

# JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis #JAVAPRO

**JAVA JETZT SO SCHNELL WIE C!**

## DIE ZUKUNFT VON JAVA

**GRAALVM - QUARKUS - MICROSTREAM**

16 **DIE MAGIE  
DER LAMBDAS**

21 **OPENJDK  
AMAZON CORRETTO**

30 **JAVAEV VS SPRING  
FÜR MICROSERVICES**

36 **QUARKUS -  
DIE ZUKUNFT VON JAVA**

40 **NATIVE ANWENDUNGEN  
MIT JAVA**

45 **MICROSTREAM 4 -  
HIGH-PERFORMANCE-PERSISTENCE**

51 **BOILERPLATE CODE  
MINIMIEREN MIT LOMBOK**

[www.jcon.one](http://www.jcon.one)

Java • Architektur • Cloud • Agile



**JCON  
2020**

[www.jcon.one](http://www.jcon.one)

**Online-Konferenz  
27. - 30. Oktober 2020**

**Kostenlos für JUG-Mitglieder !  
500 JUG-Tickets verfügbar.**

**Mit freundlicher Unterstützung unserer Partner.**

# JAVAPRO PARTNER NETWORK

Die JAVAPRO wird von den Mitgliedern des JAVAPRO Partner Network finanziert und aktiv unterstützt. Dadurch sind wir in der Lage, redaktionell unabhängig zu arbeiten und die JAVAPRO kostenlos für die gesamte Java-Community zu produzieren sowie die Java Konferenz JCON 2020 zu veranstalten.

## GOLD PARTNER

ORACLE®

*Fast Lane*  
Google Cloud

  
MicroStream

 eclipse

XDEV™

## SILBER PARTNER

 RAPIDclipse™

 JFrog

 TimoCom

trivago®

 consol  
Wir unternehmen IT

## BRONZE PARTNER

viadee®  
IT-Unternehmensberatung

raytion™

 TK  
The Technology

 GEBIT  
Solutions

X2019  
DEVCON

4Soft

MichaelPage

 ETEKTURE

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

**[www.javapro.io/partner](http://www.javapro.io/partner)**

# JAVAPRO

## Impressum

**JAVAPRO**

**Verlag:**

JAVAPRO

Mergenthalerallee 73-75

65760 Eschborn

Telefon: +49 (0)800 - 528 277 6

E-Mail: [info@javapro.io](mailto:info@javapro.io)

Website: <http://www.javapro.io>

**Chefredakteur:**

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

**Redaktion:**

[info@javapro.io](mailto:info@javapro.io)

**Gestaltung, Layout, Produktion:**

Impuls Mediengruppe GmbH

Im Gewerbepark 29

92681 Erbendorf

**Preis:** kostenfrei

**Illustrationen:**

Pixabay (Public Domain)

**Erscheinungsweise:** Vier mal jährlich

**Gründungsjahr:** 2017

Copyright (c) 2020

Impuls Mediengruppe GmbH

Alle Rechte vorbehalten.

Java(TM) ist ein eingetragenes Warenzeichen der Oracle Corporation. Javapro ist ein unabhängiges Magazin und wird nicht von der Oracle Cooperation gesponsert.

Namentlich gekennzeichnete Artikel geben nicht unbedingt die Meinung der Redaktion wieder.

## #JAVAPRO #Editorial

**C**OVID-19 ist auch an der JAVAPRO nicht spurlos vorübergegangen. Wir mussten die Produktion kurzzeitig einstellen. Aber ab sofort sind wir wieder für euch da – sogar mit verstärktem Team, ungebrochen großer Motivation und enormen Spaß an der Arbeit an diesem tollen Magazin. Ab sofort wird wieder alle drei Monate eine neue JAVAPRO Ausgabe erscheinen.

Mit dieser Ausgabe wollen wir die derzeit wohl brennendste Frage in Bezug auf Java beantworten: „Wird Java gerade von anderen Technologien überholt, soll man auch weiterhin auf Java setzen, welche Innovationen sind von Java in naher Zukunft zu erwarten?“

Seit mehr als zwanzig Jahren ist Java die führende Programmiersprache und Anwendungsplattform. Doch jetzt stehen wir an der Schwelle einer neuen Software-Epoche: Cloud-Native, Serverless-Computing, Big-Data, Künstliche Intelligenz, Virtual-Reality, Internet-of-Things und Automotive sind keine Science-Fiction mehr, sondern mittlerweile alltägliche Anforderungen in Unternehmen und Startups. Je schneller solche Systeme arbeiten, zum Beispiel KI- und Machine-Learning-Algorithmen, je schneller Massendaten und komplexe Datenstrukturen verarbeitet werden können, desto leistungsfähiger und effizienter werden die Systeme und machen bestimmte Anwendungsfälle wie selbstfahrende Autos überhaupt erst möglich, desto weniger Rechenleistung ist erforderlich um solche Anwendungen zu betreiben und desto weniger Infrastrukturkosten fallen letztendlich an. Immer mehr Unternehmen verlagern immer mehr Anwendungen in die Cloud und merken jetzt, dass sie die Übersicht verlieren und ihre Kosten explodieren. Spätestens in der Post-Corona Zeit müssen Unternehmen ihre Kosten drastisch senken und achten mehr als je zuvor auf effiziente, ressourcenschonendere Systeme. Bei Neuentwicklungen und Migrationen in die Cloud spielen zudem Entwicklungszeit und Time-to-Market eine wichtige Rolle.

Viele IT-Entscheider und noch mehr jüngere Entwickler haben mittlerweile große Zweifel, ob Java für all diese Anforderungen immer noch die richtige Technologie ist. Seit jeher hängt Java der Ruf nach kompliziert, träge, schwergewichtig und speicherhungrig zu sein und die Weiterentwicklung von Java unterliegt einem aufwändigem, bürokratischem Entscheidungsprozess, wodurch die Community auf längst fällige Innovationen oftmals viele Jahre warten muss. Manche bezeichnen Java sogar schon

als Dinosaurier. Derweil scheinen andere Sprachen wie Python, JavaScript sowie Cloud-Native- und Serverless-Ansätze mehr und mehr Java den Rang abzulaufen. Im Silicon Valley, dem Hotspot für Hightech und Innovationen, entwickelt niemand mehr mit Java, hört man dort immer häufiger und das stimmt bedenklich. Dass Oracle selbst vor zwei Jahren seine traditionelle Java-Konferenz JavaOne in eine multitechnologische Konferenz (Oracle CodeOne) umstrukturiert und sich nun auch für konkurrierende Sprachen völlig offen zeigt, erweckt zudem den Eindruck, dass selbst für den Hersteller Java nicht mehr an erster Stelle steht. Entsprechend groß war der Aufschrei in der Java Community. Als Cloud-Provider, der mit aller Macht Marktanteile gewinnen möchte, hat Oracle allerdings auch gar keine Wahl als sich auch für andere Sprachen zu öffnen. Die Gedanken dahinter sind also nachvollziehbar. Die Frage, ob man überhaupt noch auf Java setzen sollte, aber auch.

Mit dieser Ausgabe wollen wir diese brennende Frage beantworten. So viel schon mal vorab: Es gibt richtig gute Nachrichten für uns Java Entwickler: 1. So schnell und einfach sterben Dinosaurier nicht aus. Nur wenn erneut ein großer Meteorit die Erde trifft, könnte es auch für Java brenzlich werden. 2. Der Heilige Graal, der zum ewigen Leben verhelfen soll, ist bereits gefunden! Mit GraalVM und einer neuen Generation von Microservice-Frameworks wie Quarkus, Micronaut, Helidon und MicroStream entsteht gerade ein völlig neuer Java-Cloud-Native Technologie-Stack, der sich perfekt für den Einsatz in der Cloud eignet und Java auf ein völlig neues Level bringt: Java-Anwendungen genauso schnell wie mit C, Anwendungsstart in Millisekunden, minimaler Memory-Footprint und bis zu 1000 Mal schnellere Datenbankzugriffe. Das ist keine Vision, sondern heute schon möglich. In dieser Ausgabe erfahren Sie alles Wichtige über den neuen Java-Cloud-Native Stack. Das gesamte JAVAPRO-Team wünscht euch viel Spaß beim Lesen!



**Markus Kett**  
Chefredakteur  
JAVAPRO

Twitter: @MarkusKett  
LinkedIn: markuskett

## 36

FRAMEWORKS &amp; APIS

**Quarkus - die Zukunft von Java ?**

Von Bernd Fischer

Wenn es um Entwicklungszykluszeiten, Startzeiten von Anwendungen, Größe der Deployments und Kompatibilität in oder mit Containerumgebungen geht, scheinen andere Sprachen wie JavaScript, Python oder Go gerade Java den Rang abzulaufen. Mit GraalVM und Quarkus soll sich das jetzt ändern: Native Anwendungen mit Java, Performance vergleichbar mit C, Startzeiten unter einer Sekunde und dazu alle bereits bekannten Vorteilen von Java. Damit dringt Java in eine neue Dimension vor und setzt völlig neue Maßstäbe.

## 45

FRAMEWORKS &amp; APIS

**MicroStream 4 - High-Performance Java-Native Persistence**

Von Richard Fichtner

Auf Basis von GraalVM entsteht ein neuer Java-Stack für moderne Cloud-Native-Applikationen, der Java auf ein völlig neues Level bringt. Mit MicroStream 4 kommt jetzt eine hoch performante und gleichzeitig leichtgewichtige Persistence dazu die mit bis zu 1000 Mal schnelleren Datenbankzugriffen eine bisher nie dagewesene Performance erreicht. Applikationen benötigen nur noch einen Bruchteil an Rechenleistung, was Anwendern bis zu 90% Cloud-Kosten spart. Gleichzeitig vereinfacht und beschleunigt das elegante Programmiermodell die Entwicklung.

## 16

CORE JAVA

**Die Magie der Lambdas**

Lambdas - ein mächtiges Werkzeug das jeder Entwickler kennen sollte. Damit lässt sich Code kompakter und besser lesbar gestalten.

## 21

CORE JAVA

**OpenJDK Amazon-Corretto**

Das Corretto JDK kommt bei Amazon bei tausenden produktiven Workloads zum Einsatz. Grund genug sich die Vorteile genauer anzusehen.

## 25

CORE JAVA

**JSR-385 hätte Mars Orbiter retten können**

Fehlerhafte Umrechnungen können fatale Folgen haben. Mit dem neuen Units-of-Measurement-API 2.0 soll nun alles besser werden.

## 30

SERVERSIDE JAVA &amp; JAVA ENTERPRISE

**Java-EE vs. Spring für Microservices**

Auch wenn es um die Entwicklung von Microservices geht, stellt sich die Frage: Java-EE oder Spring - welcher Ansatz ist besser?

## 36

FRAMEWORKS &amp; APIS

**Quarkus - die Zukunft von Java?**

Java erreicht neue Dimensionen: Native Anwendungen, Performance wie C, Startzeit in Millisekunden und alle Vorteile von Java.

## 40

FRAMEWORKS &amp; APIS

**Native Anwendungen mit Java**

Quarkus bringt GraalVM mit vertrauten Frameworks zusammen und bietet sagenhafte Performance mit optimiertem Build-Prozess.

## 45

FRAMEWORKS &amp; APIS

**MicroStream 4 - High-Performance Java-Native Persistence**

Mit MicroStream lassen sich Objektgraphen und Teilgraphen speichern, dynamisch und updaten. Vorteil: bis 1000 Mal schnellere Queries.

## 51

FRAMEWORKS &amp; APIS

**Boilerplate-Code minimieren mit Lombok**

Lombok versteckt Boilerplate-Code wie Getter, Setter und was IDEs noch generieren. Der Code wird kürzer und besser lesbar.

## 55

FRAMEWORKS &amp; APIS

**Kafka-Connect - Drittsysteme an Kafka anbinden**

Kafka-Connect ist neben der Kafka-API der einfachere Weg um mit Kafka zu interagieren oder um Drittsysteme wie Redis anzubinden.

## 59

FRAMEWORKS &amp; APIS

**Bessere Java-Desktop Deployments**

Mit Java 14 gibt es jetzt endlich wieder anwenderfreundliche Lösungen um klassische Java-Desktop-Applikationen auszuliefern.

#JCON2020

**JCON 2020**

DIE GROSSE JAVA COMMUNITY KONFERENZ

**27. - 30. Oktober 2020 - ONLINE !**

Workshop-Day am 30. Oktober 2020

**www.jcon.one**

|                  |                              |                          |                        |                               |                              |
|------------------|------------------------------|--------------------------|------------------------|-------------------------------|------------------------------|
| <b>4</b><br>Days | <b>5</b><br>Streams parallel | <b>9</b><br>Special Days | <b>110+</b><br>Speaker | <b>150+</b><br>Sessions Live! | <b>1000+</b><br>Participants |
|------------------|------------------------------|--------------------------|------------------------|-------------------------------|------------------------------|

62

FRAMEWORKS &amp; APIS

**Keycloak individuell erweitern**

So lassen sich Anpassungen für das Identity- und Access-Management mit Keycloak vornehmen - vom eigenen Modul bis zum Logging.

68

ARCHITECTURE

**Self-Contained-Systems**

SCS ist ein Architektur- und Organisationsansatz, der Trennung von Systemen und Strukturierung größerer Teams erleichtern soll.

73

ARCHITECTURE

**Architektur-Hotspots aufspüren**

Mit neuen Konzepten zur Feature-Modularität lassen sich Architektur-Hotspots aufspüren, die den Wartungsaufwand erhöhen.

80

IDE &amp; TOOLS

**Loggen mit Struktur**

Log-Server sind komplette Datenbanksysteme. Durch passende Log-Meldungen ergeben sich für Log-Daten völlig neue Möglichkeiten.

86

DATENBANKENTWICKLUNG

**H2 für JUnit-Tests**

H2 ist eine relationale Open-Source-Datenbank, die sich für automatisierte Unit-Tests, in Prototypen und kleine Services eignet.

89

AGILE &amp; PROJEKTMANAGEMENT

**Von Nixdorf nach Kubernetes - Digitalisierung im Wandel der Zeit**

Digitalisierung ist die Zukunft. Tatsächlich? Unser Mittelstand betreibt Digitalisierung schon seit den 1970ern, zu Nixdorf Zeiten.

93

AGILE &amp; PROJEKTMANAGEMENT

**Business-Impact-Analyse mit nur 5 Slides**

Unternehmen sollten dringend ihre kritischen IT-Systeme erfassen. Eine Business-Impact-Analyse lässt sich auf 5 Folien reduzieren.

98

INNOVATIONEN

**Vernetzte Systeme mit FEPCOS-J**

FEPCOS-J beschleunigt die Entwicklung vernetzter Systeme durch die Vereinfachung von Netzwerkprogrammierung und Nebenläufigkeiten.

2

MAGAZIN

**Partner Network & Impressum**

3

MAGAZIN

**Editorial**

#JCON2020  
[www.jcon.one](http://www.jcon.one)

JAVAPRO



DIE GROSSE JAVA COMMUNITY KONFERENZ  
**27. - 30. Oktober 2020 - ONLINE !**

**Workshop-Day am 30. Oktober 2020**

[www.jcon.one](http://www.jcon.one)

**4**

Days

**5**

Streams parallel

**9**

Special Days

**110+**

Speaker

**150+**

Sessions Live!

**1000+**

Participants

JCON-ONLINE 2020 PARTNER

GOLD PARTNER

ORACLE®

eclipse

MicroStream

XDEV™

SILBER PARTNER

RAPIDclipse™

consol  
Wir unternehmen IT

JFrog

TimoCom.

ORGANISATIONS-PARTNER

JAVAPRO



JUG Oberpfalz

## JCON-Online 2020

### JCON-Online 2020

Die JCON ist die große Java Community Konferenz der JAVAPRO - dem ersten kostenlosen Fachmagazin für Java. Bedingt durch COVID-19 findet die JCON heuer erstmals nicht im UCI Kino in Düsseldorf statt, sondern als reine Online-Konferenz. Alle Sessions werden live übertragen und aufgezeichnet. Das Online-Konzept ist zwar neu für uns alle, bietet aber eine Menge Vorteile für alle Teilnehmer: keine Anreise nötig, keine Reise- und Übernachtungskosten, schnell und einfach zwischen Vorträgen wechseln, einfach einloggen, Session auswählen und dabei sein. Und zudem bietet die JCON-ONLINE 2020 noch mehr Sessions und hochkarätige Speaker und damit noch mehr Auswahl und geballtes Java Know-how als jemals zuvor.

### JCON = Core Java

Bei der JCON liegt der Schwerpunkt traditionell auf Core Java, Serverside Java, APIs und Frameworks, denn damit arbeiten wir als Java Entwickler tagtäglich. Auch heuer bilden 2 zeitgleiche Live-Streams zu diesen Themen die JCON-Hauptkonferenz.

Tag 1 **Agile Day**

**Testing Day**

**Rapidclipse Day**

Tag 2 **Architecture Day**

**Java-Cloud-Native Day**

**Cloud Development Day**

Tag 3 **DevOps Day**

**Web Development Day**

**CIO Day**



# JCON2020



# DIE GROSSE JAVA COMMUNITY KONFERENZ

27. - 30. Oktober 2020 - ONLINE!



Core Java



Serverside Java &  
Java EE



APIs &  
Frameworks



Serverless /  
Cloud Native



Microservices



Architecture



Agile



Web  
Development



Mobile  
Development



Cloud Platforms



Container



DevOps



Continuous Delivery



IDE & Tools



Testing & Quality



UI & UX



Performance



Security



Big- / Fast- / Smart-  
Data



IoT & Embedded



Innovations

#JCON2020

Unsere großartigen Speaker





# DIE GROSSE JAVA COMMUNITY KONFERENZ

## 27. - 30. Oktober 2020 - ONLINE!

### Workshop DAY

Freitag, 30. Oktober 2020 • 10 Power Workshops gleichzeitig - Live!  
Teilnehmerzahl pro Workshop begrenzt! - Beginn & Dauer: 09:00 bis 13:00 Uhr.

#### MicroStream Training Day: Create ultra-fast in-memory database apps with Java

Trainer: **Markus Kett & Florian Habermann**, *MicroStream*

Deep-Dive in die MicroStream Storage-Engine mit Live-Coding. Was gibt es Neues? Setup Storage-Targets Was ist eine Root? Konfiguration Daten speichern und laden Lazy-Loading Legacy Type Mapping / Refactoring Backup-Strategien Customizing Data Viewer Wenn es die Zeit erlaubt: JCache (JSR 107) Android

#### RapidClipse - Visual Java IDE

Trainer: **Sebastian Späth & Christian Kümmel**, *MicroStream*

In diesem Workshop erhalten Sie einen Überblick über die Funktionsweise von RapidClipse in Verbindung mit MicroStream. Während des Workshops erfahren Sie alles über das AppLayout, FlexLayout und FormLayout um eine Responsive Business-App zu erstellen. Hierbei wird das Layout einer Web-Application so flexibel gestaltet, dass dieses auf dem Computer- Desktop, Tablet und Smartphone eine gleichbleibende Benutzerfreundlichkeit bietet und der Inhalt gänzlich und schnell vom Besucher aufgenommen werden kann. Für das Speichern von Daten wird hinter den Kulissen das MicroStream Framework verwendet. Abschließend wird die fertige Anwendung als PWA-Application auf einem Server zur Verfügung gestellt.

#### Reactive Spring

Trainer: **Patrick Baumgartner**, *42talents*

In this workshop, we will use the current versions of Spring Boot, Spring Framework and Project Reactor to write Reactive code and will introduce RSocket, an application protocol to supports Reactive Streams. You'll get hands-on experience with building a Reactive application to stream data leveraging the newly available Reactive data types, Spring WebFlux, Spring Data, and RSocket. Besides that, we will work with some new tools which make dealing with errors and blocking APIs visible. Agenda • Introduction Reactive Streams, Publisher/Subscriber types and Reactor types • Getting familiar with Project Reactor • Using Reactive RabbitMQ with Spring • Using RSocket as a streaming message protocol • Using Spring WebFlux • Functional configuration API for Spring WebFlux • Using Reactive Types with Thymeleaf • Using Spring Security Reactive • Using Spring Data MongoDB to reactively stream data • Using Spring Data R2DBC • Using Reactive Redis with Spring • What's next? The goal of the workshop Participants will build and test the individual parts of a Spring Boot microservices infrastructure themselves. We will use different tools and demonstrate the interaction between them. At the end of the day, participants will take along a backpack of tools, working techniques, and terms that will help them successfully implement reactive components with Spring Boot. Target Group This workshop is targeted at developers and architects who are new to reactive programming and want to learn how to use the Spring ecosystem to get started. Requirements

#### Cloud Native Java Performance Tuning

Trainer: **Ingo Düppe**, *CROWDCODE*

Das mit Java hoch performante, verteilte Systeme entwickelt werden können, beweise zahlreiche Beispiele. Zudem kommen neuerdings Aspekte wie Cloud Native hinzu, welche den Startprozess von Java Anwendungen oder das Langzeit Speicherverhalten der JVM in den Fokus rücken. Die jüngsten JDKs bieten hierzu neue Konzepte zur Optimierungsstrategien an. Das Java Eco-System bietet ein sehr umfangreiches Feld an Methoden und Werkzeugen, um die Performance von Java-Anwendungen zu optimieren. Es werden die typischen Ursachen für die Entstehung von Performance-Engpässen gezeigt und mit welchen Strategien diese im Vorfeld vermieden werden können. Ziel des Workshops ist es den Teilnehmer die methodische Analyse der Performance von Java-Enterprise-Anwendungen zu zeigen. Hierzu werden die wichtigsten Konzepte aktueller JVMs aufgezeigt und wie Open Source- und kommerzielle Werkzeuge systematisch eingesetzt werden können.

#### Cloud Platform Hands-On Labs: Docker & Kubernetes in der Praxis

Trainer: **Thorsten Jakoby & Immanuel Haag**, *Novatec Consulting*

Sie haben zwar schon viel von Docker und Kubernetes gehört, aber noch nie selbst Hand angelegt? Pods, ReplicaSets & Ingress Controller klingen immer noch ein wenig kryptisch? Möchten Sie eigene Docker Container für Ihre Anwendung erstellen und diese anhand bewährter Methoden in Kubernetes deployen? Dann haben wir das richtige Training für Sie! Dieser Workshop vermittelt den Teilnehmern fundierte Kenntnisse und praktische Erfahrung mit Docker & Kubernetes in einer bereitgestellten Umgebung. Wir beginnen unsere Reise mit der Erläuterung der Kernkonzepte, bevor wir uns in geführter Vorgehensweise den Übungen zuwenden. Um ein umfassendes Verständnis von Docker & Kubernetes aus Entwicklersicht aufzubauen beinhalten die Übungen folgenden Ablauf: Bauen der Container einer verteilten Anwendung, Deployment auf einem Kubernetes Cluster, Verknüpfung der Komponenten untereinander, Bereitstellung für den End-User, (automatische) Skalierung und dynamische Updates ohne Downtime.

### Live a happier life using Spring Cloud Contract

Trainer: **Fabian Krüger**, Pivotal now VMware

For sure, you are a good citizen! You write a whole set of unit and integration tests for your neat and beautiful microservices. It's all clean code for sure. The message topics, along with REST APIs provided by other Teams, are used as agreed upon in a whole series of meetings, emails and slack messages. But then... comes the day when you need to integrate your microservice into the landscape of existing microservices... and... BOOM! Consumed APIs and message formats suddenly changed last minute. But somehow the information didn't reach you. You're wondering - was it communicated at all? You need to work overtime, your boss is unhappy, your family is unhappy, you're unhappy and testers file another bug - again! Have you been there? Do you want to escape this cycle? Do you want a happier life? Then this workshop is for you! Come in and see how Spring Cloud Contract can come to your rescue. Learn how to use Spring Cloud Contract to verify with generated tests and stubs that REST-APIs and messages provided by producers finally conform to the contract they have with their consumers and... get home on time to see your happy family, make your boss happy and let testers file less bugs.

### Funktionale Programmierung - Next Level

Trainer: **Jan Hauer**, EXXETA AG

Lambdas und Streams gehören mittlerweile zu unseren Standard-Werkzeugen als Entwickler. Oft sind sie aber eher Fremdkörper in einer objektorientierten Anwendung. In diesem Training machen wir uns auf Entdeckungsreise in die unbekannte Welt des funktionalen und reaktivem Entwickeln. Beispielhaft entwickeln wir eine Spring Boot Anwendung. Dabei werden wir altbekannte Probleme wie Fehlerbehandlung oder Validierung funktional angehen. Auch andere Fragen wie die Modellierung von Zustandsautomaten zum Beispiel zur Menüführung, lässt sich natürlich funktional lösen. Nebenbei lüften wir das Geheimnis von Begriffen wie Monoide, funktionen höherer Ordnung und die Rätsel wie sich funktionaler Code eigentlich in bestehende Anwendungen integrieren lässt.

### Continuous Deployment on AWS: Make releasing the most boring part of your job

Trainer: **Steffen Grunwald & Dirk Fröhner**, Amazon Web Services

You can quickly build a prototype for your next brilliant idea with cloud services. In addition to business logic, mechanisms for continuous and efficient development in a team are important. This is a call for staging changes across multiple environments, rolling out new features without interruption and rolling back when things go south. Monitoring is essential to measure the quality of a new code iteration and it also allows you to quickly identify bottlenecks in a distributed system. This workshop introduces relevant AWS services and demonstrates how they are combined to reach these goals. Guided by hands-on labs you will build a continuous deployment pipeline step by step that makes development and operations a pleasure.

### Microservice: Architektur, Implementierung und Produktion

Trainer: **Eberhard Wolff**, INNOQ

In diesem Tutorial teilen wir eine Domäne mit Domain-driven Design in mehrere Microservices auf. Dann entscheiden wir über die Integrationstechnologie und bringen die Anwendung auf Kubernetes in Produktion. Schließlich kümmern wir uns um das Monitoring, Tracing und Logging. Dabei hilft der Service Mesh Istio. So zeigt das Tutorial an einem einfachen Beispiel, wie man eine konkrete Microservices-Anwendung von der Architektur über die Implementierung bis in Produktion bringt. Statt praktischer Übungen zeigt der Workshop konkrete Beispiele und Code für alle Technologien und Ansätze. Sie stehen zusammen mit den Folien und einer Dokumentation in drei Broschüren zum Download bereit und können ein Startpunkt für die Umsetzung eigener Microservices sein.

### DevSecOps für Praktiker

Trainer: **Sven Ruppert**, JFrog

In diesem Workshop werden wir basierend auf frei verfügbaren Werkzeugen ein Java Projekt aufsetzen in dem wir die Grundlagen des DevSecOps durchgehen werden. Hierzu zählen die theoretischen Grundlagen, praktische Übungen und BestPractices die Dir im Alltag die Arbeit erleichtern werden und Dir den Einstieg in das -Sec- von DevSecOps nahebringen.



# DIE GROSSE JAVA COMMUNITY KONFERENZ

## 27. - 30. Oktober 2020 - ONLINE!

### SESSIONPLAN - TAG 1

Dienstag, 27. Oktober 2020 • Agile Day • Testing Day • RapidClipse Day  
Beginn & Dauer: 9:00 bis 22:00 Uhr

|                | Main Track (EN)                                                                                                   | Main Track                                                                                          | Agile Day                                                                                         | Testing Day                                                                                                                     | RapidClipse Day                                                                                                                                                        |
|----------------|-------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 09:00<br>10:00 | <b>The evolution of Garbage Collection Algorithms in Java</b><br>Gaurav Sen                                       | <b>Eine herausragende App: Mehr als nur Technologie</b><br>Christian Seifert                        |                                                                                                   | <b>Consumer Driven Contracts mit OpenAPI</b><br>Christian Fischer                                                               | <b>RapidClipse - Status Quo</b><br>Markus Kett                                                                                                                         |
| 10:00<br>11:00 | <b>Not dead yet - Java on Desktop</b><br>Gerrit Grunwald                                                          | <b>Java &amp; Spring Boot im Container</b><br>Michael Vitz                                          | <b>Per Anhalter durch die Entwickler-Galaxis</b><br>Christian Seifert                             | <b>Ich packe meinen Testtoolkoffer und nehme mit... - Testwerkzeuge für den Entwickler</b><br>Sandra Parsick                    | <b>Web-Components &amp; Building UIs mit RapidClipse</b><br>Sebastian Späth                                                                                            |
| 11:00<br>12:00 | <b>NoSQL Endgame</b><br>Werner Keil                                                                               | <b>Fifty Shades of Java</b><br>Hendrik Ebbers                                                       | <b>Einführung eines Peer Feedback Konzepts in Entwicklungsteams</b><br>Kajetan Mischok            | <b>Testcontainers - The past, the present and the future</b><br>Kevin Wittek                                                    | <b>Praxisbericht: Sanktionsprüfung bei einem global führenden Industrieverseherer - Eine modulare Softwarelösung mit RapidClipse und Microstream</b><br>Matthias Wenzl |
| 12:00<br>13:00 | <b>Moving beyond request-reply: how smart APIs are different</b><br>Bernd Rücker                                  | <b>Java und die GraalVM</b><br>Wolfgang Weigend                                                     | <b>Teamkultur in der Spirale der Wertesysteme</b><br>Claudia Simsek-Graf, Stefanie Schuster       | <b>Einführung in Mutation Testing</b><br>Manfred Dreese                                                                         | <b>Automatisiertes Deployment mit RapidClipse</b><br>Christian Kümmel                                                                                                  |
| 13:00<br>14:00 | <b>Java In-Memory Database-Apps with MicroStream</b><br>Florian Habermann                                         | <b>Soziotechnische Architektur: Was treibt Softwarearchitekturen und Teams an</b><br>Martin Schmidt | <b>#WORKHACKS - Efficient way of working and teamwork</b><br>Simone Engelhard, Simon Qualmann     | <b>Spock vs Supermutanten: Spezifikationstesten trifft Mutationstesten</b><br>Johannes Dienst, Ralf D. Müller                   | <b>Validierung richtig gemacht</b><br>Sebastian Späth                                                                                                                  |
| 14:00<br>15:00 | <b>Introduction to data streaming</b><br>Nicolas Fränkel                                                          |                                                                                                     |                                                                                                   | <b>A Tale of Unit Tests, Code Coverage and Code Design</b><br>Stefan Waldmann                                                   | <b>Threats, Pushes and Broadcasts in RapidClipse</b><br>Christian Kümmel                                                                                               |
| 15:00<br>16:00 | <b>What Should Happen in 2021 with Java Backends and Web Frontends</b><br>Adam Bien                               |                                                                                                     |                                                                                                   |                                                                                                                                 |                                                                                                                                                                        |
| 16:00<br>17:00 | <b>Web Components without Frameworks: Productivity, Speed and Independence</b><br>Adam Bien                       | <b>Web-Anwendungen mit RapidClipse in Rekordzeit entwickeln</b><br>Markus Kett                      | <b>Mob Programming: Learn Together. Deliver Together!</b><br>Thomas Much                          | <b>Feature- and time-based software analysis - Balancing features, effort &amp; quality</b><br>Konstantin Sokolov, Egon Wuchner | <b>Responsive Design mit RapidClipse</b><br>Sebastian Späth                                                                                                            |
| 17:00<br>18:00 | <b>Fighting Diabetes: Calculating Insulin With Automated Carb Counting Using AI, ML</b><br>Todd Sharp             | <b>Frohes Neues - Die Neuerungen von Java 14 &amp; 15</b><br>Michael Vitz                           | <b>Hypothese: User Stories sind Hypothesen!?</b><br>Konstantin Diener                             | <b>Testing: Wo die Arbeit des Entwicklers endet und die des Testers beginnt</b><br>Zdenek Jonas                                 | <b>RapidClipse Fragen und Antworten</b><br>Christian Kümmel                                                                                                            |
| 18:00<br>19:00 | <b>Chaos Engineering with Spring Boot</b><br>Patrick Baumgartner                                                  |                                                                                                     | <b>Be the Leader of Your Own Career</b><br>Thodoris Bais                                          | <b>H2 database - Einführung und Praxisbeispiele</b><br>Julius Mischok                                                           |                                                                                                                                                                        |
| 19:00<br>20:00 | <b>Serverless Java Apps in the Cloud: MicroProfile, Quarkus, and Cloud Run</b><br>Mads Opheim, Rustam Mehmandarov |                                                                                                     | <b>Neugestaltung einer komplexen Webanwendung - Best practice und Learnings</b><br>Alexander Zels |                                                                                                                                 |                                                                                                                                                                        |
| 21:00<br>22:00 | <b>It's Easy! Contributing to Open Source</b><br>Cesar Hernandez                                                  |                                                                                                     |                                                                                                   |                                                                                                                                 |                                                                                                                                                                        |

Den aktuellen Sessionplan finden Sie unter [sessions.jcon.one](https://sessions.jcon.one)

# DIE GROSSE JAVA COMMUNITY KONFERENZ

## 27. - 30. Oktober 2020 - ONLINE!



### SESSIONPLAN - TAG 2

Mittwoch, 28. Oktober 2020 • Architecture Day • Cloud Development Day • Java-Cloud-Native Day  
Beginn & Dauer: 9:00 bis 22:00 Uhr

|                | Main Track (EN)                                                                                    | Main Track                                                                                                             | Architecture Day                                                                                                       | Cloud Development Day                                                                                    | Java-Cloud-Native Day                                                                                               |
|----------------|----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| 09:00<br>10:00 | <b>Standardizing Microservice Development</b><br>Priit Liivak                                      | <b>Projects Valhalla, Loom und GraalVM</b><br>Vadym Kazulkin                                                           | <b>Hysterisch Gewachsen – Wege zur stabilen API</b><br>Marco Schulz                                                    | <b>Cloud Jump Start für Java Entwickler</b><br>Benjamin Klatt, Sebastian Sirch                           | <b>Java und die GraalVM</b><br>Wolfgang Weigend                                                                     |
| 10:00<br>11:00 | <b>High-Performance Java-Cloud-Native Apps &amp; Microservices</b><br>Markus Kett                  |                                                                                                                        |                                                                                                                        |                                                                                                          |                                                                                                                     |
| 11:00<br>12:00 | <b>Adapting Java for the Serverless world</b><br>Vadym Kazulkin                                    | <b>Kubernetes Serverless mit Knative</b><br>Roland Huss                                                                | <b>Mit der Zeit gehen – Flexible Architekturen</b><br>Falk Sippach                                                     | <b>Infrastructure as Code in Action ! - Wie wir eine Cloud Plattform gebaut haben</b><br>Thorsten Jakoby | <b>Fast and Efficient Microservices with GraalVM</b><br>Oleg Šelajev                                                |
| 12:00<br>13:00 | <b>Advanced (elastic)search for your legacy application</b><br>David Pilato                        | <b>Konsent statt Konsens - Wie Teams schneller zu tragfähigen Entscheidungen kommen</b><br>Patrick Loka, Daniel Krämer | <b>CQRS und Event Sourcing unter Strom: Lessons Learned mit Spring und Axon</b><br>Frank Scheffler, Matthias Grünewald | <b>AWS Cloud-Umgebungen einfach und sicher verwalten</b><br>Dennis Kieselhorst                           | <b>Fly like a rocket with Helidon</b><br>Dmitry Kornilov                                                            |
| 13:00<br>14:00 | <b>Mobile Java Web Applications with MVC and OpenDDR</b><br>Werner Keil                            | <b>Pragmatische Builds mit Apache Maven</b><br>Richard Fichtner                                                        | <b>Effizient arbeiten mit Architecture Decision Records (ADRs)</b><br>Johannes Dienst                                  | <b>Fail Fast IoT: accelerated innovation with cloud-powered hardware</b><br>Piet van Dongen              | <b>Introduction to Micronaut 2.0</b><br>Graeme Rocher                                                               |
| 14:00<br>15:00 | <b>IDo you know these Libraries?</b><br>Roberto Cortez                                             | <b>It's a JDK-Jungle out there - JDK 15 and OpenJDK 15</b><br>Wolfgang Weigend                                         | <b>End-to-End Portable Microservices - with Eclipse MicroProfile</b><br>Ivar Grimstad                                  | <b>Java on Azure</b><br>Sandra Ahlgrimm                                                                  | <b>Kubernetes Native Applications with Quarkus</b><br>Markus Eisele                                                 |
| 15:00<br>16:00 | <b>Open Source at Amazon</b><br>Sascha Möllering                                                   |                                                                                                                        | <b>Java-Cloud-Native Microservices Battle - Ask the Gurus - 2 hours!</b><br>Graeme Rocher, Markus Eisele               |                                                                                                          |                                                                                                                     |
| 16:00<br>17:00 |                                                                                                    |                                                                                                                        | <b>Domain Driven Design - damit Dein Code Klartext spricht</b><br>Daniel Ludwig                                        | <b>Running Kubernetes on Workstations and Edge / IoT Devices</b><br>Mario-Leander Reimer                 | <b>Java-Cloud-Native Microservices Battle - Ask the Gurus - 2 hours!</b><br>Dmitry Kornilov, Florian Habermann      |
| 17:00<br>18:00 | <b>Programming Model for Microservices- MicroProfile and Spring with comparison</b><br>Emily Jiang | <b>Micronaut in Production</b><br>Carlos Barragan                                                                      | <b>Functional Centered Design</b><br>Jan Hauer                                                                         | <b>Creating a Kubernetes Operator in Java</b><br>Rudy De Busscher                                        | <b>Getting started with MicroStream - High-Performance Java-Native Persistence</b><br>Markus Kett, Christian Kümmel |
| 18:00<br>19:00 | <b>Supercharge your Code to Get Optimal Database Performance</b><br>Gerald Venzl                   | <b>Der Weg ist das Ziel: raus aus der Legacy-Falle hin zu mehr Qualität</b><br>Matthias Gutheil                        | <b>Es muss nicht immer Kubernetes sein</b><br>Stephan Kaps                                                             | <b>Developing Skills &amp; Actions for Voice Assistants</b><br>Piet van Dongen                           |                                                                                                                     |
| 19:00<br>20:00 | <b>The Battle of the IDEs (Pokemons)</b><br>Ko Turk                                                | <b>Ich möchte auch mal einen Vortrag halten - Wo fange ich bloß an?</b><br>Falk Sippach                                | <b>ArchUnit - Unit Testing Architecture and Design</b><br>Thomas Much                                                  | <b>Spring Security 5: Demystifying Cloud Identity</b><br>Ingo Düppe                                      |                                                                                                                     |
| 20:00<br>21:00 | <b>Practical DevSecOps for private and open source projects</b><br>Sven Ruppert                    | <b>Hitchhiker's Guide To Performance – Reloaded for Java 15</b><br>Ingo Düppe                                          | <b>Data management in a Microservices world</b><br>Gerald Venzl                                                        | <b>Via Java API Kubernetes Workloads dynamisch steuern</b><br>Matthias Haeussler, Corvin Schapöhler      |                                                                                                                     |
| 21:00<br>22:00 | <b>Coding video game as a hobby, possible and (kinda) easy with Java !</b><br>Simon Bedard         |                                                                                                                        |                                                                                                                        | <b>Scaling Java Applications Vertically and Horizontally with Jelastic PaaS</b><br>Ruslan Synytsky       |                                                                                                                     |

Den aktuellen Sessionplan finden Sie unter [sessions.jcon.one](https://sessions.jcon.one)

#JCON2020



# DIE GROSSE JAVA COMMUNITY KONFERENZ

## 27. - 30. Oktober 2020 - ONLINE!

### SESSIONPLAN - TAG 3

Donnerstag, 29. Oktober 2020 • DevOps Day • Web Development Day • CIO Day  
Beginn & Dauer: 9:00 bis 20:00 Uhr

|                | Main Track (EN)                                                                                                                   | Main Track                                                                                             | DevOps Day                                                                                                                     | Web Development Day                                                                                |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| 09:00<br>10:00 | <b>A Promising Future for using Native Libraries in the Java World</b><br>Babneet Singh, Tobi Ajila                               | <b>Praxisbericht von Sebastian über FRAPORT</b><br>Sebastian Späth                                     | <b>Alles nur CI-Theater?</b><br>Konstantin Diener                                                                              | <b>ECMAScript oder TypeScript, das ist hier die Frage</b><br>Thorsten Maier                        |
| 10:00<br>11:00 | <b>Live Coding 12-Factor App with MicroProfile and Kubernetes</b><br>Emily Jiang                                                  | <b>Java class file meets JVM</b><br>Christian Packenius                                                | <b>Efficient DevOps Tooling with Java and GraalVM</b><br>Mario-Leander Reimer                                                  | <b>Einführung in GraphQL – Das bessere REST?</b><br>Niclas Roßberger                               |
| 11:00<br>12:00 | <b>Securing eHealth and eGovernment with Java</b><br>Werner Keil, Thodoris Bais                                                   | <b>Serialization &amp; Object Graph Communication</b><br>Florian Habermann                             | <b>In den sicheren Hafen: Einstieg in Container Security</b><br>Stephan Kaps                                                   | <b>From Basic Auth to OAuth: Migrating security</b><br>Nikolai Neugebauer                          |
| 12:00<br>13:00 | <b>Two steps forward, one step backward - backward compatibility in Elasticsearch</b><br>Alexander Reelsen                        | <b>Let's Talk About Sex: Breeding Programs with Evolution</b><br>Heiko Spindler                        | <b>Zehn Jahre DevOps-Bewegung – ein Rück- und Ausblick</b><br>Konstantin Diener                                                | <b>Java goes Semantic Web</b><br>Christopher Olbertz                                               |
| 13:00<br>14:00 | <b>FaaS or not to FaaS? Visible and invisible benefits of the Serverless paradigm</b><br>Vadym Kazulkin                           | <b>Functional Programming in Practice with Kotlin and Scala</b><br>Benedikt Jerat, Florian Pfeleiderer | <b>From Code to Cloud - Full production pipeline using Java, Docker and Kubernetes</b><br>François Fernandès, Matthias Seifert | <b>MicroFrontends für MicroServices</b><br>Thomas Bröll                                            |
| 14:00<br>15:00 | <b>GameJams, or the art of fusing a week-end and coffee to create a video game</b><br>Simon Bedard                                | <b>No No Hibernate</b><br>Christian Kummel                                                             | <b>Terraform your environments from scratch</b><br>Pascal Euhus                                                                | <b>Per Anhalter durch JavaScript</b><br>Lisa Maria Moritz, Lucas Dohmen                            |
| 15:00<br>16:00 |                                                                                                                                   |                                                                                                        |                                                                                                                                |                                                                                                    |
| 16:00<br>17:00 | <b>Using WebComponents with Java on the server-side and how to harden the stack against known Vulnerabilities</b><br>Sven Ruppert | <b>RapidClipse Intro</b><br>Markus Kett                                                                | <b>DevSecOps aber sicher! kontinuierlich sicherer entwickeln</b><br>Frank Pientka                                              | <b>Schnittstellenarchitektur in Zeiten sich wandelnder Frontend-Technologien</b><br>Thorsten Maier |
| 17:00<br>18:00 | <b>GraphQL, Spring und Kotlin - a match made in Heaven</b><br>Ronny Bräunlich                                                     | <b>Machine Learning in Java: Die Visual Recognition API (JSR381)</b><br>Dennis Kieselhorst             | <b>Continuous Deployment Roadmap</b><br>Stephan Kaps                                                                           | <b>What's new in JHipster 7?</b><br>Frederik Hahne                                                 |
| 18:00<br>19:00 | <b>AdoptOpenJDK - What is that, actually?</b><br>Hendrik Ebberts, George Adams                                                    |                                                                                                        | <b>Monitoring your Spring Boot and Micronaut micro-services with Micrometer</b><br>Ko Turk                                     | <b>Reactive templates, TypeScript and typed endpoints in Vaadin</b><br>Viktor Lukashov             |
| 19:00<br>20:00 | <b>Easier Microservice Persistence With Micronaut Data</b><br>Todd Sharp                                                          |                                                                                                        |                                                                                                                                |                                                                                                    |

Den aktuellen Sessionplan finden Sie unter [sessions.jcon.one](https://sessions.jcon.one)

# DIE GROSSE JAVA COMMUNITY KONFERENZ

## 27. - 30. Oktober 2020 - ONLINE!



### AGENDA

| Dienstag, 27. Oktober                                   | Mittwoch, 28. Oktober                            | Donnerstag, 29. Oktober                        | Freitag, 30. Oktober                       |
|---------------------------------------------------------|--------------------------------------------------|------------------------------------------------|--------------------------------------------|
| <b>JCON-ONLINE 2020</b>                                 |                                                  |                                                |                                            |
| <b>Haupt-Konferenz Track 1 (EN)</b><br>9:00 - 22:00 Uhr |                                                  |                                                | <b>Workshop-Day</b><br>9:00 - 13:00 Uhr    |
| <b>Haupt-Konferenz Track 2</b><br>9:00 - 22:00 Uhr      |                                                  |                                                |                                            |
| <b>JCON Special Days</b>                                |                                                  |                                                |                                            |
| <b>Agile Day</b><br>9:00 - 22:00 Uhr                    | <b>Architecture Day</b><br>9:00 - 22:00 Uhr      | <b>DevOps Day</b><br>9:00 - 20:00 Uhr          | 10 Power Workshops<br>gleichzeitig - Live! |
| <b>Testing Day</b><br>9:00 - 22:00 Uhr                  | <b>Cloud Development Day</b><br>9:00 - 22:00 Uhr | <b>Web Development Day</b><br>9:00 - 20:00 Uhr |                                            |
| <b>Rapidclipse Day</b><br>9:00 - 22:00 Uhr              | <b>Java-Cloud-Native Day</b><br>9:00 - 22:00 Uhr | <b>CIO Day</b><br>9:00 - 20:00 Uhr             |                                            |

### TICKETS

| JUG-Ticket                                                            | Konferenz Ticket                                                      | Workshop Ticket                                                                           | CIO Day Ticket                                                          |
|-----------------------------------------------------------------------|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------|
| Di / Mi / Do<br>27. - 29. Oktober                                     | Di / Mi / Do<br>27. - 29. Oktober                                     | Fr<br>30. Oktober                                                                         | Do<br>29. Oktober                                                       |
| Teilnahme an allen Konferenz<br>Sessions an allen 3<br>Konferenztagen | Teilnahme an allen Konferenz<br>Sessions an allen 3<br>Konferenztagen | Auswahl aus allen Workshops<br>am 30. Oktober, Teilnahme an<br>einem Workshop Ihrer Wahl. | Speziell für IT-Entscheider.<br>Teilnahme an allen CIO Day<br>Sessions. |
| inklusive allen Special Days<br>(ausgenommen CIO Day).                | inklusive allen Special Days<br>(ausgenommen CIO Day).                | Begrenzte Teilnehmerzahl.                                                                 | Q&A mit Speakern.                                                       |
| Q&A mit Speakern.                                                     | Q&A mit Speakern.                                                     | Intensive Q&A mit Trainer.                                                                | Zugang zu allen<br>aufgezeichneten Sessions.                            |
| Zugang zu allen<br>aufgezeichneten Sessions.                          | Zugang zu allen<br>aufgezeichneten Sessions.                          |                                                                                           |                                                                         |
| Nur für JUG-Mitglieder.<br>Max. 500 Tickets verfügbar!                |                                                                       |                                                                                           |                                                                         |
| <b>0 €</b>                                                            | <b>39 €</b>                                                           | <b>199 €</b>                                                                              | <b>399 €</b>                                                            |

Alle aufgeführten Preise sind Nettopreise zzgl. 16% ges. MwSt.!

**Jetzt anmelden unter: [www.jcon.one](http://www.jcon.one)**

**Haben Sie Fragen zur JCON 2020? Wir beraten Sie gerne!**  
**Fon: +49 (0)800 -528 277 6 | E-Mail: [info@jcon.one](mailto:info@jcon.one)**

**#JCON2020**

# Die Magie der Lambdas

Mit der JDK-Version 8 wurde der Versuch gestartet, die objektorientierte Welt mit den Vorteilen der funktionalen Programmierung zu verbinden. Dazu wurden die Lambda-Ausdrücke eingeführt, ein mächtiges aber leider oft unterschätztes Werkzeug. Da Lambda-Ausdrücke jedoch ein integraler Bestandteil des JDK und vieler Frameworks sind, kommt man in der Java-Welt nur noch schwer um sie herum.

## Code = Daten

Die mit Java 8 eingeführten Lambda-Ausdrücke sollen die objektorientierte Programmiersprache Java mit den Vorteilen bekannter Konzepte aus der funktionalen Programmierung bereichern. Dazu muss erst einmal geklärt werden, dass Daten nicht notwendigerweise nur aus Zahlen, Bytes oder Zeichenketten bestehen, die in einem Programm herumgereicht werden, sondern dass auch der Code unseres Programms nichts anderes ist als Daten. Daten können an Methoden übergeben werden. Es ist also bekanntermaßen kein Problem Daten zum Beispiel in Form eines Integers oder eines komplexen Objekts an eine Methode weiterzugeben.

Doch auch Code kann an eine Methode übergeben werden. Vor Java 8 verwendete man dazu eine Referenz auf ein Objekt. Innerhalb der Methode können dann die Methoden des Objekts aufgerufen werden und somit ist ein Zugriff auf dessen Code möglich. Oftmals macht man es sich zunutze, dass die Klasse des Objekts eine bestimmte Methode implementiert, die beispielsweise durch ein Interface oder eine abstrakte Klasse vorgegeben

ist. Bekannt sind das Interface `Runnable`, in deren Methode `run` der Code für einen Thread verpackt wird, und das Interface `ActionListener` zur Definition einer Aktion, die z.B. beim Klick auf eine Schaltfläche ausgeführt werden soll. Diese Aktion wird in der vom Interface vorgegebenen Methode `actionPerformed` definiert.

## Übergabe von Code mittels anonymer Klassen

In diesem Abschnitt möchten wir uns kurz mit einer Variante der Übergabe von Code beschäftigen, die schon recht nahe an das funktionale Konzept herankommt: den anonymen Klassen. (Listing 1) zeigt ein einfaches Interface, das eine einzige Methode zur Verfügung stellt. Das folgende Konzept funktioniert auch mit Interfaces die mehr Methoden enthalten sowie mit abstrakten Klassen. Aber wir möchten uns auf den Fall eines Interfaces mit nur einer einzigen Methode beschränken, denn diese werden noch eine wichtige Rolle spielen.

### (Listing 1)

```
public interface Printer {  
    void print(int pageCount);  
}
```

Bekanntermaßen benötigt man eine implementierende Klasse, wenn man im Programm ein Interface oder eine abstrakte Klasse verwenden möchte, denn von beiden Konstrukten können keine Objekte erzeugt werden. Eine Klasse implementiert ein Interface durch Verwendung des Schlüsselwortes **implements** wie es in (Listing 2) zu sehen ist.

### (Listing 2)

```
public class LaserPrinter implements Printer {  
    @Override  
    public void print(int pageCount) {
```

## Autor:



Christopher Olbertz arbeitet als Lehrkraft an der Hochschule für Technik und Wirtschaft des Saarlandes. Dort betreut er Studenten vor allem in den Programmiersprachen Java und C/C++ und der UML. Sein besonderes Interesse gilt modernen Frameworks wie Spring, Hibernate, JavaServer Faces und Apache Tapestry.

Blog: <https://thinkingaboutprogramming.blogspot.com/>

```

        System.out.println("Ich drucke wie ein "
            + "Laserdrucker");
    }
}

```

Von dieser Klasse kann nun ein gewöhnliches Objekt mithilfe des Schlüsselworts **new** erzeugt werden. Sowohl Interface als auch die implementierende Klasse im Beispiel ist sehr klein. Wenn die Implementierung nur an einer einzigen Stelle benötigt wird, lohnt sich meistens der Aufwand einer zusätzlichen benannten Klasse nicht. Als Alternative können sogenannte anonyme Klassen verwendet werden, die im Gegensatz zu benannten Klassen keinen Namen haben (Listing 3).

(Listing 3)

```

public static void main(String [] args) {
    print100Pages(new Printer() {
        @Override
        public void print(int pageCount) {
            System.out.println("Ich drucke wie ein "
                + "Laserdrucker");
        }
    });
}

public static void print100Pages(Printer printer) {
    printer.print(100);
}

```

Die Methode **print** erwartet als Übergabe ein Objekt, dessen Klasse das Interface **Printer** implementiert. Dieser Methode können wir ein normales Objekt einer benannten implementierenden Klasse übergeben. Im Beispiel wird jedoch eine unbekannte Klasse übergeben: Die Klassendefinition und die Objekterzeugung finden an derselben Stelle statt, indem **new** mit einem Interface aufgerufen wird. In diesem Moment implementieren wir das Interface und erzeugen gleichzeitig ein Objekt dieser Klasse. Dieses Objekt wird an die Methode **print** übergeben. Da die implementierende Klasse keinen Namen hat, können wir sie nicht noch einmal verwenden.

Um auf den Anfang dieses Artikels zurückzukommen: Wir übergeben hier Code an eine Methode, was natürlich bei der Übergabe eines gewöhnlichen Objekts wie bereits beschrieben auch geschieht, aber an dieser Stelle ist diese Tatsache auf den ersten Blick ersichtlich. Lambda-Ausdrücke funktionieren prinzipiell ähnlich wie anonyme Klassen, sie gehen nur noch einen Schritt weiter.

## Einfache Lambda-Ausdrücke

Die Syntax von Lambda-Ausdrücken sieht folgendermaßen aus:

**Parameter -> Anweisungen**

Die Syntax ist ein wenig gewöhnungsbedürftig, aber wenn man sich einmal eingelese hat, sind Lambda-Ausdrücke in vielen Fällen erheblich besser lesbar als die oben beschriebenen

anonymen Klassen. Sehen wir uns einmal die Syntax genauer an: Ein Lambda-Ausdruck kann Parameter erhalten wie eine Methode. Dann werden die Anweisungen ausgeführt, welche die Parameter verarbeiten. Im Prinzip kann man sich einen Lambda-Ausdruck als eine anonyme Methode vorstellen, also als eine Methode ohne Namen. (Listing 4) zeigt einen einfachen Lambda-Ausdruck.

(Listing 4)

```

Printer laserprinter = (int pageCount) -> {
    System.out.println("Ich drucke wie ein "
        + "Laserdrucker: " + pageCount + " Seiten!");
};
laserprinter.print(100);

```

Der Lambda-Ausdruck erzeugt ein Printer-Objekt und erhält als Übergabe die Anzahl der Seiten. Was dann geschieht, ist auf den ersten Blick allerdings nicht ersichtlich. Und genau das macht Lambda-Ausdrücke im ersten Moment etwas schwer verständlich: Welche Methode wird implementiert?

## Funktionale Interfaces

An dieser Stelle kommt eine besondere Art von Interface ins Spiel: das funktionale Interface. Es definiert genau eine einzige Methode. Das Printer-Interface aus (Listing 1) ist ein solches funktionales Interface. Und genau dieses ist für Lambda-Ausdrücke enorm wichtig. Lambda-Ausdrücke sind so kompakt, weil der Compiler sich viele Informationen selbst erschließen kann. Dazu gehört auch die Information, welche Methode eines Interface eigentlich implementiert werden soll. Das ist für den Compiler sehr einfach, denn ein Lambda-Ausdruck funktioniert nur mit einem funktionalen Interface und ein funktionales Interface kann nur eine einzige Methode enthalten. Unser Lambda-Ausdruck aus (Listing 4) erzeugt ein Printer-Objekt. Printer definiert als funktionales Interface nur die Methode **print**. Und somit weiß der Compiler, dass sich der im Lambda-Ausdruck enthaltene Code auf diese Methode bezieht. Daher wird in der letzten Zeile in (Listing 4) der Lambda-Ausdruck aufgerufen.

## Lambda-Ausdrücke vereinfachen

Lambda-Ausdrücke können noch weiter vereinfacht werden. In (Listing 5) wird das funktionale Interface **MathFunction** mit der Methode **calculate** als Beispiel aufgeführt. Die Lambda-Ausdrücke werden in (Listing 6) Schritt für Schritt vereinfacht.

(Listing 5)

```

public interface MathFunction {
    int calculate(int a, int b);
}

```

## (Listing 6)

```
MathFunction sum1 = (int a, int b) -> {
    return a + b;
};

MathFunction sum2 = (a, b) -> {
    return a + b;
};

MathFunction sum3 = (a,b) -> a + b;
```

Der erste Ausdruck in (Listing 6) ist der lange Ausdruck mit allen Informationen. Bei dem zweiten wurden die Datentypen weggelassen. Die kann der Compiler sich erschließen, da das implementierte Interface schließlich nur eine einzige Methode hat. Besteht der Lambda-Ausdruck nur aus einer einzigen Zeile, können auch die geschweiften Klammern weggelassen werden, und in diesem Fall ist es auch nicht notwendig, die Rückgabe mit **return** einzuleiten. **return** wird vom Compiler implizit angenommen. So kommen wir zu dem kleinen und charmanten Ausdruck in der letzten Zeile von (Listing 6).

## (Listing 7)

```
MathFunction sum4 = new MathFunction() {

    @Override
    public int calculate(int a, int b) {
        return a + b;
    }
};
```

Zum Vergleich zeigt (Listing 7) dieselbe Implementierung noch einmal als anonyme Klasse. Hier fällt sofort auf, dass der Lambda-Ausdruck erheblich kürzer und – nach einer Eingewöhnungsphase – besser lesbar ist.

## Sammlung funktionaler Interfaces in java.util.functions

In dem Paket `java.util.functions` wurden bereits einige funktionale Interfaces vordefiniert, die an zahlreichen Stellen in Java eingesetzt werden können. Dazu ist es wichtig zu wissen, welche Interfaces es gibt, und ihre Einsatzmöglichkeiten zu verstehen. Die folgenden funktionalen Interfaces mit den entsprechenden Merkmalen sind bereits definiert:

- **Consumer**: keine Rückgabe
- **Function**: erzeugt eine Rückgabe
- **Operator**: gibt einen Wert vom Argumenttyp zurück
- **Supplier**: hat eine Rückgabe, aber kein Argument
- **Predicate**: hat eine Rückgabe vom Typ `Boolean`

## Das funktionale Interface Consumer

**Consumer** geben keinen Wert zurück, erhalten allerdings einen Übergabeparameter. Dazu enthält das funktionale Interface eine Methode **accept**, die in einem Lambda-Ausdruck implementiert werden muss. (Listing 8) zeigt ein Beispiel für die Verwendung von **Consumer**. Von dem Interface **Consumer** gibt es mehrere Spezialisierungen, wie zum Beispiel **IntConsumer**, der im Code zu sehen ist. Der Lambda-Ausdruck berechnet das Quadrat der übergebenen Ganzzahl und gibt dieses auf dem Bildschirm aus. Das Interface **Consumer** bietet noch zahlreiche weitere Methoden, die unter anderem eine Verkettung von Ausdrücken ermöglichen. Dies ist in den letzten Zeilen von (Listing 8) zu sehen. Hier wird ein zweiter Consumer **sumConsumer** definiert. Mithilfe der Methode **andThen()** werden die beiden Consumer nacheinander ausgeführt.

## (Listing 8)

```
IntConsumer squareConsumer =
    a -> System.out.println(a * a);
squareConsumer.accept(5);

IntConsumer sumConsumer = a -> System.out.println(a + a);
sumConsumer = sumConsumer.andThen(squareConsumer)
    .andThen(sumConsumer);
sumConsumer.accept(6);
```

Im Moment addieren und multiplizieren wir immer nur eine Zahl. Was aber tun wir, wenn wir zwei Zahlen in die Berechnung miteinbeziehen möchten? Im Gegensatz zum einfachen **IntConsumer** erhält die Implementierung **BiConsumer** zwei Übergabeparameter. In (Listing 9) werden die beiden Übergaben an den **BiConsumer** multipliziert.

## (Listing 9)

```
BiConsumer<Integer, Double> biConsumer =
    (a, b) -> System.out.println(a * b);
biConsumer.accept(4, 5.5);
```

## Das funktionale Interface Function

Die Methode **applyAsX** des funktionalen Interface **Function** erhält einen Übergabeparameter und gibt einen Wert zurück. Das **X** in dem Namen der Methode steht für den Rückgabety. (Listing 10) führt zwei Beispiele für das Interface **Function** auf. Als erstes wird ein Objekt des Typs **IntToLongFunction** deklariert. Dessen Methode **applyAsLong** erhält als Übergabe einen **int** Parameter und gibt einen **long** zurück. Das zweite Beispiel zeigt die Verwendung eines Objekts vom Typ **ToIntFunction**. Dieses kann mit einem Typ für die Übergabe typisiert werden, hier `String`. Zurückgegeben wird ein Wert vom Typ **int**.

(Listing 10)

```

IntToLongFunction squareFunction =
    (x) -> {
        return x * x;
    };
long square = squareFunction.applyAsLong(6);
System.out.println("Ergebnis x^2: " + square);

ToIntFunction<String> convertToIntFunction =
    (myString) -> Integer.parseInt(myString);
int value = convertToIntFunction.applyAsInt("123");
System.out.println("String als Integer: " + value);

```

## Das funktionale Interface Operator

Wenn Übergabeparameter und Rückgabe den gleichen Datentyp haben, kann anstelle von **Function** das Interface **Operator** als nützliche Abkürzung verwendet werden. Beispiele in (Listing 11). Zuerst wird ein **IntUnaryOperator** definiert. Dieser erhält eine Übergabe vom Typ **Int**, verdoppelt dieses und gibt es wieder als **Int** zurück. Im zweiten Beispiel wird ein **DoubleBinaryOperator** eingesetzt, der zwei Double-Parameter erwartet und einen Double-Wert zurückgibt.

(Listing 11)

```

IntUnaryOperator doubleOperator = x -> {return x * 2;};
System.out.println(doubleOperator.applyAsInt(4));

DoubleBinaryOperator multiplyOperator = (x,y) -> {
    return x * y;
};
System.out.println(multiplyOperator.applyAsDouble(4.5, 6.6));

```

## Das funktionale Interface Supplier

Das Interface **Supplier** erzeugt nur eine Rückgabe, erwartet aber keine Parameter. (Listing 12) zeigt mehrere Beispiele. Im ersten wird ein **IntSupplier** erzeugt, der eine Zufallszahl erzeugt und als **Int** zurückgibt, dabei aber keinen Parameter erhält. Das zweite berechnet mithilfe eines **BooleanSupplier**, ob ein fiktives Geschäft gerade geöffnet hat. Auch in diesem Fall kommen wir ohne eine Übergabe aus.

(Listing 12)

```

IntSupplier randomSupplier = () -> {
    return new Random().nextInt(100);
};
System.out.println("Zufallszahl: " + randomSupplier.getAsInt());

BooleanSupplier shopOpenSupplier = () -> {
    LocalDateTime now = LocalDateTime.now();
    return now.getHour() < 20 && now.getHour() > 8;
};
boolean isShopOpen = shopOpenSupplier.getAsBoolean();

```

## Das funktionale Interface Predicate

Das Interface **Predicate** schließlich erlaubt die Definition von Bedingungen und deren Verkettung, indem es einen Wert vom Typ **boolean** zurückgibt. (Listing 13) zeigt zwei Beispiele für die Verwendung von **Predicate**. Im ersten Beispiel wird ein **IntPredicate** erzeugt. Dieses erhält eine Bedingung als Lambda-Ausdruck und wendet diese Bedingung auf den **int** Wert an, welcher der Methode **test()** übergeben wird.

Das zweite Beispiel zeigt die Verkettung mehrerer **IntPredicate** Objekte. Das erste **IntPredicate** prüft, ob eine Zahl größer als 0 ist, das zweite, ob eine Zahl kleiner ist als **100**. Beide Bedingungen werden mithilfe der Methode **and** mittels logischem Und miteinander verknüpft. **Predicate** stellt auch Methoden zur Verneinung oder Verknüpfung mit einem logischen Oder zur Verfügung.

(Listing 13)

```

IntPredicate negativeNumberPredicate = x -> x < 0;
System.out.println("Ist 42 negativ? " +
    negativeNumberPredicate.test(42));
System.out.println("Ist -42 negativ? " + negativeNumberPredicate.
    test(-42));

IntPredicate predicateGreaterThan0 = x -> x > 0;
IntPredicate predicateLessThan100 = x -> x < 100;
IntPredicate combinedPredicate =

    predicateGreaterThan0.and(predicateLessThan100);
System.out.println("Liegt 55 zwischen 0 und 100? " +
    combinedPredicate.test(55));
System.out.println("Liegt 550 zwischen 0 und 100? " +
    combinedPredicate.test(550));

```

## Optionals

Die beschriebenen funktionalen Interfaces und somit auch die Lambda-Ausdrücke werden in Java 8 sehr häufig eingesetzt. So enthält das JDK ab der Version 8 zahlreiche Methoden, die mit diesen fünf funktionalen Interfaces arbeiten. Ein Beispiel ist die Klasse **Optional**. Diese bieten eine Alternative zu der unsehligen Rückgabe von Null-Werten, was immer die Gefahr von **NullPointerExceptions** nach sich zieht. Ein **Optional** ist eine Art Datencontainer, der entweder einen Wert enthalten oder leer sein kann. Ein leerer **Optional** ist eine Entsprechung zu Null mit dem Unterschied, dass keine **NullPointerException** ausgelöst werden kann. Zudem kann man mithilfe eines **Optionals** eine interessante Semantik in eine Methodendefinition hineinbringen. Wenn die Methode eine **Optional**-Rückgabe aufweist, bedeutet dies, dass die Rückgabe leer sein könnte und der Aufrufer entsprechende Maßnahmen ergreifen muss, um sich abzusichern. Handelt es sich nicht um eine **Optional**-Rückgabe, wird die Semantik transportiert, dass die Rückgabe niemals Null sein kann und dass somit auch keine besonderen Maßnahmen getroffen werden können.

Eine konsequente Benutzung von `Optional` kann ein Programm somit erheblich sicherer und besser lesbar machen, weil viele ineinander verschachtelte If-Anweisungen verschwinden können, die einfach nur Rückgaben von Methoden auf Null prüfen. `Optional` ist eine sehr komplexe Klasse und deswegen werden hier nur einige grundlegende Funktionen vorgestellt.

(Listing 14) zeigt einige Unit-Tests, mit denen die Funktionsweise von `Optional` überprüft werden kann. In der ersten Test-Methode wird ein leeres `Optional` Objekt erzeugt, in der zweiten ein `Optional`, der einen String enthält. Die Methode `isPresent` prüft, ob der `Optional` einen Wert enthält. Gibt die Methode `false` zurück, handelt es sich bei dem `Optional` um eine Entsprechung von Null mit dem Unterschied, dass keine `NullPointerException` geworfen werden kann. Die Methode `get` schließlich entnimmt dem `Optional` den enthaltenen Wert, wenn der `Optional` einen Wert enthält.

(Listing 14)

```
@Test
public void whenCreatesEmptyOptional_thenCorrect() {
    final Optional<String> empty = Optional.empty();
    assertFalse(empty.isPresent());
}

@Test
public void givenNonNull_whenCreatesOptional_thenCorrect() {
    final String expectedString = "SystemTechnikLabor";
    final Optional<String> optional = Optional.of(expectedString);
    assertTrue(optional.isPresent());
    final String actualString = optional.get();
    assertEquals(expectedString, actualString);
}
```

Interessant ist die Methode `ifPresent`, die in (Listing 15) präsentiert wird. `ifPresent` erwartet als Übergabe einen Consumer, der als Lambda-Ausdruck dargestellt werden kann. Im Beispiel enthält der Consumer einen String, an er die Endung `-if` anhängt und dies in der Variablen `actualName` speichert.

(Listing 15)

```
@Test
public void givenOptional_whenIfPresentWorks_thenCorrect() {
    final String name = "SystemTechnikLabor";
    String actualName = null;
    final String expectedName = "SystemTechnikLabor-if";
    final Optional<String> underTest = Optional.of(name);
    underTest.ifPresent(myString -> actualName = myString + "-if");
    assertEquals(expectedName, actualName);
}
```

Ein letztes Beispiel ist in (Listing 16) zu sehen. Hier wird noch zusätzlich die Methode `filter` eingesetzt. Sie erhält als Übergabe ein Predicate mit einer Bedingung. Nach dieser Bedingung wird der Inhalt des `Optionals` gefiltert. Im Beispiel wird zweimal gefiltert: nach dem Jahr **2016** und dem Jahr **2017**. `filter` prüft,

ob der im `Optional` enthaltene Wert jeweils durch **2016** oder **2017** ohne Rest teilbar ist. Und `isPresent` gibt zurück, ob ein Wert enthalten ist, der diese Bedingung erfüllt. Es geht also bei `isPresent` nicht mehr alleine darum, ob ein Wert enthalten ist.

(Listing 16)

```
@Test
public void whenOptionalFilterWorks_thenCorrect() {
    final Integer year = 2016;
    final Optional<Integer> yearOptional = Optional.of(year);
    final boolean is2016 = yearOptional.filter(y -> y == 2016).
        isPresent();
    assertTrue(is2016);
    final boolean is2017 = yearOptional.filter(y -> y == 2017).
        isPresent();
    assertFalse(is2017);
}
```

Optionals sind ein gutes Beispiel wie mächtig die Kombination aus Lambda-Ausdrücken und den in Java vordefinierten funktionalen Interfaces ist. Ein weiteres Beispiel ist die Stream-API, JAVAPRO 3-2019, ab Seite 22.

## Fazit:

Lambda-Ausdrücke sind zugegebenermaßen im ersten Moment etwas gewöhnungsbedürftig. Hat man sich jedoch einmal mit ihnen angefreundet, eröffnet sich ein breites Feld an Möglichkeiten, Code kompakter und besser lesbar zu gestalten. Sie können beliebig eigene funktionale Interfaces verwenden, aber Sie können auch auf die bereits in Java definierten zurückgreifen. Diese decken bereits ein breites Spektrum an Anwendungsmöglichkeiten ab und sind sehr allgemein formuliert. Sie werden bereits in vielen Methoden des JDK angewandt und deswegen sollten Java-Programmierer sie auch unbedingt kennen und verstehen. Dennoch gilt auch bei Lambda-Ausdrücken, dass die Lesbarkeit das wichtigste Gut eines Quelltextes ist. Lambda-Ausdrücke können den Quellcode leichter lesbar machen, wenn sie kompakt sind. Sind sie jedoch zu lang, kann die Lesbarkeit schnell darunter leiden. Daher empfiehlt es sich, auch in Lambda-Ausdrücken darauf zu achten, dass sie nicht zu lang werden, und gegebenenfalls einen Teil des enthaltenen Codes in private Methoden auszulagern.

## Quellen:

1 Michael Inden: Java 8 – Die Neuerungen: Lambdas, Streams, Date And Time API und JavaFX 8 im Überblick

#JAVAPRO #CoreJava #DevOps

# OpenJDK Amazon-Corretto

Anfang 2019 wurde Amazon-Corretto 8, eine kostenlose, plattformübergreifende Distribution von OpenJDK für Java 8 veröffentlicht. Kurz darauf kam Amazon-Corretto 11 für Java 11 hinzu. Amazon setzt intern Corretto bei Tausenden von produktiven Workloads ein und hat auf Basis dieser Erfahrung Verbesserungen hinsichtlich Sicherheit, Stabilität oder Leistung im OpenJDK-Projekt beigesteuert. In diesem Artikel betrachten wir genauer, welche spezifischen Vorteile Amazon-Corretto bietet.

## Die Installation von Amazon-Corretto

Amazon-Corretto 11 ist als Headless-Variante verfügbar. Diese Variante enthält keine Laufzeitabhängigkeiten, die in der Regel mit GUI-Anwendungen wie X11 und ALSA verbunden sind, und ist für serverseitige Workloads sinnvoll. Im folgenden werden die Installationsanweisungen für Amazon-Linux 2 gezeigt. Amazon-Linux 2 ist die nächste Generation von Amazon-Linux, einem Linux-Server-Betriebssystem von Amazon-Web-Services (AWS). Es bietet eine sichere, stabile und hochleistungsfähige Ausführungsumgebung zur Entwicklung und Ausführung von Cloud- und Unternehmensanwendungen.

Option 1:  
Installieren der Headless-Variante von Amazon-Corretto 11:

### (Listing 1)

```
sudo yum install java-11-amazon-corretto-headless
```

Option 2:  
Installieren der vollständigen Variante von Amazon-Corretto 11:

### (Listing 2)

```
sudo yum install java-11-amazon-corretto
```

`/usr/lib/jvm/java-11-amazon-corretto.<cpu_arch>` ist der Installationspfad für Amazon-Corretto. Für Debian-basierte und RPM-basierte Linux-Distributionen sind nachfolgende Installationsschritte notwendig: Im ersten Schritt muss die `.rpm-Datei`

für Linux von der Download-Seite heruntergeladen und dann mithilfe von `yum localinstall` installiert werden.

### (Listing 3)

```
sudo yum localinstall <rpm-Datei>
```

## Der Turbo für Crypto

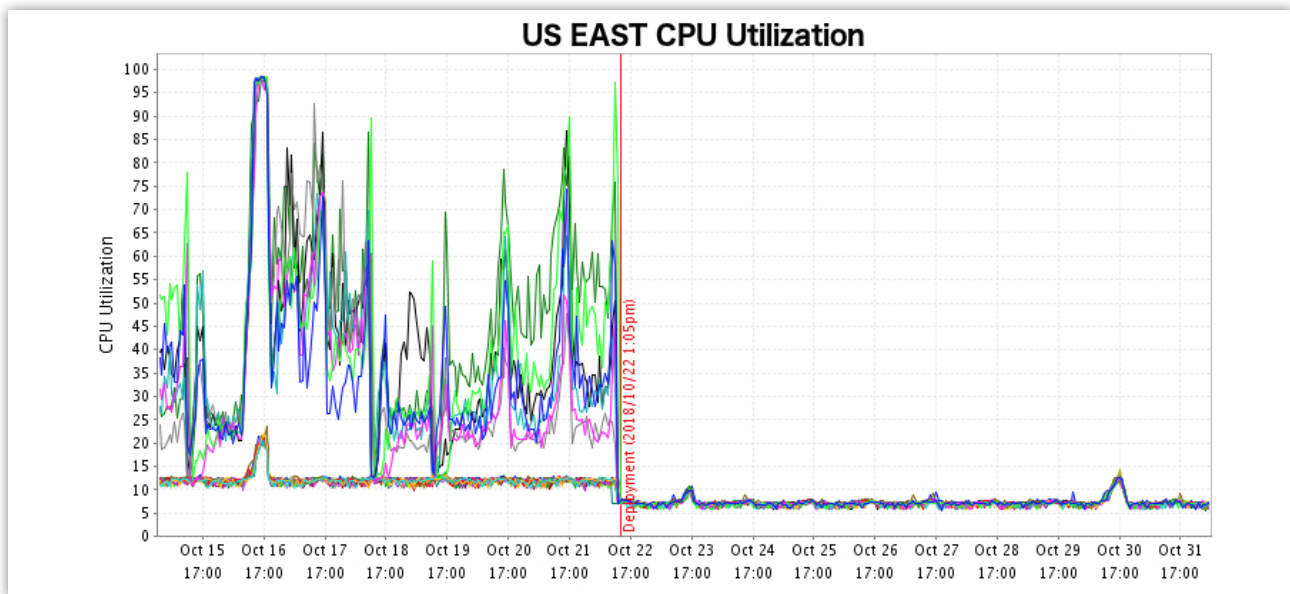
In der Vergangenheit war die Java-Kryptographie langsam und sehr CPU-intensiv. Kryptografische Operationen in Java führen zu einer erheblichen CPU-Auslastung (Utilization), Engpässen beim Durchsatz und erhöhten Betriebskosten. Um diese Probleme zu lösen, wurde der Amazon-Corretto-Crypto-Provider (ACCP) entwickelt, in dem Dutzende von kryptografischen Algorithmen

## Autor:

Sascha Möllering arbeitet seit mehr als 4 Jahren als Solutions-Architect und Solutions-Architect-Manager bei der Amazon Web Services EMEA SARL, Niederlassung Deutschland. Zuvor war er als Team-Lead und Software-Architect bei zanox in Berlin tätig. Er teilt seine Expertise mit dem Fokus auf die Bereiche Automation, Infrastructure-as-Code, Distributed-Computing, Container und JVM regelmässig in Beiträgen verschiedener IT-Magazine im deutschsprachigen Raum.



<https://aws.amazon.com/de/>  
@sascha242  
smoell@amazon.de



Effekt der Nutzung von ACCP auf einen AWS-Service (Abb. 1)

optimiert wurden. Die Leistungszuwächse variieren je nach Algorithmus und Arbeitsaufwand, aber in Benchmarks und in Produktion wurde eine Verbesserung des Durchsatzes von Services um mehr als 25% festgestellt. Im Allgemeinen profitieren langsamere Algorithmen und größere Datensätze am meisten von ACCP: der AES-GCM-Algorithmus mit ACCP ist über 28 Mal schneller als AES-GCM ohne ACCP. ACCP implementiert die Standard-Java-Cryptography-Architecture-Schnittstellen (JCA). Es ersetzt die Standard-Java-Kryptographie-Implementierungen durch die, welche libcrypto aus dem OpenSSL-Projekt bereitstellt. Als Beispiel wurde der Auslastungsgraphen eines AWS-Dienstes ausgewählt, der Transport-Layer-Security (TLS) auf Host-Ebene terminiert. In der Grafik (Abb. 1) ist deutlich der Unterschied vor und nach der Verwendung von ACCP zu sehen.

Die Nutzung von ACCP ist sehr einfach. Im ersten Schritt muss die entsprechende Abhängigkeit in der **pom.xml** eingetragen werden, falls Maven genutzt wird:

(Listing 4)

```
<dependency>
  <groupId>software.amazon.cryptools</groupId>
  <artifactId>AmazonCorrettoCryptoProvider</artifactId>
  <version>[1.0,2.0)</version>
  <classifier>linux-x86_64</classifier>
</dependency>
```

Beim Bootstrapping der Anwendung müssen die notwendigen Klassen genutzt werden, dazu sollte möglichst früh im Programmcode die folgende Zeile eingetragen werden:

(Listing 5)

```
com.amazon.corretto.crypto.provider.AmazonCorrettoCryptoProvider.
install()
```

Um zu überprüfen, ob ACCP auf dem System ordnungsgemäß funktioniert, kann eine der folgenden Aktionen durchgeführt werden.

Überprüfen, ob der Provider mit der höchsten Priorität tatsächlich ACCP ist:

(Listing 6)

```
if (Cipher.getInstance("AES/GCM/NoPadding").getProvider().
  getName().equals(AmazonCorrettoCryptoProvider.PROVIDER_NAME)) {
  // Successfully installed
}
```

Health-Check für ACCP durchführen:

(Listing 7)

```
if (AmazonCorrettoCryptoProvider.INSTANCE.getLoadingError() ==
  null && AmazonCorrettoCryptoProvider.INSTANCE.runSelfTests().
  equals(SelfTestStatus.PASSED)) {
  // Successfully installed
}
```

Sicherstellen, dass ACCP korrekt funktioniert und eine `RuntimeCryptoException` werfen, wenn dies nicht der Fall ist:

(Listing 8)

```
AmazonCorrettoCryptoProvider.INSTANCE.assertHealthy();
```

## Amazon-Corretto im Container

Amazon-Corretto kann natürlich auch im Kontext von Containern genutzt werden. Dafür existiert ein Docker-Base-Image,

das als Basis für eigene Docker-Images genutzt werden kann. Ein Dockerfile für eine Hello-World-Anwendung sieht folgendermaßen aus:

(Listing 9)

```
FROM amazoncorretto:11
RUN echo '$' \
public class Hello { \
public static void main(String[] args) { \
System.out.println("Welcome to Amazon Corretto!"); \
} \
}' > Hello.java
RUN javac Hello.java
CMD ["java", "Hello"]
```

Um dieses Image zu bauen, muss das Kommando **docker build -t hello-app** ausgeführt werden. Die Ausführung dieses Images als Docker-Container wird in (Listing 10) aufgezeigt. Darüber hinaus ist es auch möglich, ein Docker-Image auf Basis des GitHub-Repositories zu bauen (Listing 11).

(Listing 10)

```
$ docker run hello-app
Welcome to Amazon Corretto!
```

(Listing 11)

```
docker build -t amazon-corretto-11 https://raw.githubusercontent.com/corretto/corretto-docker/master/11/jre/alpine/Dockerfile
```

Nach dem erfolgreichen Bauen des Docker-Images befindet sich dieses lokal unter dem Namen **amazon-corretto-11**. Mit dem Kommando **docker run -it amazon-corretto-11** kann der Docker-Container gestartet werden. Das verwendete Dockerfile nutzt `jlink`, um nur die benötigten Pakete aus der JRE für eine eigene Runtime zu nutzen, was die JRE deutlich kompakter macht. In den meisten Fällen sind Pakete wie AWT oder Swing nicht notwendig. Zudem basiert das Base-Image auf Alpin-Linux, was sich nochmals sehr positiv auf die Größe auswirkt.

## Unterstützung für ARM64

Im Dezember 2019 hat AWS den M6g-Instanztyp veröffentlicht. Das „g“ steht hier für Graviton2, die nächste Generation eines auf Arm basierenden Chips, der von AWS entwickelt wurde und 64-Bit-Arm-Neoverse-N1-Kerne nutzt. Amazon-Corretto wird auch für die ARM64-Prozessorarchitektur für Amazon-Linux 2 angeboten<sup>1</sup>. Die Installation ist identisch mit dem Vorgehen für die x64-Architektur. Diese Prozessoren unterstützen 256-Bit-DRAM-Verschlüsselung im Dauerbetrieb.

Die M6g-Instanzen sind in 8 Größen mit 1, 2, 4, 8, 16, 32, 48 und 64 vCPUs oder als Bare-Metal-Instanzen erhältlich. Sie

unterstützen Konfigurationen mit bis zu 256 GB Speicher, 25 Gbit/s Netzwerkleistung und 19 Gbit/s EBS-Bandbreite. Diese Instanzen werden vom AWS-Nitro-System betrieben, einer Kombination aus dedizierter Hardware und einem leichtgewichtigen Hypervisor.

Typische Open-Source-Anwendungs-Stacks, die in der Regel auf x86-64-Architekturen eingesetzt werden, gehen durch die Migration auf Graviton2-basierte Instanzen mit einem bis zu 40% besserem Preis-Leistungs-Verhältnis im Vergleich zu ähnlich großen M5-Instanzen einher. M6g-Instanzen eignen sich gut für Workloads wie Application-Server, Gaming-Server, mittelgroße Datenbanken, Caching-Fleets, Web-Anwendungen und dergleichen.

## Immer auf dem neuesten Stand

In vielen Fällen ist die automatisierte Aktualisierung von Anwendungen eine notwendige Anforderung, weil die Software direkt auf Server ausgeliefert wird, die beispielsweise mit Chef oder Puppet provisioniert werden. Der Release-Schedule von Amazon-Corretto orientiert sich an den Long-Term-Support-Releases (LTS) des OpenJDK. Ab OpenJDK 11 erscheinen zukünftige LTS-Releases im Drei-Jahres-Turnus, so dass die nächste Version OpenJDK 17 im Jahre 2021 veröffentlicht werden wird. Die OpenJDK 11 Distribution Corretto 11 wird mindestens bis August 2024 vierteljährliche Updates erhalten.

Um die Kompatibilität mit der Java-SE-Plattform sicherzustellen, führt Amazon vor jedem Corretto-Release das Java-Technology-Compatibility-Kit (TCK) aus. Dabei handelt es sich um eine umfangreiche Testsuite, die von Oracle und Lizenznehmern verwendet wird, um kompatible Implementierungen der Java-Plattform sicherzustellen. Die Einschränkungen des OpenJDK gegenüber dem kommerziellen Oracle JDK bestehen natürlich auch bei Amazon-Corretto: Anwendungen, welche Funktionen nutzen, die in OpenJDK nicht verfügbar sind, wie zum Beispiel Java-Flight-Recorder, funktionieren nicht mit Corretto.

Aber nicht nur Major-Releases unterliegen einem Release-Plan: Zwischen den LTS-Releases erscheinen mindestens quartalsweise Updates. Diese Änderungen sind in der Corretto-Dokumentation<sup>2</sup> aufgeführt. Mit Amazon-Linux 2 sind zudem wichtige Updates auch im Amazon-Linux-Security-Center<sup>3</sup> aufgeführt. Die Aktualisierungen enthalten Performance-Verbesserungen und kritische Fehlerbehebungen für die Anwendungsentwicklung von Unternehmen. Aus diesem Grund sollten Änderungen kontrolliert, aber so zeitnah wie möglich ihren Weg in die Anwendung finden. Dazu existieren mehrere mögliche Vorgehensweisen, die im Folgenden kurz erläutert werden.

Beim initialen Start einer EC2-Instanz wird ein sogenanntes User-Data-Skript ausgeführt. Dadurch lassen sich Benutzerdaten an die Instanz übergeben, die man verwenden kann, um allgemeine

automatisierte Konfigurationsaufgaben durchzuführen und nach dem Start der Instanzskripte auszuführen. Ein generelles Muster um auf dem neuesten Stand zu bleiben, ist der Befehl in diesem Script am Beispiel von Amazon-Linux. Alle verfügbaren Updates werden damit zum frühestmöglichen Zeitpunkt installiert. Es besteht die Möglichkeit, nur sicherheitsrelevante Updates mit dem Zusatz **--security** zu installieren.

#### (Listing 12)

```
#!/bin/bash
yum update -y
```

Im Gegensatz zum eigenen Datacenter existieren EC2-Instanzen in vielen Fällen nicht für längere Zeit, denn sie können sehr einfach erzeugt und auch wieder terminiert werden. Eine Auto-Scaling-Gruppe enthält eine Sammlung von Amazon-EC2-Instanzen, die als logische Gruppierung zum Zweck der automatischen Skalierung und Verwaltung behandelt werden. In diesem Kontext werden Auto-Scaling-Groups eingesetzt: eine Auto-Scaling-Gruppe ermöglicht die Verwendung von Amazon-EC2-Auto-Scaling-Funktionen, wie beispielsweise Ersetzungen im Zuge von Zustandsprüfungen und Skalierungsrichtlinien.

Die zwei wichtigsten Funktionen des Amazon-EC2-Auto-Scaling-Services bestehen in der automatischen Skalierung und der Beibehaltung der Anzahl der Instanzen, falls beispielsweise bei einer EC2-Instanz Fehler auftreten sollten. Die Größe einer Auto-Scaling-Gruppe richtet sich nach der Anzahl der Instanzen, die als die gewünschte Kapazität definiert worden sind. Bei der Skalierung durch eine Auto-Scaling-Group wird somit das vorher definierte User-Data-Script ausgeführt, was zur Folge hat, dass die notwendigen Updates automatisiert eingespielt werden können.

Für einige Workloads sind kurze Startzeiten beim Auto-Scaling aufgrund eines Lastanstiegs entscheidend. Aus diesem Grund eignet sich auch die Erstellung eigener Amazon-Machine-Images (AMIs) mit Hilfe von AWS-Systems-Manager-Automation<sup>4</sup> oder zum Beispiel HashiCorp-Packer<sup>5</sup>. Ein AMI stellt die Informationen zur Verfügung, die zum Starten einer Instanz erforderlich sind.

Mit dem AWS-Systems-Manager und dessen vordefinierten Automation-Dokument **AWS-UpdateLinuxAmi** auf Basis eines Amazon-Linux-AMIs, ist die Erstellung eines AMIs denkbar einfach. Mit dem AWS-CLI wird die Automation wie in (Listing 13) gestartet. Das **PostUpdateScript** ist ein benutzerdefiniertes Skript mit Befehlen. Der AWS-Systems-Manager führt die Befehle nach der Aktualisierung aller Pakete aber noch vor der Erstellung des AMIs aus. Das Skript enthält lediglich die Zeilen aus (Listing 14).

#### (Listing 13)

```
aws ssm start-automation-execution \
  --document-name "AWS-UpdateLinuxAmi" \
```

```
--parameters \
SourceAmiId=ami-0caa0ee1474fb9a30,\
PostUpdateScript=<URL ohne Redirects, z.B. S3/ GitHubGist>
```

#### (Listing 14)

```
#!/bin/bash
yum install java-11-amazon-corretto-headless -y
```

Nach dem erfolgreichen Erstellen des AMI, enthält die Ausgabe der Automation die ID des gebauten AMIs. Wird danach die Auto-Scaling-Group über AWS-CloudFormation auf dieses AMI angepasst, übernimmt AWS-CloudFormation das Ausrollen des AMIs automatisch. Der Vorteil dieser Variante ist, dass der genaue Stand der installierten Software in Code dokumentiert ist. So ist auch ein Zurückrollen auf eine ältere Version im Fehlerfall möglich. Es ist ein durchaus übliches Vorgehen, nach einem erfolgreichen Build einer neuen Softwareversion ein aktuelles AMI mit diesem Release zu erstellen und in der Auto-Scaling-Group die einzelnen Instanzen auszutauschen. Dieses Konzept nennt sich **Immutable-Infrastructure**.

Die letzte Variante zum Update kommt ins Spiel, wenn die Instanzen nicht regelmäßig neu gestartet werden sollen. In den aktuellen Standard-AMIs für Windows- und Amazon-Linux ist der AWS-Systems-Manager-Agent vorinstalliert. Dieser übernimmt die Aktualisierung der installierten Pakete<sup>6</sup> anhand einer Patch-Baseline, die im AWS-Systems-Manager-Patch-Manager für die jeweilige Instanz konfiguriert wurde.

#### Fazit:

Amazon bietet mit der OpenJDK-Distribution Corretto ein kostenfreies, plattformübergreifendes und produktionsreifes Drop-in-Replacement für bestehende OpenJDK-Distributionen an. Corretto wird für eine Vielzahl von unterschiedlichen Betriebssystemen und zwei Prozessor-Architekturen angeboten. Durch die langfristige Unterstützung der LTS-Releases ist ein hohes Maß an Stabilität und Planungssicherheit gegeben. Dennoch sollten regelmäßig wichtige Sicherheitsaktualisierungen installiert und getestet werden.

#### Quellen:

- 1 <https://bit.ly/corretto-download-1>
- 2 <https://bit.ly/patches-c2>
- 3 <https://amzn.to/2ETfQu1>
- 4 <https://bit.ly/manager-a4>
- 5 <https://bit.ly/packer-c5>
- 6 <https://bit.ly/sysman-patch>

#JAVAPRO #CoreJava #API

# JSR-385 hätte Mars Orbiter retten können

1999 verlor die NASA den Mars-Climate-Orbiter im Wert von 125 Millionen US-Dollar, als er in die Umlaufbahn eintrat. Aufgrund einer Diskrepanz zwischen US-Customary und SI-Maßeinheiten in unterschiedlichen Teilsystemen kam das Raumschiff dem Planeten zu nahe, passierte die obere Atmosphäre und zerfiel. Dies war nicht der einzige Fall, in dem eine fehlerhafte Umrechnung zwischen Maßeinheiten katastrophale Folgen hatte, aber es war sicherlich einer der spektakulärsten und kostspieligsten.

## Immer wieder Unglücksfälle wegen Rechenfehler

- Patriot Missile: Hier war die Ursache eine ungenaue Berechnung der Zeit, die seit dem Start verstrichen ist, auf Grund von Arithmetik-Fehler im Computersystem.
- Ariane 5 Explosion: Es wurde eine Fließkommazahl umgewandelt, welche danach einen größeren Wert hatte, als durch den verfügbaren 16-Bit-signed Integer-Wert speicherbar gewesen ist.
- Beinahe Absturz einer US-Verkehrsmaschine auf dem Flug zwischen den USA und Kanada, in der Gegend der Großen Seen und Großraum Chicago. Hier wurden beim Betanken der Maschine die Volumeneinheiten US-Gallone und Britischer Gallone für Flüssigkeit und Treibstoffe verwechselt, was die tatsächlich verfügbare Menge an Kerosin erheblich reduzierte und dazu führte, dass die Maschine in der Warteschleife beim Landeanflug ohne ausreichend Treibstoff endete.
- Zahlreiche, manchmal auch tödliche Fehldosierungen von Medikamenten.

Das Maß einer körperlichen Quantität ist der Prozess des Schätzens des Verhältnisses seiner Größe zu einer anderen Quantität der gleichen Art, die wir als Einheit gegeben annehmen. Zum Beispiel 5 Meter ist eine Schätzung der Länge eines Gegenstandes im Verhältnis zu einer anderen Länge - das Messinstrument, das wir in diesem Beispielfall als die Standardmaßeinheit der Länge annehmen. Eine ähnliche Annäherung kann für finanzielle Quantitäten angenommen werden und voraussetzen, dass eine Währungseinheit richtig und allgemein gültig definiert wird.

Die Maßbezeichnung wird auch in einer loseren Art und Weise verwendet, um sich auf jeden möglichen Prozess des Zuweisens von Zahlen zu beziehen, um etwa die Richtung des Grads in einem Wesen darzustellen. Beispielsweise kennzeichnet die Mohs-Skala von Mineralhärten den Kratzerwiderstand der verschiedenen Mineralien durch die Fähigkeit eines härteren Materials, ein weicherer Material zu zerkratzen. Diese Skala gibt an, dass Quarz härter ist als Calcit, sagt aber nicht aus, um wie viel härter es ist. Es ist eine etwas willkürliche, ausschließliche Ordnungsskala. Die Quantitäten, die durch solche Skalen ausgedrückt werden, sind vergleichbar, jedoch nicht additiv. Eine Quantität kann auch von einem Zählimpuls festgestellt werden.

## Autor:

Werner Keil ist Java EE Consultant, BDD und Microservice Experte bei einem führenden Unternehmen im Logistik und IoT Bereich. Nach vergleichbarer Tätigkeit für Banken, Versicherungen und Unternehmen im Bereich Print und Automotive/Embedded, sowie als Agile Coach, Java Embedded und Eclipse RCP Experte bei einem Anbieter von Embedded und Realtime Systemen.



Webseite: <http://catmedia.us/>

Twitter: @wernerkeil

XING: [https://www.xing.com/profile/Werner\\_Keil](https://www.xing.com/profile/Werner_Keil)

LinkedIn: <https://www.linkedin.com/in/catmedia->

GitHub: <https://github.com/keilw>

Email: [werner@catmedia.us](mailto:werner@catmedia.us)

Ein Maß ist eine reale Zahl und ist nie wirklich genau, während eine Anzahl (Quantität) eine natürliche Zahl ist und genau sein kann. So können wir beispielsweise feststellen, dass sich genau 12 Eier in einem Karton befinden, indem man sie zählt. Das letzte gebrachte Beispiel birgt auch noch weitere meist regionale Besonderheiten. So werden bestimmte Anzahlen historisch bedingt gerne auch mit Synonymen verbunden bzw. durch solche oft ersetzt. Die Zahl 12 wird in der Regel als ein Dutzend bezeichnet. Auch Vielfache davon (2, 3, ... Dutzend) oder Teile davon wie etwa ein halbes Dutzend werden nicht nur umgangssprachlich, sondern auch bis heute in vielen Geschäftsbereichen verwendet. Auch die Maß (Bier) oder andere damit vergleichbare Einheiten, etwa das im britischen Raum und Einflussbereich verwendete Pint (und dessen Teile und Vielfache) sind populäre Beispiele. Während beispielsweise das Pint oder die Maß Bier sehr wohl im Sinn von Einheiten zu verstehen sind, kann eine Anzahl wie ein Dutzend genauso gut auch als Präfix, sprich als Multiplikator, zu verstehen sein und gebraucht werden. Und nicht nur in britischen Pubs, sondern auch in Münchner Biergärten und anderswo auf der Welt ist ein Dutzend Maß oder "a dozen pints" ein nicht wirklich so seltener Auftrag. Auf Präfix bzw. Multiplikator kommen wir in der Folge noch etwas genauer zu sprechen.

## Begriffe

Die Java-Klassen, die in der JSR-385-Spezifikation definiert werden, sind nicht nur auf Wissenschaft, Physik oder Mathematik allein beschränkt. Infolgedessen sollten die Bezeichnungen Quantität und die Maßeinheit in ihrer allgemeinen Begrifflichkeit angenommen werden, wie in der körperlichen Quantität oder in der körperlichen Maßeinheit, falls nicht anders angegeben. Diese Verallgemeinerung trifft zu, selbst wenn das Units-of-Measurement-API ursprünglich mit wissenschaftlichen oder physikalischen Quantitäten als primäres Ziel entworfen wurde. Die Bezeichnungen und Begriffe, die in der Spezifikation verwendet werden, sind:

- Dimension - Dimension ist einer der grundlegenden Aspekte von Quantitäten, einschließlich Länge, Masse, Zeit oder Kombinationen davon. Der Begriff des Maßes drückt eine Eigenschaft ohne irgendein Konzept der Größe aus. So können wir über Länge ohne irgendeine Implikation sprechen.
- Quantity (Quantität) - Jedwede Art quantitativer Eigenschaften oder Attribute einer Sache. Masse, Zeit, Abstand, Hitze, Längen- oder Breitengrad gehören zu den vertrauten Beispielen der quantitativen Eigenschaften.
  - ◊ Es ist keine Maßeinheit erforderlich, um eine Quantität auszudrücken, noch benötigt man eine bestimmte Arithmetik um dies zu tun. Alice kann die Masse ihrer Schuhe quantitativ bestimmen, indem sie diese in die Hand nimmt. Alice kann die Masse ihres linken Schuhs und die Masse ihres rechten Schuhs hinzufügen, indem sie beide in die Schale einer Waage legt. Quantitäten können messbar oder auch dimensionslos sein: die Zahl der Schuhe von Alice ist eine dimensionslose Quantität.
- ◊ Maßeinheiten sind erforderlich, um messbare Quantitäten in einem Computer, auf Papier, in einem Netzwerk, Gerät usw. darzustellen. Zwei oder mehr dieser Spezifikation unterstellen den Gebrauch einer bestimmten Quantität nicht als Standardmaßeinheit. Daher wird in der Regel SI (System-International) verwendet, damit das Standardsystem der Maßeinheiten den meisten Anwendungsfällen gerecht wird. Laut der JSR-385-API-Spezifikation sollen 7 Quantitäten commensurable sein, wenn sie die gleichen Maße haben, um sinnvoll miteinander verglichen werden zu können.
- Unit (Einheit) - Eine Einheit oder Maßeinheit ist eine Größe, die als Standard mit eindeutigem Wert angenommen wird. Maßeinheiten derselben Art können als Vergleichswert genommen werden. So entspricht beispielsweise ein Fuß (ft.) einer Quantität von 0,3048 Metern. Ungeachtet dessen, wie diese zu Stande kam, kann eine Maßeinheit als Quantität anderer Maßeinheiten der gleichen Art ausgedrückt werden.
  - ◊ Base-Unit (Basisinheit bzw. Grundlegende Maßeinheit)
    - Basisinheit ist eine wohl definierte Maßeinheit, die von ihrer Definition her als dimensional unabhängig von anderen grundlegenden Maßeinheiten angesehen wird. Das SI-System definiert nur 7 grundlegende Maßeinheiten (einschließlich Meter, Kilogramm und Sekunde), von denen alle weiteren SI-Maßeinheiten abgeleitet werden.
  - ◊ Derived-Unit (Abgeleitete Maßeinheit) - Derived-Unit ist eine Maßeinheit gebildet als Produkt von Ergebnissen grundlegender Maßeinheiten. Einige abgeleitete Maßeinheiten erhalten einen speziellen Namen und ein Symbol aus Komfortgründen zur einfacheren Handhabung. Im SI-System schließen abgeleitete Maßeinheiten mit speziellem Namen Hz, Newton und Volt ein.
  - ◊ System-of-Units - Ein Einheitensystem ist ein Satz grundlegender und abgeleiteter Maßeinheiten gewählt als Standards für das Spezifizieren von Maßen. Beispiele dafür sind das SI (Metrische) sowie das Imperiale (Englische) System.
  - ◊ Prefix (Präfix bzw. Multiplikator) - Ein Präfix ist ein führendes Wort, das an einer Maßeinheit angewendet werden kann, um eine dezimale Vervielfachung oder eine Verringerung der Maßeinheit zu bilden. Präfixe werden hauptsächlich im SI-System verwendet, das Kilo-, Centi-, Mili- etc. miteinschließt. Andere Kulturkreise und Regionen verwenden teilweise vom SI-System abweichende, ähnlich geartete Multiplikatorsysteme. Beispiele sind etwa die gesonderte Verwendung von Multiplikatoren wie etwa Lakh- (100.000) oder Crore- (10 Mio.) in Ländern wie Indien oder auch Teilen des Nahen Ostens. Man kann dies darauf zurückzuführen, dass unterschiedliche Währungssysteme existieren. Es lässt sich auch auf historische Zahlensysteme zurückführen, welche vom lateinischen oder arabischen Ursprung heutiger westlicher Systeme in einigen Bereichen abweichen oder in diesen in derselben Form fehlen. Dafür ist

wiederum der Gebrauch anderer, wie etwa ein Dutzend, dort nahezu unbekannt.

## Beispiel

Das Ergebnis des Messens der Wellenlänge einer einfarbigen Lichtquelle kann im Rahmen des SI-Systems mit folgenden Einheiten dargestellt werden:

$$\lambda = 698,2 \text{ nm}$$

wobei:

- $\lambda$  ist das Symbol der physikalischen Größe (Wellenlänge)  
nm ist das Symbol der physikalischen Einheit (Nanometer),  
worin:
  - ◊ n das Symbol des Verringerungsmultiplikators ist (Nano, was so viel bedeutet wie  $10^{-9}$ )
  - ◊ m das Symbol der Grundeinheit oder abgeleiteten Einheit, auf welche das Präfix anzuwenden ist.
- 698,2 ist der numerische Wert (Magnitude) der Wellenlänge in Nanometer. Die Dimension einer Länge wird in der Regel durch den Buchstaben L in dimensionaler Analysis bezeichnet. Nichtsdestotrotz erscheint diese Dimension nicht explizit, sondern kann stattdessen aus der Größe  $\lambda$  oder mithilfe der Einheit m (und dem entsprechenden Multiplikator) abgeleitet werden.

## Datentypen

### Maßeinheit

Die Einheit `javax.measure.Unit` repräsentiert eine bekannte Größe, welche sich als Standard zur Messung bewährt und durchgesetzt hat. So sind beispielsweise Kilometer und Watt Einheiten, wohingegen 5 Kilometer keine Einheit darstellt (jedenfalls nicht im Rahmen des internationalen Einheitssystems). Es ist lediglich eine Abmessung.

### Quantität

Ein grundlegendes Interface zur Darstellung messbarer Werte ist `javax.measure.Quantity`. Diese definiert und identifiziert all jene Größen, die gemessen werden können. Mittels Java-Generics können Angaben enthaltener Messwerte meist schon während des Kompilierens korrekt und ohne potenzielle Fehlerquellen gehandhabt werden. Die Quantität macht das Ergebnis eines Messvorgangs darstellbar in einer bekannten Einheit. Sie verbindet einen numerischen Wert mit einer Einheit als Ergebnis.

### Einheitssysteme

Ein Einheitssystem ist ein Satz von Einheiten, welche als Standard zur Normierung von Maßeinheiten gebräuchlich und bewährt ist. Wie etwa in vorhandenen Bereichen des JDK etablierte Systeme, so z.B. der Kalender in Form des gregorianischen Kalenders so weit wie möglich und sinnvoll abgebildet wurde, enthält JSR 385 als Grundlage die Einheiten des SI-Standards sowie die

wichtigsten und am meisten verwendeten Nicht-SI-Einheiten. Andere Systeme lassen sich als Erweiterung genauso einbinden. So sind etwa bereits erwähnte lokal etablierte Einheiten oder Multiplikatoren ebenso denkbar (und teilweise in Diskussion als Teil des API selbst) wie historische Einheiten. Darunter das Dutzend oder auch so etwas, wie der Marathon – 42,195 km (26 miles 385 yards). Ein Beispiel für Distanzen, die zwar auch selbst keine Einheit darstellen, wie 5m, aber sehr wohl in bestimmten Zusammenhängen praktisch als solche betrachtet werden können. Anders als die 5 Meter mögen 11 Meter in vergleichbarem Rahmen also durchaus ihren Platz in spezialisierten APIs haben. Solche Definitionen blieben dann aber Drittherstellern und Anwendern von JSR-385 überlassen.



Hierarchie von Einheiten und Einheitssystemen (Abb. 1)

Einheiten können auf folgende Art und Weise instanziiert werden:

- Mithilfe vordefinierter Konstanten: Der einfachste Weg dazu ist, auf Sammlungen von Konstanten zurückzugreifen, wie sie im Rahmen des JSR-385-API enthalten sind. Etwa in `SystemOfUnits` Implementationen wie SI. Die SI-Klasse stellt alle wichtigen Konstanten bereit, darunter etwa **METRE** bzw. **METER** (eine Basiseinheit) oder **WATT** (eine abgeleitete Einheit). **METRE** ist in ersterem Fall die britische Schreibweise, während **METER** in den USA und andernorts verwendet wird.
- Durch algebraische Operationen: Einheiten können dynamisch aus anderen vorhandenen Einheiten durch algebraische Operationen errechnet werden. All diese Operationen sind Methoden der `Unit`-Klasse. Diese sind vergleichbar mit etwa der `Number`-Basisklasse des JDK oder anderer Klassen zur Verwendung durch mathematische oder sonstige Berechnungen.
  - ◊ Argumente: Alle binären Operationen erwarten Argumente vom Typ `(Unit, Unit)` oder `(Unit, n)` wobei n entweder eine ganze Zahl (`Integer`) oder Fließkommazahl (`Float`) sein muss. Beispiel: m und s seien Instanzen der Klasse `Unit`. So sind zulässige Operationen etwa `m/s`, `m*s`, `m2` oder `m*1000`, wobei die verwendeten Methoden wie etwa `divide(Unit)`, `multiply(Unit)` oder `power(int)` der Instanz m in der `Unit`-Klasse definiert sind.
  - ◊ Ergebnis: Alle Ergebnisse, sowohl binär als auch unär (ein

Argument, d.h. im Fall von Methoden und Klassen die Klasse der Instanz selbst) sind jeweils wieder eine Instanz derselben Klasse `Unit`. Entweder eine neue Instanz oder eine bereits vorhandene mit entsprechendem Wert.

- Das Ergebnis hat dieselbe Dimension wie die verwendeten Argumente. Solche Operationen dienen meist dazu, neue Einheiten durch Umrechnung aus anderen gegebenen Einheiten zu erhalten. So kann etwa ein Zoll (Inch) als 2,54 cm ausgedrückt werden oder eben der Zentimeter Einheit multipliziert mit dem Faktor 2,54.
- Das Ergebnis hat eine andere Dimension als die verwendeten Argumente. Solche Berechnungen dienen der Ableitung neuer (abgeleiteter) Einheiten aus grundlegenden bereits vorhandenen Einheiten (zumeist Basis-einheiten). Werte einer solchen Ergebniseinheit sind in der Regel nicht wieder zurückkonvertierbar in die ursprüngliche Einheit. Ein Beispiel ist die Ableitung der Einheit Watt aus der Einheit Joule dividiert durch eine weitere Einheit.

| Ergebnis mit gleichbleibender Dimension                                                                                                                                                                                                                                                                                                                                                           | Ergebnis mit anderer Dimension                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Binäre Operationen: <ul style="list-style-type: none"> <li>• <code>divide</code> (double)</li> <li>• <code>divide</code> (Number)</li> <li>• <code>multiply</code> (double)</li> <li>• <code>multiply</code> (Number)</li> <li>• <code>shift</code> (double)</li> <li>• <code>shift</code> (Number)</li> <li>• <code>alternate</code> (String)</li> <li>• <code>prefix</code> (Prefix)</li> </ul> | Binäre Operationen: <ul style="list-style-type: none"> <li>• <code>pow</code> (int)</li> <li>• <code>root</code> (int)</li> <li>• <code>multiply</code> (Unit)</li> <li>• <code>divide</code> (Unit)</li> <li>• <code>transform</code> (UnitConverter)</li> </ul> Unäre Operationen: <ul style="list-style-type: none"> <li>• <code>inverse</code></li> </ul> |

Verwendbare, algebraische Operationen, die die `Unit`-Klasse zur Verfügung stellt (Abb. 2)

- Durch Interpretieren des Einheitssymbols (Parsing) - Dazu wird wie auch für die Ausgabe von Einheiten ein spezieller Formatter verwendet. Die `UnitFormat` Implementationen sind zumeist eine Subklasse von `java.text.Format` um mit existierenden Java-Bibliotheken und -Komponenten wie etwa `javax.swing.JFormattedTextField` kompatibel zu sein. Wie die meisten Format-Klassen enthält die Klasse eine Anzahl statischer `getInstance` Factory-Methoden, welche zusammen mit dem Rest des `UnitFormat`-API dafür sorgen, dass dessen Benutzer Instanzen der `Unit`-Klasse direkt aus einer `UnitFormat` Instanz erstellen können. Dieser Ansatz bietet, wenn auch nicht immer zur Kompilierzeit typensicher, so doch die größtmögliche Flexibilität. Somit entspricht dieser Ansatz auch der Verwendung statischer `AbstractUnit.parse(String)` Convenience-Methoden. Diese Art erkennt sämtliche SI-Einheiten sowie derzeit alle US-Customary Einheiten wie zum Beispiel in (Listing 1):

(Listing 1)

```
Unit<Length> meter = AbstractUnit.parse("m").asType(Length.class);
Unit<Length> feet  = AbstractUnit.parse("ft").asType(Length.class);
Unit<Energy> kilojoule = AbstractUnit.parse("kJ").asType(Energy.class);
Unit<Acceleration> mps2 = AbstractUnit.parse("m/s2").asType(Acceleration.class);
```

## Unterstützte Einheiten

Die Referenzimplementierung beinhaltet folgende Einheiten und -systeme:

- BIPM Units (wie ISO-1000), inklusive:
  - Base units
  - Dimensionless derived units
  - Derived units with special names
- SI prefixes from  $10^{-24}$  to  $10^{+24}$
- Binary prefixes

| Quantity type         | Unit constant            | Symbol           | Base |
|-----------------------|--------------------------|------------------|------|
| Acceleration          | METRES_PER_SQUARE_SECOND | m/s <sup>2</sup> |      |
| AmountOfSubstance     | MOLE                     | mol              | Y    |
| Angle                 | RADIAN                   | rad              |      |
| Area                  | SQUARE_METRE             | m <sup>2</sup>   |      |
| CatalyticActivity     | KATAL                    | kat              |      |
| Dimensionless         | ONE                      | 1                |      |
| ElectricCapacitance   | FARAD                    | F                |      |
| ElectricCharge        | COULOMB                  | C                |      |
| ElectricConductance   | SIEMENS                  | S                |      |
| ElectricCurrent       | AMPERE                   | A                | Y    |
| ElectricInductance    | HENRY                    | H                |      |
| ElectricPermittivity  | FARAD_PER_METRE          | F/m              |      |
| ElectricPotential     | VOLT                     | V                |      |
| ElectricResistance    | OHM                      | Ω                |      |
| Energy                | JOULE                    | J                |      |
| Force                 | NEWTON                   | N                |      |
| Frequency             | HERTZ                    | Hz               |      |
| Illuminance           | LUX                      | lx               |      |
| Length                | METRE                    | m                | Y    |
| LuminousFlux          | LUMEN                    | lm               |      |
| LuminousIntensity     | CANDELA                  | cd               | Y    |
| MagneticFieldStrength | AMPERE_PER_METRE         | A/m              |      |
| MagneticFlux          | WEBER                    | Wb               |      |
| MagneticFluxDensity   | TESLA                    | T                |      |
| Mass                  | KILOGRAM                 | kg               | Y    |

|                         |                   |                |   |
|-------------------------|-------------------|----------------|---|
| Power                   | WATT              | W              |   |
| Pressure                | PASCAL            | Pa             |   |
| RadiationDose Absorbed  | GRAY              | Gy             |   |
| RadiationDose Effective | SIEVERT           | Sv             |   |
| Radioactivity           | BECQUEREL         | Bq             |   |
| SolidAngle              | STERADIAN         | sr             |   |
| Speed                   | METRES_PER_SECOND | m/s            |   |
| Temperature             | KELVIN            | K              | Y |
| Time                    | SECOND            | s              | Y |
| Volume                  | CUBIC_METRE       | m <sup>3</sup> |   |

Von RI unterstützte Quantitäten und Einheiten (Abb. 3)

## Beispiele

(Listing 2) zeigt eine einfache Umrechnung von Maßeinheiten.

(Listing 2)

```
ServiceProvider provider = ServiceProvider.current();
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);
Unit<Length> sourceUnit = METRE;
Unit<Length> targetUnit = CENTI(METRE);
UnitConverter converter = sourceUnit.getConverterTo(targetUnit);
double length1 = 4.0;
double length2 = 6.0;
double result1 = converter.convert(length1);
double result2 = converter.convert(length2);
Quantity<Length> quantLength1 = lengthFactory.create(4.0, sourceUnit);
Quantity<Length> quantLength2 = lengthFactory.create(6.0, targetUnit);
Quantity<Length> quantResult = quantLength1.to(targetUnit);
```

In (Listing 3) sehen wir das Ergebnis einer Wärmebedarfsermittlung von Blockheizkraftwerken.

(Listing 3)

```
Quantity<Volume> volume = Quantities.getQuantity(1000, Units.LITRE);
System.out.println(String.format("Welche Arbeit ist nötig um %s Wasser von 50°C auf 70°C zu erhitzen?", volume));
Quantity<Temperature> deltaT = Quantities.getQuantity(20, Units.KELVIN, RELATIVE);
Quantity<Energy> energy = (Quantity<Energy>) volume.multiply(-deltaT);
Quantity<Energy> W = (energy.multiply(4200)).divide(3600);
Unit<Energy> wattHour = W.getUnit();
SimpleUnitFormat.getInstance().label(wattHour, "Wh");
Quantity<Energy> kWh = W.to(KILO(wattHour));
SimpleUnitFormat.getInstance().label(KILO(wattHour), "KWh");
System.out.println(kWh);
Quantity<Power> kiloWatt = Quantities.getQuantity(9.5,
```

```
KILO(Units.WATT));
Quantity<Time> time = (Quantity<Time>) kWh.divide(kiloWatt);
SimpleUnitFormat.getInstance().label(time.getUnit(), "h");
System.out.println(String.format("Wenn nur %s zur Verfügung stehen dauert es entsprechend länger.", kiloWatt));
System.out.println(String.format("Nämlich: %s", time));
```

In (Listing 4) wird der Body-Mass-Index (BMI) einer Person ermittelt.

(Listing 4)

```
ServiceProvider provider = ServiceProvider.current();
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);
QuantityFactory<Mass> massFactory = provider.getQuantityFactory(-Mass.class);
Quantity<Length> height = lengthFactory.create(1.83, METRE);
// for US measure units, the above line can be replaced by the following:
// Quantity<Length> heightUS = lengthFactory.create(6, FOOT);
// Quantity<Length> height = heightUS.to(METRE);
Quantity<Mass> mass = massFactory.create(85, KILOGRAM);
// for US measure units, same could be done from lb as above
Quantity<Area> squareHeight = height.multiply(height).asType(Area.class);
Quantity<?> bmi = mass.divide(squareHeight);
```

Und in (Listing 5) exemplarisch ein einfaches In-Flight Infotainment-System eines Flugzeuges, das die Zeit bis zur Landung in verschiedenen Einheitensystemen darstellen kann.

(Listing 5)

```
ServiceProvider provider = ServiceProvider.current();
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);
Airplane airplane = new Airplane("A380");
Quantity<Length> distance = lengthFactory.create(10427, KILO(-METRE));
Quantity<Speed> airplaneSpeed = airplane.getSpeed();

Quantity<Time> timeToDest = distance.divide(airplaneSpeed).asType(Time.class);

System.out.println(airplane + " flying " + airplaneSpeed);
System.out.println(airplane + " flying " + airplaneSpeed.to(MILES_PER_HOUR));
System.out.println("TTD: " + timeToDest.to(HOUR));
```

## Fazit:

Primitive Datentypen sind, wie der Name schon sagt, primitiv und sollten in erster Linie der Bildung komplexerer Datentypen dienen. Typsichere Behandlung von Werten im richtigen Kontext unter Wahrung der Semantik ist wichtig für die korrekte Auswertung von Daten und speziell im Realtime- und Embedded-Umfeld, etwa Automotive u.Ä., kann sie oft sogar über Leben und Tod entscheiden, oder zumindest über den Verlust oder Erhalt von Projekt und Budget.

#JAVAPRO #JavaEE #MicroProfile #Spring

# Java-EE vs. Spring für Microservices

Hat Jakarta-EE (ehem. Java-EE) immer noch den Ruf schwergewichtig zu sein und hat damit geringe Chancen für eine Microservice-Architektur in Betracht gezogen zu werden? Ist das Spring-Framework die bessere Antwort auf die Frage mit welcher Java-Technologie wir unsere Services bauen? Einmal mehr stehen sich die ewigen Duellanten Jakarta-EE und Spring gegenüber, dieses Mal jedoch mit dem Fokus auf Microservices.

## Autor:



Daniel Krämer ist bereits seit mehreren Jahren als Software Engineer für die anderScore GmbH tätig und begeistert sich für durchdachte Software-Architektur und strukturiertes Design. In seinen Kundenprojekten setzt er sich überwiegend mit individueller Java- und Web-Entwicklung sowie Fragestellungen rund um Migration und Integration (z.B. Microservices) auseinander. Er ist dafür bekannt, auch für komplexe Szenarien effektive und effiziente Strategien zur Test-Automatisierung zu finden. Daniel teilt seine Kenntnisse und Erfahrungen gerne mit anderen Entwicklern. Zu den Inhalten regelmäßig abgehaltener, öffentlicher Trainings zählen Themen wie Microservices, Spring, Apache Wicket und jQuery.

Webseite: <https://anderscore.com>

GitHub: <https://github.com/dkraemer-anderscore>

E-Mail: [daniel.kraemer@anderscore.com](mailto:daniel.kraemer@anderscore.com)



Maik Wolf ist bei der Kölner anderScore GmbH als Fullstack-Entwickler für Java Enterprise Projekte & Agile Coach im Kundeneinsatz. Durch seine Expertise in den Branchen Logistik, Managed-Hosting, Groß- & Einzelhandel und Dialogmarketing verfügt er über vielseitige und intensive Einsichten in verschiedene Softwarelandschaften und in ganz unterschiedliche Geschäftsanforderungen. Da sein zweites Herz für agile Arbeitsmethoden schlägt, integriert und optimiert er diese in seinen Projekten.

Webseite: <https://anderscore.com>

Blog: <https://maik-wolf.de>

Twitter: [@da\\_mwolf](https://twitter.com/da_mwolf)

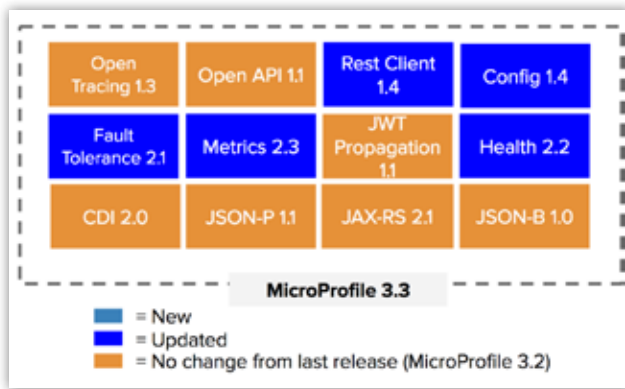
E-Mail: [maik.wolf@anderscore.com](mailto:maik.wolf@anderscore.com)

## Jakarta-EE vs. Spring - schon wieder?

Microservices mit Spring-Boot sind für viele Java-Entwickler zur Norm geworden und werfen einen erheblichen Schatten auf Jakarta-EE, wie Java-EE jetzt heißt. Durch seine beständige Evolution schafft es Spring immer wieder neue Entwicklungsansätze in die Welt zu tragen. Somit passt es sich den neuen Gegebenheiten und unseren sich ständig ändernden Anforderungen an. Spring-Boot hat sicherlich erheblich dazu beigetragen, dessen Grundlagen in der Blütezeit der Microservices entstanden sind. Gerade mit diesem Framework lassen sich schlanke Anwendungen schreiben, die in Self-Contained-JARs ausgeliefert werden. Diese und noch weitere Komponenten aus dem Spring-Ökosystem eignen sich nahezu perfekt, um den Kriterien eines Microservice gerecht zu werden.

Im Schatten steht nun Jakarta-EE mit seinen starren Spezifikationen und dem dazugehörigen, schwergewichtigen Application-Server. Doch genau diesem Nachteil haben sich viele Anbieter solcher Produkte bereits angenommen. Hervorgehoben sei dort die Entwicklung um Thorntail (zu Anfang Wildfly-Swarm). Nach dem Motto Just-enough-application-server konnte man sich bei diesem seine sog. Fractions zusammenstellen. Diese bestehen lediglich aus einzelnen Komponenten von Wildfly sowie Implementierungen anderer Frameworks. Somit war es tatsächlich möglich, einen leichtgewichtigen Application-Server in einer Self-Contained-JAR zu erzeugen. Der Weg zum schlanken Microservice wurde also mit diesen und anderen Frameworks bereits geebnet.

Zu dem neuen Hype um JEE-Microservices hat unter anderem auch das im Frühjahr 2019 erschienene Framework Quarkus beigetragen. Unter dem Slogan „Supersonic-Subatomic-Java“ verspricht das von Red Hat gesponserte Open-Source-Projekt „Container-First, Unifies Imperative and Reactive,



Bestandteile von MicroProfile 3.3<sup>3</sup> (Abb. 1)

Developer Joy & Best-of-Breed Libraries and Standards“. Quarkus ist somit die konsequente Evolution von Just-enough-application-server hin zu Container-First. Durch seine geringe Start-up-Zeit im Millisekundenbereich und der geringen Größe der Self-Contained-JARs, eignet es sich perfekt zur automatischen Skalierung in Container-getriebenen Orchestrierungs-Tools, wie zum Beispiel Kubernetes. Nun werden auch hier sog. Extensions verwendet, die gängige Funktionalitäten zum Betreiben des Microservice mitbringen. Auch dadurch fühlt sich Quarkus an vielen Stellen an wie Thorntail, was daran liegen mag, dass Design und Implementierungsideen von Thorntail 4.x Proof-of-Concept in Quarkus eingeflossen sind<sup>1</sup>. Mit diesen Extensions werden die beworbenen „Best-of-Breed Libraries and Standards“ eingebracht. Unter anderem lässt sich hier die Spezifikation für Microservices aus dem Projekt Eclipse-MicroProfile verwenden. MicroProfile seinerseits hat sich der Herausforderung angenommen, einen Standard für die Entwicklung von Microservices auf Basis von Java zu definieren. Wie schon Java-EE ist auch MicroProfile eine Sammlung von Spezifikationen, mit welcher die Entwicklung von Microservices zu einer einfachen Aufgabe werden soll. Die Sammlung beinhaltet bewährte JEE-Technologien wie CDI, JAX-RS, JSON-B & JSON-P, aber auch neue, auf Microservices zugeschnittene Spezifikationen wie Health-Check, Metrics, Config ect. Siehe JAVAPRO 2-2019, „MicroProfile für Microservices mit Java-EE“, von Alexander Ryndin<sup>2</sup>. Mit diesen Erneuerungen kann Java-EE wieder in den Ring mit Spring steigen und sich im Kampf um das bessere Framework für Microservices messen.

## Vergleichskriterien

Um die beiden Technologien miteinander vergleichen zu können, bedarf es zunächst geeigneter Kriterien. Was aber macht einen klassischen Microservice aus? Einerseits handelt es sich per Definition um ein unabhängig lauffähiges und deploybares Artefakt. Mithilfe externer Konfiguration passt es sich dynamisch an seine jeweilige Laufzeitumgebung an, wie zum Beispiel einem Kubernetes-Cluster. Da ein Service bekanntlich selten allein kommt, benötigt er weiterhin auch Schnittstellen zur Außenwelt. Aufgrund hoher Interoperabilität werden diese in der Praxis gerne als REST-Endpoints realisiert. Um jederzeit über

die korrekte Funktion und Verwendung des Service im Bilde zu sein, werden ferner ein Health-Check sowie möglichst einfach abrufbare Metriken benötigt. Natürlich handelt es sich bei den genannten Kriterien nur um eine subjektive Zusammenstellung, die keinen Anspruch auf Vollständigkeit oder Allgemeingültigkeit erhebt. Dennoch bietet sie eine erste Entscheidungsgrundlage zur Auswahl einer passenden Technologie. Nach Festlegung der Kriterien kann die Gegenüberstellung beginnen.

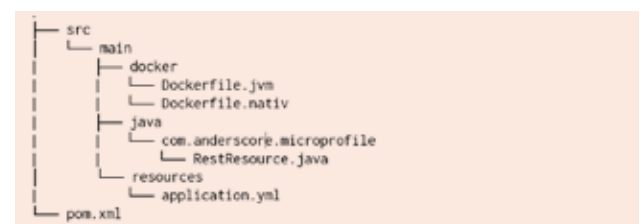
Spring-Boot sieht sich selbst als Wegbereiter für unabhängige, produktionsreife Anwendungen, die sich einfach starten lassen. Diesem Anspruch folgend bestehen die mittels hausinternem Initializr erstellten Projekte im Wesentlichen nur aus einer startbaren Application-Klasse nebst zugehöriger Konfigurationsdatei (Listing 1).

(Listing 1)



Dank Spring-Boot-Maven-Plugin genügt ein simples **mvn spring-boot:run** auf der Kommandozeile, um den Microservice mitsamt eingebettetem Web-Server (meist ein Tomcat) auf Port 8080 hochzufahren. Nicht viel anders verhält es sich bei Projekten auf Basis von MicroProfile bzw. Quarkus, welche sich ihrerseits wahlweise über den MicroProfile- bzw. Quarkus-Starter oder mittels Kommandozeile unter Verwendung des Quarkus-Maven-Plugins **mvn quarkus:create** generieren lassen (Listing 2).

(Listing 2)

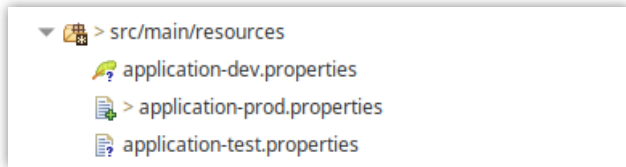


Ins Auge fallen im Vergleich lediglich zwei vorbereitete Docker-Files, welche der Kubernetes-Native-Philosophie des Quarkus-Projektes Rechnung tragen. Bevor über das Quarkus-Maven-Plugin auch hier ein eingebetteter Web-Server gestartet wird, erfolgt optional eine native Kompilierung (Listing 3).

(Listing 3)

```
$ mvn package -Pnative && ./target/microprofile-quarkus
$ mvn quarkus:dev
```

Variable Konfigurationswerte lassen sich bei Spring bequem in umgebungsspezifische Properties- oder YAML-Dateien auslagern und an nahezu beliebiger Stelle injizieren (**Abb. 2**). Profiles ermöglichen überdies auch die gezielte Aktivierung oder Deaktivierung einzelner Beans (**Listing 4**).

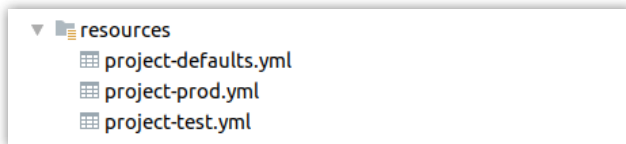


(Abb. 2)

(Listing 4)

```
mvn spring-boot:run -Dspring.profiles.active=test
```

Wenig Anlass zur Überraschung bietet der anschließende Blick auf Quarkus, weil die MicroProfile-Config-Spezifikation diesem bewährten Prinzip folgt:



(Abb. 3)

(Listing 5)

```
$ java -jar myapp-quarkus.jar -Stest
```

RESTful Webservices werden bei Spring (Boot) als **@RestController** mit zugehörigen Methoden definiert und zur Laufzeit über einen Component-Scan des Application-Contextes erfasst (**Listing 6**).

(Listing 6)

```
@RestController
@RequestMapping("/tasks")
public class TaskController {

    @Autowired
    private TaskRepository taskRepository;

    @PostMapping
    @ResponseStatus(CREATED)
    public void createTask(@RequestBody Task task) {
        taskRepository.save(task);
    }

    @GetMapping("/{id}")
    public Task findTask(@PathVariable long id) {
        return taskRepository.findById(id)
            .orElseThrow(() -> new NotFoundException(id));
    }

    // ...
}
```

Seit jeher ähneln die verwendeten Annotationen jenen des JAX-RS-Standards, welcher auch MicroProfile und damit Quarkus zugrunde liegt (**Listing 7**).

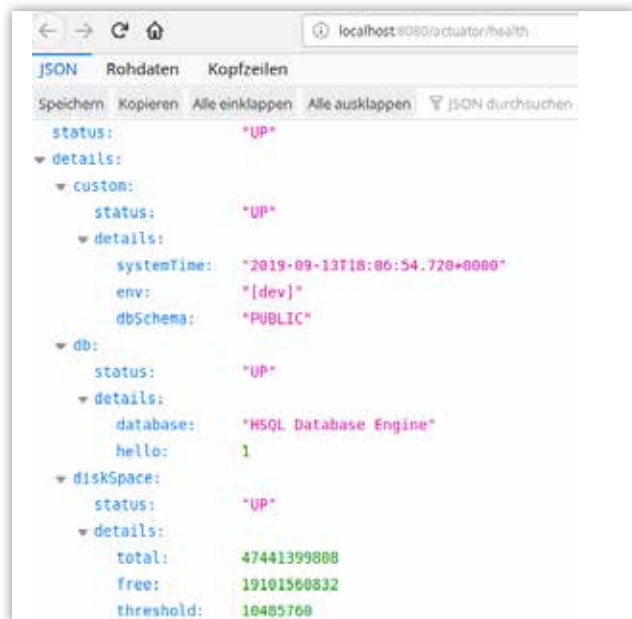
(Listing 7)

```
@Path("/tasks")
@Produces(MediaType.APPLICATION_JSON)
public interface TaskResource {

    @POST
    @Path("/{id}")
    void createTask(Task task);

    @GET
    @Path("/{id}")
    Task findTask(
        @PathParam("id") Long id
    );
    // ...
}
```

Um zur Laufzeit mehr über seinen Microservice zu erfahren, stellt Spring das Actuator-Projekt bereit. Dieses lässt sich leicht als Dependency über einen Starter einbinden und stellt daraufhin eine Vielzahl nützlicher Endpunkte über REST und JMX bereit. Ein Blick auf den Health-Endpunkt genügt bereits, um wichtige Lebensfunktionen zu messen (**Abb. 4**).



(Abb. 4)

Sind diese nicht aussagekräftig genug, lassen sich beliebig komplexe **HealthIndicators** selbst definieren und als Bean in den Context einbinden (**Listing 8**).

(Listing 8)

```
@Component
public class CustomHealthIndicator extends AbstractHealthIndicator {
    // ...
}
```

```
@Autowired
private Environment environment;

@Autowired
private DataSource dataSource;

@Override
protected void doHealthCheck(Builder builder) throws Exception {
    builder.up()
        .withDetail("systemTime", new Date())
        .withDetail("env",
            Arrays.toString(environment.getActiveProfiles()))
        .withDetail("dbSchema",
            dataSource.getConnection().getSchema());
}
```

MicroProfile verfügt mit der Health-Spezifikation über ein vergleichbares Konzept, unterscheidet jedoch grundsätzlich zwischen Liveness (Service läuft) und Readiness (Service ist bereit). Entsprechende Endpunkte stellen auch hier alle relevanten Informationen für einen Health-Check zur Verfügung (Listing 9). Implementierungsabhängige Kennzahlen werden hingegen unter **/vendor** angeboten (Listing 10).

(Listing 9)

```
{
  "gc.total;name=G1 Young Generation1": 15,
  "gc.total;name=G1 Old Generation1": 0,
  "cpu.systemLoadAverage": 1.42,
  [...]
}
```

(Listing 10)

```
{
  "bufferPool.usedMemory;name=mapped1": 0,
  "bufferPool.usedMemory;name=direct1": 368640,
  "memoryPool.usage.max;name=CodeHeap 'profiled nmethods'1":
  16832896,
  [...]
}
```

Ferner bietet auch MicroProfile die Möglichkeit, selbst definierte Metriken zu erheben und unter einem eigenen Endpoint (relativ zu **/application**) zur Verfügung zu stellen. Sollen etwa die Aufrufe einer bestimmten fachlichen oder technischen Methode gezählt werden, lässt sich dies leicht mithilfe der **@Counted** Annotation bewerkstelligen (Listing 11). Mittels **@Gauge** ist auch eine dynamische Berechnung zum Zeitpunkt des Abrufs möglich (Listing 12).

(Listing 11)

```
@Counted(unit = MetricUnits.NONE,
    name = "tasksCreated",
    absolute = true,
```

```
    displayName = "Created items",
    description = "Metrics to show how many times createTask
    method was called.",
    tags = {"tasks=create"}
)
@POST
@Path("/{id}")
public void createTask(Task task) {
    //...
}
```

(Listing 12)

```
@Inject
@ConfigProperty(name="defaultEstimation")
private Long defaultEstimation;

@Gauge(unit = "Hour", name = "defaultEstimation", absolute =
true)
public Long getDefaultEstimation() {
    return defaultEstimation;
}
```

Zur Bestimmung der (zeitlichen) Lastverteilung kann ferner die Annotation **@Metered** herangezogen werden (Listing 13). Zu guter Letzt bietet **@Timed** von Haus aus sogar ein wenig Arithmetik (Listing 14).

(Listing 13)

```
@Metered(
    name = "findTask",
    unit = MetricUnits.MINUTES,
    description = "Metrics to monitor findTask method.",
    absolute = true
)
@GET
@Path("/{id}")
public Task findTask(
    @PathParam("id") Long id
) {
    ....
};
```

(Listing 14)

```
@Metered(
    name = "findTask",
    unit = MetricUnits.MINUTES,
    description = "Metrics to monitor findTask method.",
    absolute = true
)
@GET
@Path("/{id}")
public Task findTask(
    @PathParam("id") Long id
) {
    ....
};
```

## Fazit technologisch:

Welche Technologie eignet sich denn nun besser für die Realisierung eines Microservice? Wie schon so oft präsentiert sich Spring (Boot) einmal mehr als Vorreiter und darf nach einigen Jahren intensiver Erprobung in der Praxis guten Gewissens als absolut produktionsstauglich bezeichnet werden. Spätestens seit Einführung der Auto-Configuration hält sich der erforderliche Glue-Code zum Aufsetzen eines Microservice in sehr überschaubaren Grenzen, sodass der Entwickler sich nun endgültig auf die Implementierung der Fachlogik konzentrieren kann. Insgesamt präsentiert sich das Spring-Ökosystem heutzutage als derart mächtig, dass sich schwerlich ein Bereich findet, welcher noch nicht durch ein entsprechendes Projekt nebst Starter abgedeckt wird. Schafft der Einsatz solcher Technologie im konkreten Projekt einen Mehrwert, kann dies bereits ein Entscheidungskriterium sein. An dieser Stelle lässt sich insbesondere Spring-Data heranziehen, welches die Implementierung eigener Logik für Datenbankzugriffe nahezu überflüssig macht. Eine vergleichbare Funktionalität bietet MicroProfile aktuell noch nicht an. Losgelöst davon existiert bei Quarkus mit Panache zwar ein ähnliches Konzept, das aber mit einigen Restriktionen einhergeht, wie zum Beispiel Public-Attribute in den Entities. Seit jeher bestand eine Maxime des Spring-Frameworks auch darin das Rad nicht neu zu erfinden, sondern stattdessen die Einbindung anderer, am Markt etablierter Libraries und Frameworks (Hibernate, Netflix OSS etc.) zu vereinfachen. Ist die Integration spezieller Technologien angedacht, sollte deren Unterstützung ebenfalls berücksichtigt werden. Auf der anderen Seite reduziert die Verwendung vordefinierter Starter- und Parent-POMs zwar einen eigenen Konfigurationsaufwand, erzeugt aber auch starke Abhängigkeiten, die nicht außer Acht gelassen werden sollten.

Dass totgesagte länger leben, beweist das Trio aus Jakarta-EE, MicroProfile und Quarkus eindrucksvoll. Die meisten enthaltenen Enterprise-Standards (CDI, JAX-RS, etc.) sind nicht nur hochgradig praxiserprobt, sondern auch vollständig durchspezifiziert. Vielen Entwicklern dürfte ein Großteil eines solchen Frameworks daher allzu bekannt vorkommen, sodass leicht auf bereits gesammeltes Wissen und Know-How zurückgegriffen werden kann. Vervollständigt werden diese Standards durch neue, vollständig am Bedarf moderner Microservices orientierter Spezifikationen wie etwa Health, Metrics oder Fault-Tolerance. Die Einbindung und Konfiguration von Third-Party-Libraries ist nicht mehr erforderlich, weil Implementierungen wie Quarkus diese bereits über entsprechende Extensions mitliefern. Darüber hinaus beschränkt sich die Konfiguration auch bei MicroProfile auf das Wesentliche. Beides vereinfacht das Setup deutlich und kann so zu einer Kostenersparnis führen.

Stichwort Ersparnis: Speziell Quarkus setzt auf GraalVM und kann daher nativ kompiliert werden - inklusive Live-Coding im Development-Modus. Im Ergebnis reduzieren sich Startzeit und Round-Trips nach Änderungen drastisch. Jene Tage, in denen Java-EE als schwergewichtig, langsam und kompliziert galt, gehören damit endgültig der Vergangenheit an.

Wie schon bei klassischen Application-Servern wirkt sich die Vorauswahl konkreter Implementierungen durch den jeweiligen Anbieter trotz Extensions negativ auf die Flexibilität aus. Manche Entwickler sprechen in diesem Zusammenhang zuweilen von einer Hassliebe.

Aus technischer Sicht ergibt sich also das Bild, dass man mit beiden Technologien wunderbar Microservices entwickeln kann. Man muss daher nicht zwingend von Java-EE auf Spring oder umgekehrt umsatteln.

## Fazit organisatorisch:

Viel mehr spielen auch organisatorische Faktoren eine Rolle. Insbesondere ist es ratsam abzuwägen, welche Technologie die Umsetzung individueller Anforderungen besser unterstützt. Betrachten wir zunächst Spring, so kann man dieses flexibel auf die speziellen Anforderungen innerhalb der Anwendung anpassen. Durch seine einfache Integration anderer Libraries und Frameworks kann der Funktionsumfang bei Bedarf schnell erweitert werden. Auf der anderen Seite sollte aber Wert daraufgelegt werden, dass eben diese tatsächlich auch einen Großteil der Anforderungen abdecken. Bindet man stattdessen nach Belieben weitere Projekte ein, kann der eigene Service schnell auch zu einer JAR-Hölle werden. Deshalb gilt es zunächst gründlich zu evaluieren, welche Projekte in welchem Umfang die Umsetzung der eigenen Anforderungen unterstützen. Für diese Aufgabe werden jedoch Mitarbeiter benötigt, die ausreichend Zeit und Wissen mitbringen.

Auf der anderen Seite haben wir das Trio aus Jakarta-EE, MicroProfile und Quarkus, das zu vielen Anforderungen an Microservices schon die passenden Implementierungen durch Libraries mitbringt. Diese haben sich in der Praxis bewährt und wurden von der Community als tauglich befunden. An dieser Stelle wird die Zeit der Evaluation gespart und die Mitarbeiter können sich - gemäß dem ursprünglichen Versprechen von Java-EE - auf die eigentliche Geschäftslogik konzentrieren. Problematisch wird es bei Anforderungen die vom Standard abweichen. Dann erhöht sich der Implementierungsaufwand schnell um ein Vielfaches.

Weiterhin sollte man die Vorlieben der eigenen Mitarbeiter nicht außer Acht lassen. Auch wenn beide Technologien ihre individuellen Vor- und Nachteile besitzen, fühlen sich Entwickler gewöhnlich entweder im Spring- oder im JEE-Ökosystem zu Hause. Das Wissen, welches man sich über Jahre angeeignet hat, sollte weiterhin sinnvoll genutzt und für die Umsetzung von Microservices und deren neuen Herausforderungen herangezogen werden. So muss auch kein Mitarbeiter befürchten, sich unverhofft in einer Umgebung wiederzufinden, mit der er nicht vertraut ist.

## Quellen:

- 1 Thorntail: <https://bit.ly/Thorntail>
- 2 JAVAPRO-Artikel: <https://bit.ly/JAVAPRO-a2>
- 3 Bestandteile von MicroProfile 3.3: <https://bit.ly/microprofile-m3>

**Das kostenlose Magazin !**

Einfach anfordern unter: [www.javapro.io](http://www.javapro.io)

GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

# JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

**AB SOFORT!**

## Online Redakteur

M/W/D

**Testen  
Coden  
Bloggen  
Community  
Meetups  
Konferenzen  
Home-Office**

**Anfrage an: [info@javapro.io](mailto:info@javapro.io)**

[www.javapro.io](http://www.javapro.io)

**@JAVAPROmagazin**

**[www.javapro.io](http://www.javapro.io)**

# Quarkus – die Zukunft von Java?

Wenn es um Entwicklungszykluszeiten, Startzeiten von Anwendungen, Größe der Deployments und Kompatibilität in oder mit Containerumgebungen geht, scheinen andere Sprachen wie JavaScript, Python oder Go gerade Java den Rang abzulaufen. Mit Quarkus soll sich das jetzt ändern: Native Anwendungen mit Java, Performance vergleichbar mit C, Startzeiten unter einer Sekunde und dazu alle bereits bekannten Vorteilen von Java. Damit dringt Java in eine neue Dimension vor und setzt völlig neue Maßstäbe.

**Q**uarkus<sup>1,2</sup> ist ein für die GraalVMs<sup>3</sup> optimiertes Java-Framework. Bei GraalVM handelt es sich im Grunde um zwei eigenständige virtuelle Maschinen:

- die eigentliche GraalVM ist eine mehr oder weniger gewöhnliche Java 8 zertifizierte JVM mit einigen Besonderheiten und Optimierungen sowie
- Substrate-VM, welche den JVM-Code (nicht nur Java) in plattformspezifischen Maschinencode übersetzen kann. Es werden damit dynamisch oder statisch gelinkte Executables für verschiedene Ausführungsumgebungen erzeugt.

Für die Substrate-VM gelten einige Besonderheiten. Vor allem von vielen Frameworks verwendete dynamische Java-Sprach-Features, wie Reflections und Dynamic Proxies können nur mit Einschränkungen verwendet werden bzw. bedürfen spezieller

Vorbereitungen. Es ist also leider nicht möglich, beliebigen Java-Code ohne Anpassungen on-the-fly in ein natives Executable zu übersetzen. Als Konsequenz ist es notwendig, dass die Entwickler von Frameworks diese auf die Benutzung mit der Substrate-VM vorbereiten bzw. anpassen.

Ein Framework, das genau auf diese Bedingungen optimiert wurde, ist Quarkus. Dabei werden nicht nur bekannte Standards und Frameworks, wie z.B. RESTEasy, Hibernate, Netty, Vert.x, JUnit, sondern auch weniger bekannte Bibliotheken wie zum Beispiel Panache, Narayana JTA und STM durch sogenannte Extensions eingebunden.

## Prerequisites

Für die nun folgende Arbeit mit Quarkus und der GraalVM benötigen wir die folgenden Tools:

- GraalVM 19.2.1 inklusive Native-Image-Tool
- Docker-Installation 19.03.X
- Maven 3.6.2

und die beiden Umgebungsvariablen **GRAALVM\_HOME** und **JAVA\_HOME** müssen entsprechend korrekt gesetzt werden. Das Native-Image-Tool wird in neueren GraalVM Versionen nicht mehr als Bestandteil der Basisinstallation ausgeliefert. In diesem Fall muss es nachinstalliert werden (**Listing 1**).

### (Listing 1)

```
$GRAALVM_HOME/bin/gu install native-image
```

## Autor:



Bernd Fischer beschäftigt sich seit seinem Studium der Elektrotechnik mit Software-Entwicklung und -Architektur und arbeitet heute als CTO bei der MindApproach GmbH in Dresden.

Über Assembler, Fortran, Pascal, C/C++ kam er vor mehr als 15 Jahren zur Programmiersprache Java und ist ihr seither weitestgehend treu geblieben.

Neben seiner hauptberuflichen Tätigkeit ist er in der JUG Saxony e.V. und der Dresdner Docker-Community aktiv.

## Developer Joy

Die Arbeit mit Quarkus stützt sich auf die beiden weit verbreiteten Build-Werkzeuge Gradle und Maven, wobei wir uns im Weiteren auf Maven konzentrieren. Die Erstellung neuer Projekte wird auf zwei Wegen unterstützt:

- Verwendung des Quarkus-Maven-Plugins oder
- mithilfe der Webseite <https://code.quarkus.io>, auf der die Wunschkonfiguration zusammengestellt und die daraus generierten Projektdateien als Zip-File<sup>4</sup> heruntergeladen werden können.

Auf beiden Wegen kann bequem und schnell ohne großes Dokumentationsstudium das Grundgerüst einer Anwendung erstellt werden. Die Herangehensweise ist von anderen Frameworks wie Spring bekannt und mit diesen vergleichbar. Wir halten uns daher nicht weiter bei diesem Feature auf, sondern verwenden für die folgenden Schritte ein bereits etwas weiterentwickeltes Beispielprojekt<sup>5</sup>.

Zur Unterstützung und Beschleunigung der Entwicklungsarbeit stellt Quarkus einen Developer-Mode zur Verfügung, der die üblichen Source-Code-Pfade des Projekts überwacht und auf Änderungen reagiert. Dieser Modus ist in das Maven-Plugin integriert und wird mit dem Kommando aus (Listing 2) gestartet. Der Parameter **Ddebug=false** schaltet die Erzeugung des Debug-Ports ab, über den sich ein externer Debugger mit dem Prozess verbinden könnte, und spart so ein paar Sekundenbruchteile bei der Startzeit. Die Umgebungsvariablen sorgen dafür, dass nicht die per Default konfigurierte MySQL-, sondern eine In-Memory-Datenbank verwendet wird.

### (Listing 2)

```
QUARKUS_DATASOURCE_DRIVER=org.h2.Driver \
QUARKUS_DATASOURCE_URL=jdbc:h2:mem:default \
mvn compile quarkus:dev -Ddebug=false
```

Wir müssen jetzt in einem zweiten Terminal-Fenster weiterarbeiten, da der Befehl aus (Listing 2) weiterhin aktiv ist. Dort können wir z.B. die Funktionalität eines der REST-Endpunkte testen (Listing 3). Der Source-Code des für diese Funktionalität verantwortlichen Controllers ist in (Listing 4) zu finden.

### (Listing 3)

```
$ http http://localhost:8080/hello
HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/plain;charset=UTF-8

hello again!
```

### (Listing 4)

```
package de.benfir.evl.quarkus;

import org.eclipse.microprofile.config.inject.ConfigProperty;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path( "/hello" )
public class GreetingController
{
    @ConfigProperty( name= "greeting.message" )
    String message;

    @ConfigProperty( name = "greeting.suffix", defaultValue="!" )
    String suffix;

    // STEP1
    @GET @Produces( MediaType.TEXT_PLAIN ) public String hello()

    {
        return this.message + this.suffix;
    }

    // // STEP2
    // @Inject GreetingService service;
    //
    // @GET @Produces( MediaType.TEXT_PLAIN )
    // @Path(("/{name}") )
    // public String greeting( @PathParam( "name" ) String name )
    // {
    //     return this.service.greeting( name );
    // }
    ----
```

Gut erkennbar sind die an Eclipse-Microprofile und CDI (Context-and-Dependency-Injection) angelehnten Konfigurationsmöglichkeiten. Die aktuellen Werte der mit **@ConfigProperty** annotierten Variablen können via Konfigurationsdateien wie zum Beispiel **application.properties** (Listing 5), Systemparametern wie **-Dgreeting.message=blah** oder Umgebungsvariablen wie **GREETING\_MESSAGE=blahblah** definiert werden.

### (Listing 5 – application.properties)

```
# Configuration file

# profile: default
quarkus.datasource.url=jdbc:mariadb://localhost:3306/testdb
quarkus.datasource.driver=org.mariadb.jdbc.Driver
quarkus.datasource.username=test
quarkus.datasource.password=1234
quarkus.hibernate-orm.dialect=org.hibernate.dialect.MariaDBDialect
quarkus.hibernate-orm.database.generation=update

greeting.message=hello
```

```
# profile: test
# used during testing automatically
%test.greeting.message=hello test

# profile: dev
# used during development (i.e. mvn compile quarkus:dev)
%dev.greeting.message=hello dev
----
```

Im Übrigen ist das Suffix **QUARKUS** bei Umgebungsvariablen bzw. **quarkus** bei Systemparametern für interne Anwendungen durch das Framework reserviert. Nun soll der Aufruf von **/hello** parametrierbar sein (Listing 6).

## (Listing 6)

```
http http://localhost:8080/hello/quarkus
```

Natürlich führt der Aufruf dieser URL im schnell und vor der eigentlichen Umsetzung erstellten Unit-Test zu einem **404 – Not Found** Fehler, da noch kein entsprechender Endpunkt implementiert ist. Das können wir nachholen, indem wir als Simulation der tatsächlichen Entwicklungsarbeit die Kommentarzeilen in (Listing 4) entfernen, die Änderung speichern und den Befehl aus (Listing 6) wiederholen, der dann mit Erfolg ausgeführt wird. Anhand der Ausgaben im ersten Terminal, der die Verzeichnisse überwacht und in dem nach wie vor der Quarkus-Developer-Mode aktiv ist, wird deutlich dass Quarkus die durchgeführte Änderung erkennt, den Source-Code kompiliert ("Changed source files detected, recompiling ...") und das Framework neu gestartet hat. Und das alles in nicht einmal einer Sekunde!



Super Sonic Java® (Abb. 1)

## The Usual Java-Build

Wie eingangs erwähnt, ist GraalVM auch eine ganz normale JVM und kann daher wie andere JVMs auf bekannte Weise gängige Java-Anwendungen erzeugen (Listing 7).

## (Listing 7)

```
$ mvn clean package -DskipTests=true
...
$ ls -alh ./target/*.jar
366K Nov  2 14:20 ./target/evl-quarkus-runner.jar
17K Nov  2 14:20 ./target/evl-quarkus.jar
```

Dabei entstehen im Verzeichnis **./target** zwei JAR-Dateien. Für unsere weitere Arbeit ist vor allem die größere der beiden mit dem Postfix **-runner** interessant. Sie enthält neben unserem eigenen Code zusätzlich Start-, Initialisierungs- und Verwaltungs-Code. Die direkten Projektabhängigkeiten aus dem

Maven-POM-File finden sich dagegen im Verzeichnis **./target/1ib**. Damit haben wir genug Informationen, um unsere Java-Anwendung mit analoger Konfiguration wie im Developer-Mode zu starten (Listing 8), wofür nicht einmal ganze drei Sekunden benötigt werden.

## (Listing 8)

```
QUARKUS_DATASOURCE_DRIVER=org.h2.Driver \
QUARKUS_DATASOURCE_URL=jdbc:h2:mem:default \
java -jar ./target/evl-quarkus-runner.jar
```

## Going Native

Nun wollen wir eine native, d.h. ohne Verwendung einer JVM, direkt vom Betriebssystem ausführbare Anwendung erstellen. Das erledigt für uns ein Aufruf des Quarkus-Maven-Plugins (Listing 9). Das Erzeugen nativer Anwendungen ist zeitaufwendig und benötigt nicht unerhebliche Ressourcen. Auf einem MacBook mit Intel Core i7, 2,5 GHz und 16 GByte dauerte der Prozess etwas mehr als 5 Minuten und verbrauchte in der Spitze circa 3 GByte RAM.

## (Listing 9)

```
mvn clean package -Pnative -Dskip
Tests=true
```

Danach liegt im **./target** Verzeichnis neben den beiden bereits erwähnten JAR-Files eine rund 60 MByte große ausführbare Datei namens **evl-quarkus-runner**. Da wir die Erzeugung auf einem Mac ausgeführt haben, handelt es sich um eine native Mac-OSX-Datei.

Von der für Entwicklung (Profile **dev**) und Test (Profil **test**) verwendeten In-Memory-Datenbank H2 gibt es noch keine native Version, deshalb nutzen wir das **prod** Profil und starten dafür via **./run-create-container-dmo\_mysql.sh** eine externe MySQL-Datenbank in einer lokalen Docker-Instanz. Danach kommt die eigentliche Anwendung an die Reihe (Listing 10), die sich mit der MySQL-Datenbank verbindet.

## (Listing 10)

```
QUARKUS_DATASOURCE_DRIVER=org.mariadb.jdbc.Driver \
QUARKUS_DATASOURCE_URL=jdbc:mariadb://localhost:3306/testdb \
./target/evl-quarkus-runner
```

Fast unmittelbar nach dem Drücken der Enter-Taste ist unsere Anwendung auch bereits gestartet. Das Framework selbst benötigt laut Konsolenausgabe weniger als 0,1 Sekunden für den Startvorgang!

## Going Cross Native

Nun wollen wir native Programme nicht nur für das Betriebssystem unseres Entwicklungsrechner, sondern auch für andere Betriebssysteme wie Linux erstellen. Leider unterstützt GraalVM im Moment kein direktes Cross-Compiling. Es ist deshalb notwendig den Übersetzungsprozess in der jeweiligen Zielumgebung auszuführen. Für die Umsetzung dieser Aufgabenstellung verwenden wir Docker-Desktop-CE-for-Mac und eine Art Build-Container. Anstatt die Transformation in ein natives Executable mit Hilfe des Quarkus-Maven-Plugins anzustoßen, dass dann einen Docker-Build anstößt (Listing 11), bevorzugen wir den direkten Weg. Der Befehl aus (Listing 13) erstellt auf der Basis des (Multi-Stage-) Docker-Files aus (Listing 12) ein Docker-Image. Dabei werden alle Build-Schritte im Container ausgeführt.

(Listing 11)

```
mvn clean package -Pnative -Dquarkus.native.container-build=true
```

(Listing 12)

```
# syntax=docker/dockerfile:1-experimental
FROM oracle/graalvm-ce:19.2.1 as build

ENV GRAALVM_HOME="/opt/graalvm-ce-19.2.0/"

RUN gu install native-image

WORKDIR /usr/src/app

COPY mvnw ./
COPY .mvn ./mvn
COPY pom.xml ./
COPY src ./src

RUN --mount=type=cache,sharing=locked,uid=1001,gid=1001,target=/home/root/.m2/repository ./mvnw -f /usr/src/app/pom.xml clean package -DskipTests=true -Pnative

FROM cescOFFier/native-base
COPY --from=build /usr/src/app/target/*runner ./application
CMD ["/application", "-Dquarkus.http.host=0.0.0.0"]
```

(Listing 13)

```
DOCKER_BUILDKIT=1 \
docker image build \
-f src/main/docker/evl-quarkus-svm-dyn.dockerfile \
-t benfir/evl-quarkus:0.2-svm-dyn \
.
```

Zum Abschluss zeigt (Listing 14) das Starten der Anwendung.

(Listing 14)

```
docker container run -d \
--name=quarkus \
--publish 1234:8080 \
```

```
--network dmo_net \
--env QUARKUS_DATASOURCE_DRIVER=org.mariadb.jdbc.Driver \
--env QUARKUS_DATASOURCE_URL=jdbc:mariadb://mysql:3306/testdb \
benfir/evl-quarkus:0.2-svm-dyn
```

Die Log-Ausgaben, (Listing 15) erläutert wie man diese anzeigen kann, belegen, dass auch in einem Container die Anwendung unglaublich schnell startet, und zwar rund 0,1 Sekunden. Auch wenn die implementierte Funktionalität relativ trivial ist, handelt es sich dennoch um eine vollständige REST-basierte Java-Anwendung mit Anbindung an eine externe Datenbank.

(Listing 15)

```
docker container logs quarkus
...
... INFO [io.quarkus] (main) evl-quarkus 1.0 (running on Quarkus
0.27.0) started in 0.170s.
...
```

## Fazit:

Auch wenn Quarkus noch relativ jung ist, kann es bereits einige gelungene Feature auf der Habenseite verbuchen:

- Der Developer-Mode ist gut gelungen.
- Der Java-Code kann mit einigen Einschränkungen und Besonderheiten in kleine schnell startende Anwendungen kompiliert werden.

Damit ist das Framework insbesondere für die Erstellung von Microservices oder für die Implementierung von Businesslogikbausteinen im Umfeld von Message-Brokern wie zum Beispiel Kafka geeignet und natürlich auch für Serverless-Anwendungen à la OpenFaas o.ä. Dem stehen auch einige Punkte auf der Sollseite gegenüber:

- Die Kompilierung in native Anwendungen ist ressourcen- und zeitaufwendig.
- Existierender Code und Frameworks müssen für die Verwendung in oder mit der Substrate-VM fit gemacht werden. Dafür sind u.U. weiterreichende Anpassungen notwendig.
- Die in oder mit Quarkus eingesetzten Frameworks verwenden einige, zumindest für Spring-Anwender ungewöhnliche Konzepte, zum Beispiel Panache für die Persistenz.

## Quellen:

- 1 <https://bit.ly/quarkus-q1>
- 2 <https://bit.ly/code-q2>
- 3 <https://bit.ly/graalvm-g3>
- 4 <https://bit.ly/zip-z4>
- 5 <https://bit.ly/github-q5>
- 6 <https://bit.ly/comic-c6> – combined into one picture publiziert unter: Creative Commons License Attribution 3.0 Unported (CC by 3.0 – <https://creativecommons.org/licenses/by/3.0/>)

#JAVAPRO #CloudNative #Quarkus

# Native Anwendungen mit Java

Quarkus verspricht sagenhafte Performancegewinne. Der Einstieg ist einfach, da Quarkus wie eine Sammlung bereits bekannter Frameworks erscheint. Doch Quarkus wurde speziell für den Einsatz im Container und in der Cloud entwickelt. Hohe Performance und geringer Ressourcenverbrauch stehen im Focus.

**Q**uarkus bietet einen ähnlichen Funktionsumfang wie die Platzhirsche Spring-Boot und Java-EE (jetzt Jakarta-EE). Es ist jedoch nicht das Ziel von Quarkus, die etablierten Frameworks zu verdrängen. Es gibt keinen vorgezeichneten Migrationspfad von Java-EE oder Spring nach Quarkus. In den meisten Fällen dürfte eine Migration so teuer sein, dass sie den Nutzen bei weitem überschreitet. Quarkus ist vielmehr angetreten, um neue Nischen zu besetzen.

Die Welt der Java-Entwickler hat sich geändert. Als Spring-Boot, und erst recht Java-EE, entworfen wurden, sah die typische Laufzeitumgebung noch ganz anders aus als heute. Es gab keine Cloud. Dafür war es populär die UI mit Java zu entwickeln. Bis vor einigen Jahren galten JavaScript, HTML und CSS als schwierig. Ein sehr großer Teil der Java-Entwickler fühlte sich in diesen Bereichen nicht sattelfest. Das machte Java-zentrierte UIs wie Java-Server-Faces (JSF), Spring-MVC oder GWT (ehem. Google Web-Toolkit) populär. Spring und Java-EE mussten den gesamten Bereich vom Frontend bis hin zur Datenbank abdecken.

## Autor:



Stephan ist Architekt, Entwickler und Trainer mit viel Freude an Presales und Marketing. Er pendelt regelmäßig zwischen der Java-Welt und dem JavaScript-Universum. In der Open-Source-Szene hat er sich einen Namen durch Projekte wie BootsFaces und ngx-extended-pdf-viewer gemacht. Außerdem betreibt er den Blog BeyondJava.net, wo er aktuell eine große Serie über die GraalVM und Quarkus schreibt.

Blog: <https://www.beyondjava.net>  
 Twitter: @beyondjava  
 GitHub: stephanrauh  
 E-Mail: [articles@beyondjava.de](mailto:articles@beyondjava.de)

## The Brave New World für Java-Entwickler

Heute sieht die Situation ganz anders aus. Die Branche hat sich differenziert. In vielen Firmen hat sich eine Arbeitsteilung etabliert: Java-Entwickler kümmern sich um die Business-Aufgaben im Backend, während das Frontend die Domäne der JavaScript-Entwickler ist. In der Backend-Entwicklung wiederum findet gerade eine Revolution statt. Der Megatrend Cloud-Computing wirbelt die Welt der Java-Entwickler gründlich durcheinander, zum Beispiel:

- Docker gehört nun zum Entwickleralltag.
- Mit Docker laufen Java-Programme typischerweise unter Linux.
- Application-Server, auf denen mehrere Anwendungen gleichzeitig laufen, wurden durch leichtgewichtiger Spring-Boot-Services abgelöst.
- Neue Serverless-Ansätze wie AWS-Lambda und Azure-Functions sind noch leichtgewichtiger.
- Microservices führ zu einer zunehmenden Arbeitsteilung.

Das alles ist sehr spannend und aufregend, bringt jedoch ganz neue Ebenen an Komplexität mit sich. Der Wunsch nach Vereinfachungen wird laut.

## Der Aufstieg der Micro-Frameworks

Eine ganze Reihe neuer Frameworks springt in die Bresche. Sie bezeichnen sich als Micro-Frameworks, weil sie sich auf bestimmte Aspekte der Softwareentwicklung konzentrieren. Während Spring-Boot noch ein Full-Stack-Framework war, liefern Micro-Frameworks nur noch einen Teil dieser Funktionalität. Bei den Micro-Frameworks sind das die Funktionen, die für Microservices ohne UI benötigt werden. MicroProfile, Helidon, Micronaut, Spark, Javalin und Ktor sind einige Micro-Frameworks, die auf dem Radar auftauchen. Und natürlich Quarkus.

## Altbekannte Frameworks neu kombiniert

Eine für die Einarbeitung sehr angenehme Eigenschaft von Quarkus ist, dass es auf den ersten Blick wenig Neuerungen bringt. Im Werkzeugkasten<sup>1</sup> auf der Quarkus-Homepage finden wir unter anderem Hibernate, JAX-RS, CDI, Bean-Validation, Flyway und Liquibase sowie Apache-Kafka. Viele Java-Entwickler haben mit diesen oder ähnlichen Bibliotheken schon gearbeitet. Das sorgt für einen stressfreien Start. Es gibt auch Support für Spring. Konkret werden Spring-Core, Spring-Data, Spring-Web, Spring-Security und ein Teil von Spring-Boot unterstützt. Es handelt sich um einen Kompatibilitäts-Layer; in der Praxis sind daher Einschränkungen zu erwarten. Die Botschaft ist aber auch hier klar: Wer Spring kennt, sollte sich schnell mit Quarkus wohlfühlen. Weniger bekannt sind die reaktiven Frameworks von Quarkus: Vert.x und Smallrye. Darauf wird später nochmal eingegangen. Beide sind schon seit einigen Jahren auf dem Markt. Mit anderen Worten: Quarkus bringt viele bekannte Frameworks zusammen. Auf der Homepage von Quarkus lässt sich ein Beispielprojekt generieren. Es ist ein klassisches Maven- oder Gradle-Projekt ohne große Überraschungen (Abb. 1). Die APIs sind also nicht das Alleinstellungsmerkmal von Quarkus. Was ist es stattdessen?

## Container-First Ansatz

Einige Neuerungen fallen im Screenshot doch auf: der Ordner Docker, zwei Dateien mit der Bezeichnung native im Namen und das **Maven-Plugin** von Quarkus. Auf den zweiten Blick fällt noch auf, dass anstelle der Standardbibliotheken Quarkus-Versionen von RESTEasy, JUnit, Hibernate usw. verwendet werden. Quarkus wurde für den Einsatz im Container und in der Cloud entwickelt. Daraus leiten sich verschiedene Optimierungsziele ab. Es sind die Dauerbrenner Performance, Speicherverbrauch und Kosten – jetzt aber mit anderen Schwerpunkten:

- Im Bereich Performance gewinnt die Dauer eines Kaltstarts an Bedeutung. Die Peak-Performance ist relevant bei Anwendungen, die mehrere Minuten oder Stunden laufen. Beim Einsatz in der Cloud ist das aber gar nicht mehr die Regel. Um Kosten zu sparen, werden Container schnell herunter- und nur bei Lastspitzen wieder hochgefahren. Besonders augenfällig wird das bei AWS-Lambdas und Azure-Functions. Wird die Funktion einige Minuten nicht aufgerufen, bedeutet der nächste Aufruf einen Kaltstart.
- Mit Speicherverbrauch ist nicht mehr der Heap-Memory gemeint, sondern der gesamte Speicher, den der Container braucht. Die JVM und das Betriebssystem gehören dazu.
- In der Cloud bestimmen sowohl die Rechenleistung als auch der Speicherbedarf die Kosten. Die Rechenleistung ist bei typischen Enterprise-Anwendungen sekundär: Es können jederzeit weitere Container hochgefahren werden. Damit wird, abhängig vom Preismodell des Cloudproviders, oft der Hauptspeicher zum primären Kostentreiber.

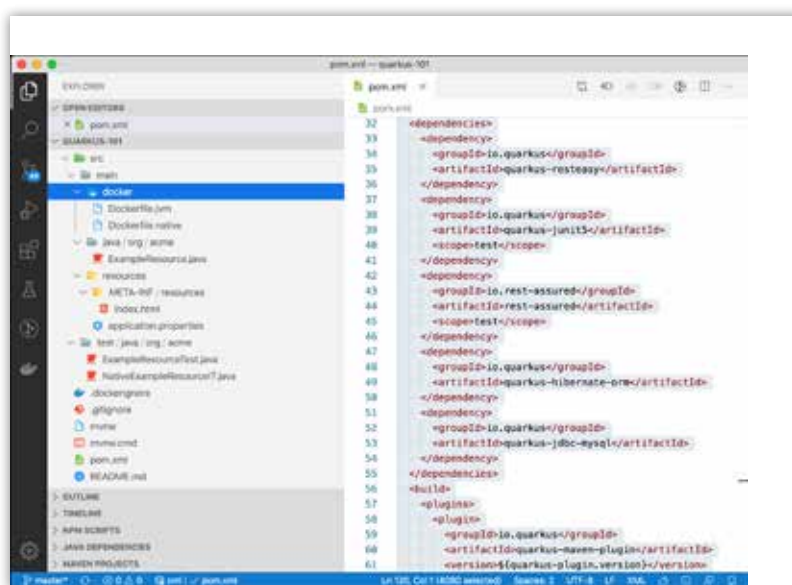
## Native Anwendungen mit GraalVM

Quarkus verschiebt einen Teil der Aufgaben vom Anwendungsstart in die Kompilierphase. Das ist nicht zuletzt der GraalVM geschuldet. GraalVM bietet einen Ahead-of-Time-Compiler (AOT-Compiler). Dieser Compiler erzeugt Maschinencode, der von der CPU direkt ausgeführt werden kann. Das hat Vor- und Nachteile. Zu den Vorteilen gehört ein dramatisch beschleunigter Anwendungsstart. Zu den Nachteilen zählen eine langfristig schlechtere Performance und der Verzicht auf Reflection, was jedoch den Verzicht auf fast alle populären Java-Frameworks bedeuten würde. Daher kann die GraalVM zur Compile-Zeit Metadaten erzeugen, mit deren Hilfe die Funktionalität der Reflection-API gerettet werden kann. Der Preis dafür sind erheblich längere Kompilierzeiten.

Ein weiterer Preis ist, dass sich die fertig kompilierte Anwendung zur Laufzeit nicht mehr ändern darf (Closed-World-Principle). In der Praxis ist das keine Einschränkung. Wenn eine Anwendung verändert werden soll, wird sie ohnehin neu kompiliert. Etwas anders sieht es bei einem traditionellen Application-Server aus. Dort konnten die WAR-Dateien zur Laufzeit installiert werden. Mit einem AOT-Compiler ist so etwas nicht möglich. Die Unterstützung des AOT-Compilers hat auch ein paar Nachteile.

## Die Aufgaben der Quarkus-Extensions

Quarkus macht aus der Not eine Tugend. Wie bereits erwähnt, werden nicht direkt die gewohnten Frameworks verwendet, sondern speziell für Quarkus angepasste Versionen, die Quarkus-Extensions. Die Erweiterung



Typischer Aufbau einer Quarkus-Anwendung (Abb. 1)

besteht darin, zur Kompilierzeit Metadaten zu sammeln und möglichst viele Aufgaben vom Anwendungsstart in die Kompilierzeit zu verschieben. Peter Palaga skizziert in seinem Vortrag<sup>2</sup>, wie das aussieht: Statt alle Beans bei jedem Anwendungsstart per Classpath-Scan zu suchen und per Reflection zu erzeugen, werden sie gleich beim Kompilieren in eine statische Hash-Table geschrieben. Damit sind sie sofort verfügbar und können schnell erzeugt werden. Je nach Scope können die Instanzen dieser Beans auch gleich erzeugt werden. Er beschreibt auch noch weitere Tricks, mit denen Quarkus den Start der Anwendung optimiert. Die statische Codeanalyse während der Kompilierzeit ermöglicht eine Art Tree-Shaking. Klassen, welche die Anwendung nicht benötigt, werden eliminiert. Das spart den Aufwand sie zu laden und zu initialisieren. Bei Spring-Boot und erst recht bei einem Application-Server macht das einen erheblichen Teil der Startzeit aus.

## Enorme Performance mit GraalVM

Ein besonders raffinierter Trick wird nicht von Quarkus, sondern von der GraalVM implementiert. Sie führt statische Initializer bereits zur Kompilierzeit aus, nimmt einen Snapshot des so entstandenen Heap und kopiert ihn in das Executable<sup>3</sup>. Wenn die Anwendung später gestartet wird, brauchen die statischen Initializer damit nicht mehr ausgeführt werden. Das Ergebnis liegt ja bereits vor.

In praktisch allen Präsentationen zur GraalVM wird gezeigt, dass der Speicherverbrauch einer nativen, mit dem AOT-Compiler erzeugten Anwendung 80% bis 90% niedriger ist als der Speicherverbrauch einer traditionellen Anwendung mit JIT-Compiler. Das ist grundsätzlich korrekt, gilt in diesem Ausmaß aber nur für relativ kleine Anwendungen. Bytecode ist erheblich kompakter als Assembler-Code. Je größer eine Anwendung ist, desto geringer wird der Vorteil des AOT-Compilers, und irgendwann drehen sich die Verhältnisse auch um. Ähnlich wird es bei den Startzeiten sein. Bei den meisten Benchmarks startet die Anwendung dank des AOT-Compilers rund tausendmal schneller als mit dem JIT-Compiler<sup>4</sup>. Führt man aber eigene Programmlogik bei Programmstart aus, wird das Ergebnis weniger überzeugend sein. Der Grundsatz des AOT-Compilers lautet, dass man ein wenig Peak-Performance gegen schnelle Verfügbarkeit eintauscht. Das sollte man beim Anwendungsdesign berücksichtigen. Für die meisten Microservices, wenn sie denn wirklich micro sind, ist das Resümee aber tatsächlich: deutlich weniger Memory-Footprint, stark reduzierte Startzeiten, aber dafür wird das Programm niemals die Performance des JIT-Compilers erreichen. Konkrete Zahlen zu nennen ist sehr schwierig, da es erheblich von der jeweiligen Aufgabe abhängt. In den meisten Fällen dürfte der JIT-Compiler jedoch rund zwei bis dreimal schneller sein.

Quarkus macht die Arbeit mit GraalVM einfacher. Wie in (Abb. 1) zu sehen, werden die Dateien zum Generieren eines nativen Executables von Quarkus standardmäßig erzeugt. Das manuelle

Erstellen dieser Datei artet laut Peter Palaga<sup>5</sup> dagegen schnell in eine endlose Liste von Parametern aus.

## Performance im Vergleich

Bei dem auf GitHub<sup>6</sup> verfügbaren Demoprojekt zu diesem Artikel handelt es sich um einen besonders kleinen Lambda-Service, der aber gerade deshalb die Unterschiede zwischen dem AOT-Compiler und dem JIT-Compiler besonders deutlich macht. Mit Java 8 und der traditionellen JVM dauert der erste Start spürbar lang.

```
> ./invoke-java-function.sh
Duration: 367.99 ms Billed Duration: 400 ms Memory Size: 128 MB
Max Memory Used: 97 MB Init Duration: 1112.91 ms
```

Dafür ist der zweite und alle folgenden Funktionsaufrufe rund 1000 Mal schneller – so lange, bis der Lambda-Service nach einer Pause wieder herunterfährt.

```
> ./invoke-java-function.sh
Duration: 0.60 ms Billed Duration: 100 ms Memory Size: 128 MB Max
Memory Used: 97 MB
```

Wenn der Lambda-Service dagegen nativ kompiliert wird, dauert der Aufruf immer rund eine Millisekunde.

```
./invoke-native-function.sh
Duration: 1.04 ms Billed Duration: 100 ms Memory Size: 128 MB Max
Memory Used: 52 MB
```

Das ist auch in diesem einfachen Beispiel, in dem es kaum etwas zu optimieren gibt, nur halb so schnell wie die Peak-Performance des JIT-Compilers. Die Messung ist jedoch zuverlässig. Bei jedem Versuch lag die Performance im Bereich zwischen 0,9 bis 1,1 Millisekunden.

## Wann lohnen sich langsamere Microservices?

Traditionell ist die CPU-Performance das Maß der Dinge. Speicher galt lange Zeit als billige Ressource. In der Cloud sieht das oft anders aus. Hier ist der Speicherverbrauch ein wesentlicher Kostentreiber. Der geringere Memory-Footprint einer nativen Anwendung erlaubt eine höhere Packungsdichte. Der Speicher, der vorher für einen einzigen Application-Server nötig war, reicht jetzt für eine ganze Schar von Containern. Wenn alle Container gleichzeitig gestartet werden, also bei horizontaler Skalierung, ist der Durchsatz größer als der Durchsatz des einen Application-Servers. Die gute Performance des JIT-Compilers kann das nicht ausgleichen.

## Wann braucht man vorhersehbare Performance?

Manchmal kommt es auch gar nicht so sehr auf die Spitzengeschwindigkeit an. In Echtzeitsystemen ist das entscheidende

Kriterium, dass das System garantiert in einer bestimmten Zeitspanne reagiert. Ein Beispiel dafür wäre eine in Java programmierte Motorsteuerung eines Autos. Der JIT-Compiler sorgt dafür, dass die CPU reichlich Leistungsreserven hat – mit Ausnahme des Kaltstarts. Java nimmt erst im Laufe der Zeit Fahrt auf. Für den Motor bedeutet das jede Menge Fehlzündungen. Womöglich sogar jedes Mal, wenn die Start-Stopp-Automatik an der Ampel den Motor ausschaltet. Der AOT-Compiler sorgt dagegen für eine vorhersagbare Performance. Sie mag etwas geringer sein, aber vor allem ist sie konstant. Die Motorsteuerung hat nicht mehr so viel Leistungsreserven, aber darauf kommt es in diesem Fall gar nicht an. Entscheidend ist, dass sie keine Fehlzündungen erzeugt.

## Quarkus ohne GraalVM

Auch ohne GraalVM profitiert man beim Kompilieren von den Features von **Quarkus**. Anwendungen starten auch mit dem klassischen JIT-Compiler i.d.R. deutlich schneller, in unseren Beispielen um den Faktor zwei bis drei. Ein weiteres hilfreiches Feature von Quarkus ist das Hot-Module-Reloading. Dazu startet man Quarkus im Entwicklungsmodus, indem man ein Terminal öffnet und auf der Kommandozeile den Befehl `./mvnw compile quarkus:dev` eingibt. Anschließend geht man genauso vor wie zum Beispiel bei Angular: Quellcode editieren, beim Speichern wird automatisch kompiliert und beim Aktualisieren des Browsers wird der neue Code ausgeführt. Kurioserweise handelt es sich um ein sehr altes Feature, das bereits in Java 1.4 enthalten war. Seinerzeit hatte IBM einen Application-Server entwickelt, der ca. fünf bis zehn Minuten zum Starten brauchte. An ein flüssiges Entwickeln war damit nicht zu denken. Als Ausweg wurde das Hot-Code-Replacement erfunden. Das ermöglicht es der IDE, geänderte Quelltexte in das laufende Programm einzufügen. Der Zustand der Anwendung bleibt dabei erhalten. Dank Hot-Code-Replacement ist das Einloggen nach jeder Programmänderung bei einer komplexen Anwendung mit UI nicht notwendig. Leider funktioniert Hot-Code-Replacement nicht besonders gut mit Frameworks wie beispielsweise Spring. Spätestens mit dem Siegeszug der testgetriebenen Entwicklung und der Fokussierung auf Backend-Entwicklung geriet das Feature in Vergessenheit. Spring Boot startet so schnell, dass das Hot-Code-Replacement nicht mehr so wichtig ist. Quarkus startet noch etwas schneller. Bei umfangreichen Test-Suiten spielt das eine große Rolle. Vor allem Integrationstests laufen mit Quarkus rund doppelt so schnell.

## Die richtige IDE für Quarkus

Der developer mode von Quarkus besteht aktuell aus einem Maven-Skript. Das Kommando `./mvnw compile quarkus:dev` startet einen Prozess, der die Dateien im Quelltextverzeichnis überwacht und bei jeder Änderung neu kompiliert. Traditionell arbeiten IDEs wie zum Beispiel Eclipse anders. Im traditionellen Modell kümmert sich die IDE darum, dass die Anwendung

kompiliert wird. Die geänderte Arbeitsweise hat einige Auswirkungen. Eine davon ist relativ überraschend, zumindest aus der Sicht eines Java-Entwicklers: Visual-Studio-Code ist ein hervorragender Editor für Quarkus. In der JavaScript-Welt, aus der Visual-Studio-Code kommt, hat es sich ohnehin weitgehend etabliert, den Compiler in einem Terminal-Fenster zu starten. Quarkus überträgt das gleiche Modell in die Java-Welt. Eclipse kann ebenfalls verwendet werden, es ist aber ein wenig mühsamer, weil Eclipse standardmäßig kein Terminalfenster mitbringt. IntelliJ dagegen bietet ein integriertes Terminal. Trotzdem lohnt es sich, Visual-Studio-Code in die engere Wahl zu nehmen: Sowohl Quarkus als auch das Java-Plugin von Visual-Studio-Code werden primär von Red Hat vorangetrieben. Beide sind gut aufeinander abgestimmt. Zum Debugging muss in allen IDEs das remote Debugging verwendet werden. Das ist ein wenig lästig, funktioniert aber tadellos.

## Reactive Programming mit Quarkus

In der JavaScript-Welt sind in den letzten Jahren die reaktive Programmierung und Non-Blocking-IO populär geworden. Quarkus überträgt diese Ideen in die Java-Welt und unterstützt sowohl imperative Programmteile als auch reaktive. Hinter der reaktiven Programmierung steckt die Idee, dass das Programm nicht blockiert, wenn auf das Ergebnis einer Datenbankabfrage oder einer REST-Calls gewartet wird. In der Zwischenzeit kann das Programm weiterlaufen und sich um andere Aufgaben kümmern. Sobald das Ergebnis da ist, wird eine Call-Back-Funktion aufgerufen. In Java sind das Lambda-Expressions. Der Unterschied wird in **(Listing 1)** gezeigt. Es handelt sich hierbei nicht um Quellcode, sondern um stark reduzierten Pseudocode. Das erste Beispiel ist JDBC-Code, welcher schon seit Java 5 bekannt ist:

(Listing 1)

```
// Pseudocode!
public List<Fruit> findAllImperative(Statement client) {
    ResultSet result = client.query("SELECT id, name FROM fruits");

    // hier muss die CPU warten

    return convertResultToArrayList(result);
}
```

Das Programm hält so lange an, bis die Datenbankabfrage ein Ergebnis zurückliefert. Viel schöner wäre es, wenn mehrere Datenbankabfragen parallel abgeschickt werden könnten. Reactive-Programming macht das möglich:

(Listing 2)

```
public Uni<List<Fruit>> findAllReactive(PgPool client) {
    return client.query("SELECT id, name FROM fruits")
        .map(rowSet -> { convertResultToArrayList(rowSet);
    });
}
```

In diesem Fall ist der Code, der die Daten verarbeitet, nicht mehr direkt in der nächsten Zeile. Hinter dem Aufruf der Methode **query()** folgt ein Lambda, das asynchron aufgerufen wird, wenn die Daten da sind. In der Zwischenzeit läuft die Anwendung weiter und könnte weitere Abfragen starten. Deswegen liefert die Methode auch keine Liste. Das **Uni<List>** ist lediglich ein Versprechen, demnächst eine List zu bekommen.

Die reaktive Programmierung erfordert ein Umdenken. Was passiert zum Beispiel, wenn in der Callback-Methode wiederum eine Datenbankabfrage passiert? Dann würde ein Versprechen auf ein Versprechen zurückgeliefert. Das erfordert etwas Übung. Quarkus erleichtert den Einstieg, indem es beide Programmierstile zulässt.

## Reactive Programming weitergedacht

Im Prinzip könnte der Datenbanktreiber auch einen Datenstrom schicken. Anders ausgedrückt: Er könnte bereits anfangen, das Ergebnis der Abfrage zu schicken, wenn er erst einen Teil der Daten gelesen hat. Der Rest der Daten kommt später und wird ebenfalls über das Lambda verarbeitet. Datenbankbetreiber, die dieses Feature bereits beherrschen, sind dem Autor nicht bekannt.

Bei vielen anderen Systemen, wie zum Beispiel Kafka ist das bereits heute Standard. Ein Anwendungsfall ist ein Chat-Programm, das den ganzen Tag im Hintergrund läuft und sofort reagiert, wenn eine Nachricht ankommt. Wenn die Architektur so konzipiert ist, dass die Daten als kontinuierlicher Strom von Ereignissen kommen, ist die asynchrone Verarbeitung in einer Callback-Methode die einfachste Lösung.

Reaktive Programmierung hat noch weitere Vorteile. Wenn die Datenbank sehr groß ist und der Datenbanktreiber mitspielt, kann die Callback-Methode bereits aufgerufen werden, wenn erst ein Teil der Daten verfügbar ist. Die Callback-Methode wird mehrfach aufgerufen. Immer dann, wenn weitere Daten ankommen.

## Java-Programme mit Quarkus als AWS-Lambda deployen

Auf [Quarkus.io](https://quarkus.io)<sup>7</sup> findet sich ein Tutorial das zeigt, wie die Unterstützung von Quarkus aussehen kann. Das Beispielprojekt wird im wesentlichen durch ein Maven-Archetype erzeugt und enthält einige Skripte, um einen Amazon-Lambda-Service mit wenigen Handgriffen zu deployen – oder auch lokal zu testen. Das erleichtert den Einstieg in die Technik deutlich. Vor allem ist es mit den Skripten sehr einfach, ein native kompiliertes Java-Programm als Lambda zu deployen. Die gleiche Anwendung mit Shell-Skripten, welche die Arbeit noch etwas weiter vereinfachen, kann man auf GitHub-Repository<sup>8</sup> ansehen. Der erste Eindruck ist positiv. Der Maven-Archetype liefert die Konfiguration von SAM mit, so dass die Lambda-Funktion lokal getestet werden kann. Die Shell-Skripte erlauben sowohl das Deployment einer traditionellen

JAR-Datei als auch das Deployment einer nativen Binärdatei. Dafür wird GraalVM als Docker-Image heruntergeladen. Der AOT-Compiler braucht also nicht lokal installiert zu werden. Das ist praktisch, bedeutet aber auch, dass es etwas länger dauert.

Die native Kompilierung zeigt am deutlichsten, dass der AWS-Lambda-Support noch im Preview-Stadium ist. Neben den unangenehm langen Kompilierzeiten fielen in der Version 1.3.2 teilweise kryptische Fehlermeldungen unangenehm auf. Die langen Laufzeiten liegen allerdings in der Natur der Sache. Die statische Code-Analyse des AOT-Compilers und die Analyse der Reflection dauern einige Zeit. Die Erzeugung der nativen Images wird am besten in einen Jenkins-Job ausgelagert, der asynchron nach dem Einchecken der Quelltexte läuft.

## Fazit:

Auf den ersten Blick ist es gar nicht so einfach zu sagen, was Quarkus ausmacht. Fast alles, was einem in den Tutorials begegnet, ist vertraut. Allenfalls das reaktive Programmieren mit Vert.x und Smallrye mag ungewohnt sein. Das eigentlich Neue an Quarkus ist, dass es all diese vertrauten Frameworks auf eine neue Weise zusammenbringt. Die Magie liegt in der Build-Pipeline. Quarkus hat den Build-Prozess für viele Frameworks so optimiert, dass sie besonders gut mit dem AOT-Compiler von GraalVM harmonieren. Und das wiederum ist ein großer Vorteil in der Cloud, wo Container häufig hoch- und runtergefahren werden.

Auch jenseits der Cloud hat Quarkus ein paar Vorteile. Der Anwendungsstart ist deutlich schneller, und das fällt in vielen Situationen auf. Unit-Tests laufen schneller, und das gilt insbesondere für Integrationstests. Natürlich hat auch Quarkus ein paar Eigenheiten und Nachteile. Die IDE wird auf die Rolle eines Editors reduziert. Der Compiler läuft in einem separaten Fenster und Debugging ist nur über Remote-Debugging möglich. Für Softwareentwickler ist aber noch etwas wichtig: Der Entwicklermodus von Quarkus bringt das Hot-Module-Reloading in die Java-Welt. Dies ist bereits seit Java 1.4 mit dem Hot-Code-Replacement bekannt, also bereits seit 18 Jahren. Allerdings ist dieses Feature meistens nicht mehr zugänglich. Es spricht einiges dafür, dass Quarkus im Bereich Cloud-Computing seine Nische finden und uns noch viele Jahre begleiten wird.

## Quellen:

- 1 <https://bit.ly/quarkus-c1>
- 2 Peter Palaga, „Quarkus from Inside“: <https://bit.ly/palaga-p2>
- 3 Peter Palaga, „Quarkus from Inside“: <https://bit.ly/palaga-p3>
- 4 Unterschied JIT- und AOT-Compiler: <https://bit.ly/graalvm-u4>
- 5 Peter Palaga, „Quarkus from Inside“: <https://bit.ly/palaga-p5>
- 6 Demoprojekt auf GitHub: <https://bit.ly/github-g>
- 7 Tutorial Quarkus: <https://bit.ly/tutorial-t7>
- 8 GitHub-Repository: <https://bit.ly/github-g>

#JAVAPRO #Frameworks #Persistence

# MicroStream 4 – High-Performance Java-Native Persistence

Native Anwendungen mit Java. Java-Services so schnell wie C. Startzeiten in Millisekunden und minimaler Memory-Footprint. Auf Basis von GraalVM entsteht ein neuer Java-Stack für moderne Cloud-Native-Applikationen, der Java auf ein völlig neues Level bringt. Mit MicroStream kommt jetzt eine hoch performante und gleichzeitig leichtgewichtige Persistence dazu die mit bis zu 1000 Mal schnelleren Datenbankzugriffen eine extrem hohe Performance bietet.

## Autor:

Richard Fichtner ist ein passionierter Java Entwickler mit mehr als 15 Jahren Erfahrung in der Softwarebranche. Er engagiert sich in der Open Source Community, um das Wissen über Java-Technologien zu verbreiten. Er spricht auf Konferenzen wie Oracle Code / JavaOne / JCON/ Clean Code Days und vielen anderen und leistet einen Beitrag zu verschiedenen Open Source Projekten wie <http://www.rapidclipse.com/>. Als Oracle Groundbreaker Ambassador Alumni und AWS Certified Solution Architect unterstützt er Teams beim Einsatz von Cloud Lösungen. Seine Interessenschwerpunkte sind Clean-Code, Cloud, neue Technologien und alles was agil ist. In seiner Freizeit organisiert er die Java User Group Oberpfalz.



**M**it GraalVM, Quarkus und leichtgewichtigen Microservice-Frameworks wie Micronaut und Helidon dringt Java in neue Dimensionen vor. Java-Anwendungen so schnell wie C-Programme, Startzeiten in Millisekunden, mit all den Stärken von Java – klingt beeindruckend. Das ist der neue Java-Cloud-Native-Stack mit GraalVM im Kern. Er wird von Oracle stark vorangetrieben, um Java fit zu machen für die Entwicklung moderner, hoch performanter und ressourcensparender Applikationen und Services für die Cloud. Das einzige was diesem Stack noch fehlt, ist eine ebenso leistungsfähige wie leichtgewichtige Persistence. Genau dafür wurde MicroStream entwickelt – seit kurzem in Version 4 mit bedeutenden Neuerungen verfügbar.

### Paradigmenwechsel: Pure Java only

Genauso wie Microservices bei der Architektur oder Container bei der Virtualisierung zu einem Paradigmenwechsel führen und damit ein Umdenken erfordern, setzt MicroStream einen Paradigmenwechsel und Umdenken bei der Datenspeicherung voraus. Gute Nachricht für Java-Entwickler: MicroStream verfolgt eine kompromisslose Pure-Java-Philosophie.

### Java bietet alles für eine ultraschnelle Datenverarbeitung

Java ist die perfekte Technologie für hoch performante Datenverarbeitung und das JDK bietet alles was man dafür braucht, unter anderem eine universelle Datenstruktur, eine typsichere Abfragesprache, extrem hohe Performance zur Laufzeit, hohe Stabilität und Zuverlässigkeit.

### Java-Objektgraphen – eine universelle Datenstruktur für jeden Anwendungsfall

In Java werden Daten in Form von Objekten bzw. komplexen Objektgraphen verwaltet, die sich hervorragend eignen um vernetzte, komplexe Daten abzubilden. Auf den ersten Blick erinnern Objektgraphen an das frühe hierarchische Datenbankmodell und bieten ähnliche Vorteile, während es die gravierenden Nachteile von damals bei Objektgraphen nicht mehr gibt. Verknüpfungen über mehrere Ebenen sind mit Zirkelbezügen möglich und n:m Beziehungen sowie der Aufbau von Indizes lassen sich mit Hilfe von Collections umsetzen. Ein Objektgraph kann sich aus Millionen von Objekten und tausenden Teilgraphen zusammensetzen. Man kann komplexeste Vernetzungen der Daten durch Zirkelbezüge abbilden, aber genauso gut einfache Listen oder gar nur ein einzelnes Objekt in einem Objektgraphen verwalten. Da alle Java-Typen sowie Collections verwendet werden können, kann man Java-Objektgraphen als eine Art Multi-Model-Datenstruktur bezeichnen.

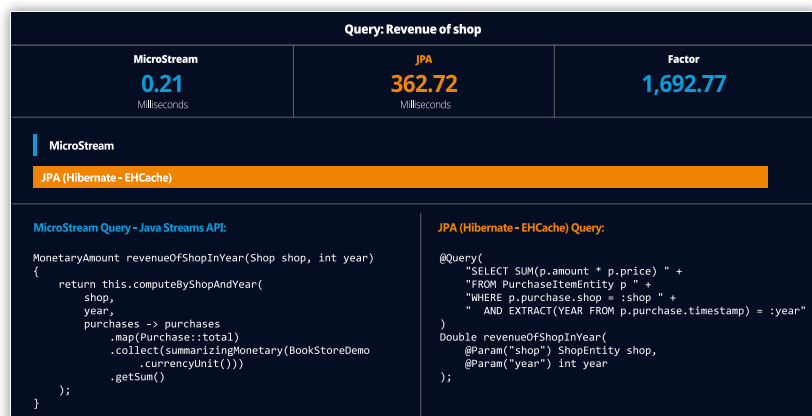
### Ultraschnelle Abfragen mit Java-Streams

Um Objektgraphen zu durchsuchen, bietet Java ab Version 8 mit der Java-Streams-API eine sehr effiziente Abfragesprache. Mit Parallel-Streams lassen sich die Suchvorgänge auf Objektgraphen zudem parallelisieren. Umso mehr Rechenkerne auf der Hardware-Ebene zur Verfügung stehen, desto schneller läuft die Suche. Das Durchsuchen komplexer Objektgraphen im RAM, selbst mit Datenmengen im Bereich mehrerer hundert Gigabyte, bewegt sich in der Regel im Bereich von nur wenigen Millisekunden. Bei gut designten Objektmodellen zum Teil nur in Mikrosekunden. Mit Java-Objektgraphen und Streams API als Abfragesprache lassen sich zweifelsohne hoch performante Java-In-Memory-Datenbanken aufbauen. Das einzige was uns in Java seit jeher fehlt, ist eine Möglichkeit, Java-Objektgraphen effizient zu speichern. Genau dafür gibt es jetzt MicroStream.

### Objektgraphen serialisieren – gute Idee nicht praxistauglich

Die Idee Objektgraphen direkt zu persistieren ist alles andere als neu. Mit Java 1.1 wurde Serialisierung eingeführt, um Objekte über ein Netzwerk zu übertragen oder in einer Datei persistieren zu können. Der größte Vorteil von Serialisierung ist, dass sie sehr einfach zu verwenden ist und keine spezielle Architektur voraussetzt – und es gibt nur ein einziges Datenmodell: Java-Klassen. Objekte müssen nicht mehr auf ein Datenbankmodell gemappt werden, sondern werden einfach in binärer Form gespeichert. Sämtliche mit Mapping verbundenen Probleme und Performanceverluste fallen vollständig weg. Die Vorteile klingen verlockend, sodass sich im Laufe der Jahre schon viele Entwickler mit diesem Ansatz auseinandergesetzt und ihre eigenen Erfahrungen damit gemacht haben.

Die Sache hat jedoch einen entscheidenden Haken. Die Java-Serialisierung ist stark limitiert, nicht sehr effizient und vorhersehbare Folgeanforderungen an Objektgraphpersistierung sind mit heutigen Serialisierungs-Frameworks nicht lösbar. Wird zum Beispiel ein Objekt serialisiert, das auf ein anderes Objekt referenziert, werden automatisch beide Objekte serialisiert. Das kann



(Abb. 1)

dazu führen, dass der gesamte Objektgraph, sprich die ganze Datenbank gespeichert wird – und umgekehrt, dass beim Deserialisieren automatisch die gesamte Datenbank in den RAM geladen wird. Ein weiteres großes Problem ist: beim Laden von Objekten erhält man Kopien dieser Objekte, die keinen Bezug mehr zum Objektgraphen besitzen. Darüber hinaus wären viele weitere Probleme zu lösen. Deshalb sind bisher alle Ansätze Objektgraphen mit Java-Serialisierung zu persistieren, mit dem Ziel OR-Mapping und relationale Datenbanken überflüssig zu machen, im Sande verlaufen.

## Persistierung und dynamisches Laden von Objektgraphen und Teilgraphen

MicroStream ist nun der Durchbruch gelungen. Denn MicroStream ermöglicht es erstmals, einzelne Objekte eines Objektgraphen oder Teilgraphen persistent zu speichern. Umgekehrt lassen sich gezielt einzelne Objekte oder Teilgraphen nachladen, die dann automatisch in den Objektgraphen im RAM gemergt werden. Man muss sich nicht mehr mit Objektkopien, Session-Kontext und Objektlebenszyklen herumplagen. Um das zu ermöglichen haben die MicroStream-Entwickler eine vollständig neue, ultraschnelle und gleichzeitig hochsichere Serialisierung entwickelt, die den Kern von MicroStream bildet. Die Performance von MicroStream-Applikationen ist förmlich atemberaubend. Datenbankabfragen werden im Vergleich zu ORM-Frameworks wie Hibernate bis zu 1000 Mal schneller ausgeführt. Das auf der MicroStream-Webseite<sup>1</sup> laufende Performance-Demo (**Abb. 1**) ist auch auf GitHub<sup>2</sup> veröffentlicht, sodass man damit selbst experimentieren kann. Damit ist MicroStream eine hoch effiziente Alternative zu JPA und ist geradezu prädestiniert für den Einsatz in Microservices mit eigener Persistenz. Die Objekte lassen sich wahlweise in relationale Datenbanken, NoSQL-Datenbanken, In-Memory-Datenbanken, In-Memory-Data-Grids, Distributed-Caches, Cloud-Blob-Stores oder in einfachen Files persistieren. Letzteres wird jedoch nur zu Testzwecken und Prototypen empfohlen.

## Starten mit MicroStream

MicroStream ist eine kleine Java-Library, die man via Maven einbindet. Zur Laufzeit läuft MicroStream embedded in derselben JVM wie die Applikation.

### (Listing 1)

```
<repositories>
<repository>
<id>microstream-releases</id>
<url>https://repo.microstream.one/repository/maven-public</url>
</repository>
</repositories>
<dependencies>
<dependency>
<groupId>one.microstream</groupId>
<artifactId>storage.embedded</artifactId>
<version>04.00.00-MS-GA</version>
```

```
</dependency>
</dependencies>
```

## Java-In-Memory-Datenbank

Ein Objektgraph im Hauptspeicher repräsentiert im MicroStream-Kontext eine Java In-Memory-Datenbank. Es ist wichtig zu verstehen, dass alle Datenbankoperationen auf dem Objektgraphen stattfinden und jede Änderung am Objektgraphen eine Änderung der Datenbank bedeutet.

Um einen persistierbaren Objektgraphen zu erzeugen, braucht man zuerst einen Startpunkt, eine sogenannte Root-Instanz. Alle Objekte, die gespeichert werden sollen, werden dann an die Root angehängt oder an Objekte, die bereits an der Root hängen.

Zu jeder Root-Instanz gehört immer eine Storage, in der die Daten in Form von Bytecode persistent gespeichert werden. Als Storage kann man eine relationale Datenbank, NoSQL Datenbanken, Cloud Blob-Stores, aber auch In-Memory-Datenbanken, In-Memory-Data-Grids, Distributed Caches, sogar Event-Streaming-Plattformen wie Kafka, aber auch einfache Dateien verwenden.

## Elegantes Objektmodell mit POJOs

Definiert wird das Datenmodell ausschließlich durch Java-Klassen. Ein Datenbankdatenmodell gibt es mit MicroStream nicht mehr und damit sind auch keine Mappings mehr nötig. Die Klassen sind reine POJOs. Es gibt weder eine Superklasse von der man ableitet, noch Interfaces die man implementieren, oder Annotations mit denen man seine Klassen erweitern muss. Alle sinnvoll persistierbaren Java-Typen werden unterstützt. Auch mit Vererbung kann MicroStream umgehen. In Bezug auf die Struktur macht MicroStream nur eine einzige Vorgabe: Alle Daten, also die gesamte Datenbank, hängt an einer Root-Instanz. In einer Applikation kann es bei Bedarf auch mehrere Root-Instanzen geben. Der Aufbau des Objektgraphen kann völlig frei modelliert werden. Für ein gutes Objektmodell ist lediglich zu beachten, dass Objekte, auf die man häufiger zugreifen muss, möglichst nahe an der Root angelegt werden, sodass möglichst wenige Nodes durchlaufen werden müssen.

### (Listing 2)

```
public class Order {
    private Product product;
    private double price;
    private int amount;
    ...
}
```

```

}

public class Customer {

    private String firstname;
    private String lastname;
    private String mail;
    private LocalDate dateOfBirth;
    private Boolean active;
    private Set<Order> orders;

    ...

}

```

## Objekte speichern

Um einen Objektgraphen zu persistieren benötigt man nur noch einen Storage-Manager, dem man die Root übergibt. Der Storage-Manager ist die Verbindung zur Storage.

### (Listing 3)

```

<dependency>
  <groupId>one.microstream</groupId>
  <artifactId>filesystem.sql</artifactId>
  <version>04.00.00-MS-GA</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.21</version>
</dependency>

final DataRoot root = new DataRoot();

// create JDBC data source
MysqlDataSource dataSource = new MysqlDataSource();
dataSource.setUrl("jdbc:mysql://host:3306/mydb");
dataSource.setUser("user");
dataSource.setPassword("secret");

// create sql file system
SqlFileSystem sqlFileSystem = SqlFileSystem.New(
    SqlConnector.Caching(SqlProviderMySql.New(dataSource))
);

// start storage manager
EmbeddedStorageManager storageManager = EmbeddedStorage.start(
    root, // root object
    sqlFileSystem.ensureDirectoryPath("microstream-storage") //
    storage directory
);

```

Um bei einem Systemausfall keine Daten zu verlieren, sollten Objektgraph und Storage möglichst immer synchron sein. Das passiert jedoch nicht automatisch, sondern durch den Aufruf einer Store-Methode. Alles weitere erledigt MicroStream. Der Entwickler entscheidet also explizit darüber, ob und wann Änderungen am Objektgraphen persistiert werden. Lediglich eine einfache Regel ist dabei zu beachten: Es muss immer das Objekt gespeichert werden, das sich geändert hat.

Um ein im Objektgraphen neu hinzugefügtes Objekt zu speichern (Insert), wird nicht das neue Objekt selbst, sondern immer dessen Owner gespeichert, indem man die Store-Methode auf dem Owner aufruft. Kommt zum Beispiel bei einem Customer eine neue Order dazu, wird nicht die Order, sondern die Order-List gespeichert in der die Order hinzugefügt wurde.

### (Listing 4)

```

List<Order> order = ...;
order.add(createNewOrder());
storageManager.store(order);

```

Möchte man dagegen ein im Objektgraphen geändertes Objekt speichern (Update), zum Beispiel eine bestehende Order die sich geändert hat, dann muss man das geänderte Objekt selbst speichern, indem man die Store-Methode direkt mit dem geänderten Order-Objekt aufruft.

### (Listing 5)

```

Order order = ...;
order.setAmount(8);
storageManager.store(order);

```

Möchte man bestimmte Members nicht mit speichern, kann man diese als transient markieren. Bei jedem Store wird das zu speichernde Objekt in der Storage hinten angehängt (Append). Das gilt auch bei Updates. Jeder Store ist eine atomare All-or-nothing-Operation, vergleichbar mit einer Transaktion im relationalen Kontext. Wenn der Store erfolgreich war, ist garantiert, dass alle Daten in der Storage vorhanden sind (Strong-Consistency). Falls der Store nicht erfolgreich war, werden bereits persistierte Daten wieder gelöscht, was einem Rollback entspricht. Es ist zu beachten, dass dann auch auf dem Objektgraphen ein Rollback erfolgen muss, um den man sich jedoch im Moment noch selbst kümmern muss. Mit Hilfe von Channels lassen sich I/O Zugriffe parallelisieren.

## Objekte löschen

Für das Löschen von Objekten ist keine spezielle Delete-Methode vorgesehen, denn dazu muss man wie in Java üblich, lediglich sämtliche Referenzen auf das zu löschende Objekt entfernen und diese Änderungen anschließend persistieren. Objekte, die nicht mehr referenziert werden, werden vom Garbage-Collector der Java-VM automatisch aus dem Speicher entfernt.

### (Listing 6)

```

List<Order> orders = ...;
orders.remove(canceledOrder);
storageManager.store(orders);

```

## Objekte laden

Beim initialen Starten der Anwendung wird der gesamte Objektgraph im RAM erzeugt. Dazu werden jedoch lediglich die Objekt-IDs sowie alle Objekte die nicht als Lazy deklariert sind, direkt in den RAM geladen. Falls genügend Hauptspeicher verfügbar ist, spricht nichts dagegen, gleich die gesamte Datenbank in den RAM zu laden. Reicht der Hauptspeicher aber nicht aus, kann man sämtliche Objekte, die man erst später braucht, als Lazy definieren. Dadurch lädt die Engine sie erst dann in den RAM, wenn sie benötigt werden. MicroStream bietet ein sehr elegantes, vollständig objektorientiertes Programmiermodell. Man braucht keine aufwändigen Selects mehr, sondern kann mit gewöhnlichen Get-Methoden direkt auf die Objekte zugreifen, ohne sich darüber Gedanken machen zu müssen, ob sich die Objekte bereits im RAM befinden oder nicht. Liegt das gewünschte Objekt noch nicht im Speicher vor, wird es von der Engine automatisch geladen. Da I/O-Operationen auch mit MicroStream teuer sind, sollte man das Laden einzelner Objekte möglichst vermeiden und stattdessen lieber ganze Teilgraphen laden. Die geladenen Objekte werden automatisch in den Objektgraphen gemerget, sodass man sich nicht mehr mit Objekt-Kopien auseinandersetzen muss.

(Listing 7)

```
public class BusinessYear
{
    private Lazy<List<Turnover>> turnovers;

    public BusinessYear()
    {
        super();
    }

    private List<Turnover> getTurnovers()
    {
        return Lazy.get(this.turnovers);
    }

    public void addTurnover(final Turnover turnover)
    {
        List<Turnover> turnovers = this.getTurnovers();
        if(turnovers == null)
        {
            this.turnovers = Lazy.Reference(turnovers =
                new ArrayList<>());
        }
        turnovers.add(turnover);
    }

    public Stream<Turnover> turnovers()
    {
        final List<Turnover> turnovers = this.getTurnovers();
        return turnovers != null ? turnovers.stream() :
            Stream.empty();
    }
}
```

Objekte, die in den RAM geladen werden, brauchen deutlich mehr Speicher als in der Storage. Das liegt unter anderem an zusätzlichen Metadaten. Um Hauptspeicher zu sparen, kann man Lazy-Referenzen jederzeit auch wieder aus dem Hauptspeicher entfernen, sobald diese nicht mehr benötigt werden. Die Objekt-ID bleibt dabei im Objektgraphen erhalten, sodass das Objekt jederzeit erneut geladen werden kann. Um Ressourcen zu sparen, ist ein Vergleich der verschiedenen Java-VMs empfehlenswert. So verspricht die OpenJ9-JVM einen bis zu 63% geringeren Memory-Footerprint und kürzere Startup-Zeiten. Sie eignet sich damit sehr gut für Projekte mit MicroStream.

(Listing 8)

```
LazyReferenceManager.set(LazyReferenceManager.New(
    Lazy.Checker(
        Duration.ofMinutes(30).toMillis(), // timeout of lazy
        access
        0.75                               // memory quota
    )
));
```

## Was, wenn sich Versionen von Klassen ändern?

Sobald man eine Klasse ändert, indem man Attribute hinzufügt, entfernt oder ändert, passt die Klasse nicht mehr zu den bereits persistierten Objekten in der Storage. IDEs bieten dafür eine Refactoring-Funktion. Die gesamte Datenbank einem Refactoring zu unterziehen, wäre gegebenenfalls sehr zeitaufwändig. MicroStream ändert deshalb die persistenten Daten erst zur Laufzeit, nachdem sie zum ersten Mal geladen wurden. Dabei wird jeder veraltete (legacy) Typ auf einen entsprechenden neuen Typ gemappt. Jede Änderung muss abschließend neu persistiert werden. Die meisten Änderungen werden von einer Heuristik automatisch erkannt. Für komplexere Fälle kann man ein eigenes Type-Mapping definieren.

## Garbage-Collection für die Storage

Damit die Storage nicht durch veraltete Objektversionen unendlich anwächst, werden diese von einem speziellen House-Keeping-Prozess, der wie ein Garbage-Collector auf File-Ebene funktioniert, automatisch aus der Storage entfernt. Dieser Prozess nimmt zudem permanent Optimierungen an den Daten in der Storage vor.

## Zugriff von außen

Mit Hilfe einer REST-Schnittstelle können auch externe Prozesse auf die Storage zugreifen, zum Beispiel für das Reporting. Auch eine graphische Benutzeroberfläche steht zur Verfügung. Mit Hilfe von GraphQL ist auch der Zugriff auf den Objektgraphen von außen möglich.

## Läuft überall, wo eine JVM verfügbar ist

MicroStream läuft grundsätzlich überall dort, wo Java ab Version 8 läuft. Unter anderem auf dem Desktop, Server, in Container-Umgebungen und auf Android. Es kann mit sämtlichen JVM-Sprachen wie Scala und Kotlin genutzt werden, ist prädestiniert für Microservices mit eigener Persistenz und läuft auch in, mit GraalVM oder Quarkus erzeugten, nativen Images. MicroStream ist in der Community Edition völlig lizenzkostenfrei verfügbar. Auch der kommerzielle Einsatz ist damit erlaubt.

## Vorteile für Anwender

Es gibt drei Hauptargumente für den Einsatz von MicroStream:

1. Enorm hohe Performance und Datendurchsatz: Damit sind neue Innovationen, Features und Softwareprodukte möglich, die bislang auf Grund zu geringer Performance nicht denkbar waren. Anwender, die Probleme mit der Performance ihrer derzeitigen Systeme haben, sollten sich MicroStream unbedingt einmal ansehen.
2. Kostenersparnis in der Cloud oder On-Premis: Immer mehr Unternehmen verlagern ihre Systeme in die Cloud. Immer mehr Unternehmen verlieren den Überblick über ihre Systeme in der Cloud und die Kosten laufen zum Teil völlig aus dem Ruder. Insbesondere in der Post-Corona Zeit müssen Unternehmen Kosten reduzieren und ihre Anwendungen kosteneffizient entwickeln und betreiben. Durch die hohe Performance von MicroStream brauchen Anwendungen nur noch einen Bruchteil an Rechenleistung. Matthias Wenzl, Geschäftsführer der Consulting-Firma Caweco und Implementierungspartner des Versicherungskonzerns Allianz, sagt: „Hätten wir unsere Anwendungen wie früher auf Basis von JPA entwickelt, hätten wir teure Cluster-Architekturen betreiben müssen. Mit MicroStream laufen heute die meisten unserer Anwendungen containerisiert auf kleinen bis mittleren virtuellen Server-Instanzen. Wir sprechen von Einsparungen von bis zu 90% Infrastrukturkosten jährlich.“
3. Hoher Entwicklungskomfort: MicroStream verfolgt eine Pure-Java-Philosophie. Das Programmiermodell ist sehr elegant und vollständig objektorientiert. Entwickler müssen ihre Klassen nicht mehr an eine Datenbank anpassen und die Komplexität der Anwendungsarchitektur wird drastisch reduziert. Der gesamte Datenbankentwicklungsprozess wird vereinfacht und beschleunigt.

## Wo ist der Haken?

MicroStream richtet sich gezielt an Java-Entwickler. Gute Kenntnisse in OOP, Design von Objektmodellen sowie Concurrency-Programmierung werden vorausgesetzt. Leider gibt es bislang noch keine SQL-Schnittstelle, über die externe Systeme

auf die Storage zugreifen können. Eine solche wurde aber bereits angekündigt. Der Zugriff von außen ist via REST und GraphQL jetzt schon möglich.

MicroStream bildet das Objektmodell nicht mehr auf ein Datenbankmodell ab, sondern persistiert die serialisierten Daten als Binaries direkt in der Datenbank. Datenbankspezifische Funktionen bleiben damit weitestgehend ungenutzt. Datenbankverfechter werden dem Pure-Java-Konzept eher skeptisch begegnen. Durch die Datenbankabstraktion mit JPA hat man in Java aber ohnehin schon diesen Weg eingeschlagen. Ein Großteil an möglichen Datenbankfunktionen bleibt bereits mit JPA außen vor, unter anderem Stored-Procedures, Stored-Functions, PL/SQL etc. Während Datenbankhersteller propagieren, Businesslogik direkt in der Datenbank auszuführen, wird mit JPA die Businesslogik in den Application-Layer verlegt und für uns selbstverständlich in Java implementiert. Nur ein wenig Businesslogik gestatten wir in der Datenbank zu laufen, z.B. Concurrency-Management. Genau das bereitet uns dann oft Probleme und wir müssen uns am Ende des Tages trotzdem wieder selbst darum kümmern. MicroStream geht diesen Weg nun konsequent zu Ende, indem man die Datenbank ausschließlich als reine Daten-Storage betrachtet und die Businesslogik vollständig in den Java-Server verlegt. Dieser Paradigmenwechsel muss verstanden werden, um die Vorteile dieses Konzepts erfassen zu können.

## Fazit:

MicroStream ist prädestiniert für Microservices, die ihre eigene Persistenz benötigen. Aber auch in großen monolithischen Anwendungen hat sich MicroStream als Alternative zu JPA bereits bewährt. Damit ist das Einsatzfeld enorm breit. Es gibt bereits zahlreiche Applikationen im Umfeld von ERP- und Branchenlösungen, meist individuelle Zusatzmodule, die ebenfalls eine eigene Persistence benötigen. Auch für Anwender von Graph-Datenbanken kann MicroStream eine sehr interessante Alternative sein.

Wer erst einmal etwas Praxiserfahrung sammeln möchte, kann MicroStream überall dort einsetzen, wo heute zum Beispiel H2-Datenbanken eingesetzt werden, unter anderem für Unit-Tests und die Entwicklung von Prototypen. Wiederum prädestiniert ist MicroStream als Persistence für Android-Apps. Hierfür bietet MicroStream einen Connector zu SQLite.

MicroStream ist als Community-Edition frei verfügbar unter [www.microstream.one](http://www.microstream.one).

## Quellen:

- 1 [www.microstream.one](http://www.microstream.one)
- 2 <http://bit.ly/github-g2>

#JAVAPRO #Framework #Lombok

# Boilerplate-Code minimieren mit Lombok

Projekt Lombok versteckt auf elegante Weise sogenannten Boilerplate-Code. Getter, Setter und all das, was Entwickler im Leben ungefähr 2 Millionen Mal per IDE generieren – all das soll mit Lombok wegfallen. Ist die Einstiegshürde erst einmal überwunden, funktioniert das wirklich.

## Autor:

Christian Grobmeier arbeitet selbst, ständig und gerne. Am liebsten mag er irgendwas mit Spring oder Solr. Er ist auch kein JavaScript-Kostverächter und treibt sich gerne mit Angular oder React rum. Hin und wieder greift er sogar zur Ruby und PHP Keule. Und wenn alles nichts mehr hilft, schreibt er ein Buch.



Webseite: <https://grobmeier.solutions>  
Twitter: <https://twitter.com/grobmeier>  
LinkedIn: <https://www.linkedin.com/in/grobmeier>  
GitHub: <https://github.com/grobmeier>  
E-Mail: [cg@grobmeier.de](mailto:cg@grobmeier.de)

**W**er erinnert sich eigentlich noch an die vielen Java-Enterprise-Beans der Version 1 oder 2, die unsäglichen Schmerzen, die einem damals bei professioneller Entwicklung widerfahren sind und die große Erleichterung die schließlich mit POJOs kam. Plain-Old-Java-Objects: Endlich testbarer Code! Code, den man auch verstehen konnte. Es gab nur zwei Regeln die man akzeptieren musste:

1. Alle Klassen müssen einen parameterlosen Konstruktor haben.
2. Alle Felder müssen Getter und Setter haben.

Verglichen mit dem was die Entwicklergemeinschaft vorher hatte, war das kein Problem. Leider müssen noch die Methoden **toString**, **equals** und **hashCode** überschrieben werden. (Listing 1) zeigt eine einfache Klasse mit zwei Konstruktoren und den Boilerplate-Code, wie er üblicherweise erwartet wird. Glücklicherweise hilft jeder gängige Editor diesen Code per Shortcuts zu generieren. Will nicht jeder diese Art von Code so schnell wie möglich hinter sich bringen? Mittlerweile hat man sich allerdings daran gewöhnt.

Aber warum erzeugt man den gleichen Code immer und immer wieder und versioniert ihn in unseren Source-Repositories? Wie stellt man sicher, dass zum Beispiel **equals** neu generiert wird, wenn beispielsweise ein neues Feld hinzugefügt wird? Muss man eigentlich Boilerplate-Code testen, wenn man ihn mit der IDE generiert? Ist es sinnvoll, Code der immer wieder auf seine Art gleich ist, zu duplizieren? Jede Code-Zeile ist eine Entscheidung die man bewusst treffen sollte. Im Fall von **get** und **set** mangelt es aber an guten, durchsetzungsfähigen Lösungen. Warum nicht einfach alles **public** machen? Puristen würden vielleicht laut vor Entsetzen aufschreien, andere dagegen sogar frohlocken.

Mit Lombok kann Boilerplate-Code endgültig ins Reich der Dinosaurier verbannt werden und der moderne Entwickler kann sich nun auf das konzentrieren, was wirklich wichtig ist.

(Listing 1)

```
public class Category {
    private Long id;
    private String name;
    private String slug;

    public Category () {
    }

    public Category (String name) {
        this.name = name;
    }

    public Long getId() {
        return this.id;
    }
}
```

```
public String getName() {
    return this.name;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;
    Category category = (Category) o;
    return Objects.equals(id, category.id) &&
        Objects.equals(name, category.name) &&
        Objects.equals(slug, category.slug);
}

@Override
public int hashCode() {
    return Objects.hash(id, name, slug);
}

@Override
public String toString() {
    return "Category{" +
        "id=" + id +
        ", name='" + name + '\'' +
        ", slug='" + slug + '\'' +
        '}';
}
}
```

## Lombok werden

Lombok zu verwenden muss eine bewusste Entscheidung sein und im Team getroffen werden. Jeder Entwickler muss dafür ein Plugin für seine IDE installieren. Auf der Lombok-Webseite finden sich die Plugins zum Download, samt Installationsanleitung. Für manche Anwender stellt bereits das eine große Hürde dar und leider ist das noch nicht alles. (Listing 2) zeigt die Abhängigkeit, die man benötigt, um die Lombok-Annotations in seinem Projekt verwenden zu können. Zudem zeigt (Listing 3), wie man Lombok via Maven in seine Compiler-Konfiguration einbindet, damit man auch außerhalb der IDE lauffähige Pakete erzeugen kann. Etwas Arbeit muss man also in seine Umgebung stecken, bevor es los gehen kann. Bei einem 50 Personenteam kann das also auch schon etwas dauern.

(Listing 2)

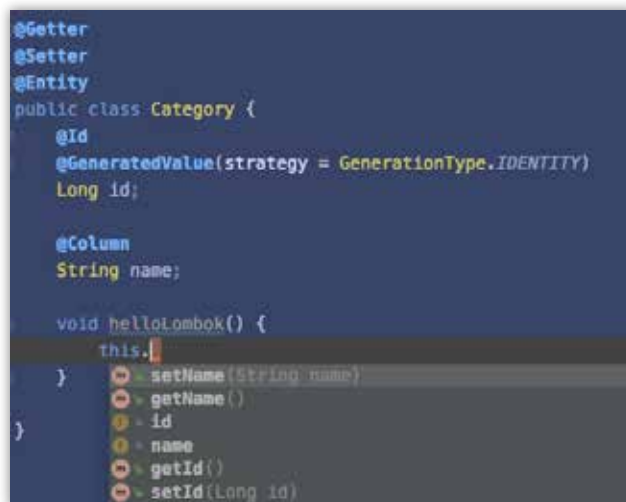
```
<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.10</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

(Listing 3)

```
<build>
  <plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>8</source>
      <target>8</target>
      <annotationProcessorPaths>
      <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.10</version>
      </path>
      </annotationProcessorPaths>
    </configuration>
  </plugin>
</plugins>
</build>
```

## Nie mehr Getter und Setter

Wenn einmal alles aufgesetzt ist, kann die **Category** Klasse leicht neu geschrieben werden (Listing 4). Eine weitere Aktion ist nicht notwendig. Das Lombok-Plugin erzeugt im Hintergrund bereits Code für die Get- und Set-Methoden. Die IDE kann diese Methoden auch bereits ganz normal verwenden. Der Entwicklungsfluss ist also nicht gestört. Kein separates Kompilieren oder Knöpfchen klicken. Alles ist einfach da (Abb. 1).



Get und Set sind einfach da (Abb. 1)

(Listing 4)

```
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Category {
    private Long id;
    private String name;
    private String slug;
```

Selbstverständlich gibt es auch Annotation für **equals**, **hashCode** und **toString**, nämlich **@ToString** und **@EqualsAndHashCode**. Wer gleich das volle Pauschalpaket möchte, verwendet nicht diese vier Annotations, sondern lediglich **@Data**, was alle vier Optionen beinhaltet (Listing 5).

(Listing 5)

```
import lombok.Data;

@Data
public class Category {
    private Long id;
    private String name;
    private String slug;
```

Es ist außerdem möglich, die **@Getter** und **@Setter** Annotations über Felder zu schreiben, womit dann nur für diese Felder Getter und Setter generiert werden.

## Wie man in die Glaskugel blickt

Lombok wird oft als Magie bezeichnet, was es irgendwie auch ist. Wer beispielsweise **@Builder** verwendet, wird feststellen, dass plötzlich eine gesamte API zum Bauen eines Objekts bereitgestellt wird. Übertragen auf das **Category** Beispiel, könnte also Code angewendet werden wie in (Listing 6).

(Listing 6)

```
@Data
@Builder
public class Category {
    private Long id;
    private String name;
    private String slug;
}

class Lombok {
    void hello() {
        Category.builder().name("Hello").build()
    }
}
```

Was geschieht da eigentlich? Wie kann man diese Lombok'sche Magie überhaupt verstehen? Und was, wenn das Team wieder weggehen möchte von Lombok? Auch das ist keine große Sache, denn dafür gibt es Delombok. Delombok ist verfügbar für einzelne Klassen über das IDE-Plugin und als Ant- oder Maven-Task, um gesamte Projekte aus der Lombok-Abhängigkeit zu befreien. Wer auf die Klasse von (Listing 6) rechts-klickt und bei IntelliJ mittels Refactoring die Option **Delombok** auswählt, wird feststellen, dass die Lombok-Annotations beseitigt werden. Stattdessen sieht man den Code, den Lombok generiert hat bzw. generieren würde. Im Falle der Builder-Annotation ist das der Code aus (Listing 7).

(Listing 7)

```
public static class CategoryBuilder {
    private Long id;
    private String name;

    CategoryBuilder() {}

    public Category.CategoryBuilder id(Long id) {
        this.id = id;
        return this;
    }

    public Category.CategoryBuilder name(String name) {
        this.name = name;
        return this;
    }

    public Category build() {
        return new Category(id, name);
    }

    public String toString() {
        return "Category.CategoryBuilder(id=" + this.id + ",
name=" + this.name + ")";
    }
}
```

## Weitere wichtige Annotations

Das Prinzip Lombok ist schnell erklärt und auch die Annotations sind schnell überblickt. Derzeit gibt es 17 stabile Features und mehrere experimentelle. Eines der wichtigsten Features ist **@Log4j2**. Es gibt diese Annotation auch für andere Logging-Frameworks. Lombok erzeugt dann einen Ready-to-use Logger für eine Klasse (Listing 8). Die meisten Entwickler haben sicher schon mehrmals eine Logger-Instanz erzeugt und dabei die falsche Klasse als Argument verwendet. Typische Fehler bei sich ständig wiederholendem Code sind oft auf Copy und Paste zurückzuführen.

(Listing 8)

```
private static final Logger log = org.apache.logging.log4j.LogManager.getLogger(Category.class);
```

Die Lombok-Annotation **@Close** ist ebenfalls sehr nützlich. Das korrekte Schließen eines **InputStream** wird mit **@Close** sehr einfach. Generiert wird Code, der im **finally** Block den **InputStream** schließt. Wer also (Listing 9) schreibt, erhält letzten Endes (Listing 10).

(Listing 9)

```
public static void main(String[] args) throws IOException {
    @Cleanup InputStream in = new FileInputStream(args[0]);
    @Cleanup OutputStream out = new FileOutputStream(args[1]);

    // do something
}
```

(Listing 10)

```
public static void main(String[] args) throws IOException {
    InputStream in = new FileInputStream(args[0]);
    try {
        OutputStream out = new FileOutputStream(args[1]);
        try {
            // do something
        } finally {
            if (out != null) {
                out.close();
            }
        }
    } finally {
        if (in != null) {
            in.close();
        }
    }
}
```

Zudem gibt es **@eval** und **@var**. Damit ist es möglich, statt dem Referenz-Typen einfach **var** (Variable) oder **val** (final) anzugeben. Allerdings verliert man dann eine wichtige Information, die man möglicherweise zum zügigen Coden benötigt. Typen sind prinzipiell nicht schlecht. Ob jedoch alles was Lombok bietet auch verwendet werden sollte, bleibt dem Team überlassen. Weiter werden Immutability, Synchronized oder die automatische Generierung von Konstruktoren der verschiedensten Art unterstützt.

## Fazit:

Ob Code-Generierung gut oder eher schlecht ist, muss jeder selbst für sich bzw. das Team entscheiden. Manche Entwickler meinen, dass auf diese einfache Art und Weise Getter und Setter zu generieren dazu verleiten würde, dass man nicht mehr über Datenkapselung nachdenkt und einfach prinzipiell alles via Get bzw. Set erreichbar macht. Es ist die Frage, ob Getter und Setter dann generell eine gute Idee sind. Ein Argument dagegen wäre, dass Entwickler die sich Getter und Setter von der IDE generieren lassen, das ja genauso gedankenlos machen können wie Entwickler die **@Getter** und **@Setter** schreiben. Die persönliche Erfahrung des Autors ist, dass der Code mit Lombok eleganter, kürzer und lesbarer geworden ist und die Einführung von Lombok problemlos war. Wer davon ausgeht, dass generierter Code generell besser und aktueller ist, der wird sich eher auf diese Magie einlassen als jemand der den gesamten Code sehen möchte.

## Quellen:

1 Projekt Lombok - <https://bit.ly/lombok-p1>

#JAVAPRO #Streaming #Kafka

# Kafka-Connect - Drittsysteme an Kafka anbinden

Kafka ist eine verteilte Streaming-Plattform die verschiedene Nutzungsszenarien unterstützt. Sie kann als Messaging- oder Speichersystem eingesetzt werden und Datenströme transformieren. Dieser Artikel beschreibt, wie Sie mit Kafka-Connect Drittsysteme wie Redis an Kafka anbinden können.

**K**afka bietet eine flexible Grundlage, um darauf komplexe Datenverarbeitungsarchitekturen zu bauen. Eine wichtige Frage bei diesen Architekturen ist, wie die Daten in die Plattform gelangen und wie sie diese wieder verlassen. Hier kommt Kafka-Connect ins Spiel, ebenfalls eine quelloffene Komponente von Kafka. Kafka-Connect ist ein Framework mit dem man externe Systeme, wie zum Beispiel Datenbanken, Key-Value-Speicher, Suchindizes oder Dateisysteme an Kafka anschließen kann. Für Kafka-Connect gibt es bereits zahlreiche vorgefertigte Konnektoren, wie zum Beispiel für Splunk oder Elasticsearch<sup>1</sup>. Diese Konnektoren teilen sich in zwei Gruppen auf: Sources und Sinks – also Quellen und Senken. Quellkonnektoren können zum Beispiel ganze Datenbanken auslesen und Änderungen in der Datenbank dann in Kafka-Topics veröffentlichen. Ein anderer Anwendungsfall ist das Sammeln von Metriken von Microservices, um sie dann dem Stream-Processing mit geringer Latenz zur Verfügung zu stellen und sie dann auswerten zu können. Sinks bringen die Daten aus den Kafka-Topics dann zum Beispiel in Suchindizes wie Elasticsearch oder in Batch-Systeme wie Hadoop, zur späteren Offline-Verarbeitung.

Nun könnte man meinen, das alles kann auch ohne ein zusätzliches Framework realisiert werden, indem man über das API von Kafka die Daten nach Kafka schreibt oder daraus liest. Worin liegen also die Vorteile von Kafka-Connect? Kafka-Connect stellt eine API bereit, die es ermöglicht, dass ein Konnektor nach einem Fehler an der Stelle fortfährt, an der er unterbrochen worden ist. Dabei wird der Konnektor automatisch neu gestartet und gegebenenfalls auch auf einen neuen Knoten innerhalb des Kafka-Clusters gestartet. Zusätzlich macht es die API einfacher mehrere Worker parallel zu starten, die dasselbe Quell- bzw. Zielsystem versorgen.

Auch wenn bereits viele verschiedene Konnektoren verfügbar sind, kann es dennoch vorkommen, dass für ein Legacy-System

oder eine Eigenentwicklung kein Konnektor zur Verfügung steht oder dass die benötigte Funktionalität nicht angeboten wird. Das stellt jedoch kein Hindernis für den Einsatz von Kafka und Kafka-Connect dar. Das Implementieren eines eigenen Konnektors ist relativ einfach und ermöglicht es praktisch jedes System mit Kafka zu verbinden. Wie bereits erwähnt unterscheidet Kafka-Connect zwischen Quellen und Senken, je nachdem ob Daten nach Kafka hinein oder hinaus bewegt werden. Es gibt also zwei verschiedene Interfaces, die je nach Anwendungsfall implementiert werden müssen. Um eine Quelle anzuschließen müssen

`org.apache.kafka.connect.source.SourceConnector` und `org.apache.kafka.connect.source.SourceTask`

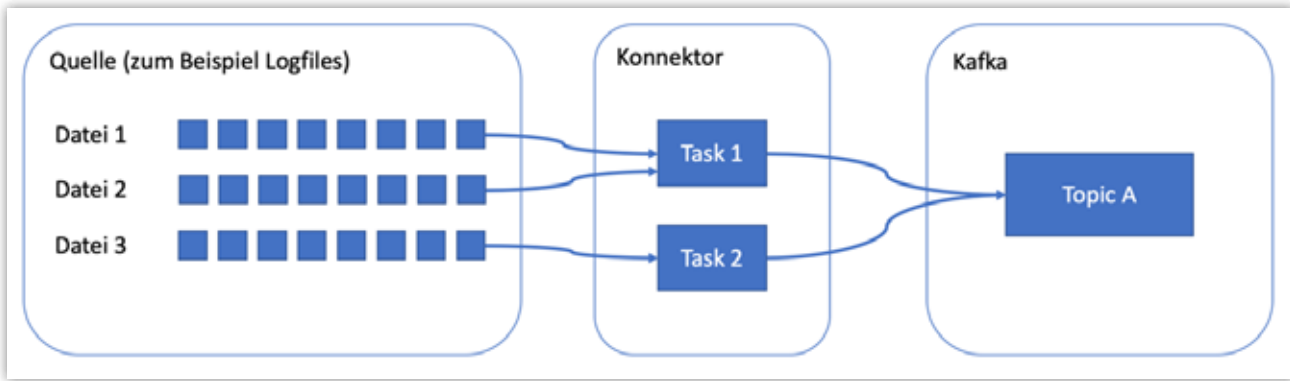
implementiert werden. Für eine Senke sind es `SinkConnector` und `SinkTask`. Ein `JDBCSourceConnector` würde zum Beispiel die Daten einer JDBC-kompatiblen Datenbank nach Kafka importieren und ein `HDFS SinkConnector` würde den Inhalt eines Kafka-Topics nach `HDFS` exportieren. Dabei ist zu beachten, dass der Konnektor selbst nicht für das Kopieren der Daten

## Autor:

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der Firma sidion. Er entwickelt seit über zwölf Jahren Geschäftsanwendungen in Java, Kotlin und PHP für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen.



Webseite: <https://www.sidion.de>  
E-Mail: [nicolai.mainiero@sidion.de](mailto:nicolai.mainiero@sidion.de)



Schemabild wie ein Quellkonnektor mit Kafka zusammenarbeitet (Abb. 1)

verantwortlich ist. Er erhält beim Starten von Kafka-Connect seine Konfiguration und leitet daraus eine Konfiguration für die Tasks ab. Die Konfiguration des Konnektors enthält zum Beispiel den Dateipfad oder die JDBC-URL und wie viele Tasks erzeugt werden sollen. Bei einem Dateikonnektor könnte zum Beispiel konfiguriert werden, dass es zwei Tasks geben darf. Der Konnektor wäre dann dafür verantwortlich, nicht nur den Dateipfad an den Task zu übergeben, sondern auch welcher Bereich aus der Datei gelesen werden soll, so dass es zwei Tasks geben kann, die sich nicht gegenseitig beeinflussen. Diese Tasks werden dann von Kafka-Connect-Workern verarbeitet<sup>2</sup>. Auch bei diesen Tasks wird wieder zwischen Quelle und Senke unterschieden. Der Konnektor kann bei Bedarf auf Änderungen des externen Systems reagieren und den Task geeignet rekonfigurieren.

Hat der Task nun die Konfiguration eines Konnektors bekommen, ist dieser dafür verantwortlich die Daten in das konfigurierte Kafka-Topic zu schreiben oder aus einem Topic zu lesen. Dabei müssen die Daten ebenfalls in verschiedene Partitionen aufgeteilt werden, analog dazu, wie Kafka Topics strukturiert. Innerhalb jeder Partition sind die Daten wieder eine geordnete Sequenz mit entsprechenden Offsets. Diese Aufteilung zu finden kann je nach Anwendung sehr einfach oder aber aufwändig sein. Wenn zum Beispiel Log-Files (Abb. 1) gelesen werden, kann jede einzelne Datei einer Partition entsprechen, jede Zeile ist ein Eintrag und der Offset ist einfach die Position innerhalb der Datei. Bei einem JDBC-Konnektor kann es da schon schwieriger werden. Eine Möglichkeit wäre, jede Tabelle auf ein Topic abzubilden. Innerhalb des Topics kann dann der Zeitstempel der letzten Abfrage genutzt werden, um eine Reihenfolge zu definieren.

## Redis-Konnektor

Wie wird nun also ein Konnektor konkret implementiert? Als Quellsystem soll ein Redis-Cache dienen, der über das Pub/Sub-System angeschlossen wird. Das ermöglicht es zu zeigen, welche Schnittstellen für Kafka-Connect implementiert werden müssen und welche Funktionen dadurch nutzbar werden, ohne dass die Komplexität zu groß wird. Damit sich der Task mit Redis verbinden kann, benötigen wir Informationen wie Redis erreichbar ist. Diese Informationen werden von Kafka-Connect

an den Redis-Konnektor weitergegeben (Listing 1). Es sind also folgende Eigenschaften notwendig: Hostname **host** oder IP-Adresse des Redis-Servers, die Portnummer **port**, unter der der Redis-Server erreichbar ist, der Kanal **channel** auf den via Pub/Sub gelauscht wird und Name des Topics **topic**, in das die Informationen geschrieben werden sollen, die von Redis empfangen werden. Dieses Beispiel ist stark vereinfacht, um die wesentlichen Implementierungsschritte zu zeigen. Ein Redis-Konnektor mit vollem Funktionsumfang würde nicht nur einen einzelnen Channel beobachten, sondern mit Hilfe von **PUBSUB CHANNELS** alle aktiven Channels von Redis abfragen und für jeden Channel dann einen Task erzeugen, der dann in sein eigenes Topic schreibt. Eines der Designziele von Kafka-Connect ist, dass die Konnektoren mit großen Mengen an Daten umgehen können, bei gleichzeitig einfacher Konfiguration.

### (Listing 1)

```
public class RedisSourceConnector extends SourceConnector {
    private String topic;
    private String host;
    private int port;
    private String channel;
}
```

Damit diese Konfigurationsparameter beim Anlegen eines Konnektors auch übergeben werden können, muss die **start** Methode geeignet überschrieben werden (Listing 2). Diese Methode bekommt eine Map aus der dann die Konfigurationseinstellungen für den Redis-Konnektor extrahiert werden.

### (Listing 2)

```
@Override
public void start(final Map<String, String> props) {
    final AbstractConfig parsedConfig = new AbstractConfig(CONFIG_DEF,
        props);
    topic = parsedConfig.getString(TOPIC_CONFIG);
    host = parsedConfig.getString(HOST);
    port = parsedConfig.getInt(PORT);
    channel = parsedConfig.getString(CHANNEL)
}
```

Die generische Map wird zunächst mit Hilfe einer Konfigurationsdefinition **CONFIG\_DEF** in eine **AbstractConfig** umgewandelt, anschließend werden dann die einzelnen Konfigurationsparameter extrahiert. Die **CONFIG\_DEF** kann im veröffentlichten Quellcode<sup>3</sup> eingesehen werden. Diese Konfigurationsdefinition wird zum Beispiel von dem Confluent-Control-Center verwendet, um eine Konfigurationsoberfläche für den Konnektor anzuzeigen.

## Redis-Task

Damit Kafka-Connect nun einen Source-Task erstellen kann, müssen noch zwei weitere Methoden überschrieben werden. Zum einen muss mitgeteilt werden, welche Klasse den Task für diesen Konnektor übernimmt und wie dieser Task konfiguriert werden muss. Man erkennt, dass in dem **RedisSourceConnector** in erster Linie die Konfiguration für die Tasks aufbereitet wird, um dann zu entscheiden wie viele Tasks erzeugt werden sollen. Um den Konnektor robuster zu machen, könnte man hier eine Validierung einbauen und prüfen ob mit den übergebenen Parametern wie **HOST** und **PORT** Redis überhaupt erreichbar ist.

Konkret müssen die Methoden **taskClass** und **taskConfigs** überschrieben werden. In **taskClass** wird die Klasse definiert, die den Task dann tatsächlich ausführt, also **RedisSourceTask**. In **taskConfigs** muss die Konfiguration für die Tasks erstellt werden (**Listing 3**). Je nach Konnektor kann dies mehr oder weniger aufwendig sein. In diesem einfachen Beispiel wird die Konfiguration 1:1 an den Task weitergegeben. Das bedeutet auch, dass genau ein Task erzeugt wird. Falls der Anwendungsfall es zulässt, dass mehr als ein Task erzeugt werden kann muss darauf geachtet werden, dass nicht mehr als im Parameter von **task Configs** angegeben erzeugt werden.

### (Listing 3)

```
@Override
public List<Map<String, String>> taskConfigs(final int maxTasks) {
    final ArrayList<Map<String, String>> configs = new ArrayList<>();
    final Map<String, String> config = new HashMap<>();
    config.put(HOST, host);
    config.put(TOPIC_CONFIG, topic);
    config.put(PORT, String.valueOf(port));
    config.put(CHANNEL, channel);
    configs.add(config);
    return configs;
}
```

Damit ist der **RedisSourceConnector** fertig implementiert. Der zugehörige **RedisSourceTask** implementiert wie die Daten kopiert werden. Doch betrachten wir zunächst, wie Kafka-Connect mit diesem Task interagiert. Das **SourceTask** Interface definiert die **poll** Methode, die von den Worker-Prozessen von Kafka-Connect wiederholt aufgerufen wird, um eine Liste von Einträgen abzurufen und diese dann in das entsprechende Topic zu schreiben. Dabei ist zu beachten, dass Kafka-Connect für jeden

Task einen dedizierten Thread zur Verfügung stellt. Das bedeutet die Implementierung von **poll** darf blockieren. Das ermöglicht es zunächst mit einer einfachen Implementierung zu starten und diese dann später zu optimieren. Bevor nun Kafka-Connect **poll** aufrufen kann, muss der Task gestartet werden. Dies geschieht indem die **start** Methode aufgerufen wird (**Listing 4**). Dort wird nun zunächst eine Verbindung zum Redis-Server aufgebaut (1), um dann den **RedisSourceTask** als Listener bei dem Pub/Sub-System von Redis zu registrieren (2).

### (Listing 4)

```
@Override
public void start(final Map<String, String> props) {
    LOG.info("Start new RedisSourceTask and open connection to Redis.");
    final String host = props.get(RedisSourceConnector.HOST);
    final int port = Integer.valueOf(props.get(RedisSourceConnector.PORT));
    final String channel = props.get(RedisSourceConnector.CHANNEL);
    topic = props.get(RedisSourceConnector.TOPIC_CONFIG);
    redisClient = RedisClient.create(RedisURI.Builder.redis(host, port).build());
    connection = redisClient.client.connectPubSub(); // 1
    connection.addListener(this); // 2
    final RedisPubSubCommands<String, String> sync = connection.sync();
    sync.subscribe(channel); // 3
    messageQueue = new ConcurrentLinkedDeque<>(); // 4
}
```

Danach wird der konfigurierte Kanal abonniert (3) und eine **Deque** (4) initialisiert (**Listing 4**). Eine **Deque** ist eine doppelseitige Warteschlange, mit der eine Verbindung zwischen der **poll** Methode und dem Event-Listener hergestellt werden kann.

### (Listing 5)

```
@Override
public void message(final String channel, final String message) {
    LOG.info("Recieved message from redis.");
    final Map<String, ?> sourcePartition = ImmutableMap.of("channel", channel);
    final Map<String, ?> sourceOffset = ImmutableMap.of();
    final SourceRecord record = new SourceRecord(sourcePartition, sourceOffset, topic, SchemaBuilder.STRING_SCHEMA, message); //1
    messageQueue.add(record); //2
}
```

Da der **RedisSourceTask** als Listener bei dem Redis-Client registriert wurde, muss nun auch das entsprechende Interface implementiert werden (**Listing 5**). Relevant ist dabei die **message** Methode, die für das Verarbeiten von Nachrichten verantwortlich ist, die aus dem Pub/Sub-System von Redis empfangen werden. Hier wird ein **SourceRecord** erstellt (1) und dann mit **add** an das Ende der Warteschlange angehängt (2). Der **SourceRecord** ist

ein von Kafka-Connect zur Verfügung gestellter Datentyp, der einen Eintrag in ein Kafka-Topic beschreibt. Dieser enthält neben den Daten auch das Topic, in das die Daten geschrieben werden sollen und Informationen woher diese Nachricht kam. Die **sourcePartition** beschreibt in diesem Fall den Ursprung der Daten - hier der Kanal, der belauscht wird. Mit dem **sourceOffset** kann angegeben werden, bis zu welcher Stelle man bereits von dem Quellsystem gelesen hat, um bei einem Neustart des Tasks wieder an dieser Stelle fortfahren zu können. Bei dem Redis-Task ist dieser Wert nicht gesetzt, da beim Pub/Sub-System die Nachrichten nicht zwischengespeichert werden, sondern nur an alle Empfänger gesendet werden, die im Moment des Sendens verbunden sind. Wenn eine Datei eingelesen wird, kann zum Beispiel die Zeilennummer als Offset verwendet werden, die gerade gelesen worden ist. Zusätzlich kann noch ein Schema mit übergeben werden, also eine Beschreibung welches Format die Nachricht hat. Dadurch ist es möglich, verschiedene Nachrichten in verschiedenen Formaten an Kafka zu senden und beim Lesen wieder anhand des Schemas zu rekonstruieren.

(Listing 6)

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
    final List<SourceRecord> records = new ArrayList<>(1024);
    int count = 0;
    SourceRecord record;
    while (count <= 1024 && null != (record = messageQueue.poll()))
    { //1
        records.add(record);
        count++;
        LOG.info("Answering poll() with {} records.", count);
    }
    return records;
}
```

Die so erzeugten Einträge in der Warteschlange werden nun durch wiederholtes Aufrufen der **poll** Methode von den Kafka-Connect Workern angeholt und in das angegebene Topic geschrieben. Dabei wird die Warteschlange abgefragt bis sie leer ist und dann an Kafka-Connect zurückgegeben. Falls ein Quell-System angeschlossen wird, das ein Quittieren der gelesenen Nachrichten unterstützt, beispielsweise eine Message-Queue, kann dies in den **commit** oder **commitrecord** Methoden realisiert werden.

## Paketierung des Konnektors

Damit ist die Implementierung eines einfachen Quellkonnektors vollständig. Um diesen Konnektor nun in Kafka-Connect zu benutzen, muss ein Archiv erstellt werden, das sowohl die Implementierung enthält als auch alle relevanten Third-Party-Bibliotheken. Dies kann mit dem Maven-Assembly-Plugin<sup>4</sup> erreicht werden (Listing 7). Dazu packt man das Artefakt (1) inklusive seiner Runtime-Dependencies (2) in ein eindeutig benanntes Zip-Archiv.

(Listing 7)

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
    <id>zip</id>
    <includeBaseDirectory>true</includeBaseDirectory>

    <formats>
        <format>zip</format>
    </formats>
    <files> <!-- 1 -->
        <file>

            <source> ${project.build.directory}/${project.artifactId}-${project.version}.jar</source>
            <outputDirectory></outputDirectory>
        </file>
    </files>
    <dependencySets> <!-- 2 -->
        <dependencySet>
            <excludes>

                <exclude>${project.groupId}:${project.artifactId}:jar:*</exclude>
            </excludes>
            <scope>runtime</scope>
        </dependencySet>
    </dependencySets>
</assembly>
```

Das so erzeugte Archiv enthält dann alles notwendige, um den Konnektor zu distribuieren. Die Installation in Kafka-Connect gestaltet sich dann auch sehr einfach. Das Archiv wird entpackt und das neu entstandene Verzeichnis wird in den Plug-in-Pfad von Kafka-Connect kopiert. Beim nächsten Starten steht der Konnektor dann zur Verfügung.

## Fazit:

Um große Datenmengen einfach nach Kafka oder aus Kafka heraus zu bewegen, kann man entweder die Kafka-API direkt benutzen oder auf Kafka-Connect zurückgreifen. Kafka-Connect bietet mit seiner API einen wesentlich einfacheren Weg, um mit Kafka zu interagieren und denkt dabei auch schon an Ausfallsicherheit, Fehlertoleranz und Performance. Die API ist leicht verständlich und ermöglicht es sehr schnell Konnektoren für beliebige Drittsysteme zu entwickeln.

## Quellen:

- 1 <https://bit.ly/kafka-k1>
- 2 <https://bit.ly/kafka-k2>
- 3 <https://bit.ly/quellcode>
- 4 <https://bit.ly/plugin-p4>  
<https://bit.ly/github-p5>

#JAVAPRO #Framework #JavaFX

# Bessere Java-Desktop Deployments

Durch gravierende Änderungen bei den Java-Versionen ab Java 9, hat sich vor allem die Entwicklung klassischer Java-Desktop-Anwendungen stark verändert. Die Problematik ist sehr vielschichtig. Doch mit Java-Version-14 gibt es jetzt endlich anwenderfreundliche Lösungen für die Auslieferung von Java-Desktop-Applikationen.

## Autor:

Sven Reinck programmiert seit seinem sechsten Lebensjahr. Da war es nicht verwunderlich, dass er später Informatik studiert hat. Im Studium lernte er die Sprache Java, welche ab diesem Zeitpunkt seine bevorzugte Programmiersprache war. 2017 entdeckte er dann den Nachfolger Kotlin, den er seitdem immer mehr schätzen gelernt hat.



Jetzt führt er mit seinem Coaching nicht nur Schulungen in Kotlin durch, sondern bietet auch eine längerfristige Begleitung, damit die Einführung von Kotlin in Firmen reibungslos funktioniert.

Im Jahr 2014 war die Java-Welt noch in Ordnung. Drei Jahre zuvor kam nach langer Zeit mit Java 7 endlich wieder eine neue Version. Dann wurde mit Java 8 eine Version mit vorher nie dagewesenen Neuerungen vorgestellt. Lambdas und Streams sollten die Art wie wir Java programmieren für immer verändern. Java 9 führte im Jahr 2017 ein Modulsystem mit dem Codenamen Jigsaw ein. Dieses Modulsystem war sehr umstritten und führte dazu, dass der Release verschoben werden musste. Dieses Modulsystem ist wohl der Grund dafür, warum viele Entwickler sich geweigert haben auf Version 9 umzusteigen. Java 9 war gleichzeitig auch der Startschuss für eine neue Release-Strategie von Oracle. Von nun an sollte jedes halbe Jahr eine neue Hauptversion erscheinen. Und mit jeder neuen Version wurden weitere starke Einschnitte in das Java-System vorgenommen, so dass sich in relativ kurzer Zeit sehr viele Änderungen ergaben, die mit der Abwärtskompatibilität gebrochen haben.

### Auswirkung auf Server-Anwendungen

Bis Java 8 beinhaltete die Java-Laufzeitumgebung auch einige Komponenten, die für den Betrieb eines Servers notwendig sind. Zu den Einschnitten, die mit Java 9 begannen, gehört unter anderem diese Komponenten zu entfernen, weswegen vor allem Nutzer solcher Server mit einigen Schwierigkeiten zu rechnen haben, wenn sie ihre Java-Version aktualisieren wollen. Viele Anbieter von Server-Software haben sich lange Zeit geweigert, Versionen über Java 8 zu unterstützen. Je nachdem in welcher Umgebung der Anwender die Software ausführt, sind diese Komponenten vorhanden oder eben nicht. Das ist nicht das was mit Java ursprünglich geplant war, doch es ist ein Teil einer durchaus notwendig gewordenen Aufräumaktion. Da Oracle den Support für Java 8 nun nach und nach einstellt, werden sich alle Entwickler, die bisher noch mit dem Umstieg gewartet haben, früher oder später mit den neueren Java-Versionen auseinandersetzen müssen.

### Auswirkung auf Desktop-Anwendungen

Mit JavaFX kam 2008 ein neues GUI-Framework, welches das inzwischen veraltete Swing ablösen sollte. Doch mit Java 11 wurde JavaFX aus der Java-Laufzeitumgebung entfernt, was für ein Desktop-System besonders unglücklich ist. Java basierte immer auf dem Konzept, dass in der Java-Laufzeitumgebung bereits alle plattformspezifischen Bibliotheken vorhanden sind, die eine Anwendung braucht. Da man sich aber jetzt nicht mehr darauf verlassen kann, dass JavaFX in der vom Anwender installierten Laufzeitumgebung bereits vorhanden ist, müssen Entwickler, die eine FX-Desktop-Anwendung ausliefern wollen, die JavaFX-Bibliotheken selbst mitliefern. Denn als sehr systemnahe Grafikbibliothek, eine der größten Stärken von JavaFX gegenüber dem früheren Swing, ist JavaFX für jede Plattform anders. Das heißt, Entwickler einer Desktop-Anwendung müssen jetzt für jede Plattform eine eigene Version bauen, auf der die Anwendung laufen soll. Das aber widerspricht dem ursprünglichen Gedanken

Write-once-run-everywhere, was heißt, dass ein Java-Programm auf allen von Java unterstützten Plattformen läuft, ohne dass dafür zusätzliche Anpassungen nötig sind. Wer jetzt jedoch eine solche plattformunabhängige JavaFX-Anwendung will, muss zu seiner Anwendung die JavaFX-Bibliotheken für alle Zielplattformen selbst mit dazu packen. Das Paket wird allerdings dadurch schnell mehrere hundert Megabyte groß, was keine optimale Lösung darstellt.

### Eine mögliche Lösung

Mit Java 9 kam mit `jlink` eine Neuerung dazu, die damals noch kaum Beachtung gefunden hat, aber nun in Verbindung mit JavaFX sehr interessant ist. `jlink` ermöglicht nämlich, eine Java-Anwendung mit genau den Teilen der Laufzeitumgebung zusammenzupacken, welche diese Anwendung benötigt. Dadurch lassen sich Anwendungen auch auf Systemen starten, auf denen keine Java-Laufzeitumgebung vorhanden ist, oder bei denen man nicht weiß, welche Java-Version installiert ist. So deployte Anwendungen startet zudem auch noch schneller, da nicht die gesamte Java-Laufzeitumgebung geladen werden muss, sondern nur die Teile, die tatsächlich benötigt werden. `jlink` kann jedoch nur dann richtig funktionieren, wenn die Anwendung vollständig das neue Modulsystem aus Version 9 einsetzt. Nur damit kann automatisch erkannt werden, welche Teile der Java-Laufzeitumgebung von der Anwendung benötigt werden.

### Das Modulsystem

Durch das Modulsystem soll vor allem möglich werden, die Sichtbarkeiten der eigenen Bibliotheken besser zu kontrollieren. Ein Beispiel dafür sind Packages mit Klassen, die nur für die eigene Verwendung bestimmt sind. In dem neuen Modulsystem muss explizit angegeben werden, welche Packages nach außen hin sichtbar sind und von anderen Anwendungen verwendet werden können. Weil klar definiert ist, welche Packages sich in welcher Bibliothek befinden, können Klassen zur Laufzeit schneller gefunden und geladen werden, wodurch sich die Geschwindigkeit der gesamten Anwendung erhöht. Um das Modulsystem im eigenen Projekt zu benutzen, muss im Root-Ordner des Quellcodes eine Datei mit dem Namen **module-info.java** erzeugt werden. In einem Maven-Projekt wäre das: **src/main/java**. Dieser kann für ein einfaches JavaFX-Projekt zum Beispiel wie in (Listing 1) aussehen.

#### (Listing 1)

```
1 module treemap {
2     requires javafx.controls;
3     exports treemap;
4 }
```

In Zeile 1 wird der Name dieses Moduls festgelegt. Zeile 2 besagt, dass dieses Modul ein Modul von JavaFX benötigt. In Zeile 3 wird

festgelegt, welche Packages dieses Modul zur Verfügung stellt. Gerade in JavaFX-Projekten ist es wichtig das Package mit der Hauptklasse zu exportieren, da das JavaFX-System sonst nicht auf diese zugreifen kann und es dadurch bereits beim Starten zu einer Fehlermeldung kommen würde.

Das Problem verschiedener Versionen der gleichen Bibliothek in einer Anwendung, wird durch das Modulsystem nicht gelöst, was ein Teil der Kritik am Modulsystem war. Mittlerweile ist es so, dass viele Bibliotheken, die man in der eigenen Anwendung verwendet, wiederum andere Bibliotheken benötigen. Es kann schnell passieren, dass über diese sogenannten transitiven Abhängigkeiten verschiedene Versionen der gleichen Bibliothek benötigt werden. Das Modulsystem wäre eine Möglichkeit gewesen dieses Problem zu lösen. Stattdessen gibt es einige Dinge bei der Verwendung zu beachten. Vieles, was vorher einfach so funktioniert hat, muss nun richtig konfiguriert werden.

Immerhin besitzt das Modulsystem einen Modus zur Abwärtskompatibilität. Somit können Bibliotheken verwendet werden, die das neue Modulsystem noch nicht nutzen. Auf diesen Kompatibilitätsmodus muss komplett verzichtet werden, wenn man JLink einsetzen möchte. Gerade in größeren Projekten kommt es schnell vor, dass viele verschiedene Bibliotheken zum Einsatz kommen. Das schließt eine Verwendung von JLink praktisch aus, weil nur wenige Libraries das neue Modulsystem anwenden. Um hier Abhilfe zu schaffen, wurde in Java 9 das Tool Jdeps integriert. Es analysiert die Abhängigkeiten des eigenen Programms und seiner Bibliotheken. So können die Informationen für das Modulsystem nachträglich generiert werden.

## Multi-Version-Bibliotheken

Mit Jdeps und JLink gibt es jedoch weitere Hürden, um eine Anwendung mit eigener Laufzeitumgebung ausliefern zu können. Mit Java 9 wurden sogenannte Multi-Version-Bibliotheken eingeführt. Es war schon immer so, dass sich der Umfang der Java-Laufzeitumgebung von Version zu Version leicht erweitert wurde. Multi-Version-Bibliotheken ermöglichen es, Teile des Programms auszutauschen, falls es auf einer anderen Umgebung ausgeführt wird. Das ist ein durchaus nützliches Feature, verhindert aber den Einsatz von Jdeps mit einer niedrigeren Version als Java 12. Denn Jdeps ist erst mit Java 12 in der Lage herkömmliche und Multi-Version-Bibliotheken miteinander zu vermischen. Wer also mit einer niedrigeren Java-Version als 12 Multi-Version-Bibliothek verwendet, kann Jdeps nicht einsetzen.

Eine Multi-Version-Bibliothek zeichnet sich dadurch aus, dass im **META-INF** Ordner ein Unterordner mit dem Namen **versions** liegt. In diesem Ordner kann dann für jede Version, für die Teile der Bibliothek ersetzt werden, jeweils ein Ordner gelegt werden dessen Name einfach nur aus der Versionsnummer besteht. Für Änderungen, die für Java-Version 10 und später gültig werden, würde der Ordner also **META-INF/versions/10** heißen. Dieser

Ordner fungiert dann als ein weiteres Wurzelverzeichnis der Bibliothek. Wird zum Beispiel die Klasse **treemap/TreeMap.class** ersetzt, hat die neue Datei den vollständigen Namen **META-INF/versions/10/treemap/TreeMap.class**. In diesem Beispiel sollte die eigene Bibliothek mindestens mit Java 10 ausgeführt werden.

## Installer für Java-Programme

Mit Java 8 war es möglich, eine JAR-Datei zu packen und diese dem Anwender zur Verfügung zu stellen. JAR-Dateien können bekanntlich per Doppelklick direkt ausgeführt werden. Die mit JLink erstellten Images sind dagegen komplexe Ordnerstrukturen, in denen der Benutzer erst die startbare Datei finden muss. Das ist zwar durch eine Batch-Datei oder etwas ähnliches lösbar, aber es ist schade, dass Java bis zu diesem Zeitpunkt keine benutzerfreundlichere Lösung dafür zur Verfügung gestellt hat. Erst mit Java 14 kam das Tool Jpackage zum JDK hinzu. Dieses Tool ist ein Ersatz für das zuvor gestrichene Tool JavaPackage, um eigenständig laufende Programme zu erzeugen. Das ist mit Java 14 endlich wieder möglich, auch wenn diese Pakete nun nicht mehr auf beliebigen Plattformen, sondern immer nur auf einer bestimmten Zielplattform laufen. Dafür kann Jpackage aber einen Installer für die jeweilige Plattform erstellen, den der Anwender anklickt, um die Anwendung wie gewohnt zu installieren und anschließend zu starten. Benutzer bekommen damit jetzt die für die jeweilige Plattform vertraute und damit bestmögliche Customer-Experience.

## Fazit:

Das eigene Projekt setzt am besten vollständig das Modulsystem ein. Sämtliche Bibliotheken müssen mit Hilfe von Jdeps analysiert werden, um herauszufinden welche Teile der Java-Laufzeitumgebung benötigt werden. Mit JLink kann ein Image geschaffen werden, welches die nötigen Teile der Java-Laufzeitumgebung enthält. Diese Anwendung kann somit auch auf einem System ausgeführt werden, welches keine oder die falsche Java-Laufzeitumgebung hat. Für eine Standard-Installationsprozedur wird das Image anschließend mit Jpackage verpackt. Zumindest für Desktop-Entwickler ist spätestens jetzt die Zeit gekommen den Schritt von Java 8 vorwärts zu gehen, denn mit Java 14 gibt es nun endlich eine Möglichkeit den Benutzern wieder die gleiche oder sogar eine bessere Experience zu bieten als früher.

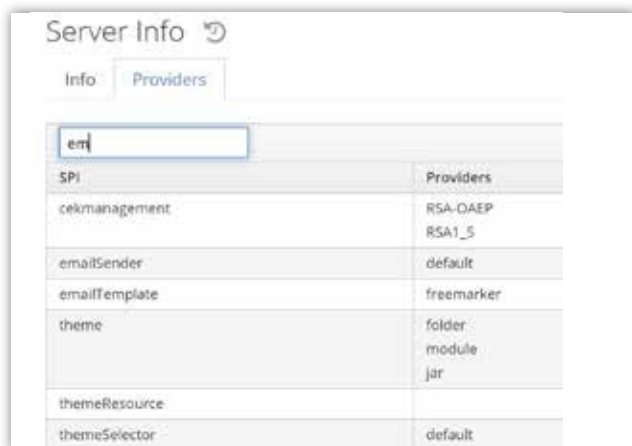
Der gesamte Prozess ist jedoch aufwändig. Denn gerade, wenn man ein größeres Projekt hat, wird man wahrscheinlich ein Build-System wie Maven verwenden, welches sich um die ganzen Bibliotheken kümmert. Dort werden Bibliotheken schnell in einer unübersichtlichen Anzahl vorhanden sein. Da Maven ohnehin sämtliche Informationen darüber hat welche Bibliotheken benötigt werden, wäre ein Maven-Plugin hilfreich, das den gesamten Prozess abbildet und ausführt. Der Autor hat dazu auf GitHub ein entsprechendes Open-Source-Projekt gestartet: <https://bit.ly/github-plugin>.

# Keycloak individuell erweitern

Die Identity- und Access-Management-Lösung Keycloak erfreut sich in den letzten Jahren weiter Verbreitung. Trotz vieler Funktionen sind in der Praxis häufig individuelle Anpassungen erforderlich. Wie kann man solche Erweiterung vornehmen - von der Bereitstellung eines eigenen Moduls, über die Registrierung eines Providers bis hin zu Aspekten wie Logging und Konfiguration?

**K**eycloak ist eine Open-Source Identity- und Accessmanagement-Lösung. Für individuelle Erweiterungen stehen in Keycloak eine Reihe von Erweiterungspunkten, sogenannte Service-Provider-Interfaces (SPI) zur Verfügung, um vorhandene Funktionalität auszutauschen oder an die eigenen Bedürfnisse anzupassen.

## Service-Provider-Interfaces in Keycloak



Übersicht der Keycloak-Provider auf der Server-Info-Page. (Abb. 1)

Keycloak bietet eine ganze Reihe an SPIs, die als Erweiterungs- bzw. Anpassungspunkte dienen können. Da die Dokumentation längst nicht alle vorhandenen SPIs auflistet, empfiehlt sich für eine Gesamtübersicht ein Blick auf Keycloaks Server-Info-Page. Hierzu reicht es aus, eine Keycloak-Instanz mittels Docker zu starten (Listing 1).

### (Listing 1)

```
docker run -d --name keycloak -p 8080:8080 -e KEYCLOAK_USER=test
-e KEYCLOAK_PASSWORD=test jboss/keycloak:9.0.2
```

Anschließend steht die Server-Info-Page mit einer Liste aller SPIs und verfügbarer Implementierung in der Administrationskonsole zur Verfügung. Der Login erfolgt mit Benutzername **test** und Passwort **test**.

Im Rahmen dieses Artikels soll die SPI **emailSender** als Beispiel dienen. Wie die Server-Info-Page zeigt, gibt es dazu in Keycloak genau eine Implementierung bzw. einen Provider namens **default**. Die Implementierung dient zum Versand von E-Mails und nutzt die SMTP-Einstellungen, die in der Konsole unter **Realm Settings** im Untermenü **Email** konfiguriert werden können. Als exemplarische Anforderung für eine Erweiterung soll es möglich sein, die Betreffzeilen aller E-Mails mit einem konfigurierbaren Präfix zu versehen. Beispielsweise soll ein Präfix **KEYCLOAK:** konfiguriert werden können, so dass eine E-Mail mit dem eigentlichen Betreff **Ihr Passwort wurde geändert** nun den Betreff **KEYCLOAK: Ihr Passwort wurde geändert** trägt.

## Ein Grundgerüst für die Implementierung

Die Implementierung einer SPI wird in der Regel als JAR-Datei ausgeliefert und später deployt. Zur Generierung eines JARS kann ein einfaches Maven-Projekt<sup>1</sup> genutzt werden. Folgende Maven-Dependencies sind für eine Implementierung der SPI notwendig und dem Projekt hinzuzufügen (Listing 2).

### Autor:



Sven-Torben Janus arbeitet als Principal Architect bei der Conciso GmbH wo er als Partner den Bereich Softwarearchitektur mitverantwortet. Er befürwortet einen agilen und praktikablen Entwurf von Softwarearchitekturen und ist Gründer des DDD Meetups Rhein/Main.

Blog-URL: <https://conciso.de/blog/author/janus/>

Twitter: <https://twitter.com/sventorben>

Github: <http://github.com/sventorben>

E-Mail: [sven-torben.janus@conciso.de](mailto:sven-torben.janus@conciso.de)

(Listing 2)

```
<project>
  <!-- ... -->
  <dependencies>
    <dependency>
      <groupId>org.keycloak</groupId>
      <artifactId>keycloak-core</artifactId>
      <version>${version.keycloak}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.keycloak</groupId>
      <artifactId>keycloak-server-spi</artifactId>
      <version>${version.keycloak}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.keycloak</groupId>
      <artifactId>keycloak-server-spi-private</artifactId>
      <version>${version.keycloak}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <!-- ... -->
</project>
```

Die SPI **emailSender** ist durch das Interface **EmailSenderProvider** definiert. Die eigene Implementierung erfolgt einfach über die Implementierung dieses Interfaces wie in (Listing 3) dargestellt.

(Listing 3)

```
public class EmailPrefixSenderProvider implements EmailSenderProvider {

    EmailPrefixSenderProvider() {
    }

    public void send(Map<String, String> config, UserModel user,
String subject, String textBody, String htmlBody) throws EmailException {
        // ... hier folgt die Implementierung
    }

    public void close() {
    }

}
```

Zusätzlich benötigt diese Implementierung eine zugehörige Factory im Sinne des Factory-Patterns. Die Implementierung erfolgt über das entsprechende Interface **EmailSenderProviderFactory** (Listing 4).

(Listing 4)

```
public class EmailPrefixSenderProviderFactory implements
EmailSenderProviderFactory {

    public EmailSenderProvider create(KeycloakSession session) {
        return new EmailPrefixSenderProvider();
    }

}
```

```
public void init(Config.Scope config) {
}

public void postInit(KeycloakSessionFactory factory) {
}

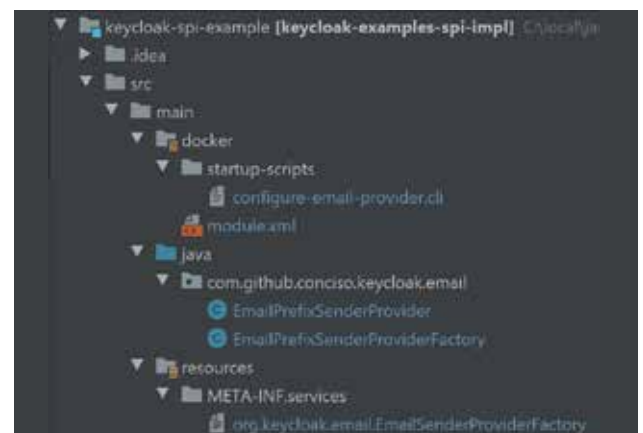
public void close() {
}

public String getId() {
    return "email-prefix";
}

}
```

Dabei wird in der Methode **create** eine Instanz des Providers erzeugt und in der Methode **getId** ein Name für die SPI-Implementierung definiert. Dieser Name ist äquivalent zum Namen **default**, der für die Standardimplementierung auf der Server-Info-Page zu sehen war.

Damit ist das Grundgerüst für eine Implementierung prinzipiell bereits erstellt. Es fehlt lediglich eine Datei, um den Provider zu registrieren, wie sie für Java-Service-Provider-Interfaces<sup>2</sup> üblich ist. Die Datei muss den Namen **org.keycloak.email.EmailSenderProviderFactory** haben und im Verzeichnis **src/main/resources/META-INF/services** liegen. Als Inhalt hat sie nur eine Zeile mit dem vollständigen Namen der Factory-Klasse: **io.github.conciso.keycloak.email.EmailPrefixSenderProviderFactory** (Abb. 2).



Dateistruktur des Maven-Projekts. (Abb. 2)

Durch den Maven-Build wird über den Befehl **mvn clean install** aus diesen Code-Artefakten eine JAR-Datei erzeugt.

## Deployment in einen Docker-Container

Die JAR-Datei muss in ein Wildfly-Modul<sup>3</sup> eingepackt werden, um sie nach Keycloak deployen zu können. Das Wildfly-Modul besteht im Prinzip aus einer Descriptor-Datei im XML-Format und der JAR-Datei selbst. Die Descriptor-Datei sieht dabei wie in (Listing 5) gezeigt aus und trägt den Namen **module.xml**. Sie liegt im Verzeichnis **src/main/docker**.

(Listing 5)

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" name="keycloak-spi-example.provider">
  <resources>
    <resource-root path="${project.build.finalName}.jar"/>
  </resources>
  <dependencies>
    <module name="org.keycloak.keycloak-core"/>
    <module name="org.keycloak.keycloak-server-spi"/>
    <module name="org.keycloak.keycloak-server-spi-private"/>
  </dependencies>
</module>
```

Sie definiert die Abhängigkeiten auf andere Keycloak-spezifische Wildfly-Module. Diese Wildfly-Module entsprechen den zuvor genannten Maven-Dependencies. Darüber hinaus definiert das Modul aus welchen JAR-Dateien es besteht. In diesem Fall wird die Maven-Property **project.build.finalName** genutzt, um den Namen der JAR-Datei anzugeben, die durch den Maven-Build erzeugt wird. Dazu ist es notwendig in Maven das Resource-Filtering zu aktivieren (Listing 6).

(Listing 6)

```
<project>
  <!-- ... -->
  <build>
    <resources>
      <resource>
        <directory>src/main/docker</directory>
        <filtering>true</filtering>
      </resource>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
    </resources>
  <!-- ... -->
</build>
<!-- ... -->
</project>
```

Das Deployment erfolgt durch Kopieren der beiden Dateien in das **modules** Verzeichnis des Keycloak- bzw. Wildfly-Servers. Nutzt man Keycloak auf Basis der offiziellen Docker-Images<sup>4</sup>, kann das im Rahmen der Erstellung eines Dockerfile geschehen, wie in (Listing 7) beschrieben.

(Listing 7)

```
FROM jboss/keycloak:9.0.2

COPY target/*.jar /opt/jboss/keycloak/modules/keycloak-spi-example/provider/main/
COPY target/classes/module.xml /opt/jboss/keycloak/modules/keycloak-spi-example/provider/main/
```

Das Image kann nun mittels Docker gebaut werden. Es geht aber auch direkt innerhalb des Maven-Builds mittels Fabric8s-Docker-Maven-Plugin (Listing 8).

(Listing 8)

```
<project>
  <!-- ... -->
  <build>
    <!-- ... -->
    <plugins>
      <!-- ... -->
      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>docker-maven-plugin</artifactId>
        <version>0.33.0</version>
        <configuration>
          <images>
            <image>

<name>${project.artifactId}:${project.version}</name>
            <build>

<dockerFileDir>${project.basedir}</dockerFileDir>
            <noCache>true</noCache>
            <tags>
              <tag>latest</tag>
            </tags>
          </build>
        </image>
      </images>
    </configuration>
    <executions>
      <execution>
        <phase>package</phase>
        <goals>
          <goal>build</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
  <!-- ... -->
</build>
<!-- ... -->
</project>
```

Der Provider ist damit in Keycloak deployt. Damit er jedoch genutzt wird, muss er noch registriert bzw. geladen werden. Dazu wird das erstellte Wildfly-Modul im Sub-Modul **keycloak-server** als Provider ergänzt. Dies erfolgt über ein Command-Line-Interface-Skript (CLI) beim Starten des Containers (Listing 9).

(Listing 9)

```
/subsystem=keycloak-server/:list-add(name=providers,value="module:keycloak-spi-example.provider")
```

Das Skript registriert das Wildfly-Modul als Provider. Keycloak wird daraufhin beim Starten das Modul nach möglichen SPI-Implementierungen durchsuchen und diese verfügbar machen. Für die automatisierte Ausführung des CLI-Skripts, stellt das offizielle Docker-Image einen Mechanismus bereit. Das Skript muss lediglich im Verzeichnis **/opt/jboss/startup-skript/** hinterlegt werden. Das kann durch Mounten eines Volumes oder durch Kopieren des Skriptes in das Image erfolgen. Im

Rahmen des Maven-Builds kann das Skript im Verzeichnis **src/main/docker/startup-scripts** liegen und mittels Dockerfiles in das Image kopiert werden (Listing 10).

(Listing 10)

```
COPY target/classes/startup-scripts/* /opt/jboss/startup-scripts/
```

Nach einem erneuten Build ist das Docker-Image einsatzbereit. Danach muss man noch den laufenden Docker-Container stoppen, mit Maven erneut den Build starten und anschließend den Container mit dem neuen Image wieder starten (Listing 11). Auf der Server-Info-Page sollte nun neben dem **default** Provider zusätzlich der neue Provider mit Namen **example-email** erscheinen.

(Listing 11)

```
docker stop keycloak && docker rm keycloak
mvn clean install
docker run --name keycloak -d -p 8080:8080 -e KEYCLOAK_USER=test
-e KEYCLOAK_PASSWORD=test keycloak-spi-example
```

## Aktivierung des Providers

Der Keycloak-Server verfügt nun über zwei Implementierungen des gleichen SPI. Es stellt sich daher die Frage, welche Implementierung herangezogen wird, wenn die SPI genutzt werden soll. Hierzu stellt Keycloak die Möglichkeit zur Aktivierung einzelner Provider im Sub-Modul **keycloak-server** bereit. Dazu wird das CLI-Skript einfach erweitert (Listing 12).

(Listing 12)

```
/subsystem=keycloak-server/spi=emailSender:add
/subsystem=keycloak-server/spi=emailSender/provider=email-prefix:add(enabled=true)
/subsystem=keycloak-server/spi=emailSender/write-attribute(name=default-provider, value=email-prefix)
```

Im ersten Schritt wird die SPI **emailSender** hinzugefügt. Anschließend wird der neue Provider für die SPI ergänzt und aktiviert (**enabled=true**). Zu guter Letzt setzt das Skript den Standard-Provider auf den neuen Provider. Das sorgt dafür, dass der Provider immer herangezogen wird, wenn aus Keycloak heraus eine E-Mail versandt werden soll.

## Logging

Beim Starten des Containers ist der Provider nun aktiviert und wird standardmäßig verwendet. Zur Prüfung wird zunächst einmal ein Logging in den Provider eingebaut. Keycloak nutzt dazu JBoss-Logging. Dazu muss die entsprechende Dependency in der **pom.xml** für den Maven-Build ergänzt werden (Listing 13).

(Listing 13)

```
<project>
  <!-- ... -->
  <dependencies>
    <!-- ... -->
    <dependency>
      <groupId>org.jboss.logging</groupId>
      <artifactId>jboss-logging</artifactId>
      <version>3.4.1.Final</version>
      <scope>provided</scope>
    </dependency>
    <!-- ... -->
  </dependencies>
  <!-- ... -->
</project>
```

Zusätzlich ist es notwendig, das entsprechende Wildfly-Modul in der **module.xml** zu referenzieren (Listing 14). Anschließend kann in der Implementierung des Providers das Logging erfolgen (Listing 15).

(Listing 14)

```
<module>
  <!-- ... -->
  <dependencies>
    <!-- ... -->
    <module name="org.jboss.logging"/>
  </dependencies>
</module>
```

(Listing 15)

```
public class EmailPrefixSenderProvider implements
EmailSenderProvider {

    private static final Logger logger =
Logger.getLogger(EmailPrefixSenderProvider.class);

    public void send(Map<String, String> config, UserModel user,
String subject, String textBody, String htmlBody) throws EmailException {
        logger.info("Sending email");
    }
    // ...
}
```

Nach erneutem Build und einem Neustart des Containers ist es möglich die Nutzung des Providers im Logfile zu beobachten. Mittels **docker logs -f keycloak** kann das Logfile verfolgt werden. Zum Testen wird in der Administrationskonsole dem angemeldeten Testbenutzer über den Menüpunkt **Users** eine E-Mail-Adresse zugewiesen. Dazu noch den Benutzer suchen, editieren, die E-Mail-Adresse im Feld **Email** eintragen und die Option **Email Verified** aktivieren und anschließend speichern (Abb. 3). Im Anschluss muss man noch im Menü **Realm Settings** unter **Email** auf Test connection klicken. Im Log sollte dann der entsprechende Eintrag **Sending email** erscheinen.



Aktivierung der E-Mail für einen Benutzer.  
(Abb. 3)

## Nutzung vorhandener Provider

Nun fehlt noch die Implementierung der eigentlichen Funktion zum E-Mail-Versand. Hierzu kann auf die Keycloak Default-Implementierung zurückgegriffen werden. Das kann auf eine von zwei Arten erfolgen: Durch direkte Instanziierung oder Provider-Lookup.

### Direkte Instanziierung

In der ersten Variante wird der Default-Provider einfach instanziiert. Die dazu notwendige Java-Klasse **org.keycloak.email.DefaultEmailSenderProvider** befindet sich im Keycloak-Modul **keycloak-services**. Dieses wird daher zunächst als Maven-Dependency in der **pom.xml** ergänzt. Darüber hinaus muss auch diesmal in der **module.xml** das entsprechende Modul referenziert werden (Listing 16). Anschließend kann der Provider instanziiert und aufgerufen werden (Listing 17).

(Listing 16)

```
<!-- pom.xml -->
<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-services</artifactId>
  <version>${version.keycloak}</version>
  <scope>provided</scope>
</dependency>
```

```
<!-- module.xml -->
<module name="org.keycloak.keycloak-services"/>
```

(Listing 17)

```
public class EmailPrefixSenderProvider implements
EmailSenderProvider {
```

```
private final KeycloakSession session;

EmailPrefixSenderProvider(KeycloakSession session) {
  this.session = session;
}

public void send(Map<String, String> config, UserModel user,
String subject, String textBody, String htmlBody) throws EmailException {
  String prefixedSubject = "KEYCLOAK: " + subject;
  EmailSenderProvider defaultProvider = new
DefaultEmailSenderProvider(session);
  defaultProvider.send(config, user, prefixedSubject, textBody,
htmlBody);
}
}
```

Dazu benötigt der **DefaultEmailSenderProvider** eine **KeycloakSession**, über die er die notwendigen Einstellungen für den SMTP-Server sowie Absenderadresse etc. ermitteln kann. Diese Informationen entsprechen den Einstellungen in der Administrationskonsole unter **Realm Settings** im Unterpunkt **Email**. Die **KeycloakSession** erhält der **EmailPrefixSenderProvider** ebenfalls über seinen Konstruktor. Entsprechend muss die **EmailPrefixSenderProviderFactory** angepasst werden (Listing 17). Alternativ könnte eine Instanz des **DefaultEmailSenderProvider** auch über die zugehörige Factory mittels **new DefaultEmailSenderProviderFactory().create(session)** erzeugt werden.

(Listing 17)

```
public EmailSenderProvider create(KeycloakSession session) {
  return new EmailPrefixSenderProvider(session);
}
```

## Provider-Lookup

Keycloak nutzt diese Provider-Factories selbst auch, um entsprechende Instanzen zu erzeugen. Dabei stehen über das Interface an der **org.keycloak.provider.ProviderFactory** weitere Methoden zur Verfügung, wie zum Beispiel **init** oder **postInit**, die hauptsächlich der weiteren Konfiguration der Provider dienen. Es empfiehlt sich daher grundsätzlich eher einen Provider-Lookup in Keycloak zu machen, anstatt eine direkte Instanziierung vorzunehmen. Der Provider-Lookup erfolgt über die **KeycloakSession** wie mittels **session.getProvider(EmailSenderProvider.class, "default")**.

Dabei gibt der erste Parameter **EmailSenderProvider.class** die Art des Providers an, der zweite Parameter **"default"** referenziert den Namen der Provider-Implementierung die genutzt werden soll. Die **KeycloakSession** stellt darüber hinaus viele weitere Methoden für den Zugriff auf unterschiedlichste Provider bereit. Hier empfiehlt sich grundsätzlich ein Blick in die Java-Dokumentation<sup>5</sup>.

## Konfiguration des Providers

Der Mechanismus, der über die **ProviderFactory** bereitgestellt wird, kann auch für die Konfiguration eigener Provider genutzt werden. Als Beispiel dient das Präfix für die Betreffzeile der E-Mail. Diese ist bislang hart als **"KEYCLOAK: "** kodiert. Um sie zu konfigurieren, wird der Provider erweitert, so dass das Präfix über den Konstruktor an den Provider übergeben werden kann (**Listing 18**).

(Listing 18)

```
public class EmailPrefixSenderProvider implements
EmailSenderProvider {

    private final KeycloakSession session;
    private final String subjectPrefix;

    EmailPrefixSenderProvider(KeycloakSession session, String
subjectPrefix) {
        this.session = session;
        this.subjectPrefix = subjectPrefix;
    }

    public void send(Map<String, String> config, UserModel user,
String subject, String textBody, String htmlBody) throws Email-
Exception {
        String prefixedSubject = subjectPrefix + subject;
        // ...
    }
}
```

Der Wert wird analog zur **KeycloakSession** über die Factory ermittelt. Hierzu kann mittels der **init** Methode in der Factory auf die Konfiguration der Provider-Implementierungen zugegriffen werden (**Listing 19**).

(Listing 19)

```
public class EmailPrefixSenderProviderFactory implements
EmailSenderProviderFactory {

    private static final String CONFIG_SUBJECT_PREFIX =
"subjectPrefix";

    private Config.Scope config;

    public EmailSenderProvider create(KeycloakSession session) {
        String subjectPrefix = config.get(CONFIG_SUBJECT_PREFIX);
        return new EmailPrefixSenderProvider(session,
subjectPrefix);
    }

    public void init(Config.Scope config) {
        this.config = config;
    }

    // ...
}
```

Die Provider-Factory definiert nun, dass unter dem Schlüssel **subjectPrefix** das entsprechende Präfix in der Konfiguration zu finden ist. Letztlich wird der Konfigurationseintrag über CLI-Skripte

im Keycloak-Sub-System für die **EmailSenderProvider** SPI vorgenommen. Dazu ist lediglich eine Anpassung im bereits vorgestellten CLI-Skript notwendig. Die Zeile, die den Email-Pre-fix-Provider hinzufügt, wird um das entsprechende Property in der Konfiguration erweitert (**Listing 20**).

(Listing 20)

```
/subsystem=keycloak-server/spi=emailSender/provider=email-pre-
fix:add(properties={subjectPrefix => "SSO: "},enabled=true)
```

Nach dieser Konfigurationsänderung werden nun alle E-Mails mit einem Präfix **"SSO: "** im Betreff verschickt. Auf diese Art und Weise können SPIs konfiguriert werden. Neben einfach Strings, werden auch Integers, Listen und boolesche Werte unterstützt. Darüber hinaus ist es möglich, Ausdrücke zu nutzen, um System-Properties wie **\${subjectPrefix}** oder Umgebungsvariablen wie **\${env.SUBJECT\_PREFIX}** aufzulösen und Standardwerte zu setzen. Die zuvor genannte Konfiguration liest so beispielsweise die Umgebungsvariable **SUBJECT\_PREFIX**. Ist diese nicht gesetzt, wird das Präfix auf **KEYCLOAK** gesetzt. Wie in vielen Container-Umgebungen heute üblich, ist damit prinzipiell auch die Konfiguration über Umgebungsvariablen möglich (**Listing 21**).

(Listing 21)

```
/subsystem=keycloak-server/spi=emailSender/provider=email-pre-
fix:add(properties={subjectPrefix => "${env.SUBJECT_PREFIX:KEY-
CLOAK}: "},enabled=true)
```

## Fazit:

Das vorgestellte Beispiel zeigt, wie einfach Erweiterungen mittels Service-Provider-Interfaces in Keycloak möglich sind. Das Beispiel ist recht trivial gewählt. Die Vielfalt an vorhandenen SPIs, wie in der Server-Info-Page ersichtlich, lässt weitaus umfangreichere Anpassungsmöglichkeiten erahnen. Dabei folgen solche Anpassungen in der Regel den vorgestellten Mustern. Dazu zählt im Kern die Implementierung der SPI durch einen konkreten Provider mit zugehöriger Factory. Ebenso ist die Registrierung der SPI und deren Konfiguration vor allem in komplexeren Szenarien zumeist notwendig. Gleiches gilt für Aspekte wie Logging oder die Nutzung vorhandener Provider über entsprechende Lookups. Sie bilden daher die Grundlagen für eigene Erweiterungen eines Keycloak-Systems.

## Quellen:

- 1 Der Code ist auf Github unter <https://bit.ly/github-g1> zum Nachlesen veröffentlicht.
- 2 Einführung in Java SPIs: <https://bit.ly/javaSPI-2>
- 3 Classloading in Wildfly: <https://bit.ly/wildfly-w3>
- 4 Keycloak auf Basis der offiziellen Docker-Images: <https://bit.ly/docker-img4>
- 5 Java-Dokumentation: <https://bit.ly/keycloak-k5>

#JAVAPRO #Architecture #Projektmanagement

# Self-Contained-Systems

Self-Contained-Systems ist sowohl ein Architektur-, als auch Organisationsansatz, der im Kern auf dem Domain-Driven-Design-Ansatz aufsetzt und erweitert. SCS soll nicht nur die Trennung und Betrieb von Systemen, sondern auch die organisatorische Strukturierung und Trennung größerer Entwicklungsteams erleichtern. Der Fokus von SCS liegt klar auf Anforderungen und nicht auf technische Implementierungsdetails.

## Autor:



Mirko ist Software Craftsman im Web/eCommerce-Umfeld. In Funktionen als Software-Engineer, Architekt und Consultant in Projekten in Deutschland und der Schweiz sammelte er Erfahrungen mit einer Vielzahl von Frameworks, Technologien und Methoden. Heute arbeitet er als IT-Analyst bei der Thalia Bücher GmbH in Münster mit Schwerpunkt auf Java, eCommerce sowie Such- und Empfehlungs-Technologien. Seine Freizeit verbringt er mit seiner Familie und hin- und wieder mit Open-Source Projekten.

Webseite: <https://www.mirkosertic.de>

Twitter @mirkosertic

GitHub <https://github.com/mirkosertic>

LinkedIn <https://www.linkedin.com/in/mirko-sertic-98882397/>

XING [https://www.xing.com/profile/Mirko\\_Sertic](https://www.xing.com/profile/Mirko_Sertic)

E-Mail: [mirko@mirkosertic.de](mailto:mirko@mirkosertic.de)

**S**elf-Contained-Systems (SCS) ist zunächst einmal ein Architekturansatz. Die Idee ist, Anforderungen durch einzelne, fachlich unabhängige Systeme abzubilden. Diese Systeme beinhalten alles, was für die Abbildung der Anforderungen notwendig ist - die Datenhaltung, die eigentliche Geschäftslogik und die komplette Benutzerschnittstelle. Diese Systeme werden dadurch self-contained.

Jedes SCS soll von einem eigenen Entwicklungsteam betreut werden. Dieser Architekturansatz soll neben der technischen Trennung der Systeme auch die organisatorische Trennung erleichtern. Durch die Vermeidung von Abhängigkeiten werden auch die Teams unabhängiger und damit flexibler. Soweit die Theorie. Nachfolgend werden Vorteile und auch mögliche Probleme bei der Implementierung des SCS-Ansatzes beleuchtet.

## Grundlagen von Self-Contained-Systems

Die Idee, fachlich unabhängige Teilsysteme zu definieren, ist nicht neu. Eric Evans hat den Begriff Domain-Driven-Design, kurz

DDD, geprägt. DDD stellt die fachlichen Anforderungen und die daraus resultierende fachliche Struktur ganz klar in den Vordergrund. DDD bietet ebenfalls Entwurfsmuster an, um die fachlichen Anforderungen zu dokumentieren und auch zu implementieren. In der DDD-Welt ist ein fachlich abgeschlossener Bereich ein Bounded-Context.

Der SCS-Ansatz baut genau auf diesen Bounded-Contexten auf und erweitert deren Definition. Neben der fachlichen Logik wird in diesem Kontext auch die Benutzerschnittstelle implementiert. Ein wichtiger Punkt ist hier, dass beim Aufruf einer Funktionalität keine synchrone Kommunikation mit Komponenten außerhalb des Self-Contained-Systems erfolgen soll. Wenn Kommunikation stattfindet, so soll sie ausschließlich asynchron erfolgen. Daraus ergeben sich mehrere Konsequenzen. Das System wird dadurch widerstandsfähiger gegenüber möglichen Fehlern, wie zum Beispiel Timeouts. Auch wird die Implementierung der Logik einfacher, da keine Patterns, wie zum Beispiel ein Bulkhead oder ein Circuit-Breaker vor die Kommunikation geschaltet werden und eventuelle Fallback-Szenarien implementiert werden müssen. Die andere Konsequenz ist die Verhinderung der weiteren Zerlegung des Bounded-Context in weitere Artefakte, ein Ansatz der auch gerne Microservices genannt wird. Weshalb? In der DDD-Welt ist der Bounded-Context eine fachliche Einheit, welche alle notwendige Logik kapselt. Wenn ein Change an dieser Einheit vorgenommen werden muss, wie beispielsweise eine Datenbankerweiterung, so betrifft dieser Change vermutlich mehr als nur die Datenbank und erstreckt sich über alle Schichten des Bounded-Context. Wenn nun diese Schichten nicht über mehrere Microservices verteilt sind, ist die Tragweite von solchen Changes einfacher zu überblicken. Es ergibt sich gar nicht mehr die Notwendigkeit auf Breaking-Changes in einer API zu achten. Durch die explizite Vermeidung von synchroner Kommunikation soll zudem das Risiko reduziert werden, einen verteilten Monolithen mit viel zu vielen und langsamen Schnittstellen zu bauen. Die Devise lautet: Weniger ist mehr. Oder pragmatischer ausgedrückt: Was nicht da ist, macht auch keine Fehler.

## Auch die Organisation verändert sich

Der SCS-Architekturansatz hat Konsequenzen auf die Organisation und die Teamstruktur. Wie schon einleitend erwähnt, soll jedes Self-Contained-System von einem Team verantwortlich betreut werden. Das bedeutet, dass sich auch die dafür notwendigen Kompetenzen in diesem Team bündeln müssen. Diese Kompetenzen wären unter anderem das Product-Ownership, Entwicklungs- sowie QA-Kompetenzen und auch Ops oder User-Experience (UX). Durch die Bündelung dieser Kompetenzen in einem Team wird das Team handlungsfähig und kann Geschwindigkeit bei der Umsetzung von neuen Features aufnehmen. Idealerweise ist also jedes Teammitglied im Full-Stack unterwegs. Architekturentscheidungen, welche nur ein SCS betreffen, werden auch Mikro-Architektur-Entscheidungen genannt. Diese Entscheidungen können schnell getroffen und

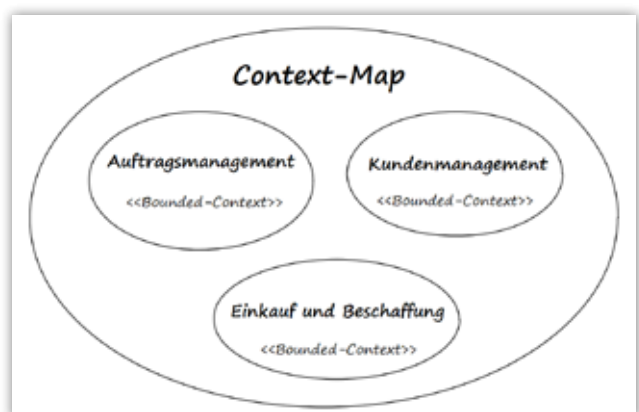
revidiert werden, da die Tragweite auf ein SCS beschränkt ist. Es sind lokale Entscheidungen, wie zum Beispiel:

- welche Java-Version,
- welches Framework,
- welche Persistenz-API oder
- ob eine hexagonale Struktur verwendet wird.

Dem gegenüber stehen Makro-Architektur-Entscheidungen. Diese Entscheidungen betreffen die komplette Systemlandschaft, haben in der Regel eine größere Tragweite, erfordern mehr Abstimmungsaufwand und sind meistens schwerer zu revidieren. Solche Entscheidungen sind zum Beispiel:

- die Anzahl der SCS,
- welche systemübergreifenden Protokolle verwendet werden sollen oder
- welche Standards bei der Präsentationstechnologie einzuhalten sind.

Die Makroarchitektur ist also auch maßgeblich Organisationsarchitektur. Wie groß kann nun ein SCS und das betreuende Team werden? Das ist eine sehr interessante Frage auf die es jedoch keine absolute Antwort gibt. Ein Team sollte ein SCS betreuen, eventuell auch mehrere, abhängig von der fachlichen Struktur und die daraus resultierende Komplexität. Der treibende Faktor hinter der Teamgröße ist also ausschließlich der fachliche Umfang. Bei SCS handelt es sich also nicht nur um eine Sammlung technischer Grundsätze, sondern vielmehr um eine Richtlinie zur Organisation von Entwicklungsteams. Ohne Anpassungen innerhalb der Organisation kann der Ansatz nicht seine Vorteile ausspielen. Im Idealfall entspricht das Organigramm der **Context-Map** aus der DDD-Welt (Abb. 1).



Beispiel Context-Map. (Abb. 1)

## Datenhaltung und Co.

Ein Self-Contained-System soll alle Funktionalitäten beinhalten, die für die Ausführung der darin enthaltenen Anwendungsfälle notwendig sind. Dies beinhaltet auch die dafür notwendigen Daten. Auf Ebene Datenhaltung ergeben sich dadurch einige

Herausforderungen. Ein Auftragsmanagement SCS benötigt wahrscheinlich die Daten für die Aufträge, und hat über diese Daten auch die Datenhoheit. Es benötigt vermutlich auch die Kundendaten, hat aber eventuell über diese Daten nicht die Datenhoheit. Es benutzt diese Daten, ist abhängig davon, verwaltet sie aber nicht. Wie kann diese Problematik nun umgesetzt werden? Die Antwort lautet: mit Datenreplikation.

Mit Datenreplikation ist hier die fachliche Replikation der Daten gemeint, und nicht die technische Replikation, wie sie zum Beispiel mit einem Master-Slave-Datenbank-Setup möglich wäre. Was ist hier der Unterschied? Eine Replikation auf technischer Ebene, sprich via Datenbankreplikation, würde zu viele Implementierungsdetails offenlegen und die Systeme zu stark aneinanderbinden. Änderungen am Datenbankschema betreffen nicht mehr nur ein SCS. Es würden technische und organisatorische Abhängigkeiten entstehen, doch genau dies soll der SCS-Ansatz ja verhindern. Ein anderes Argument ist, dass nicht alle Systeme eine vollständige Kopie der Daten benötigen und vermutlich auch anders auf die Daten zugreifen müssen. Eventuell dürfen bestimmte Daten auch nicht das Ursprungssystem verlassen, da sie besonders schutzbedürftig sind. Auch kann beispielsweise System A für den schreibenden Zugriff auf Kundendaten optimiert sein und System B ausschließlich für den lesenden Zugriff optimiert werden. Daraus ergeben sich dann andere Anforderungen an die zu verwendende Datenbank, das Datenbankschema und Normalisierungsformen etc.

Welche Möglichkeiten gibt es nun? Als gut funktionierend hat sich Event-Sourcing in Verbindung mit einem Transaktionslog erwiesen. Eine Änderung an einem Kundendatensatz im Kundenmanagement-SCS würde das Event **KundeWurdeGeändert** mit den neuen Kundendaten generieren und in ein Transaktionslog schreiben. Das Auftragsmanagement-SCS würde dieses Event aus dem Transaktionslog konsumieren, die für die Auftragsverwaltung notwendigen Daten extrahieren und in seine eigene Datenbank schreiben.

Als Transaktionslog sind Lösungen wie Apache-Kafka sehr interessant. Neben der Tatsache, dass es sich hier um eine Pull-Lösung handelt und die Komplexität auf Ebene Kafka-Server dadurch sehr klein gehalten werden kann, ist ein anderer Aspekt viel wichtiger: Kafka ermöglicht eine Zeitreise! Der Konsument von Events kann seinen Offset innerhalb des Transaktionslog zurücksetzen, um Events aus der Vergangenheit nochmals zu verarbeiten. Diese Funktionalität ist wichtig für ein Disaster-Recovery. Fehler können und werden passieren, sei es technische Probleme oder einfach nur Fehler in der Software. Durch die Zeitreise können wir eine neue Version der Software installieren, Events aus der Vergangenheit konsumieren und eventuell beschädigte Datenbestände nachträglich reparieren. Eine Funktionalität die sich als sehr nützlich herausgestellt hat.

Ein Nachteil dieser Datenreplikation ist die dadurch entstehenden Redundanz und der damit verbundene Platzverbrauch.

Wenn zum Beispiel mehrere Millionen Artikeldatensätze über mehrere Systeme repliziert werden, ist die Größenordnung nicht zu unterschätzen und kann auch ein Kostentreiber werden. Aus Kostensicht zu beachten ist ebenfalls das Lizenzmodell der Datenbank. Jedes SCS sollte seinen eigenen Datenbankserver bekommen, um beispielsweise bei Wartungsarbeiten an dem Datenbankserver nicht gleich mehrerer Systeme beeinträchtigen zu müssen. Die Lizenzkosten können explodieren. Hier gilt es, die Risiken, Vor- und Nachteile abzuwägen und eine pragmatische Lösung zu finden und diese auch so dokumentieren und zu kommunizieren.

Bei Datenreplikation darf das Thema Datenschutz nicht vernachlässigt werden. Wo dürfen Kundendaten gespeichert werden? Was ist mit Kreditkartendaten? Welche Möglichkeiten zur Auditing müssen vorgesehen werden? Welche Daten müssen nachträglich löscherbar sein? Welche Daten müssen anonymisiert werden, eventuell auch nachträglich? Betrifft das nur das Zielsystem oder auch das Transaktionslog dazwischen? Wie sieht es mit der DSGVO-Konformität aus? Gibt es noch zusätzliche Auflagen? Der zuständige Datenschutzbeauftragte ist auf jeden Fall zu konsultieren, denn aus der technischen Datenreplikation kann sehr schnell ein fachliches und rechtliches Problem entstehen.

## Das Wichtigste ist die Geschäftslogik

Bei der Implementierung der Geschäftslogik sind einige Dinge aus konzeptioneller Sicht zu beachten. Self-Contained-Systeme bauen im Kern auf Ansätzen aus dem Domain-Driven-Design auf. Domain-Driven-Design hilft auch bei der Implementierung der Geschäftslogik. Patterns wie Aggregate, Entity, Value-Object und Repository helfen, die Geschäftslogik gut zu strukturieren und zu implementieren. Eine hexagonale Architektur unterstützt diese Patterns weiter und hilft dabei, ein modulare und gut testbare Systemstruktur zu erhalten.

Probleme ergeben sich bei der Implementierung von Anwendungsfällen, welche nicht komplett innerhalb eines SCS umgesetzt werden können, weil unter anderem Zulieferer oder Geschäftspartner involviert sind. Dies verdeutlicht ein Blick auf den Anwendungsfall **Kaufen und Lagerbestand reservieren**: Wenn ein Kunde etwas bestellt, soll der Auftrag im System verbucht werden. Abhängig davon ob der Artikel noch auf Lager ist oder nicht, soll der Lagerbestand für den Kunden reserviert oder eine Nachbestellung veranlasst werden. Das Ergebnis der Verarbeitung soll dem Kunden angezeigt werden. Dieser Anwendungsfall hat es in sich. Einen Auftrag im System verbuchen ist einfach zu implementieren. Interessant ist die Reservierung von Lagerbeständen und die Nachbestellung. Die Lagerverwaltung wird vermutlich von einem anderen SCS verantwortet, genauso wie die Nachbestellungen. Die Kommunikation soll asynchron erfolgen, um die Systeme zu entkoppeln. Trotzdem soll dem Kunden nach dem Auftragseingang das Ergebnis der

Verarbeitung angezeigt werden. Welche Optionen haben wir? Aus Sicht von Design-Patterns ist das SAGA-Pattern interessant. Das SAGA-Pattern soll helfen verteilte Transaktionen zu implementieren. Neben der technischen Implementierung des Patterns müssen auch die fachlichen Anforderungen implementiert werden, wie der Aufruf der Lagerverwaltung und der Nachbestellung inklusive Fehlerbehandlung. Dies ist aus technischer Sicht herausfordernd genug. Gerade die Fehlerbehandlung ist in diesem Fall mehr fachlich als technisch getrieben. Was passiert, wenn der Aufruf der Lagerverwaltung zu lange dauert? Was passiert, wenn bei der Nachbestellung ein Fehler auftritt? Soll das System es nochmals versuchen? Wie breche ich die Verarbeitung komplett ab? Und wie bekomme ich das Ganze sinnvoll in das SCS integriert?

Eine mögliche Lösung ist hier der Einsatz einer leichtgewichtigen Workflow-Engine, welche in das Auftragsverwaltungs-SCS eingebettet ist. Eine Workflow-Engine bringt die technisch notwendige Struktur für die korrekte Umsetzung des SAGA-Patterns mit. Die Workflow-Definition kann sehr gut mit den Verantwortlichen abgestimmt und transparent gemacht werden. Der Fokus auf die Implementierung der Fachlogik geht nicht verloren. Eingebettete Workflow-Engines haben noch einen anderen Vorteil: In der Regel bildet die Workflow-Definition eine Einheit mit dem SCS. Ein Change an dem Einen betrifft meistens auch das Andere. Es macht daher Sinn, diese zusammen zu versionieren, zu pakettieren und auch zusammen zu deployen. In diesem Zusammenhang sei noch erwähnt, dass der Aufruf eines anderen Systems kein synchroner Aufruf sein sollte, da die systemübergreifende Kommunikation idealerweise asynchron erfolgen sollte. Der Aufruf ist also der Versand eines Events an das Zielsystem und das Warten auf ein entsprechendes Antwort-Event des Zielsystems inklusive Timeout-Behandlung usw. Gerade bei der Modellierung und Implementierung solcher Funktionalitäten helfen Workflow-Engines enorm.

In Verbindung mit dem **Kaufen und Lagerbestand reservieren** Anwendungsfall sieht die Implementierung dann so aus: Beim Klick auf den Kaufen-Button wird innerhalb des Auftragsmanagement-SCS der entsprechende Workflow über die Workflow-Engine gestartet. Wenn dieser Workflow innerhalb ein paar Dutzend Millisekunden erfolgreich durchläuft, kann als Antwort der Auftragsverwaltung dem Kunden der entsprechende Status des Auftrages angezeigt werden. Wenn der Workflow nicht komplett durchläuft, können wir auch dies dem Kunden mit einem entsprechenden Hinweis mitteilen und ihn dann später via E-Mail oder Push-Notification über die Verarbeitung benachrichtigen. Dieser Mechanismus hilft den Anwendungsfall korrekt zu implementieren, Lastspitzen auf den Systemen besser zu skalieren und auch noch eine kundenfreundliche Fehlerbehandlung zu implementieren.

Geschäftslogik soll in einem SCS gekapselt werden. Das bedeutet aber nicht, dass Java für die Implementierung aller

Anwendungsfälle genutzt werden muss. Gerade bei datenintensiven Anwendungen kann es sinnvoll sein, auf natives SQL zurückzugreifen anstatt ORM-Frameworks wie Hibernate zu verwenden. Der Einsatz von SQL unterwandert die Kapselung nicht, sondern bringt Geschäftslogik und Daten nur näher zusammen. Hier handelt es sich um ein Beispiel für die Mikroarchitektur, welche lokal auf ein SCS beschränkt ist. Diese Architektur und Designentscheidungen sollen vom zuständigen SCS-Team getroffen werden.

## Die Benutzerschnittstelle

Jedes Self-Contained-System soll die komplette Benutzerschnittstelle für die darin enthaltenen Anwendungsfälle beinhalten. Im Idealfall ist es eine Web-Applikation, sprich die Benutzerschnittstelle ist HTML, CSS und JavaScript. Im Zusammenhang mit SCS ist aber ein besonderes Augenmerk auf die Integration von mehreren SCS zu einer einheitlichen Benutzerschnittstelle zu lenken. Benutzer möchten unterschiedliche Funktionalitäten nutzen. Diese Funktionalitäten sollen via Link zu dem zuständigen SCS führen. Am Beispiel eines Online-Shops ergibt sich dann die Frage, wo zum Beispiel die Top-Navigation herkommt, die auf fast jeder Seite angezeigt wird. Soll diese Funktionalität redundant in mehreren SCS implementiert werden? Die Antwort lautet natürlich: nein.

Es gibt ein System, das für die Top-Navigation zuständig ist. Ein anderes System ist z.B. für die Berechnung von Empfehlungen zuständig und ein weiteres System kann Artikellisten anhand von Filterkriterien generieren. Jedes dieser Systeme liefert den HTML-Code für genau diesen Teil der Seite aus. Die einzelnen Seitenbestandteile müssen zu einer Einheit zusammengefasst werden. Hier gibt es mehrere Möglichkeiten. Eine Möglichkeit wäre, ein Content-Management-System zu nutzen, welches die Aggregation übernimmt. Diese Möglichkeit ist die mächtigste, stellt aber auch wieder die organisatorische Frage, wer dieses System betreut und weiterentwickelt. Zusätzlich entstehen auch neue Abhängigkeiten zwischen den zuständigen Teams und nicht zuletzt ein Single-Point-Of-Failure für die ganze Architektur. Eine andere, etwas leichtgewichtigere Möglichkeit ist, das Seitenlayout und Platzhalter für systemfremde Inhalte in Form von Templates in das SCS zu integrieren. Diese Platzhalter könnten SSI-Direktiven (Server-Side-Includes) oder ESI-Direktiven (Edge-Side-Includes) sein. Das Auflösen dieser Platzhalter würde dann von einem vorgelagerten Web-Server oder Varnish-Cache durchgeführt werden. In der Praxis ist eher eine leichtgewichtige Lösung ratsam, die später weiter ausgebaut werden kann, falls dies wirklich notwendig ist. Einmal in Produktion gebrachte Komplexität ist schwer zurückzubauen.

Beim Thema Edge-Side-Includes oder Caching sind noch Details bei personalisierten Inhalten zu beachten. Es soll eine möglichst hohe Cache-Hit-Rate erreicht werden. Personalisierte Inhalte sind meistens sehr schwer cachebar. Eine Lösung ist, immer das

gleiche HTML-Grundgerüst für alle Nutzer auszuspielen und die personalisierten Inhalte via JavaScript nachzuladen. Diese Strategie ist auch ohne Cache im Hintergrund sinnvoll. In Verbindung mit Progressive-Enhancement ergeben sich interessante Möglichkeiten. Warum Inhalte ausliefern, wenn sie auf dem Zielgerät nicht sichtbar sind, weil der Bildschirm zu klein ist? Warum Videos anzeigen, wenn das Zielgerät das nicht kann? All diese Fragen können nur auf dem Zielgerät und nicht auf dem Applikations-Server beantwortet werden. Der Server liefert nur das Grundgerüst der Seite über semantisches HTML aus. Der Rest wird auf dem Zielgerät entschieden und zusammengebaut.

Wie lässt sich eine einheitliche UX erreichen und gibt es eine zentrale Ablage für Styles? Was passiert mit JavaScript-Code? Jedes SCS enthält seinen eigenen JavaScript-Code sowie eigene Styles und liefert diese aus. Der Trick ist, einen komponentenbasierten Ansatz in Verbindung mit einer Modularisierung der Stylesheets und des JavaScripts zu fahren. Eine Komponente ist beispielsweise die Top-Navigation oder eine Artikelliste. Jede Komponente bringt das dafür notwendige HTML, CSS und JavaScript mit. Es sollten so wenige globale Styles wie möglich vorgegeben werden. Nur so können Funktionalitäten unabhängig und innerhalb eines SCS umgesetzt und damit letztendlich Geschwindigkeit in der Entwicklung und kurze Time-to-Market umgesetzt werden. Hier gilt es einen pragmatischen Weg zu finden.

Die Konsequenz ist, dass es kein globales Stylesheet mehr für all das gibt. Dies ist so gewollt, aus Sicht von UX aber erst einmal ein Albtraum. Bei genauerer Betrachtung von großen Online-Shops wie Amazon, stellt man jedoch auch fest, dass das Seitenlayout in den Artikellisten komplett anders ist als im Check-out. Vermutlich, weil diese Seiten von einem anderen System ausgeliefert werden. Vermutlich ist auch hier eine pragmatische Entscheidung zwischen einheitlichem Layout, Unabhängigkeit von Systemen und Teams sowie von Entwicklungszeit und Entwicklungskosten getroffen worden.

Dieser Abschnitt ist sehr HTML-lastig. HTML ist jedoch die Wahl für die Implementierung der Präsentationsschicht eines SCS. Problematisch wird es, wenn gemischte Präsentationstechnologien verwendet werden müssen. Dies ist besonders der

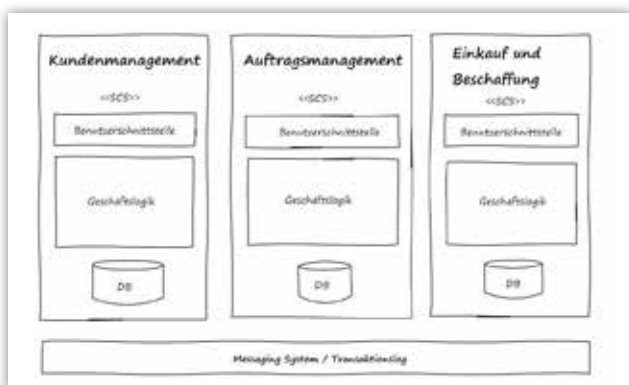
Fall, wenn bestimmte Anwendungsfälle via App zur Verfügung gestellt werden sollen. Wenn es sich bei der App um eine hybride App handelt und bereits eine Web-View integriert ist, kann HTML verwendet werden. Soll die App allerdings die nativen Möglichkeiten des Endgerätes nutzen oder komplett offlinefähig sein, so muss für die App eine andere Schnittstelle zur Verfügung gestellt werden. In diesem Fall stößt der SCS-Ansatz an seine Grenzen, da die Benutzerschnittstelle nicht mehr Teil des SCS ist, und vermutlich auch die App-Entwicklung nicht mehr Teil des eigentlichen SCS-Teams ist. In diesem Fall wäre es ein Kompromiss, das SCS um eine (REST-)API zu erweitern und diese der App zur Verfügung zu stellen. Die Funktionalität bleibt dann im SCS gekapselt und nur der App-spezifische Teil der Präsentationsschicht wird externalisiert (**Abb. 2**).

### Fazit:

Self-Contained-Systems können helfen, die Arbeit größerer Entwicklungsteams zu organisieren und zu strukturieren. Aber auch für kleinere Projekte und Teams kann es ein interessanter Ansatz sein, welcher den Fokus klar auf Anforderungen und nicht auf technische Implementierungsdetails lenkt. Die Prinzipien hinter Domain-Driven-Design werden hier angewendet. Der Grundsatz „Weniger ist mehr“ zeigt sich auch in den aktuellen Trends der Softwareentwicklung und der Rückkehr zu wenigen, aber dafür wichtigen und richtigen Services. Wie bei allen Architekturentscheidungen handelt es sich auch hier nicht um die Entscheidung zwischen Schwarz oder Weiß. Es ist immer ein Abwägen zwischen Vor- und Nachteilen und das Leben mit den damit verbundenen Konsequenzen.

Welches Framework und welche Programmiersprache sind nun die besten, um ein SCS zu implementieren? Grundsätzlich sind alle geeignet, solange sich damit die fachlichen Anforderungen umsetzen lassen und natürlich auch das Entwicklungsteam damit produktiv und nachhaltig arbeiten kann. Der SCS-Ansatz soll gerade dabei helfen, pro SCS die jeweils sinnvollste Technologie zu verwenden. In der Praxis muss hier ein Kompromiss zwischen innovativen Ansätzen und technologischen Wildwuchs gefunden werden.

Kritisch betrachtet ist der SCS-Architekturansatz zu grobgranular und lässt zu viele Details offen. Gerade der Punkt Makroarchitektur könnte mehr Details und Hinweise liefern, da genau hier die Faktoren für eine erfolgreiche Einführung versteckt liegen.



Systemübersicht. (Abb. 2)

### Quellen:

- 1 Self-Contained-Systems - <https://bit.ly/scs-s1>
- 2 Evans, Eric: Domain-Driven-Design: Tackling Complexity in the Heart of Software
- 3 Apache Kafka - <https://bit.ly/kafka-a3>
- 4 SAGA Pattern - <https://bit.ly/saga-s4>
- 5 Zeebe - <https://bit.ly/zeebe-z5>
- 6 Canoo ULC - <https://bit.ly/ulc-c6>

#JAVAPRO #Architecture

# Architektur-Hotspots aufspüren

Strukturelle Abhängigkeitsanalysen sind von begrenzter Aussagekraft. Teilweise erfassen sie die wesentlichen Architektur-Hotspots nicht, wie anhand eines Code-Beispiels exemplarisch gezeigt wird. Die vorgestellten Konzepte zur Feature-Modularität sind eine neue Art die System-Komplexität zu erfassen. Sie zielen darauf ab, Architektur-Hotspots zu identifizieren, die den (Wartungs-) Aufwand für neue Features ansteigen lassen.

**E**in weit verbreiteter Ansatz zur Bewertung von Softwaresystemen und deren Architekturen besteht in der Analyse ihrer Abhängigkeitsstruktur. Ziel ist es Abhängigkeitsstrukturen zu identifizieren, die allgemein als schwer wartbar bekannt sind, zum Beispiel zyklische Abhängigkeiten und bestimmte systemspezifische Verletzungen wie schichtübergreifende Abhängigkeiten. Es besteht kein Zweifel, dass die Abhängigkeitsanalyse wertvolle Einblicke in die Struktur eines Softwaresystems liefert und tatsächlich echte Wartungsprobleme aufdecken kann. Trotz ihrer Nützlichkeit hat die klassische Abhängigkeitsanalyse aber große Schwächen.

## Schwächen klassischer Abhängigkeitsanalysen

Sie berücksichtigt nur strukturelle Abhängigkeiten, zum Beispiel Aufruf- und Import-Abhängigkeiten. Sie versäumt es implizite Abhängigkeiten zu erkennen, wodurch verschiedene Artefakte (Dateien, Module, Klassen etc.) gleichzeitig angepasst werden müssen, ohne dass eine sichtbare Abhängigkeit zwischen ihnen besteht. Zudem ergibt die strukturelle Abhängigkeitsanalyse oft positive aber falsche Ergebnisse. Auch wenn problematische Abhängigkeitsstrukturen sicher auf potenzielle Schwierigkeiten hindeuten, müssen sie im Sinne eines erhöhten Wartungsaufwandes keine wirklichen Probleme verursachen. Die Gründe dafür können unterschiedlich sein:

- Code-Module, die von schlechten Abhängigkeitsstrukturen betroffen sind, laufen stabil und werden während des

Lebenszyklus eines Softwaresystems nicht sehr oft verändert.

- Die Entwickler sind sich der potenziell problematischen Stellen bewusst und berücksichtigen die Auswirkungen bei der Implementierung entsprechend, so dass unangenehme Überraschungen wie Folgefehler vermieden werden.
- Nicht jede zyklische Abhängigkeit ist für sich genommen problematisch. Einige könnten sogar absichtlich eingeführt werden, um große Module in kleinere aufzuteilen. Dabei sind die kleineren Module als Teil der Implementierung und weniger als Teil des Designs der Architektur anzusehen.

## Autor:

Konstantin Sokolov ist Mit-Gründer der Cape of Good Code GmbH. Er hat deren DETANGLE Software Analyse Suite, die zur Software Qualitätssicherung, Qualitätsaudits und Technical Due Diligence Projekten eingesetzt werden, maßgeblich mitgestaltet.



Ural Sezer ist Senior Engineer und Experte für Frontend-Technologien, die unter anderem bei DETANGLE zum Einsatz kommen.



Im Allgemeinen versucht die klassische Abhängigkeitsanalyse Fälle einer engen Kopplung aufzudecken. Dabei bleibt sie jedoch oft zu feingranular und kurzfristig und lässt das große Ganze außer Acht. In realen Systemen können strukturelle Kopplungen zwischen einzelnen Codemodulen oder deren Fehlen nicht als ausreichende Bedingung für eine schlechte bzw. gute Wartbarkeit angesehen werden. Es besteht kein Zweifel, dass das Vorhandensein einer engen Kopplung ein starker Indikator für Probleme bei der Wartbarkeit ist. Diese Zuversicht folgt auch aus der Erfahrung. Es scheint jedoch, dass allein die Betrachtung struktureller Kopplungen zwischen einzelnen Codemodulen zu ungenau ist. Welche Art von Kopplung ist also im Hinblick auf die Wartbarkeit wirklich wichtig?

### Von der Modulkopplung zur Idee der Feature-Kopplung

Die Beantwortung dieser Frage erfordert eine Definition der Wartbarkeit. Eine einfache könnte sein:

„Der Aufwand der erforderlich ist, um neue Funktionalitäten in ein wartbares System einzuführen, ist ungefähr proportional zur Komplexität der Funktionalität, jedoch nicht zur Komplexität des gesamten Systems.“

Einfach ausgedrückt: In einem wartbaren System sollte es nicht der Fall sein, dass die Implementierung neuer Features immer schwieriger wird und immer mehr Zeit in Anspruch nimmt. Umgekehrt bedeutet dies, dass ein System unabhängig von der Qualität seiner Abhängigkeitsstruktur oder möglicherweise anderer Metriken als wartbar zu betrachten ist, solange der durchschnittliche Aufwand pro Feature nicht weiter steigt. Wen kümmert eine enge strukturelle Kopplung und zyklische Abhängigkeiten, sofern sie nicht den gesamten Entwicklungsprozess verlangsamen? Und was ist eine strukturelle Modularisierung mit einer schönen Abhängigkeitsstruktur wert, wenn man immer beständig Projekttermine verpasst?

Ein genauerer Indikator für den potenziellen Aufwand zur Einführung neuer Features in ein System muss gefunden werden. Und dieser Indikator sollte sich wohl am besten auf die Funktionalität und deren Implementierung fokussieren. Features sind als Hauptbausteine von Softwaresystemen zu behandeln - und nicht Codemodule, Dateien oder Klassen. Es ist zu erwarten, dass die Analyse und Verbesserung der Beziehungen zwischen Features, und nicht zwischen Code-Modulen allein, genauere Aussagen über die Wartbarkeit eines Systems ermöglichen. In der Regel kann ein Feature in einem System umso schneller implementiert werden, je weniger umgebender Code verstanden, geändert und erneut getestet werden muss und je weniger unverwandter Code, der für andere Features verantwortlich ist, betroffen ist. Je weniger sich eine Feature-Implementierung mit anderen Feature-Implementierungen überschneidet und diese möglicherweise stört, desto eher ist es möglich sie in einer angemessenen

Zeit zu implementieren ohne die bestehende Funktionalität zu beeinträchtigen. Der Grad der Schnittmenge eines Features mit anderen Features im Code wird hier als Feature-Kopplung verstanden. In diesem Zusammenhang versteht man unter einem Feature:

- Eine von Menschen lesbare Beschreibung der erwarteten Funktionalität, der eine eindeutige ID zugewiesen wird. Normalerweise werden Features in Issue-Tracker-Systemen gepflegt und als Tickets mit entsprechenden Ticket- oder Issue-IDs ausgedrückt.
- Ein Satz von Commits in einem Versionskontrollsystem, die erforderlich sind um die notwendigen Code-Änderungen einzuführen, welche die Funktionalität des Features ausmachen.

Um die Feature-Kopplung programmatisch analysieren zu können muss jeder Commit, der zu einem Feature gehört, mit der entsprechenden Feature-ID verknüpft werden.

### Feature-Kopplung an einem Code-Beispiel

In diesem Abschnitt wird ein Beispiel skizzenhaft in Java implementiert. Es geht darum, Objekte in der realen Welt zu erkennen und sie auf lustige Weise zu dekorieren. Zu diesem Zweck hat die Software Zugriff auf eine drehbare Kamera. Für den ersten Meilenstein sind die folgenden User-Stories zu implementieren:

- User-Story #1: Der Benutzer möchte vom System erkannte Autos lustig dekorieren (Lustige Autos)
- User-Story #2: Der Benutzer möchte vom System erkannte Personen lustig dekorieren können (Lustige Personen)

Abkürzend wird im Folgenden an manchen Stellen User-Story #1 als das Feature "Lustige Autos" und User Story #2 als "Lustige Personen" referenziert. Als Teil von User-Story #1 wird Skeleton-Code für die Darstellung und Wahrnehmung der Welt erstellt. Es werden zwei neue Dateien hinzugefügt: **WorldModel.java** und **WorldParser.java** (Listing 1 & Listing 2).

#### (Listing 1 - WorldModel.java mit Commit 1)

```
import WorldObject
import Observer

public final class WorldModel {
    public enum EventType {}
    private HashMap<EventType, List<Observer>>
        eventObserverMap = new HashMap<>();

    public WorldModel() {
    }

    public void subscribe(Observer observer, EventType event) {
        var observerList =
            eventObserverMap.getOrDefault(event, new Array-
                List<>());
```

```

        observerList.add(observer);
        eventObserverMap.put(event, observerList);
    }

    private void notify(EventType event, WorldObject obj) {
        var observerList =
            eventObserverMap.getOrDefault(event, new Array-
            List<>());
        observerList.forEach(observer -> observer.onEvent(event,
        obj));
    }
}

```

(Listing 2 - WorldParser.java mit Commit 1)

```

import Frame
import Camera
import WorldModel

public class WorldParser {
    private Camera cam = new Camera();

    public WorldParser() {
    }

    public void mainLoop() {
        while (true) {
            processFramesFor10Seconds();
            cam.rotate(45);
        }
    }

    public void processFramesFor10Seconds() {
        var iterator = cam.getFrameIterator();
        while (iterator.hasNext()) {
            parseFrame(iterator.next());
        }
    }

    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
    }

    private byte[] lowLevelProcessing1(Frame frame) {
        // retrieve first-level raw data
        ...
    }
    ...
}

```

Die WorldModel-Klasse dient als Container von Objekten, die derzeit in der Welt repräsentiert werden, und implementiert das Observer-Muster, um bestimmte Ereignisse im Zusammenhang mit diesen Objekten zu verteilen. Die **WorldParser** Klasse beobachtet die Umgebung durch Drehen der Kamera, um konkrete Objekte zu erkennen. Weitere drei Commits implementieren die Funktionalität hinter User-Story #1, womit sich sukzessive Codes (Listing 3, Listing 4 und Listing 5) ergibt.

(Listing 3 - WorldModel.java mit Commit 2)

```

public final class WorldModel {

    public enum EventType { CAR_APPEARED }

    private ArrayList<Car> cars = new ArrayList<>();

    ...

    public void addCar(Car car) {
        cars.add(car);
        notify(EventType.CAR_APPEARED, car);
    }
}

```

(Listing 4 - WorldParser.java mit Commit 3)

```

public class WorldParser {
    ...

    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
        var car = findCar(data3);
        car.ifPresent(value -> WorldModel.getInstance().
        addCar(value));
    }

    private Optional<Car> findCar(byte[] data) {
        ...
    }
}

```

(Listing 5 - FeatureFunnyCar.java mit Commit 4)

```

public class FeatureFunnyCar extends Observer {

    public FeatureFunnyCar() {
        WorldModel.getInstance().subscribe(
            this, WorldModel.EventType.CAR_APPEARED);
    }

    @Override
    public void onEvent(WorldModel.EventType event,
        WorldObject obj) {
        makeFunnyCar((Car)obj);
    }

    private void makeFunnyCar(Car car) {
        ...
    }
}

```

Nun wird User-Story #2 (Personen lustig dekorieren) ebenfalls mittels sehr ähnlicher drei Commits umgesetzt, so dass sich am Ende zusätzlich zum neuen Code-Modul **FeatureFunny Person.java** die Module **WorldModel.java** (Listing 6) und **World Parser.java** (Listing 7) ergeben.

(Listing 6 - WorldModel.java mit Commit 7)

```
public final class WorldModel {

    public enum EventType { CAR_APPEARED, PERSON_APPEARED }

    private ArrayList<Car> cars = new ArrayList<>();
    private ArrayList<Person> persons = new ArrayList<>();

    ...

    public void addCar(Car car) {
        cars.add(car);
        notify(EventType.CAR_APPEARED, car);
    }

    public void addPerson(Person person) {
        persons.add(person);
        notify(EventType.PERSON_APPEARED, person);
    }

}
```

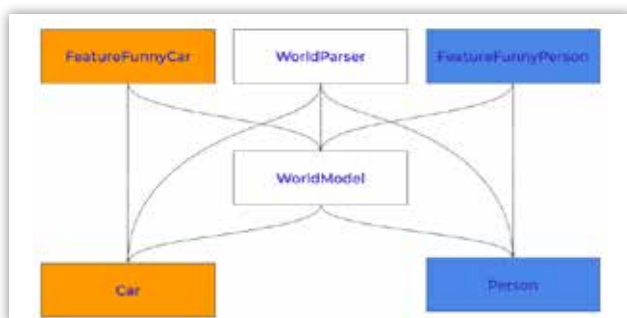
(Listing 7 - WorldParser.java mit Commit 7)

```
public class WorldParser {
    ...
    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
        var car = findCar(data3);
        car.ifPresent(value -> WorldModel.getInstance().
            addCar(value));
        var person = findPerson(data3);
        person.ifPresent(value ->
            WorldModel.getInstance().addPerson(value));
    }

    private Optional<Car> findCar(byte[] data) {
        ...
    }

    private Optional<Person> findPerson(byte[] data) {
        ...
    }
}
```

Die bisherige Commit-Historie sieht wie in (Tab. 1.) dargestellt aus.



Abhängigkeitsstruktur - hierarchisch strukturiert, azyklisch, aber der Schein trügt (Abb. 1)

|   | Commit-Message                                                        | Code-Module                                     |
|---|-----------------------------------------------------------------------|-------------------------------------------------|
| 7 | US #2: Funktion "Lustige Personen" hinzufügen.                        | FeatureFunnyPersons.java hinzufügen             |
| 6 | US #2: Erweitern des Parsers, um Personen zu erkennen                 | WorldParser.java anpassen                       |
| 5 | US #2: Erweitern des Modells, um Personen darzustellen                | WorldModel.java anpassen                        |
| 4 | US #1: Funktion "Lustige Autos" hinzufügen.                           | FeatureFunnyCars.java hinzufügen                |
| 3 | US #1: Erweitern des Parsers, um Autos zu erkennen                    | WorldParser.java ändern                         |
| 2 | US #1: Erweitern des Modells, um Autos darzustellen                   | WorldModel.java ändern                          |
| 1 | US #1: Skeleton-Code zur Welt Darstellung und -wahrnehmung hinzufügen | WorldModel.java und WorldParser.java hinzufügen |

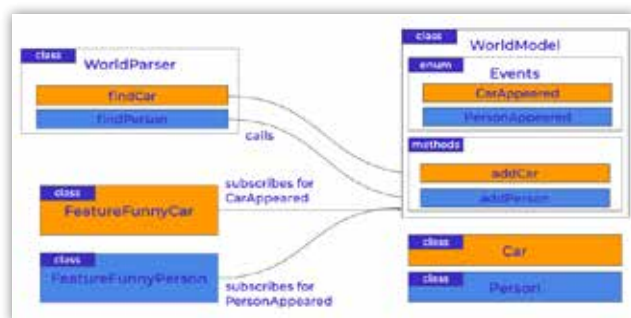
Bisherige Commit-Historie (Tab. 1)

Betrachten wir nun die Abhängigkeitsstruktur der aktuellen Codebasis (Revision 7), um erste Einblicke in die architektonische Qualität unseres Projekts zu erhalten (Abb. 1).

Auf den ersten Blick sieht die modulare Struktur des Code-Beispiels recht überzeugend aus. Es wurde eine hierarchische und azyklische Abhängigkeitsstruktur erreicht. Aber an den Abhängigkeiten des **WorldModel** zu **Car** und **Person** deuten sich die Schwächen schon an. Ein anderes detaillierteres Bild macht die Problematik noch ersichtlicher. Die orangen Markierungen kennzeichnen die Änderungen, die aufgrund User-Story #1 nötig waren, während die blauen Markierungen die Anpassungen gemäß User-Story #2 darlegen (Abb. 2)

Diese Architektur ist eher suboptimal:

1. Beim Hinzufügen jedes neuen Features müssen fast alle Module im System verändert werden.
2. In der **WorldParser** Klasse werden Erkennungsalgorithmen

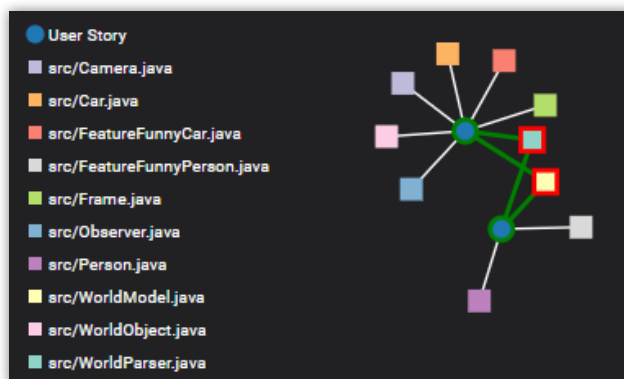


Abhängigkeitsstruktur - gut, aber mit unerkannten Architektur-Hotspots (Abb. 2)

für verschiedene Objekte vermischt, das Modul wird wachsen und unverständlich werden.

3. In der **WorldModel** Klasse ist die Situation aber noch schlimmer, weil das Code-Modul für jedes Feature an drei unterschiedlichen Stellen verändert werden muss.

Es kann argumentiert werden, dass die Architektur problematisch ist, dies aber nicht an der Abhängigkeitsstruktur erkennbar ist. Die gleichzeitig nötigen Anpassungen der Klassen **WorldModel** und **WorldParser** im Falle weiterer Features der Art „Lustige X“ werden nicht sichtbar. Daher ergibt sich die Frage: woran können wir es dann erkennen? Es wird anhand der Feature-Kopplung und deren Visualisierung ersichtlich (Abb. 3).



Feature-Kopplung visualisieren - Architektur-Hostspots erkennbar (Abb. 3)

Das Netzwerk-Diagramm beinhaltet blaue Kreise, welche die ersten beiden User-Stories darstellen. Jede User-Story ist per Kante mit den Code-Modulen verbunden, die zu ihrer jeweiligen Implementierung beigetragen haben. Code-Module werden als Rechtecke visualisiert. Es wird deutlich, dass **WorldModel.java** und **WorldParser.java** als Code-Module aufgrund beider User-Stories verändert wurden. Die Umsetzung einer weiteren User-Story zur Erfassung eines neuen Features zur Dekoration anderer Objekttypen würde wiederum diese beiden Code-Module anpassen. Diese beiden Code-Module weisen daher Verbindungen zu vielen gemeinsamen User-Stories auf. Sie stellen rot markierte architekturelle Hotspots des Systems dar, die anhand der klassischen Abhängigkeitsgraphen nicht sichtbar werden. Diese beiden Code-Module weisen eine hohe Feature-Kopplung auf, d.h. sie tragen gemeinsam zu vielen gleichen User-Stories (und dem Feature dahinter) bei.

### Komplizierte Abhängigkeiten, bessere Feature-Kopplung

Nun wird ein Refactoring durchgeführt. Die Arbeiten zum Refactoring

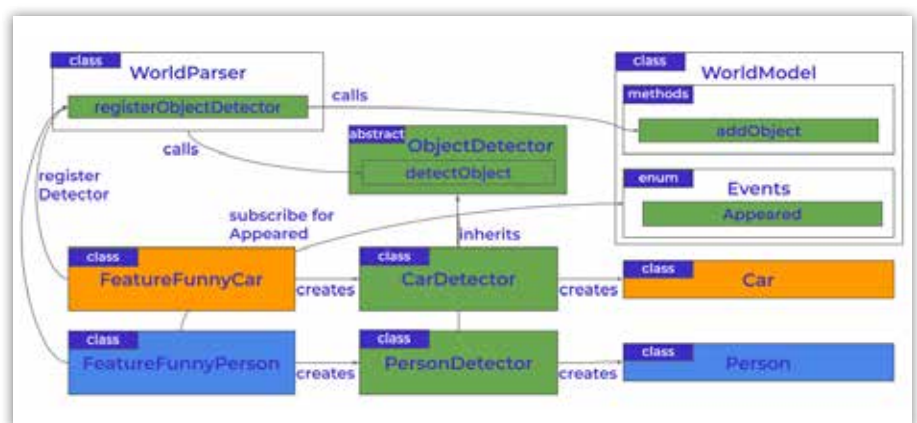
|     | Commit-Message                                                                   | Code-Module                                                                                                                                                                                                                                                                |
|-----|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8   | DR #3:<br>Refactoring Modell/<br>Parser mittels<br>Detector-Klassen              | <b>WorldParser.java</b> ändern<br><b>WorldModel.java</b> ändern<br><b>FeatureFunnyCar.java</b> ändern<br><b>FeatureFunnyPerson.java</b> ändern<br><br><b>ObjectDetector.java</b> hinzufügen<br><b>CarDetector.java</b> hinzufügen<br><b>PersonDetector.java</b> hinzufügen |
| ... | ...                                                                              | ...                                                                                                                                                                                                                                                                        |
| 1   | US #1:<br>Skeleton-Code zur<br>Weltdarstellung<br>und -wahrnehmung<br>hinzufügen | <b>WorldModel.java</b> und<br><b>WorldParser.java</b> hinzufügen                                                                                                                                                                                                           |

Commit-Änderungen (Tab. 2)

werden als eigenes Development-Requirement mit der ID #3 und dem Titel **Refactoring zur Entkoppelung des Modells und des Parsers** erfasst. Damit gehen Commit-Änderungen einher (Tab. 2).

Es wurde für jeden zu erkennenden Objekttyp eine eigene Detector-Klasse **CarDetector** und **PersonDetector** eingeführt, die von einer gemeinsamen abstrakten Klasse **ObjectDetector** abgeleitet ist. Bestehende und künftige Features, z.B. **Feature FunnyCar**, zur Dekoration von Objekten instanzieren nun einen jeweiligen **ObjectDetector** (z.B. **CarDetector**), um ihn beim Modell zu registrieren. Der jeweilige Detector erkennt das zugehörige Objekt und erzeugt das entsprechende Modell-Objekt, zum Beispiel **Car**. Das Diagramm (Abb. 4) skizziert die Kollaboration der Klassen nach dem Refactoring.

Der Vollständigkeit halber seien noch die revidierten Code-Stände der Klassen **WorldModel** und **WorldParser** aufgezeigt (Listing 8 und 9).



Feature-Kopplung visualisieren - Architektur-Hostspots ersichtlich (Abb. 4)

(Listing 8 - WorldParser.java mit Commit 8)

```
public class WorldParser {
    ...
    private ArrayList<ObjectDetector> objectDetectors = new
    ArrayList<>();

    public void registerObjectDetector(ObjectDetector
    objectDetector) {
        objectDetectors.add(objectDetector);
    }

    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
        var car = findCar(data3);
        car.ifPresent(value -> WorldModel.getInstance().
        addCar(value));
        var person = findPerson(data3);
        person.ifPresent(value ->
        WorldModel.getInstance().addPerson(value));
        for (ObjectDetector objectDetector : objectDetectors) {
            var obj = objectDetector.detectObject(data3);
            obj.ifPresent(value ->
                WorldModel.getInstance().addObject(value));
        }
    }
}
```

(Listing 9 - WorldModel.java mit Commit 8)

```
public final class WorldModel {

    public enum EventType { APPEARED }

    private ArrayList<Car> cars = new ArrayList<>();
    private ArrayList<Person> persons = new ArrayList<>();

    private HashMap<EventType, List<Observer>> eventObserverMap =
        new HashMap<>();

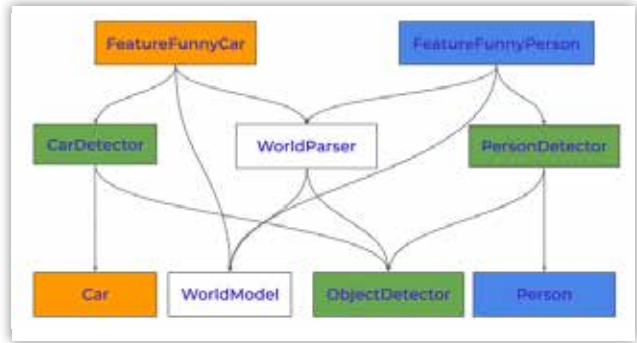
    private HashMap<Class<? extends WorldObject>, ArrayList<WorldObject>>
        typeObjMap= new HashMap<>();
    ...

    public void addObject(WorldObject obj) {
        var objList =
            typeObjMap.getDefault(obj.getClass(), new ArrayList<>());
        objList.add(obj);
        typeObjMap.put(obj.getClass(), objList);

    public void addCar(Car car) {
        ...
    }

    public void addPerson(Person person) {
        ...
    }
}
```

In (**Abb. 5**) ist eindeutig zu erkennen, dass die Abhängigkeitsstruktur komplizierter geworden ist. Dennoch stellt das Refactoring eine gravierende Architekturverbesserung dar, was sich



### Abhängigkeitsstruktur - komplizierter nach dem Refactoring (Abb. 5)

beim Hinzufügen eines weiteren Features erkennbar macht. Im Netzwerk-Diagramm in (**Abb. 6**) sind zwei neue User-Stories aufgeführt. Darunter User-Story #4: Der Benutzer möchte vom System erkannte "Gebäude lustig dekorieren" (Lustige Gebäude) und eine entsprechende User-Story zu „Lustigen Bäumen“. Es lässt sich sofort erkennen, dass die Klassen **WorldModel** und **WorldParser** keiner weiteren Anpassung für User-Story #4 und #5 (der blaue Kreis links oben und rechts unten) bedürfen. Es werden nur neue Code-Module für User-Story #4 hinzugefügt, wie zum Beispiel **FeatureFunnyBuildings.java**, **BuildingDetector.java** und **Building.java**. Sie sind nun recht straight forward zu implementieren, so dass deren Listings aus Platzgründen weggelassen wurden.



### Feature-Kopplung - neue Features sind nicht mehr gekoppelt (Abb. 6)

Diese neuen Code-Module weisen keine Feature-Kopplung auf, denn sie tragen zu genau einer User-Story bzw. einem Feature bei. In dem Cluster in der Mitte ist auch das Development-Requirement mit dem Refactoring als rot eingerahmtes Issue markiert. Da Refactorings meistens systemübergreifend sind, wurden dafür viele der bereits bestehenden Code-Module angefasst. Daher sind Refactorings für eine Evaluierung der Feature-Kopplung nicht zu berücksichtigen. Aus diesen und anderen Gründen sollten Refactorings übrigens als eigene Issues erfasst werden<sup>1</sup>.

## Feature-Modularität und seine visuellen Muster

Das höhere Ziel der Softwareentwicklung besteht im Erreichen von Code-Modularität, soweit herrscht weitestgehend Einigkeit.

Eine bisher oft betonte Ausprägung ist die bereits erwähnte Qualität der Abhängigkeitsstruktur, die sich durch eine hierarchische und azyklische Ausprägung auszeichnet. Die Beschränkungen dieser Sichtweise wurden bereits dargelegt. Eine weitere Art Code-Modularität zu erfassen, ist sicherlich das Single-Responsibility-Prinzip (SRP), von dem zwei Formulierungen bekannt sind:

„Ein Modul sollte einem, und nur einem, Akteur gegenüber verantwortlich sein.“  
Robert C. Martin<sup>2</sup>

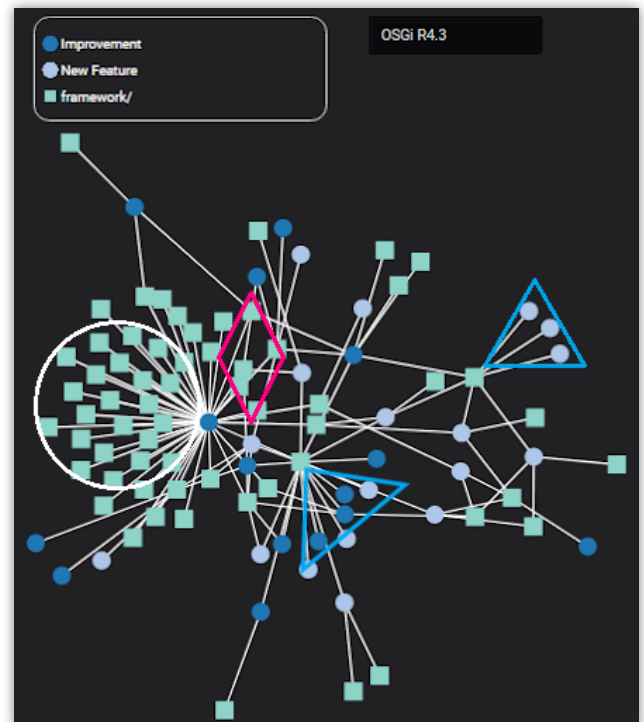
„A functional unit on a given level of abstraction should only be responsible for a single aspect of a system's requirements. An aspect of requirements is a trait or property of requirements, which can change independently of other aspects.“  
Ralf Westphal<sup>3</sup>

Laut Robert C. Martin geht es beim SRP nicht nur um die einzelnen Klassen oder Funktionen, sondern vielmehr um, durch die Anforderungen eines Akteurs definierten Sammlungen an Funktionalitäten und Datenstrukturen. Ralf Westphal verweist auf den ähnlichen Punkt, dass sich das SRP auf Requirements bezieht. Bisher gibt es keine allgemein akzeptierte Fassung dieses Prinzips trotz seiner intuitiv eingängigen Aussage. Ralf Westphal bezieht sich eher auf eine eigenständige Eigenschaft, die mehreren Requirements zugrunde liegt. Robert C. Martin bezieht sich hingegen eher auf einen Satz von Funktionalitäten, die auf einen Akteur bzw. User zurückgehen. Eine pragmatische Ausprägung wäre:

„Ein Code-Modul sollte weitestgehend zu einem Requirement, einer Funktionalität oder einem Feature beitragen.“

Analysen zeigten, dass Code-Module die dem zuwiderlaufen, nach dem Hinzufügen neuer Features oftmals von vielen Bugs und Wartungsaufwänden betroffen sind.

Die eingangs erwähnte Systemkomplexität, eine Messzahl die ausdrückt wo im System mit wesentlich mehr Aufwand zur Feature-Umsetzung zu rechnen ist, wird mit Kennzahlen der Feature-Modularität berechnet. Die Konzepte der Feature-Kopplung und Feature-Kohäsion lassen sich wieder anhand visueller Muster im Netzwerkgraphen erklären (Abb. 7), hier am Beispiel des Open-Source-Projekts Apache-Felix. Wiederum werden dunkelblaue und hellblaue Kreise für die Issues vom Typ **Feature** und **Improvement** sowie türkise Rechtecke für die **Code-Module** genutzt. Feature-Kohäsion misst den Beitrag einer Datei zu verschiedenen Features. Je größer die Anzahl der Features zu denen eine Datei beiträgt, desto stärker verletzt sie das Single-Responsibility-Prinzip, und desto kleiner ist ihre Kohäsion. Im Netzwerkgraphen sind das visuelle Dreiecke, die in (Abb. 7) blau markiert sind. Dieses visuelle Muster wird auch



Feature-Modularität - visuelle Muster (Abb. 7)

Module-Tangle genannt. Tangle steht für Knäuel, Wirrwarr, was eben dieses Modul mit seinen Beiträgen zu mehreren Features/Improvements auch darstellt. Für die Weiterentwicklung dieser Module muss erhöhter kognitiver Aufwand aufgebracht werden, um unbeabsichtigte Seiteneffekte zu vermeiden.

Die Feature-Kopplung wiederum misst die Überlappung oder Überschneidung von Features über Dateien hinweg. Viele Module tragen gemeinsam zu vielen Features auf eine chaotische Art und Weise bei. Ein Beispiel ist in (Abb. 7) als pinkes Trapez markiert. Dieses visuelle Muster wird als Diamond-Tangles bezeichnet. Diamanten deswegen, weil dies die architekturellen Hotspots des Systems darstellen. Feature-Modularität lässt sich auch dazu nutzen, um die technischen Schulden der Features, die Feature-Qualitätsschulden, im Code zu messen. Das ist auch ein Ansatz, um den grundsätzlichen Konflikt im Software-Engineering zwischen Features und Qualität der Software anzugehen.

## Quellen:

- 1 Refactoring Issues: <https://bit.ly/issues-i1>
- 2 Robert C. Martin: "Clean Architecture: A Craftsman's Guide to Software Structure and Design"
- 3 Single-Responsibility-Principle - <https://bit.ly/principle-p3>

#JAVAPRO # Tools #Logging

# Loggen mit Struktur

Log-Server bieten weit mehr als eine Volltextsuche über textuelle Log-Meldungen an. Sie sind komplette Datenbanksysteme mit komplexen Auswertemöglichkeiten. Wenn Log-Meldungen passend zu den Datenbankfähigkeiten erstellt werden, ergeben sich neue Möglichkeiten im Umgang mit Log-Daten.

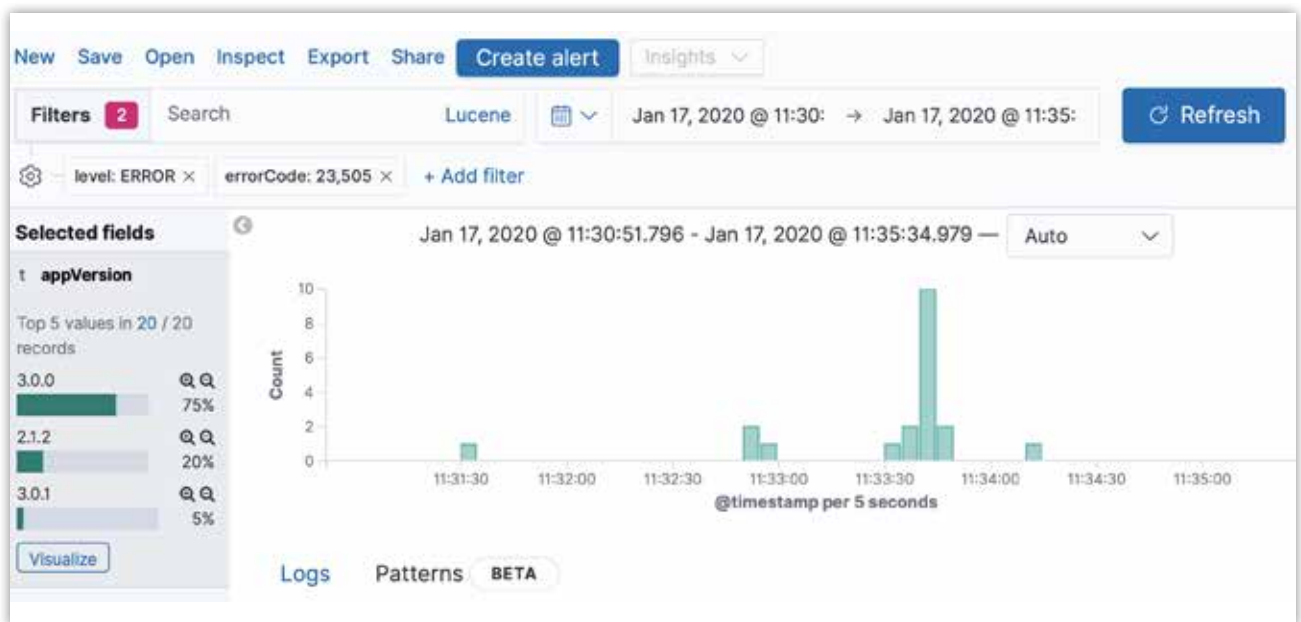
## Autor:



Roland Majchszak ist seit 2013 bei der adesso SE in Hamburg als Softwarearchitekt angestellt. Er beschäftigt sich mit der Planung und Umsetzung von Architekturen im Java-Umfeld. In letzter Zeit stehen dabei zunehmend die effiziente Umsetzung und der Betrieb von Microservice-Architekturen im Fokus.

Firmen-Webseite: [adesso.de](https://www.adesso.de)  
Email: [roland.majchszak@adesso.de](mailto:roland.majchszak@adesso.de)

Java-Log-Bibliotheken zum effizienten Schreiben von Log-Meldungen gibt es seit über 20 Jahren. Ursprünglich dienten sie als Hilfe beim Debuggen während der Entwicklung und zur Diagnose von Produktionsproblemen. Im produktiven Unternehmenseinsatz hatten Entwickler meistens keinen Zugriff auf die produktiven Log-Dateien. Die Dateien mussten erst von einem Administrator angefordert werden, der einem dann die hoffentlich richtigen Log-Dateien bereitstellte. Unter solchen Umständen dauerte es oft Stunden, bis ein Problem diagnostiziert werden konnte. Mit dem Aufkommen von Microservice-Architekturen und verteilten Anwendungen in der Cloud steigt die Anzahl der relevanten Log-Dateien rasant an. In der



Eine Auswertung in Kibana: Der Fehlercode 23505 taucht in drei verschiedenen Softwareversionen (appVersion) auf. (Abb. 1)

Vergangenheit gab es Projekte mit einer Microservice-Architektur, bei denen allein von den Java-Anwendungen über 40 Log-Dateien auf vier verschiedenen Maschinen gleichzeitig geschrieben wurden. Die Problemdiagnose nur mit Texteditoren und UNIX-Kommandozeilen-Tools ist bei einer solchen Anzahl von Dateien keine erfreuliche Aufgabe. Ein Log-Server, der über eine grafische Oberfläche eine Volltextsuche über alle Log-Dateien ermöglicht, ist unter diesen Umständen eine große Arbeitserleichterung. Da der Zugriff auf den Log-Server nicht den direkten Zugriff auf das Produktionssystem erfordert, ist es – abgesehen vom Datenschutz – auch unkritisch, dem Entwickler Zugriff auf die Produktions-Logs zu geben. Es dauert dann nicht mehr Stunden, bis ein Entwickler die richtigen Log-Meldungen sehen kann, sondern er kann ohne fremde Hilfe tätig werden und hat praktisch Live-Zugriff.

### Volltextsuche ist nicht genug

Die Datenbank eines Log-Servers speichert eine Log-Meldung nicht als Textzeile, sondern als Objekt mit mehreren Attributen. Die Textmeldung ist nur eines dieser Attribute. Andere Attribute sind zum Beispiel der Zeitstempel oder der Log-Level. Die Volltextsuchen der Log-Server sind zwar sehr schnell, Suchen auf Attributbasis sind jedoch nochmal eine Größenordnung schneller. Bei großen Log-Datenbeständen macht das einen spürbaren Unterschied. Außerdem sind Suchanfragen treffsicherer, wenn sie direkt auf ein Attribut abzielen. Wird zum Beispiel nach kurzen Zahlen über den vollen Text gesucht, gibt es regelmäßig Treffer in den Zeitstempeln. Bei der Suche nach einem Attribut sind solche False-Positives ausgeschlossen. In den GUIs der Log-Server gibt es außerdem statistische Auswertmöglichkeiten für Attributwerte. (Abb. 1) zeigt als Beispiel dafür, wie sich Meldungen mit einem gesuchten Fehlercode auf verschiedene Versionen der Software verteilen. Die gängigen Java-Log-Bibliotheken loggen

nicht nur Textmeldungen, sie können auch Log-Meldungen direkt als Objekte im JSON-Format loggen. Die Attribute können dann direkt in die Datenbank übernommen werden und stehen dort als Parameter für Abfragen bereit.

### Die Infrastruktur:

#### Log-Server und Java-Log-Bibliotheken

Log-Aggregierungs-Server sammeln die anfallenden Log-Daten ein, parsen die Log-Meldungen und speichern sie in einer Volltextdatenbank. Durch das Parsen werden die in unterschiedlichen Formaten anfallenden Log-Meldungen in ein einheitliches JSON-Format umgewandelt. Über grafische Oberflächen lassen sich die Log-Meldungen durchsuchen und anzeigen. Es gibt derzeit drei verbreitete Systeme: Der freie ELK-Stack besteht aus den Komponenten Elasticsearch als Volltextdatenbank, Logstash als Komponente zum Einsammeln sowie Parsen von Log-Daten und Kibana als grafische Benutzeroberfläche. Splunk ist ein verbreitetes kommerzielles Komplettsystem zur Log-Verarbeitung im Enterprise-Umfeld. Graylog gilt als kostengünstigere Alternative zu Splunk. Es gibt Graylog sowohl in einer Open-Source-Variante, als auch in einer kommerziellen Enterprise-Version. Es gibt Anbieter, die Cloud-basierte ELK-Stacks bereitstellen, die für einen Testzeitraum auch kostenlos ausprobiert werden dürfen.

Für das Erzeugen von Log-Meldungen gibt es aktuell zwei relevante und verbreitete Java-Log-Bibliotheken: Log4j2 und Logback. Beide stammen vom mittlerweile veralteten Log4j ab. Bezüglich Features und Performance liefern sich Logback und Log4j2 ein Kopf an Kopf Rennen. Das sehr verbreitete sl4j ist keine eigenständige Log-Bibliothek, sondern eine Fassade, die eine einheitliche API sowohl für Logback als auch für Log4j2 und andere weniger verbreitete Bibliotheken bietet. Durch sl4j werden direkte Abhängigkeiten im Code zur konkreten Log-Bibliothek

vermieden. Für Code in Bibliotheken oder Frameworks ist nicht sicher, wie und von wem er später verwendet wird. Hier sollte immer `sl4j` genutzt werden, um die Entscheidung für eine konkrete Log-Bibliothek dem späteren Verwender des Codes zu überlassen.

## Log-Meldungen richtig formatieren

Sowohl bei Log4j2 als auch bei Logback sind das Erzeugen von Log-Meldungen im Code und das Ausgeben einer Log-Meldung voneinander entkoppelt. In einer Konfigurationsdatei wird jeweils angegeben, wie Log-Meldungen ausgegeben werden sollen. Bei Log4j2 wird dazu ein Appender und eventuell ein Layout konfiguriert. Bei Logback hingegen ein Encoder. Log-Meldungen können direkt über das Netzwerk an einen Log-Server geschickt werden. Dies vereinfacht die Konfiguration des Gesamtsystems, da der Transport und das Parsen von Log-Dateien nicht selbst organisiert werden muss. Allerdings ist dann die Verfügbarkeit der Applikation an eine hinreichend schnelle und stabile Netzwerkverbindung zum Log-Server gekoppelt. Wird erst in eine Datei geloggt und dann diese Datei zum Log-Server transportiert, führt das zu einer höheren Resilienz.

Beim Schreiben in eine Log-Datei muss das textuelle Format der Log-Meldungen festgelegt werden. Da Log-Server die Daten im JSON-Format erwarten, ist es am einfachsten, gleich JSON zu loggen. Die Konfiguration der Log-Parser vereinfacht sich und zur Laufzeit wird die Rechenleistung für das Parsen reduziert. Sowohl für Log4j2 als auch für Logback gibt es passende Layouts, beziehungsweise Encoder für JSON. (Listing 1) zeigt ein Beispiel für Logback.

Log-Dateien im JSON-Format sind für Menschen schwerer zu lesen als normale Text-Log-Meldungen. Das UNIX-Kommandozeilen-Tool `jq` hilft, JSON-Meldungen lesbarer auszugeben. Zusätzlich zu einer Log-Datei im JSON-Format lassen sich in der Entwicklungsumgebung Log-Meldungen auch im Textformat auf der Konsole oder einer zusätzlichen Datei ausgeben.

## Techniken fürs Loggen

Es bleibt die Frage, wie die Attribute im Java-Code für die Log-Meldungen eingesammelt und ausgeloggt werden. In einem realen Projekt können die Log-Meldungen weit mehr als 20 Attribute besitzen. Niemand schreibt gern Log-Meldungen, bei denen er bei jedem Aufruf einer Log-Methode über 20 Parameter angeben muss. Fast alle Attribute müssen automatisch zu einer Log-Meldung hinzugefügt werden, sonst scheuen sich Entwickler, viele Log-Meldungen zu schreiben. Die Attribute einer Log-Meldung lassen sich in drei Kategorien aufteilen, die jeweils mit unterschiedlichen Techniken angegangen werden. Einige Attribute sind innerhalb der Laufzeit einer Applikation konstant. Andere Attribute sollen für alle Log-Meldungen innerhalb eines bestimmten Bereichs gelten. Ein Beispiel für einen Bereich

beziehungsweise Scope ist ein Web-Request. Schließlich gibt es Attribute, die nur für eine individuelle Log-Meldung gelten. Beispiele für konstante Attribute, die sich nach dem Start der Applikation nicht mehr ändern, sind

- der Name der zuständigen Organisationseinheit (Projektname, Abteilungsname etc.), falls der Log-Server von mehreren Projekten gemeinsam genutzt wird,
- der Name der Applikation,
- der Name des Microservice, falls eine Applikation aus mehreren Microservices besteht,
- die Software-Version,
- der Typ der Laufzeitumgebung (Entwicklung, Integration, Test, Produktion etc.) und
- die Bezeichnung der Laufzeitumgebung, auf der die Applikation läuft. Das kann der Hostname, der Name eines Clusters, Pods, Stacks oder ähnliches sein. Es kann auch zwischen den Namen der virtuellen und physischen Laufzeitumgebungen unterschieden werden.

Diese Attribute werden einmalig in der Log-Konfigurationsdatei gesetzt. Wie dies genau funktioniert hängt von der verwendeten Log-Bibliothek und vom verwendeten Formatter beziehungsweise Encoder ab. Ein einheitlicher Standard existiert leider nicht. Prinzipiell gibt es zwei Möglichkeiten die Werte in der Log-Konfiguration zu setzen. Beim Deployment kann die Konfigurationsdatei durch den Deployment-Mechanismus mit den richtigen Werten erzeugt werden. In einem Template der Log-Konfiguration werden durch Textersetzungen die gewünschten Werte gesetzt.

Die zweite Möglichkeit ist, Werte von Environment-Variablen in der Konfigurationsdatei beim Start der Anwendung auszuwerten. Dies funktioniert aber leider nicht mit allen Formattern beziehungsweise Encodern. In (Listing 1) ist ein Logback-Beispiel dargestellt.

(Listing 1)

```
<configuration>
  <appender name="jsonConsoleAppender"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.LogstashEncoder">
      <customFields>
        {
          "appName" : "logdemo",
          "appVersion": "${VERSION}"
        }
      </customFields>
    </encoder>
  </appender>
  <root level="info">
    <appender-ref ref="jsonConsoleAppender"/>
  </root>
</configuration>
```

**Logback Konfiguration mit JSON-Ausgabe und festen Attributen im Tag customFields. VERSION wird aus dem Environment gelesen. (Listing 1)**

Wenn die Attribute für alle Log-Meldungen innerhalb eines Scopes, wie zum Beispiel einem Web-Request oder einem Methodenaufrufs gelten sollen, ist der Mapped-Diagnostic-Context (MDC) das Mittel der Wahl. Er wird von sl4j, Logback und Log4j2 unterstützt. Der Mapped-Diagnostic-Context speichert eine Map mit Log-Attributen im aktuellen Thread. Die Log-Attribute werden dabei in allen Log-Meldungen ausgegeben, die innerhalb des aktuellen Threads erzeugt werden. Dies betrifft insbesondere auch Log-Meldungen, die in Frameworks beziehungsweise Libraries erzeugt werden. Wenn zum Beispiel in einem Framework eine **SQLException** geloggt wird, ist diese Log-Meldung mit allen im MDC vorhandenen Attributen versehen und kann so mit anderen Log-Meldungen korreliert werden. Gute Code-Stellen, um den MDC zu füllen, sind zum Beispiel **ServletFilter**, in denen sich Informationen aus einem Web-Request abgreifen lassen:

- Der Name des angemeldeten Benutzers
- Die aufgerufene URL
- Die IP-Adresse des Aufrufers
- Ein Hash der Session-ID
- Daten aus der Session
- Daten aus Cookies
- Andere Attribute aus dem Http-Header

Andere gute Stellen zum Füllen des MDC sind Implementierungen von fachlichen Services. Hier können fachliche Daten aus den Aufrufparametern in den MDC geschrieben werden, bevor die eigentliche Geschäftslogik ausgeführt wird. Auf jeden Fall sollten fachliche IDs im MDC abgelegt werden. Fachliche IDs sind zum Beispiel Kundennummern, Bestellnummern oder Vorgangsnummern. Kundensichtbare IDs sind besonders wertvoll, denn wenn es zu einer Störung kommt, werden Anwender diese IDs nennen. Wird das Loggen von fachlichen IDs konsequent umgesetzt, wird bei einer Suche nach einer fachlichen ID der Gesamtzusammenhang eines Vorgangs sichtbar. Es ist nicht nur die Benutzeraktion mit dem Fehler zu sehen, sondern auch alle anderen zum Vorgang gehörenden Aktionen, wie zum Beispiel Änderungen durch andere Benutzer oder durch Batch-Prozesse. Dieses mehr an Kontext hilft ungemein bei einer Fehleranalyse.

Attribute müssen wieder gelöscht werden, sobald sie nicht mehr im Scope sind. Andernfalls kann zum Beispiel ein Web-Request noch alte Attribute vom Vorgänger loggen. Um dies zu lösen, ist der MDC als Stack aufgebaut und implementiert das **Closeable** Interface, so dass **try-with** Klauseln zum Füllen des MDCs verwendet werden können. In (Listing 2) ist ein Beispiel aus einem **ServletFilter** zu sehen, der Attribute per sl4j-API in den MDC schreibt. Eine zugehörige Log-Meldung ist in (Abb.

2) zu sehen. Die Meldung wurde von Hibernate erzeugt und stammt nicht aus dem eigenen Applikations-Code.

(Listing 2)

```
try (var c1 = MDC.putCloseable("request_uri", req.getRequestURI());
    var c2 = MDC.putCloseable("actor", req.getUserPrincipal().getName());
    var c3 = MDC.putCloseable("method", req.getMethod());
    {
        chain.doFilter(req, response);
    })
```

Innerhalb eines ServletFilters wird der Mapped-Diagnostic-Context mit Attributen gefüllt.

| Key            | Value                                            |
|----------------|--------------------------------------------------|
| @timestamp     | Jan 17, 2020 @ 11:33:46.128                      |
| @version       | 1                                                |
| X-B3-SpanId    | cdea7784634489ca                                 |
| X-B3-TraceId   | cdea7784634489ca                                 |
| X-Span-Export  | false                                            |
| actor          | tom                                              |
| appName        | logdemo                                          |
| appVersion     | 3.0.0                                            |
| level          | WARN                                             |
| level_val      | 30,000                                           |
| logger_name    | org.hibernate.engine.jdbc.spi.SqlExceptionHelper |
| message        | SQL Error: 23505, SQLState: 23505                |
| method         | POST                                             |
| produktNummer  | 123                                              |
| request_uri    | /produkt/                                        |
| spanExportable | false                                            |
| spanId         | cdea7784634489ca                                 |
| tags           | _logz_http_bulk_json_0070                        |
| thread_name    | http-nio-8881-exec-2                             |
| traceId        | cdea7784634489ca                                 |

Kibana zeigt eine Logmeldung an. (Abb. 2)

Bei jeder individuellen Log-Meldung werden bereits von der Log-Bibliothek Attribute automatisch gesetzt. Dazu gehören der Zeitstempel, der Log-Level, der Meldungstext, der Name des Loggers und der Name des Threads. Von diesen Attributen können wir den Namen des Loggers, den Meldungstext und den Log-Level beeinflussen. Welcher Log-Level ist jetzt der jeweils richtige für eine Log-Meldung? Auf alle Fälle muss das einheitlich festgelegt werden. Hier ein Anwendungsbeispiel:

- **ERROR** ist für Fehler und Java-Exceptions. Hier besteht die Erwartung, dass die Ursache jeder Log-Meldung auf dem Level **ERROR** nachgegangen wird. Zusätzlich kann beim Level **ERROR** eine automatische Benachrichtigung an die Administratoren erfolgen.
- **WARN** ist eine als harmlos erkannte Fehlermeldung. Normalerweise können diese Meldungen ignoriert werden, es sei denn

sie treten in großer Zahl auf. Ein Beispiel hierfür ist ein fehlgeschlagener Login-Versuch. Dies tritt regelmäßig auf und ist kein Grund zur Sorge. Wenn sich die Anzahl von fehlgeschlagenen Login-Versuchen stark häuft, kann das ein Anzeichen für eine technische Störung im Authentisierungssystem sein, oder es handelt sich um einen Angriff. In beiden Fällen muss gehandelt werden. Über die Histogrammfunktionen der Log-Server-GUIs lassen sich Zunahmen von Log-Meldungen im Level **WARN** leicht erkennen.

- **INFO** ist eine allgemeine Meldung zum Nachvollziehen der Vorgänge in einer Anwendung. Dieser Level ist im Produktionssystem eingestellt. Die Erwartung ist, dass auf diesem Level alle Informationen geloggt werden die notwendig sind, um eine Fehlersituation im Produktionssystem eines Testsystems nachzustellen. Außerdem werden auf diesem Level Audit-Meldungen und alle Daten für statistische und fachlichen Auswertungen sowie das Monitoring geloggt.
- **DEBUG** ist auf dem Test- und Entwicklungssystem eingeschaltet. Der Level dient dazu, den Entwicklern Informationen zur Fehleranalyse bereitzustellen. Da dieser Level nicht in produktiven Systemen genutzt wird, können hier auch personenbezogene Daten und andere sensible Daten geloggt werden.
- **TRACE** wird von einigen Libraries und Frameworks genutzt. Der eigene Code benutzt diesen Level nicht.

Als erster Parameter einer Log-Meldung kann optional ein **Marker** übergeben werden. Der Umgang mit einem **Marker** kann schon in die Log-Konfigurationsdatei eingestellt werden. Log-Meldungen mit einem **Marker** können in eine andere Log-Datei als der Rest geschrieben werden. So können zum Beispiel Audit-Meldungen in eine eigene Audit-Log-Datei getrennt von den restlichen Log-Meldungen ausgeleitet werden. In (Listing 3) wird eine Audit-Meldung mit einem **Marker** geschrieben. (Listing 4) zeigt am Beispiel von Logback, wie diese Audit-Meldungen in eine eigene Log-Datei umgeleitet werden.

#### (Listing 3)

```
public final static Marker AUDIT = MarkerFactory.getMarker("AUDIT");
...
LOG.info(AUDIT, "Eine Audit Meldung für {}",nummer);
```

#### (Listing 4)

```
<appender name="AUDIT_FILE" class="ch.qos.logback.core.FileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="ch.qos.logback.classic.boolex.OnMarkerEvaluator">
```

```
    <marker>AUDIT</marker>
  </evaluator>
  <onMismatch>DENY</onMismatch>
  <onMatch>ACCEPT</onMatch>
</filter>
<file logs/audit.log</file>
<encoder class="net.logstash.logback.encoder.LogstashEncoder">
  <customFields>
    {
      "appName" : "logdemo",
      "appVersion": "${VERSION}"
    }
  </customFields>
</encoder>
</appender>
```

#### Logback-Konfiguration für das Ausleiten von Audit-Meldungen in die Log-Datei audit.log.

Key-Value-Paare können als Attribut direkt an eine Log-Meldung gehängt werden. Bei wichtigen Aktionen innerhalb der Anwendung sollte eine Event-ID als Attribut angegeben werden. Die ID sollte lesbar und eindeutig sein. Beispielsweise kann bei einem Zustandsübergang für den Schlüssel **eventId** der Name des Events als Wert gespeichert werden. Immer wenn das System eine fachliche Entscheidung trifft, kann ein Name für diese Entscheidung als Wert für **eventId** geloggt werden. Zahlenwerte (Mengen, Preise, Aufrufdauer etc.) lassen sich auch als Wert in einem Key-Value-Paar loggen. Die Verteilung der Zahlenwerte lässt sich mit dem Log-Server-GUIs statistisch auswerten.

Fehler-Codes sollten unter einem Schlüssel mit dem Wert des Codes als Attribut geloggt werden. Mit einem Fehler-Code können gleichartige Fehler gezielt gesucht und mit anderen Attributen gruppiert werden, um zum Beispiel die folgenden Fragen zu beantworten:

- Tritt der Fehler nur in Testsystemen auf oder auch im Produktionssystem?
- Mit welcher Softwareversion trat der Fehler erstmals auf (Abb. 1) ? Wurde der Fehler wirklich durch eine neue Version behoben?
- Welche Benutzer sind von dem Fehler betroffen und müssen eventuell kontaktiert werden?

Wird Logback mit dem Logstash-Encoder verwendet, können mit Hilfe der **kv** Methode sehr einfach Key-Value-Paare als Parameter geloggt werden. So führt die Code-Zeile

```
LOG.info("Mit Key Value Paar", kv("eventId","info.request"));
```

zu folgender Log-Meldung:

## (Listing 5)

```
{
  "@timestamp" : "2020-01-03T09:45:25.793+01:00",
  "@version" : "1",
  "message" : "Mit Key Value Paar",
  "logger_name" : "de.adesso.majchszak.loggingdemo.
  LoggingController",
  "thread_name" : "http-nio-8080-exec-1",
  "level" : "INFO",
  "level_value" : 20000,
  "eventId" : "info.request",
}
```

Bei Log4j2 funktioniert dieser Key-Value-Mechanismus leider nicht. Stattdessen bietet Log4j2 das **Message** Interface an. Um Key-Value-Paare über **Message** Objekte zu loggen, ist allerdings eine eigene Implementierung des **Message** Interfaces und eine eigene Implementierung eines **Appenders** notwendig.

### Noch mehr IDs zum Zusammenhalten von Log-Meldungen

In einer Microservice-Architektur wird eine Benutzerinteraktion nicht nur von einem Service allein abgearbeitet. Um die Log-Meldungen einer einzelnen Aktion auf mehreren Services verfolgen zu können, werden sogenannte Correlation-IDs benötigt, die sich automatisch bei REST, JMS oder anderen Service-Aufrufen weiter übertragen. Dies funktioniert wie folgt:

- Trifft ein Request ohne Correlation-ID ein, wird eine ID erzeugt und in den MDC geschrieben.
- Beim Aufruf eines anderen Microservice wird die Correlation-ID in die Metadaten des Aufrufs geschrieben. Bei REST-Aufrufen wird die ID in den HTTP-Header geschrieben.
- Eine in den Meta-Daten eines eingehenden Request gefundene Correlation-ID wird in den MDC geschrieben.

Im Spring-Umfeld erledigt das die Bibliothek **Sleuth** automatisch. In (Abb. 2) hat **Sleuth** die Log-Meldung um eine Correlation-ID im Attribut **traceId** angereichert.

### Es darf nicht alles geloggt werden

Sicherheitsrelevante Daten dürfen auf keinen Fall in Logs geschrieben werden. Dazu gehören unter anderem Passwörter und Kreditkarteninformationen. In vielen Logs ist häufig die Session-ID der HTTP-Session zu sehen. Bei der Verwendung eines Log-Servers darf diese ID nicht geloggt werden, da mit Kenntnis der Session-ID innerhalb des Gültigkeitszeitraums die Session eines Benutzers übernommen werden kann. Ein Log-Server macht die Session-ID viel zu schnell sichtbar. Da der konkrete Wert der Session-ID für die Log-Analyse nicht relevant ist, kann stattdessen ein Hash-Wert der Session-ID geloggt werden. Personenbezogene Daten werden am besten über

eigene Log-Kommandos in abgetrennten Log-Dateien geloggt. Marker sind dazu gut geeignet. Log-Daten können im Log-Server in unterschiedlich abgesicherte Indizes gespeichert werden. Der Personenkreis, der Zugriff auf personenbezogene Daten haben soll, kann so stark eingeschränkt werden, während der Zugriff auf den Rest der Log-Daten ausgeweitet werden kann. Durch mehrere Indizes können auch unterschiedliche Löschfristen für die Log-Daten realisiert werden. Bei personenbezogenen Daten sind eher kurze maximale Aufbewahrungsfristen gefordert, während bei Auditdaten eher lange Mindestaufbewahrungsfristen wünschenswert sind. Die Aufteilung der Log-Daten auf verschiedene Indizes mit unterschiedlichen Löschfristen löst dieses Problem elegant.

### Lebt die Applikation noch?

Ein gut strukturiertes Logging erlaubt das Monitoring einer Anwendung. Klassischerweise werden Fehlermeldungen im Log überwacht. Auf diese Weise lässt sich feststellen, dass irgend-etwas nicht funktioniert. Viel wichtiger ist es aber festzustellen, dass eine Anwendung erfolgreich arbeitet. In komplexeren Systemen kommt es hin und wieder zu Fehlersituationen, die nicht zu Fehlermeldungen führen. Beispiele dafür sind:

- Benutzer führen einen Prozessschritt in einem Fremdsystem durch und werden dort durch Fehler blockiert, die nicht im eigenen Log auftauchen. Im eigenen System ist nur zu sehen, dass nichts passiert, weil die Benutzer nicht zurückkehren. Wird ein Abgangs- und ein Rückkehr-Event geloggt, zeigt eine stark abfallende Rate der Rückkehr-Events das Problem an.
- Netzwerkprobleme mit Teilausfällen vor der eigentlichen Anwendung führen zu keinen Fehlern in den Anwendungs-Logs. Allerdings bricht die Rate der Events im Log sichtbar ein.

### Tests auch für das Logging

Auch Log-Meldungen einer Applikation sollten Tests unterzogen werden. Damit ist nicht gemeint, dass in Unit-Tests das Loggen jeder einzelnen Meldung überprüft wird, sondern dass Log-Meldungen auf Vollständigkeit und Verständlichkeit geprüft werden. Dazu werden die Use-Cases der Anwendung durchgeführt und anschließend wird geprüft, ob sich die durchgeführten Aktionen nachvollziehen lassen. Tritt bei einem Test ein Fehler in der Anwendung auf, sollte immer auch überprüft werden, ob der Fehler in den Log-Meldungen nachvollziehbar ist und ob zur Diagnose des Fehlers Daten fehlen. Wenn erst im produktiven Betrieb festgestellt wird, dass im Log Daten fehlen, ist es zu spät.

#JAVAPRO #Database #JUnit

# H2 für JUnit-Tests

H2 ist eine komplett in Java implementierte, relationale Open-Source-Datenbank, die man embedded, als Server- oder In-Memory-Datenbank verwenden kann.

H2 eignet sich unter anderem in Prototypen, kleinen Services und für automatisierte Unit-Tests.

**M**oderne Ansätze zum Systemdesign wie Microservices oder Self-Contained-Systems bieten die Möglichkeit, einzelne Aspekte eines Gesamtsystems isoliert betrachten und bearbeiten zu können. Implementierung, Erweiterung und Deployment der einzelnen Komponenten sind durch automatisierte Tests schnell und trotz zahlreicher Build- und Deployment-Pipelines sicher. Die Vorteile der Isolation kommen vor allem in der Entwicklung zum Tragen, wenn es gelingt Services ohne weitere Umgebungskonfiguration auf der Entwicklermaschine aufzusetzen und testgetrieben entwickeln zu können. Für Services und Systeme, die eine relationale Datenbank als Datenquelle nutzen, gibt es für lokal und in Pipelines ausgeführte Unit-Tests unterschiedliche Ansätze zur Einbindung einer Datenquelle. In diesem Zusammenhang wird im Folgenden das relationale Datenbankmanagementsystem H2 vorgestellt, das insbesondere für Tests auch als In-Memory Lösung konfigurierbar ist.

## Was ist H2?

H2 ist eine Open-Source-SQL-Datenbank, die komplett in Java implementiert ist. Sie lässt sich in drei Modi verwenden: Embedded, Server oder In-Memory. Im Embedded-Modus läuft die Datenbank in der gleichen JVM wie die Applikation und ist auch nur innerhalb dieser Virtual-Machine erreichbar. Dagegen kann die Datenbank im Servermodus auch von anderen JVMs aus angesprochen werden. Der In-Memory-Modus ist mit dem Embedded-Modus vergleichbar, jedoch mit dem wesentlichen Unterschied, dass der Zustand der Datenbank nicht über den Neustart der JVM persistiert wird. Die H2-Datenbank unterstützt Transaktionen und Basis-Konstrukte wie Triggers, Views und Check-Constraints. Darüber hinaus können bei der Verbindung Kompatibilitätsmodi für gängige SQL-Datenbanken konfiguriert werden, u.a. MySQL, Microsoft SQL-Server, PostgreSQL etc. Wird die Datenbank nicht im In-Memory-Modus betrieben, persistiert sie die Daten in Files, welche mittels AES-128 verschlüsselt werden können. H2 unterstützt JDBC, daher ist die Anbindung mit bewährten Standards wie JPA problemlos möglich. Die Projektwebsite<sup>1</sup> bietet einen umfassenden Überblick über weitere Features sowie eine kurze tabellarische Übersicht mit einem Vergleich anderer Datenbanken.

## Warum H2?

Als komplett in Java geschriebene Lösung lässt sich H2 per Dependency im entsprechenden Build-Tool konfigurieren und wird nach dem Klonen des Projekts auf der Entwicklermaschine automatisch beim ersten Bauen angezogen. Dadurch entfällt die Notwendigkeit, auf der Maschine eine manuelle Installation

### Autor:



Julius Mischok ist Geschäftsführer der Mischok GmbH in Augsburg. Seine Kernaufgaben sind Prozessentwicklung, sowie Coaching und Schulung der Entwicklungsteams. Aktuell fokussiert sich seine Arbeit auf die Frage, wie Software schnell und mit einer maximalen Wertschöpfung produziert werden kann. Er hat Mathematik studiert und entwickelt seit fast zwei Jahrzehnten Java.

E-Mail: [julius.mischok@mischok.de](mailto:julius.mischok@mischok.de)

Webseite: [www.mischok.de](http://www.mischok.de)

Xing: [https://www.xing.com/profile/Julius\\_Mischok](https://www.xing.com/profile/Julius_Mischok)

des Datenbanksystems vorzunehmen oder zumindest einen Container mit einer Installation hochzufahren. Insbesondere der In-Memory-Modus ermöglicht es, bei jedem Testlauf bzw. JVM-Start mit einem definierten Datenbankzustand zu starten. Dies dient vor allem der Reproduzierbarkeit automatisierter Tests. Durch die reine Integration als Dependency fügt sich H2 nahtlos in CI-/CD-Pipelines ein und es ist keine angepasste Konfiguration notwendig. Der schmale Footprint wirkt sich hier äußerst positiv auf Laufzeit und Ressourcenverbrauch aus.

## Konfiguration für den In-Memory Modus

Auf GitHub<sup>2</sup> ist ein Projekt veröffentlicht, das ein komplettes Beispiel für die Konfiguration von H2 in einer Spring-Boot-Applikation bietet. Der Service stellt ein relativ einfaches REST-Interface zur Verwaltung von Tasks zur Verfügung. Für den Produktivbetrieb ist eine PostgreSQL-Datenbank vorgesehen und auch bereits unter **src/main/resources/application.properties** konfiguriert (Listing 1). Im Repository befindet sich eine Docker-Compose-Konfiguration, die einen Container mit einer Datenbankinstanz hochfährt. Diese wird verwendet, um die Applikation lokal laufen zu lassen.

(Listing 1)

```
spring.datasource.username=local
spring.datasource.password=local
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.driverClassName=org.postgresql.Driver
```

Um die Unit-Tests nicht mehr gegen die PostgreSQL-Datenbank laufen zu lassen, sind zwei Schritte notwendig: Zunächst muss die Dependency in der Maven-Konfiguration eingebunden werden (Listing 2). Analog lässt sich auch Gradle konfigurieren.

(Listing 2)

```
...
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>test</scope>
</dependency>
...
```

Im zweiten Schritt wird eine bereits bestehende **src/test/resources/application.properties** erweitert oder die Datei angelegt. Im Wesentlichen müssen hier mindestens vier Parameter überschrieben werden: Die URL der H2-Datenbank, Benutzername und Passwort sowie der zu verwendende Treiber (Listing 3). Dieser Treiber wird durch das Hinzufügen der Dependency im Build-File bereits angezogen. Der Einschub **mem** in der Datenbank-URL führt dazu, dass die Datenbank im In-Memory-Modus betrieben wird. Mit diesem Setup lassen sich die Unit-Tests im Optimalfall bereits gegen die H2-Datenbank ausführen.

(Listing 3)

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
```

## Einbindung in Bestandscode

Gerade bei der Übernahme eines Bestandsprojekts kann die Notwendigkeit auftreten, das Projekt überhaupt erst lokal testbar zu machen. Die oben gezeigte Konfiguration ist hier ein eleganter Weg, dieses Ziel ohne umfangreiche Konfiguration außerhalb des Projekts zu erreichen. Je nach Zustand des Bestandscodes können jedoch bei der Umsetzung diverse Stolpersteine auftreten. Wurden beispielsweise bis dato keine Datenbankmigrationsskripte zur Schemagenerierung verwendet, müssen Tabellen und weitere Strukturen für die Testläufe auf anderem Wege angelegt werden. Als einfachster Ansatz ist zu prüfen, ob die Tabellenstruktur anhand von JPA-Entities angelegt werden kann. Hierzu kann der Konfigurationsparameter **spring.jpa.hibernate.ddl-auto=create-drop** in die Testkonfiguration aufgenommen werden. Sollte dieses Vorgehen keinen Erfolg bringen, besteht die Möglichkeit ein Datenbankmigrations-Tool wie zum Beispiel Flyway<sup>3</sup> für die Testläufe einzubinden. Hierfür wird dann ein einzelnes Migrationsskript angelegt, das die entsprechende Tabellenstruktur in H2 erstellt.

Ist für den Produktivbetrieb bereits ein Datenbank-Migrations-Tool konfiguriert, kann es zu dem Umstand kommen, dass einzelne Migrations-Skripte inkompatibel zu H2 sind. Wurden beispielsweise bisher herstellerepezifischer Dialekt oder Strukturen in den Skripten verwendet, kann der Kompatibilitätsmodus von H2 unter Umständen Abhilfe schaffen. In Fällen, in denen auch dieser Ansatz versagt, bleibt als letzte Möglichkeit einzelne Skripte in den Test-Resources zu überschreiben, diese also in Standard-SQL zu übersetzen. Meistens lässt sich durch diese Schritte ein ausreichendes Setup herstellen, um sinnvolle Unit-Tests erstellen zu können. Diese können lokal und in Pipelines ausgeführt werden und sind schnell ein wertvoller Baustein zur Qualitätssicherung. Dies ist in der Gesamtbetrachtung in jedem Fall mit den für die Konfiguration anfallenden Aufwänden in Relation zu setzen.

## Kritische Betrachtung

Nachdem nun die Fragen rund um Konfiguration und eventuelle Stolpersteine, kurz die technischen Möglichkeiten, geklärt sind, ist noch eine Betrachtung aus Sicht der nachhaltigen Softwarearchitektur notwendig. Insbesondere muss die Frage beantwortet werden, ob es vertretbar ist, für automatisierte Tests und in der Produktion unterschiedliche Datenbanksysteme zu verwenden. Pauschal ist dies sicherlich zu verneinen. Agile Methoden in der Softwareentwicklung zielen unter anderem darauf ab, die Zeit

zu minimieren, in der Fehler auftreten. Ein Setup unterschiedlicher Datenbanksysteme kann dazu führen, dass ein spezifischer Effekt erst nach der Integration und nicht bereits beim lokalen Testlauf auftritt. In der Gesamtbetrachtung wird demgegenüber die Zeitersparnis durch die wegfallende, lokale Konfiguration und in den Pipelines gestellt. Im Kontext einer Microservice-Architektur kann die Entscheidung außerdem durchaus positiven Einfluss auf das Softwaredesign haben: Die parallele Verwendung zweier Datenbanksysteme zwingt zu der Verwendung von Standards bei der Entwicklung, beispielsweise die Verwendung von SQL in Migrations-Skripten. Dies führt langfristig zu einer schwachen Kopplung des Microservices an das verwendete Datenbanksystem und erleichtert einen Umzug auf ein anderes System im Produktivbetrieb, sofern dafür in Zukunft die Notwendigkeit besteht. Zudem steigt die Fokussierung des Services auf die nach außen zur Verfügung gestellte Schnittstelle, wenn die Funktion von den technischen Möglichkeiten der dahinterliegenden Datenbank entkoppelt ist. Als Faustregel lässt sich folgendes formulieren: Je kleiner das Projekt und je mehr die Struktur der Daten anstelle von Abfrageperformance im Vordergrund stehen, desto eher bietet der Einsatz unterschiedlicher Datenbanksysteme für Produktivbetrieb und automatisierte Tests mehr Vor- als Nachteile. Letztlich muss für jede Applikation eine individuelle Antwort gefunden und begründet werden.

## Die Grenzen von H2

Mit dem Aufkommen von NoSQL-Datenbanken hat sich der Markt für Datenbanksysteme wesentlich gewandelt. Relationale Datenbanksysteme punkten immer noch bei der Konsistenz, Transaktionssicherheit und Abfrageperformance in Anwendungsfällen, in denen viel Optimierungspotential in der Datenbank selbst ausgeschöpft werden kann. Wenn die Performance eine fachliche Anforderung an die Applikation ist, macht es Sinn, die Möglichkeiten des dahinterliegenden Systems vollends auszureizen. Hier ist vom Einsatz von H2 als Datenbank für Unit-Tests abzuraten, da andernfalls für einen Teil der Anforderungen keine testgetriebene Entwicklung möglich ist. Gleiches gilt, wenn durch fachliche Anforderungen die Verwendung herstellerspezifischer Strukturen des verwendeten Datenbanksystems notwendig ist.

Kommt der Einsatz von H2 aus oben genannten Gründen nicht in Frage, bieten Testcontainer die erste Alternative. Mit dieser Technologie lassen sich ohne weitere externe Konfigurationen, Container mit einer spezifischen Datenbank wie beispielsweise PostgreSQL aus dem Java-Code heraus erstellen und als Datenquelle einbinden. Im Vergleich zur Konfiguration von H2 muss jedoch etwas mehr Aufwand betrieben werden. Insbesondere werden Java-Klassen erstellt, um den Testcontainer entsprechend aufzusetzen. Dies führt, wenn auch in überschaubarem Maße, zu höherer Komplexität im Projekt und einer größeren Code-Base.

Als letzte Lösung kann auch ein lokal zu startender Docker-Container dienen. Hierfür lässt sich ein Docker-File oder eine

Docker-Compose-Konfiguration in das Repository einchecken. Die Konfiguration in diesem Fall bleibt ähnlich schlank, wie bei der Verwendung von H2. Jedoch werden auf der Entwicklungsmaschine zusätzlich installierte Komponenten vorausgesetzt und die Datenbank wird nicht ohne weiteres für jeden Testlauf auf einen leeren Zustand zurückgesetzt. Zudem erfordert dieser Ansatz auf jeder Entwicklungsmaschine und in jeder Pipeline zusätzliche Konfigurationsschritte.

## Weitere Einsatzmöglichkeiten

Neben dem ausführlich vorgestellten Einsatz von H2 für automatisierte Tests gibt es noch weitere Szenarien, in denen sich die Verwendung anbietet. Für das Prototyping kann H2 seinen Vorteil des geringen Footprints voll ausspielen. In der ersten Phase der Entwicklung eines Produktprototyps wird hierbei auf die Konfiguration einer externen Datenbanklösung verzichtet. Diese kleine Ersparnis an Zeit und Komplexität multipliziert sich über die Anzahl der lokalen Entwicklungsumgebungen und die Deployment-Stages und ist mitunter nicht unerheblich. Durch die Verwendung von standardkonformem SQL und Abstraktions-Layern wie JPA ist beim Übergang eines Prototypen zur tatsächlichen Produktimplementierung einfach der Wechsel auf ein anderes Datenbanksystem möglich.

Andere sinnvolle Einsatzfelder sind Szenarien, in denen es auf eine schlanke Installation des gesamten Systems ankommt. Kleine Services, die im Wesentlichen von wenigen Nutzern verwendet werden, sind hierfür ein Beispiel. Durch den Wegfall einer separaten Datenbankinstallation mit den entsprechenden Betriebsaufgaben wie Updates, Backups etc. entfallen über die Lebensdauer der Applikation nicht unerhebliche Aufwände. H2 bringt sogar ein kleines Webinterface mit, das im Produktivbetrieb in einer Spring-Boot-Applikation per `spring.h2.console.enabled=true` aktiviert werden kann.

## Fazit:

Das relationale Datenbankmanagementsystem H2 bietet sich durch seine schlanke Konfiguration und den schmalen Footprint zur Verwendung in Prototypen oder als Datenbank für Unit-Tests an. Das System lässt sich auch in Bestandsprojekten nachrüsten, in denen zuvor auf automatisierte Tests verzichtet wurde. So wird in der entsprechenden Applikation nachhaltig die Qualität gesteigert. Durch die Verwendung von Standard SQL und Abstraktionsschichten in der Anbindung führt die Verwendung zu loser Kopplung der Applikation an das verwendete Datenbanksystem und zu einem insgesamt verbesserten Applikationsdesign.

## Quellen:

- 1 H2-Webseite: <https://bit.ly/h2database>
- 2 Github-Projekt: <https://bit.ly/project-g2>
- 3 Flywaydb: <https://bit.ly/flywaydb>

#JAVAPRO #Digitalisierung #Projektmanagement

# Von Nixdorf nach Kubernetien – Digitalisierung im Wandel der Zeit

Die Politik entdeckt die Digitalisierung als Zukunftsthema und gibt den Rat: „Die Digitalisierung ist auch für Mittelständler eine Pflichtaufgabe“<sup>1</sup>. Dabei ist die Digitalisierung im Mittelstand schon lange angekommen – lange bevor das Internet zum Neuland in der Politik deklariert wurde.

## Digitalisierung 1.0

Mit der mittleren Datentechnik, die in den 70er Jahren des vorherigen Jahrhunderts vor allem durch Nixdorf in Deutschland und Europa geprägt wurde, wurden Computer auch für mittelständische Unternehmen erschwinglich. So hat das Optica Abrechnungszentrum bereits früh auf Unterstützung durch die elektronische Datenverarbeitung (EDV), wie IT früher hieß, gesetzt. Damit sich die teure Investition rechnete, waren die Prozesse auf den Computer ausgerichtet, um Leerlauf zu vermeiden. In mehreren großen Schreibbüros mit 40 bis 50 vorwiegend weiblichen Mitarbeitern wurden Abrechnungsdaten an Datensammlersystemen erfasst und gespeichert, anfangs auf Lochkarten, später auf Magnetbändern. Zum Abrechnungslauf wurden die Bänder auf ein Nixdorf 8850 eingespielt und anschließend archiviert. Benutzerschnittstellen waren in der EDV-Bronzezeit kein Thema – der Benutzer hatte sich nach dem Rechner zu richten. Vorherrschend waren Batch-Betrieb und das EVA-Prinzip:

## Autor:

Oliver Böhm ist derzeit beim Optica Abrechnungszentrum für Fragen rund um die Java-Entwicklung verantwortlich — nicht nur für aktuelle Themen, sondern auch für Legacy-Systeme. Passend dazu hat er im letzten Jahrhundert Informatik an der Uni Stuttgart studiert. Seit 1999 entwickelt er mit Java. Neben seiner hauptberuflichen Tätigkeit als Java-Archäologe und -Architekt ist er Board-Mitglied der JUG Stuttgart und organisiert dort neben den Stuttgarter Testtagen Code Retreats zusammen mit der Softwarekammer Stuttgart.



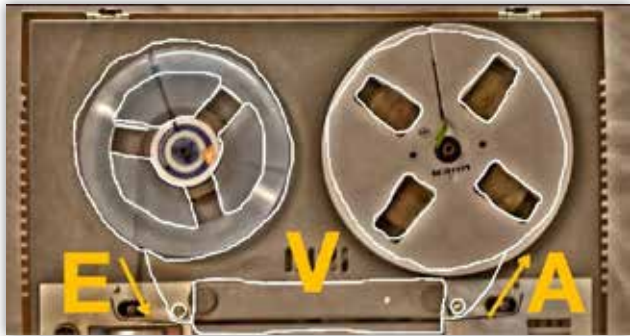
Twitter: @oboehm, @JitStorm

E-Mail: o.boehm@optica.de

Xing: [https://www.xing.com/profile/Oliver\\_Boehm2/](https://www.xing.com/profile/Oliver_Boehm2/)

Blog: <https://oli.blogger.de/>

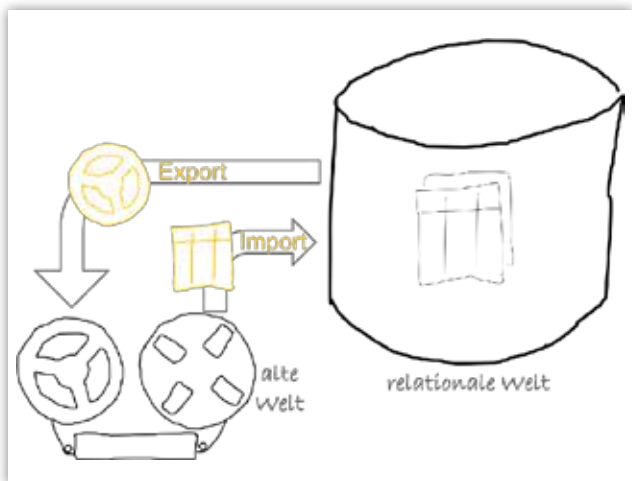
Eingabe – Verarbeitung – Ausgabe (**Abb. 1**). Hierbei gab es einfache, meist grüne Masken, die über die Tastatur und Enter-Taste bedient wurden. Und gute Datentypisten/-innen konnten die Masken blind und in hoher Geschwindigkeit bedienen.



Das EVA-Prinzip (**Abb. 1**)

## Digitalisierung 2.0

In den 80er Jahren kamen dann relationale Datenbanken auf. Diese dienten nicht nur der standardisierten Ablage von Daten, sondern erlaubten es auch, Beziehungen (mathematisch: Relationen) auszudrücken. Als 1985 das Gigabyte RAM noch etwa 100.000 DM kostete, stand vor allem die effiziente Speicherung von Daten im Vordergrund. Gleichzeitig fand mit dem Aufkommen des Personal-Computers eine Dezentralisierung statt. Im Mittelpunkt stand eine relationale Datenbank, während die Eingabe und (Vor-)Verarbeitung bereits auf PCs stattfand. Zusammen mit der aufkommenden Vernetzung war dies die Geburtsstunde vieler Client-Server-Architekturen, die teilweise auch heute noch im Einsatz sind. Und auch hier gab es bereits die ersten digitalen Transformationen. Es galt die Records und Datenbänder aus der Nixdorf Ära in die neue, relationale Welt zu übersetzen – und umgekehrt (**Abb. 2**). Schließlich liefen auf dem alten System lebenswichtige Prozesse, mit denen Geld erwirtschaftet wurde. Und diese Prozesse wollen mit Daten bzw. Magnetbändern versorgt werden.

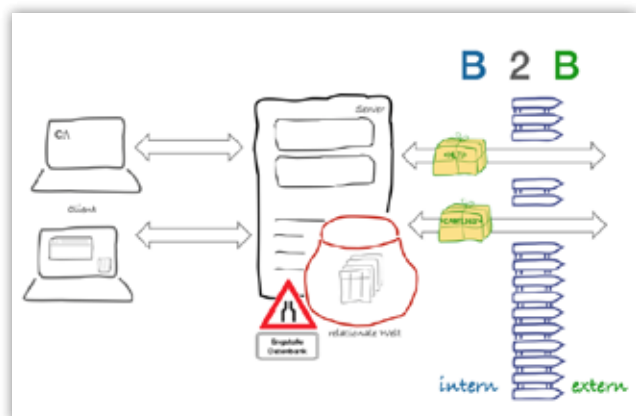


Datenaustausch zwischen alter und neuer Welt (**Abb. 2**)

In einigen Bereichen wie Banken und Versicherungen sind diese Transformationen auch heute noch im Einsatz. Wer schon einmal mit einer COBOL-Copy-Strecke zu kämpfen hatte, weiß wovon die Rede ist.

## World-Wide-Web

Waren es anfangs noch lokale Netze, die den Datenaustausch durch Disketten oder andere Datenträger ablösten, wurde vor allem nach der Geburtsstunde des World-Wide-Webs 1989 am CERN der elektronische Austausch mit externen Partnern immer wichtiger. War es bis dahin noch üblich, Bestellungen, Rechnungen und andere Unterlagen in Papierform auszutauschen, entwickelte sich ein wilder B2B- und B2C-Schnittstellen-Zoo mit dem Ziel, diesen Medienbruch zu reduzieren (**Abb. 3**). Mit dem Austausch von Daten ist immer auch die Frage des Formats von Bedeutung. Waren es anfangs meist binäre Protokolle, wurde ab 1998 immer häufiger XML für die Definition von Austauschformaten eingesetzt. XML spielt auch für die Idee von Service-Oriented-Architecture (SOA) eine zentrale Rolle, die ihre Hypephase Anfang der 2000er Jahre hatte.



Datenaustausch mit Business-Partnern (**Abb. 3**)

Auch wenn um 2000 die ersten NoSQL-Datenbanken aufkamen, änderte sich datenbanktechnisch nicht viel: im Mittelpunkt vieler Anwendungen und Architekturen steht nach wie vor eine relationale Datenbank, die aber langsam zum Flaschenhals zu werden droht. Nun kann man zwar Datenbanken clustern, aber dies bedeutet nicht zwangsläufig einen Performancegewinn, sondern hängt stark davon ab, ob sich die Daten anhand der Zugriffe aufteilen lassen. Dies kann beim Datenbankdesign berücksichtigt werden, aber die Realität sieht oft so aus, dass neue Anforderungen meist mit einer Anreicherung von Datenbanktabellen um neue Spalten einhergehen. Und auch wenn es sinnvoller wäre eine Tabelle aufzuteilen, so scheitert das oft daran, dass sich um die Datenbank eine ganze Menge von Anwendungen gruppiert haben, die eine grundlegende Änderung des Datenbankdesigns unmöglich machen. Was also tun, wenn sich die Datenbank nicht clustern lässt? Und wie lassen sich neue Anforderungen schneller umsetzen als mit der aktuellen Architektur?

## Microservices

Ein Resultat dieser Überlegungen war die Aufteilung der Datenbank und der Anwendungslandschaft nach Domain-Driven-Design in verschiedene Bereiche (Stammdaten, Rechnungen, Rezepte etc.). Darin sollte jede Domain ihr eigenes Datenbankschema bekommen. Architektonisch sollte der Zugriff dann nur über Microservices erfolgen. Damit wird die Datenbank zum Implementierungsdetail und kann später auch durch eine NoSQL-Datenbank ausgetauscht werden (Abb. 4).



**Aufteilung und Kapselung der Datenbank über Microservices (Abb. 4)**

Warum Microservices? Bei Microservices geht es immer um die Performance der Softwareentwickler, nicht um die der Software<sup>2</sup>. Natürlich gibt es auch andere Vorteile wie bessere Skalierbarkeit und Unabhängigkeit von der Programmiersprache, aber man erkaufte sich diese Vorteile durch eine höhere Komplexität der Anwendungslandschaft. Und Microservices allein lösen nicht das Problem der Aufteilung unserer zentralen Datenbank. Prinzipiell gibt es dafür zwei Strategien:

1. Beibehaltung der zentralen Datenbank und Kapselung durch Microservices.
2. Microservices mit eigener Datenbank + Synchronisierung mit zentraler Datenbank.

Strategie 1 bringt die Gefahr, dass viele Anwendungen die Kapselung umgehen können und neue Anforderungen zu einem weiteren Aufblähen der Datenbank führen. Langfristig hat daher die 2. Strategie das größere Potential. Allerdings birgt es auch größere Risiken: man muss die zentrale Datenbank mit ihren Redundanzen und potenziellen Inkonsistenzen mit den internen Datenbanken der einzelnen Services synchronisieren. Das ist ein nicht ganz triviales Problem. Technisch gelöst wird das über Events, die bei Datenänderungen erzeugt werden (zum Beispiel per Datenbank-Trigger oder über den Microservices selbst). Ob man jetzt hier RabbitMQ, Kafka oder andere Lösungen einsetzt, hängt von verschiedenen Randbedingungen ab. Das Prinzip ist aber das gleiche – Events werden zusammen mit den Daten in Messages

verpackt und an die Abonnenten verschickt, die so über Datenänderungen informiert werden (Abb. 5).



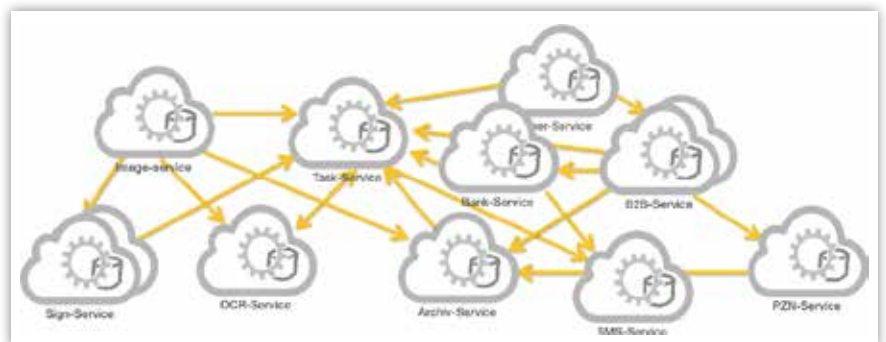
**Rabbit MQ (Abb. 5)**

## Wolkige Aussichten

Gab es früher nun einen oder einige wenige physikalische Server im Rechenzentrum, so sind es heute viele virtuelle Server. Damit stellt sich die Frage, auf welche Server verteilen wir am besten unsere Microservices. Und wie skalieren wir, wenn wir Lastspitzen abfangen wollen oder müssen.

Eine weitere Frage ist die Frage des Testens. Microservices kommen selten allein, sondern sind oft von anderen Seiten abhängig. Wie kann ich das testen, wenn dann auch noch weitere Abhängigkeiten wie eine Datenbank mit Testdaten und ein Elastic-Search-Server dazukommen?

Früher oder später stößt man bei der Suche nach der Antwort auf Docker, einer leichtgewichtigen Virtualisierungslösung mit vielen vorgefertigten Containern. Es gibt Container für Java, für Datenbanken, für Webserver, kurzum – für alles Mögliche. Darauf aufbauend kann man eigene Container bauen und sich damit sein eigenes kleines virtuelles Rechenzentrum auf seinem Rechner aufbauen – ideal für den kleinen Integrationstest zwischendurch oder auch einfach nur zum Ausprobieren, ob für ein bestimmtes Problem vielleicht eine NoSQL-Datenbank die bessere Alternative wäre. Endlich ist man auch nicht mehr vom Admin abhängig, wenn bestimmte Dienste oder Rechenkapazitäten gebraucht werden. Man sucht sich einfach den passenden Container aus und packt ihn in die Cloud. Und zum Deployment in Produktion ist es dann nur noch ein kleiner Schritt, oder? (Abb. 6).



**Microservice-Landschaft (Abb. 6)**

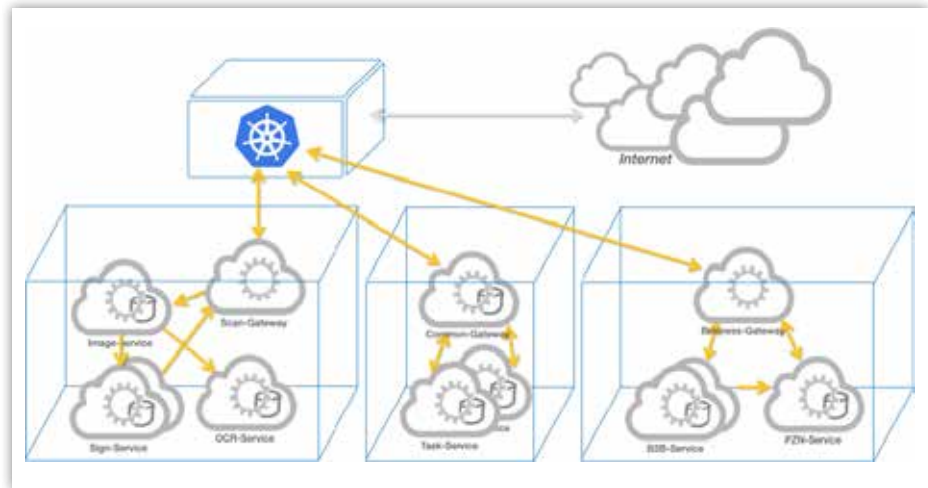
Stop! Spätestens ab hier wird es jetzt kritisch. In Produktion ist das Gewässer rauer und hier herrschen andere Gesetze. Auch wir mussten erst lernen, den Admin frühzeitig mit einzubinden und ein DevOps-Team aufzubauen, das für unsere Basis-Container verantwortlich ist und sie für die Gefahren auf rauer See härten. Es sorgt dafür, dass man im Falle von Sturmschäden rechtzeitig vom Ausfall einzelner Systeme benachrichtigt wird und das Fehleranalysen betrieben werden können. Aus Sicherheitsüberlegungen werden bei uns die Rechnerkapazitäten intern bereitgestellt. Wer das nicht kann, für den ist die Frage nach einem vertrauensvollen Cloud-Provider überlebenswichtig. Neben der Frage der Datensicherheit ist vor allem die Frage zu klären, wie man im Fehlerfall an wichtige Daten und Protokolle herankommt, um eine Analyse erfolgsversprechend durchführen zu können.

## Die Reise nach Kubernetes

Mit der steigenden Anzahl von Containern wird die effektive Zuteilung der Ressourcen immer wichtiger. Welcher Container bekommt wieviel Speicher oder CPUs? Welchen Container starte ich lieber mehrfach? Wie skaliere ich und wie überwinde ich das Ganze? Für die Antwort auf diese Fragen setzen wir auf Kubernetes, um unsere Container-Landschaft zu orchestrieren. Der Orchestrierung sollte aber immer eine gute Planung vorausgehen. Welche Microservices gehören zusammen und sollten am besten in ein eigenes Self-Contained-System (SCS) untergebracht werden? Welche gemeinsamen Dienste müssen bereitgestellt werden? Wer darf mit wem kommunizieren? Wie wird Autorisierung und Authentisierung gehandhabt? Wie soll Load-Balancing gesteuert werden? Wo soll das ganze laufen – in einer internen, externen oder hybriden Cloud? Für die Beantwortung dieser Fragen braucht es neben den verantwortlichen Architekten ein DevOps-Team mit entsprechender Kubernetes-Kompetenz. Und damit schließt sich der Kreis zu den Anfängen. Auch im Nixdorf- und Mainframe-Zeitalter war die zentrale Administration Voraussetzung für einen effektiven und störungsfreien Betrieb. Eine schöne Oberfläche allein reicht nicht, um auf Dauer erfolgreich zu sein (Abb. 7).

## Fazit:

Im Gegensatz zur Politik ist die Digitalisierung im Mittelstand schon längst angekommen. Musste man sich zu Beginn der Digitalisierung noch fragen, ob sich die teure Hardware lohnt und wie man sie am besten auslastet, setzt man heute auf Virtualisierung und Verlagerung der Ressourcen in die Cloud, um die Kosten



Orchestrierung über Kubernetes (Abb. 7)

in den Griff zu bekommen – unerheblich, ob die Cloud bei Amazon, Azure oder selbst betrieben wird. Eine selbstbetriebene Cloud muss dabei nicht unbedingt teuer sein, wenn man auf unendliche Skalierung verzichten kann. Und sie bietet Vorteile bei der Sicherheit:

- Daten können nicht abgegriffen werden,
- Daten können nicht verloren gehen,
- beim Ausfall der Cloud ist man nicht von anderen abhängig (auch Public-Clouds können ausfallen<sup>3 4 5</sup>).

Auch wenn der Datenschutz oft als hinderlich angesehen wird, so kann er auch ein Wettbewerbsvorteil sein. Gerade im Gesundheitsbereich, wo man mit hochsensiblen Daten umgehen muss, ist Datenschutz unabdingbar und es gilt das Risiko von Datenlecks zu minimieren, auch in Zeiten von Corona und Home-Office. Neben dem schleppenden Breitbandausbau seit mehr als 10 Jahren ist die Rekrutierung von IT-Fachkräften eines der größeren Probleme bei der weiteren Entwicklung. Auch wenn sich Studienabgänger oft von großen Firmen und Globalisierung verleiten lassen, hat der Mittelstand den Vorteil, dass für ihn Entwickler keine Human-Resources sind, sondern Mitarbeiter, die Spaß an der Arbeit haben sollen. Hinderlich an der Digitalisierung sind analoge Schnittstellen, wie sie sowohl im Gesundheitsbereich, aber auch bei manchen Verwaltungsämtern zu finden sind. Wenn aber die Politik wie anfangs in der Corona-Krise mehr auf Experten, als auf Interessenverbände und Lobbyisten hört, dann klappt das auch dort mit der Digitalisierung. Im Mittelstand jedenfalls ist die Digitalisierung schon seit Jahrzehnten angekommen.

## Quellen:

- 1 Peter Altmeier: <https://bit.ly/altmeier-p1>
- 2 Hanna Prinz Zitat: <https://bit.ly/prinz-h2>
- 3 <https://bit.ly/heise-h3>
- 4 <https://bit.ly/heise-h4>
- 5 <https://bit.ly/heise-h5>

#JAVAPRO #Projektmanagement #Keycloak

# Business-Impact-Analyse mit nur 5 Slides

Um zukünftigen Schadensszenarien begegnen zu können, müssen Unternehmen im Kontext eines Notfall- bzw. Risikomanagements eine Erhebung ihrer kritischen IT-Systeme durchführen, insbesondere im Rahmen von IT-Architektur-Reviews, Assessments, Audits oder weil die Compliance dies erfordert. In der Praxis ist eine solche Betrachtung viel zu umfänglich. Eine Business-Impact-Analyse kann jedoch mit vereinfachtem Umfang auf fünf Schaubilder reduziert werden.

Eine gängige und anerkannte Methode für die Erhebung kritischer IT-Systeme ist das Notfallmanagement nach BSI 100-4<sup>6</sup>. Der Umfang dieses Vorgehens betrachtet jedoch nicht nur die IT-Infrastruktur, sondern eben ein unternehmensweites Notfallmanagement inklusive Organisation und Prozesse. Nicht selten ist ein Analyseprojekt in diesem Kontext bis zu mehrere Hundert PT groß. In der Praxis ist diese Betrachtung viel zu umfänglich, da man im ersten Schritt nur den Technologie-Stack betrachten möchte. Sofern Prozesse und Organisation nicht im Fokus stehen, kann eine Business-Impact-Analyse (BIA) mit vereinfachtem Umfang auf fünf Schaubilder reduziert werden.

## BSI 100-4 als Basis für die Erhebung

Der Vorteil einer BIA liegt darin, dass ein strukturierter Blick auf IT-Systeme oder IT-Verbunde durchgeführt werden kann und wesentliche Informationen zur Kritikalität einzelner Fachanwendungen oder Technologiekomponenten ermittelt werden können. Im Gegensatz zur üblichen Kritikalitätseinstufung, die bei vielen IT-Dienstleistern existieren und in der Regel nur Systeme im Kontext eines Kerngeschäftsprozesses und deren zugeordneten IT-Systemen berücksichtigen, prüft eine BIA-Erhebung<sup>7</sup> die Auswirkung eines IT-Systemausfalles auf folgende Schadensszenarien:

- die finanzielle Auswirkung
- die Beeinträchtigung der Aufgabenerfüllung
- die Auswirkung auf Gesetze, Vorschriften und Verträge
- die Auswirkungen auf das Image des Unternehmens
- die Auswirkungen auf die persönliche Unversehrtheit<sup>1</sup>

Fachanwendungen sind Informationssysteme die Kernprozesse, Managementprozesse oder Unterstützungsprozesse innerhalb

eines Unternehmens abbilden. Unterhalb der Fachanwendungen muss ein Technologie-Stack existieren, damit Informationssysteme aufgebaut, betrieben und überwacht werden können. Dieser Technologie-Stack besteht wiederum aus Technologiekomponenten wie Hardware (Server, Netzwerk, Storage, Backup etc.) und Software (Betriebssysteme, Middleware, Datenbanken etc.). Im Rahmen der BIA-Erhebung wird jede relevante Fachanwendung und bzw. oder Technologiekomponente auf jedes der vier vorab genannten Schadensszenarien geprüft. Maßgeblich dabei ist die Ausfallzeit (< 4 Std, < 24Std, < 3Tage, < 3-10 Tage, > 10 Tage). Alle Daten werden anschließend in einer umfangreichen Dokumentation mit Word oder Excel erfasst und nachfolgend analysiert. Der BSI-Standard zum Notfallmanagement wird aktuell überarbeitet und zukünftig in BSI200-4<sup>8</sup> fortgeführt.

## Autor:

Gerd Herbertz, Jahrgang 1969, ist ausgebildeter Dipl. Ing (FH) Elektrotechnik. Bei adesso verantwortet er im Geschäftsfeld IT-Management Consulting das Competence Center Infrastruktur & Technologie, welches sich mit den Technologiestacks Microsoft und Linux beschäftigt. Speziell im Linux-Umfeld liegt der Fokus auf Konzeption, Aufbau und Übergabe von DevOps-Umgebungen im Rahmen von Kundenprojekten. Neben dem reinen Engineering und der Technologieberatung wird das Portfolio des Competence Centers durch IT-Management- und Strategieberatung abgerundet.



## Die Anwendungs- und Technologielandkarte

Zur Erfassung und komprimierten Darstellung einer BIA auf fünf Slides muss zuerst eine Aufnahme der IT-Architektur bestehend aus Fachanwendungen und Technologien erfolgen. Wesentliche Elemente dieser Darstellung sind:

- Organisationseinheiten
- Systemkritikalitäten
- Fachanwendungen
- Technologiekomponenten
- Grad der Anwendungskomplexität (sofern bekannt)
- Zusatzinformationen:
  - ◊ oIT-Service<sup>2</sup> wird extern erbracht
  - ◊ oIT-Service in Ablösung
  - ◊ oIT-Service in Entwicklung
  - ◊ okeine relevante Komponente im Kontext der Kritikalität<sup>3</sup>

Die Zuordnung von Fachanwendungen zur Organisationseinheit muss dabei wie folgt umgesetzt werden:

- Fachanwendungen, die nur durch eine Organisationseinheit verwendet werden, werden nur dort einmalig aufgeführt.
- Fachanwendungen die durch mehrere Organisationseinheiten verwendet werden, müssen jeder Organisationseinheit einzeln zugewiesen werden<sup>4</sup>.
- Fachanwendungen, die alle Fachbereiche nutzen, gehören in die Rubrik **Übergreifend**.

Im Kontext der Bestandsaufnahme wird die grundsätzliche Kritikalität der Fachanwendungen und Technologiekomponenten erhoben. In diesem Zusammenhang werden die Anwendungen mit einem Farbcode entsprechend eingefärbt (Abb. 1).

Diese Darstellung bildet dann die Grundlage der BIA-Erhebung. Als Zwischenformat empfiehlt sich hier die Nutzung einer Excel-Liste zur Sammlung und Abstimmung aller Fachanwendungen

und Technologiekomponenten. Nach Abstimmung der Liste kann dann die grafische Landkarte innerhalb von PowerPoint erstellt werden. Diese Landkarte bildet die Basis für die Durchführung einer BIA-Light. (Abb. 2).

## BIA-Erhebung

Die Erfassung aller wesentlichen Informationen gemäß der im Modul **Business-Impact-Analyse** vorgegebenen Schadensszenarien erfasst je Schadensszenario vier Auswirkungsgrade, die dort entsprechend beschrieben sind:

1. Niedrig
2. Mittel
3. Hoch
4. Sehr hoch

Um ein direktes Mapping zu den drei Farbcodes in der Landkarte gemäß (Abb. 2) zu bekommen, werden die Auswirkungsgrade **Hoch** und **Sehr hoch** zusammengefasst. Die obige Variante beschreibt dabei die genaue Variante. Das Problem liegt in der Regel darin, hier eine adäquate Quantifizierung des Schadens zu ermitteln, zum Beispiel den finanziellen Schaden. Es empfiehlt sich daher einen eher pragmatischen Ansatz zu wählen. Am besten auf Basis einer direkten Experteneinschätzung des Fachbereiches beziehungsweise der IT. Die BIA-Erhebung unterscheidet diese Aufnahme zusätzlich in zwei Szenarien:

- Kalenderjahr: folglich der normale Geschäftsbetrieb
- besondere Termine und Ereignisse

Hier empfiehlt sich ebenfalls zur Vereinfachung immer die höchstmögliche Auswirkung einmalig zu erfassen. Am besten auf Basis eines gemeinsamen Workshops mit direkter Anpassung des Farbcodes je Schadensszenario, in Abhängigkeit des Auswirkungsgrades für jede Fachanwendung beziehungsweise Technologiekomponente.

### Aufnahme der IT-Architektur

- Definition der IT-Systeme im Kontext der Kritikalität -

Zur Erfassung der IT-Architektur bestehend aus Technologien, Fachanwendungen und Werkzeugen der TUIV werden folgende IT-Systeme im Kontext der Kritikalität unterschieden

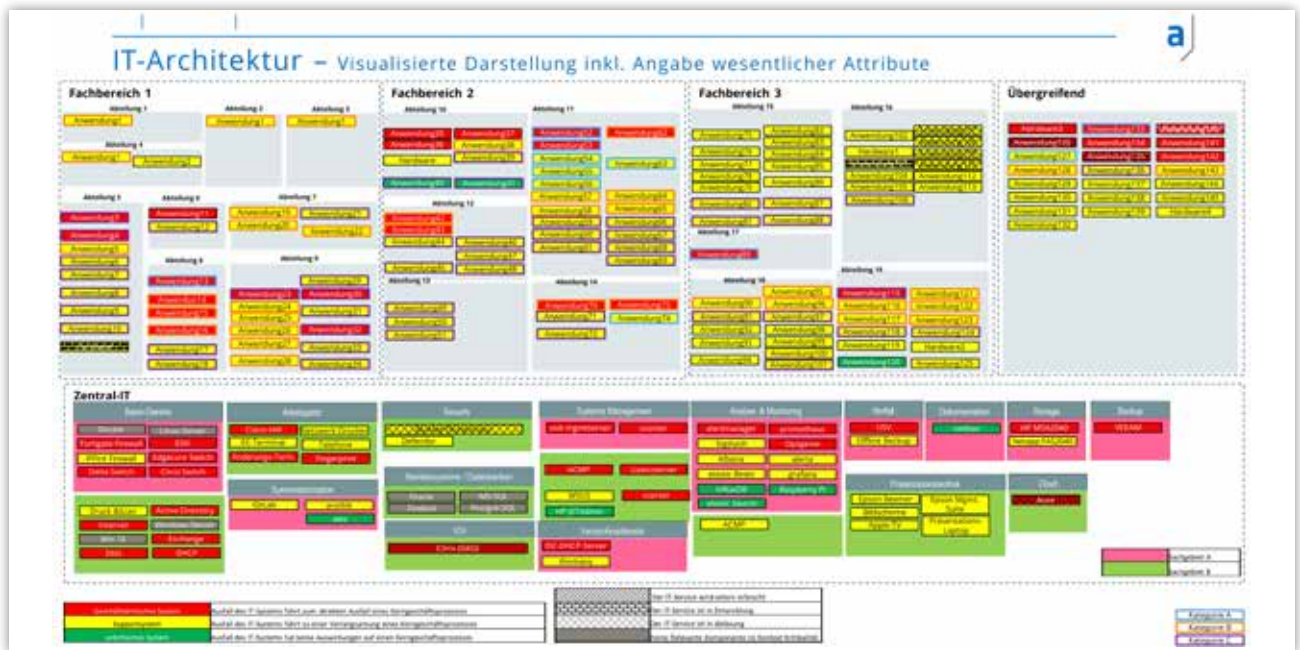
|                            |                                                                                  |
|----------------------------|----------------------------------------------------------------------------------|
| geschäftskritisches System | Ausfall des IT-Systems führt zum direkten Ausfall eines Kerngeschäftsprozesses   |
| Supportsystem              | Ausfall des IT-Systems führt zu einer Verlangsamung eines Kerngeschäftsprozesses |
| unkritisches System        | Ausfall des IT-Systems hat keine Auswirkungen auf einen Kerngeschäftsprozesses   |

**Geschäftskritisches System:** IT-System (Fachanwendung, Technologie, Werkzeug) das bei Ausfall zu einem „direkten“ und damit wirksamen Ausfall eines Kerngeschäftsprozesses führt

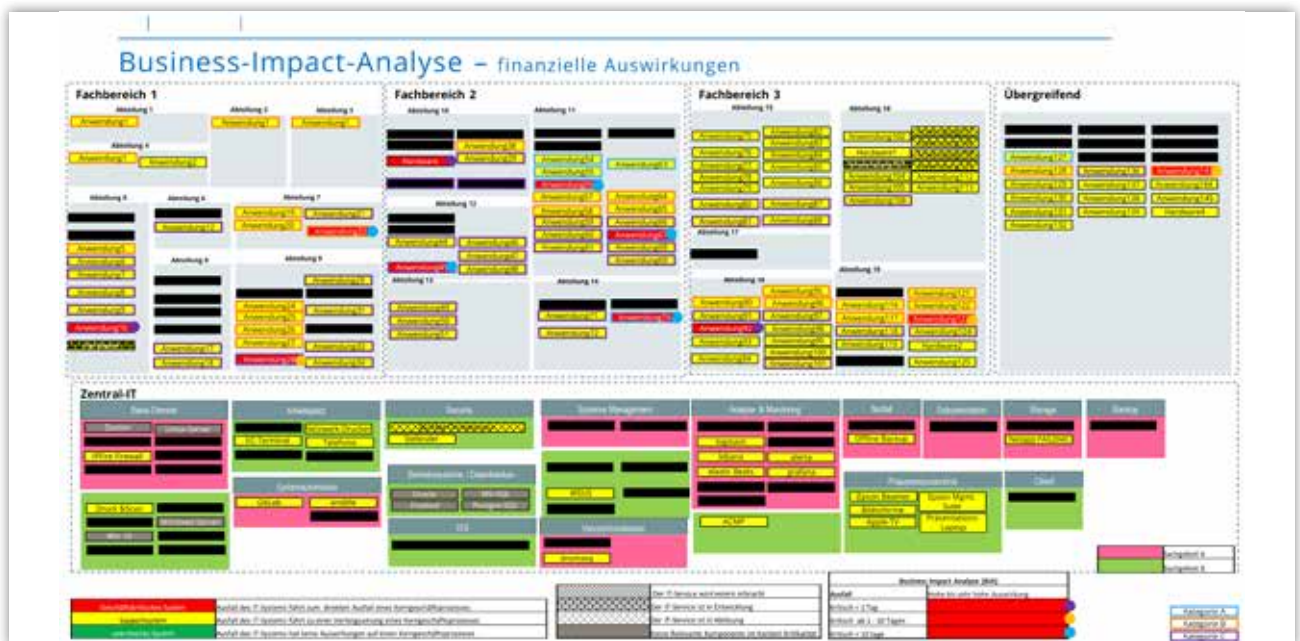
**Supportsystem:** IT-System (Fachanwendung, Technologie, Werkzeug) das bei Ausfall Kerngeschäftsprozesse verlangsamt und mittelfristig zu wirksamen Schaden führen kann

**Unkritisches System:** IT-System (Fachanwendung, Technologie, Werkzeug) das bei Ausfall zu keinem Zeitpunkt einen Kerngeschäftsprozess beeinflusst

Übersicht der Systemkritikalitäten (Abb.1)<sup>9</sup>



Beispielkarte Technologie + Fachanwendungen (Abb. 2)<sup>10</sup>



BIA: Finanzielle Auswirkung (Abb. 3)<sup>9</sup>

Bevor nun die Farbcodierung und damit die Erstellung der BIA beginnt, muss eine Anpassung der Anwendungs- und Technologielandkarte erfolgen. Und zwar müssen alle vorab kritischen Systeme (**Rot**), sowie alle unkritischen Systeme (**Grün**) ausgeblendet werden, indem sie schwarz ausgefüllt werden, denn

- kritische Systeme bleiben auch immer kritisch und
- unkritische Systeme werden niemals kritisch.

Nun ist die Basis für die eigentliche BIA-Light geschaffen worden und die Erhebung kann beginnen.

## Schadensszenario: Finanzielle Auswirkung

Betrachtet man nun die erste Folie zur BIA-Light, stellt man fest, dass eine weitere Legende dazugekommen ist. Hier wird grafisch vereinfacht dargestellt, welche identifizierten, neuen, kritischen Fachanwendungen bzw. Technologiekomponenten nach welcher Laufzeit kritisch<sup>5</sup> werden. Dazu werden in dem anonymisierten Beispiel einfach Farbpunkte an das jeweilige Item platziert und damit farblich kenntlich gemacht. Auch hier wird wieder vereinfacht und die Ausfallzeit wie folgt kategorisiert bzw. gebündelt:

- kritisch < 1 Tag
- kritisch 1 - 10 Tage
- kritisch > 10 Tage

Falls dies nicht dem Wunsch oder Anspruch genügt, kann hier detaillierter vorgegangen werden (**Abb. 3**). Hier wurden zum Beispiel keinerlei finanzielle Risiken aus Sicht der IT (Technologiekomponenten) erfasst, sondern ausschließlich kritische Fachanwendungen neu identifiziert. Dieses Vorgehen wird nun für die restlichen drei Schadensszenarien wiederholt angewendet.

### Schadensszenario: Beeinträchtigung der Aufgabenerfüllung

In (**Abb. 4**) erkennt man wie massiv die Aufgabenerfüllung bei längeren Laufzeiten eingeschränkt wird. Dies ist typisch für Unternehmen mit hoher Kundeninteraktion oder in den Bereichen Fertigung, Forschung und Entwicklung. Zu sehen ist ein massiver Anstieg an zusätzlichen kritischen Fachanwendungen im Bereich **1-10 Ausfallzeit**. Nach ungefähr zehn Tagen Ausfallzeit hat sich die Anzahl der vorab kritisch identifizierten Anwendungen aus (**Abb. 2**) fast verdoppelt.

### Schadensszenario: Verstoß gegen Gesetze, Vorschriften und Verträge

Im Vergleich zu den beiden vorherigen Schadensszenarien ist hier zu sehen, dass generell weniger Objekte betroffen sind. Bei Unternehmen, die Compliance-Standards wie VAIT, BAIT oder KAIT unterliegen, würde es bezüglich der Menge etwas anders aussehen.

Zu guter Letzt zeigt die Betrachtung des Imageschadens, dass auch hier relativ viele zusätzliche Fachanwendungen und Technologiekomponenten betroffen sind. Dies ist für Unternehmen mit starker öffentlicher Wahrnehmung und davon abhängigen Geschäftsmodellen sehr relevant.

### Was nun?

An diesem Punkt stellt sich die Frage was man mit diesen erhobenen, prägnanten Informationen eigentlich macht. Die Tatsache, dass es weitere, kritische Informationssysteme gibt, kann und muss dazu genutzt werden, die dortigen aktuell verwendeten Backup- und Recovery-Prozeduren zu überprüfen. Die BIA-Light liefert einen guten Fahrplan mit abgeleiteter Priorisierung, um aktuell nicht erfasste und behandelte Fachanwendungen und bzw. oder Technologien zu behandeln.

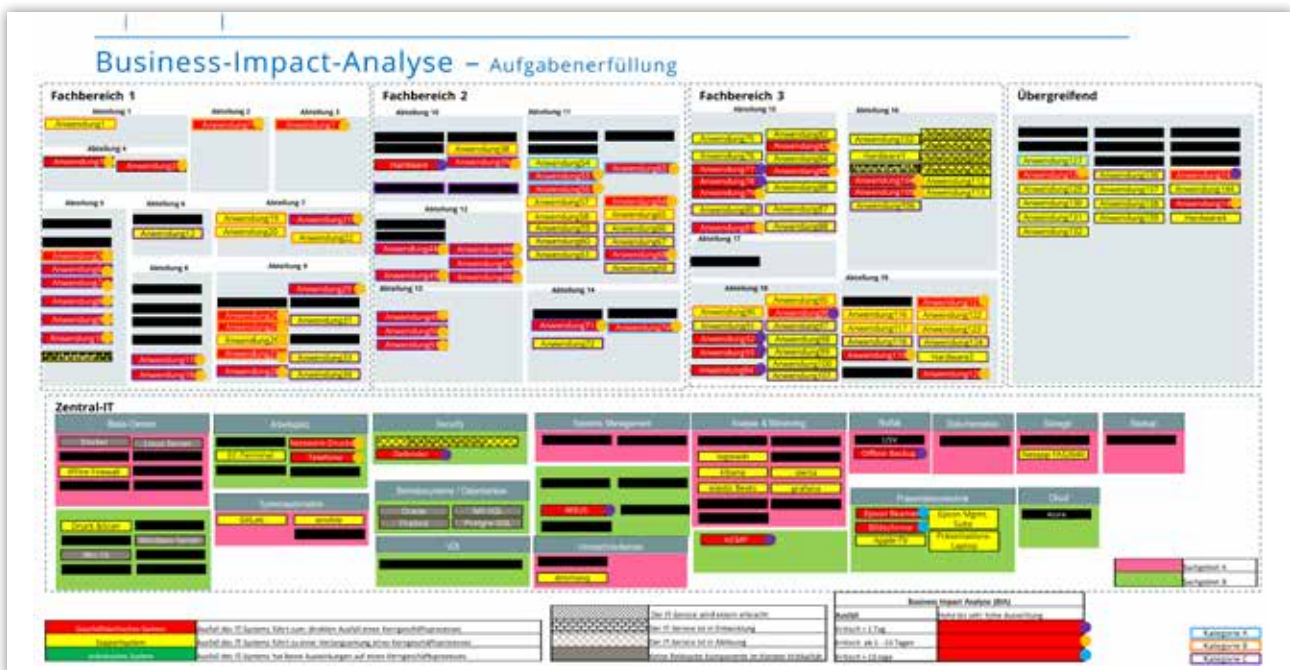
Für Fachanwendungen kann dies bedeuten:

- die IT-Architektur hochverfügbarer zu gestalten
- ein adäquates Monitoring bereitzustellen, um kritische Systemzustände frühzeitig zu erfassen
- Fachtests zu automatisieren
- die Wiederherstellung einer konsistenten Anwendung regelmäßig zu prüfen oder zumindest Stichproben durchzuführen

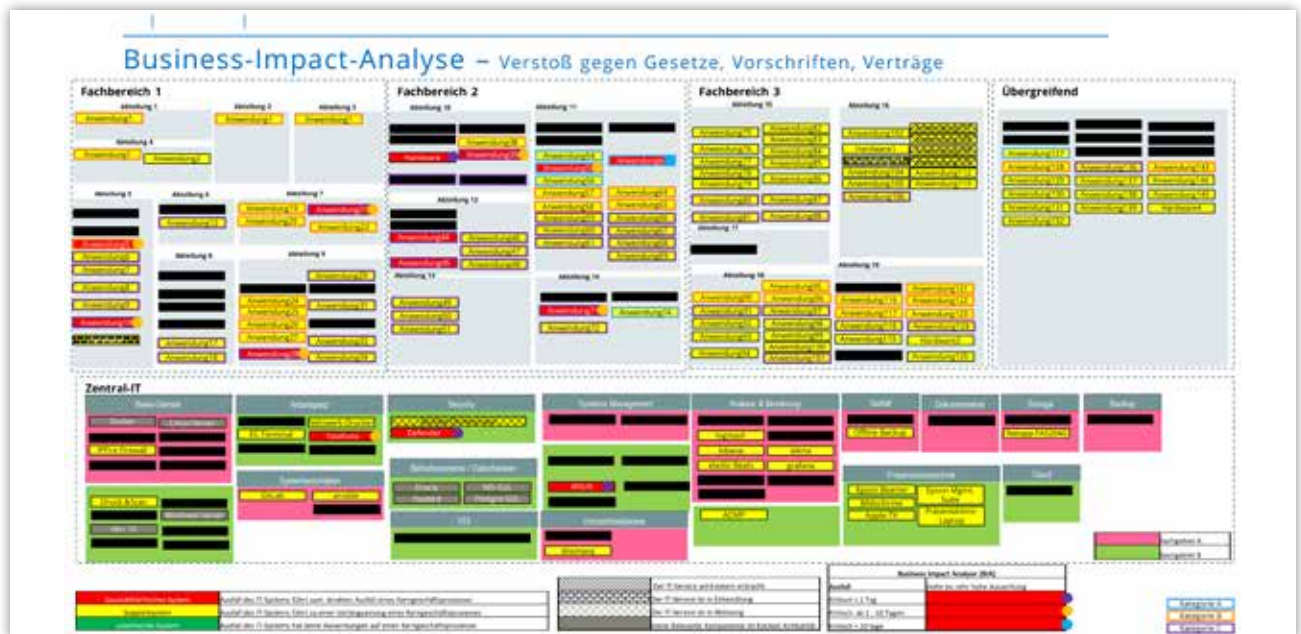
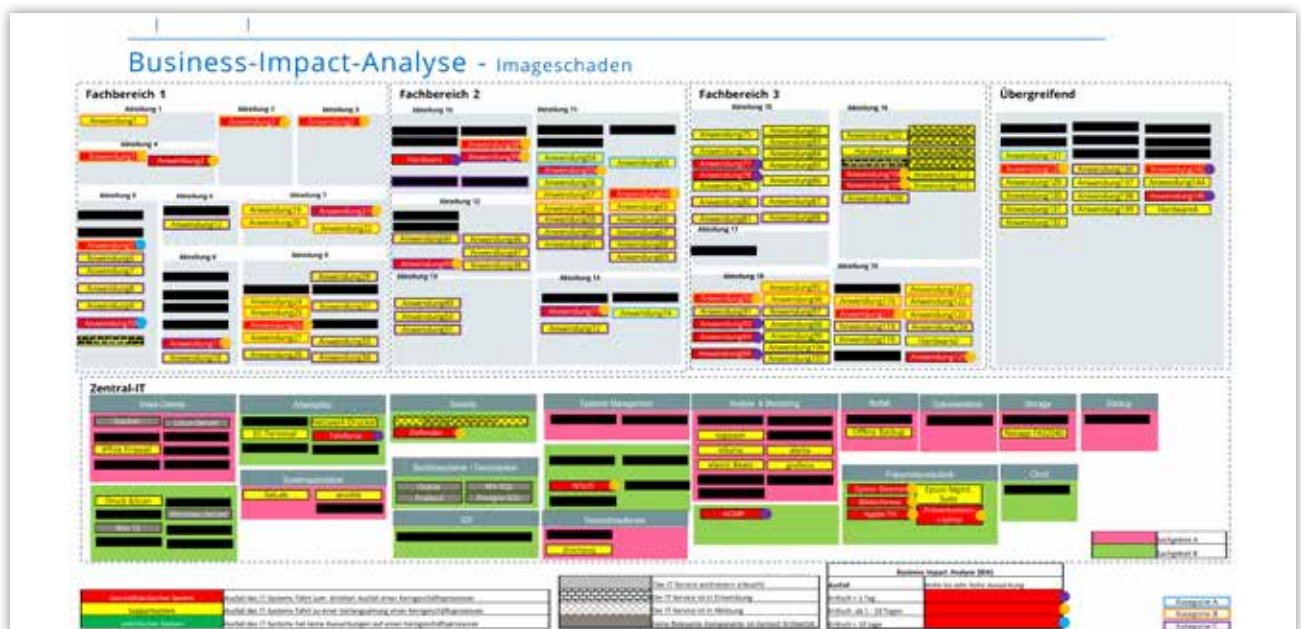
Für Technologiekomponenten bedeutet dies:

- hochverfügbar zu sein
- wesentliche Statusinformationen für das Monitoring zu liefern
- Technologien zu konsolidieren

Weiterhin hat diese eher kurze aber sehr detaillierte Aufnahme den Vorteil, dass erste Schritte zur Einführung von Business-Continuity-Management (BCM) erfolgt sind. Dies wird immer relevanter, da mehr und mehr Branchen gesteigertes Interesse auf ein BCM legen werden.



BIA: Beeinträchtigung der Aufgabenerfüllung (Abb. 4)<sup>9</sup>

BIA: Auswirkung auf Gesetze, Vorschriften, Verträge (Abb. 5)<sup>9</sup>BIA: Auswirkungen auf das Image (Abb. 6)<sup>9</sup>

### Fazit:

Das hier aufgezeigte Vorgehen spart Zeit und Geld, um einen ersten Blick auf das Unternehmen und dessen IT-Systeme sowie deren Kritikalität zu erlangen. Die Darstellung ist managementtauglich und kann auch von Nichttechnologen verstanden und angewendet werden. Um eine genauere Übersicht zu geben, kann neben den vorgestellten Slides eine weitere Vertiefung im Rahmen von Analysen vorgenommen werden, die zum Beispiel die veränderten Kritikalitäten im Kontext der Ausfallzeit noch einmal aufbereitet darstellen. Ausgehend vom Normalzustand in (Abb.2) ist eine tabellarische Übersicht der kritischen Fachanwendungen und Technologiekomponenten, die zusätzlich je Schadenszenario hinzukommen, sinnvoll.

- 1 Für BIA-Erhebungen bei IT-Dienstleistungsunternehmen ist dies in der Regel nicht erforderlich
- 2 Ein IT-Service wird durch die Kombination von Informationstechnologie, Menschen und Prozessen gebildet
- 3 Dies bezieht sich auf Basistechnologien, die ohne einen entsprechenden Nutzungskontext keine Kritikalität haben (also z.B. Datenbanken, Client/Server-Software)
- 4 Bei hochkomplexen Umgebungen muss ggf. vereinfacht werden, damit die Darstellung nicht überladen wird
- 5 Entspricht dem Auswirkungsgrad „hoch/sehr hoch“ -> Farbcode: „Rot“

### Quellen:

- 6 BSI 100-4: <https://bit.ly/BSI-100-4>
- 7 BIA-Erhebung: <https://bit.ly/bsi-bund-BIA>
- 8 BSI200-04: <https://bit.ly/BSI-200-04>
- 9 adesso AG

#JAVAPRO #Innovationen

# Vernetzte Systeme mit FEPCOS-J

FEPCOS-J ist ein Programmiergerüst für Java, das die Entwicklung von zusammengesetzten, vernetzten Systemen, wie zum Beispiel Roboterschwärme, beschleunigt, indem es die benötigte Netzwerkprogrammierung automatisiert und die Spezifikation von Nebenläufigkeit vereinfacht. Wir stellen die Grundlagen und mit einem einfachen Beispiel die Anwendung von FEPCOS-J vor.

## Autor:



Dipl.-Inf. Univ. Gerhard Fuchs hat 18 Jahre Erfahrung mit Java und entwickelt Programmierkonzepte und Netzwerkprotokolle. Er hat an der FAU Erlangen-Nürnberg u.a. Netzwerkprogrammierung gelehrt und Programmierkonzepte für Robotersensornetze und die dafür benötigte Aufgabenverteilung erforscht. Derzeit arbeitet er am FEPCOS-Projekt.

Webseite: <http://fepcos.info>

E-Mail: [fuchs@fepcos.info](mailto:fuchs@fepcos.info)

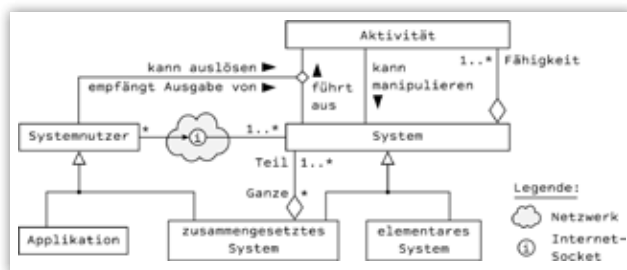
RSS-Feed: <http://fepcos.info/news.rss>

„Das Ganze ist mehr als die Summe seiner Teile.“  
Aristoteles (384 v.Chr. - 322 v.Chr.). Dieses Prinzip ist auf ein vernetztes, technisches System, das aus Teilsystemen zusammengesetzt ist, übertragbar. Zum Beispiel ist bei einem Roboterschwarm der Schwarm das Ganze und die Roboter sind die Teile. Jeder Roboter kann für sich fahren. Der Schwarm kann mehr. Er kann zum Beispiel ausschwärmen, indem die einzelnen Roboter in verschiedene Richtungen fahren. Dies ist ein möglicher Mehrwert. Allerdings bekommt man diesen nicht geschenkt. Bei der Entwicklung eines vernetzten, technischen Systems, das ein aus Teilen zusammengesetztes Ganzes ist, ist mehr zu beachten als bei einem einzelnen System. Die Programmierung derartiger Systeme umfasst Netzwerkkommunikation und systemübergreifende Nebenläufigkeit, da die Teile kommunizieren und hochgradig parallel arbeiten. Dies ist ein Mehraufwand.

Im Rahmen des FEP-COS-Projekts<sup>1</sup> entsteht FEP-COS-J. FEP-COS steht für den englischen Ausdruck **Framework-for-the-Programming-of-Composed-Systems**. FEP-COS-J verringert den Mehraufwand, indem es die Programmierung der Netzwirkommunikation automatisiert und die Spezifikation von Nebenläufigkeit vereinfacht. FEP-COS-J stellt anwendungsspezifische Java-Annotationen<sup>2</sup>, den FEP-COS-J-Processor **fjp** und den FEP-COS-J-Exporter **fjx** zur Verfügung. Die Java-Annotationen erlauben es, derartige Systeme strukturiert zu programmieren. **fjp** wertet die Annotationen aus und generiert Code; **fjx** ermöglicht den Zugriff auf die Systeme innerhalb eines Netzwerks.

## Grundlegende Begriffe

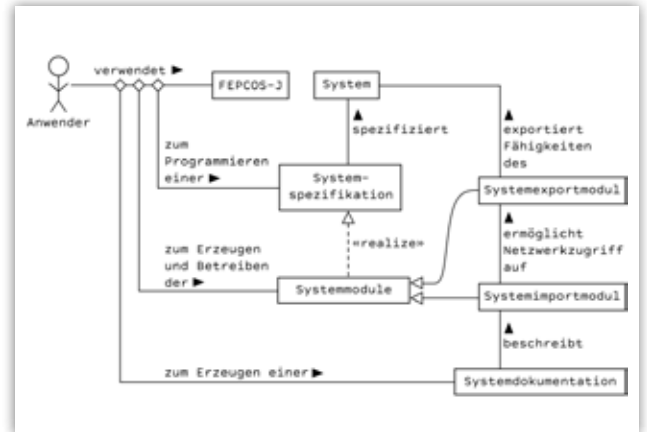
FEP-COS-J basiert auf dem in (Abb. 1) dargestellten Modell. Ein System besitzt mindestens eine Fähigkeit, sprich eine **Aktivität**, die es innerhalb einer endlichen Zeit ausführen kann. Eine **Aktivität** kann ein System manipulieren. Ein elementares System oder ein zusammengesetztes System ist ein System. Ein zusammengesetztes System ist ein Ganzes, das aus mindestens einem System bzw. Teil besteht. Ein System kann ein Teil von beliebig vielen Ganzen sein. Ein **Systemnutzer** greift über ein Internet-Socket innerhalb eines Netzwerks auf mindestens ein System zu. Ein **Systemnutzer** kann so das Ausführen einer **Aktivität** anstoßen und die resultierende Ausgabe empfangen. Auf ein System können beliebig viele **Systemnutzer** zugreifen. Eine Applikation oder ein zusammengesetztes System ist ein möglicher **Systemnutzer**.



Grundlegendes Modell von FEP-COS-J (Abb. 1)

## Verwendungsmöglichkeiten von FEP-COS-J

Ein Anwender von FEP-COS-J, nachfolgend nur noch Anwender genannt, verwendet FEP-COS-J zum Programmieren einer Systemspezifikation, zum Erzeugen und Betreiben von Systemmodulen und zum Erzeugen einer Systemdokumentation (Abb. 2). Die Systemspezifikation spezifiziert ein System; die Systemmodule realisieren die Systemspezifikation. Das **Systemexportmodul** und das **Systemimportmodul** sind Systemmodule. Das **Systemexportmodul** exportiert die Fähigkeiten des Systems, das **Systemimportmodul** ermöglicht einen Netzwerkzugriff darauf. Die Systemdokumentation beschreibt das **Systemimportmodul**, sprich die Schnittstellen, mit denen ein Systemnutzer auf das **Systemexportmodul** und somit auf die Fähigkeiten des Systems zugreifen kann.



Verwendungsmöglichkeiten von FEP-COS-J (Abb. 2)

FEP-COS-J ermöglicht:

- die Spezifikation der Fähigkeiten eines Systems. Der Anwender legt diese fest. FEP-COS-J kümmert sich darum, dass auf diese innerhalb eines Netzwerks zugegriffen werden kann und automatisiert die dafür benötigte Netzwerkprogrammierung.
- die Spezifikation der Teile eines Systems. Der Anwender legt diese fest. FEP-COS-J kümmert sich um deren Verwaltung und sorgt dafür, dass die benötigten Netzwerkverbindungen automatisch auf- und abgebaut werden.
- einen blockierenden und einen nebenläufigen Zugriff auf die Fähigkeiten eines Systems. Der Anwender muss nur die **Aktivität** programmieren. FEP-COS-J stellt die entsprechenden Schnittstellen bereit und generiert die dazugehörigen JAVADOC-Kommentare.

FEP-COS-J benötigt zwingend:

- Java 11 oder höher, da es in Java-Version 11 geschrieben ist und auf dem Modulsystem aufbaut
- ein IPv4-Netzwerk, über das die Java VMs kommunizieren können

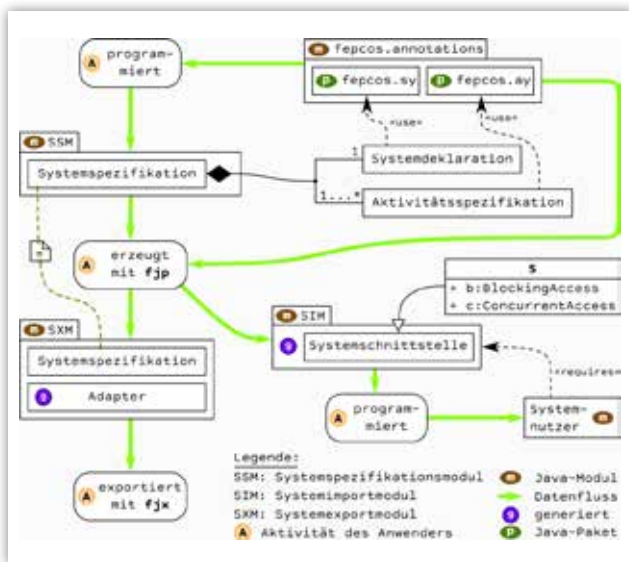
FEP-COS-J ist derzeit noch in Entwicklung und stellt noch keine Authentifizierungs- und Verschlüsselungsmechanismen bereit. Das bedeutet, dass jeder, der Zugriff auf das Netzwerk hat, über einen uneingeschränkten Zugriff auf die Fähigkeiten der sich darin befindenden Systeme verfügt. Es obliegt dem Anwender, das Netzwerk gegebenenfalls vor unbefugten Zugriffen abzusichern. FEP-COS-J wird auf Debian-basierten Linux-Computern mit OpenJDK ab Version 11 entwickelt und getestet. Ob es mit anderen Betriebssystemen und Java-VMs funktioniert, muss sich zeigen.

## Das Prinzip von FEP-COS-J

(Abb. 3) zeigt die prinzipielle Anwendung von FEP-COS-J, insbesondere des FEP-COS-J-Processors **fjp** und des FEP-COS-J-Exporters **fjx**.

# Der Anwender

- programmiert die **Systemspezifikation**, die aus genau einer **Systemdeklaration** und mindestens einer **Aktivitätsspezifikation** besteht, innerhalb eines **Systemspezifikationsmoduls** unter Verwendung der Annotationen von FEPCOS-J. Er verwendet zum Programmieren der **Systemdeklaration** die in (Tab. 1) aufgelisteten Annotationen des Pakets **fepcos.sy** und zum Programmieren einer **Aktivitätsspezifikation** die in (Tab. 2) aufgelisteten Annotationen des Pakets **fepcos.ay**. Die beiden Pakete befinden sich im Java-Modul **fepcos.annotations**.
- erzeugt mit **fjp** aus dem **Systemspezifikationsmodul** das **Systemexportmodul** und das **Systemimportmodul**. Hierbei wertet **fjp** die Annotationen aus. Das **Systemexportmodul** enthält die vom Anwender programmierte **Systemspezifikation** und einen von **fjp** generierten Adapter. Das **Systemimportmodul** ist komplett von **fjp** generiert und enthält eine Systemschnittstelle, die **S.b** und **S.c** umfasst. **S.b** ermöglicht einen blockierenden Zugriff (**BlockingAccess**) auf das System und **S.c** erlaubt einen nebenläufigen Zugriff (**ConcurrentAccess**) darauf.
- exportiert das **Systemexportmodul** mit **fjx**.
- verwendet die Systemschnittstelle des **Systemimportmoduls** zum Programmieren eines **Systemnutzers**, welcher deshalb das **Systemimportmodul** benötigt.



Anwendung von FEPCOS-J (Abb. 3)

|               |                                                                                                                                                                                                                                                                                                                                   |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>@SYDec</b> | <p>Markiert die Klasse einer Systemdeklaration. Als Parameter kann ein String(-Array) übergeben werden, dessen Inhalt <b>fjp</b> als Textbaustein für die Systemdokumentation verwendet.</p> <p>Beispiel: <b>@SYDec(„Ein nettes System“) class SYS { ... }</b> besagt, dass die Klasse <b>SYS</b> eine Systemdeklaration ist.</p> |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

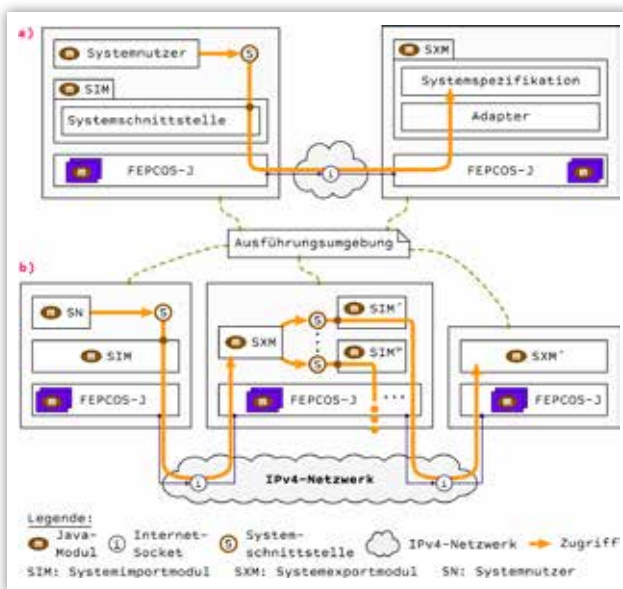
|               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>@Cap</b>   | <p>Markiert eine Objektvariablendeklaration und deklariert eine Fähigkeit. Der Typ der Deklaration muss die Klasse einer Aktivitätsspezifikation sein. Der Name ist frei wählbar. <b>@Cap</b> muss mindestens einmal in einer Systemdeklaration vorkommen.</p> <p>Beispiel: <b>@Cap Größe sagHallo;</b> deklariert eine Fähigkeit <b>sagHallo</b>, wobei die <b>Aktivität Größe</b> ist.</p>                                                                                                                                                                                                  |
| <b>@Part</b>  | <p>Markiert eine Objektvariablendeklaration und spezifiziert einen Teil. Der Typ der Deklaration muss die Klasse einer Systemschnittstelle sein. Der Name ist frei wählbar. Als Parameter müssen eine IP4-Adresse <b>ip</b> und ein Port <b>port</b> angegeben werden. <b>@Part</b> kann beliebig oft in einer Systemdeklaration vorkommen.</p> <p>Beispiel:<br/><b>@Part(ip="10.0.0.2", port=22222) hallo.system.S hs;</b> deklariert einen Teil <b>hs</b> dessen Systemschnittstelle <b>hallo.system.S</b> ist und der über das Internet-Socket <b>10.0.0.2:22222</b> adressierbar ist.</p> |
| <b>@Start</b> | <p>Markiert eine Methode, die beim Starten des Exports des Systems aufgerufen werden soll. Dies ermöglicht zum Beispiel das Starten von Hardware-Treibern. <b>@Start</b> darf maximal einmal in einer Systemdeklaration vorkommen.</p> <p>Beispiel: <b>@Start void start() { ... }</b></p>                                                                                                                                                                                                                                                                                                    |
| <b>@Stop</b>  | <p>Markiert eine Methode, die beim Beenden des Exports des Systems aufgerufen werden soll. Dies ermöglicht zum Beispiel das Beenden von Hardware-Treibern. <b>@Stop</b> darf maximal einmal in einer Systemdeklaration vorkommen.</p> <p>Beispiel: <b>@Stop void stop() { ... }</b></p>                                                                                                                                                                                                                                                                                                       |

Für eine Systemdeklaration benötigte Annotationen aus dem Paket **fepcos.sy** (Tab. 1)

|                  |                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>@AYSpec</b>   | <p>Markiert die Klasse einer Aktivitätsspezifikation. Als Parameter kann ein String(-Array) übergeben werden, dessen Inhalt <b>fjp</b> als Textbaustein für die Systemdokumentation verwendet.</p> <p>Beispiel:<br/><b>@AYSpec(„Grüßt den Benutzer ...“) class Größe { ... }</b> besagt, dass die Klasse <b>Größe</b> eine Aktivitätsspezifikation ist.</p>                         |
| <b>@Behavior</b> | <p>Markiert eine Methode, die das gewünschte Verhalten des Systems beim Ausführen der <b>Aktivität</b> festlegt. Optional kann der Methode die Systemdeklaration als Parameter übergeben werden, sodass diese manipuliert werden kann. <b>@Behavior</b> muss genau einmal in einer Aktivitätsspezifikation vorkommen.</p> <p>Beispiel: <b>@Behavior void go(SYS2 s) { ... }</b></p> |

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>@In</b>  | Markiert eine Objektvariablendeklaration, die einen Eingabeparameter spezifiziert. Als Parameter der Annotation kann ein String übergeben werden, dessen Inhalt <b>fjp</b> als Textbaustein für die Systemdokumentation verwendet. Als Typ der Deklaration sind <b>byte</b> , <b>char</b> , <b>short</b> , <b>int</b> , <b>long</b> , <b>float</b> , <b>double</b> , <b>String</b> und <b>byte[]</b> erlaubt. