagier der Passagierliste hinzugefügt und das Ganze bezahlt. Alles funktioniert fehlerlos.

Jetzt wird es Zeit, den Code anzusehen *(siehe Listing 1)*. Die Werte der LRA-Annotationen sind schon von den EJBs her bekannt – es sind jedoch keine EJBs. Der Timeout wird später noch betrachtet. Nun wird in der Arbeitsmethode der LRA-Header „injected", der die LRA identifiziert. Die Buchung selbst wird durchgeführt und im Erfolgsfall der Downstream-Service, also der Flightplan-Service, gerufen.

Ist auch die Aktualisierung der Passagierliste erfolgreich, wird schließlich die Bezahlung ausgeführt *(siehe Listing 2)*. Wichtig ist, dass hier ein „accepted", also ein 202, zurückgegeben wird. Bei einem 200 würde ja sonst schon das Ende der LRA ausgelöst. Bei einem Fehler wird sofort kompensiert.

Hiermit wird in jedem Fall das Ende der LRA eingeleitet *(siehe Listing 3)*. Je nach Erfolg des Payment endet die LRA als „complete" oder als „cancel". Die LRA soll so einfach gehalten werden, dass im Erfolgsfall nichts mehr zu tun ist *(siehe Listing 4)*. Mit dem „complete" der letzten Action wird die LRA als „completed" beendet *(siehe Listing 5)*.

Sollte nun ein „compensate" ausgelöst werden *(siehe Abbildung 9)*, so müssen immer alle aktiven Actions einer LRA kompensiert werden *(siehe Listings 6, 7* und *8)*. Das kann auch schon früher enden *(siehe Abbildung 10)*, ohne das Payment zu berücksichtigen.

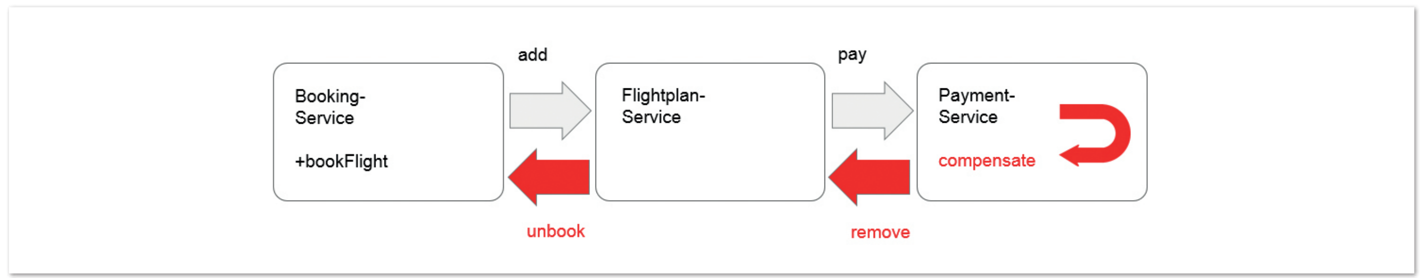


Abbildung 9: Flugbuchung mit Kompensation (Quelle: Matthias Koch)

```java
@Compensate
@Path("/compensate")
@PUT
public Response compensatePayment(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  //Service is responsible for availability of data for compensation.
  PaymentInfo paymentInfoCompensate = getPaymentInfo(lraId);
  // Check here if payment was accomplished.
  if (!undoPay(paymentInfoCompensate)) {

    // Can not be resolved immediate.
    backOut(lraId, paymentInfoCompensate);
  }
  return Response.accepted().build();
}
```

Listing 6: Kompensation eines Participant (Quelle: Matthias Koch)

```java
@Compensate
@Path("/compensate")
@PUT
public Response removeFlightlist(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Service is responsible for availability of data for compensation.
  PassengerInfo passengerInfoCompensate = getPasssengerInfo(lraId);
  // Check if passenger was added first.
  if (!removePassenger(passengerInfoCompensate)) {

    backOut(lraId, passengerInfoCompensate);
  }
  return Response.accepted().build();
}
```

Listing 7: Kompensation eines Participant (Quelle: Matthias Koch)

```java
@Compensate
@Path("/compensate")
@PUT
public Response compensateBooking(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Service is responsible for availability of data for compensation.
  BookInfo bookInfoCompensate = getBookInfo(lraId);
  if (!unbookFlight(bookInfoCompensate)) {

    backOut(lraId, bookInfoCompensate);
  }
  return Response.serverError().build();
}
```

Listing 8: Abschluss der LRA mit Kompensation (Quelle: Matthias Koch)

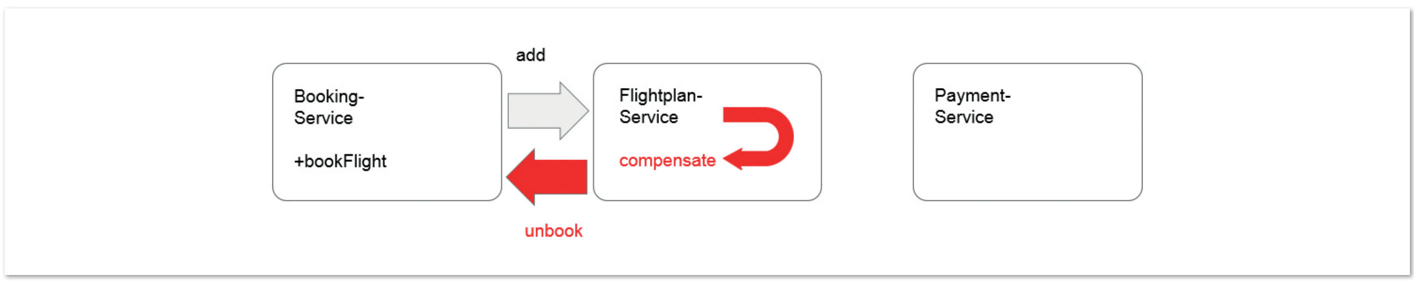


Abbildung 10: Flugbuchung mit vorzeitiger Kompensation (Quelle: Matthias Koch)

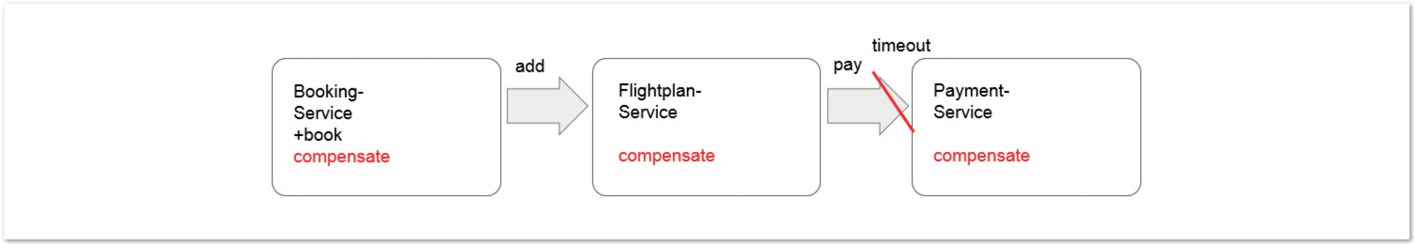


Abbildung 11: Flugbuchung mit Timeout (Quelle: Matthias Koch)

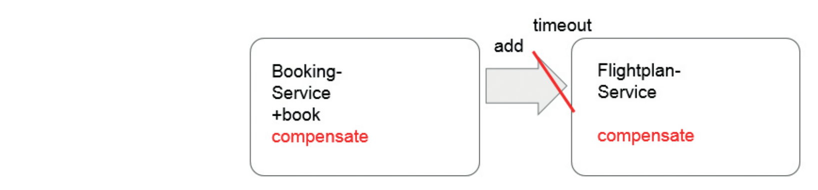


Abbildung 12: Flugbuchung mit Timeout (Quelle: Matthias Koch)

Außerdem kann ein Timeout in einem verteilten System immer auftreten *(siehe Abbildung 11)*. Wichtig ist hierbei, dass auch das Payment gegebenenfalls kompensiert wird, und das bereits als Erstes. Tritt der Timeout schon früher ein *(siehe Abbildung 12)*, ist das Payment gar nicht mit involviert.

Fazit

Abschließend werden noch einmal die Ziele betrachtet. Es wurde kein Fall vergessen, was durch das State-Modell der LRAs gewährleistet wird. Und letztendlich wird auch immer wieder ein global konsistenter Zustand erreicht. Den Code-Beispielen ist zu entnehmen, dass an keiner Stelle Daten gelockt werden, und das Framework kommt nahezu ohne Boilerplate-Code aus.

Zusammenfassend lässt sich feststellen, dass es leider immer noch wenige Lösungsansätze gibt, um Transaktionen in verteilten Microservices zu implementieren. Klassische Lösungen, die stark auf dem ACID-Prinzip basieren, wie zum Beispiel X/Open XA, sind nicht sinnvoll anwendbar.

Das Saga-Pattern und insbesondere die betrachtete Eclipse-Microprofile-LRA sind dagegen eine mögliche Variante, Transaktionen in Microservices zu implementieren. Allerdings gilt auch hier die Einschränkung, dass Transaktionen mit Echtzeit-Charakter oder starken Konsistenzansprüchen nicht sinnvoll implementierbar sind.

Quellen

[1] Dr. Eric Brewer (2000): *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing.* ACM New York, New York, USA.

[2] Seth Gilbert & Nancy Lynch (2002): *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.* ACM SIGACT News Volume 33 Issue 2, ACM New York, New York, USA

[3] XA-Spezifikation: *http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf*

[4] Héctor García-Molina, Kenneth Salem: *Sagas.* *ftp://ftp.cs.princeton.edu/reports/1987/070.pdf*

[5] Arnon Rotem-Gal-Oz (2012): *SOA Patterns.* Manning, Shelter Island, New York, USA

[6] Saga Pattern: *https://cararuseugeniu.blogspot.com/p/saga-pattern.html*

[7] Narula, McCaffrey, Kingsbury: *Distributed Sagas*: *https://github.com/aphyr/dist-sagas/blob/master/sagas.pdf*

[8] Eventuate Tram Saga: *https://github.com/eventuate-tram/eventuate-tram-sagas*

[9] Axon Framework: *https://axoniq.io/*

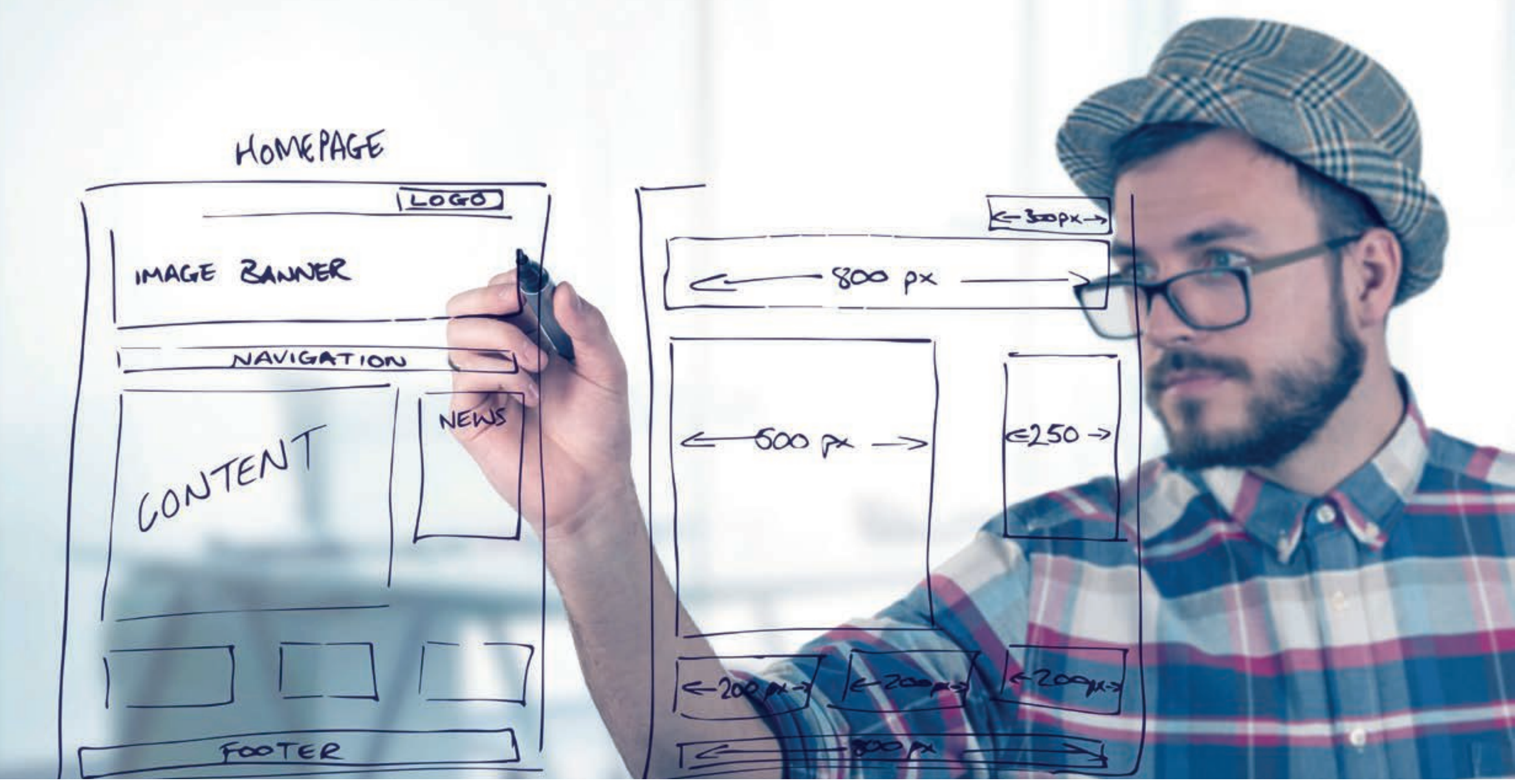
[10] Long Running Actions for MicroProfile: *https://github.com/eclipse/microprofile-lra*

Matthias Koch
sidion
matthias.koch@sidion.de

Matthias Koch ist Diplom-Informatiker und arbeitet als Senior Software Developer bei der Firma sidion. Er entwickelt seit mehr als 20 Jahren Geschäftsanwendungen für unterschiedlichste Kundenprojekte – vor allem mit Java. Bei seinen Code- und Architekturreviews setzt er auf das Prinzip „Clean Code". Ferner interessiert er sich für funktionale Programmierung, Secure Coding und Continuous Delivery. Außerdem ist er Dozent an der Hochschule für Technik in Stuttgart.

Markus Grabert
sidion
markus.grabert@sidion.de

Markus Grabert ist Diplom-Informatiker und arbeitet als Senior Projektmanager bei der Firma sidion. Seit über 20 Jahren ist er in verschiedenen Rollen, als Netzwerk- und Unix-Admin, meist aber als Softwareentwickler tätig. Derzeit arbeitet er in Kundenprojekten als Softwarearchitekt mit Java. Seine Schwerpunkte liegen in und rund um die Themen Agile Project Management, Java, DevOps und Linux.



Langfristige, effiziente Web-UI-Entwicklung für Unternehmensanwendungen

Björn Müller, CaptainCasa GmbH

Werfen wir einen Blick auf die Web-UI-Frameworks, die allein von Google initiiert wurden: Im Jahr 2006 wurde GWT – Google Web Toolkit – gestartet, 2009 war die Geburtsstunde von AngularJS. Im Jahre 2012 folgte der Versuch, Dart als Programmiersprache für das Web zu etablieren. Seit 2015 gibt es Polymer und 2016 wurde das Framework Angular ins Rennen geschickt. Wir alle wissen: Google ist nicht der einzige aktive Player im Bereich von Web-UI-Frameworks, die Aufzählung ließe sich beliebig fortsetzen. Es ist also weiterhin eine spannende Zeit für Web-UI-Entwickler! Noch spannender – dieses Mal vielleicht nicht ganz so positiv gemeint – ist die Zeit für Unternehmen, die heute in die langfristige Entwicklung von Softwareanwendungen investieren. Die dürfen es nämlich finanziell ausbaden, wenn ein verwendetes Framework „out" ist und vom Hersteller nicht mehr aktiv gefördert wird.

Welche Bedeutung spielt Langfristigkeit?

Nicht für alle Arten von Anwendungen ist der Aspekt Langfristigkeit relevant. Spielt diese keine Rolle, dann sind auch die folgenden Überlegungen redundant und man benötigt keine explizite, eigene UI-Architektur, sondern kann direkt mit den Architekturvorgaben des Wunsch-UI-Frameworks loslegen. In diesem Fall sind beispielsweise JavaScript und React oder auch TypeScript mit Angular eine Option, um direkt zu beginnen.

Bei Unternehmensanwendungen spielt die Langfristigkeit in der Regel eine große Rolle. Ihre Entwicklung dauert lange und die Nutzungszeit beim Kunden ist noch länger. Schreibt man heute beispielsweise eine Anwendung in den Bereichen Lagerlogistik, Behördenverwaltung oder Personalverwaltung, dauert es ein bis zwei Jahre, bis der erste Release für den Kunden produktionstauglich ist. Danach beginnt der Vertrieb – Jahr für Jahr kommen also neue Kunden hinzu. Die Nutzungszeit beim Kunden beträgt zehn Jahre oder mehr. Schnell erreicht man einen Gesamtlebenszyklus der Anwendung von über 15 Jahren. Hinsichtlich der Wahl der UI-Architektur ergeben sich somit entsprechende Anforderungen, die einerseits in Richtung Langfristigkeit und andererseits in Richtung Effizienz gehen. In *Abbildung 1* ist ein Beispiel für eine Sachbearbeiter-Anwendung zu sehen.

