Java 2020 – State of the Art

Dossier für Java-Entwickler:

GraalVM, Spring Boot, Kubernetes,
Domain-driven Design & Machine Learning

jaxcon jax.konferenz videosjaxenter jax

jax.de

Inhalt

Oracles vielsprachige Virtual Machine im Java-Ökosystem

GraalVM startet durch

Die GraalVM ist eine universelle Virtuelle Maschine (VM) für Anwendungen, die in den JVM-basierten Programmiersprachen Java, Scala und Kotlin, den dynamischen Sprachen JavaScript, R, Ruby und Python und den LLVM-basierten Sprachen C/C++ geschrieben wurden [1]. Im Oktober 2019 wurde die auf dem JDK-8-Update 231 basierende GraalVM 19.2.1 veröffentlicht, die Performanceverbesserungen und polyglotte Unterstützung für verschiedene Programmiersprachen bietet. Sie ermöglicht die Sprachinteroperabilität in einer gemeinsamen Laufzeitumgebung und kann eigenständig oder im Kontext von OpenJDK, Node.js sowie der Oracle-Datenbank betrieben werden. Die GraalVM kann wahlweise als Open Source Community Edition (CE) oder als Enterprise Edition (EE) mit OTN-Lizenz verwendet werden.

von Wolfgang Weigend

Die ursprüngliche Entwicklung der GraalVM entstand vor mehreren Jahren in den Oracle Labs und führte zur ersten serienreifen Version GraalVM 19.0 im Mai 2019. Sie fügt sich nahtlos in das Java-Ökosystem ein und bietet mit ihrer Vielsprachigkeit [2] eine Heimat für Programmiersprachen, die mit einem Bytecodecompiler ausgestattet sind. Die GraalVM verwendet das JDK 8 und kann mit dem JDK 11, OpenJDK 11 und höheren Versionen auf Command-Line-Ebene experimentell aktiviert werden:

```
java -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI
                                      -XX:+UseJVMCICompiler –jar my_file.jar
```

JDK mit JVM

Innerhalb des Java Development Kit (JDK) sind die Entwicklungswerkzeuge und der Java-Compiler javac angesiedelt. Die Ausführung von Java-Programmen übernimmt das Java Runtime Environment (JRE), entweder als eigenständige Ablaufumgebung oder als Bestandteil des JDK, inklusive der Java HotSpot Virtual Machine (JVM), wie in **Abbildung 1** dargestellt. Der Class Loader lädt die erzeugten Java-Klassendateien, der Java-Bytecode wird verifiziert und im ersten Schritt ohne Optimierung vom Interpreter ausgeführt. Die Optimierung von Methoden übernimmt der JIT-Compiler, der anhand von Profiling gesammelten Ausführungsinformationen entscheidet, wie eine Methodenoptimierung durchgeführt werden kann, bevor der Maschinencode erzeugt wird. Die HotSpot VM beheimatet zwei JIT-Compiler, den C1-Client-Compiler zur Minimierung der Start-up-Zeit und den C2-Server-Compiler, mit dem Fokus auf Durchsatzsteigerung und für dauerhafte Performanceverbesserungen. Der C2-Compiler soll den ausgeführten Code intensiv analysieren und die dabei erkannten Optimierungen besser in Maschinencode umsetzen können.

JIT-Compiler mit Tiered Compilation und Graal-Compiler

Beim C1-Client-Compiler kann die Tiered Compilation, durch *java –client -XX:+TieredCompilation* aktiviert

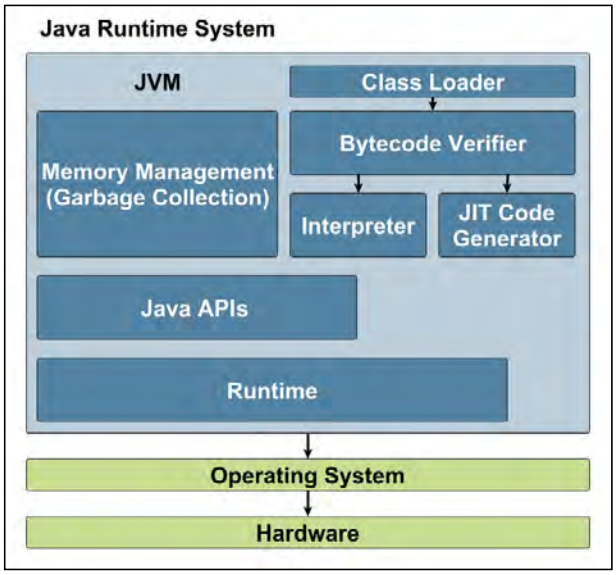

Abb. 1: Java Runtime mit JVM

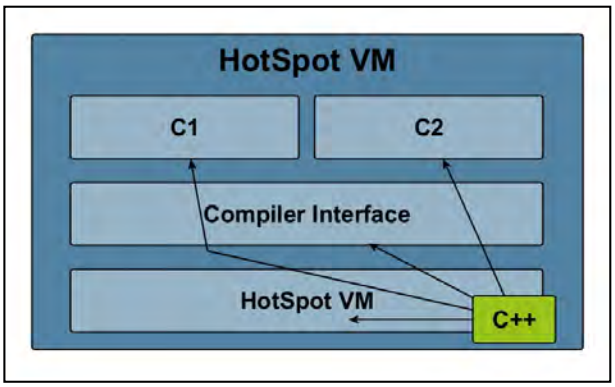

Abb. 2: HotSpot VM implementiert mit C++

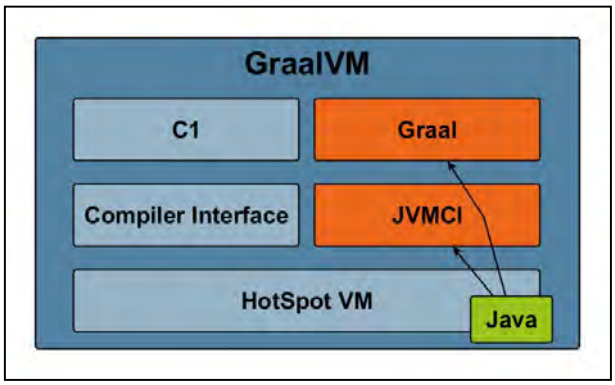

Abb. 3: GraalVM mit Graal-Compiler in Java

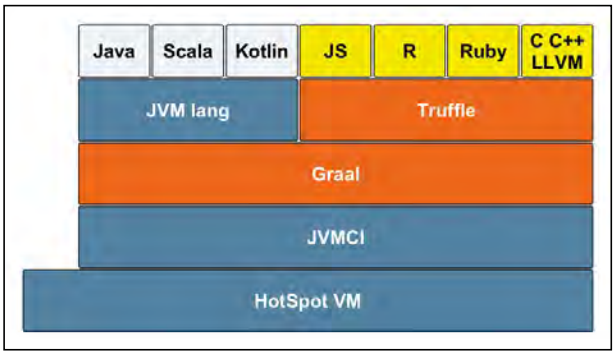

Abb. 4: HotSpot VM mit Graal für JVM-Sprachen

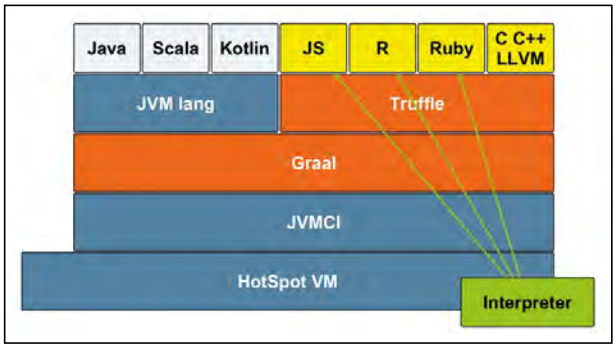

Abb. 5: HotSpot VM mit Graal für dynamische Sprachen

werden, aber der C2-Server-Compiler läuft ohne Tiered Compilation. Die Tiered-Compilation-Ausführungslevel sind wie folgt definiert:

Level 0: interpretierter Code, Level 1: einfacher C1-kompilierter Code ohne Profiling, Level 2: eingeschränkter C1-kompilierter Code mit geringem Profiling, Level 3: vollständiger C1-kompilierter Code mit komplettem Profiling und Level 4: C2-kompilierter Code, der die Profile-Daten der vorherigen Schritte benutzt. Das Java Virtual Machine Compiler Interface (JVMCI) [3] ermöglicht den Austausch vom bisherigen C2-Compiler, der in C++ implementiert wurde (**Abb. 2**), durch den vollständig in Java geschriebenen Graal-Compiler (**Abb. 3**). Mit der neuen Just-in-time-Compilertechnologie werden Java-Programme schneller ausgeführt und der optimierte JIT-Compiler übernimmt die Umwandlung vom Java-Bytecode zum Maschinencode. Vorteilhaft ist dabei, dass die Speicherallokierung direkt auf dem Java Heap erfolgt und dass der Bootstrap-Mechanismus begünstigt wird, da der optimierte Graal-Compiler als Bestandteil der JVM in einer Sprache geschrieben ist, die auch die JVM ausführt.

Polyglotte GraalVM

Neben den JVM-basierten Sprachen können auch andere Sprachen, die mit dem GraalVM Language Implementation Framework implementiert wurden, direkt von Java aus aufgerufen werden (**Abb. 4**). Die Ausführung von den dynamischen Sprachen JavaScript, R, Ruby, Python erfolgt über das Truffle Language Implementation Framework, mit einem API für Spra-

chinterpreter (**Abb. 5**). Truffle ist ein Language Abstract Syntax Tree Interpreter, der es ermöglicht, andere Sprachen mit der GraalVM auszuführen. Die LLVM-basierten Sprachen C/C++ werden über den LLVM-Bitcode-Interpreter Sulong eingebunden. Sulong ist in Java geschrieben und verwendet das Truffle Framework und Graal als dynamischen Compiler. Mit Sulong kann man C/C++, Fortran und andere Programmiersprachen, die in LLVM-Bitcode umgewandelt werden können, mit der GraalVM ausführen. Um ein Programm auszuführen, muss das Programm mit einem LLVM Frontend wie Clang zu LLVM-Bitcode kompiliert werden.

GraalVM Native Image

Mit der GraalVM können native Images erzeugt werden und diese nativen Binaries werden direkt im Betriebssystem ausgeführt, ohne die JVM benutzen zu müssen. Bei der Native-Image-Erzeugung wird eine statische Analyse angewandt, um jeglichen Code zu finden, der über die Java-Main-Methode erreichbar ist, und um dann eine vollständige AOT-Kompilierung (AOT) durchzuführen. Die resultierende native Binärdatei enthält das gesamte Programm zur Ausführung in Maschinencode (**Abb. 6**) und durch den optimierten AOT-Compiler werden schnellere Start-up-Zeiten erreicht. Das native Binary kann mit anderen nativen Programmen verknüpft werden und optional den GraalVM-Compiler für die zusätzliche JIT-Kompilierungsunterstützung enthalten, um andere, auf GraalVM basierende Sprachen, auszuführen. Zur Leistungssteigerung können native Images mit profilgesteuerten Optimierungen erstellt werden, die bereits in einer

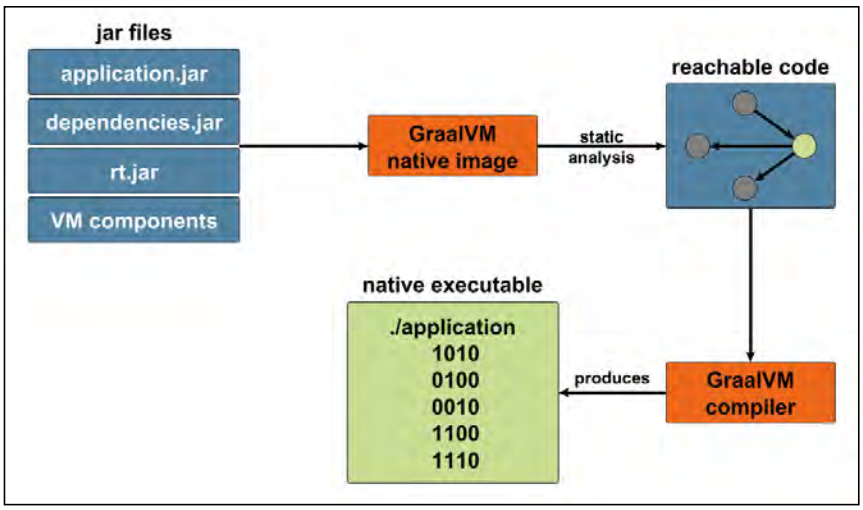

Abb. 6: GraalVM Native Image

früheren Anwendungsausführung gesammelt wurden.

Für die Java-Programmausführung der GraalVM ist die Substrate VM zuständig, die aus Java-Anwendungen über AOT-Kompilierung native Binärdateien erzeugt und die Laufzeitaspekte wie Threads und Speicher verwaltet. Das Substrate VM Framework benötigt die Installation von mx und den auf ein JVMCI-fähiges JDK 8 gesetzten *JAVA_HOME*-Pfad, um die AOT-Kompilierung von Java-Applikationen zu beginnen. Das in Python geschriebene Command-Line-Tool mx dient zur Verwaltung der Entwicklung von Java-Code. Es enthält einen Mechanismus zum Spezifizieren von Abhängigkeiten sowie zum Erzeugen, Testen, Ausführen und Ändern des Codes und der erstellten Artefakte. Das Tool mx enthält Entwicklungsunterstützung für Code, der auf mehrere Quell-Repositories verteilt ist.

Die Native-Image-Kompilierung ist abhängig von der lokal eingesetzten Toolchain, d. h. glibc-devel, zlib-devel (Headerdateien für die C-Bibliothek und zlib) und gcc müssen auf dem System verfügbar sein. Das Erstellen von Images unter Windows ist noch experimentell. Unter Windows muss Python 2.7, Git for Windows und das Windows SDK for Windows 7 installiert sein. Alle Builds müssen über den Windows SDK 7.1 Command Prompt ausgeführt werden. Nach dem Repository Cloning kann das „Hello World"-Beispiel in Listing 1 ausgeführt werden.

Listing 1: Substrate VM Quick-Start

```
cd substratevm
mx build    // Compile all Java and native code

echo "public class HelloWorld { public static void main(String[] args) {
                System.out.println(\"Hello World\"); } }" > HelloWorld.java
$JAVA_HOME/bin/javac HelloWorld.java
mx native-image HelloWorld
./helloworld
```

Die GraalVM-Source-Repository-Komponenten auf GitHub:

- GraalVM Standard Development Kit (SDK) (Kasten „GraalVM SDK") ist eine API-Sammlung der GraalVM.
- GraalVM-Compiler: geschrieben in Java, getrimmt auf hohe Leistung und Erweiterbarkeit, der sowohl die dynamische als auch die statische Kompilierung unterstützt. Der Compiler kann in die Java HotSpot VM integriert sein oder eigenständig ausgeführt werden.
- Truffle Language Implementation Framework: ebenfalls in Java geschrieben, bildet die Basis zur Unterstützung von anderen Sprachen. Über ein Truffle API können Programmiersprachen interpreter gebaut werden, die über die GraalVM ausführbar sind. Truffle ist eine Open-Source-Bibliothek zum Erstellen von Programmiersprachenimplementierungen als Interpreter für selbstmodifizierende Abstract Syntax Trees (AST).

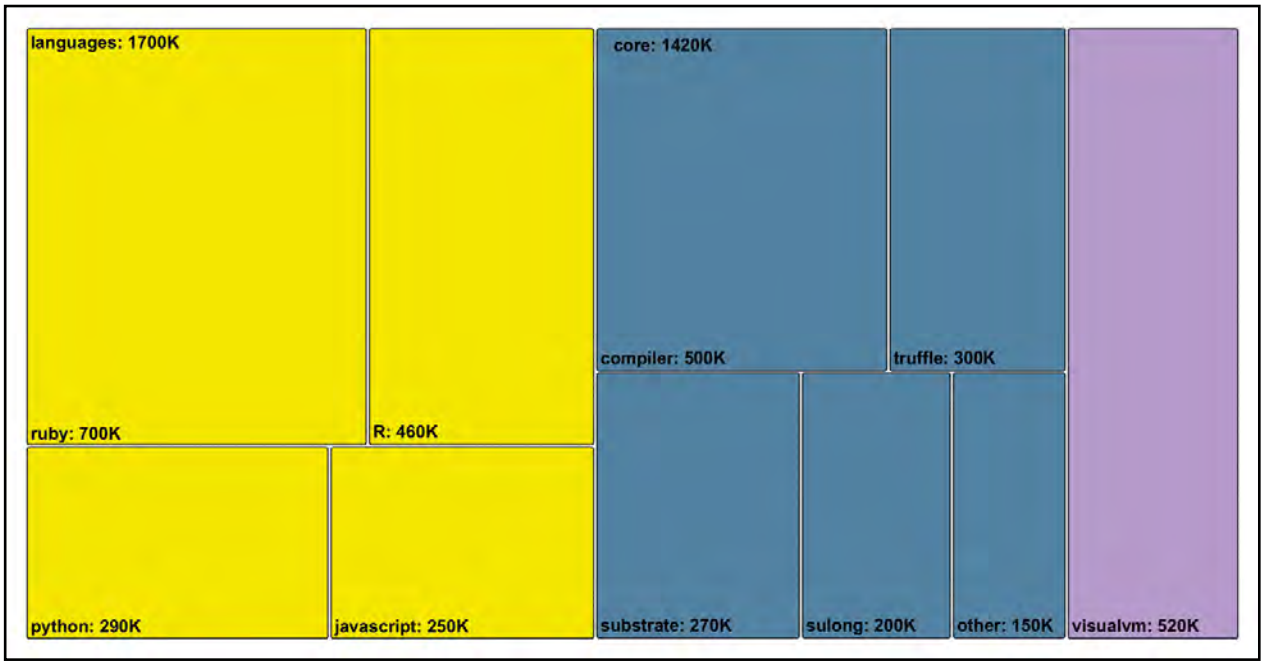

Abb. 7: 3,6 Millionen Open-Source-Codezeilen aktiv gewartet für GraalVM

Die wachsende Anzahl von gemeinsam genutztem Implementierungscode senkt den Sprachimplementierungsaufwand signifikant, und von Vorteil ist ein universelles Instrumentierungs-Framework, das mehrsprachiges Debugging und andere externe Entwicklungswerkzeuge unterstützt.

- Graal-Tool-Set zur GraalVM-Sprachimplementierung mit dem Instrumentation Framework, basierend auf Truffle, ist sprachunabhängig. Die Tools können mit jeder Sprache verwendet werden, die mit Truffle implementiert wurde und die Tests mit dem Technology Compatibility Kit (TCK) erfolgreich durchlaufen hat.
- Substrate VM Framework zur AOT-Kompilierung von Java-Anwendungen in ausführbare Images oder gemeinsam genutzte Objekte, erstellt unter den Bedingungen einer festgelegten und abgeschlossenen Umgebung.
- Sulong Engine zum Ausführen von LLVM-Bitcode auf GraalVM. Sulong ist in Java geschrieben und verwendet das Truffle-Implementierungs-Framework und Graal als dynamischen Compiler.

GraalVM SDK

- org.graalvm.polyglot-Module beinhalten APIs zum Einbinden von Graal-Sprachen in Java-Host-Applikationen.
- org.graalvm.collections-Module beinhalten Memory-effiziente Common-Collection-Datenstrukturen, die übergreifend von Graal-Projekten benutzt werden.
- org.graalvm.options-Module beinhalten wiederverwendbare Klassen als Option.

- TRegex ist eine Implementierung eines Subsets regulärer ECMAScript-Ausdrücke, die mit dem GraalVM-Compiler und dem Truffle API reguläre Ausdrücke ausführen können.
- VM Suite beinhaltet Komponenten zur Erstellung eines modularen GraalVM Image. Die VM ermöglicht die Erzeugung von benutzerdefinierten GraalVM-Distributionen sowie installierbare Komponenten. Die definierte GraalVM-Basisdistribution enthält ein JVMCI-fähiges JDK, das GraalVM SDK, Truffle und den GraalVM-Komponenteninstaller. Weitere Komponenten können über den dynamischen Suite-Import hinzugefügt werden.

GraalVM Open Source

Die GraalVM Community Edition basiert auf 3,6 Millionen Open-Source-Codezeilen [4], die vom GraalVM-Team und anderen Entwicklern stammen, sowie auf zusätzlichen Millionen Source-Code-Zeilen aus Projekten, auf die GraalVM angewiesen ist, wie Java, Node.js und andere (**Abb. 7**).

Versionsauswahl und Installation

Die GraalVM gibt es in zwei Ausführungen [6]: Zum einen gibt es die frei einsetzbare Open Source GraalVM Community Edition, die aus den GraalVM GitHub Sourcen erstellt und als vorgefertigte Binärdatei für Linux-, macOSX- und für die Windows-Plattformen auf x86-64-Bit-Systemen angeboten wird. Die Windows-Unterstützung ist experimentell. Die Lizenzierung [8] der Open-Source-Komponenten [5] ist im Kasten „Lizenzierung der GraalVM-Komponenten“ dargestellt.

Zum anderen gibt es die optimierte GraalVM Enterprise Edition, die für den produktiven Einsatz kostenpflichtig ist und nur über die Oracle-Technolo-

gy-Network-Lizenzvereinbarung (OTN-Lizenz) [9] angeboten wird. Im Gegensatz zur Community Edition bietet die GraalVM Enterprise Edition eine verbesserte Leistungsfähigkeit, Sicherheit und Skalierbarkeit, die für den Produktivbetrieb von Anwendungen relevant sind. Die GraalVM Enterprise Edition ist für Entwicklung, Test, Prototyping und Demozwecke kostenlos und kann als Binärdatei für Linux-, MacOS-X- und für die Windows-Plattformen auf x86/64-Bit-Systemen vom Oracle Technology Network heruntergeladen werden. Die Windows-Unterstützung ist experimentell (Listing 2).

Die GraalVM-Installation ist in wenigen Minuten vollzogen und wird als **.tar.gz*- oder **.zip*-Archivdatei heruntergeladen und in ein Verzeichnis entpackt. Dort muss die Variable *JAVA_HOME* auf den Pfad mit dem verwendeten JDK gesetzt sein.

Die Grundinstallation der GraalVM EE beinhaltet die JVM, den GraalVM-Compiler, den LLVM-Bitcode-Interpreter und die JavaScript-Laufzeitumgebung mit Node.js-Unterstützung.

Performance

Messungen der GraalVM mit einer Cloudinfrastruktur [7] haben ergeben, dass die Compileroptimierungen der GraalVM Enterprise Edition Version 19.1 die Objektallokierungen reduzieren und dass sich die Ausführungsgeschwindigkeit vom Cloud-Monitoring-Service gegenüber dem JDK 8 verbessert.

Als Resultat sind weniger Garbage-Collection-Pausen bei geringerem Rechenleistungsverbrauch für die äquivalente Workload-Ausführung nachweisbar. Der Cloud-Monitoring-Service konsumierte insgesamt fünf Prozent weniger Prozessorleistung, erzielte einen höheren Durchsatz und verbrauchte weniger Zeit für die Garbage Collection und andere Systemaktivitäten. Beim Monitoring-Service wurden im Vergleich zu Java-8-Update 212 die Garbage-Collection-Zeit um 25 Prozent minimiert und die Applikationspausenzeiten um 17 Prozent verringert. Der Durchsatz erhöhte sich um zehn Prozent mit den GraalVM-Optimierungen bei der Anzahl der Transaktionen pro Sekunde im Vergleich zu Java-8-Update 212.

Listing 2: GraalVM EE 19.2.1

```
C:\graalVM\bin>java -version
java version "1.8.0_231"
Java(TM) SE Runtime Environment (build 1.8.0_231-b11)
Java HotSpot(TM) 64-Bit GraalVM EE 19.2.1 (build 25.231-b11-jvmci-
19.2-b03, mixed mode)
```

Lizenzierung der GraalVM-Komponenten

- Truffle Framework mit seinem abhängigen GraalVM SDK ist lizenziert unter der Universal Permissive License
- Tools Project ist lizenziert unter der GPL 2 mit Classpath Exception
- TRegex Project ist lizenziert unter der GPL 2 mit Classpath Exception
- GraalVM-Compiler ist lizenziert unter der GPL 2 mit Classpath Exception
- Substrate VM ist lizenziert unter der GPL 2 mit Classpath Exception
- Sulong ist lizenziert unter 3-clause BSD
- VM ist lizenziert unter der GPL 2 mit Classpath Exception

Fazit

Bei aller Euphorie angesichts der innovativen und polyglotten GraalVM mit ihren ungeahnten Einsatzmöglichkeiten muss sich die Praxistauglichkeit dieser jungen Technologie erst bewähren. Die GraalVM-AOT-Compilerfähigkeit zur Erstellung von eigenständig ausführbaren Native Images und ihre – gegenüber der JVM – verbesserte Start-up-Zeit bei geringerem Memory-Verbrauch lassen es zu, dass Java in den Bereich der Systementwicklung vordringen kann. Zudem gelangen die Vorteile der GraalVM Native Images direkt zu den Java Microservices Frameworks.

Mit der neuen GraalVM und der darin enthaltenen nativen Image-Funktionalität können auch kleinere und leistungsfähigere JavaFX-Anwendungen mit der aktuellen, ein Cross-Plattform-Deployment ermöglichenden JDK-Version und der korrespondierenden JavaFX-Version erstellt werden.

Insgesamt hat die GraalVM das Potenzial, die Java-Landschaft grundlegend zu verändern. Das könnte sich im Segment Mobile- und Embedded-Technologie bemerkbar machen. Auch die Entwicklungsbeiträge zum Ökosystem mit GraalVM und gemeinsamen Open-Source-Projekten haben zugenommen, wie Eclipse Vert.x Toolkit, Fn Project, Gluon Client Plugin, Helidon, Micronaut, Picocli Java Command Line Parser und Quarkus zeigen.

Wolfgang Weigend arbeitet als Sen. leitender Systemberater bei der Oracle Deutschland B.V. & Co. KG. Er beschäftigt sich mit Java-Technologie und Architektur für unternehmensweite Anwendungsentwicklung.

Links & Literatur

[1] https://www.graalvm.org

[2] https://www.graalvm.org/docs/reference-manual/polyglot/

[3] https://openjdk.java.net/jeps/243

[4] https://www.graalvm.org/community/opensource/

[5] https://github.com/oracle/graal/blob/master/README.md

[6] https://www.graalvm.org/downloads/

[7] https://blogs.oracle.com/cloud-infrastructure/graalvm-powers-oracle-cloud-infrastructure

[8] https://github.com/oracle/graal#license

[9] https://www.oracle.com/downloads/licenses/graalvm-otn-license.html

Spring Boot 2.2: die neuen Features in der Übersicht

Zwei und zwei ergibt eine neue Generation

Spring Boot 2.2 ist da! Zu den Highlights der neuen Version, die im Oktober dieses Jahres erschien, gehören die Unterstützung für Java 13 und die Aktualisierung zahlreicher Bibliotheken. In diesem Artikel beleuchten wir diese und alle weiteren Neuerungen eingehend.

von Michael Simons

Nachdem ich während der Veröffentlichung meines Spring-Boot-Buchs [1] den mehr als umfangreichen Änderungen von Spring Boot 1.5 nach 2.0 quasi hinterher geschrieben habe, hielten sich grundsätzliche Änderungen in Spring Boot 2.1 [2] in überschaubaren Grenzen. In meinen Augen waren das natürlich der Support für das Spring Framework 5.1 und Änderungen in der Default-Bean-Konfiguration: standardmäßig ist es seit Spring Boot 2.1 nicht mehr erlaubt, einmal definierte Beans in weiteren *@Configuration*-Klassen zu überschreiben.

Im Enterprise-Alltag fiel insbesondere auf, dass die *JpaProperties* keine Hibernate-spezifischen Einstellungen enthalten. Diese wanderten in *HibernateProperties*, inklusive neuer Präfixe für entsprechende Konfigurationsdateien. Auch hier hilft das Tooling (*org.springframework.boot:spring-boot-properties-migrator*).

Welche Java-Version für Spring Boot 2.2?

Spring Boot 2.1 brachte volle Unterstützung für Java 11 mit, blieb aber mit Java 8 kompatibel. Die Releasekadenz von Java ist nunmehr halbjährlich, wie sieht es mit 2.2 aus? Das Spring-Team bei Pivotal verfolgt die Java-Releases und testete den Milestone M5 bereits mit Java 13, im finalen Release ist die Unterstützung von JDK 13 und damit des aktuellen LTS-Java-Release enthalten. Es wunderte mich zudem nicht, dass das Spring Framework mit 5.2 eine andere Version als Java 8 als minimale Anforderung hat.

Jakarta EE

Ist das nicht ein Artikel über Spring Boot? Natürlich, aber das Spring Framework und damit Spring Boot setzt natürlich auf offene, funktionierende Standards, unter anderem Java EE. Im Mai 2019 wurde bekannt, dass die Java Enterprise Edition (Java EE) nach dem Umzug zur Eclipse Foundation und der Umbenennung zu Jakarta EE noch weitere Aufgaben vor der Brust hat: Eine Einigung hinsichtlich Marken- und Namensrechten konnte nicht gefunden werden, der Name Java ist nicht nur als Java EE schützenswert, sondern auch als Teil der API-Spezifikation. Das heißt, existierende javax-Packages müssen unter dem Jakarta-EE-Mantel eingefroren und dürfen nicht weiterentwickelt werden. Weiterentwicklung darf nur unter anderem Namen erfolgen. Finden Änderungen und Erweiterungen statt, muss das Package umbenannt und das API angepasst werden. Bestehende Java-EE-Spezifikationen im *javax*-Namensraum sind mit zusätzlichen Auflagen verbunden.

Für alles nach Jakarta EE 8 besteht als realistische Möglichkeit nur, schnellstmöglich alle Spezifikationen und APIs von *javax* zu *jakarta* zu portieren. Genau das passiert, und Spring Boot bzw. das Spring Framework ziehen an dieser Stelle mit: Soweit möglich, werden Jakarta-EE-Spezifikationen eingesetzt, die bereits portiert sind. Stand heute sind das Jakarta Mail und Jakarta EL.

Allgemeine Betrachtungen

Die ursprünglichen Ziele Spring Boots (einfache Konfigurierbarkeit von Spring-Framework-Anwendungen und Erstellung ausführbarer Anwendungen) werden meiner Meinung nach immer mehr von Dependency-Management überschattet. Spring Boot ist enorm erfolgreich, unter anderem weil es viele Abhängigkeiten im Kontext des Spring-Ökosystems konfiguriert. Umgekehrt rückt der Erfolg von Spring Boot in den Fokus von

Reaktive Transaktionen sind ein bemerkenswertes Feature, aber weder in der Erstellung noch in der Benutzung einfach, auch wenn der deklarative Ansatz das Gegenteil suggeriert.

Herstellern, die möchten, dass ihr Produkt möglichst gut auf der Plattform funktioniert.

Die Liste gemanagter Dependencies wächst und wächst und mit ihr die Menge der Libraries, die bei einem Versionssprung ebenfalls aktualisiert werden müssen. Hervorgehoben werden in der Regel das Spring Framework selbst, Spring Security, Hibernate und JUnit. Spring Boot 2.1 machte den Sprung auf Hibernate 5.3, Spring Boot 2.2 setzt auf Hibernate 5.4. In den Release Notes fallen insbesondere AssertJ 3.12 und Hamcrest 2.1 mit Breaking Changes auf, weitere erwähnenswerte Upgrades sind:

- Artemis 2.7
- Elasticsearch 6.7.2
- Hazelcast 3.12
- Jedis 3.1
- Jersey 2.29
- Kafka 2.2
- Kotlin 1.3.31
- Lombok 1.18.8
- Netty 4.1.36
- Solr 8.0
- Tomcat 9.0.19

Die Liste aktualisierter Spring-eigener Module ist ähnlich umfangreich:

- Spring AMQP
- Spring Batch
- Spring Data Moore
- Spring Framework
- Spring HATEOAS
- Spring Integration
- Spring Kafka
- Spring Security
- Spring Session Corn
- Reactor Dysprosium

In den meisten Fällen hängt Spring Boot von diesen Projekten hinsichtlich Autokonfiguration und Ähnlichem ab. Teilweise bestehen aber auch umgekehrte Abhängigkeiten, meistens über das Spring Framework selbst. Natürlich bringen die einzelnen Spring-Projekte wieder Abhängigkeiten mit sich, die Liste der zu lesenden Changelogs wird gefühlt immer größer.

Ich vertrete nach wie vor die Meinung, dass Spring Boot die vernünftigste Lösung ist, im Jahr 2019 ein neues Spring-Projekt aufzusetzen, aber die Form des Projekts gilt es zu überdenken: Multi-Module-Projekte haben sich für mich als ausgesprochen schwierig erwiesen und zwar hinsichtlich der Wartbarkeit. Soll in einem Module, beispielsweise dem Datenbanklayer – und das gilt unabhängig davon, ob dieser JPA heißt oder es sich um das von mir gepflegte Spring Data Neo4j bzw. das Neo4j-OGM-Modul handelt –, nur Spring Data aktualisiert werden, bedeutet das, dass ich damit auch das Spring Framework aktualisieren und damit alle Abhängigkeiten auf vom Framework unterstützte Versionen heben muss. Ein oftmals erheblicher Aufwand, der zwar geleistet werden kann, aber mit der Größe der Anwendungen und der Anzahl der Module wächst. Hier einen vernünftigen Architekturschnitt zu finden, ist wichtiger denn je.

Performance

Neue Frameworks wie Micronaut [3] und Quarkus [4], die – frei von über 15 Jahren Entwicklung – verschiedene Aspekte eines Anwendungs-Framework neu denken konnten und tatsächlich auf Microservices fokussiert sind, haben das Java-Umfeld in den letzten Monaten aufgewirbelt. Zusammen mit der GraalVM [5], die es – neben vielen anderen coolen Dingen – erlaubt, native Binaries aus Java-Programmen zu erzeugen, rücken so Dinge wie Start-up-Zeit und auch Speicherverbrauch noch weiter in den Fokus.

Spring Boot und das Spring Framework adressieren in den Versionen 2.2 und 5.2 viele dieser Punkte. Die eingebaute automatische Konfiguration (siehe dazu einen meiner Vorträge [6]) verwendet nun keine Proxies

mehr und Injection Points werden nur dann tatsächlich angewandt, wenn eine entsprechende Bean erzeugt wird.

Gerade Quarkus hat durch Anpassungen in Hibernate extrem viel Performance gewinnen können. Gleiches gilt für Spring-Anwendungen: Hibernates Entity Scanning wird in Spring Boot 2.2 gar nicht mehr genutzt, sondern nur noch die von Spring bereitgestellte Persistence Unit. Das ist im Übrigen ein Ansatz, den wir mit unserem Next-Generation-Spring-Data-Modul für Neo4j [7] ebenfalls verfolgen. Wer mehr darüber lesen möchte, sollte in den Blogpost von Gerrit Meier schauen: Welcome SDN/RX [8].

Eine Opt-in-Verbesserung ist die Möglichkeit, alle Beans *lazy* zu initialisieren. Durch Setzen von *spring.main.lazy-initialization* auf *true* wird das Feature eingeschaltet. Der Start kann schneller sein, der erste HTTP Request wird hingegen langsamer sein. Weiterhin fallen einige mögliche Fehler erst nach der vollständigen Initialisierung und nicht bereits beim Start auf. Funktionen wie Tomcats *MBean Registry*, die zwar nützlich sind, aber oft nicht genutzt werden, sind per Default abgeschaltet. Das reduziert Speicherverbrauch.

Reaktive Transaktionen

Die Spring Data Stores MongoDB und Redis unterstützen nun seit einiger Zeit reaktive Programmierung. Mit Reactive Relational Database Access (R2DBC) [9] sowie serverseitigen reaktiven Transaktionen innerhalb der Graphdatenbank Neo4j 4.0 [10] und auch in MongoDB wurde der Ruf nach applikationsseitigem Support für reaktive Transaktionen unüberhörbar laut. Mark Paluch schrieb bereits im Mai über das zu diesem Zeitpunkt im Spring Framework veröffentliche Feature: Reactive Transactions with Spring [11].

Es geht bei diesem Feature eben nicht um modulspezifische Lösungen wie *.inTransaction* oder Ähnliches, sondern um die Unterstützung des bekannten deklarativen Ansatzes (*@Transactional*) sowie ein Pendant zum *TransactionTemplate*. Spring 5.2 bietet ab 5.2 M2 ein SPI für *ReactiveTransactionManager* an. Auf Basis dieses API kann *@Transactional* auf alle Methoden angewandt werden, die reaktive Datentypen (*Flux<>* und *Mono<>*) zurückgeben. Der *TransactionalOperator* ist das Pendant zum *TransactionTemplate*.

Reaktive Transaktionen speichern ihre Ressourcen – Status der Transaktion und anderes – nicht in einem *ThreadLocal*, sondern im Kontext von Project Reaktor [12]. Das ist insofern erwähnenswert, als es bedeutet, dass reaktive Transaktionen mit Spring nur mit reaktiven Datentypen aus Project Reactor, nicht aber mit RX Java 2 funktionieren, obwohl RX Java 2 weiterhin von Spring unterstützt wird. Deklarative, reaktive Transaktionen sowie der *TransactionalOperator* werden von folgenden Spring-Data-Modulen unterstützt:

- R2DBC mit Spring Data R2DBC ab 1.0 M2
- MongoDB mit Spring Data MongoDB ab 2.2 M4
- Neo4j 4.0 mit Spring Data Neo4j RX ab 1.0.0-beta01

SDN/RX wurde Ende August von Gerrit Meier öffentlich vorgestellt: Welcome SDN/RX [13]. Dieser englische Post von Mark Paluch ist in diesem Zusammenhang wärmstens zu empfehlen: Reactive Relational Database Transactions [14].

Reaktive Transaktionen sind ein bemerkenswertes Feature, aber weder einfach in der Erstellung noch einfach in der Benutzung, auch wenn der deklarative Ansatz das Gegenteil suggeriert. Ich bin gespannt, welchen Anklang es – gerade im relationalen Enterprise-Umfeld – findet und mit welchen Erwartungshaltungen es eingesetzt werden wird.

RSocket Server

RSocket [15] ist ein binäres Protokoll auf Anwendungsebene, das die Semantik von reaktiven Datenströmen [16] über binäre Transportprotokolle wie TCP und WebSocket implementiert. RSocket ermöglicht verschiedene Formen reaktiver Programmierung:

- Request/Response (Stream of one)
- Request/Stream (Stream of many)
- Fire and Forget (no Response)
- Channel (bi-directional Streams)

RSocket wird direkt vom Messaging-Modul des Spring Framework unterstützt, sowohl auf Server- als auch auf Clientseite. Durch spezielle *@Controller*, die *@MessageMapping* einsetzen, kann es analog zu HTTP-Controllern genutzt werden, ohne dass ein neues Paradigma für die Programmierung erlernt werden müsste. Spring

Boot 2.2 wird dies mit automatischer Konfiguration unterstützen.

Wichtige Aspekte hier sind, in welcher Art und Weise Payloads serialisiert und wieder deserialisiert werden. Es stehen CBOR und JSON zur Verfügung. Der Server benötigt entweder einen aktiven WebFlux-Server oder eine explizite Konfiguration eines eingebetteten RSocket-Servers. Ebenfalls neu ist ein *RSocketRequester*, der innerhalb von Services genutzt werden kann, um mit RSocket-Servern zu kommunizieren.

Weitere erwähnenswerte Features

Freemarker, eine Templatesprache, ermöglicht die Nutzung von *.ftl* und *.ftlh*. In *.ftl*-Dateien wird HTML in interpolierten Blöcken (*${}*) per Default nicht escaped. Das ist unsicher, und *.ftlh*-Dateien drehen das Verhalten entsprechend um. Spring Boot 2.2 wird *.fthlh* als Defaulterweiterung für Freemarker-Dateien verwenden. Konstruktor-Injection ist kein neues Feature, der Support für *@DefaultValue*, *@DateTimeFormat* und andere hingegen schon.

Nach langer Zeit ist Spring HATEOAS endlich aus der Alpha heraus und in Version 1 [17] veröffentlicht worden. Spring Boot 2.2 wird Spring HATEOAS vollumfänglich unterstützen. Neo4j 3.4+ hat hingegen nicht nur vollständigen Support für alle wichtigen Java-8-Datums- und Zeittypen, sondern auch für räumliche Datentypen. Spring Boot 2.2 konfiguriert OGM nun entsprechend, falls die notwendigen Abhängigkeiten auf dem Klassenpfad sind. Ebenso wurde die eingebettete Variante von Neo4j in das Dependency-Management mit aufgenommen. Das vereinfacht das Testen von Neo4j-Anwendungen.

Zu guter Letzt wurden die Konfigurationseigenschaften *logging.file* und *logging.path* als „deprecated“ markiert und sinnvollerweise durch *logging.file.name* beziehungsweise *logging.file.path* ersetzt.

Übersicht der verfügbaren Changelogs

Grundlage für den Artikel sind – abgesehen von eigener Arbeit – die öffentlich verfügbaren Changelogs. Ich kann nur empfehlen, die entsprechenden Wiki-Pages auf GitHub zu verfolgen, um für ein Upgrade gewappnet zu sein. Ebenso möchte ich die Empfehlung des Spring-Teams wiederholen: Ein Upgrade sollte in Phasen erfolgen. Falls die Anwendung noch auf Spring Boot 2.0 basiert, sollte zuerst auf 2.1, dann auf 2.2 aktualisiert werden.

Michael Simons ist Vater, Ehemann und Radfahrer. Er arbeitet als Software Engineer bei Neo4j und beschäftigt sich dort mit dem Spring-Data-Modul für die gleichnamige Graphdatenbank Neo4j. Michael ist Java Champion, Mitgründer und aktueller Leiter der Eurego JUG. Nachdem er im April 2018 das erste deutschsprachige Spring-Boot-Buch veröffentlicht hat, schreibt er aktuell wieder vermehrt in seinem Blog über Datenbanken, Java, Spring und Softwarearchitektur.

Links & Literatur

[1] http://springbootbuch.de

[2] https://jaxenter.de/spring-boot-2-1-76822

[3] https://micronaut.io

[4] https://quarkus.io

[5] https://www.graalvm.org

[6] https://speakerdeck.com/michaelsimons/die-magie-hinter-spring-boot-1

[7] https://neo4j.com

[8] https://medium.com/neo4j/welcome-sdn-%EF%B8%8Frx-22c8fe6cd955

[9] https://r2dbc.io

[10] https://neo4j.com/blog/neo4j-enterprise-edition-4-0-milestone-release-2/

[11] https://spring.io/blog/2019/05/16/reactive-transactions-with-spring

[12] https://projectreactor.io

[13] https://medium.com/neo4j/welcome-sdn-%EF%B8%8Frx-22c8fe6cd955

[14] https://paluch.biz/blog/179-reactive-relational-database-transactions.html

[15] https://rsocket.io

[16] http://www.reactive-streams.org

[17] https://spring.io/blog/2019/03/05/spring-hateoas-1-0-m1-released

Wenn Java EE und MicroProfile auf Kubernetes und Istio treffen …

Ein Cloud-Native-Starter-Projekt

Nicht jede neue Anwendung muss als Microservices-Architektur „cloud-native" entwickelt werden, aber das Thema Cloud-native hat mittlerweile sicherlich den Hypestatus überwunden und sich als feste Größe in der Anwendungsentwicklung etabliert. Allerdings bedeutet der Umstieg auf Cloud-native, dass man viel Neues lernen und alte Gewohnheiten aufgeben muss. Das kann schon viel auf einmal sein, wie wir selbst erfahren haben.

von Harald Uebele

Anfang des Jahres haben wir in unserem Team entschieden, dass wir uns mit dem Thema Java EE und Container beschäftigen wollen. Java EE, weil es immer noch die meistverwendete Sprache im Enterprise-Umfeld ist, und Container, da sie praktisch unumgänglich sind, wenn es um Cloud-native-Anwendungen geht. Cloudtechnologie ist ja auch ein wichtiger Baustein für die Themen Digitalisierung und Anwendungsmodernisierung im genannten Enterprise-Umfeld. Unser Ziel war es, herauszufinden, ob man für Anforderungen an Ausfallsicherheit, Testing, Monitoring usw. besser die Funktionen einer Plattform wie Istio [1] verwendet oder die Funktionen eines Frameworks wie MicroProfile [2]. Dabei stellten wir fest, dass es im Internet dafür zwar viele Beispiele gibt, diese aber entweder nur einzelne Aspekte betrachten, auf proprietären Produkten basieren oder so komplex waren, dass sie auf einem typischen Notebook nicht installierbar wären. Deshalb entstand unser Cloud-Native-Starter-Projekt, das komplett auf Open Source basiert, selbst Open Source und mit Hilfe von Skripten für jeden einfach installierbar ist, und von uns auf GitHub veröffentlicht wurde [3]. Wir wollten dabei alle (oder zumindest die meisten) Aspekte abdecken, die einen Anwendungsentwickler im Enterprise-Umfeld zum Thema Cloud-native interessieren könnten:

- Java und REST APIs
- Traffic Routing (Dark Launches, Canary Deployments)
- Resiliency (Verhalten der Anwendung im Fehlerfall)
- Authentication, Authorization
- Verteiltes Logging und Monitoring
- Health Checks
- Konfiguration
- Persistenz mit Java Persistence API

Zusätzlich wollten wir es auf möglichst vielen Umgebungen realisieren – im Moment haben wir Skripte für:

- Minikube
- IBM Cloud Kubernetes Service
- Minishift
- OpenShift auf der IBM Cloud

Im GitHub-Projekt sind momentan weit über 20 Blogartikel verlinkt, die wir zu einzelnen Themenbereichen geschrieben haben. Einige davon zitiere ich in diesem Artikel, und es ist zusätzlich ein kompletter Hands-on-Workshop zum Thema Cloud-native Applications enthalten [4].

Grundlagen

Nachfolgend die Definition für Cloud-native auf der Seite der Cloud Native Computing Foundation [5] (in meiner Übersetzung):

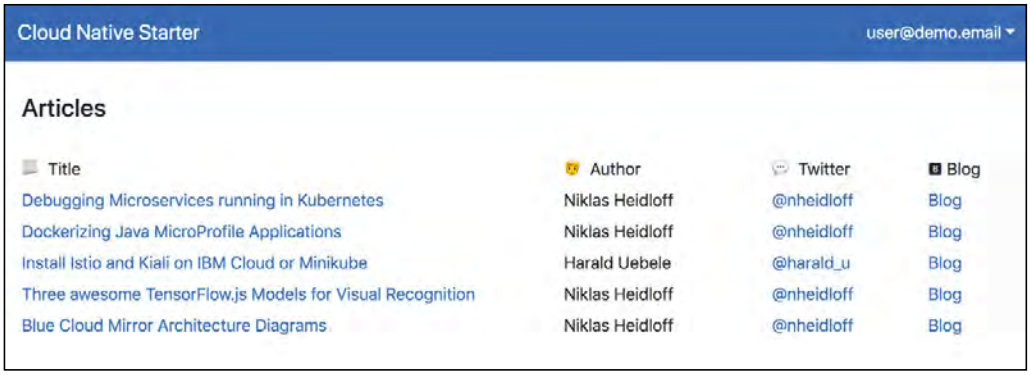

Abb. 1: Cloud Native Starter: Web-App als Frontend

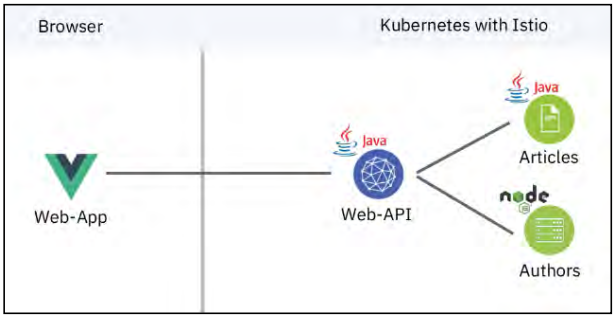

Abb. 2: Cloud-Native-Starter-Architektur

„Cloud-native Computing verwendet einen Open-Source-Softwarestack, um Anwendungen als Sammlung von Microservices zu deployen, jeden einzelnen Microservice in einen eigenen Container zu packen und diese Container dynamisch zu orchestrieren, um die Ressourcenauslastung zu optimieren. Die nativen Cloudtechnologien ermöglichen es Softwareentwicklern, tolle Produkte schneller zu erstellen."

In anderen Worten: Statt als einen einzigen Monolith entwickeln wir unsere Cloud-native-Anwendung aus einzelnen Microservices. Diese werden als Container (Docker) Images in Containern deployt. Um das Orchestrieren der Container kümmert sich Kubernetes, da es mittlerweile der De-facto-Standard für Containerorchestrierung und außerdem Open Source ist.

Basis für die Microservices sind besagte Docker Images, die mit Hilfe von Dockerfiles gebaut werden. Alle Java-basierten Microservices von Cloud Native Starter verwenden diesen Open-Source-Stack:

- OpenJ9, die Open Source JVM von IBM [6]
- OpenJDK, kann OpenJ9 als JVM verwenden [7]
- MicroProfile als Java EE Framework [2]
- Open Liberty als Server Runtime und MicroProfile-Implementierung [8]

Im Dockerfile sieht das dann einfach so aus:

```
FROM open-liberty:microProfile2-java11
```

Hier stellt sich vielleicht die Frage: Java und Microservices? Ernsthaft? Ja, der Anspruch von MicroProfile ist es, Enterprise Java für eine Microservices-Architektur zu optimieren, und das funktioniert auch.

Die Cloud-Native-Starter-Anwendung

Unsere Beispielanwendung ist eine einfache Microservices-Architektur. Sie besteht aus einem Frontend, das im Browser aufgerufen wird und in Node.js mit dem Vue.js Framework geschrieben ist. Sie zeigt eine Liste unserer Blogeinträge an, für jeden Eintrag den Titel, den Autor und als Details das Twitter Handle und den Link zum Blog (**Abb. 1**).

Die komplette Anwendung besteht neben diesem Frontend aus drei Microservices (**Abb. 2**):

1. Web-API: Backend-for-Frontend-Pattern, REST API für die Artikelliste
2. Articles: REST API für Titel und Autor
3. Authors: REST API für Twitter Handle und Link zum Blog für einen Autor

Damit die Anwendung richtig funktioniert, müssen diese drei Microservices deployt sein und natürlich auch laufen, dafür ist Kubernetes zuständig. Kubernetes sorgt außerdem durch Namensauflösung (Name Resolution) dafür, dass der Web-API-Service den Arti-

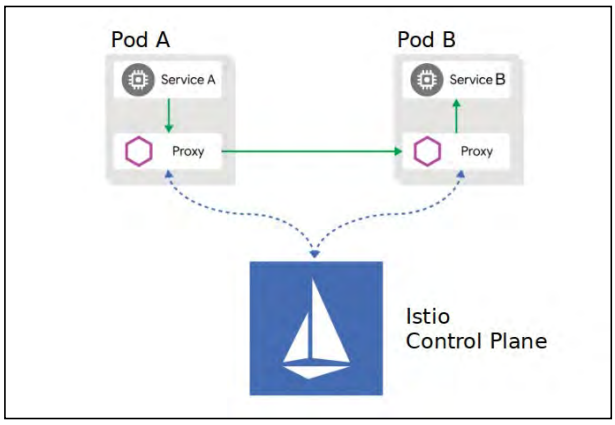

Abb. 3: Istio-Architektur

cles Service und den Authors Service ansprechen kann, ohne Details wie IP-Adresse und dergleichen von ihnen zu kennen. MicroProfile spielt in diesem Zusammenhang mehrere Rollen:

- Das REST API wird über JAX-RS bereitgestellt
- MicroProfile Open API erlaubt die Dokumentation des REST API direkt im Code (Swagger)
- Web-API benutzt den MicroProfile REST Client für den Zugriff auf das Articles API und das Authors API

Plattform

Zu Kubernetes als Werkzeug zur Containerorchestrierung muss man sicherlich nicht mehr viel sagen. Es ist inzwischen bei jedem großen Cloudanbieter verfügbar und wird auch oft für Cloudanwendungen im eigenen Rechenzentrum eingesetzt, z. B. als Red Hat OpenShift.

Ein anderer wichtiger Bestandteil ist Istio [1] als Service Mesh, es ist wie Kubernetes Open Source. Istio bekommt viel Unterstützung, es sind mittlerweile viele und auch große Firmen beteiligt, neben den Projektgründern Google, IBM und Lyft z. B. Pivotal und Red Hat. Es kann nicht nur Kubernetes als Plattform nutzen, sondern auch Nomad und Mesos. Der Begriff Service Mesh beschreibt ein Netzwerk aus Microservices, die zusammen eine Applikation darstellen, und ihre Interaktionen. Es ist eine Art transparente Schicht, die es erlaubt, die Microservices miteinander zu verbinden, diese Verbindungen abzusichern und die Services zu überwachen (**Abb. 3**).

Dafür installiert Istio in den Pod eines Microservice ein Proxy namens Envoy, das manchmal auch als Sidecar, also Beiwagen, bezeichnet wird. Wie ein Beiwagen neben dem Motorrad, so sitzt der Envoy neben dem Microservices-Container. Die Verbindung und damit der Datenfluss zwischen den Services läuft nur über den Envoy, die Services der Istio-ControlPlane kommunizieren mit den Envoys und konfigurieren das Routing, installieren Zertifikate, sammeln Telemetriedaten usw. Der Vorteil des Sidecars liegt darin, dass kein Code und keine zusätzlichen Bibliotheken in die eigentlichen Microservices eingepflegt werden müssen, das macht die ganze Sache sprachunabhängig, d. h., es funktioniert in Java genau wie in jeder anderen Programmiersprache.

