

JavamagazinTM

Java | Architektur | Software-Innovation

java 14

Born to write Code



Ausgabe 5.2020

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15



Anzeige



Get your motor runnin'

Die Freiheit, so sagt mancher, hat man nur erlebt, wenn man einmal mit der Harley die Vereinigten Staaten durchquert hat – nach Möglichkeit auf der legendären Route 66. Im Ohr natürlich den Titelsong aller, die es lieben, sich auf das Ross aus Schwermetall zu schwingen und in die Abendsonne zu brausen. Doch betrachtet man die Textzeilen der Bikerhymne ganz genau, kommt man auch als Java-Entwickler auf seine Kosten:

Get your motor runnin'
Head out on the highway
Looking for adventure
And whatever comes our way

Das bedeutet ja wohl nichts anderes, als dass man seine JVM anschmeißt und auf der Datenautobahn nach Abenteuern sucht. Meist haben diese Abenteuer vermutlich die Form von Bugs, Problemen, die sich nicht so einfach mit Standard-Java-Code lösen lassen, und Versionsupdates für unsere liebste Programmiersprache. Gerade Letztere sorgen spätestens seit der Einführung des Modulsystems in Java 9 (Codename: Jigsaw) für größere Schwierigkeiten als jeder Krieg zwischen zwei Motorradgangs – die Folge: Java 8 ist bis heute die am weitesten verbreitete Version, obwohl es schon seit rund sechs Jahren auf dem Markt ist. Gerade im Unternehmensbereich gibt es viele Bergaufbremser, wenn es um Updates auf neue Versionen geht.

Mit Java 14 steht nun das nächste Modell im Schaufenster und „Evel Knievel“ Sippach hat es vom Lenker bis zum Rücklicht getestet. Besonderes Augenmerk auf die Type Records hat Tim „Mainz Angel“ Zöller gelegt, worüber er einen ausführlichen Erfahrungsbericht schreibt. Natürlich wollten wir auch vom „MC Java Experts“ eine Meinung haben, weshalb diesmal auch ein Interview Teil des Schwerpunkts ist. Abgerundet wird das Ganze durch unsere obligatorische Infografik zum Release.

Wer sich lieber mit der Verkleidung als mit den inneren Werten seines Java-Bikes beschäftigt, den nimmt Manfred „Austrian Outlaw“ Steyer mit auf eine faszinierende Reise ins Angular-Universum. Auch dort gab es mit Angular 9 vor Kurzem eine Veröffentlichung, die es in sich hatte und unter anderem den neuen Ivy-Compiler bereithält. Doch natürlich darf man die Sicherheit nicht vergessen, Stichwort: Helmpflicht. SysCall-FILTER stellen dabei im Container- und Kubernetes-Umfeld einen sehr guten Schutz dar, wie Andreas „The Kube“ Jaekel in seinem Artikel feststellt.

Wie Sie sehen, liebe Bikerinnen und Biker, auch unsere große Themenvielfalt bietet in diesem Monat wieder die Freiheit, sich auf dem Highway to Java auszutoben. Vielleicht eine gute Alternative dazu, tatsächlich vor die Tür zu gehen, was dieser Tage ohnehin keine gute Idee zu sein scheint. Bleiben Sie daheim und denken Sie immer daran: Sie sind born to write Code.

In diesem Sinne: Immer eine Handbreit Luft unter dem Vorderrad!

Dominik Mohilo | Redakteur



@JavaMagazin



10 – 32 | Java 14

© Anjar Sumawi/Shutterstock.com
© Morevector/Shutterstock.com

„Languages must evolve, or they risk becoming irrelevant“, sagte Brian Goetz (Oracle) im November 2019 während seiner Präsentation „Java Language Futures“ bei der Devoxx in Belgien. Er ist als Java Language Architect maßgeblich daran beteiligt, dass Java trotz seiner 25 Jahre noch lange nicht zum alten Eisen gehört. In diesem Artikel werfen wir einen Blick auf die Neuerungen des JDK 14.



58 | Elasticsearch auf Kubernetes

© akud/Shutterstock.com

Vor nicht langer Zeit war es undenkbar, persistente Daten auf Kubernetes zu stellen. Kluge Ops-Teams hätten eher auf einen vergleichbaren Cloud-Service gesetzt und die Funktionalität für Simplizität und ein ruhiges Gewissen geopfert. In den letzten Jahren ist Kubernetes aber gereift und in neue Bereiche vorgestoßen.



70 | Java neuer HTTP-Client

© fasseodesignen/Shutterstock.com

Es ist enorm wichtig, dass eine so populäre Programmiersprache wie Java eine zeitgemäße Unterstützung für den Versand von HTTP Requests mitbringt. Überraschenderweise war das sehr lange Zeit nicht der Fall.



81 | Angular 9

© ECHOY/shutterstock.com

Angular 9 bringt Ivy in einer abwärtskompatiblen Variante, damit kleinere Bundles. Die i18n-Lösung wurde umfangreich überarbeitet und einige Ecken abgerundet. So stehen dem Entwickler und zukünftigen Versionen von Angular neue Möglichkeiten zur Verfügung.



88 | Containersicherheit

© Makstrorm/Shutterstock.com

Container und Plattformen wie Kubernetes sind mittlerweile ein beliebtes Angriffsziel von Hackern. In diesem Artikel zeigt Andreas Jaekel, Head of PaaS Development bei IONOS, wie man seine Container absichern kann, denn von Natur aus sicher sind sie nicht.

Magazin

6 News

Titelthema

10 Java – die Vierzehnte

Die Neuerungen der aktuellen Version auf einen Blick
Falk Sippach

18 Java 14: Diese JEPs sind Teil des JDks

Infografik

24 Rekordverdächtig strukturiert

Ein Blick auf JEP 359: Java Records
Tim Zöller

28 Die Java-Elefantenrunde

Unter der Lupe: Java 14

Enterprise

34 Kolumne: EnterpriseTales

Cloud-Computing is someone else's computer
Lars Kölpin

Architektur

38 Qualität ist besser!

Nichtfunktionale Anforderungen: Struktur, Organisation und Wartung
Marco Schulz

44 Algorithmen in der Java-Praxis

Teil 2: Sortier- und Suchalgorithmen – ein Überblick
Dr. Veikko Krypczyk und Elena Bochkor

Data

54 Kampf den schlechten Gewohnheiten

Code in Machine-Learning-Modellen: Komplexität vermeiden
David Tan

58 ECK macht es einfach

Natives Elasticsearch auf Kubernetes
Dimitri Marx

Cloud Computing

66 Migration nach AWS

Teil 5: Verschlüsselung für Java-Entwickler
Steffen Grunwald

Web

70 Was lange währt...

Javas neuer HTTP-Client
Thilo Frotscher

76 Bessere Integrationstests mit WireMock

Die Anbindung an externe HTTP Services richtig testen
Ronny Bräunlich

81 Was bringt Angular 9?

Ivy voll integriert, Lazy-Loading-Komponenten und Lokalisierung
von Manfred Steyer

DevOps

86 Für das Plus an Produktivität und Effizienz...

Coden in Go
Dewet Diener

Security

88 Sicherheit geht vor – auch bei Containern

Container, Docker & Kubernetes: mehr Sicherheit mit SysCall-Filtern
Andreas Jaekel

Internet of Things

92 Amazon IoT für Java

Device Shadows/Device Twins
Tam Hanna

Standards

3 Editorial

7 JUG-Kalender

98 Impressum, Inserentenverzeichnis, Vorschau, Empfehlungen



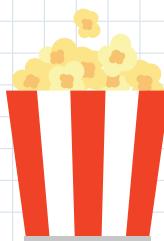
Leserbriefe und Feedback zu den Artikeln des Java Magazins bitte an redaktion@javamagazin.de.

@nnnotations

TWEET DES MONATS

If you ever code something that “feels like a hack but it works”, just remember that a CPU is literally a rock that we tricked into thinking.

@daisyowl



TV-TIPP

Die Themen Quarkus und GraalVM haben in den letzten Monaten für einiges Aufsehen in der Java-Welt gesorgt. In seiner Session auf der W-JAX 2019 in München zeigt Peter Palaga, Software-Engineer für JBoss Fuse bei Red Hat, wie Quarkus Laufzeitkosten eliminiert und wie es dafür die Vorteile der GraalVM einsetzt. <https://tinyurl.com/w5wwqn8>



Karen Hoyos
Backend-Software-Engineer
im Babbel-Payment-Team
<https://tinyurl.com/rsnodoj>

“Es gab eine Menge überraschte Gesichter, als ich über meinen Beruf sprach und Leute annahmen, dass ich eine Frontend-Entwicklerin sei, nur weil [redacted] ich eine Frau bin und man Frauen für kreativer hält.”

Frisch von JAXenter

JEP 372: Entfernen der JavaScript-Engine Nashorn
<https://tinyurl.com/wpuavjb>

Für eine polyglotte Zukunft: **GraalVM Project Advisory Board** gegründet
<https://tinyurl.com/r68ruo4>

Apache NetBeans 11.3 liefert Support für JDK 14
<https://tinyurl.com/v66j8db>

JEP 374: Das Ende von Biased Locking
<https://tinyurl.com/r4s9pej>



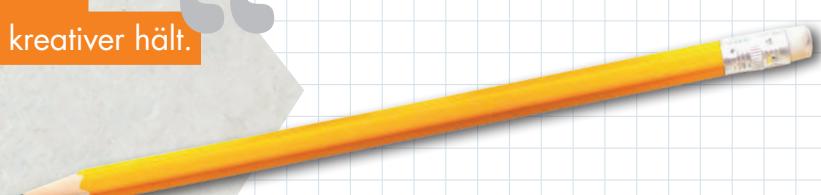
LET'S TALK!



Für viele schon Teil des Alltags, für andere noch ein etwas wolkiges Thema: Serverless. Im Zuge unseres aktuellen Schwerpunkts haben wir mit zwei Experten zum Thema gesprochen:

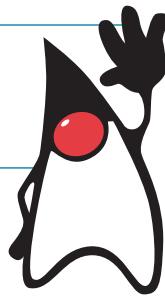


CHRISTIAN BANNES, Lead Developer bei der ip.labs GmbH, und **VADYM KAZULKIN**, Head of Technology Strategy, ebenfalls bei der ip.labs GmbH.
<https://bit.ly/34kXOZT>



JUG-Kalender

Neues aus den User Groups



WER?	WAS?	WO?
JUG Berlin-Brandenburg	07.04.2020: Monatlicher Stammtisch der JUG Berlin-Brandenburg	https://tinyurl.com/rrq3wzp
JUG Augsburg	14.04.2020: Self-Driving Car: Computer Vision und Machine Learning (Intro)	https://tinyurl.com/wwns8cn
JUG Ingolstadt	14.04.2020: Java-Treff Elasticsearch Testing & Security	https://tinyurl.com/u3d4k3l
JUG Saarland	16.04.2020: Integrationstests mit Docker und Testcontainer – Kevin Witteck	https://tinyurl.com/vktk5lx
JSUG Austria	20.04.2020: April Meetup: All about Quarkus!	https://tinyurl.com/vzlx8lr
JUG Switzerland	21.04.2020: Quarkus	https://tinyurl.com/y5t3vhuz
JUG Stuttgart	22.04.2020: Haskell – 7 Languages in 7 Months	https://tinyurl.com/rcaf7da
JUG Darmstadt	23.04.2020: JUG DA: Kubernetes Native Spring Applications with Quarkus (Markus Eisele)	https://tinyurl.com/qwn8fc2
JUG Saxony	23.04.2020: Infrastructure as Microservices – Alternativen zum Monolithen Kubernetes	https://tinyurl.com/qs3wa5m
JUG Hannover	24.04.2020: Java Stammtisch	https://tinyurl.com/ua4mrw2
JUG Augsburg	28.04.2020: Neuronale Netze in der Praxis – Integration in Java Webservices	https://tinyurl.com/v7q8yf6
JUG Frankfurt	29.04.2020: Up Next – Kotlin! (Sebastian Aigner)	https://tinyurl.com/wyk6oml
JUG Görlitz	29.04.2020: JDK 14 und GraalVM im Java Ökosystem (Wolfgang Weigend)	https://tinyurl.com/szljqqa4

Alle Angaben ohne Gewähr. Da Termine sich kurzfristig ändern können, überprüfen Sie diese bitte auf der jeweiligen JUG-Website.



Java ist nicht nur eine Insel, sondern auch ein weites Feld – dementsprechend steht das Java Magazin allen kreativen Köpfen offen, die dieses Feld gemeinsam mit uns beackern wollen. Habt Ihr ein spannendes Thema in petto, das einen technischen Mehrwert für unsere Leser bietet, auf werbliche Elemente verzichtet und die breite Java-Community anspricht, dann seid Ihr bei uns herzlich willkommen. Also: Keine Angst vor der eigenen Courage und einfach einen kurzen Abstract an redaktion@javamagazin.de schicken.



Beginning Jakarta EE

von Peter Späth

J2EE – neuerdings Jakarta – war in den letzten Jahren vor allem aufgrund der diversen Namens- und Eigentümeränderungen im Gespräch. Im Apress-Verlag erscheint nun ein Lehrbuch von Peter Späth, das angehenden Entwicklern zu einer holistischen Sichtweise verhelfen soll.

Das erste Kapitel beginnt mit der Vorstellung der Industriestandards, die die von Jakarta propagierte Interoperabilität gewährleisten. Im nächsten Schritt erklärt der Autor Themen wie Client-Server-Topologie und Microservice, um danach auf den GlassFish-Server überzugehen. Vorteilhaft ist an dieser Vorgehensweise, dass alle Komponenten kostenlos zur Verfügung stehen. Das zweite Kapitel expliziert die Konfiguration des Servers, während das dritte Kapitel die Einrichtung von Eclipse für die J2EE-Entwicklung demonstriert. An dieser Stelle beschreibt der Autor die Umsetzung erster Programmbeispiele, die das Projektelement und die diversen Bibliotheken der Umgebung vorstellen.

Im vierten Kapitel folgt ein praktisches Beispiel: Späth programmiert eine auf JSF basierende Miniapplikation. Im Backend setzt er übrigens ausschließlich auf CDI. Er erklärt

die früher gern verwendeten Managed Beans für veraltet und behandelt sie nicht weiter. Die Erklärung der Templatesprache und des Data Bindings fällt hervorragend aus. So interessant die Experimente mit JSF auch sind, so selten braucht man sie in der Praxis. Muss Schlomo Nor malentwickler eine Webapplikation realisieren, denkt er so gut wie immer an eine SPA.

Das fünfte Kapitel beginnt mit einer Kurzvorstellung eines Minimalbeispiels, um danach auf REST, HTTP-Verben und das JSON-Austauschformat einzugehen. Auch hier entsteht nebenbei ein kleines Beispiel, in dem der angehende Entwickler die Konzepte live aufgeführt bekommt. Dabei zeigt der Autor übrigens keinerlei Scheu vor Codesamples. Stellenweise findet sich schon einmal ein halbseitiges Listing, das dank der sauberen Formatierung allerdings problemlos lesbar bleibt.

Wer Jakarta als reines HTTP Framework verwendet, verliert einen Gutteil der fortgeschrittenen Funktionen. Das sechste Kapitel beginnt mit der Vorstellung des JPA, das den Datenbankzugriff ermöglicht. Lobenswert ist hier, dass Späth alles „zu Fuß“ realisiert: Die

in Eclipse enthaltene Datenbank-Engine wird zwar in einem Kasten erwähnt, kommt aber im Interesse der Vendor-Unabhängigkeit im Rest des Buchs nicht mehr zur Sprache.

Das darauffolgende Kapitel zu Enterprise Java Beans stellt eine weitere Art der Vorhaltung von Backend-Logik vor. Im darauffolgenden Abschnitt dreht sich alles um das Parsen von XML. Der Autor stellt fortgeschrittene Szenarien vor, bei denen Nichtkenner wegen diverser Tricks und Kniffe schon mal einige Stunden verlieren können. Das ist kein Selbstzweck, da zwei Kapitel zum Messaging unter Verwendung von JMS und zur Zustands-Engine JTA folgen.

Viele Java-Lehrbücher lassen den Entwickler liegen, nachdem die realisierte Applikation in der IDE läuft. Späth erklärt auch Enterprise-lastige Themen wie die Absicherung einer Jakarta-Applikation oder ihre Auslieferung in Form von Artefakten. Die im allgemeinen leidigen Themen des Loggings und des Monitorings reißt das Werk ebenfalls an, um dem angehenden Administrator ein Sammelsurium an Basistechnologien an die Hand zu geben. Danach folgt noch ein Appendix zur HTML-Templatesprache und die Lösungen der in den diversen Kapiteln platzierten Möglichkeiten zur Selbstkontrolle.

Wer sich schon immer in die schöne neue Welt von Jakarta einarbeiten wollte, findet mit „Beginning Jakarta EE“ ein kompaktes und didaktisch sauberes Werk. Lassen Sie sich von der englischen Sprache nicht abschrecken – der Text ist leicht verständlich.

Tam Hanna



Peter Späth

Beginning Jakarta EE

Enterprise Edition for Java: From Novice to Professional

444 Seiten, 29,95 Euro

Apress, 2019

ISBN: 978-3-446-45466-8

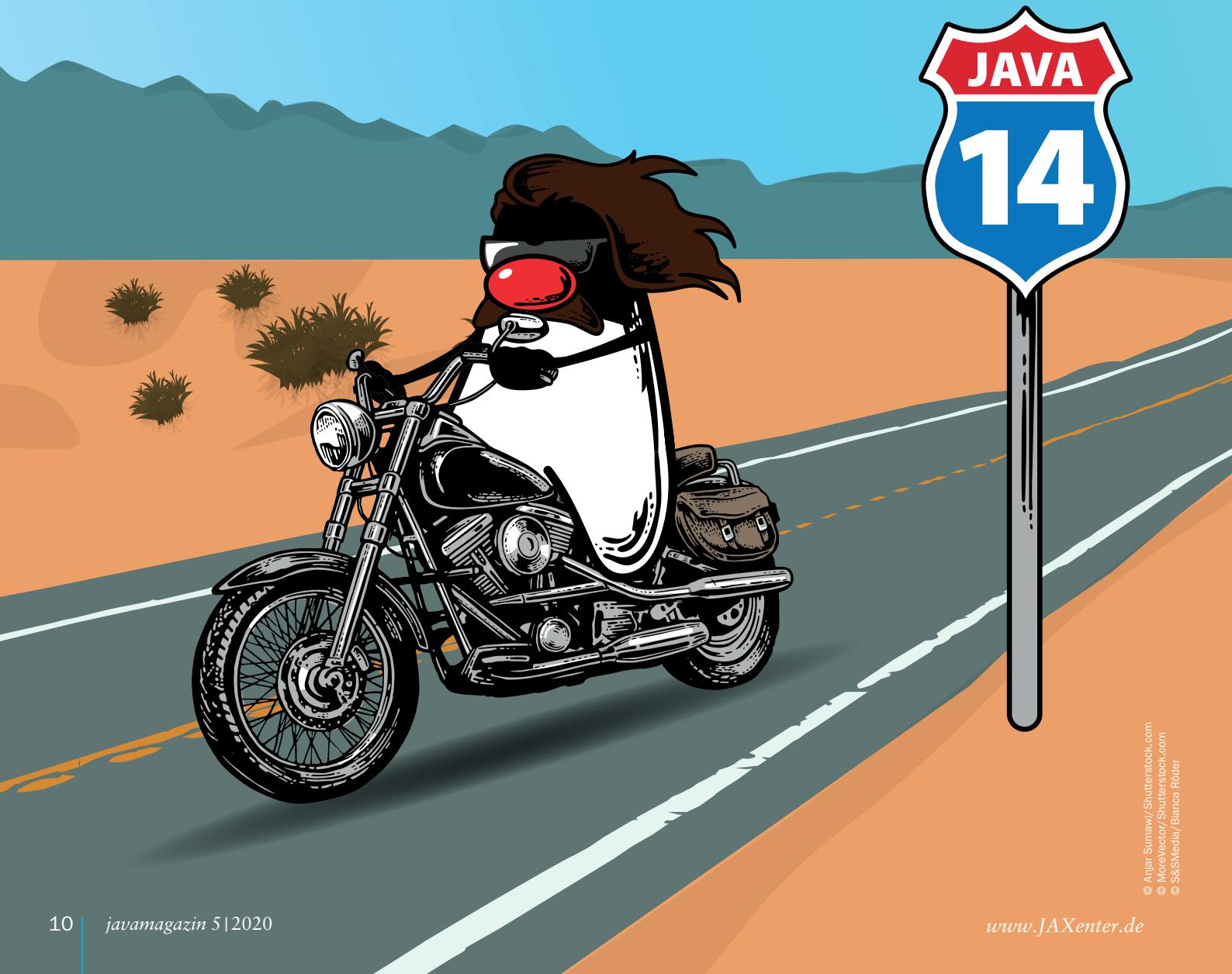
Anzeige

Die Neuerungen der aktuellen Version auf einen Blick

Java - die Vierzehnte

„Languages must evolve, or they risk becoming irrelevant“, sagte Brian Goetz (Oracle) im November 2019 während seiner Präsentation „Java Language Futures“ bei der Devoxx in Belgien. Er ist als Java Language Architect maßgeblich daran beteiligt, dass Java trotz seiner 25 Jahre noch lange nicht zum alten Eisen gehört. In diesem Artikel werfen wir einen Blick auf die Neuerungen des JDK 14.

von Falk Sippach



Oracle hat in den vergangenen Jahren einige wegweisende Entscheidungen getroffen. Dazu zählt das neue, halbjährliche Releasemodell mit den Preview-Features und den kürzeren Veröffentlichungs- und Feedbackzyklen für neue Funktionen. Das Lizenzmodell wurde geändert, das Oracle JDK wird nicht mehr kostenfrei angeboten. Das hat den Wettbewerb angekurbelt, und so bekommt man nun von diversen Anbietern, auch von Oracle, freie Distributionen des OpenJDK. Das ist seit Java 11 binärkompatibel zum Oracle JDK und steht unter einer Open-Source-Lizenz.

Vor anderthalb Jahren ist im Herbst 2019 mit Java 11 die letzte LTS-Version erschienen. Seitdem gab es bei den beiden folgenden Major-Releases jeweils nur eine überschaubare Menge an neuen Features. In den JDK-Inkubator-Projekten (Amber, Valhalla, Loom, ...) wird aber bereits an vielen neuen Ideen gearbeitet, und so verwundert es nicht, dass der Funktionsumfang beim gerade veröffentlichten JDK 14 wieder deutlich größer ausfällt. Und auch wenn nur wenige die neue Version in Produktion einsetzen werden, sollte man trotzdem frühzeitig einen Blick auf die Neuerungen werfen und ggf. Feedback zu den Preview-Funktionen geben. Nur so wird sichergestellt, dass die neuen Features bis zur Finalisierung im nächsten LTS-Release, das als Java 17 im Herbst 2021 erscheinen wird, produktionsreif sind.

Die folgenden Java Enhancement Proposals (JEP) wurden umgesetzt [1]. Wir wollen in diesem Artikel die aus Entwicklersicht interessanten Themen genauer unter die Lupe nehmen.

- JEP 305: Pattern Matching for instanceof (Preview)
- JEP 343: Packaging Tool (Incubator)
- JEP 345: NUMA-Aware Memory Allocation for G1
- JEP 349: JFR Event Streaming
- JEP 352: Non-Volatile Mapped Byte Buffers
- JEP 358: Helpful NullPointerExceptions
- JEP 359: Records (Preview)
- JEP 361: Switch Expressions (Standard)
- JEP 362: Deprecate the Solaris and SPARC Ports
- JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector
- JEP 364: ZGC on macOS
- JEP 365: ZGC on Windows
- JEP 366: Deprecate the ParallelScavenge + SerialOld GC Combination
- JEP 367: Remove the Pack200 Tools and API
- JEP 368: Text Blocks (Second Preview)
- JEP 370: Foreign-Memory Access API (Incubator)

JEP 305: Pattern Matching für instanceof

Das Pattern-Matching-Konzept kommt bereits seit den 1960er Jahren bei diversen Programmiersprachen zum Einsatz. Zu den moderneren Vertretern zählen unter anderem Haskell und Scala. Ein Pattern ist eine Kombination aus einem Prädikat, das auf eine Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern

Durch die Switch Expressions stehen den Entwicklern nun zwei neue Syntaxvarianten mit einer kürzeren, klareren und weniger fehleranfälligen Syntax zur Verfügung.

die entsprechenden Inhalte zugewiesen. Die Intention ist die Destrukturierung von Objekten, also das Aufspalten in ihre Bestandteile.

Bisher konnte man in Java im Switch Statement nur nach den Datentypen *Integer*, *String* und *Enum* unterscheiden. Durch die Einführung der Switch Expression in Java 12 wurde aber bereits der erste Schritt hin zum Pattern Matching vollzogen. Mit Java 14 können wir nun zusätzlich Pattern Matching beim *instanceof*-Operator nutzen. Dabei werden unnötige Casts vermieden, zudem erhöht sich durch die verringerte Redundanz die Lesbarkeit.

Vorher musste man beispielsweise für das Prüfen auf einen leeren String bzw. eine leere Collection wie folgt vorgehen:

```
boolean isEmpty( Object o ) {
    return null ||
    instanceof String && ((String) o).isEmpty() ||
    instanceof Collection && ((Collection) o).isEmpty();
}
```

Jetzt kann man beim Check mit *instanceof* den Wert direkt einer Variablen zuweisen und darauf weitere Aufrufe ausführen:

```
boolean isEmpty( Object o ) {
    return o == null ||
    o instanceof String s && s.isEmpty() ||
    o instanceof Collection c && c.isEmpty();
}
```

Der Unterschied mag marginal erscheinen. Die Puristen unter den Java-Entwicklern sparen damit allerdings eine kleine, aber dennoch lästige Redundanz ein.

Die Switch Expressions hatte man zunächst in Java 12 und 13 jeweils als Preview-Feature eingeführt. Sie wurden nun im JEP 361 finalisiert. Dadurch stehen den Entwicklern zwei neue Syntaxvarianten mit einer kürzeren, klareren und weniger fehleranfälligen Semantik zur Verfügung. Das Ergebnis der Expression kann einer Variablen zugewiesen oder als Wert aus einer Methode zurückgegeben werden (Listing 1). Weitere Details können dem Artikel zu Java 13 [2] entnommen werden.

JEP 358: Helpful NullPointerExceptions

Der unbeabsichtigte Zugriff auf leere Referenzen ist auch bei Java-Entwicklern gefürchtet. Nach eigener Aussage von Sir Tony Hoare war seine Erfindung der Nullreferenz ein Fehler mit Folgen in Höhe von vielen Milliarden Dollar. Und das nur, weil es bei der Entwicklung der Sprache Algol in den 60er Jahren einfach so leicht zu implementieren war.

In Java gibt es weder vom Compiler noch von der Laufzeitumgebung Unterstützung beim Umgang mit Nullreferenzen. Mit diversen Workarounds lassen sich diese leidigen Exceptions vermeiden. Der einfachste Weg stellt die Prüfungen auf null dar. Leider ist dieses Vorgehen sehr mühselig und wird immer genau dann vergessen, wenn man den Check gebraucht hätte. Mit der seit dem JDK 8 enthaltenen Wrapper-Klasse *Optional* kann man über das API explizit den Aufrufer darauf hinweisen, dass ein Wert null sein kann und er darauf reagieren muss. Somit kann man nicht mehr aus Versehen in eine Nullreferenz hineinlaufen, sondern muss

Listing 1

```
String developerRating( int numberOfChildren ) {
    return switch (numberOfChildren) {
        case 0 -> "open source contributor";
        case 1, 2 -> "junior";
        case 3 -> "senior";
        default -> {
            if (numberOfChildren < 0)
                throw new IndexOutOfBoundsException( numberOfChildren );
            yield "manager";
        }
    };
}
```

Listing 2

```
Stream.of( man, woman )
    .map( p -> p.partner() )
    .map( p -> p.name() )
    .collect( Collectors.toUnmodifiableList() );

Result: java.lang.NullPointerException: Cannot invoke "Person.name()"
because "<parameter1>" is null
```

Listing 3

```
int calculate() {
    Integer a = 2, b = 4, x = null;
    return a + b * x;
}
calculate();

Result: java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()"
because "x" is null
```

explizit mit dem möglicherweise leeren Wert umgehen. Dieses Vorgehen bietet sich unter anderem bei Rückgabewerten von öffentlichen Schnittstellen an, kostet aber auch eine extra Indirektionsschicht, da man den eigentlichen Wert immer auspacken muss.

In anderen Sprachen wurden längst Hilfsmittel in Syntax und Compiler eingebaut, wie zum Beispiel bei Groovy das *NullObjectPattern* und der Safe Navigation Operator (*some?.method()*). Bei Kotlin kann man explizit zwischen Typen, die nicht leer sein dürfen und solchen, bei deren Referenz auch null erlaubt ist, unterscheiden. Mit den NullPointerExceptions werden wir in Java auch künftig leben müssen. Aber immerhin erleichtern uns die als Preview-Feature eingeführten Helpful NullPointerExceptions nun die Fehlersuche im Ausnahmefall. Damit beim Werfen einer NullPointerException die notwendigen Informationen eingefügt werden, muss man beim Starten die Option *-XX:+ShowCodeDetails-InExceptionMessages* aktivieren. Ist dann in einer Aufrufkette ein Wert null, bekommt man eine aussagekräftige Meldung:

```
man.partner().name()
```

```
Result: java.lang.NullPointerException: Cannot invoke "Person.name()"
because the return value of "Person.partner()" is null
```

Bei Lambdaausdrücken braucht es eine Spezialbehandlung. Ist zum Beispiel der Parameter einer Lambdafunction null, bekommt man standardmäßig die in Listing 2 gezeigte, unzureichende Fehlermeldung. Damit der korrekte Parametername angezeigt wird, muss der Quellcode mit der Option *-g:vars* kompiliert werden. Das Resultat:

```
java.lang.NullPointerException: Cannot invoke "Person.name()"
because "p" is null
```

Bei Methodenreferenzen gibt es aktuell leider noch keinen Hinweis im Fall eines leeren Parameters:

```
Stream.of( man, woman )
    .map( Person::partner )
    .map( Person::name )
    .collect( Collectors.toUnmodifiableList() )

Result: java.lang.NullPointerException
```

Setzt man aber wie hier im Beispiel jeden *Stream*-Methodenaufruf in eine neue Zeile, lässt sich die problematische Codezeile sehr schnell eingrenzen. Herausfordernd waren NullPointerExceptions bisher auch beim automatischen Boxing/Unboxing. Wird auch hier der Compilerparameter *-g:vars* aktiviert, bekommt man ebenfalls die neue hilfreiche Fehlermeldung (Listing 3).

JEP 359: Records

Die wahrscheinlich spannendste und gleichzeitig auch überraschendste Neuerung dürfte die Einführung der

Anzeige

Record-Typen sein. Sie wurden noch relativ spät in das Release von Java 14 aufgenommen. Dabei handelt es sich um eine eingeschränkte Form der Klassendeklaration, ähnlich den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Die kompakte Syntax könnte Bibliotheken wie Lombok in Zukunft obsolet machen. Kevlin Henney sieht außerdem noch folgenden Vorteil [3]: „I think one of the interesting side effects of the Java record feature is that, in practice, it will help expose how much Java code really is getter/setter-oriented rather than object-oriented.“ Die einfache Definition einer Person mit zwei Feldern sieht man hier:

```
public record Person( String name, Person partner ) {}
```

Eine erweiterte Variante mit einem zusätzlichen Konstruktor, damit nur das Feld *name* Pflicht ist, lässt sich ebenfalls realisieren:

```
public record Person( String name, Person partner ) {
    public Person( String name ) { this( name, null ); }
    public String getNameInUppercase() { return name.toUpperCase(); }
}
```

Erzeugt wird vom Compiler eine unveränderbare (immutable) Klasse, die neben den beiden Attributen und

Listing 4

```
public final class Person extends Record {
    private final String name;
    private final Person partner;

    public Person(String name) { this(name, null); }
    public Person(String name, Person partner) { this.name = name; this.partner = partner; }

    public String getNameInUppercase() { return name.toUpperCase(); }
    public String toString() { /* ... */ }
    public final int hashCode() { /* ... */ }
    public final boolean equals(Object o) { /* ... */ }
    public String name() { return name; }
    public Person partner() { return partner; }
}
```

Listing 5

```
var man = new Person("Adam");
var woman = new Person("Eve", man);
woman.toString(); // ==> "Person[name=Eve, partner=Person[name=Adam, partner=null]]"

woman.partner().name(); // ==> "Adam"
woman.getNameInUppercase(); // ==> "EVE"

// Deep equals
new Person("Eve", new Person("Adam")).equals( woman ); // ==> true
```

den eigenen Methoden natürlich auch noch die Implementierungen für die Accessoren (keine Getter!), den Konstruktor sowie *equals/hashcode* und *toString* enthält (Listing 4).

Die Verwendung gestaltet sich erwartungsgemäß. Man sieht dem Aufrufer nicht an, dass Record-Typen instanziiert werden (Listing 5).

Records sind übrigens keine klassischen Java Beans, da sie keine echten Getter enthalten. Man kann aber über die gleichnamigen Methoden auf die Membervariablen zugreifen. Records können im Übrigen auch Annotationen oder Javadocs enthalten. Im Body dürfen zudem statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Nicht erlaubt ist die Definition von weiteren Instanzfeldern außerhalb des Record-Headers. Weitere Details dazu finden sich im Artikel von Tim Zöller in dieser Ausgabe.

JEP 368: Text Blocks

Ursprünglich als Raw String Literals bereits für Java 12 geplant, hat man dann in Java 13 zunächst eine abgespeckte Variante in Form von mehrzeiligen Strings namens Text Blocks eingeführt. Insbesondere für HTML-Templates und SQL-Skripte erhöht sich dadurch die Lesbarkeit enorm (Listing 6).

Neu hinzugekommen sind jetzt zwei Escape-Sequenzen, mit denen man die Formatierung eines Text Blocks anpassen kann. Um zum Beispiel einen Zeilenumbruch zu verwenden, der aber nicht explizit in der Ausgabe erscheinen soll, kann man am Zeilenende einfach einen \ (Backslash) einfügen. Dadurch bekommt man einen String mit einer langen Zeile, im Quellcode dürfen aber für die Übersichtlichkeit Umbrüche verwendet werden (Listing 7).

Die neue Escape-Sequenz \s wird zu einem Leerzeichen übersetzt. Dadurch kann man beispielsweise erreichen, dass Leerzeichen am Zeilenende nicht automatisch abgeschnitten (getrimmt) werden und man eine feste Zeichenbreite je Zeile erhält:

```
String colors = """
    red \s
    green\s
    blue \s
""";
```

Was gibt es noch Neues?

Neben den beschriebenen Features, die hauptsächlich für Entwickler interessant sind, gibt es diverse andere Änderungen. Im JEP 352 wurde das *FileChannel* API erweitert, um die Erzeugung von *MappedByteBuffer*-Instanzen zu ermöglichen. Die arbeiten, im Gegensatz zum volatilen Speicher, dem RAM, auf nichtflüchtigen Datenspeichern (NVM, non-volatile Memory). Die Zielplattform ist allerdings Linux x64. Auch bei der Garbage Collection hat sich einiges getan. So wurde der Concurrent Mark Sweep (CMS) Garbage Collector entfernt. Dafür gibt es den ZGC jetzt auch für macOS und Windows.

Bei kritischen Java-Anwendung wird empfohlen, die Flight-Recording-Funktion in Produktion zu aktivieren. Der folgende Befehl startet eine Java-Anwendung mit Flight Recording und schreibt die Informationen in die `recording.jfr`, wobei immer die Daten eines Tages aufgehoben werden:

```
java \
-XX:+FlightRecorder \
-XX:StartFlightRecording=disk=true, \
filename=recording.jfr,dumponexit=true,maxage=1d \
-jar application.jar
```

Listing 6

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

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

Listing 7

```
String text = """
Lorem ipsum dolor sit amet, consectetur adipiscing \
elit, sed do eiusmod tempor incididunt ut labore \
et dolore magna aliqua.\n""";
// statt
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
    "elit, sed do eiusmod tempor incididunt ut labore " +
    "et dolore magna aliqua.;"
```

Listing 8

```
import jdk.jfr.consumer.RecordingStream;
import java.time.Duration;

try ( var rs = new RecordingStream() ) {
    rs.enable( "jdk.CPULoad" ).withPeriod( Duration.ofSeconds( 1 ) );
    rs.onEvent( "jdk.CPULoad", event -> {
        System.out.printf( "%.1f %% %n", event.getFloat( "machineTotal" )
            * 100 );
    });
    rs.start();
}
```

Mit jpackage, dem in Java 14 eingeführten Nachfolger von javapackager, können nun wieder eigenständige Java-Installationsdateien erstellt werden.

Normalerweise kann man die Daten dann mit dem Tool JDK Mission Control (JMC) auslesen und analysieren. Neu im JDK 14 ist, dass man auch aus der Anwendung heraus asynchron die Events abfragen kann (Listing 8).

Im JDK 8 gab es das Tool javapackager, das aber leider mitsamt JavaFX in Version 11 aus Java entfernt wurde. In Java 14 wird nun der Nachfolger jpackage eingeführt (JEP 343: Packaging Tool), mit dem wieder eigenständige Java-Installationsdateien erstellt werden können. Ihre Basis ist die Java-Anwendung mitsamt einer Laufzeitumgebung. Das Tool baut aus diesem Input ein lauffähiges Binärfakt, das sämtliche Abhängigkeiten enthält (Formate: *msi, exe, pkg in a dmg, app in a dmg, deb und rpm*).

Fazit

Java ist nicht tot, lang lebe Java! Die halbjährlichen OpenJDK-Releases tun der Sprache und der Plattform gut [4]. Diesmal gab es sogar deutlich mehr neue Funktionen also noch bei Java 12 und 13. Und es gibt noch viele Features, die in zukünftigen Versionen auf ihren Einsatz warten. Uns Java-Entwicklern wird also nicht langweilig werden, die Zukunft sieht weiterhin rosig aus. Im September 2020 erwartet uns bereits Java 15.



Falk Sippach hat zwanzig Jahre Erfahrung mit Java und ist bei OIO – den Java-Experten der Trivadis – als Trainer, Softwareentwickler und -architekt tätig. Er publiziert regelmäßig auf Blogs, in Fachartikeln und spricht auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit Anderen die örtliche Java User Group.

@sippssack

Links & Literatur

- [1] <https://openjdk.java.net/projects/jdk/14/>
- [2] <https://jaxenter.de/java-13-neue-features-87053>
- [3] <https://twitter.com/KevlinHenney/status/1226094836404670464?s=03>
- [4] Codebeispiele: <https://github.com/jonatan-kazmierczak/java-new-features/blob/master/java14.md>

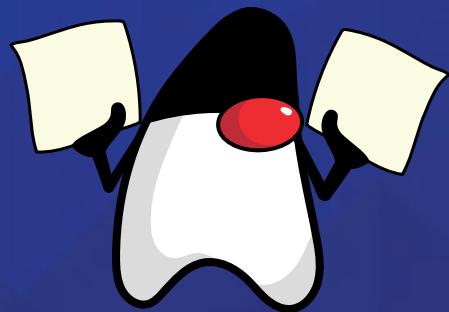
Anzeige

Anzeige

Java 14: Diese JEPs sind Teil des JDKs

JEP 305 – Pattern Matching for instanceof (Preview)

Im Zuge des Projekts Amber wird unter anderem am sogenannten Pattern Matching für Java gearbeitet. Für den Operator `instanceof` wird das Pattern Matching in Java 14 Wirklichkeit. Durch Pattern Matching soll die Programmiersprache Java prägnanter und sicherer werden. Die „Form“ von Objekten kann so präzise definiert werden, woraufhin diese dann von Statements und Expressions gegen den eigenen Input getestet werden. Die Nutzung von Pattern Matching in `instanceof` könnte für einen starken Rückgang nötiger Typumwandlungen in Java-Anwendungen sorgen. In zukünftigen Java-Versionen soll das Pattern Matching für weitere Sprachkonstrukte wie Switch Expressions kommen.



JEP 343 – Packaging Tool

Wer hätte gedacht, dass man JavaFX vielleicht doch noch einmal vermissen würde? Mit JDK 8 wurde auch das Tool `javapackager` veröffentlicht, das als Teil von JavaFX im Kit enthalten war. Nachdem JavaFX allerdings mit dem Release von JDK 11 aus Java herausflog, war auch der beliebte `javapackager` nicht mehr verfügbar. Das Packaging-Tool sorgte dafür, dass Java-Anwendungen so verpackt werden konnten, dass sie wie alle anderen Programme installiert werden konnten. Für Windows-Nutzer konnten so etwa `*.exe`-Dateien erstellt werden, deren enthaltene Java-Anwendung dann per Doppelklick installiert wurde. Da das Tool schmerzlich vermisst wird, wird ein neues Werkzeug mit Namen `jpackage` dessen Aufgabe übernehmen. Nutzer können so endlich wieder eigenständige Java-Installationsdateien erstellen, deren Basis die Java-Anwendung und ein Laufzeit-Image sind. Das Tool nimmt diesen Input und erstellt ein Image einer Java-Anwendung, die sämtliche Abhängigkeiten enthält (Formate: `msi, exe, pkg` in `dmg, app` in `dmg, deb` und `rpm`).



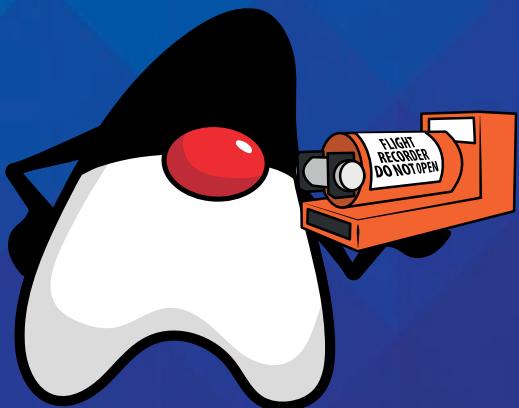
JEP 345 – NUMA-Aware Memory Allocation for G1

Mehrkerンprozessoren sind mittlerweile der allgemeine Standard. In einer NUMA-Arbeitsspeicher-Architektur erhält jeder Kern des Prozessors einen lokalen Arbeitsspeicher, auf den den anderen Kernen allerdings Zugriff gewährt wird. JEP 345 sieht vor, den G1 Garbage Collector mit der Möglichkeit auszustatten, solche Architekturen vorteilhaft zu nutzen. So soll unter anderem die Performance auf sehr leistungsstarken Maschinen erhöht werden. JEP 345 dient dabei ausschließlich der Implementierung des NUMA-Supports für den G1 Garbage Collector, nur für das Speichermanagement (Memory Allocation) und auch nur unter Linux. Ob also diese Unterstützung von NUMA-Architekturen auch für andere Garbage Collectors kommt oder für andere Teile wie etwa das `Task Queue Stealing`, ist nicht bekannt.



JEP 349 – JFR Event Streaming

Der Java Flight Recorder (JFR) ist mittlerweile Teil des OpenJDKs und damit frei verfügbar. Mit JEP 349 wurde vorgeschlagen, ein API zu erstellen, über das die vom Java Flight Recorder erhobenen Daten für die kontinuierliche Überwachung von aktiven und inaktiven Anwendungen genutzt werden können. Dabei sollen die gleichen Events aufgenommen werden wie bei der Nicht-Streaming-Variante; der Overhead soll weniger als ein Prozent betragen. Event Streaming würde so mit der Nicht-Streaming-Variante gleichzeitig durchgeführt werden. Die in JEP 349 eingeführte Funktion soll allerdings keine synchronen Callbacks für den jeweiligen Konsumenten ermöglichen, auch Daten aus Recordings, die im Zwischenspeicher liegen, sollen nicht verfügbar gemacht werden. Technisch würde das Package `jdk.jfr.consumer` im Modul `jdk.jfr` durch die Funktionalität erweitert werden, asynchron auf Events zugreifen zu können.



JEP 352 – Non-Volatile Mapped Byte Buffers

Mit JEP 352 wird der *MappedByteBuffer* um die Fähigkeit erweitert, den Zugriff auf nichtflüchtige Datenspeicher (NVMs) zu erweitern. Diese Speicher (auch als persistente Speicher bekannt) werden eingesetzt, um Daten dauerhaft zu speichern. Das Java Enhancement Proposal hat in Bezug auf das JDK API ein neues Modul und eine neue Klasse im Gepäck: Das Modul `jdk.nio.mapmode` kann genutzt werden, um den *MappedByteBuffer* zu erstellen (*read-write* oder *read-only*), der über eine Datei auf einem NVC gemappt ist. Die Klasse *ManagementFactory* macht es möglich, eine Liste von *BufferPoolMXBean*-Instanzen über die Methode `List<T> getPlatformMXBeans(Class<T>)` zu erhalten. Diese wird so modifiziert, dass sie gewisse Statistiken für sämtliche durch obiges Modul gemappte *MappedByteBuffer*-Instanzen erfasst.