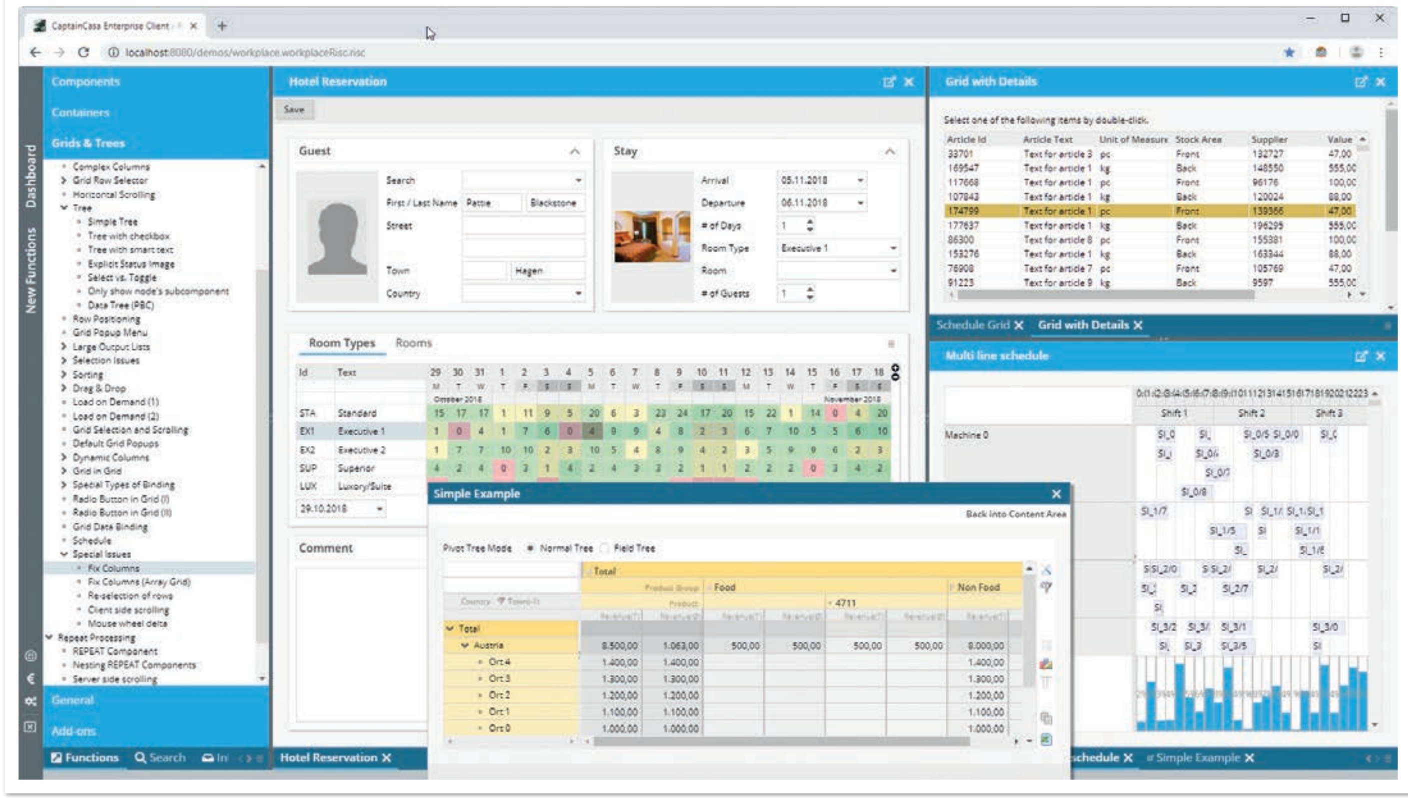


Abbildung 1: Sachbearbeiter-Anwendung im Web (Quelle: Björn Müller)

Framework-Volatilität muss durch Architektur aufgefangen werden! Kehrt im Bereich von Web-UI-Frameworks jemals Ruhe ein? Davon ist aufgrund vergangener Erfahrungen nicht auszugehen. Somit gilt es, für die Teile der Entwicklungen, die ein besonderes Augenmerk auf Langfristigkeit und Effizienz legen, entsprechend zu reagieren, indem Architekturen definiert werden, die eine enge Kopplung zwischen Interaktionslogik und Web-UI-Framework verhindern.

Hier geht es allerdings nicht um die altbekannte Entkopplung der „Oberfläche" von der „Anwendungslogik", sondern um die Entkopplung der „Rendering-Logik" von der „Interaktionslogik" innerhalb der Oberfläche. Im Idealfall sind beide so voneinander entkoppelt, dass die Rendering-Logik alle Abhängigkeiten zum konkreten Web-UI-Framework kapselt – und die Interaktionslogik nur noch die anwendungssemantische Steuerung der Interaktion enthält. Wie ein Button gerendert wird, wie er optisch disabled oder enabled wird – all das weiß die Rendering-Logik. Warum er überhaupt enabled oder disabled ist, wie auf seinen Druck hin reagiert wird, beinhaltet die Interaktionslogik.

Prinzipien einer solchen Architektur

Das Hauptprinzip einer solchen Architektur ist die Entkopplung von Rendering und Interaktion. Um diese zu erreichen kommt man nicht umhin zwischen den beiden Ebenen ein abstrahiertes Control-Modell zu definieren: Die Interaktionslogik baut Dialoge in diesem Control-Modell auf – typischerweise als Hierarchie von Control-Objekten. Die Rendering-Logik setzt das Control-Modell in konkrete, physische Controls um. Entsprechend werden Events von der physischen Kontrolle abgegriffen und an das abstrahierte Control-Objekt weitergegeben. Die Interaktionslogik der Anwendung arbeitet also mit einem abstrakten Control-Baum, der durch die Rendering-Logik materialisiert wird. Die Vorteile einer solchen Architektur liegen auf der Hand: Nur noch einige UI-Spezialisten kümmern sich um die Rendering-Logik – und müssen sich mit der durchaus komplexen Web-UI-Entwicklung befassen. Die Interaktionslogik-Entwickler hingegen arbeiten auf einer abstrahierten UI-Schicht und können sich auf die Fachlichkeit ihrer Interaktion konzentrieren. Somit muss nicht jeder Entwickler ein Web-UI-Fachmann sein.

Client-zentrisch oder Server-zentrisch?

Die große Frage lautet nun: Auf welcher Seite läuft die Interaktionsverarbeitung – auf Client- oder auf Server-Seite?

Client-zentrische UI-Architektur

Der heutige Mainstream liegt hier auf der Client-Seite *(siehe Abbildung 2)*. Im Browser läuft ein JavaScript-Programm als Single Page Application. In diesem Programm wird sowohl die Rendering-Logik als auch die Interaktionslogik abgearbeitet. Die Interaktionslogik greift beispielsweise per REST-API auf die Server-seitige Anwendungslogik zu. Diese Architektur ist zugegebenermaßen eine sehr einfache – zunächst im positiven Sinne! Sie folgt dem pauschalen Denken „hinten die APIs und die Business-Logik", vorne „alles rund ums Frontend". Sie hat allerdings auch gewichtige Nachteile und birgt Risiken, die zum Vorschein kommen, wenn die Anwendung wächst und wächst. Vom Modell her entspricht diese Architektur dem, was man in den 90er Jahren „Fat Client" genannt hat: Der Frontend-Client ist relativ mächtig und seine Größe steigt mit der der Anwendung.

Die Interaktionsentwicklung im Frontend muss sehr genau darauf achten, wie und wann sie welche Daten vom Server holt: Die Anzahl der Roundtrips muss minimiert werden, Puffer im Client müssen aufgebaut werden, die APIs der Server-Logik müssen angepasst werden (aus „n" Aufrufen muss ein einziger gemacht werden). An-

sonsten passiert genau das, was vielen Fat-Client-Anwendungen in den 90ern auch passierte: Sobald zwischen Server und Client eine langsame Leitung liegt, werden die Anwendungen erschreckend langsam. Zudem ist die Verteilung der Anwendungslogik immer eine Herausforderung. Ein Beispiel: Macht man eine Preisberechnung über einen API-Call im Server mit entsprechendem Roundtrip? Oder berechnet man den Preis lieber schon einmal vorne im Client vor, damit das Ergebnis direkt bei der Eingabe ohne Roundtrip angezeigt wird? Und wenn man es vorberechnet: Wie stellt man sicher, dass die Vorberechnung im Client genau dem entspricht, was die Logik im Server macht? Man sieht also: Sobald die Anwendung wächst, wachsen auch die Probleme!

Server-zentrische UI-Architektur
Die Server-zentrische Architektur ist zunächst einmal etwas komplexer *(siehe Abbildung 3)*. Das Grundprinzip ist dennoch einfach: Im Client läuft ein (JavaScript-)Programm als Single Page Application. Dieses Programm ist ein generischer Renderer, der nichts anderes macht, als Dialogbeschreibungen, die vom Server kommen, auszurendern und Ereignisse (Maus-Clicks, Feldeingaben etc.) geordnet an den Server zu geben. Der Renderer weiß nicht, ob das, was er vom Server als Layout-Beschreibung erhält, nun ein Bestell-Dialog oder ein Stammdaten-Dialog ist – er ist semantisch vom Anwendungsinhalt entkoppelt. Früher hat man einen solchen Renderer als „Thin Client" bezeichnet. Dieser Begriff war allerdings missverständlich, weil unter „thin" allzu oft ein Mangel an UI-Features verstanden wurde.

Konkret kann das so aussehen: Der Client startet, kommuniziert zum Server und erhält den ersten Dialog, beispielsweise als XML-Layoutbeschreibung. Der Client stellt diesen Dialog dar – der Benutzer arbeitet nun mit diesem Dialog. Zu bestimmten Zeitpunkten (zum Beispiel, wenn der Benutzer einen Button drückt oder eine Eingabe in einem wichtigen Feld tätigt) werden die Datenänderungen und das Event per Request an den Server übertragen – und dort an die Interaktionslogik weitergereicht. Die Interaktionslogik arbeitet nun, entscheidet, ob der aktuelle Dialog weiter eingegeben oder ob er verändert werden soll, (beispielsweise durch das Einblenden neuer Felder) und berechnet für bestimmte Felder neue Werte. Das Ergebnis wird per XML wieder zum Client geschickt, sodass dieser nun weiß, dass ein Update des aktuellen Dialoges vollzogen werden muss. Das XML muss dabei nur die Änderungen am Layout enthalten, es ist also nicht notwendig, jedes Mal die gesamte Dialogbeschreibung zu senden.

Man sieht also: Das Rendering findet vorne im Client statt, die Anwendungsverarbeitung der Interaktion hinten im Server.

Maximale Entkopplung
Der naheliegendste Vorteil der Server-zentrischen UI-Verarbeitung ist der einer maximalen Entkopplung des Rendering-Geschehens von der Anwendungsverarbeitung: Der Rendering-Client ist nicht nur logisch, sondern sogar physikalisch per explizitem Netzwerkprotokoll abgetrennt. Entsprechend kann auf Basis dieses Protokolls die Technologie des Rendering-Clients frei gewählt werden. Technologische Wandel in der Web-UI-Welt schlagen nicht auf die Anwendungsverarbeitung durch. Der Rendering-Client kann vor der Anwendung ausgetauscht werden, ohne dass die Anwendung davon betroffen ist.

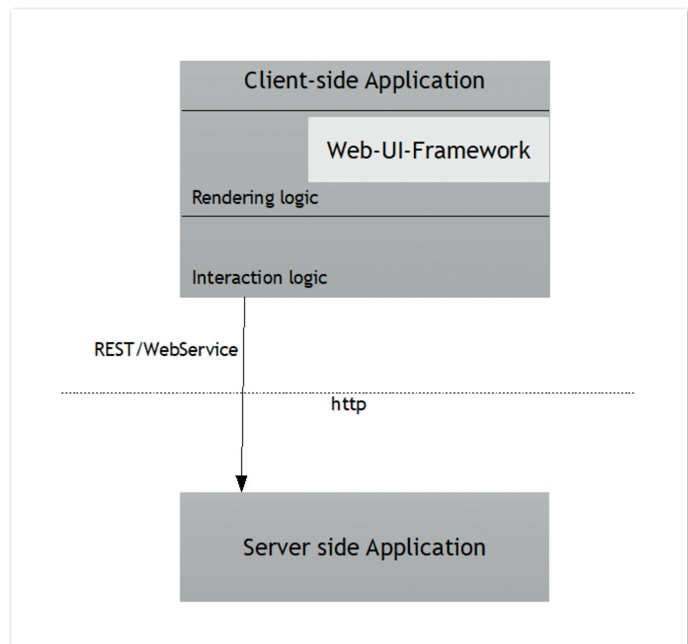


Abbildung 2: Client-zentrische UI-Architektur (Quelle: Björn Müller)

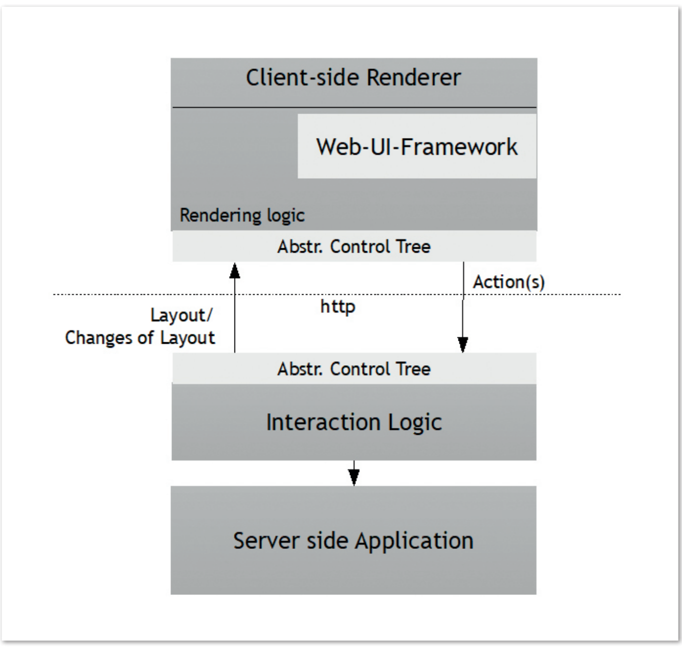


Abbildung 3: Server-zentrische UI-Architektur (Quelle: Björn Müller)

Dafür, dass dies nicht nur theoretisches Gedankengut ist, gibt es konkrete Beispiele: Noch immer läuft eine nicht ganz kleine Anzahl von Anwendungen auf „alten" Terminalprotokollen (3270 und folgende) – die Grün-auf-schwarz-Dialoge haben es also geschafft, sich seit den frühen 80ern bis heute zu bewahren. Wesentlich präsenter ist das Beispiel des HTML-Browsers selbst, der beweist, dass auf einem definierten Protokoll unterschiedlichste Client-Implementierungen langfristig auf unterschiedlichste Server zugreifen können. Ich möchte auch den CaptainCasa Enterprise Client erwähnen. Ein Framework, das auf Basis einer Entkopplung im Jahre 2007 mit einem Java Swing Client startete, dann 2012 einen JavaFX-basierten Client anbot und im Jahre 2017 auf einen Web-basierten Client umstieg – ohne Änderung der Anwendungsverarbeitung im Server, allein durch den Austausch des vorgeschalteten UI-Clients.

Effizienz-Vorteile

Die Server-zentrische Verarbeitung bietet aufgrund ihrer Architektur eine Menge an Effizienz-Vorteilen. Zunächst einmal ist die Interaktionsentwicklung auf der Server-Seite vom direkten Umgang mit Web-Technologien entbunden. Die Anforderungen an das Know-how des Anwendungsdialog-Entwicklers sind also wesentlich geringer als in einer direkten Web-Frontend-Entwicklung. Der Rendering-Client wird von „Web-UI-Profis" entwickelt und betrieben, das Testen neuer Browser-Versionen ist unabhängig vom Test der Anwendung.

Die Interaktionsverarbeitung findet im Server statt und ist deswegen unabhängig von den Programmiersprachen des Clients. Insbesondere findet sie nicht in JavaScript statt – sondern zum Beispiel im Java-Umfeld mit „ganz normalem Java".

Die Interaktionsentwicklung findet nah zur eigentlichen Anwendung statt. Während im Client-zentrischen Ansatz die eigentliche Anwendungslogik nur über Remote-APIs mit potenziell langsamer Leitung zugreifbar ist, kann im Server-zentrischen Ansatz direkt und performant auf die Anwendungslogik zugegriffen werden. Dies bedeutet wiederum, dass die Verteilung der Logik einfacher zu strukturieren ist: Ein Vorziehen von Logik in den Client aus Performance-Gründen ist nicht notwendig.

Die Roundtrips zwischen Client und Server sind klar geregelt, sowohl von ihrer Häufigkeit als auch vom Datenvolumen her. Immer, wenn in der Dialogverarbeitung eine semantische Prüfung oder Verarbeitung notwendig ist, muss ein Roundtrip zwischen Client und Server erfolgen. Die Roundtrip-Größe ist bestimmt und begrenzt durch das, was auf dem Bildschirm dargestellt wird, da zwischen Client und Server Layout-Beschreibungen ausgetauscht werden – und keine semantischen Anwendungsinhalte. Die Größe des Rendering-Clients ist unabhängig von der Größe der Anwendung. Der Rendering-Client wird nicht „dicker", wenn die Anwendung umfangreicher wird.

Wo liegen die Nachteile des Server-zentrischen Ansatzes?

Beginnen wir mit der technologischen Sicht: Der Server-zentrische Ansatz braucht einen Server – und dieser wird bei Interaktionen des Benutzers auch regelmäßig angesprochen. Phasen, in denen der Benutzer autark – ohne Server – mit dem Frontend interagiert, kann es also per se nicht geben – die Anwendung steht und fällt mit der Verfügbarkeit eines Netzwerkes. Weiterhin muss man sich bewusst sein, dass eine Server-seitige Interaktionsverarbeitung mit einem Server-seitigen Halten von „State" verbunden ist. Zu einer Client-Anmeldung gehört auf dem Server eine zugeordnete Session. Wie groß der Session-State konkret ist, hängt von der Implementierung der Anwendung ab und bestimmt wesentlich die Skalierbarkeit des Anwendungssystems. Bei einer Client-zentrischen Verarbeitung wird auf den Server normalerweise „stateless" zugegriffen, was der potenziellen Skalierbarkeit natürlich zugutekommt. Die Skalierbarkeit kann im Server-zentrischen Ansatz durch Hinzunahme von Server-Knoten und Lastverteilung erhöht werden. Eine Skalierbarkeit zu „Zehntausenden" von Benutzern ist nicht problematisch, eine Skalierung zu „Millionen" von Benutzern dagegen stellt ein Problem dar. Ein Beispiel: Es gibt frei verfügbare Frameworks zum Server-zentrischen Entwickeln des Web-UI von Anwendungen.

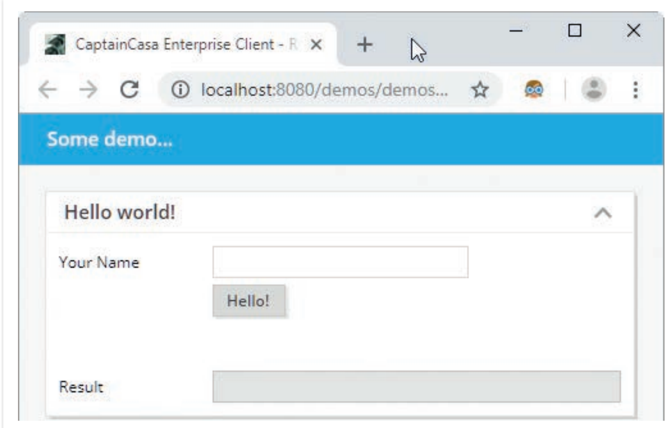


Abbildung 4: „Hello World" (Quelle: Björn Müller)

Das prominenteste ist „Vaadin", im deutschsprachigen Raum ist der „CaptainCasa Enterprise Client" bekannt. Daneben gibt es eine Reihe von Implementierungen des Server-zentrischen Ansatzes innerhalb größerer Software-Unternehmen. Wie sieht die konkrete Entwicklung einer „Hello World!"-Seite mit einem solchen Framework – hier „CaptainCasa Enterprise Client" – aus *(siehe Abbildung 4)*?

Der Entwickler definiert die Seite auf dem Server auf Basis einer Control-Bibliothek – entweder dynamisch per Java-API oder per XML-Definition *(siehe Listing 1)*.

Im Layout sind die Controls hierarchisch angeordnet. Innerhalb der Attribute gibt es sogenannte Expressions, die auf eine Server-seitige Java-Klasse zeigen. Felder sind mit Java-Properties verbunden, Buttons mit Java-Methoden *(siehe Listing 2)*.

Die Entwicklung der Web-Oberfläche geschieht somit ohne eine Zeile Programmierung von JavaScript und ohne jegliche Kenntnis von HTML-/CSS-/Web-Frameworks. Mehr ist nicht notwendig – alles Weitere geschieht automatisch zur Laufzeit: Im Browser wird ein JavaScript-Programm gestartet (der Rendering Client), dieser fordert vom Server per HTTP-Kommunikation das aktuelle Layout mit seinen Daten an – und rendert es im Browser aus. Feldeingaben des Benutzers werden in der Regel im Client gepuffert und führen nicht zu einem sofortigem Roundtrip; dieser erfolgt erst, wenn der Benutzer den „Hello"-Button drückt. Die gepufferten Daten und das Event werden per HTTP-Kommunikation zum Server transferiert, dort werden zunächst alle Daten übergeben (set, set, set), danach wird das Event ausgeführt (onHello) und anschließend wird geprüft (get, get, get), ob sich an den Daten etwas geändert hat. Änderungen werden zum Client zurücktransferiert und der Client führt ein Update seines Web-UI durch. Kleiner Exkurs für „Insider": Die Dialogdefinition und das Binding zwischen Dialog und Programm erinnert an JavaServer Faces (JSF), denn CaptainCasa setzt in der Server-Verarbeitung JSF ein, allerdings nicht auf die klassische Art und Weise. Im klassischen JSF-Einsatz erzeugt der Server HTML, das an den Browser geht – im CaptainCasa-Einsatz erzeugt der Server XML, das an den JavaScript Rendering Client geht, der im Browser als Single Page Application läuft.

Nutzungsszenarien und Fazit

Die am Anfang des Textes beschriebenen Problemstellungen bezüglich Langfristigkeit und Effizienz im Bereich der Web-UI-Ent-

```
<t:rowtitlebar text="Some demo..." />
<t:rowbodypane >
    <t:row >
        <t:foldablepane rowdistance="5" text="Hello world!" width="100%" >
            <t:row>
                <t:label text="Your Name" width="120" />
                <t:field text="#{d.DemoHelloWorld.name}" width="200" />
            </t:row>
            <t:row>
                <t:coldistance id="g_ccpreview_9" width="120" />
                <t:button actionListener="#{d.DemoHelloWorld.onHello}" text="Hello!" />
            </t:row>
            <t:rowdistance height="30" />
            <t:row id="g_ccpreview_12" >
                <t:label text="Result" width="120" />
                <t:field enabled="false" text="#{d.DemoHelloWorld.output}"
                         width="100%" />
            </t:row>
        </t:foldablepane>
    </t:row>
</t:rowbodypane>
<t:rowstatusbar/>
```

Listing 1

```
public class DemoHelloWorld extends DemoBasePageBean
{
    String m_name;
    String m_output;

    public DemoHelloWorld()
    {
    }

    public void setName(String value) { m_name = value; }
    public String getName() { return m_name; }

    public String getOutput() { return m_output; }

    public void onHello(ActionEvent ae)
    {
        if (m_name == null)
            m_output = "No name set.";
        else
            m_output = "Hello World, "+m_name+"!";
    }
}
```

Listing 2

wicklung von Unternehmensanwendungen werden durch den Server-zentrischen Ansatz gelöst. Die Aufgabenteilung zwischen einer Rendering-Entwicklung und einer Interaktions-Entwicklung ergibt für größere Vorhaben mit Hunderten von Dialogen mehr als Sinn und sorgt dafür, dass einem Framework- und Control-Wildwuchs am Client Einhalt geboten wird. Die Skalierungsfähigkeit ist mehr als ausreichend für Unternehmensanwendungen, die vornehmlich von Mitarbeitern bedient werden.

Umgekehrt gilt: Wo Millionen von Nutzern auf die Anwendung warten, wo explizit gewollt ist, dass Anwendungsentwickler auf alle Web-UI-Features direkt zugreifen können und/oder wo autarke Client-Szenarien zu finden sind, ist der Client-zentrische Ansatz sinnvoller. In diesen Szenarien darf man dann eben auch nicht erstaunt sein, wenn das Web-UI-Framework, das heute „in" ist, in drei bis vier Jahren durch einen neuen „Hype" abgelöst wird. Wichtig ist, beide Architekturen zu kennen, zu verstehen und gezielt einzusetzen, statt Framework- oder Prototyp-getrieben automatisch in eine Architektur hineinzurutschen.

Björn Müller
CaptainCasa GmbH
bjoern.mueller@CaptainCasa.com

Björn Müller ist seit 2001 im Bereich Oberflächentechnologien und -architekturen für Geschäftsanwendungen tätig, im selben Jahr hat er auch die Casabac Technologies GmbH gegründet (seit 2005 Bestandteil der Software AG). 2007 gründete er die CaptainCasa-Community und entwickelte das Client-Framework CaptainCasa Enterprise Client. Im Jahr 2016 entwickelte er die RISC-HTML-Methode als langfristig stabile Grundlage für anspruchsvolle Web-UI-Entwicklungen.

ORAWORLD
Das e-Magazine für alle Oracle-Anwender!
EOUC
EMEA ORACLE USERGROUP COMMUNITY
• Spannende Geschichten aus der Oracle-Welt
• Technologische Hintergrundartikel
• Leben und Arbeiten heute und morgen
• Einblicke in andere User Groups weltweit
• Neues (und Altes) aus der Welt der Nerds
• Comics, Fun Facts und Infografiken
Jetzt Artikel einreichen oder Thema vorschlagen!
Jetzt e-Magazine herunterladen
www.oraworld.org

„So entwickelt sich eine App weiter: Man beobachtet, was die User brauchen und was für sie wichtig ist"

Michal Harakal ist langjähriger Experte in der App-Entwicklung und fokussiert sich besonders auf Consumer-Apps, die sich an bestimmte Zielgruppen richten und auf die Bedürfnisse des User abgestimmt sind. Im Interview mit Sanela Lukavica, ehemalige Mitarbeiterin der DOAG, verrät er, worauf es bei einer guten App wirklich ankommt. Außerdem geht er auf die Unterschiede bei der Programmierung mit nativen und webbasierten Apps ein.

Du hast nach 13 Jahren im Maschinenbau 2010 deinen beruflichen Wechsel in die mobile Welt von Android vollzogen. Wie hat sich deiner Meinung nach die App-Entwicklung in den letzten Jahren verändert?

Michal Harakal: Ich bin von Anfang an bei Android dabei und durfte die Entwicklungen mitverfolgen. Am wichtigsten ist sicher der Punkt, dass die Entwicklung professionalisiert wurde. Am Anfang ging es um die Ideen, die man an mobilen Geräten, die völlig neue Möglichkeiten mitbrachten, umsetzen wollte. Zum Beispiel, wie die Bedienung einer App auf einem Gerät mit einem sehr kleinen Bildschirm aussehen soll. Über die Jahre hat sich das geändert: Die Apps werden jetzt auch in größeren Teams entwickelt.

Die Qualitätssicherung ist wichtig geworden. Das, was man aus der normalen Softwareentwicklung kennt, Themen wie Software-Architektur, Testabdeckung, Testautomatisierung oder statische Code-Analysen, also das, was eigentlich bei Backend-System-Software gang und gäbe war, ist auch in die Entwicklung von mobilen Apps eingeflossen. Und das betrifft beide Plattformen, sowohl Android als auch iOS. Aus meiner Sicht haben wir manchmal das Rad quasi neu erfunden, das heißt wir haben neue Lösungen gesucht und gefunden, die für die Entwicklung von mobilen Apps spezifisch waren. Inzwischen ist es so, dass die Anforderungen an das Programmieren einer guten App sich nicht viel von einer Backend-System-Software unterscheiden.

Grundsätzlich heißt es ja, dass das Kontrollsystem von Apple um einiges strenger sei als das von Android. Wie ist deine Erfahrung: Veröffentlicht man bei Google schneller Apps als bei Apple?

Michal Harakal: Die Handhabung ist tatsächlich unterschiedlich. Die Apps für den App Store gehen durch eine Freigabeprozedur, die aus Sicht der Entwickler nicht wirklich transparent ist, zumindest war es anfangs. Inzwischen ist weitgehend bekannt, was zu tun ist und wann eine App Gefahr läuft, abgewiesen zu werden. Die Qualitätssicherung sowie die Bearbeitungszeiten, die Apple beansprucht, haben sich deutlich verändert. Im Moment geht alles viel schneller vonstatten. Im Store publizieren ist aber wirklich der letzte Schritt und könnte von einem anderem Team als den Entwicklern übernommen werden.

Dennoch geht bei Android vieles schneller. Ist das automatisch auch mit einer größeren Freiheit für dich als Entwickler verknüpft?

Michal Harakal: Bei Android gibt es die Freiheit, dass man die Apps sehr flexibel in den Store stellen kann. Natürlich kann es passieren, dass einem größere Bugs durch die Lappen gehen. Dann ist es von Vorteil, dass man in der Lage ist, die Bugs zügig zu fixen und die neue Version in den Store zu stellen. Das könnte man als Freiheit ansehen. Es führt aber andererseits dazu, dass die Qualität der App vielleicht nicht so gut ist. In der Vergangenheit ist es mir zum Beispiel schon mal passiert, dass sehr viele Nutzer eine Konferenz-App bereits installiert hatten und diese partout nicht aktualisiert haben, obwohl dies nötig gewesen wäre, um ein paar Bugs zu beheben. Zudem wurde über diverse Kommunikationskanäle bekanntgegeben, dass es Fixes gab. Auswertungen haben dann ergeben, dass die Hälfte der Leute weiterhin die erste fehlerhafte Version benutzt haben. Vor diesem Hintergrund ist es einfach wichtig, Apps in den Store zu stellen, die keine Fehler aufweisen. Da ist das Thema Qualitätssicherung absolut wichtig. Natürlich ist diese Freiheit schnell zu agieren von Vorteil, aber in dieser Situation hat sie mir nicht geholfen. Für mich als Entwickler oder Konsument sind es zwei sehr unterschiedliche Verfahren, und ob Google oder Apple zuerst veröffentlichen, spielt für mich als Entwickler keine bedeutende Rolle. Ein Indie-Entwickler, bei dem das direkte Verkaufen von Apps oder InApp-Käufe Einnahmequellen sind, wird das Thema sicher anders sehen. Meine Perspektive unterscheidet sich: Die Apps, die ich bisher entwickelt habe, waren bisher immer ein Teil oder eine Erweiterung eines bestimmten Service, sei es eine Konferenz-App oder eine Festival-App, bei der man auf anderen Wegen den Service monetarisiert.

Michal, dein Fokus liegt auf Consumer-Apps. Du hast zum Beispiel an einer rollstuhlgerechten Karte namens „Wheelmap.org" gearbeitet. Versucht man, als Entwickler dem Konsumenten gerecht zu werden, das heißt beschäftigt man sich eingehend mit den Schwierigkeiten, die dieser tagtäglich hat oder rückst du technische Aspekte in den Vordergrund?

Michal Harakal: Bei den Apps steht in der Tat der Nutzer im Fokus. Die erwähnte App war sehr spezifisch. Es ging darum, den Leuten dabei zu helfen, Restaurants, Büros und so weiter mit barrierefreiem Zugang zu finden. Das ist nun schon etwas länger her. Der Gründer des Wheelmap-Projekts ist ein Rollstuhlfahrer. Er und seine Freunde haben irgendwann gemerkt, dass sie sich immer in demselben Café treffen, um Ideen auszutauschen, weil sie kein anderes kannten, zu dem man als Rollstuhlfahrer einen barrierefreien Zugang hatte. Sie dachten sich also, dass man die Plätze kartographieren könnte, um so auch die anderen Menschen dazu zu bringen, mehr auszugehen. Dadurch konnte man nicht nur das Wissen über passende Orte teilen, sondern es entstand indirekt ein gewisser Druck auf die Betreiber und die Besitzer. Ich habe die Verantwortlichen damals bei einer Open-Street-Maps-Konferenz kennengelernt. Nicht nur die App und die Website haben die Daten verbessert, sondern auch begleitende Projekte. Zum Beispiel haben sie mit Schülern zusammengearbeitet, die Smartphones bekommen und viele Orte kartographiert haben. Was ich aus der User-Perspektive sehr gut fand, war die Kennzeichnung, ob ein Ort rollstuhlgerecht ist oder nicht. Es gibt eine bestimmte DIN-Norm in Deutschland, die exakt vorgibt, wie hoch eine Treppenstufe sein darf, damit sie als rollstuhlgerecht gilt. Sie haben sich stattdessen für ein Ampelsystem entschieden. Grün hieß: alles okay. Gelb: Der Rollstuhlfahrer kommt ohne Schwierigkeiten rein, es kann aber sein, dass es gewisse Hürden vor Ort gibt, wie zum Beispiel eine Toilette, die sich unter einer Treppe befindet. Schließlich rot: Der Zutritt ist erschwert. Somit wurde klargemacht, ob man reinkommt und falls ja, unter welchen Bedingungen. Das ist der Unterschied zu den offiziellen, rechtlichen Vorgaben – eine einfache, pragmatische Lösung.

