GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

Sept. / 2021

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

JAVAPRO

DOMAIN DRIVEN DESIGN

GRUNDLAGEN UND PRAXIS

Mit freundlicher Unterstützung unserer Partner.

JAVAPRO PARTNER NETWORK

Die JAVAPRO wird von den Mitgliedern des JAVAPRO Partner Network finanziert und aktiv unterstützt. Dadurch sind wir in der Lage, redaktionell unabhängig zu arbeiten und die JAVAPRO kostenlos für die gesamte Java-Community zu produzieren sowie die Java Konferenz JCON 2021 zu veranstalten.

GOLD PARTNER

ORACLE · Fast Lane · MicroStream · eclipse · XDEV

SILBER PARTNER

RAPID clipse · consol Wir unternehmen IT · trivago

BRONZE PARTNER

viadee IT-Unternehmensberatung · raytion · GEBIT Solutions · JCON 2019 DEVCON · 4DSoft · Michael Page · ETECTURE

Werden auch Sie Mitglied des JAVAPRO Partner Network und unterstützen Sie die JAVAPRO - Das kostenlose Fachmagazin für die Java Community.

www.javapro.io/partner

JAVAPRO

Impressum

JAVAPRO

Verlag:
JAVAPRO
Im Gewerbepark 29
92681 Erbendorf
Telefon: +49 (0) 800 5282 776
E-Mail: info@javapro.io
Website: http://www.javapro.io

Chefredakteur:
Markus Kett (V.i.S.d.P.)

Redaktion:
info@javapro.io

Gestaltung, Layout, Produktion:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf

Preis: kostenfrei

Illustrationen:
Pixabay (Public Domain)

Erscheinungsweise: Vier mal jährlich

Gründungsjahr: 2017

#JAVAPRO #Editorial

Liebe JAVAPRO Leser,

Domain-Driven-Design (DDD) ist mittlerweile schon 17 Jahre alt, aber durch den immer stärkeren Trend weg von monolithischen Anwendungen und hin zu Microservices ist DDD so aktuell wie nie zuvor.

Eine Microservicearchitektur setzt die Aufteilung der Anwendung in viele, möglichst kleine Services voraus, die ganz nach dem Design-Prinzip Separation-of-Concerns jeweils nur für eine ganz bestimmte Aufgabe zuständig sind und anders als Module völlig unabhängig voneinander in einem eigenen Prozess und self-contained, d.h. in ihrem eigenen Container laufen. Ein solches System bietet vor allem für große und geschäftskritische Anwendungen sehr vielversprechende Vorteile: Bei einem Ausfall einzelner Services läuft das Gesamtsystem weiter, wenn auch mit entsprechenden Einschränkungen. Services, die nicht permanent benötigt werden, können heruntergefahren werden, was Infrastrukturkosten spart und die in Zukunft für Unternehmen immer wichtiger werdende Klimabilanz verbessert. Die einzelnen Services können je nach Notwendigkeit in eigenen Intervallen und damit sehr viel einfacher gewartet, weiterentwickelt und ggf. migriert werden. Die Entwickler müssen nicht mehr das gesamte System beherrschen, sondern nur noch die Services an denen sie arbeiten.

Die Theorie klingt überzeugend. Bei der Umsetzung stehen Entwicklerteams jedoch vor großen Herausforderungen, auf die man sich intensiv vorbereiten sollte. Zum einen sind dies technische Herausforderungen. Anstelle von Full-Stack-Frameworks wie Jakarta EE oder Spring kommen moderne Microservice-Frameworks wie Quarkus, Helidon, Micronaut oder Spring Boot zum Einsatz. Die Services laufen nicht mehr auf der JVM, sondern werden mit GraalVM als Native-Image deployt, wodurch ein Service bis zu 50 Mal schneller startet und bis zu 5 Mal weniger Speicher verbraucht. Für Deployment, Test und Produktivbetrieb werden eine Container-Plattform wie Kubernetes, eine gut vorbereitete CI-/CD-Pipeline mit automatisierten Tests und Tools für das Monitoring benötigt. Richtig anspruchsvoll wird es beim Thema Persistenz und Datenkonsistenz. Alle diese Themen behandeln wir deshalb regelmäßig in der JAVAPRO.

Die meisten Projekte scheitern jedoch bekanntlich nicht an technischen Herausforderungen, sondern an organisatorischen – und damit sind wir beim Titelthema dieser Ausgabe. Für die meisten Teams stellt sich neben technischen Herausforderungen die noch viel wichtigere Frage, wie die Anwendung am besten in einzelne Services aufgeteilt werden soll. Was genau soll oder darf von einem einzelnen Microservice abgedeckt werden und wie groß darf ein einzelner Service eigentlich sein? Domain-Driven-Design ist prädestiniert genau diese Fragen zu beantworten. Da sich viele Entwickler noch nicht intensiv damit beschäftigt haben, beginnen wir in dieser Ausgabe bei den Grundlagen von Domain-Driven-Design, liefern viele wertvolle Tipps für die Praxis und machen dann einen Deep-Dive in das Thema.

Markus Kett
Chefredakteur
JAVAPRO

Twitter: @MarkusKett
LinkedIn: markuskett

#JCON2021
www.jcon.one

JAVAPRO

OCTOBER
5-8

JCON 2021

JCON-ONLINE 2021 LIVE !

Internationale Java Community Conference

5. - 8. OKTOBER

Alle 4 Tage inkl. Workshop-Day nur 99,- EUR. Für alle Java-User-Group Mitglieder frei !

www.jcon.one

4	5	8	110+	150+	2500+
Days	Streams parallel	Haupt-Themen	Speaker	Sessions Live !	Teilnehmer Online

PARTNER 2021

GOLD PARTNER

ORACLE XDEV MicroStream

helidon.io Open Liberty payara MICRONAUT azul RAPID clipse

ORGANISATIONS-PARTNER

JAVAPRO XDEV

JCON-ONLINE 2021

Die JCON-ONLINE 2021 ist inzwischen eine der größten und beliebtesten Java Community Online-Konferenzen und die einzige Java-Konferenz, zu der alle Java-User-Group Mitglieder freien und uneingeschränkten Zugang haben.

Die 5. JCON wird erneut eine reine Online-Konferenz. Alle Sessions werden jedoch live gestreamt. Letztes Jahr waren über 2.100 Teilnehmer aus insgesamt 51 Ländern weltweit online mit dabei. In diesem Jahr werden weit mehr als 2.500 Teilinehmer erwartet.

Auf der JCON liegt der Fokus traditionell auf Core-Java, Enterprise Java, Frameworks und APIs. Auch heuer bilden 2 Live-Streams dazu die Hauptkonferenz. Des weiteren gibt es an 3 Tagen einen Live-Stream zu unserem diesjährigen Hauptthema 'Java-Cloud-Native und Microservices'. Zudem gibt es Live-Stream zu den Themen Agile, Cloud-Platforms, DevOps, Architecture, Web-Development und Low-Code. Am Freitag, den 8. Oktober findet erneut ein Online-Workshop-Day mit bis zu 20 hochkarätigen Online-Workshops statt. Insgesamt bietet die JCON auch heuer wieder 5 Live-Streams zeitgleich, 6 spannende Keynotes und bis zu 150 Vorträge und ist gespickt mit über 110 internationalen Top-Speakern. Die Teilnahme kostet nur sensationelle 99,00 Euro für alle 4 Tage mit Zugang zu allen Sessions, Workshops und Aufzeichnungen. Alle Java-User-Group Mitglieder können ein kostenloses JUG-Ticket buchen und erhalten damit freien Zugang zur JCON-ONLINE 2021. Insgesamt sind 1.000 JUG-Tickets verfügbar. First come, first serve!

Die JCON-ONLINE 2021 wird organisiert von der Java-User-Group Oberpfalz in Kooperation mit der JAVAPRO und wird von 17 internationalen Java-User-Groups unterstützt.

#JCON2021

OCTOBER 5-8

JCON 2021

JCON-ONLINE 2021 LIVE!

Internationale Java Community Conference

5. - 8. OKTOBER

OTOBER 5-8

Featured Speaker

Adam Bien
Key Account Manager und Advisor
Oracle

Emily Jiang
Liberty Cloud Native Architect
and Chief Advocate at IBM

Cesar Hernandez
Senior Software Engineer
Tomitribe

Uwe Friedrichsen
Key Account Manager und Advisor
Oracle

Eberhard Wolff
Fellow
innoQ

Graeme Rocher
Architect
Oracle Labs

Hendrik Ebbers
JavaPunk working at Karakun
and Eclipse Adoptium

Ivar Grimstad
Java Champion, Jakarta EE Developer
Advocate at Eclipse Foundation

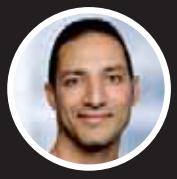
Kathryn Kodama
Developer at Open Liberty Developer
Experience Team at IBM

Martin Lippert
Spring Tools Lead
Vmware

Mary Grygleski
President of Chicago-JUG,
Senior Developer Advocate at IBM

Mete Adamel
Developer Advocate
Google

Markus Kett
Editor in Chief at JAVAPRO Magazine
CEO at MicroStream



Mohammed Taman
Java Champion, JCP Member,
Chief Solutions Architect at effortel

Oleg Šelajev
Developer Advocate GraalVM
Oracle

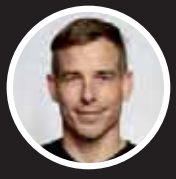
Reza Rahman
Principal Program Manager
Java on Azure at Microsoft

Robin Moffatt
Senior Developer Advocate
Confluent

Rudy De Busscher
Developer Advocate
Payara Ltd.

Rory Preddy
Senior Cloud Advocate
Microsoft

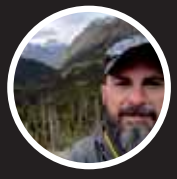
Richard Fichtner
Leiter JUG-Oberpfalz,
CEO at XDEV Software GmbH



Sven Ruppert
Developer Advocate
JFrog

Tomáš Langer
Helidon Architect
Oracle

Werner Keil
Director
Creative Arts & Technologies



Todd Sharp
Developer Advocate Cloud and
Autonomous DB at Oracle

Burt Ceckwith
Micronaut Developer
Oracle Labs

Carola Lilienthal
Geschäftsführerin
Workplace Solutions GmbH

Dmitry Alexandrov
Senior Principle Developer
Oracle

Grace Jansen
Developer Advocate
IBM

Wolfgang Weigend
Oracle Solution Engineer Java & GraalVM
Oracle

Marcus Hellberg
Head of Community
Vaadin

... und weitere 70 internationale Speaker !

JCON-ONLINE 2021 LIVE!

Internationale Java Community Conference

5. - 8. OKTOBER

OTOBER 5-8

AGENDA

Dienstag 5. Oktober	Mittwoch 6. Oktober	Donnerstag 7. Oktober	Freitag 8. Oktober
2 Main Streams - Core Java, Serverside Java, APIs			
Main-Conference Stream 1 (Englisch) 9:00 - 22:00 Uhr			**Training Day** 9:00 - 13:00 Uhr 20 hochkarätige Workshops
Main-Conference Stream 2 9:00 - 22:00 Uhr			
Special Topic Streams			
Microservices Live-Stream	**Microservices** Live-Stream	**Microservices** Live-Stream	
Agile Live-Stream	**Cloud Platforms** Live-Stream	**DevOps** Live-Stream	
Low-Code Live-Stream	**Architecture** Live-Stream	**Web Development** Live-Stream	

TICKETS

JUG Ticket

- Gültig für alle 4 Konferenz-Tage
- Zugang zu allen Online-Sessions
- Zugang zu allen Workshops
- Fragen direkt an Speaker stellen
- Zugang zu allen Aufzeichnungen
- Nur für Mitglieder einer Java-User-Group
- Max. 1.000 JUG-Tickets verfügbar!

0.00 EUR

4-Tages Ticket

- Gültig für alle 4 Konferenz-Tage
- Zugang zu allen Online-Sessions
- Zugang zu allen Workshops
- Fragen direkt an Speaker stellen
- Zugang zu allen Aufzeichnungen

NUR 99.00 EUR

Alle auf dieser Seite aufgeführten Preise sind Nettopreise zzgl. 19% MwSt.!

Max. 1.000 JUG-Tickets verfügbar. First come, first serve!

Jetzt JUG-Ticket sichern: www.jcon.one

Haben Sie Fragen zur JCON 2020 ? Wir beraten Sie gerne!

Fon: +49 (0)800 – 52 82 776 | E-Mail: info@jcon.one

#JCON2021

#JAVAPRO #CoreJava #Performance

Primitive-Types vs. Wrapper-Classes

Heutige Anforderungen stellen ständig neue Herausforderungen in Sachen Performance an die Software. Wer bereits alle algorithmischen Optimierungen ausgereizt hat, muss sich mit den Low-Level-Funktionen der JVM auseinandersetzen. Dieser Artikel zeigt, was man durch das Verwenden von Primitive-Types gegenüber ihren Wrapper-Classes-Pendants an Performance gewinnen und an Speicher einsparen kann und wie man einen Micro-Benchmark für beide Szenarien schreibt.

Die meisten Softwareprojekte haben eine über die Jahre gewachsene Codebase, an der durchgehend gearbeitet wird. Dabei fallen immer wieder Code-Stellen auf, die teilweise sehr alt sind und von Zeit zu Zeit Probleme verursachen. Manchmal entstehen die Probleme auch durch Refactorings und das Ändern weiter Teile der Software. Parallel zu dieser Entwicklung steigen die Anforderungen der Kunden stetig. Die abzubildenden Datenmodelle sind in ihrer Komplexität in den letzten Jahren um ein Vielfaches gewachsen. Die Anforderungen großer, heutiger Kunden und Konzerne stellen ständig neue Herausforderungen in Sachen Performance an die Software. Bei der gegenwärtigen Menge an Operationen machen auch kleine Optimierungen viel aus, insbesondere wenn algorithmische Verbesserungen wie das Reduzieren der Komplexität bereits ausgereizt wurden.

Autor:

Björn Stahl ist Senior Java Developer bei der LucaNet AG und arbeitet im TeamPerformance. Dort kümmert er sich um die Analyse von Performanceproblemen. Er testet und entwickelt Tools, die die Analysen vereinfachen. Nebenbei beschäftigt er sich weiter mit Microservices, High-Traffic Low-Latency Applications und Garbage Collection sowie Performance Tuning.

https://www.lucanet.de/
https://twitter.com/Mr_Steel
https://github.com/Mr-Steel

“There is no such thing as free lunch”

Zur Optimierung einer Software gehört standardmäßig die Analyse von Performance-Problemen mit einem Profiler. Während einer dieser Sessions fielen dem Autor sehr viele `toLong()` und `toInt()` sowie `Integer.valueOf()` und `Long.valueOf()` Aufrufe auf, welche einen Hinweis auf eine starke Verwendung von Autoboxing darstellen. Diese benötigten nicht den größten

Teil der verbrauchten Zeit, deshalb blieben sie längere Zeit unbeachtet. Da die Software bereits gut optimiert war, blieben nicht mehr viele Möglichkeiten, noch mehr Performance herauszuholen. Wie groß ist also das mögliche Optimierungspotential beim Autoboxing?

Autoboxing

Autoboxing stellt eines dieser schönen Features dar, die man bewusst nicht mehr wahrnimmt, aber häufig viel Arbeit abnehmen. Es macht insbesondere den Umgang mit Collections einfacher, da man sich nicht mehr explizit um die Konvertierung eines Primitive-Types zu der korrespondieren Wrapper-Class und umgekehrt kümmern muss. Dieses Feature gibt es bereits seit Java 5, deshalb hier noch mal eine kurze Erinnerung, was Autoboxing genau macht. Autoboxing konvertiert primitive Typen in die korrespondierenden Objekte und umgekehrt. Oracle bietet auf seiner Tutorial-Seite einen eigenen Abschnitt zu diesem Thema. Für unsere Messungen ist interessant, was der Compiler im Hintergrund macht.

(Listing 1)

```
List<Integer> list = new ArrayList<>();
    for (int i=1; i<50; i+=2) {
        list.add(i);
    }
```

(Listing 2)

```
List<Integer> list = new ArrayList<>();
    for (int i=1; i<50; i+=2) {
        list.add(Integer.calueOf(i));
    }
```

Wir haben hier also mindestens einen zusätzlichen Methodenaufruf pro Iteration. Was kostet also dieser versteckte Aufruf?

Micro-Benchmark to the rescue!

Für diese Art der Messung kommt ein Micro-Benchmark zum Einsatz. Doch was genau ist das? Ein Micro-Benchmark, ist ein kleiner künstlicher Benchmark welcher dazu gedacht ist, eine Methode oder einen Algorithmus zu testen und verschiedene Szenarien und/oder Implementierungen miteinander zu vergleichen. Der Einsatz ist vor allem dann sinnvoll, wenn einzelne Methoden oder Code-Abschnitte getestet werden sollen, die auf einem kritischen Pfad liegen. Beispiele dafür sind die Verwendung von verschiedenen Datenstrukturen, Vergleich von Sortiermethoden und ähnliches. Die korrekte Messung eines solchen Methodenaufrufs ist nicht trivial, da die Methode nur wenige Nanosekunden braucht. Hinzu kommt, dass bei jeder Messung kleine Fehler und Seiteneffekte das Ergebnis verfälschen. Beispielsweise kann die Ausführung auf einem Laptop durch das Heruntertakten der CPU auf Grund von Hitzeproblemen zu erheblichen Schwankungen in den Resultaten führen. Dies muss man immer beachten und beispielsweise durch entsprechende Kühlung verhindern. Zudem sollte das Netzteil während der Messungen angeschlossen werden. Andere Seiteneffekte entstehen wiederum durch die Java-Virtual-Machine selbst. Eine der Stärken des JIT-Compilers, die Optimierung des Codes während der Ausführung, wird bei der Messung zu einem großen Fallstrick. Zuverlässige Vergleiche von Messergebnissen werden damit sehr schwierig.

JMH – Java-Micro-Benchmarking Harness[1]

Mit Hilfe des Java-Micro-Benchmarking Harness ist es deutlich einfacher geworden, einen verlässlichen Micro-Benchmark zu schreiben. JMH ermöglicht es die JVM explizit vorzuwärmen, d.h. gezielt so viele Aufrufe zu tätigen, dass der JIT-Compiler getriggert wird und die Optimierungen zum Messzeitpunkt bereits so weit möglich vorgenommen wurden. Es werden auch einige Optimierungen wie Dead-Code-Elimination verhindert, da diese beim Benchmark eher hinderlich sind. Hierfür ist es wichtig bei jedem Benchmark ein Ergebnis zurückzugeben, damit genau das verhindert wird. Wenn dies nicht so einfach möglich ist oder auch mehrere Ergebnisse berechnet werden, dann kann ein sogenanntes Black-Hole-Objekt verwendet werden. Dieses Objekt stellt eine Consume-Methode zur Verfügung und gaukelt bei deren Aufruf mit den Ergebnissen der JVM vor, dass diese Ergebnisse auch tatsächlich verwendet und damit nicht wegoptimiert werden dürfen.
Weiterhin ist die Ausführungsumgebung interessant für den Performancegewinn. Dabei spielt nicht nur die Hardware, sondern auch die Version der JVM eine entscheidende Rolle. Es werden ständig Optimierungen vorgenommen, welche unterschiedliche Auswirkungen auf die Ausführungsgeschwindigkeit haben. Mit jeder neuen Version, besser mit jedem Update, sollte ein Benchmark noch einmal ausgeführt werden. Nur so kann man sicherstellen, dass die Performance der kritischen Stellen in der Applikation mindestens gleichbleibt. Für diese Zwecke bietet sich am Besten ein dedizierter Jenkins-Job oder ein anderes CI-Tool an. Hierbei muss unbedingt darauf geachtet werden, dass der ausführende Job isoliert läuft. In der Praxis bedeutet das, eine isolierte Jenkins-Instanz wird erzeugt, auf der nur ein Job gleichzeitig ausgeführt werden darf und auf der nichts anderes im Hintergrund läuft. Die Messergebnisse können sonst massiv verzerrt werden und führen zu falschen Schlussfolgerungen.

Zeit zum Messen

Ein erster Micro-Benchmark soll klären, wie sich das Autoboxing auf die Performance einer einfachen Addition von 2 Zahlenwerten auswirkt. Hierfür schreiben wir 3 Methoden, welche die 3 unterschiedlichen Fälle abbilden: Addition zweier `Int` Primitive-Types,

Addition eines `Int` Primitive-Types mit einem Integer-Objekt und die Addition zweier Integer-Objekte.

```
AutoBoxingBenchmark.sum_Int_Int           avgt  2,460 ns
AutoBoxingBenchmark.sum_Int_Integer       avgt  4,822 ns
AutoBoxingBenchmark.sum_Integer_Integer   avgt  4,983 ns
```

Das Ergebnis zeigt unerwartet deutlich: die Addition der `Int` Primitive-Types ist fast doppelt so schnell verglichen mit der Addition zweier Integer-Objekte. Hier macht sich eindeutig der zusätzliche Methodenaufruf bei jedem Integer-Objekt bemerkbar. Doch wie sieht dieser Effekt aus, wenn die Addition in einer Schleife stattfindet? In der Praxis werden eher selten nur zwei einzelne Werte addiert. Der erwartete Effekt sollte hier etwas geringer ausfallen, da die JVM diverse Möglichkeiten hat, Schleifen in der Ausführung zu optimieren.

Java unterstützt derzeit keine Listen von Primitive-Types, deshalb muss für die Messung eine zusätzliche Bibliothek verwendet werden, die dieses Feature unterstützt. Es gibt seit geraumer Zeit alternative Implementierungen der Java-Collections und eine davon sind die Eclipse-Collections[2]. Ursprünglich im Jahr 2012 von Goldman Sachs unter dem Namen GS-Collections auf GitHub veröffentlicht, wurde die Bibliothek 2015 zur Eclipse-Foundation migriert und wird dort kontinuierlich weiterentwickelt. Die Eclipse-Collections bieten für Listen und Maps Implementierungen an, die Primitive-Typs unterstützen. Die Bibliothek bietet aber noch weit mehr, z.B. Immutable-Maps/Lists. Dabei handelt es sich um performantere Implementierungen der Standard-Maps/Lists, die direkt als Drop-In-Replacement verwendet werden können, da die Standard Java-Interfaces implementiert werden. Des Weiteren gibt es sogar neue Datenstrukturen wie Bags und MultiMaps.

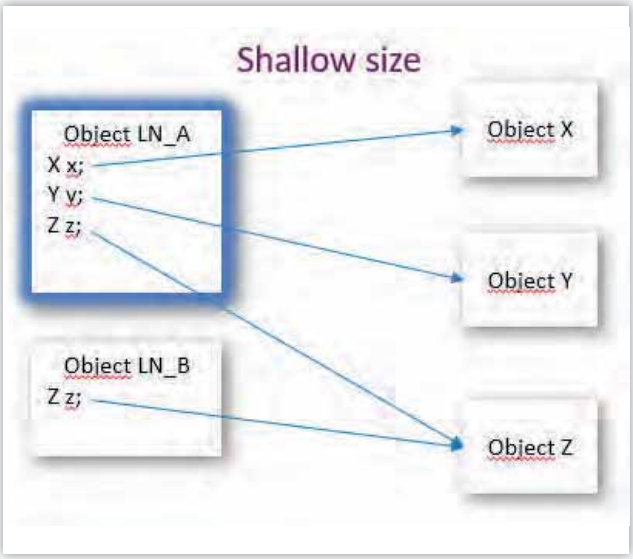
Ein direkter Vergleich mit der optimierten Eclipse-Collections Implementierung hätte wenig Aussagekraft, daher enthält der nächste Benchmark die optimierte List-Implementierung FastList der Eclipse-Collections als Äquivalent zur ArrayList aus dem Standard-JDK.

```
AutoBoxingBenchmark.sum_Integer_Loop_ArrayList
avgt      163,591 ns
AutoBoxingBenchmark.sum_Integer_Loop_FastList
avgt      147,967 ns
AutoBoxingBenchmark.sum_Int_Loop_PrimitiveList
avgt      97,714 ns
```

Laufzeit ist nicht alles

Neben dem klar messbaren Gewinn an Performance beim Micro-Benchmark bei der Verwendung von Primitive-Types statt Objekten, ist auch der geringere Speicherverbrauch in der Praxis nicht zu vernachlässigen. Wenn es darum geht, das letzte Quäntchen Performance herauszuholen, kann es sich auch lohnen, den Speicherverbrauch genau unter die Lupe zu nehmen. Jedes Objekt muss erstellt und später durch den Garbage-Collector wieder abgeräumt werden, was ebenfalls Prozessorleistung und Zeit kostet. Selbst wenn man nur kurzlebige Listen oder andere Datenstrukturen mit vielen Objekten verwendet, kann man hier durch die Verwendung von Primitive-Types den Druck auf den Garbage-Collector verringern und indirekt ebenfalls die Performance verbessern.

Den Speicherverbrauch von Java-Objekten zu bestimmen gestaltet sich nicht ganz einfach. Tatsächlich gibt es hier mehrere Möglichkeiten, die Größe eines Objekts zu beschreiben. Eine davon ist die Shallow-Size. Dabei handelt es sich um die Größe des Objekts selbst inklusive der Referenzen zu anderen Objekten **(Abb. 1)**.

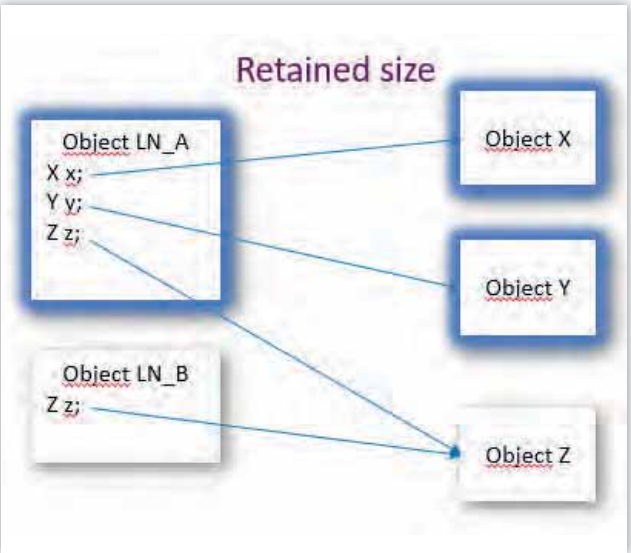


Shallow-Size. (Abb. 1)

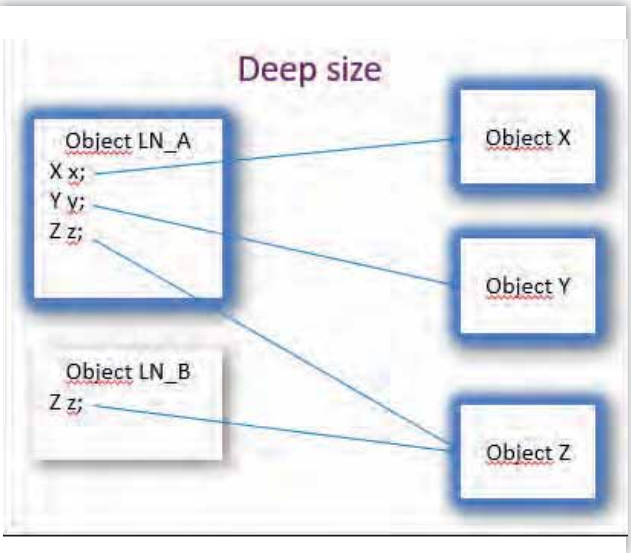
Zudem gibt es die Retained-Size **(Abb. 2)**, das ist die Größe des Objekts inklusive der Referenzen zu anderen Objekten plus die Größe der Objekte, die nur von diesem Objekt referenziert werden. Das ist der Speicher, der durch den Garbage-Collector freigegeben wird, wenn das Objekt abgeräumt wird.

Zuletzt gibt es die Deep-Size **(Abb. 3)**. Das ist die Summe der Größe des Objekts plus der Größe aller referenzierten Objekte. Das zeigt den Maximalwert des Speicherverbrauchs des Objekts an, wenn keines der referenzierten Objekte andere eingehende Referenzen hat, quasi den Worst-Case.

Im Normalfall bestimmt man diese Größen mit einem Profiler. Allerdings ist es gerade bei großen Applikationen schwierig einen genau definierten Zeitpunkt zu messen, in welchem sich das Objekt im gewünschten zu messenden Zustand befindet. Es gibt jedoch eine praktische Bibliothek namens Jamm[3]. Diese stellt einen Java-Agent zur Verfügung, mit dessen Hilfe man die Größe von Objekten messen kann. Allerdings mit der Einschränkung,



Retained-Size. (Abb. 2)



Deep-Size. (Abb. 3)

dass nur die Shallow- und Deep-Sizes gemessen werden können, was jedoch in den meisten Fällen völlig ausreichend ist. Die Verwendung ist relativ einfach. Man instanziiert ein `MemoryMeter-Objekt` und hat dann zwei Methoden zur Messung zur Verfügung: `measure(object)` und `measureDeep(object)`. Die `measure(object)` Methode bestimmt die Shallow-Size des übergebenen Objektes, die andere Methode, wie der Name schon vermuten lässt, die Deep-Size des Objekts. Die Dokumentation gibt selbst als Disclaimer an, dass die Objektgrößen nur Annäherungen sind, die in der Praxis jedoch sehr genau sind. Hier als Beispiel die Messung einer ArrayList. Der vollständige Code des Benchmarks befindet sich auf Github.

(Listing 3)

```
  MemoryMeter meter = new MemoryMeter();
  List<Integer> integerArrayList = new ArrayList<>(initialCapa-
city);
  long measureIntegerArrayList = meter.measure(integerArrayList);
  long measureDeepIntegerArrayList = meter.measureDeep(integerAr-
rayList);
  System.out.println("ArrayList size norm: " + measureIntegerAr-
rayList);
  System.out.println("ArrayList size deep: " + measureDeepInte-
gerArrayList);
```

Die Ergebnisse sind recht deutlich. Während ein einzelnes Objekt natürlich nicht viel ausmacht (4 Bytes int gegenüber 16 Bytes Integer), sieht es in einer Liste vielleicht schon völlig anders aus. Eine Messung über jeweils eine Liste mit 250 Einträgen macht dies deutlich:

ArrayList mit Integer-Objekten: `5040 Bytes`
IntList mit Primitive-Ints: `1040 Bytes`

Die List mit den Primitive-Types kommt also mit fast einem Fünftel des Speicherbedarfs gegenüber der ArrayList mit der Wrapper-Class aus. Wenn also der Speicher in einer Anwendung knapp ist oder einfach wenig Kapazität zur Verfügung steht, sollte die Verwendung einer alternativen Listenimplementierung mit Primitive-Types in Betracht gezogen werden.

Fazit:

Der größte Performance-Boost ist immer noch eine Operation, die gar nicht erst ausgeführt werden muss. Wenn aber bereits alle algorithmischen Optimierungen ausgelotet wurden, dann muss man sich mit den Low-Level-Funktionen der JVM auseinandersetzen.

Dieser Artikel stellt dar, was man durch das Verwenden von Primitive-Types gegenüber ihren Wrapper-Classes-Pendants an Performance gewinnen oder auch an Speicher einsparen kann. Getreu dem Motto „Benchmarking beats lots of theory and speculation" wurde auch gezeigt, wie ein entsprechender Micro-Benchmark für beide Szenarien geschrieben werden kann.
Hier kann natürlich nur ein kleiner Einblick in das Thema und die möglichen Alternativen gegeben werden. Es gibt neben den vorgestellten Bibliotheken selbstverständlich auch noch weitere.

Quellen:

1 http://bit.ly/jmh-1
2 http://bit.ly/collections-c2
3 http://bit.ly/jamm-3

#JAVAPRO #CloudNative #Kubernetes #Quarkus

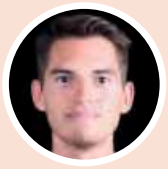
Kubernetes-Native mit Quarkus und AWS-EKS-Fargate

In den letzten Jahrzehnten war die Client-Server-Architektur der Standard für die Anwendungsentwicklung. Nach und nach entstanden eine neue Reihe von Anwendungen und Architekturstilen, welche die Art und Weise beeinflussten, wie Code geschrieben wird und Anwendungen ausgerollt und ausgeführt werden. Dieser Artikel zeigt auf, wie Entwickler heute Native-Java-Anwendungen mit Quarkus auf Amazon-Elastic-Kubernetes-Service (EKS) mit AWS-Fargate erstellen und die Herausforderungen der modernen Anwendungsentwicklung bewältigen können.

Die Bereitstellung von Microservices, reaktiven Anwendungen, Event-getriebenen Microservices und Serverless-Anwendungen in der Cloud haben sich als wichtige Architekturmuster in modernen Anwendungssystemen etabliert. Gleichzeitig bringt die Entwicklung nativer Java-basierter Anwendungen in der Cloud Herausforderungen mit sich, wie zum Beispiel native Integration mit der Container-Infrastruktur, langsame Startzeit und großen Platzbedarf. Klassische Java-EE-Anwendungen sind in vielen Fällen als Drei-Schichten-Architektur entwickelt, die auf einer einzigen Maschine eingesetzt werden und den gesamten Speicher dieses Systems für die Arbeitslast reservieren. Oft wurden mehrere Anwendungen auf ein und demselben Rechner eingesetzt, um die verfügbaren Ressourcen optimal zu nutzen, was bei klassischen Applikations-Servern über das Deployment mehrerer WAR- oder EAR-Archive erreicht wird.

Heutzutage zeichnet sich im Hinblick auf die moderne Architektur deutlich ab, dass diese Anwendungen in kleinere Komponenten aufgeteilt und so orchestriert werden, dass sie auf einer Container-Plattform wie Amazon-Elastic-Container-Services (ECS)[1], Kubernetes oder serverless wie in AWS-Lambda[2] oder AWS-Fargate[3] laufen. AWS-Fargate ist eine Technologie, die bei Bedarf Rechenkapazität in der richtigen Größe für Container auf Abruf bereitstellt. Mit AWS-Fargate müssen keine Gruppen von virtuellen Maschinen mehr bereitgestellt, konfiguriert, gewartet oder skaliert werden, um Container auszuführen. Damit entfällt die Notwendigkeit, Server-Typen auszuwählen oder zu entscheiden, wann die Node-Groups skaliert werden sollen. Eine interessante aktuelle Entwicklung in der Java-Community ist die Implementierung von Java-Anwendungen für Container und Serverless-Workloads mit entsprechend optimierten Plattformen und Frameworks. Im weiteren Verlauf des Artikels wird anhand eines Beispiels gezeigt, wie eine solche Anwendung aussehen kann.

Autor:



Luis Morales arbeitet als Solutions-Architect bei AWS. Zuvor war er als Softwareentwickler und Softwarearchitekt vorwiegend in Cloud Native Microservice Projekten tätig. Er ist spezialisiert auf die Themen Software-Engineering, testgetriebene Entwicklung, verteilte Systeme, Everything-as-Code und Sicherheit.

https://aws.amazon.com/de/
lluim@amazon.de

Ovidiu ist Solution-Architect bei AWS und unterstützt Entwickler und Unternehmen leidenschaftlich dabei, moderne Anwendungen mit den Transformationsfähigkeiten der AWS-Dienste zu erstellen. Seine Berufserfahrung begann 2007 bestehend aus verschiedenen Rollen als Softwareentwickler, Datenbank- und Sys-Administrator und Devops-Ingenieur.

https://aws.amazon.com/de/
ovi@amazon.de

Herausforderungen bei klassischen Java-Anwendungen

Eine der Herausforderungen bei Java-EE-Applikations-Servern besteht darin, dass diese für monolithische Anwendungen optimiert sind, bei denen beim Start der Anwendung viel dynamisches Wiring und Annotation-Scanning durchgeführt wird, was einen kurzfristigen Burst verursacht. Dieses dynamische Verhalten ist im Kontext von klassischen Servern in den meisten Fällen durch entsprechende Überprovisionierung von Ressourcen unproblematisch. Im Kontext von Containern und Serverless kann dieses Verhalten zu unerwarteteten Seiteneffekten wie lange Kaltstartzeiten führen, welche die Skalierung unnötig verlängern. Bei einem Migrationsansatz, der auf Lift-and-Shift basiert, also die Anwendungen unverändert in Container packt, bringt das einige Nachteile mit sich, wie übergroße Container-Images, erhöhte Kaltstartzeiten und ineffiziente Ressourcennutzung. Außerdem beeinträchtigt die Verwendung von Frameworks mit intensiver Nutzung von Reflections die Startzeit und Laufzeit der Anwendung negativ.

Aufbau eines nativen Java-Stacks

Damit Java in einer Container-nativen Infrastruktur effizient arbeiten kann, hat es in den letzten Jahren bemerkenswerte Anstrengungen in Richtung effizienter Ausführungsumgebungen gegeben. Mit der Einführung von Eclipse-MicroProfile[4] sind Entwickler in der Lage, ihre Anwendungen je nach ihren geschäftlichen und technischen Anforderungen auf verschiedenen portablen Laufzeitumgebungen laufen zu lassen. Es existieren verschiedene Implementierungen von MicroProfile, wie zum Beispiel Quarkus[5], das sich auf die effiziente Ausführung von Java-Anwendungen in Containern im Allgemeinen und Kubernetes im Besonderen konzentriert.

Was ist Quarkus?

Das Ziel von Quarkus ist es, Java zu einer führenden Plattform auf Kubernetes und Serverless-Umgebungen zu machen. Es bietet Entwicklern eine einheitliche Konfiguration, das Live-Nachladen der Applikation und eine einfache Möglichkeit zur nativen Code-Generierung. Außerdem bietet es die Möglichkeit, nicht blockierende und imperative Entwicklungsstile in einem einzigen Modell zu vereinen, um ein breiteres Spektrum verteilter Anwendungsarchitekturen optimal anzusprechen. Um die Entwicklung einfach und flexibel zu gestalten, baut Quarkus auf bewährten Standards auf. Von offiziellen Standards wie MicroProfile, spezialisierten Frameworks wie Vert.x, Dependency-Injection auf der Basis von CDI, JAX-RS-Annotationen zur Definition von REST-Endpunkten und vielen mehr. Quarkus wurde mit dem Gedanken der Container-First-Strategie entwickelt. Das bedeutet, dass es für eine geringe Speichernutzung und eine schnelle Startzeit optimiert ist, was es ideal für eine schnelle Skalierung in einer Container-Umgebung macht. Hierfür wird die Anwendung während der Erstellungszeit so optimiert, dass sie zur Laufzeit nur die tatsächlich benötigten Klassen enthält und die Verwendung von Reflections reduziert wird. Bei der Kompilation in ein natives Artefakt wird der Großteil des Startup-Codes bereits ausgeführt und das Ergebnis in die ausführbare Datei serialisiert. Für die Ausführung der Anwendung ist Quarkus für die OpenJDK-HotSpot-JVM[6] und GraalVM[7] zugeschnitten. Wir werden uns auf letztere konzentrieren, da sie laut GraalVM-Website den Speicherbedarf um das bis zu 5-fache und die Startzeit um das 50-fache reduzieren kann.

GraalVM-Integration

Quarkus lässt sich nahtlos in die GraalVM integrieren, so dass Java-Code in eine native Binärdatei kompiliert werden kann, sprich Maschinen-Code, der ohne JVM läuft. Das bringt die Leistungsvorteile einer statisch gelinkten Binärdatei und reduziert die Zeit, die für die Bedienung der ersten Anfrage benötigt wird. Die Leistungsverbesserung ist beachtlich. Zum Beispiel startet eine kleine Demo-REST Anwendung, die zu einer nativen Binärdatei kompiliert wurde, in 0,016 Sekunden. GraalVM kompiliert nicht nur Java-Code zu nativen Binärdateien, sie ersetzt auch den Just-in-Time (JIT) C2-Compiler, den es seit Java 1.3 gibt, durch einen Ahead-of-Time Compiler (AOT). Der GraalVM-Compiler ist vollständig in Java implementiert und verwendet das neue JVM-Compiler-Interface (JVMCI), das modernere Optimierungen für die Ausführung von Java-Code ermöglicht.

Die Demo-Anwendung

Für die Verdeutlichung, wie Quarkus zur Implementierung leichtgewichtiger Java-basierter Anwendungen im Kontext von Containern eingesetzt werden kann, wird im Folgenden eine Demo-Anwendung genauer besprochen, die über ein GitHub-Repository[8] verfügbar ist. Es wird behandelt, wie die Anwendung zu einer nativen ausführbaren Datei kompiliert, in ein Docker-Image verpackt und mit AWS-Fargate in Amazon-EKS ausgeführt wird.

Der Einstieg in Quarkus ist ziemlich simpel, denn mit nur einem Maven-Bootstrap-Befehl[9] kann ein erstes Projekt erstellt werden. In unserem Fall ist das ein Projekt mit Maven-Struktur, einer initialen Ressource, einer Landing-Page, einer Anwendungskonfigurationsdatei und Dockerfile-Dateien sowohl für den nativen als auch für den JVM-Modus in `src/main/docker` **(Listing 1)**.

(Listing 1)

```
mvn io.quarkus:quarkus-maven-plugin:1.8.1.Final:create \
    -DprojectGroupId=com.amazon.example \
    -DprojectArtifactId=aws-quarkus-demo \
    -DclassName="com.amazon.example.resource.UserResource" \
    -Dpath="/users"
```

Die Architektur der Anwendung ist einfach gehalten und fokussiert sich im Wesentlichen auf einige wenige Klassen, die einen

REST-Service implementieren, der alle Benutzerdaten in einem Key-Value-Store speichert, in unserem Fall Amazon-DynamoDB[10]. Quarkus bietet eine Erweiterung für Amazon-DynamoDB[11]an, die auf AWS-Java-SDK 2.x[12] basiert. Hier handelt es sich um eine umfassende Neuentwicklung der 1.x Code-Basis, die zwei Programmiermodelle bietet: klassische Blocking-Calls und einen asynchronen Ansatz **(Abb. 1)**.

Damit Kubernetes prüfen kann, ob der Dienst erreichbar ist, wurde ein HealthCheck in der Datei `src/main/java/com/amazon/example/resource/HealthResource.java` definiert. Dieser verwendet JAX-RS-Annotationen zur Definition der REST-Endpunkte.

(Listing 2)

```
package com.amazon.example.resource;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/health")
public class HealthResource {

   @GET
   @Produces(MediaType.APPLICATION_JSON)
   public String check() {
      return "OK";
   }
}
```

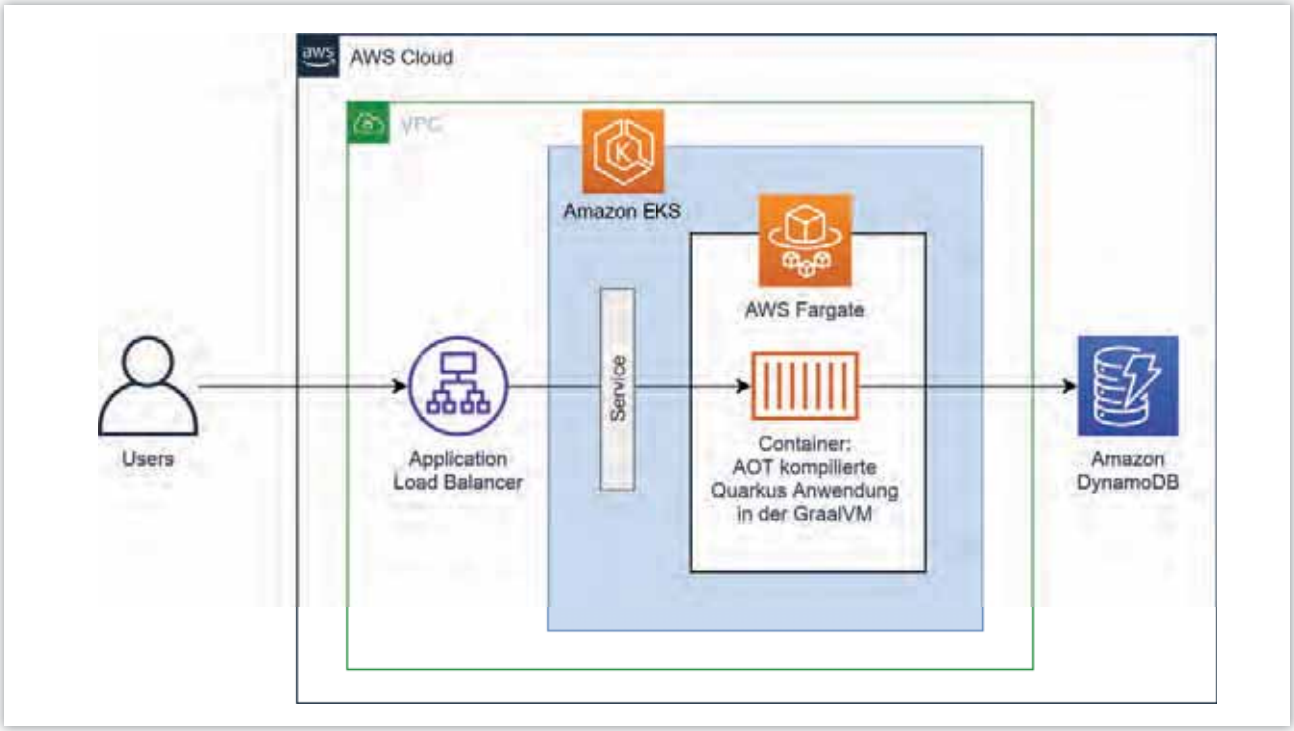
Zum Ausführen dieser Anwendung verfügt Quarkus über einen Modus speziell für die Entwicklungsphase. Dieser ermöglicht einen Hot-Reload mit Kompilierung im Hintergrund, das heißt wenn eine Java- oder eine Ressourcen-Datei geändert und der Browser aktualisiert wird, werden diese Änderungen automatisch wirksam. Die Aktualisierung des Browsers löst einen Scan des Projektes aus. Wenn Änderungen festgestellt werden, werden die Java-Dateien neu kompiliert und die Anwendung wird neu bereitgestellt. Die verschickte Anfrage wird dann von der neu bereitgestellten Anwendung bearbeitet. Mit `./mvnw compile quarkus:dev` kann die Anwendung im Entwicklungsmodus ausgeführt werden. Sobald diese gestartet wird, kann der bereitgestellte Endpunkt unter `http://localhost:8080/health` angefordert werden.

Erstellung der Anwendung

Für die Erstellung der Applikation sind drei einfache Schritte notwendig:

1. Bauen der Applikation
2. Erzeugung des Docker-Container-Images
3. Deployment des Images in Amazon-EKS

Um die Applikation zu bauen, wird das Maven-Profil `native` genutzt, während das Standardprofil ein Uber-JAR erzeugt, eine eigenständige Applikation mit allen notwendigen Abhängigkeiten. Diese ausführbare Datei wird als natives Image bezeichnet und enthält die Anwendungsklassen, Klassen aus ihren Abhängigkeiten, Laufzeitbibliotheksklassen aus JDK und statisch verknüpften nativen Code aus dem JDK. Es wird nicht in der Java-VM ausgeführt, enthält jedoch die erforderlichen Komponenten wie Speicherverwaltung und Thread-Planung von einer anderen virtuellen Maschine namens Substrate-VM[13].



Architektur der Demo-Anwendung (Abb. 1)

Der `package` Befehl führt automatisch alle Tests aus. Da diese eine Abhängigkeit zu Amazon-DynamoDB haben, besteht entweder die Möglichkeit die Tests gegen Amazon-DynamoDB in AWS direkt laufen zu lassen oder auf die lokale Variante[14] auszuweichen **(Listing 3)**.

(Listing 3)

```
$ docker run -p 8000:8000 amazon/dynamodb-local -jar DynamoD-
BLocal.jar -inMemory -sharedDb
$ ./mvwn package
```

Mit dem Kommandozeilenparameter -DskipTests können die Integrationstests übersprungen werden.

Um die Vorteile der GaalVM nutzen und ein natives Runnable erzeugen zu können, wird das Profil `nativ` verwendet, **(Listing 4)**.

(Listing 4)

```
$ ./mvnw package -Pnative -Dquarkus.native.container-build=true
-DskipTests
```

Das erzeugte Artefakt kann direkt über `./target/user-service-1.0-SNAPSHOT-runner` ausgeführt werden. Abschließend muss dieses noch in ein Docker-Container-Image verpackt werden **(Listing 5)**.

(Listing 5)

```
$ docker build -f src/main/docker/Dockerfile.native -t quarkus/
user-service-native .
```

Die entsprechende Docker-Datei nutzt hierfür einen sogenannten Multi-Stage-Docker-Build. Damit kann der Build-Prozess in mehrere Teilschritte zerlegt werden. Dies hat den Vorteil, dass zur Laufzeit die Abhängigkeiten für das Bauen des Images nicht inkludiert werden müssen. Das reduziert die Größe und die Anzahl der Abhängigkeiten in unserem Laufzeit-Image. In unserem Fall wird das `ubi-quarkus-native-image:20.1.0` als Base-Image genutzt und es werden die notwendigen TLS-Dateien (SunEC-Bibliothek und die entsprechenden Zertifikate) kopiert und in das Laufzeit-Image eingefügt **(Listing 6)**.

(Listing 6)

```
FROM quay.io/quarkus/ubi-quarkus-native-image:20.1.0-java11 as
nativebuilder
RUN mkdir -p /tmp/ssl-libs/lib \
  && cp /opt/graalvm/lib/security/cacerts /tmp/ssl-libs \
  && cp /opt/graalvm/lib/libsunec.so /tmp/ssl-libs/lib/

FROM registry.access.redhat.com/ubi8/ubi-minimal as runti-
me-builder
WORKDIR /work/
COPY target/*-runner /work/application
*COPY --from=nativebuilder /tmp/ssl-libs/ /work/*
RUN chmod 775 /work
EXPOSE 8080
CMD ["./application", \
     "-Dquarkus.http.host=0.0.0.0", \
     "-Djava.library.path=/work/lib"*, \*
     "-Djavax.net.ssl.trustStore=/work/cacerts"]
```

Das erzeugte Docker-Image startet bei Tests auf einer EC2-Instanz `m5.large` im JVM-Mode in 1,3 Sekunden, das native in 0,012 Sekunden. Für andere Workloads kann das Ergebnis natürlich gänzlich anders sein. Mit Hilfe von Docker-Push kann das Image in eine beliebige Docker-Registry (wie beispielsweise DockerHub, Quay, Amazon-ECR etc.) verschoben werden.

Nun, da das Docker-Image erstellt und in das bevorzugte Repository gepusht wurde, kann die Anwendung auf Amazon-EKS mit AWS-Fargate ausgeführt werden. Für diesen Fall bietet Quarkus eine spezielle Kubernetes-Extension. Die Quarkus-Community hat über 90 Erweiterungen[15] entwickelt, die zusätzliche Optimierungen und Integrationen des Frameworks bieten, einschließlich der Möglichkeit, die Dateien für die Bereitstellung von Kubernetes-Manifests[16] zu generieren. Die Verwendung dieser Erweiterung erzeugt eine initiale Deployment-Datei unter dem Ordner `target/kubernetes`, die weiter verbessert werden kann.

Für unsere Demoanwendung soll erreicht werden, dass diese von der DynamoDB-Users-Tabelle lesen sowie in diese schreiben kann. Dazu wird die Funktion IAM-Rollen für Service-Accounts für EKS[17] verwendet, indem eine AWS-IAM-Rolle einem neuen Kubernetes-`ServiceAccount` zugeordnet wird. Um den Einsatz wiederholbar und versionierbar zu machen, wird die Verwendung eines Code-Werkzeugs zur Definition der zugrunde liegenden Ressourcen verwendet. In diesem Fall wurde das AWS-Cloud-Development-Kit (CDK)[18] verwendet, damit die Ressourcen in einer sicheren, wiederholbaren Weise durch AWS-CloudFormation[19] bereitgestellt werden können **(Listing 7)**.

(Listing 7)

```
sa = cluster.addServiceAccount('quarkus-service-account', { ...
});

 deployment = {
  apiVersion: "apps/v1",
  kind: "Deployment",
  metadata: { name: "quarkus-demo-deploy" },
  spec: {
    template: {
      spec: {
        serviceAccountName: sa.serviceAccountName,
        containers: [ { ... } ]
  ... }}}};

table = new dynamodb.Table(this, 'Users', { ... });

table.grantReadWriteData(sa.role)
```

Um die Anwendung gegenüber DynamoDB zu authentifizieren, benötigt diese außerdem einen Security-Token (temporäre Zugriffsschlüssel) für die entsprechende IAM-Rolle. Diese bekommt unsere Anwendung über den AWS-Security-Token-Service (STS)[20]. Dafür wurde die entsprechende Abhängigkeit in die `pom.xml` inkludiert **(Listing 8)**.

(Listing 8)

```
<dependency>
      <groupId>software.amazon.awssdk</groupId>
      <artifactId>sts</artifactId>
</dependency>
```

Da das STS-SDK mit Reflections arbeitet, um den Security-Token zur Laufzeit zu holen, muss es für das Kompilieren des nativen Images mit GraalVM entsprechend konfiguriert werden. Dies erfolgt über die Reflection-Konfigurations-Datei `src/main/resources/reflect-config.json` **(Listing 9)**.

(Listing 9)

```
[
    {
        "name" :"software.amazon.awssdk.services.sts.internal.
StsWebIdentityCredentialsProviderFactory",
        "allDeclaredConstructors": true,
        "allPublicConstructors": true,
        "allDeclaredMethods": true,
        "allPublicMethods": true
    }
]
```

Nun muss das native Maven-Profil um zusätzliche Build-Argumente erweitert werden, um die Reflection-Konfiguration mit anzugeben **(Listing 10)**.

(Listing 10)

```
<properties>
    <quarkus.package.type>native</quarkus.package.type>
    <quarkus.native.additional-build-args>
        -H:ReflectionConfigurationFiles=reflect-config.json
    </quarkus.native.additional-build-args>
</properties>
```

Der neu bereitgestellte `UserService Dienst` ist von ausserhalb des Amazon-EKS-Clusters nicht erreichbar, da dieser nur an einen lokalen 8080-Port bindet. Um die Anwendung vom Internet aus erreichbar zu machen, wurde eine zusätzliche Kubernetes-Application-Load-Balancer-Ingress-Controller[21]-Ressource erstellt. Dadurch wird für unsere Ingress-Resource automatisch ein AWS-Application-Load-Balancer mit einem öffentlichen DNS und einem Listener auf HTTP erstellt, der die eingehende Anfrage an den Kubernetes-Service und entsprechend an einen der Pods weiterleitet **(Listing 11)**.

(Listing 11)

```
service = {
  apiVersion: "v1",
  kind: "Service",
  metadata: { name: "quarkus-demo-svc" },
  spec: {
    type: "NodePort",
    ports: [ { port: 8080, targetPort: 8080 } ],
    selector: appLabel
  }
};

 ingress = {
  apiVersion: "extensions/v1beta1",
  kind: "Ingress",
  metadata: {
  name: "quarkus-demo-ingress",
  annotations: {
    "kubernetes.io/ingress.class": "alb",
    "alb.ingress.kubernetes.io/scheme": "internet-facing",
    "alb.ingress.kubernetes.io/healthcheck-path": "/health"
  },
  labels: appLabel
  },
  spec: { rules: [{ http: { paths: [{
        path: "/*",
        backend: {
          serviceName: "quarkus-demo-svc",
          servicePort: 8080
        }}]}}]
  }
};
```

Mit den Kommandos in **(Listing 12)** kann die Infrastruktur und die Anwendung in AWS in der `region us-east-1` bereitgestellt werden.

(Listing 12)

```
$ npm install -g aws-cdk
$ cd fargate/eks_cdk
$ npm install
$ cdk bootstrap  // configures the CDK in the region/acount
$ cdk diff       // Lists the resources to be deployed
$ cdk deploy     // Deploys the CloudFormation template
```

Das EKS-CDK-Konstrukt definiert in unserem Stack einen CloudFormation-Output, der den auszuführenden Befehl zur Konfiguration des kubectl-Clients enthält **(Listing 13)**.

(Listing 13 - Outputs)

```
EksCdkJsStack.quarkusdemoclusterConfigCommand = aws eks update-kubeconfig
    --name quarkus-demo-cluster --region us-east-1 --role-arn
arn:aws:iam::*:role/*
```

Der CloudFormation-Stack Output `LoadBalancerDNS` ist der DNS-Eintrag des Load-Balancers **(Listing 14)**.

(Listing 14 - Outputs)

```
EksCdkStack.LoadBalancerDNS = aaa-quarkus.us-east-1.elb.amazo-
naws.com
```

Nachdem die Infrastruktur und der DNS-Endpunkt eingerichtet sind, kann die Anwendung wie in (Listing 15) beschrieben getestet werden.

(Listing 15)

```
$ curl -v http://<lb-url>/health // aus HealthResource.java
Resource
$ curl -v -d '{"userName":"hmueller", "firstName":"Hans", "last-
Name":"Mueller", "age":"35"}' -H "Content-Type: application/json"
-X POST http://<lb-url>/users
$ curl -v http://<lb-url>/users/<user-id>
$ curl -v http://<lb-url>/users
$ curl -v -X DELETE http://<lb-url>/users/<user-id>
```

Abschließende Überlegungen zu Quarkus-Projekten

Ein paar Hinweise, die beim Erstellen nativer Quarkus-Anwendungen auf AWS zu beachten sind:

- Bei der Kompilierung, Konfiguration und Bereitstellung der Anwendung sollte eine CI/CD-Pipeline genutzt werden, um den Prozess wiederholbar, transparent, schnell und frei von menschlichen Fehlern zu machen.
- Wenn bereits eine Spring-Anwendung vorliegt und diese migriert werden soll, bietet Quarkus bereits einige Features, die zu Spring-Bibliotheken wie Spring-DI[22], Spring-Web[23] oder Spring-Data-JPA[24] kompatibel sind.
- Es existieren einige Einschränkungen[25] bei der Verwendung des Native-Images, wie zum Beispiel eingeschränkte Verwendung der Reflections und dynamisches Laden von Klassen.
- Alle Quarkus-Erweiterungen ermöglichen die Abstimmung des generierten Manifests mit Hilfe der `application.properties` Datei, wie im All-Config-Guide[26] beschrieben.
- Um Containern, die auf EKS mit Fargate aufgestellt werden, eine zusätzliche Berechtigung zu erteilen, ist es notwendig die AWS-Funktion IAM-Roles-for-Service-Account zu verwenden. Auf diese Weise fügt der Kubernetes-Scheduler die Web-Token-Datei zur Laufzeit in den Pod ein. Die Anwendung sollte dann die Rolle mit `assume-role-with-web-identity` übernehmen.

Bereinigen

Um Kosten zu sparen können alle diese Ressourcen mit einem einzigen Befehl bereinigt werden, sobald die Tests fertig sind (Listing 16).

(Listing 16)

```
$ cdk destroy
```

Fazit:

Wie in der Demo-Anwendung gezeigt wurde, ist es einfach in die Softwareentwicklung mit Quarkus einzusteigen. Innerhalb weniger Minuten kann ein grundlegender REST-Dienst aufgebaut, kompiliert und auf Amazon-EKS mit AWS-Fargate ausgeführt werden. Darüber hinaus wurde CDK verwendet, um die benötigte Infrastruktur als Code zu definieren und das Native-Image auf Fargate bereitzustellen. Und mit der Möglichkeit, einem bestimmten Pod in Amazon-EKS unter Verwendung von ServiceAccounts feingranulare Privilegien zu gewähren, konnte auf andere AWS-Dienste wie Amazon-DynamoDB auf sichere Weise zugegriffen werden. Die Anwendung kann an dieser Stelle einfach erweitert, bzw. an entsprechende Usecases angepasst werden. Darüber hinaus lässt sich Amazon-EKS sehr gut mit anderen AWS-Diensten wie Amazon-CloudWatch zur Überwachung oder AWS-X-Ray zum Logging und Tracing integrieren. Die Verwendung des AWS-Managed-Service reduziert den Aufwand für Installation, Betrieb und Wartung der Infrastruktur. Entwickler können sich dadurch auf den Code und die Bereitstellung interessanter und relevanter Anwendungen konzentrieren. Quarkus bietet mit den im Artikel beschriebenen Möglichkeiten ein simples und solides Fundament für die Entwicklung von Cloud-Native-Anwendungen. Genau wie bei anderen Plattformen und Frameworks gilt aber auch hier: Ein gutes Anwendungsdesign ist die Basis für eine performante Applikation.

Quellen:

```
 1 http://bit.ly/ecs-e1
 2 http://bit.ly/lambda-2
 3 http://bit.ly/fargate-3
 4 http://bit.ly/microprofile-4
 5 https://quarkus.io/
 6 http://bit.ly/hotspot-6
 7 https://www.graalvm.org/
 8 http://bit.ly/quarkus-demo-8
 9 http://bit.ly/bootstrap-9
10 http://bit.ly/dynamodb-a10
11 http://bit.ly/dynamodb-q11
12 http://bit.ly/sdk2-12
13 http://bit.ly/substratevm-13
14 http://bit.ly/variante-14
15 http://bit.ly/extentions-15
16 http://bit.ly/kubernetes-16
17 http://bit.ly/iam-17
18 http://bit.ly/cdk-18
19 http://bit.ly/formation-19
20 http://bit.ly/sts-a20
21 http://bit.ly/alb-21
22 http://bit.ly/spring-di22
23 http://bit.ly/spring-w23
24 http://bit.ly/jpa-24
25 http://bit.ly/limitations-25
26 http://bit.ly/guide-26
```

#JAVAPRO #CloudNative #SpringBoot #GraalVM

Native-Images mit Spring-Boot und GraalVM

GraalVM-Native-Image erzeugt aus Java-Code native und daher sehr schnelle Binärdateien. Dazu müssen jedoch alle Abhängigkeiten der Anwendung bereits zur Übersetzungszeit bekannt sein. Besonderheiten ergeben sich unter anderem durch die Verwendung von Mechanismen wie Reflection, Dynamic-Proxys, Classpath-Scanning oder der CGLIB, die in Spring-Boot eingesetzt werden. Dieser Artikel zeigt, wie sich mit Spring-Boot Native-Images erzeugen lassen.

Beim Starten einer Java-Anwendung wird der gesamte Byte-Code zunächst ausschließlich interpretiert. Im Laufe der Zeit werden häufig genutzte Codepassagen, sogenannte Hotspots nach und nach in Maschinencode übersetzt. Die Übersetzung erfolgt durch den Just-in-Time-Compiler (JIT). Dieses Vorgehen gab der bis heute populärsten Java-Virtual-Machine den Namen Hotspot-VM.

Autor:

Thorsten Maier arbeitet als Principal-Consultant bei Trivadis. Er teilt seit mehr als 10 Jahren unter anderem als Spring-Trainer, Speaker, Berater und Autor sein Wissen und seine Erfahrungen mit anderen. Ihn bewegt die Frage, wie sich modernste Technologien in gewachsene Umgebungen einbinden lassen und wann man besser auf Bestehendes zurückgreifen sollte.

https://www.trivadis.com/
https://blog.oio.de/
@ThorstenMaier
https://www.linkedin.com/in/maier-thorsten
Thorsten.Maier@trivadis.com

Der größte Vorteil dieser Technik liegt neben der Plattformunabhängigkeit des Codes darin, dass der JIT-Compiler dynamische Optimierungen zur Laufzeit vornehmen kann. Dies ist mit einem Ahead-of-Time-Compiler (AOT) wie beispielsweise GCC nicht möglich. Dieser Compiler muss alle Optimierungen durchführen, bevor die Anwendung gestartet wird und ist daher nicht in der Lage, auf die speziellen Laufzeiteigenschaften des Codes einzugehen. Eine der wichtigsten Optimierungen moderner Compiler sind Inline-Methoden. Diese gehen allerdings auf Kosten des Speicherbedarfs und sollten daher nur an häufig genutzten Codestellen zum Einsatz kommen. Ohne Ausführung des Codes sind solche Hotspots viel schwieriger zu lokalisieren.

Auch der Nachteil eines JIT-Compilers liegt auf der Hand. Der Übersetzungsprozess zur Laufzeit benötigt Ressourcen in Form von Hauptspeicher und CPU-Zyklen. Diese stehen der eigentlichen Anwendung daher nicht zur Verfügung. Dieser Nachteil wird jedoch durch die höhere Qualität des Codes, insbesondere bei langlaufenden Server-Anwendungen, ausgeglichen. Was aber ist mit Anwendungen, die nur eine sehr kurze Lebensdauer haben oder bei denen es auf besonders schnelle Startzeiten ankommt? In einem solchen Umfeld wäre es vorteilhaft, den gesamten Code vor dem Start in Maschinencode zu übersetzen, um den Aufwand zur Laufzeit zu sparen. An dieser Stelle kommt GraalVM ins Spiel.

GraalVM

Mit GraalVM stellt Oracle eine noch junge Hochleistungslaufzeitumgebung zur Verfügung, die signifikante Verbesserungen der Anwendungsleistung und -effizienz[1] verspricht. Neben Java unterstützt GraalVM auch JavaScript, LLVM-basierte Sprachen wie C und C++ sowie andere dynamische Sprachen wie Ruby, Python oder R.

Beim Einsatz mit Java unterscheidet sich GraalVM kaum von einem klassischen Oracle-JDK. Lediglich der alte, in C++ geschriebene Just-in-Time-Compiler C2 wird durch den neuen GraalVM-Compiler ersetzt. Dieser neue JIT-Compiler wurde in Java geschrieben. Das bedeutet aber auch, dass der Code des GraalVM-Compilers wie jeder andere Java-Code in der JVM ausgeführt wird. Der Code wird zunächst interpretiert und im Laufe der Zeit werden Hotspots zu Maschinencode kompiliert. Erst nach einer längeren Laufzeit entsteht optimierter und damit schnellerer Code. Da sich das Laufzeitverhalten des JIT-Compilers zwischen verschiedenen Ausführungen nicht wesentlich unterscheidet, wird der resultierende Code trotz des vergleichsweisen hohen Aufwands immer sehr ähnlich sein. Um diese Laufzeitnachteile zu eliminieren, kann der GraalVM-Compiler als vorkompilierte Bibliothek verwendet werden.

GraalVM-Native-Image und Spring-Boot

Mit GraalVM-Native-Image kann Java-Code bereits vor der Ausführung in ein eigenständig ausführbares Artefakt, ein so genanntes Native-Image, kompiliert werden. Dies geschieht unter der Annahme einer geschlossenen Welt, in der alle notwendigen Quellcodeteile zum Zeitpunkt der Übersetzung bereits bekannt und verfügbar sind. Diese Technologie wird verwendet, um den GraalVM-Compiler in die vorkompilierte Bibliothek namens libgraal zu übersetzen. GraalVM-Native-Image ist jedoch nicht auf den neuen JIT beschränkt. Stattdessen ist dieser Ansatz in der Lage, beliebigen Java-Code vor der Ausführung in Maschinencode zu übersetzen. In Microservice-Umgebungen verspricht GraalVM eine bis zu 50 Mal schnellere Startzeit und bis zu fünf Mal weniger Speicherbedarf[2]. Laufzeit und Speicherbedarf sind zugleich die zentralen Kostentreiber in modernen Cloud-Umgebungen. Die sich daraus potenziell ergebenden Kostenvorteile sprechen für die Überlegung, in Zukunft nicht nur Microservices, sondern alle Java-Anwendungen als Native-Image bereitzustellen, um von diesen Vorteilen profitieren zu können.

Spring gehört nach wie vor zu den beliebtesten Frameworks im Java-Umfeld. Vor allem wenn es um die Implementierung einer Microservices-Architektur geht, gab es lange Zeit wenig ernstzunehmende Alternativen zu Spring bzw. dem darauf aufsetzenden Spring-Boot. Bei Microservices wiederum kommen die Vorteile der Native-Images besonders gut zum Tragen. Es erscheint daher naheliegend, diese beide Techniken zu kombinieren.

Spring ist im Kern flexibel und erweiterbar. Daher tut sich das Framework zunächst recht schwer mit dem Closed-World-Ansatz von GraalVM-Native-Image. Dieser Artikel zeigt im Folgenden die wichtigsten Schritte die notwendig sind, um aus einer Spring-Boot-Anwendung ein Native-Image zu erzeugen und erläutert die notwendigen Konfigurationen.

Native-Image Limitierungen

Nach der Installation von GraalVM kann Native-Image mit dem GraalVM-Updater (gu) hinzugefügt werden. Dies geschieht mit dem Befehl `gu install native-image`. Danach kann ein vorhandenes Java-JAR mit `native-image -jar app.jar` in ein Native-Image übersetzt werden. So weit, so einfach.

Der Closed-World-Ansatz von GraalVM-Native-Image bringt jedoch einige Einschränkungen mit sich.
Infolgedessen liefert der Native-Image-Befehl allzu oft beim ersten Versuch nicht das gewünschte binäre Artefakt. Alle Einschränkungen beziehen sich darauf, dass zum Zeitpunkt der Übersetzung alle später auszuführenden Codeteile bereits bekannt sein müssen. Am Beispiel der Java-Reflection-API sollte schnell klar werden, dass dies bei Java nicht immer der Fall ist. Daneben gibt es weitere Funktionen, die entweder gar nicht, nur in eingeschränkter Form oder nur mit zusätzlicher Konfiguration in einem Native-Image verwendet werden können[3]:

- Dynamic-Proxy
- Dynamic-Class-Loading
- JCA (Java-Cryptography-Architecture)
- JNI (Java-Native-Interface)
- Invokedynamic-Bytecode and Method-Handles
- Serialization
- Security-Manager
- Class-Initializers
- Finalizers
- Threads
- Unsafe-Memory-Access

Reflection und Component-Scan

Spring-Boot-Anwendungen nutzen standardmäßig einen Component-Scan, um auf dem Klassenpfad nach Komponenten zu suchen. Das Konzept des Klassenpfades existiert allerdings in einer Binärdatei nicht mehr. Somit kann auch ein normaler Component-Scan nicht mehr eingesetzt werden. Statt des dynamischen Ansatzes muss auf eine statische Konfiguration zurückgegriffen werden. Bereits seit Spring 5.0 gibt es mit dem Spring-Context-Indexer einen Annotation-Processor mit dessen Hilfe in der Build-Phase eine statische Liste aller Komponenten erzeugen kann. Dieser Index wird unter `META-INF/spring.components` abgelegt. Ist er vorhanden, werden die darin enthaltenen Einträge anstelle des dynamischen

Klassenpfad-Scans eingesetzt. Um den Index zu generieren, wird eine zusätzliche Abhängigkeit in die Build-Datei hinzugefügt. (Listing 1) zeigt dies am Beispiel Maven[4].

(Listing 1)

```
<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-indexer</artifactId>
  </dependency>
</dependencies>
```

Die Komponenten werden beim Starten der Spring-Anwendung per Reflection instanziiert. (Listing 2) zeigt anhand eines einfachen Beispiels, warum die Verwendung von Reflection mit dem Closed-World-Ansatz von GraalVM-Native-Image ebenfalls eine zusätzliche Konfiguration notwendig macht[5]. In der ersten Zeile der Main-Methode wird eine Klasse anhand ihres Namens geladen. Die Besonderheit hierbei ist, dass der Name der Klasse von außen als Befehlszeilenargument an die Java-Anwendung übergeben wird. Was in der virtuellen Maschine ohne Probleme möglich ist, führt beim Kompilieren mit GraalVM-Native-Image zwangsläufig zu Problemen. Welche Klasse soll in das binäre Artefakt gepackt werden? Der AOT-Ansatz stößt hier schlichtweg an seine Grenzen.

(Listing 2)

```
public class HelloReflection {
   public static void main(String[] args) throws Exception {
      Class<?> class1 = Class.forName(args[0]);
      Object instance = class1.newInstance();
      Method method = class1.getMethod("doIt");
      method.invoke(instance);
   }
}
```

Die Klassen und Methoden, die über Reflection eingebunden und aufgerufen werden sollen, können über eine zusätzliche Konfiguration bekannt gemacht werden. (Listing 3) zeigt eine Beispielkonfiguration für die Klasse aus (Listing 2). Die JSON-Konfiguration ist unter `META-INF/native-image/reflect-config.json` gespeichert und wird daher während des Übersetzungsprozesses automatisch eingebunden. Die Konfiguration enthält den voll qualifizierten Klassennamen und die Methoden, die in das binäre Artefakt aufgenommen werden sollen. Neben der Methode `doIt` muss auch der parameterlose Konstruktor (`<init>`) explizit aufgeführt werden.

(Listing 3)

```
[{
  "name":"com.trivadis.MyComponent",
  "methods":[
    {"name":"<init>","parameterTypes":[] },
    {"name":"doIt","parameterTypes":[] }
  ]
}]
```

Um diesen Konfigurationsprozess zu automatisieren, gibt es von GraalVM einen Tracing-Agent[6], der das Verhalten einer Java-Anwendung während der Ausführung aufzeichnet und bei der Erstellung der Konfigurationsdateien hilft. Der Tracing-Agent hilft unter anderem beim Umgang mit Funktionen wie der Java-Reflection. Die Einbindung des Agenten beim Starten der Java-Anwendung ist in (Listing 4) dargestellt. Der Agent erstellt die notwendige `reflect-config.json` und legt sie im angegebenen Verzeichnis `META-INF/native-image` ab. Bereits bei kleinen Spring-Anwendungen werden dabei erstaunlich große Konfigurationsdateien erzeugt.

(Listing 4)

```
$JAVA_HOME/bin/java -agentlib:native-image-agent=config-out-
put-dir= META-INF/native-image
```

Durch die Kombination aus Spring-Indexer und der expliziten Reflection-Konfiguration sind bereits wesentliche Teile der Dynamik einer Spring-Anwendung in einer statischen Konfiguration festgeschrieben worden. Auf dem Weg hin zu einem Binärartefakt sind allerdings noch weitere Hürden zu überwinden.

Dynamic-Proxys und weitere Einschränkungen

Spring nutzt an unterschiedlichen Stellen dynamisch zur Laufzeit generierte Proxys. Dabei werden zwei Implementierungsvarianten unterstützt. Die Generierung kann über CGLIB oder den JDK-eigenen Mechanismus Dynamic-Proxy erfolgen[7]. In Spring-Boot wird standardmäßig CGLIB als Implementierung verwendet[8]. Native-Images unterstützen allerdings lediglich die Verwendung von Dynamic-Proxy[9]. (Listing 5) zeigt die notwendigen Konfigurationsanpassung, um Spring-Boot auf die Verwendung dieser Variante zu beschränken. Der aufgeführte Parameter `proxyBeanMethods` wurde mit Spring-Boot 2.2 eingeführt.

(Listing 5)

```
@SpringBootApplication(proxyBeanMethods=false)
```

Weiterhin benötigt die Verwendung von Dynamic-Proxys im Zusammenspiel mit Native-Image ebenfalls wieder eine zusätzliche Konfiguration, damit die notwendigen Proxys bereits zum Übersetzungszeitpunkt bekannt sind. Der im letzten Abschnitt beschriebene GraalVM-Agent generiert neben der Reflection-Konfiguration auch eine `proxy-config.json`. Diese wird benötigt, damit bereits zum Kompilierungszeitpunkt bekannt ist,

welche Dynamic-Proxys erstellt werden müssen. Ähnlich hilfreich ist der Agent auch beim Umgang mit Java-Native-Interface (JNI) und Klassenpfadressourcen. Analog zu den beiden gezeigten Funktionen Reflection und Dynamic-Proxy werden hierfür die Konfigurationsdateien `jni-config.json`[10] und `resource-config.json`[11] benötigt. Beide Konfigurationen werden ebenfalls automatisch durch den Agent erzeugt.

GraalVM-Feature

Wie gezeigt ist der vorgestellte GraalVM-Agent sehr hilfreich, wenn es darum geht, dynamische Anteile der Spring-Boot-Anwendung durch eine statische Konfiguration zu ersetzen. Zwei Nachteile dieses Ansatzes sollten jedoch nicht verschwiegen werden. Erstens muss die Anwendung einmal gestartet werden, um die Konfigurationsdateien zu erstellen. Zweitens können die statischen Konfigurationen nur die Einträge enthalten, die unmittelbar beim Start der Anwendung geladen werden. Spring ermöglicht darüber hinaus allerdings auch zu einem späteren Zeitpunkt den Zugriff auf dynamische Ressourcen. Der Zugriff auf diese zusätzlichen Ressourcen muss manuell in den generierten Konfigurationsdateien bekannt gemacht werden, damit das Binärartefakt fehlerfrei funktionieren kann.

Letztlich behandelt der gezeigte GraalVM-Agent eine Spring-Boot-Anwendung wie jede andere Java-Anwendung und versucht den Zugriff auf dynamische Ressourcen zu erkennen. Die Native-Image-Generierung greift anschließend lediglich auf die zuvor durch den Agent erstellten statischen Konfigurationsdateien zu. Da dieser Ansatz keine Kenntnis über speziellen Spring-Mechanismen hat, existieren zwangsläufig Einschränkungen. Diese Einschränkungen können zumindest teilweise durch die Erweiterung der Analysephase von `native-image` umgangen werden. Hierzu wird die Analysephase über eine Erweiterungsschnittstelle um zusätzliches Wissen zum Beispiel über die internen Abläufe eines Frameworks wie Spring erweitert. Hierzu ist die Implementierung eines sogenannten Features[12] für GraalVM-Native-Image notwendig. Ein solches Feature für die direkte Unterstützung von Spring befindet sich gerade in Entwicklung und kann bereits getestet werden[13]. Das aktuelle Release Spring 5.3 bietet ebenfalls Verbesserungen für die Unterstützung dieses Features.

Einfach ausgedrückt fügt das Feature zusätzliche Informationen zu den typischen Konfigurationselementen einer Spring-Boot-Anwendung wie z. B. den Annotationen `@Component` und `@Configuration` hinzu, so dass diese Zusatzinformationen zur Erstellung des binären Artefakts verwendet werden können. GraalVM-Native-Image bekommt durch das Feature einen Einblick in die Auswirkungen der Konfigurationselemente einer Spring-Anwendung. Es ist und bleibt jedoch eine Analyse, die auf statischem Code basiert, der zu diesem Zweck nicht ausgeführt wird.

Das Feature wird, wie in **(Listing 6)** gezeigt, als zusätzliche Abhängigkeit in der Maven `pom.xml` bekannt gemacht. Alle weiteren notwendigen Konfigurationen zur Nutzung des Features sind in der Dokumentation zu finden[14]. Dazu gehört unter anderem auch die Angabe der Startklasse der Anwendung, so dass dem Analyseprozess von Native-Image ein definierter Einstiegspunkt für die Analyse zur Verfügung steht. Diese Konfiguration ist im Übrigen auch für die Nutzung ohne das Feature notwendig, damit die Generierung des binären Artefakts erfolgreich durchgeführt werden kann.

(Listing 6)

```
<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-graalvm-native</artifactId>
    <version>0.7.1</version>
  </dependency>
</dependencies>
```

Hybrid - Das Beste aus beiden Welten

Der anfangs betrachtete GraalVM-Agent hat keine Kenntnis über die interne Funktionsweise der Spring-Anwendung. Er hat jedoch den Vorteil, dass er die tatsächlichen Abläufe der Java-Anwendung zur Laufzeit nachverfolgen kann. Das zuletzt betrachtete Spring-GraalVM-Native-Image-Feature kennt hingegen zwar die internen Abläufe von Spring, muss dafür allerdings auf die Ausführung des Codes verzichten. Weder der Agent noch das Feature werden fehlerfreie Ergebnisse liefern. In aller Regel müssen manuelle Änderungen an den Konfigurationen vorgenommen werden. Je nach Anwendungsfall kann jedoch der eine Ansatz die Schwächen des jeweils anderen ausgleichen. Daher kann es manchmal sinnvoll sein, beide Ansätze zu kombinieren. Dabei wird die Anwendung zunächst mit dem Agent gestartet, um die statischen Konfigurationsdateien zu erzeugen. Anschließend wird der Erzeugungsprozess des Binärartefakts mit dem Feature erweitert. Das Feature ist in der Lage, die statischen Konfigurationen zu erkennen und diese gegebenenfalls um weitere Informationen anzureichern.

Wie bereits erwähnt, wird sich die Unterstützung von Native-Images für Spring-Boot-Anwendungen in den kommenden Monaten deutlich verbessern. Sowohl der GraalVM-Agent als auch das Spring-GraalVM-Native-Image-Feature werden derzeit intensiv weiterentwickelt. Ebenfalls fließen Konfigurationsvereinfachungen in den Kern des Spring-Frameworks, um den Analyseprozess künftig zu erleichtern. Mit der Kombination aus Agent und Feature sowie etwas Geduld für das Finetuning der in aller Regel noch notwendigen weiteren Konfigurationseinstellungen, lassen sich bereits heute Native-Images aus Spring-Boot-Anwendungen generieren. Für das Finetuning lassen sich dem `native-image` Befehl zahlreiche Befehlszeilenargumente

übergeben. (Listing 7) zeigt die im offiziellen Spring-Github-Repository[15] vorgeschlagene Kombination an Parametern für die Erstellung einer typischen Spring-Web-Anwendung mit integriertem Tomcat. Darin ist unter anderem zu sehen, dass die Verwendung von HTTP manuell aktiviert werden muss.

(Listing 7)

```
-H:+PrintAOTCompilation -H:+PrintHeapHistogram -H:EnableURLProto-
cols=http -Dspring.native.mode=hybrid -Dspring.backgroundpreini-
tializer.ignore=true -Dspring.spel.ignore=true -Dspring.native.
remove-yaml-support=true -Dspring.xml.ignore=true -Dspring.
native.remove-jmx-support=true
```

Fazit und Ausblick:

Mit GraalVM-Native-Image lassen sich beeindruckende Verbesserungen bei Startzeit und Speicherverbrauch einer Spring-Anwendung erzielen. Die konkreten Verbesserungen unterscheiden sich je nach Anwendungsfall und Umgebung. In der Regel sind sie jedoch signifikant genug, damit sich für viele ein Blick auf diese noch recht junge Technologie lohnt. Im Moment ist der Weg zu einer nativ kompilierten Spring-Boot-Anwendung noch steinig, wie die im letzten Abschnitt aufgeführte Parameteraufliistung zeigt. Mit zukünftigen Versionen von GraalVM und Spring wird dieses Problem jedoch verschwinden. Beide Seiten bewegen sich derzeit aufeinander zu, so dass Spring-Anwendungen schon bald deutlich einfacher von den Vorteilen der Native-Images profitieren können. Die in der Java-Welt ungewohnte Plattformabhängigkeit und deutlich längere Build-Zeiten aufgrund von Analysen während des Kompilierungsprozesses sind in vielen Fällen sicherlich vertretbar.

Quellen:

1 https://www.graalvm.org/
2 http://bit.ly/medium-m2
3 http://bit.ly/3l8abBG
4 http://bit.ly/30yxPhs
5 http://bit.ly/30CBZVG
6 http://bit.ly/3ckGY2y
7 http://bit.ly/2OQ0kUY
8 http://bit.ly/3vgsiKz
9 http://bit.ly/3tc95rp
10 http://bit.ly/2Ook6Y1
11 http://bit.ly/2Na7gfh
12 http://bit.ly/3taIxXE
13 http://bit.ly/3vszJhU
14 http://bit.ly/3vsA7No
15 http://bit.ly/3l7mqyC

#JAVAPRO #SpringBoot #Security #Authentication

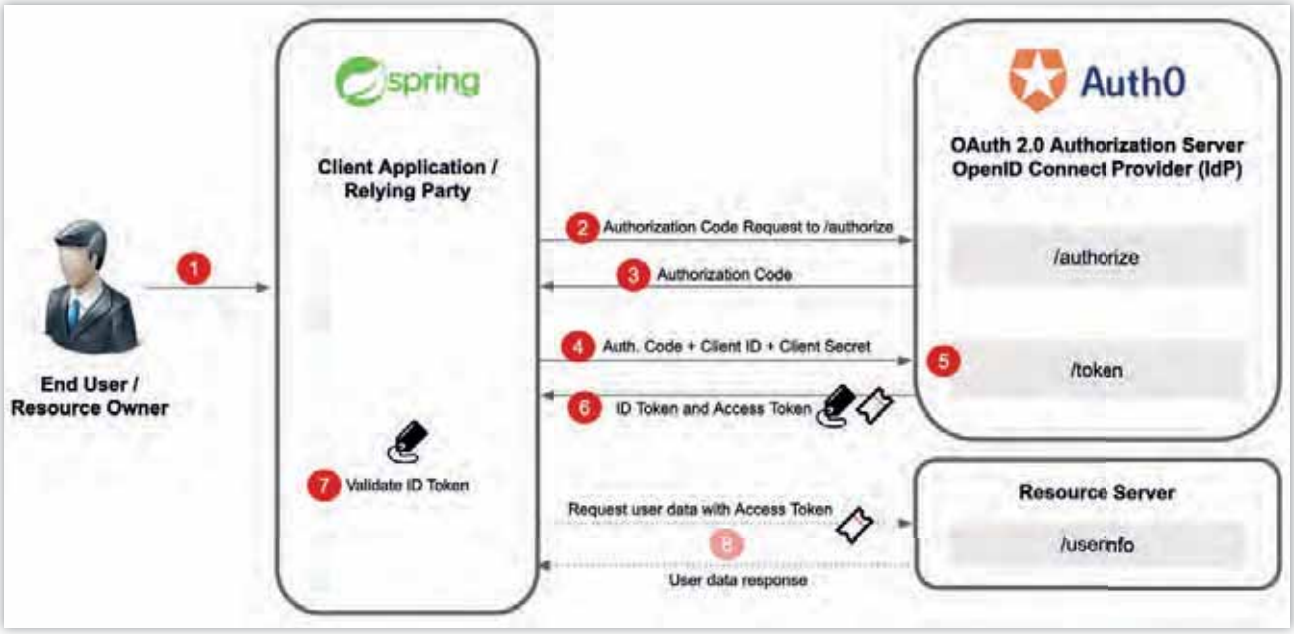
Authentifizierung mit OpenID-Connect

Der Artikel zeigt, wie Benutzern einer Spring-MVC-Applikation ermöglicht werden kann, sich mittels OpenID-Connect über einen eigenen Identity-Provider (IdP) zu authentifizieren, ohne selbst einen Autorisierungsserver implementieren zu müssen. Im zweiten Schritt wird gezeigt, wie Entwickler mühelos weitere Social-IDPs, wie zum Beispiel Facebook, Twitter, LinkedIn etc. ohne viel Mehraufwand föderiert anbinden können.

Spring-Security ist nicht nur ein mächtiges Framework zur Authentifizierung und Autorisierung von Benutzern, sondern auch der De-Facto Standard zur Absicherung von Java-Applikationen. In Verbindung mit einer Identity-as-a-Service-Plattform (IDaaS) lassen sich sicherheitsrelevante Features in kurzer Zeit und mit wenig eigenem Code abbilden. Mit Spring 5.1 wurde der OAuth 2.0 und OpenID-Connect (OIDC)-Unterstützung rundum erneuert bzw. reimplementiert. Während das OAuth 2.0 Protokoll zur Autorisierung bzw. Access-Delegation auf Ressourcen wie APIs oder sonstige Backends dient, handelt es sich bei OpenID-Connect um einen Identity-Layer, der auf OAuth 2.0 aufsetzt und rein der Authentifizierung des Benutzers dient.

Autor:

Mathias Conradt ist Senior-Solutions-Engineer bei Auth0 und beschäftigt sich vorwiegend mit Identity & Access Management Lösungen rund um OAuth und OpenID Connect.



OpenID-Connect mit Authorization-Code-Grant[1] (Abb. 1)

OpenID-Connect und der OAuth 2.0 Authorization-Code-Grant

Konzeptionell sieht der Ablauf einer OpenID-Connect basierten Authentifizierung gemäß dem OAuth 2.0 Authorization-Code-Grant wie in (Abb 1.) aus.

1. Der Benutzer ruft die Spring-Applikation im Browser auf.
2. Die Spring-Applikation bzw. Spring-Security leitet den Benutzer aufgrund der Sicherheitskonfiguration zum Autorisierungs-Server/IdP (`/authorize` Endpunkt) weiter, in diesem Fall `Auth0`. Sofern keine aktive Session beim `IdP` besteht, wird dem Benutzer eine Login-Seite und/oder ein Consent-Dialog angezeigt. Der Benutzer authentifiziert sich mittels einer der konfigurierten Login-Optionen, zum Beispiel `Benutzername / Passwort`. Er sieht daraufhin einen Consent-Dialog der die Berechtigungen, zum Beispiel `Auslesen des Benutzerprofils` auflistet, die der Autorisierungs-Server/IdP `Auth0` der anfragenden Spring-Applikation gewähren wird.
3. Der Autorisierungs-Server/IdP `Auth0` leitet den User mit einem `Authorization-Code` zurück an die Spring-Applikation.
4. Spring-Security sendet den Code an den Autorisierungsserver/IdP (`/oauth/token` Endpunkt) zusammen mit `Client ID` und `Client Secret`.
5. Der Autorisierungs-Server/IdP `Auth0` verifiziert den `Code`, `Client ID` und `Client Secret`.
6. Der Autorisierungs-Server /IdP `Auth0` gibt `ID Token` und `Access Token` und optional `Refresh Token` an die Spring-Applikation zurück.
7. Die Spring-Applikation validiert und dekodiert den `ID Token` und kann dessen Inhalt (Payload) zur Anzeige des Benutzerprofils nutzen.
8. Optional kann die Spring-Applikation den `Access-Token` nutzen, um weitere Benutzerinformationen vom gemäß OpenID-Connect standardisierten UserInfo-Endpunkt des Autorisierungsserver/IdP abzurufen. In diesem Beispiel reicht der `ID Token` allein aus.

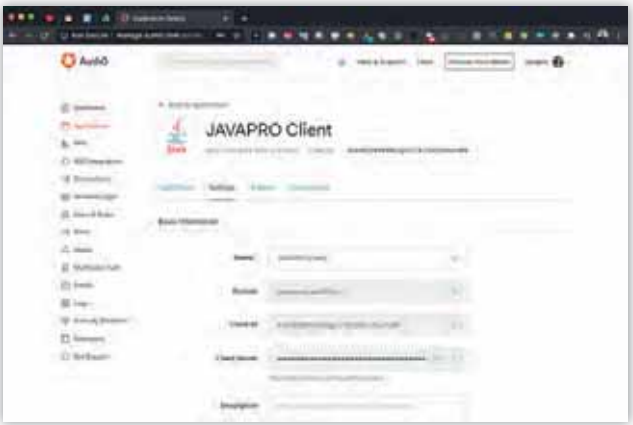
Theoretisch kann für eine reine OpenID-Connect-Authentifizierung, bei der lediglich der ID-Token und kein Access-Token verwendet wird, auch der Implicit-Grant mit Form-Post-Response-Mode[2] in Erwägung gezogen werden, allerdings wird dieser seitens Spring Security 5.1 derzeit nicht unterstützt. Dies würde die generelle Komplexität verringern und das Verwalten von Client-Secrets unnötig machen.

Die Implementierung erfolgt in zwei Schritten. Zuerst wird der Identity-Provider seitens Auth0 konfiguriert, danach erfolgt die Integration in ein Spring-Boot- bzw. Spring-Security-Projekt.

IdP-Konfiguration seitens Auth0

Auf `https.//Auth0.com` wird zunächst ein kostenfreier Account sowie Mandanten erstellt. Die geplante Java-Applikation im Auth0-Dashboard wird unter dem Punkt `Applications` registriert. Als Technologie-Stack wird Regular-Web-App und Java-Spring-MVC gewählt. Nach erfolgreichem Anlegen der Client-Applikation und Wechsel zum Settings-Tab erhält man eine Client-ID sowie ein Client-Secret. Dieses wird später für die Konfiguration des Auth0-IdP in der Spring-Applikation benötigt. Es werden noch in den Settings die erlaubten Callback-URLs konfiguriert `http://localhost:3000/login/oauth2/code/Auth0` sowie die erlaubten Logout-URLs auf `http://localhost:3000`. Optional kann ein Icon vergeben werden. In diesem Beispiel wird das Java-Logo verwendet (Abb. 2).

Nun kann mit der eigentlichen App-Entwicklung begonnen werden.



Client-Application-Settings im Auth0-Dashboard (Abb. 2)

Spring-Boot Projektinitialisierung

Es wird ein Spring-Boot-Projekt mit folgenden, wesentlichen für OpenID-Connect relevanten Abhängigkeiten erstellt (Listing 1). Das gesamte Beispielprojekt ist auf GitHub[3] verfügbar.

(Listing 1)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>Spring Security-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>Spring Security-oauth2-jose</artifactId>
</dependency>
```

Um die Anwendung für nicht authentifizierten Zugriffen zu schützen, wird zunächst eine Konfigurationsklasse erstellt, die von `WebSecuritConfigurerAdapter` ableitet. In ihr wird die `configure` Methode überschrieben und die Sicherheitsregeln für eingehende HTTP-Requests definiert.
Im Beispiel soll für die gesamte Applikation eine Authentifizierung vorausgesetzt sein. Zusätzlich soll OAuth2-Login aktiviert werden. Schließlich wird noch ein eigener `LogoutHandler` definiert, der den Default-Handler überschreibt (Listing 2).

(Listing 2)

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepo-
sitory;
    private final LogoutController logoutController;

    public SecurityConfig(LogoutController logoutController) {
        this.logoutController = logoutController;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2Login()
            .and()
            .logout()
            .addLogoutHandler(logoutController);
        }
    }
}
```

Der `oauth2Login` holt sich die Konfiguration des zu verwendenden Identity-Provider aus der Applikationskonfiguration `application.yml`. Zunächst wird der `Auth0` als Identity-Provider definiert (Listing 3).

(Listing 3)

```
[...]
spring:
  security:
    oauth2:
      client:
        registration:
          Auth0:
            client-name: JAVAPRO Client
            client-id: {clientID}
            client-secret: {clientSecret}
            scope:
            - openid
            - profile
            - email
            - offline_access
            authorization-grant-type: authorization_code
            logout-uri: https://javapro.eu.Auth0.com/v2/
logout?client_id=[...]
        provider:
          Auth0:
            issuer-uri: https://javapro.eu.Auth0.com/
            user-name-attribute: name
```

Die in (Listing 3) gezeigte Konfiguration unterteilt sich in zwei Teile: es wird zum einen der Identity-Provider definiert (Listing 4).

(Listing 4)

```
provider:
  Auth0:
    issuer-uri: https://javapro.eu.Auth0.com/
```

Über die Issuer-URI ist Spring im Stande, sich über das als OpenID-Connect-Discovery[4] standardisierte Verfahren die notwendigen OAuth 2.0 Endpunkte wie Authorization-Endpoint und Token-Endpoint zur Kommunikation zwischen Client-Applikation (Spring) und Autorisierungs-Server (`Auth0`) selbständig herauszusuchen. Der zweite Teil der `application.yml` **(Listing 3)** registriert die zugehörige Client-Applikation **(Listing 5)**.

(Listing 5)

```
registration:
  google:
    client-name: JAVAPRO Client
    client-id: {clientID}
    client-secret: {clientSecret}
    scope:
      - openid
      - profile
      - email
      - offline_access
    authorization-grant-type: authorization_code
    logout-uri: [...]
```

Der `client-name` kann hierbei beliebig gewählt werden. Es bietet sich an für eine bessere Übersicht den gleichen Namen zu verwenden wie in `Auth0` bei der Registrierung der Applikation. Als `scope` wird `openid profile email offline_access` angefragt. Hiermit wird angezeigt, dass es sich um einen OpenID-Connect-Request handelt, der Profildaten des Users zurückerhalten und neben Access- und ID-Token auch ein Refresh-Token erhalten möchte (`offline_access`). Da es sich um eine reguläre Web-Applikation mit Backend handelt, wird der Authorization-Code-Grant von OAuth 2.0 verwendet, zumal Spring-Security aktuell auch nur diesen unterstützt: `authorization-grant-type: authorization_code`
Als nächstes wird ein Controller benötigt. Der `HomeController`, der authentifizierte Requests an `http://localhost:3000/` entgegennimmt und das Benutzerprofil - extrahierte Claims des ID-Tokens - darstellt. Dazu nimmt eine Methode `index` die GET-Anfragen am Root-Pfad entgegen. In ihr steht sowohl ein `Model` Objekt zur Verfügung, als auch der erhaltene OAuth2AuthenticationToken, welcher eine `OAuth2-Authentifizierung` repräsentiert und die Token-Information beinhaltet **(Listing 6)**.

(Listing 6)

```
@Controller
public class HomeController {
    public HomeController(OAuth2AuthorizedClientService authorizedClientService) {
        this.authorizedClientService = authorizedClientService;
    }
    @RequestMapping("/")
    public String index(Model model, OAuth2AuthenticationToken
authentication) {
        model.addAttribute("userName", authentication.getName());
        model.addAttribute("clientName", authorizedClient.getCli-
entRegistration().getClientName());
        model.addAttribute("userinfo", authentication.getPrin-
cipal().getAttributes());
        return "index";
    }
}
```

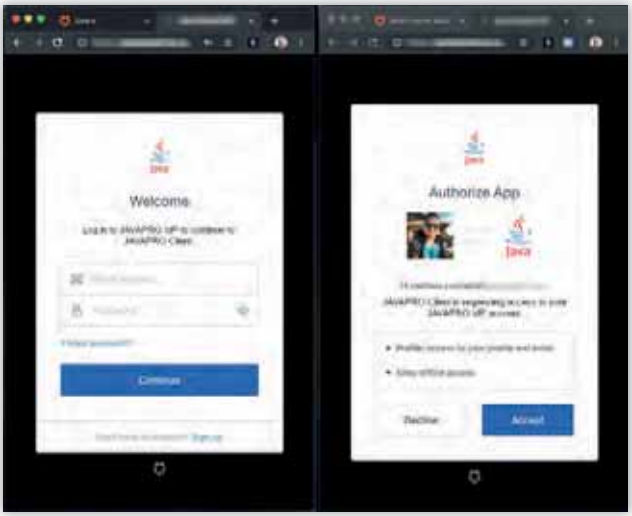
Die Claims aus dem ID-Token, die letztlich die Benutzer-Attribute und somit Benutzerprofil darstellen, werden mittels `authentication.getPrincipal().getAttributes()` erhalten und können diese direkt an die View (`index.html`) unter dem gewählten Attribut-Namen `userInfo` übergeben. Erfreulicherweise übernimmt Spring-Security OAuth- bzw. OpenID-Support automatisch den OAuth 2.0 gemäßen Tausch von Autorisierungscode gegen einen Access- und ID-Token. Auch wird das Dekodieren und Validieren des ID-Tokens, welches vom Identity-Provider als Base64Url-dekodierter und signierter JSON-Web-Token (JWT)[5] zurückkommt, automatisch vorgenommen und nimmt dem Entwickler bereits einiges an manueller Implementierungsarbeit ab. Die zugehörige View, welche Thymeleaf als Template-Library verwendet und an die das Benutzerprofil über das `Model` übergeben wird, ist in **(Listing 7)** beschrieben. Sie zeigt neben dem Namen der Client-Applikation das Avatar-Foto des Benutzers sowie das Profil an. Das Benutzerfoto wird seitens `Auth0` automatisch vom Gravatar-Dienst auf Basis der bei der Registrierung verwendeten Benutzer-Mail-Adresse aufgesucht, sofern vorhanden.

(Listing 7)

```
[...]
<h1>OAuth 2.0 Login with Spring Security</h1>
<div>
  You are successfully logged in <span style="font-weight:bold"
th:text="${userName}"></span>
  via the OAuth 2.0 Client <span style="font-weight:bold"
th:text="${clientName}"></span>
</div>
<div><img th:src="${userinfo.get('picture')}" height="100"
width="100"/></div>
<div>
  User info from ID token:
  <ul>
    <li th:each="attribute : ${userinfo}">
      <span style="font-weight:bold" th:text="${attribute.
key}"/>: <span th:text="${attribute.value}"></span>
    </li>
    </ul>
</div>
[...]
```

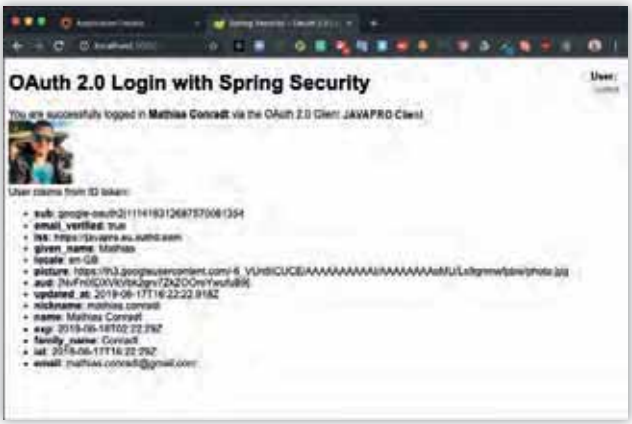
Die Client-Applikation wird mittels `mvn spring-boot:run` gestartet und die URL `http://localhost:3000` wird im Browser

aufgerufen. Da noch keine Authentifizierung vorliegt, wird man sofort zum konfigurierten Identity-Provider `Auth0` weitergeleitet und aufgefordert sich einzuloggen bzw. einen Account zu erstellen, sofern dieser noch nicht besteht. Außerdem erscheint ein Consent-Screen, mit der die Einwilligung eingeholt wird, dass die Spring-Anwendung Zugriff auf das Profil bei `Auth0` erhalten darf **(Abb. 3)**.



Login- und Registrierungsseite des Identity Providers. (Abb. 3)

Nach erfolgreicher Authentifizierung wird das Benutzerprofil, konkret die sogenannten Claims des ID-Tokens, dargestellt **(Abb. 4)**.



Benutzerprofil. (Abb. 4)

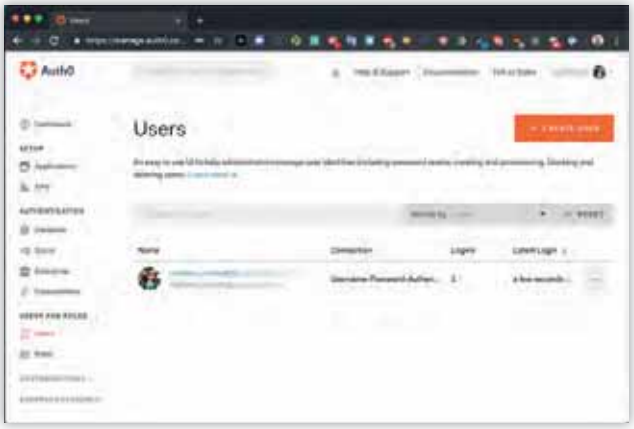
Entwickler haben in Bezug auf den Logout die Möglichkeit zu entscheiden, ob beim Klicken des Logout-Buttons lediglich die Session der Anwendung beendet wird oder der Benutzer auch am IdP ausgeloggt werden soll. Soll der Benutzer auch am IdP ausgeloggt werden, so muss ein eigener `LogoutController` erstellt werden, der in der `SecurityConfig` registriert wird. Der `LogoutController` ruft den entsprechenden Logout-Endpunkt des IdP auf, in diesem Beispiel `Auth0`. Der IdP-seitige Logout-Endpunkt wird aus der `application.yml` ausgelesen und ist als `logout-uri` definiert **(Listing 8)**.

(Listing 8)

```
https://javapro.eu.Auth0.com/v2/logout?client_id={clientID}&re-
turnTo=http://localhost:3000.

@Controller
public class LogoutController extends SecurityContextLogout-
Handler {

    private final ClientRegistrationRepository clientRegistrati-
onRepository;
    private Logger logger = LoggerFactory.getLogger(LogoutCont-
roller.class);

    @Value("${spring.security.oauth2.client.registration.Auth0.
logout-uri}")
    private String logoutUrl;

    public LogoutController(ClientRegistrationRepository client-
RegistrationRepository) {
        this.clientRegistrationRepository = clientRegistrationRe-
pository;
    }

    @Override
    public void logout(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Authentication authen-
tication) {
        super.logout(httpServletRequest, httpServletResponse,
authentication);
        try {
            httpServletResponse.sendRedirect(logoutUrl);
        } catch (IOException ioe) {
            logger.error("Error logging out.", ioe);
        }
    }
}
```

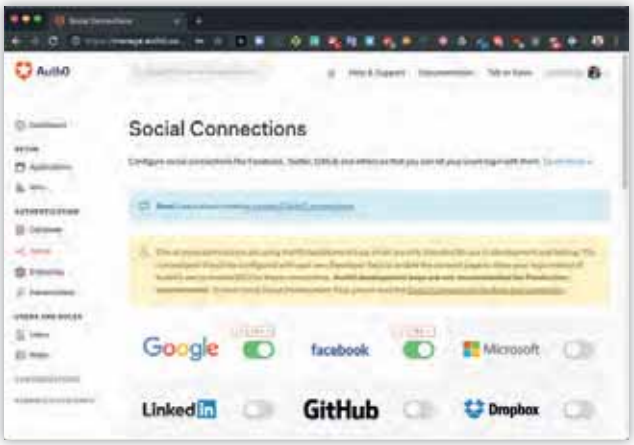
Entscheidet sich der Entwickler dazu, beim Logout lediglich die aktuelle Session der Anwendung zu beenden, nicht jedoch die des IdP, kann auf den `LogoutController` sowie die Methoden `logout` und `addLogoutHandler` in der `SecurityConfig` Konfigurationsklasse verzichtet werden.