Verwende ich besser Plattform oder Framework, Istio oder MicroProfile?

Kubernetes und Istio sind also die Plattform, MicroProfile das Framework für unsere Java-basierten Services. Istio und MicroProfile haben zum Teil überlappende Funktionen, beide bieten z. B. Funktionen für Fehlertoleranz, Metriken und Tracing. Welche Version soll man einsetzen? Es folgen zwei Beispiele:

Beispiel 1: Nehmen wir das Thema Fehlertoleranz und besonders Fallback: Nur ich als Entwickler der Businesslogik weiß, wie ich mit einem Verbindungsfehler umgehen kann, ohne dass meine Anwendung abstürzt oder der Benutzer mit unverständlichen Fehlermeldungen irritiert wird, d. h. für Fallback sollte ich auf jeden Fall die Fallback-Methode von MicroProfile einsetzen [9].

Abbildung 4 zeigt einen Codeausschnitt aus dem Web-API von Cloud Native Starter. An der markierten Stelle wird versucht, fünf Artikel vom Articles Service zu bekommen. Wenn das nicht funktioniert, weil der Service nicht verfügbar ist, wird die Methode aufgerufen, die in der *@Fallback*-Annotation angeben ist. *lastReadArticles* liefert die zuletzt gelesenen Artikel, die gecacht

```
private List<Article> lastReadArticles;

public List<Article> fallbackNoArticlesService() {
  return lastReadArticles;
}

@Fallback(fallbackMethod = "fallbackNoArticlesService")
public List<Article> getArticles() throws NoDataAccess {

  List<Article> articles = new ArrayList<Article>();
  List<CoreArticle> coreArticles = new ArrayList<CoreArticle>();

  try {
    coreArticles = DataAccessManager.getArticlesDataAccess().getArticles(5);
  } catch (NoConnectivity e) {
    throw new NoDataAccess(e);
  }
```

Abb. 4: MicroProfile Fallback

Listing 1: „deployment-v1.yaml"

```
kind: Deployment
...
metadata:
  name: web-api-v1
spec:
  template:
    metadata:
      labels:
        app: web-api
        version: v1
...
```

wurden. Damit bekommt der Anwender keinen Fehler 500, sondern eine Liste von Artikeln, die vielleicht nicht ganz aktuell ist.

Beispiel 2: Eine der Stärken einer Microservices-Architektur ist die Möglichkeit, einen Service unabhängig von den anderen zu verändern (natürlich nur, solange ich das REST API nicht verändere), z. B. um einen Fehler zu beheben oder eine neue Funktion einzubauen. Bei den großen Internetplattformen testet man neue Funktionen oder Fixe gerne live, indem man sie ohne Ankündigung einer kleinen Testgruppe „unterschiebt", also einen Dark Launch oder Canary Test macht. Das ist eine meiner Lieblingsdemos für Istio und eine der ganz starken Istio- und damit Plattformfunktionen, sie heißt Traffic Routing.

Wenn ich eine neue Version eines Microservice in Kubernetes ausrolle (*kubectl apply …*), dann passiert ein Rolling Update, d. h. Kubernetes startet die neue Version und beendet die alte. Wenn danach etwas schiefgeht, mache ich ein Rollback, d. h. die neue Version wird beendet und die alte wieder gestartet. Das ist ein eher grober Test. Natürlich gibt es Möglichkeiten, Canary Testing oder Dark Launches in Kubernetes auszuführen, aber in Istio ist es ganz einfach.

Ich deploye beide Versionen parallel, mit eigenen Namen, und versehe sie aber mit Labels, z. B. *app: web-api, version: v1* für Version 1 (Listing 1). Version 2 findet sich in Listing 2.

Die Kubernetes-Service-Beschreibung sieht wie in Listing 3 aus, mit ihr werden beide Versionen des Web-API (v1 und v2) gleichermaßen angesprochen (50/50, *selector: app: web-api*).

Das ist alles ganz normales Kubernetes-Geschäft. Mit Istio kommen weitere Objekte hinzu, zuerst eine *DestinationRule*, über die man fürs Traffic Routing sogenannte Subsets definiert (Listing 4).

Und als weiteres Objekt ein *VirtualService*, in dem die eigentlichen „Verkehrsregeln" stehen, diese sind in unserem Fall etwas komplexer, da wir das Traffic Routing für einen Microservice verwenden, der direkt mit dem Frontend kommuniziert. Das heißt, wir verwenden ein Istio Ingress Gateway (*default-gateway-ingress-http*), um von der Web-App im Browser auf den Web-API-Service zuzugreifen. Dieses Istio Ingress Gateway enthält selbst wieder einen Envoy-Proxy, den wir für das Traffic Routing konfigurieren können (Listing 5).

Der *VirtualService* sorgt jetzt dafür, dass jeder Zugriff (*hosts: „*"*) auf den Istio Ingress über den URI */web-api/v1/getmultiple* zu 80 Prozent (*weight: 80*) über die Version 1 des Web-API geleitet wird, zu 20 Prozent über die Version 2. Man kann das auch schön in Kiali, dem Istio Dashboard, sehen (**Abb. 5**).

Listing 2: „deployment-v2.yaml"

```
kind: Deployment
...
metadata:
  name: web-api-v2
spec:
  template:
    metadata:
      labels:
        app: web-api
        version: v2
    ...
```

Listing 3: „service.yaml"

```
kind: Service
...
metadata:
  name: web-api
  labels:
    app: web-api
spec:
  selector:
    app: web-api
  ...
```

Listing 4: „destinationrule.yaml"

```
apiVersion: networking.istio.io/
                        v1alpha3
kind: DestinationRule
metadata:
  name: web-api
spec:
  host: web-api
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Listing 5: „virtualservice.yaml"

```
apiVersion: networking.istio.io/
                        v1alpha3
kind: VirtualService
metadata:
  name: virtualservice-ingress-
                web-api-web-app
spec:
  hosts:
  - "*"
  gateways:
  - default-gateway-ingress-http
  http:
  - match:
    - uri:
        prefix: /web-api/v1/
                        getmultiple
    route:
    - destination:
        host: web-api
        subset: v1
      weight: 80
    - destination:
        host: web-api
        subset: v2
      weight: 20
```

Was gibt es noch?

Istio und MicroProfile haben noch wesentlich mehr Funktionen, die ich hier nur noch anreißen möchte, wir haben alles in Blogartikeln dokumentiert.

Authorization und Authentication sorgen dafür, dass nur bestimmte und bekannte Anwender meine Anwendung oder sensitive Teile meiner Anwendung verwenden dürfen. Beides wird von Istio und von MicroProfile über Standards wie OpenID und JWT (JSON Web Token) in unterschiedlichem Umfang abgedeckt.

In unserem Beispiel verwenden wir den Cloud Service IBM App ID [10], dieser ist entweder selbst Identity Provider oder aber Proxy für Social Identity Provider wie Google und Facebook oder auch Enterprise Provider

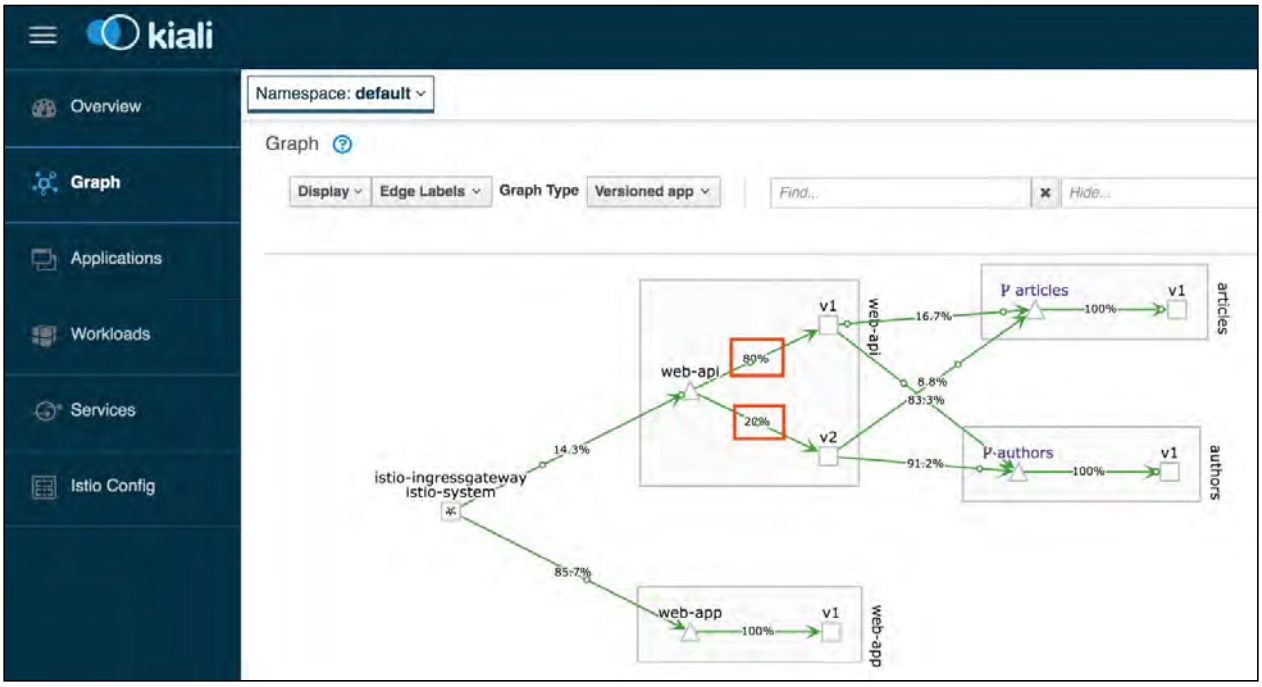

Abb. 5: Traffic Routing sichtbar gemacht in Kiali

im eigenen Rechenzentrum wie Active Directory über SAML. Diese Identity Provider haben eigene Log-in-Seiten, d. h., als Entwickler muss ich mich darum nicht kümmern, und bei Enterprise Providern funktioniert dann womöglich auch unternehmensweites Single Sign-on. Zu diesem Themenbereich hat mein Kollege Niklas Heidloff Grundlagenforschung betrieben und diese in zwei Artikeln dokumentiert [11], [12].

Ein weiteres wichtiges Thema für Microservices-Architekturen ist Observability. Eine Microservices-Architektur ist eine sehr lebendige und flüchtige Umgebung: Objekte (wie z. B. Pods) kommen und gehen; das kommt von der Skalierung, weil Fehler auftreten, oder als Nebeneffekt der Lastverteilung. Im Gegensatz zu einer monolithischen Anwendung haben wir keine einzelne Logdatei, sondern ganz viele verschiedene Stellen, an denen Logs weggeschrieben werden. Wir müssen uns also mit neuen Werkzeugen beschäftigen:

- *Distributed Tracing*: Ein Aufruf des Web-API in unserem Beispiel resultiert in nachgeordneten Aufrufen des Articles Services und des Authors Service. Mit Werkzeugen wie Jaeger [13] kann man diese „Trace Spans“ gemeinsam betrachten. Sowohl Istio als auch MicroProfile unterstützen Jaeger.
- *Monitoring und Metriken*: Istio enthält und MicroProfile unterstützt Prometheus als Monitoring- und Metrics-Werkzeug [14]. Anstatt sich selbst um den Betrieb eines Monitoring-Service zu kümmern, kann man aber auch externe Werkzeuge wie Sysdig benutzen, die beispielsweise auf der IBM Cloud als Service angeboten werden [15].
- *Central Logging*: In unserem Beispiel haben wir drei Microservices, die an drei verschiedenen Stellen ihre Logs schreiben. Im Fehlerfall möchte ich nicht an drei Stellen suchen müssen. Und das ist ja noch ein ganz simples Beispiel. In Microservices-Architekturen sind Distributed- oder Central-Logging-Systeme ganz essenziell. Ich kann in meinem Kubernetes Cluster z. B. Fluentd installieren und so ein System selbst betreiben – oder, wie eben für Monitoring

erwähnt, einen Logging-Service meines Cloudanbieters nutzen, z. B. LogDNA auf der IBM Cloud [15].
- *Health Check*: MicroProfile stellt über eine einfache Annotation ein Health-Check-Interface zur Verfügung [16]. Damit kann Kubernetes ganz einfach feststellen, wann mein Service bereit ist, Requests zu verarbeiten. Das kann beim Start einer Java-Anwendung ja durchaus einen Moment länger dauern.

Fazit

Wir haben mit unserem Projekt Cloud Native Starter ein einfaches Beispiel für eine moderne Cloud-native-Anwendung entwickelt, anhand dessen man die wesentlichen Aspekte dieser Art Anwendung lernen kann. Uns kam es darauf an, dass jeder dieses Beispiel nutzen kann, daher haben wir es auf Open-Source-Technologien aufgebaut und so schlank gehalten, dass es auf einem einigermaßen aktuellen Notebook lokal in Minikube betrieben werden kann. Nicht zuletzt haben wir das Projekt selbst zu Open Source erklärt. Um den Start in diese Welt so einfach wie möglich zu gestalten, haben wir jede Menge Skripte geschrieben, die so viel wie möglich automatisieren. Wir haben mittlerweile doppelt so viel Shell Code wie Java in GitHub. Aber damit kann man Cloud Native Starter in etwa einer Stunde komplett selbst aufsetzen. Es gibt also keinen Grund, es nicht selbst einmal auszuprobieren! Ich habe bei diesem Projekt auf jeden Fall viel gelernt.

Harald Uebele ist Developer Advocate bei IBM mit Standort Stuttgart. Er beschäftigt sich seit sechs Jahren mit Cloudtechnologien und ist ein großer Fan von Open Source und Linux.

Links & Literatur

[1] Istio Service Mesh: https://istio.io

[2] MicroProfile: https://microprofile.io

[3] https://github.com/IBM/cloud-native-starter

[4] Cloud-Native-Workshop: https://github.com/IBM/cloud-native-starter/tree/master/workshop

[5] https://www.cncf.io

[6] OpenJ9: https://www.eclipse.org/openj9/

[7] OpenJDK: https://openjdk.java.net

[8] Open Liberty: https://openliberty.io

[9] Developing resilient Microservices with Istio and MicroProfile: http://heidloff.net/article/resiliency-microservice-microprofile-java-istio

[10] IBM App ID: https://cloud.ibm.com/docs/services/appid?topic=appid-about

[11] Authorization in Cloud-Native Apps in Istio via OpenID: http://heidloff.net/article/authentication-authorization-openid-connect-istio

[12] Authorization in Microservices with MicroProfile: http://heidloff.net/article/authorization-microservices-java-microprofile/

[13] Istio und Jaeger: https://istio.io/docs/tasks/telemetry/distributed-tracing/jaeger/

[14] Istio und Prometheus: https://istio.io/docs/tasks/telemetry/metrics/collecting-metrics/

[15] What's going on (in my cluster)?: https://haralduebele.blog/2019/04/08/whats-going-on-in-my-cluster/

[16] Implementing Health Checks with MicroProfile and Istio: http://heidloff.net/article/implementing-health-checks-microprofile-istio

Entwurf einer funktionalen Softwarearchitektur

Hearts ist Trumpf!

Der Entwurf von nachhaltigen Softwarearchitekturen ist eine Herausforderung: Mit der Größe steigt in vielen klassisch objektorientierten Softwareprojekten die Komplexität überproportional an. Durch viel Disziplin und regelmäßige Refaktorisierungen lässt sich das Problem eine Weile in Schach halten, aber die wechselseitigen Abhängigkeiten und komplexen Abläufe von Zustandsveränderungen nehmen mit der Zeit trotzdem zu. Die funktionale Softwarearchitektur geht an die Strukturierung großer Systeme anders heran als objektorientierte Ansätze und vermeidet so viele Quellen von Komplexität und Wechselwirkungen im System.

von Michael Sperber und Peter Thiemann

Funktionale Softwarearchitektur steht für das Ergebnis eines Softwareentwurfs mit den Mitteln der funktionalen Programmierung. Sie zeichnet sich unter anderem durch folgende Aspekte aus:

- An die Stelle des Objekts mit gekapseltem Zustand tritt die Funktion, die auf unveränderlichen Daten arbeitet.
- Funktionale Sprachen (ob statisch oder dynamisch) erlauben ein von Typen getriebenes, systematisches Design von Datenmodellen und Funktionen.
- Statt starrer hierarchischer Strukturen entstehen flexible, in die funktionale Programmiersprache eingebettete domänenspezifische Sprachen.

Wir konzentrieren uns in diesem Artikel auf den ersten Punkt, also den Umgang mit Funktionen und unveränderlichen Daten. Dabei werden wir auch die Rolle von Typen beleuchten. Der Code zu diesem Artikel findet sich auf GitHub [1].

Funktionale Programmiersprachen

Funktionale Softwarearchitektur ist in (fast) jeder Programmiersprache möglich, aber in einer funktionalen Sprache wie Haskell, OCaml, Clojure, Scala, Elixir, Erlang, F# oder Swift ist diese Herangehensweise besonders natürlich. Funktionale Softwarearchitektur wird in der Regel als Code ausgedrückt, also nicht in Form von Diagrammen. Entsprechend benutzen wir für die Beispiele in diesem Artikel die funktionale Sprache Haskell [2], die besonders kurze und elegante Programme ermöglicht. Keine Sorge: Wir erläutern den Code, sodass er auch ohne Vorkenntnisse in Haskell lesbar ist. Wer dadurch auf Haskell neugierig geworden ist, kann sich eine Einführung in funktionale Programmierung [3], ein Buch zu Haskell [4] oder einen Onlinekurs [5] zu Gemüte führen.

Überblick

Wir erklären den Entwurf einer funktionalen Softwarearchitektur anhand des Kartenspiel Hearts [6], von dem wir nur die wichtigsten Teile umsetzen.

Hearts wird zu viert gespielt. In jeder Runde eröffnet eine Spielerin, indem sie eine Karte ausspielt (zu Beginn des Spiels muss das die Kreuz-Zwei sein.) Die nächste Spielerin muss, wenn möglich, eine Karte mit der gleichen Farbe wie die Eröffnungskarte ausspielen. Andernfalls darf sie eine beliebige Karte abwerfen. Haben alle Spielerinnen eine Karte ausgespielt, muss die Spielerin den Stich einziehen, deren Karte die gleiche Farbe wie die Eröffnungskarte sowie den höchsten Wert hat. Ziel ist, mit den eingezogenen Karten einen möglichst geringen Punktestand zu erreichen. Dabei zählt die Pik-Dame 13 Punkte und jede Herzkarte einen Punkt, alle weiteren Karten null Punkte.

Modellierung des Spielablaufs

Als Basis des Entwurfs verwenden wir ein klassisches taktisches Entwurfsmuster aus dem Domain-driven Design (DDD) [7] und modellieren das Kartenspiel auf der Basis von Domain Events. Die Events repräsentieren jedes Ereignis, das im Spielverlauf passiert ist, die Commands repräsentieren Wünsche der Beteiligten, dass etwas passiert. Die Architektur ist auf diese Weise offen für eine spätere Umstellung auf Client-Server-Betrieb, Microservices oder Event Sourcing.

Abbildung 1 zeigt den Ablauf: Jede Spielerin nimmt Events entgegen – was im Spiel gerade passiert ist – und

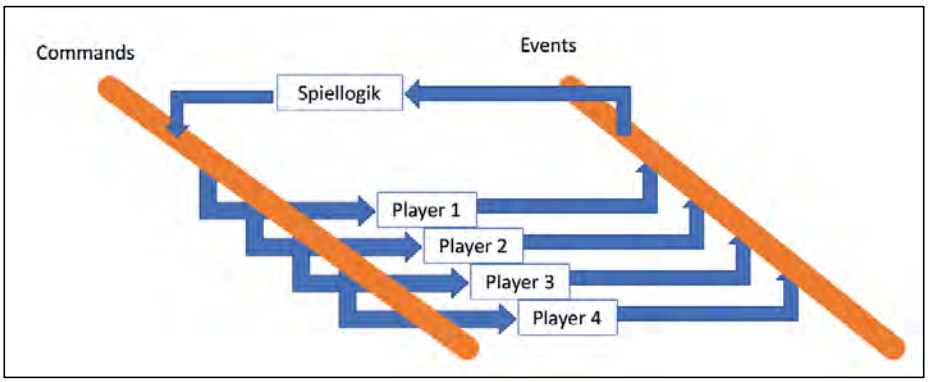

Abb. 1: Spielablauf durch Commands und Events

generiert dafür Commands, die Spielzüge repräsentieren. Die Spiellogikkomponente nimmt die Commands entgegen, überprüft sie auf Korrektheit (War die Spielerin überhaupt dran? War der Spielzug regelkonform?) und generiert ihrerseits daraus wieder Events. Das Entwurfsmuster ist das gleiche wie in objektorientiertem DDD, aber die Umsetzung unterscheidet sich durch die Verwendung von unveränderlichen Daten und Funktionen.

Im Beispiel stellen wir die Kommunikation zwischen den einzelnen Komponenten der Architektur direkt mit Funktionsaufrufen her, aber auch andere Mechanismen – nebenläufige Prozesse oder Microservices – sind möglich.

Programmieren mit unveränderlichen Daten

Eine Vorbemerkung: „Unveränderliche Daten" bedeutet, dass es keine Zuweisungen gibt, die Attribute von Objekten verändern können. Wenn Veränderung modelliert werden soll, so generiert ein funktionales Programm neue Objekte. Diese Vorgehensweise bietet enorme Vorteile:

- Es gibt niemals Probleme mit verdeckten Zustandsänderungen durch Methodenaufrufe oder nebenläufige Prozesse.
- Es gibt keine inkonsistenten Zwischenzustände, wenn ein Programm erst das eine Feld, dann das nächste etc. setzt.
- Das Programm kann problemlos durch ein Gedächtnis erweitert werden, das zum Beispiel zu früheren Spielständen zurückkehrt, wenn eine Spielerin ihren Zug zurücknimmt.
- Es gibt keine verstecken Abhängigkeiten durch die Kommunikation von Zustand hinter den Kulissen.

Aus den gleichen Gründen ist in Java das Programmieren mit Value-Objekten oft nützlich.

Karten modellieren

Die konkrete Modellierung beginnt mit den Spielkarten. Der Datentyp *Card* ist ein Record-Typ (analog zu einem POJO in Java) und legt damit fest, dass eine Karte eine Farbe (*suit*) und einen Wert (*rank*) hat:

```
data Card = Card { suit :: Suit,
                   rank :: Rank }
```

Weiter werden noch Definitionen von *suit* und *rank* benötigt:

```
data Suit = Diamonds | Clubs |
Spades | Hearts

data Rank = Numeric Integer | Jack
| Queen | King | Ace
```

Hier handelt es sich um Aufzählungen (vergleichbar mit *enum* in Java). Das | steht für „Oder", entsprechend steht dort: Ein Suit ist Diamonds oder Clubs oder Spades oder Hearts. Bei *Rank* ist es ähnlich – zusätzlich hat eine der Alternativen, *Numeric*, ein Feld vom Typ *Integer*, das den Wert einer Zahlenspielkarte angibt.

Hier ist die Definition der Kreuz-Zwei auf Basis dieser Datentypdefinition in Form einer Gleichung:

```
twoOfClubs = Card Clubs (Numeric 2)
```

Das Beispiel zeigt, dass Card als Konstruktorfunktion agiert. Außerdem auffällig: In Haskell werden Funktionsaufrufe ohne Klammern und Komma geschrieben, Klammern dienen nur zum Gruppieren. Die Deklaration von Cards definiert auch die Getter-Funktionen *suit* und *rank*, die wie Funktionen verwendet werden.

Kartenspiele und Hände

Für Hearts wird ein kompletter Satz Karten benötigt. Dieser wird durch folgende Definitionen generiert, die jeweils eine Liste aller Farben, eine Liste aller Werte und schließlich daraus eine Liste aller Karten (also aller Kombinationen aus Farben und Werten) konstruiert:

```
allSuits :: [Suit]
allSuits = [Spades, Hearts, Diamonds, Clubs]

allRanks :: [Rank]
allRanks = [Numeric i | i <- [2..10]] ++ [Jack, Queen, King, Ace]

deck :: [Card]
deck = [Card suit rank | rank <- allRanks, suit <- allSuits]
```

Jede Definition wird von einer Typdeklaration wie *allSuits :: [Suit]* begleitet. Das bedeutet, dass *allSuits* eine Liste (die eckigen Klammern) von Farben ist. Die Definitionen für *allRanks* und *deck* benutzen sogenannte Comprehension-Syntax (Pythons Comprehensions sind von den funktionalen Sprachen Haskell und Miranda inspiriert), um die Werte und schließlich alle Karten aufzuzählen.

Für die Umsetzung eines Kartenspiels müssen die Karten repräsentiert werden, die eine Spielerin auf der Hand hat: als Menge (*set*) von Karten.

```
type Hand = Set Card
```

Der Typ *Set* ist ein generischer Typ, der von der Standardbibliothek importiert wird. Listing 1 zeigt die Typsignaturen der Funktionen aus *Set*.

Für Hand wird eine *type*-Deklaration verwendet, die ein Typsynonym definiert.

Einige Hilfsdefinitionen erleichtern den Umgang mit dem Typ *Hand*. Die Funktion *isHandEmpty* hat den Typ *Hand -> Bool*, was eine „Funktion, die eine Hand als Eingabe nimmt und ein Bool als Ausgabe liefert" beschreibt. In der Gleichung für *isHandEmpty* steht nach dem Funktionsname der Name des Parameters *hand*:

```
isHandEmpty :: Hand -> Bool
isHandEmpty hand = Set.null hand
```

Die Funktion *containsCard* nimmt zwei Argumente und prüft mit Hilfe der Library-Funktion *Set.member*, ob eine gegebene Karte zu einer Hand gehört:

```
containsCard :: Card -> Hand -> Bool
containsCard card hand = Set.member card hand
```

Der Typ *Card -> Hand -> Bool* wird erst verständlich, wenn er von rechts geklammert wird: *Card -> (Hand -> Bool)*. Das bedeutet, dass die Funktion zunächst eine Karte akzeptiert und dann eine Funktion liefert, die ihrerseits eine Hand akzeptiert und dann einen booleschen Wert zurückliefert. Streng genommen kennt Haskell nur einstellige Funktionen und simuliert mehrstellige Funktionen durch diese Technik [8].

Die nächste Funktion zeigt beispielhaft, wie der Umgang mit unveränderlichen Daten funktioniert – *removeCard* entfernt eine Karte aus einer Hand:

```
removeCard :: Card -> Hand -> Hand
removeCard card hand = Set.delete card hand
```

In idiomatischem Java hätte das Hand-Objekt eine Methode *void removeCard(Card card)*, die den Zustand des Hand-Objekts entsprechend verändert. Nicht so in der funktionalen Programmierung, wo *removeCard* eine neue Hand liefert und die alte Hand unverändert lässt. Nach

```
hand2 = removeCard card hand1
```

ist *hand1* immer noch die „alte" Hand und *hand2* die neue.

Das ist in Haskell nicht nur eine Konvention: Eine Funktion kann nicht einfach so Objekte verändern, es handelt sich im Sprachgebrauch der funktionalen Programmierung immer um eine „reine" oder „pure" Funktion. In der rein funktionalen Programmierung ist die Typsignatur enorm nützlich, weil sie alles aufführt, was in die Funktion hinein und wieder hinausgeht: Es gibt keine versteckten Abhängigkeiten zu globalem Zustand, und alle Ausgaben stehen hinter dem rechten Pfeil der Signatur. Daher sind die rechten und linken Seiten einer Definitionsgleichung überall im Programm jederzeit austauschbar.

Typvariablen und Higher-Order-Funktionen

Da Hearts ein sogenanntes Stichspiel ist, hat der Code einen Typ für den Stich (engl.: trick). Dieser muss mitführen, wer welche Karte ausgespielt hat, um nach einer Runde zu entscheiden, wer den Stich einziehen muss:

```
type PlayerName = String

type Trick = [(PlayerName, Card)]
```

Die Typdefinition von *Trick* besagt, dass ein Stich eine Liste (die eckigen Klammern) von Zwei-Tupeln (die runden Klammern innendrin) ist. Wir verwenden hier Listen, da die Reihenfolge der Karten wichtig ist, wenn es darum geht, welche Spielerin den Stich bekommt. Listen werden in funktionalen Sprachen von hinten nach vorn aufgebaut, die zuletzt ausgespielte Karte ist also vorn.

Wenn der Stich eingezogen wird, zählen nur noch die Karten, nicht mehr, von wem sie stammen. Dafür ist folgende Funktion nützlich:

```
cardsOfTrick :: Trick -> [Card]
cardsOfTrick trick = map snd trick
```

Was die Funktion macht, ist wieder aus der Typsignatur ersichtlich. Interessant ist hier aber auch die Implementierung, weil sie die eingebaute Higher-Order-Funktion *map* bemüht, deren Typsignatur so aussieht:

```
map :: (a -> b) -> [a] ->[b]
```

a und *b* sind Typvariablen, das Pendant zu Java Generics. Ausgesprochen steht dort: *map* akzeptiert eine Funktion, die aus einem *a* ein *b* macht, sowie eine Liste von *a*s und liefert eine Liste von *b*s. Bei *cardsOfTrick* ist *a* das Zwei-Tupel *(PlayerName, Card)* und *b* ist *Card*. Die Funktion *snd* extrahiert aus dem Zwei-Tupel die zweite Komponente und hat deshalb folgenden Typ:

```
snd :: (a, b) -> b
```

Solche generischen Funktionen gibt es (inzwischen) auch in Java, aber in funktionalen Sprachen kommen

Listing 1

```
Set.null :: Set -> Bool
Set.member :: a -> Set a -> Bool
Set.insert :: a -> Set.Set a -> Set.Set a
Set.delete :: a -> Set a -> Set a
```

sie im Zusammenhang mit Higher-Order-Funktionen viel häufiger zur Anwendung. Sie sind ein wichtiger Aspekt funktionaler Architektur: Diese macht nicht an der konkreten Modellierung von fachlichem Wissen halt, sondern erlaubt den Entwicklerinnen und Entwicklern, Abstraktionen zu bilden, die das fachliche Wissen verallgemeinern.

Das Zusammenspiel von generischen Higher-Order-Funktionen und anderen reinen Funktionen liefert die Basis für ein extrem mächtiges Konstruktionsprinzip: Weil Funktionen nie etwas Verstecktes machen, können sie bedenkenlos zu immer größeren Gebilden zusammengestöpselt werden. In OO-Sprachen wächst jedoch mit der Größe das Risiko, dass versteckte Effekte unerwünschte Wechselwirkungen haben. Deshalb sind Disziplin und eigene architektonische Patterns notwendig, um die resultierende Komplexität in den Griff zu bekommen. In der funktionalen Programmierung ist das nicht so. Dementsprechend ist dort das „Programmieren im Großen“ dem „Programmieren im Kleinen“ ziemlich ähnlich.

Spiellogik

Die Spiellogik bildet den Mittelbau der Architektur. Ein Event Storming liefert folgende Event-Klassen:

- Die Karten wurden ausgeteilt.
- Eine neue Spielerin ist an der Reihe.
- Eine Spielerin hat eine bestimmte Karte ausgespielt.
- Eine Spielerin hat den Stich aufgenommen.
- Eine Spielerin hat versucht, eine unzulässige Karte auszuspielen.
- Das Spiel ist vorbei.

Diese Klassen lassen sich direkt in eine Typdefinition übersetzen:

```
data GameEvent =
    HandsDealt (Map PlayerName Hand)
  | PlayerTurn PlayerName
  | CardPlayed PlayerName Card
  | TrickTaken PlayerName Trick
  | IllegalMove PlayerName
  | GameOver
```

Das *HandsDealt* Event trägt eine „Map“ zwischen Spielernamen und ihren Karten mit sich. *HandsDealt* ist als Domain Event eigentlich ungünstig, weil es an alle Spielerinnen verteilt wird, die so nicht nur ihre eigene Hand erfahren, sondern auch die der anderen. Wir haben es hier aus Gründen der Einfachheit gewählt, aber besser wären separate *HandDealt* Events für jeweils nur eine Spielerin, von denen jede Spielerin nur jeweils ihres bekommt. Der Verlauf eines Spiels kann immer aus dessen Folge von Events rekonstruiert werden. Es gibt nur zwei Klassen von Commands:

```
data GameCommand =
    DealHands (Map PlayerName Hand)
  | PlayCard PlayerName Card
```

Die erste Klasse ist das direkte Pendant zu *HandsDealt*; sie setzt das Spiel zu Beginn in Gang. Die zweite repräsentiert den Versuch einer Spielerin, eine bestimmte Karte auszuspielen.

Die Spielregeln werden durch die Verarbeitung von Commands zu Events implementiert. Die Regeln beziehen sich auf den Zustand des Spiels: welche Karten ausgespielt werden dürfen, welche Spielerin als Nächstes dran ist etc. Von diesem Zustand wird die Beantwortung folgender Fragen verlangt:

- Wer sind die Spieler und in welcher Reihenfolge spielen sie?
- Was hat jede Spielerin auf der Hand?
- Welche Karten hat jede Spielerin eingezogen?
- Was liegt auf dem Stich?

Das alles wird durch eine weitere Record-Definition repräsentiert:

```
data GameState =
  GameState
  { gameStatePlayers :: [PlayerName],
    gameStateHands   :: PlayerHands,
    gameStateStacks  :: PlayerStacks,
```

```
  gameStateTrick   :: Trick
}
```

Die Liste im Feld *gameStatePlayers* wird dabei immer so rotiert, dass die nächste Spielerin vorn steht. Für die beiden Felder *gameStateHands* und *gameStateStacks* müssen jeweils Karten *pro* Spieler vorgehalten werden, darum sind die dazugehörigen Typen Synonyme für Maps:

```
type PlayerStacks = Map PlayerName (Set Card)
type PlayerHands  = Map PlayerName Hand
```

Auch bei der Umsetzung der Spielregeln macht sich die funktionale Architektur bemerkbar: Während des Spiels wird der Zustand nicht verändert, sondern jede Änderung erzeugt einen neuen Zustand. Die zentrale Funktion für die Verarbeitung eines Commands hat deswegen folgende Signatur:

```
processGameCommand :: GameCommand -> GameState ->
                                        (GameState, [GameEvent])
```

Mit anderen Worten: Command rein, (Repräsentation des) Zustand(s) vorher rein, Tupel aus dem neuen Zustand bzw. Repräsentation des neuen Zustands und Liste resultierender Events raus. Hier ist die Implementierung der Gleichung für das *DealHands* Command:

```
processGameCommand (DealHands hands) state =
  let event = HandsDealt hands
  in (processGameEvent event state, [event])
```

Listing 2

```
processGameCommand (PlayCard player card) state =
  if playValid state player card
  then
    let event1 = CardPlayed player card
        state1 = processGameEvent event1 state
    in  if turnOver state1 then
          let trick = gameStateTrick state1
            trickTaker = whoTakesTrick trick
            event2 = TrickTaken trickTaker trick
            state2 = processGameEvent event2 state1
            event3 = if gameOver state2
                     then GameOver
                     else PlayerTurn trickTaker
            state3 = processGameEvent event3 state2
          in (state3, [event1, event2, event3])
        else
          let event2 = PlayerTurn (nextPlayer state1)
              state2 = processGameEvent event2 state1
          in (state2, [event1, event2])
  else
    (state, [IllegalMove player, PlayerTurn player])
```

Da dieser Befehl von der „Spielleitung“ kommt, führt er immer zu einem *HandsDealt* Event. Der Effekt des Events auf den Zustand wird durch die Funktion *processGameEvent* berechnet, deren Definition aus Platzgründen fehlt, aber deren Arbeitsweise sich wieder gut an der Typsignatur ablesen lässt:

```
processGameEvent :: GameEvent -> GameState -> GameState
```

Die Kernlogik ist in der Gleichung für das *PlayCard*-Command in Listing 2 zu finden. Diese verlässt sich auf die Hilfsfunktionen *playValid* (die einen Spielzug auf Korrektheit überprüft), *whoTakesTrick* (die berechnet, wer den Stich einziehen muss) und *gameOver* (die feststellt, ob das Spiel vorbei ist). Der Code führt eine Reihe von Fallunterscheidungen in Form von *if ... then ... else* durch und bindet lokale Variablen – insbesondere Events *event1, event2* etc. und Zwischenzustände *state1, state2* – mit dem Sprachkonstrukt *let* (Listing 2).

Es ist deutlich zu sehen, dass der Zustand niemals verändert wird und dass alle Zwischenzustände separate, voneinander unabhängige Objekte sind.

Zustand verwalten

Die Funktion *processGameCommand* bildet die Abfolge von Zuständen durch unterschiedliche Variablen und die Abhängigkeiten dazwischen ab. Das ist in vielen Situationen gerade richtig, hier aber unnötig fehleranfällig – wenn zum Beispiel irgendwo statt *state3* mal *state2* steht. Unnötig ist es deshalb, weil die Funktion einen sequenziellen Prozess abbildet – und dem wäre besser durch eine sequenzielle Notation gedient.

Sequenzielle Prozesse lassen sich gut durch Monaden beschreiben, ein typisch funktionales Entwurfsmuster. Mit Monaden entstehen immer noch Funktionen, aber die Notation wechselt in eine sequenzielle Form. Diese hat Zugriff auf einen Kontext, der sowohl gelesen als auch beschrieben werden kann. Anders als in Java ist aber genau definiert, was in dem Kontext alles steht und was also die monadische Form kann und was nicht.

In Listing 3 steht eine monadische Version von *processGameCommandM*. Sie unterscheidet sich von der funktionalen Version folgendermaßen:

- Der Zustand und die Event-Liste sind implizit.
- Die Hilfsfunktion *processAndPublishEventM* verarbeitet den Effekt eines Events auf den Zustand und „speichert" das Event ab.
- An die Stelle der Hilfsfunktionen *playValid*, *turnOver*, *currentTrick*, *gameOver* treten monadische Versionen mit *M* hinten am Namen jeweils ohne *state*-Argument, die den Spielzustand aus dem Kontext beziehen. Auch anstelle des normalen *if* tritt die monadische Version *ifM*.
- Wenn mehrere monadische Aktionen hintereinander laufen, werden sie in einem *do*-Block untergebracht, ähnlich der geschweiften Klammern in Java. Dort wird <- benutzt, um das Ergebnis einer monadischen Operation an eine Variable zu binden.

Das Ergebnis in Listing 3 sieht zumindest strukturell fast wie ein Java-Programm aus.

Die Funktion sieht zwar aus, als würde sie Änderungen durchführen – tatsächlich aber liefert sie nur eine Beschreibung dieser Änderungen. Diese Beschreibung muss dann explizit ausgewertet werden. Im Fall von Zustand heißt das zum Beispiel, dass aus der Beschreibung eine Funktion wird, die einen Zustand als Eingabe akzeptiert und einen neuen Zustand als Ausgabe liefert.

Welche Änderungen in der Monade gemacht werden können, ist durch den Typ der Funktion eingeschränkt:

```
processGameCommandM :: GameInterface m => GameCommand -> m ()
```

Der Typ zerfällt in zwei Teile, die durch den Doppelpfeil => getrennt sind. Im rechten Teil taucht eine Typvariable *m* auf, die für die Monade steht. Links davon steht ein sogenannter Constraint, der sagt, was für Eigenschaften *m* haben muss. In diesem Fall steht dort, dass *m* das Interface *GameInterface* erfüllen muss – dazu gleich mehr. Wenn dem so ist, liefert *processGameCommandM* zu jedem Kommando eine Berechnung, die *()* produziert, das steht für „kein explizites Ergebnis" – die Ergebnisse sind alle implizit. *GameInterface* hat folgende Definition:

```
type GameInterface m = (MonadState GameState m, MonadWriter
                                                [GameEvent] m)
```

Das *GameInterface* enthält zwei Features, die *processGameCommandM* benutzen darf: sie darf auf den Spielzustand zugreifen (*MonadState GameState*) und sie darf *GameEvents* bekannt geben. Mehr kann *processGameCommandM* nicht tun. Da ist also der wesentliche Unterschied zu Java – das heißt, nicht ganz: In Java gibt es ja die *throws*-Klausel an Methoden, die besagt, welche Exceptions eine Funktion werfen kann.

Da der Zustand nicht mehr explizit herumgereicht wird, muss sich eine Aktion wie *playValidM* den aktuellen *GameState* erst besorgen, indem sie die Funktion *State.get* benutzt, die zu *MonadState* gehört:

```
playValidM :: MonadState GameState m => PlayerName -> Card -> m Bool
playValidM playerName card = do
  state <- State.get
  return (playValid state playerName card)
```

Diese Funktion hat noch wenigere Anforderungen an die Monade, denn sie verlangt nur den Zustandsanteil. Die Aktion *state <- State.get* besorgt den aktuellen Zustand, der für die folgenden Aktionen in der Variable *state* zur Verfügung steht. Das verbleibende *return* gibt den Zustand unverändert weiter und liefert als Ergebnis den Wert von *playValid*. Auf die gleiche Art funktionieren auch die Aktionen *turnOverM*, *currentTrickM*, *gameOverM* und *nextPlayerM*. Die Funktion *processAndPublishEventM* verarbeitet Events und verschickt sie:

```
processAndPublishEventM :: GameInterface m => GameEvent -> m ()
processAndPublishEventM gameEvent =
  do processGameEventM gameEvent
     Writer.tell [gameEvent]
```

Listing 3

```
processGameCommandM (DealHands playerHands) =
  processAndPublishEventM (HandsDealt playerHands)
processGameCommandM (PlayCard playerName card)
  ifM (playValidM playerName card)
    (do
      processAndPublishEventM (CardPlayed playerName card)
      ifM turnover
        (do
          trick <- currentTrickM
          let trickTaker = whoTakesTrick trick
          processAndPublishEventM (TrickTaken trickTaker trick)
          ifM gameOverM
            (processAndPublishEventM (GameOver))
            (processAndPublishEventM (PlayerTurn trickTaker)))
        (do                    -- not turnover
          nextPlayer <- nextPlayerM
          processAndPublishEventM (PlayerTurn nextPlayer)))
    (do                        -- not playValid
      nextPlayer <- nextPlayerM
      processAndPublishEventM (IllegalMove nextPlayer)
      processAndPublishEventM (PlayerTurn nextPlayer))
```

Die Funktion benutzt das andere Feature in *GameInterface – MonadWriter*, wo es eine Funktion *Writer.tell* gibt, die das Event speichert und bekanntgibt. Außerdem benutzt sie die monadische Version von *processGameEvent*, deren erste Gleichung folgendermaßen lautet:

```
processGameEventM :: GameInterface m => GameEvent -> m ()
processGameEventM (HandsDealt playerHands) =
  do gameState <- State.get
     State.put (gameState { gameStateHands = playerHands })
```

Sie holt sich also den impliziten Zustand und schreibt dann eine neue Version zurück, in der die verteilen Karten vermerkt sind.

Alle Funktionen mit Effekten in solche monadische Form zu bringen, ist anstrengend. Es sorgt aber dafür, dass an den Funktionssignaturen deutlich zu sehen ist, welches Arsenal von Effekten sie benötigen. Außerdem ist das Resultat jeweils immer noch eine Funktion, die unter kontrollierten Umständen aufgerufen werden kann.

Spielerlogik

Während die Spiellogik Kommandos entgegennimmt und dafür Events generiert, funktioniert die Spielerlogik genau andersherum. Sie nimmt Events entgegen und liefert als Antwort Kommandos, die an die Spiellogik weitergegeben werden.

Die Implementierung der Spielerlogik funktioniert wieder mit Monaden, um die Formulierung zu erleichtern und gleichzeitig explizit zu machen, welcher Effekte sich eine Spielerin bedient. Das heißt, jede Spielerin wird in einer abstrakten Monade gestartet, von der sie nur weiß, dass sie Kommandos an die Spiellogik schicken kann und Zugriff auf I/O-Operationen hat – zum Beispiel, um über ein GUI zu interagieren oder den Telefonjoker anzurufen. Diese Features werden als Constraints ausgedrückt:

```
type PlayerInterface m = (MonadIO m, MonadWriter [GameCommand] m)
```

Jede Spielerin kann nun für sich selbst entscheiden, welche weiteren Features sie lokal verwenden möchte. Typischerweise verwaltet jede Spielerin ihre eigene Version vom Spielzustand, weil sie nicht auf den *GameState* der Spiellogik zugreifen kann. Die Ausgestaltung dieses Spielzustands ist der Spiellogik völlig gleichgültig und kann auch von jeder Spielerin anders gehandhabt werden. Diese Spielerlogik wird natürlich durch eine Funktion repräsentiert, die zusammen mit dem Namen der Spielerin in ein Record verpackt wird:

```
data Player =
  Player {
    playerName :: PlayerName,
    eventProcessor :: forall m . PlayerInterface m => GameEvent -> m Player
  }
```

Im *Player* hat eine Spielerin einen Namen und eine Event-Prozessor-Funktion, die ein *GameEvent* als Aktion in einer Spielermonade *m* interpretiert. Dieses *m* kann beliebig gewählt werden (das wird durch das *forall m* ausgedrückt) und kann sich darauf verlassen, dass das *PlayerInterface* vom Aufrufer zur Verfügung gestellt wird, also das Gegenstück zu *GameInterface*.

In der Regel wird eine Spielerin im Spiel dazulernen wollen – also ihre Beobachtung des Spielverlaufs benutzen, um möglichst gute Spielzüge auszuwählen. Dazu muss sie sich Dinge merken, und das tut sie, indem ihr *eventProcessor* ein neues *Player*-Objekt zurückliefert, das in der nächsten Runde den Platz des alten einnimmt.

Es bleibt die Implementierung einer Spielerin. Um die Regeln einzuhalten, muss jede Spielerin sich daran erinnern, welche Karten sie auf der Hand hat und was auf dem Stich liegt. Der Typ dazu sieht so aus:

```
data PlayerState =
  PlayerState { playerHand  :: Hand,
                playerTrick :: Trick }
```

Hilfreich ist die Funktion *playerProcessGameEventM*, die den Spielerzustand entsprechend den eingehenden Events ändert. Sie benötigt das normale *PlayerInterface*, aber auch einen *PlayerState* als Zustand:

```
playerProcessGameEventM :: (MonadState PlayerState m, PlayerInterface m)
                                  => PlayerName -> GameEvent -> m ()
```

Der Code in Listing 4 implementiert eine Spielerin, die bei Spielbeginn die Kreuz-Zwei ausspielt, falls sie die Karte hat. Danach wählt sie jeweils eine passende Karte

Listing 4

```
playAlongProcessEventM ::
  (MonadState PlayerState m, PlayerInterface m) =>
    PlayerName -> GameEvent -> m ()
playAlongProcessEventM playerName event =
  do playerProcessGameEventM playerName event
     playerState <- State.get
     case event of
       HandsDealt _ ->
         if Set.member twoOfClubs (playerHand playerState)
         then Writer.tell [PlayCard playerName twoOfClubs]
         else return ()

       PlayerTurn turnPlayerName ->
         if playerName == turnPlayerName
         then do card <- playAlongCard
                 Writer.tell [PlayCard playerName card]
         else  return ()

       _ -> return ()
```

mit der Hilfsfunktion *playAlongCard* aus und spielt sie aus. Das tut sie, indem sie *Writer.tell* mit einem *PlayCard*-Command aufruft. Um verschiedene Events zu unterscheiden, benutzt die Funktion ein *case*-Konstrukt, das analog zu *switch* in Java funktioniert – der letzte Zweig _ entspricht dem Java-*default*.

Diese Spielstrategie wird von der folgenden Funktion in einem *Player*-Objekt verpackt. Die Strategie akzeptiert einen expliziten Zustand vom Typ *PlayerState*. Die dazugehörige Beschreibung der Zustandsveränderung wird in *playAlongProcessEventM* mit Hilfe der eingebauten Funktion *State.execStateT* explizit gefüttert und auch wieder herausgeholt, unter dem Namen *nextPlayerState* – und der wird dann im nächsten Player verwendet.

```
playAlongPlayer :: PlayerName -> PlayerState -> Player
playAlongPlayer playerName playerState =
  let nextPlayerM event =
    do nextPlayerState <-
       State.execStateT (playAlongProcessEventM playerName event) playerState
    return (playAlongPlayer playerName nextPlayerState)
  in Player playerName nextPlayerM
```

Der Aufruf *State.execStateT* zeigt, dass *playAlongProcessEventM* trotz der monadischen Form eine ganz normale Funktion ist. Sie mutiert keine Variablen wie das in Java der Fall wäre. Stattdessen muss ein Programm sie explizit durch *State.execStateT* mit einem Anfangszustand aufrufen und bekommt dann einen expliziten Resultatzustand zurück.

Zusammenfassung

Es fehlen noch einige Funktionen, um die Spiellogik mit der Spielerlogik zu verdrahten. Diese fehlen hier aus Platzgründen, aber der vollständige Code ist im GitHub Repository zu finden.

Wichtig war uns, zu demonstrieren, wie Monaden Effekte beschreiben und in den Typsignaturen einschränken. Funktionale Softwarearchitektur bedeutet dementsprechend erst einmal, dass ein Entwickler nicht so leicht Architekturprinzipien verletzen kann: Nicht mal so eben schnell ein Attribut verändern oder – speziell in Haskell – externe Effekte auslösen, weil es gerade so passt. Stattdessen führen neue Zustände immer gleich zu neuen Objekten, und alle Effekte müssen explizit deklariert werden. Das ist für Entwicklerinnen mit OO-Hintergrund gewöhnungsbedürftig. Diese Einschränkungen bringen aber Verbesserungen der Architekturqualität mit sich: geringere Kopplung, weniger Abhängigkeiten, deklarierte und kontrollierte Effekte – das alles steigert die Robustheit, Flexibilität und Wartbarkeit des Codes.

Michael Sperber ist Geschäftsführer der Active Group GmbH. Er wendet seit über 25 Jahren funktionale Programmierung in Forschung, Lehre und industrieller Entwicklung an. Er ist Mitbegründer des Blogs funktionale-programmierung.de und Mitorganisator der jährlichen Entwicklerkonferenz BOB. Er ist außerdem Mitautor des iSAQB-Advanced-Curriculums „FUNAR – Funktionale Softwarearchitektur".

michael.sperber@active-group.de

Peter Thiemann ist Professor für Informatik an der Universität Freiburg und leitet dort den Arbeitsbereich Programmiersprachen. Er ist einer der führenden Experten zur funktionalen Programmierung, der partiellen Auswertung, domänenspezifischen Sprachen und zahlreichen anderen Themen der Softwaretechnik. Seine aktuelle Forschung beschäftigt sich mit statischen und dynamischen Analysemethoden für JavaScript sowie Typsystemen für Protokolle.

thiemann@informatik.uni-freiburg.de

Links & Literatur

[1] https://github.com/funktionale-programmierung/hearts/

[2] https://www.haskell.org/

[3] Sperber, Michael; Klaeren, Herbert: „Schreibe Dein Programm!": https://www.deinprogramm.de/

[4] Hutton, Graham: „Programming in Haskell", 2nd edition, 2016

[5] z.B. Breitners, Joachim: „Haskell for Readers": http://haskell-for-readers.nomeata.de/

[6] Hearts auf Wikipedia: https://de.wikipedia.org/wiki/Hearts

[7] Vaughn, Vernon: „Domain-Driven Design Distilled", Pearson, 2016

[8] Funktionale Programmierer sprechen von Curried Functions und Currying: https://de.wikipedia.org/wiki/Currying

Nachhaltige Angular-Anwendungen mit taktischem DDD und Monorepos

Domain-driven Design in Angular?

Taktisches DDD hilft bei der Beherrschung der steigenden Komplexität in Single Page Applications (SPAs) und harmoniert noch dazu gut mit den in der Angular-Welt anzutreffenden Gepflogenheiten. Wie lassen sich bewährte Architekturkonzepte wie z. B. DDD nun in Verbindung mit modernen JavaScript-Businessanwendungen nutzen?

von Manfred Steyer

Geschäfts- und Industrieanwendungen sind in der Regel langlebig. Eine Lebenszeitspanne von einer oder mehreren Dekaden ist keine Seltenheit. Dazu kommt, dass immer mehr Teile dieser Anwendungen zur Steigerung der Benutzerfreundlichkeit ins Frontend wandern und dort mittels JavaScript implementiert sind. Somit stellt sich die Frage, wie sich bewährte Architekturkonzepte in der Welt von JavaScript Frameworks einsetzen lassen.

Dieser Artikel liefert eine Antwort, die sich in der Praxis des Autors bereits mehrfach bewährt hat. Es geht dabei um die Nutzung von Domain-driven Design in Angular-Anwendungen sowie um die gemeinsame Nutzung von Best Practices aus beiden Welten. Da bereits in [1] über den Einsatz von Strategic Design in Angular-Anwendungen geschrieben wurde, fokussiert sich dieser Artikel auf die andere Seite der Medaille: Tactical Design. Die verwendeten Beispiele finden sich wie immer in meinem GitHub-Account unter [2].

Vertikale und horizontale Trennlinien

Domain-driven Design sieht vor, dass ein Gesamtsystem in mehrere kleine, möglichst autarke Subdomänen zu untergliedern ist. Jede Subdomäne ist separat zu modellieren und erhält ihre eigenen Entitäten, die den jeweiligen Geschäftsbereich bestmöglich widerspiegeln. Dieses Vorgehen nennt sich auch Strategic Design [1]. Sind diese Subdomänen erst einmal identifiziert, stellt sich die Frage, wie sie strukturiert werden sollen. Eine klassische Vorgehensweise sieht die Unterteilung in Schichten vor. Diesen Ansatz verfolgt auch der vorliegende Text (**Abb. 1**).

Alternativ zur Schichtentrennung lassen sich natürlich auch eine hexagonale Architektur oder Ideen aus Clean Architecture einsetzen. Dank des in Angular integrierten Dependency-Injection-Mechanismus gestalten sich auch solche Implementierungen sehr gradlinig.

Wie **Abbildung 1** zeigt, führt die verfolgte Vorgehensweise zu einer vertikalen Unterteilung nach Subdomänen und zu einer zusätzlichen horizontalen Unterteilung nach Schichten. Für jene Aspekte, die domänenübergrei-

Kategorie	Beschreibung	Beispielhafte Inhalte
feature	Beinhaltet Komponenten für einen Use Case	book-flight-component
api	Exportiert Building Blocks aus der aktuellen Subdomäne für andere	flight (aus Domain-Schicht)
ui	Beinhaltet sogenannte „dumme Komponenten" (Dumb Components), die Use-Case-agnostisch sind und somit wiederverwendet werden können	datetime-component address-component adress-pipe
domain	Beinhaltet jene Teile des Domänenmodells, die clientseitig zum Einsatz kommen	flight passenger
util	Beinhalten allgemeine Hilfsfunktionen	formatDate

Tabelle 1: Kategorisierung von Schichten und Bibliotheken

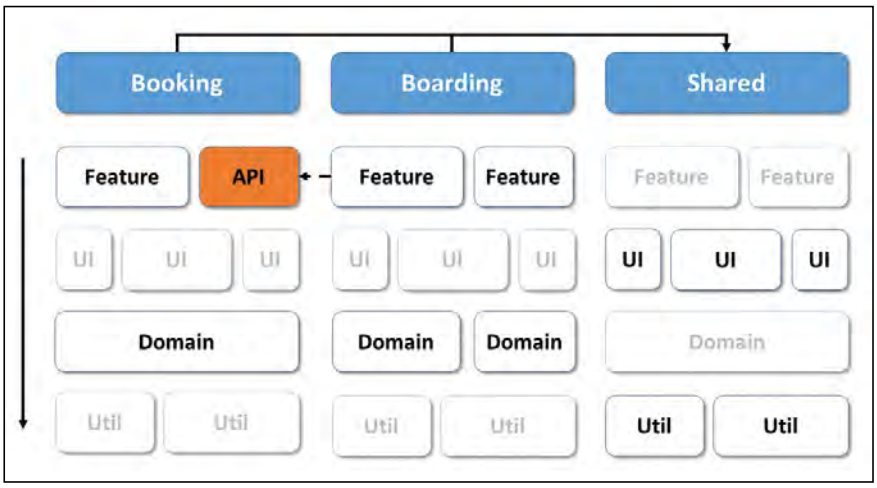

Abb. 1: Domänen und Layer

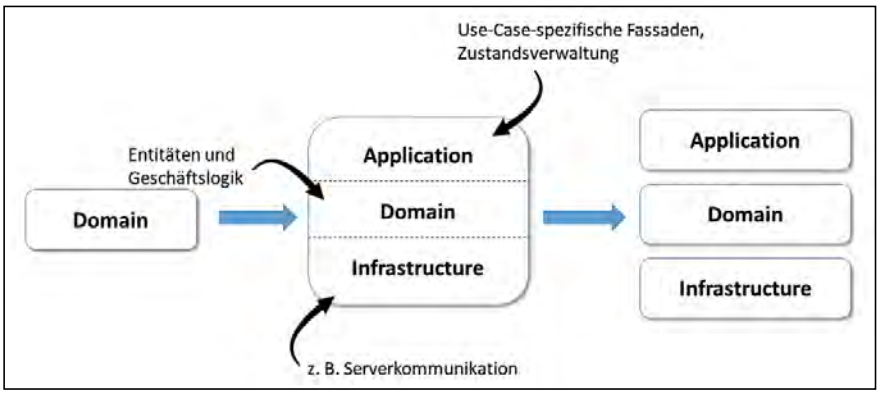

Abb. 2: Den Domain-Layer isolieren

fend zu nutzen sind, kommt ein zusätzlicher vertikaler Abschnitt mit der Bezeichnung *shared* zum Einsatz. Dessen fachliche Teile entsprechen dem von DDD vorgeschlagenen Shared Kernel. Zusätzlich beherbergt er technische Bibliotheken, z. B. für Authentifizierung oder Logging.

Jede Schicht erhält nun eine oder mehrere Bibliotheken. Zugriffsregeln zwischen diesen Bibliotheken führen zu einer losen Kopplung und somit zu einer gesteigerten Wartbarkeit. Typischerweise legt man fest, dass jede Schicht nur mit darunterliegenden Schichten kommunizieren darf, aber auch, dass domänenübergreifende Zugriffe lediglich über den Shared-Bereich erlaubt sind. Um zu verhindern, dass zu viel im Shared-Bereich landet, nutzt der hier vorgestellte Ansatz auch APIs, die Building Blocks für andere Domänen veröffentlichen. Das entspricht der Idee von Open Services in DDD.

In Anlehnung an [3] unterscheidet der hier vorgeschlagene Ansatz zwischen fünf Kategorien von Schichten bzw. Bibliotheken (Tabelle 1).

Diese vollständige Architekturmatrix wirkt ein wenig erdrückend, aber wie so oft wird auch hier nichts so heiß gegessen, wie es gekocht wird. Wie die ausgegrauten Blöcke in **Abbildung 1** andeuten, befinden sich die meisten Util-Bibliotheken nur im Shared-Bereich, zumal Aspekte wie Authentifizierung oder Logging systemübergreifend zum Einsatz kommen sollen. Dasselbe gilt auch für allgemeine UI-Bibliotheken, die ein systemweites Look and Feel sicherstellen.

Die Use-Case-spezifischen Featurebibliotheken und die domänenspezifischen Domain-Bibliotheken befinden sich hingegen in der Regel nicht im Shared-Bereich. Das wäre zwar im Sinne eines Shared Kernels konform zu Ideen von DDD, da es jedoch zu geteilten Verantwortungsbereichen, mehr Abstimmungsaufwand und Breaking Changes führen kann, sollte damit sparsam umgegangen werden.

Die Domäne isolieren

Um die Domänenlogik zu isolieren, werden ihr Fassaden [4] vorangestellt. Diese bereiten die Domänenlogik für jeweils einen Use Case auf und kümmern sich auch um die Verwaltung von Zuständen (**Abb. 2**).

Während Fassaden gerade im Angular-Umfeld sehr beliebt sind, korreliert diese Idee auch wunderbar mit DDD, wo von Application Services die Rede ist. Auch Infrastrukturangelegenheiten werden von der eigentlichen Domänenlogik getrennt. Bei SPAs handelt es sich hierbei meist um Serverzugriffe. Somit ergeben sich drei weitere Schichten: Die Application-Schicht mit Fassaden, die eigentliche Domänenschicht und die Infrastrukturschicht.

Diese Schichten können nun ebenfalls in eigene Bibliotheken verpackt werden. Zur Vereinfachung kann man auch dazu übergehen, sie in einer einzigen Bibliothek, die entsprechend untergliedert wird, zu verstauen. Vor allem vor dem Hintergrund, dass die Schichten meist gemeinsam genutzt werden und nur für Unit-Tests ausgetauscht werden müssen, kann diese Entscheidung sinnvoll sein.

Umsetzung mit einem Monorepo

Nachdem die Bestandteile unserer Architektur festgelegt wurden, stellt sich die Frage, wie sie sich in der Welt von Angular umsetzen lassen. Ein sehr üblicher und auch von Google selbst beschrittener Weg ist der Einsatz von Monorepos. Dabei handelt es sich um ein Code-Repository, das sämtliche Bibliotheken eines Softwaresystems beinhaltet. Monorepos vereinfachen unter anderem die Nutzung von geteiltem Code wie dem zuvor diskutierten Shared-Bereich, da dieser nun nicht mehr versioniert und verteilt werden muss.

Abb. 3: Abbildung der Domänen und Layer im Monorepo

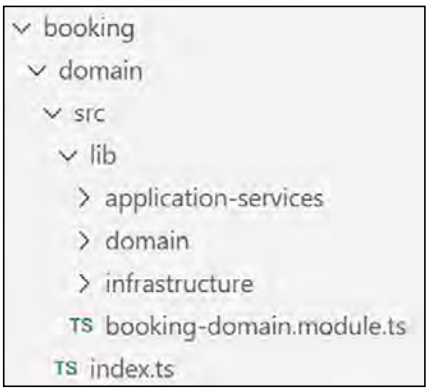

Abb. 4: Isolation des Domänenmodells

Stattdessen befinden sich immer die aktuellsten stabilen Versionen jeder Library im Master-Branch.

Während sich mittlerweile ein mit dem Angular CLI erstelltes Projekt als Monorepo nutzen lässt, bietet das beliebte Werkzeug Nx [5] noch einige zusätzliches Möglichkeiten, die gerade bei großen Unternehmenslösungen wertvoll sind. Dazu gehört die zuvor diskutierte Möglichkeit, Zugriffsbeschränkungen zwischen Bibliotheken einzuführen. Das verhindert, dass jede Bibliothek auf jede andere zugreift und sich somit ein stark gekoppeltes Gesamtsystem ergibt. Daneben kann Nx durch einen Blick in die Git History auch erkennen, welche Bibliotheken von den letzten Codeänderungen betroffen sind. Diese Information nutzt es, um nur diese Bibliotheken neu zu kompilieren bzw. nur deren Tests laufen zu lassen. Offensichtlich spart das eine Menge Zeit bei großen Systemen, die als Ganzes in einem Repository hinterlegt sind. Um eine Bibliothek im Monorepo zu erstellen, reicht eine Anweisung:

```
ng generate library domain
                    --directory boarding
```

Der von Nx nachgerüstete Schalter *directory* gibt ein optionales Unterverzeichnis an, in dem die Bibliotheken abzulegen sind. Auf diese Weise lassen sie sich nach den Domänen des Systems gruppieren (**Abb. 3**).

Die Namen der Bibliotheken spiegeln die Schichten wider. Weist eine Schicht mehrere Bibliotheken auf, bietet es sich an, diese Namen als Präfix zu nutzen. Somit ergeben sich Bezeichnungen wie *feature-search* oder *feature-edit*. Zur Isolation des eigentlichen Domänenmodells unterteilt das hier betrachtete Beispiel die Domain-Bibliothek in die drei genannten weiteren Schichten (**Abb. 4**).

Entitäten

Um die Strukturierung der Domain-Schicht drehen sich die Ideen aus dem DDD-Teilbereich Tactical Design. Im Zentrum stehen die Entitäten, die in den meisten Fällen der Server liefert. Sie spiegeln die Beschaffenheit der Domäne wider und basieren auf dem dort üblichen Vokabular. Listing 1 zeigt neben einem *enum* zwei Entitäten, die den in objektorientierten Sprachen wie Java oder C# üblichen Gepflogenheiten entsprechen.

Listing 1

```
public enum BoardingStatus {
  WAIT_FOR_BOARDING,
  BOARDED,
  NO_SHOW
}

public class BoardingList {
  private int id;
  private int flightId;
  private List<BoardingListEntry> entries;
  private boolean completed;

  // Getters and Setters

  public void setStatus(int passengerId, BoardingStatus status) {
  // Complex logic to update status
  }
}

public class BoardingListEntry {
  private int id;
  private BoardingStatus status;

  // Getters and Setters

}
```

Wie in der Objektorientierung üblich, nutzen diese Entitäten Information Hiding, um sicherzustellen, dass ihr Zustand konsistent bleibt. Sie implementieren das mit privaten Feldern und öffentlichen Methoden, die darauf operieren. Außerdem kapseln diese Entitäten nicht nur Daten, sondern auch darauf optierende Logiken. Diesen Umstand deutet zumindest die Methode *setStatus* an. Lediglich für Fälle, wo sich Logiken nicht sinnvoll in einer Entität unterbringen lassen, definiert DDD sogenannte Domain Services.

Entitäten, die lediglich Datenstrukturen repräsentieren, sind in DDD hingegen verpönt. Abwertend bezeichnet die Community sie als blutarm (Anemic Domain Model). Von einem objektorientierten Standpunkt gesehen, mag das auch korrekt sein. Allerdings steht bei Sprachen wie JavaScript und TypeScript die Objektorientierung weniger im Vordergrund. Es handelt sich vielmehr um Multiparadigmensprachen, in denen die funktionale Programmierung (FP) einen besonders starken Stellenwert hat. Da jedoch die funktionale Programmierung die Trennung von Datenstrukturen und Logik propagiert, sind Domain-Modelle hier zwangsweise blutarm. Werke, die sich mit funktionalem DDD beschäftigen, unterstreichen das (vgl. [6], [7]), und selbst „Domain-Driven Design Distilled“ von Vauhgn Vernon [8], das als eines der Standardwerke für DDD gilt und primär auf OOP setzt, gesteht ein, dass diese Regeländerung in der FP notwendig ist. Das vorhin betrachtete Entitätenmodell würde demnach in TypeScript in einen Datenteil (Listing 2) und einen Logikteil (Listing 3) getrennt.

Bei diesen Entitäten fällt auch auf, dass sie öffentliche Eigenschaften nutzen. Auch das ist in der FP durchaus üblich, während der exzessive Einsatz von Gettern und Settern, die lediglich an private Eigenschaften delegieren, oftmals belächelt wird – andere Paradigmen, andere Sitten.

Nun lässt sich natürlich vortrefflich darüber streiten, was der bessere Stil ist. Viel interessanter ist jedoch die Frage, wie die funktionale Welt inkonsistente Zustände vermeidet. Die Antwort ist verblüffend einfach: Datenstrukturen sind bevorzugt unveränderbar (immutable). Das Schlüsselwort *readonly* in Listing 2 unterstreicht das. Ein Programmteil, der solche Objekte ändern möchte, muss das entprechende Objekt also klonen, und haben andere Programmteile erst mal ein Objekt für ihre Zwecke validiert, können sie davon ausgehen, dass es valide bleibt.

Aggregate

Um den Überblick über die Bestandteile eines Domänenmodells zu behalten, fasst Tactical DDD Entitäten zu Aggregaten zusammen. In Listing 2 bilden zum Beispiel *BoardingList* und *BoardingListEntry* ein Aggregat. Der Zustand sämtlicher Bestandteile eines Aggregats muss als Gesamtes konsistent sein. Beispielsweise könnte man im betrachteten Fall festlegen, dass *completed* in der *BoardingList* nur dann auf *true* gesetzt werden darf, wenn kein *BoardingListEntry* den Status *WAIT_FOR_BOARDING* hat. Außerdem dürfen verschiede-

Listing 2

```
export type BoardingStatus = 'WAIT_FOR_BOARDING'
                           | 'BOARDED' | 'NO_SHOW';

export interface BoardingList {
  readonly id: number;
  readonly flightId: number;
  readonly entries: BoardingListEntry[];
  readonly completed: boolean;
}

export interface BoardingListEntry {
  readonly passengerId: number;
  readonly status: BoardingStatus;
}
```

Listing 3

```
export function updateBoardingStatus(
                boardingList: BoardingList,
                passengerId: number,
                status: BoardingStatus): Promise<BoardingList> {

  // Complex logic to update status

}
```

Listing 4

```
@Injectable({ providedIn: 'root' })
export class FlightFacade {

  private flightsSubject = new BehaviorSubject<Flight[]>([]);
  public flights$ = this.flightsSubject.asObservable();

  constructor(private flightService: FlightService) {
  }

  search(from: string, to: string, urgent: boolean): void {
    this.flightService.find(from, to, urgent).subscribe(
      flights => {
        this.flightsSubject.next(flights)
      },
      err => {
        console.error('err', err);
      }
    );
  }
}
```

ne Aggregate nicht über Objektreferenzen auf einander verweisen. Stattdessen können sie IDs verwenden. Das soll eine unnötige Kopplung zwischen Aggregaten verhindern. Große Domänen lassen sich somit auf kleinere Gruppen von Aggregaten herunterbrechen.

Vernon schlägt in [8] vor, Aggregate so klein wie möglich zu gestalten. Zunächst einmal solle man jede einzelne Entität als Aggregat ansehen und dann Aggregate, die ohne Zeitverzögerung gemeinsam geändert werden müssen, miteinander verschmelzen.

Fassaden aka Application Services

Die Aufgabe der Application Services ist es, Details des Domänenmodells für bestimmte Use Cases aufzubereiten. Diese Idee erfreut sich in der Welt von Angular auch unabhängig von DDD seit einiger Zeit großer Beliebtheit. Man spricht hierbei auch von Fassaden (Facades) [4].

Das Beispiel in Listing 4 veranschaulicht solch eine Fassade in Form eines Angular Service für das Suchen nach Flügen.

Während es mittlerweile zum guten Ton gehört, serverseitige Services zustandslos zu gestalten, trifft das nicht für Services in SPAs zu. Eine SPA hat nun einmal einen Zustand, und genau das erhöht auch die Benutzerfreundlichkeit: Man möchte eben nicht alle Informationen immer und immer wieder vom Server abrufen. Diesen Umstand spiegelt auch die betrachtete Fassade wider, indem sie die abgerufenen Flüge für eine spätere Verwendung innerhalb des Use Case vorhält. Dazu nutzt sie Observables. Das bedeutet, dass die Fassade Angular, aber auch andere Systembestandteile informieren kann, wenn sich Zustände ändern.

Events und State Management

Die Implementierung im letzten Abschnitt funktioniert so lange gut, solange nur eine oder wenige Komponenten mit dem Zustand interagieren. Besonders schlimm wird es, wenn viele verschiedene Komponenten den Zustand verändern und somit Inkonsistenzen entstehen. Es können aber auch zyklische Abhängigkeiten entstehen. Deswegen müssen sich Entwicklungsteams bei SPAs früher oder später die Frage stellen, wie damit umzugehen ist.

Auf der anderen Seite ist der Einsatz von Observables eine gute Idee. Ein damit geschaffenes reaktives System verbessert die Datenbindungsperformance und sorgt für Entkopplung, da sich Sender und Empfänger theoretisch nicht direkt kennen müssen. Das passt auch zu DDD, wo der Einsatz von Domain Events mittlerweile zur Tagesordnung gehört: Passiert in einem Anwendungsteil etwas Interessantes, versendet er ein Domain Event und andere Anwendungsteile können darauf reagieren. Im betrachteten Fall könnte solch ein Domain Event anzeigen, dass sich ein Passagier oder alle Passagiere eines Flugs nun im Zustand *BOARDED* befinden.

Diese beiden Fliegen, State Management und Eventing, lassen sich mit dem populären Redux-Muster auf einmal schlagen. Der vorliegende Text soll zwar keine Einführung in Redux werden, nichtsdestotrotz beleuchten die nächsten Beispiele, wie sich eine darauf aufbauende Architektur anfühlt. Dazu kommt NgRx Store, der De-facto-Standard für Redux im Angular-Umfeld, zum Einsatz.

Beim Einsatz von NgRx Store wird zunächst der Anwendungszustand für jedes Feature der Anwendung modelliert. Das passiert typischerweise durch die Bereitstellung von Interfaces. Für Events, aber auch für Kommandos, die zu einer Zustandsänderung führen, sind sogenannte Actions zu modellieren. Im Wesentlichen sind das Objekte mit einem Typ und weiteren Properties. Der Typ beschreibt die Art des Events bzw. Kommandos (Listing 5).

Der Typ der hier betrachteten *Action* lässt darauf schließen, dass die *Action* darüber informiert, dass Flüge geladen wurden. Die Properties beinhalten diese Flüge. Die einzelnen Programmteile – man spricht bei NgRx auch von Features – erhalten *Reducer*, die auf die *Actions* reagieren und den globalen Anwendungszustand aktualisieren. Der *Reducer* in Listing 6 reagiert beispielsweise auf die zuvor betrachtete *loadFlightsSuccess* Action. Er nimmt den aktuellen Zustand sowie die *Action* mit ihren Properties entgegen und verändert daraufhin den Zustand. Genaugenommen klont er den Zustand, da NgRx Store die vorhin diskutierten Immutables erzwingt. Den so entstehenden neuen Zustand liefert der *Reducer* zurück.

Um nun eine *Action* auszulösen, besorgt sich ein Programmteil via Dependency Injection den sogenannten Store, der sich um die Verwaltung der Zustände kümmert. Seine *dispatch*-Methode versendet die gewünschte Aktion an alle *Reducer*, ohne diese direkt zu kennen:

```
this.store.dispatch(loadFlightsSuccess({flights}));
```

Ähnlich einfach gestaltet sich das Abrufen von Daten aus dem Store:

Listing 5

```
export const loadFlightsSuccess = createAction(
  '[Booking] Load Flights Success',
  props<{ flights: Flight[] }>()
);
```

Listing 6

```
const bookingReducer = createReducer(
  initialState,

  on(loadFlightsSuccess, (state, action) =>
    [...]
  )
);
```

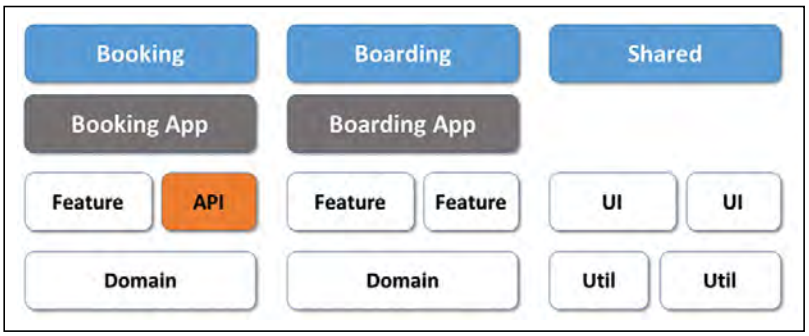

Abb. 5: DDD auf dem Client als Grundlage für Micro Frontends

```
public flights$ = this.store.pipe(select(state => state.booking.flights));
```

Hierbei erhält der Aufrufer nicht den Wert selbst, sondern ein Observable, das diesen Wert repräsentiert. Das bedeutet, dass sich der Aufrufer über jede Änderung informieren lassen kann.

Der Einsatz von Redux lässt sich für den Rest der Anwendung hinter der im letzten Abschnitt besprochenen Fassade verbergen. Der Konsument der Fassade merkt also von einem eventuellen Einsatz von Redux nichts, und somit kann ein Team Redux auch erst im Nachhinein einführen, wenn die Komplexität bestimmter Anwendungsfälle es rechtfertigt.

Domain-driven Design und Micro Frontends?

Die Ideen von Domain-driven Design lassen sich bekanntlich für das Erstellen von Microservices-Architekturen nutzen. Genauso lässt sich DDD auf dem Client auch als Grundlage für Micro Frontends verwenden. Ob sich nämlich ein Deployment-Monolith, Micro Frontends oder irgendetwas dazwischen ergibt, hängt von der Verwendung des Monorepos ab. Richtet das Team pro Domäne eine eigene Anwendung im Monorepo ein, geht es einen großen Schritt in Richtung Micro Frontends (**Abb. 5**).

Die Zugriffsbeschränkungen sichern eine lose Kopplung und ermöglichen sogar eine spätere Aufteilung auf mehrere Repositories, falls das für die Entkopplung als vorteilhaft angesehen wird. Dann kann von Micro Frontends im klassischen Sinn einer Microservices-Architektur gesprochen werden. Das Team muss sich dann jedoch, wie bei Microservices üblich, um das Versionieren und Verteilen der Bibliotheken aus dem Shared-Bereich kümmern.

Fazit

Moderne Single Page Applications sind häufig mehr als Empfänger von Data-Transfer-Objekten. Sie beinhalten einiges an Logik, und das erhöht die Komplexität. Ideen von DDD helfen, die dadurch entstehende Komplexität zu beherrschen. Aufgrund des objektfunktionalen Charakters von TypeScript und den dort vorherrschenden Gepflogenheiten sind ein paar Regeländerungen notwendig.

Der Einsatz von Monorepos mit mehreren Bibliotheken, die nach Domänen gruppiert werden, hilft beim Aufbau der grundsätzlichen Struktur, und Zugriffsbeschränkungen zwischen Bibliotheken verhindern eine Kopplung zwischen Domänen. Fassaden bereiten das Domänenmodell für einzelne Use Cases auf und kümmern sich um das Vorhalten von Zuständen. Bei Bedarf lässt sich hierzu Redux hinter der Fassade nutzen, ohne dass der Rest der Anwendung etwas davon bemerkt. In diesem Fall erhält man auch einen Eventing-Mechanismus frei Haus. Ganz nebenbei schafft ein Team durch den Einsatz von DDD am Client auch die Voraussetzung für Micro Frontends.

Manfred Steyer (www.softwarearchitekt.at) ist Trainer und Berater mit Fokus auf Angular sowie Google Developer Expert und trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. In seinem aktuellen Buch zu Angular behandelt er die vielen Seiten des populären JavaScript Frameworks aus der Feder von Google.

Links & Literatur

[1] Steyer, Manfred: „Große Business-Apps mit Angular meistern", in: Windows Developer 5.2019

[2] https://github.com/manfredsteyer/angular-ddd

[3] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book

[4] https://medium.com/@thomasburlesonIA/ngrx-facades-better-state-management-82a04b9a1e39

[5] https://nx.dev/

[6] Ghosh, Debasish: „Functional and Reactive Domain Modeling", Manning, 2016

[7] Walaschin, Scott: „Domain Modeling Made Functional": https://pragprog.com/book/swdddf/domain-modeling-made-functional

[8] Vaughn, Vernon: „Domain-Driven Design Distilled", Addison-Wesley Professional, 2016

Was in zehn Jahren DevOps-Bewegung alles passiert ist

Die erste Etappe einer langen Reise …

Vor rund zehn Jahren kam der Begriff DevOps auf. Seitdem hat sich technisch einiges verändert. Außerdem gibt es ganz neue Kollaborationsansätze in der IT.

von Konstantin Diener

Im Jahr 2019 ist die DevOps-Bewegung zehn Jahre alt geworden. Die öffentliche Initialzündung war ein Vortrag von John Allspaw und Paul Hammond von Flickr auf der O'Reilly Velocity Conference 2009 [1]. In diesem Vortrag beschreiben die beiden die seinerzeit übliche Arbeitsteilung zwischen Softwareentwicklung (Dev) und -betrieb (Ops) folgendermaßen: „Dev's job is to add new features. Ops's job is to keep the site stable and fast." Sie argumentieren, dass beide Parteien das Business unterstützen sollten und diese Unterstützung nur durch kontinuierlichen Change möglich sei. Damit der kontinuierliche Change aber nicht zu Fehlern und Ausfällen führt, brauche es neue Tools und kulturelle Ansätze („Lowering the risk of change through tools and culture").

Wie sehr sich die Welt von einem Tag auf den anderen ändern kann, ist im Jahr zuvor am Beispiel der Bank Lehman Brothers und der weltweiten Finanzkrise nur allzu deutlich geworden. Die Wellen dieser Krise reichen weit bis ins Jahr 2009 hinein. In diesem Jahr meldeten unter anderem der Autohersteller GM und Arcandor AG (ehemals KarstadtQuelle AG) Insolvenz an. Einzig die politische Lage wirkt aus heutiger Sicht stabiler: Mit Barack Obama wurde ein Hoffnungsträger als US-Präsident vereidigt und aus der Bundestagswahl gingen CDU und FDP mit einer eindeutigen Regierungsmehrheit hervor. Die schnelle Bildung von Zweiparteienkoalitionen ist heute seltener geworden.

Die IT-Landschaft dieser Tage war auch noch sehr viel klassischer als heute: Das iPhone ist erst zwei Jahre alt und beginnt gerade erst, die mobile Internetnutzung zu verändern. Das Hosting von Anwendungen findet in der Regel ganz klassisch in Rechenzentren statt. AWS ist erst ca. drei Jahre alt und Google startet mit der Google App Engine for Java [2]. Beide sind zu diesem Zeitpunkt noch keine echten Alternativen zum On-Premise Hosting. Das bedeutet auch, dass IT-Infrastruktur zu dieser Zeit noch ein erhebliches Investment darstellte. Meist wurden schon zu Beginn einer Softwareentwicklung die Hardwaresysteme für den späteren Betrieb dimensioniert und bestellt. Die Kosten für diese Investition waren mindestens fünfstellige Eurobeträge.

Die Java-Community diskutierte im Jahr 2009 natürlich über die Übernahme von Sun Microsystems durch Oracle und darüber, welche Auswirkungen diese Entwicklung auf die Java-Plattform haben würde. Technisch beschäftigte sich die deutschsprachige Community

Abb. 1: Cover der Java-Magazin-Ausgabe 1.2009

Abb. 2: Vorschläge aus der Präsentation von Allspaw und Hammond

im Java Magazin damit, ob REST oder SOAP für Web Services das richtige Paradigma wären (**Abb. 1**) oder ob man eher auf EJB 3.0 oder auf Spring 3.0 setzen sollte. OSGi war der Hoffnungsträger in Bezug auf feingranulare, modulare Architekturen, und im Enterprise-Bereich waren SOA und ESB die relevanten Themen.

Im Java Magazin und auf zahlreichen Konferenzen ließ sich beobachten, dass außerdem das Thema Agilität bzw. agile Softwareentwicklung mehr und mehr diskutiert wurde, aber noch viel erklärungsbedürftiger war als heute. In der Kolumne „Perspektivenwechsel“ im Java Magazin erhielten die Leser Tipps für die praktische Anwendung der noch relativ neuen Konzepte Scrum, Kanban, TDD oder XP. Die meisten agilen Ansätze blieben in der Realität des Jahres 2009 allerdings Insellösungen: Selbst wenn das Scrum-Team nach jedem Sprint ein Potentially Shippable Product Increment lieferte, schaffte es dieses Softwareartefakt allerhöchstens auf ein Entwicklungs- oder Testsystem. Die Auslieferung auf Integrations- oder gar Produktionssysteme fand viel seltener statt – oft nur einmal im Quartal.

Für diese Zeit war es revolutionär, dass die beiden Flickr-Mitarbeiter schon in ihrem Vortragstitel von zehn oder mehr Deployments am Tag (!) sprachen und eine Reihe von Tools und Eigenschaften ihrer Kultur vorstellten, um eine bessere Kooperation von Dev und Ops zu erreichen. Folgende Tooling-Aspekte zählten sie auf (**Abb. 2**):

1. *Automated Infrastructure:* Entgegen der zu dieser Zeit gängigen Praxis (manuelle Installation) plädierten die beiden für eine Provisionierung der Infrastruktur mit Werkzeugen wie Chef oder Puppet.
2. *Shared Version Control:* Die Skripte für die automatisierte Provisionierung der Infrastruktur sollten im selben Versionskontrollsystem wie der Anwendungscode liegen.
3. *One Step Build and deploy:* Ebenso wie die Provisionierung der Infrastruktur sollten auch Deployments in einem automatisierten und vor allem nachvollziehbaren Prozess stattfinden („Wer hat was zu welchem Zeitpunkt deployt“).

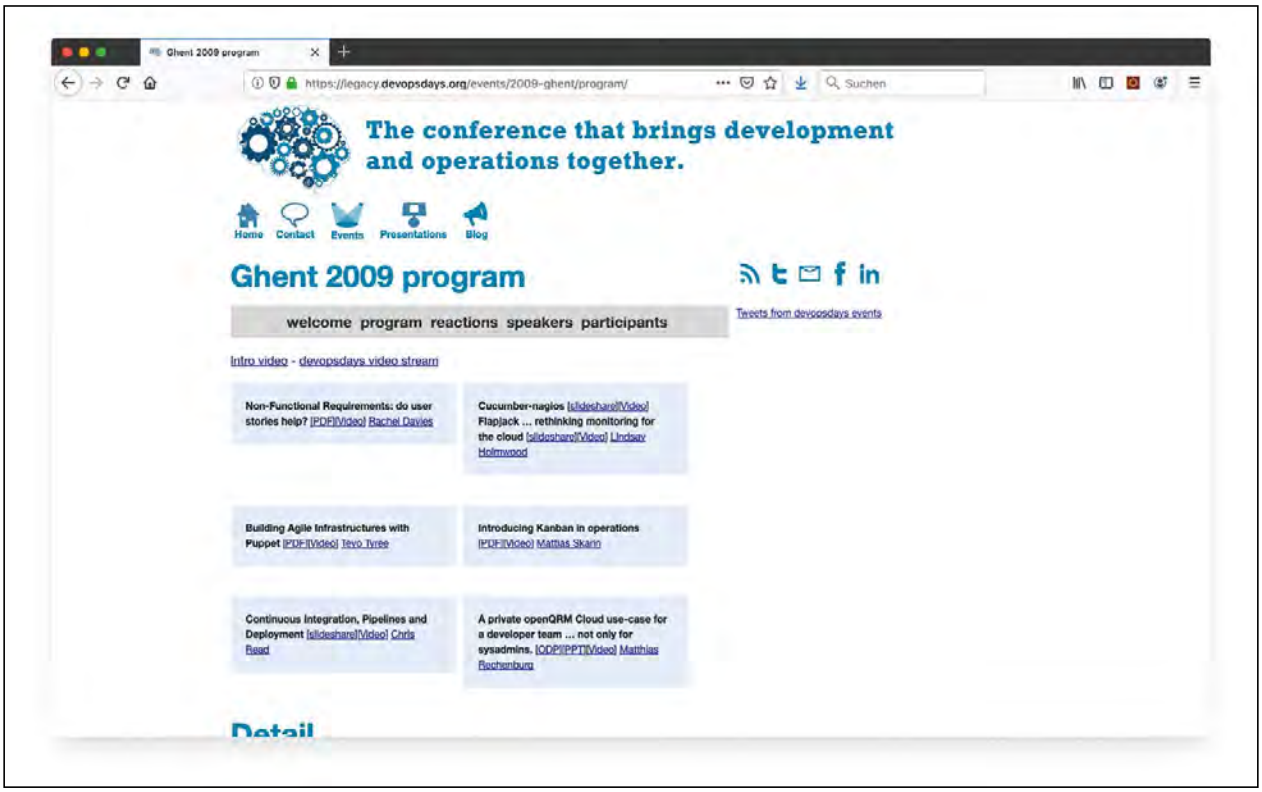

Abb. 3: Programm der ersten DevOps Days im Jahr 2009

Abb. 4: Bestandteile des CAMS-Modells

4. *Feature flags:* Die beiden argumentierten, dass man bei Webanwendungen andere Versionierungsansätze als bei Desktopsoftware brauchen würde, und plädierten für den Einsatz von Trunk Based Development und Feature-Flags [3].
5. *Shared Metrics:* Für diese Zeit revolutionär war auch der Ansatz, gemeinsame Systemmonitoringwerkzeuge für Entwicklung und Betrieb bereitzustellen.
6. *IRC and IM robots* war ein Vorläufer dessen, was später als ChatOps bekannt wurde.

Der wichtigste Ratschlag in kultureller Hinsicht war der des gegenseitigen Respekts. Nach wie vor bedeutet das insbesondere, mit Vorurteilen aufzuräumen und die jeweils andere Seite als Experten auf ihrem Gebiet anzuerkennen. Diese Expertise soll aber nicht zu geheimem Herrschaftswissen führen, vielmehr sollen alle Aspekte der Entwicklung und des Betriebs transparent werden (Einbindung von Ops in Featurediskussionen bzw. Einbindung von Dev bei Infrastrukturänderungen). Nur so entsteht Vertrauen, das unabdingbar für eine sinnvolle Kollaboration ist. Darüber hinaus sollten alle Beteiligten immer unterstellen, dass jeder sein Bestes für das Business gibt. Sie sind sich sicher, dass trotzdem Fehler passieren werden und es eine gesunde Fehlerkultur ohne Blaming geben muss.

Der Begriff DevOps tauchte in dem Flickr-Vortrag noch nicht auf, tatsächlich wurde das Thema noch im Jahr 2008 eher unter dem Begriff „Agile Infrastructure“ oder „Agile Systems Administration“ diskutiert. Der Begriff DevOps entstand später im Jahr 2009 auf einer Konferenz in Gent. Inspiriert durch den Vortrag von Allspaw und Hammond sowie durch seine eigenen Erfahrungen rief Patrick Debois die DevOps Days ins Leben [4]. Auch dort ging es sowohl um Tooling bzw. Automatisierung als auch um kulturelle Aspekte der Kollaboration (**Abb. 3**).

DevOps ist mehr als Tooling ...

Mit steigender Popularität wurde der Begriff DevOps immer häufiger mit Tooling bzw. Automatisierung gleichgesetzt – vor allem in den Marketingbotschaften. Dabei ging es aber in der Regel nicht um Tooling-Prinzipien, wie Allspaw und Hammond sie beschrieben hatten, sondern eher um das Versprechen, man müsse nur Tool XYZ einsetzen, um erfolgreich zu sein. Wie so oft stellte sich auch hier schnell Ernüchterung ein. Eine bessere Kollaboration und damit häufigere, stabile Softwareauslieferungen lassen sich eben nicht einfach durch ein Tool erreichen.

Das Buch „The Phoenix Project“ [5], das mittlerweile als Standardwerk der DevOps-Bewegung gilt, beschrieb einige Jahre später sehr anschaulich, dass DevOps vor allem einen Kulturwandel bedeutete. Die Geschichte spielt in der fiktiven Firma Parts Unlimited. Und obwohl es um das strategische IT-Projekt des Unternehmens und Herausforderungen des IT-Betriebs geht, dreht sich sehr wenig um konkrete Tools oder Technologien. Ich selbst war beim ersten Lesen sogar etwas überrascht, weil ich mehr grundlegend Neues erwartet hatte. Stattdessen kannte ich viele Ansätze des sehr gut geschriebenen Buches bereits aus Lean, Kanban etc. Auch hier stellt sich wie bei Allspaw und Hammond wieder die Frage, ob DevOps nicht einfach Agilität über die Softwareentwicklung hinaus bedeutet. Genau wie in der agilen Softwareentwicklung ist es nicht mit der Einführung eines Tools getan. Es handelt sich vielmehr um eine Reise, die kontinuierliche Anstrengungen bedeutet und eigentlich nie zu Ende ist.

Der „2018 State of DevOps Report“ [6], [7] versucht, diese Reise zu beschreiben und in sechs verschiedene Stationen zu gliedern. Stufe 0 ist sozusagen der ständige Reisebegleiter und zugleich die Grundlage einer DevOps-Reise. Die Inhalte dieser Stufe orientieren sich stark am sogenannten CAMS-Modell. CAMS bedeutet Culture, Automation, Measurement und Sharing (**Abb. 4**). Das

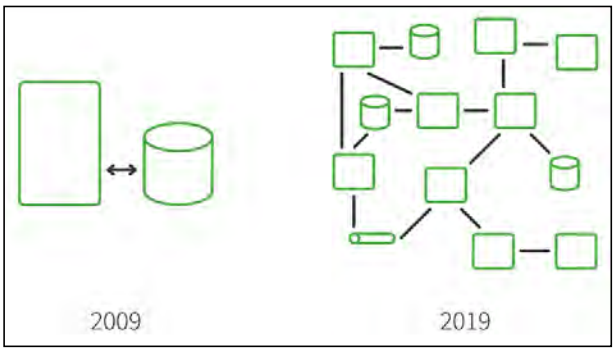

Abb. 5: Anzahl und Größe von Deployment-Einheiten 2009 und heute

Modell greift zahlreiche Aspekte wieder auf, die Allspaw und Hammond in ihrem Vortrag beschrieben haben. Automation bedeutet in diesem Zusammenhang die Automatisierung von Tests und konsequente CI/CD.

Auch wenn DevOps also in erster Linie als Kulturwandel zu verstehen ist, hat der technologische Fortschritt in den letzten Jahren diesen Wandel ohne Zweifel ermöglicht oder beschleunigt.

… aber Tools und Technologien sind Enabler für DevOps

Die kurze Rückblende am Anfang des Artikels enthielt auch das Thema OSGi. Diese Komponententechnologie versprach kleinere Komponenten, die sich leichter zur Laufzeit austauschen lassen. OSGi bedeutete damit eine Abkehr von der bisherigen Praxis, möglichst große Deployment-Artefakte zu bauen und auszuliefern. Im Jahr 2009 bestand eine Java-Anwendung in der Regel aus einer großen EAR-Datei, die in einem Application-Server deployt wurde. Neben diesen beiden gab es nur wenige, ebenfalls sehr große Anwendungskomponenten (z. B. Datenbank, Message Broker, ESB).

OSGi hat den Trend eingeleitet, wieder in vielen kleineren Deployment-Einheiten zu denken. Dieser Trend hat sich in den folgenden Jahren verstetigt, auch wenn OSGi heute nicht mehr die gleiche Rolle spielt. Eine moderne Anwendung besteht heute aus Dutzenden kleiner Einzelteile (**Abb. 5**). Bei diesen kleinen Einzelteilen handelt es sich oft um Microservices. Ein sehr kleiner Service kapselt eine oder wenige fachliche Problemstellungen (Bounded Context), enthält technisch aber alle notwendigen Bestandteile wie einen Webserver etc., um autonom einsatzfähig zu sein. Im Gegensatz zu OSGi-Modulen lassen sich diese Services viel einfacher unabhängig voneinander skalieren.

Ermöglicht wurde diese Entwicklung durch das Aufkommen von Containertechnologien – in erster Linie Docker. Container erlauben auf einfache Weise, alle notwendigen Softwarebestandteile (Applikationsartefakt, Webserver etc.) transportierbar in einen Container zu packen und ohne Installation auf einer beliebigen Basisplattform auszuführen. Das Java-Motto „Write once, run anywhere“ wird damit auf Betriebssystemebene angewendet. Für etliche Application-Server, Datenbanken etc. existieren fertige Images, die einfach um eigene Komponenten und Konfigurationen ergänzt werden können.

Der zweite Turbo für immer kleinteiliger werdende Anwendungsarchitekturen und vor allem deren Skalierung sind die Plattformen der führenden Public-Cloud-Anbieter. Vor rund zehn Jahren war die Bereitstellung von Rechenkapazität und Speicher ein langfristiges Investment und mit langen Vorlaufzeiten verbunden. Heute ist beides in der Cloud nur wenige Mausklicks entfernt und kann ebenso schnell wieder abgeschaltet werden. Interessanterweise gab es sogar schon einen Vortrag zu einem Cloudthema im Programm der DevOps Days in Gent.

Auch die Begriffe Rechenkapazität und Speicher passen eher zur IT-Infrastruktur-Welt des Jahres 2009 als in die heutige Zeit. Damals war es gängige Praxis, Root-Server zu mieten und dann einen Datenbankcluster, einen Suchindex, eine NoSQL-Datenbank oder einen Application-Server auf diesen Maschinen von Hand zu installieren und insbesondere dann später auch zu betreiben und zu patchen. Dieser Ansatz wird zunehmend durch Managed Services ersetzt. Statt den Datenbankserver selbst zu betreiben, wird er als Service gemietet. Der Managed-Service-Provider kann diesen Service in der Regel viel günstiger und sicherer anbieten als seine Kunden. Zum einen bedingt durch Skaleneffekte, zum anderen, weil die meisten Managed Services wieder auf Basis-Services der großen Public-Cloud-Anbieter wie AWS basieren. Mit Managed Services setzt sich ein Trend fort, der mit der Nutzung von Programmbibliotheken seinen Anfang genommen hat. Vor zehn Jahren haben die Softwareentwickler nach passenden Java-Bibliotheken gesucht, um nicht alles selbst entwickeln zu müssen. Heute suchen sie zusätzlich noch nach passenden Managed Services für Infrastrukturkomponenten. Solche Services existieren für alle denkbaren Anwendungsfälle (z. B. Messaging, Sprachausgabe, Caching, Authentication, relationale Datenbanksysteme, NoSQL-Datenbanken, Suchindizes, File Storage etc.). Managed-Services-Anbieter nehmen dem Kunden dabei immer mehr Arbeit ab. Manche Services, wie z. B. Heroku, gehen so weit, dass die Entwickler nur noch ihr Artefakt deployen müssen und nach wenigen Sekunden die laufende Webanwendung im Browser aufrufen können – auf Wunsch auch in einer geclusterten Version.

Der aktuelle Höhepunkt dieser Entwicklung ist Function as a Service (FaaS) – der bekannteste Vertreter aus dem Serverless-Umfeld. Im FaaS-Modell deployt der Kunde ein Stück Code, das dann zu bestimmten Ereignissen aufgerufen wird (Veränderungen in einem Clouddateisystem, einkommende Requests, asynchrone Nachrichten aus Messagingsystemen oder zeitgesteuert). Dem Kunden wir dabei nur die tatsächliche Ausführungszeit in Rechnung gestellt. Die Kosten für die Infrastruktur lassen sich so nahezu linear mit der Last auf der Anwendung skalieren.

Mit der Anzahl der Einzelteile einer Anwendung steigt der Bedarf der Orchestrierung dieser Teile. Es

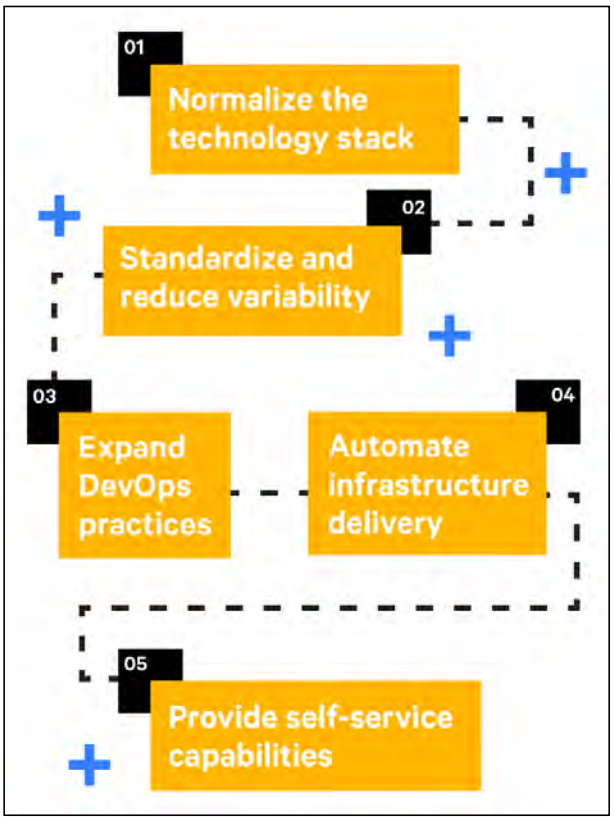

Abb. 6: Die Schritte einer DevOps-Reise (Quelle: State of DevOps Report 2018)

braucht ein Betriebssystem, auf dem z. B. Docker-Container betrieben werden können. Das bekannteste Betriebssystem für Container ist heute Kubernetes. Kubernetes bildet eine Schicht zwischen den Containern und den Maschinen, auf denen diese Container ausgeführt werden. Es kann Container anhand ihrer Anforderungen an Speicher, Disk oder Prozessor auf die passenden Maschinen verteilen, abgestürzte Container neu starten und unterstützt beim Update von Containern. Aufsetzend auf einem Betriebssystem wie diesem gibt es außerdem heute Service Meshes wie Istio, die bei der Verwaltung einer großen Anzahl von Microservices helfen.

Tooling – eher Praktiken als konkrete Tools

Der wichtigste Tooling-Grundsatz aus der Präsentation von Allspaw und Hammond war die Infrastrukturautomatisierung (Infrastructure as Code). Im Jahr 2009 war damit eher noch gemeint, mit Tools wie Chef oder Puppet auf einem Root-Server Komponenten wie einen Application-Server o. Ä. automatisiert zu installieren und mit den passenden Anwendungsartefakten zu bestücken. Dabei handelte es sich bereits um einen großen Fortschritt, wenn man sich überlegt, dass diese Installationen vorher in der Regel manuell durchgeführt wurden. Eine Automatisierung erhöht die Geschwindigkeit, reduziert Fehler und erlaubt z. B. Entwicklungsteams, produktionsgleiche Testumgebungen zu haben. Außerdem werden Änderungen an den Infrastrukturkomponenten so für alle nachvollziehbar. Ist doch mal ein Fehler passiert, lassen sich die Änderungen zurückrollen oder die komplette Umgebung lässt sich von Grund auf neu aufsetzen. Letzteres war 2009 weitgehend undenkbar. Die meisten Serverumgebungen waren irgendwann einmal aufgesetzt worden und wurden dann sukzessive erweitert. Gab es Probleme mit der Umgebung, wurde sie von den Ops-Mitarbeitern regelrecht gesund gepflegt. Man sagt heute dazu, dass die Umgebungen wie Haustiere behandelt und gehätschelt werden [8]. Mit dem Aufkommen von Virtualisierungs- und vor allem von Public-Cloud-Lösungen ist es sehr viel einfacher geworden, fehlerhafte Infrastrukturkomponenten zu stoppen und durch neue zu ersetzen. Mit Infrastructure as Code rückt dabei die programmatische Beschreibung der Infrastruktur in den Vordergrund, weil in der Cloud beliebig viele Kopien davon erzeugt werden können. In diesem Vorgehen werden die Infrastrukturkomponenten als Nutzvieh bezeichnet. Die Cloud schützt aber nicht vor Haustierinfrastruktur. Auch hier gibt es immer wieder Komponenten, die schnell manuell in der Weboberfläche erstellt wurden.

Durch den gestiegenen Einsatz von Managed Services hat sich der Fokus von Infrastructure as Code in den letzten Jahren verschoben. Chef und Puppet sind noch sehr programmatisch bzw. ablauforientiert – nicht ohne Grund heißen die Artefakte in Chef Rezepte. Die Infrastructure-as-Code-Artefakte für Managed Services

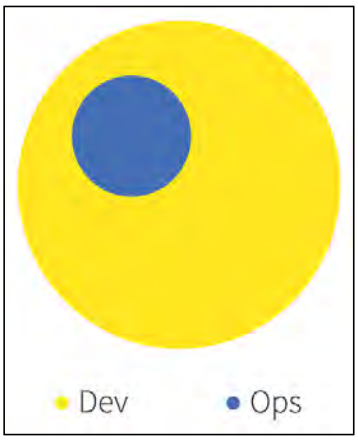

Abb. 7: Ein-Team-Modell

sind hingegen eher deskriptiv und beschreiben den gewünschten Zielzustand.

Es gibt immer wieder Firmen, in denen DevOps mit Infrastrukturautomatisierung gleichgesetzt wird. Entsprechende Initiativen beschränken sich auf diesen Aspekt oder beginnen zumindest damit, die Bereitstellung von Infrastruktur zu automatisieren. Laut „State of DevOps Report 2018“ sind diese Initiativen in der Regel zum Scheitern verurteilt. So ist es z. B. wenig zielführend, in Infrastrukturautomatisierung zu investieren, wenn es im Unternehmen eine Vielzahl unterschiedlicher Technologiestacks gibt. Im bereits angesprochenen sechsstufigen Modell der DevOps-Reise aus demselben

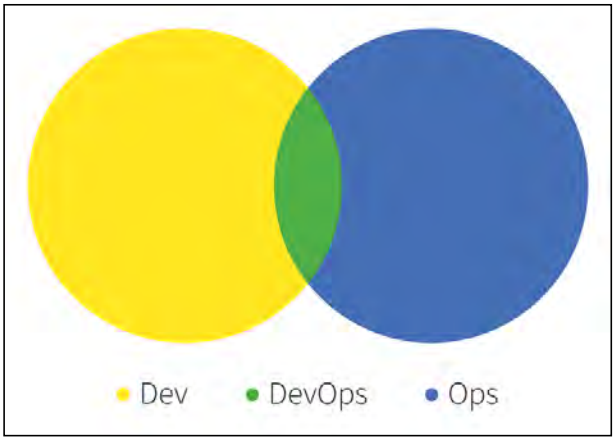

Abb. 8: Kollaborationsmodell

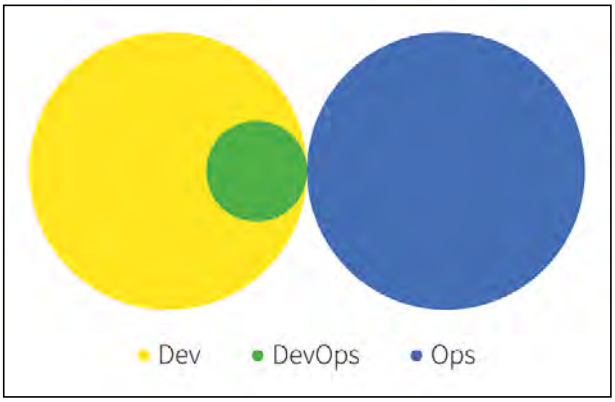

Abb. 9: Infrastructure-as-a-Service-Modell

Report ist Infrastrukturautomatisierung demnach auch erst der vorletzte Schritt [6], [7]. Vorher ist es für das Unternehmen wichtig, seinen Technologiestack auf eine möglichst kleine Anzahl von Standardkomponenten zu reduzieren (**Abb. 6**).

Überraschenderweise sind einige der Praktiken von Allspaw und Hammond auch rund zehn Jahre später nicht flächendeckend umgesetzt. So gibt es auf der einen Seite noch Firmen, die keine sinnvolle automatisierte Testbasis haben und von echter Continuous Integration bzw. Continuous Delivery [3] noch weit entfernt sind. Andere Firmen haben das agile Mantra hinter CI, „Wenn etwas schmerzhaft und aufwendig ist, tue es früh und oft“ verinnerlicht und holen das Thema Security mit DevSecOps aus seinem Stiefkindstatus, wie es Jahre zuvor mit dem Thema automatisierte Tests passiert ist. Auch ein gemeinsames Monitoring gibt es in etlichen Firmen noch nicht.

DevOps – eine Kultur der Kollaboration und Transparenz

Dabei ist vor allem ein gemeinsames Monitoring für eine bessere Kollaboration zwischen Softwareentwicklung und -betrieb überaus wichtig. Und die Stärkung der Kollaboration zwischen Dev und Ops bleibt das Kernstück der DevOps-Philosophie. Wie sollen diese beiden Gruppen aber besser miteinander kollaborieren, wenn sie bislang getrennte Wege in unterschiedlichen Silos gegangen sind? Darauf gibt es nicht die eine richtige Antwort und es heißt auch nicht zwangsläufig, dass die Ops-Teams aufgelöst werden müssen. Matthew Skelton hat eine Webseite aufgebaut, auf der er zahlreiche verschiedene Patterns und Antipatterns beschreibt [9]. Dabei sind nach meiner Meinung die folgenden drei am häufigsten anzutreffen:

- *Ein Team aus Dev und Ops:* Die vorher getrennten Teams verschmelzen zu einem Team und übernehmen zukünftig gemeinsam die Entwicklung und den Betrieb der Anwendung (**Abb. 7**). Dieses Modell kommt dem Leitsatz „You build it, you run it“ von Werner Vogels, dem CTO von Amazon, am nächsten.
- *Kollaboration:* Dev und Ops kollaborieren z. B. über Hospitationsmodelle miteinander bzw. nehmen gegenseitig an Meetings wie Sprint Planning oder Stand-up teil (**Abb. 8**). Außerdem stellt das Dev-Team ordentliche Run Books etc. für das Ops-Team bereit.
- *Infrastructure as a Service:* Das Ops-Team stellt Services bereit, die das Entwicklungsteam selbstständig verwenden kann. Dafür muss das Ops-Team nicht zur selben Firma gehören, sondern kann z. B. bei einem Cloudanbieter sitzen (**Abb. 9**).

Ein gängiges Antipattern ist in der Realität leider auch häufig anzutreffen. Die beiden Teams für Dev und Ops bleiben bestehen und haben keinen direkten Kontakt zueinander. Zwischen den beiden steht ein DevOps-Team,

das für Infrastrukturautomatisierung etc. zuständig ist.

Google hat mit dem Site Reliability Engineering ein weiteres Muster für sich entwickelt, das mittlerweile auch von anderen Firmen übernommen wurde. Die Kernidee dieses Konzepts ist, Ops-Teams aus Software Engineers zu bilden. Software Engineers haben laut Google ein intrinsisches Interesse daran, Abläufe zu automatisieren und zu verbessern. Dadurch sollen Anwendungen entstehen, die sich im Wesentlichen ohne menschliches Zutun selbst überwachen und im Notfall heilen. Aus diesem Grund darf ein SRE-Team auch maximal 50 Prozent seiner Zeit für die Behebung von Incidents oder andere klassische Ops-Tätigkeiten aufwenden. Der Großteil der Zeit muss in Automatisierungen und Verbesserungen fließen.

Ein weiteres Konzept aus dem Site Reliability Engineering ist das des Error Budgets. Dabei geht man davon aus, dass nahezu keine Softwareanwendung eine Verfügbarkeit von 100 Prozent benötigt. Was allein schon deshalb sinnvoll ist, weil Infrastrukturkomponenten wie WLAN, Mobilfunknetz, Internetanschluss etc., die zwischen dem Nutzer und der Anwendung liegen, auch keine solch hohe Verfügbarkeit haben. Wenn die Verfügbarkeit also kleiner als 100 Prozent ist, wird sie meist über die Anzahl der Neuner hinter dem Komma definiert (je mehr, desto teurer). Die Differenz zwischen 100 Prozent und der angestrebten Verfügbarkeit ist das Error Budget (quasi eine prognostizierte Nichtverfügbarkeit). Diese Zeit kann für geplante Downtimes verwendet werden, wird aber auch durch ungeplante Ausfälle reduziert.

An der Einstellung zu Ausfällen und Fehlern zeigt sich ein grundlegender Unterschied zum Jahr 2009. Der IT-Betrieb hatte traditionell das Ziel, Ausfälle und Fehler auf null zu reduzieren. In Zeiten komplexer, verteilter Infrastruktur können Ausfälle und Fehler nicht mehr (zu sinnvollen Kosten) ausgeschlossen werden, deshalb versucht man den Umgang damit zu professionalisieren. Dazu gehört zum einen das absichtliche Provozieren von Ausfällen im Chaos Engineering (Ausfall oder Beeinträchtigungen von Infrastrukturkomponenten) und zum anderen eine gesunde Fehlerkultur, wie sie auch schon Allspaw und Hammond in ihrem Vortrag thematisierten. Eine solche Kultur bedeutet, dass man akzeptiert, dass Fehler passieren, man aber trainiert, sinnvoll mit ihnen umzugehen und aus ihnen zu lernen. Ein Mittel dafür sind sogenannte Blameless Post-mortems. Dabei handelt es sich um spezielle Retrospektiven, die einem festen Muster folgen [11]. Die Ergebnisse werden dokumentiert und als Lernmaterial allen anderen Teams in der Organisation zur Verfügung gestellt.

Ein Team, das ein Blameless Post-Mortem durchführt, steht vor der Herausforderung, den Ablauf eines Incidents und die verschiedenen Einflussfaktoren lückenlos zu rekonstruieren. Besonders einfach wird diese Forensik, wenn man wie GitHub alle Deployments, Metriken etc. im Group-Chat-Tool abgebildet hat. Dieses Vorgehen wird als Chat Ops bezeichnet, und wir erinnern uns, dass auch der Vortrag von Allspaw und Hammond eine Vorstufe davon enthielt.

Fazit und Ausblick

Wenn wir auf die Anfänge der DevOps-Bewegung im Jahr 2009 zurückschauen, hat sich im technischen Bereich vieles verändert. Kulturell und methodisch sind die Gedanken insbesondere von Allspaw und Hammond überraschend aktuell.

Doch zunächst zu den technischen Aspekten. Sebastian Meyen schreibt im Editorial des Java Magazins 6.2009:

„Die Idee, das Thema Hardware und Basisinfrastruktur im Sinne des Cloud Computings zu organisieren, wird sich indes ohne Zweifel durchsetzen. Bis dahin ist aber noch ein steiniger Weg zu gehen. Noch existiert kein Konsens, wie viel und in welcher Weise z. B. Java unterstützt werden soll, noch haben sich die Cloud-Anbieter nicht auf gemeinsame Standards geeinigt, um größtmögliche Kompatibilität zu erzielen. Die Cloud-Revolution steht gerade erst am Anfang und es konkurrieren vor allem hochproprietäre Angebote um die Gunst der Kunden.“

Zehn Jahre später gibt es kaum noch Unternehmen, die sich nicht mit dem Thema Cloud auseinandersetzen (müssen). Gemeinsame Industriestandards sind dabei nicht direkt entstanden. Mittlerweile schickt sich aber Kubernetes an, der De-facto-Standard als Cloudbetriebssystem zu werden.

Anders als im Jahr 2009 bedeutet Cloud-Computing heute immer seltener, Rechenleistung und Storage in Form von virtuellen Servern zu mieten und dort Infrastrukturkomponenten zu installieren. Stattdessen mieten Unternehmen zunehmend sogenannte Managed Services wie Datenbanken, Webserver etc., bei denen sich der Anbieter um Skalierbarkeit, Ausfallsicherheit und das Patching kümmert. Dieser Trend wird sich in den nächsten Jahren verstetigen. Damit folgt die Entwicklung im Infrastrukturbereich der in der Softwareentwicklung. Genauso wie eine moderne Anwendung heute zum größten Teil aus fremdem Code in Form von Libraries besteht, werden auch mehr und mehr Infrastrukturkomponenten durch Managed Services ersetzt. Am Ende wird nur eine sehr schmale Schicht mit anwendungsspezifischen Aspekten übrigbleiben. Das klingt eigentlich wie das hohe Ziel der Softwareentwicklung: Die Entwickler können sich vollständig auf ihre Businesslogik konzentrieren und alle anderen Aspekte werden ihnen abgenommen. Mit diesem Anspruch waren auch Enterprise JavaBeans (EJBs) seinerzeit angetreten. Genau wie bei EJBs ist die Komplexität aber nicht verschwunden, sondern wandert an andere Stellen – damals in die komplexen XML-Deployment-Deskriptoren, heute in die komplexen Infrastructure-as-Code-Beschreibungen. Es bleibt dabei, dass sich Komplexität in der IT nicht abschaffen, sondern nur verschieben lässt.

Dies soll kein Plädoyer gegen Infrastructure as Code sein – im Gegenteil. Schon 2009 forderten Allspaw und Hammond die Automatisierung von Infrastruktur ein. Daraus ergeben sich viele Vorteile. Neue Umgebungen können innerhalb kürzester Zeit erzeugt werden, Deployments sind sicherer und für alle nachvollziehbar und sorgen dafür, dass wir uns von den Haustieren in der IT verabschieden.

Infrastrukturautomatisierung ist einer der offensichtlichsten Teile der DevOps-Philosophie und wird relativ gut durch Tools unterstützt. Aus diesem Grund gibt es leider immer noch Unternehmen, die Infrastrukturautomatisierung und DevOps gleichsetzen. Dabei können Initiativen in diesem Bereich nur erfolgreich sein, wenn vorher bestimmte Hausaufgaben wie die Bereinigung der Technologiestacks durchgeführt werden.

Noch häufiger wird allerdings die kulturelle Seite der DevOps-Idee vergessen, deren Grundstein Allspaw und Hammond schon 2009 legten. Ein wichtiges Prinzip der DevOps-Kultur ist das transparente Teilen von Informationen. Das beginnt mit den gemeinsamen Metriken, geht weiter über das CAMS-Modell, in dem das S für Sharing steht, und findet sich heute auch in Praktiken wie Blameless Post-Mortems oder ChatOps wieder. Das zweite Prinzip, das sich wie ein roter Faden durch alle erfolgreichen DevOps-Ansätze zieht, ist eine gesunde Fehlerkultur. In modernen Organisationen akzeptieren wir heute, dass Fehler passieren können und werden. Wir versuchen, möglichst gut damit umzugehen.

Diese beiden Prinzipien sind das Fundament einer vertrauensvollen Kollaboration zwischen Dev und Ops, um gemeinsam das Business voranzubringen. Wenn diese Kollaboration sich erfolgreich entwickelt und die beiden Silos verschmelzen, werden oft die anderen Silos in der Organisation sichtbar, die sich rund um Dev und Ops befinden: Fachbereiche, Produktmanagement, Support, Marketing, Sales. Die DevOps-Reise macht in diesem Fall nicht halt, sondern versucht, immer mehr Silos in der Organisation aufzulösen, um das Business erfolgreich zu machen. So entstehen mit der Zeit echte cross-funktionale Produktteams.

Technologisch haben sich viele Firmen bereits auf die DevOps-Reise begeben und erste Fortschritte gemacht, in kultureller Hinsicht gibt es aber etliche, die noch am Anfang der Reise stehen.

Konstantin Diener ist CTO bei cosee. Das Unternehmen wurde im Jahr 2009 gegründet und übernahm früh auch den Betrieb der Lösungen, die es für Kunden entwickelt. Erst Monate später fiel den Mitarbeitern auf, dass DevOps cosees Vorgehen sehr gut beschreibt. Seitdem beschäftigt sich Konstantin mit Agilität und DevOps, schreibt darüber und spricht auf Konferenzen.

https://cosee.biz @onkelkodi @coseeaner

Links & Literatur

[1] https://de.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr

[2] Seemann, Michael: „Cloud Computing mit Java“, in: Java Magazin 7.2009

[3] Diener, Konstantin: „(Un)regelmäßige Integration“, in: Java Magazin 12.2018, https://jaxenter.de/devops-stories-unregelmaessige-integration-stolpersteine-fuer-continuous-integration-77965

[4] https://devops.com/the-origins-of-devops-whats-in-a-name/

[5] Kim, Gene; Behr, Kevin; Spafford, George: „The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win“, It Revolution Press, 2013

[6] https://inthecloud.withgoogle.com/state-of-devops-18/dl-cd.html

[7] Diener, Konstantin: „Wenn einer eine (DevOps-)Reise tut …“, in: Java Magazin 7.2019, https://jaxenter.de/devopsstories-kolumne-devops-reise-84452

[8] Diener, Konstantin: „Keine Haustiere in der IT!“, in: Java Magazin 5.2019, https://jaxenter.de/devops-cloud-infrastruktur-cattle-83174

[9] https://web.devopstopologies.com

[10] https://landing.google.com/sre/sre-book/toc/index.html

[11] https://landing.google.com/sre/sre-book/chapters/postmortem-culture/

Training von TensorFlow-Modellen mit JVM-Sprachen

Deep Learning: nicht nur in Python

Zwar gibt es mit Frameworks wie DL4J mächtige und umfangreiche Machine-Learning-Lösungen für die JVM, dennoch kann es in der Praxis vorkommen, dass der Einsatz von TensorFlow notwendig wird. Das kann beispielsweise der Fall sein, wenn es einen bestimmten Algorithmus nur in einer TensorFlow-Implementierung gibt und der Portierungsaufwand in ein anderes Framework zu hoch ist. Zwar interagiert man mit TensorFlow über ein Python API, die zugrunde liegende Engine jedoch ist in C++ geschrieben. Mit Hilfe der TensorFlow-Java-Wrapper-Bibliothek kann man deshalb sowohl Training als auch Inferenz von TensorFlow-Modellen aus der JVM heraus betreiben, ohne auf Python angewiesen zu sein. So können bestehende Schnittstellen, Datenquellen und Infrastruktur mit TensorFlow integriert werden, ohne die JVM zu verlassen.

von Christoph Henkelmann

KI und Deep Learning sind immer noch in aller Munde und trotz einiger erster Rückschläge, beispielsweise im Bereich selbstfahrender Autos, ist das Potenzial von Deep Learning noch lange nicht ausgeschöpft. Auch gibt es noch viele Bereiche der IT, in denen das Thema gerade erst richtig Fahrt aufnimmt. Daher ist es besonders wichtig, zu schauen, wie man Deep-Learning-Systeme auf der JVM realisieren kann, denn Java (sowohl die Sprache als auch die Plattform) sind immer noch dominierende Technologien im Enterprise-Bereich.

TensorFlow ist eins der wichtigsten Frameworks im Deep-Learning-Bereich und trotz der steigenden Popularität von Keras noch immer nicht wegzudenken, insbesondere da KI-Platzhirsch Google die Entwicklung weiter vorantreibt. In diesem Artikel wird gezeigt, wie TensorFlow aus einer JVM heraus sowohl für das Training von TensorFlow-Modellen als auch für die Inferenz genutzt werden kann.

Wofür ist die Kombination aus TensorFlow und JVM geeignet?

Möchte man auf der JVM Deep Learning betreiben, ist normalerweise DL4J das Mittel der Wahl, da es als einziges professionelles Deep Learning Framework wirklich auf der JVM zu Hause ist. TensorFlow wird hauptsächlich – wie viele Machine Learning Frameworks – mit Python genutzt. Es gibt jedoch Gründe, TensorFlow aus einem JVM-Kontext heraus zu nutzen:

- Man möchte ein Verfahren nutzen, für das es bei TensorFlow, nicht aber bei DL4J eine Implementierung gibt, und für das der Portierungsaufwand zu hoch ist.
- Man arbeitet mit einem Data-Science-Team, das gewohnt ist, mit TensorFlow und Python zu arbeiten, aber die Zielinfrastruktur läuft auf der JVM.
- Die für das Training notwendigen Daten liegen in einer Java-Infrastruktur (Datenbanken, eigene Dateiformate, APIs), und um an die Daten zu gelangen, müsste existierender Schnittstellencode von Java nach Python portiert werden.

Die JVM-TensorFlow-Kombination ist also immer dann sinnvoll, wenn eine existierende Java-Umgebung vorhanden ist und aus personellen oder projekttechnischen Gründen trotzdem TensorFlow für Deep Learning genutzt werden sollte (Kasten: „TensorFlow und JVM – immer eine gute Idee?“).

Wie funktioniert TensorFlow?

Bevor man ein neues Framework einsetzt, ist es wichtig, sich ein wenig damit auseinanderzusetzen, was unter der Haube passiert (Kasten: „TensorFlow Begriffs-Cheat-Sheet“). Bei TensorFlow denkt man zunächst einmal an KI und neuronale Netze, aber technisch gesehen handelt es sich vor allem um ein Framework, das komplexe, sich immer wiederholende parallele Berechnungen auf Tensoren ausführen kann – und das, wenn möglich, GPU-beschleunigt. Auch wenn Deep Learning das

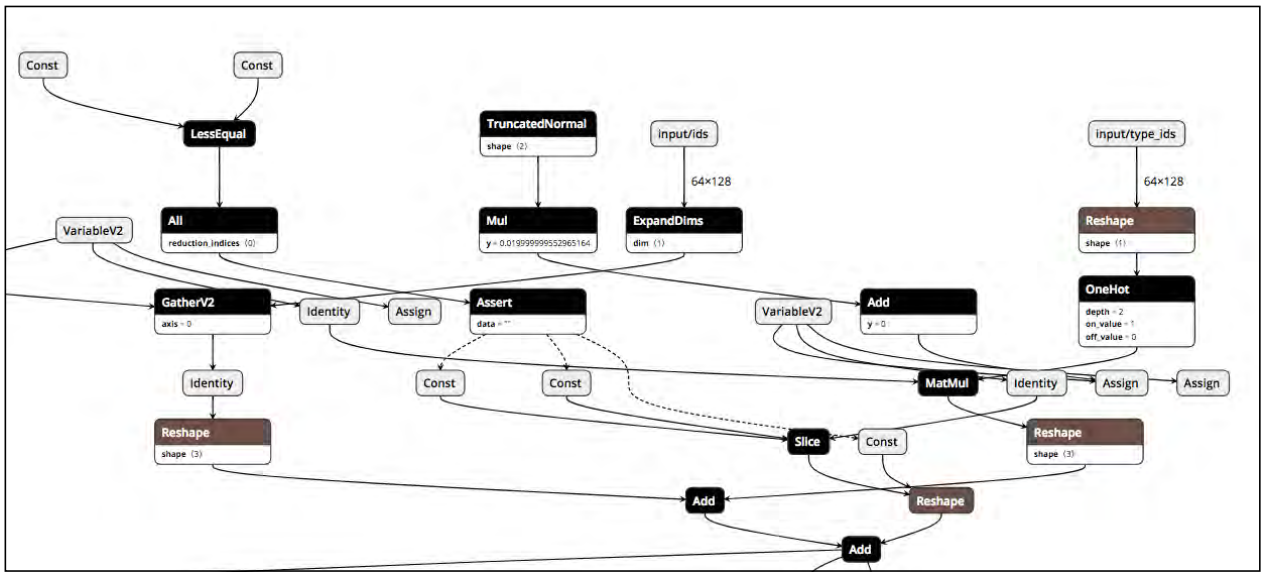

Abb. 1: Ein (kleiner) Ausschnitt aus einem TensorFlow-Graphen: Die Zahlen an den Kanten zeigen die Größe des Tensors an, der durch sie fließt, die Pfeile die Richtung

Hauptanwendungsgebiet für TensorFlow ist, kann man auch beliebige andere Berechnungen damit durchführen.

Ein TensorFlow-Programm – oder besser: die Konfiguration einer Berechnung – ist in TensorFlow immer wie ein Graph aufgebaut. Die Knoten des Graphen stellen Operationen dar, wie etwa Addieren oder Multiplizieren, aber auch Laden und Speichern. Alles was TensorFlow tut, findet in den Knoten eines vorher definierten Berechnungsgraphen statt.

Die Knoten (Operationen) des Graphen sind durch Kanten verbunden, durch die die Daten in Form von Tensoren fließen. Daher auch der Name TensorFlow.

Alle Berechnungen in TensorFlow finden in einer sogenannten Session statt. In der Session wird entweder ein fertiger Graph geladen oder stückweise durch API-Aufrufe ein neuer Graph erzeugt. Spezielle Knoten im Graphen können Variablen enthalten. Damit der Graph funktioniert, müssen diese initialisiert werden. Ist das geschehen und existiert eine Session mit fertigem, initialisiertem Graphen, interagiert man mit TensorFlow nur noch durch das Aufrufen von Operationen im Graphen. Was dabei berechnet wird, hängt davon ab, welche Aus-

TensorFlow und JVM – immer eine gute Idee?

Auch wenn es gute Gründe für diese Kombination geben kann, ist es auch wichtig, anzusprechen, was dagegensprechen kann. Insbesondere die Wahl von TensorFlow sollte gut bedacht sein:

- TensorFlow ist kein geeignetes Framework für Deep- oder Machine-Learning-Anfänger.
- TensorFlow ist nicht benutzerfreundlich: Das API ändert sich schnell, oft ist in dem Wust von Anleitungen nicht klar, welcher Weg der geeignete ist.
- TensorFlow ist nicht besser, nur weil es von Google ist: Deep Learning ist Mathematik, und die ist für alle gleich. TensorFlow erzeugt keine „schlaueren" KIs als andere Frameworks. Auch ist es nicht schneller als die Alternativen (allerdings auch nicht „dümmer" oder langsamer).

Möchte man in Deep Learning einsteigen und auf der JVM bleiben, ist der Einsatz von DL4J absolut zu empfehlen. Gerade für professionelle Enterprise-Projekte ist DL4J eine gute Wahl. Aber auch wenn man sozusagen über den Zaun blicken und ein wenig Python ausprobieren möchte, lohnt es sich, nach TensorFlow-Alternativen zu schauen. Hier ist man aktuell bei Keras besser aufgehoben, einem wesentlich angenehmeren API sei Dank.

gabeknoten des Graphen abgefragt werden. Es wird also nicht der gesamte Graph ausgeführt, sondern nur die Operationen, die Input für den abgefragten Knoten liefern, dann wiederum deren Eingabeknoten etc. zurück zu den Eingabeoperationen, die mit dem notwendigen Inputtensoren befüllt sein müssen.

Das wichtige bei TensorFlow ist, dass alle Operationen automatisch für den Nutzer differenziert werden – das braucht man für das Training von neuronalen Netzen. Da es jedoch automatisch geschieht, kann man es als Anwender getrost ausblenden.

Der Graph wird in der Regel durch ein Python API definiert. Er ist zwar mit Hilfsprogrammen grafisch darstellbar (**Abb. 1**), solche Darstellungen dienen aber nur dem Debugging, es wird nicht grafisch programmiert wie in einer visuellen Programmiersprache, wie etwa LabView.

Obwohl in den meisten Beispielen Python genutzt wird, um mit TensorFlow zu interagieren, ist die eigentliche Engine in C/C++ geschrieben. Man kann TensorFlow also mit jeder Sprache nutzen, die C-Funktionen aufrufen kann. Damit können wir also auch von der JVM aus Berechnungen in TensorFlow durchführen.

TensorFlow-Training und -Inferenz unter Python

Das Training eines TensorFlow-Modells unter Python (Kasten: „tf.data oder Feeding?“) lässt sich in folgende Schritte unterteilen:

- Erzeugen des Graphen, entweder über mehrere API Calls, die den Graphen zusammensetzen, oder durch das Laden einer **.pb*-Datei, die den Graphen enthält
- Erzeugen einer Session für den Graphen
- Initialisieren der Graphvariablen, entweder durch Aufrufen einer speziellen Operation im Graphen, die die Variablen mit Defaultwerten füllt, oder durch Laden eines vortrainierten Modells

Nach diesen drei Schritten hat man eine lauffähige TensorFlow-Session mit einem funktionierenden Modell. Wollen wir dieses nun (weiter-)trainieren, werden immer wieder die folgenden drei Schritte in einer Schleife ausgeführt, bis das Modell genug gelernt hat – entweder indem man vorher eine feste Anzahl an Trainingsschritten festlegt oder indem man wartet, bis der Fehler beim Training unter ein gewisses Maß sinkt:

- Eingabedaten in Arrays packen und Eingabetensoren zuordnen
- Ausgabeknoten auswählen und in eine Liste packen
- Die Session ausführen: Über einen speziellen Befehl wird die Session veranlasst, die notwendigen Operationen zum Erzeugen der gewählten Ausgaben auszuführen

Wo aber findet hier das Training statt? Es geschieht durch das Ausführen der richtigen Ausgabeknoten. Für TensorFlow gibt es keinen Unterschied zwischen Training und Inferenz, es werden einfach mathematische Operationen im Berechnungsgraphen ausgeführt. Führen diese dazu, dass ein neuronales Netz bessere Gewichte lernt, um ein Problem zu lösen, sprechen wir von Training. Die API-Aufrufe für Training und jede andere

TensorFlow Begriffs-Cheat-Sheet

- *Tensor*: Die Grundlage für Berechnungen in TensorFlow. Ein Tensor ist eigentlich ein Objekt aus der linearen Algebra, für unsere Zwecke reicht es aber völlig aus, einen Tensor als mehrdimensionales Array (meist aus *float*- oder *double*-Werten, manchmal auch *char* oder *boolean*) zu betrachten. TensorFlow nutzt Tensoren für alles. Alle Daten, die TensorFlow konsumiert, produziert und intern nutzt, sind in Tensoren verpackt – daher auch der Name.
- *Graph*: Die Definition von TensorFlow-Berechnungsabläufen wird üblicherweise in einer Datei mit dem Namen *graph.pb* im ProtoBuf-Binärformat gespeichert, analog zu einer Java- *.class*-Datei.
- *Training*: Beim Training eines Machine-Learning-Verfahrens werden dem Algorithmus immer wieder Daten und erwartete Ergebnisse vorgelegt, woraufhin dieser die internen Parameter des Modells anpasst, um das Ergebnis zu verbessern. Manchmal spricht man hier auch von „Lernen“, obwohl es mit menschlichem Lernen wenig zu tun hat.
- *Inferenz*: Je nach Anwendung möchte man mit einem Machine-Learning-Verfahren klassifizieren, vorhersagen, übersetzen, Inhalte erzeugen und vieles mehr. All diese Anwendungen werden unter dem Begriff Inferenz zusammengefasst. Inferenz bedeutet also so viel wie „ein Verfahren anwenden, um ein Ergebnis zu erhalten“. Also das, was wir nach dem Training im Liveeinsatz die meiste Zeit tun möchten. Bei der Inferenz lernt ein Verfahren nichts.
- *Modell*: Die gelernten Parameter eines Machine-Learning-Verfahrens, zum Beispiel eines neuronalen Netzes; das ist das Ergebnis des Lernprozesses und nötig, um Ergebnisse zu erhalten (quasi der Variablenzustand des Graphen). Es wird über mehrere Dateien verteilt gespeichert in einer **.index*- und mehreren **.data*-Dateien, zum Beispiel **.data-0000-of-0001*, wobei die erste Zahl die fortlaufende Nummer der Datei angibt, die zweite die Gesamtanzahl.
- *Session*: Kontext, in dem TensorFlow ausgeführt wird, wie eine laufende JVM-Instanz; um TensorFlow zu nutzen, muss man eine Session erzeugen, in die ein Graph geladen wird, der mit einem Modell initialisiert wird. In Java muss eine JVM-Instanz gestartet werden, in die Klassen geladen werden, die mit Konstruktorparametern instanziiert werden.

Art von Nutzung sind jedoch die gleichen. Beim Training besteht unser Input aus den zu lernenden Daten (zum Beispiel einem Bild als zweidimensionalem Tensor und dem Label „Hund“ oder „Katze“ in Form einer Integer-ID in einem nulldimensionalen Tensor). Durch das Ausführen der richtigen Knoten aktualisiert TensorFlow manche Variablen im Graphen, um die Vorhersage zu verbessern. Der Hauptunterschied zwischen Training und Inferenz ist, dass wir während des Trainings in regelmäßigen Abständen den aktuellen Zustand der Graphvariablen – die sich ja ständig ändern – speichern, während das bei der Inferenz nutzlos ist, da sie konstant bleiben.

Das TensorFlow Java API

Da TensorFlow intern in C/C++ implementiert ist, können wir nun alle Operationen, die in Python für das Training oder die Inferenz notwendig sind, über JNI aufrufen.

Erfreulicherweise muss man sich nicht mehr selbst die Mühe machen, das Low-Level C API mit JNI zu wrappen. Diese Aufgabe hat Google bereits für uns erledigt, die notwendigen Bibliotheken stehen wie gewohnt auf Maven Central zur Verfügung. Es gibt vier unterschiedliche Artefakte, alle in der Gruppe org.tensorflow:

- *tensorflow:* Ein Metapaket mit Abhängigkeiten auf libtensorflow und libtensorflow_jni; um Verwirrung zu vermeiden, sollte es nicht benutzt werden.
- *libtensorflow:* Das API, gegen das man in Java programmiert; das ist die Compile- und Runtime-Abhängigkeit und der zentrale Einstiegspunkt.
- *libtensorflow_jni:* Enthält die nativen CPU-Abhängigkeiten für libtensorflow; dieses Artefakt braucht man zur Laufzeit, wenn man einen Rechner ohne GPU nutzt; es enthält nativen Code für Windows, Linux und Mac; TensorFlow ist komplett enthalten, man muss auf dem ausführenden System weder Python oder TensorFlow installieren.
- *libtensorflow_jni_gpu:* Das GPU-Äquivalent zu libtensorflow_jni; diese Abhängigkeit sollte man nutzen, wenn man einen Rechner mit NVIDIA GPU nutzt und Cuda und cuDNN korrekt installiert sind; es funktioniert nur unter Windows und Linux, unter macOS gibt es keinen GPU-Support für TensorFlow.

Die Versionsnummern der Java Wrapper entsprechen der Versionsnummer der enthaltenen TensorFlow-Version. Hier sollte man einfach immer das neueste stabile Release geben. Aufpassen muss man nur, wenn der Code auf einem Rechner mit GPU ausgeführt werden soll (Kasten: „Auswahl der zu nutzenden GPU“). Nicht jede TensorFlow-Version unterstützt jede CUDA- und cuDNN-Version. (CUDA ist ein spezieller NVIDIA-Treiber, um Grafikkarten für parallele Berechnungen zu nutzen, cuDNN eine auf CUDA basierende Bibliothek für neuronale Netze.) Es muss darauf geachtet werden, das CUDA- und TensorFlow-Version zueinander passen. Aktuell unterstützen alle TensorFlow-Versionen ab 1.13 die gleiche CUDA-Version: 10.0. Hier hat man mit einer Java-basierten Lösung bei der Installation der fertigen Software schon einen großen Vorteil gegenüber einer Python-Software. Dank Maven bringt unser resultierendes Artefakt alle Abhängigkeiten bereits mit. Weder Python noch TensorFlow noch irgendwelche Python-Bibliotheken müssen vorinstalliert und die Installationen mit einem Tool wie Anaconda verwaltet werden.

tf.data oder Feeding?

Trainiert man ein TensorFlow-Modell in Python, ergeben sich zwei Möglichkeiten, Trainingsdaten in den Graphen zu laden: das tf.data API oder sogenanntes „feeding“, also das Übergeben einzelner Daten für jeden Berechnungsschritt. Das tf.data API ist intern in C implementiert, direkt in den Graphen integriert und daher sehr schnell – dafür aber auch kompliziert in der Nutzung und sehr schwer zu debuggen. Die Feeding-Methode ist leicht zu nutzen und verständlich, man braucht aber zur Laufzeit Python-Code – daher bremst hier meistens Python die teurere Grafikkarte aus und wertvolle GPU-Kapazität wird nicht genutzt. Welchen Ansatz nutzen wir aber nun in Java? Glücklicherweise ist Java um Größenordnungen schneller als Python, daher erhalten wir hier das Beste aus beiden Welten: Wir können die leicht zu verstehende Feeding-Methode nutzen und bekommen trotzdem die volle Performance. Aus diesem Grund lassen wir in diesem Artikel das tf.data API außen vor, wir brauchen es einfach nicht.

Auswahl der zu nutzenden GPU

Auf Systemen mit mehreren GPUs möchte man manchmal nicht alle GPUs blockieren, zum Beispiel um mehrere Trainings parallel laufen zu lassen. Hierzu kann man den TensorFlow-Graphen, der normalerweise automatisch die GPU oder GPUs allokiert, so konfigurieren, dass nur eine bestimmte GPU genutzt wird. Das hat aber den großen Nachteil, dass dann der Graph auf eine bestimmte GPU „hart verdrahtet“ wird und nur noch auf dieser GPU einsetzbar ist. Viel bequemer ist es, die GPUs per Umgebungsvariable ein- bzw. auszublenden, bevor man die JVM startet. Das geht leicht mit der Umgebungsvariable *CUDA_VISIBLE_DEVICES*. Hier kann man eine kommaseparierte Liste von CUDA-Devices angeben, die in der aktuellen Shell sichtbar sein sollen. Vorsicht: Die Nummerierung beginnt bei 1, nicht bei 0. Folgender Konsolenbefehl aktiviert beispielsweise nur die zweite Grafikkarte für TensorFlow (oder andere Deep Learning Frameworks):

```
export CUDA_VISIBLE_DEVICES=2
```

Es sollte nicht die Top-Level-Abhängigkeit tensorflow genutzt werden, sondern besser direkt libtensorflow und eine der *_jni-Implementierungen. Grund hierfür ist, dass das tensorflow-Artefakt eine Abhängigkeit auf libtensorflow_jni (die CPU-Variante) mitbringt. Fügt man nun libtensorflow_jni_gpu hinzu, wird trotzdem der CPU-native Code genutzt und man wundert sich, warum trotz GPU alles so langsam läuft. Die Gradle-Abhängigkeiten für das TensorFlow-Training auf der GPU sehen dann z. B. so aus:

```
compile "org.tensorflow:libtensorflow:1.14.0"
runtimeOnly "org.tensorflow:libtensorflow_jni_gpu:1.14.0"
```

Das für Training und Inferenz notwendige Java API ist einfach und überschaubar. Wichtig sind nur vier Klassen: *Graph*, *Session*, *Tensor* und *Tensors*. Wie diese richtig eingesetzt werden, sehen wir jetzt, indem wir die Python-typischen Trainingsschritte in Java nachbauen.

TensorFlow-Training unter Java

Der erste Schritt im Training ist die Definition des Graphen. Leider müssen wir hier gleich zu Beginn den ersten, aber einzigen Kompromiss eingehen. Zwar kann auch ein Graph über das Java API schrittweise aufgebaut werden, aber bei vielen Knotentypen erzeugt das Python API viele notwendige Helferknoten automatisch mit, die für die reibungslose Nutzung des Graphen notwendig sind. Um das in Java nachzubauen, bedürfte es eines sehr detaillierten Wissens über die Interna des Python API. Daher muss dieser Schritt im Vorfeld einmal in Python durchgeführt werden. Die resultierende Graphdatei legen wir dann als Java Resource ab, um sie dann wieder in der JVM zu laden. Das Speichern des aktuellen Graphen in Python ist denkbar einfach:

```
with open(filename, 'wb') as f:
  f.write(tf.get_default_graph().as_graph_def().SerializeToString())
```

Wichtig dabei: Auch wenn hier *SerializeToString()* aufgerufen wird, ist das Resultat doch eine Binärdatei.

Listing 1

```
# Dieser Python-Befehl legt einen Knoten zum Initialisieren an.
init_op = tf.global_variables_initializer()
# Der Saver ist eine Hilfsklasse, die in Python ein Modell speichert.
saver = tf.train.Saver()
# Speichern ist eine Graphenoperation
# und kann nur in einer Session ausgeführt werden.
with tf.Session() as sess:
  # Variablen initialisieren
  sess.run(init_op)
  # Zustand speichern
  save_path = saver.save(sess, filename)
```

Bei komplexen Modellen lässt sich durch Transfer Training ein bereits existierender Zustand weitertrainieren und anpassen.

Bequemlichkeitshalber sollte man hier auch gleich die initialisierten Variablen speichern. Zwar ist das Initialisieren der Variablen im Graphen von der JVM aus einfach, wählt man aber immer diesen Ablauf, ist es hinterher leicht, bei komplexen Modellen sogenanntes Transfer Training zu betreiben. Dabei wird ein bereits existierender Zustand eines Modells weitertrainiert und angepasst (Listing 1).

Nun haben wir Graph und Modell gespeichert und können es in Java trainieren und den Graphen ausführen. Die folgenden Beispiele sind der Kürze halber in Kotlin, lassen sich aber auf jede JVM-Sprache übertragen:

```
//leeren Graphen erzeugen
val graph = Graph()
//*.pb Datei laden - entweder aus einer Datei oder aus den Ressourcen
```

```
val graphDefBytes = javaClass.getResource(resourceName).readBytes()
//Graphen aus Datei rekonstruieren
graph.importGraphDef(graphDefBytes)
```

Nun haben wir den TensorFlow-Graphen in der JVM geladen. Um etwas damit zu machen, brauchen wir eine Session:

```
val session = Session(graph)
```

Bevor wir richtig loslegen können, muss nur noch der letzte Stand der Variable geladen werden. Das kann entweder die initial in Python gespeicherte Datei sein oder der letzte Stand eines vorangegangenen Trainings, zum Beispiel, um ein Training fortzusetzen. Das Laden von Variablen ist nur eine Operation im TensorFlow-Graphen. Diese Operation braucht als Input einen in einen Tensor gepackten String, der den Namen der **.index*-Datei ohne das Suffix enthält, also *foo* statt *foo.index*.

Hier brauchen wir zum ersten Mal die *Tensors*-Klasse. Diese enthält Hilfsfunktionen, um Java-Datentypen in *Tensor*-Objekte zu verpacken. Dabei wird automatisch darauf geachtet, dass der *Tensor* die richtige Form hat. Wichtig für jedes *Tensor*-Objekt: Es enthält Speicher, der außerhalb der JVM allokiert wurde. Daher muss es manuell geschlossen werden, wofür es das *Closable*-Interface implementiert. In Java muss für jeden Tensor ein eigener Block *try{…} finally { tensor.close(); }* angelegt werden. In Kotlin geht das zum Glück viel einfacher mit *use:*

```
Tensors.create(path).use { pathTensor ->
  session.runner().feed("save/Const", pathTensor)
                  .addTarget("save/restore_all")
                  .run()
}
```

Hier sieht man alle notwendigen Teile einer TensorFlow-Aktion auf der JVM:

Listing 2

```
fun train(inputs: Array<FloatArray>, labels: IntArray) {
  withResources {
    val results: List<Tensor<*>> = session.runner()
      .feed("inputs", Tensors.create(inputs).use())
      .feed("labels", Tensors.create(labels).use())
      .fetch("total_loss:0")
      .fetch("accuracy:0")
      .fetch("prediction")
      .addTarget("optimize").run().useAll()
    val trainingError = results[0].floatValue()
    val accuracy      = results[1].floatValue()
    val prediction    = results[2].intValue()
  }
}
```

- Es wird ein Runner für die Session erzeugt; diese Klasse hat ein Builder API, das definiert, was ausgeführt werden soll.
- Der Inputknoten für das Laden und Speichern („*save/Const*“) wird mit dem Tensor gefüllt, der den Dateinamen enthält.
- Der Zielknoten für das Laden wird als Ziel definiert.
- Die Aktion wird ausgeführt.

Der Trick für alle Operationen ist es, ihre Namen zu kennen. Da man aber den Graphen selbst zuvor baut und man beim Erzeugen eines Knotens den Namen definieren kann, kann man diese selbst wählen. Ausnahme sind die Knoten zum Laden und Speichern, die immer die hier angegebenen Namen haben.

Nun haben wir bereits alle Operationen gesehen, die es zur Interaktion mit TensorFlow von der JVM aus braucht. Einen Trainingsschritt durchzuführen, ist nun denkbar einfach. Nehmen wir an, unser Input ist ein Array aus geladenen Bildern. Die Schwarz-Weiß-Werte der Pixel sind in *float*-Werte im Bereich 0-1 umgewan-

Listing 3

```
class Resources : AutoCloseable {
  private val resources = mutableListOf<AutoCloseable>()

  fun <T: AutoCloseable> T.use(): T {
    resources += this
    return this
  }

  fun <T: Collection<AutoCloseable>> T.useAll(): T {
    resources.addAll(this)
    return this
  }

  override fun close() {
    var exception: Exception? = null
    for (resource in resources.reversed()) {
      try {
        resource.close()
      } catch (closeException: Exception) {
        if (exception == null) {
          exception = closeException
        } else {
          exception.addSuppressed(closeException)
        }
      }
    }
    if (exception != null) throw exception
  }
}

inline fun <T> withResources(block: Resources.() -> T): T =
  Resources().use(block)
```

delt. Jedes Bild gehört zu einer Klasse, die durch einen *int*-Wert definiert ist, zum Beispiel 0 = Hund, 1 = Katze. Dann ist der Input für einen Batch (es werden immer mehrere Bilder gleichzeitig trainiert) ein *float[][][]*-Array, das die Bilder enthält, und ein *int[]*-Array, das die zu lernenden Klassen enthält. Ein Trainingsschritt kann nun wie folgt ausgeführt werden (Listing 2).

Wir sehen wieder das gleiche Muster: Ein Runner wird erzeugt, die Inputs in Tensoren gepackt, das Ziel ausgewählt („*optimize*“) und die Aktion ausgeführt. Eine Neuerung haben wir jetzt allerdings: Wir erhalten Werte zurück. Die Namen der zurückzugebenden Knoten werden mit *fetch* definiert. Die Namen enthalten hier noch ein Suffix: „:0“. Das heißt, dass es sich um Knoten mit mehreren Outputs handelt, das *:0*-Suffix bedeutet, dass der Output des Knotens mit Index 0 zurückgegeben werden soll.

Der Output ist eine Liste von *Tensor*-Objekten. Die kann man in diverse Primitivtypen und Arrays umwandeln, um das Ergebnis verfügbar zu machen. Wichtig hierbei: Auch die vom API erzeugten *Tensor*-Objekte müssen geschlossen werden. Normalerweise müsste hierzu in einem *finally*-Block über die Einträge der Liste iteriert und diese geschlossen werden. Das ist allerdings sehr unhandlich und schlecht zu lesen. Deshalb ist es nützlich, ein erweitertes *use* API in Kotlin zu definieren, mit dem innerhalb eines Blocks mehrere Objekte mit *use* oder *useAll* (für Listen von *Closables*) markiert werden, die dann anschließend sicher geschlossen werden (Listing 3).

Mit diesem nützlichen Trick können nun alle Tensoren innerhalb eines TensorFlow-Aufrufs bequem und sicher geschlossen werden.

Bei der Inferenz unter Java wird es wirklich einfach. Wir erinnern uns: Jede Aktion auf dem TensorFlow-Graphen wird durchgeführt, indem Eingabeknoten mit Inputtensoren befüllt und die richtigen Ausgabeknoten abgefragt werden. Für unser Beispiel oben heißt das: Der Code bleibt gleich, es werden lediglich die Inputs für die richtige Lösung (*labels*) nicht gesetzt. Logisch, denn die kennen wir ja noch nicht. Bei der Ausgabe rufen wir die Knoten für die Fehlerberechnung und die Aktualisierung des neuronalen Netzes nicht auf (*total_loss:0*, *accuracy:0*, *optimize*), wir lernen also nicht. Stattdessen fragen wir nur das Ergebnis ab (*prediction*). Da für die Berechnung des Ergebnisses der Input der Lösungen nicht nötig ist, funktioniert alles wie bisher: Es kommt zu keinem Fehler, da der Teil des Graphen, der das neuronale Netz trainiert, inaktiv bleibt.

Praktische Erfahrungen

Das hier vorgestellte Verfahren ist nicht nur ein interessantes Experiment, der Autor hat es bereits in mehreren kommerziellen Projekten erfolgreich eingesetzt. Dabei haben sich mehrere Vorteile im praktischen Einsatz herauskristallisiert:

- Das Java API ist schnell und effizient: Es ergeben sich keine Performanceeinbußen im Vergleich zur reinen Python-Anwendung. Ganz im Gegenteil: Da Java gerade für Aufgaben wie das Einlesen von Daten und Preprocessing sehr viel schneller als Python ist, ist es sogar leichter, einen performanten Trainingsprozess zu implementieren.
- Das Training läuft über mehrere Tage absolut stabil, Googles Java-Implementierung hat sich als sehr verlässlich herausgestellt.
- Das Deployment des fertigen Produkts ist wesentlich einfacher als das von auf Python-basierten, da nur eine Java-Laufzeitumgebung und die richtigen CUDA-Treiber vorhanden sein müssen – alle Abhängigkeiten sind Teil der Java-TensorFlow-Bibliothek.
- Das Low-Level-Persistenz-API von TensorFlow (wie hier vorgestellt) ist leichter zu nutzen als viele der „offiziellen“ Verfahren wie beispielsweise Estimators.

Der einzige wirkliche Nachteil ist, dass ein Teil des Projekts weiterhin Python-basiert ist – die Definition des Graphen. Man braucht also ein Team, das zumindest zum Teil auch in der Python-Welt zu Hause ist.

Christoph Henkelmann hat in Bonn Informatik studiert und ist als technischer Geschäftsführer der DIVISIO GmbH (https://divis.io) für die Integration von KI-Verfahren in Enterprise-Projekten zuständig.

Im Porträt: Sandra Parsick, freiberufliche Softwareentwicklerin

Frauen in der Tech-Branche

In unserer Artikelserie „Women in Tech" stellen wir inspirierende Frauen vor, die erfolgreich in der IT-Branche Fuß gefasst haben. Heute im Fokus: Sandra Parsick, freiberufliche Softwareentwicklerin.

Madeleine Domogalla

Was hat dein Interesse für die Tech-Branche geweckt?

Mein Interesse an Tech besteht schon seit ich denken kann. Angefangen hat es damit, dass ich zu Hause lieber meinen Vater beim Handwerklichen helfen wollte, als meiner Mutter beim Kochen. Wenn ich bei meinen Großeltern zu Besuch war, dann habe ich auch sehr gerne meinem Opa, der Bauingenieur war, bei der Arbeit zugeschaut. Er hat mir dann auch immer kleine Mathematikaufgaben zum Knobeln gegeben.

Die Interesse zum Computer hat dann mein Cousin bei mir geweckt. Wir hatten zu Hause kein Computer, aber mein Cousin hatte einen. Wenn wir bei meiner Tante zu Besuch waren, dann habe ich den Jungs gerne beim Computerspielen zugeguckt und wollte es auch mal ausprobieren. Ich hab dann meinen Eltern lange in den Ohren gelegen, dass wir auch einen Computer brauchen. Irgendwann haben sie sich erbarmt und dann lief es klassisch für mich: Erst spielte ich viele Computerspiele, dann wurde ausprobiert was man noch so mit dem Computer machen kann.

Vorbilder und Unterstützer

Das eine Vorbild habe ich nicht, aber ich habe mich von vielen Personen inspirieren lassen. Meine Eltern, eigentlich meine ganze Familie, hat mich in dem was ich vorhatte, immer 100 Prozent unterstützt. Später im Job waren es hauptsächlich Männer, die mich gefördert haben und mir Mut zugesprochen haben. Ich denke, das lag aber auch daran, dass ich die einzige Technikerin im Team war.

Ein Tag in Sandras Leben

Ich bin als Freiberufler selbstständig und bin als Entwickler im Java-Umfeld unterwegs. Neben der normalen Softwareentwicklung, helfe ich den Firmen ihre Entwicklungsprozesse zu automatisieren. Ich habe eigentlich keinen normalen Arbeitsalltag. Es kommt auf den Auftrag an: Manchmal arbeite ich ganz normal als Entwicklerin in den Teams mit, manchmal berate ich die Teams oder ich vermittle in Schulungen und Workshop Wissen.

Für mich stand schon früh fest, dass ich was Technisches machen will. Eine Zeit lang wollte ich KFZ-Technik studieren, weil ich unbedingt Testfahrerin werden wollte. Ich hab aber schnell gemerkt, dass ich kein Talent für Physik gehabt hatte, da ich mir die Physik immer über die Mathematik erklärt habe. So hat es schnell herauskristallisiert, dass es entweder etwas mit Mathematik oder mit Informatik wird. Das hat sich auch schnell in meinen Leistungskursen für das Abitur widergespiegelt – Mathematik und Informatik. Ich hab mich dann nach dem Abitur an der Universität Bonn für Mathematik und Informatik eingeschrieben. Schnell bemerkte ich, dass mir Mathe dann doch zu abstrakt ist und ich habe mich komplett auf die Informatik konzentriert und das Studium darin auch beendet. Danach habe ich in der QA-Abteilung bei einem Start-up angefangen. Nach einem Jahr bat ich dann um eine Versetzung in die Entwicklung. Dort hatte ich Glück gehabt, dass ich an einen Senior-Entwickler geraten bin, der meinte, aus mir soll eine vernünftige Entwicklerin werden und nahm mich unter seine Fittiche.

Nach paar Jobwechsel und wo ich wieder nach neuen Herausforderungen suchte, haben mich meine Eltern darauf hingewiesen, dass sie die Uhr danach stellen können, wann ich mit meinem aktuellen Job unzufrieden bin und vielleicht sollte ich was grundlegendes ändern. Zeitgleich haben Kollegen, die als Freiberufler bei mir in den Firmen waren, mir geraten, versuch es doch mal mit der Selbstständigkeit, Nach einem Mitarbeiterjahresgespräch, das nicht zu meiner Zufriedenheit ablief, habe ich dann am nächsten Tag gekündigt und meinte okay, jetzt hast du drei Monate Zeit dich auf die Selbstständigkeit vorzubereiten. Mittlerweile bin ich länger selbstständig als ich angestellt war und zufrieden damit.

Wieso gibt es nicht mehr Frauen in der Tech-Branche?

Ich glaube, das es kulturelle Gründe. Ich weiß von Frauen, die in der DDR aufgewachsen sind, dass bei denen z. B. Informatik ein typischer Studienfach für Frauen war. Auch kann man das heutzutage noch beobachten, dass z. B. in den osteuropäischen und südamerikanischen Ländern der Frauenanteil in der IT weit höher ist als bei uns. Ich hab auch mal einen Artikel gelesen, der beschrieb, dass in den westlichen Ländern, der Frauenanteil in der Datenverarbeitung bis zur Einführung des Personal Computer auch höher war als heute.

Formell gibt es keine Hürden für Frauen, aber auf einer emotionalen Ebene. Es gibt immer noch genug Menschen, egal welchen Geschlechts, die dir als Frau irrationale Ratschläge geben, nur weil du als Frau etwas in deren Augen frauenuntypisches machen willst. Als ich meine Leistungskurse für das Abitur gewählt haben, gab es Stimmen, die meinten, ich sollte doch besser Deutsch nehmen, obwohl meine Note gerade für die Kombination Mathematik und Informatik sprachen.

Als ich mich selbstständig machen wollte, gab es Stimmen, die das Ganze für ein Hirngespinst hielten und mich auch fragten, wann ich mir endlich wieder einen vernünftigen Job suche. Ich hab mich davon nicht beirren lassen, aber ich kann mir gut vorstellen, das anderen Frauen sich davon beeinflusst lassen und dann diesen Ratschlägen folgen, obwohl sie total irrational sind.

Frauen in MINT-Fächern

Viele sind erstaunt, wenn sie hören, dass ich als Entwicklerin viel mit Leuten reden muss. Daher versuche ich Schülerinnen zu motivieren, wenn sie was „mit Menschen“ machen wollen, sich ein IT-Beruf anzuschauen, da man heutzutage dort viel mit Menschen zu tun hat. Viele Produkte würden nicht so sehr auf Männer ausgelegt werden. Ein Beispiel sind die Gesundheits-Apps, die lange Zeit keine Funktionen hatten wie z. B. Menstruationskalender.

Vielleicht müssen wir auch die IT-Berufe mehr als soziale Berufe interpretieren. Neue technische Innovationen haben Einfluss auf die Gesellschaft und die Gesellschaft hat Einfluss auf die Technik. Ein weiterer Aspekt ist, wenn die Teams durchmischter sind, dann gehen die Menschen sozialer miteinander um.

Ich glaube, wir werden immer eine Diversity-Debatte haben. Diversity macht ja nicht bei den Geschlechtern halt.

Hindernisse

Hürden und Hindernisse gab es bestimmt. Doch ich schlug dann halt einen anderen Weg ein oder bin über die Steine gegangen, um an mein Ziel zu kommen bzw. es ergaben sich andere, neue und zum Teil spannendere Perspektiven. Wenn Menschen zusammen arbeiten, dann menschelt es halt. Ich hab es nicht auf mein Geschlecht bezogen.

Klischees gibt es sicherlich. Wenn Leute erfahren, dass ich in der IT-Branche arbeite, dann wird meistens davon ausgegangen, dass ich im Projektmanagement tätig bin. Wenn ich sie aufkläre, dass ich entwickle, gehen sie immer davon aus, dass ich im Frontend-Bereich entwickle. Ich persönlich hab damit kein Problem. Ich kläre die Leute auf und dann ist gut.

Tipps & Tricks

Wenn man Interesse an Technik hat, dann soll man sich nicht von irrationalen Ratschlägen beirren lassen und einfach machen. Auch wenn man kein Interesse an Technik hat und eher was Soziales machen will, ist die Tech-Branche dafür gut geeignet, da wir komplexe Probleme lösen müssen, die sich nur in Teams mit verschiedenen Hintergründen lösen lassen.

Generell kann ich allen nur raten, macht das woran ihr Interesse habt und wenn ihr zu viele Zweifler um euch habt, dann sucht euch Leute, die euch unterstützen.

Sandra Parsick ist als freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld tätig. Seit 2008 beschäftigt sie sich mit agiler Softwareentwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen in den Bereichen Java-Enterprise-Anwendungen, agile Methoden, Software Craftsmanship und der Automatisierung von Softwareentwicklungsprozessen. In ihrer Freizeit engagiert sie sich in der Softwerkskammer Ruhrgebiet.