Ein weiteres Beispiel für die Konsumenten-Orientierung ist der Weg, wie die Daten in der Datenbank landen. Die Daten liegen beim Open-Street-Map-Projekt. Um Datenvandalismus zu vermeiden, lassen sie sich nicht so einfach verändern. Den Benutzern, die ohne einen neuen Account registrieren zu müssen, schnell einsteigen wollten, wurde ein Sammelaccount generiert. Hier wurden die Daten gesammelt, ausgewertet und mit der Datenbank synchronisiert. Später ergab sich auch die Möglichkeit für die Leute, die das seriös gemacht haben und die es wichtig fanden, Credits dafür zu bekommen. Sie konnten sich anmelden und die Eintragungen unter ihrem eigenen Namen veröffentlichen. So entwickelt sich eine App schließlich immer weiter: Man beobachtet, was die User brauchen und was für sie wichtig ist.

Die Zahl der weltweiten Nutzung von Apps ist in der Zeit von 2013 bis heute zurückgegangen. Deutschland ist eine Ausnahme, aber in anderen Ländern, insbesondere in den USA, spielen Apps nicht mehr dieselbe Rolle wie früher. Wie siehst du die zukünftige Entwicklung?

Michal Harakal: Mit Blick auf die nativen Apps ist das eine Frage, die mich in letzter Zeit tatsächlich stark beschäftigt. Sie betrifft mich als Entwickler auf unterschiedlichen Ebenen. Wie werden wir Apps zukünftig entwickeln? Auf einer Seite können in Stores nur qualitativ

hochwertige Apps Erfolg haben. Auf der anderen Seite ist es immer wichtiger, mit welchem Aufwand man eine richtig gute App entwickeln kann. Wie schnell ist man in der Lage, auf Kundenwünsche und Veränderungen zu reagieren? Und werden Apps fehlerfrei zur Verfügung gestellt oder ist der Kunde den Fehlern ausgeliefert? Wenn man eine native App schreiben will, braucht man viel spezielles Wissen.

Das Betriebssystem von Android ist sehr eigenartig, wenn es darum geht, wie man zum Beispiel mit Hintergrundprozessen oder Daten umgeht. Es gibt ein paar generelle philosophische Entscheidungen, die von Anfang an festgelegt und getroffen wurden, um gewisse Ziele zu erreichen. Sehr viele Entscheidungen rollen heute noch technische Grundlagen aus, die aus einer Zeit stammen, als noch nicht so starke CPUs vorhanden waren und die Batterien noch nicht so leistungsstark waren – und die meisten gelten immer noch.
Wenn ich als Entwickler an einem Screen arbeite, muss ich immer daran denken, dass ein Telefonanruf kommen kann, den der User sofort annehmen möchte. Ich muss mich darum kümmern, dass ein Text in einem Eingabefeld, der nur zur Hälfte geschrieben wurde, nicht verschwindet, wenn der Nutzer das Gespräch beendet hat. Das sind die kleinen Details, die man als Nutzer erst erkennt, wenn man mit einer schlechten App konfrontiert wird.

Nativ ist nicht die einzige Möglichkeit, wie man eine mobile App schreiben kann. Webbasierte (HTML5-) Apps sind von Anfang an vorgesehen gewesen. Vor iOS war das iPhone aber die einzige Möglichkeit, das zu tun. Bei Android war es wunderbar zu beobachten, wie Google stark ins Web investierte, zum Beispiel mit der Entwicklung des Chrome-Browsers, oder dass auf den ersten Android-Entwicklerkonferenzen Web-orientierte Talks genauso stark vertreten waren wie die für Android.

Mit Hybrid-Frameworks versucht man, die Details einer mobilen Plattform unter Abstraktion zu verstecken. Sobald ein neues Framework kommt, heißt es: Man muss es einmal schreiben, und es läuft überall. Typische Cross-Platform-Lösungen wie Xamarin haben zwar ihre Nischen gefunden, aber eine größere Verbreitung haben sie nicht erreicht. Trotzdem wird Android weiterhin sehr stark nativ entwickelt. Das hat mit den technischen Grundlagen und den Entscheidungen zu tun, die zu Beginn getroffen wurden. Dass man zum Beispiel Android-Apps in Java schreibt oder jetzt in Kotlin. Aber es zeichnet sich tatsächlich ab, dass man mit der neuen Generation von Cross-Platform-Lösungen wie Flutter oder React native Apps bauen kann, die gut genug sind. Ich sehe es weiterhin trotzdem kritisch, aufgrund von Erfahrungen, die andere bereits gemacht haben, und hinsichtlich der Tatsache, dass man Komplexität und Unterschiede der mobilen Plattformen nicht einfach mit einer zusätzlichen Schicht Framework und dem Einsatz einer weiteren Programmiersprache lösen kann. Ich sehe die Lösung im Code Sharing, und das ist es auch, was ich in Zukunft erwarte. Durch Einsatz einer passenden Software-Architektur kann man Domain- und Business-Logik-Plattformen neutral schreiben und auf beiden Plattformen verwenden. Google hat Kotlin bereits zur „Nummer-eins-Sprache" erklärt. Massive Investitionen sind deutlich, an Werkzeugen, Bibliotheken, Blogs und Beispielen zu spüren. Auch das gesamte Ökosystem mit zahlreichen 3rd-Party-Libraries entwickelt sich in einer unglaublichen Geschwindigkeit. Mit Kotlin Multiplatform und Kotlin Native entsteht ein weiterer wichtiger Bestandteil beim Code Sharing.

Zur Person

Michal Harakal arbeitet seit 2010 als Android Entwickler bei der Deutschen Telekom AG. In die spannende Welt der mobilen Geräte ist er nach langjähriger Berufserfahrung als Entwickler von Mess- und Datenauswertungssystemen sowie Steuerungen für industrielle Maschinen gewechselt. Sein Werkzeugkasten, in dem Java und seit Neuestem auch Kotlin einen besonderen Platz haben, ist vielseitig gefüllt. Bei Android ist er von der ersten Stunde an dabei. Michal teilt gerne seine Erfahrung und ist im Open-Source-Projekten aktiv.

Was spricht gegen webbasierte Lösungen?

Michal Harakal: Für mich funktionieren die webbasierten Lösungen, also HTML5-Apps, die im Browser laufen, nur bedingt oder gar nicht – und zwar aus mehreren Gründen. Der erste Grund sind die Web-Standards selbst. Die Unterstützung für moderne Web-Standards hinkt gerade bei älteren Android-Versionen hinterher und damit schließt man manche Nutzer aus oder es bedarf größeren Aufwands, um sie zu unterstützen. Ein weiteres wichtiges Argument ist die UI. Fühlt sich eine Web-App eher fremd an, müssen für eine schöne Integration in das Betriebssystem große Anstrengungen unternommen werden. Für mich zählt aber die Startzeit. Wenn man eine App mit Webbrowser baut, muss ein Webbrowser starten. Dieser ist aber eine großgewichtige Komponente, die viele Ressourcen verbraucht, sprich Speicher und CPU. Er braucht immer ein paar Hundert Millisekunden, um zu starten. Das bedeutet, man zieht sein Handy aus der Tasche, startet das Handy, startet die App und wartet schon mal eine halbe Sekunde, bis es losgeht.

Wie groß ist denn der Aufwand, App-Starts zu optimieren?

Michal Harakal: Eine App braucht erst mal ein, zwei Sekunden, um zu starten. Bei Apps, die man oft auch unterwegs nutzt, zählt aber jede Sekunde. Wir haben mehrmals komplette Bibliotheken gewechselt, um den App-Start zu beschleunigen und die Software-Entwicklung zu verbessern. Schon damals hat man erkannt, dass die Clean-Code- und SOLID-Prinzipien auch bei mobilen Apps zu einer Source-Code-Qualität führen und „Dependency Injection" benutzt wurde. Wir haben dann in der Android-Entwicklung ein Dependency-

Injection-Framework namens „Roboguice" benutzt, eine Android-spezifische Variante vom „Guice", welche aus der Java-Welt kam. Technisch bedingt, wie die Introspektion („Reflection") in der Laufzeitumgebung auf dem Android implementiert wurde, hat es lange gedauert, bis alle Instanzen von allen notwendigen Objekten erzeugt waren und bis der ganze Dependecy-Baum aufgebaut wurde. Das hat dazu geführt, dass wir zwar einen sauberen Code geschrieben haben, aber die App zu langsam startete. Wenn irgendwann so ein Stand erreicht ist, dass man mit solchen Dingen beschäftigt ist, wie Startseiten von Screens oder wie viele Daten über die Leitung übertragen werden, ist es zu viel. So suchte man immer wieder nach neuen Lösungen, die nicht nur die Aufgabe der Dependency Injection erledigen sollten, sondern auch Android-gerecht sind. Dafür wurde ein riesiger Aufwand betrieben. Als ein weiteres Beispiel dient das Date/Time-API: Mit der JSR 310 wurde ein neues Date/Time-API in Java 8 eingeführt. Um das auf älteren Java-Systemen oder bei älteren Androids zu nutzen, bei denen Java 8 noch nicht unterstützt ist, gab es eine Portierung (Backport). Zusätzlich zu der Java-Portierung gab es noch eine Android-spezifische Variante. Der einzige Grund für die Android-spezifische Portierung war die Beschleunigung des App-Starts. Das ist ein riesiger Aufwand, denn jemand muss das tatsächlich pflegen, releasen und testen. Dennoch ist es die paar Millisekunden allemal wert!

Um Programmiersprachen wird häufig ein gewisser Hype betrieben. Wie ist deine Erfahrung?

Michal Harakal: Ich kann mich noch erinnern, als Scala vor vier, fünf Jahren noch sehr populär war. Das war eine moderne Sprache, etwas Neues, Anderes. Sie hat zu besserem Code mit weniger Abstürzen geführt. Die Android-Community mit Top-Entwicklern hat einen enormen Aufwand betrieben, um Scala auf Android zu bringen. Allerding hat man relativ schnell gemerkt, dass es ohne offizielle Unterstützung schwierig wurde. Es gab mehrere Gründe, die es erschwert haben, Apps wirklich produktiv zu entwickeln. Dann wurde es ruhig um Scala. Im Herbst 2017 habe ich meinen ersten Vortrag über Kotlin besucht. Ich dachte: Wiederholt sich die Geschichte mit Scala oder ist es etwas Neues? Ich wartete also erstmal ab, bis ich selbst Kotlin probiert hatte. Seitdem ist Kotlin auch die offizielle Sprache von Google für Android geworden, was Google nicht nur deklariert, sondern auch mit unterschiedlichen Schritten deutlich untermauert hat. Inzwischen bin ich begeisterter Kotlin-Nutzer und bevorzuge Projekte, die in Kotlin geschrieben werden. Mir persönlich macht es einfach Spaß. Inzwischen fängt Kotlin an, ein Eigenleben zu führen. Als Sprache ist Kotlin sehr stabil. Es kommen täglich neue Libraries dazu und bestimmte Bereiche entwickeln sich immer noch in großen Sprüngen (Kotlin Scripting, Multiplattform, JavaScript oder WebAssembly). Von größter Bedeutung sind aus meiner Sicht Kotlin/Native und Kotlin Multiplatform, denn gerade die Unterstützung für iOS sehe ich als essenziell an. Ich habe vor kurzem in München einen Meetup-Vortrag gehalten und über meine Erfahrungen mit Kotlin Multiplatform berichtet.

Wie schreibt man eine gute App?

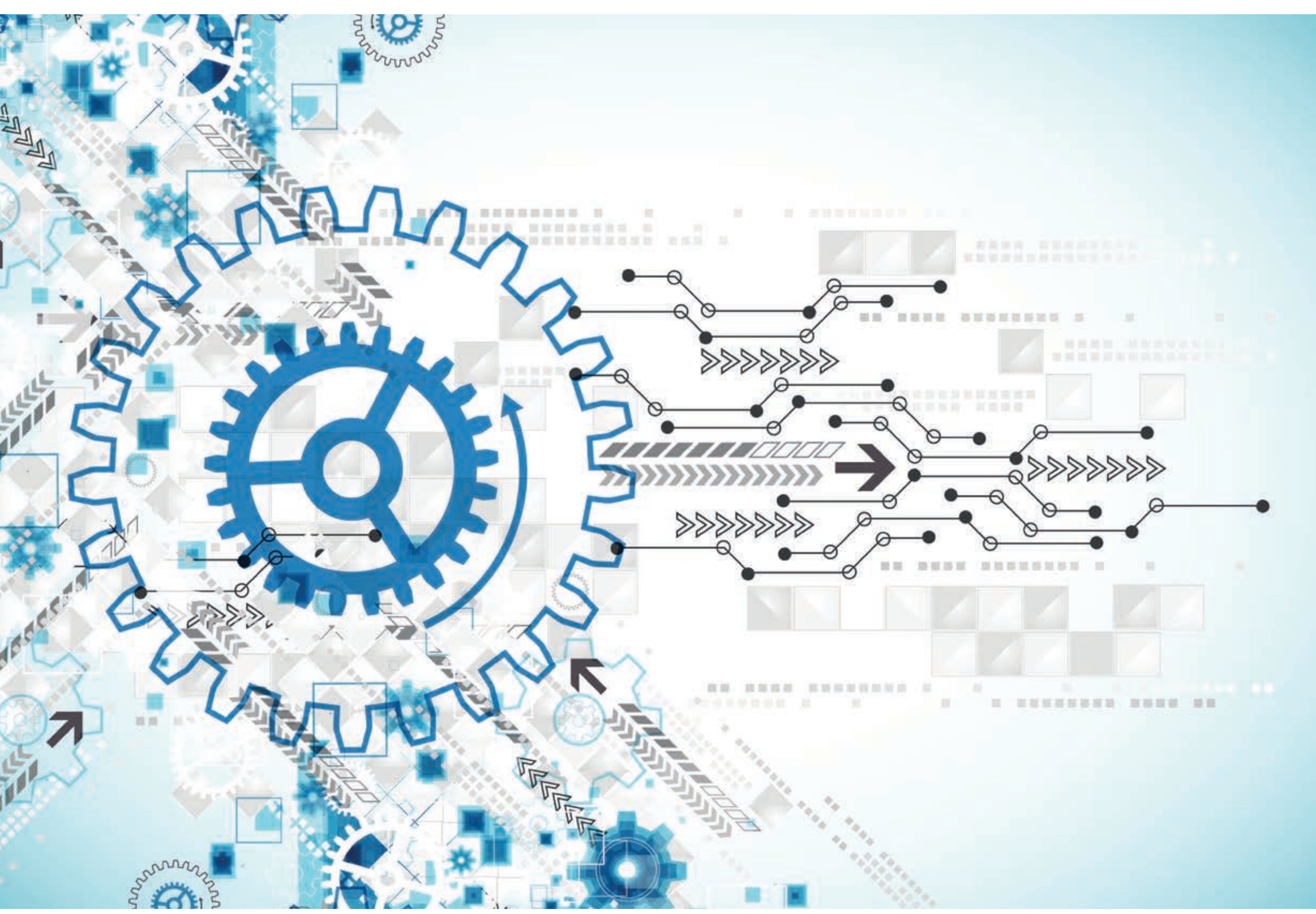
Michal Harakal: Es gibt viele Kriterien, anhand derer man eine App als gut bewerten kann. Am Ende ist es der Benutzer, der die App bewertet. Ich selbst bin davon überzeugt, dass eine gute App auf einer guten technischen Grundlage basieren sollte. Eindeutige Software-Architektur, die zum Entwicklungsteam passt, die Komponenten-basiert ist, mit einer klaren Trennung zwischen Zuständigkeiten, der Darstellungsebene, Domain und Business-Logik. Genauso wichtig sind die Praktiken, wie man entwickelt, angefangen mit Code Reviews, Testautomatisierung bis hin zu einer sehr hohen Testabdeckung. Man muss sich zunächst genau überlegen, was zu tun ist.

Ein wichtiges Merkmal einer guten App ist, dass sie sehr flüssig reagiert. Generell finden viele länger dauernde Operationen im Hintergrund statt, um das UI nicht zu blockieren (um zum Beispiel das besagte Ruckeln beim Scrollen von langen Listen zu vermeiden). Wir als Entwickler verbringen viel Zeit damit, das UI mit der Operation im Hintergrund zu synchronisieren. Möchte man zum Beispiel eine Datei herunterladen, so findet das Herunterladen selbst in einem parallelen Thread und nicht im Haupt-UI-Thread statt. Ich möchte aber trotzdem den Upload-Verlauf anzeigen. Das heißt, die zwei müssen irgendwie miteinander kommunizieren. Das macht die ganze Sache komplex.

Ein weiteres und oft unterschätztes Merkmal ist die App-Größe. Als Nutzer wird man nicht wirklich damit konfrontiert, aber es gibt in Android eine bestimmte Begrenzung der nutzbaren Anzahl von Methoden auf etwa 64.000. Es heißt DEX-Limit. Mit der Zeit nahm die Komplexität der Apps zu, sodass dieses Limit überschritten wurde. Natürlich kamen Lösungen von Google dazu. Diese führten wiederum zu weiteren Herausforderungen. Ich versuche stets zu prüfen, welchen Einfluss die Bibliotheken auf die Größe der App haben. Die Bibliotheken beeinflussen die Startzeit oder auch die Größe der APK (Android Package). Die Größe wiederum hat mit dem Verschwimmen von nativen Apps und Web-Apps zu tun. Bei Android gibt es schon seit zwei Jahren das Konzept von Instant Apps, wo man eine native App ohne den PlayStore installieren kann. Theoretisch könnte man also einen Link auf der Web-Seite haben und die App würde sofort auf das Handy geladen werden und funktionieren, ohne dass man in den Store geht. Es gibt natürlich eine Begrenzung bezüglich der Größe von Apps. Aber dadurch, dass sie so klein sind, werden sie sofort heruntergeladen und gestartet.