Auth0-Dashboard, Social-Login und MFA-Konfiguration



Benutzerübersicht im Auth0-Dashboard (Abb. 5)

Im Auth0 Dashboard kann nun unter dem Punkt Users & Roles auch der soeben registrierte Benutzer gefunden werden (Abb. 5). Um jetzt neben der Auth0 Datenbank weitere föderierte Social-Identity-Provider wie Google und Facebook zu integrieren (Abb. 6) sowie beispielsweise Multi-Factor-Authentication (MFA) zu aktivieren (Abb. 7), kann dies rein über die Konfiguration innerhalb von Auth0 vorgenommen werden, ohne dabei die Java-Applikation anfassen zu müssen.



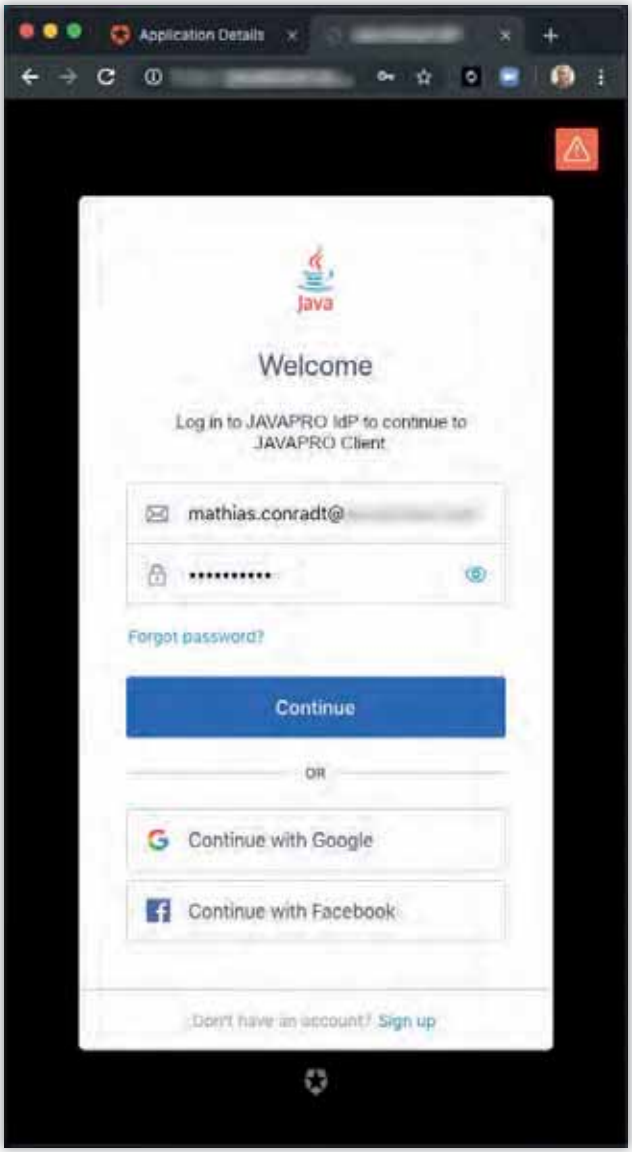
Aktivierung weiterer föderierter Social-Identity-Provider (Abb. 6)

Aktivierung von MFA mittels Google-Authenticator (Abb. 7)

Loggt man sich nun aus der Web-Applikation unter http://localhost:3000 aus und versucht sich erneut einzuloggen, stellt man fest, dass nun sowohl zwei weitere Buttons für Facebook und Google als Möglichkeit zur Authentifizierung zur Verfügung stehen (Abb. 8). Auch die Aufforderung nach dem Login erscheint, die Multi-Factor-Authenfication (MFA) über das Scannen eines QR-Codes zu initialisieren.

Fazit:

Ein OAuth 2.0 bzw. OpenID-Connect-basierter Login ist in Spring Security 5.1 mit sehr wenig Zeilen Code und minimaler Konfiguration möglich. Das Framework nimmt dem Entwickler einiges



Login-Widget mit aktivierten Social-Connections (Abb. 8)

an Implementierungsarbeit ab, was die Details des OAuth2-Protokolls angeht. Statt weitere Social-Identity-Provider direkt in der Java-Applikation zu konfigurieren, werden diese stattdessen mittels Auth0 als Broker föderiert integriert. Dies hat den Vorteil, dass die Konfiguration an zentraler Stelle vorgenommen werden kann. Bei der Entwicklung weiterer Client-Applikationen skaliert dies hervorragend und vermeidet doppelten Aufwand. Dies gilt auch für die Bereitstellung von MFA. Als netter Seiteneffekt kann so ebenfalls ein Single-Sign-On über alle Applikationen hinweg bereitgestellt werden.

Quellen:

```
1 http://bit.ly/grant-a1
2 http://bit.ly/oauth-m2
3 http://bit.ly/github-3
4 http://bit.ly/openid-4
5 https://jwt.io/
```

#JAVAPRO #CoreJava #Architecture #API

Zukunftssicheres API-Design

Viele Ideen sind auf dem Papier hervorragend. Oft fehlt aber das Wissen, wie man brillante Konzepte in den eigenen Alltag einbauen kann. Dieser Workshop soll die Lücke zwischen Theorie und Praxis schließen und zeigt, mit welchen Maßnahmen man langfristig zu einer stabile API gelangt.

Bei der Entwicklung kommerzieller Software ist vielen Beteiligten oft nicht klar, dass die Anwendung für lange Zeit in Benutzung sein wird. Da sich unsere Welt stetig im Wandel befindet, ist es leicht abzusehen, dass im Laufe der Jahre große und kleine Änderungen der Anwendung ausstehen werden. Zu einer richtigen Herausforderung wird das Vorhaben, wenn die zu erweiternde Anwendung nicht für sich isoliert ist, sondern mit anderen Systemkomponenten kommuniziert. Denn das bedeutet für die Konsumenten der eigenen Anwendung in den meisten Fällen, dass sie ebenfalls angepasst werden müssen. Ein einzelner Stein wird so schnell zu einer Lawine. Mit einem guten Lawinenschutz lässt sich die Situation gut beherrschen. Das gelingt aber nur, wenn man berücksichtigt, dass die im Nachfolgenden beschriebenen Maßnahmen ausschließlich für eine Prävention gedacht sind. Hat sich die Gewalt aber erst einmal entfesselt, kann ihr kaum noch etwas entgegengesetzt werden. Deshalb wird zuerst geklärt, was eine API ausmacht.

Verhandlungssache

Ein Softwareprojekt besteht aus verschieden Komponenten, denen spezialisierte Aufgaben zuteilwerden. Die wichtigsten sind Quelltext, Konfiguration und Persistenz. Der Artikel befasst sich hauptsächlich mit dem Bereich Quelltext. Es sind keine Neuigkeiten, dass stets gegen Interfaces implementiert werden soll. Diese Grundlage bekommt man bereits in der Einführung der Objektorientierten Programmierung vermittelt. Bei der täglichen Arbeit sieht man aber sehr oft, dass so manchem Entwickler die Bedeutung der Forderung gegen Interfaces zu entwickeln nicht

Autor:

Marco Schulz (Twitter: @ElmarDott) studierte an der HS Merseburg Diplominformatik. Sein persönlicher Schwerpunkt liegt in Softwarearchitekturen, der Automatisierung des Softwareentwicklungsprozess und dem Softwarekonfigurationsmanagement. Seit über fünfzehn Jahren realisiert er in internationalen Projekten für nahmenhafte Unternehmen auf unterschiedlichen Plattformen umfangreiche Webapplikationen. Er ist freier Consultant, Trainer und Autor verschiedener Fachartikel. Sein persönlicher Blog lautet https://enRebaja.wordpress.com, sie erreichen ihn unter: marco.schulz@outlook.com

immer ganz klar ist, obwohl bei der Verwendung der Java-Standard-API dies die übliche Praxis ist. Das klassische Beispiel ist in **(Listing 1)** beschrieben.

(Listing 1)

```
List<String> collection = new ArrayList<>();
```

Diese kurze Zeile nutzt das Interface `List`, welches als eine ArrayList implementiert wurde. Es ist zu sehen, dass kein Anhängsel in Form eines `I` die Schnittstelle kennzeichnet. Auch die zugehörige Implementierung trägt kein `Impl` im Namen. Das ist auch gut so. Besonders bei der Implementierungsklasse könnten verschiedene Lösungen erwünscht sein. Dann ist es wichtig, diese gut zu kennzeichnen und leicht durch den Namen unterscheidbar zu halten. `ListImpl` und `ListImpl2` sind verständlicherweise nicht so gut wie `ArrayList` und `LinkedList` auseinander zu halten. Damit konnte auch schon der erste Punkt einer stringenten und sprechenden Namenskonvention geklärt werden.

Die Programmteile werden im nächsten Schritt behandelt. Es soll nach Möglichkeit nicht für Konsumenten der Anwendung nach außen gegeben werden, dass es sich um Hilfsklassen handelt. Ein Teil der Lösung liegt in der Struktur, wie die Packages zu organisieren sind. Ein sehr praktikabler Weg ist:

- `my.package.path.business`: enthält sämtliche Interfaces
- `my.package.path.application`: enthält die Implementierungen der Interfaces
- `my.package.path.application.hepler`: enthält interne Hilfsklassen

Bereits über diese simple Architektur signalisiert man anderen Programmierern, dass es keine gute Idee ist, Klassen aus dem Package `helper` zu benutzen. Ab Java 9 gibt es noch eine weitreichendere Restriktion, das Verwenden interner Hilfsklassen zu unterbinden. Die Modularisierung, welche mit dem Projekt Jigsaw[1] in Java 9 Einzug genommen hat, erlaubt es im Module-Descriptor `module-info.java` Packages nach außen hin zu verstecken.

Separatisten und ihre Flucht vor der Masse

Schaut man sich die meisten Spezifikationen etwas genauer an, stellt man fest, dass viele Schnittstellen in eigene Bibliotheken ausgelagert wurden. Technologisch betrachte würde das auf das vorherige Beispiel bezogen bedeuten, dass das Package `business`, welches die Interfaces enthält, in eine eigene Bibliothek ausgelagert wird. Die Trennung von API und der zugehörigen Implementierung erlaubt es grundsätzlich Implementierungen leichter gegeneinander auszutauschen. Es gestattet außerdem einem Auftraggeber einen stärkeren Einfluss auf die Umsetzung seines Projektes bei seinem Vertragspartner auszuüben, indem der Hersteller die API durch den Auftraggeber vorgefertigt bekommt. Damit es auch tatsächlich so klappt, wie es ursprünglich gedacht war, sind aber ein paar Regeln zu beachten.

Beispiel 1: JDBC. Es ist bekannt, dass die Java-Database-Connectivity ein Standard ist, um an eine Applikation verschiedenste Datenbanksysteme anbinden zu können. Sieht man von den Problemen bei der Nutzung von nativem SQL einmal ab, können JDBC-Treiber von MySQL nicht ohne weiteres durch PostgreSQL oder Oracle ersetzt werden. Schließlich weicht jeder Hersteller bei seiner Implementierung vom Standard mehr oder weniger stark ab und stellt darüber hinaus weitere exklusive Funktionalität des eigenen Produktes über den Treiber mit zur Verfügung. Entscheidet man sich im eigenen Projekt diese Zusatz-Features massiv nutzen zu wollen, ist es mit der leichten Austauschbarkeit vorüber.

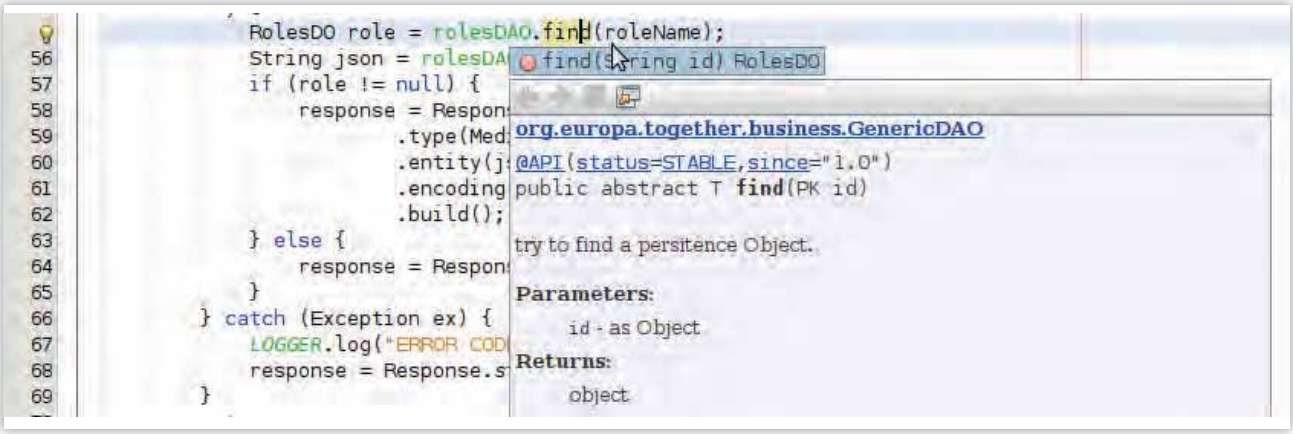
Beispiel 2: XML. Hier hat man gleich die Wahl zwischen mehreren Standards. Es ist klar, dass die APIs von SAX, DOM und StAX nicht zueinander kompatibel sind. Will man beispielsweise wegen einer besseren Performance von DOM zum ereignisbasierten SAX wechseln, kann das unter Umständen umfangreiche Codeänderungen nach sich ziehen.

Beispiel 3: PDF. Zu guter Letzt noch ein Szenario von einem Standard, der keinen Standard hat. Das Portable-Document-Format selbst ist zwar ein Standard wie Dokumentdateien aufgebaut werden, aber bei der Implementierung von Programmbibliotheken, die für die eigene Anwendung genutzt werden können, köchelt jeder Hersteller sein eigenes Süppchen.
Die Beispiele zeigen die üblichen Probleme auf, die im täglichen Projektgeschäft zu meistern sind. Eine kleine Regel bewirkt schon großes: Fremdbibliotheken nur dann nutzen, wenn es wirklich notwendig ist. Schließlich birgt jede verwendete Abhängigkeit auch ein potenzielles Sicherheitsrisiko. Es ist auch nicht notwendig eine Bibliothek mit der Größe von einigen MB einzubinden, um damit drei Zeilen Code einzusparen, die benötigt werden, um einen String auf leer und Null zu prüfen.

Musterknaben

Wenn man sich für eine externe Bibliothek entschieden hat, so ist es immer vorteilhaft sich anfänglich die Arbeit zu machen und die Funktionalität über eine eigene Klasse zu kapseln, welche man dann exzessiv nutzen kann. Dies ist im Projekt `TP-CORE` auf GitHub[2] an mehreren Stellen zu sehen. Der Logger kapselt die Funktionalität von SLF4J und Logback. Im Vergleich zu den PdfRenderer ist die Signatur der Methoden von den verwendeten Logging-Bibliotheken unabhängig und kann somit leichter über eine zentrale Stelle ausgetauscht werden.

Um externe Bibliotheken in der eigenen Applikation möglichst zu kapseln, stehen die Entwurfsmuster Wrapper, Fassade und



Suggestion in Netbeans mit @API Annotation in der JavaDoc (Abb. 1)

Proxy zur Verfügung. Wrapper, auch Adaptor-Muster genannt, gehört in die Gruppe der Strukturmuster. Der Wrapper koppelt eine Schnittstelle zu einer anderen, die nicht kompatibel sind. Fassade ist ebenfalls ein Strukturmuster, das mehrere Schnittstellen zu einer vereinfachten Schnittstelle bündelt. Proxy, auch Stellvertreter genannt, gehört ebenfalls in die Kategorie der Strukturmuster. Proxies sind eine Verallgemeinerung einer komplexen Schnittstelle. Es kann als Komplementär der Fassade verstanden werden, die mehrere Schnittstellen zu einer einzigen zusammenführt.

Sicher ist es wichtig in der Theorie, diese unterschiedlichen Szenarien zu trennen, um sie korrekt beschreiben zu können. In der Praxis ist es aber unkritisch, wenn zur Kapselung externer Funktionalität Mischformen der hier kurz vorgestellten Entwurfsmuster entstehen. Für alle diejenigen, die sich intensiver mit Design-Pattern auseinander setzen möchten, dem sei das Buch „Entwurfsmuster von Kopf bis Fuß“[3] ans Herz gelegt.

Klassentreffen

Ein weiterer Schritt auf dem Weg zu einer stabilen API ist eine ausführliche Dokumentation. Basierend auf den bisher besprochenen Schnittstellen gibt es eine kleine Bibliothek, mit der verschiedene Methoden, basierend der API-Version, annotiert werden können. Neben Informationen zum Status und der Version können für Klassen über das Attribute `consumers` die primären Implementierungen aufgeführt werden. Um API-Gaurdian dem eigenen Projekt zuzufügen sind nur wenige Zeilen der `POM` hinzuzufügen **(Listing 2)**.

Die Auszeichnung der Methoden und Klassen ist ebenso leicht. Die Annotation `@API` hat die Attribute: `status`, `since` und `consumers`. Für Status sind die folgenden Werte möglich:

- `DEPRECATED`: Veraltet, sollte nicht weiterverwendet werden.
- `EXPERIMENTAL`: Kennzeichnet neue Funktionen, auf die der Hersteller gerne Feedback erhalten würde. Mit Vorsicht verwenden, da hier stets Änderungen erfolgen können.
- `INTERNAL`: Nur zur internen Verwendung, kann ohne Vorwarnung entfallen.
- `STABLE`: Rückwärts kompatibles Feature, das für die bestehende Major-Version unverändert bleibt.
- `MAINTAINED`: Sichert die Rückwärtsstabilität auch für das künftige Major-Release zu.

Nachdem nun sämtliche Interfaces mit diesen nützlichen Metainformationen angereichert wurden, stellt sich die Frage, wo der Mehrwert zu finden ist. Dazu hilft ein Blick auf **(Abb. 1)**, welche den Arbeitsalltag demonstriert.

Für Service-basierte RESTful APIs gibt es Swagger[4]. Auch hier wird der Ansatz verfolgt, aus Annotationen eine API-Dokumentation zu erstellen. Swagger selbst scannt allerdings Java-Webservice-Annotationen, anstatt eigene einzuführen. Die Verwendung ist ebenfalls recht leicht umzusetzen. Man muss lediglich das Swagger-Maven-Plugin einbinden und in der Konfiguration die Packages angeben, in denen die Webservices residieren **(Listing 3)**. Anschließend wird bei jedem Build eine Beschreibung in Form einer JSON-Datei erstellt, aus der dann Swagger-UI eine ausführbare Dokumentation generiert. Swagger-UI selbst wiederum ist als Docker-Image auf DockerHub[5] verfügbar.

(Listing 2)

```
<dependency>
    <groupId>org.apiguardian</groupId>
    <artifactId>apiguardian-api</artifactId>
    <version>1.1.0</version>
</dependency>
```

(Listing 3)

```
<plugin>
    <groupId>io.swagger.core.v3</groupId>
    <artifactId>swagger-maven-plugin</artifactId>
    <configuration>
        <outputFileName>swagger</outputFileName>
        <outputFormat>JSON</outputFormat>
```

```
    <resourcePackages>
            <package>org.europa.together.service</package>
        </resourcePackages>
        <outputPath>${project.build.directory}</outputPath>
    </configuration>
</plugin>
```

Swagger-UI Dokumentation der TP-ACL RESTful API (Abb. 2)

Versionierung ist für APIs ein wichtiger Punkt. Unter Verwendung des Semantic-Versioning lässt sich bereits einiges von der Versionsnummer ablesen. In Bezug auf eine API ist das Major-Segment von Bedeutung. Diese erste Ziffer kennzeichnet API-Änderungen, die inkompatibel zueinander sind. Eine solche Inkompatibilität ist das Entfernen von Klassen oder Methoden. Aber auch das Ändern bestehender Singnaturen oder der Rückgabewert einer Methode erfordern bei Konsumenten im Rahmen einer Umstellung Anpassungen. Es ist immer eine gute Entscheidung, Arbeiten zu bündeln und eher selten zu veröffentlichen, die Inkompatibilitäten verursachen. Dies zeugt von Stabilität im Projekt. Auch für Web-APIs ist eine Versionierung angeraten. Diese lässt sich am besten über die URL erreichen, in die eine Versionsnummer eingefügt wird. Es hat sich bewährt, lediglich bei Inkompatibilitäten die Version hochzuzählen.

Beziehungsstress

Der große Vorteil eines RESTful-Service mit allem gut auszukommen, ist zugleich der größte Fluch. Denn das bedeutet, dass hier viel Sorgfalt walten muss, da viele Klienten versorgt werden. Da die Schnittstelle eine Ansammlung von URIs darstellt, liegt das Augenmerk auf den Implementierungsdetails. Dazu wird ein Beispiel aus dem ebenfalls auf GitHub verfügbaren Projekt `TP-ACL` genutzt **(Listing 4)**. Den Ausschnitt findet man im Try-Block der `fetchRole` Methode der `RoleService` Klasse.

(Listing 4)

```
RolesDO role = rolesDAO.find(roleName);
String json = rolesDAO.serializeAsJson(role);
if (role != null) {
    response = Response.status(Response.Status.OK)
            .type(MediaType.APPLICATION_JSON)
            .entity(json)
            .encoding("UTF-8")
            .build();
} else {
    response = Response.status(Response.Status.NOT_FOUND).
build();
}
```

Die GET-Anfrage liefert den Fehlercode `404` zurück, falls eine Rolle nicht gefunden wird. Bei der Implementierung der einzelnen Aktionen `GET`, `PUT`, `DELETE` etc. einer Ressource wie Rolle, genügt es nicht, einfach nur den sogenannten Happy-Path umzusetzen. Bereits während des Entwurfes sollte berücksichtigt werden, welche Stadien eine solche Aktion annehmen kann. Für die Implementierung eines Konsumenten (Client) ist es schon ein beachtlicher Unterschied, ob eine Anfrage gescheitert ist, die nicht mit 200 abgeschlossen werden kann, weil die Ressource nicht existiert (`404`) oder weil der Zugriff verweigert wurde (`403`).

Fazit.

Wenn wir von einer API sprechen, dann bedeutet das, dass es sich um eine Schnittstelle handelt, die von anderen Programmen genutzt werden kann. Der Wechsel einer Major-Version indiziert Konsumenten der API, dass Inkompatibilität zur vorherigen Version vorhanden ist und möglicherweise Anpassungen erforderlich sind. Dabei ist es völlig irrelevant, um welche API-Art es sich handelt, ob die Verwendung der Anwendung öffentlich oder eine Methode unternehmensintern ist. Die daraus resultierenden Konsequenzen sind identisch. Aus diesem Grund sollte man sich mit den nach außen sichtbaren Bereichen seiner Anwendung gewissenhaft auseinandersetzen. Arbeiten, welche zu einer API-Inkompatibilität führen, sollten durch das Release-Management gebündelt und möglichst nicht mehr als einmal pro Jahr veröffentlicht werden. Auch an dieser Stelle zeigt sich, wie wichtig regelmäßige Codeinspektionen für eine stringente Qualität sind.

Quellen:

1 http://bit.ly/jigsaw-1
2 http://bit.ly/tp-core-2
3 E. Freeman, 2015, „Entwurfsmuster von Kopf bis Fuß“ 2. Auflage, O'Reilly, ISBN: 9783955619862
4 https://swagger.io
5 http://bit.ly/swagger-ui-s5

#JAVAPRO #DomainDrivenDesign

Domain-Driven-Design Fundamentals

Domain-Driven-Design (DDD) ist eine von Eric Evans formulierte Designtechnik, die das Augenmerk der Softwareentwicklung auf den fachlichen Sinn lenkt und diesen als Zentrum oder dem Herz der Software anlegt.

Prof. David West[1] berichtete 2017 im Rahmen einer Keynote von seinem ersten Job als Programmierer bei einer Bank, als deren erster IT-Mitarbeiter er angestellt wurde. Er besprach sich regelmäßig mit seinen Bankkollegen über Aussehen und Leistungsumfang der Software. Es war ein Dialog, von dem beide Seiten profitierten. Auf der einen Seite die Experten mit dem Fachwissen wie eine Bank funktioniert, auf der anderen Seite der Softwareingenieur mit dem Know-how, wie Software funktioniert und was diese leisten kann.

Die späteren Jahre waren geprägt von einem Auseinanderdriften der verschiedenen Disziplinen. Fachabteilungen wurden konsolidiert, IT wurde ausgelagert und oft nur als notwendiges Übel betrachtet. Fachabteilungen, welche den Wunsch nach Digitalisierung hatten, konnten oft nur über Zwischenstellen wie dem Anforderungsmanagement und nach enormem bürokratischem Aufwand mit Entwicklern in Kontakt treten. Der eigentliche Sinn und Mehrwehrt der Software blieb dabei oft auf der Strecke. Auch heute noch erleben wir allzu oft, dass mehr über Prozesse und Zuständigkeiten als über den eigentlichen Sinn und Ziele einer Software gesprochen wird.

Ein anderer Aspekt dieser Spaltung war und ist bis heute die starke Neigung zu einer technischen Zentrierung in der Entwicklung. Ein gutes Beispiel stellt dafür die klassische 3-Tier-Architektur dar. In großen Monolithen bis zu Microservices findet sich die Struktur in Controllern, Services und Repository-Layern. Diese Architektur ist per se nicht schlecht, lenkt das Augenmerk aber schnell auf eine rein technische Sicht. Fachliche Belange werden leicht übersehen und resultieren dann in einer stark verteilten Fachlichkeit in den einzelnen Schichten ohne klare Trennung. Von Berechnungen im Frontend bis zu komplizierten Fachregeltabellen in relationalen Datenbanken finden sich überall Bruchstücke der Fachdomäne. Technische Aspekte sind zwar sauber getrennt, fachliche Logik aber wild im System verteilt. Das Resultat ist nicht selten ein Big-Ball-of-Mud[2].
Sich ändernde Anforderungen oder neue Erkenntnisse in den Unternehmensprozessen ziehen dabei einen eklatanten Aufwand für die Anpassung der bestehenden Software nach sich.

Erst in den letzten Jahren hat sich diese Tendenz gedreht und mit agilen Modellen, wie Scrum oder Kanban sind Elemente eingezogen, welche die Kommunikation weg von Dokumenten und wieder mehr zu den Menschen hin führt. Hier setzt nun Domain-Driven-Design an, um die durch die verbesserte Kommunikation erlangten Erkenntnisse in bessere Software einfließen zu lassen. DDD hilft hier mit Entwicklungs-Pattern und Vorgehensmodellen den fachlichen Sinn einer Software in den Kern der Diskussionen und der Entwicklung zu führen. Der Untertitel von Eric Evans Werk bringt es hier gut auf den Punkt: „Tackling Complexity in the Heart of Software“[3]

Autor:

Christian Schmidt, 34 Jahre aus Siegburg, ist seit 2010 bei der tarent Solutions GmbH in Bonn tätig. Hauptsächlich mit Java, Kotlin und Go entwickelt er moderne Microservices. Gerade die innere Architektur der Microservices liegt dabei mit Domain-Driven-Design mit CQRS oder hexagonaler Architektur im Fokus.
Außerhalb des Codes arbeitet er bereits seit einigen Jahren in modernen Microservicearchitekturen und kümmert sich um die Integration in CI/CD Systeme und Cloudmigrationen.

Die Domain

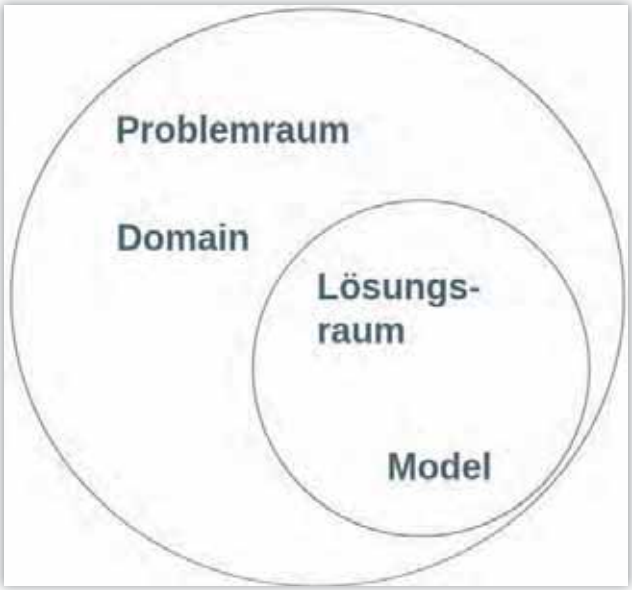
In DDD ist die Domäne der Dreh- und Angelpunkt einer zu erstellenden Software. Sie ist die fachliche Realität eines Unternehmens, einer Organisation oder einer Abteilung. Sie umfasst alle Dinge, Prozesse, Menschen, Begriffe und immateriellen Werte, die ein Geschäftsfeld oder Fachraum ausmacht und umschließt. Den Rahmen, den die Domain dabei bildet, kann das gesamte Unternehmen nebst Kunden und angeschlossenen Prozessen umfassen, aber auch lediglich einen Teilaspekt eines Teams innerhalb einer Abteilung. Die Größen und Abgrenzungen sind hier fließend.

Sehr oft ist bereits in Dokumenten und Prozessen das Wissen über die Domain beschrieben, aber gerade das immaterielle Wissen in den Köpfen von Mitarbeitern stellt meist einen großen Teil der Wirklichkeit dar[4]. Dieses Wissen zu extrahieren und zu destillieren ist eines der größten Aufgabenfelder denen man sich stellen kann und muss. Dabei ist es wichtig, dass sich nicht nur Businessanalysten und Product-Owner, sondern und gerade Entwickler mit dieser Domain auseinandersetzten. Denn nur was man versteht, kann man später auch modellieren und entwickeln. Ein interessantes Workshopkonzept ist hier das Event-Storming von Alberto Brandolino, welches eine große Brücke zwischen Fachwissen, Modellierung und späterer Softwarearchitektur schlägt und alle Beteiligten in einen Raum einlädt[5]. Innerhalb der Domain öffnet sich nun der Problemraum für die Entwicklung. Also das abgesteckte Feld oder der Business-Case, welches durch eine softwareseitige Umsetzung ergänzt, verbessert oder ersetzt werden soll. Als Beispiel einer Domain kann die Buchhaltungsabteilung eines Unternehmens betrachtet werden und der Problemraum bildet der Prozess der jährliches Bilanzerstellung innerhalb der Buchhaltung. Hier beginnt der Prozess[6].

Modell

Die Domain ist zu umfassend und komplex, um sie vollständig zu beschreiben. Denn die Erfassung der Domain zu 100% beschreibt das Unternehmen als Ganzes. Meist benötigt man nur eine kleine Teilmenge für eine ausreichend technische Abbildung. Dies bildet das Modell. Konkreter ist das Modell das eine hinreichende Abbildung der Domain darstellt **(Abb.1)**. Es beschreibt Organisationsteile so, wie sie später in einer Software umgesetzt werden. So wie die Domain einen Problemraum öffnet, bildet das Modell den nötigen Lösungsraum dafür ab. Das Modell ist dabei immer nur ein Ausschnitt der Wirklichkeit, niemals die Wirklichkeit selbst. Als Beispiel können hier verschiedene Weltkarten herhalten. Eine Karte bildet die Topografie ab, eine andere die wirtschaftlichen Aspekte, wiederum eine andere das Straßenlayout etc.

Und ebenso kann es auch verschiedene Modelle derselben Domain geben. Modelle können sich überschneiden, aber auch disjunkt zueinander stehen. Es kommt immer darauf an was einem nützt, um die Domain für eigene Zwecke hinreichend genau zu beschreiben. Man muss nicht die ganze Welt bereisen, um alles Notwendige für eine Aufzählung der Kontinente zu wissen.



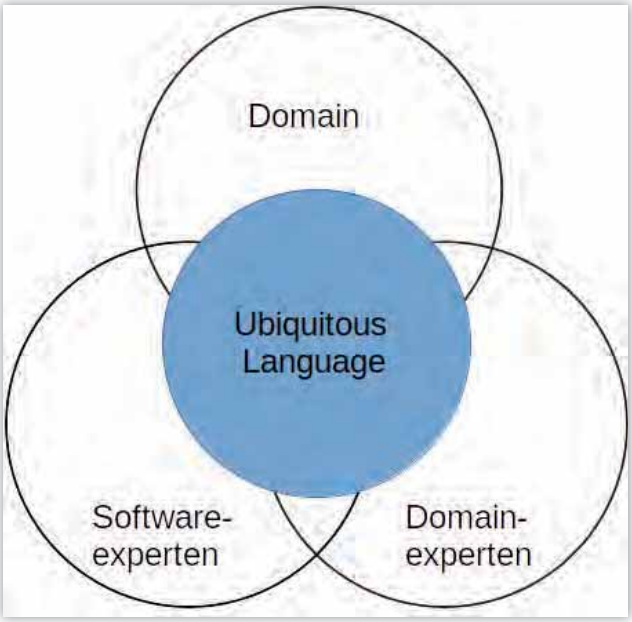
Lösungsraum innerhalb des Problemraums – Das Modell als Abbildung der Domain (Abb. 1)

Ein Modell zu erstellen ist immer ein iterativer und vor allem ein kreativer und kommunikativer Prozess. Modelle sind niemals starr. Ein Modell ist kein Ding, dass man einmal dokumentiert und für immer ablegt. Es ist vielmehr ein kontinuierlicher Prozess, um das Modell immer den entsprechenden Kenntnisstand der Domain und den gestellten Anforderungen anzupassen. Ein Modell ist auch nicht nur das UML-Diagramm, das im Wiki liegt. Ein Modell kann auch ein UML-Diagramm sein, das schnell an einem Whiteboard skizziert wurde. Ein Modell stellt aber viel mehr dar. Es ist der fachliche Code im Programm oder eine Tabellenstruktur in einer Datenbank. Es ist Text, Bild, Sprache und die Gesamtheit der mentalen Modelle aller beteiligter Personen[7].

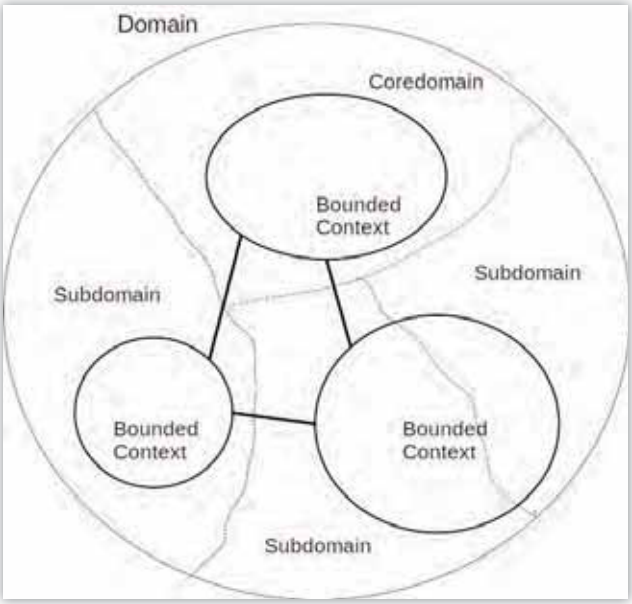
Ubiquitous-Language

Befasst man sich als Entwickler mit der Domain und will man sie verstehen, so fallen einem Fachbegriffe auf, die man nicht zuordnen kann, die synonym benutzt werden oder von denen man noch nie etwas gehört hat. Fachsprache ist immer kontextbezogen und nur in einem kleinen Kreis der Wissenden nutzbar. Auch Entwickler haben ihre eigene Fachsprache. Es ist nicht nur die Aufgabe des Entwicklers sich mit der Sprache der Fachdomäne zu befassen und diese zu verstehen, sondern auch Personen der abzubildenden Fachdomäne sollen und müssen sich mit der Fachsprache des Entwicklers befassen, wenn sie den weiteren Prozess begleiten und verstehen wollen. Der Prozess des gegenseitigen Kennen- und Verstehenlernens ist, wie so vieles in der Softwareentwicklung, ein kontinuierlicher Prozess, der nie abgeschlossen ist. Nach einer Zeit der Zusammenarbeit entwickelt sich in einem Team eine neue Sprache, die Begriffe aus beiden Welten und ggf. auch Neuschöpfungen oder Neuinterpretationen enthält. Sie ist ausreichend, um die Domain in den notwendigen Facetten zu beschreiben.

DDD nennt diese Sprache die Ubiquitous-Language oder frei übersetzt, die allgegenwärtige Sprache. Allgegenwärtig soll hier wörtlich genommen werden. Begriffe und Phrasen können und sollen sich in Gesprächen, Diagrammen, Dokumentationen, im Modell und auch im Programmcode wiederfinden. Allein dadurch ist gewährleistet, dass das Verständnis der Domain ebenfalls allgegenwärtig ist. Eine Fachperson kann ein UM-Diagramm wesentlich einfacher verstehen, wenn sie dort die ihr bekannten Begriffe findet. Ebenso können Entwickler in Gesprächen Fachprozesse leichter abbilden, wenn sie die Fachbegriffe verstehen. Im Alltag ist ein schnell zugängliches Glossar sehr praktisch um den Überblick zu behalten, bis sich Begriffe gefestigt haben.



Die Ubiquitous-Language im Dreieck der Experten und der Domain (Abb. 2)

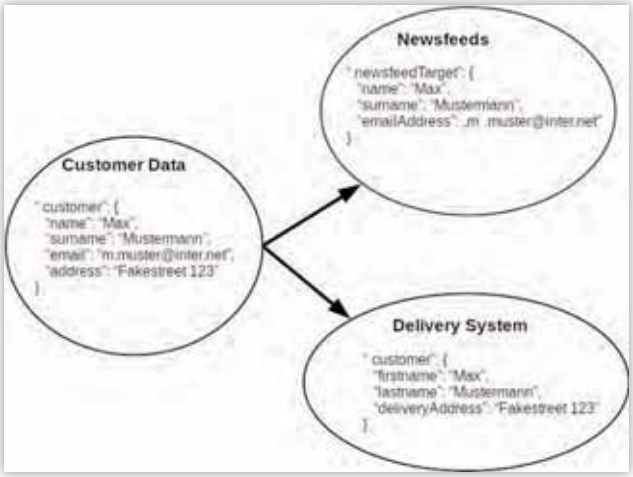


Sub-Domain und Bounded-Context (Abb. 3)

Bounded-Context

Betrachtet man eine beliebig komplexe Domain, so kommt man in der Modellierung schnell an seine Grenzen, sowohl auf grobem Level als auch in beliebigen Detailgraden. Einzelne Versuche erkennt man oft an wandgroßen Diagrammen auf DIN A0 Poster. Wie sehr diese fertigen Diagramme noch etwas mit der Realität der Domain, aber auch mit der aktuellen Software zu tun haben, sei dahingestellt. DDD bietet hier das Konzept der Sub-Domain und darauf aufbauend des Bounded-Context an. Eine Sub-Domain ist dabei immer ein abgegrenzter Teilbereich der umschließenden Domain, beispielswiese eine Abteilung innerhalb eines Unternehmens oder ein Prozess innerhalb eines Fachteams. So wie eine Sub-Domain einen Problemraum innerhalb der Domain bildet, so bildet darauf aufbauend der Bounded-Context einen Lösungsraum für die Sub-Domain.

(Abb. 3)[8] zeigt eine umfassende Domain mit vier identifizierten Sub-Domains, welche einen jeweiligen Problemraum aufspannen. Die Core-Domain nimmt dabei einen speziellen Fall ein und bildet die Kernkompetenz einer Unternehmung ab. Also die `Sub-Domain`, welche für den Erfolg der gesamten Domain am wichtigsten ist. Darauf aufbauend sind drei `Bounded-Contexts` zu erkennen. Diese bilden hier für sich einen eigenen Lösungsraum für den darunter liegenden Problemraum. Zu sehen ist hier auch, dass ein Kontext nicht die gesamte `Sub-Domain` einnehmen muss. Auch hier gilt, dass der Lösungsraum nur die notwendigen Aspekte abbilden soll, die für eine Umsetzung notwendig sind. Ein spezieller Fall bilden Bounded-Contexts, die Teile von mehreren `Sub-Domains` abbilden. Hierbei handelt es sich meist um zugekaufte Produkte wie ERP-Systeme oder Legacy-Systeme, die nicht genau auf die unterliegende Organisationsstruktur passen und Elemente aus anderen `Sub-Domains` mit einfließen lassen. Auch hier gilt, dass ein Kontext nicht starr ist. Über die Zeit finden sich ggf. bessere Schnitte oder es werden neue Sub-Domains identifiziert, bzw. eine bestehende `Sub-Domain` geteilt. Im Alltag bildet ein Bounded-Context einen abgegrenzten fachlichen Block, der zum Beispiel von einem Team bearbeitet werden kann. Das hat zur Folge, dass die bereits vorgestellten Elemente wie Modell und Ubiquitous-Language nun nicht mehr die gesamte Domain beschreiben, sondern immer nur genau ihren Bounded-Context. Jedes Team entwickelt für seinen Bounded-Context über die Zeit seine eigene Ubiquitous-Language und sein Modell. Das macht vor allem daher Sinn, da ein Bounded-Context eine reale Sub-Domain abbildet. Auch im Alltag findet sich zum Beispiel in einer Abteilung eine abgegrenzte Terminologie, die hier Einzug in den Kontext hält. Die Ubiquitous-Language ist eben keine universelle Sprache, die auf beliebige Abteilungen oder Entwicklungsteams passt. Sie ist lediglich im Kontext der Entwicklung gebräuchlich, verständlich und teambezogen.



Drei Bounded-Context mit jeweils eigenen Modellen und Sprachen (Abb. 4)

Strategic-Design

(Abb. 4) zeigt exemplarisch, wie drei Kontexte in Beziehung stehen. So kann ein simpler Begriff, wie `customer` im Team `Customer-Data` eine ganz andere Bedeutung und Abhän-gigkeiten haben, wie im Team `Delivery-System`. Im Team `Newsfeed` gibt es sehr ähnliche Daten, aber einen ganzen anderen Begriff. Bounded-Contexts übernehmen nicht einfach andere Modelle, sondern überführen sie in ihre eigene Ubiqui-tous-Language, bzw. in ihr eigenes Modell. Gerade die Beziehungen zwischen Bounded-Contexts und damit auch einzelnen Teams ist einer der wichtigsten Elemente innerhalb von DDD, da hier auf ein Team oft nicht berechenbare Einflüsse einwirken. Ein Modell oder Ubiquitous-Language liegt immer in der Hoheit eines Kontexts und damit eines Teams. Aber wie es an Informa-tionen von anderen Kontexten kommt, bzw. Informationen an andere Kontexte weitergibt, ist eine Sache der Kommunikation. DDD gibt hier verschiedene Patterns an, wie zwei Kontexte oder Teams miteinander interagieren können:

- Shared-Kernel: Zwei Kontexte teilen sich eine gemeinsame Code-Basis. Eine Bibliothek oder Teile eines alten, abzulösenden Systems. Die Sprache dieser Code-Basis muss nichts mit der Ubiquitous-Language des jeweiligen Kontexts zu tun haben.
- Customer-Supplier: Zwei Teams stehen in direkter Beziehung als Versorger und Konsument. Die Kommunikation stimmt und man kann sich auf eine gemeinsame Schnittstellensprache einigen und kommuniziert Änderungen zeitnah und oft.
- Partnership: Die Partnership ist die engste Beziehung. Beide Teams sind nur zusammen erfolgreich oder nicht. Beide Teams haben eine starke Motivation zu einer optimalen Beziehung.
- Conformist: Wie beim Customer-Supplier, nur liefert das versorgende System sein eigenes Modell aus und besitzt keine Motivation zur Änderung, wie zum Beispiel ein zugekauftes System. Das konsumierende Team nimmt die Schnittstelle wie sie ist.
- Anticorruption-Layer: Im Falle eines Shared-Kernels oder einer Conformist-Beziehung möchte das Team sein Modell schützen. Von außen gegebene Fremdmodelle werden zunächst in das eigene Modell übersetzt und validiert, bevor es in den eigenen Kontext gegeben wird.
- Separate-Ways: Der abhängige Kontext bietet dem Team so marginale Funktionalität, dass eine Eigenentwicklung schneller und kostengünstiger ist als eine teamübergreifende Kommunikation und damit Abhängigkeiten.
- Open-Host-Service & Published-Language: Das Team definiert eine öffentliche, allgemeine Schnittstelle in ihrem Kontext. Abhängige Teams binden sich per Conformist-Pattern an, oder beide Teams bilden eine Customer-Supplier-Beziehung.

(Abb. 3) bildet, mit den Informationen über den Beziehungstyp, eine Context-Map. Sie ist ein wichtiges Diagramm während der ersten Modellierungsphasen und stellt den eigenen Kontext in die Beziehung mit der Außenwelt.

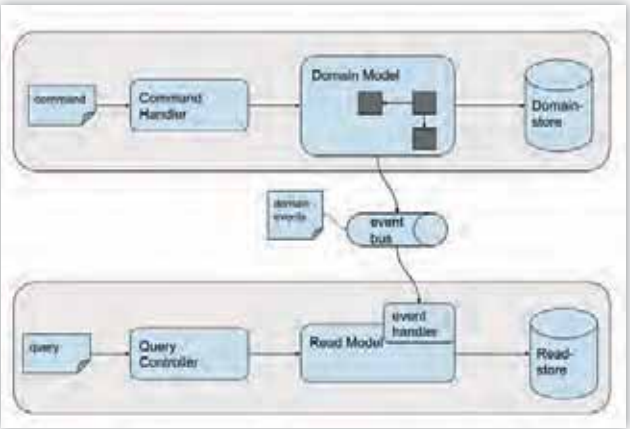
Softwarearchitektur

Entwickelt man nach DDD, so geben die Kernelement wenig bis gar keine Angaben darüber, wie eine Softwarearchitektur aussehen soll. Einzig das Modell wird gut beschrieben und sollte immer den Kern der Software bilden. APIs, Persistenz und Frameworks sollten darum gebaut werden und möglichst keinen Einfluss auf das eigentliche Modell haben. In den letzten Jahren haben sich einige Arten als stark kompatibel mit DDD heraus-gestellt, woraus hier zwei kurz skizziert werden sollen: CQRS und hexagonale Architektur. Daneben gibt es auch viele weitere Arten, denen allen gemein ist, dass sie einen stark abgekapseltes Domain-Modell als fachlichen Kern haben.

Command-Query-Responsibility-Segregation (CQRS)[9]

Diese Architektur mit dem etwas sperrigen Begriff bietet sich bei einer stärkeren Trennung von schreibenden und lesenden Zugriffen an. Das Modell wird hier als Domain-Model in das Zentrum der Command-Ebene gerückt. Über Commands, wie zum Beispiel `UpdateAddress`, werden Zustandsübergänge im Modell initiiert. Wichtig dabei ist, dass nur das Domain-Mo-dell die Commands abarbeitet, also der interne Zustand des Modells lediglich mit einem Command überführt oder durch einen Fehler quittiert wird. Zustandsänderung von außen sollten möglichst vermieden werden. Fand eine Zustandsänderung statt, so manifestiert sich diese in einem Domain-Event. Analog zu dem Command `UpdateAdress` zum Beispiel das Event `AdressUpdated`.

Domain-Events werden von einem oder mehreren Event-Handler abgefangen und in ein Read-Model überführt. Dieses Modell muss nicht zwangsläufig alle Daten des Domain-

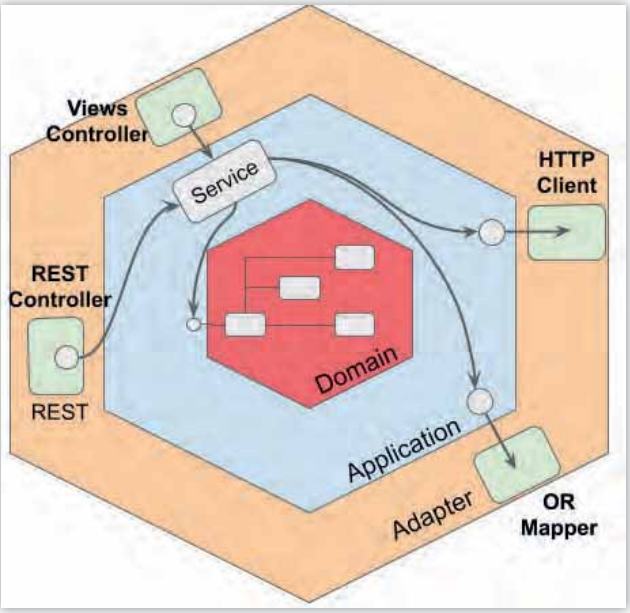


CQRS Architektur (Abb. 5)

Modells beinhalten und kann sogar andere Quellen anziehen. Hier lassen sich gut Grenzen über Bounded-Contexts abbilden, aber auch innerhalb eines Context verschiedene Darstellungen der Domain umsetzen.

Hexagonale Architektur / Ports- and Adapters

In dieser Architektur befindet sich das Modell, hier Domain genannt, im Zentrum einer Schalenstruktur mit zwei weiteren Schichten: der Application und dem Adapter. Die Domain ist der fachliche Kern und bietet der Application definierte Schnittstellen, die den internen Zustand ändern können. Wie bei CQRS ist auch hier wichtig, dass das Modell nur über diese Schnittstellen geändert werden darf. Eine direkte Änderung von außensollte vermieden werden[10]. Die Application-Schicht nutzt die Methoden des Modells und definiert darüber Schnittstellen, Ports genannt, welche Usecases abbilden. Services kümmern sich dann um die Aggregationsaufgaben.



Hexagonale oder Ports- and Adapters Architektur (Abb. 6)

(Listing 1)

```
package Application

public interface AddressPort {
    Customer updateAddress(CustomerUuid uuid, Address address);
}

public class AddressService implements AddressPort {
    CustomerRepositoryPort customerRepository;

    @Override
    public Customer updateAddress(CustomerUuid uuid, Address
address) {
        Customer customer = customerRepository.load(uuid);
        // ... null check
        Customer changedCustomer = customer.addAddress(address);
        // ... error handling
        customerRepository.save(changedCustomer);
        return changedCustomer;
    }
}
```

Die Application definiert einen Port `AdressPort` mit einer Methode - die Implementierung findet ebenfalls in der Application statt. Hier kommt es aber lediglich zu einer Aggregation verschiedener Elemente. Zu sehen ist zum Beispiel ein `CustomerRepositoryPort`, welcher ebenfalls definiert wurde. Über diese Schnittstelle wird anhand der `uuid` ein `Customer` als Domain-Modell geladen und die Adresse über die fachliche API an das Modell gegeben. Im Erfolgsfall wird das so geänderte Modell über den `RepositoryPort` gespeichert. Die Schicht Adapter bietet schlussendlich die technische Anbindung in Form von Frameworks und APIs. So kann zum Beispiel der `CustomerRepositoryPort` per JPA implementiert werden. Aber auch eine Implementierung zu einer NoSQL-Datenbank ist möglich. Daneben kann der `AddressPort` zum Beispiel von einem REST-Controller genutzt werden, um eine Schnittstelle über das Web zur Verfügung zu stellen.

Das Modell

Kern einer jeden DDD Software ist das Modell. Dieses zu konstruieren ist allerdings nicht trivial, modelliert es doch die Realität so viel wie nötig und so wenig wie möglich. Terminologie und fachliche Logik sind initialer Bestandteil des Modells und sollten auch nur hier zu finden sein. Vaughn Vernon gibt in seinem Werk über mehrere Kapitel eine sehr gute Einführung über viele abstrakte Elemente, die ein Modell ausmachen. Folgende Übersicht gibt daher nur einen kurzen Abriss darüber und ist auch nicht vollständig.

Entity:

Ein Entity ist ein konkretes Ding aus der Realität. Das kann eine konkrete Person sein, aber auch ein Container mit einem Ziel.

Entities haben immer eine ID und sind immer im Bounded-Context, wenn nicht sogar weltweit eindeutig. Auf Programmebene sollten Entities einen Mutable-State besitzen und dürfen bei gleichen Werten nicht gleich sein, da sie zwei trotzdem unterschiedliche reale Objekte beschreiben. Entities besitzen die meisten Methoden, da hier viele Interaktionen stattfinden können.

(Listing 2)

```
public class Customer {
    CustomerUuid uuid;

    void updateAddress(Address address);
    RegistrationState registerNewEmail(EmailAddress email);
}
```

Value-Objects:

Values-Objects sind dagegen Objekte ohne eine eigene Identität. Zum Beispiel eine Farbe oder ein Name. Programmatisch sollten Value-Objects immutable sein. Objekte mit denselben Werten sollen gleich sein. Man sollte sich nicht scheuen, in der Domain auch einfachste Begriffe mit einem Value-Object zu versehen. Denn zwei Strings lassen sich immer vergleichen und ersetzen. Ein Name aber nicht mit einer Farbe. Auch haben Namen und Farben gegebenenfalls andere Validierungsregeln bezüglich ihres Wertes.

(Listing 3)

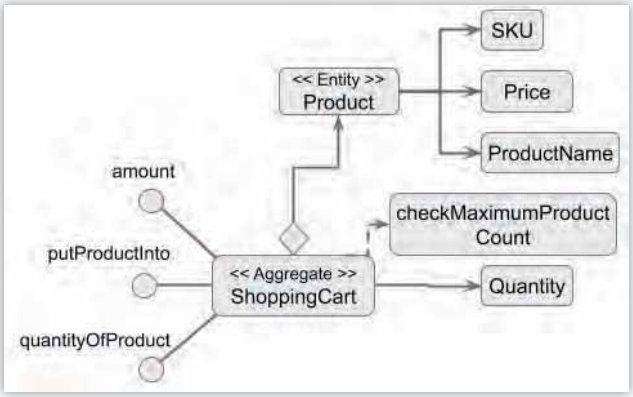
```
public class Name {
    private String value;
}

public class Farbe {
    private String value;
}
```

Aggregate:

Ein Aggregate ist eine Sammlung von Domain-Objekten, die unter einem gemeinsamen Begriff zu finden sind. So könnte im Bestellwesen ein Warenkorb das Aggregate sein. Bestehend aus dem Warenkorb an sich, vielen Produkten und deren Anzahl, dazugehörigem Preis und Zusatzinformationen. Die Außenwelt sollte lediglich über Aggregate mit den Domain-Objekten kommunizieren.

Die Ubiquitous-Language im Modell:

Domain-Begriffe sollen sich auch vor allem im Programmcode wiederfinden. Im deutschen Sprachraum stellt sich hier immer die Frage, ob sich deutsche Begriffe im Code als englisches Pendant wiederfinden sollen oder ob man direkt das deutsche Synonym benutzt. Einen goldenen Weg gibt es nicht und man sollte von Fall zu Fall entscheiden wie man vorgehen möchte. Englische



Beispiel eines Domainmodels mit einem Aggregate (Abb. 7)

Übersetzungen fördern sicherlich die Möglichkeit der Internationalisierung. Anderseits sollte man sich auch die Frage stellen, warum für eine geplante internationalisierung nicht bereits die Domain-Sprache Englisch ist. Deutsche, originäre Begriffe, die sich nicht oder nur schwer übersetzten lassen, sparen viel Zeit und Diskussion, wenn man sie einfach im Code verwendet. Ein schönes Beispiel ist hier das Einschreiben Einwurf im deutschen Postverkehr. Übersetzungen sind schnell irreführend. Die deutsche PLZ wird gerne mit ZIP-Code übersetzt. Leider sind PLZs und ZIP-Codes zwei unterschiedliche Konzepte der deutschen und US-Amerikanischen Adresssysteme. Zwei ähnliche Begriffe aus zwei unterschiedlichen Domains. Eine deutsche PLZ, die aufgrund einer falschen Übersetzung mit einem fertigen ZIP-Code-Validator geprüft wird, erzeugt schnell Bug-Tickets.

Fazit:

Domain-Driven-Design ist in seiner Gänze sicherlich keine leichte Lektüre, aber es lohnt sich. Denn egal auf welcher Ebene man sich der Softwareentwicklung befindet, es hilft den Grundzug von Domain-Driven-Design zu verinnerlichen: Den fachlichen Kern und den Sinn der Software in das Zentrum der Entwicklung und des Denkens zu rücken. Denn auch DDD ist keine neue Erfindung, sondern rückt viele bekannte Elemente der Softwareentwicklung einfach in ein neues Licht und in einen neuen Fokus.

Quellen:

1 https://bit.ly/youtube-west-w1
2 https://bit.ly/mud-m2
3 vgl. [evans]4vgl. [vernon], Seite 43 & 44
5 vgl. [brandolino], Kapitel 1
6 vgl. [vernon], Kapitel 2, S. 56,
7 vgl. [vernon], Kapitel 2, S. 66 - 71
8 vgl. [vernon], Kapitel 2, S. 509 https://bit.ly/fowler-f9
9 vgl. [vernon] Kapitel 4, S. 125 ff.

#JAVAPRO #DomainDrivenDesign

DDD-Bounded-Context Deep-Dive

„Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt." Prägnanter als der Philosoph Ludwig Wittgenstein kann man die Maxime des technischen Systemschnitts nicht formulieren. „Und diese Welt heißt Bounded-Context" müsste man aus der Perspektive des Domain-Driven Design (DDD) ergänzen.

Dieser Artikel thematisiert die sinnvolle Zerlegung komplexer Fachlichkeit in Systeme. Ein tieferes Verständnis des Bounded-Context ist sein Ziel. Dafür werden zunächst die Kanoniker Eric Evans (Schöpfer und Autor von Domain-Driven Design) und Vaughn Vernon (Autor von Implementing Domain-Driven Design) befragt. Um besser zu verstehen, wie Systemgrenzen gefunden werden, kommt im Anschluss die Wissenschaft zu Wort. Weil technische Systeme zwar Ergebnis, aber nicht Ausgangspunkt einer Modellierung sind, ist die Informatik keine Ansprechpartnerin für dieses Problem. Auskunft über soziale und sprachliche Unternehmenswelten erteilen Philosophie und Soziologie. Deren Antworten werden zitiert. Fragen der Modellierung neuer Fachlichkeiten bilden den Abschluss.

Kontext im Domain-Driven-Design

Bounded-Context ist im DDD der Schlüsselbegriff für Grenzziehungen. „Explicitly define the context within which a model applies. Boundaries in terms of team organization ..., and physical manifestations such as code bases and database schemas"[1]. Teams und Artefakte werden anhand der Grenzen eines Bounded-Context geschnitten. Für einen Bounded-Context wird ein Domänen-Modell entwickelt. Der Bounded-Context ist der Kontext, in dem eine eindeutige Fachsprache mit klaren Begriffen gesprochen wird: „Within each Bounded-Context, you will have a coherent dialect of the Ubiquitous Language."[2] Ubiquitous-Language ist die Fachsprache, die in einem Projekt allgegenwärtig (ubiquitous) sein soll. Sie wird von der Fachabteilung gesprochen. Sie soll von den Entwickelnden gesprochen werden. Auf ihr basiert die Namensgebung im Quellcode und das gesamte Domänen-Modell. Während Eric Evans 2004 noch an eine unternehmensweite Ubiquitous-Language glaubte und die verschiedenen Bounded-Contexte auf den Dialekten dieser Sprache basieren ließ, ist für Vaughn Vernon 9 Jahre später in seinem Buch „Implementing Domain-Driven Design“ klar: „There is one Ubiquitous Language per Bounded-Context." und: „It's chiefly a linguistic boundary."[3]

Was bedeutet das unabhängig von den zitierten Autoren? In einem Unternehmen werden verschiedene Sprachen gesprochen. Wenn das Marketing den Begriff Produkt verwendet, bedeutet es etwas anderes, als wenn die Logistikabteilung ihn benutzt. Für den Entwurf von Systemen sind diese sprachlichen Grenzen entscheidend. Es gilt Bereiche zu identifizieren, in denen eine einheitliche Sprache gesprochen wird, in denen die Begriffe eine klare Bedeutung haben. An diesen Bereichen sind die technischen Systeme auszurichten. Teamzuständigkeiten, Codebasen, Datenbanken, alle Ressourcen sollen sich an diesen Grenzen orientieren.
So charmant und einfach dieser Gedanke auf den ersten Blick erscheint, so kompliziert und gar nicht mehr intuitiv ist er auf den Zweiten. Wie soll in einem Unternehmen ein Geschäftsprozess

Autor:

Jens Himmelreich, geboren 1964. Nach einer Ausbildung zum Energieanlagenelektroniker, Studium der Informatik und der Philosophie an der Universität Bremen. 2006 mit fünf Kollegen Gründung von 'neuland - Büro für Informatik'. Bis heute dort beschäftigt.

Webseite: http://jenshimmelreich.de
Blog-Url: http://jens.himmelreich.de/blog
Firmen-Webseite: https://neuland-bfi.de/
Twitter: https://twitter.com/jenshimmelreich
Email: jens.himmelreich@posteo.de

realisiert werden, wenn alle unterschiedliche Sprachen sprechen? Warum kann man diese Sprachen nicht vereinheitlichen? Wäre das nicht im Interesse aller? Ist das Zulassen von unterschiedlichen Begriffen eine Schludrigkeit, der durch ein sprachliches Aufräumen begegnet werden kann? Wie kann man diese Grenzen, wenn sie sich denn wirklich nicht verhindern lassen, erkennen? Es scheint sich zu lohnen, diesem zweiten Blick eine ausführlichere Betrachtung zu widmen.

Der Wunsch nach einem unternehmensweit einheitlichen Datenmodell ist ein alter Traum der Informatik. Wenn es gelänge solch ein Modell zu etablieren, würden Redundanzen der Vergangenheit angehören. Es gäbe genau ein Modell für Kunden, für Produkte und für alle anderen zentralen Entitäten. Dieser Glaube speist sich aus einer einfachen und intuitiven Überzeugung. Die Dinge, auf die unsere Begriffe verweisen, gibt es ja auch genau einmal. Alle Produktbegriffe die in einem Unternehmen implementiert werden, alle Kunden die in Domänen-Modellen modelliert werden, referenzieren letztendlich auf den gleichen Gegenstand: den einen Kunden, das eine Produkt.

Kontext und Philosophie

Ludwig Wittgenstein, ein Philosoph der ersten Hälfte des 20. Jahrhunderts, bringt das in seinem Frühwerk auf den Begriff: „Der Name bedeutet den Gegenstand."[4] Er formuliert in seiner Schrift Tractatus-Logico-Philosophicus eine Abbildtheorie der Sprache. Er hegt die Hoffnung, Sprache ließe sich pflegen, so dass Doppeldeutigkeiten verschwinden würden. Sein Ideal ist eine Sprache, die genau das leistet, was wir in unserem unternehmensweiten Datenmodell bräuchten: Eindeutigkeit.

Wittgenstein ist der Meinung, alle grundlegenden Fragen der Philosophie beantwortet zu haben. Er zieht sich zurück. Nach 10 Jahren als Dorfschullehrer, Gärtnergehilfe, Innenarchitekt und Künstler nimmt er seine philosophische Arbeit in Cambridge wieder auf. Dort wendet er sich von seiner frühen Theorie der Idealsprache ab. Die natürliche Sprache, wie sie von den Menschen gesprochen wird, wird sein Referenzpunkt. Er versteht sie nicht mehr als defizitär. „Die Bedeutung eines Wortes ist sein Gebrauch in der Sprache." schreibt er in §43 der Philosophischen Untersuchungen[5]. Bedeutung wird nicht mehr durch eine Referenz auf Außersprachliches bestimmt, sondern durch die Benutzung eines Wortes in der Sprache.

Zurück zur Modellierung. Wittgenstein zufolge verweisen Worte auf einen Handlungskontext. In einem anderen Handlungskontext kann das gleiche Wort eine andere Funktion und eine andere Bedeutung haben. Und genau das drückt der Begriff Bounded-Context aus. Gegen einen unternehmensweiten Kontext, gegen eine Referenzauffassung von Worten ist der Begriff Bounded-Context ein Instrument, um unternehmensintern Handlungskontexte zu identifizieren. In einem Handlungskontext wird eine Sprache gesprochen. Diese Sprache dient dazu, die Handlungen in diesem Kontext zu koordinieren. Die Worte in diesem Kontext erfahren ihren Feinschliff in der tagtäglichen Praxis. Deshalb verwundert es nicht, wenn die Begriffe Kunde und Produkt im Marketing eine gänzliche andere Verwendungsweise erfahren als in der Logistik. Sie sind Instrumente, um andere Probleme zu lösen. Für die Logistik gibt es viele Produkte mit der gleichen Artikelnummer. Alle unterscheiden sich. Jedes hat seinen physischen Ort: im Regal, auf dem Weg zum Kunden. Für das Marketing unterscheiden sich diese Exemplare nicht. Ein Produkt ist Mitglied einer Familie. Es existiert in Größen und in unterschiedlichen Ausprägungen (Farben, Längen etc.). Der Logistik sind diese Familien egal - Hauptsache sie passen ins Regal.

Wittgenstein zeigt, warum es den Traum vom unternehmensweiten Datenmodell gibt, denn jeder Begriff referenziert ein Ding der wirklichen Welt. Aber Wittgenstein zeigt auch, warum präzise Begriffe nur in einem konkreten Handlungskontext entstehen. Nur dort werden sie von der Praxis ausgebildet, nur dort erfahren sie eine scharfe Semantik. Die Modellierung von Programmen muss sich an präzisen, klar umrissenen Begriffen, an einer ausgearbeiteten Fachlichkeit orientieren. Dort, wo diese Begriffe, wo diese explizite Fachlichkeit nicht existiert, muss sie hergestellt werden. Bounded-Context ist die Aufforderung, in einem Unternehmen die sprachlichen Grenzen - mithin die Grenzen der Handlungsfelder - zu erkennen und für ihre Separierung zu sorgen. Separierung der Teams, des Codes, der Verantwortlichkeiten aufgrund der Grenzen, die uns die Sprache zeigt. „Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt."[6] Oder, wie es im DDD heißen müsste: die Grenzen meiner Systeme.

Kontext und Soziologie

Wittgenstein hilft zu verstehen, warum es überhaupt begrenzte Kontexte für die Bedeutung von Begriffen gibt und wie diese Kontexte grundsätzlich zustande kommen. Um zu verstehen, wie sich unterschiedliche Handlungskontexte in Unternehmen herausbilden, ist die Philosophie überfragt. Organisationssoziologische Betrachtungen helfen hier weiter. Niklas Luhmann rekapituliert in seiner Schrift „Zweckbegriff und Systemrationalität"[7] die Grundlagen für lokale Rationalitäten in Organisationen. Wie alle anderen Organisationen sind für Luhmann Unternehmen Systeme. Systeme haben ein Grundproblem, welches sie lösen müssen. Sie leben in einer Umwelt, die immer komplizierter ist als die Systeme selbst. Wie sie mit diesem Komplexitätsgefälle umgehen, wie sie die Komplexität der Umwelt auf ein verständliches Maß reduzieren, ist für sie die zentrale Frage. Die Grundstrategien, die eine Organisation einsetzt, um die Komplexität ihrer Umwelt auf ein bearbeitbares Maß zu reduzieren, sind: Umweltdifferenzierung, Teilsystembildung und lokale Rationalität.

Umweltdifferenzierung bedeutet, dass eine Organisation ihre Umwelt als unterschiedliche Umwelten wahrnimmt. Ein Unternehmen ist nicht einfach irgendwie im Markt unterwegs. Ein Unternehmen differenziert seine Umwelt. Es gibt Kunden, welche die Produkte des Unternehmens erwerben, es gibt Lieferanten, welche die Rohstoffe liefern, die zu Produkten verarbeitet werden. Es gibt Mitarbeitende, die die Arbeit verrichten. Es gibt rechtliche Regelungen, die bestimmen, in welchen Grenzen

Arbeitsprozesse vollzogen werden müssen. Es gibt eine Öffentlichkeit, die via Medien das Unternehmen beobachtet.
Teilsystembildung bezeichnet die Ausbildung von Teilsystemen, die für Umweltbereiche zuständig sind. Die Marketingabteilung beobachtet den Kunden, der Einkauf verhandelt mit den Lieferanten, die Personalabteilung stellt Mitarbeitende ein, die Rechtsabteilung liest Verordnungen und überwacht Prozesse, die PR-Abteilung teilt der Presse ihre Sicht der Dinge mit. Jede dieser Abteilungen bildet eine lokale Rationalität aus. Das Marketing beginnt die Produkte mit den Augen der Kunden zu sehen. Der Einkauf denkt in den Rhythmen seiner Lieferanten. Die Rechtsabteilung spricht die Sprache der Gesetze und Verordnungen. Die PR-Abteilung vernetzt sich mit der lokalen Presse. Das alles heißt: Die Bereiche des Unternehmens betten sich in die sachlichen, sozialen und zeitlichen Bezüge ihrer Umwelt ein.

Durch diese Strategie gelingt es einem Unternehmen, ein hohes Maß an Umweltkomplexität zu handhaben. Aber es zahlt auch einen Preis. Komplexität, die bisher an der Schnittstelle zur Umwelt existierte, wird in das eigene Unternehmen geholt. Die Marketingfrau, die jetzt die Sprache des Kunden spricht, versteht den Mann aus dem Einkauf nicht mehr, der wie ein Lieferant denkt. Der Kollege aus der Rechtsabteilung, der keine Schwierigkeiten hat, komplizierte Gesetze zu lesen, wird von der Kollegin der PR-Abteilung gemieden, weil niemand versteht, was er sagt.
Interne Komplexität ist der Preis für eine bessere Anpassung an die Umwelt. Für den Bau von IT-Systemen heißt das: lokale Rationalitäten identifizieren und in ihrem Kontext IT-Systeme bauen. Der Versuch, übergreifende Begriffe zu etablieren und umfassende Datenmodelle zu implementieren, macht die Werkzeuge, die für den spezifischen Bereich entwickelt wurden, stumpf. Unternehmensweite Homogenität verringert die Anpassungsfähigkeit an die Umwelt.
DDD, Bounded-Context und Ubiquitous-Language sind die Übersetzung der organisationssoziologischen Begriffe: Umweltdifferenzierung, Teilsystembildung und lokale Rationalität, in die Modellierung von IT-Systemen. Sie werden durch eine systemtheoretische Sichtweise auf Organisationen geerdet.

Was bedeutet das für die konkrete Modellierung? Die Identifikation von Bounded-Contexts ist die Schlüsselfrage für den Systemschnitt. Die Sensibilität für unterschiedliche Sprachen, die in einem Unternehmen gesprochen werden, ist ein gutes Hilfsmittel, um Handlungskontexte zu identifizieren. Die Identifikation der spezifischen Umwelten, mit der ein Unternehmensbereich interagiert, ist eine weitere Heuristik, Bounded-Contexts zu erkennen.

Kontext und die Neue Welt

Was, wenn der Bereich für den ein IT-System entwickelt werden soll, noch gar nicht existiert? Wenn der Bereich mit dem System entsteht? Hier ist es nicht möglich auf die Sprache der Fachabteilung zu hören. Sie ist erst im Entstehen. Sie wird sich zunächst an die Sprachen anlehnen, die im Unternehmen bereits gesprochen werden. Ihre Mitarbeitenden werden vorher an anderen Stellen des Unternehmens gearbeitet haben und zunächst ihre Sprache und Sicht der Dinge mitbringen. Ein typischer Fall für diese Situation ist die Erweiterung um einen digitalen Vertriebskanal, die in den letzten 20 Jahren alle Handelsunternehmen erfahren haben. Das Internet wurde zunächst als marginaler Vertriebskanal aufgebaut und für seine Systeme stand die Frage, welcher Sprache sie sich bedienen. Die eigene Fachsprache der Internet-Abteilungen existierte noch nicht, aber die Frage nach der adressierten Umwelt half. Auch für eine neue Abteilung gilt, dass sie eine Umwelt hat. Wird diese Umwelt bereits im Unternehmen adressiert und gibt es eine Fachsprache, die sich mit dieser Umwelt verschränkt hat, dann ist dies eine gute Quelle für die Domänensprache des neuen Bereiches. Gibt es das nicht - und das Internet war eine neue Umwelt für alle Unternehmen - gilt es etwas Neues zu schaffen, in die neue Welt hineinzuhören und die Sprache der Umwelt zu adaptieren. Mittlerweile ist es klar, dass das Internet eine eigene Sprache spricht. Wer sich nicht mit Web, SEO, Page-Rank, Recommendations und PayPal auskennt, wer nicht die Sprache des Web spricht, hat keine Chance und wer das zuerst erkannte, hatte einen klaren Wettbewerbsvorteil.

Zusammengefasst

Der zentrale Begriff für den Schnitt von Systemen ist Bounded-Context. Bounded-Contexts zeichnen sich durch eine gemeinsame Fachsprache aus. In einem Unternehmen gibt es viele Bounded-Contexts. Ein Bounded-Context verweist auf einen Handlungskontext. Das Ausbilden eines Bounded-Context mit einer eigenen Fachsprache ist ein Mittel eines Systems, hier eines Unternehmens, um die Komplexität seiner Umwelt zu meistern. Der Preis, den es für die höhere Anpassungsfähigkeit zahlt, ist interne Komplexität - im Unternehmen werden verschiedene Sprachen gesprochen. Die Sensibilität für Fachsprachen ist ein gutes Hilfsmittel für den Systementwurf. Eine weitere Hilfe ist es, zu erkennen, welche Umwelt der Kontext adressiert. Werden in einem Unternehmen neue Bereiche aufgebaut, trägt die Frage der Fachsprache für die Modellierung nicht, während die adressierte Umwelt immer noch ein gutes Indiz ist. Mithilfe dieses tieferen Verständnisses für die Designstrategien des DDD schützt man sich vor verlockenden Abkürzungen oder: Wer hätte nicht schon einmal davon geträumt, ein System als Quelle aller Wahrheit zu implementieren?

Quellen:

1 [Eva04, S. 336]
2 [Eva04, S. 345]
3 [Ver13, S. 25]
4 [Wit63, 3.203]
5 [Wit03]
6 [Wit63, 5.6]
7 [Luh73, 182ff]

MicroStream

Java-Native Object Graph Persistence.

Realize ultra-fast in-memory data processing with pure Java.
Microsecond Query Time. Low-Latency Data Access.
Gigantic Data Throughput and Workloads.

MicroStream 5 is Now Open Source

We are very excited to announce that the new MicroStream version 5 is now available as Open Source. The source code is published under the Eclipse Public License (EPL) on our GitHub page.

MicroStream is used productively in business-critical projects for more than 6 years. It's proven, stable and has a high code quality. Now, it's time to open source it. Open Source provides strong value and great benefits for our community. The source code is now avialable for anyone to read. Debugging, finding and fixing bugs is possible. Security issues can be found and eliminated faster. It gives you the flexibility to customize and extend MicroStream on your own. Permanently reviews of the source code and numerous testing cycles by developers across the world will increase the stability of MicroStream. This will even improve the already high quality of MicroStream. The entire community can collaborate, support other community members and contribute to making MicroStream better.

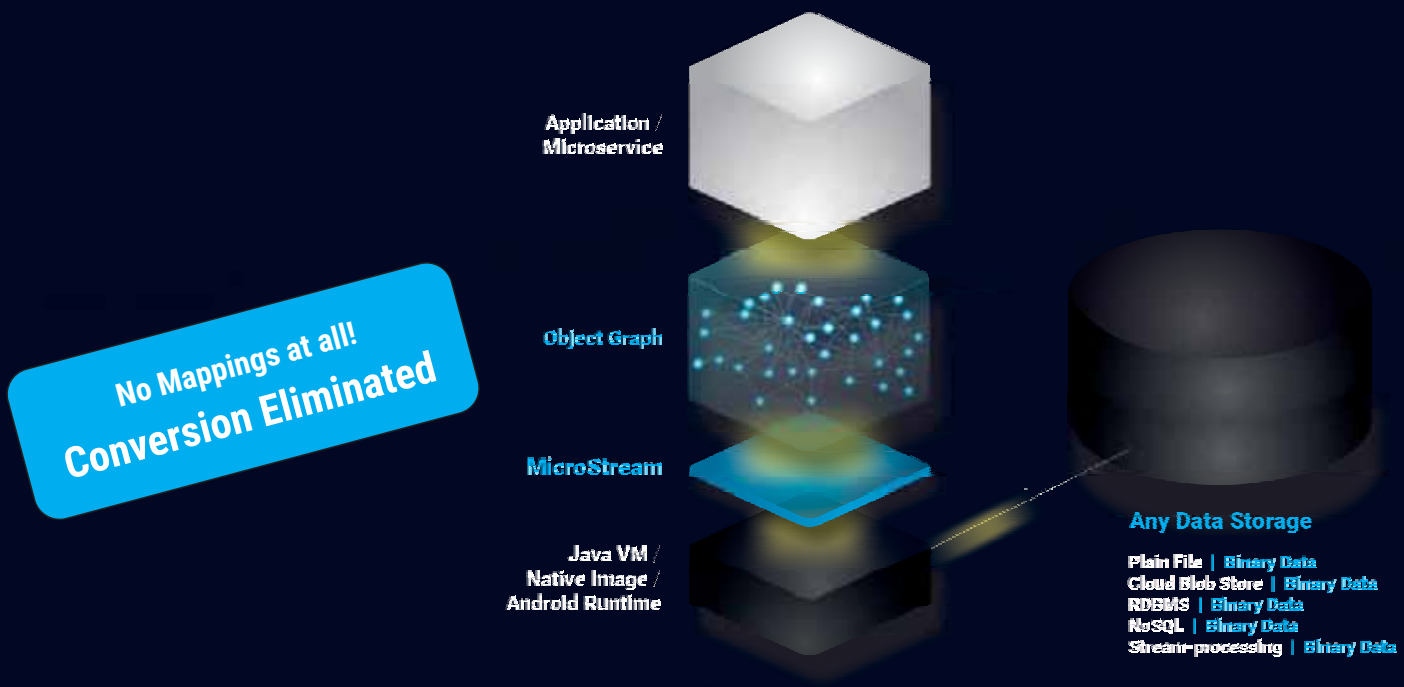
Using MicroStream becomes even easier. You don't need to worry about the license. Just use it, get started quickly, and check out whether it can solve your business problem, build great solutions. You can start small and quickly and scale up without limitations. Last, but not least, developers love open source. This increases the enthusiasm and motivation as well as the satisfaction of the developers.

We're very excited to see your ideas and contributions to MicroStream.

MicroStream Persistence

MicroStream is joyfully easy to use Java API you can download via Maven.

MicroStream is a Java-native object graph persistence engine for storing any complex Java object graph or any single subgraph and restoring it in RAM at any time by using a fundamentally new serialization concept designed from scratch. With MicroStream, not only the entire object graph, but also partial subgraphs, or only single objects can be restored in RAM on demand. Beyond serialization, MicroStream is ACID transaction safe, can handle your class changes, provides a garbage collector for the storage, multi-threaded IO, and connectors for various data storages.



- ✓ Incredible performance boost
- ✓ Saves lots of CPU power
- ✓ Simple architecture
- ✓ Pure Java, developer's joy
- ✓ Eliminates expensive latencies
- ✓ Reduces your infrastructure costs
- ✓ Reduces the development effort and costs

www.microstream.one

#JAVAPRO #Cloud #CI #CD

Cloud-Infrastruktur per Code mit Terraform

Infrastructure-as-Code ist die Bereitstellung von Rechenzentrumsinfrastruktur per Code anstelle von physischen Hardware-Konfigurationen. Mit Terraform ist jedoch nicht nur Infrastruktur als Code möglich, integriert in eine CI-Pipeline kann es auch für Deployments eingesetzt werden.

Mittlerweile gibt es eine ganze Sammlung an Tools, um Infrastructure-as-Code zu nutzen. Einige Cloud-Anbieter bieten eigene Lösungen an, wie zum Beispiel AW-CloudFormation oder der Google-Deployment-Manager, um nur zwei zu nennen. Hashicorp stellt mit Terraform eine Open-Source-Multi-Cloud-Lösung dafür bereit. Terraform bietet die Möglichkeit, Infrastruktur im eigens entwickelten deklarativen HCL-Format (Hashicorp-Configuration-Language) zu beschreiben und unterstützt dabei zahlreiche Cloud-Anbieter. Dabei beschränkt Terraform sich nicht nur auf eine statische YAML-ähnliche Syntax, sondern stellt auch zahlreiche Funktionen wie Schleifenkonstrukte, Typisierungen und Module bereit. Dank der Möglichkeit den Zustand der Infrastruktur, den Terraform-State, zum Beispiel auf einem Cloud-Speicher zu halten, eignet sich Terraform auch hervorragend für verteilte Teams und Kollaborationen.
Dies macht das Tool sehr flexibel im Einsatz und unterstützt beim Erstellen einer homogenen, skalierbaren Infrastruktur. Terraform erfreut sich zunehmend einer breiten Community, die für viele Cloud-Komponenten bereits einfach zu nutzende Module bereitstellt.

Autor:

Pascal Euhus arbeitet seit 3 Jahren als DevOps-Engineer bei Reservix GmbH. Hauptsächlich kümmert er sich um Automatisierungen und Cloud Betrieb. Zusätzlich ist er als Dozent an der dualen Hochschule Lörrach tätig. Besonderes Interesse gilt der Etablierung und Verbesserung von DevOps-Kultur in Unternehmen sowie Cloud-Migrationen.

Twitter: @pascal_euhus
LinkedIn: https://www.linkedin.com/in/pascal-euhus-611309164
GitHub: PacoVK
Mail: pascal.euhus@gmx.de

Vorbereitung

Für den Einsatz von Terraform sollte man bereits etwas Erfahrung in der Cloud mitbringen. Jede Ressource, die mit Terraform erstellt wird, muss in einem sogenannten Plan deklariert werden. Anders als beim Erstellen von Infrastruktur über diverse Web-Oberflächen muss man also wissen, welche Ressourcen insgesamt konfiguriert werden müssen (Rechte, Rollen, Firewall Richtlinien, etc.).
Einmal definiert kann die Infrastruktur eines Terraform-Plans aber dann jederzeit skalierbar reproduziert werden. Um einen Web-Service, der in einem eigenen Rechenzentrum auf einem Docker-Host läuft, für interne Zwecke in Zukunft in der Cloud zu betreiben, benötigt es vereinfacht eine dedizierte Cloud-Umgebung, welche wie das eigene Rechenzentrum vom öffentlichen Netzwerk abgekapselt ist, eine Docker-Umgebung und einen Load-Balancer **(Abb. 1)**.
Das folgende Beispiel baut die Infrastruktur für AWS mit Terraform auf, für andere Cloud-Anbieter gibt es vergleichbare Ressourcen. Der Terraform-Plan enthält also die in **(Listing 1)** gezeigte rudimentäre Konfiguration.

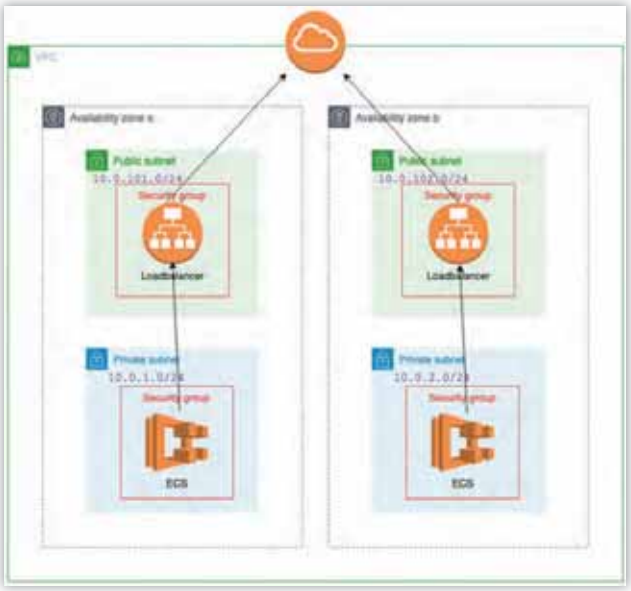
(Listing 1)

```
provider "aws" {
  max_retries = 1337
  region  = "eu-central-1"
  profile = "default"
}
```

Praktischerweise ist eine Gliederung in Dateien pro Ressourcen-einheit innerhalb eines Terraform-Plans ähnlich (Abb. 2) ratsam, um die Übersicht nicht zu verlieren.

Die private Public-Cloud

Was das eigene Rechenzentrum mit seinen getrennten Netzen und DMZ darstellt, das entspricht in der Public-Cloud die VPC

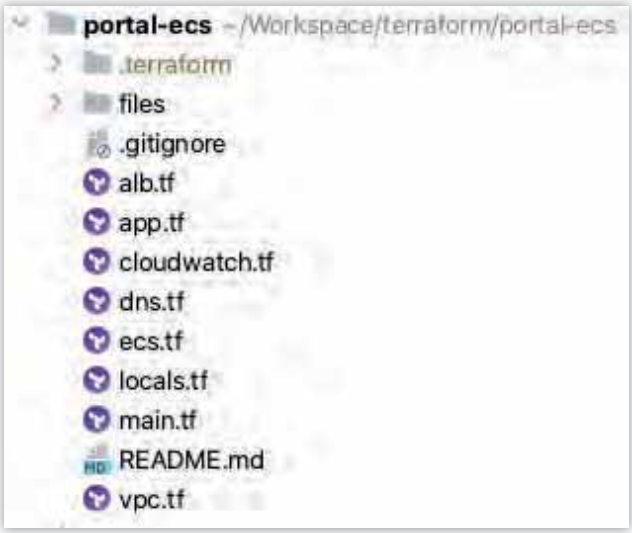


Zielarchitektur bei AWS (Abb. 1)

(Virtual-Private-Cloud). Bei AWS hat jeder Account per Default eine rudimentär konfigurierte VPC. Möchte man seine eigenen CIDR-Blöcke verteilen oder auch verschiedene Produkte voneinander isolieren, kann man weitere VPCs erstellen. Sieht man die VPC als wiederverwendbare Basisinfrastruktur, kann man entweder ein auf eigene Bedürfnisse angepasstes Modul mit Terraform schreiben oder wie in (Listing 2) das offizielle Modul nehmen[1].

(Listing 2)

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "my-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["eu-central-1a", "eu-central-1b", "eu-cent-
ral-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24",
"10.0.103.0/24"]
  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Terraform = "true"
    Environment = "dev"
  }
}
```



Terraform Projektstruktur (Abb. 2)

Nachdem die VPC und damit die Basis steht, kann nun mit der Docker-Infrastruktur weiter gemacht werden. Im Kontext von AWS und diesem Beispiel ist das der Elastic-Container-Service (ECS). Um es einfach zu halten, wird hier der Full-Managed-Service Fargate verwendet. Grundsätzlich sieht der Aufbau eines ECS-Clusters aber wie in (Listing 3) aus.

(Listing 3)

```
resource "aws_ecs_cluster" "my_cluster" {
  name = "my-cluster"
}
```

Im Wesentlichen fehlt nun die Migration der Anwendung auf die neue Docker-Umgebung. Aus Gründen der Leserlichkeit wird auf eine ausführliche Darstellung der Ressourcen verzichtet. (Listing 4) zeigt dennoch komprimiert auf, welche Komponenten noch fehlen.

(Listing 4)

```
resource "aws_cloudwatch_log_group" "logging"  {...}

resource "aws_ecs_task_definition" "container"  {...}

resource "aws_ecs_service" "service" {...}

module "security_groups" {...}

resource "aws_alb" "lb"  {...}

resource "aws_alb_target_group" "lb_target_group"  {...}

resource "aws_lb_listener" "lb_listener" {...}

resource "aws_route53_record" "lb_dns_alias"  {...}
```

Ein Terraform-Plan kann schnell sehr lang und unübersichtlich werden. Es empfiehlt sich daher Pläne logisch voneinander zu trennen. Dank der Module ist dies recht einfach möglich und erspart eine Menge doppelten Code. Es lohnt sich auch einen Blick in die Terraform-Registry mit den offiziellen Modulen für die jeweiligen Cloud-Anbieter[2].

Um den Plan auszuführen, muss Terraform zuerst alle Abhängigkeiten herunterladen, wie zum Beispiel Provider, Module etc. Anschließend kann der Plan ausgeführt und die gewünschte Infrastruktur erstellt werden. **(Listing 5)** zeigt den Ablauf dazu.

(Listing 5)

```
> terraform init
> terraform apply

#### Anstehende Änderungen werden ersichtlich mit
> terraform plan
```

Deployment? - Aber bitte automatisch

Mit diesem Setup kann man nun in diversen Accounts stets dieselbe Umgebung in Minuten aufziehen. Der Terraform-Code kann jetzt aber auch für das Deployment, integriert in eine CI/CD-Pipeline, genutzt werden. Das bedeutet nicht, dass der Release tatsächlich voll automatisch passieren muss. Aber zumindest bis zu den geplanten Änderungen kann Terraform ohne Bedenken automatisch ausgeführt werden. Die stärkste Integration bietet aktuell GitHub mit seinen Actions, die auch aktiv von Hashicorp entwickelt werden[3]. Aber auch in andere CI-Systeme lässt sich Terraform hervorragend integrieren. **(Listing 6)** zeigt eine einfache GitLab-CI-Pipeline.

(Listing 6)

```
.terraform:
  image: hashicorp/terraform

validate:
  extends: .terraform
  stage: validate
  script:
- terraform init -backend=false
- terraform validate

plan:
  extends: .terraform
  stage: plan
  artifacts:
paths:
  - review.tfplan
  script:
- terraform init
- terraform plan
  -out=review.tfplan
apply:
  extends: .terraform
  stage: apply
  when: manual
  only:
refs:
  - master
  dependencies:
- plan
  script:
- terraform init
- terraform apply
  -auto-approve
  review.tfplan
```

Letztendlich gibt es nun Entwicklern die Möglichkeit, Infrastruktur selbst nachzuvollziehen und zu pflegen. Durch eine entsprechende CI/CD-Pipeline wird nicht nur das Deployment der Software automatisiert, es eröffnet auch die Möglichkeiten Infrastrukturänderungen gleich wie Softwareänderungen zu testen, zu reviewen und zu deployen.

Homogene Umgebungen mit Terraform

Mithilfe des Konzepts von Workspaces[4] können Pläne über einen oder mehrere Accounts verwaltet werden. Dadurch kann eine Code-Basis dafür genutzt werden, eine Test-/UAT-Umgebung (User-Acceptance-Testing) analog zur Produktion aufzubauen. Das resultiert nicht nur in erweiterte Testbarkeit für die Softwarekomponenten, sondern eröffnet die Möglichkeit, Infrastrukturänderungen besser testbar zu machen, bevor man sie in Produktion ausrollt. Eine beispielhafte Account-Struktur ist in **(Abb. 3)** dargestellt. Dank der Möglichkeit der Interpolationen auch auf Provider-Ebene **(Listing 7)**, kann eine Code-Basis auf diese verteilten Accounts konsistent ausgerollt werden.

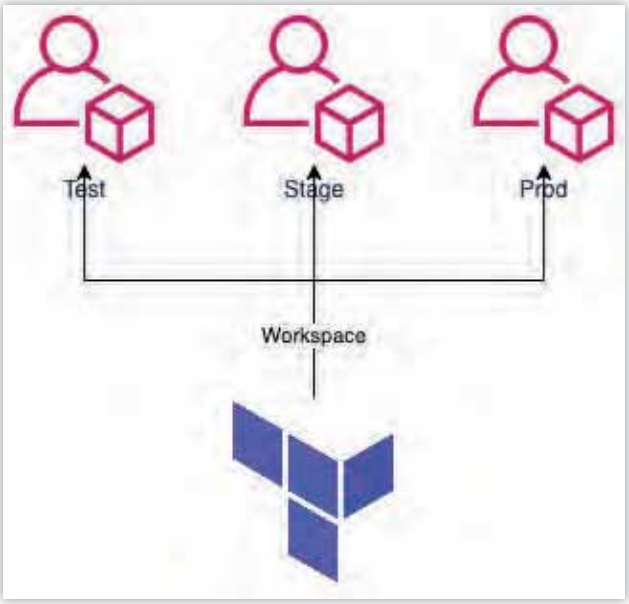
(Listing 7)

```
provider "aws" {
 max_retries = 1337
 region  = "eu-central-1"
 profile = terraform.workspace
}
```

Daraus ergeben sich weitere Vorteile, sodass zum Beispiel im Falle einer Account-Kompromittierung simpel eine Migration aller Ressourcen in einen neuen Account vollzogen werden kann.

Terraform und die Fallstricke

Terraform nutzt die APIs der Cloud-Anbieter. Grundsätzlich ein valider Ansatz, dennoch kann ein Terraform-Plan sehr schnell sehr komplex werden. Durch Kapselung in Module reduziert sich zwar die Anzahl der Code-Zeilen, die Zahl der Ressourcen und damit



Multi-Accounts - Getrennte Accounts je Umgebung (Abb. 3)

der Requests gegen die APIs allerdings nicht. Grundsätzlich gilt es hier eine logische Trennung der Komponenten eines Stacks vorzunehmen. Beispielsweise muss eine VPC-Konfiguration, die je nach Ausmaß mehr als 150 Ressourcen beinhalten kann, nicht zwingend bei anwendungsspezifischen Ressourcen wie zum Beispiel Load-Balancer oder API-Gateway liegen. Eine Entzerrung der Komponenten in verschiedene kleinere Pläne verkürzt die Terraform-Laufzeiten erheblich. Für fast jede Ressource gibt es die Möglichkeit auf Ressourcen außerhalb des Terraform-Planes zuzugreifen **(Listing 8)**.

(Listing 8)

```
data "aws_vpc" "vpc" {
  cidr_block = "10.0.0.0/16"
}

data "aws_subnet_ids" "public" {
  vpc_id = data.aws_vpc.vpc.id

  tags = {
Tier = "Public"
  }
}
```

Ebenso gibt es eine Infrastruktur, auf die zum Beispiel aus Compliance-Gründen nicht jeder Zugriff haben soll. Terraform nutzt hier lediglich die IAM-Strukturen der Cloud-Anbieter, also die lokalen Zugriffsschlüssel desjenigen der Terraform ausführt. Hat man also alle Ressourcen in einem Plan, muss derjenige, der den Plan ausführen will, auch Lese-/Schreibrechte auf diese Infrastruktur haben. Versehentliche Änderungen an der Basis-Netzwerktopologie sind sicher ein großes Risiko, was durch die Einbindung in vollautomatische CI-Pipelines maximiert wird. Auch hier minimieren getrennte Terraform-Pläne für kritische Infrastruktur das Risiko unbefugter Änderungen.

Dem Terraform-State, der den Ist-/Soll-Zustand der Infrastruktur darstellt, gilt besonderes Augenmerk. Ein Backup und redundante Speicherung wird hierfür empfohlen. Da vor jedem Ausführen des Terraform-Plans der Ist-Zustand der Ressourcen bei AWS mit dem Soll-Zustand (Terraform-Deklarationen) verglichen wird, muss gewährleistet werden, dass Terraform-Ressourcen nicht über die Web-Konsole verändert werden. Die Folge davon sind nicht vorhersehbare und nicht persistierte Änderungen.

Fazit:

Infrastructure-as-Code löst nicht nur die Probleme wie fehlende Dokumentationen über die Infrastruktur, sie eröffnet auch viele Möglichkeiten der Automatisierung und erleichtert den Umzug in die Cloud. Terraform bietet mit seiner Flexibilität aber noch mehr als IaC, vielmehr bringt es IaC noch näher an die Softwareentwicklung. Besonders erwähnenswert ist Hashicorps neueste Adaption, die auf Terraform aufbaut und Konstrukte aus Amazons AWS-CDK (Cloud-Development-Kit) übernimmt. AWS-CDK und Terraform-CDK haben das Potenzial IaC erneut disruptiv zu verändern. Die Grenzen zwischen Software- und Infrastrukturentwicklung verschwimmen immer mehr. So ist es möglich mit den CDKs seine Infrastruktur in derselben Sprache zu entwickeln wie die Applikation. Aktuell stehen für Terraform-CDK-TypeScript, Python und Java zur Auswahl[5].

Während IaC, trotz Terraforms Modulen und Funktionen aktuell noch sehr viel Code bedeutet, kann ein äquivalenter Terraform-Plan im CDK nur noch auf wenige Zeilen gestaucht werden. Durch eine adäquate Implementierung und Bereitstellung von Funktionen können nötige generische Ressourcen ähnlich wie in der Web-Konsole im Hintergrund erstellt werden. Somit ist ein derart tiefes Wissen über Zusammenhänge von Ressourcen nicht zwingend nötig für die Nutzung des Tools. Dadurch lässt sich die Lernkurve deutlich reduzieren. Wer sich vorab mit den Möglichkeiten eines Cloud-Development-Kits beschäftigen möchte, kann sich das AWS-CDK ansehen, welches auf dem AWS-spezifischen IaC-Tool CloudFormation aufbaut.

Quellen:

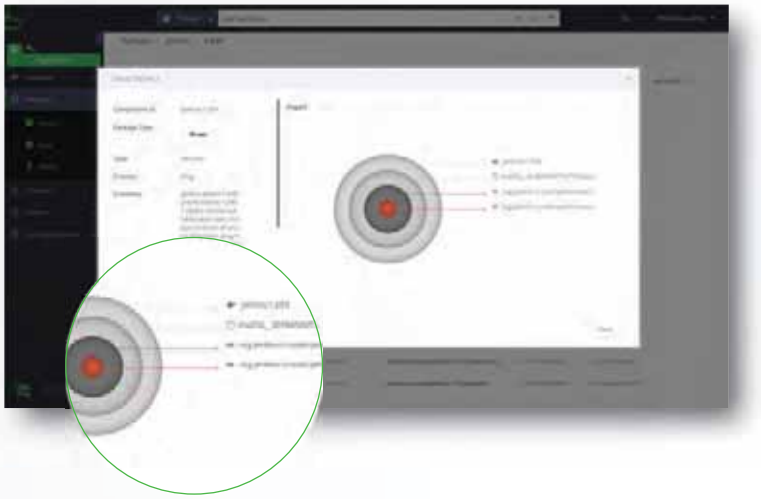
1 https://bit.ly/2P2wmgL
2 https://bit.ly/3bY7lfS
3 http://bit.ly/3vBcZMH
4 http://bit.ly/3s3O2qY
5 http://bit.ly/3r2pRb4

The Liquid Software Company

JFROG XRAY

Reduce Your Security and Compliance Risk

OVER 50% OF PUBLICLY AVAILABLE DOCKER HUB IMAGES CONTAIN AT LEAST ONE CRITICAL VULNERABILITY!



Not only that, some of these are known high severity vulnerabilities to be exact. If you're using Docker containers to deploy your applications, then your application code is potentially exposed to some serious exploits.

JFrog Xray has the unique capability to scan and recursively peel away all the different layers and their dependencies, ensuring that every software component included in your Docker image has been scanned for all known vulnerabilities and license compliance risks.

If you're using Kubernetes and Artifactory as your Kubernetes registry, Xray's native integration with Artifactory will enable you to identify any vulnerabilities so they don't pollute your deployments.

UNIVERSAL ARTIFACT ANALYSIS

JFrog Xray is a universal Software Composition Analysis solution that supports all major package types and integrations, knowing how to unpack each one and what every underlying layer contains. Each unpacked component is examined to uncover potential vulnerabilities and license compliance violations.

maven npm python™ GO php

alpine Linux debian nuget sbt Gradle ivy

Try it for yourself - *Start Free:*
https://jfrog.com/artifactory/start-free/

Read more on jfrog.com or sign up to one of our free English or German webinars!

#JAVAPRO #Cloud #CloudNative #Kubernetes

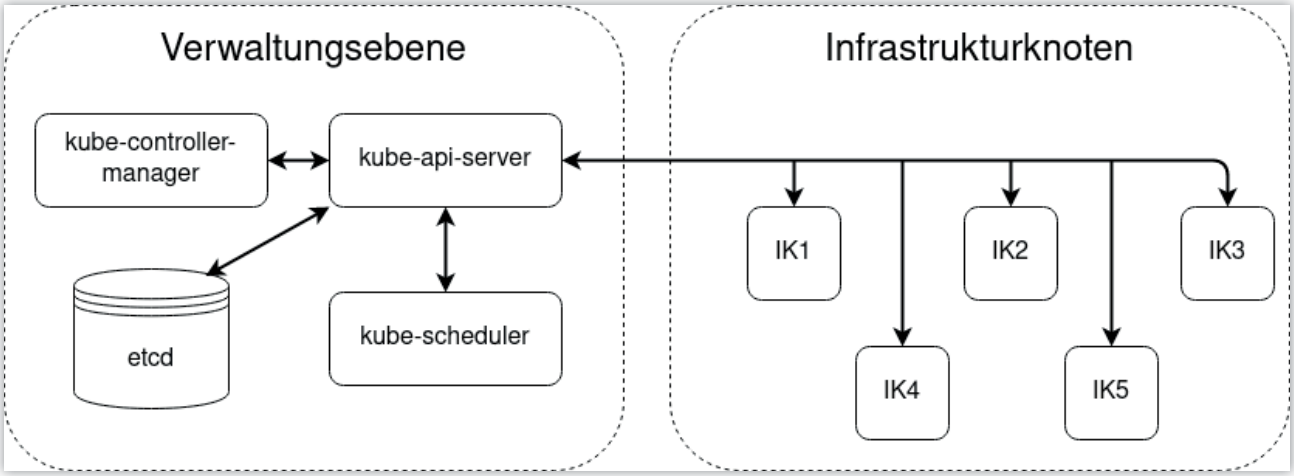
Kubernetes-Entwicklung in Java

Kubernetes ist zum De-facto-Standard geworden, um containerisierte Cloud-native Anwendungen zu betreiben. Es ist Ressourcen-basiert und kann mit benutzerdefinierten Ressourcen erweitert werden. Um diese zu betreiben und zu verwalten, werden sogenannte Kubernetes-Operatoren eingesetzt. Die meisten Operatoren werden in Golang entwickelt, weil Kubernetes in dieser Programmiersprache geschrieben ist. Daneben können aber auch Java und andere Programmiersprachen verwendet werden.

Autor:

Alexander Ryndin ist bei der Consol Software GmbH als Software-Engineer tätig. Er beschäftigt sich mit den Themen rund um Microservices, Java EE, MicroProfile, Kubernetes, AWS und Cloud-native Anwendungen.

Firmen-Webseite https://labs.consol.de/
GitHub https://github.com/progaddict/
E-Mail-Adresse Alexander.Ryndin@consol.de

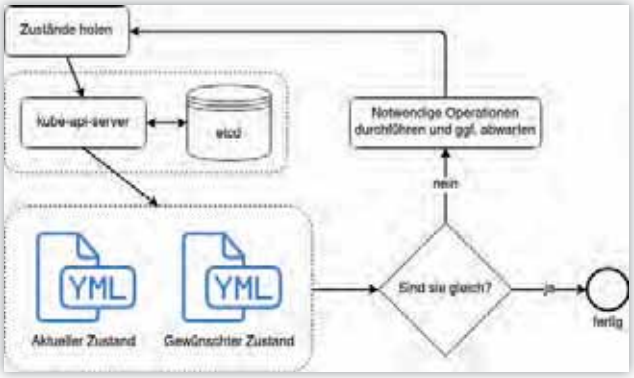


Vereinfachte Architektur von Kubernetes (Abb. 1)

Kernfunktionen von Kubernetes sind Verteilung, Scheduling und Verwaltung der containerisierten Workloads. Es gibt mehrere Ansätze, um ein System mit diesen Kernfunktionen zu implementieren. Die Entwickler von Kubernetes haben dafür folgende grundlegende Prinzipien gewählt:

1. Ressourcenorientierte Architektur mit RESTful-Interfaces
2. Deklarative Beschreibung der Ressourcen
3. Controller-Muster (Controller-Pattern)
4. Flache Hierarchien und flexible Verknüpfung von Ressourcen
5. Erweiterbarkeit

Das erste Prinzip gestaltet die allgemeine Architektur des Systems. Alle Kubernetes-Objekte, wie zum Beispiel Pods, Services, Deploy-ments sind Ressourcen, die via HTTP erzeugt, gelesen, modifiziert und gelöscht werden können. Die Komponente `kube-api-server` **(Abb. 1)** ist dafür verantwortlich. Kubernetes hat also eine ressourcenorientierte Architektur, die über REST-Schnittstellen verwaltet wird.
(Abb. 1) zeigt auch, dass die Kubernetes-Komponenten in zwei Gruppen aufgeteilt sind. Die erste Gruppe bilden die Komponenten, die den gesamten Kubernetes-Cluster steuern. Hierbei handelt es sich um die Verwaltungsebene (Control-Plane). Die zweite Gruppe bilden die Infrastrukturknoten `IK1` bis `IK5`, welche die tatsächliche Arbeit leisten beziehungsweise die Container ausführen. Die Komponente `kube-scheduler` entscheidet dabei, welcher Infrastrukturknoten welchen Pod ausführt **(Abb. 1)**. In[1] sind die Architektur und ihre Komponenten detaillierter erklärt.



Controller-Muster (Abb. 2)

Das zweite Prinzip weist auf die Definition des gewünschten Zustandes der Ressourcen hin. Kubernetes soll selbst die notwendigen Operationen, wie beispielsweise das Hoch- oder Runterskalieren durchführen, um diesen gewünschten Zustand zu erreichen. Die Ressourcen und deren gewünschter Zustand sind typischerweise in YAML definiert. Diese Information wird dann in der Komponente `etcd` gespeichert **(Abb. 1)**. `etcd` ist eine verteilte Key-Value-Datenbank[2], die zusätzlich die Führerselektion (Leader-Election) anbietet. Die notwendigen Operationen werden von Kubernetes-Controllern durchgeführt. Diese Controller werden von der Komponente `kube-controller-manager` verwaltet **(Abb. 1)**. Sie sind nach dem Controller-Muster gebaut, welches das dritte Prinzip darstellt **(Abb. 2)**.
Das Controller-Muster[4] kann als eine Schleife betrachtet werden, in welcher der Controller die Operationen typischerweise mittels Kubernetes-API-Aufrufen durchführt, bis die Ressource den gewünschten Zustand erreicht hat. Wenn beispielsweise ein Deployment auf `Null` heruntersklaiert wird, sieht der entsprechende Replication-Controller, dass in dem gewünschten Zustand keine Pods laufen sollen und löscht diese.

Das vierte Prinzip manifestiert sich durch Labels, durch die Ressourcen einander flexibel referenzieren können. Alle Kubernetes-Ressourcen können mit `key=value` Labels versehen werden. Eine Menge dieser Labels definiert somit eine Gruppe von Ressourcen, die mit diesen Labels markiert sind. Ähnlich definieren Hashtags auf Twitter eine Gruppe von Tweets, die diese Hashtags beinhalten. Genauso wie Hashtags auf Tweets verweisen, referenzieren Labels Kubernetes-Ressourcen. Ohne Labels müssten Deployments, Services und Pods in einer hierarchischen Baumstruktur definiert werden, um sie miteinander zu verknüpfen. Das wäre allerdings rigide und stark miteinander

verkoppelt. Labels bieten somit einen universalen Mechanismus an, um Ressourcen zu referenzieren. So werden beispielsweise Ressourcen als zugehörig zu anderen Ressourcen markiert und solche rigiden hierarchischen Strukturen vermieden.

Das fünfte Prinzip ist für jedes moderne System sehr wichtig, denn es gibt immer Situationen, in denen das System mit einer spezifischen und vorher unerwarteten Funktion erweitert werden soll. Da es nicht möglich ist, ein Werkzeug für alle Zwecke zu entwickeln, sollte es erweiterbar sein. Kubernetes bietet zwei Erweiterungsmechanismen an: Custom-Resources und den API-Aggregation-Layer.

Diese fünf Prinzipien haben Kubernetes zur Plattform[5] zum Bauen von Plattformen gemacht. Es ist sehr flexibel, erweiterbar und läuft sowohl On-Premises als auch in einer Public-Cloud.

CustomResourceDefinition und Operator-Duo

Custom-Resources[6] sind erst in Kubernetes 1.7 erschienen, vorher gab es Third-Party-Resource, die in der Version deprecated wurde. Eine Custom-Resource ist eine benutzerdefinierte Ressource, die von Kubernetes via bekannte REST-API beziehungsweise per Kommandozeilen-Tool kubectl verwaltet werden kann. Um eine Custom-Resource in Kubernetes zu integrieren, sind `CustomResourceDefinition` und ein Operator notwendig.

`CustomResourceDefinition` ist selbst eine Kubernetes-Ressource, welche die benutzerdefinierte Ressource beschreibt (Meta-Ressource) und dem System bekannt macht. Sie kann als ein YAML-Dokument betrachtet werden. Es definiert, welche Felder die benutzerdefinierte Ressource hat. Dies ist notwendig, damit Kubernetes die neue Ressource zum Beispiel während der Erzeugung validieren kann. Das veranschaulicht folgendes Beispiel einer Dog-Custom-Resource **(Listing 1)**.

(Listing 1)