JEP 358 – Helpful NullPointerExceptions

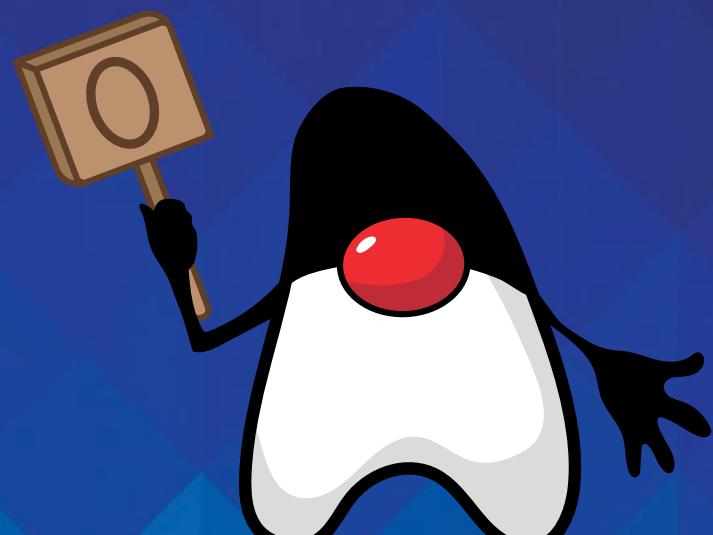
Programme, und da machen auch Java-Anwendungen keine Ausnahme, bestehen in der Regel aus einer Vielzahl an Methoden, Objekten, Funktionen, Annotationen und so weiter. Und in diesem großen Sammelsurium aus Kleinteilen kann praktisch überall eine sogenannte *NullPointerException* (NPE) auftreten. Dieses Problem soll von JEP 358 gelöst werden. Durch eine Analyse der Bytecode-Anweisungen eines Programms soll die JVM zukünftig präzise aufzeigen können, welche Variable einen Nullwert ergibt. Daraufhin soll die JVM eine *null-detail message* in der *NullPointerException* ausgeben, die neben der bislang üblichen Meldung erscheint. Das obige Beispiel würde dann wie folgt aussehen:

```
Exception in thread "main" java.lang.NullPointerException:  
    Cannot assign field 'i' because 'a' is null.  
    at Prog.main(Prog.java:5)
```

Entsprechend würde die Nachricht für das etwas komplexere Beispiel (`a.b.c.i = 99;`) so erscheinen:

```
Exception in thread "main" java.lang.NullPointerException:  
    Cannot read field 'c' because 'a.b' is null.  
    at Prog.main(Prog.java:5)
```

Ziel des JEP 358 ist es, so die Autoren, Entwicklern wichtige und hilfreiche Informationen für die Fehlerbehebung direkt an die Hand zu geben. Neue Entwickler wären so weniger verwirrt und besorgt über NPEs, und das allgemeine Verständnis eines Programms wird so auch verbessert.



JEP 359 – Records (Preview)

JEP 359 bringt *Records* als Preview-Feature für Java. Bei *Records* handelt es sich um einen neuen Typ, der im Zuge des Projekts Valhalla entwickelt wurde. *Records* stellen – wie beispielsweise auch *enums* – eine eingeschränkte Form der Deklaration *class* dar. *Records* unterscheiden sich von klassischen Klassen darin, dass sie ihr API nicht von dessen Repräsentation entkoppeln können. Die Freiheit, die dabei verloren geht, wird aber durch die gewonnene Präzision wettgemacht.

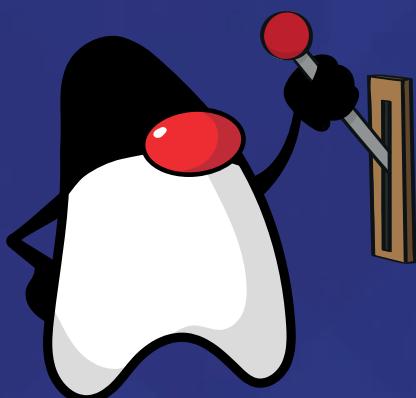
Im Proposal zu den Records hieß es dazu, dass ein *Record* „der Zustand, der gesamte Zustand und nichts als der Zustand“ sei. Er besteht aus einem Namen, der Statusbeschreibung und dem Body:

```
record Point(int x, int y) {}
```

Records bleiben dabei Klassen, auch wenn sie eingeschränkt sind. So können sie etwa Annotationen oder



Javadocs enthalten und deren Body u. a. statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklarieren. Was sie allerdings nicht vermögen, ist die Erweiterung anderer Klassen oder die Deklaration von Instanzfeldern (mit der Ausnahme der Statuskomponenten, die im Header des *Records* deklariert wurden).

**JEP 361 – Switch Expressions (Standard)**

Mit JEP 325 (Switch Expressions) wurde vorgeschlagen, die *switch*-Anweisung zu erweitern, sodass sie entweder als Anweisung (Statement) oder als Ausdruck (Expression) genutzt werden kann. Beide Formen sollten dabei in der Lage sein, entweder traditionelle oder simplifizierte Variablen und Kontrollstrukturen zu nutzen. Ziel dieses JEPs war es, das tägliche Programmieren zu vereinfachen und den Weg für Pattern Matching (JEP 305) in Verbindung mit der Anweisung *switch* zu ebnen.

Durch diesen Vorschlag ändert sich zunächst die Schreibweise beim *case*. Neben dem offensichtlichen Pfeil statt des Doppelpunkts können seit Java 12 testweise auch mehrere Werte aufgeführt werden. Bemerkenswert ist vor allem, dass kein *break* mehr benötigt wird.

Mit JEP 354 wurde von Gavin Bierman vorgeschlagen, die Funktionsweise noch ein wenig anzupassen. Um einen Wert von einer Switch Expression auszugeben, soll nun die *value*-Anweisung durch eine *yield*-Anweisung ersetzt werden. Ansonsten blieben die Switch Expressions unverändert und sind nun Teil von Java 14.

JEP 362 – Deprecate the Solaris and SPARC Ports

Das Betriebssystem Solaris stammt noch aus den Beständen von Sun Microsystems und ist mittlerweile nicht mehr wirklich zeitgemäß. Entsprechend wenig überraschend ist daher der Wunsch von Oracle, die Ports für Solaris/SPARC, Solaris/x64 und Linux/SPARC als deprecated zu markieren. Der nächste Schritt ist dann laut JEP 362, sie in einem zukünftigen Update endgültig loszuwerden. Es sei allerdings angemerkt, dass alte Java-Versionen (bis JDK 14) auf alten Systemen unverändert laufen sollen, inklusive der entsprechenden Ports.

Ziel des Ganzen ist es, sich um andere Features kümmern zu können. Sollte sich allerdings doch eine Gruppe engagierter und interessierter Entwickler finden, die die Ports betreuen und verwalten wollen, könnte die Entfernung aus dem JDK allerdings auch zu einem späteren Zeitpunkt noch gekippt werden.



JEP 363 – Remove the Concurrent Mark Sweep (CMS) Garbage Collector

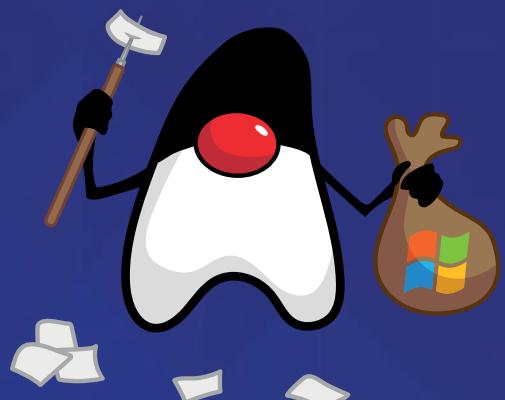
JEP 363 sieht vor, den Concurrent Mark Sweep (CMS) Garbage Collector, der bereits in Java 9 (JEP 291) als veraltet markiert worden ist, endgültig zu entfernen. Zwei Jahre hatten, so JEP-Autor Thomas Schatzl, interessierte User Zeit, sich des Projekts anzunehmen. Da dies nicht der Fall war, wird nun die letzte Reise für Mark Sweep vorbereitet.

Nutzer älterer Java-Versionen, die auf den CMS GC setzen, können aber aufatmen: Es ist nicht geplant, den Garbage Collector aus früheren Releases des JDKs zu entfernen. Gleicher gilt, auch wenn dies eigentlich nicht erwähnt werden muss, für andere Müllsammler wie Shenandoah oder ZGC. Diese bleiben im JDK enthalten.



JEP 364 – ZGC on macOS

Auch JEP 364 befasst sich mit der Garbage Collection und wurde von Erik Österlund erstellt. Ziel des Proposals ist es, den Z Garbage Collector auch für macOS-Nutzer als Option zur Verfügung zu stellen. Teil des JEPs ist auch die Funktionalität des Collectors, ungenutzten Speicher für das System wieder freizugeben, wie es in JEP 351 vorgeschlagen wurde. Diese Funktionalität ist seit Java 13 im JDK enthalten.



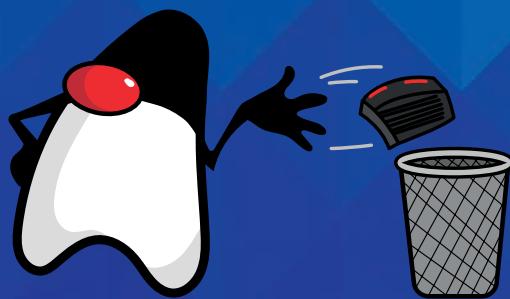
JEP 365 – ZGC on Windows

JEP 365 ist praktisch der gleiche Vorschlag wie JEP 364. Der einzige Unterschied ist, dass es bei JEP 365 um die Bereitstellung des Z Garbage Collectors (ZGC) für Windows geht. Der Großteil der Codebasis des ZGC ist plattformunabhängig, daher war es nur ein geringer Entwicklungsaufwand, ihn auch für Windows fit zu machen. Einschränkungen gibt es bei der Kompatibilität zu älteren Windows-Versionen: Windows 10 und Windows Server sollten nicht älter als Version 1803 sein, da ältere Versionen nicht das benötigte API für Speicherreservierungen beinhalten.



JEP 366 – Deprecate the ParallelScavenge + SerialOld GC Combination

Auch in JEP 366 geht es um die Garbage Collectors, bzw. in diesem Fall um eine Kombination aus zwei Müllsammlern: ParallelScavenge und SerialOld. Die Kombination der Algorithmen der beiden werden als deprecated markiert, da wohl nur sehr wenige Entwickler sie in dieser Form einsetzen. Am Ende geht es hier um den Kosten-Nutzen-Faktor, denn diese Kombination aktuell zu halten, erfordert einen erheblichen Aufwand, bedenkt man die Seltenheit der tatsächlichen Nutzung. Komplett ausgebaut wird die Kombination, die via `-XX:UseParallelGC` und `-XX:-UseParallelOldGC` aufgerufen wird, allerdings nicht. Der Garbage Collector Parallel ist allerdings in Augen von Thomas Schatzl, der dieses JEP vorgeschlagen hat, eine gute Alternative.



JEP 367 – Remove the Pack200 Tools and API

Das Kompressionsschema Pack200, das für die Komprimierung von JAR-Dateien genutzt wird, ist bereits seit Java 11 als veraltet markiert. Gemäß JEP 367 werden die Tools pack200 und unpack200 sowie das Pack200 API aus dem Package `java.util.jar` entfernt. Die Technik ist seinerzeit in Java 5 eingeführt worden und diente als Reaktion auf die damals noch sehr begrenzte Bandbreite (56k-Modems) und den geringen Speicherplatz auf Festplatten. Mit Java 9 wurden bereits vor einiger Zeit neue Kompressionsschemata eingeführt, Entwicklern wird empfohlen, jlink zu nutzen.

JEP 368 – Text Blocks (Second Preview)

JEP 326 sah vor, sogenannte Raw String Literals der Programmiersprache bereits in Version 12 hinzuzufügen. Ein solches Raw String Literal kann sich über mehrere Codezeilen erstrecken und interpretiert keine Escape-Sequenzen wie `\n` oder Unicode-Escape-Sequenzen wie `\uXXXX`. Im Gegensatz zu traditionellen Stringliteralen sollten die neuen, ergänzenden Raw Literals nicht interpretiert, sondern wie sie sind als String erzeugt werden. Sozusagen roh und unbehandelt sollten sie es Entwicklern leichter machen, Zeichensequenzen in lesbbarer Form und frei von Java-Indikatoren ausdrücken zu können. Dieses JEP wurde viel diskutiert und schließlich auf die lange Bank geschoben, da es dringender Überarbeitungen bedürfe, wie Brian Goetz es auf der Amber-Spec-Experts-Mailing-Liste zur Debatte stellte. Engagierte Java-Entwickler wurden dazu aufgerufen, dort Input zu geben und sich an dem Prozess zu beteiligen.

In Java 13 war JEP 355 als Preview-Feature enthalten, mit dem der neue Typ *Text Block* eingeführt werden sollte. Die Bemühungen des JEPs basierten auf den Grundlagen, die bereits für JEP 326 und die Raw String Literals geschaffen wurde. Die im Vorschlag beschriebenen Textblöcke sollen dem entsprechenden JEP Draft zufolge mehrzeilige Stringliteralen darstellen. Ihre Formatierung soll einheitlichen Regeln unterliegen und sich an der Darstellung des Texts im Quellcode orientieren, so dass Entwickler nicht zwingend auf Befehle zurückgreifen müssen, um das Layout des Texts zu beeinflussen. Zu den Zielen, die mit der Einführung von Text Blocks verbunden werden, gehört unter anderem das leichtere Gestalten mehrzeiliger Strings für Programmierer. Durch die Regeln zur automatischen Formatierung wären bei einer Einführung des Typs keine Escape-Sequenzen im Stil von `\n` nötig. Insbesondere solche Java-Programme, die Code anderer Programmiersprachen innerhalb von Strings enthalten, sollen mit dem neuen Format besser lesbar werden.

JEP 368 führt nun die zweite Preview ein. Basierend auf dem Feedback wurden dem Feature zwei neue Escape-Sequenzen hinzugefügt, die eine feingranulare Kontrolle der Verarbeitung von Newlines und Whitespace erlauben: `\<line-terminator>` und `\s`. Erstere Es-

cape-Sequenz kann genutzt werden, um einen Umbruch für sehr lange Stringliterale zu forcieren, ohne sie in kleinere „Substrings“ aufzuteilen. Ohne `\<line-terminator>` Escape-Sequenz:

```
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
    "elit, sed do eiusmod tempor incididunt ut labore " +
    "et dolore magna aliqua.;"
```

Mit `\<line-terminator>` Escape-Sequenz:

```
String text = """
    Lorem ipsum dolor sit amet, consectetur adipiscing \
    elit, sed do eiusmod tempor incididunt ut labore \
    et dolore magna aliqua.\
""";
```

Die Escape-Sequenz `\s` lässt sich einfach als einzelnes Leerzeichen (`\u0020`) beschreiben. Damit lässt sich im unteren Beispiel sicherstellen, dass die Zeilen exakt 6 Stellen haben und nicht länger sind:

```
String colors = """
    red \s
    green\s
    blue \s
""";
```

Diese Escape-Sequenz kann in den neuen Text Blocks, aber auch in klassischen Stringliteralen genutzt werden.



JEP 370: Foreign-Memory Access API (Incubator)

Es gibt tonnenweise Java-Bibliotheken und auch -Anwendungen, die auf fremden Speicher zugreifen. Prominente Beispiele sind Ignite, mapDB, memcached oder Nettrys ByteBuf API. Dennoch stellt das Java API selbst keine gute Möglichkeit hierfür zur Verfügung, obwohl dadurch Kosten in Sachen Garbage Collection (besonders bei der Verwaltung großen Caches) gespart und Speicher über mehrere Prozesse hinweg geteilt werden kann. Auch das Serialisieren und Deserialisieren von Speicherinhalten via Mappings von Dateien in den Speichern wird durch den Zugriff auf fremden Speicher möglich (etwa via mmap).

Mit JEP 370 wird ein passendes Java API ins JDK implementiert, mit dem Java-Anwendungen sicher und effizient auf fremden Speicher zugreifen können, der außerhalb des Java Heaps angesiedelt ist. Wichtig sind die drei Prämissen: Allgemeingültigkeit, Sicherheit und Determinismus.



Anzeige

Ersteres bedeutet, dass ein einziges API in Verbindung mit unterschiedlichen fremden Speichern arbeiten sollte (gemeint sind nativer Speicher, persistenter Speicher etc.). Die Sicherheit der JVM ist das höchste Gut, daher sollte das API nicht dazu fähig sein, diese zu unterwandern – völlig egal, welcher Speicher zum Einsatz kommt. Zudem (Determinismus) sollte die Freigabe von Speicher explizit im Quelltext erfolgen.

Dieses JEP wurde im Zuge des Project Panama entwickelt und stellt eine klare Alternative zur Verwendung von existierenden APIs wie ByteBuffer, Unsafe oder JNI dar. Natürlich hilft die Implementierung beim Erreichen der generellen Maxime des Projekts: der Unterstützung von nativer Interoperabilität von Java.

Ein Blick auf JEP 359: Java Records

Rekordverdächtig strukturiert

Datenklassen, also Java-Klassen, deren einziger Zweck darin besteht, Daten zu halten und diese über Getter und Setter zugänglich zu machen, gehören in vielen Softwareprojekten zu den größten Sammelstellen von Boilerplate-Code. Für jede neue Klasse jeweils Konstruktoren, die Methoden *equals*, *hashCode* und *toString* und für jedes Feld noch einen Getter und einen Setter zu erstellen, ist für viele Entwickler eine verhasste Zeremonie geworden – sofern sie nicht direkt Bibliotheken wie Lombok einsetzen, um dieser zu entgehen. JEP 359 soll Abhilfe schaffen.

von Tim Zöller

Mit JEP 359 werden Records in die JVM eingeführt – wenn auch vorerst nur als Preview Feature. Um sie ausprobieren zu können, müssen sowohl der Compiler als auch das erstellte Programm mit dem Flag `--enable-preview` ausgeführt werden.

Das Problem, das mit Records gelöst werden soll

Was genau macht das Erstellen von Datenklassen in Java so mühsam? Das Herzstück solcher Klassen ist die definierte Liste von Instanzvariablen, die die Zustandsbeschreibung eines Objekts darstellen. Möchten Entwickler eine Klasse entwerfen, die einen Würfel widerspiegelt, kommen sie mit drei Variablen aus: Höhe, Breite und Tiefe. Um mit Instanzen des Typs Würfel arbeiten zu können, müssen sie allerdings weitere Formalitäten erfüllen: Die Klasse benötigt Konstruktoren, Getter und Setter. Ebenfalls müssen die Methoden *equals*, *hashCode* und *toString* überschrieben werden. Diese gesamte Arbeit läuft in den allermeisten Fällen so gleichförmig ab, dass Entwickler sie automatisiert von ihrer Entwicklungsumgebung vornehmen lassen: Für jede Instanzvariable werden Getter und Setter erstellt, alle sollen bei *hashCode*, *equals* und *toString* berücksichtigt werden (oder schlimmer: *hashCode* und *equals* werden erst gar nicht implementiert). Oft wird auch noch ein Konstruktor definiert, der sämtliche Variablen als Parameter enthalten kann, um ein Objekt vollständig initialisiert erstellen zu können. Nach dieser Prozedur hat die Klasse dann knapp 65 Zeilen Code, von denen etwa fünf ausreichen, um die wichtigste Information zu beschreiben – den Namen, den Modifier und welchen Zustand sie hält.

Natürlich können an einem Würfel auch seine Oberfläche, Kantenlänge oder sein Volumen interessant

sein – diese Werte leiten sich aber von seiner Zustandsbeschreibung ab und stellen keine neuen Variablen dar. Um das Volumen des Würfels zu erhalten, würden Entwickler eine Methode *calculateVolume()* bereitstellen, die die Länge, Höhe und Breite des Würfels multipliziert. Auch diese Methoden enthalten Informationen, die für alle Entwickler des Teams wichtig sind, um die Eigenschaften eines Würfels zu verstehen. Sie zwischen den anderen neun Methoden zu identifizieren, die vorhin von der Entwicklungsumgebung generiert wurden, kann auf den ersten Blick unter Umständen schwierig sein. Die Klasse vermischt das „Was“ mit dem „Wie“, sie hat eine unnötig hohe kognitive Komplexität. Mit der Einführung von Records soll exakt dieses Problem vermieden werden, indem Daten auch als Daten modelliert werden und der Zugriff darauf sich aus dem Kontext ergibt.

Aufbau und Struktur von Records

Records sind eine neue Typendeklaration in Java und stellen eine spezialisierte Form einer Klasse dar. Sie sind für einen klar definierten Zweck gedacht, haben gegenüber herkömmlichen Klassen aber Einschränkungen, ähnlich wie Enums. Die Zustandsbeschreibung des Records wird über die Definition von sogenannten Komponenten vorgenommen, die aus einem Typ und einer Bezeichnung bestehen. Ein simpler Record, der einen Würfel mit den Komponenten „Höhe“, „Breite“ und „Tiefe“ beschreibt, findet sich nachfolgend:

```
public record Wuerfel(int hoehe, int breite, int tiefe) {  
}
```

Eine vergleichbar aufgebaute herkömmliche Klasse zeigt Listing 1.

Auffällig hierbei ist, dass die Komponenten des Records einen Teil seiner Deklaration darstellen und nicht im Körper der Klasse definiert werden. Jede dieser Komponenten entspricht einer implizit definierten, finalen Instanzvariable sowie einer gleichnamigen Zugriffsmethode, die sowohl implizit als auch explizit definiert sein kann. Alle Komponenten sind darüber hinaus Bestandteil des Standardkonstruktors. Das „Wie“, also die Art und Weise, wie auf den Zustand eines Objekts zugegriffen werden kann, leitet sich also aus seiner Definition ab und muss nicht explizit aufgeschrieben werden. Mit folgendem Code kann demnach eine Instanz des oben gezeigten Records-Würfels erstellt und auf den Wert der Höhe zugegriffen werden:

```
var wuerfel = new Wuerfel(10, 10, 10);
wuerfel.hoehe();
> 10
```

Records verfügen nicht über Setter, da ihre Instanzvariablen, wie schon beschrieben, implizit final sind und folglich nach der Instanziierung nicht mehr aktualisiert werden können. Darüber hinaus müssen bei Records die Methoden *equals*, *hashCode* und *toString* nicht implementiert werden – sie ergeben sich ebenfalls aus den Komponenten. Bei Bedarf können sie überschrieben und damit angepasst werden.

```
wuerfel.toString();
> Wuerfel[hoehe=10, breite=10, tiefe=10]

var w1 = new Wuerfel(10, 10, 10);
var w2 = new Wuerfel(10, 10, 10);
w1.equals(w2);
> true
```

Aus der Tatsache, dass die Zugriffsmethoden denselben Namen tragen wie ihre Komponenten, folgt ein Problem: Einige Komponentennamen würden Methoden überschreiben, die beispielsweise zur Klasse *Object* gehören, beispielsweise *getClass*, *finalize* oder auch *toString*. Aus diesem Grund sind folgende Bezeichnungen für Records verboten und führen zu einem Compile-Fehler: *clone*, *finalize*, *getClass*, *hashCode*, *notify*, *notifyAll*, *readObjectNoData*, *readResolve*, *serialPersistentFields*, *serialVersionUID*, *toString*, *wait*, *writeReplace*.

Weitere Attribute von Records

Records können um statische Klassenvariablen ergänzt werden, beispielsweise um Literale zu benennen. Zusätzliche Instanzvariablen hingegen dürfen nicht deklariert werden, da diese zum Zustand eines Objekts gehören würden – dieser soll aber ausschließlich über die Komponenten eines Records definiert sein. Zusätzliche Methoden, etwa um Kalkulationen mit dem Zustand eines Objekts durchzuführen, sind erlaubt (Listing 2). Ebenfalls möglich sind statische Methoden und statische Initialisierungsblöcke.

Konstruktoren

Der Zustand eines Objekts wird bei dessen Erzeugung über den Konstruktor definiert und kann anschließend nicht mehr geändert werden – zumindest auf der definierten Ebene. Im Text zum JEP wird dieser Zustand „shallowly immutable“ genannt. Da die Zugriffsmethoden von Komponenten Referenzen auf die enthaltenen Objekte liefern, ist es möglich, die Objekte, auf die sie verweisen, zu manipulieren. Tatsächlich unveränderliche Records sind nicht möglich – ähnlich wie auch bei

Listing 1

```
public final class Wuerfel {

    private final int height;
    private final int width;
    private final int depth;

    public Wuerfel(int height, int width, int depth) {
        this.height = height;
        this.width = width;
        this.depth = depth;
    }

    public int getHeight() {
        return height;
    }

    public int getWidth() {
        return width;
    }

    public int getDepth() {
        return depth;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Wuerfel wuerfel = (Wuerfel) o;
        return height == wuerfel.height &&
            width == wuerfel.width &&
            depth == wuerfel.depth;
    }

    @Override
    public int hashCode() {
        return Objects.hash(height, width, depth);
    }

    @Override
    public String toString() {
        return "Wuerfel{" +
            "height=" + height +
            ", width=" + width +
            ", depth=" + depth +
            '}';
    }
}
```

normalen Klassen können die Referenzen zu gekapselten Werten entweichen.

Bei umfangreicheren Records, die etwa über mehr als zehn Komponenten verfügen, kann die Leserlichkeit von Code leiden. Während sich beim Aufruf von Settern anhand der Methodennamen noch der Kontext der Daten erkennen lässt, ist dies bei großen Parameterlisten in Konstruktoren weit weniger offensichtlich, beispielsweise in Listing 3. Die Darstellung mag durch die Benutzung von Literalen anstelle von sprechenden Variablennamen zwar überspitzt sein, das Problem der

schwierigen Lesbarkeit existiert nichtsdestotrotz. Dieses Problem wird durch die Unterstützung von Entwicklungsumgebungen zwar abgeschwächt (die Early Access Preview von IntelliJ zeigt beispielsweise schon vor jedem Konstruktorparameter den Variablenamen an), greift aber etwa bei der Betrachtung des Codes in einer Diff View nicht. Hier würden benannte Parameter helfen, wie sie zum Beispiel bei Kotlin verwendet werden. Mit diesem Gedanken spielte auch Brian Goetz schon in einem Artikel, der sich mit Möglichkeiten von Datenklassen und „Sealed Types“ beschäftigt [1].

Alternativ könnte dieses Problem mit einem Builder-Ansatz umgangen werden, wie Lombok ihn schon für Datenklassen ermöglicht. Eine Bibliothek, die der Autor dieses Artikels zu diesem Zweck geschrieben hat, findet sich auf GitHub [2].

Der Standardkonstruktor ist bei Records, anders als bei normalen Klassen, nicht parameterlos, sondern enthält Parameter für alle Komponenten des Records. Da jede Record-Instanz nach dem Erzeugen nicht mehr geändert werden kann, bietet sich der Konstruktor an, um zu prüfen, ob das Objekt vollständig und in sich schlüssig initialisiert wurde. In Listing 4 kann man sehen, dass Konstruktoren in Records genau so definiert werden können wie in üblichen Klassen. Leider führt das aber dazu, dass die elegante, kurzgefasste Zustandsbeschreibung des Records mehrfach wiederholt wird. Jeder Variablenname wird noch viermal getippt: in der Parameterliste des Konstruktors, in der Validierung und zweimal bei der Zuweisung zur Instanzvariable. Da dies der gewünschten Einfachheit von Records entgegenläuft, wurde eine Kurzschreibweise für Konstruktoren von Records eingeführt, die in Listing 5 aufgeführt ist. Sie benötigt weder eine Parameterliste noch die Zuweisung von Parametern zu den Instanzvariablen – beide werden implizit erledigt. Entwickler können sich ausschließlich auf die Validierungslogik konzentrieren, also erneut auf das „Was“, nicht auf das „Wie“.

Vererbung

Records sind implizit final, von ihnen kann also keine andere Klasse erben. Betrachten wir die Grammatik von Records in Listing 6, fällt noch etwas auf: Records dürfen zwar ein oder mehrere Interfaces implementieren, nicht jedoch von einer Oberklasse oder einem anderen erben. Der Codeabschnitt in Listing 7 zeigt einen Record, der ein Interface implementiert. Diese Regeln zur Vererbung leiten sich erneut daraus ab, dass Records durch ihre Zustandsbeschreibung definiert sein sollen. Übliche Java-Klassen können ihren Zustand hingegen beliebig definieren und manipulieren. Würde ein Record von einer solchen Klasse erben, wäre diese Tatsache nicht mehr sicherzustellen. Auch das Erben von einem anderen Record würde die Klarheit der Deklaration verwässern. Ein Record würde weitere Komponenten erben, die Teil seiner Zustandsbeschreibung wären. Damit wäre ebendiese Zustandsbeschreibung über mehrere Typen verteilt und schwieriger zu erfassen.

Listing 2

```
public record Wuerfel(int hoehe, int
breite, int tiefe) {
    private final static int DICHTE = 3;
    public int masse() {
        return volumen() * DICHTE;
    }
    public int volumen() {
        return hoehe * breite * tiefe;
    }
}
```

Listing 3

```
var game = new Game(new
Person("Rüdiger",
"Behrens",
LocalDate.of(1982, 3, 17),
186),
new Person("Frank",
"Meier",
LocalDate.of(1990, 2, 2),
170),
10,
20);
```

Listing 4

```
public record Wuerfel(int hoehe, int breite, int tiefe) {
    public Wuerfel(int hoehe, int breite, int tiefe) {
        if(hoehe < 0 || breite < 0 || tiefe < 0) {
            throw new IllegalArgumentException("Werte dürfen nicht negativ sein!");
        }
        this.hoehe = hoehe;
        this.breite = breite;
        this.tiefe = tiefe;
    }
}
```

Listing 5

```
public record Wuerfel(int hoehe, int breite, int tiefe) {
    public Wuerfel {
        if (hoehe < 0 || breite < 0 || tiefe < 0) {
            throw new IllegalArgumentException("Werte dürfen nicht negativ sein!");
        }
    }
}
```

Annotations und Reflection

Sowohl Records selbst als auch ihre Komponenten können mit Annotationen versehen werden. Für Annotationen auf Typebene gelten dieselben Regeln wie bei Klassen: Die Annotation muss *TYPE* als Target erlauben. An den Komponenten eines Records können Annotationen genutzt werden, die als Target *PARAMETER*, *FIELD*, *METHOD* oder das mit Records neu eingeführte Target *RECORD_COMPONENT* ermöglichen. Die Vielzahl erlaubter Annotationsziele ergibt sich aus den implizit abgeleiteten Elementen einer Record-Komponente: Konstruktorparameter, Instanzvariablen und Methoden.

Das Java Reflection API wurde ebenfalls erweitert, um die Arbeit mit Records zu ermöglichen. Der Typ *java.lang.Class* wurde um zwei Methoden erweitert: Mittels *isRecord()* lässt sich feststellen, ob es sich bei einer Klasse um einen Record handelt. Die Methode *getRecordComponents()* liefert ein Array vom neuen Typ *java.lang.reflect.RecordComponent* zurück, der Infor-

Listing 6

```
RecordDeclaration:
{ClassModifier} record TypeIdentifier [TypeParameters]
  (RecordComponents) [SuperInterfaces] [RecordBody]

RecordComponents:
{RecordComponent {, RecordComponent} }

RecordComponent:
{Annotation} UnannType Identifier

RecordBody:
{ {RecordBodyDeclaration} }

RecordBodyDeclaration:
ClassBodyDeclaration
RecordConstructorDeclaration

RecordConstructorDeclaration:
{Annotation} {ConstructorModifier} [TypeParameters] SimpleName
[Throws] ConstructorBody
```

Listing 7

```
public interface Koerper {
    int volumen();
}

public record Wuerfel(int hoehe, int breite, int tiefe) implements Koerper {

    @Override
    public int volumen() {
        return hoehe * breite * tiefe;
    }
}
```

mationen zu den Komponenten eines Records enthält, wie zum Beispiel den Namen, Datentypen, generische Typen, Annotationen oder die Zugriffsmethode.

Zusammenfassung und Ausblick

Mit Records erhält Java einen interessanten, neuen Typ – wenn auch vorerst lediglich als Preview Feature. Der Plan, Daten kurz und prägnant eben auch als Daten zu modellieren und von der üblichen Zeremonie zu befreien, befindet sich auf einem guten Weg und wird von vielen Entwicklern sehnsgütig erwartet. Nicht nur wird das Erzeugen von immer gleichförmigem Code obsolet und es entfällt die Gefahr, bei der Implementierung die wichtigen Methoden *equals* und *hashCode* zu vergessen. Vielmehr kann die Lesbarkeit von Code in Zukunft vielmehr stark vereinfacht werden, sodass die wichtigen Attribute eines Typs schneller erfasst werden können. Interessant wird sein, wie gängige Bibliotheken mit dem unveränderlichen Charakter der Record-Instanzen umgehen. Derzeit benutzen viele Mapper und Serialisierungstools die Setter von Datenklassen, um sie mit Werten zu füllen. Hier ist eine Umstellung auf einen konstruktorbasierten Ansatz zu erwarten, um auch Records behandeln zu können.

Die Einschränkungen, die der neue Typ mit sich bringt, werden dafür sorgen, dass Records keineswegs die herkömmlichen, veränderlichen Datentypen in allen Situationen ablösen werden. Das ist auch explizit nicht beabsichtigt. Diese Restriktionen wurden aus gutem Grund und mit Blick auf die Zukunft beschlossen: Aufgrund der (von Menschen und Compilern) leicht zu erfassenden Zustandsbeschreibung von Records eignen sie sich in Kombination mit Sealed Types (JEP 360 [3]) sehr gut für Pattern Matching und Destrukturierung und damit für einen weiteren großen Schritt hin zu weiteren funktionalen Features in Java [4].

Es bleibt abzuwarten, welche Änderungen an den hier beschriebenen Konzepten in der Previewphase noch vorgenommen werden. Die Erfahrung mit bisherigen Previewfeatures zeigt, dass auf das Feedback der Community generell eingegangen wird und getätigte Annahmen noch einmal überprüft werden können. Trotzdem lässt sich schon jetzt absehen, dass Records eine große Bereicherung für Java sein werden, die die Tür zu weiteren Innovationen weit aufstößt.



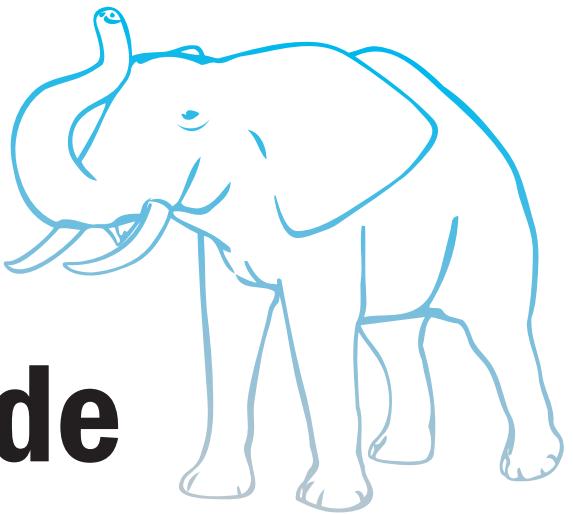
Tim Zöller ist Mitgründer der Java Usergroup Mainz und Consultant und Teamleiter bei der ilum:e informatik ag in Mainz. Er entwickelt seit über zehn Jahren Software mit Java und arbeitet momentan am liebsten mit Eclipse MicroProfile, Quarkus oder Clojure.

Links & Literatur

- [1] <https://cr.openjdk.java.net/~briangoetz/amber/datum.html>
- [2] <https://github.com/javahippie/jukebox>
- [3] <https://openjdk.java.net/jeps/360>
- [4] <http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html>

Unter der Lupe: Java 14

Die Java-Elefantenrunde



Java Magazin: Java 14 ist gerade erschienen, daher die obligatorische Frage: Was ist euer Highlight des Releases?

Oliver B. Fischer: Das wohl bekannteste Feature der Textblöcke ist natürlich offensichtlich, weil es wesentlich zur positiven Developer Experience beitragen wird. Persönlich halte ich den Einstieg ins Pattern Matching mit JEP 303 Pattern Matching for instanceof für interessant, ebenso wie JEP 361 für Switch Expressions.

Dr. Heinz Kabutz: Vor einigen Monaten hat sich Brian Goetz auf der Java-Champion-Mailingliste darüber beschwert, dass sie bei Oracle nicht genug Feedback zu ihren Vorschaufunktionen seitens der Community bekommen. Daraufhin beschloss ich, meine Website [1] von nun an immer mit der neuesten Previewversion von Java zu betreiben. Wenn man sich die Website jetzt ansieht, sieht man oben „Running on Java 14-ea+29-1384 (Preview)“. Es ist ein bisschen Arbeit, das so zu machen. Neben meinem Master Branch habe ich drei weitere Branches für die Java-14-Funktionen: java14-preview, jep359 (Records) und jep368 (Text Blocks). Im Allgemeinen arbeite ich im Master, aber die drei anderen Branches werden dann abgeleitet.

Man könnte vermuten, dass JavaSpecialists.eu nur eine Hobbywebsite ist und dass es nicht viel Risiko bergen würde, wenn sie down wäre. Das ist jedoch nicht der Fall. Sie ist mein täglich Brot. Ohne sie hättet ihr nie etwas von mir gehört. Ich kann mir auf keinen Fall leisten, dass sie abgeschaltet wird, und setze also ein ständiges Monitoring ein, um zu überprüfen, ob alles noch funktioniert.



Oliver B. Fischer arbeitet bei der E-Post Development GmbH und organisiert die JUG Berlin-Bранdenburg.

Aber zum Kern der Frage: Von allen Funktionen, die in Java 14 enthalten sind, gefallen mir Records (JEP359) am besten. Eine interessante Neuerung ist jedoch ebenfalls, dass ConditionNode innerhalb des AbstractQueuedSynchronizer jetzt auch ein ManagedBlocker ist. Das bedeutet, dass die Bedingung von ReentrantLock gut mit dem ForkJoinPool und damit mit parallelen Streams zusammenspielt. LinkedBlockingQueue basiert auf ReentrantLock, und so arbeitet auch das gut zusammen. Ich habe unter [2] über die Möglichkeiten geschrieben und schlage einen Hack vor, um LinkedBlockingQueue zu fixen. Mit Java 14 funktioniert es out of the box.

Markus Günther: Mein persönliches Highlight ist die Einführung des Record-Konzepts (JEP-359). Records sind zwar noch ein Previewfeature in Java 14 und man muss das Feature explizit aktivieren, um es zu verwenden. Sie fügen sich aber logisch konsistent in das etablierte Java-Typsystem ein und bieten großes Potenzial. Mit dem aktuellen Release bedeutet das, dass man für einfache Datenhaltungsklassen nicht länger ausschweifenden Boilerplate-Code schreiben muss. Stattdessen kapselt man alle relevanten Attribute in Form eines Records und lässt den Compiler einen sinnvollen Defaultkonstruktor, Zugriffsmethoden zum Lesen und Schreiben der Attribute und einen Vertrag für *equals* und *hashCode* etc. generieren. Auf den zweiten Blick erkennt man, dass Records den Weg für weitere interessante Features ebnen: Records könnten beispielsweise wunderbar mit Pattern Matching (JEP Draft: Pattern Matching for Switch [3]) in Form von Deconstruction Patterns zusammenspielen. Records sind also definitiv ein Konzept, das man im Auge behalten sollte.

Tim Riemer: Die Switch Expressions sind nun erwachsen, d. h. mit dem Java-14-Release vom Previewfeature zu einem Standardfeature erhoben worden. Ein weiteres Feature, von dem man als Entwickler in seiner täglichen Arbeit profitieren wird, ist JEP-358: Helpful NullPointerExceptions. Daneben sind Records und Pat-

tern Matching for instanceof als interessante Preview-features dabei.

Jens Schauder: Java 14 ist ein echtes Feuerwerk in meinen Augen. Es ist schwierig, sich zwischen Pattern Matching for instanceof, besseren NPEs, Switch Expressions und Records zu entscheiden. Wenn ich mich entscheiden muss, wähle ich das Pattern Matching for instanceof.

Es ist etwas, was in der Codebasis von Spring Data wirklich praktisch wäre. Records finde ich eigentlich noch besser, aber sie sind als Preview gekennzeichnet.

Michael Vitz: Dieses Mal finde ich es schwer, ein einzelnes Highlight zu nennen. Für die Sprache selbst ist es wohl das Preview von Records, die die Erstellung von klassischen Java Beans extrem erleichtern. Abseits davon finde ich die neuen NullPointerExceptions super, da mich diese sehr unaufgeregt und ohne dass ich etwas tun muss, unterstützen.

JM: Anschließend an die erste Frage: Welches Feature vermisst ihr in Java 14?

Fischer: Ganz klar Stringinterpolation für Strings und Textblöcke.

Günther: Das ist jetzt nicht zwingend auf Java 14 bezogen, aber ich vermisse nach wie vor die Möglichkeit, lokale Variablen durch *val* (analog zu dem seit Java 10 existierenden *var* für veränderliche Variablen) als unveränderlich zu deklarieren.

Riemer: Eigentlich vermisste ich nichts, was aber auch damit zu tun hat, dass man bei einem sechsmontatigen Releasezyklus keine riesigen Neuheiten erwartet. Persönlich fände ich es schön, wenn man erste Ergebnisse von Project Loom und Valhalla sehen könnte.

Schauder: Ich bin ja kein Sprachdesigner. Daher kann ich gar nicht so recht beurteilen, was sinnvoll ist und was nicht. Ich bin immer beeindruckt, wenn ich mich mit Leuten unterhalte, die von so etwas Ahnung haben. Aber ich denke, ein Typsystem könnte mehr leisten, als das von Java es momentan tut. Union Types habe ich mir schon oft gewünscht. Aber wie gesagt, ich habe keine Ahnung, was für Auswirkungen auf das große Ganze das hätte und ob es die Mühe wert wäre.

Vitz: Ehrlich gesagt, weiß ich das gar nicht. Für mich ist Java bereits seit Java 8 sehr rund. Ab und an merke ich, dass etwas gefehlt hat, wenn ich ein neues Feature sehe. Alles in allem sticht für mich persönlich aktuell aber nichts akut heraus.

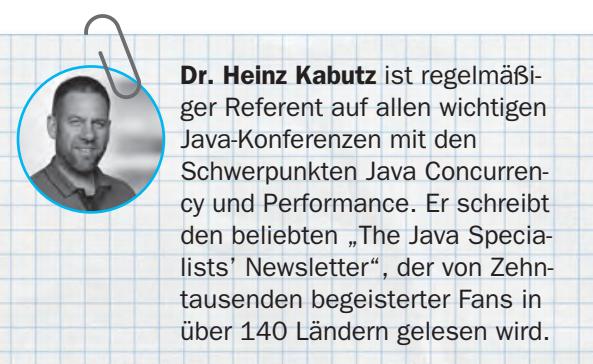
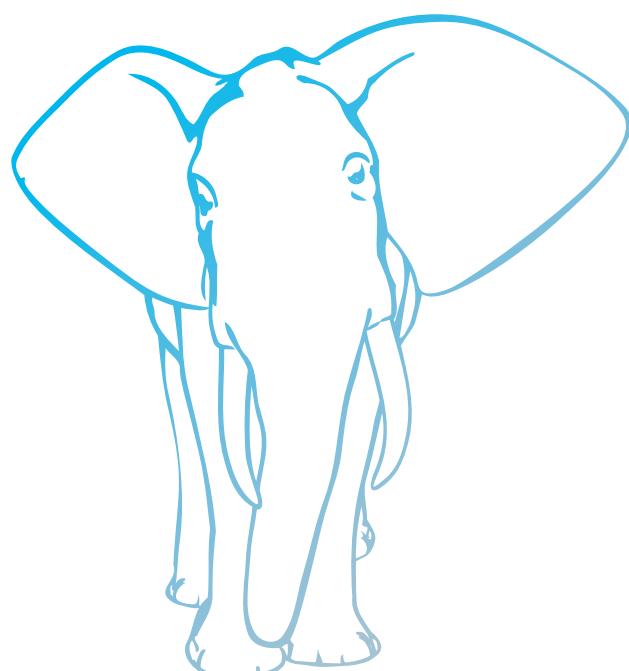


Markus Günther ist freiberuflicher Softwareentwickler und -architekt. Er beschäftigt sich aktuell mit Fast-Data-Architekturen und unterstützt seine Kunden bei der Umsetzung robuster, skalierfähiger Systeme.

JM: Werdet ihr bzw. werden eure Kunden sofort auf das neue Release updaten oder lohnt sich das für euch bzw. eure Kunden nicht?

Fischer: Java 14 ist keine LTS-Version, was für viele Kunden verständlicherweise ein wichtiges Kriterium für den Produktionseinsatz darstellt. Auf der anderen Seite ist die Fähigkeit, schnell auf eine höhere JDK-Version umsteigen zu können, auch ein Qualitätsindikator für die Softwareentwicklung in einer Organisation. Pauschale Aussagen sind nicht möglich, nur Empfehlungen. Wo möglich, würde ich immer relativ bald auf die nächste JDK-Version umsteigen.

Günther: Meine Kunden arbeiten hauptsächlich noch mit Java 8. Stellenweise beobachte ich das selbst bei der technologischen Auswahl innerhalb von Greenfield-Projekten: Manch einer nutzt die Chance und steigt auf Java 11 um, andere dagegen nutzen weiterhin Java 8 als Basis. Bei der zuletzt genannten Gruppe geht das in der Regel damit einher, dass der Anwendungsentwurf ohne Blick auf künftige Migrationen (beispielsweise Modularisierung mit dem JPMS) stattfindet. Bei älteren Systemen auf Basis von Java 8 mit monolithischer Struktur ist eine Migration auch durchaus nicht trivial, wenn der Monolith nicht bereits eine strikte Trennung in fachliche Module aufweist. Diesen Invest muss man irgendwann tragen, aber nach meiner Wahrnehmung scheuen sich die Kunden insbesondere bei Bestandsprojekten davor. Und wenn nicht, dann wäre mein Ratschlag, sich zunächst dem Sprung auf eine



Dr. Heinz Kabutz ist regelmäßiger Referent auf allen wichtigen Java-Konferenzen mit den Schwerpunkten Java Concurrency und Performance. Er schreibt den beliebten „The Java Specialists’ Newsletter“, der von Zehntausenden begeisterter Fans in über 140 Ländern gelesen wird.



Tim Riemer ist als Softwarearchitekt mit Schwerpunkt Integration bei der FNT GmbH in Heiligenhaus beschäftigt. Neben seinem Job befasst er sich mit aktuellen Themen rund um Softwarearchitektur, Spring Boot, Build-Automation und Kotlin.

zordan_f

LTS-Version (Java 11) zu widmen und sich erst danach – sofern die Kosten/Nutzen-Rechnung das rechtfertigt – auf die kürzeren Releasezyklen einzustellen.

Riemer: Aktuell setzen wir Java 11 bzw. 12 in Produktion ein. Unseren Teams steht es aber frei, auf eine aktuellere Version zu wechseln, doch ich persönlich sehe bei den neuen Features in Java 14 das Kosten/Nutzen-Verhältnis, den der Aufwand eines Upgrades bedeutet, nicht.

Schauder: Bis Spring selbst Java 14 einsetzt, also wir unseren Code in Java 14 schreiben, wird es wohl noch ein gehöriges Weilchen dauern. Aber unterstützen werden wir es sicherlich zeitnah.

Zum Beispiel hat es schon die ersten Experimente für die Unterstützung von Records in Spring Data gegeben.

Vitz: Viele meiner Kunden haben gerade erst ein Update von JDK 8 auf 11 hinter sich und werden tendenziell nicht direkt das nächste Upgrade auf 14 durchführen. Vermutlich wird ein Großteil hier auf JDK 17 warten und somit auf das nächste Release mit Long-Term-Support wechseln.

JM: Welche Schwierigkeiten seht ihr dabei, die neuen Java-Versionen gleich in Produktion einzusetzen?

Fischer: Wir müssen zwischen neuen Sprachfeatures und neuen JDK-Versionen unterscheiden. Gerade das Update von JDK-Versionen kann zu Überraschungen führen, beispielsweise wenn die Versionssprünge groß sind und es seit dem letzten Update Änderungen im Bereich Security gibt. Generell sollte jede Software auch mit der nächsthöheren

JDK-Version in der CI-Pipeline oder in Testumgebungen getestet werden.

Neue Sprachfeatures sollten nur dann in Produktion gehen, wenn sie final sind. Neue Sprachfeatures zuerst in Tests einzusetzen, ist ein guter Weg, um früh erste Erfahrungen mit ihnen zu sammeln.

Günther: Selbst bei Releases, die, gemessen an ihrem Featureumfang, einen eher evolutionären Charakter haben, kann sich das Verhalten der Anwendung nach dem Versionsupgrade im Produktivbetrieb anders zeigen als gewohnt. Es ist daher unabdingbar, ein solches Upgrade mit sinnvollen Regressions- und Kompatibilitätstests zu begleiten.

Riemer: Größere Schwierigkeiten sehe ich bei uns nicht, aber das Entfernen des Concurrent Mark Sweep (CMS) Garbage Collectors könnte natürlich bei einigen ein reibungsloses Update verhindern.

Schauder: Als Bibliothek/Framework kann Spring nicht voraussetzen, dass alle auf die aktuelle Version wechseln. Wie in der Vergangenheit auch wird es einen zeitlich entspannten Übergang geben, in dem man Java 14 nutzen kann, aber auch noch wesentlich ältere Versionen unterstützt werden. Unabhängig von Spring hoffe ich doch, dass wir die Schmerzen der Einführung des Modulsystems hinter uns gelassen haben und ein Versionsupgrade recht problemlos ist, wenn man eine Anwendung baut.

Vitz: Ich glaube, es liegt vor allem am Vertrauen. Obwohl die am JDK entwickelnden Menschen stets betonen, dass ein Upgrade von z. B. 13 auf 14 nur ein kleiner Schritt ist, wird das nicht so wahrgenommen. Viele meiner Kunden möchten gerne ein längeres Intervall mit Updates erhalten. Zudem müssen aktuell zusätzlich zum JDK-Update auch Bibliotheken, Build-Tools oder die IDE upgedatet werden. Außerdem werden mittlerweile ja tatsächlich vorher deprecate Teile aus dem JDK entfernt, was zu zusätzlichen Aufwand führen kann.

JM: Was hältet ihr von JEP 371 Hidden Classes, das möglicherweise mit einer der nächsten Java-Versionen kommen wird? Viele unserer Leser scheinen sich besonders dafür zu interessieren ...

Riemer: Ich sehe Hidden Classes eher als relevant für Framework- oder Sprachentwickler. Nach meinem Verständnis geht es darum, dass aktuell nicht unterschieden werden kann, ob der Bytecode einer Klasse



Jens Schauder ist leidenschaftlicher Softwareentwickler. Er arbeitet bei der T-Systems on site services GmbH als Senior Consultant. Besonders interessiert ihn das Thema Qualität in der Softwareentwicklung. In seiner Freizeit entwickelt er Degrapph.

Programmiersprachen geraten entweder in Vergessenheit oder sie entwickeln sich weiter und kopieren dabei fleißig Features von anderen Sprachen – so ist das auch bei Java der Fall.

dynamisch generiert oder statisch erzeugt wurde. Hier helfen Hidden Classes, die Sichtbarkeit und Lebenszeit einzuschränken.

Vitz: Da ich primär Anwendungen mit Java und eher selten Frameworks baue, hoffe ich, dass dieses Feature auf mich wenig Auswirkungen hat. Das Versprechen, dass nun von Frameworks generierte Klassen nicht in Stack Traces auftauchen, kann sich allerdings als hilfreich erweisen und Stack Traces etwas besser lesbar machen. Ich lasse mich hier gerne positiv überraschen.

JM: Nähert sich Java durch Features wie JEP 358 Helpful NullPointerExceptions „modernen“ Sprachen wie Kotlin an?

Fischer: Wurde auch Zeit, kann ich da nur sagen. Ein gutes Beispiel, denn es zeigt, dass auch kleinere Veränderungen die Developer Experience wesentlich verbessern können.

Kabutz: „Modern“ ist gut. Die Herausforderung bei einer Sprache wie Kotlin besteht darin, dass sie auch ältere JVMs unterstützen muss. Eine ganze Reihe von schönen neuen Java-Performanceverbesserungen sind noch nicht in Kotlin verfügbar. Zum Beispiel wurde String überarbeitet, um ein schnelles Anhängen mit + zu ermöglichen. Der von Kotlin generierte Bytecode ist jedoch immer noch der von Java 8.

Mir gefällt, wie Kotlin die gefürchtete NullPointerException vermeidet. Leider sind die Helpful NullPointerExceptions ziemlich weit davon entfernt. Stattdessen zeigen sie einfach einige zusätzliche Informationen über das, was Null war, anstatt die Möglichkeit der NullPointerException ganz zu vermeiden.

Günther: Den Trend, sich an guten und etablierten Features aus anderen Programmiersprachen zu orientieren, kann man seit längerer Zeit beobachten. Insbesondere Scala hat gezeigt, dass ein objektfunktionales Programmiermodell durchaus Stärken hat. Und es ist sicher auch ein Inkubator dafür gewesen, dass mit Java 8 Lambdaausdrücke Einzug in die Sprache gehalten haben. Mit JEP 305: Pattern Matching for instanceof (Preview), mit JEP 359: Records, JEP 361: Switch Expressions und mit JEP 368: Text Blocks, die uns Java 14 bringt, setzt sich dieser Trend – ungeachtet dessen, aus welcher Ecke die Inspiration stammt – fort und reduziert die Lücke zu anderen Programmiersprachen.

Riemer: Kotlin hat meiner Meinung nach schon sehr viel richtig gemacht, wenn es um Entwicklerproduktivität und -Usability geht. Dahingehend finde ich es eine positive Entwicklung, wenn man sich auch nach 25 Jahren

Java noch einem zentralen Thema wie NPEs widmet und, was ja ein erklärtes Ziel des JEPs ist, neuen Entwicklern die Verwirrung im Umgang mit NPEs nehmen will.

Schauder: Ich habe eine sehr ähnliche Frage gehört, als Lambdas eingeführt wurden. Die Vergleichssprachen, die damals angeführt wurden, waren aber Scala und Groovy. Ich kenne weder Groovy noch Kotlin genauer, aber Scala hatte damals schon Sprachfeatures, die über das hinausgehen, was Java jetzt hat. Ich gehe davon aus, dass die Situation mit Kotlin ähnlich ist. Aber eigentlich ist die Prämisse falsch. Programmiersprachen geraten entweder in Vergessenheit oder sie entwickeln sich weiter und kopieren dabei fleißig Features von anderen Sprachen. Java zeigt mit diesem Release, dass es sich fleißig – aber auch sehr besonnen – Dinge von anderen Sprachen abschaut. Es wird immer jede Menge Sprachen geben, die das eine oder andere zusätzliche neue Feature haben, aber das ist irrelevant, solange man nicht Papers über neue Sprachfeatures schreiben will.

Vitz: Ich empfinde die neuen NullPointerExceptions als deutliche Verbesserung. Zu erraten, welche Variable Null war, entfällt mit ihnen, was mir bei mancher Fehlersuche wirklich weiterhilft. Ich finde allerdings nicht, dass sich Java durch genau dieses Feature an Kotlin annähert. Koltins Konzept mit Null umzugehen, ist deutlich ausgefeilter und steckt ja bereits mit im Compiler. Im Allgemeinen ist aber zu beobachten, dass sich das JDK an Konzepten bedient, die in anderen Sprachen, wie auch Kotlin bereits erprobt wurden. Das finde ich sehr gut, da sich die Sprache so weiterentwickelt, ohne jedes neue Feature selbst komplett erproben zu müssen.

JM: Welche Wünsche bzw. Präferenzen habt ihr für das ebenfalls noch in diesem Jahr erscheinende Java 15?

Kabutz: Dass sie Records so schnell wie möglich fertigstellen. Das ganze Konzept scheint gut durchdacht, auch die Art und Weise, wie die Serialisierung funktioniert, ist clever.

Günther: Mit Spannung beobachte ich den Fortschritt der Arbeit an Pattern Matching for Switch, das sich aktuell noch im Status eines JEP Draft befindet. Insbesondere eine sinnvolle Kombination mit den in Java 14 eingeführten Records kann die Art und Weise (Stichwort: Deconstruction Patterns), wie wir künftig Java-Code formulieren, beeinflussen. Aber so sehr ich mir das wünsche, für das noch in diesem Jahr erscheinende Java 15 ist die Verfügbarkeit dieses Features doch eher unrealistisch.



Michael Vitz verfügt über mehrjährige Erfahrung in der Entwicklung, Wartung und im Betrieb von Anwendungen auf der JVM. Aktuell beschäftigt er sich vor allem mit den Themen Microservices, Cloud-Architekturen, DevOps, Spring Framework und Clojure. Als Senior Consultant bei INNOQ hilft er Kunden, warrbare und wertschaffende Software zu entwickeln.

Riemer: Ich bin auf jeden Fall gespannt auf die Features JEP 198: Light-Weight JSON API und JEP 218: Generics over Primitive Types.

Schauder: Ich vertraue da voll und ganz den Leuten, die sich wirklich damit auskennen. Die haben ihren Job in den letzten circa 25 Jahren gut gemacht, die werden das auch weiterhin tun.

Vitz: In den letzten Tagen habe ich erst bewusst bemerkt, dass dem seit JDK 11 vorhandenen HTTP-Client ein Konzept fehlt, um ausgehende Requests oder eingehende Responses zu filtern. Damit lässt sich beispielsweise vor dem Senden automatisiert ein Header mit Trace-ID hinzufügen. Es wäre super, wenn das ergänzt würde. Ich glaube aber, dass das bis jetzt noch nicht auf dem Radar des JDKs aufgetaucht ist.

JM: Was haltet ihr generell von der Beschleunigung, die Java durch den sechsmonatigen Releasezyklus erlebt?

Fischer: Ehrlich gesagt stehe ich ihr mit gemischten Gefühlen gegenüber, sehe sie aber eher positiv, wenn ich bedenke, wie lange es früher gedauert hat, bis neue Sprachfeatures Einzug hielten. Auf der anderen Seite zeigt die Geschichte des JEP 355 für Textblöcke, beginnend mit dem dann zurückgezogenen JEP 326 für Raw String Literals, dass sich nicht alles in Sechsmonatsblö-

cke pressen lässt. Rund wären Textblöcke für mich erst mit Stringinterpolation. Auch die neuen Switch Expressions sind aus meiner Sicht nicht vollkommen, weil sie sich im Fall-Through-Verhalten von der parallel existierenden Switch-Anweisung unterscheiden. Das wird viele Programmierfehler verursachen. Es bleibt dabei: Eile mit Weile.

Kabutz: Es ist ein wenig chaotisch und schwer zu merken, welche Features zu welcher Version gehören. Die meisten Firmen arbeiten noch mit Java 8. Diejenigen, die sich weiterentwickelt haben, verwenden Java 11. Nur sehr wenige sind mutig (oder dumm) genug, um mit der neuesten hochmodernen JVM zu arbeiten.

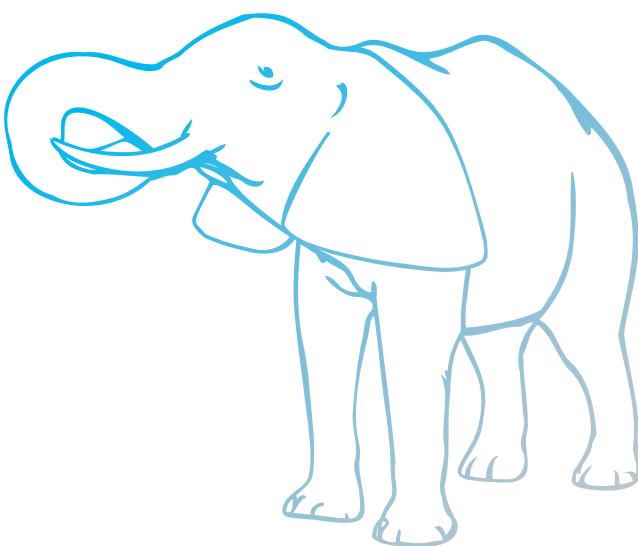
Dennoch macht es mir Spaß, die neuen Funktionen nachzuschlagen und auszuprobieren. Es fühlt sich ein wenig an wie in den frühen Tagen von Java, als wir ziemlich regelmäßig große Fortschritte verbuchen konnten.

Günther: Ich bin zwiegespalten. Zum einen ermöglicht der beschleunigte Releasezyklus die zeitnahe Veröffentlichung wichtiger Features. Das gibt uns das Gefühl, mit einer lebendigen und modernen Sprache zu arbeiten, und ist absolut zu begrüßen. Zum anderen stelle ich aber fest, dass Non-LTS-Releases in der Regel keine besondere Rolle spielen.

Riemer: Durch einen kürzeren Releasezyklus wird das Gefühl erzeugt, dass sich die Sprache stetig weiterentwickelt, wodurch sie präsenter und auch frischer wirkt. Ich erinnere mich noch sehr gut daran, wie ewig sich die Zeit zwischen den Releases von Java 6, Java 7 und Java 8 angefühlt hat.

Schauder: Früher arbeitete ich in einem Konzern, in dem die sogenannte neue Java-Version erst eingesetzt wurde, wenn die alte aus dem Support lief. Jetzt arbeite ich an einem Framework, das in solchen Konzernen oft genutzt wird. D. h. ich arbeite wieder mit relativ alten Versionen und es wird ein Weilchen dauern, bis ich die neuen Features wirklich nutzen kann. Es hat sich also nicht so viel geändert. Überhaupt hat sich doch nicht die Geschwindigkeit geändert, sondern nur die Kadenz, oder?

Vitz: Auch wenn sich das Modell in der Praxis bei meinen Kunden bisher nicht etabliert hat, finde ich die Beschleunigung gut. Sie gibt gerade den JDK-Entwicklern die Möglichkeit, viel früher Feedback einzuholen. In Kombination mit den Experimental-Features erhöht dieses Vorgehen die Wahrscheinlichkeit, gute APIs und Sprachfeatures zu entwickeln, ungemein.



Links & Literatur

[1] <https://www.javaspecialists.eu/>

[2] <https://www.javaspecialists.eu/archive/Issue223.html>

[3] <https://openjdk.java.net/jeps/8213076>

Anzeige

Kolumne: EnterpriseTales

von Lars Kölpin



Cloud-Computing is someone else's computer

Schon lange hören wir von Kunden immer wieder von der Absicht, in die Cloud zu wollen. So viele Befürworter hinter dem Geschäftsmodell stehen, so viele Kritiker stehen ihm gegenüber. Bei Vorfällen hört man dann immer öfter in scherhaftem Ton, dass Cloud-Computing nur „someone else's“ Computer sei. Doch ist das tatsächlich noch so? Steckt hinter Cloud-Computing wirklich nur der Computer von jemand anderem, oder hat sich das Geschäftsmodell in den letzten Jahren gewandelt?

Vom zentralen Team zum dezentralen Team

Früher war das Entwicklerdasein doch so einfach. Der Kunde bzw. die Fachabteilung hatte einen Wunsch, gab uns ein bisschen Zeit, den Wunsch umzusetzen, und ein paar Monate später sahen wir uns mit der fertigen Software wieder. Diese wurde an die Testabteilung übergeben. Nachdem sie nach einer mehrwöchigen Testphase abgenommen wurde, wurde die Software in Produktion genommen. Doch es hat sich etwas verändert. Unternehmen setzen in ihrer IT-Abteilung heutzutage vermehrt auf agile Methoden, Extreme Programming und praktizieren den „Continuous Process“ – um genau zu sein, also das permanente Integrieren von Veränderungen im Code sowie kleine Releasezyklen.

Als der Streaminggigant Netflix das Konzept der Microservices etablierte, gab es eine regelrechte Revolution. Unternehmen wollten keine große Software – genannt Monolith – mehr, sondern kleine, unabhängige Dienste, die gemeinsam eine Softwarelandschaft erge-

ben. Jeder dieser Dienste wird dabei von einem separaten Team betrieben, das einen reibungslosen Ablauf des jeweiligen Dienstes gewährleistet. Frei nach dem Motto: „You build it, you run it.“ Der Entwickler wandelt sich damit einhergehend in vielen Unternehmen immer mehr vom reinen Developer zu einem vollständigen Produktbetreiber eines Dienstes. Damit verbunden migrieren immer mehr Unternehmen ihre Infrastruktur in die Cloud. Doch warum ist das so?

Damit Software Werte für das Unternehmen generieren kann, müssen diese in irgendeiner Weise auf einem physikalischen Computer laufen. Das führt dazu, dass Unternehmen vor dem eigentlichen Release der Software in eine Infrastruktur investieren müssen, indem sie physikalische Rechner anschaffen und dafür sorgen, dass sie störungsfrei laufen. Software, die in Produktion betrieben wird, sollte der Erfahrung nach isoliert, also auf für die Software eigener oder virtualisierter Hardware, betrieben werden, damit ein System bei falscher Konfiguration andere Systeme nicht beeinflusst. Wird die Software von Endkonsumenten besser angenommen als erwartet und wird zu wenig in eine Infrastruktur investiert, muss trotzdem eine Verfügbarkeit sichergestellt werden. Wird im Gegensatz dazu das Produkt weniger gut angenommen, wurde für die bereitgestellte Infrastruktur unnötiges Geld ausgegeben. Die perfekte Investition für das Betreiben der Rechner ist deshalb kaum vorherzusagen. Wird die Software international verwendet, muss darüber hinaus eine gute Antwortzeit in den verschiedenen Regionen der Welt garantiert werden. Läuft der Speicher des Servers voll, muss er erweitert werden. Damit sichergestellt werden kann, dass keine Daten verloren gehen, darf das regelmäßige Erstellen eines Backups natürlich nicht fehlen. Die Kosten der Software bestehen also nicht nur aus der reinen Entwicklung, sondern beinhalten auch versteckte Kosten von Hardware, Monitoring und Wartung der Hardware. Das sind Kostenpunkte, die vor allem durch

Porträt



Lars Kölpin ist Enterprise-Entwickler bei der OPEN KNOWLEDGE GmbH in Oldenburg. Zu seinen Leidenschaften gehören moderne Frontend-Technologien, vor allem im Zusammenspiel mit modernen Cloud-Architekturen.

die vielen kleinen Teams und Dienste, die der Microservices-Ansatz mit sich bringt, multipliziert werden.

Cloud-Computing damals

Amazon erfuhr die Probleme im Rahmen des Weihnachtsgeschäfts am eigenen Leib und extrahierte 2006 die Amazon Web Services (AWS) und damit das Geschäftsmodell des Cloud-Computings. Mit dem Dienst war es erstmals möglich, Rechenpower im Minutentakt dynamisch zu mieten – und vor allem wieder zu kündigen. Die Hardware, auf der die Software läuft, die zu Stoßzeiten, wie beispielsweise zur Weihnachtszeit, mehr Rechenpower benötigte, konnte auf Knopfdruck oder sogar vollautomatisch skaliert werden. Infrastructure as a Service war geboren. Aber das Modell bringt Probleme mit sich. Zwar wird die Infrastruktur bereitgestellt, jegliche Software, die auf der Infrastruktur betrieben wird, wie beispielsweise das Betriebssystem und die auf dem System laufende Laufzeitumgebung, müssen jedoch selbst gepflegt werden. Das Administrieren und Konfigurieren der Systeme bleiben deshalb nicht aus. Neue Entwickler im Team müssen deshalb häufig zusätzlich zu dem gewählten Tech-Stack auch noch Fähigkeiten in der Administration von Linux-Systemen oder Ähnlichem lernen.

Es folgten deshalb kurze Zeit später die ersten Services, die statt reiner Hardware nun Hardware inklusive Laufzeitumgebung anboten. Dazu zählen z. B. Heroku oder der Dienst der Google Cloud Platform AppEngine, bei denen das reine Artefakt einer Software weltweit betrieben werden kann. Die Dienste bieten die Laufzeitumgebung in einer bestimmten Version für das Artefakt (im Fall von Java: JAR-Dateien) an. Das konkrete Betriebssystem, inklusive Sicherheitsupdates und Ähnlichem, wird komplett vom Anbieter verwaltet. Dabei wird ein tieferes Verständnis des Betriebssystems, wie beispielsweise Linux, nicht weiter benötigt. Die darunterliegende Hardware und deren Skalierung werden abstrahiert.

Auf die Spitze treibt Amazon das Konzept der verwalteten Hardware und Software mit dem Dienst Lambda und dem damit verbundenen Konzept von Function as a Service, bei dem einzelne Funktionen auf der Infrastruktur von Amazon hochgefahren, ausgeführt und dann direkt wieder runtergefahren werden. In Rechnung gestellt werden die reine Ausführungszeit sowie die Anzahl der Ausführungen. Als Auslöser dieser Funktionen dienen AWS interne Ereignisse oder externe Auslöser wie HTTP-Anfragen.

Die Herangehensweise, dass Server nicht mehr im eigenen Rechenzentrum betrieben, sondern (ggf. virtuell) angemietet werden, hat sich mittlerweile etabliert. Bei dem Blick darauf, was die internen Rechenzentren so tun, fiel allerdings schnell auf, dass es nicht nur das Betreiben von Servern war, sondern auch das Betreiben von Datenbanken. Daher war es eine gravierende Veränderung, dass bei den gängigen Anbietern Datenbanken wie PostgreSQL, MySQL oder Oracle mit wenigen Klicks erstellt und betrieben werden konnten – automatische Backups inklusive. Was gut für die nutzenden Un-

Anzeige

ternehmen ist, stellt ein Problem für die Cloud-Anbieter dar. Sich vom Konkurrenten in diesem Bereich abzuheben, ist nur begrenzt möglich, weshalb die Anbieter das Angebot häufig um Eigenentwicklungen für ihre spezifischen Anwendungsfälle erweiterten. Ein solcher Dienst ist zum Beispiel DynamoDB, eine verwaltete NoSQL-Datenbank, die das Problem einer weltweit skalierbaren Datenhaltung löst, dafür aber mit Eigenheiten daherkommt.

Cloud-Computing heute

Durch die positive Annahme der spezifischen Dienste ist ein weiterer Trend zu beobachten: Neben dem Angebot von Infrastruktur und klassischen Anwendungen wie Datenbanken und Datenhaltung gibt es immer mehr Dienste, die ein Wissen für bestimmte Nischen abdecken.

Nimmt man als Beispiel dazu den Dienst Recognition von Amazon, so bietet dieser den Kunden Zugriff auf ein riesiges Machine-Learning-Netz, um Bilder und Videos zu analysieren, ohne sich tiefer mit dem Thema auskennen zu müssen. Mit Textract kann der Traum vom „paperless Office“ realisiert werden. Der Dienst kann gescannte Dokumente analysieren und auswerten, ohne dass ein Mensch dazu die Daten mühselig in ein Computersystem übertragen muss. Das Spannende: Theoretisch muss dabei nicht eine Zeile Code geschrieben werden.

Die wenigsten Applikationen kommen ohne Querschnittsfunktionen, wie z. B. eine Authentifizierung, aus. Ohne tieferes kryptografisches und mathematisches Verständnis ist es meist schwer möglich, eine wirklich sichere Benutzerverwaltung zu implementieren. Nicht selten werden deshalb mehrere Wochen zum Feinjustieren investiert, ohne einen wirklichen Mehrwert der Software für Unternehmen zu bieten. Durch das Einbinden der entsprechenden Dienste, wie beispielsweise Amazons Cognito, steht dem System im Gegensatz dazu mit wenig Code ein kompletter Authentifizierungsserver – inklusive granularer Rechteverwaltung für die eigenen Dienste des Cloud-Anbieters – zur Verfügung. Es ist also ein klarer Trend zu erkennen: Cloud-Computing wandelt sich immer mehr vom Anbieter reiner Infrastruktur zum Anbieter von Lösungen.

Das Kaufen von fertiger Software zur Integration in die eigene Systemlandschaft und damit verbundenem Wissen, das nicht der eigenen Kernkompetenz entspricht, ist keine neue Idee. In der Vergangenheit bedeutete das aber immer, dass solche Software im eigenen Rechenzentrum betrieben werden musste und sehr viel Aufwand in die Integration der heterogenen Teilsysteme geflossen ist. Die Kosten für diesen Integrationsaufwand sind häufig nicht abzuschätzen, obwohl sie eigentlich bei der Kalkulation auf die Anschaffungs- und Lizenzkosten der Drittanbieter aufgeschlagen werden müssten. Vor diesem Hintergrund ist es besonders interessant, dass der Fokus der Cloud-Anbieter in den vergangenen Jahren neben dem Anbieten der Dienste zunehmend auf deren wechselseitiger Integration liegt. Wer sich für einen Cloud-Anbieter entscheidet, kauft damit nicht nur

spezifisches Wissen und Lösungen, sondern ein ganzes Ökosystem von integrierten Diensten ein.

Wo es vor einigen Jahren noch undenkbar war, Dienste, wie beispielsweise einen Dokumentenscanner oder Bilderkennung, außerhalb des eigenen Rechenzentrums zu betreiben, stehen diese heute mit einem einzigen Klick und ohne eine vorherige große monetäre Investition direkt zur Verfügung. Unternehmen erhalten Zugriff auf eine riesige Sammlung von Infrastruktur und Wissen der Giganten von Google und Amazon. Sie zahlen nur das, was sie wirklich brauchen, und müssen nicht einmal die darunterliegende Rechenpower betreiben.

Aber wie heißt es doch so schön: Es ist nicht alles Gold, was glänzt. Nicht zuletzt zeigen Vorfälle wie der bei Microsofts Azure [1], dass die Nutzung vollständig verwalteter Dienste nicht ganz ohne Risiken daherkommt. Dort wurden Anfang 2019 automatisiert Datensätze aus Datenbanken gelöscht und waren unwiederbringlich verloren. Durch das Einkaufen und Nutzen der Dienste und das damit verbundene Wissen verliert das eigene Unternehmen einerseits diese Kompetenzen, und erhöht andererseits zugleich die Abhängigkeit vom Anbieter. Weil ein Dienst selten allein daherkommt, und diese Dienste oft miteinander verzahnt bzw. sehr einfach mit anderen Diensten zu integrieren sind, kaufen sich Unternehmen häufig unbewusst in das Ökosystem der Cloud-Anbieter ein. Damit macht sich das eigene Unternehmen abhängig von dessen Preispolitik und Service Level Agreements. Auch deshalb entschied sich z. B. der Speicherdienst Dropbox in den letzten Jahren, die Daten aus dem Simple Storage Service von Amazon in ein eigenes Datenzentrum mit eigener Software zu migrieren [2].

Fazit

Also ist Cloud-Computing tatsächlich nur „someone else's“ Computer? Meiner Meinung nach ist diese Aussage nicht ganz einfach zu beantworten. Im Allgemeinen muss zunächst zwischen Infrastruktur und angebotenen Diensten unterschieden werden. Betrachtet man den infrastrukturellen Aspekt, so ist Cloud-Computing nicht nur „someone else's“ Computer, sondern eher „someone else's globally distributed datacenter with engineers working 24/7 to ensure its availability“. Dennoch darf nicht vergessen werden: Es herrscht die Einschränkung, dass Kompetenzen abgegeben werden. Gibt es einen Vorfall im Datenzentrum, ist man selbst zunächst machtlos. Betrachtet man im Gegenzug den Wandel und setzt ein Unternehmen vermehrt auf die Dienste eines Anbieters, so kauft sich dieser eher „someone else's knowledge“ und „someone else's ecosystem“ ein. Und das ist als eine Chance und ein Risiko zugleich zu verstehen.

Links & Literatur

[1] <https://www.golem.de/news/microsoft-azure-loescht-aus-versehen-datenbanken-von-kunden-1901-139101.html>

[2] <https://t3n.de/news/weggang-aws-dropbox-spart-75-962140/>

Anzeige

Nichtfunktionale Anforderungen: Struktur, Organisation und Wartung

Qualität ist besser!

Um der Definition von Codequalität gerecht zu werden, genügt es nicht, sich einzig auf den Quelltext zu konzentrieren. Auch Struktur, Organisation und Wartung spielen eine wichtige Rolle.

von Marco Schulz

Aus Erfahrung wissen die meisten von uns, wie schwierig es ist, auszudrücken, was wir unter Qualität verstehen. Warum ist das so? Es gibt viele verschiedene Ansichten zum Thema Qualität, und jede von ihnen hat ihre Bedeutung. Im Kontext eines Projekts muss die Definition von Qualität mit den Bedürfnissen und dem Budget des Projekts übereinstimmen. Der Versuch, Perfektionismus zu erreichen, kann kontraproduktiv sein, wenn ein Projekt erfolgreich beendet werden soll. Wir werden unsere Überlegungen auf der Grundlage eines 1976 von Barry W. Boehm verfassten Aufsatzes [1] beginnen. Boehm beleuchtet die verschiedenen Aspekte der Softwarequalität und den zugehörigen Kontext. Schauen wir uns das etwas genauer an.

Wenn wir über Qualität sprechen, sollten wir uns auf drei Bereiche konzentrieren: Codestruktur, Korrektheit der Implementierung und Wartbarkeit. Viele Manager kümmern sich nur um die ersten beiden Aspekte, der Bereich Wartung wird von ihnen meist vollständig vernachlässigt. Das ist gefährlich, da das Risiko einer Investition für Unternehmen in die individuelle Entwicklung einer Anwendung nicht eingegangen wird, um diese nur für kurze Zeit in Produktion zu nutzen. Abhängig von der Komplexität der Anwendung kann der Preis für die Erstellung leicht mehrere Hunderttausend Euros erreichen. Da ist es mehr als verständlich, dass der erwartete Geschäftsnutzen solcher Aktivitäten als hoch eingeschätzt wird. Eine Lebensdauer von zehn Jahren und mehr in der Produktion ist üblich. Um die Vorteile auch künftig zu sichern, sind Anpassungen obligatorisch. Das impliziert einen starken Fokus auf die Wartung. Sauberer Code bedeutet nicht, dass sich Ihre Anwendung einfach ändern kann. Ein sehr leicht ver-

ständlicher Artikel [2], der dieses Thema berührt, wurde von Dan Abramov geschrieben. Bevor wir weiter darauf eingehen, wie Wartung definiert werden kann, werden wir den ersten Punkt diskutieren: die Struktur.

Einrüstarbeiten

Ein häufig unterschätzter Aspekt in Entwicklungsabteilungen ist ein fehlender Standard für Projektstrukturen. Eine klare Definition, wo Dateien abgelegt werden müssen, hilft Teammitgliedern, sogenannte Points of Interest (POI) schnell zu identifizieren. Eine solche Metastruktur für Java-Projekte wird beispielsweise vom Build-Werkzeug Maven definiert. Vor mehr als einem Jahrzehnt haben Unternehmen den Einsatz von Maven in ihren Projekten ausprobiert und das Tool an die vorhandene Verzeichnisstruktur angepasst. Das führte zu umfangreichen Wartungsarbeiten, da im Laufe der Zeit immer mehr Infrastrukturtools für die Softwareentwicklung hinzugekommen sind, wie beispielsweise SonarQube zur Codeanalyse. Solche Tools arbeiten nach dem von Maven definierten Standard. Das bedeutet, dass jede Anpassung des Standards an eigene Strukturen den Erfolg der Integration neuer Tools oder des Austauschs eines vorhandenen Tools gegen ein anderes beeinflusst.

Ein weiterer zu betrachtender Aspekt ist die unternehmensweit definierte Metaarchitektur. Wenn möglich, sollte jedes Projekt der gleichen Metaarchitektur folgen. Das reduziert die Zeit, die ein neuer Entwickler benötigt, um einem bestehenden Team beizutreten und produktiv zu werden. Eine solche Metaarchitektur muss für Adaptionen offen sein, was in zwei einfachen Schritten erreicht werden kann:

- Kümmern Sie sich nicht um zu viele Details.
- Folgen Sie dem KISS-Prinzip (Keep it simple, stupid!).

Wenn andere Menschen Ihre Absichten leicht nachvollziehen können, akzeptieren sie eine Empfehlung eher, da sie unter den genannten Umständen einleuchtend ist.

Ein klassisches Muster, das gegen das KISS-Prinzip verstößt, besteht darin, dass Standards stark angepasst werden. Ein sehr gutes Beispiel für die fatalen Auswirkungen einer starken Anpassung beschreibt George Schlossnagle [3]. In Kapitel 21 seines Buchs erklärt er die Probleme, die für ein Team entstanden, als der ursprüngliche PHP-Kern angepasst wurde, anstatt dem empfohlenen Weg zu folgen und Erweiterungen zu nutzen. Das führte dazu, dass jedes Update der PHP-Version manuell bearbeitet werden musste, um die eigenen Entwicklungen in den neuen Kern aufzunehmen. In diesem Zusammenhang bilden die besprochenen Maßnahmen hinsichtlich Struktur, Architektur und KISS bereits drei Qualitätsziele, die einfach umzusetzen sind.

Das auf GitHub gepostete Open-Source-Projekt TP-CORE [4] befasst sich mit den oben genannten Punkten: Struktur, Architektur und KISS. Dort wird gezeigt, wie Sie den beschriebenen Ansatz in die Praxis umsetzen können. Diese kleine Java-Bibliothek hält die Maven-Konvention mit ihrer Verzeichnisstruktur streng ein. Das vereinfacht unter anderem auch die Komplexität der zu schreibenden Build-Logik. Zur schnellen Kompatibilitätserkennung werden Releases durch semantische Versionierung [5] definiert. Als Architektur wurde das Schichtenmodell gewählt, das wir nachfolgend in den wichtigsten Punkten erläutern. Eine ausführliche Beschreibung findet sich unter [6].

Die wichtigsten Architekturentscheidungen basieren auf folgenden Überlegungen: Jede Ebene wird durch ein eigenes Paket definiert und die Dateien folgen ebenfalls einer strengen Regel. Es wird kein spezielles Prä- oder Postfix verwendet. Die Logger-Funktionalität wird beispielsweise von einer Schnittstelle namens *Logger* und der entsprechenden Implementierung *LogbackLogger* deklariert. Die APIs können im Paket business und in den Implementierungsklassen im Paket application gefunden werden. Namen wie *ILogger* und *LoggerImpl* sollten vermieden werden. Stellen Sie sich ein Projekt vor, das vor zehn Jahren gestartet wurde und dessen *LoggerImpl* auf Log4j basiert. Jetzt entsteht eine neue Anforderung und das Loglevel muss zur Laufzeit aktualisiert werden. Um diese Herausforderung zu lösen, könnte die Log4j-Bibliothek durch Logback ersetzt werden. Jetzt ist es verständlich, warum es eine gute Idee ist, die Implementierungsklasse wie die Schnittstelle zu benennen und das mit den Implementierungsdetails zu kombinieren – es erleichtert die Wartung erheblich! Gleiche Konventionen finden Sie auch im Java-Stan-

dard-API. Das Interface *List* wird von einer *ArrayList* implementiert. Offensichtlich ist die Schnittstelle nicht als *IList* und die Implementierung nicht als *ListImpl* benannt.

Zusammenfassend wurde hier ein vollständiger Regelsatz definiert, der auch messbar ist und unser Verständnis der strukturellen Qualität beschreibt. Erfahrungsgemäß sollte diese Beschreibung kurz sein. Wenn andere Menschen Ihre Absichten leicht nachvollziehen können, akzeptieren sie eine Empfehlung eher, da sie unter den genannten Umständen einleuchtend ist. Darüber hinaus erkennt der Architekt Regelverstöße weitaus schneller.

Erfolgsmesslatte

Der schwierigste Teil ist es, sauberen Code über einen langen Zeitraum zu erhalten und vor Erosion zu schützen. Einige Ratschläge sind an sich nicht schlecht, aber im Kontext Ihres Projekts möglicherweise nicht so nützlich. Daher habe ich nachfolgend einige Grundregeln zusammengestellt, die in der Praxis meist schon angewendet werden.

Meiner Meinung nach wäre die wichtigste Regel, immer die Compilerwarnung zu aktivieren, egal welche Programmiersprache Sie verwenden. Alle Compilerwarnungen müssen behoben werden, wenn eine Version vorbereitet wird. Organisationen wie die NASA, die sich mit kritischer Software befassen, wenden diese Regel in ihren Projekten strikt an, was zu äußerst erfolgreichen Resultaten führt.

Codierungskonventionen zu Klassenbenennung, Zeilenlänge und API-Dokumentation wie Javadoc können von Tools wie Checkstyle einfach definiert und eingehalten werden. Dieser Prozess kann während des Builds vollautomatisch ausgeführt werden. Aber Achtung: Selbst wenn die Codeprüfungen ohne Warnungen bestanden werden, bedeutet das nicht, dass alles optimal funktioniert. Javadoc ist beispielsweise problematisch. Mit einem automatisierten Checkstyle kann sichergestellt werden, dass die API-Dokumentation vorhanden ist, obwohl wir keine Ahnung von der Qualität der Beschreibungen haben.

An dieser Stelle sollte es nicht erforderlich sein, die Vorteile von Tests zu erörtern. Lassen Sie mich lieber einen Überblick über die Testabdeckung geben. Der Industriestandard von 85 Prozent Testabdeckung sollte eingehalten werden, da eine Abdeckung von weniger als 85 Prozent die komplexen Teile Ihrer Anwendung nicht erreicht. Eine hundertprozentige Abdeckung be-

lastet lediglich Ihr Budget, ohne dass sich daraus weitere Vorteile ergeben. Ein Paradebeispiel dafür ist das TP-CORE-Projekt, dessen Testabdeckung meist zwischen 92 und 95 Prozent liegt. Das wurde getan, um zu verdeutlichen, ab wann sich mit vertretbarem Aufwand keine aussagekräftigen Tests mehr schreiben lassen. Auch Mock-Objekte würden keine weitere Aussagekraft bei den Tests bringen.

Wie bereits erläutert, enthält die Businessschicht nur Schnittstellen, die das API definieren. Diese Ebene ist ausdrücklich von der Code Coverage ausgeschlossen. Ein anderes Paket heißt internal und enthält versteckte Implementierungen wie den SAX *DocumentHandler*. Aufgrund der Abhängigkeiten, an die der *DocumentHandler* gebunden ist, ist es selbst mit Mocks sehr schwierig, diese Klasse direkt zu testen. Das ist an sich auch völlig unproblematisch, da ihr Zweck nur der interne Gebrauch ist. Darüber hinaus wird die Klasse von der Implementierung des *DocumentHandler* implizit getestet. Um eine höhere Abdeckung zu erreichen, könnte es auch eine Option sein, alle internen Implementierungen von Überprüfungen auszuschließen. Es ist jedoch immer eine gute Idee, die implizite Abdeckung dieser Klassen zu beobachten, um Aspekte zu erkennen, die sonst möglicherweise nicht erkannt werden.

Neben den Unit-Tests auf niedriger Ebene sollten auch automatisierte Abnahmetests [7] durchgeführt werden. Wenn Sie diese Punkte genau beachten, können Sie eine Vielzahl von Problemen vermeiden. Aber vertrauen Sie vollautomatischen Prüfungen niemals blind: Regelmäßig wiederholte manuelle Codeinspektionen sind immer obligatorisch, insbesondere bei der Arbeit mit externen Anbietern. In unserem Vortrag auf der JCON 2019 [8] haben wir gezeigt, wie einfach die Testabdeckung gefälscht werden kann. Um andere Schwachstellen zu

erkennen, können Sie zusätzliche Checker wie etwa SpotBugs ausführen. Tests zeigen nicht an, dass eine Anwendung fehlerfrei ist, sie zeigen jedoch ein definiertes Verhalten für implementierte Funktionen an.

SCM-Suites wie GitLab oder Microsoft Azure unterstützen seit einiger Zeit Pull-Anfragen, die vor langer Zeit auf GitHub eingeführt wurden. Diese Workflows sind per se nichts Neues. IBM Synergy verwendete eine ähnliche Strategie. Ein Build Manager war dafür verantwortlich, die Änderungen der Entwickler in der Codebasis zusammenzuführen. Das führte dazu, dass alle vom Entwickler durchgeführten Revisionen vom Build Manager in das Repository aufgenommen wurden. Der Build Manager verfügte nicht über ausreichend fundierte Kenntnisse, um die Implementierungsqualität beurteilen zu können. Es wurde schnell zur üblichen Praxis, die Commits durchzuwinken und lediglich darauf zu achten, dass der Build nicht kaputt geht und die Kompliierung immer ein Artefakt erzeugt.

Unternehmen haben als neue Strategie Pull-Request entdeckt, um sie in ihren Projekten einzusetzen. Den dahinterstehenden Dictatorship-Workflow, wie er zu Recht in Open-Source-Projekten angewendet wird, hinterfragen sie hingegen nicht auf Sinnhaftigkeit. Denn das sagt implizit, dass dem eigenen Entwicklungsteam weder Vertrauen entgegengebracht noch Kompetenz zugetraut wird. Heutzutage treffen Manager häufig die Entscheidung, Pull-Requests als Qualitätsschranke zu verwenden. Nach meiner Erfahrung verlangsamt das die Produktivität, da es einige Zeit dauert, bis die Änderungen in der Codebasis verfügbar sind. Das Verständnis von Branch- und Merge-Mechanismen hilft Ihnen bei der Entscheidung für die Nutzung eines einfachen Branch-Modells, wie beispielsweise Release Branch Lines. Auf diesen Branches arbeiten dann Tools wie SonarQube, um das geforderte Qualitätsziel zu sichern. Wenn ein Projekt einen orchestrierten Build mit einer definierten Reihenfolge für die Erstellung von Artefakten benötigt, haben Sie einen starken Hinweis auf eine notwendige Refaktorierung.

Die Kopplung zwischen Klassen und Modulen wird oft unterschätzt. Es ist sehr schwierig, eine automatisierte Visualisierung für die Bindungen von Modulen zu haben. Sie werden sehr schnell feststellen, welchen Effekt es hat, wenn eine starke Kopplung aufgrund zunehmender Komplexität Ihre Build-Logik unnötig verkompliziert.

Wiederholungstäter

Seien Sie versichert, nichts ist so sicher wie die Änderung! Es ist eine Herausforderung, eine Anwendung für Anpassungen offen zu halten. Einige der vorherigen Empfehlungen haben implizite Auswirkungen auf die zukünftige Wartbarkeit. Eine gute Source-Qualität vereinfacht das Unterfangen durchaus. Es gibt jedoch keine Garantien. Im schlimmsten Fall ist das Ende des Produktlebenszyklus (EOL) erreicht, wenn obligatorische Verbesserungen oder Änderungen beispielsweise

Checkliste

- Befolgen Sie etablierte Standards
- KISS – Keep it simple, stupid!
- Gleiche Verzeichnisstruktur für verschiedene Projekte
- Einfache Metaarchitektur, die weitgehend in anderen Projekten wiederverwendet werden kann
- Codierungsstyles definieren und befolgen
- Es werden keine Compilerwarnungen akzeptiert
- Testabdeckung um die 85 Prozent
- Vermeiden Sie Bibliotheken von Drittanbietern soweit möglich
- Verwenden Sie nicht mehr als eine Technologie für ein bestimmtes Problem (z. B. JSON)
- Decken Sie Fremdcode durch ein Entwurfsmuster ab
- Vermeiden Sie starke Objekt-Modul-Kopplungen

Eine lose Kopplung bringt zahlreiche Vorteile für die Wartung und Wiederverwertung mit sich. Dieses Ziel zu erreichen ist nicht so schwer, wie es auf den ersten Blick erscheinen mag.

wegen einer erodierten Codebasis nicht mehr realisiert werden können. Das tritt weit häufiger auf, als man annimmt.

Wie bereits erwähnt, bringt eine lose Kopplung zahlreiche Vorteile für die Wartung und Wiederverwendung mit sich. Dieses Ziel zu erreichen ist nicht so schwierig, wie es auf den ersten Blick erscheinen mag. Versuchen Sie zunächst, die Verwendung von Drittanbieterbibliotheken so weit wie möglich zu vermeiden. Nur um zu überprüfen, ob ein String leer oder null ist, ist es nicht notwendig, sich von einer externen Bibliothek abhängig zu machen. Diese wenigen Zeilen sind schnell selbst formuliert. Ein zweiter wichtiger Punkt, der in Bezug auf externe Bibliotheken berücksichtigt werden muss: Nur eine Bibliothek zur Lösung eines Problems. Wenn beispielsweise im Projekt JSON verwendet wird, entscheiden Sie sich für eine Implementierung und integrieren Sie nicht verschiedene Artefakte mit gleicher Funktionalität. Das wirkt sich auch stark auf die Sicherheit aus: Ein Artefakt von Drittanbietern, dessen Verwendung wir vermeiden können, kann keine Sicherheitslücken verursachen.

Zudem ist es nach einer Entscheidung für eine externe Bibliothek sehr hilfreich, diese für das gesamte Projekt zu kapseln. Das gelingt über die Verwendung von Entwurfsmustern wie Proxy, Fassade oder Wrapper und gestattet einen einfacheren Austausch, da die Codeänderungen so nicht gleich über die gesamte Codebasis verteilt werden. Sie müssen also nicht alles auf einmal ändern, wenn Sie den Anweisungen zum Benennen der Implementierungsklasse und Bereitstellen einer Schnittstelle folgen. Obwohl ein SCM auf Zusammenarbeit ausgelegt ist, gibt es Einschränkungen, wenn mehr als eine Person dieselbe Datei bearbeitet. Durch die Verwendung eines Entwurfsmusters zum Verstecken von Information können Sie Ihre Änderungen iterativ aktualisieren.

Fazit

Wie wir gesehen haben, ist eine nichtfunktionale Anforderung nicht so schwer zu beschreiben. Mit einer kurzen Checkliste (Kasten: „Checkliste“) können Sie die wichtigsten Aspekte für Ihr Projekt klar definieren. Es ist nicht erforderlich, alle Punkte für jeden Code-Commit im Repository zu überprüfen. Das würde höchstwahrscheinlich nur die Kosten erhöhen und führt nicht zu mehr Vorteilen. Eine vollständige Überprüfung etwa wenige Tage vor einem Release stellt in einem agilen Kontext eine effektive Lösung dar. POIs zur Qualitäts-

sicherung sind die Überarbeitungen der Codebasis für ein Release. Das gibt Ihnen eine vergleichbare Statistik und hilft, die Zuverlässigkeit von Schätzungen zu erhöhen. Schlussendlich sollte das Hauptanliegen sein, ein hohes Maß an Automatisierung in der Infrastruktur zu erreichen; Continuous Integration z. B. ist äußerst hilfreich, befreit Sie jedoch nicht von manuellen Code Reviews.



Marco Schulz studierte an der HS Merseburg Diplominformatik. Sein persönlicher Schwerpunkt liegt auf Softwarearchitekturen, der Automatisierung des Softwareentwicklungsprozesses und dem Softwarekonfigurationsmanagement. Seit über fünfzehn Jahren entwickelt er für namhafte Unternehmen auf unterschiedlichen Plattformen umfangreiche Webapplikationen. Er ist freier Consultant, Trainer und Autor verschiedener Fachartikel.



<https://enRebaja.wordpress.com>



marco.schulz@outlook.com



@ElmarDott

Links & Literatur

- [1] <https://csse.usc.edu/TECHRPTS/1976/usccse76-501/usccse76-501.pdf>
- [2] <https://overreacted.io/goodbye-clean-code/>
- [3] Schlossnagle, George: „Advanced PHP Programming: Developing Large-Scale Web Applications with PHP 5“, Pearson Education, 2007
- [4] <https://github.com/ElmarDott/TP-CORE>
- [5] <https://semver.org/>
- [6] <https://enrebaja.wordpress.com/together-platform/>
- [7] <https://dzone.com/articles/acceptance-tests-in-java-with-jgiven>
- [8] <https://www.youtube.com/watch?v=V0wVfG0tpM>

Anzeige

Anzeige

Teil 2: Sortier- und Suchalgorithmen – ein Überblick

Algorithmen in der Java-Praxis

Sortier- und Suchalgorithmen gehören zu den Basisalgorithmen und sind in vielen Klassenbibliotheken implementiert. Muss man sehr umfassende Datenbestände sortieren oder in diesen suchen, spielt die Vorgehensweise aus Gründen der Performance eine wichtige Rolle. Parallel Computing kann das Sortieren beschleunigen. Auch diese Verfahren basieren auf den Basisverfahren. Es gibt also genügend Gründe, diesen auf den Grund zu gehen.

von Dr. Veikko Krypczyk und Elena Bochkor

In diesem Teil der Artikelserie beschäftigen wir uns mit Sortier- und Suchverfahren in Theorie und Praxis. Die theoretischen Erkenntnisse befähigen uns zum einen, die Verfahren zu verstehen, und zum anderen, deren Leistungsfähigkeit, insbesondere mit Blick auf die Verarbeitungsgeschwindigkeit, zu beurteilen. Man sollte zunächst das Chaos sortieren, bevor man sich an die Suche macht. Was im Alltag gilt, hat auch seine Berechtigung in der Informatik. Starten wir mit den Sortieralgorithmen und gehen dann über zum Suchen in den Daten.

Grundlagen des Sortierens

Mit Hilfe eines Algorithmus möchte man eine Menge von Objekten in eine definierte Reihenfolge bringen. Um welche Art von Objekten es sich handelt, spielt dabei keine Rolle. Im einfachsten Fall handelt es sich um eine Menge (Liste) von Zahlen, die zum Beispiel in eine aufsteigende Reihenfolge gebracht werden sollen. Grundsätzlich können wir jedoch beliebige Objekte sortieren. Voraussetzung dafür ist, dass es ein Sortierkriterium gibt, nach dem das Sortieren erfolgt. Dieses Sortierkriterium (Key) muss es erlauben, dass man die Objekte miteinander vergleichen kann. Es müssen also Vorgänger- und Nachfolgerbeziehungen zwischen zwei zu vergleichenden Ele-

menten bezüglich des Sortierkriteriums herstellbar sein. Typische Beispiele sind die lexikographische Ordnung von Zeichenketten oder die numerische Ordnung von Zahlen. Diese Forderung wird als Trichotomie bezeichnet und besagt, dass für alle Elemente einer Menge bezüglich deren Schlüssel a und b gilt:

- $a < b$ oder
- $a = b$ oder
- $a > b$

Die Transitivität gewährleistet, dass die Sortierung widerspruchsfrei durchgeführt wird: Ist das Element a vor dem Element b und das Element b vor dem Element c in der Liste einzusortieren, so folgt unweigerlich, dass das Element a vor dem Element c anzugeordnen ist. Eine andere Reihenfolge wäre unlogisch. Für den Datentyp Zahl muss man diese Vergleichsoperationen nicht explizit implementieren, da sie implizit in jeder Programmiersprache vorhanden sind. Mit anderen Worten: Dass der Zusammenhang 2 ist kleiner als 3 gilt, ist offensichtlich und führt in der Logik der Programmiersprache ($2 < 3$) zu einer wahren Aussage. Möchte man andere (beliebige) Objekte miteinander vergleichen, muss man die Vergleichsoperationen explizit programmieren. In Java würde man das zum Beispiel tun, indem man die Klassen *Comparable* und *Comparator* nutzt. Dabei bedeuten:

- **Comparable:** Implementiert eine Klasse *Comparable*, so können sich die Objekte selbst mit anderen Objekten vergleichen.
- **Comparator:** Eine implementierende Klasse, die sich *Comparator* nennt, nimmt zwei Objekte an und vergleicht sie.

Algorithmen – State of the Art und Praxis mit Java

Teil 1: Algorithmen als Kernelemente des Programms – Definition und Klassen

Teil 2: Sortier- und Suchalgorithmen – ein Überblick

Teil 3: Metaheuristiken zur Lösung komplexer Probleme

Teil 4: Ausgewählte Bibliotheken für Java – schneller in der Praxis

Im ersten Fall ist die Vergleichsfunktion Bestandteil der Funktionalität der Klasse des Objekts, im zweiten Fall ist der Vergleich nicht in der Klasse, sondern extern im *Comparator* implementiert. Mit *Comparable* kann man nur ein Vergleichskriterium pro Datenklasse implementieren, da es nur eine Methode *compareTo(...)* gibt. Mittels *Comparator* kann man dagegen mehrere Vergleichskriterien integrieren, sofern das notwendig ist. Das Interface *Comparable* hat folgende Syntax:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

Die Klasse *Comparator* ist wie folgt definiert:

```
public interface Comparator {
    int compare(Object o1, Object o2);
    boolean equals(Object o);
}
```

Betrachten Sie dazu Listing 1. Wir implementieren eine Datenklasse namens *Person*, die das Interface *Comparable* implementiert. Diese muss nun die Methode *compareTo(...)* beinhalten. Diese Methode nimmt den Vergleich der Werte des Objekts mit den Werten eines anderen Objektes der gleichen Datenklasse, hier der Klasse *Person* vor. Die Vergleichsoperation ist damit also Bestandteil der Datenklasse. In Listing 2 haben wir dagegen eine eigene Klasse *ComparatorForPerson* definiert. Diese übernimmt das Vergleichen von zwei Objekten der Klasse *Person*. Das Ergebnis des Vergleichs wird bei beiden Varianten über den ganzzahligen Rückgabewert (-1; 0; 1) mit der Bedeutung (kleiner als, gleich, größer als) angegeben.

Wichtige Gütemerkmale eines Sortieralgorithmus sind die Zahl der durchschnittlich benötigten Iterationen, bis die gewünschte Reihenfolge der Elemente vorliegt, und

Listing 1: Datenklasse, die das Interface Comparable implementiert

```
public class Person implements Comparable<Person> {
    private String name;
    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public int compareTo(Person o) {
        if(o == null) return -1;
        else if(o.getName() == null && name == null) return 0;
        else if(o.getName() != null && name == null) return -1;
        else if(o.getName() == null && name != null) return 1;
        else return name.compareTo(o.getName());
    }
}
```

Listing 2: Klasse ComparatorForPerson zum Vergleichen der Objekte der Klasse Person

```
public class ComparatorForPerson implements java.util.Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        if(o1 == null && o2 == null) return 0;
        else if(o1 == null && o2 != null) return -1;
        else if(o1 != null && o2 == null) return 1;
        else return o1.getName().compareTo(o2.getName());
    }
}
```

Anzeige

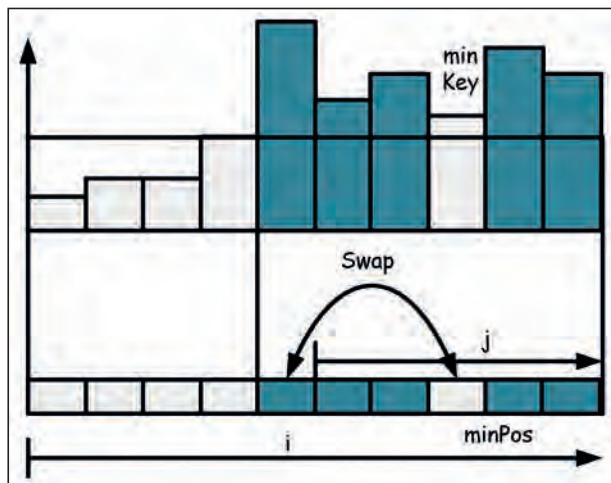


Abb. 1: Selection Sort – Tausch des nächst kleineren Elements mit der aktuellen Position

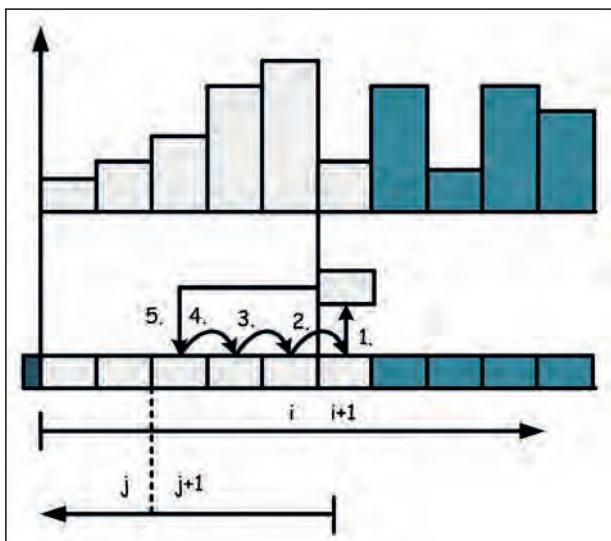


Abb. 2: Insertion Sort – jedes Element wird an der aktuell passenden Position eingesortiert

der benötigte Speicherplatzbedarf. Die Zahl der Iterationen gibt die Komplexität des Algorithmus an. In diesem Zusammenhang spricht man auch von den Kosten des Sortierens. Sortierverfahren können in interne und externe Verfahren unterteilt werden. Ist es möglich, die zu sortierenden Daten komplett im Hauptspeicher, d. h. innerhalb einer Datenstruktur, zum Beispiel einem Array, zu sortieren, so liegt ein internes Verfahren vor. Bei größeren Datenbeständen ist es nicht handhabbar, sämtliche Daten innerhalb des Arbeitsspeichers zu halten. Zum Einsatz kommen dann externe Speichermedien. Bei externen Algorithmen ist die Effizienz besonders wichtig, da die Lese- und Schreibzugriffe dann einen relativ hohen Zeitbedarf beanspruchen. Ein Sortieralgorithmus wird als stabil bezeichnet, wenn er die relative Reihenfolge von Elementen beibehält, die den gleichen Wert bezüglich des Sortierschlüssels aufweisen. Ein Beispiel: Es liegt eine Liste mit Namen und Alter der Personen vor, die alphabetisch sortiert ist. Diese soll nach dem Alter sortiert werden. Ein stabiler Algorithmus

gewährleistet, dass nach dem Sortiervorgang Personen mit dem gleichen Alter weiterhin alphabetisch sortiert sind. Ist ein Sortieralgorithmus zunächst nicht stabil, kann man diesen über Modifikationen i. d. R. als stabile Variante implementieren. Typische und bekannte Sortieralgorithmen sind:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Mergesort
- Quicksort

Diese Sortieralgorithmen sollte man kennen und ihre prinzipielle Vorgehensweise verstanden haben. Wir sehen uns diese in den kommenden Textabschnitten zunächst konzeptionell an und betrachten dann die Implementierung in Java. Um bei großen Datenmengen schneller zu werden, kann man nur auf eine parallele Vorgehensweise wechseln. Auch dies wird nachfolgend thematisiert.

Selection Sort

Es handelt sich hierbei um einen sehr einfachen und intuitiv arbeitenden Algorithmus. Dieser arbeitet nach dem folgenden Prinzip: Es wird das kleinste Element aus der Liste der zu sortierenden Elemente ausgewählt und mit dem ersten Element vertauscht (Swap-Operation). Danach sucht man das nächst kleinere Element und vertauscht es mit dem zweiten Element. Dieses Vorgehen wird fortgesetzt, bis alle Elemente der Liste in der richtigen (in diesem Fall aufsteigenden Reihenfolge) angeordnet sind. Das Prinzip sehen Sie in Abbildung 1, die eine beispielhafte Swap-Operation während der Laufzeit des Algorithmus darstellt.

Im Beispiel sind bereits die ersten Elemente sortiert. Es wird das nächst kleinere Element gesucht (*minKey*) und mit dem Element an der aktuellen Position (*i*) getauscht. Der Algorithmus stellt sich daher auch sehr einfach dar. Wir benötigen eine Hilfsvariable (Zwischenspeicher) und drei Zuweisungen. Einen Vorschlag zur Implementierung in Java finden Sie in Listing 3.

Listing 3: Selection Sort in Java

```
public static int[] SelectionSort(int[] items) {
    for (int i = 0; i < items.length - 1; i++) {
        for (int j = i + 1; j < items.length; j++) {
            if (items[i] > items[j]) {
                int temp = items[i];
                items[i] = items[j];
                items[j] = temp;
            }
        }
    }
    return items;
}
```

Der Algorithmus ist einfach zu implementieren. Die Ausführungszeit ist unabhängig vom Sortierzustand der Ausgangsdaten.

Insertion Sort

Dieser Algorithmus entspricht dem üblicherweise intuitiven Vorgehen beim Einsortieren von Gegenständen oder beim Sortieren von Spielkarten. Man geht dabei wie folgt vor: Das erste Element wird als bereits sortiert betrachtet. Dann nimmt man das nächste Element und sortiert dieses entweder davor oder danach ein, je nach Größe des Schlüsselwerts. Mit dem nächsten Element geht man genauso vor. Mit der Zeit wird der sortierte Bereich immer größer, d.h., dass man immer ein Element aus dem unsortierten Bereich entnimmt und dieses an der richtigen Position des sortierten Bereichs einfügt. Abbildung 2 zeigt das Prinzip des Algorithmus und Listing 4 den Quellcode in Java. Insert Sort eignet sich besonders gut für bestimmte Arten von nicht zufällig angeordneten Arrays, d. h., wenn bereits eine gewisse Ordnung der Elemente vorliegt. In diesem Fall arbeitet der Algorithmus schnell.

Bubble Sort

Die Idee beruht darauf, dass man solange systematisch benachbarte Elemente miteinander vergleicht und bei Bedarf austauscht, bis der Datenbestand sortiert ist. Soll eine Liste von Zahlen (*Keys*) in eine aufsteigende Reihenfolge gebracht werden, so durchläuft man das Feld von vorne nach hinten. Ist das betrachtete Element größer als sein rechter Nachbar, werden die Elemente vertauscht. Auf diese Weise wandern größere Elemente systematisch an das Ende der Liste. Man sagt umgangssprachlich auch, dass sie wie eine Luftblase nach oben aufsteigen (daher Bubble Sort). Im nächsten Schritt geht man die Liste wieder von vorne durch, die Zahl der unsortierten Elemente ist jedoch bereits um eines weniger geworden. Mit jedem Schritt wächst der sortierte Bereich (rechts) und der unsortierte Bereich (links) schrumpft. Abbildung 3 zeigt das Prinzip des Bubble-Sort-Algorithmus. Den Quellcode in Java finden Sie in Listing 5.

Listing 4: Insertion Sort in Java

```
public static int[] InsertionSort(int[] items) {
    int temp;
    for (int i = 1; i < items.length; i++) {
        temp = items[i];
        int j = i;
        while (j > 0 && items[j - 1] > temp) {
            items[j] = items[j - 1];
            j--;
        }
        items[j] = temp;
    }
    return items;
}
```

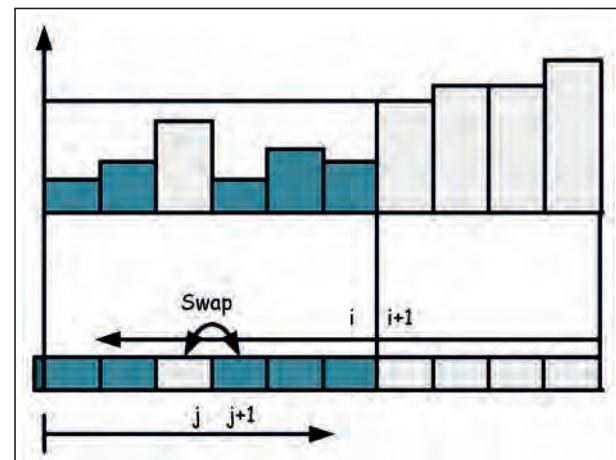


Abb. 3: Bubble Sort basiert auf einem systematischen Austausch der Elemente

Quick Sort und paralleler Quicksort

Quicksort ist ein Algorithmus, der nach dem Prinzip „Teile und herrsche“ (divide and conquer) arbeitet. Dazu wird die zu sortierende Liste in zwei Teillisten (linke und rechte Teilliste) getrennt. Als Nächstes wird ein sogenanntes Pivotelement aus der Liste ausgewählt. Alle Elemente, die kleiner als das Pivotelement sind, kommen in die linke Teilliste. Alle Elemente, die größer als das Pivotelement sind, kommen in die rechte Teilliste. Elemente, die gleich groß sind, können entweder der einen oder der anderen Liste zugeordnet werden. Nach der Aufteilung sind die Elemente der linken Liste kleiner oder gleich den Elementen der rechten Liste. Anschließend muss man also noch jede Teilliste in sich sortieren, um die Sortierung zu vollenden. Dazu wird der Quicksort-Algorithmus jeweils auf der linken und auf der rechten Teilliste ausgeführt, d. h. rekursiv angewendet. Dazu wird jede Teilliste wiederum in zwei Teillisten aufgeteilt, neue Pivotelemente bestimmt und jede Teilliste wiederum sortiert. Wenn eine Teilliste nur noch die Länge eins oder null hat, so ist sie bereits sortiert und es erfolgt der Abbruch der Rekursion. Das Prinzip des Algorithmus finden Sie in Abbildung 4 und den Quellcode für die Programmiersprache Java in Listing 6. Dazu sind einige wenige Anmerkungen angebracht. Der Algorith-

Listing 5: Bubble Sort in Java

```
public static int[] BubbleSort(int[] items) {
    int temp;
    for(int i=1; i<items.length; i++) {
        for(int j=0; j<items.length-i; j++) {
            if(items[j]>items[j+1]) {
                temp=items[j];
                items[j]=items[j+1];
                items[j+1]=temp;
            }
        }
    }
    return items;
}
```

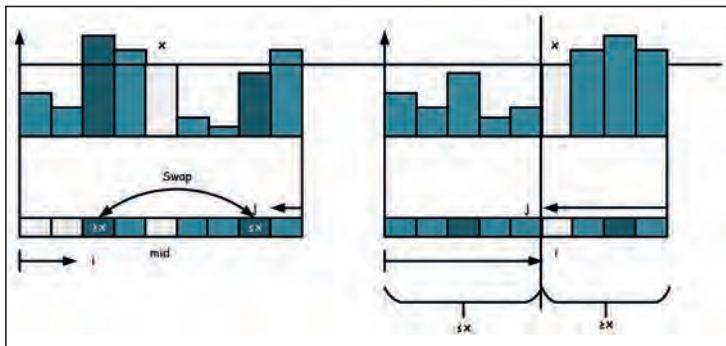


Abb. 4: Quick Sort basiert auf dem Prinzip von „Teile und herrsche“

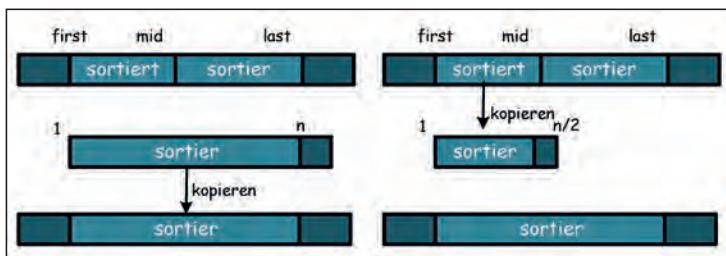


Abb. 5: Merge Sort basiert auf dem Mischen von zwei sortierten Teilstücken

mus arbeitet rekursiv, d. h., die Methode *QuickSort(...)* ruft sich selbst wieder auf. Die zweite (private) Methode ist für die Partitionierung der Liste in die beiden Teil-

Listing 6: Quick Sort in Java

```
public static void QuickSort(int[] items, int start, int end){

    int partition = partition(items, start, end);
    if(partition>start) {
        QuickSort(items, start, partition - 1);
    }
    if(partition+1<end) {
        QuickSort(items, partition + 1, end);
    }
}

private static int partition(int[] items, int start, int end){
    int pivot = items[end];

    for(int i=start; i<end; i++){
        if(items[i]<pivot){
            int temp= items[start];
            items[start]=items[i];
            items[i]=temp;
            start++;
        }
    }

    int temp = items[start];
    items[start] = pivot;
    items[end] = temp;
    return start;
}
```

listen anhand des Pivotelements zuständig. Beim ersten Aufruf wird der Algorithmus mittels *QuickSort(...)* gestartet und als Parameter werden die zu sortierende Liste und als Start und Endparameter die Werte 0 und die Anzahl der Elemente der Liste übergeben.

Bereits an dieser Stelle können wir erwähnen, dass der Sortieralgorithmus Quick Sort seinen Namen nicht umsonst trägt. In den meisten Fällen weist der Algorithmus ein besseres Laufzeitverhalten auf als die anderen Algorithmen. Aus der Konstruktion des Vorgehens, d. h. der Aufteilung der zu durchsuchenden Elemente auf einzelne Bereiche, lässt sich erahnen, dass sich der Quick-Sort-Algorithmus auch für eine parallele Verarbeitung eignet. In diesem Fall würde man die Aufgaben zur Sortierung der Teilmengen einzelnen Prozessen übertragen. Parallel Varianten des Quick-Sort-Algorithmus sind der Regular-Sampling- und der Hyper-Quick-Sort-Algorithmus. Eine Darstellung dieser parallel arbeitenden Algorithmen findet sich zum Beispiel in [1]. Möchte man diese Algorithmen für spezielle Sortieraufgaben implementieren, muss man sich mit der parallelen Programmierung (Threads) auseinandersetzen, da man den Suchraum in Teilaufgaben zerlegen und diesen den Prozessoren zuweisen muss. Für einfachere Anwendungen geht es aber mit Hilfe der Java-Klassenbibliothek auch schneller, worüber wir gleich ausführlicher schreiben.

Merge Sort

Merge Sort arbeitet ebenfalls nach dem Prinzip, die gesamte Liste der zu sortierenden Objekte aufzuteilen, d. h., dass auch dieser Algorithmus nach dem „Teile und herrsche“-Verfahren arbeitet: Die Gesamtmenge der Objekte wird in zwei Hälften geteilt. Beide Hälften werden nun unabhängig voneinander sortiert und dann wieder gemischt. Beim Mischen werden immer die vordersten Elemente der sortierten Hälften verglichen und das jeweils kleinere in die Gesamtmenge übernommen. Es ist zu beachten, dass der Schritt der Zerlegung rekursiv so lange fortgesetzt wird, bis nur noch ein Element der Menge übrig ist. Abbildung 5 zeigt das Prinzip am Beispiel und Listing 7 den zugehörigen Quellcode in Java. Die steuernde Methode lautet *MergeSort(...)*. Diese nimmt die zu sortierenden Elemente entgegen und ruft sich erneut rekursiv selbst auf. Die private Methode *merge(...)* vermischt beide sortierten Teillisten. Es gibt unterschiedliche Varianten und Implementierungen des Merge-Sort-Algorithmus. Insbesondere unterscheiden sie sich bezüglich Stabilität und Speicherplatzbedarf. Im durchschnittlichen Fall ist Merge Sort langsamer als Quick Sort.

Indirektes Sortieren

Die Anzahl der notwendigen Operationen bestimmt das Laufzeitverhalten des betreffenden Algorithmus. Dabei ist es entscheidend, wie schnell die Tausch- und Kopieroperationen der betreffenden Objekte ausgeführt werden. In unseren einfachen Beispielen haben wir lediglich Listen von Zahlen sortiert. In diesem Fall

Listing 7: Marge Sort in Java

```
public static int[] MergeSort(int[] items) {  
  
    if (items.length > 1) {  
        int mitte = (int) (items.length / 2);  
  
        int[] links = new int[mitte];  
        for (int i = 0; i <= mitte - 1; i++) {  
            links[i] = items[i];  
        }  
  
        int[] rechts = new int[items.length - mitte];  
        for (int i = mitte; i <= items.length - 1; i++) {  
            rechts[i - mitte] = items[i];  
        }  
  
        links = MergeSort(links);  
        rechts = MergeSort(rechts);  
  
        return merge(links, rechts);  
    } else {  
        return items;  
    }  
}  
  
private static int[] merge(int[] links, int[] rechts) {  
    int[] neuArray = new int[links.length + rechts.length];  
    int indexLinks = 0;  
    int indexRechts = 0;  
    int indexErgebnis = 0;  
  
    while (indexLinks < links.length && indexRechts < rechts.length) {  
        if (links[indexLinks] < rechts[indexRechts]) {  
            neuArray[indexErgebnis] = links[indexLinks];  
            indexLinks += 1;  
        } else {  
            neuArray[indexErgebnis] = rechts[indexRechts];  
            indexRechts += 1;  
        }  
        indexErgebnis += 1;  
    }  
  
    while (indexLinks < links.length) {  
        neuArray[indexErgebnis] = links[indexLinks];  
        indexLinks += 1;  
        indexErgebnis += 1;  
    }  
  
    while (indexRechts < rechts.length) {  
        neuArray[indexErgebnis] = rechts[indexRechts];  
        indexRechts += 1;  
        indexErgebnis += 1;  
    }  
  
    return neuArray;  
}
```

Anzeige

werden die Operationen ohne Verzögerungen ausgeführt. In der Praxis werden jedoch i. d. R. komplexere Datenobjekte anhand von Schlüsseln sortiert. Diese Objekte werden auch in Listen (Arrays) gespeichert. Das Sortieren durch Tausch- und Kopieroperationen führt dann zu einem größeren Zeitbedarf. Bei sehr großen zu sortierenden Mengen kann das relevant sein. Die Idee des indirekten Sortierens besteht nun darin, dass man nicht den Datenbestand selbst sortiert, sondern ein Feld von Zeigern, die auf den Datenbestand verweisen. Dazu ist ein Hilfsfeld notwendig, das auf den ursprünglichen Datenbestand, d. h. auf die einzelnen Objekte, zeigt. Das Hilfsfeld kann, da es nur Zeigerwerte beinhaltet, wiederum deutlich schneller sortiert werden. Voraussetzung für die Anwendung des indirekten Sortierens ist es, dass man die Adressen der ursprünglichen Objekte kennt.

Sortierverfahren vergleichen

Wir haben an dieser Stelle einige einfache (Selection Sort, Insert Sort) und komplexere Algorithmen (Quick Sort, Merge Sort) zum Sortieren von Datenbeständen vorgestellt. Der Vergleich dieser und anderer Algorithmen basiert auf den folgenden Parametern:

- *Komplexitätsangabe zum Laufzeitverhalten mit den Dimensionen:* bester Fall, durchschnittlicher Fall und schlechterer Fall.
- *Stabilität des Algorithmus:* Wird durch die Sortierung bei gleichem Schlüsselwert die Reihenfolge der Elemente beibehalten?
- *Speicherplatz:* Wird zusätzlicher Speicherplatz zum Sortieren benötigt?

Eine umfassende Darstellung nach diesen Kriterien findet man zum Beispiel in [2].

Sortieren in der Praxis

Um Datenmengen zu sortieren, müssen Sie nicht jedes Mal das Rad neu erfinden. Sortieren kann man eine Liste (Array) bekanntermaßen durch den Aufruf einer simplen Methode. In der Klasse *Arrays* aus dem Package *java.util*, finden Sie eine Vielzahl von Methoden zum Sortieren eines Arrays unterschiedlicher Datentypen, zum Beispiel *static void sort(double[] a)* ohne die Angabe eines Sortierbereiches oder *static void sort(double[] a, int fromIndex, int toIndex)* mit der entsprechenden Eingrenzung des zu sortierenden Bereiches für die übergebene Datenstruktur; in diesem Fall ein Array mit Daten des Typs *double*.

Auch für generische Datentypen stehen entsprechende Methoden bereit, zum Beispiel in der Form *static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)* unter Berücksichtigung eines *Comparators*.

Ein Blick in die Dokumentation zeigt, dass diese Sortieralgorithmen auf dem Dual-Pivot Quick Sort von Vladimir Yaroslavskiy, Jon Bentley und Joshua Bloch

beruhen und damit bereits eine optimierte Variante des oben beschriebenen Quick-Sort-Algorithmus darstellen [3]. Das wirkt sich i. d. R. positiv auf die Laufzeit aus.

Sehr interessant für das Sortieren großer Datenbestände ist das Vorhandensein paralleler Sortieralgorithmen in der genannten Klasse *Arrays*. Diese Methoden stehen in Analogie zu den o. g. nicht parallelen Varianten zur Verfügung, also zum Beispiel *static void parallelSort(double[] a)*.

Intern wird hier laut Dokumentation auf eine parallele Variante des Merge-Algorithmus zurückgegriffen. Über diese Methoden kann in der Praxis die parallele Verarbeitung, d. h. das Sortieren genutzt werden, ohne dass man sich mit der Thread-Programmierung herumschlagen muss.

Suchen

Kommen wir zum zweiten großen Thema: den Suchverfahren. Suchverfahren sind Algorithmen, die einen Suchraum nach einem bestimmten Muster oder nach dem Vorhandensein eines Objekts durchsuchen. Man unterscheidet einfache und heuristische Suchalgorithmen. Einfache Suchalgorithmen durchsuchen den Suchraum mittels eines intuitiven Vorgehens. Diese systematische Vorgehensweise führt zwar theoretisch immer zum Ziel, es kann jedoch bei einem sehr großen Suchraum zu lange dauern. Zu den einfachen Suchalgorithmen gehören die folgenden:

- lineare Suche (sequenzielle Suche)
- binäre Suche
- Suche in Bäumen
- Suche in Graphen

Gegenüber einfachen Suchverfahren nutzen heuristische Suchalgorithmen das Wissen über den Suchraum (beispielsweise die Datenverteilung), um die benötigte Suchzeit zu reduzieren. Diese heuristischen Suchverfahren versuchen, durch intelligentes Suchen den Prozess abzukürzen. Heuristiken werden auch als lokale Suchverfahren bezeichnet. Allerdings kann das Auffinden der optimalen Lösung nicht garantiert werden. Aufgrund der speziellen Anpassung der Verfahren an die Problemstellung werden sie als problemspezifische Heuristiken bezeichnet. Eine Einteilung in Konstruktionsverfahren und Verbesserungsverfahren ist üblich. Mit Hilfe von Konstruktionsverfahren wird zunächst eine Startlösung erzeugt, deren Güte meist noch weit vom Optimum entfernt ist. Danach wird oft versucht, diese Lösung schrittweise zu verbessern. Noch einen Schritt weiter geht der Ansatz von sogenannten Metaheuristiken. Dabei handelt es sich um allgemeine, d. h. problemunabhängige Verfahrensansätze. Im Gegensatz zu den lokalen Suchverfahren ist es deren Ziel, die Suche über den gesamten Lösungsraum zu steuern und damit die Beschränkungen der lokalen Suchverfahren aufzuheben. Wir kommen im dritten Teil der Serie auf dieses Thema zurück. Zunächst geht es um die Basics, d. h. um die einfachen Suchverfahren.

Anzeige

Lineare (sequenzielle) Suche

Die sequenzielle Suche beruht auf dem Ansatz, einen Datenbestand vollständig zu durchsuchen, bis das passende Element gefunden wird. Man prüft also jedes einzelne Element darauf, ob es dem Suchkriterium entspricht. Der Algorithmus hat demzufolge den folgenden Ablauf:

- Lege den Suchschlüssel fest.
- Vergleiche jedes Element mit dem Suchschlüssel.
- Führe folgende Prüfung durch: im Erfolgsfall: Liefere die Indexposition beim ersten Auffinden des Objekts und beende das Suchen. Kein Erfolg: Gehe zum nächsten Objekt und vergleiche es.
- Gib eine Meldung über das erfolglose Suchen zurück, wenn das Ende des Datenbestands erreicht wurde, ohne dass das Objekt gefunden wurde.

Der Aufwand für diesen Algorithmus ist linear, d. h. $O(n)$. Die Anzahl der nötigen Vergleichsoperationen hängt direkt von der Anzahl der Elemente (n) im Datenbestand ab. Im Durchschnitt sind dies $(n/2)$ Vergleiche.

Binäre Suche

Sind die zu durchsuchenden Elemente geordnet, dann kann man die binäre Suche anwenden. Dies ist im Vorfeld ggf. über einen Sortieralgorithmus sicherzustellen. Die binäre Suche erfolgt wiederum nach dem Prinzip „Teile und herrsche“. Dazu teilt man die zu durchsuchenden Daten in zwei Hälften und ermittelt dann, in welcher Hälfte das zu suchende Element vorkommt. Die andere Hälfte kann ignoriert werden. So wird bereits im ersten Suchschritt die Menge der zu durchsuchenden Daten um 50 Prozent reduziert. Bei der binären Suche ist der Zeitaufwand nur noch $O(n) = \log_2 n$, also erheblich geringer als bei der linearen Suche.

Suche in Bäumen

Bei binären Suchbäumen handelt es sich um eine spezielle Datenstruktur. Dabei sind die Elemente im linken Teilbaum stets kleiner als die Wurzel. Im Gegensatz dazu sind alle Elemente im rechten Unterbaum stets größer als die Wurzel. Jeder Knoten besitzt höchstens zwei Kinder. Ziel des Einsatzes eines binären Suchbaums ist es, die Suchvorgänge zu verkürzen. Besonders wichtig ist dies bei umfangreichen Datenbeständen. Das Durchlaufen eines Baumes wird auch als Traversieren bezeichnet. Folgende Durchlaufordnungen sind üblich:

- *Preorder-Reihenfolge*: Es wird zuerst die Wurzel des Baums besucht. Danach werden rekursiv der linke und dann der rechte Teilbaum durchlaufen.
- *Postorder-Reihenfolge*: Es werden zuerst die Kindknoten (rekursiv) durchlaufen, danach der Elternknoten.
- *Inorder-Reihenfolge*: Es wird erst der linke Teilbaum des Knotens in der Postorder-Reihenfolge durchsucht, dann der Elternknoten und danach der rechte Teilbaum des Knotens in der Postorder-Reihenfolge.

Für die Verwendung als binärer Suchbaum muss ein Merkmal in Form eines Schlüssels (Key) in die Daten eingefügt werden. Anhand dieses Schlüssels werden die Daten innerhalb des Baums abgelegt, d. h., dass die zugehörigen Datenspeicher den Knoten und nicht den Kanten oder den Blättern zugeordnet sind. Dabei sollte der Schlüssel selbst Bestandteil der Daten oder aus den Daten zu berechnen sein. Anhand des Schlüssels wird unterschieden, ob links oder rechts unterhalb des Knotens gesucht werden soll. Der Aufruf wird rekursiv fortgesetzt, bis das gewünschte Element gefunden wurde. Weist kein Schlüssel eines Knotens den gewünschten Wert auf, so wird die Suche erfolglos beendet. Oder anders ausgedrückt: Das gewünschte Element war nicht dabei.

Zusammenfassung und Ausblick

Sortieren und Suchen gehören zu den Basics der Algorithmen. Wir haben gesehen, dass auch neuere Ansätze wie parallele Sortierverfahren auf den Grundalgorithmen beruhen. In der kommenden Ausgabe werden wir uns erneut mit der Suche beschäftigen. Dort geht es aber nicht darum, ein bestimmtes Objekt in einer Liste von gleichartigen Objekten zu finden, sondern wir suchen mit Metaheuristiken nach einer möglichst guten Lösung. Das ist immer dann sinnvoll, wenn die Suche nach der besten Lösung viel zu lange dauern würde, weil der Suchraum (d. h. Möglichkeiten, Datenbestand) viel zu groß ist.



Dr. Veikko Krypczyk ist begeisterter Entwickler und Fachautor.



Elena Bochkor arbeitet am Entwurf und Design mobiler Anwendungen und Webseiten.

Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter <http://larinet.com>.

Links & Literatur

- [1] Mohamed, Magdi: „Parallel Quicksort in Hypercubes“, https://www.researchgate.net/publication/234794781_Parallel_Quicksort_in_Hypercubes
- [2] <https://de.wikipedia.org/wiki/Sortierverfahren>
- [3] <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

Anzeige

Code in Machine-Learning-Modellen: Komplexität vermeiden

Kampf den schlechten Gewohnheiten

Es ist bekannt, dass der Code für Machine Learning (ML) schnell unordentlich [1] werden kann. Mit welchen Techniken lassen sich schlechte Programmiergewohnheiten vermeiden, die den Code unnötig kompliziert machen? Und welche Methoden erleichtern den Umgang mit der Komplexität?

von David Tan

Code, mit dem ML-Modelle trainiert werden sollen, wird normalerweise in Jupyter Notebooks geschrieben. Er steckt zum einen voller nebensächlicher Elemente (z.B. Druckanweisungen, gut lesbare DataFrames und Datenvisualisierungen), zum anderen voller Glue-Code ohne Abstraktionen, Modularisierung und automatisierte Tests. In einem echten Projekt führt das zu einem Chaos, das niemand mehr überblicken kann. Schlechte Programmiergewohnheiten resultieren in schwer verständlichem Code, dessen Änderung mühsam und fehleranfällig ist. Für Data Scientists und Entwickler*innen wird es dadurch immer schwieriger, ihre ML-Lösungen weiterzuentwickeln.

Wodurch wird Code kompliziert?

Komplex ist etwas, das aus mehreren miteinander verbundenen Teilen besteht. Mit jedem beweglichen Teil, das wir ergänzen, machen wir den Code komplexer und müssen wir ein weiteres Element im Auge behalten. Zwar können wir der grundlegenden Komplexität eines Problems nicht entgehen, doch machen wir die Dinge oft unnötig kompliziert und erhöhen die kognitive Belastung, zum Beispiel mit den folgenden schlechten An gewohnheiten:

Listing 1

```
# bad example
df = get_data()
print(df)
# do_other_stuff()
# do_some_more_stuff()

df.head()
print(df.columns)
# do_so_much_stuff()
```

Listing 2

```
# good example
df = get_data()
model = train_model(df)

# do_so_much_stuff()
model = train_model(df)
```

- *Fehlende Abstraktionen:* Schreiben wir den gesamten Code in ein einziges Python-Notebook oder -Skript, ohne ihn in Funktionen oder Klassen zu abstrahieren, zwingen wir die Leser*innen, viele Zeilen Code zu lesen, um das „Wie“ zu verstehen und herauszufinden, wozu der Code dient.

- *Lange Funktionen, die mehrere Dinge erledigen:* Wir müssen Zwischenzustände von Datentransformationen im Kopf behalten, während wir an einem anderen Teil der Funktion arbeiten.

- *Verzicht auf Unit-Tests:* Beim Refactoring können wir nur durch den Neustart des Kernels und die Ausführung des/der gesamten Notebooks überprüfen, ob alles noch funktioniert. Wir müssen uns mit der Komplexität der Codebasis auseinandersetzen, selbst wenn wir nur an einem kleinen Teil davon arbeiten möchten.

Komplexität ist unvermeidlich, aber sie lässt sich aufgliedern. Für Datenbereinigung, Feature Engineering, Fehlerkorrekturen, Verarbeitung neuer Daten usw. wird laufend neuer Code hinzugefügt. Wenn wir unsere Codebasis nicht sorgsam pflegen und regelmäßig Refactorings vornehmen (wofür wir allerdings Unit-Tests brauchen), sind Unordnung und Komplexität garantiert. In den folgenden Abschnitten stellen wir nicht nur häufige schlechte Gewohnheiten vor, die die Komplexität erhöhen, sondern auch gute Gewohnheiten, die den Umgang mit Komplexität erleichtern.

Ordnung im Code halten

Unordentlicher Code ist kompliziert, weil er sich nur schwer verstehen und ändern lässt. So wird das Ändern von Code als Reaktion auf Geschäftsanforderungen immer schwieriger und manchmal sogar unmöglich. Eine dieser schlechten Programmiergewohnheiten (auch Code Smell genannt) ist toter Code. Dabei handelt es sich um Code, der ausgeführt wird, dessen Ergebnis aber in keiner weiteren Berechnung verwendet

wird. Eine überflüssige Sache, die Entwickler*innen beim Programmieren im Kopf behalten müssen. Zur Veranschaulichung zwei Codebeispiele in Listing 1 und Listing 2.

Vorgehensweisen für sauberen Code wurden bereits ausführlich für mehrere Sprachen [2] beschrieben, einschließlich Python [3]. Wir haben diese Grundsätze für „Clean Code“ angepasst. Sie finden sich auch unter [4]:

- Design (Codebeispiele unter [5])
- Interna nicht offenlegen (Details zur Implementierung verborgen)
- Überflüssiges (Codebeispiele unter [6])
- Toten Code entfernen
- Printanweisungen vermeiden – auch geschönte Printanweisungen, wie `head()`, `df.describe()` und `df.plot()`
- Variablen (Codebeispiele unter [7])
- Namen der Variablen sollten Zweck anzeigen
- Funktionen (Codebeispiele unter [8])
- DRY-Prinzip („Don’t repeat yourself“) mit Hilfe von Funktionen anwenden
- Funktionen sollten eine einzige Aufgabe erledigen

Funktionen nutzen, um Komplexität durch Abstraktion zu reduzieren

Funktionen vereinfachen unseren Code, indem sie komplizierte Umsetzungsdetails in einer Abstraktion zusammenfassen und durch eine einfachere Darstellung ersetzen – ihren Namen. Hierzu ein Beispiel: Sie sind in einem Restaurant und bekommen eine Speisekarte. Anstatt der Namen der Gerichte finden Sie in der Speisekarte das Rezept für jedes Gericht. Es ist einfacher, wenn in der Speisekarte diese ganzen Schritte (also die Umsetzungsdetails) verborgen bleiben und wir stattdessen den Namen des Gerichts erfahren. Zur Veranschaulichung ein Codebeispiel aus einem Notebook im Titanic-Wettbewerb von Kaggle (Listing 3), das nach dem Refactoring folgendermaßen aussieht:

```
df['FareBand'] = categorize_column(df['Fare'], num_bins=4)
```

Was haben wir durch die Abstraktion in Funktionen und die damit geringere Komplexität erreicht?

- Lesbarkeit. Anhand der Schnittstelle (also `categorize_column()`) lässt sich erkennen, welche Aufgabe die Funktion hat. Es ist nicht nötig, jede Zeile zu lesen oder im Internet Dinge zu recherchieren (z. B. `qcut`). Wer anhand des Namens und der Verwendung noch nicht verstanden hat, wozu die Funktion dient, kann sich die Unit-Tests oder die Definition der Funktion ansehen.
- Da es sich nun um eine Funktion handelt, können wir leicht einen Unit-Test dafür schreiben. Ändern wir versehentlich das Verhalten der Funktion, scheitern die Unit-Tests, und wir erhalten innerhalb von Millisekunden eine Rückmeldung.

- Um an einer beliebigen Spalte dieselbe Transformation vorzunehmen (z. B. Alter oder Einkommen), brauchen wir nur jeweils eine Zeile Code statt mehrerer.

Durch das Refactoring in Funktionen kann das gesamte Notebook einfacher und eleganter gestaltet werden (Listings 4 und 5).

Die mentale Last ist nun deutlich geringer. Wir müssen nicht mehr zeilenweise Umsetzungsdetails lesen, um den gesamten Fluss zu verstehen. Stattdessen wird die Komplexität durch Abstraktionen (d. h. Funktionen) verringert: Wir erfahren, was die Funktionen tun, müssen uns aber nicht die Mühe machen, herauszufinden, wie sie es tun.

Code möglichst schnell aus Jupyter Notebooks verschieben

Im Bereich Innenausstattung gibt es ein Konzept – das Gesetz der ebenen Flächen (Law of Flat Surfaces) –, nach

Listing 3

```
pd.qcut(df['Fare'], q=4, retbins=True)[1] # returns array([0., 7.8958,
14.4542, 31.275, 512.3292])

df.loc[ df['Fare'] <= 7.90, 'Fare'] = 0 df.loc[(df['Fare'] > 7.90) & (df['Fare']
<= 14.454), 'Fare'] = 1 df.loc[(df['Fare'] > 14.454) & (df['Fare'] <= 31),
'Fare'] = 2 df.loc[ df['Fare'] > 31, 'Fare'] = 3
df['Fare'] = df['Fare'].astype(int)
df['FareBand'] = df['Fare']
```

Listing 4

```
# bad example
See notebook

# good example
df = impute_nans(df, categorical_columns=['Embarked'],
Continuous_columns=['Fare', 'Age'])

df = add_derived_title(df)
df = encode_title(df)
df = add_is_alone_column(df)
df = add_categorical_columns(df)
X, y = split_features_and_labels(df)
```

Listing 5

```
# an even better example. Notice how this reads like a story
prepare_data = compose(impute_nans,
add_derived_title,
encode_title,
add_is_alone_column,
add_categorical_columns,
split_features_and_labels)

X, y = prepare_data(df)
```

dem jede ebene Fläche in einem Zuhause Unordnung anzieht. Jupyter Notebooks sind die ebenen Flächen der ML-Welt. Natürlich eignen sich Jupyter Notebooks



Abb. 1: Große Mengen Code verzögern das Feedback

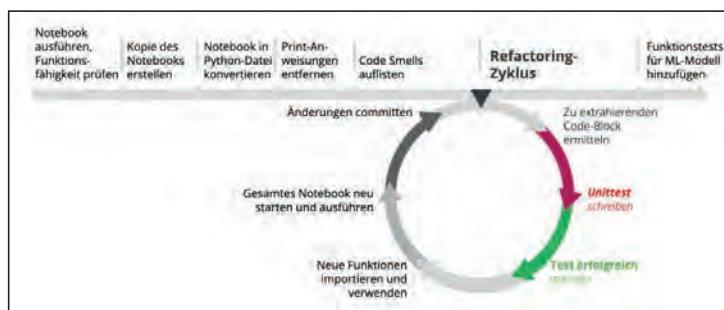


Abb. 2: Refactoring eines Jupyter Notebooks [9]

```

CLEAN-CODE-DATA-SCIENCE
├── getting-started-with-titanic-dataset-refactor... A
├── getting-started-with-titanic-dataset-refactor... A
├── getting-started-with-titanic-dataset-refactor... A
└── getting-started-with-titanic-dataset.ipynb
    └── submission_titanic.csv
    └── _pycache_
        └── test_preprocessing.py
        └── model_training.py
        └── preprocessing.py
        └── train.py
        └── type_hints_example.py
    └── _README.md
    └── .gitignore
    └── Dockerfile
    └── README.md
    └── refactoring_notebooks.md
    └── requirements.txt
    └── OUTLINE

src
└── preprocessing.py
    25 def categorize_column(series, num_bins):
    26     bins = pd.cut(series, num_bins, retbins=True)[1]
    27     return pd.Series(np.digitize(series, bins, right=True))
    28
    29
    30
    31 def add_is_alone_column(df):
    32     df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
    33
    34     df['IsAlone'] = 0
    35     df.loc[df['FamilySize'] == 1, 'IsAlone'] = 1
    36
    37     df = df.drop(['FamilySize'], axis = 1)
    38
    39     return df
    40
    41 def do_something():
    42     # this is what we're really working on
    43     pass
    44
    45 def impute_nans(df, categorical_columns=[], continuous_columns=[]):
    46     for column in categorical_columns + continuous_columns:
    47         if column in categorical_columns:
    48             replacement = df[column].dropna().mode()[0]
    49         if column in continuous_columns:
    50             replacement = df[column].dropna().median()
    51
    52
    53
    54
    55
    56
    57
    58
    59
    60
    61
    62
    63
    64
    65
    66
    67
    68
    69
    70
    71
    72
    73
    74
    75
    76
    77
    78
    79
    80
    81
    82
    83
    84
    85
    86
    87
    88
    89
    90
    91
    92
    93
    94
    95
    96
    97
    98
    99
    100
    101
    102
    103
    104
    105
    106
    107
    108
    109
    110
    111
    112
    113
    114
    115
    116
    117
    118
    119
    120
    121
    122
    123
    124
    125
    126
    127
    128
    129
    130
    131
    132
    133
    134
    135
    136
    137
    138
    139
    140
    141
    142
    143
    144
    145
    146
    147
    148
    149
    150
    151
    152
    153
    154
    155
    156
    157
    158
    159
    160
    161
    162
    163
    164
    165
    166
    167
    168
    169
    170
    171
    172
    173
    174
    175
    176
    177
    178
    179
    180
    181
    182
    183
    184
    185
    186
    187
    188
    189
    190
    191
    192
    193
    194
    195
    196
    197
    198
    199
    200
    201
    202
    203
    204
    205
    206
    207
    208
    209
    210
    211
    212
    213
    214
    215
    216
    217
    218
    219
    220
    221
    222
    223
    224
    225
    226
    227
    228
    229
    230
    231
    232
    233
    234
    235
    236
    237
    238
    239
    240
    241
    242
    243
    244
    245
    246
    247
    248
    249
    250
    251
    252
    253
    254
    255
    256
    257
    258
    259
    260
    261
    262
    263
    264
    265
    266
    267
    268
    269
    270
    271
    272
    273
    274
    275
    276
    277
    278
    279
    280
    281
    282
    283
    284
    285
    286
    287
    288
    289
    290
    291
    292
    293
    294
    295
    296
    297
    298
    299
    300
    301
    302
    303
    304
    305
    306
    307
    308
    309
    310
    311
    312
    313
    314
    315
    316
    317
    318
    319
    320
    321
    322
    323
    324
    325
    326
    327
    328
    329
    330
    331
    332
    333
    334
    335
    336
    337
    338
    339
    340
    341
    342
    343
    344
    345
    346
    347
    348
    349
    350
    351
    352
    353
    354
    355
    356
    357
    358
    359
    360
    361
    362
    363
    364
    365
    366
    367
    368
    369
    370
    371
    372
    373
    374
    375
    376
    377
    378
    379
    380
    381
    382
    383
    384
    385
    386
    387
    388
    389
    390
    391
    392
    393
    394
    395
    396
    397
    398
    399
    400
    401
    402
    403
    404
    405
    406
    407
    408
    409
    410
    411
    412
    413
    414
    415
    416
    417
    418
    419
    420
    421
    422
    423
    424
    425
    426
    427
    428
    429
    430
    431
    432
    433
    434
    435
    436
    437
    438
    439
    440
    441
    442
    443
    444
    445
    446
    447
    448
    449
    450
    451
    452
    453
    454
    455
    456
    457
    458
    459
    460
    461
    462
    463
    464
    465
    466
    467
    468
    469
    470
    471
    472
    473
    474
    475
    476
    477
    478
    479
    480
    481
    482
    483
    484
    485
    486
    487
    488
    489
    490
    491
    492
    493
    494
    495
    496
    497
    498
    499
    500
    501
    502
    503
    504
    505
    506
    507
    508
    509
    510
    511
    512
    513
    514
    515
    516
    517
    518
    519
    520
    521
    522
    523
    524
    525
    526
    527
    528
    529
    530
    531
    532
    533
    534
    535
    536
    537
    538
    539
    540
    541
    542
    543
    544
    545
    546
    547
    548
    549
    550
    551
    552
    553
    554
    555
    556
    557
    558
    559
    560
    561
    562
    563
    564
    565
    566
    567
    568
    569
    570
    571
    572
    573
    574
    575
    576
    577
    578
    579
    580
    581
    582
    583
    584
    585
    586
    587
    588
    589
    590
    591
    592
    593
    594
    595
    596
    597
    598
    599
    600
    601
    602
    603
    604
    605
    606
    607
    608
    609
    610
    611
    612
    613
    614
    615
    616
    617
    618
    619
    620
    621
    622
    623
    624
    625
    626
    627
    628
    629
    630
    631
    632
    633
    634
    635
    636
    637
    638
    639
    640
    641
    642
    643
    644
    645
    646
    647
    648
    649
    650
    651
    652
    653
    654
    655
    656
    657
    658
    659
    660
    661
    662
    663
    664
    665
    666
    667
    668
    669
    670
    671
    672
    673
    674
    675
    676
    677
    678
    679
    680
    681
    682
    683
    684
    685
    686
    687
    688
    689
    690
    691
    692
    693
    694
    695
    696
    697
    698
    699
    700
    701
    702
    703
    704
    705
    706
    707
    708
    709
    710
    711
    712
    713
    714
    715
    716
    717
    718
    719
    720
    721
    722
    723
    724
    725
    726
    727
    728
    729
    730
    731
    732
    733
    734
    735
    736
    737
    738
    739
    740
    741
    742
    743
    744
    745
    746
    747
    748
    749
    750
    751
    752
    753
    754
    755
    756
    757
    758
    759
    760
    761
    762
    763
    764
    765
    766
    767
    768
    769
    770
    771
    772
    773
    774
    775
    776
    777
    778
    779
    780
    781
    782
    783
    784
    785
    786
    787
    788
    789
    790
    791
    792
    793
    794
    795
    796
    797
    798
    799
    800
    801
    802
    803
    804
    805
    806
    807
    808
    809
    810
    811
    812
    813
    814
    815
    816
    817
    818
    819
    820
    821
    822
    823
    824
    825
    826
    827
    828
    829
    830
    831
    832
    833
    834
    835
    836
    837
    838
    839
    840
    841
    842
    843
    844
    845
    846
    847
    848
    849
    850
    851
    852
    853
    854
    855
    856
    857
    858
    859
    860
    861
    862
    863
    864
    865
    866
    867
    868
    869
    870
    871
    872
    873
    874
    875
    876
    877
    878
    879
    880
    881
    882
    883
    884
    885
    886
    887
    888
    889
    890
    891
    892
    893
    894
    895
    896
    897
    898
    899
    900
    901
    902
    903
    904
    905
    906
    907
    908
    909
    910
    911
    912
    913
    914
    915
    916
    917
    918
    919
    920
    921
    922
    923
    924
    925
    926
    927
    928
    929
    930
    931
    932
    933
    934
    935
    936
    937
    938
    939
    940
    941
    942
    943
    944
    945
    946
    947
    948
    949
    950
    951
    952
    953
    954
    955
    956
    957
    958
    959
    960
    961
    962
    963
    964
    965
    966
    967
    968
    969
    970
    971
    972
    973
    974
    975
    976
    977
    978
    979
    980
    981
    982
    983
    984
    985
    986
    987
    988
    989
    990
    991
    992
    993
    994
    995
    996
    997
    998
    999
    1000
    1001
    1002
    1003
    1004
    1005
    1006
    1007
    1008
    1009
    1010
    1011
    1012
    1013
    1014
    1015
    1016
    1017
    1018
    1019
    1020
    1021
    1022
    1023
    1024
    1025
    1026
    1027
    1028
    1029
    1030
    1031
    1032
    1033
    1034
    1035
    1036
    1037
    1038
    1039
    1040
    1041
    1042
    1043
    1044
    1045
    1046
    1047
    1048
    1049
    1050
    1051
    1052
    1053
    1054
    1055
    1056
    1057
    1058
    1059
    1060
    1061
    1062
    1063
    1064
    1065
    1066
    1067
    1068
    1069
    1070
    1071
    1072
    1073
    1074
    1075
    1076
    1077
    1078
    1079
    1080
    1081
    1082
    1083
    1084
    1085
    1086
    1087
    1088
    1089
    1090
    1091
    1092
    1093
    1094
    1095
    1096
    1097
    1098
    1099
    1100
    1101
    1102
    1103
    1104
    1105
    1106
    1107
    1108
    1109
    1110
    1111
    1112
    1113
    1114
    1115
    1116
    1117
    1118
    1119
    1120
    1121
    1122
    1123
    1124
    1125
    1126
    1127
    1128
    1129
    1130
    1131
    1132
    1133
    1134
    1135
    1136
    1137
    1138
    1139
    1140
    1141
    1142
    1143
    1144
    1145
    1146
    1147
    1148
    1149
    1150
    1151
    1152
    1153
    1154
    1155
    1156
    1157
    1158
    1159
    1160
    1161
    1162
    1163
    1164
    1165
    1166
    1167
    1168
    1169
    1170
    1171
    1172
    1173
    1174
    1175
    1176
    1177
    1178
    1179
    1180
    1181
    1182
    1183
    1184
    1185
    1186
    1187
    1188
    1189
    1190
    1191
    1192
    1193
    1194
    1195
    1196
    1197
    1198
    1199
    1200
    1201
    1202
    1203
    1204
    1205
    1206
    1207
    1208
    1209
    1210
    1211
    1212
    1213
    1214
    1215
    1216
    1217
    1218
    1219
    1220
    1221
    1222
    1223
    1224
    1225
    1226
    1227
    1228
    1229
    1230
    1231
    1232
    1233
    1234
    1235
    1236
    1237
    1238
    1239
    1240
    1241
    1242
    1243
    1244
    1245
    1246
    1247
    1248
    1249
    1250
    1251
    1252
    1253
    1254
    1255
    1256
    1257
    1258
    1259
    1260
    1261
    1262
    1263
    1264
    1265
    1266
    1267
    1268
    1269
    1270
    1271
    1272
    1273
    1274
    1275
    1276
    1277
    1278
    1279
    1280
    1281
    1282
    1283
    1284
    1285
    1286
    1287
    1288
    1289
    1290
    1291
    1292
    1293
    1294
    1295
    1296
    1297
    1298
    1299
    1300
    1301
    1302
    1303
    1304
    1305
    1306
    1307
    1308
    1309
    1310
    1311
    1312
    1313
    1314
    1315
    1316
    1317
    1318
    1319
    1320
    1321
    1322
    1323
    1324
    1325
    1326
    1327
    1328
    1329
    1330
    1331
    1332
    1333
    1334
    1335
    1336
    1337
    1338
    1339
    1340
    1341
    1342
    1343
    1344
    1345
    1346
    1347
    1348
    1349
    1350
    1351
    1352
    1353
    1354
    1355
    1356
    1357
    1358
    1359
    1360
    1361
    1362
    1363
    1364
    1365
    1366
    1367
    1368
    1369
    1370
    1371
    1372
    1373
    1374
    1375
    1376
    1377
    1378
    1379
    1380
    1381
    1382
    1383
    1384
    1385
    1386
    1387
    1388
    1389
    1390
    1391
    1392
    1393
    1394
    1395
    1396
    1397
    1398
    1399
    1400
    1401
    1402
    1403
    1404
    1405
    1406
    1407
    1408
    1409
    1410
    1411
    1412
    1413
    1414
    1415
    1416
    1417
    1418
    1419
    1420
    1421
    1422
    1423
    1424
    1425
    1426
    1427
    1428
    1429
    1430
    1431
    1432
    1433
    1434
    1435
    1436
    1437
    1438
    1439
    1440
    1441
    1442
    1443
    1444
    1445
    1446
    1447
    1448
    1449
    1450
    1451
    1452
    1453
    1454
    1455
    1456
    1457
    1458
    1459
    1460
    1461
    1462
    1463
    1464
    1465
    1466
    1467
    1468
    1469
    1470
    1471
    1472
    1473
    1474
    1475
    1476
    1477
    1478
    1479
    1480
    1481
    1482
    1483
    1484
    1485
    1486
    1487
    1488
    1489
    1490
    1491
    1492
    1493
    1494
    1495
    1496
    1497
    1498
    1499
    1500
    1501
    1502
    1503
    1504
    1505
    1506
    1507
    1508
    1509
    1510
    1511
    1512
    1513
    1514
    1515
    1516
    1517
    1518
    1519
    1520
    1521
    1522
    1523
    1524
    1525
    1526
    1527
    1528
    1529
    1530
    1531
    1532
    1533
    1534
    1535
    1536
    1537
    1538
    1539
    1540
    1541
    1542
    1543
    1544
    1545
    1546
    1547
    1548
    1549
    1550
    1551
    1552
    1553
    1554
    1555
    1556
    1557
    1558
    1559
    1560
    1561
    1562
    1563
    1564
    1565
    1566
    1567
    1568
    1569
    1570
    1571
    1572
    1573
    1574
    1575
    1576
    1577
    1578
    1579
    1580
    1581
    1582
    1583
    1584
    1585
    1586
    1587
    1588
    1589
    1590
    1591
    1592
    1593
    1594
    1595
    1596
    1597
    1598
    1599
    1600
    1601
    1602
    1603
    1604
    1605
    1606
    1607
    1608
    1609
    1610
    1611
    1612
    1613
    1614
    1615
    1616
    1617
    1618
    1619
    1620
    1621
    1622
    1623
    1624
    1625
    1626
    1627
    1628
    1629
    1630
    1631
    1632
    1633
    1634
    1635
    1636
    1637
    1638

```

Kleine und häufige Commits vornehmen

Ohne kleine und häufige Commits erhöhen wir die mentale Belastung. Während wir an einem bestimmten Problem arbeiten, wird bei Änderungen für frühere Probleme immer noch angezeigt, dass kein Commit stattgefunden hat. Das lenkt unseren Blick ab und erschwert die Konzentration auf das aktuelle Problem. Betrachten Sie die Abbildungen 3 und 4. Können Sie feststellen, an welcher Funktion wir gerade arbeiten? Bei welchem Bild fällt es Ihnen leichter? Kleine und häufige Commits bringen mehrere Vorteile:

- Weniger visuelle Ablenkungen und kognitive Last.
- Hat ein Commit stattgefunden, müssen wir nicht mehr befürchten, funktionierenden Code versehentlich zu zerstören.
- Neben Rot-grün-Refactoring [13] haben wir die Möglichkeit, Rot-rot-rot rückgängig zu machen [14]. Machen wir versehentlich etwas kaputt, greifen wir auf den letzten Commit zurück und nehmen einen neuen Anlauf. So müssen wir keine Probleme rückgängig machen, die bei dem Versuch entstanden sind, das eigentliche Problem zu beheben.

Wie klein sollte so ein Commit sein? Einen Commit sollte man immer dann vornehmen, wenn eine einzelne Gruppe logisch verknüpfter Änderungen und Testdurchläufe vorliegt. Dabei sollte man nach dem Wörtchen „und“ in der Commit-Nachricht Ausschau halten, z. B. „Explorative Datenanalyse hinzufügen und Sätze in Tokens aufsplitten und Mustertrainingscode restrukturieren“. Diese drei Änderungen sollten in drei logische Commits aufgeteilt werden. In dieser Situation können Sie `git add -patch` [15] nutzen, um Code für den Commit in kleinere Portionen aufzusplitten.

Fazit

Diese guten Gewohnheiten helfen uns dabei, die Komplexität in Machine-Learning- und Data-Science-Projekten unter Kontrolle zu halten. Das wiederum führt

Listing 6

```
import unittest
from sklearn.metrics import precision_score, recall_score

from src.train import prepare_data_and_train_model

class TestModelMetrics(unittest.TestCase):
    def test_model_precision_score_should_be_above_threshold(self):
        model, X_test, Y_test = prepare_data_and_train_model()
        Y_pred = model.predict(X_test)

        precision = precision_score(Y_test, Y_pred)

        self.assertGreaterEqual(precision, 0.7)
```

zu agileren und produktiveren Datenprojekten. Weitere Informationen darüber, wie Unternehmen Machine Learning dank Continuous-Delivery-Methoden agiler machen, finden sich unter [16] und [17].



David Tan ist seit zwei Jahren bei ThoughtWorks und arbeitete im Regierungssektor in einer nichttechnischen Rolle, bevor er sich für eine Karriere im Software Engineering entschied. In den letzten zwei Jahren hat er an mehreren Machine-Learning-Projekten gearbeitet, bei denen es um Aufgaben wie Börsenkursprognosen, Betrugsschutz und die Erkennung von Beer Quantity Image Recognition ging. Er ist auch Ausbilder für das ThoughtWorks-JumpStart!-Programm. David ist leidenschaftlich an der agilen Softwareentwicklung und an Wissensaustausch interessiert. In seiner Freizeit verbringt er als frischgebackener Vater gerne Zeit mit seiner Familie.

Links & Literatur

- [1] <https://github.com/davified/clean-code-ml/blob/master/notebooks/titanic-original.ipynb>
- [2] <https://github.com/abiodunjames/Awesome-Clean-Code-Resources>
- [3] <https://github.com/zedr/clean-code-python>
- [4] <https://github.com/davified/clean-code-ml>
- [5] <https://github.com/davified/clean-code-ml/blob/master/docs/design.md>
- [6] <https://github.com/davified/clean-code-ml/blob/master/docs/disposables.md>
- [7] <https://github.com/davified/clean-code-ml/blob/master/docs/variables.md>
- [8] <https://github.com/davified/clean-code-ml/blob/master/docs/functions.md>
- [9] <https://github.com/davified/clean-code-ml/blob/master/docs/refactoring-process.md>
- [10] <https://github.com/davified/clean-code-ml/blob/master/docs/refactoring-exercise.md>
- [11] <https://www.thoughtworks.com/insights/blog/test-driven-development-best-thing-has-happened-software-design>
- [12] https://github.com/davified/clean-code-ml/blob/master/src/tests/test_model_metrics.py
- [13] <https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html>
- [14] <https://www.facebook.com/notes/kent-beck/one-bite-at-a-time-partitioning-complexity/1716882961677894/>
- [15] <https://nuclearsquid.com/writings/git-add/>
- [16] <https://www.thoughtworks.com/insights/articles/intelligent-enterprise-series-cd4ml>
- [17] <https://www.thoughtworks.com/insights/blog/getting-smart-applying-continuous-delivery-data-science-drive-car-sales>



© akud/Shutterstock.com

Natives Elasticsearch auf Kubernetes

ECK macht es einfach

Vor nicht langer Zeit war es undenkbar, persistente Daten auf Kubernetes zu stellen. Kluge Ops-Teams hätten eher auf einen vergleichbaren Cloud-Service gesetzt und die Funktionalität für Simplizität und ein ruhiges Gewissen geopfert. In den letzten Jahren ist Kubernetes aber gereift und in Bereiche vorgestoßen, die zuvor nicht annähernd als geeignet für die Containerorchestrierung angesehen wurden.

von Dimitri Marx

Kubernetes hat an Stabilität und Funktionsumfang zugenommen und sich von einem Tool zu einer richtigen Plattform entwickelt. Eines der mächtigsten Features der letzten Releases ist die Kombination von Custom Resource Definitions (CRDs) und dem Operator Pattern.

Das Operator Pattern (oder einfach Operators) stellt ein sehr mächtiges Konzept dar, mit dem Softwareanbieter Betriebsmuster in Kubernetes abstrahieren können. Der Einsatz des Operator Patterns kann die notwendige Menge an Tools, Skripten und manueller Arbeit zum Betreiben komplexer Software massiv reduzieren. Operators erlauben es, innerhalb eines Dokuments – in einem sogenannten Kubernetes-Manifest – das gesamte Deployment zu beschreiben, nicht nur die Artefakte, die deployt werden. Auch wenn man eine Datenbank sehr

schnell auf Kubernetes deployen kann, wenn man Helm oder ein Kubernetes-Manifest verwendet, ist es doch meist Aufgabe des Endnutzers, für Sicherheit, Back-ups und Nutzerverwaltung zu sorgen.

Im Gegensatz dazu können viele dieser Aufgaben in einem System, das via CRDs und Operators deployt wird, automatisiert und mit der Kubernetes Configuration Language definiert werden. Diese Herangehensweise macht es leichter, bestehende Cluster und deren Zustand bzw. Verhalten zu durchblicken, die Konfiguration zu zentralisieren und Best Practices bestimmter Anbieter anzuwenden. Obwohl das Operator Pattern noch relativ neu ist, wird es bereits bei vielen Anbietern (inkl. Elastic) eingesetzt, um einen Betrieb der eigenen Programme auf Kubernetes zu ermöglichen.

Für Fans von Elasticsearch und dem wachsenden Elastic Stack bietet sich so die Möglichkeit, die Vorteile

der Standarddistribution nativ auf Kubernetes zu nutzen. Elastic hat hierfür kürzlich Elastic Cloud on Kubernetes (ECK) veröffentlicht, es soll zum offiziellen Weg werden, Elastic Cloud auf Kubernetes auszuführen. Ein recht großer Funktionsumfang ist im Operator bereits enthalten (Kasten: „Fünf Vorteile des nativen Betreibens von Elasticsearch auf Kubernetes via ECK“).

Bis vor Kurzem lief die Installation eines produktionsfertigen Elastic Stack ziemlich standardmäßig ab. Man erstellt ein Set von Masters, um den Cluster zu managen, dann werden Nodes hinzugefügt und konfiguriert, sie können verschiedene Rollen einnehmen. Je nach Automatisierung war das Management des Elastic-Clusters sehr umfangreich, und eine gewisse Orchestrierung wurde nötig, um sicherzugehen, dass der Cluster aktiv und bereit für Traffic ist.

Das Erstellen einer produktionsfertigen Elastic-Umgebung ist nicht annähernd so kompliziert wie das anderer Enterprise-Software. Doch es ist komplex genug, dass ein gewisses Maß an Planung, Wissen, Automatisierung und Ressourcen unabdinglich ist, um dessen Bereitschaft zur Nutzung sicherzustellen. Mit ECK existiert eine gute Alternative für das Deployment von Elastic, mit der die Vorteile von Kubernetes und des Operator Patterns genutzt werden können. Auch wenn dies das generelle Clustermanagement nicht komplett obsolet macht, werden so einige Arbeitsschritte obsolet und einige Aufgaben erleichtert. Elemente wie Clusterupgrades, das

Absichern des Clusters und dynamische Clusterskalierung sind in ECK out of the box verfügbar. Gleches gilt für eine vereinfachte Automatisierung.

Als Beispiel einer ECK-Installation wollen wir einen produktionsreifen Elastic Stack auf einem existierenden Kubernetes-Cluster installieren. Dieser kann entweder lokal via Minikube oder K3s erstellt werden, oder man deployt ihn auf einem verwalteten Cluster in der Cloud (etwa GCP oder EKS). Um zu starten, müssen wir die Custom Resource Definitions und die Operators auf dem Cluster installieren: `kubectl apply -f https://download.elastic.co/downloads/eck/1.0.0-beta1/all-in-one.yaml`. Der Befehl wendet alle Ressourcen an, aus denen sich die verschiedenen Komponenten-ECK zusammensetzen.

Jetzt haben wir die ECK-Komponenten installiert und können unsere Elastic-Konfiguration erstellen. In diesem Beispiel werden wir ein robustes Layout mit mehreren Mastern, separaten Daten-Nodes und dedizierten Ingress-Servern erstellen. Wir beginnen mit der Erstellung der Master-Nodes. Erstellen Sie eine neue Datei `elastic.yaml` und fügen den Code aus Listing 1 ein.

Die ersten beiden Zeilen definieren, welches Kubernetes API wir für die Erstellung der neuen Ressourcen verwenden werden; in diesem Fall ein API, das über das Elastic CRD und eine Ressourcenart von Elasticsearch bereitgestellt wird. Dieses Beispiel zeigt die Vorteile von CRDs als System zur Erweiterung von Kubernetes, das

Listing 1

```
apiVersion: elasticsearch.k8s.elastic.co/v1beta1
kind: Elasticsearch
metadata:
  name: monitoring
  namespace: monitoring
spec:
  version: 7.4.0
  nodeSets:
    - name: master
      count: 1
      config:
        node.master: true
        node.data: false
        node.ingest: false
        node.store.allow mmap: false
        resources:
          requests:
            memory: 4Gi
            cpu: 2
          limits:
            memory: 4Gi
            cpu: 2
        env:
          - name: ES_JAVA_OPTS
            value: "-Xms3g -Xmx3g"
  volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        storageClassName: standard
```

Fünf Vorteile des nativen Betreibens von Elasticsearch auf Kubernetes via ECK

- **Sicherheit von Beginn an:** ECK [1] konfiguriert die Sicherheitseinstellungen, Node to Node TLS, Zertifikate und einen Defaultuser für jeden Cluster automatisch.
- **Kubernetes-native Elasticsearch-Ressourcen:** Elasticsearch kann genau wie jede andere Kubernetes-Ressource genutzt werden. Es gibt keine Notwen-
- digkeit, endlose Kubernetes Pods, Services und Secrets zu konfigurieren.
- **Best Practices von Elastic:** ECK funktioniert im täglichen Elasticsearch-Betrieb nach den etablierten Prinzipien – von der Skalierung bis zum Versionswechsel.
- **Exklusive Elastic-Features:** Zugang zu allen Funktionen von Elastic, inklusive Elastic SIEM [2], Observability [3], Logs [4], Infrastruktur [5] und mehr.
- **Fortschrittliche Topologie:** Nutzer können die Vielseitigkeit der eigenen Kubernetes-Infrastruktur auf das Elasticsearch Deployment anwenden. So können etwa auch Hot-Warm-Cold-Architekturen [6] genutzt werden, um die Kosten zu reduzieren.

es Produkten erlaubt, Funktionalität in einem bestimmten Namespace hinzuzufügen.

Als Nächstes definieren wir einige Metadaten: Den Namen des Elastic Stack und den Namespace, in dem wir ihn erstellen werden. Das Hinzufügen eines Namespace hilft dabei, den ECK-Stack zu organisieren, sodass Sie Ihre Monitoring-Workload einfach von Ihren anderen Workloads trennen können. Das ermöglicht eine granularere Sicherheit innerhalb Ihres Kubernetes-Clusters. Wird der Namespace weggelassen, erstellt das ECK CRD alle Ressourcen im Standard-Namespace.

Wir kommen zum Kern der Konfiguration, der Spezifikation. Das ist im Wesentlichen der einzige Ort, an dem Konfigurationselemente für ECK-Komponenten abgelegt werden. In diesem Beispiel spezifizieren wir `nodeSet`; eine Menge von Elastic Nodes, seien es Master, Daten oder Ingests. Es ist eine Option, die in die Nodeset-Konfiguration übertragen wird. Die Nodes werden generiert und der zu erstellenden Elastic-Konfiguration hinzugefügt. Für unse-

Listing 2

```
podTemplate:
  spec:
    initContainers:
      - name: sysctl
        securityContext:
          privileged: true
        command: ['sh', '-c', 'sysctl -w vm.max_map_count=262144']
```

Listing 3

```
- name: ingest
  count: 2
  config:
    node.master: false
    node.data: false
    node.ingest: true
    node.store.allow_mmap: false
  resources:
    limits:
      memory: 4Gi
      cpu: 4
  env:
    - name: ES_JAVA_OPTS
      value: "-Xms3g -Xmx3g"
  volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
  spec:
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
    storageClassName: ssd
```

re Konfiguration setzen wir dieses spezielle Nodeset als Master Nodes ein und stellen sicher, dass wir drei haben. Das wird im Allgemeinen als Best Practice angesehen, da es (Quorum-basierte) Entscheidungsfindung innerhalb des Stacks ermöglicht. Im Allgemeinen sollten Sie immer eine ungerade Anzahl von Master Nodes haben.

Jetzt übermitteln wir die Konfigurationselemente. ECK unterstützt die meisten Optionen wie Nicht-ECK-Installationen. Besonders hervorzuheben ist in diesem Beispiel die Option `node.store.allow_mmap`. Dieses Konfigurationselement ist in manchen Fällen für Kubernetes-Nodes notwendig, die eine niedrige Kerneinstellung für `vm.max_map_count` besitzen. Diese Einstellung hat jedoch Auswirkungen auf die Leistung bei Stacks mit hohem Durchsatz, sodass sie nicht für stark ausgelastete Cluster empfohlen wird. Stattdessen sollte die Konfiguration aus Listing 2 zum Manifest innerhalb des Nodeset Tree hinzugefügt werden.

Diese Einstellung gibt die Kerneloptionen an den darunterliegenden Node weiter. Sie setzt jedoch voraus, dass Sie in der Lage sind, privilegierte Container auszuführen. Da die Privilegierung keineswegs sicher ist, insbesondere auf hochsicheren Kubernetes-Clustern, lohnt es sich, bei der Planung Ihres Elastic ECK Rollouts zu prüfen, ob sie verfügbar ist.

Wir kehren zu Listing 1 zurück und schauen uns das Festlegen von Ressourcenzuweisungen auf verschiedenen Ebenen innerhalb des Stacks an. Werden diese Einstellungen nicht vorgenommen, verwendet das CRD die Standardeinstellungen, was die Gesamtleistung des Clusters erheblich beeinflussen kann.

Die ersten beiden Elemente in diesem Teil, `resources` und `env`, setzen die Limitierungen des Volumens der verschiedenen Ressourcen, die das Nodeset verbrauchen darf. In unserem Beispiel darf jeder unserer drei Master Nodes 4 GB RAM und zwei CPU-Kerne aus dem Kubernetes-Cluster nutzen. Ohne diese Konfiguration können die Elastic Instances ein weitaus größeres Volumen an Ressourcen verbrauchen, als Sie vielleicht wünschen, was das Planen von Arbeitslasten innerhalb des Clusters beeinträchtigen kann. `env` setzt eine Umgebungsvariable, um die Limitierung an die JVM zu übergeben. Ohne diese Einstellung sind die JVM-Optionen auf 1 GB RAM voreingestellt, d. h. unabhängig davon, wie viel Sie im Kubernetes-Cluster zulassen, kann die JVM sie nicht effektiv nutzen.

Schließlich richten wir das `volumeclaimTemplate` ein. Es ist entscheidend, da es definiert, wie und wo die Daten für Elastic gespeichert werden. Standardmäßig setzt ECK einen Speicheranspruch von 1 GB innerhalb des Clusters. Diese Vorgabe stellt sicher, dass Daten von den Nodes getrennt gehalten werden, sodass die Informationen bei einem Node Replacement oder einer Löschung nicht verloren gehen. Er stellt jedoch nicht sicher, dass auf diesem Volumenanspruch genügend Platz oder Performance für eine Produktionslast vorhanden ist.

Stattdessen setzen wir, wie in diesem Beispiel, ein explizites `volumeclaimTemplate` und verwenden eine

storageClassName:-Einstellung, um sicherzustellen, dass der richtige Laufwerkstyp verwendet wird. Der Typ und die Größe des Laufwerks hängen von Ihrem Kubernetes-Anbieter ab. Meist wird ein Standard (plattenbasierte Speicherung oder langsame SSD) und eine Form von schnellerem Speicher (wie schnelle SSD oder auch exotischere schnelle Speicherung) angeboten. Auch hier ist die Abstimmung des Speichers auf die Workload einer der wichtigsten Aspekte von Elastic Deployments. ECK macht es leicht, den richtigen Speicher dem richtigen Node zur Verfügung zu stellen.

Nun, da wir unseren Master definiert haben, können wir einige Ingest Nodes definieren, indem wir die Konfiguration aus Listing 3 in unsere *elastic.yaml*-Datei hinzufügen.

Wie Sie erkennen können, sieht dies dem vorherigen Beispiel sehr ähnlich, da die gleiche CRD sie erstellt hat. In diesem Fall ändern wir drei Dinge, um dieses Nodeset für die Aufnahme bereit zu machen: *count*:, Grenzen für die *cpu*:-Limits und *storageClassName*:. Dieser Unterschied in den Einstellungen soll die besondere Rolle der Ingest Nodes berücksichtigen, da sie genug CPU zum Parsen eingehender Anfragen, eine schnelle Festplatte, um sie zu puffern, und kein Quorum benötigen.

Um unsere Daten-Nodes mit der gleichen Vorlage zu erstellen, fügen wir den Code aus Listing 4 in die Datei *elastic.yaml* ein.

Auch hier haben wir einige kleinere Änderungen vorgenommen: die Anzahl auf vier Nodes geändert, die CPU auf vier Cores erhöht und eine große Menge an High-Speed-Disks konfiguriert.

Wir haben jetzt alles, was wir brauchen, um Elastic in unserem Kubernetes-Cluster einzusetzen. Zunächst ist es jedoch sinnvoll, auf das Konzept der Pod Affinity einzugehen. Dieses erlaubt es Kubernetes, die Pods so zu planen, dass sie entweder auf dem gleichen Node eingeplant werden oder, im Fall der Anti-Affinity, getrennt gehalten werden, wo immer es möglich ist. Diese Eigenschaft ist besonders relevant im Fall von Elastic, wo man bestimmte Nodes sowohl aus Performance- als auch aus Zuverlässigkeitssgründen auseinanderhalten möchte.

Das Scheduling ist besonders für die Master relevant, da der Ausfall eines Kubernetes Node mit mehreren Mastern die Funktionsfähigkeit des Elastic Stack gefährden kann. Standardmäßig legt ECK eine Standard-Policy der Pod Anti-Affinity für alle von ihm erstellten Ressourcen fest, um sicherzustellen, dass Kubernetes versucht, sie auseinanderzuhalten. Diese Voreinstellung ist in den meisten Fällen problemlos, kann aber bei Bedarf mit der üblichen Kubernetes-Konfiguration überschrieben werden: <https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity>. Es ist erwähnenswert, dass man, um die Elastic-Komponenten korrekt getrennt zu halten, genügend Kubernetes Nodes benötigt, um dies zu unterstützen. Zum Beispiel braucht man bei drei Master-, zwei Daten- und zwei Ingest Nodes mindestens sieben Nodes, um sie

Anzeige

Listing 4

```
- name: data
  count: 2
  config:
    node.master: false
    node.data: true
    node.ingest: true
    node.store.allow_mmap: false
  resources:
    limits:
      memory: 3Gi
      cpu: 4
    env:
  - name: ES_JAVA_OPTS
    value: "-Xms3g -Xmx3g"
  volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 100Gi
        storageClassName: ssd
```

ECK legt auch ein Standard-Pod-Disruption-Budget für den Elastic Stack fest. Standardmäßig ist es auf 1 gesetzt, was bedeutet, dass immer nur ein Pod außer Betrieb sein kann.

getrennt zu halten, und diese Nodes müssen in der Lage sein, die Elastic Pods zu planen. Beim Ausführen des Stacks ist es sinnvoll, das Ausmaß der Co-Lokalisierung von Pods zu überwachen. Falls dies geschieht, sollte dem entweder durch das Festlegen von eindeutigen Richtlinien für die Anti-Affinity oder durch das Erhöhen der Anzahl der Kubernetes-Nodes entgegengewirkt werden.

ECK legt auch ein Standard-Pod-Disruption-Budget (pdb) für den Elastic Stack fest. Standardmäßig ist es auf 1 gesetzt, was bedeutet, dass immer nur ein Pod außer Betrieb sein kann. Diese Voreinstellung stellt sicher, dass Kubernetes bei der Umplanung der Elastic Pods versucht, diese auf jeweils einen Pod zu begrenzen. Diese Begrenzung ist bei der Aktualisierung des Stacks relevant, da sie sicherstellt, dass ein rolling Update erzwungen wird. Bei größeren Stapeln, bei denen mehr Pods gleichzeitig sicher außer Betrieb sein können, können Sie diese Grenze explizit höher setzen. Jetzt haben wir einige der Elemente, die ECK zur Gewährleistung der Stabilität hinzufügt, abgedeckt, sodass wir unseren Elastic-Cluster einsetzen können: `kubectl apply -f elastic.yaml`. Das wendet die Konfiguration auf Ihren Cluster an. Innerhalb weniger Minuten sollten Sie einen neuen Cluster eingerichtet haben, was Sie mit dem folgenden Befehl überprüfen können: `kubectl get pods -n elastic-cluster`. Sie sollten Folgendes erhalten:

monitor-es-mdi-0	1/1	Running	0	6m21s
monitor-es-mdi-1	1/1	Running	0	6m21s
monitor-es-mdi-2	1/1	Running	0	6m21s
monitor-kb-56bdd68b65-97wnz	1/1	Running	0	6m21s

Listing 5

```
apiVersion: kibana.k8s.elastic.co/v1beta1
kind: Kibana
metadata:
  name: monitoring
  namespace: monitoring
spec:
  version: 7.4.0
  count: 2
  elasticsearchRef:
    name: monitoring
  secureSettings:
    - secretName: kibana-enc-key
```

Schließlich müssen wir Kibana zu unserer Installation hinzufügen. Kibana wird mit einem separaten CRD zum Elastic-Cluster definiert. Fügen Sie den Code aus Listing 5 zu Ihrer `elastic.yaml`-Datei hinzu (oder sogar eine separate `yaml`-Datei, wenn Sie es vorziehen, sie getrennt zu halten).

Das sollte Ihnen mittlerweile ziemlich bekannt vorkommen. Wie bei den Elastic-Komponenten beginnen wir damit, dass wir am Anfang des Dokuments erklären, dass wir die ECK CRDs verwenden, und fragen nach einem Ressourcentyp *Kibana*. Wie beim Elastic Stack geben wir ihm dann sowohl einen Namen als auch einen Namespace. Die Spezifikationsversion ist wieder sehr ähnlich. Hier geben wir die Version von Kibana an, die wir installieren möchten, plus die Anzahl der Nodes, die wir haben wollen. Da wir ein Production Deployment anstreben, sind zwei die minimale Anzahl, die wir benötigen, da sie einen kontinuierlichen Service ermöglicht, wenn ein Pod außer Betrieb ist (je nach Umfang und SLAs, unter denen Sie arbeiten, können sogar mehr als zwei gewünscht werden). Die Einführung mehrerer Nodes führt zu einem zusätzlichen manuellen Schritt, da jeder Node im Cluster den gleichen Encryption Key verwenden muss. Um ihn zu ergänzen, fügen Sie mit dem folgenden Terminalbefehl ein neues Secret zu Ihrem Kube-Cluster hinzu: `kubectl create secret generic kibana-enc-key --from-literal=xpack.security.encryptionKey=94d2263b1ead716ae228277049f19975aff8644b4fcfe429c95143c1e90938md`

Der `xpack.security.encryptionKey` kann alles sein, was Sie möchten, solange er mindestens 32 Zeichen lang ist. Weitere Details zu dieser Einstellung finden sich im Bereich der Sicherheitseinstellungen in Kibana [7].

Jetzt haben wir einen Security Key festgelegt, wir können damit den Rest dieses Teils der Konfiguration abschließen. Das nächste Element, `elasticsearchRef:`, ist der Name des Elasticsearch-Clusters, mit dem wir diese Kibana-Instanz verbinden wollen. Dies sollte so eingestellt werden, dass es mit `name:` übereinstimmt, der in den Metadaten des Elastic-Cluster-Dokuments eingestellt ist, in diesem Fall *Monitoring*. Schließlich teilen wir den Kibana-Instanzen mit, wo der Security Key zu finden ist, den wir mit dem vorherigen Terminalbefehl mit dem `secureSettings`-Key eingerichtet haben. Das definiert das Grundlayout unseres Kibana-Clusters. Allerdings sollten wir, ähnlich wie beim Elastic Stack, gewisse Bedingungen für die Ressourcennutzung festlegen. Es kann auch sein, dass wir den Kibana-Dienst über einen Load Balancer verfügbar machen wollen, entweder

Anzeige

intern oder extern. Dazu fügen Sie das Code-Snippet aus Listing 6 zu Ihrer *kibana.yaml*-Datei hinzu.

Dieser Code gibt unserem Kibana-Cluster den letzten Schliff. Zu Beginn fügen wir ein Konfigurations-element hinzu, das an Kibana übergeben wird. Diese spezielle Einstellung *console.enabled*: schaltet die Entwicklerkonsole in Kibana aus und kann in sichereren Einstellungen wünschenswert sein. Der nächste Teil des Dokuments (*podTemplate*) ist im Wesentlichen das Gleiche wie das, was wir im Elastic-Cluster einstellen, und definiert, wie viele Ressourcen Kibana mindestens benötigt. Die *http*:-Einstellung ist der Ort, an dem wir den Benutzern unseren Kibana-Cluster präsentieren können, ohne dass sie einen Kube-Proxy oder andere, eher manuelle Methoden verwenden müssen, um in das Kubernetes-Cluster-Netzwerk zu gelangen. Das erlaubt uns, zwei wichtige Elemente zu definieren: die Art und Weise, wie der Kubernetes-Dienst definiert wird, und zusätzliche DNS-Namen innerhalb des generierten Zertifikats. Standardmäßig erstellt das ECK CRD Dienste als NodePorts. Das stellt sicher, dass der Zugriff nur von innerhalb des Clusters kommen kann, und bedeutet im Allgemeinen, dass Sie sich in den jeweiligen Node einloggen müssen, um Kibana zu sehen. Durch das Über-schreiben dieser Vorgabe zu einem *LoadBalancer*-Typ wird der Dienst von außerhalb des Clusters zugänglich gemacht. Die letzte Einstellung *tls*: ermöglicht es, den vom Dienst erstellten Zertifikaten zusätzliche DNS-Namen hinzuzufügen. Das sollte etwaige Zertifikatsfehler bei der externen Verbindung verhindern.

Wenn Sie mit den Einstellungen zufrieden sind, speichern und wenden Sie diese mit dem Befehl *kubectl apply -f kibana.yaml* an. Dadurch werden die Einstellungen auf

Listing 6

```
config:
  console.enabled: false
podTemplate:
  spec:
    containers:
      - name: kibana
    resources:
      requests:
        memory: 1Gi
        cpu: 0.5
      limits:
        memory: 2Gi
        cpu: 2
    http:
      service:
        spec:
          type: LoadBalancer
      tls:
        selfSignedCertificate:
          subjectAltNames:
            - dns: monitoring.example.com
```

Ihren Kubernetes-Cluster angewendet. Wenn Sie keinen Load Balancer verwendet haben, sollten Sie innerhalb weniger Minuten in der Lage sein, sich mittels eines Kube-Proxys mit dem Nodeport zu verbinden. Oder mit der Load-Balancer-Adresse, wenn Sie den Kibana-Log-in-Bildschirm angezeigt bekommen. Um sich einzuloggen, benutzen Sie den Benutzernamen von Elastic und rufen mit folgendem Terminalbefehl das automatisch gesetzte Passwort ab: *kubectl get secret -n monitoring monitoring-es-elastic-user -o=jsonpath='{.data.admin-password}' | base64 --decode; echo*. Beachten Sie besonders *-n*, um sicherzugehen, den richtigen Namespace anzuwählen, der in der Clustererstellung generiert wurde. Das sollte Ihnen ein Passwort liefern, um sich in Kibana einzuloggen.

Hoffentlich haben Sie jetzt einen laufenden, produktionsbereiten Elastic-Cluster mit einer Kibana-Konsole, mit der Sie ihn abfragen können. Dieser Artikel hat die Benutzerfreundlichkeit und Leistungsfähigkeit des ECK-Produkts untersucht und gezeigt, wo es konfiguriert werden kann, um es widerstandsfähig und bereit für Benutzer zu machen. In einem nächsten Artikel werden wir untersuchen, wie wir mit Hilfe von Metricbeat und Filebeat-Daten in unserem neuen Cluster bekommen können.



Dimitri Marx ist Solutions Architect bei Elastic. Er hat Informatik studiert und arbeitet seit seinem Abschluss an Open-Source-Projekten. Dimitri unterstützt Kunden dabei, den Wert von Suchtechnologien zu verstehen und wie man diese im Enterprise-Kontext einsetzen kann. Logging, Security und Analysemetriken gehören zu seinen Tätigkeitsfeldern, genau wie die Lösung der größten Herausforderungen auf dem Gebiet der strukturierten und unstrukturierten Daten.

Links & Literatur

- [1] <https://www.elastic.co/de/blog/announcing-elastic-cloud-on-kubernetes-eck-0-9-0-alpha-2>
- [2] <https://www.elastic.co/de/blog/elastic-siem-7-3-0-released>
- [3] <https://www.elastic.co/de/blog/kubecon-2019-elastic-doubles-down-on-observability-and-orchestration-for-kubernetes>
- [4] <https://www.elastic.co/de/blog/elastic-logs-app-released>
- [5] <https://www.elastic.co/de/blog/elastic-infrastructure-7-4-0-released>
- [6] <https://www.elastic.co/de/blog/implementing-hot-warm-cold-in-elasticsearch-with-index-lifecycle-management>
- [7] <https://www.elastic.co/guide/en/kibana/current/security-settings-kb.html>

Anzeige

Teil 5: Verschlüsselung für Java-Entwickler

Migration nach AWS

In den vier vorangegangenen Teilen dieser Serie haben wir uns mit der schrittweisen Optimierung einer Anwendung hin zu einer Serverless Architecture beschäftigt. Dieser Teil legt den Fokus auf eine der wichtigsten Sicherheitsmaßnahmen jeder Anwendung: Verschlüsselung von Daten.

von Steffen Grunwald

Bei Amazon Web Services (AWS) empfehlen wir das Architekturprinzip „Encrypt Everything“. Es war selten so kostengünstig und einfach, eben dieses Prinzip auf hohem Standard zu verwenden, wie nun mit Cloud-Diensten. Dieser Artikel zeigt, wie Daten auf verschiedenen Ebenen verschlüsselt werden können – und welche AWS-Dienste diesbezüglich relevant sind. Dabei arbeiten wir uns von der einfachen One-Click-Lösung vor bis hin zur Verschlüsselung auf Feldebene in einer Datenbank.

Verschlüsselung ist ein wichtiges Werkzeug, das Nutzern der Cloud hohe Sicherheit gibt, den alleinigen Zugriff auf ihre Daten zu kontrollieren. Auch regulatorische und industriespezifische Anforderungen legen nahe, sensible Daten durch eine Anwendung vor unautorisiertem Zugriff oder Veränderung zu schützen.

Im Allgemeinen wird unterschieden zwischen Verschlüsselung at rest (zur Persistierung) und in transit (zur Übertragung). Dieser Artikel konzentriert sich auf die Verschlüsselung persistenter Daten. Sie gilt als zusätzlicher Schutz neben der Berechtigungsvergabe von Aktionen für bestimmte Ressourcen mit Hilfe von AWS Identity and Access Management (IAM) und z. B. Berechtigungen, die bei einer relationalen Datenbank für Benutzer vergeben werden.

Artikelserie

- Teil 1: Warum wechseln?
- Teil 2: Verwendung von Managed Services
- Teil 3: Verwendung von Containern
- Teil 4: Serverless mit AWS Lambda
- Teil 5: Verschlüsselung für Java-Entwickler**
- Teil 6: Optimierung der Architektur

Die große Herausforderung bei Verschlüsselung ist, dass wirklich alles richtig gemacht werden muss. Ein kleiner Fehler bei der Wahl des Schlüssels und des Verschlüsselungsverfahrens kann dazu führen, dass der Schutz, der durch Verschlüsselung erreicht werden soll, nicht effektiv ist. Durch eine falsche Handhabung des Schlüssels kann dieser in fremde Hände gelangen oder verloren gehen. Im ersten Fall werden damit möglicherweise Daten offenbart. Im anderen können die Daten nicht mehr entschlüsselt werden und sind für immer nutzlos.

Key Management mit Hardware Security Modules

Bevor wir in die Verwendung von Schlüsseln eintauen, betrachten wir die Erzeugung und Speicherung von Schlüsseln. Zu diesem Zweck gibt es Hard- und Softwarelösungen, die an Kriterien wie Wartbarkeit, Kosten, Ausfallsicherheit, Skalierbarkeit, Schutz vor Schlüsselextraktion und Manipulation, Integration in andere Lösungen und Auditierbarkeit gemessen werden. Hardware Security Modules (HSM) bieten einen sehr hohen Schutz vor Schlüsselextraktion. Ver- und Entschlüsselung finden in dem Modul statt und werden protokolliert. Erkennt das Modul Anzeichen physischer Manipulation, werden die Schlüssel automatisch vernichtet. Viele Kunden entscheiden sich wegen des hohen Schutzniveaus bei der Verwaltung des Schlüsselmaterials im selbstverwalteten Rechenzentrum für eine derartige Lösung und erwarten eine solche auch in der Cloud.

Schlüsselverwaltung im AWS Key Management Service

In AWS können Kunden Schlüssel durch den AWS Key Management Service (KMS) verwalten. Sie legen darin sogenannte Customer Master Keys (CMK) an (Kasten: „Arten von Customer Master Keys“). Ein CMK enthält Metadaten, die Verbindung zu dem IAM-System sowie sogenannte HSM Backing Keys (HBK). Die HBKs sind

das Schlüsselmaterial, mit dem KMS effektiv operiert und mit dem Inhalten durch KMS ver- und entschlüsselt werden. AWS übernimmt die Skalierung und Wartung der an das KMS Frontend angeschlossenen HSMs. KMS bietet über sein Application Programming Interface (API) eine hohe Integration mit derzeit über fünfzig anderen AWS-Diensten [1], wie zum Beispiel Amazon Elastic Block Store Volumes (EBS) oder dem Amazon Relational Database Service (RDS). Es kann mit Hilfe eines AWS Software Developer Kits (SDK) auch direkt aus einer Anwendung heraus genutzt werden. Jede Verwendung von CMKs wird lückenlos in AWS CloudTrail protokolliert – zwecks Auditierung davon, wann und von wem welcher Schlüssel wie für welche Ressource verwendet wurde. Diese Eigenschaften werden regelmäßig von unabhängigen Prüfern validiert und zertifiziert, zum Beispiel in den Compliance-Programmen FIPS 140-2 und SOC, die über AWS Artifact [2] einsehbar sind. Für mehr Informationen zu KMS sei hier auf das Whitepaper zu KMS Cryptographic Details [3] verwiesen.

Envelope Encryption

Eine einzige Ver- oder Entschlüsselungsoperation von KMS kann bis zu vier Kilobyte Daten verschlüsseln. Anwendungen verschlüsseln aber oft Daten nicht direkt mit KMS, sondern nutzen Envelope Encryption. Dabei erzeugt KMS einen Data Key und gibt diesen unter einem CMK verschlüsselt und im Klartext an die Anwendung. Die Anwendung verschlüsselt die Daten mit dem Data Key, speichert den verschlüsselten Data Key zusammen mit den verschlüsselten Daten und löscht deren jeweilige Klartexte aus dem Speicher. Dieser Mechanismus vermeidet Latenz bei der Datenübertragung: Die erzeugten 256-Bit-Schlüssel sind meist kleiner und schneller zu übermitteln als die zu verschlüsselnden Daten. Außerdem ist für die Verschlüsselung von Objekten, die größer als vier Kilobyte sind, nur ein einziger Aufruf in KMS erforderlich.

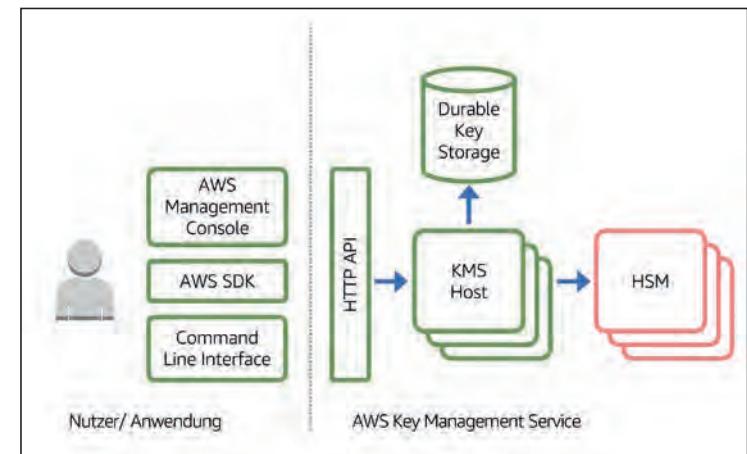


Abb. 1: Schematischer Aufbau des AWS Key Management Service

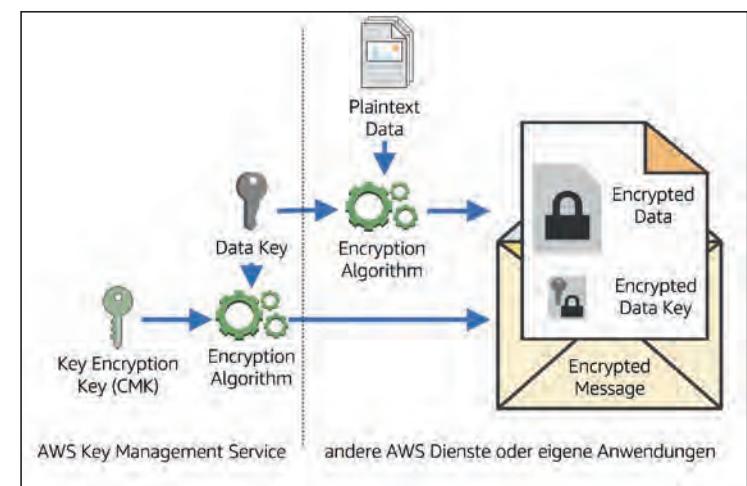


Abb. 2: Bei der Envelope Encryption liefert KMS den Data Key, die Anwendung speichert ihn zusammen mit den Daten verschlüsselt ab

Volumes und Datenbanken mit einem Klick verschlüsseln

Ähnlich wie im vorangegangenen Beispiel verschlüsselt auch Amazon EBS Volumes einer EC2-Instanz mit Hilfe von Envelope Encryption und KMS. Der Benutzer

Arten von Customer Master Keys

CMK werden in den folgenden Varianten von AWS-Diensten verwendet:

- **AWS-owned CMKs** werden von AWS außerhalb des Kunden-AWS-Accounts verwaltet. Tabellen in der NoSQL-Datenbank Amazon DynamoDB werden z. B. immer mit einem solchen CMK verschlüsselt, wenn der Kunde keinen anderen CMK bestimmt. Die CMKs sind regionsspezifisch (z. B. für Frankfurt oder Irland), werden aber für mehrere AWS-Accounts verwendet.
- **AWS-managed CMKs** werden von AWS verwaltet. Sie sind kunden-, regions- und Service-spezifisch, d. h. der CMK, der sich zum Beispiel hinter dem Alias `aws/s3` verbirgt, wird in nur einem AWS-Account und in einer Region verwendet.
- **Customer-managed CMKs** werden vom Kunden verwaltet. Im Gegensatz zu den zuvor genannten CMKs kann der Kunde hier

selbst über die Key Policy Deaktivierung, Löschung und Rotation bestimmen.

- **Customer-managed CMKs with imported key material** erlauben dem Kunden, eigenes Schlüsselmaterial zu importieren. Der Kunde kann es bei Bedarf löschen und zu einem späteren Zeitpunkt wieder importieren.

Die Wahl einer CMK-Variante ist abhängig von den Anforderungen und dem Service. Oft fällt die Wahl auf Customer-managed CMKs, wenn ein Kunde den Lebenszyklus des CMKs voll kontrollieren will und Berechtigungen feingranular erteilen möchte. Dem stehen im Vergleich zu einem selbst verwalteten HSM geringe Kosten von derzeit einem US-Dollar pro Monat und Schlüsselversion gegenüber. Hinzu kommen nur noch die Kosten für die Anzahl der Anfragen – z. B. 0,03 US-Dollar pro 10 000 Anfragen in der EU-Region (Frankfurt) von AWS.

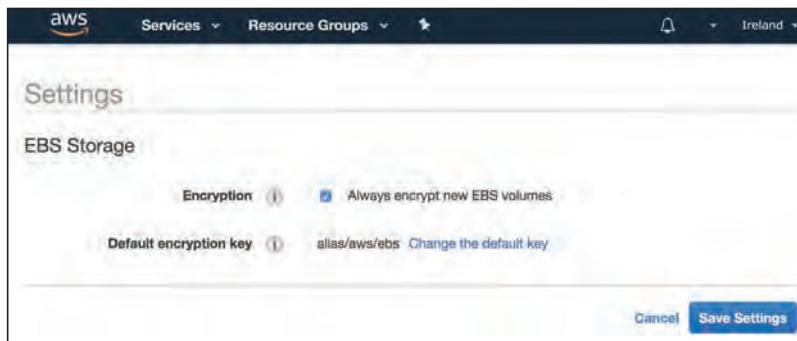


Abb. 3: Die Verschlüsselung eines EBS Volumes in der AWS Console mit einem Klick

wählt dazu bei der Erstellung einer EC2-Instanz den CMK aus. Das EBS-Subsystem generiert durch KMS individuell für jedes Volume einen verschlüsselten Data Key und übergibt diesen an den EC2-Host. Der Host nutzt KMS bei jedem Start der EC2-Instanz, um den Data Key zu entschlüsseln. Nur der verschlüsselte Data Key wird dauerhaft gespeichert. Bei der Erstellung einer Datenbank in Amazon RDS greift derselbe Mechanismus. Amazon RDS nutzt die Amazon-EBS-Verschlüsselung, um Datenbank-Volumes vollständig zu verschlüsseln.

Um eine generelle Verschlüsselung von EC2 Volumes für eine Region zu erzwingen, wählt ein Benutzer einen CMK in den EC2-Settings [4] aus und legt fest, dass EC2 in der jeweiligen Region ausschließlich verschlüsselte Volumes erstellen darf.

Feingranulare Verschlüsselung von Daten in der Anwendung

Mit der zuvor beschriebenen Verschlüsselung der Datenbank auf Volume-Ebene sind Daten des Volume nur lesbar, wenn der Zugriff von RDS auf den CMK erteilt wurde, der bei der Erstellung der Datenbank vergeben wurde. Ist die Datenbank gestartet, kann eine Anwendung jedoch alle Daten in der Datenbank im Klartext lesen – entsprechende Datenbankberechtigungen vorausgesetzt.

Eine Anwendung kann zusätzlich KMS nutzen, um Daten auf Feldebene mit Envelope Encryption zu ver-

Listing 1

```
String password = "verysecret";
AwsCrypto crypto = new AwsCrypto();
Map<String, String> encContext = Collections.singletonMap("USER",
                                                               "steffeng");
KmsMasterKeyProvider keyProvider = KmsMasterKeyProvider.builder()
    .withDefaultRegion("eu-central-1")
    .withKeysForEncryption("arn:aws:kms:eu-central-1:<ACCOUNT_ID>;alias/
                           prod/passwordManager")
    .build();
CryptoResult<String, KmsMasterKey> cryptoResult =
    crypto.encryptString(keyProvider, password, encContext);
String cipherText = cryptoResult.getResult();
```

schlüsseln. Zum Beispiel könnten die Passwörter in einem Passwortmanager so individuell verschlüsselt werden. Der Vorteil ist der feingranulare Schutz mit einem individuellen Schlüssel pro Datensatz, eine umfangreiche Protokollierung aller Schlüsseloperationen und der Nachweis der Integrität der Daten.

Die Schritte, die eine Anwendung zur Verschlüsselung machen muss, sind folgende:

1. Die Anwendung fordert einen neuen Data Key zu einem CMK in KMS an. KMS liefert den Data Key sowohl im Klartext als auch verschlüsselt mit dem CMK an die Anwendung.
2. Die Anwendung verschlüsselt ein Feld mit dem Data Key mit einem geeigneten Verschlüsselungsalgorithmus.
3. Das verschlüsselte Feld und der verschlüsselte Data Key werden von der Anwendung in einem einzigen Chiffriertext enkodiert.
4. Die Anwendung speichert den Chiffriertext in der Datenbank.
5. Die Anwendung löscht den unverschlüsselten Data Key aus dem Speicher.

Die Anwendung könnte zwar direkt das KMS API nutzen. Das AWS Encryption SDK for Java [5] erleichtert jedoch die Schritte 1 bis 3 mit wenigen Zeilen Code, wie in Listing 1 zu sehen ist.

Das AWS Encryption SDK (AwsCrypto) verschlüsselt das geheime Passwort in Frankfurt (*eu-central-1*) unter dem Schlüssel mit dem Alias *prod/passwordManager*. Dabei wird der Encryption Context mit dem Benutzernamen des Passwortbesitzers mit in den Chiffriertext eingebunden. *cipherText* ist ein langer String wie z. B. *AYADeE74xUvbUyPPx9/...*.

Mit den Defaulteinstellungen aus dem Codebeispiel verwendet das AWS Encryption SDK zur Verschlüsselung den Advanced Encryption Standard (AES) im Galois Counter Mode (GCM) mit 256-Bit-Schlüsseln.

Encryption Context

In der dritten Codezeile fällt der Encryption Context auf, der einen zusätzlichen Schutzmechanismus bietet. Wenn ein Angreifer die Datenbank unseres Passwortmanagers vollständig unter Kontrolle brächte, könnte er alle Werte auslesen und verändern. Ohne CMK-Berechtigung sind die Informationen nutzlos. Tauscht er jedoch ein eigenes verschlüsseltes Passwort gegen das eines anderen Benutzers aus, ist ihm der Zugriff auf den Klartext des Passworts über die Anwendung möglich. Der Encryption Context schützt vor diesem Szenario. Die Verschlüsselung bindet den optionalen Encryption Context als sogenannte Additionally Authenticated Data in den Chiffriertext ein. Beim Entschlüsseln wird

damit eine Manipulation erkannt. Im Codebeispiel ist der Benutzername, dem das Passwort gehört, in den Encryption Context eingebunden. Es lassen sich aber noch weitere Informationen wie die ID der Passwortseite oder der Umgebungsname hinzufügen.

Ein Codebeispiel für eine Entschlüsselungsoperation ist in Listing 2 aufgeführt. Sollte der Encryption Context nach der Entschlüsselung abweichen, deutet das auf einen Fehler oder eine Manipulation hin. Auf keinen Fall sollte dann der Wert im Klartext dem Benutzer gezeigt werden.

Der Chiffriertext wird vom AWS Encryption SDK entschlüsselt. Nach der Verschlüsselung ermittelt der Code, ob der entschlüsselte Encryption Context die erwarteten Werte enthält, dargestellt durch die Variable *contextMatch*.

Nicht nur das AWS Encryption SDK, sondern auch das KMS API für Ver- und Entschlüsselung unterstützen einen Encryption Context. Im Unterschied zum eigenen Anwendungscode wird der Encryption Context automatisch von KMS nach der Entschlüsselung abgeglichen, und es besteht keine Möglichkeit, den Klartext bei Abweichungen von KMS zu erhalten.

Berechtigungen auf Schlüssel feingranular vergeben

Wie bei vielen AWS-Ressourcen ist eine explizite Vergabe von Berechtigungen notwendig. Ohne Berechtigung darf kein Benutzer, keine Rolle und auch kein AWS-Dienst einen Schlüssel verwenden.

Berechtigungen werden über eine dedizierte Key Policy an jedem Customer-managed CMK vergeben. Jeder CMK, der über die AWS Management Console angelegt wird, erhält eine Key Policy, die die Berechtigungen an IAM delegiert. Das heißt, jeder Benutzer mit entsprechenden IAM-Berechtigungen kann den Schlüssel verwenden. Durch Änderung der Key Policy lassen sich Verantwortlichkeiten teilen und sehr genau eingrenzen. Werden Aktionen für feste Benutzer und Rollen festgelegt, darf z.B. nur ein bestimmter Benutzer den CMK administrieren oder löschen, während nur eine den Produktionsservern zugeordnete Rolle den CMK zur Ver- oder Entschlüsselung nutzen darf.

Listing 2

```
String cipherText = "AYADeE74xUvbUyPPx9RlzvlAEJ0AbwACAARVU0VSAAhzdGVmZmVuZwAVYXdzLWNyeXB0by1wdWJ[...]";
AwsCrypto crypto = new AwsCrypto();
Map<String, String> encContext = Collections.singletonMap("USER", "steffeng");
KmsMasterKeyProvider keyProvider = KmsMasterKeyProvider.builder()
    .withDefaultRegion("eu-central-1")
    .withKeysForEncryption("arn:aws:kms:eu-central-1:<ACCOUNT_ID>:alias/prod/passwordManager")
    .build();
CryptoResult<String, KmsMasterKey> cryptoResult =
    crypto.decryptString(keyProvider, cipherText);
boolean contextMatch = encContext.entrySet().stream()
    .allMatch(entry -> entry.getValue()
        .equals(cryptoResult.getEncryptionContext().get(entry.getKey())));
String password = cryptoResult.getResult();
```

Zur Entschlüsselung eines Volumes braucht sogar Amazon EBS die Berechtigung, einen CMK zu nutzen. Vor der Verwendung eines Volumes vergibt der Benutzer einen sogenannten Grant für die Entschlüsselung eines Data Keys an Amazon EBS. Der Grant schreibt über den Encryption Context vor, dass die Entschlüsselung ausschließlich für dieses Volume stattfinden darf.

Fazit

AWS Key Management Service (KMS) ist derzeit mit über fünfzig AWS-Diensten integriert. So ist für die Verschlüsselung von Amazon EBS Volumes und Datenbanken nur ein einziger Klick notwendig. Je nach Schutzbedarf lässt sich das KMS API auch in der Anwendung nutzen. Das AWS Encryption SDK for Java vereinfacht die Verschlüsselung sensibler Daten auf Feldebene. KMS nutzt Hardware Security Modules, bietet lückenlose Protokollierung aller Ver- und Entschlüsselungsoperationen und ein feingranulares Rechtemanagement. Damit ist es möglich, mit minimalem Aufwand auf hohem Standard alles zu verschlüsseln.



Steffen Grunwald ist Solutions Architect bei Amazon Web Services EMEA SARL. Er unterstützt dort überwiegend Kunden aus dem Enterprise-Umfeld auf dem Weg in die Cloud.

@steffeng

Links & Literatur

- [1] <https://aws.amazon.com/kms/features/>
- [2] <https://console.aws.amazon.com/artifact>
- [3] <https://d0.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf>
- [4] <https://console.aws.amazon.com/ec2/home?#EC2Settings:java.html>
- [5] <https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/java.html>



Javas neuer HTTP-Client

Was lange währt ...

Aktuelle Trends der Softwareentwicklung wie Microservices oder Cloud führen dazu, dass immer häufiger verteilte Systeme entstehen. Diese erfordern Kommunikation, und HTTP zählt hier zu den am häufigsten eingesetzten Protokollen. Somit ist es enorm wichtig, dass eine so populäre Programmiersprache wie Java eine zeitgemäße Unterstützung für den Versand von HTTP Requests mitbringt. Überraschenderweise war das sehr lange Zeit nicht der Fall.

von Thilo Frotscher

Über viele Jahre hinweg bestand die HTTP-Unterstützung von Java SE aus der (abstrakten) Klasse *HttpURLConnection* und davon abgeleiteten Klassen. *HttpURLConnection* wurde mit Java 1.1 im Jahr 1997 Komponenten-(!) eingeführt. Zwar wurde die Klasse über die Jahre immer mal wieder leicht erweitert, im Grunde mussten Entwickler aber bis zum Erscheinen von Java SE 11 mit einer vollkommen veralteten Unterstützung für HTTP leben. Ein Umstand, der doch stark verwundert, bedenkt man die Wichtigkeit des Protokolls in der Softwareentwicklung der letzten zehn bis fünfzehn Jahre. So existieren in *HttpURLConnection* keine dedizierten Methoden für das Setzen und Auslesen von HTTP-Headern oder zur Authentifizierung. Weiterhin gibt es keine eingebaute Unterstützung für asynchrone Requests. Entwickler müssen sich um das Erzeugen,

Starten und Verwalten nebenläufiger Threads also selbst kümmern. Es gibt keine Unterstützung für PATCH oder gar HTTP/2. Und zu allem Überfluss ist auch der Programmieraufwand beim Einsatz von *HttpURLConnection* recht hoch, da das API der Klasse hoffnungslos veraltet und gemessen an aktuellen Standards sehr umständlich ist. So erfordert der einfache Versand eines HTTP Requests eine aus heutiger Sicht beinahe absurd anmutende Anzahl von Codezeilen (Listing 1).

All das führte in der Praxis dazu, dass viele Entwickler seit Jahren einen Bogen um *HttpURLConnection* machten. Stattdessen kamen häufig Frameworks oder Third-Party-Bibliotheken wie etwa Apache HttpClient [1] zum Einsatz. Nicht zu unterschätzen ist insbesondere die fehlende Unterstützung für HTTP/2. Denn durch Verbesserungen wie Headerkompression, nur eine einzige Verbindung pro Server und Multiplexing wird die Kommunikation mit HTTP/2 deutlich effizienter.

Der neue HttpClient

Mit Java SE 11 wurde endlich Abhilfe geschaffen und die (abstrakte) Klasse *HttpClient* in den Standard-sprachumfang aufgenommen. Sie bietet ein modernes API auf Basis des Builder-Patterns, sodass unterschiedliche Aspekte wie SSL-Details, Proxy Selector, Authenticator, Time-out für den Verbindungsaufbau, Verhalten bei Redirects oder ein Cookie-Handler auf einfache Weise konfiguriert werden können (Listing 2).

Das gilt auch für die zu verwendende Protokollversion. *HttpClient* sendet alle Requests standardmäßig mit HTTP/2. Unterstützt ein Server diese Protokollversion

Listing 1: Versand von HTTP Requests mit HttpURLConnection

```
try {
    URL url = new URL("http://...");

    HttpURLConnection con = (HttpURLConnection) url.openConnection();
    con.setRequestMethod("GET");

    // set request headers
    con.setRequestProperty("Accept", "application/json");

    // set timeouts (default is infinity!)
    con.setConnectTimeout(1000);
    con.setReadTimeout(5000);

    // execute the request
    con.connect();

    // determine status
    int responseStatus = con.getResponseCode();

    // read response body
    Reader streamReader;

    if (responseStatus > 399) {
        streamReader = new InputStreamReader(con.getErrorStream());
    } else {
        streamReader = new InputStreamReader(con.getInputStream());
    }

    BufferedReader in = new BufferedReader(streamReader);

    String inputLine;
    StringBuffer content = new StringBuffer();

    while ((inputLine = in.readLine()) != null) {
        content.append(inputLine);
    }
    in.close();

    con.disconnect();

} catch (IOException e) {
    // handle exception
}
```

noch nicht, wird der Request automatisch auf HTTP/1.1 herabgestuft. Alternativ können Anwendungsentwickler aber das Protokoll mit Hilfe des Builders auch gleich auf HTTP/1.1 festlegen, falls das sinnvoll erscheint.

Nachdem eine Instanz von *HttpClient* konfiguriert wurde, kann eine beliebige Anzahl von Requests damit versendet werden. Es sollte darauf geachtet werden, dass der *HttpClient* tatsächlich wiederverwendet und nicht für jeden Request ein neuer *HttpClient* erzeugt wird. Denn der *HttpClient* erzeugt hinter den Kulissen einen Verbindungspool. Das ermöglicht es, bestehende Verbindungen wiederzuverwenden und so den Overhead des wiederholten Verbindungsaufbaus zu umgehen. Die Kapazität des Verbindungs pools ist standardmäßig unbegrenzt und einzelne Verbindungen werden bis zu 1 200 Sekunden bzw. 20 Minuten gehalten (Keepalive Time-out). Beides kann mit den System-Properties *jdk.httpclient.keepalive.timeout* und *jdk.httpclient.connectionPoolSize* konfiguriert werden. Diese sind entweder beim Start der virtuellen Maschine anzugeben oder mittels *System.getProperty(...)* zu setzen, in jedem Fall aber bevor die Klasse *jdk.internal.net.http.ConnectionPool* geladen wird. Später können die Werte nicht mehr geändert werden.

Jeder Request wird durch eine Instanz von *HttpRequest* repräsentiert, zu deren Erzeugung ebenfalls ein Builder API bereitsteht. Auf diesem Weg können Einzelheiten wie URI, Request-Header, Request-Methode oder Read Time-out gesetzt werden (Listing 3).

Im Falle eines POST oder PUT Requests ist zudem eine Datenquelle notwendig, aus der der Body des Requests befüllt wird. Hierfür sind sogenannte *BodyPublisher* zuständig, die die Daten beispielsweise aus einem String, einem *InputStream*, einem Bytearray oder einer Datei auslesen (Listing 4).

Ist der *HttpRequest* zusammengebaut, kann er versendet werden. Für einen synchronen Versand dient dabei die Methode *send*. Sie erwartet eine Instanz von *HttpRequest* und einen *BodyHandler* für die Verarbeitung der Response. Ein *BodyHandler* ermöglicht es, den Statuscode und die HTTP-Header der Response

Listing 2: Konfiguration eines HttpClient

```
Authenticator myAuthenticator = new Authenticator() {
    @Override
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication("bob", "bobsPwd".toCharArray());
    }
};

HttpClient httpClient =
    HttpClient.newBuilder()
        .authenticator(myAuthenticator)
        .connectTimeout(Duration.ofMillis(500))
        .followRedirects(HttpClient.Redirect.NORMAL)
        .version(HttpClient.Version.HTTP_2)
        .build();
```

zu untersuchen, bevor der eigentliche Response Body ausgelesen wird. Der *BodyHandler* erzeugt hierfür einen *BodySubscriber*, der das Gegenstück zum *BodyPublisher* darstellt, der beim Versand von Requests zum Einsatz kommt. Das heißt, der *BodySubscriber* liest den Body der HTTP Response aus und wandelt diesen in einen String, ein Bytearray, einen *InputStream* oder eine Datei um. Listing 5 zeigt ein typisches Beispiel dafür, wie ein synchroner Request versendet und der Response Body in einen String geleitet wird. Der Rückgabewert der Methode *send* ist dabei vom Typ *HttpResponse<>*. Diese bietet Zugriff auf den Statuscode, Response-Header und natürlich auf den Response Body.

Reactive Streams

Wie sich anhand der genannten Begrifflichkeiten bereits erahnen lässt, sind die Request und Response Bodies als Reactive Streams [2] umgesetzt. Der *HttpClient* dient dabei als Subscriber des Request Body und als Publisher des Response Body. Für Anwendungsentwickler bieten die Klassen *HttpRequest* und *HttpResponse* eine Reihe von Factory-Methoden, mit deren Hilfe bereits mitgelieferte Request Publisher und Response Subscriber für gängige Datentypen wie String, Dateien oder Bytearrays erzeugt werden können (Listing 6). Sie sammeln entweder Daten an, bis der übergeordnete Java-Typ (z. B. ein String) erzeugt werden kann, oder sie streamen die Daten, etwa im Fall einer Datei, in den Request Body hinein. Alternativ können die Interfaces *BodySubscriber* und *BodyPublisher* auch selbst implementiert werden, wenn benutzerdefinierte Reactive Streams zum Einsatz kommen sollen.

Listing 3: Erzeugen eines HttpRequest für GET

```
HttpRequest getRequest =
    HttpRequest.newBuilder()
        .uri(URI.create("http://..."))
        .timeout(Duration.ofMillis(1000))
        .header("Accept", "application/json")
        .GET()
        .build();
```

Listing 4: Erzeugen eines HttpRequest für POST

```
Path pathToJsonFile = Paths.get("my-data.json");

HttpRequest postRequest =
    HttpRequest.newBuilder()
        .uri(URI.create("http://..."))
        .timeout(Duration.ofMillis(1000))
        .header("Content-type", "application/json")
        .POST(BodyPublishers.ofFile(pathToJsonFile))
        .build();
```

Zudem gibt es Adapter, die einen Publisher oder Subscriber aus *java.util.concurrent.Flow* in einen *BodyPublisher* bzw. *BodySubscriber* umwandeln:

```
HttpRequest.BodyPublishers::fromPublisher(...)
```

```
HttpResponse.BodyHandlers::fromSubscriber(...)
```

```
HttpResponse.BodyHandlers::fromLineSubscriber(...)
```

Asynchroner Versand

Eine weitere wichtige Verbesserung, die der neue *HttpClient* mitbringt, ist die direkte Unterstützung asynchroner Kommunikation, also des Versands von HTTP Requests in separaten Threads. *HttpURLConnection* hatte hierfür keine eingebaute Unterstützung. Daher mussten sich Anwendungsentwickler selbst um die Verwaltung der notwendigen Threads kümmern, insbesondere auch um die Prüfung, ob für einen asynchronen Request bereits eine Response vorliegt oder ob eventuell ein Fehler aufgetreten ist. Das gestaltet sich besonders bei verschachtelten bzw. aufeinander aufbauenden asynchronen Aufrufen recht schnell sehr komplex. Dabei entsteht stark verschachtelter Code, der schwer lesbar und wartbar ist.

HttpClient unterstützt den asynchronen Versand von HTTP Requests mit Hilfe der Methode *sendAsync*. Diese liefert als Rückgabewert nicht etwa ein Future, sondern ein *CompletableFuture*. Die Klasse *CompletableFuture* und das von ihr implementierte Interface *CompletionStage* wurden bereits mit Java SE 8 eingeführt, sind

Listing 5: Versand und Einlesen des Response Bodys in einen String

```
HttpResponse<String> response =
    httpClient.send(postRequest, BodyHandlers.ofString());

if (response.statusCode() != 201) {
    // handle unexpected status code
}

String responseBody = response.body();
```

Listing 6: Erzeugen von Request Publisher und Response Subscriber

```
HttpRequest.BodyPublishers::ofByteArray(byte[])
HttpRequest.BodyPublishers::offile(Path)
HttpRequest.BodyPublishers::ofString(String)
HttpRequest.BodyPublishers::ofInputStream(Supplier<InputStream>)

HttpResponse.BodyHandlers::ofByteArray()
HttpResponse.BodyHandlers::ofString()
HttpResponse.BodyHandlers::offile(Path)
HttpResponse.BodyHandlers::discarding()
```

Anzeige

aber erstaunlich vielen Java-Entwicklern noch immer unbekannt. Die zugrunde liegenden Konzepte können im Rahmen dieses Artikels zwar nicht umfassend erläutert werden; zusammenfassend lässt sich aber festhalten, dass *CompletableFuture* und *CompletionStage* dabei helfen, mehrstufige asynchrone Abläufe mit deutlich weniger Aufwand zu implementieren und dabei gleichzeitig wesentlich besser lesbaren Code zu produzieren.

Mehrstufige Abläufe sind gerade auch im Fall von HTTP APIs an der Tagesordnung. Ein Beispiel hierfür wäre etwa ein Anwendungsfall, bei dem im ersten Schritt Preisanfragen an mehrere Lieferanten gesendet werden, aus deren Antworten dann im zweiten Schritt der beste (oder auch der durchschnittliche) Einkaufspreis ermittelt wird. Im dritten Schritt wird der so ermittelte Einkaufspreis an einen weiteren Service übermittelt, der daraus anhand unterschiedlicher Faktoren einen Verkaufspreis ermittelt. Im vierten und letzten Schritt soll dieser Verkaufspreis in einer Benutzungsoberfläche angezeigt oder anderweitig veröffentlicht werden.

All dies auf synchrone Art und Weise zu implementieren, könnte dazu führen, dass die Anwendung eine ganze Weile blockiert ist. Denn die HTTP Requests an die beteiligten Services können aufgrund von Verzögerungen oder Verbindungsschwierigkeiten schon mal länger dauern. Während dieser Wartezeit sollte die Anwendung jedoch sinnvollere Dinge erledigen, als nur zu warten. Beispielsweise könnten in der Zwischenzeit weitere Anfragen oder Benutzereingaben verarbeitet werden. Eine nebenläufige Umsetzung des Anwendungsfalls ist demnach empfehlenswert. Zudem sollten die Preisanfragen an die unterschiedlichen Lieferanten nicht nacheinander, sondern idealerweise parallel erfolgen. Schließlich könnte man überlegen, dass nicht der günstigste Lieferant gewinnt, sondern derjenige, der am schnellsten eine Preisauskunft sendet, oder zumindest der günstigste, der innerhalb einer maximalen Wartezeit antwortet.

Listing 7: Asynchroner Versand

```
HttpClient httpClient = ...

HttpRequest priceEnquiryRequest =
    HttpRequest.newBuilder()
        .uri(URI.create("http://..."))
        .timeout(Duration.ofMillis(1000))
        .header("Accept", "application/json")
        .GET()
        .build();

CompletableFuture<HttpResponse<String>> futureHttpResponse =
    httpClient.sendAsync(priceEnquiryRequest, BodyHandlers.ofString());

futureHttpResponse
    .thenApply(this::extractPurchasePrice)
    .exceptionally(this::fallbackToMostRecentPurchasePrice)
    .thenApply(this::calculateSellingPrice)
    .thenAccept(this::publishSellingPrice);
```

CompletableFuture greift Anwendungsentwicklern nun zunächst dadurch unter die Arme, dass notwendige Threads hinter den Kulissen gestartet werden. Diese Threads werden einem Threadpool entnommen, der optional vom Anwendungsentwickler selbst erstellt und an *CompletableFuture* übergeben wird. Im Zusammenspiel mit *HttpClient* würde an dessen *Builder* ein anwendungsspezifischer *ThreadPool* übergeben werden, der vom *HttpClient* dann wiederum an *CompletableFuture* weitergereicht wird.

Ein weiteres wichtiges Feature von *CompletableFuture* besteht darin, dass im Fall mehrstufiger Prozesse das Warten auf die Beendigung von Threads sowie das Abholen der Ergebnisse und deren Übergabe an die nächste Stufe ebenfalls komplett hinter den Kulissen geschieht. Auch das stellt eine enorme Erleichterung für den Anwendungsentwickler dar. Man spricht hier auch vom sogenannten Push-Verfahren, bei dem die Ergebnisse der vorherigen Verarbeitungsstufe (Stage) automatisch in die nächste Stufe geschoben werden, sobald sie vorliegen. Das ist deutlich eleganter als die von Future bekannten Pull- und Poll-Verfahren, bei denen man aktiv nachfragen und gegebenenfalls warten muss, um ein Ergebnis zu erhalten und dieses dann anschließend auch selbst weiterreichen zu müssen. Listing 7 zeigt beispielhaft, wie *HttpClient* und *CompletableFuture* gemeinsam dabei helfen, mehrstufige asynchrone Abläufe zu implementieren.

Fazit

Nachdem Java-Entwickler über viele Jahre mit einer hoffnungslos veralteten Unterstützung für den Versand von HTTP Requests leben und aus diesem Grund oftmals auf Third-Party-Bibliotheken ausweichen mussten, wurde mit Java SE 11 endlich Abhilfe geschaffen. Der neue *HttpClient* bietet ein zeitgemäßes API, Unterstützung für HTTP/2, Reactive Streams und asynchronen Versand auf Basis von *CompletableFuture*. In vielen Fällen wird es dadurch möglich, bestehende Projekte um eine externe Abhängigkeit zu erleichtern, den Code deutlich lesbarer und wartbarer zu gestalten sowie die Effizienz der Kommunikation zu verbessern. Für die vielen Entwicklerteams und zahlreichen Anwendungen, die noch immer auf dem Stand von Java SE 8 verharren, ist der neue *HttpClient* ein weiterer Grund, endlich auf Java SE 11 (oder neuer) zu migrieren.



Thilo Frotscher ist freiberuflicher Softwarearchitekt und Trainer mit über 20 Jahren praktischer Erfahrung. Als Experte für Enterprise Java, APIs und Systemintegration unterstützt er seine Kunden überwiegend durch Entwicklung, Reviews oder die Durchführung von maßgeschneiderten Schulungen. Thilo ist (Co-)Autor mehrerer Bücher, hat zahlreiche Fachartikel verfasst und spricht regelmäßig auf Fachkonferenzen und Schulungsveranstaltungen sowie bei Java User Groups.

feedback@frotscher.com [@thfro](https://twitter.com/thfro)

Links & Literatur

- [1] Apache HttpComponents: <https://hc.apache.org/>
- [2] Reactive Streams: <https://www.reactive-streams.org/>

Anzeige

Die Anbindung an externe HTTP Services richtig testen

Bessere Integrations-tests mit WireMock

Mit der verstärkten Verbreitung von Microservices steigen auch die Abhängigkeiten zwischen den verschiedenen Anwendungen. Aufgrund der eigenen Datenhoheit müssen Services auf die ein oder andere Art Informationen miteinander austauschen, um ihr (Business-)Ziel zu erreichen. Häufig ist REST bzw. HTTP dafür das Mittel der Wahl. Egal, ob man der klassischen Testpyramide oder einem der neuen Ansätze wie der Testhonigwabe [1] folgt: Irgendwann während der Entwicklung sollte die Kommunikation mit Integrationstests geprüft werden. Das ist der Punkt, an dem WireMock [2] ins Spiel kommt.

von Ronny Bräunlich

Als Beispiel für diesen Artikel soll uns ein einfaches System, bestehend aus zwei Services, dienen, wobei sich der zweite unserer Kontrolle entzieht. **Abbildung 1** stellt das Ganze stark vereinfacht dar.

Unsere Anwendung bietet eine REST-Schnittstelle an, über die sich Daten abfragen lassen. Diese befinden sich aber nicht in der eigenen Datenhaltung, sondern müssen von einem weiteren, externen Service geholt werden. Unsere Implementierung könnte ein Anti-Corruption-Layer im Sinne von Domain-driven Design darstellen, die externen Daten aufbereiten oder die geholten Daten für Berechnungen benötigen (z. B. aktuelle Wechselkurse). Die Möglichkeiten sind zahlreich.

Um unser Beispiel nicht abstrakt halten zu müssen, wird der externe Service durch das Chuck Norris Fact API [3] dargestellt. Der Web Service bietet eine gut dokumentierte Schnittstelle, die uns im JSON-Format Fak-

ten über den allseits bekannten Schauspieler liefert. Das gesamte Beispiel basiert auf Spring, kann aber leicht auf andere Frameworks übertragen werden.

Wir verwenden einen einfachen REST Controller namens *ChuckNorrisFactController* als API für unseren Microservice. Neben den Businessklassen gibt es den *ChuckNorrisService*. Dieser kapselt den Aufruf zum externen API. Er verwendet das Spring RestTemplate, um den HTTP-Aufruf auszuführen. Es wird ein zufälliger

Listing 1

```
@RestController
public class ChuckNorrisFactController {
    private final ChuckNorrisService chuckNorrisService;

    @Autowired
    public ChuckNorrisFactController(ChuckNorrisService chuckNorrisService) {
        this.chuckNorrisService = chuckNorrisService;
    }

    @GetMapping(path = "/fact", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    public ChuckNorrisFact getFact(){
        return chuckNorrisService.retrieveFact();
    }
}
```

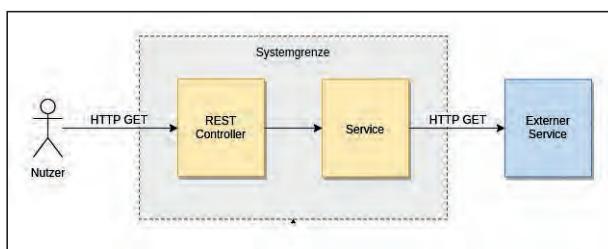


Abb. 1: Beispielsystem

Fakt abgerufen und das erhaltene JSON geringfügig verändert. Wir ignorieren den Status, der im gleichnamigen Feld der Antwort steckt und geben einfach den Wert des Felds *joke* zurück. Listing 1 zeigt den Controller und Listing 2 die einfache Implementierung des Service, die wir im Laufe des Artikels verbessern werden. Das komplette Beispiel ist auf GitHub zu finden [4].

Was oft zu sehen ist, sind Tests, die das RestTemplate mocken und eine vorgefertigte Antwort zurückgeben. Neben den üblichen Unit-Tests zur Überprüfung der Erfolgsfälle gibt es meist einen Test, der den Fehlerfall abdeckt, also mit einem 4xx- oder 5xx-Statuscode antwortet. Listing 3 zeigt diesen typischen Testfall mit Hilfe von Mockito [5] Mocks.

Test und Implementierung erfüllen die meisten Anforderungen, die man an den Code haben könnte. Alles ist mehr oder weniger lesbar und wir decken den Fehlerfall ab. Die ResponseEntity gibt einen 503-Fehlercode zurück und die Anwendung stürzt nicht ab. Der Nutzer erhält zwar immer denselben Fakt, aber unser Service wird durch externe Fehler nicht beeinträchtigt. Alle Tests sind grün und unsere Anwendung ist einsatzbereit.

Doch leider funktioniert Springs RestTemplate so nicht. Die Methodensignatur von *getForEntity* gibt uns einen kleinen Hinweis auf das tatsächliche Verhalten. Dort steht, dass eine RestClientException geworfen werden kann. Hier unterscheidet sich das gemockte RestTemplate von der tatsächlichen Implementierung. Wir werden niemals eine ResponseEntity mit einem 4xx- oder 5xx-Statuscode erhalten. Das RestTemplate

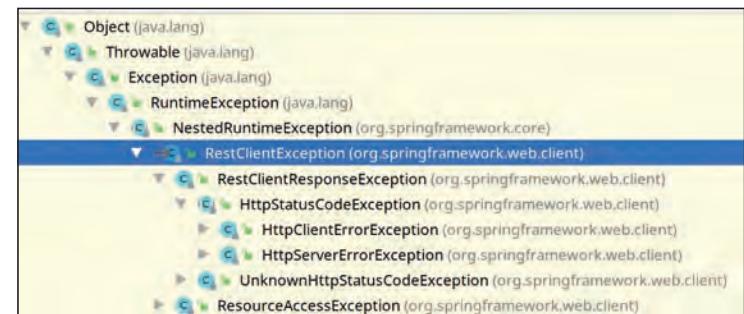


Abb. 2: Klassenhierarchie der RestClientException

wird eine Unterklasse von RestClientException werfen. Wenn wir einen Blick auf die Klassenhierarchie in Abbildung 2 werfen, erhalten wir einen guten Eindruck davon, welche Exception möglicherweise geworfen wird. Sehen wir also, wie wir den Test und unsere Implementierung verbessern können.

WireMock eilt zur Rettung

WireMock simuliert Web Services, indem es einen Mock-Server startet und Antworten zurückgibt, die vorher konfiguriert wurden. Der Einstieg in Tests mit WireMock ist relativ einfach und das Mocking von Anfragen dank einer guten DSL ebenfalls. Für JUnit 4 gibt es eine WireMockRule, die beim Starten und Stoppen des Servers hilft. Für JUnit 5 muss dies aktuell selbst gemacht werden. Im Beispielprojekt befindet sich der *ChuckNorrisServiceIntegrationTest*. Es handelt sich um einen Spring-Boot-Test auf Basis von JUnit 4. Der wichtigste Teil darin ist die *ClassRule*:

```

@Service
public class ChuckNorrisService {

    static final ChuckNorrisFact BACKUP_FACT = new ChuckNorrisFact(
        -1L, "It works on my machine" always holds true for Chuck
        Norris.");}

    private final RestTemplate restTemplate;
    private final String url;

    @Autowired
    public ChuckNorrisService(RestTemplate restTemplate, @Value("${fact.
        url}") String url) {
        this.restTemplate = restTemplate;
        this.url = url;
    }

    public ChuckNorrisFact retrieveFact() {
        ResponseEntity<ChuckNorrisFactResponse> response = restTemplate.
            getForEntity(url, ChuckNorrisFactResponse.class);
        return Optional.ofNullable(response.getBody()).map(ChuckNorrisFact-
            Response::getFact).orElse(BACKUP_FACT);
    }
}
  
```

```

@ClassRule
public static WireMockRule wireMockRule = new WireMockRule();
  
```

Wie bereits erwähnt, startet und stoppt sie den WireMock-Server. Die Regel kann auch als normale *@Rule* verwendet werden, um den Server für jeden einzelnen Test zu starten und zu stoppen. In unserem Fall ist das nicht nötig. Im Test gibt es verschiedene

Listing 3

```

@Test
public void shouldReturnBackupFactInCaseOfError() {
    String url = "http://localhost:8080";
    RestTemplate mockTemplate = mock(RestTemplate.class);
    ResponseEntity<ChuckNorrisFactResponse> responseEntity = new
        ResponseEntity<>(HttpStatus.SERVICE_UNAVAILABLE);
    when(mockTemplate.getForEntity(url, ChuckNorrisFactResponse.class)).
        thenReturn(responseEntity);
    var service = new ChuckNorrisService(mockTemplate, url);

    ChuckNorrisFact retrieved = service.retrieveFact();

    assertThat(retrieved).isEqualTo(ChuckNorrisService.BACKUP_FACT);
}
  
```

Listing 4

```
public void configureWireMockForOkResponse(ChuckNorrisFact fact) throws
    JsonProcessingException {
    ChuckNorrisFactResponse chuckNorrisFactResponse = new ChuckNorrisFactResponse("succ-
        cess", fact);
    stubFor(get(urlEqualTo("/jokes/random"))
        .willReturn(okJson(OBJECT_MAPPER.writeValueAsString(chuckNorrisFactResponse))));
}

private void configureWireMockForErrorResponse() {
    stubFor(get(urlEqualTo("/jokes/random"))
        .willReturn(serverError()));
}
```

Listing 5

```
public ChuckNorrisFact retrieveFact() {
    try {
        ResponseEntity<ChuckNorrisFactResponse> response = restTemplate.getForEntity(url,
            ChuckNorrisFactResponse.class);
        return Optional.ofNullable(response.getBody()).map(ChuckNorrisFactResponse::get-
            Fact).orElse(BACKUP_FACT);
    } catch (HttpStatusCodeException e){
        return BACKUP_FACT;
    }
}
```

Listing 6

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@RunWith(SpringRunner.class)
@ContextConfiguration(initializers = ChuckNorrisServiceIntegrationTest.
    WireMockPortInitializer.class)

@TestPropertySource(properties = {
    "fact.url=http://localhost:${wiremock.port}/jokes/random"
})
public class ChuckNorrisServiceIntegrationTest {

    private static final ObjectMapper OBJECT_MAPPER = new ObjectMapper();

    @ClassRule
    public static WireMockRule wireMockRule = new WireMockRule();

    @Autowired
    private ChuckNorrisService service;

    static class WireMockPortInitializer implements ApplicationContextInitializer<Config-
        urableApplicationContext> {

        @Override
        public void initialize(ConfigurableApplicationContext configurableApplicationContext) {
            TestPropertyValues.of("wiremock.port:" + wireMockRule.port())
                .applyTo(configurableApplicationContext);
        }
    }
    ...
}
```

configureWireMockFor...-Methoden. Listing 4 zeigt die Methoden für den Erfolgs- und den Fehlerfall. Beide enthalten die Anweisungen an WireMock, wann welche Antwort zurückgegeben werden soll. Die Aufteilung der WireMock-Konfiguration in mehrere Methoden ist ein Ansatz, der sich in meinen Tests bewährt hat. Die einzelnen Tests rufen dann die Methode auf, die sie benötigen. So ist immer klar zu sehen, was WireMock antworten soll. Natürlich könnte auch alles zusammen in einer großen *@Before*-Methode konfiguriert werden.

Alle Methoden in Listing 4 werden statisch von *com.github.tomakehurst.wiremock.client.WireMock* importiert. Für den Erfolgsfall konfigurieren wir einen Stub, der bei einem HTTP GET auf den Pfad */jokes/random* mit einem JSON-Objekt antwortet. Die *okJson()*-Methode ist nur eine Abkürzung für eine Antwort mit dem Statuscode 200 und JSON-Inhalt. Um unser Antwortobjekt in JSON umzuwandeln, hilft uns der Jackson [6] ObjectMapper. Wie zu sehen ist, ist die Konfiguration für den Fehlerfall noch einfacher. Dank der WireMock DSL kann auf einen Blick erfasst werden, was für ein Aufruf erwartet wird und wie er beantwortet werden soll.

Wenn wir nun den Integrationstest mit WireMock starten, sehen wir, dass unsere Implementierung den Fehlerfall wirklich nicht abdeckt und der Test aufgrund der vom RestTemplate geworfenen Exception fehlschlägt. Folglich müssen wir unseren Code um Exception Handling erweitern, wie in Listing 5 zu sehen.

Damit ist bereits gezeigt, wie WireMock für einfache Anfrage-Antwort-Tests verwendet werden kann. Mit Hilfe der DSL konfigurieren wir, welche Anfragen erwartet und welche Antworten zurückgeliefert werden sollen. Sollte eine Anfrage kommen, für die keine Antwort konfiguriert wurde, teilt uns WireMock die erhaltenen Parameter mit und warum sie nicht passten. So kann entweder der Test oder der Code angepasst werden.

Ein dynamischer Port für WireMock

Nachdem man eine Weile mit WireMock und Spring gearbeitet hat, und im Besonderen, wenn die Integrations tests in einer Cloud-Umgebung ausgeführt werden, tritt ein Problem auf, für das es bisher noch keine WireMock-native Lösung gibt. Um WireMock mit unserem RestTemplate ansprechen zu können, müssen wir den Port kennen. Aber wenn wir einen Port hart codieren, kann es sein, dass er in der Cloud-Umgebung, die sich meist unserer Kontrolle entzieht, belegt ist. Abhilfe schafft hier ein Spring *ApplicationContextInitializer*.

Schauen wir zurück auf Listing 2. Dem Service wird der URL zum Chuck Norris Fact API über die *@Value*-Annotation injiziert. Das bedeutet, dass wir den URL über eine *@TestPropertySource*-Annotation überschreiben können. Allerdings müssen wir den dynamischen Port mit den Test-Properties zusammen bekommen. Genau hier hilft uns der *ApplicationContextInitializer*. Wir fügen dem Anwendungskontext den dynamisch

Anzeige

zugewiesenen Port hinzu und können dann im Beispiel über die Property \${wiremock.port} darauf verweisen. Der einzige Nachteil ist, dass nun eine @ClassRule verwendet werden muss. Andernfalls kann nicht auf den Port zugegriffen werden, bevor die Spring-Anwendung initialisiert wurde. Die ersten Zeilen des angepassten Tests mit dem Initializer als innere Klasse sind in Listing 6 zu sehen.

Timeouts

WireMock bietet viel mehr Möglichkeiten für Testfälle als nur einfache Antworten auf GET- oder POST-Anfragen. Ein weiterer Testfall, der abgedeckt werden sollte, ist das Testen von Timeouts. Oftmals wird, teils aus Unwissen, teils aus Unachtsamkeit, vergessen, Timeouts an RestTemplates oder URLConnections zu setzen. Ohne Time-outs warten beide standardmäßig unendlich lange auf Antworten. Im besten Fall wird das fehlende Time-out nicht bemerkt, im schlimmsten Fall warten alle Threads auf eine Antwort, die nie ankommen wird, und unsere Anwendung wird unbenutzbar. Daher sollten wir einen Test hinzufügen, der einen Time-out simuliert. Natürlich könnten wir eine verzögerte Antwort auch mit einem Mockito-Mock erzeugen, aber in diesem Fall würden wir wieder raten, wie sich das RestTemplate verhält. Eine Verzögerung zu simulieren, lässt sich mit der WireMock DSL einfach umsetzen, wie Listing 7 zeigt.

`withFixedDelay()` erwartet einen *int*-Wert, der die Verzögerung in Millisekunden repräsentiert. Alternativ gäbe es noch Methoden, um eine zufällige Verzögerung zu simulieren und die Methode `withChunkedDribble-`

Listing 7

```
private void configureWireMockForSlowResponse() throws JsonProcessingException {
    ChuckNorrisFactResponse chuckNorrisFactResponse = new ChuckNorrisFactResponse("success", new ChuckNorrisFact(1L, ""));
    stubFor(get(urlEqualTo("/jokes/random"))
        .willReturn(
            okJson(OBJECT_MAPPER.writeValueAsString(chuckNorrisFactResponse))
            .withFixedDelay((int) Duration.ofSeconds(10L).toMillis())));
}
```

Listing 8

```
public ChuckNorrisFact retrieveFact() {
    try {
        ResponseEntity<ChuckNorrisFactResponse> response = restTemplate.getForEntity(url,
            ChuckNorrisFactResponse.class);
        return Optional.ofNullable(response.getBody()).map(ChuckNorrisFactResponse::getFact).orElse(BACKUP_FACT);
    } catch (RestClientException e){
        return BACKUP_FACT;
    }
}
```

`Delay()`, die die Antwort in mehrere Blöcke aufteilt und sie über einen bestimmten Zeitraum zurückgibt.

Nachdem wir ein Time-out auf unserem `restTemplate` gesetzt und einen Test für eine langsame Antwort hinzugefügt haben, sehen wir, dass `restTemplate` im Fall eines Time-outs eine `ResourceAccessException` wirft. Folglich genügt unsere Anpassung aus Listing 5 noch nicht. Um die Exception zu fangen, könnten wir natürlich einen weiteren Catch-Block hinzufügen oder den vorhandenen Catch-Block zu einem Multi-Catch umwandeln. Ein Blick zurück auf Abbildung 2 zeigt uns aber, dass wir die Oberklasse beider Exceptions, die `RestClientException` fangen können. Listing 8 zeigt die finale Version unserer Methode.

Fazit

Dieser Artikel sollte zwei Aspekte genauer beleuchten. Zum einen sollte die Bedeutung von Integrationstest herausgestellt werden. Unit-Tests sind ein wichtiger Teil der Entwicklung, aber Mocks helfen nur bis zu einem bestimmten Punkt. Deshalb sollten die Unit-Tests durch Integrationstests ergänzt werden.

Zum anderen sollte gezeigt werden, wie Integrations- tests mit WireMock geschrieben werden und wie sie uns bei der Entwicklung helfen können. Als Leser haben Sie hoffentlich ein Gefühl für die Nutzung und den Funktionsumfang von WireMock bekommen. Wer noch mehr über WireMock wissen möchte, sei auf die Dokumentation verwiesen [7]. Die Möglichkeiten für Tests gehen weit über einfache HTTP-GET-Anfragen hinaus. Man könnte z. B. Headerwerte noch genauer prüfen, und mit seinem Stateful Behaviour bietet WireMock die Option, weitaus komplexere Szenarien zu simulieren.



Ronny Bräunlich war nach seinem Master am KAIST zwei Jahre als IT-Consultant bei der codecentric AG beschäftigt. Seit 2019 arbeitet er bei der ProMaterial GmbH als Lead Developer und versucht dort, mit Hilfe guter Tests und neuester Technologien die Baustoffbranche zu digitalisieren. Nebenbei ist er noch Maintainer der Gating JDBC Erweiterung und der camunda BPM Platform OSGi Integration.



@code_n_roll



<https://blog.code-n-roll.dev>

Links & Literatur

- [1] <https://labs.spotify.com/2018/01/11/testing-of-microservices/>
- [2] <http://wiremock.org/>
- [3] <http://api.icndb.com/>
- [4] <https://github.com/rbraeunlich/wiremock-integrationtest>
- [5] <https://site.mockito.org/>
- [6] <https://github.com/FasterXML/jackson>
- [7] <http://wiremock.org/docs/>



Ivy voll integriert, Lazy-Loading-Komponenten und Lokalisierung

Was bringt Angular 9?

Angular 9 bringt Ivy in einer abwärtskompatiblen Variante, damit kleinere Bundles. Die i18n-Lösung wurde umfangreich überarbeitet und einige Ecken abgerundet. So stehen dem Entwickler und zukünftigen Versionen von Angular neue Möglichkeiten zur Verfügung.

von Manfred Steyer

Mit Angular 9 soll es so weit sein: Ivy wird endlich Standard. Das Angular-Team hat sehr viel Energie investiert, um Ivy abwärtskompatibel zum Vorgänger ViewEngine zu machen. Ein Blick auf den Changelog [1] verrät, dass genau das der Fokus von Version 9 war. Abseits davon gibt es auch sehr nette Neuerungen. Ich stelle anhand von Beispielen diejenigen vor, die uns die tägliche Arbeit vereinfachen werden. Dazu verwende ich ein paar Beispiele, die sich in meinem GitHub Repository finden [2].

Update

Das Update auf Angular 9 ist, wie schon von den Vorgängerversionen gewohnt, sehr geradlinig. Der CLI-Befehl `ng update @angular/core @angular/cli` reicht aus, um die Bibliotheken auf den neuesten Stand zu heben. Eventuelle Breaking Changes werden dabei soweit als möglich automatisiert berücksichtigt. Möglich machen das Migrationsskripte, die auf dem CLI-internen

Codegenerator Schematics basieren und den bestehenden Quellcode geringfügig modifizieren. Für die Fälle, in denen wir manuell eingreifen müssen, geben die verwendeten Schematics eine Meldung aus. Weitere Informationen zur Migration finden sich unter [3].

Ivy by default

Die wichtigste Neuerung bei Angular 9 ist wohl, dass der neue Ivy-Compiler, an dem seit vielen Monaten gearbeitet wird, standardmäßig aktiviert ist. Unsere Bundles werden damit nach der Umstellung auf Version 9 um bis zu 40 Prozent kleiner. Wie sehr unsere Anwendung dieses Potenzial ausschöpfen kann, hängt von ihrem Aufbau ab.

Um Breaking Changes zu vermeiden, wurde besonderes Augenmerk auf Abwärtskompatibilität gelegt. Das hat auch damit zu tun, dass bei Google über 1 500 Angular-Anwendungen im Einsatz sind. Diese Anwendungen sollen auch noch nach der Umstellung auf Version 9 funktionieren und halfen gleichzeitig, die Qualität von Ivy sicherzustellen. Zusätzlich kam eine Vielzahl weit

Abb. 1:
Dashboard
mit Lazy
Loading

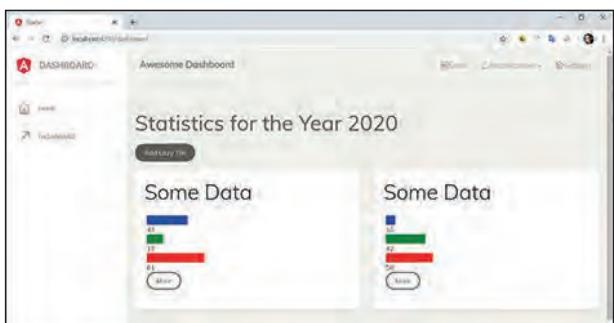
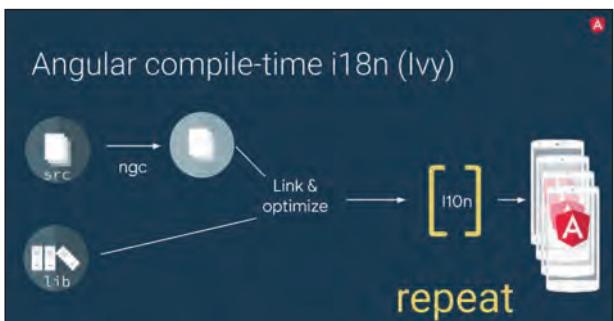


Abb. 2:
Funktions-
weise von
@angular/
localize [5]



verbreiteter Angular-Bibliotheken zur Qualitätssicherung zum Einsatz. Da das Angular-Team den gesamten Unterbau austauschen musste, ist Ivy kein einfaches Unterfangen. Daher kann es vorkommen, dass Angular Ivy in Randfällen den einen oder anderen Fehler zu Tage fördert. In diesen Fällen können wir Ivy mit der Eigenschaft `enableIvy` in der `tsconfig.json` deaktivieren:

```
"angularCompilerOptions": {
  "enableIvy": false
}
```

Das Angular-Team ist an solchen Fällen interessiert und freut sich über entsprechende Bug Reports unter [4].

Lazy Loading von Komponenten

Auf den ersten Blick bringt Ivy kleinere Bundles. Doch die Architektur von Ivy hat noch viel mehr zu bieten, sodass

Listing 1

```
export class DashboardPageComponent implements OnInit, OnChanges {
  [...]

  @ViewChild('vc', {read: ViewContainerRef, static: true})
  viewContainer: ViewContainerRef;

  [...]

  constructor(
    private injector: Injector,
    private cfr: ComponentFactoryResolver) {}

  [...]
}
```

wir in den nächsten Releases neue, darauf aufbauende Features erwarten können. Ein neues Feature ist heute schon verfügbar: Lazy Loading von Komponenten. Es war zwar schon von Anfang an integriert, doch mussten immer ganze Angular-Module geladen werden, da ViewEngine die Metadaten für den Einsatz der einzelnen Komponenten auf Modulebene untergebracht hat. Ivy verstaut diese Metadaten nun beim Kompilieren direkt in den Komponenten, somit können sie auch separat bezogen werden. Um den Einsatz dieser neuen Möglichkeit zu verdeutlichen, nutze ich hier ein einfaches Dashboard, das die anzugezeigenden Kacheln per Lazy Loading bezieht (Abb. 1). Hierfür benötigen wir zunächst einen Platzhalter, in den die Komponente geladen werden kann. Das kann jeder beliebige Tag sein, solange er mit einer Templatevariablen markiert wird: `<ng-container #vc> </ng-container>`. Templatevariablen beginnen, wie man sieht, mit einer Raute. Die zugehörige Komponente kann dieses Element als `ViewChild` laden (Listing 1).

Außerdem benötigen wir für das dynamische Erzeugen der `lazy`-Komponente den aktuellen `Injector` sowie einen `ComponentFactoryResolver`. Beides lässt sich in den Konstruktor injizieren. Danach ist alles sehr geradlinig: Über einen dynamischen Import lässt sich die `lazy`-Komponente laden und der `ComponentFactoryResolver` ermittelt die Factory, die wir zum Instanziieren der Komponente benötigen (Listing 2). Die Methode `createComponent` des `ViewContainer` nimmt die Factory entgegen und erzeugt eine Instanz der Komponente. Damit sie beim Dependency-Injection-Mechanismus von Angular angeschlossen wird, gilt es, auch den aktuellen `Injector` zu übergeben. Anschließend ruft das Beispiel die Komponenteninstanz ab, setzt Eigenschaften und ruft die `ngOnChanges`-Methode auf. Das Ergebnis ist eine `DashboardTileComponent`, die mitten im `ViewContainer` erscheint.

Eine weitere kleine, aber feine Neuerung ist, dass die Anwendung solche dynamischen Komponenten nicht

Listing 2

```
import('../dashboard-tile/dashboard-tile.component').then(m => {
  const comp = m.DashboardTileComponent;

  // Only b/c of compatibility; will not be needed in future!
  const factory =
    this.cfr.resolveComponentFactory(comp);

  const compRef = this.viewContainer.createComponent(
    factory, null, this.injector);

  const compInstance = compRef.instance;

  compInstance.a = Math.round(Math.random() * 100);
  compInstance.b = Math.round(Math.random() * 100);
  compInstance.c = Math.round(Math.random() * 100);
  compInstance.ngOnChanges();
});
```

mehr in den `entryComponents` registrieren muss. Dieses für viele ohnehin schwer verständliche Konstrukt existiert nur mehr für den Fall eines Fallbacks auf `ViewEngine`.

Besseres i18n mit @angular/localize

Die seit Version 2 in Angular integrierte Lokalisierungslösung (i18n) war in erster Linie auf Performance getrimmt. Sie erzeugt pro Locale (Kombination aus Sprache und Land) einen Build und passt beim Komprimieren die Übersetzungstexte ein, sodass sich zur Laufzeit kein Overhead ergibt. Das hatte ein paar Nachteile:

- Das Erstellen all dieser Builds war zeitintensiv.
- Es bestand keine Möglichkeit, die Übersetzungstexte zur Laufzeit festzulegen.
- Es war nicht möglich, Übersetzungstexte programmatisch zu nutzen.
- Die Sprache konnte nicht zur Laufzeit geändert werden. Stattdessen musste die Anwendung den Benutzer auf die gewünschte Sprachversion weiterleiten.

Mit Angular 9 erscheint eine neue Lösung, die die meisten dieser Nachteile kompensiert und dabei trotzdem keine Abstriche in Sachen Performance macht. Sie nennt sich `@angular/localize` und lässt sich über das gleichnamige npm-Paket beziehen. Das neue `@angular/localize` erzeugt zunächst einen einzigen Build. Pro Locale legt sie danach eine Kopie an und bringt dort die Übersetzungstexte ein (**Abb. 2**).

Außerdem ergänzt `@angular/localize` jede Kopie um Metadaten für die Formatierung von Zahlen und Datumswerten entsprechend den Gebräuchen der jeweiligen Sprache. Wir müssen also nicht mehr die benötigten Metadaten beim Programmstart importieren und registrieren. Wie im Angular-Umfeld üblich, lässt sich die Bibliothek via `ng add` beziehen: `ng add @angular/localize`. Danach können wir die zu übersetzenden Texte in den Templates mit dem Attribut `i18n` versehen. Um dem Übersetzungsstudio Kontextinformationen zu bieten, erhält dieses Attribut einen Wert mit der Bedeutung und einer Beschreibung: `<h1 i18n="meaning|description@@home.hello">Hello World!</h1>`.

Beide Informationen sind optional und werden mit einer Pipe voneinander getrennt. Ebenso optional ist die ID, die nach zwei @-Symbolen erscheinen kann. Fehlt diese ID, erzeugt Angular selbst eine. Das ist jedoch problematisch, denn wenn sich das Markup ändert, vergibt Angular einen neuen Wert dafür.

Nachdem alle Texte mit `i18n` markiert wurden, extrahiert der Befehl `ng xi18n` sie in eine XML-Datei. Diese ist pro Sprache zu kopieren und um entsprechende Übersetzungen zu ergänzen (Listing 3) [6].

Standardmäßig liegen diese Dateien im XLF-(XML-Localization-Interchange-File-)Format vor. Alternativ dazu lässt sich das CLI anweisen, stattdessen XLF2 oder XMB (XML Message Bundles) zu nutzen. Diese Formate werden häufig von Übersetzungsstudios unterstützt. Die übersetzten Dateien sind anschließend in der `angular.json` zu registrieren (Listing 4).

Mit ein paar weiteren Eigenschaften lässt sich auch das gewählte Format festlegen, sofern vom Standardformat XLF abgewichen wurde [7]. Außerdem ist hier das Locale, in dem die Templates vor der Übersetzung vorliegen, als `sourceLocale` einzutragen.

Die Anweisung `ng build -localize` erzeugt nun eine Version pro Locale (**Abb. 3**). Wie erwähnt, lässt sich diese Aufgabe verhältnismäßig schnell erledigen, weil das CLI im Gegensatz zu früher nur ein einziges Mal kompiliert und dann nur noch die Texte tauscht. Startet man nun einen Webserver im Verzeichnis `dist/projektname`, lässt sich eine der Sprachversionen durch Anhängen des jeweiligen Locales an den URL wählen.

Das neue `@angular/localize` erlaubt jedoch auch die Nutzung der Übersetzungstexte zur Laufzeit. Dazu sind sogenannte Tagged Template Strings zu nutzen: `title = $localize`:@@home.hello:Hello World!``. Als Tag kommt hier der globale Bezeichner `$localize` zum Einsatz. Außerdem wird der String mit der ID des jeweiligen Texts eingeleitet. Diese befindet sich zwischen zwei Doppelpunkten. Danach kommt der Standardwert. Derzeit extrahiert `ng xi18n` solche Texte zwar noch nicht automatisch, aber das hält uns nicht davon ab, bestehende Einträge auf diese Weise zu nutzen oder neue manuell in die XML-Dateien einzutragen.

Eine weitere sehr erwartete Möglichkeit ist das Festlegen der Übersetzungstexte zur Laufzeit. Um diese Runtime Translations zu unterstützen, sind sie lediglich in Form von Key/Value-Pairs festzulegen:

```
import { loadTranslations } from '@angular/localize';

loadTranslations({
  'home.hello': 'Küss die Hand!'
});
```

Listing 3

```
<trans-unit id="home.hello" datatype="html">
  <source>Hello World!</source>
  <target>Hallo Welt!</target>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.component.html</context>
    <context context-type="linenumber">1</context>
  </context-group>
  <note priority="1" from="description">description</note>
  <note priority="1" from="meaning">meaning</note>
</trans-unit>
```

Listing 4

```
"i18n": {
  "locales": {
    "de": "messages.de.xlf",
    "fr": "messages.fr.xlf"
  },
  "sourceLocale": "en-US"
},
```

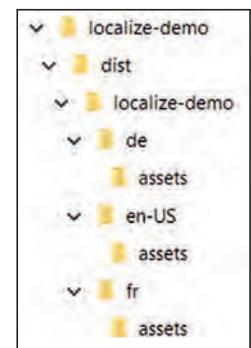


Abb. 3: Sprachversionen im dist-Ordner

Das muss vor dem Laden von Angular erfolgen, daher ist diese Codestrecke in ein eigenes Bundle auszulagern. Aus diesem Grund weist das beiliegende Beispiel diesen Aufruf in der *polyfills.ts* auf. Außerdem muss sich hier die Anwendung selbst ums Laden der richtigen Metadaten für die Formatierung von Zahlen und Datumswerten kümmern [8].

Any und platform

Die mit Version 6 eingeführten Treeshakable Providers machen die Arbeit mit Services um einiges einfacher. Version 9 setzt da noch einen drauf, indem es zwei weitere Werte für *providedIn* festlegt: *any* und *platform*:

```
@Injectable({ providedIn: 'any' })
export class LoggerConfig {
  loggerName = 'Default';
  enableDebug = true;
}
```

any bewirkt, dass jeder Scope auf Modulebene eine eigene Service-Instanz erhält. Das bedeutet, dass es für alle *lazy*-Module eine eigene Instanz gibt und eine weitere für alle restlichen Nicht-*lazy*-Module. Scopes auf Komponentenebene betrifft das nicht. So lässt sich die Notwendigkeit für *forRoot* und vor allem für *forChild*-Methoden reduzieren. Letztere haben den Vorteil, dass sie auch Konfigurationsparameter entgegennehmen können. Beim Einsatz von *any* muss die Anwendung diese anders festlegen, z. B. im Konstruktor des jeweiligen Modules (Listing 5).

Die ebenfalls ergänzte Einstellung *platform* registriert einen Service im Platform Injector. Er befindet sich eine Ebene über *root* und enthält Angular-interne Services.

Dev-Server für Server-side Rendering

Wenngleich Server-side Rendering (SSR) kein Thema ist, das jedes Projekt betrifft, ist es doch für das Angular-Team strategisch wichtig. Es erlaubt, mit Angular in den Bereich öffentlicher, SEO-kritischer Seiten vorzudringen. Bis jetzt war das Entwickeln solcher Lösungen lästig, zumal der Entwicklungswebserver nur die browserbasierte und nicht die serverbasierte Version des Projekts nach einer Änderung neu erstellte. Um auch die Auswirkung auf den serverseitigen Betrieb zu testen, musste man einen neuen Build erzeugen. Damit ist nun Schluss, für SSR steht ein Builder zur Verfügung, der beide Versionen neu erstellt und das Browserfenster aktualisiert. Wir sehen

Listing 5

```
@NgModule({ [...] })
export class FlightBookingModule {
  constructor(private loggerConfig: LoggerConfig) {
    loggerConfig.loggerName = 'FlightBooking';
    loggerConfig.enableDebug = false;
  }
}
```

somit sofort alle Auswirkungen unserer Änderungen. Dafür ist nach dem Hinzufügen des entsprechenden @nguniversal-Pakets das npm-Skript *dev:ssr* zu starten:

```
ng add @nguniversal/express-engine@next
npm run dev:ssr
```

Da hier zwei Builds parallel stattfinden müssen, ist die Performance nicht ganz so gut wie beim klassischen Development-Server. Nichtsdestotrotz verbessert es das Entwicklererlebnis erheblich gegenüber dem Status quo.

Fazit und Ausblick

Mit Angular 9 bekommen wir das lang ersehnte Ivy. Dank seines durchdachten Aufbaus macht es Angular selbst besser treeshakable und führt somit in vielen Fällen zu deutlich kleineren Bundles. Außerdem ermöglicht es Lazy Loading von Komponenten.

Um Ivy zu unterstützen, musste auch die integrierte i18n-Lösung überarbeitet werden. Das hat das Angular-Team zum Anlass genommen, die Implementierung zu verbessern: Der Build-Vorgang wurde drastisch beschleunigt, und wir können Übersetzungstexte programmatisch bereitstellen, aber auch konsumieren. Daneben erleichtert die Einstellung *any* für *providedIn* das Konfigurieren von Bibliotheken, indem es dafür sorgt, dass jedes *lazy*-Modul eine eigene Instanz erhält. Es kann eine Alternative zu *forRoot* und + darstellen und ist Mosaiksteinchen bei Bestrebungen, das Angular-Modulsystem optional zu gestalten.

Auch wenn sich das alles sehr aufregend anhört, wird es erst nach Angular 9 so richtig spannend. Denn Ivy bietet Potenzial für viele Neuerungen, denen sich das Angular-Team nun widmen kann.



Manfred Steyer ist Trainer und Berater mit Fokus auf Angular, Google Developer Expert und Trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java-Magazin und Heise Developer. Unter www.ANGULARArchitects.io bieten er und sein Team Angular-Schulungen und Beratung in Deutschland, Österreich und der Schweiz an.

www.softwarearchitekt.at, www.ANGULARArchitects.io

Links & Literatur

- [1] <https://github.com/angular/angular/blob/master/CHANGELOG.md>
- [2] <https://github.com/manfredsteyer/angular9-examples>
- [3] <https://update.angular.io>
- [4] <https://github.com/angular/angular>
- [5] Abbildung 2 ist mit freundlicher Genehmigung von Minko Gechev aus dem Angular-Team übernommen
- [6] <https://angular.io/cli/xi18n>
- [7] <https://angular.io/guide/i18n#merge-with-the-aot-compiler>
- [8] <https://www.softwarearchitekt.at/aktuelles/lazy-loading-locales-with-angular/>

Anzeige

Coden in Go

Für das Plus an Produktivität und Effizienz...

Die Programmiersprache Go wurde geschaffen, um ein Maximum an Effizienz und Produktivität zu erreichen. Programmierer, die bereits mit Java oder PHP vertraut sind, können in nur wenigen Wochen die wichtigsten Grundlagen von Go erlernen (viele werden es schlussendlich den „alten“ Sprachen vorziehen). In diesem Artikel untersucht Dewet Diener, VP of Engineering bei Curve, die Vor- und Nachteile von Go und, wie die testgetriebene Entwicklung (TDD) die Arbeit erleichtert.

von Dewet Diener

Go wurde von Google entwickelt und erschien 2009 als umfassende Programmiersprache, die zur Maximierung der Produktivität geschaffen wurde. Die Sprache wurde zudem mit der Absicht designt, Mängel in etablierten Sprachen nicht zu wiederholen. Obwohl Go eine relativ junge Sprache ist, hat sie bereits eine große Fangemeinde von Entwicklern angezogen, und deshalb möchte ich ein wenig darüber sprechen, warum wir bei Curve Go lieben und wie wir die Sprache einsetzen, um unser Ziel zu erreichen, das Bankwesen in die Cloud zu verlagern.

Go ist so etwas wie eine verfeinerte Programmiersprache: Sie funktioniert nach dem WYSIWYG-Prinzip (What you see is what you get), also klar lesbarer Code und weniger komplexe Abstraktionen. Die Sprache selbst ist einfach zu nutzen und leicht zu erlernen. Dennoch kann es, gerade weil das Ökosystem wie die Sprache noch relativ neu ist, schwierig sein, Entwickler mit umfangreichen Go-Kenntnissen zu finden.

Nutzereffizienz im Fokus

Im Gegensatz zu anderen Programmiersprachen wurde Go allerdings vor allem im Hinblick auf Nutzereffizienz entwickelt. Daher können Nutzer mit Java- oder PHP-Hintergrund innerhalb weniger Wochen in der Anwendung von Go geschult werden – und unserer Erfahrung nach bevorzugen viele von ihnen anschließend Go gegenüber den älteren Artgenossen.

Wir bei Curve setzen uns stark für die testgetriebene Entwicklung (TDD) ein, und das Framework von Go ist gut auf diese Methodik eingestellt. Durch die einfache

Benennung einer Datei in *foo_test.go* und das Hinzufügen strukturierter Testfunktionen innerhalb dieser Datei wird Go Ihre Unit-Tests schnell und effizient ausführen. Diese innovative Funktion steigert die Produktivität, da sie eine stärkere Konzentration auf die testgetriebene Entwicklung und verbesserte Peer-Review-Möglichkeiten erlaubt.

Auch im Hinblick auf die Optimierung der Produktion verfügt Go über viele wichtige Eigenschaften, etwa einen relativ geringen Speicher-Footprint, was den Einsatz in besonders großen Projekten erleichtert. Überdies bietet Go von Haus aus eine einfache Cross-Kompilierung zu anderen Architekturen. Da der Go-Code in ein einzelnes statisches Binary kompiliert wird, ist die Containerisierung ein Kinderspiel und macht es durch eine entsprechende Erweiterung fast trivial, Go in jeder hochverfügbar Umgebung (wie Kubernetes) einzusetzen.

Go stellt einen Mechanismus für das Absichern von Workloads bereit und hat sehr leichtgewichtige Produktionscontainer ohne Abhängigkeiten nach außen. Das macht das Erstellen, Deployen und Warten einer Go-basierten Anwendung sehr einfach und sicher und stellt einen guten Grund für Unternehmen dar, sich für Go zu entscheiden. Das gilt insbesondere für Firmen, die auf Microservices setzen wollen.

Ein modernes Ökosystem

Die technologischen Ökosysteme moderner Softwareentwicklung wachsen rapide und Go wurde mit diesem Gedanken im Hinterkopf entwickelt. Die Programmiersprache bietet beispielsweise als Teil ihrer Standardbibliothek alles, was man zur Erstellung von

APIs benötigt. Die Funktionalität ist einfach zu benutzen, und der leistungsfähige HTTP-Server beseitigt einige der üblichen Probleme, die oft auftreten, wenn Teams ein neues Projekt entwerfen – etwas, das bei einigen anderen populären Sprachen wie Java und Node oft zu Unannehmlichkeiten führt.

Go löst auch Formatierungsschwierigkeiten auf eine sehr geschmeidige Art und Weise, nämlich durch die in die Sprache selbst eingebaute automatische Formatierung. Konflikte werden so vollständig eliminiert, was wiederum die Produktivität und den Fokus der Teams steigert.

So sehr ich auch ein Verfechter von Go bin, so ist die Sprache natürlich nicht ohne Mängel. Ein strittiger Punkt ist, dass Go keine explizite Schnittstelle hat, ein Konzept, an das viele Entwickler gewöhnt sind. Obwohl das kein großer Nachteil ist, kann es lästig sein, die für die eigene Struktur am besten geeignete Schnittstelle zu wählen. Das liegt daran, dass man nicht einfach schreibt „X implementiert Y“, wie man es vielleicht von anderen Sprachen kennt. Allerdings ist das etwas, was man nach kurzer Zeit kaum noch wahrnimmt.

Dependency Management

Dependency Management ist ebenfalls etwas, das offenbar nicht Teil des ursprünglichen Entwurfs von Googles Go-Team war. Die Open-Source-Community griff aber schnell ein und schuf glide und dep, die das Problem allerdings nicht vollständig lösten. Mit Go 1.11 wurde die Unterstützung für Module eingeführt, was aktuell das offizielle Tool für das Dependency Management darstellt.

Diese Herausforderungen schmälern nicht das Potenzial von Go als effizienter Programmiersprache, die signifikante Vorteile gegenüber anderen Sprachen beinhaltet. Go zieht die Aufmerksamkeit von eifrigeren Entwicklern auf der ganzen Welt auf sich und das Projekt liegt stark im Fokus des allgemeinen Interesses. Und auch die Open-Source-Gemeinschaft rund um Go blüht, dank interessanter Projekte wie Docker und Kubernetes, immer weiter auf.

Die frischen und einfallsreichen, aber auch die einfachen Ideen sind es, was Go für uns ausmacht: Die Sprache ist aufregend und hilft uns bei Curve, schnell voranzukommen und bessere Produkte zu erstellen.

Anzeige



Dewet Diener ist derzeit VP of Engineering bei Curve. Dewet verfügt über mehr als zwanzig Jahre Erfahrung in der Branche, hauptsächlich in den Bereichen Software und Site Reliability Engineering, die sich über eine Reihe von verschiedenen Sektoren und eine Reihe von eklektischen Unternehmen erstreckt. Er verbrachte mehr als sechs Jahre bei Google mit der Entwicklung von Partner Solutions, half bei der Gründung von Klarismo, einem Start-up-Unternehmen für automatisierte MRI-Bildanalyse, und vor kurzem bei BenevolentAI, wo er als VP of Engineering tätig war.



© Makstom/Shutterstock.com

Container, Docker & Kubernetes: mehr Sicherheit mit SysCall-Filtern

Sicherheit geht vor – auch bei Containern

Container und Plattformen wie Kubernetes sind mittlerweile ein beliebtes Angriffsziel von Hackern. In diesem Artikel zeigt Andreas Jaekel, Head of PaaS Development bei IONOS, wie man seine Container absichern kann, denn von Natur aus sicher sind sie nicht.

von Andreas Jaekel

Kaum ein Entwickler kommt ohne Container und Kubernetes aus, die Containertechnologie erleichtert das Arbeiten mit Microservices und in agilen Teams ungemein. Kubernetes hat in den vergangenen fünf Jahren eine Erfolgsgeschichte hingelegt und ist als Standardtool für die Containerorchestrierung etabliert. Doch beliebte Technologien erhalten auch mehr Aufmerksamkeit von Hackern, die Malware verbreiten wollen. Das betrifft auch Container. Sie sind nicht per se vertrauenswürdig – selbst solche, die wir selbst erstellt haben, können gehackt werden. Malware kann in Images enthalten sein oder in die Container heruntergeladen werden. Zwar sollten generische Container die Eindringlinge in Schach halten, doch können wir das Sicherheitsniveau weiter erhöhen. Ein Ansatz ist, eine bekannte Linux-Funktion zu instrumentalisieren und Containern vorzugeben, welche System Calls (SysCalls) sie ausführen dürfen.

SysCalls – was versteckt sich dahinter?

Jedem Linux-Prozess wird beim Starten ein kleines Stück Arbeitsspeicher zur Verfügung gestellt. Der Code kann dann völlig frei auf diesem Speicherabschnitt arbeiten, also beispielsweise Rechenoperationen durchführen. Für alles andere muss er den Kernel um Erlaubnis bitten. Hier eine kleine Auswahl:

- Schreib- und Leseberechtigungen (*write/read*)
- Erstellung eines Verzeichnisses (*mkdir*)
- Start eines neuen Prozesses (*fork*)
- Abrufen der Tageszeit (*gettimeofday*)

Der Code erhält die Erlaubnis, indem er die Anfrage per SysCall an den Kernel stellt, der daraufhin die Zugriffsrechte prüft, bevor er der Anfrage nachkommt.

Es gibt ungefähr 330 SysCalls in Linux, die unter [1] zu finden sind – sie sind übrigens unabhängig von der Programmiersprache immer gleich. *write* bleibt *write*, egal ob in C oder GoLang.

Warum ist eine Filterung von SysCalls sinnvoll?

Man kann grob drei Einfallstore unterscheiden, über die Container attackiert werden:

- Backdoors in Docker Upstream Images
- Bugs in Anwendungen
- Schwachstellen in SysCalls im Linux-Kernel

SysCalls sind genau das richtige Werkzeug, um Programme und Container davon abzuhalten, etwas zu tun, was sie nicht sollten. Ein Beispiel: Nutzen Sie NGINX, können Sie gleich alle SysCalls filtern, bei denen Sie sicher sind, dass NGINX sie nicht benötigt. In Abbildung 1 ist ein kleiner Ausschnitt davon zu sehen.

Damit machen Sie die Nutzung von NGINX sicherer und schränken es nicht in seiner Funktionsfähigkeit ein.

Welche Filter sollten wir setzen?

Das vorige Beispiel sieht einfach aus. Doch wie findet man nun heraus, welche SysCalls eine Anwendung überhaupt nutzt? Filtert man zu viele davon, funktionieren die Anwendungen nicht mehr. Bei zu wenigen Filtern bleibt Angreifern zu viel Raum. Vorweg: Es ist schwierig oder fast unmöglich, eine perfekte Filterliste zu erstellen. Es gibt jedoch fünf unterschiedliche Vorgehensweisen, sich einer geeigneten Filterliste zumindest anzunähern:

- Quelltext lesen, und zwar alles – inklusive aller Libraries:** Das ist im Grunde die einzige Möglichkeit, wirklich sicherzugehen, schadhaften Code auszuschließen. Aufgrund der schier unendlichen Abhängigkeiten ist dieser Weg in der Praxis jedoch kaum gangbar.
- Trial and Error:** Natürlich können Sie auch einfach drauflosprobieren und Filter setzen. Bei rund 330 SysCalls und Unmengen an verschiedenen Kombinationen ist auch dieses Vorgehen praktisch nicht umsetzbar.
- Educated Guessing:** Eine bessere Variante ist es da schon, gezielt zu raten. Das ergibt natürlich nur dann Sinn, wenn das Wissen rund um Softwaredesign und SysCalls gut ausgeprägt ist. Selbst dann ist es wahrscheinlich, dass entweder zu viele oder zu wenige Filter gesetzt werden. Das Educated Guessing ist besser als gar nichts, von einem gut funktionierenden Filter jedoch weit entfernt.
- Analyse der Binary:** Auch aus den Binaries könnte man theoretisch herauslesen, welche SysCalls genutzt werden. Der Vorteil liegt hier darin, dass Binaries immer in Maschinensprache vorliegen. Jedoch erzeugen unterschiedliche Sprachen und Compiler auch unterschiedlichen Maschinencode. So lässt sich automatisiert kaum feststellen, welche SysCalls darin enthalten sind und welche nicht.
- Call Tracing mit strace:** SysCalls, die eine Anwendung ausführt, können mit einem Tracing-Tool nachverfolgt werden. Das lässt sich zum Beispiel während eines Testlaufs oder in einer CI Pipeline durchführen.

• kill	• reboot	• rename	• sethostname
• ptrace	• mkdir	• rmdir	• setpriority
• swapoff	• link	• umount	• init_module
• truncate	• creat	• chroot	• delete_module
• setxattr	• mount	• symlink	• quotactl
• capset	• setuid	• swapon	• etc.

Abb. 1: Gefilterte SysCalls

Linux bietet hierfür das Tool *strace* – ein sehr gutes Werkzeug zum Debuggen und zur Fehlerbehebung. Zusätzlich zeigt *strace* Call-Parameter an. In Listing 1 ist ein Beispiel im Counter Mode zu sehen:

Zusammenfassend ist festzuhalten:

- Es ist schwierig, eine gute Filterliste zu erstellen.
- Zu enge Filter hindern die Anwendung daran, fehlerfrei zu funktionieren.
- Zu lasche Filter öffnen Malware Tür und Tor.
- Ich empfehle, zunächst mit *strace* alle nötigen SysCalls nachzuverfolgen.
- Im zweiten Schritt ist es sinnvoll, den Filter über Educated Guessing zu verfeinern.

In der Praxis: Erstellung eines eBPF mit Seccomp

Linux kann vor jedem SysCall kleine State Machines ausführen. Diese Programme müssen als sogenannte „extended Berkeley Packet Filter“, kurz eBPF, ausgeliefert werden. Sie können per *SysCall bpf()* in den Kernel geladen werden.

eBPF können für verschiedene Dinge verwendet werden, z. B. zur Messung der Performance, zum Debuggen oder Tracing. Jedoch ist es sehr komplex, ein eBPF zu erstellen. Und: Sie wollen ja nur SysCalls filtern und keine

Listing 1

```
> strace -c -S name ./helloworld

Hello World!

% time    seconds   usecs/call  calls  errors syscall
-----
0.00  0.000000      0       1  arch_prctl
0.00  0.000000      0       4      brk
0.00  0.000000      0       1  execve
0.00  0.000000      0       1  uname
0.00  0.000000      0       1  write
-----
100.00  0.000000          8      total
```



neue Programmiersprache lernen. Dabei hilft uns Seccomp BPF. Es verbirgt die Komplexität von eBPF und kann individuelle SysCalls filtern und außerdem

- vorgeben, dass ein SysCall ausgeführt wurde, obwohl das nicht der Fall war,
- gefälschte Ergebnisse und Fehlernummern zurücksenden
- Breakpoints erzeugen.

Listing 2

```
int
main(int argc, char *argv[])
{
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(getpid), 0);
    seccomp_load(ctx);
    pid_t pid = getpid();
    /* never reached: process killed */
    return 0;
}
```

Listing 3

```
unsigned char buf[BUF_SIZE];
int fd = open("data.raw", 0);
int rc = seccomp_rule_add(
    ctx,
    SCMP_ACT_ALLOW,
    SCMP_SYS(read), 3,
    SCMP_A0(SCMP_CMP_EQ, fd),
    SCMP_A1(SCMP_CMP_EQ, (scmp_datum_t)buf),
    SCMP_A2(SCMP_CMP_LE, BUF_SIZE));
```

Listing 4

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "syscalls": [
        {
            "names": [
                "accept",
                "access",
                ...
            ],
            "action": "SCMP_ACT_ALLOW",
            "args": [],
            "comment": "",
            "includes": {},
            "excludes": {}
        }
    ]
}
```

Seccomp ist also ein gutes Tool, um zu testen, um Fehler zu injizieren oder zum Debugging. In Listing 2 ist ein Beispiel zu sehen, wie Seccomp in C genutzt wird. Es kann, wie in Listing 3 zu sehen, auf Basis bestimmter Parameter filtern. Dadurch filtert es *read()*-Aufrufe, die nicht alle der drei folgenden Bedingungen erfüllen:

- Die Daten werden aus genau dem File Descriptor gelesen, der zuvor mit *open()* erstellt wurde.
- Die gelesenen Daten werden im dafür vorgesehenen Speicherbereich *buf* abgelegt.
- Es werden nicht mehr Daten gelesen als in *buf* hineinpassen.

Nach Parametern zu filtern, kann in vielen Fällen sinnvoll sein, zum Beispiel um

- read-only SysCalls zu erzwingen,
- die *reads* und *writes* auf Standardeingabe und -ausgabe zu beschränken,
- *setuid()* auf eine bestimmte User-ID zu beschränken,
- ausschließlich SIGHUP-Signale zu erlauben und
- zu verhindern, dass zu großzügige Dateiberechtigungen gesetzt werden.

Das Filtern auf Basis von Parametern hat jedoch auch seine Grenzen. So können nur Wertparameter (Pass by value) ausgewertet werden. Sie können daher keinen Blick in Zeichenfolgen oder Strukturen werfen. So kann beispielsweise *open()* nicht auf bestimmte Verzeichnisnamen beschränkt werden.

Anwendung auf Container und K8s

Die gute Nachricht ist, dass Seccomp in Docker v1.10 hinzugefügt wurde. Standardmäßig werden bereits

Listing 5

```
{
    "names": [
        "Ptrace"
    ],
    "action": "SCMP_ACT_ALLOW",
    "args": null,
    "comment": "",
    "includes": {
        "minKernel": "4.8"
    },
    "excludes": {}
}
```

Listing 6

```
[...]
metadata:
  labels:
    app: problemsolver
  annotations:
    kubernetes.io/psp: allowseccomp
    seccomp.security.alpha.kubernetes.io/pod: localhost/custom-profile.json
[...]
```

44 SysCalls geblockt, inklusive `reboot()`. Diese unerwünschten SysCalls werden nicht ausgeführt, das Programm wird jedoch nicht beendet. Für die Erstellung eines benutzerdefinierten Filters ist es zu empfehlen, mit den Standardeinstellungen zu beginnen und sie wie gewünscht anzupassen. Benutzerdefinierte Filter werden als JSON-Dateien ausgegeben. Wie das in Docker aussieht, ist in Listing 4 zu sehen. Die in Listing 4 aufgelisteten System Calls werden unabhängig von ihren Parametern erlaubt. In Listing 5 wird `ptrace()` erlaubt, doch nur auf Kernels, die mindestens Linux-4.8 entsprechen. Um das JSON-Filter-Set in Docker zu laden (Docker nennt das übrigens seccomp profile) hilft:

```
# docker run -ti --rm --security-opt seccomp:custom_filter.json alpine /bin/sh
```

Dabei sollte man beachten, dass ein benutzerdefinierter Filter den Standardfilter ersetzt und nicht etwa ergänzt. Zudem gilt der Filter für den gesamten Container.

SysCall Filter in Kubernetes

Seit Version 1.3 sind SysCall-Filter über Seccomp auch in Kubernetes möglich und werden von den meisten Laufzeitumgebungen unterstützt, nicht nur von Docker. Seccomp-Profile beziehen sich immer auf den gesamten Pod, nicht nur auf einzelne Container. Um benutzerdefinierte Profile über Seccomp anzulegen, müssen Sie die Pod-Sicherheitseinstellungen (Pod Security Policies) im K8s-Cluster aktivieren und danach eine Pod-Sicherheits-einstellung definieren, die es erlaubt, Seccomp-Profile zu nutzen. Über die Erstellung einer Rollenbindung (Role Binding) erlauben Sie dies auch den Pods. Um die Sicherheitseinstellungen zu aktivieren, fügen Sie zumindest eine permissive Policy hinzu und erstellen außerdem mindestens eine passende Rolle und eine Rollenbindung für den Namespace `kube-system`. Andernfalls kann K8s keine Pods mehr starten. Fügen Sie dann die `PodSecurityPolicy` zur Liste der aktivierten Admission Controller hinzu:

```
kube-apiserver \
--enable-admission-plugins= \
PodSecurityPolicy,LimitRanger ...
```

Im nächsten Schritt können Sie Seccomp-Profile bereitstellen, indem Sie diese erstellen und den Worker Nodes in `kubelet --seccomp-profile-root=` verfügbar machen (Default: `/var/lib/kubelet/seccomp`). Um nun einen Seccomp-Filter auf einen Pod anzuwenden, müssen Sie das in Listing 6 Gezeigte im Pod ergänzen. Zum Starten können Sie unter [2] ein Beispiel herunterladen.

Zusammenfassung

Einen geeigneten Filter zu erstellen, ist nicht einfach. Ist er zu großzügig, dient er nicht der Abwehr von Malware. Bei einem zu eng gestrickten Filter kann es vorkommen, dass die Applikation gar nicht erst ausführbar ist. Hinzu kommt, dass die Docker-Standardeinstellungen bereits recht ausgereift sind.

Dennoch kann es in bestimmten Szenarien Sinn machen, die Zeit in die Erstellung eines eigenen Filters zu investieren. Insbesondere, wenn Sie Ihre Anwendung gut kennen, ist das ein Quick Win. Auch wenn Sie eine sehr sichere Docker-Umgebung benötigen (FinTech u. a.), lohnt sich der Aufwand. Gerade als Container Hoster ist es wertvoll, selbst zu definieren, welche SysCalls ausgeführt werden dürfen und welche nicht.



Andreas Jaekel ist als Leiter der PaaS-Entwicklung zuständig für die Entwicklung des Managed Kubernetes Angebots von IONOS. Seit er 1995 mit Linux- und Unix-Systemen arbeitet, hat er Erfahrungen als System- und Anwendungsentwickler, DevOps-Ingenieur, Systemmanager und auch als Endanwender gesammelt. Nach ausgedehnten Abenteuern in den Bereichen Webhosting, E-Mail-Verarbeitung und Cloud-Storage gehören zu seinen jüngsten Projekten die Lösungen Managed OpenStack und Kubernetes.

Links & Literatur

[1] <https://syscalls.kernelgrok.com/>

[2] <https://github.com/ionos-enterprise/K8s-seccomp-demo>

DevOpsCon

8. – 11. JUNI 2020 | BERLIN

EXPO: 9. – 10. JUNI 2020



Die Konferenz für Continuous Delivery, Microservices, Containers, Cloud & Lean Business

- 4 Tage mit mehr als 80 Workshops, Sessions und Keynotes
- 60+ Speaker und Trainer führender IT-Unternehmen
- 9+ Power Workshops mit Livecoding und Anwendung neuer Ideen

www.devopscon.io/berlin/



Kubernetes Workshop

Erkan Yanar (linsenraum.de)

Kubernetes ist der Industriestandard für Containerorchestrierung. Doch es ist weit mehr als nur ein weiteres Tool: Es ist die eierlegende Wollmilchsau der Containerinfrastrukturen. Dieser Workshop soll zeigen, warum Kubernetes als zukünftige und alleinige Infrastruktur zum Container-Deployment gedacht werden sollte. Der Zugriff auf ein Kubernetes-Cluster ist vollkommen ausreichend, um die Container zu betreiben. Der Entwickler bekommt nicht nur eine Umgebung, um seine Container hochverfügbar auszurollen, und ein Lifecycle-Management – richtig eingerichtet bringt Kubernetes viel mehr.

Device Shadows/Device Twins

Amazon IoT für Java

Device Shadows bzw. Device Twins sind ein interessanter Weg, um IoT-Verbünde auch beim Verlust von Teilen der Internetverbindung am Leben zu erhalten. Wie das funktioniert, klärt der folgende Artikel.

von Tam Hanna

Bevor wir in medias res gehen, wollen wir noch eine Runde Unterschiedsbingo spielen. In der heutigen Welt hört man nämlich immer wieder vom Thema des Digital Twin. Ein Digital Twin ist im Grunde genommen ein neuartiger Begriff für eine umfangreiche Softwaresimulation eines realen Systems, die einem Consulting- oder Beratungsunternehmen große Umsätze bringen soll.

Unsere hier verwendeten Device Shadows arbeiten auf einem wesentlich „niedrigeren“ Level. Sie fungieren als eine Art Cache zwischen Gerät und Logik, um bei kurzfristigen Verbindungsaußfällen das Weiterfunktionieren des Systems zu garantieren.

Wie funktioniert ein Device Shadow?

Wie so oft in der Welt der Cloud-getriebenen Informatik gilt auch hier, dass 5 000 Wege nach Rom führen. Im Hause Amazon setzt man zur Realisierung der als Device Shadow bezeichneten Funktion auf eine Gruppe von MQTT-Kanälen, über die sowohl Server als auch Client Änderungswünsche miteinander austauschen.

Hinter dem unter [1] im Detail beschriebenen Verfahren steckt der Gedanke, dass das Serversystem beim Nichtbestehen einer Verbindung zum Endgerät mit den im Device Shadow vorliegenden Informationen arbeitet. Kommt das Gerät später wieder online, prüft es Änderungen am Server und aktualisiert den Shadow mit seinen aktuellen Messdaten. Zudem kann das Gerät auch nach Zustandsinformationen befragen, die es auf seine Umgebung anwendet.

Listing 1

```
public static void main(String[] args) {  
    String clientEndpoint = "a3gw153wutvel-ats.iot.eu-west-1.amazonaws.com";  
    String clientId = "TamsGeraetInEclipse";  
  
    AWSIoTMqttClient client = new AWSIoTMqttClient(clientEndpoint,  
                                                    clientId, "AaaaW", "iaaa5", null);  
  
    try {  
        client.connect();  
        ...  
    } catch (Exception e) {  
        System.out.println("Caught an exception: " + e.getMessage());  
        return;  
    }  
}
```

Ein Klassiker sind alle Arten von langsam verändernden Zuständen wie beispielsweise eine Solltemperatur für einen Raum – dass Sie einen Device Shadow nicht zum Steuern einer latenzkritischen Aufgabe wie beispielsweise einer Turbinendrehzahl einsetzen sollten, nimmt der Autor als logisch an und führt es hier primär zur Vermeidung rechtlicher oder tödlicher Probleme durch Dummheit an.

Beschwörung eines Schattens

Im Artikel „AWS IoT Device SDK for Java“ [2] in der letzten Ausgabe haben wir festgestellt, dass das SDK vergleichsweise umfangreich ist. Als Lösung nutzten wir den Apache-Maven-Paketmanager, der die notwendigen .jar- und sonstigen Dateien für uns automatisch bereitstellte. Öffnen Sie die Eclipse IDE, und kehren Sie in das im letzten Artikel erzeugte Projektskelett zurück. Ersetzen Sie den im Einsprungpunkt befindlichen Code danach durch die in Listing 1 gezeigten Passagen.

Auch diesmal müssen wir damit beginnen, eine Instanz der Klasse `AWSIoTDevice` zu erzeugen und unter Verwendung der beiden im letzten Heft generierten IDs am AWS-Server auszuweisen. Die beiden Schlüsselstrings sollten sich nicht geändert haben, wenn Sie im AWS Backend zwischenzeitlich keine Änderungen vorgenommen haben.

Überzeugen Sie sich im ersten Schritt vom korrekten Funktionieren unseres Programms, um danach die folgenden Änderungen durchzuführen:

```
try {  
    ...  
    client.connect();  
  
    String state = "{\"state\":{\"reported\":{\"sensor\":3.0}}};  
    device.update(state);  
}
```

Die Klasse `AWSIoTDevice` dient als Dreh- und Angriffspunkt zwischen einzelnen im AWS IoT Backend angelegten Geräteinstanzen. Der übergebene String `TamsSensorAktorDevice` muss dabei nicht unbedingt sprechend sein – wichtig ist aus Sicht von Amazon nur, dass er ein weltweit einzigartiges Gerät beschreibt. Wer beispielsweise mit einem ESP32 arbeitet oder sein Gerät mit einem Sensorchip ausstattet, kann die im Sensor

Abb. 1: OneWire-Peripheriegeräte bringen im Allgemeinen eine Seriennummer mit

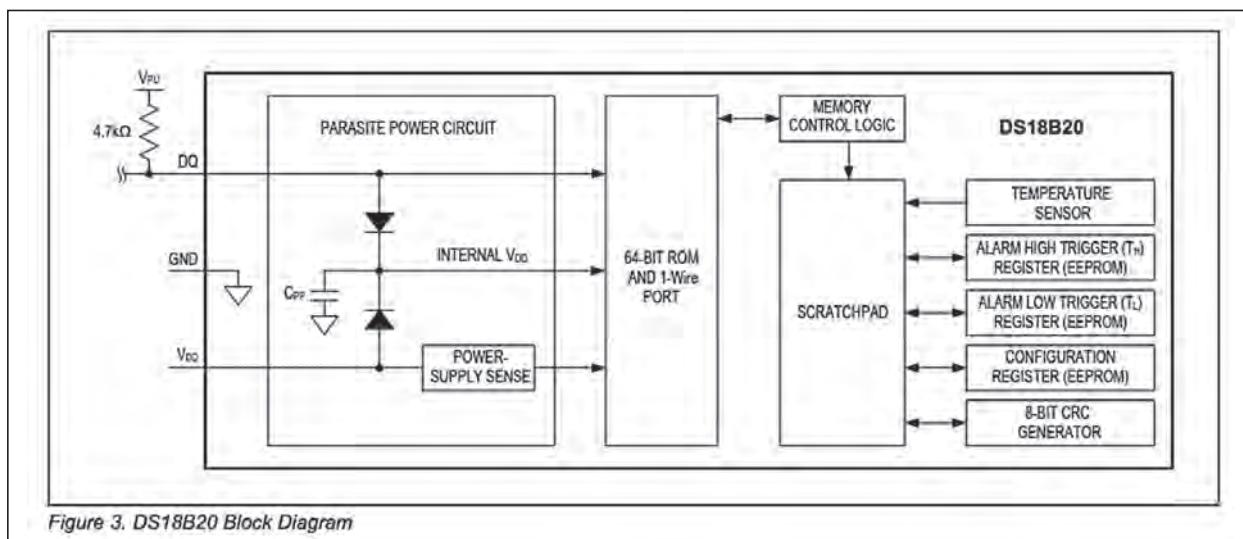


Figure 3. DS18B20 Block Diagram

enthaltenen Seriennummerndaten auch als ID für das Gesamtgerät verwenden. Eine dankbare Quelle sind dafür alle per OneWire verbundenen Sensoren – Abbildung 1 zeigt den relevanten Ausschnitt aus dem Datenblatt des weitverbreiteten DS18B20-Temperatursensors.

Wer das Programm im vorliegenden Zustand ausführt, stellt fest, dass es durchläuft. Im Backend bemerkt man allerdings nichts vom neu angelegten Geräteprofil.

Zur Umgehung dieses Problems müssen wir unseren Code ein wenig weiter anpassen, um Informationen in den Device Shadow zu schreiben:

```
try {
    ...
    client.connect();

    String state = "{\"state\":{\"reported\":{\"sensor\":3.0}}}";
    device.update(state);
```

Die Amazon-Dokumentation ist an dieser Stelle übrigens veraltet – die Updatefunktion erwartet nicht mehr ein *Date*-Objekt, sondern stattdessen einen JSON-String. Diese Vorgehensweise ist vernünftig, da es in Java – je nach Konfiguration Ihrer Arbeitsumgebung – schon eine oder sogar mehrere Stackklassen gibt.

Dieser an sich vernünftig wirkende Code lässt sich in der Cloud ebenfalls problemlos ausführen. Leider sehen wir auch hier im Backend kein Ergebnis. Zur Überprü-

fung des korrekten Verhaltens ändern wir den in *state* angelieferten Sensorwert, und schreiben ihn zudem vor der Anpassung des Werts nach außen:

```
System.out.println(device.get());
String state = "{\"state\":{\"reported\":{\"sensor\":9.0}}}";
device.update(state);
```

Wer die neue Version des Programms zum ersten Mal ausführt, bekommt den alten Sensorwert ausgegeben – es ist erwiesen, dass der Device Shadow die per Update-methode eingelieferten Werte zwischen verschiedenen Aufrufen unseres Beispielprogramms persistiert.

Misstrauische Naturen „adjustieren“ an dieser Stelle den Wert, der an AWS IoT Device übergeben wird. Im Rahmen des nächsten Starts scheitert das Programm mit der in Abbildung 2 gezeigten Fehlermeldung – findet das AWS Backend keinen zu einem Gerät passenden Shadow, so scheitert der Aufruf der Methode.

Schatten lesen

Nachdem erwiesen ist, dass die per Deviceklasse in Richtung der Cloud geschriebenen Informationen für das Programm selbst lesbar sind, wollen wir einen primitiven Client-Server-Flow realisieren. Wir hatten im oben genannten Artikel festgestellt, dass jede Verbindung zwischen einer Applikation und dem AWS Backend eine

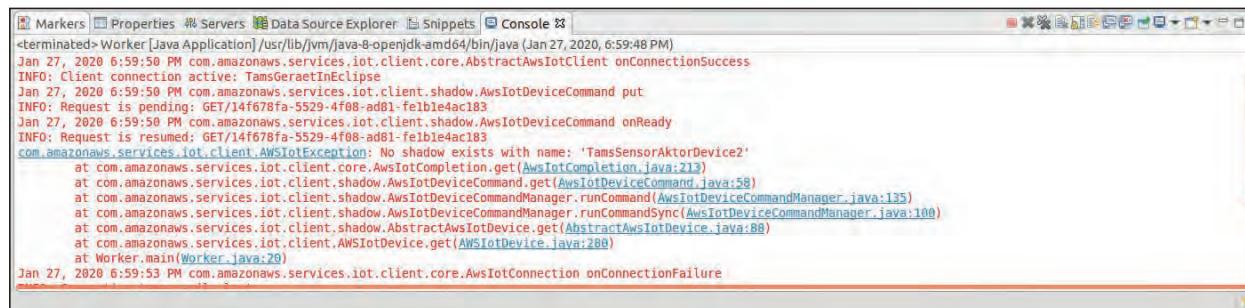


Abb. 2: Das „direkte“ Laden des Device Shadow führt zu Exceptions

weltweit einzigartige Client-ID aufweisen muss. Der bequemste Weg zur Befriedigung dieser Bedingung besteht darin, in Eclipse ein weiteres Maven-Projekt anzulegen, das AWS SDK abermals zu importieren und eine weitere Instanz von AWS-IoT-MQTT-Client zu erzeugen. Die Authentifizierungsstrings dürfen Sie wie bisher gewohnt eins zu eins übernehmen. Wichtig ist nur, dass der Wert der Client-ID nun anders lautet. Ein kleiner Test des Autors trug beispielsweise folgenden Namen:

```
public static void main(String[] args) {
    String clientEndpoint = "...amazonaws.com";
    String clientId = "TamsLeserInEclipse";

    AWSIoTMqttClient client = new AWSIoTMqttClient(clientEndpoint, clientId,
                                                    "AkaaaaaIW", "i2JRaaaal5", null);
```

Die zweite Version des Programms unterscheidet sich von der bisher verwendeten Applikation nur dadurch, dass wir in Client-ID nun einen anderen Namen übergeben. Zur Laufzeit bedeutet das, dass die beiden Programme zumindest in der Theorie gleichzeitig mit dem AWS Backend verbunden sein dürfen.

Für einen ersten Test reicht es dann aus, die folgende Payload einzufügen:

```
try {
    AWSIoTDevice device = new AWSIoTDevice("TamsSensorAktorDevice");
    client.attach(device);
    client.connect();
    System.out.println(device.get());
```

AWSIoTDevice bekommt den weiter oben erstmals verwendeten String nochmals übergeben, um eine Verbindung zur selben Geräteinstanz in der Cloud herzustellen. An dieser Stelle können Sie das Leseprogramm auch schon ausführen – es wird in der Konsole den weiter oben in den Device Shadow geschriebenen String ausgeben:

```
{"state":{"reported":{"sensor":9.0}},
```

Im Interesse eines besseren Verständnisses des Device Shadow API möchte ich an dieser Stelle einen kurzen Blick auf den vom Server zurückgelieferten String werfen. Er beginnt mit der Sequenz `{"state":{`, die im all-

Listing 2

```
import com.amazonaws.services.iot.client.AWSIoTDeviceProperty;

public class ShadowWorker extends AWSIoTDevice {
    ...
    @AWSIoTDeviceProperty
    private String myStringVal;
    @AWSIoTDeviceProperty
    private float myFloatVal;
}
```

gemeinen den weiter oben übergebenen String mit dem Temperaturattribut aufweist.

Weiter unten findet sich dann allerdings noch der folgende zweite Teil:

```
"metadata":{"reported":{"sensor":{"timestamp":1580147822}}}, "version":3,
"timestamp":1580417922,"clientToken":
"481d1504-b23d-4a73-845b-552dd66580a8"}
```

Amazon realisiert als Teil von AWS für uns eine Art Versionierungssystem, das sich um das Hinzufügen diverser Metadateninformationen kümmert. Neben der Möglichkeit zum Festlegen von *reported*-Attributen, die von der Gerätehardware ausgezeichnet und für die Verwendung in der Cloud vorgesehen sind, gibt es auch noch die Verwendung von *desired*-Informationen. Diese gehen – logischerweise – den entgegengesetzten Weg. Wir werden ihre Verwendung in den folgenden Schritten näher ansehen.

Für Sie als Entwickler ist hier vor allem wichtig, dass sich der Inhalt des Strings jederzeit ansehen lässt – wenn sich Ihr Programm seltsam verhält, ist dies eine vernünftige Option zur Problembehebung.

Bequemere Schattenverarbeitung

Unser Device Shadow mag an dieser Stelle als Speicher funktionieren. Wirklich bequem ist das Handling nicht; wenn wir mehr als nur den einen Wert ablegen würden, müssten wir uns beispielsweise mit einer JSON-Deserialisierungsbibliothek rumärgern. Erfreulicherweise bietet das AWS IoT Device SDK for Java einen bequemeren Weg an, der diese auf den ersten Blick lästige Aufgabe wesentlich vereinfacht. Hierzu müssen wir im ersten Schritt in das Schreibprogramm zurückkehren, in dem wir eine von *AWSIoTDevice* abgelegte Klasse anlegen. Die absolute Basisvariante, die sich auf die Erfüllung der Anforderungen des Compilers beschränkt, sieht folgendermaßen aus:

```
public class ShadowWorker extends AWSIoTDevice {
    public ShadowWorker(String thingName) {
        super(thingName);
    }
}
```

ShadowWorker unterscheidet sich insofern von der gewöhnlichen Programmierung, als wir nun die abgeleitete Klasse verwenden. Zur Korrelation zwischen dem Objekt und der Repräsentation in der Cloud ist natürlich abermals ein Gerätestring erforderlich, der das System bzw. das Endgerät einzigartig identifiziert.

Das eigentliche Anliegen der in der Cloud zu speichernden Elemente erfolgt dann über das Attribut `@AWSIoTDeviceProperty`. Für einen ersten kleinen Versuch wollen wir unserem soeben angelegten *ShadowWorker* nach dem in Listing 2 gezeigten Schema zwei Attribute hinzufügen.

Für einen ersten Test müssen wir den Zugriffscode dann von der bisher verwendeten Klasse auch auf eine Instanz des Workers umstellen:

```
try {
    ShadowWorker device = new ShadowWorker("TamsSensorAktorDevice");
    client.attach(device);
    client.connect();
```

An sich ist das Programm an dieser Stelle zur Ausführung bereit – die Kompilation läuft problemlos durch, und die JVM beginnt auch mit dem Programmstart. Während der eigentlichen Ausführung sehen wir dann allerdings Fehler, die nach der in Listing 3 gezeigten Bauart aufgebaut sind.

Amazon realisiert das AWS IoT Device SDK for Java unter Verwendung des *jackson*-JSON-Serialisierers. Er erwartet, dass jede mit dem Attribut *AWSIoTDevice-Property* ausgestattete Variable durch eine Getter- und eine Setter-Methode flankiert wird. Diese auf den ersten Blick pedantisch erscheinende Vorgehensweise ist insofern gerechtfertigt, als die Amazon Engine diese beiden Methoden zum Anmelden der vom Server ankommen den und zum Ernten von am Gerät befindlichen Informationen einspannen wird.

Zur Behebung der Fehlermeldungen reicht es aus, nach dem in Listing 4 gezeigten Schema sowohl Getter als auch Setter zu realisieren.

Im Interesse der besseren Überwachbarkeit des resultierenden Programms sollten Sie an dieser Stelle sowohl für die Getter als auch für die Setter Logging-Operationen platzieren. Im Interesse der Kompaktheit drucken wir diesen Code an dieser Stelle nicht ab, da er sich aus der Logik ergibt.

Wer das mit Instrumentierung ausgestattete Programm zur Ausführung freigibt, sieht nach dem Start hohe Aktivität. Danach ruft das AWS IoT Device SDK for Java von Haus aus alle fünf Sekunden einmal die Methode *getMyStringVal* auf. Wenn Sie auch in der für die *Float*-Variable vorgesehenen Methode Instrumentierung platzieren, finden sich zudem Aufrufe von *getMyFloatVal*.

Dieses auf den ersten Blick seltsame Eigenleben des AWS IoT Device SDK for Java ist darin begründet, dass die in *reported* vorgehaltenen Informationen am Server immer ein zumindest halbwegs aktuelles Abbild der physikalischen Realität um das Endgerät darstellen sollen. Von Haus aus legt Amazon dabei eine Übertragung für alle fünf Sekunden fest, was aus Sicht des „Buchhändlers“ einen akzeptablen Kompromiss zwischen Aktualität und Datenverbrauch darstellt.

Erfreulicherweise lässt sich die Aktualisierungsgeschwindigkeit des SDK über die *setReportInterval*-Methode beeinflussen:

```
long reportInterval = 15000; // milliseconds.
device.setReportInterval(reportInterval);
```

Als besonders haarig erweist sich eigentlich nur, dass das AWS IoT Device SDK for Java den Aktualisierungsgeschwindigkeitswert in Millisekunden erwartet. Schon aufgrund von Netzwerklatenzen, RTC-Ungenauigkeit

ten und Co. dürfen Sie hier allerdings keine an einen Totalisator mit GPS-Disziplinierung erinnernde Genauigkeit erwarten.

An dieser Stelle können wir das Leseprogramm abermals ausführen. Es wird nun den folgenden String in die Konsole ausgeben, den wir hier im Interesse der Übersichtlichkeit gekürzt abdrucken:

```
{"state":{"reported":{"sensor":9.0,"myFloatVal":0.0}),"metadata":
    {"reported":{"sensor":{"timestamp":1580419131,"myFloatVal":
        {"timestamp":1580419144}}}, "version":14, ...}
```

Auffällig ist, dass der von Haus aus leere Stringwert nicht im Device Shadow auftaucht. Stattdessen erscheint neu nur das Float, da die Software Standarddefaultwerte von null nicht von einer Null unterscheiden kann, die ein Algorithmus im Rahmen einer Messung in die Speicherstelle abgelegt hat.

Zur Herstellung eines konsistenteren Zustands könnten wir das Schreibprogramm folgendermaßen adaptieren:

```
client.connect();
device.delete();
device.setMyStringVal("Hallo Welt!");
device.setMyFloatVal((float) 2.2);
```

Die neue Version ruft im ersten Schritt *delete* auf, um alle schon im Device Shadow befindlichen Informationen zu löschen. Danach aktualisieren wir die Inhalte, um dem Leseprogramm einen konsistenten Zustand anzubieten. Wer es an dieser Stelle abermals zur Ausführung freigibt, bekommt nun das folgende Ergebnis:

```
{"state":{"reported":{"myStringVal":"Hallo Welt!","myFloatVal":2.2}}}
```

Ergebnisüberprüfung im Backend

An dieser Stelle sind Sie möglicherweise dazu motiviert, fortgeschrittene Versuche durchzuführen. Hierbei ist es wünschenswert, wenn man die Inhalte des Device Shad-

Listing 3

```
com.fasterxml.jackson.databind.JsonMappingException: Unexpected IOException
    (of type java.io.IOException): java.lang.IllegalArgumentException: java.lang.
    NoSuchElementException: ShadowWorker.getMyStringVal()
    at com.fasterxml.jackson.databind.JsonMappingException.fromUnexpectedIOE(JsonMapp
        ingException.java:338)
```

Listing 4

```
public String getMyStringVal() {
    return myStringVal;
}
public void setMyStringVal(String _x) {
    myStringVal = _x;
}
public float getMyFloatVal() {
    return myFloatVal;
}
public void setMyFloatVal(float _x) {
    myFloatVal = _x;
}
```

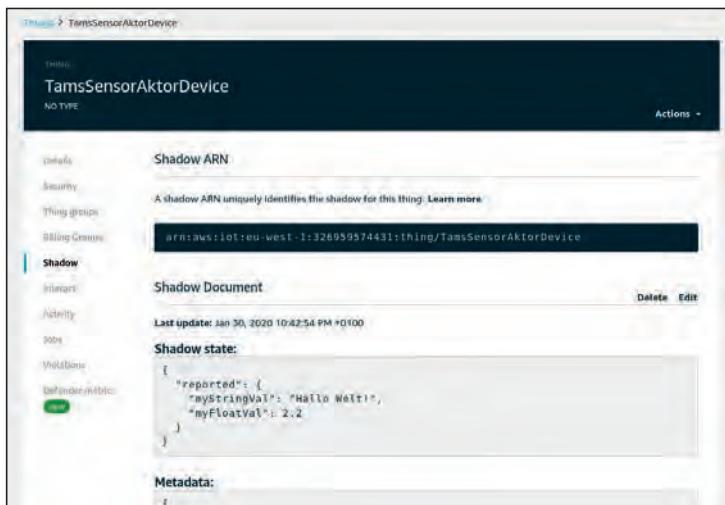


Abb. 3: Nach der Extraeinladung erscheinen die Inhalte am Bildschirm



Abb. 4: Die Änderung vom Server wirkt sofort

ows bzw. sogar des ganzen Geräts auch im weiter oben verwendeten Browser-Backend verwenden kann.

Amazon erweist sich dabei insofern als unkooperativ, als der Buchhändler die diesbezüglichen Informationen von Haus aus nicht anzeigt. Sie müssen stattdessen im ersten Schritt in die Rubrik **MANAGE | THINGS** wechseln, die Sie bei einem „jungfräulichen“ AWS-Konto mit der Aufforderung „Register a thing“. begrüßt. Klicken Sie den Knopf an, um den in der Fachliteratur auch als an Boarding bezeichneten Gerätanmeldeprozess zu aktivieren.

Cloud-Anbieter liefern sich seit längerer Zeit durchaus erbitterte Gefechte darum, wer die Anmeldung einer größeren Kohorte von Hardwareendgeräten am bequemsten bewerkstelligt. Da wir im Moment allerdings nur die Informationen unseres in Java lebenden Geräts sichtbar machen wollen, entscheiden wir uns für den zur Option „Register a single AWS IoT thing“ gehörenden Aktionsknopf.

Im ersten Schritt erfragt der Assistent vergleichsweise umfangreiche Konfigurationseinstellungen und erlaubt auch, unser Ding einer Gruppe zuzuweisen. Im Interesse

Listing 5

```
{
  "reported": {
    "myStringVal": "Hallo Welt!",
    "myFloatVal": 2.2
  },
  "desired": {
    "myStringVal": "Hallo SUS!",
    "myFloatVal": 3.3
  }
}
```

der Bequemlichkeit vergeben wir hier als Namen den String *TamsSensorAktorDevice*, der von weiter oben bekannt sein dürfte. Der Rest des Formulars bleibt unberührt. Im zweiten Schritt entscheiden wir uns für die Option „Skip certificate and create thing“. Sie weist das AWS Backend dazu an, unser Ding ohne fortgeschrittene typografische Zertifikate zur Identitätssicherung zu erzeugen.

An dieser Stelle ist die Arbeit auch schon abgeschlossen, die Ergebnisse des Prozesses lassen sich, wie in **Abbildung 3** gezeigt, sofort sehen.

Hier bietet sich ein kleiner Test an: Starten Sie das Schreibprogramm und versuchen Sie danach, unter Verwendung des Shadow-State-Werkzeugs eine Änderung des Strings durchzuführen. Der Amazon-Server wird diese zwar im ersten Schritt quittieren, um sie danach aber wieder abzulehnen. Ursache dieses auf den ersten Blick unnatürlich erscheinenden Verhaltens ist, dass das Device-Shadow-Dokument neben dem hier bearbeiteten Block auch einen *Desired*-Block aufweist, der für die Übertragung von am Server durchgeföhrten Änderungen in Richtung der physikalischen Hardware verantwortlich ist.

Zur Behebung des Problems müssen wir, wie in Listing 5 gezeigt, den JSON-String adaptieren, um die gewünschte Änderung über das *Desired*-Feld anzuliefern.

Wundern Sie sich nicht, wenn während der Abarbeitung der Payload kurzfristig ein JSON-String mit einem Deltablock aufscheint. Er liefert Informationen darüber, welche Änderung im Datenbestand gerade erfolgt ist. Der Lohn der Mühen ist die in **Abbildung 4** gezeigte Aktualisierung, die sich auch am Endgerät auswirkt.

Fazit

Device Shadows mögen bei der Realisierung von Eins-zu-eins-Verbindungen zwischen Cloud-Dienst und Server nicht wirklich hilfreich sein. Ihre Macht spielen sie in dem Moment aus, in dem der Entwickler kompliziertere Verfahren realisieren möchte und sich dabei nicht mit dem Verbindungszustand rumärgern will. Das ist übrigens eine Aufgabe, an der wenig Embedded-erfahrene Entwickler routiniert verzweifeln.

Damit wollen wir unsere Reise in die Welt des AWS API für Java allerdings auch schon beenden. Wir hoffen, dass Sie mit diesem kleinen Exkurs jede Menge Spaß hatten.



Tam Hanna befasst sich seit der Zeit des Palm IIIc mit der Programmierung und Anwendung von Handcomputern. Er entwickelt Programme für diverse Plattformen, betreibt Onlinenewsdienste zum Thema und steht für Fragen, Trainings und Vorträge gern zur Verfügung.



tamhan@tamogemon.com



Crazy Electronics Lab | @tam.hanna



@tamhanna

Links & Literatur

[1] <https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-data-flow.html>

[2] Hanna, Tam: „AWS IoT Device SDK for Java“, Java Magazin 4.2020

Anzeige

Vorschau auf die Ausgabe 6.2020

Javalin, Ktor, Spring Fu, Micronaut und Co. – Orientierung im Infrastrukturdschungel

In der Java-Welt gewinnen Microframeworks wie Javalin, Ktor, Spring Fu und Micronaut immer mehr an Bedeutung. Warum? Minimalistische Web-Frameworks fokussieren sich beim Bau modularer Anwendungen auf die zentralen Konzepte und stellen so die Developer Experience in den Vordergrund. Zudem zeichnen sie sich durch ihre klare Cloudausrichtung und die Eignung für die leichtgewichtige Erstellung von Microservices aus. Im nächsten Java Magazin nehmen wir dieses spannende und wichtige Thema in den Blick.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 6. Mai 2020

Querschau

PHPmagazin

Ausgabe 3.2020 | www.phpmagazin.de

- **PHPUnit 9: Informationen zur Major-Version aus erster Hand**
- **PhpStorm effektiv nutzen: 5 Tipps vom Profi**
- **Der Änderung auf der Spur: Changelogs mit dem CLI-Tool Changelogger**

entwickler

Ausgabe 3.2020 | www.entwickler-magazin.de

- **Progressive Web Apps: Mit Project Fugu in die Zukunft**
- **Vue.js 3: Die neue Major-Version kommt**
- **Flutter: Ab in die Future**

windows .developer

Ausgabe 4.2020 | www.windows-developer.de

- **Starke Typen: TypeScript-3-Neuerungen im Überblick**
- **Der 6+1-Punkte-Plan: Erstellen einer Micro-Frontend-Shell**
- **SQL Server 2019: Neue Features, Grundlagen und Einsatzszenarien**

Inserenten

Adesso	35	JAX	16
www.adesso.de		www.jax.de	
ARS	9	Lufthansa Industry Solutions	65
www.ars.de		www.lufthansa-industry-solutions.com	
API Conference	73	ML Conference	42
www.apiconference.net		www.mlconference.ai	
CaptainCasa GmbH	49	NTT DATA Deutschland	37
www.captaincasa.co		de.nttdata.com	
Develop Your Future	99	OIO/Trivadis Germany GmbH	45
www.develop-your-future.com		www.oio.de	
Entwickler Akademie	23, 33, 51, 53, 73, 79, 87, 97	Serverless Architecture Conference	75
www.entwickler-akademie.de		www.serverless-architecture.io	
entwickler.kiosk	2, 100	Software & Support Media GmbH	61
www.entwickler-kiosk.de		www.sandsmedia.com	
entwickler.tutorials	85	webinale	63
www.entwickler-tutorials.de		www.webinale.de	
International JavaScript Conference	13		
www.javascript-conference.com			

Verlag:

Software & Support Media GmbH

Anschrift der Redaktion:

Java Magazin

Software & Support Media GmbH

Schwedlerstraße 8

D-60314 Frankfurt am Main

Tel. +49 (0) 69 630089-0

Fax. +49 (0) 69 630089-89

redaktion@javamagazin.de

www.javamagazin.de

Chefredakteur:

Sebastian Meyen

Redaktion:

Katharina Degenmann, Dominik Mohilo,

Marius Nied, Hartmut Schlosser

Chefin vom Dienst/Leitung Schlussredaktion:

Frauke Pesch

Schlussredaktion:

Jonas Bergmeister, Dr. Anne Lorenz

Leitung Grafik & Produktion:

Jens Mainz

Layout, Titel:

Tobias Dorn, Simon Hußnagel, Dominique Kalbassi, Theresa Radig, Bianca Röder, Maria Rudi,

Sibel Sarli, Sandra Schalk, Vincent Schlothauer, Michael Schütze

Autoren dieser Ausgabe:

Elena Bochkor, Ronny Bräunlich, Dewet Diener, Oliver

B. Fischer, Thilo Frotscher, Steffen Grunwald, Dr. Heinrich

Kabutz, Markus Günther, Tam Hanna, Andreas Jaekel,

Lars Kölpin, Dr. Veikko Krypczyk, Tim Riemer, Jens

Schauder, Marco Schulz, Falk Sippach, David Tan,

Michael Vitz, Tim Zöller

Anzeigenverkauf:

Software & Support Media GmbH

Anika Stock

Tel.: +49 (0) 69 630089-22

Fax: +49 (0) 69 630089-89

anika.stock@sandsmedia.com

Es gilt die Anzeigenpreisliste Mediadaten 2020

Pressevertrieb:

PressUp GmbH

Tel +49(0) 4041448-411

www.pressup.de

Druck:

Westdeutsche Verlags- und Druckerei GmbH

Kurhessenstraße 4 – 6

64546 Mörfelden-Walldorf

ISSN: 1619-795X

Abonnement und Betreuung:

Leserservice Java Magazin

65341 Eltville

Tel.: +49 (0) 6123 9238-239

Fax: +49 (0) 6123 9238-244

javamagazin@vuservice.de

Abonnementpreise der Zeitschrift:

Inland: 12 Ausgaben € 118,80

Europ. Ausland: 12 Ausgaben € 134,80

Studentenpreis (Inland) 12 Ausgaben € 95,00

Studentenpreis (Ausland): 12 Ausgaben € 105,30

Einzelverkaufspreis:

Deutschland: € 9,80

Österreich: € 10,80

Schweiz: sFr 19,50

Luxemburg: € 11,15

Erscheinungsweise: monatlich

© 2020 Software & Support Media GmbH

Alle Rechte, auch für Übersetzungen, sind vorbehalten.

Reproduktionen jeglicher Art (Fotokopie, Nachdruck,

Mikrofilm oder Erfassung auf elektronischen Datenträgern) nur mit schriftlicher Genehmigung des Verlages.

Eine Haftung für die Richtigkeit der Veröffentlichungen kann trotz Prüfung durch die Redaktion vom Herausgeber nicht übernommen werden. Honorierte Artikel gehen

in das Verfügungsrecht des Verlags über. Mit der Über-

gabe der Manuskripte und Abbildungen an den Verlag erteilt der Verfasser dem Herausgeber das Exklusivitäts-

recht zur Veröffentlichung. Für unverlangt eingeschickte

Manuskripte, Fotos und Abbildungen keine Gewähr.

Java™ ist ein eingetragenes Warenzeichen von Oracle

und/oder ihren Tochtergesellschaften.

Anzeige

Anzeige