Das sind nur einige Beispiele für all die Faktoren, die bei der App-Entwicklung berücksichtigt werden müssen.

Vielen Dank für das Interview, Michal!



Automatisierung und Durchblick mit (Git-)Hub und (Git-)Lab

Kai Schmidt, selbstständig

GitHub und GitLab bieten Funktionen, die über das Versionsverwaltungssystem Git hinausgehen. Während Git eine gute Unterstützung für die Kommandozeile bietet und dessen Nutzung weit verbreitet scheint, werden GitHub und GitLab weitestgehend über das Webinterface bedient. Eine Automatisierbarkeit des Webinterface wäre beispielsweise über Selenium oder TestCafé möglich, jedoch umständlich und zeit- sowie wartungsintensiv. Glücklicherweise bieten beide Portale jeweils ein umfangreiches REST-API, über das viel möglich ist. Außerdem ist es weniger inkompatiblen Änderungen unterworfen als das Webinterface. Gleichzeitig ist die Nutzung des API aber mit einer erhöhten Einarbeitungszeit verbunden. Erleichterungen schaffen hierbei die Kommandozeilen-Tools Hub und Lab, die ein leicht erlernbares CLI zur Verfügung stellen und wesentliche Funktionen der jeweiligen Portale abdecken. Dieser Artikel soll einen Überblick über den Funktionsumfang der beiden Tools geben.

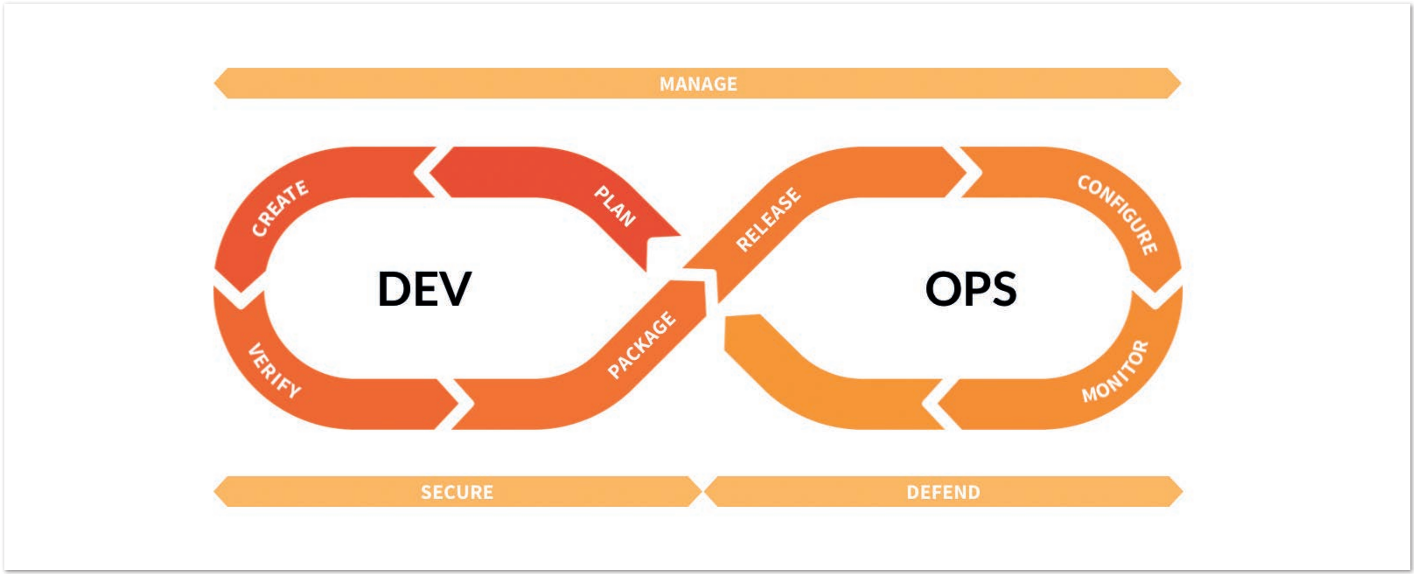


Abbildung 1: Der DevOps Lifecycle mit GitLab [5] (Quelle: Kai Schmidt)

Für uns sehr selbstverständlich gewordene Funktionen wie Pull- beziehungsweise Merge-Requests sind keine nativen Bestandteile von Git und daher nicht im Git-CLI enthalten. Die in GitHub *[1]* verfügbaren zusätzlichen Funktionen werden über das Kommandozeilen-Tool Hub *[2]* erweitert.

Insbesondere GitLab *[3]* führt viele weitere Funktionen ein, die über Git hinausgehen, um eine Abdeckung ihres DevOps Lifecycle *(siehe Abbildung 1)* zu erreichen. Diese umfassen die Planungsphase neuer Feature-Ideen bis hin zum Monitoring, und sie stehen zum Teil nur in kommerziellen Versionen zur Verfügung. Sehr bekannte und in der kostenfreien Version verfügbare Funktionen von GitLab sind das Erstellen von Issues. Sie versehen dieselbigen mit Labeln und Meilensteinen wie zum Beispiel die in GitLab eingebaute CI-Funktionalität. An Hub angelehnt, wird ein gewisser Anteil der GitLab-Funktionen über das Kommandozeilen-Tool Lab *[4]* zur Verfügung gestellt.

Am Anfang steht die Installation

Die Installation von Hub gestaltet sich durch die Paketmanager der unterschiedlichen Betriebssysteme (beispielsweise Homebrew oder Apt) sehr einfach und ist der Readme des Projekts zu entnehmen. Für Windows stehen ZIP-Dateien für den Download von Hub bereit, über Scoop oder Chocolatey können sie installiert werden. Damit lässt sich Hub direkt verwenden. Hub authentifiziert sich dabei auf dem gleichen Weg wie Git (HTTPS mit Benutzername/Passwort oder SSH).

Wer die gleiche Paketmanager-Abdeckung für Lab erwartet, wird etwas enttäuscht. Dennoch ist die Installation laut Readme ein Einzeiler. Unter Ubuntu muss der Einzeiler für eine erfolgreiche Installation leicht umgebaut werden *(siehe Listing 1)*.

Im Gegensatz zu Hub authentifiziert sich Lab über einen API Key, der in GitLab hinterlegt wird. Die zugehörige Konfiguration gestaltet sich durch den dargestellten Link nach dem Start des ersten Lab-Befehls recht einfach. Sollte der GitLab-Host oder der Token falsch sein, funktioniert kein einziger Lab-Befehl mehr. Beide Informationen lassen sich in der Datei `~/.config/lab.hcl` anpassen. Dies ist insbesondere dann praktisch, wenn man sich einen neuen Token generieren oder die gleiche Datei auf einem weiteren Rechner verwenden möchte.

Falls Windows-Veteranen die Nutzung von Scoop scheuen, lässt sich Lab (wie auch Hub) alternativ unter der WSL (Windows Subsystem for Linux) nutzen. In einer Ubuntu-WSL gestaltet sich die Installation wie unter einem Ubuntu-Betriebssystem. Beschreibungen zur Einrichtung der WSL finden sich unter *[6]* und *[7]*.

Erleichterte Git-Funktionalitäten

Durch das Wissen, mit welcher Repository-Plattform das jeweilige Projekt verbunden ist, wird die Nutzung einiger bekannter Git-Befehle erleichtert. Hat man eines der Tools installiert, lässt sich ein Repository durch die Kombination aus Benutzer- und Projektnamen auf den jeweiligen Rechner klonen. Dies macht die komplette URL auf das Repository überflüssig. *Listing 2* zeigt das Klonen der Projekte mit den entsprechenden Tools.

Die vollständige Liste der Funktionen sind bei Hub in der Readme verlinkt. Bei Lab führt ein Aufruf von Lab in der Kommandozeile weiter. Während Hub Man Pages (Befehl `man hub` oder `man hub-clone`) unterstützt, erhält man bei Lab detaillierte Informationen der einzelnen Befehle über das Flag `-h`.

```
$ cd /tmp/
$ wget https://raw.githubusercontent.com/zaquestion/lab/master/install.sh
$ chmod u+x ./install.sh
# Falls curl noch nicht installiert sein sollte
$ sudo apt install curl
$ ./install.sh
```

Listing 1: Installation von Lab auf Ubuntu

```
# "git clone git://github.com/github/hub.git" wird zu
$ hub clone github/hub

# "git clone git@gitlab.com:gitlab-org/manage.git" wird zu
$ lab clone gitlab-org/manage
```

Listing 2: Klonen eines Repository mit Hub beziehungsweise Lab

```
1 function localBranchToMR {
2   if git rev-parse --is-inside-work-tree > /dev/null 2>&1; then
3     git checkout $1
4     git push -u origin $1
5     lab mr create origin develop -d
6   else
7     echo "Not a git repository"
8  fi;
9 }
```

Listing 3: Funktion zur Erstellung eines Merge-Requests aus einem lokalen Branch

Die aus Git bekannten, erweiterten Formen lassen sich sowohl für Hub als auch für Lab grob in die folgenden Kategorien einteilen:

- Leichtere Adressierbarkeit des Remote (wie das bereits beschriebene `clone`)
- Einfachere Unterstützung verschiedener Remotes
- Arbeiten mit Forks, Pull- beziehungsweise Merge-Requests und Submodules (auf diese Weise kann ein Check-out direkt auf einem Pull-Request erfolgen: `hub checkout https://github.com/myuser/myproject/pull/1`)

Auch wenn die Funktionsweisen der Git-Erweiterungen ähnlich sind, ist der Funktionsumfang unterschiedlich. Ebenso unterscheiden sich die Anwendung der Befehle beziehungsweise der Kontroll-Flags zwischen Hub und Lab häufig. Für die konkrete Nutzungsweise der einzelnen Befehle sei auf die jeweilige Dokumentation verwiesen.

Mit Pull-Requests arbeiten

Es gibt bereits einige Artikel – zum Beispiel den Blog Post von Phillip Krüger *[8]* – , die beschreiben, wie komplexere Use Cases mit mehreren Git-Kommandos abgedeckt oder die Nutzung von Git-Kommandos vereinfacht werden können. Je nach Arbeitsprozess decken ebenso andere Tools wie Git Flow *[9]* entsprechende Bedürfnisse ab. Dank Hub und Lab können nun mit Leichtigkeit und sehr flexibel die Zusatzfunktionen der entsprechenden Repository-Plattformen genutzt und ebenso in Skripte eingebunden werden. Wiederkehrende Abläufe sind nun leicht automatisierbar. Je nach Persönlichkeit stellt man fest, dass der Bedarf, auf die jeweiligen Webseiten des Portals zu wechseln, hierdurch sehr stark begrenzt ist und die Kommandozeile immer mehr zum Cockpit für die Portale mutiert. Falls man bestimmte Tätigkeiten dennoch über die Webseite durchführen möchte, ist diese nun über `hub browse` beziehungsweise `lab project browse` leicht erreichbar.

Hub und Lab bieten Möglichkeiten der Verwaltung von Repositories (zum Beispiel Erstellen und Löschen derselbigen) über die Erstellung von Issues bis hin zum Arbeiten mit Pull-Requests an. Die genauen Aufrufe sind der jeweiligen Dokumentation entnehmbar. Um im Rahmen dieses Artikels ein Gefühl dafür zu geben, wie mit den zusätzlich angebotenen Funktionen der Repository-Plattformen mit Hub und Lab gearbeitet wird, beschränke ich mich im Folgenden auf die Funktionalität für Pull-Requests. Pull- und Merge-Requests sind weitestgehend synonym. Während GitHub diesen Prozess nach dem initialen Schritt benennt (Pull), hat sich GitLab entschieden, sich an der finalen Aktion zu orientieren (Merge) *[10]*.

Falls man in einem Projekt mit Feature Branches und Pull-Requests arbeitet, lässt sich leicht ein lokaler Branch in einen Pull-Request umwandeln und beispielsweise als `function` in die `.bashrc` einbinden. *Listing 3* zeigt ein Beispiel, um aus einem lokalen Branch in GitLab einen Merge-Request für den Develop Branch zu erstellen und den Quell-Branch beim Mergen automatisch wieder zu löschen (Flag -d). Bei Angabe eines Parameters wird für den benannten Branch ein Merge-Request erzeugt. Ohne Parameter wird für den gerade ausgecheckten Branch der Merge-Request erstellt. Hub-Nutzer müssen den Lab-Befehl in Zeile 5 wie folgt anpassen: `hub pull-request -b develop -m "merge branch to develop"`

Nach dem erfolgreichen Aufruf von `localBranchToMR` wird die URL und damit auch die ID des Pull-Requests angezeigt, die gegebenenfalls in ein Issue-Tracking-Tool übertragen werden könnte. Alternativ lässt sich jederzeit die Liste der Pull-Requests über den Aufruf `hub pr list` beziehungsweise `lab mr list` anzeigen.

Wäre die ID des Pull-Requests 1, würde der Merge – beispielsweise nach dem Abschluss eines Reviews – mit dem Befehl hub merge `https://github.com/myuser/myproject/pull/1` beziehungsweise `lab mr merge origin 1` ausgeführt werden. Der Unterschied ist, dass bei Hub der Merge lokal geschieht und entsprechend mit einem `git push` abgeschlossen werden muss. Bei Lab geschieht der Merge bereits remote. Läuft die GitLab-Pipeline des Quell-Branch noch, wird der Merge-Befehl an GitLab gesendet. GitLab führt den Merge – wie über das Webinterface – erst nach erfolgreichem Pipeline-Lauf durch. Über den abschließenden Status des Merge wird man von GitLab per E-Mail benachrichtigt.

Die Unterstützung für Merge-Requests bei GitLab-Repositories lässt sich ebenso über Plug-ins in IntelliJ IDEA einbinden. Eine Suche nach GitLab im Marketplace gibt Aufschluss darüber. Für GitHub ist dem Autor selbiges jedoch nicht bekannt. Dank Hub und Lab erhält

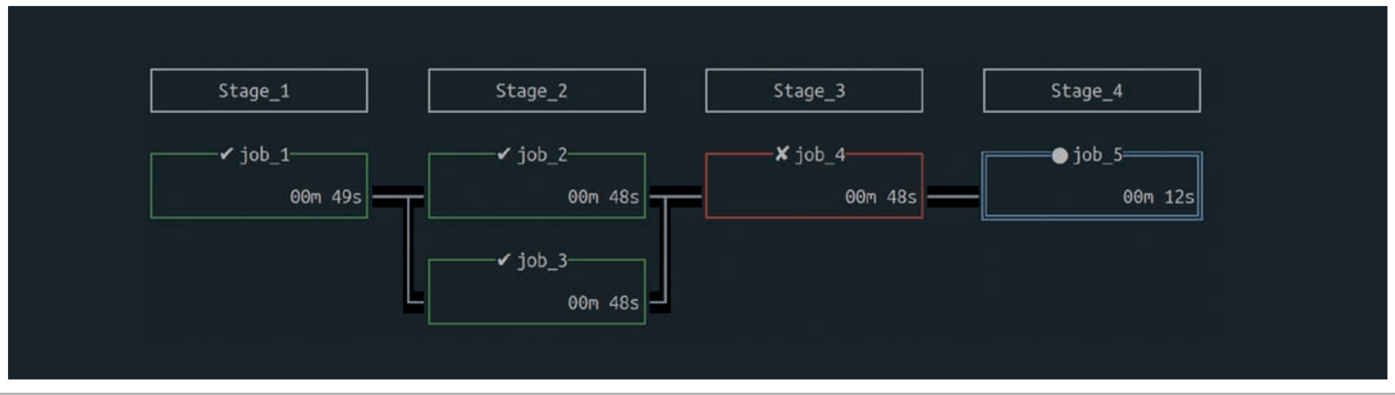


Abbildung 2: Die CI-Pipeline in der Kommandozeile (Quelle: Kai Schmidt)

man eine prächtige Unterstützung für jede beliebige IDE (beziehungsweise der dort eingeblendeten Konsole), um entsprechende Tätigkeiten verrichten zu können – unabhängig von einem Support durch Plug-ins.

Die CI-Pipeline in Textform

Mein persönliches Lieblingsfeature von Lab ist die Darstellung der CI-Pipeline. Nach einem Aufruf von `lab ci view origin develop` wird der aktuelle beziehungsweise letzte Pipeline-Lauf des Develop Branch angezeigt *(siehe Abbildung 2)*.

Wie gerade angedeutet, lässt sich diese Sicht praktischerweise in der Konsole der IDE einblenden, was *Abbildung 3* veranschaulichen soll. Durch IDE-Plug-ins erhält man diese Möglichkeit derzeit nicht. Jeder Job der Pipeline erhält ein Kästchen, in dem sowohl der Jobname als auch die verstrichene Zeit für den Job dargestellt wird. Bereits erfolgreich gelaufene Jobs werden in Grün dargestellt und erhalten einen Haken vor dem Jobnamen. Fehlgeschlagene Jobs sind rot gekennzeichnet und bekommen ein X. Noch nicht gelaufene Jobs bleiben schwarz. Laufende Jobs sind blau und erhalten einen ausgefüllten Kreis. Parallel laufende Jobs werden untereinander angezeigt.

Über die Cursor-Tasten kann man einen beliebigen Job auswählen. Welcher der Jobs gerade den Fokus erhalten hat, wird über eine doppelte Umrahmung der Box dargestellt. Den Job unter dem Fokus kann man durch einen Tastendruck auf `r` starten beziehungsweise erneut versuchen. Durch den Druck auf `c` wird der Job abgebrochen. Meist wird man sich jedoch für die Logs interessieren, auf die man über `t` Einblick erhält *(siehe Abbildung 4)*. Ein erneuter Druck geleitet uns wieder zurück zur Pipeline-Übersicht. Mit `T` werden die bisherigen Job-Ausgaben auf der Konsole wiedergegeben, sodass die Ausgaben auch nach Verlassen der Pipeline-Ansicht noch verfügbar sind.

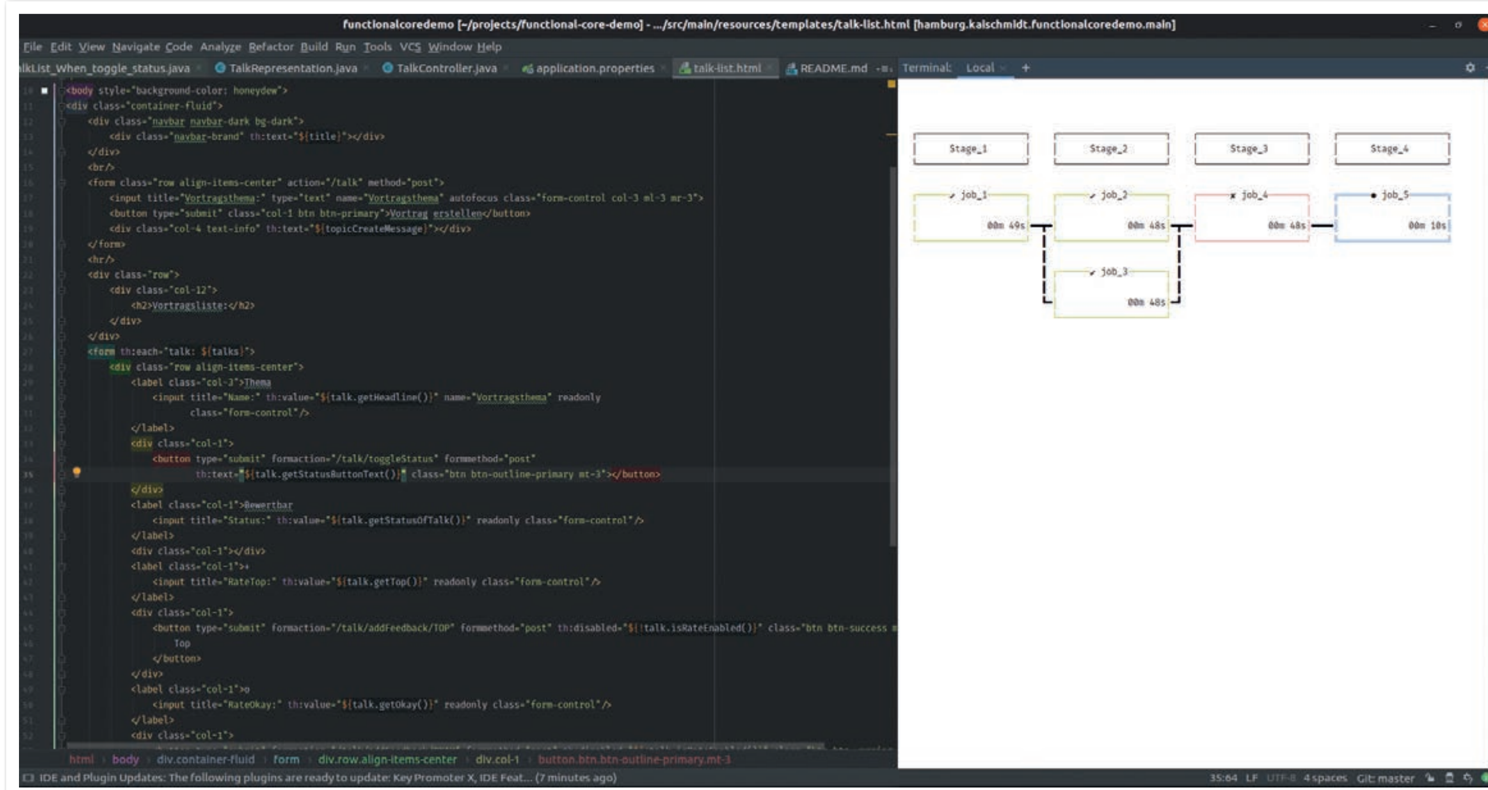


Abbildung 3: Die Pipeline in der Konsole einer IDE (hier IntelliJ IDEA) (Quelle: Kai Schmidt)

```
Showing logs for job_4 job #271929455
Running with gitlab-runner 12.1.0 (de7731dd)
  on docker-auto-scale ed2dce3a
Using Docker executor with image ruby:2.5 ...
Pulling docker image ruby:2.5 ...
Using docker image sha256:869f833217cc783f4e2bc309d5eb1d194ca980464a0176e2077f03c95eb72991 for ruby:2.5
...
Running on runner-ed2dce3a-project-13437446-concurrent-0 via runner-ed2dce3a-srm-1565736091-67fda8e1...
Fetching changes with git depth set to 50...
Initialized empty Git repository in /builds/electronickai/lab-demo/.git/
Created fresh repository.
From https://gitlab.com/electronickai/lab-demo
 * [new branch]      master     -> origin/master
Checking out 83b13e0f as master...

Skipping Git submodules setup
$ echo "Failing Job 4 in Stage 3"
Failing Job 4 in Stage 3
$ exit 1
ERROR: Job failed: exit code 1
```

Abbildung 4: Beispielhaftes Log eines Jobs (Quelle: Kai Schmidt)

```
$ alias git=hub

# “hub clone github/hub” wird zu
$ git clone github/hub

# Hub-Funktionalitäten laufen nun augenscheinlich mit git
$ git pull-request -b develop

# Funktionen ohne direkte Hub-Unterstützung funktionieren weiter wie gewohnt
$ git branch -a
```

Listing 4: Aliasing am Beispiel von Hub

Falls man mit der WSL arbeitet, beschreiben die Referenzen *[11]* und *[12]* die Einbindung derselben in IntelliJ und in Eclipse.

Erleichterung durch Aliasing und Tab Completion

Je nach persönlichem Geschmack ist es ratsam, für das jeweilige CLI-Tool ein Aliasing auf `git` zu definieren. Sämtliche Befehle, die von Hub beziehungsweise Lab nicht direkt interpretiert werden können, werden weiterhin zu Git durchgeschleust, sodass sich ein Gefühl einstellt, als seien die Funktionen bereits immer direkt in Git enthalten gewesen *(siehe Listing 4)*.

Den Funktionsumfang von Hub und Lab hat man schnell gelernt. Beide Tools beschreiben in ihrer Readme, wie das Aliasing auf UNIX-basierten Systemen eingerichtet werden kann. Selbstverständlich kann man je nach Vorliebe den Alias wechseln, sobald man auf die andere Plattform wechselt. Oder man stellt den Alias für die Plattform ein, die man zum Großteil verwendet. Lab erkennt aber auch Hub, sodass es ebenfalls möglich ist, Lab als alleinige Aufrufschnittstelle über alle drei CLIs zu verwenden. Der beispielhafte Aufruf von

```
$ lab version
git version 2.20.1
hub version 2.7.0
lab version 0.16.0
```

Listing 5: Lab verträgt sich sowohl mit Hub als auch mit Git

```
# Für ein GitLab-Repository
$ lab mr create origin develop

# Für ein GitHub-Repository unter Verwendung von Lab
$ lab pull-request -b develop -m “merge branch to develop”
```

Listing 6: Die Befehle zwischen Lab und Hub sind nicht kompatibel

`lab version` resultiert dann in der Ausgabe von *Listing 5*. Dennoch müssen die Befehle von Hub genutzt werden, um erfolgreich mit einem GitHub-Repository umgehen zu können – wie *Listing 6* zeigt.

Als weitere Erleichterung beim Tippen lassen sich für beide Tools bash- und zsh-Tab-Completions einrichten, deren Installation ebenfalls der Readme-Seite des jeweiligen Projekts zu entnehmen ist.

Fazit

Hub und Lab portieren die Zusatzfunktionen der beliebten Portale GitHub und GitLab auf die Kommandozeile. Die Befehle der beiden Tools sind eingängig und stehen – zumindest indirekt dank WSL oder Scoop – auch in Windows-Umgebungen zur Verfügung. Das einzige festgestellte Manko der WSL ist der ohne Wirkung verbleibende Absprung in den Browser über die jeweiligen browse-Befehle. Der Workaround über `lab mr show` und der damit angezeigten und klickbaren URL genügt jedoch meist als Workaround.

Auch wenn Lab noch keine 1.0-Version erreicht hat, machen beide Tools einen sehr ausgereiften Eindruck und verfügen über ein Issue-Tracking auf den jeweiligen GitHub-Seiten. Falls Befehle oder deren Flags etwas schwer zu merken sind oder sich wiederkehrende Befehlsfolgen erkennen lassen, kann man diese nun sehr leicht mit einem Bash-Skript automatisieren. Durch „Sourcing" in der .bashrc beziehungsweise .zsh sind sie automatisch als entsprechender Kommandozeilen-Aufruf verfügbar.

Man wird sich entweder GitHub oder GitLab als heiligen Gral auserkoren haben und somit auf das jeweilige Tool stürzen. Für Nutzer beider Portale sind Hub und Lab weiterhin einzeln nutzbar. Bei Bedarf bringt man über Lab nicht nur Lab selbst, sondern – wenn auch nicht ideal – ebenso Hub und Git unter eine „CLI-Haube".

Quellen

[1] *https://github.com*
[2] *https://github.com/github/hub*
[3] *https://gitlab.com*
[4] *https://github.com/zaquestion/lab*
[5] *https://about.gitlab.com/stages-devops-lifecycle/*
[6] *https://docs.microsoft.com/de-de/windows/wsl/install-win10*
[7] *https://docs.microsoft.com/de-de/windows/wsl/install-manual*
[8] *https://www.phillip-kruger.com/post/some_bash_functions_for_git/*
[9] *https://github.com/nvie/gitflow*
[10] *https://docs.gitlab.com/ee/workflow/gitlab_flow.html#mergepull-requests-with-gitlab-flow*
[11] *https://medium.com/@thomas.schmidt.0001/how-to-use-ubuntu-bash-on-windows-10-as-the-intellij-idea-terminal-334fd9a10d8c*
[12] *https://superuser.com/questions/1416461/opening-linux-sub-system-for-windows-shell-from-eclipse-photon-ide*

Kai Schmidt
selbstständig
mail@kai-schmidt.hamburg

Kai Schmidt ist freiberuflicher Software-Entwickler und -Architekt. Zuvor war er bei den IT-Beratungsunternehmen .msg systems ag und Capgemini angestellt und in seiner über 10-jährigen Projekterfahrung größtenteils in Java- und C#-Projekten in den Bereichen Logistik, Flugzeugbau sowie Handel tätig. Hub und Lab haben ihn sofort begeistert, nachdem er vor einigen Wochen über Twitter-Tweets Kenntnis davon gewonnen hatte. Heute berät und beteiligt er sich gerne an betrieblichen Anwendungssystemen und ist in der JUG sowie für Kids4IT engagiert. Kai ist auch als Speaker auf Konferenzen und Meetups aktiv.

Java aktuell

Mehr Informationen zum Magazin und Abo unter:
https://www.ijug.eu/de/java-aktuell

Edit and P(r)ay – Oder doch lieber testen?

Anja Papenfuß-Straub, ING Deutschland

Tests und Testautomatisierung sind nichts Neues, allerdings ist ihre Bedeutung in den letzten Jahren gestiegen. Mit der zunehmenden Digitalisierung ändert sich das Kundenverhalten massiv. Von uns als Entwickler wird die zeitnahe Umsetzung und Live-Stellung neuer Features erwartet. Dabei sehen wir uns mit zunehmend schnellerer technologischer Entwicklung konfrontiert. Wir müssen sicherstellen, dass unser Code funktioniert, und trotzdem fällt uns die Testautomatisierung unserer Software noch schwer. Dieser Artikel soll ermutigen und zeigen, dass Testen sinnvoll und gar nicht so schwer ist.

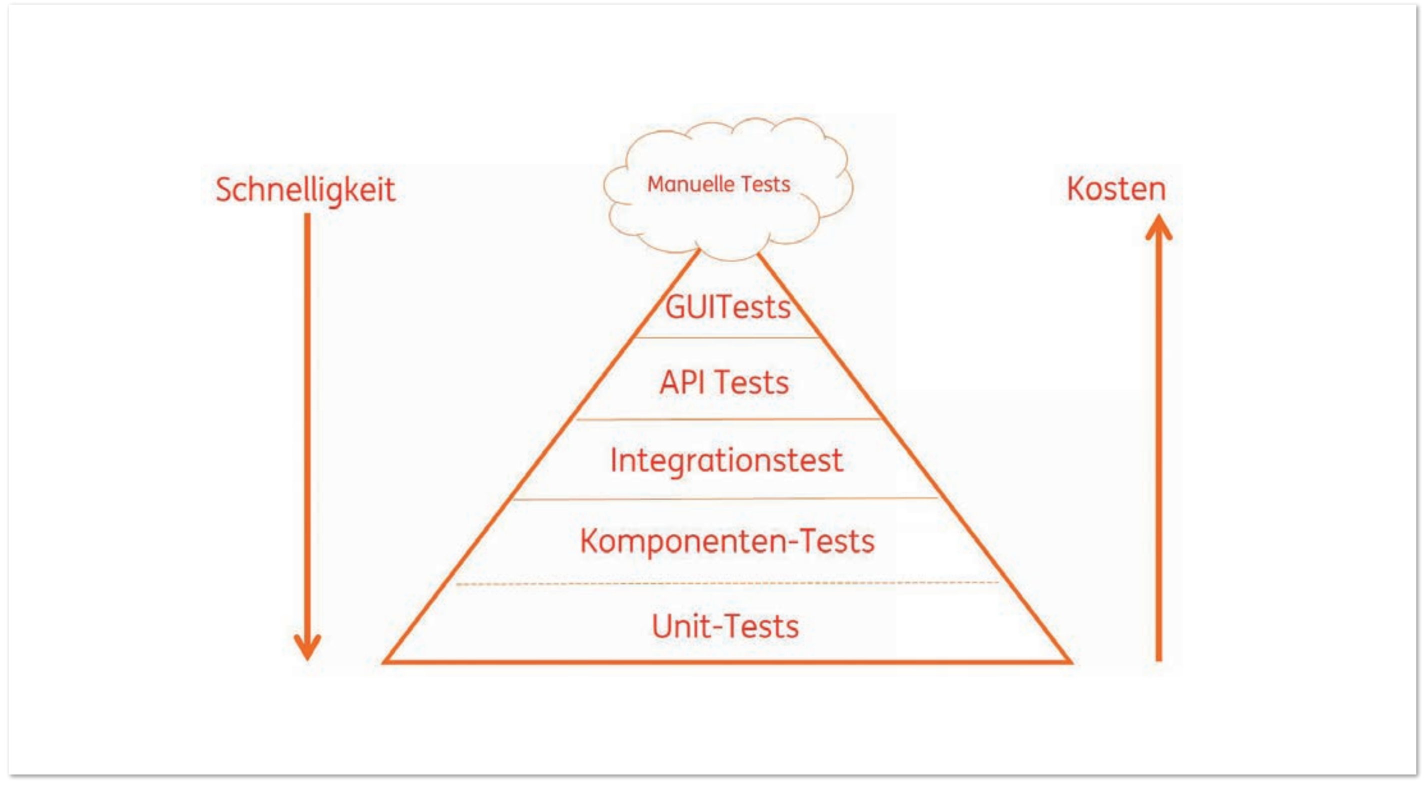


Abbildung 1: Testpyramide (© Anja Papenfuß-Straub)

Vor über zehn Jahren sah die Softwareentwicklung noch anders aus. Ich kann mich noch gut an umfangreiche Debugging-Sessions erinnern, um Fehler auf dem Produktivsystem zu finden. Bei einem meiner ehemaligen Arbeitgeber hing am IT-Chef-Büro ein riesiges Poster: „Testen ist was für Mädchen, Jungs gehen live!". Unit-Tests waren explizit verboten, sie galten als zu teuer. Die Tester durften aufwendige End-2-End-Tests schreiben (damals HttpUnit-Tests). Wir waren froh über diese Tests, brachten sie doch zumindest ein kleines Gefühl von Sicherheit. Jeder, der Fehler auf Produktionssystemen suchen und beheben muss, weiß, wie nervenaufreibend so etwas ist. Erstaunlicherweise gab es relativ wenig größere Zwischenfälle. Immerhin plante unser Kunde eine lange Testphase vor jedem Release ein. Das half ebenfalls, noch Fehler zu entdecken, und wir hatten nur wenige Releases pro Jahr.

Doch die Zeiten ändern sich. Was dieser digitalen Welt fehlt, ist Geduld. Darum sind die Unternehmen zunehmend unter Druck, auf Kundenwünsche zeitnah zu reagieren. Damit sind auch wir als Entwickler gefordert, immer schneller neue Features zu liefern und diese in kurzen Release-Zyklen produktiv zu stellen. Dabei dürfen sich Code-Qualität und Security keinesfalls verschlechtern. Mittlerweile arbeiten die meisten von uns in agilen Teams und sind als „DevOps" verantwortlich für Entwicklung und Betrieb unserer Software. Wir können nicht mehr auf lange Testphasen durch Fachtester und Fachteams vor Live-Stellungen zählen. Darum ist eine Testautomatisierung zwingend erforderlich. In der agilen Arbeitsweise sind darum Tests bereits Teil der „Definition of Done" und ein Kriterium für die Abnahme von Features. Sie sind die Grundvoraussetzung zur Absicherung unserer Software. Die Unternehmen haben erkannt, dass Produktionsfehler um ein Vielfaches teurer sind als Fehler, die während der Entwicklungsphase erkannt werden. Im schlimmsten Fall drohen Reputationsverlust bei den Nutzern und finanzielle Einbußen.

Software Lifecycle

Um die Bedeutung zu verstehen, die dem Testen tatsächlich gebührt, lohnt sich ein Blick auf den Software Lifecycle. Software wird ständig verändert und weiterentwickelt, zumindest erfolgreiche Software. Wir ändern Code ganz einfach über unsere Entwicklungsumgebung. Ändern und erweitern wir unseren Code, ohne dass wir die Code-Struktur anpassen, wird unser Code starr und dadurch schlechter änderbar. Der Grad der Unordnung innerhalb unseres Codes nimmt zu, auch bekannt als Software-Entropie.

Bei jeder neuen zu implementierenden Funktion müssen wir einen Kompromiss finden zwischen den begehrten neuen Features und der vorhandenen Code-Basis. Gilt unser Augenmerk nur der Entwicklung von neuem Code zulasten der Bestandsbasis, führt das zuerst zum Verlust der strukturellen Qualität (Architektur, Entwurfsmuster etc.) und im schlimmsten Fall auch zum Verlust der funktionalen Qualität (Fachlichkeit). Im Laufe der Jahre hat wohl fast jeder Entwickler seine Erfahrungen mit Legacy-Anwendungen gemacht. Oft war die ursprüngliche Absicht des Code-Erstellers nach mehreren Anpassungen nicht mehr einfach erkennbar. Wir sprechen in diesem Fall von Software-Erosion beziehungsweise von technischen Schulden.

Schnelle Änderungen an diesem Code sind kaum oder nur langsam möglich, sie sind fehleranfällig und teuer. Software lebt aber von Veränderungen und muss anpassbar bleiben. Das heißt, dass eine Anpassung der Software einfach und schnell durchführbar ist. Dabei ist die Struktur des Codes klar erkennbar und der Code ist ohne großen Aufwand veränderbar. Die einzige Möglichkeit, Software-Erosion unter Beachtung der Software-Entropie zu verhindern, ist regelmäßiges Refactoring. Um unsere Neuentwicklungen und Refactorings abzusichern, braucht es einen Test-Harnisch.

```
@Autowired
private SearchManager classUnderTest;

@Test
public void findName(){
  //Arrange
  SearchQuery query = new SearchQuery("name");
  //Act
  List<SearchResult> results = classUnderTest.find("name");
  assertThat(results).extracting(SearchResult::name)
     .containsOnly("name");
}
```

Listing 1: Tripple A (Arrange Act Assert) mit assertj als Assertion Framework

Der Test-Harnisch fungiert als Sicherheitsnetz und besteht aus vielen einfachen und gut wartbaren Entwicklertests (Unit- bzw. Komponententests) sowie wenigen Integrationstest (Testpyramide, siehe *Abbildung 1*). Die Testpyramide definiert die Testarten und deren Umfang unter Beachtung von Kosten und Nutzen. Günstige und schnelle Testarten werden bevorzugt (Unit-Tests). Mit jeder Anpassung und Erweiterung unseres Codes vergrößern wir die Test-Basis und verfeinern die Maschen unseres Sicherheitsnetzes. Das hilft uns, die strukturelle und funktionale Qualität zu erhalten beziehungsweise zu verbessern.

Code Coverage und Testqualität

Code Coverage, ein prozentualer Wert, der die Testabdeckung im Code misst, wird gern als Druckmittel und Heilsbringer verwendet. Zahlen wie diese werden auch vom Management gut verstanden. Aber welche Aussagekraft hat dieser Wert? Wenn wir ehrlich sind, ist eine hohe Code Coverage noch lange kein Garant für gute und sinnvolle Tests. Code Coverage prüft nur die Codezeilen und Abzweigungen, die bei einem Test durchlaufen wurden. Die Validierung des Testergebnisses und die Richtigkeit unseres Codes spielen dabei keine Rolle. Verlassen wir uns also rein auf die Testabdeckung, wiegen wir uns in falscher Sicherheit.

Trotzdem bin ich nicht grundsätzlich gegen Code-Coverage-Vorgaben. Eine geringe Code Coverage ist immerhin ein Indiz für notwendige Nachbesserungen in Sachen Tests. In Kombination mit statischer Code-Analyse sind durchaus gute Ergebnisse zu erzielen, um die Test- beziehungsweise Code-Qualität zu erhalten und zu verbessern.

Was aber macht einen guten Test aus? Kurz gesagt sollte ein Test dem **F.I.R.S.T**-Prinzip entsprechen, dem Grundsatz für Test-Driven-Development (TDD):

- **Fast:** Die Testlaufzeiten sollten gering sein, bei reinen Unit-Tests sogar im Millisekunden-Bereich. Integrationstests können auch länger laufen.
- **Isolated/Independent:** Testmethoden einer solchen Testklasse dürfen sich nicht gegenseitig bedingen. Sollten Tests aufeinander aufbauen, wird die Wartung dieser Tests extrem erschwert. Die Testmethoden können nicht unabhängig voneinander einzeln ausgeführt werden.
- **Repeatable:** Die Tests sollen wiederholbar sein und bei jedem Lauf das gleiche Ergebnis liefern.
- **Self-Verifying:** Es sollte nur ein Aspekt pro Test getestet werden. Idealerweise besteht ein Test aus Arrange (Initialisieren des Test-Setups), Act (Ausführen des zu testenden Codes) und Assert (der Validierung des Ergebnisses), siehe *Listing 1*. Durch die automatische Assertion ist eine manuelle Verifikation unnötig. Darum: Spart euch Logausgaben in Tests! Wer soll die lesen?
- **Thorough/Timely:** Das Ziel ist nicht, eine hundertprozentige Code Coverage zu haben, sondern einen Test-Harnisch aufzubauen, dem wir vertrauen. Und zu guter Letzt sollten Tests immer vor dem Implementieren der Funktionalität geschrieben werden (TDD).

Code-Qualität

Bei umfangreichen Refactorings in großen Bestandssystemen habe ich leidvoll erfahren: Einer der größten Hinderungsgründe, um gute Tests zu schreiben, ist und bleibt schlechte Code-Qualität. Schlechte Code-Qualität ist meist die Hauptursache, warum wir das Schreiben von Tests als aufwendig und langsam empfinden. Im Folgenden ein paar Beispiele, die jeden Entwickler, der Tests schreiben soll, in die Knie zwingen.

Hohe Komplexität: Wer kennt sie nicht, die Gott-Klassen. Riesig und sehr komplex vermitteln sie den Eindruck, alles zu können und jeden zu kennen. Neben Angst erzeugen diese Klassen vor allem Ärger beim Testen. Das Test-Setup für diese Klasse wird extrem aufwendig und die Wartung teuer. Außerdem leidet die Lesbarkeit. So wenig, wie der Code der Klasse verstanden wird, so wenig wird auch der Test dafür verstanden. Nebenbei ist so eine hohe Komplexität immer ein Indiz für Code Smell, zum Beispiel wenn das Single Responsibility Principle (SRP) nicht beachtet wurde. Das heißt, die Klasse enthält viele Funktionen, für die sie eigentlich nicht zuständig ist. Hier sollten wir die Abhängigkeiten analysieren, die Gemeinsamkeiten extrahieren und in separate Klassen überführen, die dann einzeln auch viel besser testbar sind.

Falsch verwendete Vererbung: Im Informatik-Studium wird uns Vererbung immer noch als Allheilmittel verkauft. Aber Hand aufs Herz, verwenden wir Vererbung wirklich immer richtig? Ist uns das Liskovsche Substitutionsprinzip immer gegenwärtig, wenn wir programmieren? Wir sind dazu erzogen worden, Code-Duplizierung zu vermeiden, und dabei dem Irrglauben verfallen, dass Vererbung hier der richtige Weg sei. Vererbung ist allerdings die engste Form der Kopplung. Angesicht der Bedeutung der Änderbarkeit von Software ist jedoch eine lose Kopplung (Komposition anstelle von Vererbung) zu bevorzugen („composition over inheritance"). Kompositionsklassen sind die geschicktere Möglichkeit, um Code-Duplizierung und eine enge Kopplung zu vermeiden. Sie sind in der Regel besser und isoliert testbar.

```java
public final class SpecialDateUtil {

     public static String getTimeOfDay() {
        DateTime time = new DateTime();
        if (time.getHour() >= 0 && time.getHour() < 6) {
          return "Night";
        } if (time.getHour() >= 6 && time.getHour() < 12) {
          return "Morning";
        } if (time.getHour() >= 12 && time.getHour() < 18) {
          return "Afternoon";
        } return "Evening";
     }
}
```

Listing 2: Statische Klasse mit statischer Methode

```java
@Test
public void getTimeOfDay_6AM_Morning() {
  try
  {
     // Setup: change system time to 6 AM
     ...
     String timeOfDay = SpecialDateUtil.getTimeOfDay();

     assertThat(timeOfDay).isEqualTo("Morning");
  }
  finally
  {
     // Teardown: roll system time back
     ...
  }
}
```

Listing 3: Testmethode für Listing 2

Statische Klassen: Die Namen dieser Klassen enden meist auf „Util" oder „Commons" *(siehe Listing 2)*. Es ist sehr einfach, diese zu schreiben und zu verwenden. Schwierig wird hier aber das Testen vor allem für die Klassen, die diesen statischen Code verwenden *(siehe Listing 3)*. Das Setup solcher Tests wird komplizierter, da die Elemente zum Durchlaufen des statischen Codes vorher initialisiert werden müssen. Statische Klassen sind ein Indiz für eine Design-Schwäche, die darauf hinweist, dass hier eine Methode ist, die eigentlich zu einer anderen Klasse gehört. Die Wahrheit ist: Statische Klassen mutieren oft zu riesigen Monstern, zu einer Halde für Methoden, die Entwickler sinnvoll für andere halten. Oft haben wir uns wenig Gedanken gemacht, wo diese statischen Methoden eigentlich hingehören – im Idealfall zur Klasse, welche die statische Methode aufruft. Statische Klassen enthalten prozeduralen Code, alles andere ergibt zumindest keinen Sinn. Wir bewegen uns mit Java als Programmiersprache allerdings in der objektorientierten Welt. Passt prozeduraler Code zum Unit-Testing? Beim Unit-Test sollen wir über Instanziierung ein Stück der Applikation isoliert testen. Dabei versuchen wir, alle Abhängigkeiten durch Mocks etc. zu ersetzen. Prozeduraler Code kennt jedoch so ein „wiring" gar nicht, da gibt es keine Objekte, keinen Code und keine Daten, die separierbar sind. Darum ist das Testen hier so schwierig *(siehe Listing 3)*.

Der Test für die Klasse aus *Listing 2* wird, je nach Ausführzeitpunkt des Tests, ein anderes Ergebnis liefern. Das heißt, die Methode enthält einen veränderbaren globalen State und verhält sich nicht deterministisch.

Um das nicht-deterministische Verhalten im Test zu berücksichtigen, muss jeweils das System-Datum kompliziert gesetzt und nach dem Test zurückgesetzt werden. Aus einem reinen Unit-Test wird dann schnell ein Integrationstest, in dem umständlich ein Environment gesetzt werden muss. Ein Black-Box-Test ist nicht möglich, denn wir müssen den Code kennen, um ihn zu testen. Die if-else-Kaskade macht das Testen auch umfangreicher, da wir für eine zu testende Methode vier Testmethoden schreiben müssen.

Wir wollen selbst bestimmen, welchen Wert „timeOfDay" *(siehe Listing 2)* bekommt, damit wir nicht noch einmal die Testaufwände aus *Listing 3* duplizieren. Die SpecialDateUtil-Klasse enthält nicht

```java
public class FlyerHeadlineGenerator {

  public String createFlyerHeadline(){
     String timeOfDay = SpecialDateUtil.getTimeOfDay();
     return "Welcome to our event next thursday " + timeOfDay;
  }
}
```

Listing 4: Verwendung statischer Klasse aus Listing 2

```
public class DateWordGenerator {

  public String getTimeOfDay(DateTime time) {
     if (time.getHour() >= 0 && time.getHour() < 6) {
       return "Night";
     } if (time.getHour() >= 6 && time.getHour() < 12) {
       return "Morning";
     } if (time.getHour() >= 12 && time.getHour() < 18) {
       return "Afternoon";
     } return "Evening";
  }
}
```

Listing 5: Code-Anpassung für bessere Testbarkeit

nur einen globalen State, sondern sie ist auch noch statisch und dadurch nur mit Zusatz-Frameworks mockbar.

Um den Code testbarer zu machen, lösen wir den globalen State auf und übergeben das DateTime-Objekt als Parameter. Klasse und Methoden sind nicht mehr statisch und können zum Beispiel als Singleton Instance weiterleben. Weitere Beispiele für Code, der Testbarkeit behindert:

- Logik im „Constructor"
- Erzeugen eigener Abhängigkeiten durch „new Objects" innerhalb von Methoden
- Verletzung des Prinzips „law of demeter"
- if-else-Kaskaden im Code
- Abhängigkeiten zu riesigen Kontext-Objekten
- Verwenden statischer Initializer

Fazit

Testen ist sinnvoll und testgetriebene Entwicklung bringt jede Menge Vorteile. Immerhin hilft sie, saubere und testbare Architekturen zu bauen sowie regelmäßige und abgesicherte Refactorings durchzuführen. Die Akzeptanz eines Vorgehens, unabhängig davon, ob es sich um Testautomatisierung oder andere Themen handelt, hat viel mit unserer inneren Einstellung zu tun. Wir hängen an unseren Gewohnheiten, sie geben uns ein Gefühl von Sicherheit. Wir sind eher bereit, diese zu verändern, wenn wir positive Erfahrungen machen. Mit der Erkenntnis, dass wir durch testgetriebene Entwicklung sicherstellen können, dass unser Code funktioniert, und dass wir befähigt werden, bessere Qualität zu liefern, wächst auch die Bereitschaft, Zeit in das Schreiben von Tests zu investieren.

Statische Unternehmensvorgaben sind sinnvoll, aber diese allein helfen nicht. Der Wandel vom auftragsbezogenen Abarbeiten fachlicher Anforderungen hin zu verantwortungsbezogenem Entwickeln von Software funktioniert nur mit der Veränderung unserer Grundeinstellung. Unser Ziel als Entwickler sollte letztendlich sein, gute, wartbare und stabile Software zu schaffen, die den Ansprüchen an funktionale und qualitativ gute Software gerecht wird.

Wir sollten als selbstorganisierte Teams fähig sein, der Forderung nach neuen Features selbstbewusst zu begegnen. Die Verantwortung eines Entwickler-Teams geht über die Entwicklung neuer Fachlichkeit hinaus und braucht Zeit. Letztendlich müssen alle Beteiligten diesen Weg gehen wollen. Am Ende gewinnen wir alle, weil wir nur dadurch nachhaltig Schnelligkeit und Sicherheit garantieren können. Also lasst uns immer daran denken:

„Softwaretesting is not about finding bugs. It's about delivering great software"

(© Harry Robinson, Software Test Lead, Microsoft)

Quellen

[1] Frank Westphal - Testgetriebene Entwicklung mit JUnit & FIT, dpunkt.verlag, 2006

[2] *http://misko.hevery.com/2008/07/24/how-to-write-3v1l-untestable-code/*

[3] *http://misko.hevery.com/2008/12/15/static-methods-are-death-to-testability/*

[4] *https://www.atlassian.com/blog/add-ons/deliver-faster-and-better-software-using-test-automation*

[5] *http://www.nils-haldenwang.de/german/das-liskovsche-substitutionsprinzip-in-java-invarianz-kovarianz-kontravarianz*

[6] *https://www.codepedia.org/toptal/unit-tests-how-to-write-testable-code-and-why-it-matters/*

Anja Papenfuß-Straub

ING Deutschland

Anja.Papenfuss-Straub@ing.de

Anja hat fast 19 Jahre Erfahrung als Java-Softwareentwicklerin und ist seit über sechs Jahren Java-Backend-Entwicklerin und Application Architect bei der ING Deutschland am Standort Nürnberg. Dabei liegt ihr Fokus auf Code-Qualität und Testbarkeit. Sie hat die ING Deutschland aktiv bei der Einführung neuer Testautomatisierungs-Vorgaben für die Softwareentwicklung begleitet und die Entwickler bei der Umsetzung unterstützt.

„Man kann selbst entscheiden, ob man Teil des Wandels sein möchte oder diesen ignoriert“

Lars Thomsen, future matters AG, ist Zukunftsforscher und teilt als Speaker auf Veranstaltungen sein Wissen. Im Interview mit Christian Luda, DOAG Dienstleistungen GmbH, gibt Thomsen einen Einblick in seine Arbeit sowie aktuelle Trend-Themen der Zukunft und schätzt ein, was uns noch erwarten könnte.

Herr Thomsen, was hat Sie bewogen, Zukunftsforscher zu werden?

Lars Thomsen: Schon als Kind wurde bei uns zuhause viel über die Zukunft gesprochen und diskutiert. Dabei wurden wir Kinder von meiner Mutter immer ermutigt, unserer Fantasie freien Lauf zu lassen (sie war Kindergärtnerin), und von meinem Vater ermahnt, nicht nur Luftschlösser zu planen, sondern auch darüber nachzudenken, wie man eine solche Idee umsetzen könnte (er war Ingenieur). Diese Kombination aus Kreativität, Neugier und technischem Sachverstand waren die Zutaten und Grundlagen, die auch heute noch meine Arbeit prägen.

Welche Rolle hat Science-Fiction bei Ihrer Faszination für die Zukunft gespielt?

Lars Thomsen: Science-Fiction arbeitet ebenfalls mit der Kombination aus Fantasie und technologischen Utopien. Die meisten Science-Fiction, Werke folgen in ihrer Projektion einer klaren Logik und sind oft aus technischer Sicht sehr plausibel und begründbar. In der Zukunftsforschung arbeiten wir ähnlich. Die Serie „Raumschiff Enterprise“ ist übrigens entstanden, weil der Produzent Gene Roddenberry damals führende Wissenschaftler aus unterschiedlichsten Disziplinen in einem Think Tank hatte, die träumen durften, wohin sich ihr Fachgebiet in 300 Jahren entwickeln könnte.

Wir leben in einer Zeit großen Wandels. Was sagen Sie Menschen, die aufgrund der Veränderungen Angst vor der Zukunft haben?

Lars Thomsen: Menschen haben Angst vor den Dingen, die sie nicht kennen oder nicht ändern können. Je mehr man sich mit der Zukunft beschäftigt, desto weniger Angst hat man vor ihr und dem Wandel, den sie bringt. Das ist nicht neu: Wandel ist das Ergebnis von Innovationen. Alles, was wir heute in unserer Zivilisation an Annehmlichkeiten täglich nutzen, war irgendwann mal eine Utopie und wurde auch sicher von vielen als „Quatsch" abgelehnt. Aber es haben sich die Innovationen durchgesetzt, die für Menschen einen Sinn machten und ihnen einen Mehrwert brachten. Ob nun elektrischer Strom, Demokratien, Trinkwasser aus einer zentralen Wasserversorgung oder das Automobil – jede Innovation war für viele Menschen und Industrien mit einem Wandel verbunden. Man kann selbst entscheiden, ob man Teil des Wandels sein möchte oder diesen ignoriert beziehungsweise bekämpft.

Welche Zeitspanne betrachten Sie als Zukunftsforscher und inwiefern hat sich diese durch die Digitalisierung verändert?

Lars Thomsen: Wir betrachten in der Regel den Zeitraum der kommenden 520 Wochen, sprich zehn Jahre. Uns interessieren vor allem sogenannte „Tipping Points", also Umbrüche und Disruptionen aufgrund technischer, sozialer und wirtschaftlicher Trends, Innovationen und Veränderungen. Zudem muss man die Wechselbeziehungen der Trends untereinander betrachten. Die Digitalisierung beispielsweise ist isoliert betrachtet recht uninteressant, aber ihre Auswirkungen auf Arbeit, Gesellschaft und Industrien enorm, und sie wird mit künstlicher Intelligenz noch einmal eine ganz andere Dynamik erhalten.

Ist Ihr Job durch den digitalen Wandel schwieriger geworden?

Lars Thomsen: Unser Job wird durch die Digitalisierung sowohl einfacher, da wir viel schneller recherchieren und kommunizieren können, aber auch komplexer, da die Veränderungsdynamik stark gestiegen ist. Trotzdem bleiben wir bei unseren Zeiträumen, in denen wir die jeweiligen Entwicklungs-Roadmaps ausarbeiten.

Können Sie kurz umreißen, wie sehr künstliche Intelligenz unser Leben verändern wird?

Lars Thomsen: Künstliche Intelligenz wird nach unserer Meinung das Leben, die Arbeit, die Wirtschaft und die Gesellschaft in den kommenden 10 Jahren mehr verändern als der der Siegeszug der Dampfmaschine, der das Leben der Menschen in der zweiten Hälfte des 19. Jahrhundert veränderte. Das liegt daran, dass wir es mit einem völlig neuen technologischen Paradigma zu tun haben: Erstmals können Maschinen lernen. Wir sind zwar noch sehr am Anfang dieser Entwicklung, aber wenn man sich vergegenwärtigt, dass wir Menschen nur so weit gekommen sind, weil wir in der Lage waren, zu lernen und nicht immer nur „das gleiche Programm" abzuspielen, dann kann man erahnen, was für eine Sprengkraft künstliche Intelligenz langfristig hat.

Inwieweit ist die Bedeutung dieses Themas schon in Wirtschaft, Politik und Gesellschaft angekommen?

Lars Thomsen: Leider ist die Auseinandersetzung mit der Technologie, ihren Möglichkeiten, Grenzen und Gefahren derzeit sowohl in Wirtschaft, Politik und Gesellschaft mangelhaft und oftmals naiv. Daher machen wir uns in der Gruppe der Zukunftsforscher sehr große Sorgen, ob wir in Deutschland und Europa die Potenziale dieser neuen Schlüsseltechnologie tatsächlich heben können.

Zur Person

Lars Thomsen gehört zu den führenden Zukunftsforschern weltweit. 1968 in Hamburg geboren, gilt als einer der einflussreichsten Vordenker für die Zukunft der Arbeit, Energie, Mobilität und Künstliche Intelligenz. Er ist Gründer und Chief Futurist der future matters AG in Zürich, welche mehr als 800 Unternehmen in Europa und Übersee bei der Entwicklung von Zukunftsstrategien und Geschäftsmodellen begleitet.

Sie haben die Bedeutung von sogenannten Tipping Points erwähnt. Können Sie anhand eines Beispiels aus der Vergangenheit einen solchen Punkt beschreiben?

Lars Thomsen: Ein Tipping Point ist der Zeitpunkt, an dem eine neue Problemlösung günstiger, attraktiver und sinnvoller als die bisherige Lösung wird. Nehmen wir das Automobil als Beispiel: Als die ersten Autos entwickelt und gebaut wurden, waren von Pferden gezogene Kutschen für viele noch eine ganze Weile die bessere Wahl. Doch das Auto wurde durch Innovationen am Produkt und in der Herstellung immer besser, günstiger und leistungsfähiger, so dass der Punkt kam, an dem Pferde und Pferdekutschen nicht mehr mithalten konnten. Die ersten Autokäufer wurden noch belächelt und Hufschmiede und Pferdehändler haben sich lange eingeredet, dass auch in Zukunft das Pferd dominieren würde, aber es kam anders – und ab dem Tipping Point (Model T von Ford) sehr schnell.

Welche weiteren Trends und Technologien werden – neben KI – aus Ihrer Sicht unsere Zukunft besonders prägen? Wo erwarten Sie Tipping Points?

Lars Thomsen: Wir beobachten derzeit 22 Trends mit Tipping Points – es gibt tatsächlich viele Umbrüche. Einige der spannendsten sind derzeit Service-Robotics, also Maschinen, die in Interaktion mit Menschen und menschlichen Umfeldern funktionieren. Daneben findet man im Bereich der Nahrungsmittelproduktion derzeit viele Anzeichen für bevorstehende Tipping Points, wie beim „Vertical Farming" oder bei der künstlichen Fleischproduktion. Aber auch in den Bereichen Energie, Bildung, Materialen und Werkstoffe, Mobili-

tätssysteme und Medizin der dritten Generation ist in den kommenden 520 Wochen einiges an Um- und Durchbrüchen zu erwarten.

Die großen Zukunftsvisionen scheinen heute im Silicon Valley zu entstehen. Besteht die Gefahr, dass einige wenige Konzerne über unsere Zukunft bestimmen und wir in Europa den Anschluss verlieren?

Lars Thomsen: Zukunft wird dort gemacht, wo Menschen ihre Ideen am effizientesten ausprobieren und umsetzen können. Dafür braucht es Risikokapital, Talente und unternehmerischen Mut, auch mal Fehler zu machen. Es fehlt uns in Europa nicht an der Ausbildung oder den Ideen, aber wir laufen Gefahr, immer mehr zu Bedenkenträgern als zu Innovatoren zu werden. Ich frage mich oftmals, ob, wenn heute erst das Automobil erfunden werden würde, wir in Europa es als „Quatsch" ablehnen und bekämpfen würden.

Wo sehen Sie besonderen Handlungsbedarf?

Lars Thomsen: Wir brauchen mehr Risikokapital und mutige Unternehmerinnen und Unternehmer, sonst verlieren wir immer mehr Boden. Zudem haben wir viel zu lange den Fokus auf „inkrementelle" Innovation, also Produktverbesserungen gesetzt, aber disruptive Innovation vernachlässigt. Wir brauchen wieder viel mehr visionäre Menschen – und zwar in jedem Bereich unserer Gesellschaft. Veränderungen kommen nicht von allein, sie folgen einer Vision. Ich habe aber den Eindruck, dass die Gesellschaft und Politik hierfür nun wieder etwas offener geworden sind, als das noch in den letzten Jahren der Fall war.

Knapp ein Drittel Ihrer Zeit verbringen Sie mit Entdeckungs- und Forschungsreisen. Wie kann man sich so eine Reise vorstellen und können Sie uns verraten, welchen Ort Sie zuletzt bereist und was Sie dort erforscht haben?

Lars Thomsen: Meist verfolge ich einen spezifischen Megatrend und recherchiere dafür viel zu den Daten, Fakten und Szenarien sowie die wichtigsten Protagonisten. Ich nehme dann Kontakt mit diesen Personen auf und spreche mit ihnen über die Entwicklungen und über deren und unsere Szenarien zur zukünftigen Entwicklung. Oftmals ergeben sich daraus für beide Seiten sehr interessante Erkenntnisse. Derzeit bin ich in den USA und besuche zahlreiche Gründer, Investoren und Wissenschaftler zum Thema „Future Food Production". Nachdem wir mit 30 bis 50 Experten deren Einschätzungen, Erwartungen und Projektionen erörtert haben, verorten wir die mit diesem Trend zu erwartenden Tipping Points in unsere Roadmaps. Zukunft erforscht man am besten, indem man mit den Menschen darüber spricht, die daran arbeiten.

Gibt es Orte auf der Welt, an denen sich die Zukunft besonders gut erforschen lässt? Welche Orte sind das und was zeichnet Sie aus?

Lars Thomsen: Es gibt mittlerweile viele „Innovation-Hotspots", die teilweise sehr unterschiedliche Themen oder Fokus-Technologien haben. Silicon Valley ist nach wie vor einer der Orte, wo man erstaunlich viel erfahren und sehen kann – vor allem, wenn es um die Mobilität von Menschen und Gütern in der Zukunft geht. Aber mittlerweile verbringe ich auch recht viel Zeit in China, das sich in vielen der zukünftigen Schlüsselindustrien enorm gut aufgestellt hat. Aber Orte sind nur die Plätze, wo interessante und visionäre Menschen leben, arbeiten und sich austauschen. Daher treffe ich diese auch gern auf Messen oder Technologie-Festivals oder Konferenzen wie dem South by Southwest (SXSW) in Austin, Texas.

Welchen Anteil haben Spekulationen und Utopien bei der Zukunftsforschung? Braucht ein Zukunftsforscher ein gewisses Maß an Fantasie?

Lars Thomsen: Absolut! Zukunft beschreibt eine Zeit, die noch nicht hier ist. Aber man kann und sollte sich den Raum und die Zeit nehmen, sich mit Hilfe von Vorstellungskraft und Fantasie eine Utopie oder ein Szenario vorzustellen. Dann nämlich wird einem klarer, warum Menschen an diesen Visionen arbeiten und was ihre Motivation ist. Spekulation ist immer dann angesagt, wenn diese auf der Grundlage von Logiken, Modellen, Daten und Fakten basiert, die eine Spekulation rechtfertigen kann. Letztlich beeinflussen Utopien und Spekulation sogar die Zukunft: Immer, wenn Menschen diese teilen, verändert sich das Denken und die Dynamik von Entwicklungen in einem System.

Wie hilfreich ist bei Ihrer Arbeit ein Blick in die Vergangenheit? Macht es etwa Sinn, die industrielle Revolution des 19. Jahrhunderts mit den aktuellen Umbrüchen zu vergleichen?

Lars Thomsen: Aus der Vergangenheit kann man gewisse Muster erkennen und ableiten, die auch in der Zukunft ihre Gültigkeit haben werden. Menschliches Verhalten lässt sich durch das Studium der Vergangenheit besser begreifen, als wenn man einfach nur die Gegenwart betrachten würde. Dabei wird einem klar, dass Innovation immer schon der Treiber der Veränderungen in Gesellschaft, Arbeit und dem Leben der Menschen war. Die soziale Akzeptanz von Durchbruchsinnovationen war auch in der Vergangenheit am Anfang einer Entwicklung sehr niedrig. Als zum Beispiel auf der Weltausstellung in Paris im Jahr 1900 elektrischer Strom „für Jedermann" in Aussicht gestellt wurde, war die Mehrzahl der Bevölkerung strikt dagegen, elektrischen Strom in Häuser zu legen. Heute sehen wir ähnliche Vorbehalte gegen künstliche Intelligenz, Elektroautos, Roboter und viele andere Technologien.

Wo sehen Sie die deutsche IT-Branche in 10 Jahren? Was sind die größten Herausforderungen?

Lars Thomsen: Künstliche Intelligenz bedeutet für die IT-Branche die wohl wichtigste Herausforderung, weil sie im Kern etwas ganz anderes darstellt, als die Batch-Verarbeitung oder klassische Programmierung. Machine Learning stellt ein komplett neues Paradigma dar, wie in Zukunft Menschen und Maschinen interagieren. Die Abkürzung IT steht für Informationstechnologie. Man muss nicht unbedingt ein Zukunftsforscher sein, um zu erkennen, dass es um viel mehr als Informationen geht: Es geht um Intelligenz und das Zusammenspiel von menschlicher und künstlicher Intelligenz in fast jedem Aspekt der Arbeit, Gesellschaft, Bildung und Wirtschaft. Das bedeutet, dass auch die IT-Branche und die Menschen, die in ihr arbeiten, neu denken und neu lernen müssen. Und nicht nur unsere Kinder – wir alle!

Vielen Dank für das Interview, Herr Thomsen!

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

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@doag.org

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: Original: © mamanamsai | *https://de.123rf.com*
S. 11: Bild © bakhtiarzein | *https://de.fotolia.com*
S. 13: Bild © Freepik | *https://de.freepik.com*
S. 17: Bild © mrcocoa | *https://de.123rf.com*
S. 19: Bild © scyther5 | *https://de.123rf.com*
S. 21: Bild © aurielaki | *https://de.123rf.com*
S. 26: Kaffee © coolvector | *https://de.freepik.com*
S. 26: Flammen © Freepik | *https://de.freepik.com*
S. 29: Bild © Валентин Амосенков | *https://de.123rf.com*
S. 38: Bild © ronstik | *https://de.123rf.com*
S. 44 + 46: Bilder © Adriana Harakalova
S. 48: Bild © Oleksandr Omelchenko | *https://de.123rf.com*
S. 54: Bild © wrightstudio | *https://de.123rf.com*

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

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

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

Alle Rechte vorbehalten. Jegliche Vervielfältigung oder Weiterverbreitung in jedem Medium als Ganzes oder in Teilen bedarf der schriftlichen Zustimmung des Verlags.

Die Informationen und Angaben in dieser Publikation wurden nach bestem Wissen und Gewissen recherchiert. Die Nutzung dieser Informationen und Angaben geschieht allein auf eigene Verantwortung. Eine Haftung für die Richtigkeit der Informationen und Angaben, insbesondere für die Anwendbarkeit im Einzelfall, wird nicht übernommen. Meinungen stellen die Ansichten der jeweiligen Autoren dar und geben nicht notwendigerweise die Ansicht der Herausgeber wieder.

Inserentenverzeichnis

Cofinpro S. 15
DOAG U 3, U 4
EOUC S. 43
iJUG S. 10, S. 53
Virtual7 U 2

2019
DOAG
Konferenz + Ausstellung
19. - 22. November in Nürnberg
PROGRAMM
ONLINE
2019.doag.org
Eventpartner:
AOUG
SOUG
swiss oracle user group
iJUG
Verbund
ORACLE

17. - 19. März 2020 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von:

Community Partner:

www.javaland.eu