```
apiVersion: myk8sextensions.com/v1alpha1
kind: Dog
metadata:
  name: kr
spec:
  breed: 'German Shepherd'
  callName: 'Kommissar Rex'
  # laut dem 4. Prinzip
  # sollen rigide hierarchische Strukturen
  # mithilfe von Labels vermieden werden
  caughtSelector:
    type: criminal
  workedWithSelector:
    type: inspector
```

Wenn die Ressource via `kubectl apply -f kr.yml` erzeugt wird, muss Kubernetes wissen, dass diese Ressource die Felder `breed`, `callName`, `caughtSelector` und `workedWithSelector` hat. Genau das definiert die entsprechende `CustomResourceDefinition`. Im Folgenden wird beschrieben, wie eine `CustomResourceDefinition` genau konstruiert werden muss.

Die Custom-Resource wird nach der Validierung in `etcd` gespeichert. Ohne Weiteres kann Kubernetes jedoch nicht wissen, wie es die Ressource verarbeiten soll. Sollen beispielsweise Pods nach der Erzeugung der Ressource gestartet werden oder soll ein Sidecar-Container in einige bestehende Pods injiziert werden? An diesem Punkt kommt der Operator ins Spiel. Ein Operator ist nichts anderes als eine in Kubernetes laufende Anwendung, wie zum Beispiel ein Pod. Diese legt fest, was gemacht werden soll, wenn eine Ressource erzeugt, modifiziert und gelöscht wird. Der Operator ruft dabei typischerweise die Kubernetes-API auf, um beispielsweise von der Custom-Resource abhängige Kubernetes-Ressourcen zu erzeugen, zu modifizieren oder zu löschen. Technisch gesehen sind hier keine Grenzen gesetzt. Der Operator kann beispielsweise auch einen externen Dienst aufrufen.
Eine `CustomResourceDefinition` und ein Operator bilden somit ein Duo. Die erste beschreibt, laut dem 2. Prinzip, den gewünschten Zustand der entsprechenden Custom-Resource. Der zweite sorgt dafür, dass dieser Zustand erreicht wird, indem er dem Controller-Muster aus dem 3. Prinzip folgt. Wie genau der Zustand aus der Definition der Ressource interpretiert wird und wie er erreicht werden kann, beziehungsweise jene Semantik, die hinter der Definition steht, wird dem Entwickler bei der Implementierung überlassen.

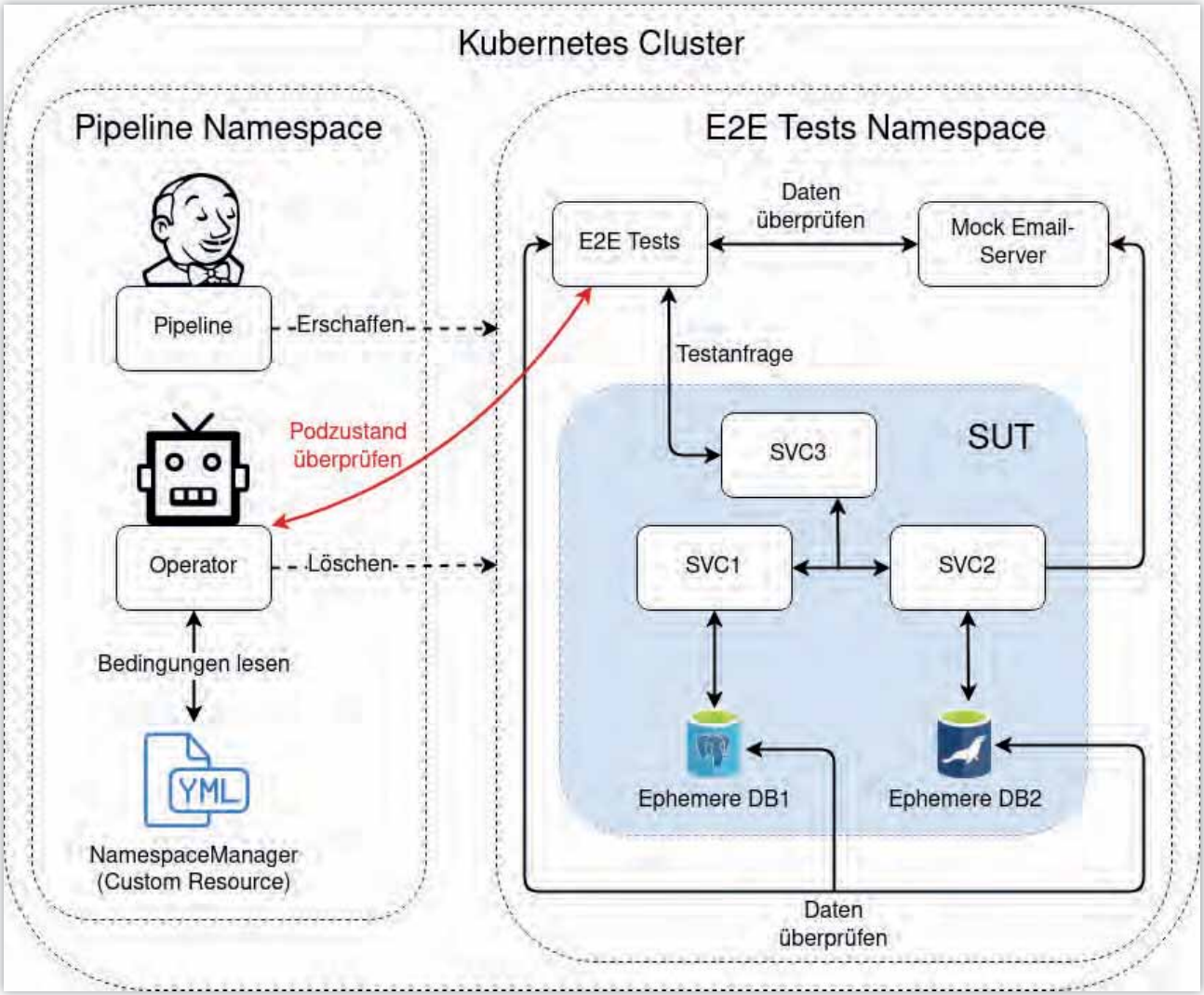
Ohne Operator macht eine Custom-Resource keinen Sinn, denn es geschieht in diesem Fall gar nichts, wenn die Ressource erzeugt, modifiziert oder gelöscht wird. Ein Operator ohne Custom-Resource kann wiederum durchaus nützlich sein. Ein Beispiel für einen derartigen Operator wäre ein Slack-Bot:

Dieser würde das Entwicklerteam in Slack notifizieren, wenn dieser feststellt, dass alle Pods in dem vom Operator eingesetzten Namespace fertig sind, beziehungsweise wenn die Pods nicht in der Phase `Pending` oder `Running` vorkommen[7]. Der Operator würde in diesem Fall nur die üblichen Kubernetes-Ressourcen, nämlich die Pods beobachten und daher keine speziellen benutzerdefinierten Ressourcen benötigen.

K8S-Namespace-Manager

K8S-Namespace-Manager[8] ist ein Beispieloperator, der als Proof-of-Concept dient, damit Operatoren auch in Java geschrieben werden können. Im Weiteren zeigen wir durch den Entwicklungsprozess dieses Operators auf, welche Schritte benötigt werden, um einen Kubernetes-Operator in Java zu entwickeln.

Im ersten Schritt wird das Problem definiert. Es geht in diesem Fall um die Verwaltung von Namespaces, die für E2E-Tests



K8S-Namespace-Manager-Deployment (Abb. 3)

(End-To-End) benutzt werden und deshalb relativ flüchtig beziehungsweise kurzlebig sind (Abb. 3). Die Pipeline deployt in einen separaten Namespace, das `SUT` (`System-Under-Test`), welches Microservice-basiert ist (`SVC1`, `SVC2` und `SVC3`) und Persistenz hat (die ephemeren `DB1` und `DB2`), zusammen mit den E2E-Tests und Mocks (zum Beispiel `E-Mail-Server-Mock`).

Die Tests laufen im Pod `E2E-Tests`. Sie schicken Anfragen gegen `SUT` und überprüfen die Antwort, die `SUT` zurückgibt. Sie können auch Mocks und Daten in den von dem `SUT` benutzten Datenbanken überprüfen. Wenn die Tests erfolgreich beendet sind, soll das `E2E-Tests` Namespace aufgeräumt beziehungsweise gelöscht werden. Das ist die Funktion des K8S-Namespace-Operators. Der Operator überprüft den Zustand des Pods, in dem die Tests laufen und sobald der Pod nicht mehr läuft und keine Fehler aufzeigt, soll der Namespace gelöscht werden.

Bei der Definition des Problems können die Fähigkeiten des Operators noch ein wenig erweitert werden. Es gibt folgende typische Situationen, wie das in (Abb. 3) beschriebene `E2E-Tests` Szenario:

- `E2E-Tests` haben mit einem Fehler geendet, der Pod ist zum Beispiel in der Phase `Failed`[9] . Damit die Entwickler diesen analysieren können, soll der Operator nichts unternehmen beziehungsweise nichts löschen oder herunterskalieren.
- `E2E-Tests` waren erfolgreich. Der E2E-Namespace soll gelöscht werden.
- `E2E-Tests` waren erfolgreich. Deployments in E2E-Namespace sollen auf `Null` herunterskaliert werden, aber der Namespace kann bestehen bleiben.
- `E2E-Tests` waren erfolgreich. Der Operator soll einen externen Dienst via HTTP aufrufen (WebHook-Funktion).
- Der `E2E-Namespace` darf nicht mehr als beispielsweise drei Stunden existieren. Wenn diese TTL (Time-To-Live) abgelaufen ist, soll er gelöscht werden – unabhängig von dem Status der Tests.
- Es soll möglich sein, mehrere Bedingungen miteinander zu kombinieren. Es soll zum Beispiel möglich sein, gleichzeitig die TTL und die Bedingung des Testergebnisses zu definieren.

Der Implementierungsansatz des K8S-Namespace-Manager baut darauf, dass die hier gelisteten Bedingungen (das Ergebnis

der Tests und Time-To-Live) und Aktionen (Löschen, Runterskalieren oder der WebHook-Aufruf), in einer Custom-Resource beinhaltet sind. Der in Java implementierte Operator soll sie via Kubernetes-API lesen **(Abb. 3)** und anwenden.

Die Custom-Resource nennt sich `NamespaceManager` und beschreibt deklarativ, welche Aktion angewendet werden soll, wenn eine Bedingung erfüllt ist. Dafür dient das Feld `policies` im Beispiel in **(Listing 2)**.

(Listing 2)

```
apiVersion: k8s.consol.de/v1beta1
kind: NamespaceManager
metadata:
  name: example-01
  namespace: example-01
spec:
  policies:
    - name: delete namespace when TTL elapses
      condition:
        type: TTL
        params:
          TTLSeconds: 40
      action:
        type: DELETE
```

Das Beispiel befindet sich auf GitHub[10]. `Policies` ist eine Liste, die Paare von Bedingung-Aktion beinhaltet. Wenn der Operator feststellt, dass die Bedingung erfüllt ist, soll er die entsprechende Aktion anwenden. Bedingungen und Aktionen können frei kombiniert werden. Möglich sind folgende Bedingungen:

- `TTL` – wenn einfach eine bestimmte Frist in Sekunden abläuft. Die Frist beginnt ab der Erschaffung des Namespace, Kubernetes hat das `creationTimestamp` erzeugte Feld dafür.
- `PODS_SUCCEED` – wenn die durch eingegebene Labels spezifizierten Pods die Phase `Succeeded` erreichen oder wenn keine dieser Pods gefunden werden, falls sie vom Kubernetes automatisch aufgeräumt werden.

Folgende Aktionen werden unterstützt:

- `DELETE` – der Namespace wird gelöscht.
- `SCALE_DOWN` – die durch eingegebene Labels spezifizierten Deployments und StatefulSets werden auf `Null` herunterskaliert.
- `WEBHOOK` – der eingegebene WebHook wird via `HTTP POST` aufgerufen und die Payload beinhaltet dabei die Information über den Namespace und die eingeschlagene Policy.

Weitere Beispiele finden sich auf GitHub[11]. Der Name des Namespace wird entweder durch das Feld `namespace` definiert oder falls nicht vorhanden, wird vom Kubernetes Feld `metadata` der Name der `NamespaceManager Ressource` verwendet. Bevor die `NamespaceManager` Ressourcen via `kubectl apply -f` erzeugt werden können, soll Kubernetes beigebracht werden, welche Struktur diese haben. Dafür dient die Meta-Ressource `CustomResourceDefinition` **(Listing 3)**.

(Listing 3)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: namespacemanagers.k8s.consol.de
spec:
  group: k8s.consol.de
  versions:
    - name: v1beta1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                # ... einige Felder ...
                policies:
                  type: array
                  minItems: 1
                  items:
                    type: object
                    properties:
                      name:
                        type: string
                        pattern: '^\w[\w ]{1,90}\w$'
                      condition:
                        type: object
                        properties:
                          type:
                            type: string
                            enum:
                              - TTL
                              - PODS_SUCCEED
                          # ... weitere Felder
                          # des verschachteltes Objektes...
  scope: Namespaced
  names:
    plural: namespacemanagers
    singular: namespacemanager
    kind: NamespaceManager
    shortNames:
      - nsman
```

Die volle `CustomResourceDefinition` findet sich auf GitHub[12]. **(Listing 3)** zeigt wichtige Teile der Definition. Eine davon ist die API-Gruppe-Definition `k8s.consol.de`. Alle Kubernetes-Ressourcen sind in API-Gruppen unterteilt. Die `CustomResourceDefinitions` gehören beispielsweise zur API-Gruppe `apiextensions.k8s.io`. Custom-Resources sollen diese Art und Weise der Organisation der Ressourcen auch berücksichtigen.

Der zweite Teil ist die Definition der Version `v1beta1` und Struktur der Custom-Resource `openAPIV3Schema`. Die Definition der Struktur soll der OpenAPI-Spezifikation[13] folgen. Dank dieser Spezifikation kann auch eine Validierung erfolgen. Beispielsweise soll der Name einer Policy den in JavaScript regulären Ausdruck

`'^\w[\w ]{1,90}\w$'` erfüllen. Praktisch bedeutet dies, dass der Name mit einem alphanumerischen Zeichen beginnen muss, nur alphanumerische Zeichen und Leerzeichen beinhalten und nicht länger als 90 Zeichen sein darf [14].

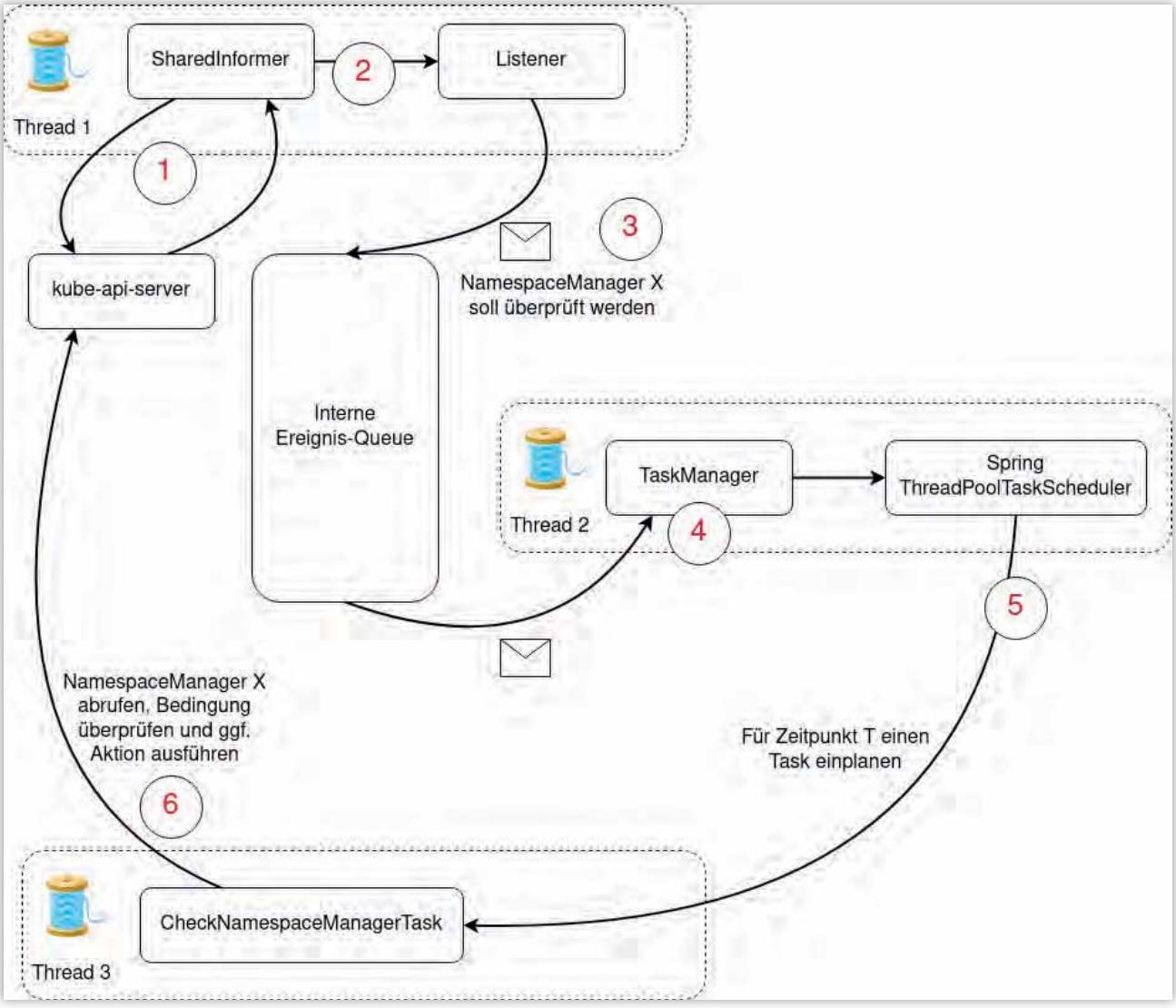
Der dritte Teil beschreibt, welchen Geltungsbereich (Scope) die neue Custom-Resource haben soll. Sie kann entweder für den Namespace oder für den Cluster gelten.

Schließlich definiert der vierte Teil, wie die Custom-Resources benannt und wie sie von kubectl aus referenziert werden können. In **(Listing 3)** erhält die neue Custom-Resource den Namen `NamespaceManager` durch `kind: NamespaceManager`. Sie kann via kubectl folgendermaßen referenziert werden:

- kubectl get `namespacemanager` foo
- kubectl get `namespacemanagers`
- kubectl get `nsman` bar

Nun fehlt nur noch der Operator, der diese neuen Custom-Resources lesen und deren Policies anwenden muss. Die Implementierung findet sich auf GitHub[8] bzw. in der Kubernetes-Dokumentation[9]. Sie ist eine Spring-Boot-Anwendung, die auf dem Fabric8-Kubernetes-Client[15] basiert. Diese Bibliothek stellt eine Portierung des offiziellen Golang-Kubernetes-Client dar und beinhaltet einige wichtige API-relevante Konzepte:

- Es gibt sogenannte `SharedInformers`[16] . Das sind Klassen, die in einer unendlichen Schleife regelmäßig Kubernetes-API abrufen, um Kubernetes-Ressourcen zu beobachten.
- `SharedInformers` ermöglichen es, Listeners für einen bestimmten Ressourcentyp anzumelden, wie zum Beispiel Pods oder Deployments. Auch für Custom-Resources wie die `NamespaceManager` Ressource ist dies möglich. Der Listener wird notifiziert, sobald eine Ressource erzeugt, modifiziert oder gelöscht wird.



Funktionsweise des Operators (Abb. 4)

Die Verarbeitung der `NamespaceManagers` ist somit ereignisgetrieben, denn der Listener wird aufgerufen, wenn eine Erzeugung, Modifikation oder das Löschen eines `NamespaceManagers` erfolgt und der Operator darauf agieren soll.

Die Godoc-Dokumentation des Kubernetes-Client empfiehlt, das Empfangen der Ereignisse und die Verarbeitung mit einer Ereignis-Queue zu entkoppeln[17]. In der Java-Welt wäre dies beispielsweise eine BlockingQueue. Bei der Verarbeitung können zwar die Bedingungen eines `NamespaceManagers` überprüft werden, sie müssen jedoch entweder regelmäßig erfolgen, wie im Fall der `PODS_SUCCEED` Bedingung oder zu einem bestimmten Zeitpunkt geschehen, wie im Fall der `TTL` Bedingung. Die Logik ist daher so implementiert, dass bei der Verarbeitung der Zeitpunkt berechnet wird, wann die Überprüfung der Bedingungen stattfinden soll. Die tatsächliche Überprüfung passiert dann asynchron in einem Task **(Abb. 4)**.

Der Betrieb des Operators geschieht in sechs Schritten **(Abb. 4)**. Im ersten Schritt holt der `SharedInformer` sich über die Kubernetes-API die Ressourcen. Im zweiten Schritt stellt der `Informer` fest, ob es Änderungen beziehungsweise neue, modifizierte oder gelöschte Ressourcen gibt. Wenn ja, wird die entsprechende Methode des `Listeners` aufgerufen. Diese zwei Schritte sind in der `Fabric8-Client` Bibliothek implementiert. Der `Listener` muss allerdings vom Entwickler implementiert werden.

Im dritten Schritt, wenn der `Listener` aufgerufen wird, wird ein Ereignis in die `Interne Ereignis-Queue` beziehungsweise in die BlockingQueue gespeichert. Es wird dann in einem separaten Thread vom `TaskManager` im vierten Schritt gelesen. Der `TaskManager` berechnet den nächsten Zeitpunkt `T`, wann die Bedingungen des `NamespaceManagers` überprüft werden sollen. Für diesen Zeitpunkt `T` wird via Spring `ThreadPoolTaskScheduler` ein Task im fünften Schritt eingeplant. Wenn der Zeitpunkt kommt, wird der entsprechende Task in einem separaten Thread im sechsten Schritt aufgerufen. Dieser Task holt sich den `NamespaceManager` über die Kubernetes-API, liest dessen Bedingungen, überprüft sie und wendet gegebenenfalls die entsprechenden Aktionen an.

Fazit und Ausblick:

Kubernetes basiert auf fünf grundlegenden Prinzipien, welche der Plattform zu einer branchenweiten Akzeptanz verholfen und sie zu einer flexiblen Plattform zum Bauen von Plattformen gemacht hat. Eines dieser Prinzipien ist die Erweiterbarkeit, die Operatoren sind deren konkrete Implementierung. Dank der Kubernetes-REST-API können Operatoren in einer beliebigen Programmiersprache entwickelt werden, die das HTTP-Protokoll unterstützt. Java ist hierfür sehr gut geeignet.

Operatoren können nach ihrem Automatisierungsgrad klassifiziert werden[18]: von Basic-Install (niedrig) bis zu Autopilot (sehr hoch). Diese Klassifizierung impliziert, dass die autonome Software, die zur Autopilotstufe gehört und sich selbst installieren, überwachen und upgraden kann, die Zukunft von Kubernetes als Plattform darstellt.

Es ist möglich, dass Kubernetes künftig als Medium dienen wird, um Cloud-native Software mittels der Operatoren zu distribuieren und zu verwalten. Eine ähnliche Rolle hat das heutige Betriebssystem, auf dem Anwendungen installiert, betrieben und upgedatet werden. Der erste Schritt in diese Richtung wurde bereits vollzogen: Im OperatorHub[19] kann jeder seinen Operator veröffentlichen, der zum Beispiel ein Produkt ins Kubernetes-Cluster installiert und verwaltet. Der Hub ähnelt somit den Paket-Repositories, die heutzutage für klassische Software genutzt werden.

Quellen:

1 Etcd: https://bit.ly/etcd-e1
2 Kubernetes-Komponenten: https://bit.ly/components-k2
3 Etcd: https://bit.ly/etcd-e1
4 Kubernetes-Controller: https://bit.ly/controller-c4
5 Kubernetes als Plattform zum Bauen von Plattformen: https://bit.ly/platform-p5
6 Custom-Resources: https://bit.ly/resources-r6
7 Kubernetes 1.7 Ankündigung: https://bit.ly/blog-b7
8 K8S-Namepace-Manager: https://bit.ly/K8S-k8
9 Pod Lebenszyklus: https://bit.ly/lifecycle-p9
10 Beispiel 1: https://bit.ly/example-e1
11 Weitere Beispiele: https://bit.ly/examples-e11
12 CustomResourceDefinition für NamespaceManager: https://bit.ly/definition-c12
13 OpenAPI-Spezifikation: https://bit.ly/specification-s13
14 OpenAPI-Pattern-Typ: https://bit.ly/types-t14
15 Fabric8-Kubernetes-Client: https://bit.ly/client-c15
16 SharedInformer-Godoc: https://bit.ly/SharedInformer-s16
17 DeltaFIFO-Godoc: https://bit.ly/DeltaFIFO-d17
18 Operator-Capability-Model: https://bit.ly/blog-o18
19 OperatorHub: https://bit.ly/operatorhub-o19

#JAVAPRO #CI #CD #Testing #Quality

CI/CD-Pipelines per Console überwachen

Bei den zunehmend schneller werdenden Entwicklungszyklen ist es sinnvoll und praktisch, den Stand der ausgeführten CI/CD-Pipelines stets im Überblick zu behalten, anstatt nur System-Mails zu erhalten. Die jeweils mitgelieferten Web-Clients ermöglichen einen direkten Blick in die Systeme. Ein weit besserer und auf die eigenen Bedürfnisse zugeschnittener Überblick kann über die jeweiligen Kommandozeilen-Tools aufgebaut werden. In diesem Artikel wird beschrieben, warum dies sinnvoll sein kann und wie dies am Beispiel von GitLab-CI und Concourse-CI aufgebaut wird.

Softwaresysteme werden heutzutage meist in einer Ansammlung von kleineren Teams von je ca. 7 Personen pro Teilprojekt entwickelt. Früher und manchmal auch heute noch sind mehr als 50 Entwickler für eine monolithische Anwendung aktiv. Damals wie heute sind diese Teams zwar nicht ausschließlich mit Softwareentwicklern bestückt, jedoch stellen sie meist mehr als die Hälfte des Teams dar. Das bedeutet, dass mehrere Personen unterschiedlichste Stellen einer Software bearbeiten. Da diese Teilarbeiten zu einem Softwareprodukt zusammengeführt werden müssen, ist es sinnvoll die Qualität und Gesamtfunktionalität der Software stets sicherzustellen – und das an zentraler Stelle, möglichst nahe an der Versionsverwaltung. Dies ist eine wesentliche Grundidee von Continuous-Integration (CI). Integrationsschwierigkeiten sind von vielfältiger Natur, verdeutlicht durch einige Beispiele:

- Qualitative Aussagen: Kompiliert der Code? Sind die automatisierten Tests erfolgreich durchgelaufen?
- Quantitative Aussagen: Sind alle Methoden der öffentlichen API dokumentiert? Wie verhält sich die Testabdeckung über die Zeit?

Je früher Integrationsschwierigkeiten erkannt werden und je genauer sie sich einer Änderung zuordnen lassen, desto geringer ist der Aufwand, diese zu beheben. Damit zahlt man auf drei Prinzipien des agilen Manifests[1] ein:

- “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”
- “Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.”
- “Working software is the primary measure of progress.”

Insbesondere in den heutigen Projektlandschaften, in denen die Projekte in Unterprojekte von kleinen Teams aufgebaut und in eine Microservice-artige Struktur gewandelt werden, ist die Funktionalität der einzelnen Services wichtig. Microservices werden unabhängig voneinander in das vorhandene System

Autor:

Kai Schmidt ist freiberuflicher Softwareentwickler und -Architekt. Zuvor war er bei den IT-Beratungsunternehmen .msg systems ag und Capgemini angestellt und in seiner über 15-jährigen Projekterfahrung größtenteils in Java- und C#-Projekten in den Bereichen Logistik, Flugzeugbau sowie Handel tätig.

Blog-URL / Firmen-Webseite: www.kai-schmidt.hamburg
Twitter: https://twitter.com/electronickai/
XING: http://www.xing.to/kaischmidt
GitHub: https://github.com/electronickai/
Emailadresse: mail@kai-schmidt.hamburg

automatisiert produktiv geschaltet. Mehrere Deployments innerhalb weniger Stunden oder Minuten sind an der Tagesordnung. In diesen Umgebungen hat man keine Möglichkeiten, einen eingefrorenen Stand des Gesamtsystems testen zu können. Die Funktionsweise zwischen den Systemen werden vermehrt über Consumer-Driven-Contract-Tests (CDC-Test)[2] sichergestellt, anstatt diese gegen die abhängigen Instanzen laufen zu lassen. In diesen Systemen bieten die CI-Pipelines ein weit größeres Spektrum als die Ausführung automatisierter Tests. Sie bieten zusätzlich die Paketierung (JAR-Packages oder eher Container-Images) sowie weitere Testphasen, zudem die automatisierte (stufenweise) Auslieferung in das Produktionssystem und gegebenenfalls ein automatisiertes Zurückrollen im Falle von festgestellten Fehlverhalten. Wird ein System stetig und automatisiert ausgerollt, spricht man von Continuous-Delivery (CD) bzw. einer Continuous-Delivery-Pipeline.

Speziell in der Java-Welt verbreitete sich Hudson, das mittlerweile mehr als 15 Jahre auf dem Buckel hat. Weit bekannter wurde der im Jahr 2011 aus Hudson hervorgegangene Fork Jenkins. Ein deutlich erkennbarer Trend in den letzten Jahren ist die Möglichkeit, den jeweiligen Aufbau der Pipeline(s) gemeinsam mit dem Code einchecken zu können (Configure-as-Code). Mit jedem Stand, den man aus dem Repository betrachtet, ist somit klar wie er getestet oder in die Produktion ausgeliefert wurde. Dies wurde für Jenkins beispielsweise erst über ein Plugin namens Pipeline-DSL ermöglicht und ist mittlerweile, wie derzeit bei Jenkins-X[3], in den meisten modernen CI/CD-Systemen Standard. Zuvor hatte man sich die Pipelines in dem vom Repository losgelösten CI/CD-System über die UI konfiguriert. Obwohl es mittlerweile selbstverständlich geworden ist, diese Systeme über Code zu konfigurieren, wird die UI weiterhin dazu genutzt, den aktuellen Stand der Pipelines im Überblick zu behalten.

Warum der Überblick über die Kommandozeile?

Ständiges, automatisiertes und kurzfristiges Feedback über die Qualität des Codes und der gemachten Anpassungen sind für Entwickler eine ungemein große Hilfe während der Arbeit. Dennoch sollte man als Entwickler durch diese, nach jedem Push parallel angestoßene Tätigkeit nicht aufgehalten werden. Wer nach einem Commit erstmal die Pipeline betrachtet und ansieht, ob alles weiterhin wie geplant funktioniert, gerät aus dem Flow. Der Wechsel zum Browser stellt einen Context-Switch dar, der schnell von der eigentlichen Arbeit und von den nächsten Tätigkeiten ablenkt, die sich im Kopf angesammelt haben. Sinnvoller ist es den parallelen Ablauf zwar im Auge zu behalten, aber gleichzeitig direkt die nächsten Tätigkeiten fortführen zu können. Möglich erscheint dies über Mails, Messenger oder einem großen Teammonitor an der Wand. Diese Prozesswerkzeuge können den Flow sowohl unterstützen als auch stören. In der jetzigen Zeit des Homeoffice sind Teammonitore eher einsam. Wer einen schnellen oder detaillierteren Einblick in den Zustand der Pipeline werfen möchte, ist mit der Kommandozeile gut beraten.

Die Kommandozeile kann sich als wahrer Schatz an vorgefertigten IDE-Plugins herausstellen. In allen gängigen IDEs kann man sich das CLI (Command-Line-Interface) einblenden lassen. Wir sind gewohnt darüber Datenbanken zu starten, mit dem (Git-)Repository zu hantieren, Code in der JShell auszuführen oder einen Gradle-Build anzustoßen. Ein Blick in die Pipelines zum gerade hochgeladenen Code-Stand gewährt sich über diesen Weg erstaunlicherweise kaum jemand. An den zwei unterschiedlich aufgebauten CI-Systemen GitLab-CI und Concourse-CI wird im Folgenden gezeigt, wie einfach dies möglich ist.

GitLab-CI

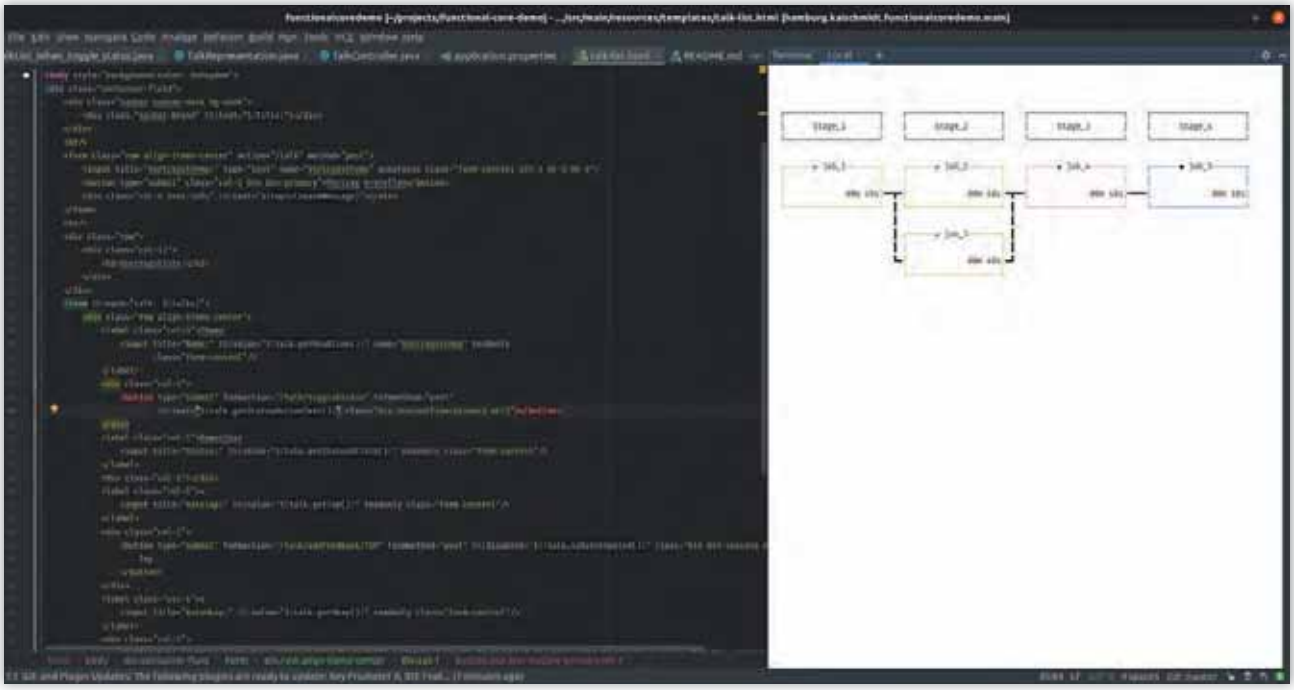
GitLab[4] führt einige Funktionen ein, die über Git hinausgehen. Diese umfassen die Planungsphase neuer Feature-Ideen bis hin zum Monitoring und stehen teilweise ausschließlich in kommerziellen Versionen zur Verfügung. Sehr bekannte und in der kostenfreien Version verfügbaren Funktionen von GitLab stellen das Erstellen von Issues und Versehen derselbigen mit Labeln und Meilensteinen sowie die in GitLab eingebaute CI-Funktionalität dar. Wer sich basierend auf diesem Artikel einen lokalen GitLab-Server inklusive zugehöriger Runner aufsetzen möchte, sei auf die zusammengestellte Anleitung[5] verwiesen. Das Aufsetzen einer lokalen GitLab-CI-Umgebung muss gegebenenfalls auf die persönlichen Bedürfnisse angepasst werden und ist damit trotz Docker teilweise etwas umständlich. GitLab selbst bietet kein CLI zu dem verfügbaren Funktionsumfang. Der wesentliche Anteil der Funktionen wird glücklicherweise über das Open-Source-Kommandozeilen-Tool Lab[6] abgedeckt.

GitLab-CI auf der Kommandozeile

Das Shell-Skript auf der Projektseite von Lab vereinheitlicht die Installation auf allen gängigen Linux-Systemen. Windows-Veteranen können Lab über den Paketmanager Scoop installieren. MacOS-Nutzer installieren per Homebrew.

Lab identifiziert sich über einen API-Key, der in GitLab hinterlegt wird. Nach dem erstmaligen Start von Lab wird man durch die notwendige Konfiguration geleitet, sodass dies kein großes Hindernis darstellen sollte. 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 praktisch, wenn man sich einen neuen Token generieren möchte, Konfigurationen für verschiedene GitLab-Hosts vorhalten oder die gleiche Datei auf weiteren Rechnern verwenden möchte.

Nach der Einrichtung der CI-Konfiguration über die Datei `.gitlab-ci.yml` und dem Anstarten einer Pipeline über ein Git-Push kann über den Aufruf von `lab ci view` der aktuelle beziehungsweise letzte Pipeline-Lauf des derzeit aktiven Branches angezeigt werden. Über `lab ci create` lässt sich ein neuer Pipeline-Lauf manuell anstarten.



Die Pipeline in der Konsole einer IDE - hier IntelliJ-IDEA (Abb. 1)

Farbe	Symbol	Bedeutung	Taste	Aktion
Schwarz		noch nicht gestartet	/	start bzw. (r)estart
Gelb	●	manuell anzustarten	p	start bzw (p)lay
Blau	●	laufend / aktiv	c	abbrechen / (c)ancel
Grün	✓	erfolgreich	t	Anzeige der Logs
Rot	✖	fehlgeschlagen	T	Logs in die Konsole

Übersicht der UI von Lab-CI-View (Tabelle 1)

Angenehm wäre ein IDE-Plugin, das einen vergleichbaren Einblick in die laufende Pipeline gewährt. Wenn überhaupt bieten IDE-Plugins lediglich Links auf die CI-Seiten als Absprungpunkte in den Browser. Wie bereits angedeutet, lässt sich Lab praktischerweise in der Konsole der IDE einblenden (Abb. 1).

Jeder Job der Pipeline erhält ein Kästchen, in dem sowohl der Job-Name als auch die verstrichene Zeit für den Job dargestellt werden. Über Farbcodes und Symbole lassen sich deren Status entnehmen (Tabelle 1). Die Cursor-Tasten dienen zur Auswahl eines Jobs. Welcher der Jobs gerade den Fokus erhalten hat, wird

Beispielhaftes Log eines Jobs (Abb. 2)

über eine doppelte Umrahmung der Box dargestellt. Auf diesen lassen sich unterschiedliche Aktionen ausführen (Tabelle 1) und beispielsweise die Log-Ausgaben des Jobs anzeigen (Abb. 2).

Für weniger interaktive Ansichten bietet Lab zwei weitere Möglichkeiten. Zum einen gewährt `lab ci status` einen Überblick über den Status der jeweiligen Jobs (Listing 1). Zum anderen liefert `lab ci trace` die Logs eines einzelnen Jobs. Hierüber lässt sich ein Docker-Tag herauskopieren oder der fehlgeschlagene Job analysieren, zum Beispiel für den Job `failing_parallel_rider` mit dem Aufruf `lab ci trace origin failing_parallel_rider`. Dies ist eine Alternative zur Auswahl des Jobs über `lab ci view` und anschließenden Tastendruck T. Die resultierende Ausgabe ist vergleichbar mit (Abb. 2).

(Listing 1 - Der aktuelle Status einer Pipeline mit Lab-CI-Status)

```
Stage:             Name                          - Status
synchronous_stage: single_rider                  - success
parallel_stage:    parallel_rider1               - success
parallel_stage:    parallel_rider2               - success
parallel_stage:    failing_parallel_rider        - failed
manual_stage:      manual_rider                  - manual
```

Dass GitLab den Fokus eher auf die Grundfunktionalität und die UI legt, erkennt man daran, dass die CLI-Unterstützung in einem unabhängigen Open-Source-Projekt entstanden ist. Ähnlich ist dies bei Bamboo, dessen CLI-Unterstützung durch Bob-Swift, einer Marke von Appfire, entwickelt wird. Als konträres Beispiel lässt Concourse Interaktionen fast ausschließlich über die Kommandozeile zu.

Concourse

Anders als GitLab konzentriert sich Concourse[7] ausschließlich auf die Aufgaben eines CI/CD-Systems. Es startet nach den konfigurierten Regeln die notwendigen Jobs und zeigt deren Ergebnisse anschaulich an - nicht mehr, aber auch nicht weniger. Concourse hat seine Stärken in der Visualisierung der Job-Läufe **(Abb. 3)** und sichert aufgrund der Container-Infrastruktur eine hohe Isoliertheit und eine hohe Zustandslosigkeit der ausführenden Einheiten. Den Ansatz Configure-as-Code nimmt Concourse sehr ernst. Wirklich alles wird über Konfigurationsdateien definiert. Die UI bietet, abgesehen von der Möglichkeit Job-Läufe manuell zu triggern und Jobs und Pipelines einzufrieren, ausschließlich lesende Zugriffe. Die zugehörigen UIs sind sehr ansprechend gestaltet und bieten selbst bei aufwändigen Pipeline-Strukturen einen guten Überblick. Auf einem für jeden im Team sichtbaren Monitor kann der Stand der letzten Job-Läufe jederzeit leicht verfolgt werden. Für die Zugriffe auf entfernte Systeme, wie beispielsweise auf das Git-Repository oder S3-Buckets, Credential-Stores oder ähnliches, gibt es eine breite Auswahl an sogenannten Resources. Hierüber können leicht die wesentlichen Parameter konfiguriert werden. Diese Resources werden zusätzlich genutzt, um einzelne Jobs miteinander zu verbinden.

Beispiel einer Pipeline-Sicht in Concourse (Abb. 3)

Ein weiterer wesentlicher Unterschied ist, dass Concourse im Gegensatz zu GitLab nicht im Vorfeld eines Pipeline-Laufs die auszuführenden Schritte ermittelt und diese auch nicht anzeigt. Die Jobs starten dynamisch auf Änderungen der konfigurierten Resources. Hat ein Job ein Ergebnis produziert, das für einen weiteren Job als Input dient, läuft dieser nachfolgende Job eigenständig los.

Hierdurch sieht man allerdings nicht direkt, mit welchem ursprünglichen Push das zustande gekommene Ergebnis zusammenhängt. So passiert es manchmal, dass Anhand der Start- und Laufzeiten der Jobs versucht wird, die darunterliegende Pipeline-Struktur für sich zu rekonstruieren, um den im Falle eines Fehlers verursachenden Commit zu identifizieren. Zusätzlich kann es passieren, dass mehrere Git-Pushes von verschiedenen Entwicklern in einem Lauf zusammengeführt werden, weil der Trigger für Git alle 60 Sekunden auf das Repository pollt. Im Gegensatz zu GitLab ist eine lokale Version von Concourse vergleichsweise schnell aufgesetzt. Die Installation ist auf der Webseite erklärt. Zum Zeitpunkt der Artikelentstehung genügt das Herunterladen und Starten der vorgefertigten `docker-compose` Datei **(Listing 2)**.

(Listing 2 - Installation von Concourse)

```
wget https://concourse-ci.org/docker-compose.yml
docker-compose up -d
```

Über einen Browser auf `localhost:8080` kann man sich nun mit den Credentials `test/test` auf der Oberfläche einwählen.

Concourse auf der Kommandozeile

Da die UI von Concourse keine schreibenden Zugriffe auf das System zulässt und die Konfigurationsdatei nicht automatisiert über eine Namenskonvention einer im Repository befindlichen Datei geladen wird, ist für den Einsatz von Concourse das mitgelieferte Kommandozeilen-Tool fly unerlässlich. Es lässt sich in der UI jederzeit über das Icon unten rechts für das in Verwendung befindliche Betriebssystem herunterladen und in den Pfad legen. Der Login auf der Kommandozeile kann mit **fl**y `-t local login -c http://localhost:8080 -u test -p test` erfolgen. Nun sind viele weitere Kommandos über die Kommandozeile möglich. Für den Aufbau, das Einspielen der Pipeline-Konfigurationen und für die Anwendung der im Folgenden genannten Befehle sei auf das Tutorial[8] verwiesen, das ebenso von Concourse selbst verlinkt wird.

Das Absetzen eines Kommandos benötigt immer das Target, das beim Login über `-t` definiert wurde. Dies erschwert Fehleingaben, macht den Umgang allerdings recht gesprächig. Insbesondere wenn man genau auf einem Target bestimmten Jobs und Pipelines arbeitet, wird man in der Praxis dazu übergehen, Aliase für häufig verwendete Kommandos anzulegen. Aus Gründen der Übersicht wird im weiteren Verlauf auf diese Parameter verzichtet.

Alle auf dem Target ausgeführten Builds und deren Jobs lassen sich über **fl**y `Builds` auflisten **(Listing 3)**. Eine Sicht, die man in der GitLab-UI ausschließlich auf Projektebene dargestellt bekommt und auch mit Lab nicht möglich ist, da Lab ausschließlich Informationen über den aktuellsten Pipeline-Lauf des jeweiligen Branches einholt. Für Entwickler, die auf wenigen Projekten arbeiten, ist dies verzichtbar. Möchte man jedoch beispielsweise als Operator viele Pipelines mitsamt Historie im Blick haben, ist diese Ansicht Gold wert. Über die Parameter `-p` und `-j` lässt sich auf eine bestimmte Pipeline bzw. einen bestimmten Job filtern.

(Listing 3 - Ausgabe von fly-Builds)

```
id  pipeline/job            build     status              start
        …
3   playground/play         1         succeeded 2020-11-
20@06:43:19+0100…
2   hello/hello-world       2         succeeded 2020-11-
16@09:22:47+0100…
1   hello/hello-world       1         errored             2020-
11-16@09:21:57+0100         …

… end              duration  team
… 2020-11-20@06:43:25+0100             6s                  main
… 2020-11-16@09:22:50+0100             3s                  main
… 2020-11-16@09:22:15+0100             18s                 main
```

Einen Einblick in die Log-Ausgabe erhält man mit fly-Watch unter Angabe des jeweiligen Jobnamens (Listing 4).

(Listing 4 - Ausgabe von fly-Watch)

```
initializing
selected worker: 648239a108d2
running echo Hello, world!
Hello, world!
succeeded
```

Sollte keine Ressource für den Start der Pipeline sorgen, beispielsweise Git über einen erfolgten Git-Push, lässt sich ein Job über **fly** `trigger-job` manuell starten. Dies funktioniert auch für Jobs, die bereits automatisiert gestartet wurden und beispielsweise nicht zum gewünschten Ergebnis führten. Eine vergleichbar benutzerfreundliche und interaktive Sicht wie über die UI oder `lab ci view` gibt es leider nicht. Dennoch hat fly eine bunte Funktionspalette. Über das bereits vorgestellte fly-Builds lassen sich die jeweils interessanten Aspekte der Darstellung leicht herausfiltern und als Eingabe für weitere fly-Befehle nutzen. Zur Anregung hier Beispiele von Bash-Befehlen:

(Listing 5 - Auszug eines erstellten Docker-Tags)

```
fly -t local watch -j /hello/dockerize | grep "Successfully
tagged"
```

Ausgaben von `docker build` sind gesprächig. Die Zeile mit dem Tag wird daher über `grep` herausgefiltert. Hierüber lässt sich der dahinter ausgegebene Docker-Tag leicht kopieren. Mit Lab ist eine ähnliche Herangehensweise über den Befehl `lab ci trace` möglich.

(Listing 6 - Log-Ausgabe des letzten fehlgeschlagenen Jobs der hello Pipeline)

```
fly -t local Builds -p hello | awk '$4=="errored" {system("fly -t
local watch -j" $2);exit;}'
```

Mit `awk` können die Werte der jeweiligen Spalten über deren Index zugegriffen werden. Wenn in der Statusspalte (`$4`) der Eintrag `"errored"` steht, sollen die Log-Ausgaben des Jobs, der in der Spalte `pipeline/job` (`$2`) steht, ausgegeben werden. Das `exit` am Schluss lässt die Verarbeitung abbrechen und die weiteren Jobs mit dem Status `"errored"` werden nicht mehr berücksichtigt. Mit Lab ist dies nur indirekt über `lab ci view` möglich.

(Listing 7 – Erneutes Antriggern des letzten fehlgeschlagenen Jobs der hello Pipeline mit gleichzeitiger Log-Ausgabe)

```
fly -t local Builds -p hello | awk '$4=="errored" {system("fly -t
local tj -w -j" $2);exit;}'
```

`tj` ist eine Kurzform für `trigger-job`. Das Flag `-w` sorgt dafür, dass das Log des neu gestarteten Jobs direkt angezeigt wird. Mit Lab ist dies nur indirekt über `lab ci view` möglich.

(Listing 8 - Anzeige aller Builds nach einem bestimmten Datum - hier der 19.11.2020)

```
fly -t local Builds | tr '@' ' ' | awk '$5 > "2020-11-19"'
```

Über das `tr` Kommando nutzt man das `@` Zeichen der Ausgabe von **fly** `Builds`, um das Datum für die Weiterverarbeitung mit `awk` als eigene Spalte zu trennen. Mit Lab ist dies derzeit nicht möglich.

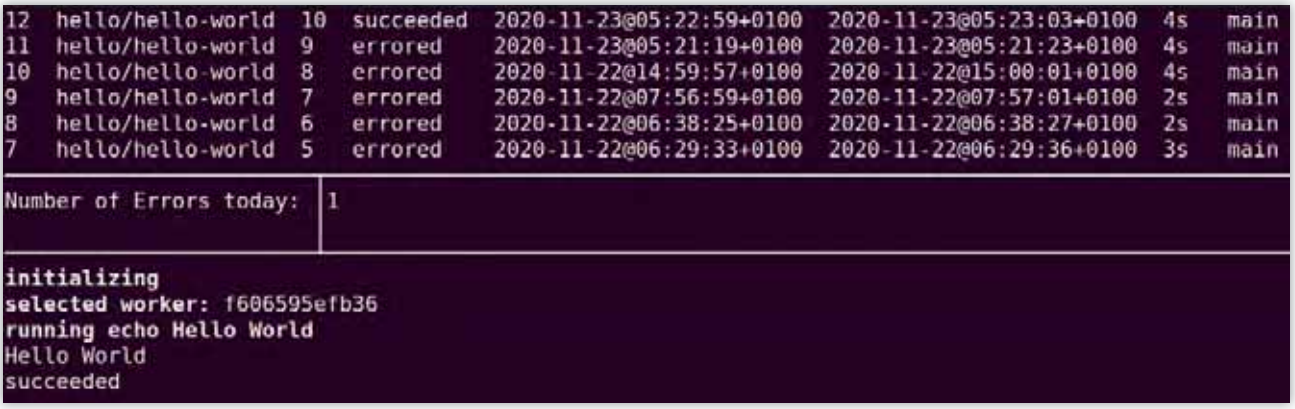
(Listing 9 - Anzahl der fehlgeschlagenen Jobs am heutigen Tag)

```
fly -t local Builds -p hello | tr '@' ' ' | awk 'BEGIN {count=0}
$5 == today && $4=="errored" {count++}
END {print count}' today=$(date +"%Y-%m-%d")
```

Das aktuelle Datum wird im Format `yyyy-MM-dd` an `awk` über die Variable `today` übergeben, um dieses mit dem Datum der Spalte `start` vergleichen zu können. Über die `BEGIN` Regel wird eine Variable `count` initialisiert, die genutzt wird um die heute fehlerhaften Läufe zu zählen. Nach der Verarbeitung der Ausgabe von fly-Builds wird die Anzahl über die `END` Regel ausgegeben. Dieser Befehl hat als Einzeiler eine nicht zu verleugnende Komplexität, soll aber zeigen, wie dynamisch sich mit den hier vorgestellten Befehlen Auswertungen erstellen lassen. Mit Lab ist dies derzeit nicht möglich.

Das personalisierte CI-Cockpit

Häufig ist man an einer Übersicht interessiert, die den Zustand der derzeit aktiven Branches (eventuell ein Feature-Branch) und vielleicht gleichzeitig noch einen weiteren Branch (wie z. B. `dev` oder `master`) anzeigen soll. Oder man ist speziell an den Konsolenausgaben einzelner Jobs der Pipeline stärker interessiert als an anderen. An Jobs, die Tests laufen lassen oder bestimmte



Cockpit-Ansicht für eine Pipeline (Abb. 4)

Docker-Images erstellen, deren Tags man sich kopieren möchte. Egal welches CI-System man einsetzt, man muss sich häufig umständlich durch die Branches und Jobs der Web-UIs klicken. Die wahre Kraft der CI-Kommandozeilenbefehle entwickelt sich in Kombination Terminal-Multiplexern bzw. Konsolenemulatoren. Hierüber lassen sich die jeweils relevanten Sichten oder Pipelines leicht in getrennte Fensterbereiche oder weiteren Tabs zusammenstellen. (Abb. 4) zeigt ein Beispiel mit den letzten 6 Läufen einer Pipeline, die Anzahl der Fehler am heutigen Tag und die Log-Ausgabe des letzten Jobs. Am Tag der Abbildung (`23.11.`) ist bisher ein Fehler aufgetreten. Die weiteren angezeigten Fehler entstanden bereits am `22.11.`

Im Rahmen dieses Artikels werden nur einzelne CLI-Befehle dargestellt. Selbstverständlich lassen sich ebenso ganze Abläufe über Stapelverarbeitungen automatisieren und in das Cockpit einbauen sowie auch Functions und Aliase nutzen. Auf diese Weise erhält man eine Pipeline-Kommandozentrale, die auf die eigenen Bedürfnisse zugeschnitten ist.

Möchte man dieses maßgeschneiderte Cockpit von Teamkollegen mitnutzen oder gar vom gesamten Team pflegen lassen, sollte man darauf achten, dass sich der eingesetzte Multi-plexer entweder direkt über Skripte in der gewünschten Struktur (Anzeigebereiche und die darin ausgeführten Befehle) aufbauen lässt oder dass die Konfiguration exportiert und erneut eingebunden werden kann. Dabei sollten die jeweiligen Definitionen menschenlesbar sowie leicht anpassbar sein und ihren Platz im eingesetzten Versionskontrollsystem finden. Bleiben die Übersichten in der IDE-Konsole überschaubar, kann man sich diese in die IDE einbinden. Die IDEs bieten meist eine Möglichkeit, die verwendete Konsole bzw. Multiplexer zu konfigurieren, oder man startet vom IDE-Terminal das Script, das den Multiplexer aufbaut.

Fazit:

Die hier dargestellten CI-Systeme sind zwei Beispiele aus vielen, wie beispielsweise Circle-CI, Travis-CI, Drone, GoCD, Teamcity und Bamboo. Eine ausgereifte CLI-Unterstützung ist wie ein Multi-Plattform-Plugin für alle IDEs, die den Entwickler helfen können im Flow zu bleiben und den Entwicklungsprozess zu unterstützen. Insbesondere dann, wenn man kein passendes IDE-Plugin findet das die eigenen Bedürfnisse erfüllt. Mit Terminal-Multiplexern bzw. -Emulatoren wie tmux (Linux) oder ConEmu (Windows) lassen sich effizient nutzbare Cockpits erstellen und deren Konfiguration über das Repository mit den anderen Teamkollegen teilen. An zwei Beispielen wurde dargestellt, dass sich die über die Kommandozeile aufgebauten Übersichten aufgrund der inneren Architektur und unterschiedlichen Konzepte sehr unterscheiden können.

Daher ist es empfehlenswert, die CLI-Unterstützung der CI-Systeme für die eigenen Bedürfnisse zu beleuchten und eine Umgebung aufzusetzen, über die man schnell einen umfassenden Blick des aktuellen CI-Status erhalten kann. Diese Möglichkeiten können in der Auswahl eines CI-Systems einen interessanten Entscheidungsfaktor darstellen. Es ist stets wichtig den Überblick über die automatisierten Prozesse zu behalten. Zudem sollte man möglichst einfach in diese Automatisierungen eingreifen können, falls etwas anders läuft wie ursprünglich geplant. Automatisierte Tests und gut aufgesetzte Pipelines sind wesentliche Werkzeuge für hohe Softwarequalität und ein nicht zu unterschätzendes Bindeglied auf dem Weg zu Continuous-Delivery.

Quellen:

1 Manifest: https://bit.ly/3vETAuj
2 Consumer-Driven-Contracts: http://bit.ly/38WTuEy
3 Jenkins-X: https://bit.ly/3eUsklo
4 GitLab: http://bit.ly/3cM36mO
5 GitLab: http://bit.ly/3eZfvGy
6 Lab: https://bit.ly/3s1NPEB
7 Concourse: https://bit.ly/3lDuSWE
8 Tutorial: http://bit.ly/3lwmU1d

#JAVAPRO #DomainDrivenDesign #UML

Domänen-Modelle mit Tools

Domain-Driven-Design zu betreiben läuft fast zwangsläufig auf geschäftsspezifische Ausdrucksformen hinaus, für die es keine vorgefertigten Tools gibt. Die Folge sind zuweilen Wegwerfmodelle auf Papier, die sich nicht mit dem Geschäft weiterentwickeln. Dabei sollte die Digitalisierung von Arbeitsmaterialien für IT-Spezialisten selbstverständlich sein. Am Beispiel der offenen Notation für Event-Storming soll sich zeigen, was das Customizing moderner Modellierungs-Tools hier leisten kann.

Eines der Kernziele von Domain-Driven-Design ist die Definition einer gemeinsamen Sprache und Ausdrucksweise innerhalb eines Geschäfts. Sie soll für beteiligte Geschäftsleute wie IT-Kräfte gleichermaßen geeignet sein, was ebenso Potential wie Herausforderungen birgt. Es erscheint zu Beginn eines Projekts viel bequemer, sich nicht zu sehr mit der Sprach- und Ausdruckweise der anderen beschäftigen zu müssen. Doch getrennte Sprachwelten führen zu getrennten Denksilos, zu tiefgreifenden Missverständnissen und zu ineffizienter Arbeit. Eine gemeinsame Ausdrucksform muss also her. Neben einem gemeinsamen Vokabular im eigentlichen sprachlichen Sinne geht es dabei um eine Form von Symbolik, die es erlaubt, Zusammenhänge des Geschäfts sichtbar und im wahrsten Sinne des Wortes begreifbar zu machen. Ganz egal, ob es sich dabei um Prozessabläufe, Dokumentenstrukturen oder Mondphasen handelt. Was wirklich wichtig ist, entscheidet das jeweilige Geschäft. Die Symbolik soll allen Beteiligten einen Verständnisgewinn bringen und ihnen erlauben, gemeinsam das Geschäft und die dahinter liegende Software zu entwerfen und sukzessive weiterzuentwickeln. So etwas geht meistens nicht in rein textueller Form, weil erfahrungsbedingt, das menschliche Gehirn bildliche Darstellungen besser verarbeitet als Texte. Eine zumindest halbgrafische Ausdrucksform wäre wünschenswert, aber bitte ohne Einstiegshürde, sondern im Gegenteil mit motivierendem Aufforderungscharakter.

Autor:

Baris Güldali ist Managing-Consultant bei S&N CQM GmbH. Als Trainer für Practitioner in Agile-Quality (PAQ) und Agile-Coach begleitet er agile Teams und unterstützt diese bei der Erreichung ihrer Sprintziele. Er organisiert Community-Events mit Themenschwerpunkten Agilität und Softwarequalität.

Jan Lessner ist Senior-Consultant und Softwarearchitekt bei S&N Invent GmbH und im Projektbereich für Bilanzanalyse tätig. Er engagiert sich in verschiedenen Open-Source-Projekten für ultraschlanke Enterprise-Lösungen, ist Buchautor und leidenschaftlicher Voll-Nerd und Clean-Coder. UML und DDD begleiten seine Arbeit seit über 10 Jahren.

Event-Storming als Tool-fremde Notation

Geringe Einstiegshürde heißt, klein anfangen und dann schrittweise bedarfsgerecht erweitern. Alberto Brandolini hat unter dem Begriff Event-Storming eine Methode entwickelt, die diesen Grundgedanken konsequent adressiert[1]. Es beginnt schon damit, dass ein Event-Storming-Workshop gar nicht so benannt wird, um nicht den Eindruck zu erwecken, die Beteiligten müssten einem strengen Korsett von Ausdrucksformen folgen. Alles beginnt mit den Events als einziges Ausdrucksmittel, die als orangene Klebezettel auf einer großen, leeren Wand oder einem Fußboden gesammelt und anschließend in zeitliche Reihenfolgende gebracht werden **(Abb. 1)**. Keine Stühle, keine Tische vor der Modellwand – alle Beteiligten sollen gleichberechtigt Zugang haben und mitgestalten. Events benennen ein Substantiv und ein Verb in Vergangenheitsform, z.B. Bestellung storniert - um sich gleichzeitig den Geschäftsobjekten wie den Geschäftsprozessen zu nähern. Welche Ausdrucksmittel später noch dazu kommen, entscheidet sich im Entstehungs- und gemeinsamen Lernprozess. Es ist ein bisschen wie Fußball: Ein Ball, ein leerer Platz und ein paar Leute mit Spaß an der Sache und los gehts! Man

kann sehr lange Fußballspielen, bevor man die Abseitsregel und Torlinientechnik braucht. Event-Storming birgt grade einmal 8 vorgedachte Modellierungselemente und überlässt alles Weitere den Bedürfnissen der jeweiligen Domäne. Entscheidend ist eine Grundregel, in der sich die Philosophie von Domain-Driven-Design widerspiegelt: „Only talk about visible things“[2] . Taucht ein Begriff auf, muss er Teil des sichtbaren Modells werden, auf den jeder Beteiligte mit dem Finger zeigen kann, wenn er davon spricht.

Event-Storming (Abb. 1)

Event-Storming dient hier nur beispielhaft als eine Methodik kollaborativer Modellentwicklung. Es gibt auch andere Methoden wie z.B. Example-Mapping[3] oder Feature-Mapping[4], die für unterschiedliche Voraussetzungen unterschiedlich geeignet sind. Aber all diesen Methoden ist ein grundlegendes Dilemma gemein:

- Sie starten mit sehr einfachen Notationsformen und zeigen sich offen für domänenspezifische Ergänzungen, um den Denkraum der Beteiligten nicht zu beschneiden.
- Begriffe, über die man redet, müssen im Modell sichtbar werden.
- Es kann aus Prinzip keine vorgefertigten Tools, für die im Vorhinein noch nicht bekannten Ausdrucksformen geben.
- Weil dem so ist, haben kollaborativ entwickelte Modelle die Tendenz im Zustand reiner Papiermodelle zu verbleiben statt eine digitale Form zu bekommen, in der sie effizient weiterentwickelt werden können[5]. Abfotografierte Tafelbilder sind keine Lösung.
- Das ursprüngliche Modell ist in kürzester Zeit nicht mehr aktuell. Die Beteiligten sind gezwungen genau das zu machen, was Domänenmodelle vermeiden sollen, nämlich über Dinge und Zusammenhänge zu reden, die im Modell nicht sichtbar sind.

Die Frage, was mit dem Modell nach einem Modellierungs-Workshop passiert, wird oft unbefriedigend oder mit Schulterzucken beantwortet[6]. Aber in jedem iterativen Entwicklungsprozess, insbesondere in agilen Vorgehensweisen sollte klar sein, dass ein Modell nur dann eine hilfreiche Arbeitsgrundlage bildet, wenn es sich weiterentwickelt, wie das Geschäft und der Code auch. Und vor allem in der IT-Branche sollte völlig klar sein, dass etwas faul ist, wenn wir unsere eigenen Arbeitsmittel nicht digitalisieren.

Tool-gestützte Domänenmodelle

Wie digitalisiert man also die Modelle, die in Workshops in haptischer Form entstehen? Mit ein paar Kompromissen ist ein gangbarer Weg das Customizing bestehender Modellierungs-Tools. Allen voran seien hier ausgereifte UML-Tools genannt, die schon basierend auf den Eigenschaften von UML selbst eine gewisse Offenheit für nennenswertes Customizing haben müssen. UML (Unified-Modelling-Language) hat in der IT fast jeder schon gehört und umso erstaunlicher ist die wenig verbreitete Nutzung. UML gilt als kompliziert und unattraktiv und infolgedessen als zu wenig nützlich, was aus Sicht erfahrener UML-Nutzer weitgehend auf Missverständnissen beruht.

Die wahrgenommene Kompliziertheit geht im Wesentlichen auf hohe Einstiegshürden der Tools zurück. UML als Ausdrucksform ist in seinen Grundzügen sehr simpel, die Bedienung der Tools hingegen für Anfänger sperrig. Das ist aber nicht schlimm, da nicht alle in der Entwicklung eines Geschäfts involvierten Personen auch die Tools bedienen müssen. Beim Erlernen der Ausdrucksformen kommt es auf das Vorgehen und eine behutsame Einführung anhand domänenbedeutsamer Begriffe an. Das verhält sich nicht anders als bei salonfähigen Ausdrucksformen wie Event-Storming, deren Stärke aus einer gruppenfähigen Methodik erwächst. Fußballspielen fängt auch nicht mit der Abseitsregel an.

Die wahrgenommene Unattraktivität liegt vor allem an billigen Tools. Kein Open-Source-Tool und keine kostenlose Variante der kommerziellen Tools bringen die Stärken von UML in brauchbarem Maße zum Fliegen und jegliches Domain-Driven-Design erst recht nicht. Das zeigt sich schon bei kleinen Dingen wie der Frage, ob das Tool Zeilenumbrüche in zusammengesetzten Begriffen verträgt, um auch längere Begriffe vernünftig in Diagrammen darstellen zu können. Aber vor allem müssen die Tools die in UML verankerten Erweiterungsmechanismen wie das Meta-Modell, UML-Profile, Stereotypen und Tagged-Values konsequent unterstützen. Ohne finanzielle Investition gibt es all dies leider nicht. Die folgenden Beispiele basieren auf den Tools `MagicDraw`[7] und Enterprise-Architect[8]. Beides sind hochentwickelte, nicht ganz billige UML-Boliden, die sich aber kostenlos für begrenzte Zeit evaluieren lassen.

Reicht denn nicht Power-Point?

Die Verführung ist groß: Einstiegshürde fast Null, Ausdrucksfreiheit fast unbegrenzt. Aber leider ist das ein Holzweg, denn die besondere Kraft von DDD liegt ja grade darin, eine formale Notation zu (er)finden, deren Bestandteile sich auf stringente Weise in die Software abbilden. Jedes Radio basiert auf einem Schaltplan, jedes Haus auf einer Bauzeichnung und Software braucht ein Pendant zu solchen halbgrafischen und dennoch formalen Konstruktionsunterlagen. In dem Artikel „Domain-Driven-Design fördert Software Craftsmanship“[9] ist nachzulesen, wie das mit den Bordmitteln der UML und entsprechend ausgereiften Tools machbar ist. Hier geht es jetzt noch einen Schritt weiter.

Beispielmodell für Event-Storming (Abb. 2)

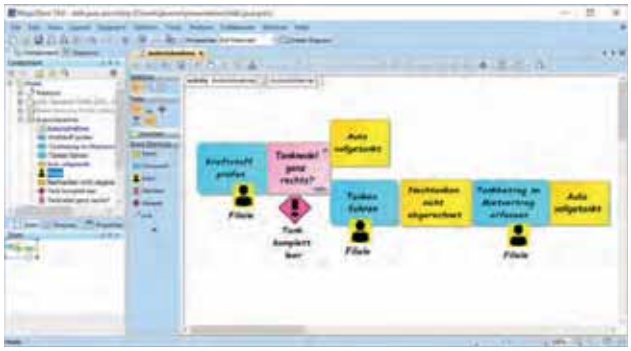
Das Beispiel

Ein kleines Beispiel aus einem Event-Storming-Modell soll im Folgenden als Prüfstein dienen. Es stammt aus der Domäne einer Autovermietung und stellt einen Teilprozess bei der Rücknahme eines Mietwagens dar. Mietwagen sollen von den Kunden vollgetankt abgegeben werden, was es aber durch die Mitarbeiter zu kontrollieren gilt. Ist dies nicht der Fall, muss nachgetankt werden und Aufwand und Kraftstoffkosten kommen zusätzlich auf die Rechnung. In (Abb. 2) ist dieser Vorgang über eine Abfolge von Events und Commands und einer Geschäftsregel dargestellt. Jemand hat noch den Extremfall hinzugefügt, dass der Tank völlig leer ist und der Wagen nicht einmal mehr zur Tankstelle gefahren werden kann. Der Fall verbleibt als sogenannter `Hotspot` im Modell, d.h. er wird vielleicht irgendwann einmal formal ausgearbeitet, aber nicht zum jetzigen Zeitpunkt.
Zeitliche Abfolge wird in Event-Storming durch Anordnung der Klebezettel von links nach rechts ausgedrückt. Wie das Beispiel zeigt, können dieselben Events mehrfach auftauchen – im Beispiel das Event `Auto vollgetankt`. Alberto Brandolini erwähnt diesen Aspekt in seiner weitverbreiteten Session „100.000 Orange Stickies Later“[10] als notwendigen Widerspruch gegenüber Objektorientiertem Design, wo dem DRY-Prinzip und der Vermeidung von Duplikation eine besondere Bedeutung zukommt. Interessanterweise ist das in UML aber kein Widerspruch, denn ein wesentliches Grundkonzept besteht in der Unterscheidung zwischen dem Modell einerseits und dem Diagramm als eine Sicht auf das Modell andererseits. Was wir in (Abb. 2) sehen, ist im Sinne von UML ein Diagramm, und in diesem kann ein und dasselbe Modellelement beliebig oft auftauchen, ohne dass es damit redundant im Modell auftaucht. Hier unterscheiden sich richtige Modellierungswerkzeuge erheblich von Malwerkzeugen.

Event-Storming mit MagicDraw

(Abb. 3) zeigt, wie der in (Abb. 2) mit Klebezetteln modellierte Ablauf im Modellierungs-Tool MagicDraw aussehen kann, wenn das Tool mit Hilfe seiner Customizing-Fähigkeiten um einen Editor für Event-Storming erweitert wurde. Auf der linken Seite sind die Modell-Elemente in einer Explorer-Ansicht zu sehen, auf der rechten Seite ist ein Diagramm zu sehen. Dazwischen befindet sich eine zum Diagramm-Editor gehörende Tool-Palette, um Elemente eines Event-Storming-Modells anzulegen. Will der Anwender keine neuen Elemente anlegen, sondern lediglich bestehende Modellelemente auf dem geöffneten Diagramm platzieren, zieht er diese per Drag&Drop aus dem Modell-Explorer auf das Diagramm.

Auf dem Diagramm sind die Elemente genauso angeordnet wie in der manuellen Modellierung. Das beinhaltet einen wichtigen Punkt bei der Digitalisierung: Nach der Übernahme der Ergebnisse aus einem Event-Storming-Workshop in ein Tool soll der Wiedererkennungseffekt für die Beteiligten so hoch wie möglich sein. Kollaborative Workshops schaffen eine hohe Identifikation der Teilnehmer mit den erarbeiteten Materialien. Diese Energie gilt es mit einzufangen. Der Übertragungsvorgang darf daher nur mit minimalen Verfremdungen verbunden sein, und folglich ist das Customizing der Tools besonders wichtig.



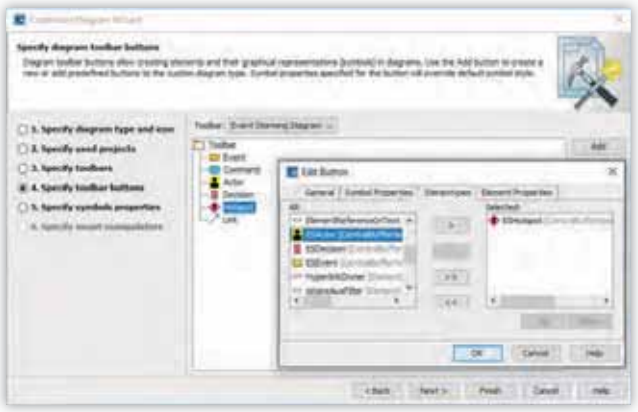
Beispielmodell in MagicDraw (Abb. 3)

Wie kriegt man nun `MagicDraw` dazu, eine Modellierung für Event-Storming zu ermöglichen?
Wesentliche Grundlage für das Customizing sind zunächst einige Eigenschaften von UML selbst, die sich in jedem ausgereiften Tool wiederfinden:

- UML unterscheidet verschiedene Diagrammarten. Da die Darstellungsformen für kollaborative Modellierung in der Regel eher schlicht sind, sonst würde man ja wieder Berührungsängste aufbauen und die Kollaboration wäre dahin, findet man immer eine passende Diagrammart als Grundlage. Wegen des Aspekts der zeitlichen Abfolge und dem Fokus auf Verben beim Event-Storming bietet sich ein Aktivitätsdiagramm als Grundlage an.
- UML erlaubt die Definition sogenannter Stereotypen, die dazu dienen, die in UML vordefinierten Elementarten nach eigenen Vorstellungen weiter zu spezialisieren. UML kennt zum Beispiel eine sogenannte Action als Element von Aktivitätsdiagrammen. Ein Command eines Event-Storming-Modells ließe sich also durch eine UML-Action ausdrücken, die mit einem domänenspezifisch erfundenen Stereotyp `ESCommand` assoziiert wird.
- Stereotypen lassen sich individuell innerhalb eines Modells definieren, aber UML beinhaltet auch ein Konzept sogenannter UML-Profile. In einem UML-Profil lassen sich Definitionen sammeln, die zur Wiederverwendung in verschiedenen Modellen geeignet sind.

Da das Konzept der Stereotypen schon sehr lange besteht, bieten ausgereifte Tools die Möglichkeit, mit Stereotypen verschiedene Darstellungseigenschaften zu verknüpfen. Im Customizing für MagicDraw sind z.B. Schriftgrößen und Fonts, Icons, Hintergrund-, Schrift- und Rahmenfarben und Verschiedenes mehr verknüpfbar. Eine Action mit Stereotyp `ESCommand` erhält beispielsweise einen blauen Hintergrund. MagicDraw unterstützt auch Farbverläufe und Schatten, was der Darstellung einen plastischen Eindruck gibt und noch stärker an Zettel auf einer Wand erinnert.

Nun wäre es unpraktisch, wenn der Anwender immer erst UML-Grundelemente anlegen und einzeln stereotypisieren müsste. In MagicDraw lässt sich dies durch sogenannte Custom-Diagrams automatisieren. Über einen Wizard kann hier ein eigener spezialisierter Diagrammtyp definiert werden, der nur die Platzierung bestimmter Elemente mit vorgegebener Stereotypisierung zulässt. Im Beispiel also ein Event-Storming-Diagramm als spezielles UML-Aktivitätsdiagramm, auf dem nur Events, Commands, Policies usw. zugelassen sind. Für das einfache Platzieren steht dem Anwender des Diagrammtyps später eine entsprechende Toolbar zur Verfügung. Die Custom-Diagrams legen auch fest, welche Elemente mit welchen anderen in Beziehung gesetzt werden dürfen. Tatsächlich sind die Elemente in **(Abb. 3)** nicht nur zeitlich nebeneinander angeordnet, sondern über stereotypisierte Dependency-Links assoziiert. Dass im Diagramm keine entsprechenden Pfeile zu sehen sind, liegt an der Platzierung Kante auf Kante. Zieht man die Elemente ein wenig auseinander, kommen die Links zum Vorschein. Die Verlinkung ist unbedingt sinnvoll, denn in UML entsteht durch Anordnung im Diagramm allein keine Beziehung zwischen den Elementen. Durch die Links bleiben diese Beziehungen dauerhaft bestehen, auch wenn einzelne Diagramme entfernt oder umgestaltet werden.



MagicDraw Custom-Diagram-Wizard (Abb. 4)

Alles in allem lässt sich mit dem Customizing von MagicDraw eine neue Diagrammform ganz passabel ergänzen. Mit Übung und einer konkreten Vorstellung, was eine neue Diagrammform leisten soll, lässt sich eine solche in ein paar Stunden einführen. Für Ungeübte ist der Anfang allerdings mühevoll. Das Customizing ist nur mäßig dokumentiert und die Arbeitszyklen sind sehr lang, weil manche Änderungen der UML-Profile und Diagrammdeskriptoren nur durch Neustart des Tools wirksam werden. Und das dauert. Für großflächige Icons sollte man Vektorgrafiken verwenden, um keine unattraktiven Verpixelungen beim Skalieren der Elemente zu riskieren. Mit der einen oder anderen Unschönheit muss man am Ende dennoch leben. Hier ein paar Beispiele:

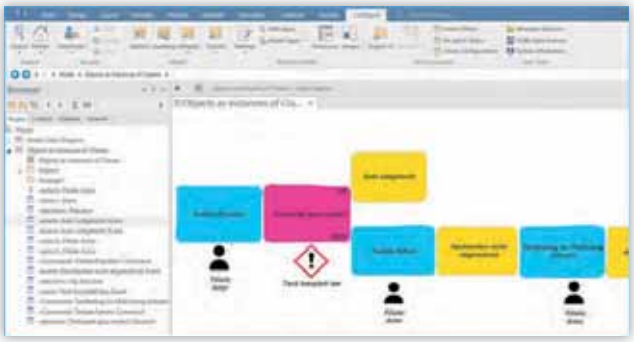
- Man kann nicht explizit festlegen, welches Diagrammelement über oder unter einem anderen liegen soll. Überlappende Platzierungen können daher problematisch sein, siehe `Hotspot` in **(Abb. 3)**.
- Beziehungsprüfungen in den Custom-Diagrams funktionieren nur bedingt.
- Beim Drag&Drop aus dem Modell-Explorer werden nicht alle stilistischen Eigenschaften korrekt initialisiert.

Einige dieser Probleme sind bereits aus Version 16 von `MagicDraw` bekannt und bestehen in der aktuellen Version 19 immer noch. Customizing ist also offensichtlich einer der weniger betretenen Pfade. Der Vollständigkeit halber sei erwähnt, dass über ein umfangreiches Java-API noch wesentlich tiefere Eingriffe in das Tool möglich sind und sich eigene Plug-Ins schreiben lassen. Ratsam ist das zum Zwecke des DDD-Customizings aber nicht. Die Plugins sind an die jeweilige Installation des Tools gekoppelt und nicht an die UML-Profile oder die Modelle. Es entstehen dadurch schnell Versionierungsprobleme. Andere Tools wie Modelio (ehemals Objecteering) haben in der Hinsicht ein ausgefeilteres Konzept.

Event-Storming mit Enterprise-Architect

Auch Enterprise-Architect (EA) von Sparx-Systems bietet Funktionen für Customizing an, mit denen domänenspezifische

Modellierung möglich wird. Diese basieren genauso wie bei MagicDraw auf Stereotypen und deren Sammlung in Form von UML-Profilen. Mittels Stereotypen kann die Bedeutung der Modellelemente und Elementbeziehungen festgelegt werden. **(Abb. 5)** zeigt, wie das Event-Storming-Modell aus der Autovermietung auf Basis eines spezialisierten Objektdiagramms im Enterprise-Architect aussieht. Die Objekte repräsentieren die Modellelemente von Event-Storming und die Objektbeziehungen repräsentieren die zeitliche Anordnung von Modellelementen.

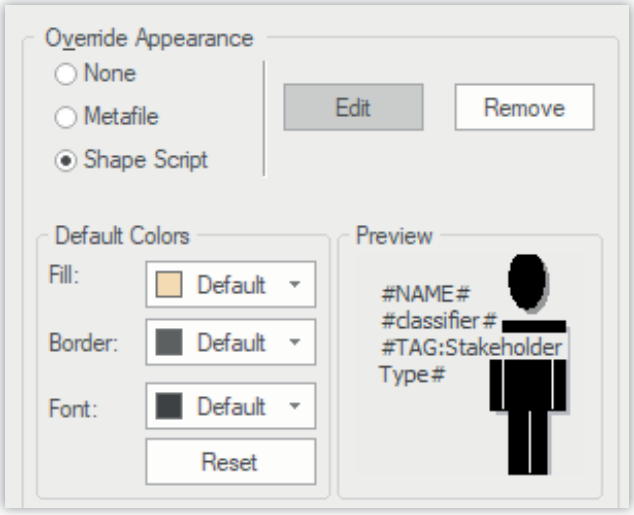


Beispielmodell in Enterprise-Architect (Abb. 5)

EA bietet zwei unterschiedliche Möglichkeiten, um auf der Basis von Stereotypen die Darstellung von Modellelementen zu steuern

- Metafile: mit dieser Option bekommen Modellelemente eine Vektorgrafik zugewiesen, die die konkrete grafische Repräsentation von EA für einen Modelltyp überschreibt. In **(Abb. 5)** ist dieses Verfahren für das Objekt `Filiale` vom Typ `Actor` verwendet worden. Im Gegensatz zu MagicDraw sind Überlappungen von Modellelementen hier kein Problem. Der Anwender kann interaktiv festlegen, welche Elemente im Vorder- und welche im Hintergrund stehen sollen. Eine Kopplung dieser Eigenschaft an Elementtypen ist allerdings nicht möglich.

- Shape-Script: Mit dieser Option kann die Darstellung der Modellelemente programmatisch zum Darstellungszeitpunkt definiert werden. Dabei geht es um verschiede Darstellungseigenschaften wie Anordnung, Farben, Formen und Beschriftungen der Modellelemente, die in einer einfachen Skriptsprache festgelegt werden. **(Abb. 7)** zeigt ein Shape-Script für `Event`, welches als ein Rechteck mit gerundeten Ecken und der Füllfarbe Orange dargestellt wird. Als Beschriftung kommt der Objektname zentriert angeordnet auf das Rechteck. Shape-Scripts bieten auch die Möglichkeit, innerhalb eines Scripts Metafiles einzubinden und diese an die anderen Eigenschaften anzupassen. Dieser Ansatz liegt den Events und Commands in **(Abb. 5)** zugrunde, wobei zu erkennen ist, dass es zu Unregelmäßigkeiten im Rendering der Grafiken kommen kann.



Überschreiben der Darstellung Appearance in EA (Abb. 6)

```
shape main {
  h_align = "center";
  v_align = "center";
  setfillcolor(255,215,0);
  RoundRect(0, 0, 100, 90, 20, 20);
  println("#NAME#");
}
```

Shape-Script für Modellelemente von Typ Event (Abb. 7)

Das relativ einfache Event-Storming-Beispiel lässt sich mittels Vektorgrafiken und Shape-Script recht ordentlich im EA herstellen. Die Darstellung kommt der ursprünglichen Form aus Klebezetteln jedenfalls so nahe, dass ein guter Wiedererkennungseffekt entsteht. Allerdings erfordert das Customizing im EA ebenso tiefe Einblicke in das Tool und Kenntnisse der UML-Modellierung wie bei MagicDraw. Der zeitliche Aufwand für das Customizing liegt bei entsprechender Übung etwa in der gleichen Größenordnung. Auch hier kommt es im Detail zu kleineren Schwierigkeiten, mit denen sich der Anwender arrangieren muss, wie z.B. das oben erwähnte Rendering-Problem. Und auch im EA kann man noch wesentlich tiefer in das Tool eingreifen. Das Stichwort hierfür sind die sogenannten MDG-Technologies.

Wohin die Reise geht

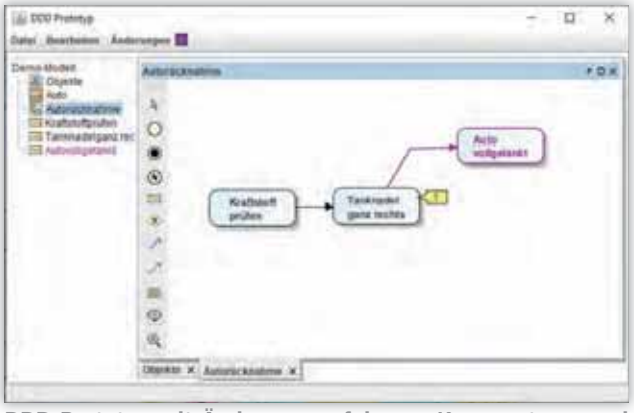
Customizing von Modellierungs-Tools lässt sich definitiv nicht aus dem Ärmel schüttelt. Das Ergebnis ist für den Anwender der Modellierung eine komfortable Sache – der Weg dorthin erscheint allerdings steinig. Und dennoch ist er wirtschaftlich betrachtet attraktiv, wenn man sich in einer Domäne bewegt, in der Domain-Driven-Design ganz generell einen Mehrwert darstellt. Papier- und PowerPoint-Modelle sind dann langfristig keine adäquaten Mittel mehr. Für den Einstieg spielen Tools noch keine Rolle. Beginnen sollte alles mit großen, freien Flächen und

ohne schwierige Formalismen – Hauptsache visuell, wenigstens halbgrafisch und von Kollaboration geprägt, denn da liegt die Magie von DDD. Zeichnet sich aber ab, dass DDD der richtige Weg für die Schaffung von Software in einem Geschäft ist, dann müssen professionelle Tools her, um Modelle zu digitalisieren. Was auch immer einem dabei an Stolpersteinen auffallen mag, die derzeit verfügbaren Tools sind allemal gut genug, um eine domänenspezifische Darstellungsformen beizubringen.
Ein praktischer Tipp: Hat man in seiner Firma bereits ein leistungsfähiges Tool zur Verfügung, dann sollte in den ersten Workshops zur Grundsteinlegung von Domänenmodellen zumindest eine Person teilnehmen, die grob über die Grenzen des Customizings des Tools Bescheid weiß. Das vermeidet die Erfindung von Ausdrucksformen, die hinterher nicht machbar sind. Und außerdem ist empfehlenswert: Keine Angst vor UML! Eigene Notationen sind möglich aber nicht immer ein Muss. In „Domain-Driven-Design fördert Software Craftsmanship“ ist dargestellt, dass moderne Tools auch für äußerliche Attraktivität von Modellen Sorge tragen und allein dadurch Berührungsängste nehmen.

Um die Modellierung am Ende vollends zu einem zentralen Bestandteil des Software-Engineerings zu machen, wie es die Intention von DDD ist, gibt es auf lange Sicht ein paar Knackpunkte mit den heutigen Tools. Sie treten erst nach längerer Zeit des Modellierens zutage, und vielleicht braucht es irgendwann eine ganz neue Generation von Tools, um die DDD-gestützte Arbeit zu optimieren. Hier ein paar Beispiele für Mankos auf den zweiten Blick:

- Im Gegensatz zur IDE eines Entwicklers laufen die Modellierungs-Tools meist nicht permanent, sondern werden bei Bedarf geöffnet. Die meisten Tools starten aber sehr träge, so dass es lange dauert, bis der Anwender sich im Modell etwas anschauen kann. Bild- und HTML-Exports schaffen etwas Abhilfe, aber die sind natürlich ständig veraltet. Schnell startende Tools wären wünschenswert, um flüssige Arbeit zu gewährleisten.
- Ein Modell muss lebendig sein und ständig von einer Iteration zur nächsten angepasst werden. Dafür braucht es eine Änderungsverfolgung, wie z.B. in Microsoft Word, damit die Entwickler erkennen, was sich gegenüber einem vorherigen Stand geändert hat und als Nächstes implementiert werden muss. Hier haben die gängigen Tools nichts Praktikables zu bieten. Gleiches gilt für Kommentierung, um z.B. Review-Ergebnisse festzuhalten.
- Was UML bereits auf Ebene der Modellelemente vorlebt – Redundanzfreiheit im Modell aber mehrfache Nutzung in Diagrammen – braucht DDD auch auf Ebene von Bezeichnern und Texten. Das wird in dem kleinen Beispielmodell in (Abb. 5) für die Rückgabe eines Autos bereits deutlich. Dort taucht ein Event mit dem zusammengesetzten Bezeichner `Auto vollgetankt` auf, wobei `Auto` sicher ein Begriff des zentralen Vokabulars einer Autovermietung ist. Es taucht hier aber nicht als Referenz auf einen Modellbegriff auf, sondern einfach als Buchstabenfolge `A`, `u`, `t` und `o`. Also redundant, ohne eine formelle Möglichkeit die Abhängigkeiten zu verfolgen (außer eine Volltextsuche). Aber dann können wir wieder PowerPoint verwenden statt zu modellieren.

(Abb. 8) zeigt einen Screenshot eines Prototyps, der einige solcher Eigenschaften mitbringt. Bis eine neue Generation solcher DDD-Tools zur Verfügung steht, kann man mit den vorhandenen Tools schon sehr weit kommen: Fußball ohne Torlinientechnik!



DDD-Prototyp mit Änderungsverfolgung, Kommentaren und Referenzen in Bezeichnern (Abb. 8)

Quellen:

1 Brandolini, Alberto (c. 2017). Introducing Event-Storming. Leanpub, 2017
2 https://bit.ly/38O0K5J
3 http://bit.ly/3bZygIe
4 http://bit.ly/3ltBpmM
5 https://bit.ly/38SiIEd
6 https://bit.ly/38SjIrX
7 https://bit.ly/2NCPEJc
8 https://bit.ly/30VrXPy
9 Lessner, Jan; Güldali, Baris. Domain-Driven Design fördert Software Craftsmanship, JavaSPEKTRUM 2/2020
10 https://bit.ly/30TdwLF

#JAVAPRO #Kotlin #iOS #Mobile

Multi-Plattform Entwicklung mit Kotlin/Native

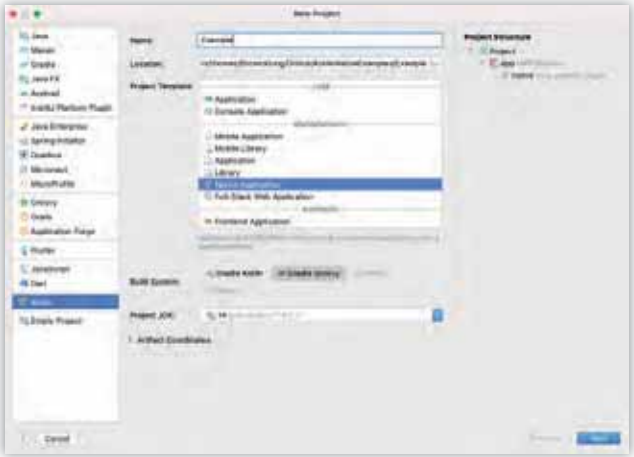
Kotlins Erfolgsgeschichte beginnt vor fast zehn Jahren als Sprache für die JVM. Seit 2019 ist die moderne, funktionale Alternative zu Java erste Wahl bei neuen Android-App-Projekten. Mit Kotlin/Native zielt Erfinder JetBrains auf wichtige Plattformen ohne virtuelle Maschine ab.

Die im Mai 2019 von Google verkündete Entscheidung Java den Rücken zu kehren und bei neuen Android-Apps stattdessen auf Kotlin zu setzen, dürfte kaum jemanden überrascht haben. Die nach einer Insel im Finnischen Meerbusen benannte Sprache wurde schon viel früher (2017) offiziell unterstützt. Kein Wunder, denn Android-Studio basiert auf IntelliJ. Und diese IDE stammt ebenfalls von JetBrains. Außerdem musste sich der Suchmaschinenprimus keine Sorgen machen, Entwickler mit einer Nischensprache zu irritieren, denn Kotlin findet auch bei der Entwicklung von Unternehmensanwendungen und Microservices immer mehr Freunde. Dank eines Transpilers nach JavaScript kann die Sprache auch im Web-Umfeld genutzt werden.

Raus aus der virtuellen Maschine

Bislang außen vor geblieben sind allerdings Plattformen, die eine Ausführung von Code in Laufzeitumgebungen oder virtuellen Maschinen nicht vorsehen. Das ist zum Beispiel unter iOS der Fall. Auch in ihren Ressourcen eingeschränkte Geräte wie der Raspberry-Pi profitieren von einer unmittelbaren Ausführung. Diese Lücken soll Kotlin/Native schließen. Die Hauptbestandteile der Technologie sind ein LLVM-basiertes (Low-Level-Virtual-Machine) Backend für den Kotlin-Compiler sowie native Implementierungen der Standardbibliothek. Aktuell werden Android, iOS und dessen Derivate iPadOS, watchOS und tvOS, macOS sowie Windows, Linux und WebAssembly (wasm32) unterstützt. Damit Kotlin-Code mit nativen Bibliotheken kommunizieren kann, generiert der Compiler neben einer ausführbaren Datei statische oder dynamische Bibliotheken mit Headern für C und C++ sowie ein Apple-Framework für Objective-C- und Swift-Projekte. Vorhandene statische oder dynamische C-Bibliotheken sowie Swift- und Objective-C-Frameworks können mit Kotlin-Code angesprochen werden.

Kotlin/Native wird am einfachsten über Gradle und das Kotlin-Multi-Platform-Plugin angesprochen. Es bietet sich an, hierfür IntelliJ zu verwenden, denn mit dessen Projektassistent **(Abb. 1)**



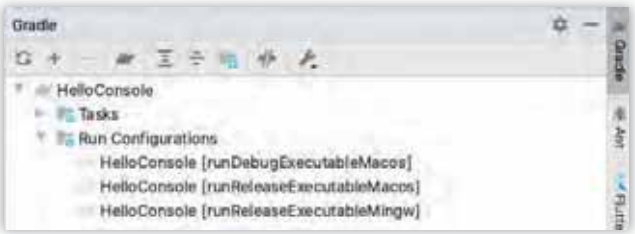
Der Projektassistent von IntelliJ (Abb. 1)

Autor:

Thomas Künneth arbeitet als Principal-Consultant und Head-of-Mobile bei der MATHEMA Software GmbH. Er ist Android-Entwickler der ersten Stunde, regelmäßiger Sprecher auf nationalen und internationalen Konferenzen und Autor zahlreicher Artikel und Bücher zu Java, Eclipse, Android und Mobile-Computing.

lassen sich sehr bequem Kotlin/Native-Projekte erzeugen. Die Plugins für Eclipse und NetBeans können das leider nicht. Das Bauen mit Gradle auf der Kommandozeile ist selbstverständlich problemlos möglich.

Für das weitere Verständnis wichtig ist der Begriff Plattform. Hierunter versteht JetBrains die JVM, JavaScript und Umgebungen ohne Laufzeitumgebung, also Kotlin/Native. Eine wichtige Rolle spielen dabei so genannte Targets. Sie sind für das Bauen, Testen und Paketieren eines vollständigen Stücks Software für eine Plattform verantwortlich. Das können zum Beispiel ausführbare Dateien oder gemeinsam genutzte Bibliotheken sein. Targets enthalten üblicherweise zwei Zweige: `main` für den eigentlichen Code und `test` für Unit-Tests. Ein Zweig wird `source set` genannt. Welches Target der Projektassistent anlegt, ist vom Betriebssystem des Entwicklungsrechners abhängig. Unter Windows entsteht beispielsweise ein `mingw` Target. Die ausführbare Datei lässt sich in bequem in der IDE durch Doppelklick **(Abb. 2)** oder natürlich der Eingabeaufforderung bzw. PowerShell (Windows), Shell (Linux) bzw. Terminal (macOS) starten.



Das Programm mit Gradle starten (Abb. 2)

Targets und ihre Quelltextverzeichnisse werden wie üblich in der Datei `build.gradle` festgelegt. Für das `mingwX64` Target enthält das Build-File **(Listing 1)** standardmäßig nur die Definition zum Bau einer ausführbaren Datei. Mit (`binaries { … executable { … } }`). `entryPoint` wird die aufzurufende Funktion festgelegt. Mit `runTask?.args` lassen sich Parameter für den Start mit Gradle übergeben. Andere Targets können weitere Parameter erfordern, zum Beispiel die Definition eines Build-Hosts. Das ist nötig, um Code für fremde Plattformen zu generieren.

(Listing 1): Die Datei buildgradle

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.4.10'
}
repositories {
    mavenCentral()
    maven { url 'https://dl.bintray.com/kotlin/kotlin-eap' }
}
kotlin {
    def hostOS = System.getProperty("os.name")
    mingwX64("mingw") {
        binaries {
            executable {
                entryPoint = 'com.thomaskuenneth.helloconsole.
main'
                runTask?.args(hostOS)
            }
        }
    }
    ...
    sourceSets {
        mingwMain {
        }
        mingwTest {
        }
        commonMain {
        }
        commonTest {
        }
        macosMain {
        }
    }
}
```

Um zusätzlich eine gemeinsam genutzte Bibliothek einschließlich C-Header-Dateien zu erzeugen, muss die Datei `build.gradle` um ein Dateinamenfragment ergänzt werden. Es wird im Bereich `sharedLib` unterhalb von `binaries` dem Bezeichner `baseName` zugewiesen **(Listing 2)**.

(Listing 2 - Eine gemeinsam genutzte Bibliothek erzeugen)

```
...
binaries {
  ...
  sharedLib {
    baseName = 'libnative'
  }
}
...
```

Während des Bauens entstehen dann die Dateien `libnative.dll`, `libnative_api.h` und `libnative.def`. Alle beim Build entstandenen Artefakte werden im Projektverzeichnis unter build/bin abgelegt. Jedes Target hat sein eigenes Unterverzeichnis, beispielsweise `macos` und `mingw`.

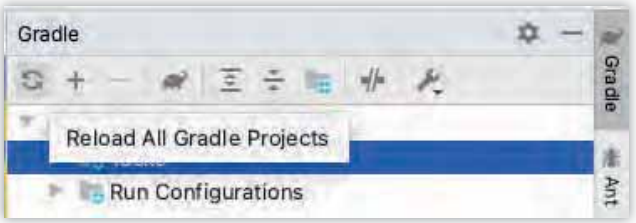
Mehrere Plattformen unterstützen

Projekte, die Code für mehrere Plattformen (Targets) enthalten, werden Multi-Platform-Projekte genannt. Hier übernimmt JetBrains eine Vorgehensweise, die sich bei anderen Cross-Platform-Frameworks ebenfalls bewährt hat: plattformspezifischer und plattformübergreifender Quelltext wird einfach in bestimmten Bereichen des Projektbaumes abgelegt und vom Build-System entsprechend behandelt. Der IntelliJ-Projektassistent enthält mehrere Vorlagen für Multi-Platform-Projekte. Zum Beispiel ermöglicht `Multiplatform Library` die Wiederverwendung von Kotlin-Code auf den Plattformen JVM, JS und Native. `Mobile Application` ist für Apps für Android und iOS gedacht. Die Wahl der am besten geeigneten Vorlage hängt also von der Art des zu erstellenden Projekts und der zu unterstützenden Plattformen ab. Konzeptionell wird gemeinsamer Code analog zu Targets behandelt. Allerdings gibt es in der

Datei `build.gradle` im Block `kotlin { … }` keinen eigenen Bereich. Stattdessen wird `sourceSets { … }` um die Blöcke `commonMain { … }` und `commonTest { … }` erweitert. Außerdem muss in der Datei `gradle.properties` folgende Zeile enthalten sein. Sie legt fest, dass gemeinsame `source sets` verwendet werden sollen:

kotlin.import.noCommonSourceSets=false

Nach Änderungen an `build.gradle` sollte im Werkzeugfenster `Gradle` das Symbol `Reload All Gradle Projects` angeklickt werden, damit IntelliJ etwaige neue Targets und `source sets` erkennt **(Abb. 3)**.



Alle Gradle-Projekte neu laden (Abb. 3)

In `common` gehört klassische Geschäftslogik, beispielsweise Berechnungen und Regeln – also alles, was auf jeder Plattform gleich funktioniert. Auch `main` ist hier gut aufgehoben. **(Listing 3)** zeigt die Datei `HelloConsole.kt` des Beispiel-Projekts `HelloConsole`.

(Listing 3 - Die Datei HelloConsole.kt)

```
package com.thomaskuenneth.helloconsole

fun main(args: Array<String>) {
  val name = if (args.size == 1)
    args[0]
  else
    getName()
  print("Hello, $name")
}
```

`main` gibt die berühmte Grußfloskel `Hello, …` auf der Console aus. Der Name kann als Parameter beim Start übergeben werden. Andernfalls wird er zur Laufzeit erfragt. Da das beim Start in der IDE nicht funktioniert, wird in `build.gradle` mit

```
runTask?.args(hostOS)
```

am besten ein Standardwert gesetzt. Die Variable `hostOS` sieht folgendermaßen aus:

```
def hostOS = System.getProperty("os.name")
```

Die Funktionen `print` und `getName` werden in `Greeter.kt` definiert **(Listing 4)**.

(Listing 4 - Die Datei Greeter.kt in commonMain)

```
package com.thomaskuenneth.helloconsole

expect fun getName(): String
expect fun print(s: String)
```

Um in plattformübergreifendem Code plattformspezifische Funktionen aufrufen zu können, hat JetBrains die beiden Schlüsselwörter `expect` und `actual` eingeführt. Ersteres wird für die Definition einer Funktion verwendet. `actual` leitet die jeweiligen Implementierungen ein. Sie müssen in allen Targets vorhanden sein. Wie eine Implementierung aussehen kann, ist in **(Listing 5)** zu sehen. Es zeigt die Verwendung von so genannten Platform-Bibliotheken. Darunter versteht JetBrains vorübersetzte Bibliotheken, die zu einem bestimmten Target gehören. Diese gewähren den Zugriff auf native Betriebssystemdienste und gehören zur Kotlin/Native-Distribution.

(Listing 5 - Beispiel für die Kommunikation mit Platform-Bibliotheken)

```
package com.thomaskuenneth.helloconsole

import kotlinx.cinterop.ByteVar
import kotlinx.cinterop.CPointer
import kotlinx.cinterop.toKString
import platform.posix.*

@ExperimentalUnsignedTypes
@Suppress("UNCHECKED_CAST")
actual fun getName(): String {
    val size: size_t = 100.toULong()
    val buf = malloc(size) as CPointer<ByteVar>
    fgets(buf, 100, stdin)
    val result = buf.toKString()
    free(buf)
    return result
}

actual fun print(s: String) {
    printf("%s\n", s)
}
```

Die POSIX-Plattform-Bibliothek steht für alle Unix- und Windows-basierten Targets einschließlich Android und iOS zur Verfügung, nicht aber für WebAssembly. Sie enthält Bindings für die Implementierung des POSIX-Standards auf der jeweiligen Plattform. In `getName` wird zunächst mit `malloc` ein 100 Byte großer Speicherblock alloziert und an `fgets` übergeben. Nach Drücken der Enter-Taste wandelt `toKString` den eingegebenen Text in einen Kotlin-String um. Vor dem Verlassen der Funktion muss mit `free` der Puffer wieder freigegeben werden. In der Funktion `print` wird der übergebene String nur als variables Argument an die Bibliotheksfunktion `printf` weitergereicht. Der Formatstring `"%s\n"` sorgt dafür, dass es als Zeichenkette interpretiert wird.

Was geht?

Die Plattform-Bibliotheken erlauben nicht nur einfache Konsolenanwendungen, sondern gestatten auch den Zugriff auf die jeweiligen UI-Frameworks. Es ist also durchaus möglich mit Kotlin/Native Win32- oder macOS-Apps mit Fenstern und Menüleisten zu schreiben. Als kleinen Vorgeschmack zeigt **(Listing 6)**, wie ein sehr einfaches Win32-Fenster auf den Bildschirm gebracht wird. Es ist in **(Abb. 4)** zu sehen.

(Listing 6 - Die Datei SampleMingw.kt des Beispiels WindowsExample)

```
package sample

import kotlinx.cinterop.convert
import platform.windows.MB_ICONINFORMATION
import platform.windows.MB_OK
import platform.windows.MessageBoxW

fun main(args: Array<String>) {
    val caption = "Kotlin/Native"
    val name = if (args.isNotEmpty()) args[0] else "Windows"
    val message = "Hello, $name"
    MessageBoxW(null, message,
            caption, (MB_OK or MB_ICONINFORMATION).convert())
}
```

Letztlich besteht dieses triviale Beispiel nur aus dem Aufruf der Funktion `MessageBoxW`. Sie erhält vier Parameter: ein Fenster-Handle `null`, eine Überschrift, eine Nachricht und ein Typ `.convert()` wandelt einen `Int` Wert in den Datentyp `platform.windows.UINT` um.

Ein einfaches Windows-Fenster (Abb. 4)

Ob das auch im großen Stil sinnvoll ist, muss natürlich individuell bewertet werden. Schließlich erfordern Programme, die über eine einfache Demo hinaus gehen, eine solide Kenntnis der jeweiligen Plattform. Ist diese vorhanden, kann meist auch gleich die Standardsprache genutzt werden.

Gemeinsam genutzte Bibliotheken

Kotlin/Native ist derzeit noch kein klassisches Cross-Platform-Framework, denn von diesen wird üblicherweise auch die Abstraktion der Benutzeroberfläche erwartet. Dass die Reise dorthin geht, zeigt die im Herbst 2020 vorgestellte Vorschauversion von Jetpack-Compose für den Desktop. Sie bringt Jetpack-Compose, das neue deklarative UI-Framework für Android, auf macOS, Linux und Windows. Damit werden plattformübergreifende Oberflächen mit Kotlin jenseits der beiden etablierten mobilen Plattformen möglich.

Die Einsatzmöglichkeiten von Ktolin/Native sind aber auch jetzt schon vielfältig. So lässt sich die gesamte Client-seitige Anwendungslogik über mehrere Plattformen hinweg wiederverwenden, wenn die über die Jahre entstanden und bewährten Muster zur Trennung von Daten, Oberfläche und Logik vernünftig umgesetzt werden. In diesem Fall werden für die jeweiligen Clients nur die Oberflächen einschließlich App-Rahmen geschrieben. Diese orchestrieren die Navigation innerhalb der Anwendung und rufen bei Bedarf die in Kotlin geschriebenen Geschäftslogikkomponenten auf. Wie so etwas aussehen kann, zeigt das Beispiel `iOSExample`. Es berechnet eine Fibonacci-Zahl. Der Kotlin-Code ist in **(Listing 7)** zu sehen, während **(Abb. 7)** die App zur Laufzeit zeigt.

(Listing 7 - Eine Fibonacci-Folge berechnen)

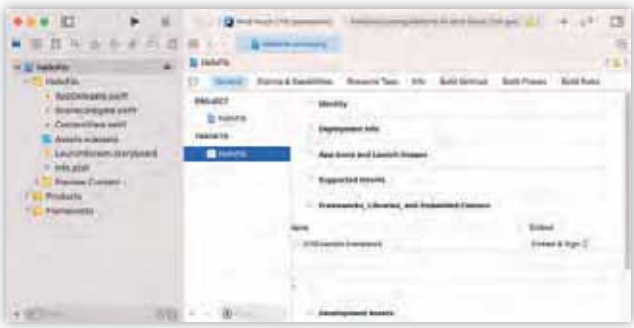
```
package com.thomaskuenneth.iosexample

actual fun fib(n: Int): Int {
  return when (n) {
    0 -> 0
    1 -> 1
    else -> if (n >= 2) fib(n - 1) + fib(n - 2) else 0
  }
}
```

Die Funktion `fib` wird in einem Swift/SwiftUI-Projekt verwendet. Damit das funktioniert, wird sie als so genanntes Apple-Framework exportiert. **(Listing 8)** zeigt einen Auszug aus der Datei `buid.gradle`.

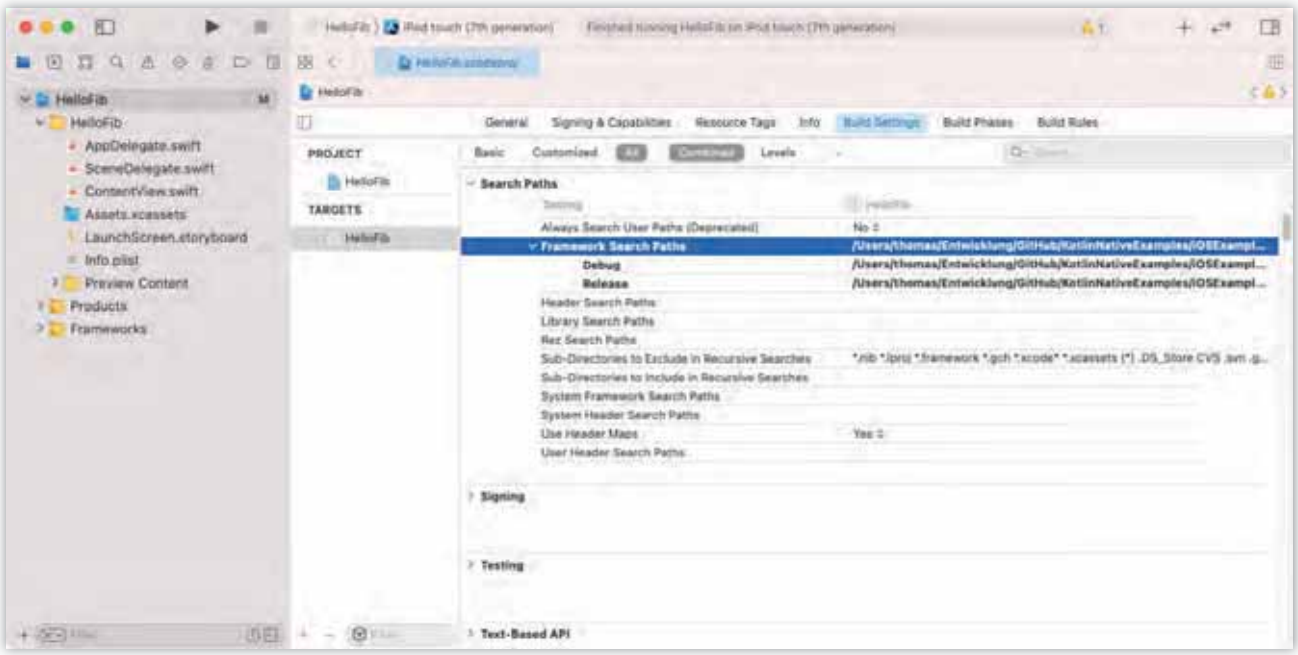
(Listing 8 - Ein Apple-Framework erzeugen)

```
kotlin {
  iosX64("ios") {
    binaries {
      framework()
    }
  }
  …
```

In Xcode muss das Framework unter `General – Frameworks, Libraries and Embedded Content` und `Build Settings – Search Paths - Framework Search Paths` hinzugefügt werden. Das ist in **(Abb. 5)** und **(Abb. 6)** zu sehen.



General – Frameworks, Libraries and Embedded Content (Abb. 5)



Build Settings – Search Paths - Framework Search Paths (Abb. 6)

Den Aufruf von **fib** aus Swift zeigt **(Listing 9)**. Mit `import iOSExample` wird die Bibliothek importiert. `Int32` repräsentiert den Kotlin-Datentyp `Int`. Beim Anklicken von + und – wird der Wert innerhalb bestimmter Grenzen inkrementiert bzw. dekrementiert.

(Listing 9 - Eine Kotlin-Funktion in Swift aufrufen)

```
import SwiftUI
import iOSExample

struct ContentView: View {

  @State private var n: Int32 = 2

  var body: some View {
    VStack {
      Stepper(onIncrement: {
        if (self.n < 30) {
          self.n += 1
        }
      }, onDecrement: {
        if (self.n > 1) {
          self.n -= 1
        }
      }) {
        Text("fib(\(n)) = \(FibKt.fib(n: n))")
      }.padding()
    }
  }
}

struct ContentView_Previews: PreviewProvider {
  static var previews: some View {
    ContentView()
  }
}
```

Das Ergebnis des Funktionsaufrufs wird von Swift direkt verwendet, um das Label der `Stepper` Komponente zu aktualisieren.

iOSExample im iOS-Simulator (Abb. 7)

Fazit:

Noch ist Kotlin/Native nicht fertig. Teile der Technologien aber auch der Sprache selbst sind als experimentell gekennzeichnet, beispielsweise vorzeichenlose Integerzahlen. Es muss deshalb damit gerechnet werden, dass sich bei einem Versionswechsel das eine oder andere ändert. Trotzdem kann es sich lohnen, schon jetzt einen Blick auf Kotlin/Native zu werfen.

Vor allem wenn generell ein Umstieg auf Kotlin ansteht oder bereits stattgefunden hat. Spaß macht das Experimentieren mit Kotlin/Native auf jeden Fall. Die im Artikel gezeigten Beispiele stehen auf `GitHub` zur Verfügung.

Quellen:

1 http://bit.ly/3s2u0wU

#JAVAPRO #FunctionalProgramming #Clojure #DSL

DSLs ganz einfach mit Clojure

Clojure-Programme sind kompakter als ihre Java-Pendants und ermöglichen damit die Konzentration auf das Wesentliche. Wichtiger noch: Clojure ist eine funktionale Sprache und ermöglicht damit oft eine andere Sicht auf die Domäne als das klassisch objektorientierte Java. Dieser Artikel demonstriert das anhand einer kleinen domänenspezifischen Sprache für Bilder.

Autor:

Dr. Michael Sperber ist Geschäftsführer der Active Group GmbH, die Individualsoftware ausschließlich mit funktionaler Programmierung entwickelt. Er ist international anerkannter Experte für funktionale Programmierung und wendet sie seit über 20 Jahren in Forschung, Lehre und industrieller Entwicklung an. Außerdem hat er zahlreiche Fachartikel und Bücher zum Thema verfasst, sowie das Curriculum für das iSAQB-Advanced-Modul "Funktionale Software-Architektur". Michael Sperber ist Mitbegründer des Blogs funktionale-programmierung.de und Mitorganisator der Entwicklerkonferenz BOB.

Clojure hat sich als alternative Programmiersprache auf der JVM fest etabliert. Die Sprache ist ausgereift, das Ökosystem enthält viele nützliche Libraries und Frameworks und die kleine Community ist sehr aktiv. Nicht nur die Nische von Clojure ist klein, sondern die Programmiersprache selbst ebenfalls und damit schnell und leicht erlernbar. Anhand einer kleinen domänenspezifischen Sprache für Bilder wird dies nachfolgend demonstriert.
Aus Sicht der objektorientierten Programmierung sollte aus jedem Substantiv in einer Domänenbeschreibung ein Objekt werden: Ein Bild sollte demnach durch ein Objekt repräsentiert werden. In Java AWT gibt es dafür das Interface `java.awt.image.Image` mit der relevanten Implementierung `BufferedImage`. Diese Klasse hat exemplarisch die Methode `setRGB`.

(Listing 1)

```
void setRGB(int x, int y, int rgb)
```

Sie setzt also in einem Raster aus Pixeln einen Pixel auf eine bestimmte Farbe. Ein BufferedImage ist also zunächst noch gar nicht das gewünschte Bild. Das entsteht erst durch eine Folge von Methodenaufrufen, welche die Farbe bestimmter Pixel ändert. Die Idee ist sehr technisch: Ein Bild besteht demnach aus rechteckig angeordneten Pixeln jeweils bestimmter Farbe.
Die funktionale Programmierung versucht, die Frage „Was ist ein Bild?" auf höherer Ebene zu beantworten. Wenn Menschen ein Bild betrachten, so sehen sie an jeder Stelle des Bildes eine bestimmte Farbe. Das Konzept des Pixels nehmen Menschen nicht direkt wahr. Die Idee - an jeder Stelle des Bildes eine bestimmte Farbe - lässt sich aber auch ganz ohne Pixel formulieren, nämlich als eine Funktion von Stelle zu Farbe. Diese Idee verfolgt das System Pan, das Conal Elliott[1] 2003 veröffentlichte. Der Artikel zeichnet diese Idee vereinfacht in Clojure nach. Den Anfang macht die Datenmodellierung. Zu Beginn müssen die Begriffe Stelle und Farbe modelliert werden. Eine Stelle wird als kartesische Koordinate mit `x` und `y` Felder modelliert. In Clojure sieht das wie folgt aus:

(Listing 2)

```
(defrecord Point [x y])
```

Clojure ist ein Lisp-Dialekt. Das heißt, dass zusammengehörende Konstrukte immer von Klammern umschlossen sind. Das `defrecord` kennzeichnet die Definition eines Records, also eines Typs für zusammengesetzte Daten mit `x` und `y` Feld. In Java wäre das ein Plain-Old-Java-Object und in der Tat erzeugt Clojure für die obige Deklaration eine POJO-Klasse dafür namens `Point` mit `x` und `y` Feldern. Die Record-Deklaration zeigt, dass Clojure eine dynamisch typisierte Sprache ist. Man muss bei `x` und `y` nicht angeben welche Typen sie haben, da dies erst zur Laufzeit entschieden wird. Für Koordinaten werden hier immer Zahlen verwendet. Der Konstruktor von `Point` heißt in Clojure `Point`. und entsprechend sieht die Konstruktion eines Punkts wie in **(Listing 3)** aus. Zwei Beispielpunkte namens `point1` und `point2` werden in **(Listing 4)** definiert:

(Listing 3)

```
(Point. 1 2)
```

(Listing 4)

```
(def point1 (Point. 1 2))
(def point2 (Point. 0.5 4))
```

Für `Point` sind außerdem automatisch zwei Getter definiert mit den Namen `:x` und `:y`. Anders als in Java sind dies aber keine Methoden sondern Funktionen. Aufgerufen werden diese auch mit Klammern, wie zum Beispiel:

- `(:x point1)` liefert `1`
- `(:y point2)` liefert `4`

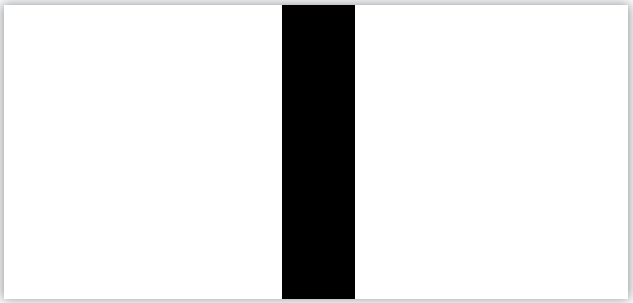
Eine Farbe ist in diesem Artikel ein Boolean `true` oder `false` für schwarz respektive weiß. Das ist leicht zu verallgemeinern auf beliebige Farben und Transparenz. Koordinaten und Farben sind also definiert. Ein Bild ist eine Funktion, die für Koordinaten (`Point` Objekt) eine Farbe liefert, also `true` oder `false` **(Listing 5)**.
(Listing 5)

```
(defn vstrip
   [p]
   (<= (Math/abs (:x p)) 0.5))
```

`defn` definiert in Clojure eine Funktion. In diesem Fall eine Funktion, die `vstrip` heißt und in eckigen Klammern einen Parameter namens `p` hat. Da Clojure eine funktionale Sprache ist, liefert jede Funktion einen Wert. Darum wäre `return` redundant. Diese Funktion liefert also das Ergebnis eines Vergleichs mit `<=`. Hier sieht man, dass auch `<=` nur eine Funktion in Clojure ist und entsprechend vor die Argumente und mit Klammern geschrieben wird. `Math/abs` ist die statische Java-Methode aus der `Math Klasse`. Die Funktion liefert also `true`, wenn die X-Koordinate von `p` zwischen `-0,5` und `+0,5` liegt, ansonsten `false`. Um herauszubekommen, welche Farbe eine bestimmte Koordinate hat, kann `vstrip` einfach aufgerufen werden:

- `(vstrip (Point. 0 0))` liefert `true` (schwarz)
- `(vstrip (Point. 0.6 0))` liefert `false` (weiß)

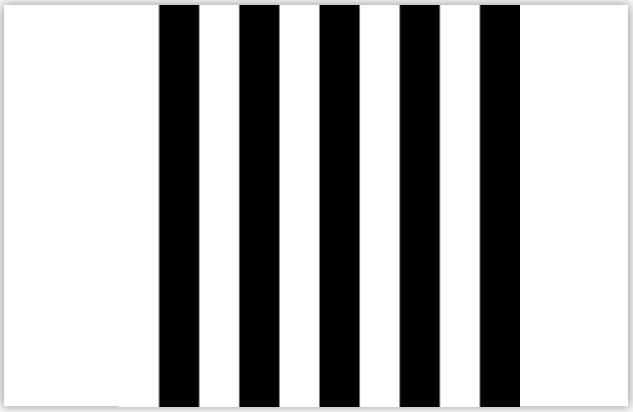
Das Bild wird in **(Abb.1)** gezeigt. Genauer gesagt, ein Ausschnitt des Bildes von jeweils `-2` bis `+2`, denn das Bild ist zumindest konzeptuell unendlich groß. `vstrip` ist ziemlich langweilig. Marginal interessanter erscheint **(Listing 6)**, welches **(Abb. 2)** ergibt.



Aufruf zur Farbbestimmung der Koordinate (Abb. 1)

(Listing 6)

```
(defn vstripes
  [p]
  (even? (int (Math/floor (:x p)))))
```



Gefängnismuster (Abb.2)

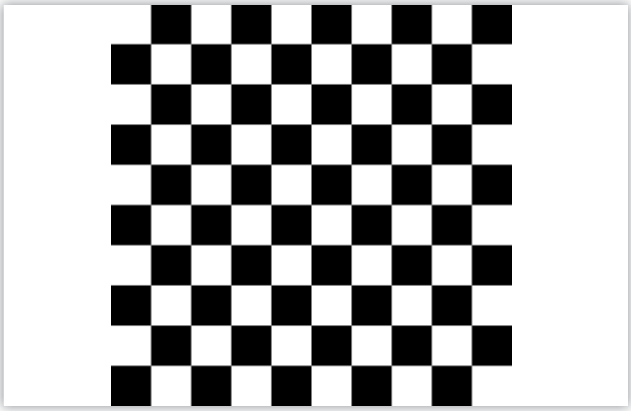
Die Funktion `int` macht ein Integer aus dem `double`, das bei `Math/floor` herausgekommen ist. Die Funktion `even?` testet, ob ein Integer gerade ist oder nicht. Das Fragezeichen gehört in Clojure zum Namen und ist Konvention für Funktionen, die ein Boolean liefern. In Java würde die Funktion `isEven` heißen. Die gleiche Idee lässt sich auch in der Horizontalen verwirklichen (Listing 7), weg vom Gefängnismuster (Abb. 2) hin zu einer Tischdecke (Listing 8) (Abb. 3).

(Listing 7)

```
(defn hstripes
  [p]
  (even? (int (Math/floor (:y p)))))
```

(Listing 8)

```
(defn checker
  [p]
  (even? (+ (int (Math/floor (:x p))) (int (Math/floor (:y p))))))
```



Tischdeckenmuster (Abb. 3)

Hier kann man erkennen, wo der Unterschied zwischen der objektorientierten Herangehensweise von `java.awt.image.Image` und diesem funktionalen Modell liegt. `java.awt.image.Image` ist geprägt durch eine Implementierungsidee (Pixel), während die funktionale Sicht auf der mathematischen Essenz der Idee eines Bildes beruht. Zur Implementierung, wie aus der Darstellung ein Bild auf dem Bildschirm wird, ist bislang noch gar nichts bekannt.

Kombinatormodelle

Das Bild `checker` mit dem Schachbrettmuster wurde oben direkt definiert. Aber `checker` kann auch mit Hilfe von `vstripes` und `hstripes` definiert werden (Listing 9).

(Listing 9)

```
(defn checker
  [p]
  (not= (vstripes p) (hstripes p)))
```

Die Funktion `not=` testet auf „nicht gleich", ist also effektiv eine Exklusiv-Oder-Verknüpfung (XOR) auf den Farben von `vstripes` und `hstripes`. Das `p` wird von `checker` an `vstripes` und `hstripes` durchgeschleift. Die beiden Bilder werden als Ganzes `xor` kombiniert. Diese Idee, zwei Bilder `xor` zu verknüpfen lässt sich aus `checker` herausabstrahieren (Listing 10). Der Bindestrich gehört zum Namen. In Java würde man einen Underscore verwenden.

(Listing 10)

```
(defn img-xor
  [img1 img2]
  (fn [p]
    (not= (img1 p) (img2 p))))
```

Diese Funktion nimmt zwei Bilder wie zum Beispiel `vstripes` oder `hstripes` als Argumente und liefert wieder ein Bild, also eine Funktion. Das `fn` ist das Clojure-Pendant zum Lambda-Ausdruck in Java und stellt eine Funktion her, die in diesem Fall ein Point-Objekt akzeptiert und eine Farbe liefert, mit dem gleichen

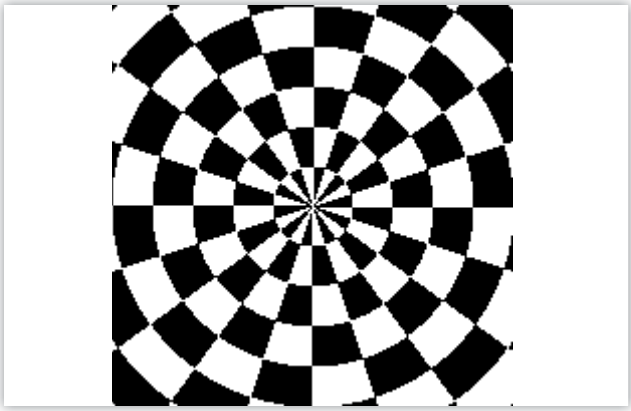
Rumpf wie schon zuletzt bei `checker`. Das kann wie in **(Listing11)** definiert werden. Damit ist sofort die Essenz von `checker` klar, die bei der ersten Definition doch schwieriger zu sehen war.

(Listing 11)

```
(def checker
  (img-xor hstripes vstripes))
```

Die Implementierung von Funktionen wie `img-xor`, die auf Bildern als Ganzes operieren, also sogenannte Kombinatoren, macht aus Clojure zusammen mit einer Library dieser Funktionen effektiv eine domänenspezifische Sprache.

Koordinatentransformation



Kreisförmiges Bild (Abb. 4)

In **(Abb. 4)** zeigt sich ein weiteres Bild. Intuitiv kann man sehen, dass es eine ähnliche Idee wie `checker` umsetzt, nur im Kreis statt im Quadrat. Ideal wäre, wenn gerade das Konzept „im Kreis statt im Quadrat" als Code ausgedrückt werden könnte. Und tatsächlich gibt es Koordinaten, die im Kreis statt im Quadrat funktionieren, sogenannte Polarkoordinaten. Diese bestehen nicht aus X- und Y-Koordinate. Stattdessen stellt man sich vor, dass zu einem Punkt ein Strahl vom Ursprung gezeichnet wird. Die Polarkoordinaten sind dann die Länge des Strahls (auch der Radius, meist r) und der Winkel des Strahls zur X-Achse (meist p). Die kartesischen Quadratkoordinaten müssen in Polarkoordinaten umgerechnet werden. Das macht die Funktion in **(Listing 12)**, deren Formel man aus einer handelsüblichen Formelsammlung beziehen kann.

(Listing 12)

```
(defn to-polar
  [p]
  (Point. (distance-from-origin (:x p) (:y p))
    (Math/atan2 (:x p) (:y p))))
```

Die Hilfsfunktion `distance-from-origin` berechnet den Abstand vom Ursprung ebenfalls mit einer Standardformel **(Listing 13)**.

(Listing 13)

```
(defn distance-from-origin
  [x y]
  (Math/sqrt (+ (* x x) (* y y))))
```

In **(Abb. 4)** wechselt die Farbe bei einer Kreisumdrehung insgesamt 20 Mal, also 10 Mal hin und zurück. Entsprechend muss der Winkel in den Polarkoordinaten angepasst werden, so dass eine Kreisumdrehung (zweimal Pi) gerade 20 entspricht. Das macht die folgende Hilfsfunktion `turn`, die bei Polarkoordinaten den Winkel (der im y Feld steht) entsprechend skaliert **(Listing 14)**.

(Listing 14)

```
(defn turn
  [n]
  (fn [p]
    (Point. (:x p)
    (* (:y p)
       (/ n Math/PI)))))
```

Die `turn` Funktion akzeptiert also als Parameter `n` die Anzahl der Farbwechsel pro Umdrehung und liefert eine Funktion, die ein Point-Objekt entsprechend transformiert. Um jetzt das Schachbrettmuster im Kreis zu bilden, müssen die Koordinaten zunächst mit `to-polar` in Polarkoordinaten umgewandelt und dann mit `turn` der Winkel skaliert werden, damit das ursprüngliche `checker` dann die Farbe ausrechnen kann. Die Koordinaten werden also durch eine kleine Pipeline geleitet, die aus drei Funktionen besteht. Mathematisch gesehen werden die Funktionen komponiert. Darum heißt die eingebaute Funktion in Clojure dafür auch `comp` und wird so benutzt, um `polar-checker` zu definieren **(Listing 15)**

(Listing 15)

```
(defn polar-checker
  [n]
  (comp checker (turn n) to-polar))
```

Auch hier sieht man gut, wie die funktionale Sichtweise die Essenz der Idee dieses Bildes herausstreicht und wie Clojure es erlaubt, die Idee in nur wenigen Zeilen Code zum Ausdruck zu bringen. Mit etwas Sinn für Formelspielereien kann man so richtig hübsche Bilder erzeugen. **(Listing 16)** zeigt, dass man auch umgekehrt von Polarkoordinaten in kartesische Koordinaten zurückrechnen kann.

(Listing 16)

```
(defn from-polar
  [p]
```

```
(Point. (* (:x p) (Math/cos (:y p)))
    (* (:x p) (Math/sin (:y p)))))
```

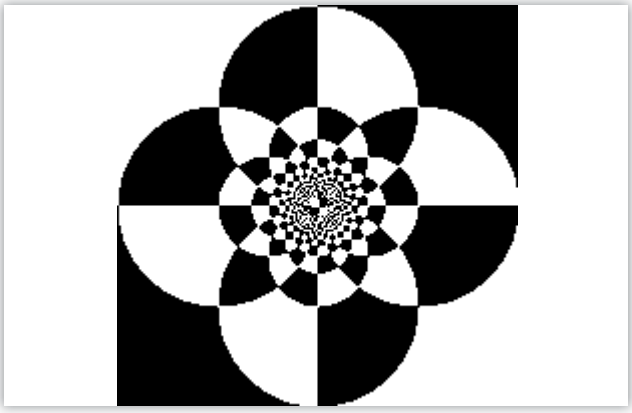
Das kann man benutzen, um auf den Polarkoordinaten Transformationen durchzuführen, indem die Koordinaten erst ins Polarformat überführt, dann transformiert und anschließend in kartesische Koordinaten zurückgerechnet werden. Auch das ist eine Pipeline mit `comp` **(Listing 17)**. Die Funktion **(Listing 18)** bildet so den Kehrwert des Polarradius, stülpt also innen nach außen. Diese kann man jetzt mit `checker` kombinieren **(Listing 19)**. Das Ergebnis wird in **(Abb. 5)** gezeigt.

(Listing 17)

```
(defn from-polar-transformation
  [trafo]
  (comp from-polar trafo to-polar))

(Listing 18)
(def invert-polar-radius
  (from-polar-transformation
   (fn [p]
     (Point. (if (zero? (:x p))
         0.0
         (/ 1.0 (:x p)))
       (:y p)))))
```

(Listing 19)

```
(def rad-invert-checker
  (comp checker invert-polar-radius))
```



Ergebnis (Abb. 5)

Bilder auf den Bildschirm

Die Essenz der Bilder ist schön und gut, aber trotzdem würde man sie natürlich gern sehen. Entsprechend geht es in diesem Abschnitt darum, zum Konzept die Implementierung zu liefern und demonstriert damit die andere Seite von Clojure, nämlich die Interoperabilität mit Java. Für die Darstellung der Bilder werden einige AWT- und Swing-Klassen benötigt, die in den Namespace importiert werden. Der Namespace ist das Clojure-Pendant zum Java-Package, der am Beginn einer Datei deklariert wird **(Listing 20)**.

(Listing 20)

```
(ns javapro.functional-images
  (:import (java.awt Color Graphics Image BorderLayout)
     (javax.swing JFrame JLabel ImageIcon)
     java.awt.image.BufferedImage))
```

Damit kann die Funktion `image->bitmap` aus dem Clojure-Image eine handfeste Bitmap-Date erzeugen. **(Listing 21)** zeigt den Header dieser Funktion.

(Listing 21)

```
(defn image->bitmap
  [image width height x-min x-max y-min y-max]
```

Der Pfeil im Namen bildet die Konvention für Funktionen, die ein Objekt in ein anderes konvertieren. Bei den Parametern ist `image` das Clojure-Bild, `width` und `height` sind Höhe und Breite des Bildes in Pixel und `x-min`, `x-max`, `y-min` und `y-max` sind der Koordinaten-Ausschnitt aus dem Bild, der angezeigt werden soll. Als nächstes werden einige lokale Variablen benötigt. Das geht in Clojure mit dem Konstrukt `let`. Dort stehen in eckigen Klammern die Namen lokaler Variablen und Ausdrücke für deren Werte, danach können die lokalen Variablen verwendet werden **(Listing 22)**.

(Listing 22)

```
(let [buffered-image (BufferedImage.
              width height
              BufferedImage/TYPE_INT_ARGB)
      graphics (.getGraphics buffered-image)
      xinc (/ (- x-max x-min)
        (double width))
      yinc (/ (- y-max y-min)
        (double height))]
```

Für die erste Variable `buffered-image` erzeugt die Funktion ein BufferedImage-Objekt, in das die Funktion das Bild painten wird. Wie bei den Records ist `BufferedImage` der Name des Konstruktors für diese Klasse. Danach wird aus `buffered-image` der Grafik-Context extrahiert. Dafür besitzt `BufferedImage` die Methode `getGraphics`. Der Java-Aufruf dieser Methode `buffered-image.getGraphics()` wird in Clojure als Funktionsaufruf mit `.getGraphics` geschrieben. Die beiden Bindungen für `xinc` und `yinc` rechnen schließlich aus, wie breit und wie hoch ein Pixel im Koordinatensystem des Bildes ist. Die Funktion `double` macht aus den Integern `width` und `height` jeweils Doubles. Jetzt muss die Funktion über die Pixel des Bildes

extrahieren, jeweils die Farbe ermitteln, diese im Grafik-Context als kleine Rechtecke zeichnen und schließlich am Ende das `BufferedImage` Objekt zurückliefern. Das `doseq` Konstrukt ist das Pendant zum `foreach` und Java und führt hier zwei geschachtelte Schleifen über alle Pixel des Bildes aus **(Listing 23)**.

(Listing 23)

```
(doseq [x (range 0 width)
  y (range 0 height)]
  (let [black? (image (Point. (+ x-min (* xinc x)) (+ y-min (*
yinc y))))
color (if black?
     Color/BLACK
     Color/WHITE)]
    (.setColor graphics color)
    (.fillRect graphics x y 1 1)))

buffered-image))
```

Das Image-Objekt, welches `image->bitmap` liefert, kann nun mit folgender Funktion, die nahezu ausschließlich aus Aufrufen von Java-Methoden besteht, zum Beispiel in einem Fenster angezeigt werden **(Listing 24)**.

(Listing 24)

```
(defn display-image!
  [image width height x-min x-max y-min y-max]

(let [bitmap (image->bitmap image width height x-min x-max y-min
y-max)
  frame (JFrame. "Image")
label (JLabel.)]
    (.setIcon label (ImageIcon. bitmap))
    (.setSize frame width height)
    (.setDefaultCloseOperation frame JFrame/EXIT_ON_CLOSE)
    (.add (.getContentPane frame) label BorderLayout/CENTER)
    (.pack frame)
    (.setVisible frame true)))
```

Fazit:

Dieser Artikel gibt einen Einblick in beide Seiten von Clojure: Die Seite, welche Domänenkonzepte in Reinform modelliert und die andere Seite, welche ihre Implementierung ermöglicht. Das alles geht auch in Java. Allerdings mit einigen kleinen, aber wesentlichen Abstrichen mit der Begründung, dass Java zwar inzwischen Lambda-Ausdrücke aus der funktionalen Programmierung importiert hat, aber immer noch zwischen Methoden und Funktionen unterscheidet. In Clojure hingegen kann die Idee des Domänenmodells in Reinform ausgedrückt werden, wobei ganz natürlich eine domänenspezifische Sprache entsteht. Dieser Prozess ist damit ein natürlicher Bestandteil der Arbeit mit Clojure. Clojure kann dann auch die Implementierung des Konzepts und die Anbindung an Betriebssystem und UI begleiten. Die Sprache hat viel zu bieten, ist aber insgesamt deutlich kleiner als Java und damit einfach zu beherrschen. Und vor allem: Es macht sehr viel Spaß!

Quellen:

1 http://bit.ly/3tpvTEm

TIMOCOM

Dein Hub der Möglichkeiten

als Java Software Entwickler (m/w/d) oder
Software Architekt / remote (m/w/d).

Werde Teil eines agilen
Teams und bewirb Dich jetzt.
jobs.timocom.de

4 Reasons to Choose Payara Platform Enterprise

Payara Platform Enterprise is a cloud-native middleware application platform supporting mission critical production systems with reliable and secure deployments of Jakarta EE (Java EE) applications on premise, in the cloud, or hybrid environments. A stable platform with monthly releases, bug fixes, and a 10-year software lifecycle, the Payara Platform is aggressively compatible with the ecosystem, cloud vendors, Docker, and Kubernetes.

Software. **Security.** **Stability.** **Support.**

Modern App Development with the Payara Platform

Jakarta EE enables community-driven collaboration and open innovation for the cloud to help developers create portable cloud-native applications.

- Migrate existing Java EE applications to the cloud – no code rewriting required
- Build new, cloud-native Jakarta EE applications on public cloud
- Deploy with your cloud platform's IaaS
- Achieve elastic, scalable deployment with Docker
- Support full clustering & high availability on Kubernetes

payara® MICRO

payara® SERVER

Download datasheets and learn more at payara.fish/resources

#JAVAPRO #Testing #Quality

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

Die Konzepte der Testautomatisierung sind nichts Neues, allerdings ist ihre Bedeutung in den letzten Jahren gestiegen. Mit der zunehmenden Digitalisierung ändert sich das Kundenverhalten und damit der Markt massiv. Von Entwicklern wird die zeitnahe Umsetzung und Live-Stellung neuer Features erwartet. Sie sehen sich mit zunehmend schnellerer technologischer Entwicklung konfrontiert. Trotz des erhöhten Zeitdrucks dürfen Ansprüche an Software-Qualität nicht verloren gehen. Dieser Artikel soll zeigen, wie die Testautomatisierung dabei hilft.

Autor:

Anja Papenfuß-Straub ist Application-Architektin bei der ING Deutschland. Ihr Schwerpunkt liegt auf Architektur, Codequalität und Testbarkeit. Sie arbeitet seit 3 Jahren an der technischen Umsetzung regulatorischer Vorgaben, zum Beispiel zum Thema DSGVO.

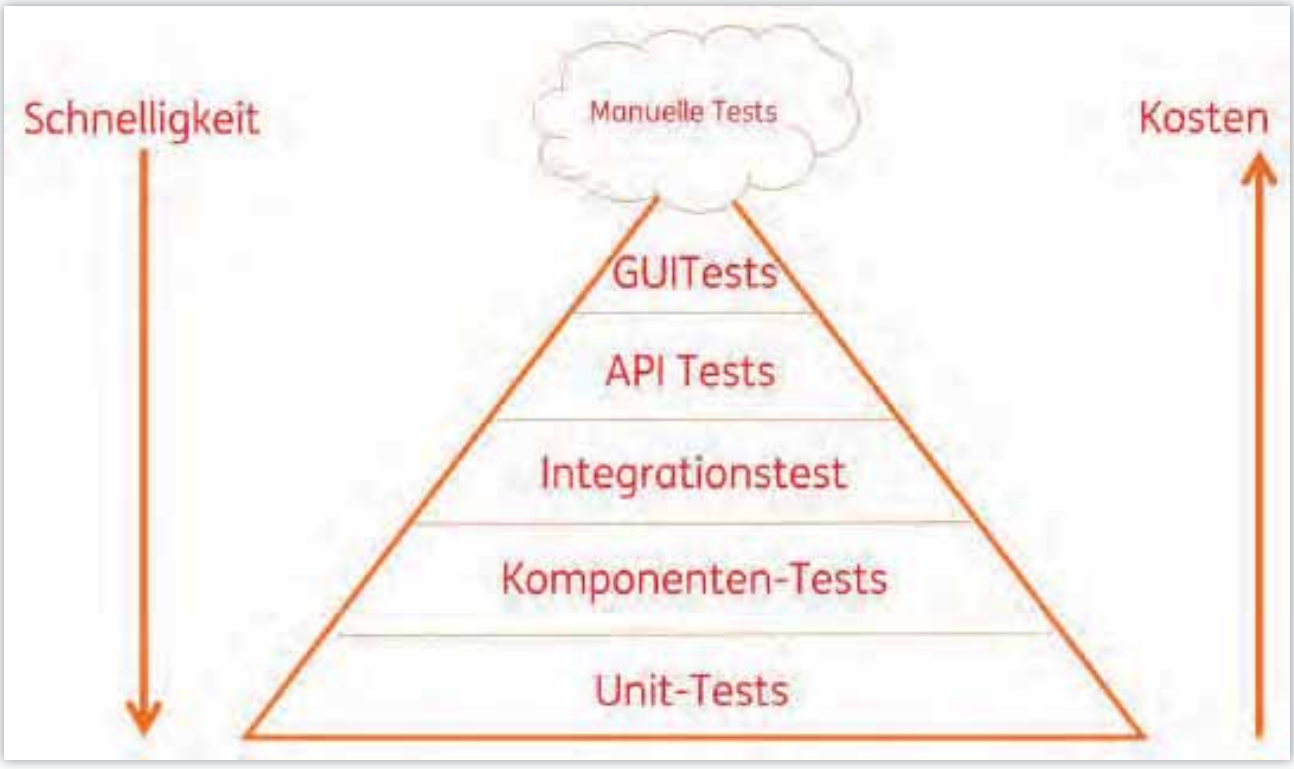
Vor über 10 Jahren wurde Software noch anders entwickelt und getestet. Es gab umfangreiche Debugging-Sessions, um Fehler auf dem Produktivsystem zu finden. Es existierten Poster mit: „Testen ist was für Mädchen, Jungs gehen live!“ Unit-Tests waren häufig nicht erwünscht, da sie als zu teuer galten. Tester durften aufwendige End-2-End-Tests (http-Unit-Tests) schreiben und Entwickler waren froh darüber, da sie doch zumindest ein kleines Gefühl der Sicherheit brachten. Es ist bekannt, wie nervenaufreibend es ist, Fehler auf Produktionssystemen suchen und beheben zu müssen. Dass es erstaunlicherweise relativ gut funktioniert, ist auf lange Testphasen vor jedem Release sowie auf jährlich geringe Anzahl an Releases zurückzuführen. In den langen und aufwendigen Testphasen konnten teilweise noch Fehler entdeckt und behoben werden.

Die Zeiten ändern sich. Was dieser digitalen Welt fehlt, ist Geduld. Dazu drängen viele Mitbewerber auf den Markt. Unternehmen sind 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. Mittlerweise arbeiten die meisten in agilen Teams und sind als DevOps verantwortlich für Entwicklung und Betrieb der Software. Es kann nicht mehr auf lange Testphasen durch Fachtester und Fachteams vor Live-Stellungen gezählt werden. 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. Die Unternehmen selbst haben erkannt, dass Produktionsfehler um ein Vielfaches teurer sind als Fehler, die während der Entwicklungsphase erkannt werden. Im schlimmsten Fall drohen Reputationsverlust 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. Code kann einfach über die Entwicklungsumgebung geändert und erweitert werden. Erfolgt dies jedoch ohne Anpassung der Code-Struktur, wird der Code starr und dadurch schlecht wartbar. Der Grad der Unordnung innerhalb unseres Codes nimmt zu, auch bekannt als Software-Entropie.

Bei jeder neu zu implementierenden Funktion muss ein Kompromiss gefunden werden, zwischen den begehrten neuen Features und der vorhandenen Code-Basis. Gilt das Augenmerk nur der Entwicklung von neuem Code zu Lasten der Bestandsbasis, führt das zuerst zum Verlust der strukturellen Qualität zum Beispiel in der Architektur, Entwurfsmuster etc. und im schlimmsten Fall auch zum Verlust der fachlichen, funktionalen Qualität. Im Laufe der Jahre hat fast jeder Entwickler seine Erfahrungen mit Legacy-Anwendungen gemacht. Oft war die ursprüngliche Absicht des Code-Erstellers nach mehreren Anpassungen nicht mehr eindeutig erkennbar. Man spricht in diesem Fall von Softwareerosion bzw. technischen Schulden.



Testpyramide von Michael Cohn (Abb. 1)

Schnelle Änderungen an diesem Code sind kaum noch möglich, sie sind fehleranfällig und teuer. Software lebt aber von Veränderungen und muss anpassbar bleiben. Das heißt, die Anpassung der Software erweist sich als einfach und schnell durchführbar. Die Struktur des Codes ist klar erkennbar und der Code ohne großen Aufwand veränderbar. Die einzige Möglichkeit, Softwareerosion unter Beachtung der Software-Entropie zu verhindern, ist regelmäßiges Refactoring.

Test-Harnisch

Um Neuentwicklungen und Refactorings abzusichern, braucht es einen Test-Harnisch. Er fungiert als Sicherheitsnetz und besteht aus vielen einfachen und gut wartbaren Entwicklertests, den Unit- bzw. Komponententests sowie wenigen Integrationstest. Letztere sind in der Wartung und wegen den Laufzeiten teurer. Die Testpyramide **(Abb. 1)** definiert die Testarten bzw. Testlevels und deren Umfang unter Beachtung von Kosten und Nutzen. Günstige und schnelle Testarten wie zum Beispiel Unit-Tests werden bevorzugt. Mit jeder Anpassung und Erweiterung des Codes wird die Testbasis vergrößert und die Maschen des Sicherheitsnetzes verfeinert. Das hilft, die strukturelle und funktionale Qualität zu erhalten bzw. zu verbessern.

Bedeutung von Code-Coverage

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? Bei einer realistischen Betrachtung ist eine hohe Code-Coverage noch lange keine Garantie für gute und sinnvolle Tests. Code-Coverage prüft nur die Code-Zeilen und Abzweigungen, die bei einem Test durchlaufen wurden. Die Validierung des Testergebnisses und die Richtigkeit des Codes spielen dabei keine Rolle. Verlassen wir uns also rein auf die Testabdeckung, wiegen wir uns in falscher Sicherheit.

Trotzdem sind Code-Coverage-Vorgaben mit sinnvollen Werten, d. h. keine 100%, richtig. Eine geringe Code-Coverage ist ein Indiz für notwendige Nachbesserungen in Sachen Tests. In Kombination mit statischer Code-Analyse sind damit gute Ergebnisse zu erzielen, um die Test- bzw. Code-Qualität zu erhalten und zu verbessern.

Testqualität

Was also macht einen guten Test aus?

- `Fast`: Die Testlaufzeiten sollten gering sein, bei reinen Unit-Tests sogar im Millisekundenbereich. 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 stark erschwert. Die Testmethoden müssen unabhängig voneinander ausführbar sein.
- `Repeatable`: Die Tests sollen wiederholbar sein und bei jedem Lauf das gleiche Ergebnis liefern.
- `Self-Verifying`: Ein Test muss sich selbst automatisch überprüfen. Es sollte nur ein Aspekt pro Test getestet werden. Dabei haben Testmethoden immer den gleichen Aufbau und folgen dem Triple-A-Prinzip **(Listing 1)**. Ein Test besteht aus Arrange: Initialisieren des Test-Setups, Act: Ausführen des zu testenden Codes und Assert: Validierung des Ergebnisses. Durch die automatische Assertion werden eine manuelle Verifikation genauso wie Log-Ausgaben in Tests unnötig. Die Log-Ausgaben verlangsamen den Test und sind, da sie nicht gelesen werden, keine Überprüfung des Testergebnisses!
- `Thorough/Timely`: Tests werden vor dem eigentlichen Code geschrieben, siehe Test-Driven-Development.

(Listing 1)

```
   @Test
   public void findName(){
//Arrange
SearchQuery query = new SearchQuery("name");
//Act

List<SearchResult> results = classUnderTest.find(query);
     //Assert
     assertThat(results).extracting(SearchResult::name)
.containsOnly("name");
   }
```

Code-Qualität

Einer der größten Hinderungsgründe gute Tests zu schreiben, ist erfahrungsgemäß schlechte Code-Qualität, u.a. auf Grund von technischen Schulden und Nichtbeachtung der Clean-Code-Prinzipien. Diese machen das Schreiben von Tests aufwendig und langsam. Beispiele:

- `Hohe Komplexität`: Riesige und sehr komplexe Klassen: 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, z.B. wenn das Single-Responsibility-Principle als eines der Designprinzipien (SRP) nicht beachtet wurde. Das heißt, die Klasse enthält viele Funktionen für die sie eigentlich nicht zuständig ist. Hier sollten die Abhängigkeiten analysiert, die Gemeinsamkeiten extrahiert und in separate Klassen überführt werden, die dann einzeln auch viel besser testbar sind.
- `Falsch verwendete Vererbung`: Im Informatik-Studium wird Vererbung immer noch als das Fundament der objektorientierten Entwicklung angepriesen. Aber wird Vererbung

in der Praxis tatsächlich immer richtig verwendet? Ist das Liskovsche-Substitutionsprinzip immer gegenwärtig, wenn Entwickler programmieren? Entwicklern wird vermittelt, dass Code-Duplizierung vermieden werden soll und Vererbung der richtige Weg wäre. Vererbung ist die engste Form der Kopplung. Angesicht der Bedeutung der Änderbarkeit von Software ist jedoch eine lose Kopplung zu bevorzugen (Composition-over-Inheritance). Kompositionsklassen sind die eindeutig bessere Wahl, um Code-Duplizierung und eine enge Kopplung zu vermeiden und zudem besser, isoliert zu testen.

- `Statische Klassen`: Die Namen dieser Klassen enden meist auf `Util` oder `Commons` **(Listing 2)**. Es ist sehr einfach diese zu schreiben und auf diese im Code zu referenzieren. Schwierig wird hier aber das Testen vor allem für die Klassen, welche diesen statischen Code verwenden **(Listing 4)**. 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 Designschwäche, die darauf hinweist: Hier ist eine Methode, die eigentlich zu einer anderen Klasse gehört. Statische Klassen mutieren oft zu riesigen Monstern, einer Halde für Methoden, die Entwickler sinnvoll für Andere halten. Man macht sich oft wenig Gedanken, wo diese statischen Methoden eigentlich hingehören. Im Idealfall zur Klasse, welche die statische Methode aufruft. Statische Klassen enthalten prozeduralen Code, alles andere macht zumindest keinen Sinn. Wir bewegen uns mit Java allerdings in der objektorientierten Welt. Passt prozeduraler Code zum Unit-Testing? Beim Unit-Test soll über Instanziierung ein Stück der Applikation isoliert testen. Dabei wird versucht, alle Abhängigkeiten durch Mocks etc. zu ersetzen. Prozeduraler Code kennt aber so ein Wiring gar nicht, es gibt keine Objekte, Code und Daten, die separierbar sind. Darum ist das Testen hier so schwierig **(Listing 3)**.

(Listing 2)

```
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";
  }
}
```

Der Test **(Listing 3)** für die Klasse **(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.

(Listing 3)

```
@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
      ...
  }
}
```

Um das nicht-deterministische Verhalten im Test zu berücksichtigen, muss jeweils das Systemdatum 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. Man muss den Code erkennen, um ihn zu testen. Die If-Else-Kaskade macht das Testen noch umfangreicher, da für eine einzelne zu testende Methode vier Testmethoden geschrieben werden müssen.

(Listing 4)

```
public class FlyerHeadlineGenerator {

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

Es soll bestimmt werden, welchen Wert `timeOfDay` bekommt, damit nicht noch einmal die Testaufwände aus **(Listing 3)** dupliziert werden. Die `SpecialDateUtil` Klasse enthält nicht nur einen globalen State, sondern ist auch noch statisch und dadurch nur mit Zusatzframeworks mockbar, wie zum Beispiel Powermock oder einer neueren Mockito-Version.

Um den Code testbarer zu machen, wird der global State aufgelöst und das Objekt `DataTime` als Parameter **(Listing 5)** übergeben. Klasse und Methoden sind nicht mehr statisch und können zum Beispiel als Singleton-Instance weiterleben. Das löst sowohl das Design- als auch das Testproblem.

(Listing 5)

```
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";
  }
}
```

Weitere Beispiele für schlecht testbaren Code:

- Logik im Constructor
- Erzeugen von eigenen Abhängigkeiten durch new Objects innerhalb von Methoden
- Verletzung des Prinzips Law-of-Demeter
- If-Else-Kaskaden im Code
- Abhängigkeiten zu riesigen Kontextobjekten
- Verwenden von statischen Initializern

Clean-Code-Prinzipien helfen solche Fallstricke zu vermeiden, denn diese Prinzipien berücksichtigen bereits Aspekte der Testbarkeit und Codequalität.

Test-First und Test-Driven-Development

Wir hören oft Begrifflichkeiten wie Test-First und Test-Driven-Development (TDD). Irritierenderweise werden diese oft vermischt bzw. gleichgestellt. Es gibt aber einen wesentlichen Unterschied. Bei Test-First wird der Test vor der Implementierung des Produktions-Codes geschrieben, also kein Produktions-Code ohne roten Test. Während Test-First hauptsächlich den zeitlichen Aspekt betrachtet, also wann der Test geschrieben wird, geht TDD darüber hinaus. Genau wie bei Test-First wird wieder mit einem Test begonnen und dann der Produktions-Code implementiert, bis der Test erfolgreich durchläuft. Dann refaktoriert man den Produktions-Code wieder, um eventuelle Prinzipienverletzungen zu eliminieren (YAGNI[1], SoD[2], SRP[3], KISS[4], DRY[5] etc.) und den Code zu verbessern.

Der wesentliche Unterschied liegt also in der wiederkehrenden Refactoring-Tätigkeit. Man sorgt nicht nur dafür, dass der Test durchläuft und die geforderte Funktionalität sichergestellt ist, sondern nutzt auch die neu gewonnene Sicherheit, um den Code kontinuierlich zu verbessern. Test-First ist also ein Bestandteil von Test-Driven-Development, aber Test-Driven-Development ist das ideale Instrument, um den Test-Harnisch zu erstellen bzw. zu erweitern. Schnelleres Feedback zum Code ist die Folge. Schon beim Entstehen des Codes wird dafür gesorgt, dass nicht nur die Funktionalität abgesichert, sondern auch die Codequalität sichergestellt und verbessert wird.

Fazit:

Statische Unternehmensvorgaben wie zum Beispiel Code-Coverage oder Vorgaben zur Testautomatisierung sind sinnvoll. Diese allein helfen aber nicht. Der Wandel vom auftragsbezogenen Abarbeiten fachlicher Anforderungen hin zu verantwortungsbezogenem Entwickeln von Software, geht nur mit der Veränderung der Grundeinstellung. Die Akzeptanz für die Testautomatisierung hat viel mit der 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. 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. Dabei hilft uns die testgetriebene Entwicklung wie bei Test-Driven-Development. Mit der Erkenntnis, dass wir durch testgetriebene Entwicklung sicherstellen können, dass unser Code funktioniert, wir bessere Qualität liefern und wir auf manuelle Testaufwände während der Entwicklung verzichten können, wächst auch die Bereitschaft für das Schreiben von automatisierten Tests. Wir können mit einem guten Gefühl und ohne Angst Software produktiv stellen.

In einer agilen Welt sollten selbstorganisierte Teams fähig sein, der Forderung nach neuen Features selbstbewusst zu begegnen. Die Verantwortung eines Entwicklerteams geht über die Entwicklung neuer Fachlichkeit hinaus und braucht Zeit. Zeit, die durch Testautomatisierung und stetige Verbesserung wiedergewonnen wird. Letztendlich müssen alle Beteiligten diesen Weg gehen wollen. Am Ende gewinnen alle, weil dadurch nachhaltig Schnelligkeit, Qualität und Sicherheit garantiert werden kann.

„Softwaretesting ist 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. Untestable Code - http://bit.ly/3tq8Bya
3. Probleme statischer Methoden - http://bit.ly/3eRm2mE
4. Liskovsches Substitutionsprinzip - http://www.nils-haldenwang.de
5. Why testable Code - http://bit.ly/3lprRJm
6. Clean Code Developer - http://bit.ly/3qXTxWV

Wie skaliert man Agilität? Eine Zeitreise.

Dieser Artikel zeigt auf, wie sich die Organisation von Softwareentwicklung historisch entwickelt hat. Er vermittelt, welche Methoden heute zur Verfügung stehen, um jene Techniken erfolgreich auf größere Teams zu übertragen, die im Kleinen so gut funktionieren.

Agilität steht hier für die breite Palette an Methoden, die letztendlich alle auf jenen Haltungen basieren, wie sie beispielsweise im Manifest für Agile Softwareentwicklung[1] zum Ausdruck gebracht werden. Diese Methoden zeichnen sich dadurch aus, dass sie in der Praxis gut funktionieren. Teams, die einem gut eingespielten Scrum-Prozess folgen, sind in der Lage kontinuierlich hohe Qualität zu liefern und flexibel auf Änderungen zu reagieren. Ebenso real ist aber auch folgende Erkenntnis. Was gut für ein Startup mit 20 Entwicklern funktioniert, taugt nur bedingt für Firmen, die mit tausenden Entwicklern an historisch gewachsenen Produkten arbeiten. Um genau dieses Spannungsfeld soll es hier gehen.

Autor:

Ich habe an der Universität Karlsruhe studiert und dort 1999 meinen Abschluss als Diplom-Informatiker gemacht. Danach war ich zunächst zwei Jahre als Java-Entwickler beim Compart Systemhaus in Böblingen beschäftigt, bevor ich 2001 zur IBM Deutschland Research & Development GmbH gewechselt habe. Dort bin ich seither in wechselnden Rollen in den Bereichen zSystems Hardware und Firmware Entwicklung im Labor Böblingen tätig.

In den letzten Jahren habe ich mich intensiv mit der Frage beschäftigt, wie im Kontext eines komplexen Produkts, der IBM z Systems (aka „Mainframe“), und eines großen, global verteilt agierenden Teams von Entwicklern agile Entwicklungsprozesse gestaltet werden können.

Jetzt ist es mir ein Anliegen, diese Erfahrungen an alle Interessierten weiterzugeben.

Firmen-Webseite: https://www.ibm.com/de-de/marketing/entwicklung/index.html
Emailadresse: edwin.guenthner@de.ibm.com

Sprung in die IT-Steinzeit: Worum ging es damals?

Antiquiert formuliert lautet die Antwort:

- „the problems of achieving sufficient reliability in the data systems which are becoming increasingly integrated into the central activities of modern society”
- „the difficulties of meeting schedules and specifications on large software projects”

Diese Zeilen stammen aus dem Report[2] zur NATO-Konferenz Software-Engineering aus dem Jahr 1968 und markieren den Beginn der systematischen Auseinandersetzung mit der Softwarekrise. Jenem ärgerlichen Umstand, dass die Kosten für die Entwicklung der Software schon damals die Kosten für die Anschaffung der Hardware überholt haben.

Ein erster Ansatz zum Umgang mit diesem Problem wurde 1970 von Winston Royce veröffentlicht. Sein Artikel „Managing the Development of Large Software Systems“[3] beschreibt die Entwicklung von Software in verschiedenen Phasen und wird häufig als erste theoretische Grundlage dessen gesehen, was wir heute als Wasserfallmodell kennen. Bemerkenswerterweise hat Royce bereits damals erkannt, dass die strikte Einteilung in Phasen riskant und fehleranfällig ist. Er schlägt deshalb vor, Kunden möglichst früh einzubinden: „It is important to involve the customer in formal way so that he has committed himself at earlier points before final delivery. To give the contractor free rein between requirement definition and operation is inviting trouble.”

Diese frühen Einsichten hatten jedoch wenig Einfluss auf die schnelle und weite Verbreitung des Wasserfallmodells in der Softwareentwicklung. Die Bemühungen zur Standardisierung durch Behörden der US-Regierung wie dem DOD-STD-2167 aus dem Jahr 1985 legten die Grundlage dafür, dass Softwareentwicklung fast über Jahrzehnte hinweg untrennbar mit dem Wasserfallmodell verknüpft war. Das ändert aber nichts an der frühen Erkenntnis von Royce. Beim Wasserfallmodell besteht ein signifikantes Risiko, nicht das zu liefern, was der Kunde haben will.

Ein Versuch zur Lösung dieses Problems wurde 1979 von Barry Boehm zum ersten Mal vorgestellt, das V-Modell. In diesem Entwicklungsmodell werden ganz dediziert zwei Aspekte betrachtet: Validieren (machen wir das Richtige?) und Verifizieren (machen wir es richtig?). In Deutschland wurde dieser Ansatz früh vom Bundesministerium für Verteidigung aufgegriffen und eigenständig weiterentwickelt. Inzwischen erfolgt die Weiterentwicklung durch den Beauftragten der Bundesregierung für Informationstechnik (CIO-Bund). Die aktuelle Version 2.3 des V-Modells XT[4] wurde im Mai 2019 veröffentlicht und erlaubt inzwischen auch eine iterative Vorgehensweise. Es ist daher ein Irrtum zu glauben, dass diese Ansätze heute ausschließlich aus historischem Interesse Relevanz besitzen. Auch die erwähnten Regierungsbehörden der USA haben frühzeitig erkannt, dass penibles Festhalten an starren Regeln schnell zu hoher Inflexibilität führen kann. Das Problem der hohen Kosten durch das Wasserfallmodell wurden keinesfalls gelöst. Ein bekanntes Paradebeispiel für Wasserfallsoftware ist der Code für das Space-Shuttle der NASA. 17 Fehler in 420.000 Zeilen Code ist vermutlich ein Qualitätsrekord[5]. Aber wer kann oder will über 1.000 US-Dollar pro Zeile Code ausgeben?

Aus der Kostenproblematik ergibt sich eine weitere spannende Frage: Wenn die Entwicklung so viel kostet, wie kann man dann vor der potenziellen Vergabe von Aufträgen das Risiko abschätzen, dass der Lieferant in der Lage ist, die gewünschte Software zum geplanten Termin mit der gewünschten Qualität und innerhalb eines bestimmten Kostenrahmens zu entwickeln?

Ein neuer Blickwinkel

Und wieder kam eine prägende Antwort aus den USA. Am Software-Engineering-Institute (SEI) der Carnegie-Mellon-University (CMU) wurde ab Mitte der 1980er Jahre am Capability-Maturity-Model (CMM) gearbeitet. Ziel war es, den Reifegrad eines Unternehmens, welches Software entwickelt, zu verstehen und zu verbessern. Wer sich bisher keine Gedanken über seinen Entwicklungsprozess gemacht hat, wird sich in diesem Modell auf Stufe 1 Initial finden. Durch geeignete Maßnahmen kann eine Firma sich Stufe für Stufe verbessern, um am Ende idealerweise die (utopische) Stufe 5 Optimizing zu erreichen. Mit jeder Stufe steigt dabei Wissen und Kontrolle über den eigenen Entwicklungsprozess bis hin zum Punkt der kontinuierlichen Verbesserung aller Abläufe. Wir sehen, die Idee des beständigen Nachdenkens „was tun wir eigentlich?" und „wie könnten wir es besser machen?" ist mitnichten eine Erfindung des neuen Jahrtausends und des agilen Manifests. Aber woran liegt es dann, dass wir heute in Blogs, Magazinen und auf Konferenzen zwar von Agilität und Retrospektiven hören, aber nur selten im Zusammenhang mit CMM?

Meiner Meinung nach liegt eine der Ursachen darin, dass diese Methoden zwar Werkzeuge für Entscheider und Projektmanager liefern, aber dem einzelnen Entwickler keine Hilfestellung bieten, um einen guten Job zu machen. Es überrascht wenig, dass dieses Manko nicht erst gestern entdeckt wurde. Am erwähnten SEI entstand unter Watts Humphrey in den 1990er Jahren der Personal-Software-Process (PSP) und später der Team-Software-Process (TSP). Ziel war ganz konkret für einzelne Entwickler bzw. Teams Methoden bereit zu stellen, die es ermöglichen, die verschiedenen Aufgaben wie Entwurf, Implementierung und Testen besser zu organisieren und dadurch verlässlich planbar zu machen. In der Theorie spannende und interessante Modelle, die jedoch in der Praxis , zumindest im deutschsprachigen Raum. Speziell der PSP zeichnet sich durch einen hohen Grad an Formalismus aus. Da werden Lines-of-Code abgeschätzt und später gezählt, immer wieder Checklisten erstellt und erweitert und mit erheblichem Aufwand Statistik betrieben. Konsequent umgesetzt kann man damit klar formulierte Programmieraufgaben mit hoher Qualität lösen.

Der Übergang in die Neuzeit

Aber diejenigen Fragen, die sich in einem realen Arbeitsumfeld stellen, wurden auch vom PSP nicht beantwortet. Glücklicherweise begannen Kent Beck, Martin Fowler und andere Praktiker ebenfalls Mitte der 1990er Jahre an dem zu arbeiten, was als Extreme-Programming (XP) die Sicht auf Softwareentwicklung in gänzlich andere Bahnen lenkte. Auch XP beinhaltet formale Techniken, aber im Mittelpunkt aller Aktivitäten steht der Mensch. Das bedeutet: Basierend auf gemeinsamen Werten suchen sich einzelne Entwickler und Teams diejenigen Methoden, die es ihnen ermöglichen, die gemeinsamen Ziele zu erreichen. Zwei Aspekte sind dabei von zentraler Bedeutung: Flexibilität und humane Arbeitsbedingungen. Das erklärte Ziel von XP ist die Fähigkeit eines Teams, flexibel auf Änderungen reagieren zu können. Aber nicht um jeden Preis. Im Gegenteil: Es geht um die Etablierung nachhaltiger Vorgehensweisen, die dauerhaft aufrechterhalten werden können. Also Vorgehensweisen, die ihre Stärken auch daraus ziehen, dass sie den Menschen als soziales Wesen und Individuum anerkennen. Aus dieser Haltung heraus ergibt sich eine holistische Herangehensweise, die sowohl technische Details als auch soziale Aspekte explizit zu verknüpfen sucht. Alle wesentlichen Merkmale, die wir heute mit Agilität verbinden, u.a. kurze Iterationen, kontinuierliche Integration, offene Kommunikation als Grundlage, beständige Einbeziehung des Kunden usw., hat XP schon vor über 20 Jahren auf den Weg gebracht. Provokant formuliert könnte man sagen, dass die bekannten Konzepte wie der Scrum-Prozess und das Agile Manifest, die nach XP kamen, lediglich unterschiedliche Aspekte von XP weiter verfeinert haben.

Wie bereits eingangs erwähnt, vereint diese Methoden die Tatsache, dass sie in der Regel in der Praxis gut funktionieren. Vielleicht genauso wichtig ist, dass die Zufriedenheit der Teilnehmer an agilen Prozessen meistens als gut, gar sehr gut bewertet wird. Aber wie ebenfalls bereits erwähnt, funktionieren XP oder Scrum nicht in jedem Kontext einfach gut. Das ist auch

kein Anspruch dieser Methoden und Konzepte. Der Scrum-Prozess beschäftigt sich primär mit der Organisation eines Scrum-Teams. Bereits die Frage, ob Scrum darüber hinaus auch ein geeignetes Framework für Entscheider auf Unternehmensebene darstellt, wird heftig diskutiert und die Antworten fallen unterschiedlich aus. Wer Hilfestellungen zur Organisation von Teams mit Tausenden von Entwicklern sucht, wird hier jedenfalls keine befriedigenden Antworten finden.

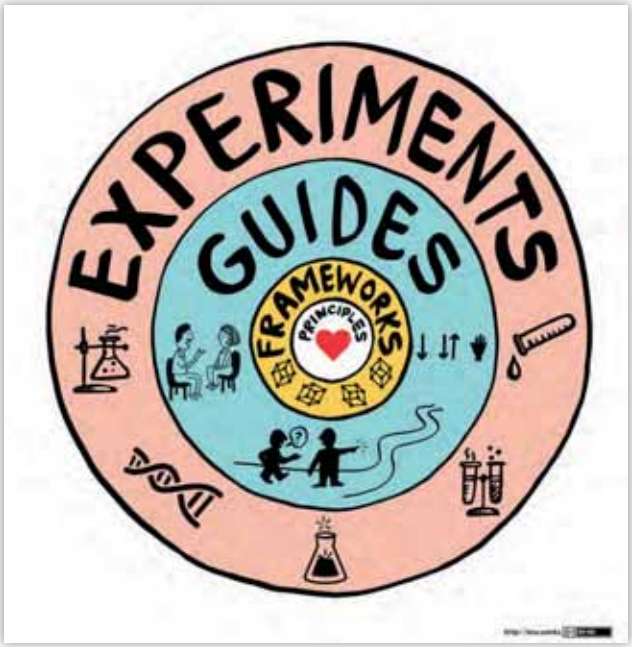
Vom Team zum Konzern

Im Weiteren soll es genau darum gehen: Wie kann man die Vorteile agiler Entwicklungsmodelle auf Teams mit 200 oder 2.000 Personen übertragen? Zwei der bekanntesten Methoden werden im Folgenden vorgestellt, beginnend mit dem Scaled-Agile-Framework, kurz SAFe. SAFe entwickelte sich aus dem Buch Agile-Software-Requirements, welches Dean Leffingwell 2011 veröffentlich hat. Das erklärte Ziel ist es, die bekannten agilen Praktiken mit dem Konzept des Lean-Management zu verbinden und das Management komplexer Portfolios von Produkten möglich zu machen.
Dreh- und Angelpunkt bei SAFe ist das SAFe-Big-Picture[6] **(Abb. 1)**:

SAFe for Lean-Enterprises 5.0: Portfolio SAFe (Abb. 1)

Im Big-Picture finden sich alle Fragestellungen, für die SAFe Hilfestellungen anbietet. Zu jedem Element in dieser Grafik stellt SAFe ausführliche Artikel bereit, die das spezifische Thema detailliert besprechen und innerhalb des Frameworks einordnen. SAFe legt großen Wert darauf, seine Nutzer umfassend zu informieren und ihnen eine vollständige Landkarte bereitzustellen. Um nur ein Beispiel zu nennen: SAFe definiert als zentrale Komponente den Agile-Release-Train (ART). Dabei handelt es sich um ein langlebiges Team agiler Teams, welches zusammen mit weiteren Akteuren inkrementell Solutions innerhalb eines ValueStreams entwickelt und ausliefert. Der ART steht in Beziehung zu verschiedenen agilen Organisationsformen wie zum Beispiel XP, Scrum, Kanban und andere. Die Verwaltung der Arbeitspakete erfolgt wenig überraschend durch Team Backlogs, die auf höherer Ebene durch zentrale Program-Backlogs koordiniert werden. Das Big-Picture definiert auf diese Weise alle relevanten Elemente, die in einer großen Organisation betrachtet werden sollten.

Diese Wegmarken werden ausführlich beschrieben und von unterschiedlichen Richtungen heraus betrachtet. Insgesamt ist SAFe daher besonders für diejenigen geeignet, die sich am Anfang der Reise befinden. Wer sich eher in der Welt des Wasserfallmodells und ausführlich beschriebener Prozesse verortet, für den bietet SAFe konkrete Hilfestellungen mit hohem Detailgrad. Für jene, die schon länger auf der Reise sind und nach neuen Wegen suchen, um eine bereits in vielen Teilen agile Organisation auf die nächste Stufe der Evolution zu heben, lohnt sich der Blick auf andere Frameworks. Ein Beispiel wäre LargE-Scale-Scrum (LeSS), welches seit 2013 von Bas Vodde entwickelt wird. Auch für LeSS gibt eine Übersichtsgrafik[7], die alle Themengebiete und Grundbestandteile der Methode übersichtlich darstellt. Besonders interessant ist das LeSS Complete Picture **(Abb. 2)**.



The LeSS-Complete-Picture (Abb. 2)

Was dabei sofort auffällt sind Minimalismus und Prioritäten. Die Essenz von LeSS sind nicht dutzende verschiedener Themenblöcke, sondern die vier Bausteine Prinzipien, Frameworks, Anleitungen und Experimente. Ganz offensichtlich nimmt den meisten Raum die Kategorie Experimente ein.

Ziel von LeSS ist es nicht, die agile Welt komplett neu zu erfinden. Es geht vielmehr darum, die bekannten und etablierten Prinzipien und Elemente des Scrum-Teamprozesses im Kontext einer größeren Organisation zu transportieren und zu leben. In diesem Sinne ist es eine (Rück-)Besinnung darauf, wie Agilität in kleineren Teams erfolgreich gestaltet wird. Die nächsten beiden Kreise machen klar, dass auch LeSS den Nutzern vielfältige Hilfestellungen an die Hand gibt, um jene Kernideen von Scrum in große Teams zu tragen. Am wichtigsten jedoch ist die Ansage: „Du musst selbst herausfinden, was für Dich das Richtige ist". Anders formuliert besteht das eigentliche Ziel von LeSS darin, die Nutzer

in die Lage zu versetzen, sinnvolle Fragestellungen und Thesen zu entwickeln, um diese dann durch Experimente zu beantworten bzw. zu validieren. LeSS will nicht vorschreiben, wie eine Organisation umgebaut oder Prozesse gestaltet werden sollten. Ziel ist es vielmehr, den Anwender in die Lage zu versetzen, diejenigen individuellen Fragen zu finden, deren Beantwortung genau die aktuelle Situation nachhaltig verbessern wird.

Neben SAFe und LeSS gibt es eine ganze Reihe weiterer Methoden: Disciplined-Agile-Delivery (DAD), Nexus, Enterprise-Scrum, Scrum@Scale, um nur einige zu nennen. Dabei ist sicherlich zutreffend, dass alle guten Modelle versuchen, den Anwendern nahe zu bringen: „Grau ist alle Theorie".

Nach einer eher theoretischen Betrachtung wird nun konkret dargelegt, wie Skalierung von Agilität im Arbeitsumfeld des Autors aussieht.

Exkurs: IBM-System-Z

Das Team System-Z-Firmware-Development (IBM-Entwicklungslabor in Böblingen) entwickelt jegliche Software, die ein IBM-Z-Mainframe-Kunde zusammen mit der Hardware erhält. Das ist der wesentliche Unterschied im Vergleich zu Betriebssystemen wie z//VM oder z//OS, die getrennt vertrieben werden. Die Definition des Begriffs Firmware ist deswegen so wage, weil ein breites Spektrum an Komponenten betreut wird.

Das beginnt bei Kernfunktionen, wie dem BIOS des Mainframes (geschrieben in C++, PL8, PLX, IBM Z Assembler), geht über umfangreiche Steuer- und Regelsoftware zur Kontrolle der physikalischen Hardware (C, C++) und endet bei komplexer Software zur Verwaltung logischer Partitionen/Virtueller Maschinen, in denen die Kunden ihre Betriebssysteminstanzen betreiben (Java-Backend, JavaScript-Frontend). Passend zu diesem hohen Grad an technischer Diversität gestaltet sich die organisatorische Seite. Das bestehende Team aus etwa 500 Mitarbeitern verteilt sich global über Standorte in Deutschland, den USA und Indien. Das Umfeld ist sehr heterogen, mit jungen Uni-Abgängern auf der einen Seite und erfahrenen Mitarbeitern, die manchmal seit 30 Jahren die gleiche Komponente betreuen, am anderen Ende des Spektrums. Die große Herausforderung, die sich für das Team stellt, ergibt sich aus dem Ende von Moore's-Law.

IBM entwickelt die IBM-Z-Mainframes kontinuierlich weiter. In der Regel wird alle 2 bis 3 Jahre die nächste Generation Mainframes aufgelegt. Das bedeutet konkret, dass beispielsweise die Prozessor-Chips komplett neu entwickelt werden. Bis vor einigen Jahren ergab sich daraus automatisch eine signifikante Steigerung der Leistung des Gesamtsystems. Für viele Kunden war das ein wesentlicher Grund für den Kauf eines neuen Mainframes. In diesem Modell ist die primäre Aufgabe von Firmware, die Geschwindigkeitsvorteile und neuen Funktionen der Hardware effizient nutzbar zu machen. Dementsprechend diktiert die Verfügbarkeit der neuen Hardware die Zeitpläne für die Entwicklung der Firmware. Insgesamt geht es hier um tausende von Entwicklern, die über Jahre darauf hinarbeiten, zu einem bestimmten Termin die nächste Systemgeneration auf den Markt zu bringen. Diese Vorgehensweise hat für mehr als 20 Jahre sowohl für IBM als auch deren Kunden gut funktioniert. John Markoff brachte es bereits 2015 präzise auf den Punkt: „Smaller, Faster, Cheaper, Over"[8]. Moore's-Law gilt jedoch nicht mehr, denn ein neuer Prozessor in der neuesten Technologie ist nicht automatisch viel schneller als das Vorgängermodell.

IBM legt Wert darauf, dass jede nächste Generation mehr Leistung erbringt als das aktuelle Modell. Nur ist eben der Zuwachs nicht mehr in den Größenordnungen machbar, die noch vor 5 oder 10 Jahren üblich waren. Daraus folgt eine hohe Notwendigkeit, neue Kaufanreize zu schaffen. Konkret bedeutet das, dass der IBM-Z-Mainframe auf allen Ebenen attraktiver werden muss. Zum Beispiel dadurch, dass die Firmware nicht mehr nur die grundlegenden Funktionen der unterliegenden Hardware verfügbar macht, sondern darüber hinaus komplexe Funktionen bereitstellt, die das Management des Mainframes in allen Bereichen für die Kunden massiv vereinfacht und verbessert.

Dabei ist es von großer Bedeutung, Lösungen nicht im Elfenbeinturm zu entwerfen, die an den vielfältigen Bedürfnissen der Kunden vorbeigehen. Um das Richtige zu liefern, setzt IBM auf die IBM-Design-Thinking-Methode[9]. Deren Ziel ist es, die Benutzung jedweden Produkts auf die Bedürfnisse der Endanwender zu optimieren.

Zusammengefasst ergibt sich daher für das Team die Notwendigkeit, einerseits vorausschauend die wichtigen technischen Trends frühzeitig zu erkennen und mit hoher Qualität in das Produkt einfließen zu lassen. Das geschieht in intensiver Zusammenarbeit mit Kunden, aber unter Beibehaltung der erwähnten langen Release-Zyklen.

Es ist klar, dass die existierenden Prozesse und Strukturen, die sich sehr stark am Wasserfallmodell orientieren und seit 15 - 20 Jahren immer weiter verfeinert wurden, keine angemessene Grundlage für die notwendige Neuausrichtung des Teams bilden.

Agilität für den Mainframe

Die Antwort basiert auf drei Säulen:

1. Seit über 10 Jahren werden regelmäßig Schulungen zu agilen Entwicklungsmethoden durchgeführt. Ziel ist es, die breite Masse der Entwickler mit neuen Konzepten und Ansätzen vertraut zu machen. Dabei wurde und wird großer Wert daraufgelegt, die verschiedenen existierenden Rollen, wie zum Beispiel Entwicklung, Teamführung und Projektmanagement gleichermaßen anzusprechen und gemeinsam neue Wege zu entwickeln.

2. Die bereits erwähnte Methode IBM-Design-Thinking mit dem Ziel, neue Ideen möglichst schnell mit Kunden zu prüfen und weiterzuentwickeln. Besonders wichtig ist dabei, die Aktivitäten der Design-Teams möglichst eng mit der eigentlichen Produktentwicklung zu verbinden.
3. Einer dediziert neuen Organisationsform, basierend auf dem Spotify-Modell[10].

Oberflächlich betrachtet ist das Spotify-Modell eine Methode, um die Struktur einer größeren Organisation so zu gestalten, dass ein hohes Maß an Autonomie entstehen kann, während gleichzeitig darauf geachtet wird, dass die einzelnen Teams auf die gleichen Ziele hin ausgerichtet sind. Auf der untersten Stufe stehen dabei die sogenannten Squads. Das sind agile Teams, die idealerweise die vollständige Verantwortung für eine autarke Komponente haben, sich also holistisch um Design, Entwicklung und Test kümmern. Alle Squads, die am gleichen Produkt arbeiten, kommen im Tribe zusammen. Weiterhin existieren innerhalb eines Tribes die sogenannten Chapter.

Deren Sinn besteht darin, den Austausch über bestimmte Interessengebiete innerhalb des Tribe voranzubringen. Ein gängiges Beispiel sind Chapter für Frontend- bzw. Backend-Entwicklung. Daneben gibt es noch Guilds. Das sind Interessenverbände, die über mehrere Tribes hinweg agieren. Das alles fließt in dem Team im sogenannten Alliance-Board zusammen. Hier kommen regelmäßig Management, Tribe-Leader, Projektmanagement und technische Führungskräfte zusammen, damit die gemeinsamen Ziele nicht aus dem Fokus geraten.

Entscheidend ist aber an dieser Stelle die Haltung aller Beteiligten. Das primäre Ziel liegt in der Schaffung neuer Freiheitsgrade. Squad ist nicht nur eine andere Bezeichnung für Team. Es geht vielmehr darum, den Mitgliedern der Squads die Möglichkeit zu geben, die eigenen Interessen und Stärken konstruktiv einzubringen. Das gelingt weder über Nacht noch mit wenigen großen Würfen. Vielmehr sind Kontinuität und Beständigkeit gefragt. Gleichzeitig bedarf es hier der Balance. Die Basisaufgaben müssen weiterhin mit der Qualität erledigt werden, die die Kunden erwarten. Andererseits soll jedes Mitglied des Teams die Möglichkeit haben, neue Ansätze zu erproben, zu verfeinern und zu verwerfen.

Es ist nicht überraschend, dass man in der Realität irgendwo in der Mitte landet. Es gibt Gruppen, die sich in der alten Welt wohler fühlen und sich nur mit kleineren Schritten auf diese neue, weniger formal definierte Zukunft einlassen. Aber es existieren auch Squads, die das neue Modell jeden Tag neu leben und dadurch in der Lage sind, die potentiellen Konflikte zwischen Innovationsfreude, Qualität und Termindruck weitestgehend aufzulösen. Außerdem zeigt sich, Rosinenpicken ist erlaubt und sinnvoll. Projektmanager orientieren sich beispielsweise an verschiedenen Elementen aus dem SAFe-Kosmos, was die Neugestaltung des Portfolio-Backlog-Managements angeht.

Die Möglichkeiten zur Einflussnahme durch das höhere Management sind eher begrenzt. Es geht in erster Linie darum, positive Anreize zu schaffen, da ein erfolgreicher Kulturwandel hin zum autonomen Entwickler nicht von oben verordnet werden kann. Gelebte Führungsverantwortung bedeutet hier vor allem, aktiv an der Lösungsfindung mitzuwirken, wenn Konflikte mit externen Gruppen innerhalb der IBM offenbar werden, die ihre Ursache in unterschiedlichen Grundhaltungen zur Agilität haben.

Fazit:

Der wichtigste Aspekt bei der Organisation von Softwareentwicklung in großen Teams liegt nach Ansicht des Autors nicht darin, in einem Video, einem Fachartikel oder einem guten Buch die allgemein gültigen Regeln und Antworten zu suchen, die alle Probleme auf Jahre hinaus lösen. Das Ziel ist stattdessen, die richtigen Fragen zu identifizieren und herauszufinden, wie die eigene Organisation tickt und welche ihrer Stärken dazu geeignet sind, die aktuellen Herausforderungen zu meistern. Aber auch immer wieder sich klar zu machen, dass auch gute, funktionierende Lösungen nicht für die Ewigkeit sind, sondern regelmäßig auf den Prüfstand gehören.

Zum Abschluss noch ein Gedankenexperiment, welches dem Konzept des „Intentional Living"[11] entlehnt ist:

Tun Sie die Dinge, weil Sie es können? Oder agieren Sie, weil Sie sich bewusst entschieden haben, genau so zu handeln?

Sowohl in der Software-Entwicklung als auch im realen Leben kann man sich treiben lassen, um dann dort zu landen, wohin die Umstände einen führen. Man kann sich aber auch überlegen wo man hin möchte, um sich bewusst für den Weg zu entscheiden, der in Richtung dieses Ziels führt.

Welchen Weg gehen Sie heute? Und ist das der Weg, den Sie gehen wollen?

Quellen:

1 Manifest: https://bit.ly/3r3xj5V
2 NATO-Report: https://bit.ly/30ZUSC6
3 Artikel Winston Royce: https://bit.ly/3lw2DsF
4 V-Modell XT: http://bit.ly/2P97AM9
5 Qualitätsrekord NASA-Code: http://bit.ly/3qXAwUl
6 SAFe-Big-Picture: http://bit.ly/3seSvao
7 LeSS: https://bit.ly/3vNmici
8 The New York Times, by John Markoff: http://nyti.ms/3vQLLBC
9 IBM: https://ibm.co/393YDKS
10 Spotify-Modell: http://bit.ly/3vNnJYe
11 Intentional Living: https://bit.ly/3r939OD

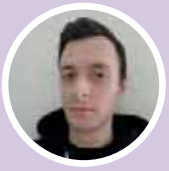
#JAVAPRO #Blockchain #Krypto #Ethereum

Dezentrale Apps mit Ethereum

Dieser Artikel gibt einen Einblick in die Technologie Ethereum und zeigt, wie in Java mithilfe der Web3j-Library mit einem Ethereum-Netzwerk interagiert werden kann. Zusätzlich wird ein Beispiel gegeben, bei dem durch die Programmiersprache Solidity ein Smart-Contract auf Ethereum ausgeführt wird, um so einen Einblick in die Entwicklung von Decentralized-Applications (DApps) zu bekommen.

Autor:

Kevin Wittek ist Testcontainers-Co-Maintainer, Oracle-Groundbreaker-Ambassador und aktives Mitglied der Java- und Open-Source-Community. Er schreibt gerne MATLAB- und Python-Programme, um seine Frau bei der Durchführung von verhaltenswissenschaftlichen Experimenten mit Tauben zu unterstützen. Zudem spielt er E-Gitarre und ist in seinem zweiten Leben Musiker. Nach langjähriger Tätigkeit in der Industrie als Softwareentwickler promoviert Kevin aktuell zum Thema Verifikation von Smart-Contracts und leitet das Blockchain-Lab am Institut für Internet-Sicherheit in Gelsenkirchen an der Westfälischen Hochschule.



Marius Weise und Dominik Krakau sind Studenten an der Westfälischen Hochschulen in Gelsenkirchen im Bachelorstudiengang Informatik. Sie unterstützen außerdem das Blockchain-Lab im Institut für Internet-Sicherheit als Studentische Hilfskräfte.

Ethereum ist ein dezentrales System und basiert auf der Blockchain-Technologie mit der besonderen Eigenschaft, dass dort Programme angelegt, verwaltet und ausgeführt werden können. Diese Programme werden Smart-Contracts genannt und in Ethereum mithilfe einer virtuellen Maschine, der Ethereum-Virtual-Machine ausgeführt. Diese VM führt Ethereum-Byte-Code aus, der wiederum aus verschiedenen speziellen Hochsprachen kompiliert werden kann. Die erste und bekannteste dieser Ethereum-Programmiersprachen ist Solidity.

Mithilfe von Solidity können Anwendungen entwickelt werden, sogenannte Smart-Contracts, die in einem Ethereum-Netzwerk bereitgestellt und ausgeführt werden. Solidity ähnelt von der Syntax einer Mischung der Sprache JavaScript und C, besitzt allerdings auch viele Charakteristika einer objektorientierten Sprache. Neben Solidity existieren noch weitere Programmiersprachen, die in Ethereum-Byte-Code kompiliert werden können, wie zum Beispiel Viper. Diese befinden sich allerdings überwiegend in einem frühen und experimentellen Zustand und sind aktuell nicht für den Produktiveinsatz empfohlen.

DApps entwickeln mit Web3j

Eine dezentralisierte Anwendung (DApp) zeichnet sich dadurch aus, dass sowohl die Daten als auch die Ausführung des Programmcodes nicht durch eine zentrale Instanz kontrolliert werden, wie zum Beispiel einem Unternehmen. Vielmehr werden Zustandsänderungen automatisch in einem (Peer-to-Peer) Netzwerk verteilt und synchronisiert. Das Vertrauen in die Korrektheit von einer zentralen Institution (Institutional-Trust) in ein heterogenes Netzwerk sowie die Annahme der Korrektheit der eingesetzten Protokolle (Distributed-Trust) wird also verschoben.

Mithilfe von Ethereum kann ein solches Blockchain-basiertes Peer-to-Peer Netzwerk aufgebaut werden. Hierbei kann grundsätzlich zwischen öffentlichen (Internet) und privaten (Intranet) Netzwerken unterschieden werden. Das bekannteste Beispiel für ein öffentliches Ethereum-Netzwerk ist das Ethereum-Mainnet, aber es existieren auch weitere öffentliche Testnetzwerke, wie Ropsten, Kovan oder Rinkeby. Des Weiteren existieren auch domänenspezifische Netzwerke wie zum Beispiel bloxberg[1] im Kontext von Wissenschaft und Forschung.

Darüber hinaus besteht auch die Möglichkeit ein lokales und privates Netzwerk für eigene Testzwecke aufzubauen. Ein solches Testnetzwerk kann sehr einfach mit dem Tool Ganache[2] erstellt werden. Ganache generiert automatisch eine Anzahl an Account-Adressen, die bereits über die entsprechende Menge an Ether-Tokens für die Durchführung von Transaktionen verfügen. In Ethereum werden diese Tokens verwendet, um die Ausführung von Smart-Contract-Funktionen zu bezahlen.

Die Entwicklung einer DApp lässt sich an einer einfachen Beispielanwendung zeigen. In diesem Fall soll eine öffentliche Umfrage durchgeführt werden. Die Umfrage enthält eine Fragestellung und eine binäre Antwortmöglichkeit. Die Anwendung besteht vornehmlich aus zwei Komponenten: Zum einen ein Ethereum-Smart-Contract, der in seinen Eigenschaften Ähnlichkeiten zu klassischen Server-Anwendungen besitzt (im speziellen Ähnlichkeiten zu Serverless-Functions) und einer Java-Client-Anwendung. Mit dieser Java-Client-Anwendung und mit dem Smart-Contract - und somit transitiv mit dem Ethereum-Netzwerk - kann so interagiert werden, um an der Umfrage teilzunehmen. Der nachfolgende Code stellt die Grundstruktur des Smart-Contracts dar **(Listing 1)**.

(Listing 1)

```
pragma solidity ^0.5.0;

contract Poll {

   address[] voters;
   uint votersCount;

   mapping(address => bool) voterHasVoted;

   string pollName;

 mapping(uint => PollOption) pollOptions;

    struct PollOption {
        string name;
        uint voteCount;
    }

// […]

}
```

Innerhalb des Contracts können, ähnlich wie in einer Java-Klasse, verschiedene Variablen in Feldern abgelegt werden. In dem Array `voters` werden alle Adressen der teilnehmenden Wähler abgespeichert. Mit der map `votersHasVoted`, für die Prüfung von Ethereum Adressen auf einen Wahrheitswert, kann später performant geprüft werden, ob ein Wähler bereits abgestimmt hat, ohne über das gesamte `voters` Array zu iterieren. Desweiteren wird ein `struct` definiert. Auswahlmöglichkeit `PooOption` besitzt einen Namen `string` und einen Zähler `uint` zur Speicherung der für diese Wahloption abgegebenen Stimmen.

Ganz ähnlich zu objektorientierten Programmiersprachen wie Java wird ein Konstruktor benötigt, der die Smart-Contract Felder initialisieren kann **(Listing 2)**. In der aktuellen Version von Solidity lassen sich Arrays noch nicht als Funktionsargumente verwenden. Deswegen soll in diesem Beispiel der Einfachheit halber nur auf die Initialisierung des Smart-Contracts mit zwei Auswahlmöglichkeiten eingegangen werden. Der Umfragename und die beiden Auswahlmöglichkeiten werden als Parameter übergeben und initialisieren mit diesen die structs `PollOption` und `Poll`.

(Listing 2)

```
    constructor(string memory _pollName, string memory _firstPol-
lOption, string memory _secondPollOption) public {

        votersCount = 0;
        pollName = _pollName;

        PollOption memory questionnaireOption1;
        questionnaireOption1.name = _firstPollOption;
        questionnaireOption1.voteCount = 0;
        pollOptions[0] = questionnaireOption1;

        PollOption memory questionnaireOption2;
        questionnaireOption2.name = _secondPollOption;
        questionnaireOption2.voteCount = 0;
        pollOptions[1] = questionnaireOption2;

}
```

Des Weiteren wird eine Funktion `vote` benötigt, mit der die Fachlogik abgebildet werden kann. In dieser wird geprüft, ob die Adresse des Funktionsaufrufers schon abgestimmt hat. Sollte dies nicht der Fall sein, wir der `votecount` der gewählten Auswahlmöglichkeit um 1 inkrementiert **(Listing 3)**.

(Listing 3)

```
    function vote(uint choose) public {
        if(!voterHasVoted[msg.sender]) {
            voters[votersCount] = msg.sender;
            (choose == 0)
                ? questionnaire.pollOptions[0].voteCount += 1
                :  questionnaire.pollOptions[1].voteCount += 1;
             voterHasVoted[msg.sender] = true;
             votersCount++;
        }
    }
```

Um aus der Java-Anwendung herauslesend auf den Zustand des Smart-Contracts zugreifen zu können, müssen Getter mit atomaren Datentypen hinzugefügt werden. Eine Rückgabe von komplexeren Datentypen, wie zum Beispiel Arrays, ist mit der aktuellen Solidity Version nicht möglich **(Listing 4)**.

(Listing 4)

```
    function getPollName() public view returns(string memory) {
        return questionnaire.name;
    }
    function getPollOption(uint optionIndex) public view
returns(string memory) {
        return questionnaire.pollOptions[optionIndex].name;
    }
    function getPollOptionVoteCount(uint optionIndex) public view
returns(uint) {
        return questionnaire.pollOptions[optionIndex].voteCount;
    }
```

Die Ethereum-Online-IDE[3] stellt vermutlich die einfachste Möglichkeit dar, Smart-Contracts online zu entwickeln, zu kompilieren und zu testen. Eine stärker professionalisierbare Variante stellt die Nutzung von Truffel[4] als lokales Solidity-Build-Tool dar. Zum Testen des Smart-Contracts wird hierbei allerdings ein lokales Ethereum-Netzwerk, wie zum Beispiel Ganache benötigt. Dank der Open-Source-Library Web3j ist die Integration des Smart-Contracts in die Java-Anwendung mit wenig Aufwand möglich und Web3j kann einfach als Maven- oder Gradle-`Dependency` zum eigenen Projekt hinzugefügt werden **(Listing 5)**. Darüber hinaus wird ebenfalls ein Maven- und Gradle-`Plugin` angeboten, mit dem es möglich ist, Solidity-Quelltext in Ethereum-Byte-Code zu übersetzen **(Listing 6)**.

(Listing 5)

```
<dependency>
    <groupId>org.web3j</groupId>
    <artifactId>core</artifactId>
    <version>5.0.0</version>
</dependency>
```

(Listing 6)

```
<plugin>
    <groupId>org.web3j</groupId>

<artifactId>web3j-maven-plugin</artifactId>
    <version>4.5.11</version>
    <configuration>
        <packageName>contract</packageName>
        <sourceDestination>src/main/java/generated</sourceDesti-
nation>
        <nativeJavaType>true</nativeJavaType>
        <outputFormat>java,bin</outputFormat>
        <soliditySourceFiles>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.sol</include>
            </includes>
        </soliditySourceFiles>
        <outputDirectory>
            <java>src/main/java</java>
            <bin>src/main/resources</bin>
            <abi>src/main/resources</abi>
        </outputDirectory>
        <contract>
            <includes>
                <include>greeter</include>
            </includes>
            <excludes>
                <exclude>mortal</exclude>
            </excludes>
        </contract>
        <pathPrefixes>
            <pathPrefix>dep=../dependencies</pathPrefix>
        </pathPrefixes>
    </configuration>
</plugin>
```

Im Falle von Maven geschieht dies anhand des Generate-Sources-Maven-Goals, welches nicht nur den Solidity-Quelltext kompiliert, sondern auch eine Java-Proxy-Klasse für den Smart-Contract erstellt, die direkt als API fungiert.

Die Interaktion mit dem auf der Ethereum-Blockchain veröffentlichten Smart-Contract geschieht in der Regel per HTTP oder WebSocket, wobei die Ethereum-API JSON-RPC aufgerufen werden kann. Diese technischen Details werden bei der Verwendung von Web3j allerdings nahezu vollständig gekapselt. Für die Interaktion mit einem Ethereum-Netzwerk muss ein Web3j-Client mit der Adresse eines Netzwerksknotens initialisiert werden: Zum Beispiel `localhost:8545`, bei der Verwendung eines lokalen Testnetzwerks mit Ganache **(Listing 7)**.

(Listing 7)

```
Web3j web3j = Web3j.build(new HttpService("http://local-
host:8545"));
```

Eine Besonderheit bei dezentralen Anwendungen ist die Tatsache, dass auf klassische User-Accounts verzichtet wird. Stattdessen geschieht die Identifikation eines Users mithilfe einer Wallet. Eine Wallet beschreibt hierbei einen Software- oder Hardware-Speicher für Schlüsselmaterial, in der Regel öffentliche und private Schlüssel für die Verwendung von asymmetrischer Kryptographie. Web3j erlaubt es, verschiedene Wallets zu laden.

Um das Beispiel zu vereinfachen, soll hier allerdings ein privater Schlüssel als hartkodierter String verwendet werden - diesen Ansatz biete NIEMALS in Produktion verwenden. Diese Wallet kann nun von Web3j dafür verwendet werden, Transaktionen zu signieren, aber auch die Transaktionskosten mit der in der Wallet hinterlegten Adresse zu bezahlen. Die Adresse kann aus dem öffentlichen Schlüssel eindeutig abgeleitet werden. Das deployen eines Smart-Contracts ist ebenfalls eine Transaktion **(Listing 8)**.

(Listing 8)

```
Credentials credentials = Credentials.create("privatekey");
Poll survey = Poll.deploy(web3j, credentials, new DefaultGasPro-
vider(), "Coffee or Tee?" , "Coffee", "Tee").send();
```

Durch das deployen des Smart-Contracts im Ethereum-Netzwerk wird gleichzeitig ein Java-Objekt vom Typ `Poll` initialisiert. Dieses kann direkt verwendet werden, um Funktionen des Smart-Contracts aufzurufen.

Grundlegend existieren in einem Ethereum-Smart-Contract zwei Klassen von Funktionen: Zum einen Transaktions-Funktionen, welche den Zustand eines Smart-Contracts ändern und daher mit Rechenaufwand und Gebühren belegt sind - sogenannten Gaskosten, die mit Ether bezahlt werden können.

Die `vote` Funktion ist ein Beispiel für diese Klasse. Zum anderen `view` und `pure` Funktionen, die entweder den Zustand eines Smart-Contracts auslesen (`view`) oder Funktionen im engeren mathematischen Sinne sind und nur auf transienten Daten arbeiten (`pure`). Die Getter im Poll-Smart-Contract sind Beispiele für `view` Funktionen. Der Aufruf aus Java heraus erfolgt allerdings vollkommen transparent **(Listing 9)**.

(Listing 9)

```
// get name of the poll
var result = poll.getPollName().send();

// get results for first option
result = poll.getPollOption(BigInteger.valueOf(0)).send();

// vote for option 2
poll.vote(BigInteger.valueOf(1)).send()

// get results for second option
result = poll.getPollOption(BigInteger.valueOf(1)).send();
```

In diesem Beispiel wird die synchrone Web3j-API verwendet. Darüber hinaus stellt Web3j allerdings auch ein asynchrones und ein reaktives Programmiermodell zur Verfügung.

Fazit:

Obwohl Ethereum auf den ersten Blick komplex und ungewohnt erscheinen mag, soll dieses Beispiel zeigen, dass viele Konzepte und Methodiken ganz ähnlich zu bekannten Technologien funktionieren. Die Programmierung von Smart-Contracts bietet viele Ähnlichkeiten zur objektorientierten Programmierung und die Interaktion mit einem, in einem Ethereum-Netzwerk deployten Smart-Contract, ist nicht weit von der Integration gegen einen klassischen Web-Service entfernt. Der Einsatz von Blockchain-Technologie im allgemeinen und Ethereum im speziellen sind sicherlich nicht in jedem Software-Projekt notwendig, aber das Verständnis der Konzepte und ein Basiswissen in Umgang und Umsetzung sind ein wertvoller Bestandteil im Werkzeugkasten eines modernen Softwerkers.

Quellen:

1 Bloxberg: https://bloxberg.org/
2 Ganache: http://bit.ly/3tHtkgR
3 Ethereum-Online-IDE: http://bit.ly/3saAhqy
4 Truffle: http://bit.ly/3tP3ZSx

Java Cloud-Native

Microservice Entwicklung

Online-Trainings

GraalVM · MicroProfile · Quarkus · Micronaut · Payara · NEW! MicroStream

Microservice-Entwicklung mit Java

Cloud-Computing und Microservices verändern die gesamte Softwarebranche. Nach einer O'Reilly Umfrage entwickeln bereits 30% aller großen Unternehmen Microservices. Knapp 25% nutzen Microservices noch nicht. Die Hauptvorteile für Organisationen sind eine wesentlich bessere Feature-Flexibilität, schnellere Reaktionsmöglichkeit auf neue Anforderungen, bessere Skalierbarkeit sowie eine bessere Produktivität und Zufriedenheit der Entwickler. Mehr als 80% aller Entwickler weltweit gehen davon aus, dass sich Microservices als Standardarchitektur etablieren werden, jedoch besitzt die Hälfte aller Entwickler noch nicht über das nötige Know-how. Um das zu ändern, starten wir jetzt ein einzigartiges Microservice Kursprogramm für Java Entwickler.

Auch Java verändert sich gerade radikal. Die JVM sowie Full-Stack-Frameworks wie Java EE und Spring sind für den Betrieb von Microservices ungeeignet. Das Problem sind zu lange Startzeiten und zu hoher Ressourcenverbrauch. bei den einzelnen Services. Deshalb gibt es jetzt für Microservices einen völlig neuen Java-Technologie-Stack. Anstelle von Java EE oder Spring treten moderne Microservice-Frameworks wie Quarkus, Micronaut oder Helidon. Damit sind Startzeiten in Millisekunden, geringer Speicherverbrauch und mit einer MicroStream Integration bis zu 1000 Mal schneller Datenbankzugriffe möglich. Mit unseren Online-Kursen erhalten Sie das gesamte Know-how, das Sie für die Entwicklung von Microservices in Java brauchen.

Weltweite Experten für Technologie-Training und -Beratung

Fast Lane

Verwirklichen Sie Ihre Karriereziele, erweitern Sie Ihre Kenntnisse über modernste IT-Lösungen und weisen Sie Ihr Know-how durch eine IT-Zertifizierung nach! Fast Lane bietet IT-Schulungen für führende Technologieanbieter wie AWS, Cisco, Google, Microsoft, NetApp, VMware und andere weltweit an. Ob als IT-Training im Klassenraum, Online-Kurs, E-Learning oder Blended Learning, bei uns finden Sie die passende Lösung für Ihre Aus- und Weiterbildung!

Fast Lane

Testen Sie jetzt unsere Schulungen kostenlos und unverbindlich. Exklusiv für JAVAPRO Leser:

2 Schulungen kostenlos buchen !

Alle hier aufgeführten Schulungen & alle Termine in 2021 wählbar. Kontingent max. 150 Buchungen. First come, first serve!

JAVAPRO Buchungs-Code:

JAVAPRO2021

Hinweis: Buchungen mit JAVAPRO-Code garantiert 0,00 Euro. Buchungs-Code pro Person 2 Mal erlaubt. Buchungen über das erlaubte Kontingent von 2 Schulungen hinaus sowie Mehrfachbuchungen werden von uns automatisch kostenlos storniert. Sie buchen kostenlos und ohne Risiko. Der Rechtsweg ist ausgeschlossen.

Open Liberty | helidon.io | spring boot | azul

Company Training & Microservice Entwicklung

XDEV
www.xdev-software.de

GraalVM - Online Training Live		
GraalVM: Build Native Images	1 Tag	**890 €**

MicroStream - Online Training Live		
MicroStream - Fundamentals	2 Tage	**1.690 €**
MicroStream - Advanced	2 Tage	**1.890 €**

Helidon - Online Training Live		
Helidon & **MicroProfile - Fundamentals**	2 Tage	**1.690 €**
Helidon MP & MicroProfile - Advanced	2 Tage	**1.890 €**
Helidon SE - Advanced	2 Tage	**1.890 €**

Open Liberty - Online Training Live		
Open Liberty & **MicroProfile - Fundamentals**	2 Tage	**1.690 €**
Open Liberty & **MicroProfile - Advanced**	2 Tage	**1.890 €**

Quarkus - Online Training Live		
Quarkus & **MicroProfile - Fundamentals**	2 Tage	**1.690 €**
Quarkus & **MicroProfile - Advanced**	2 Tage	**1.890 €**

Payara Micro - Online Training Live		
Payara Micro & **MicroProfile - Fundamentals**	2 Tage	**1.690 €**
Payara Micro & **MicroProfile - Advanced**	2 Tage	**1.890 €**

Micronaut - Online Training Live		
Micronaut - Fundamentals	2 Tage	**1.690 €**
Micronaut - Advanced	2 Tage	**1.890 €**

Spring Boot - Online Training Live		
Spring Boot Cloud-Native - Fundamentals	2 Tage	**1.690 €**
Spring Boot Cloud-Native - Advanced	2 Tage	**1.890 €**

Alle Preise verstehen sich zzgl. der aktuellen gesetzlichen MwSt.

Termine & Buchung:

www.microservices.education

Alle Schulungen finden online statt.
Bei Fragen zu den verschiedenen Formaten beraten wir Sie gern:

+49 40 253 346 - 10

Jetzt gleich
2 x Schulungen kostenlos
sichern !

#JCON2021
www.jcon.one

JAVAPRO

OCTOBER
5-8

JCON 2021

JCON-ONLINE 2021 LIVE!

Internationale Java Community Conference

5. - 8. OKTOBER

Alle 4 Tage inkl. Workshop-Day nur 99,- EUR. Für alle Java-User-Group Mitglieder frei !

www.jcon.one

4	5	8	110+	150+	2500+
Days	Streams parallel	Haupt-Themen	Speaker	Sessions Live !	Teilnehmer Online

PARTNER 2021

GOLD PARTNER

ORACLE XDEV MicroStream

helidon.io Open Liberty payara MICRONAUT azul RAPID clipse

ORGANISATIONS-PARTNER

JAVAPRO XDEV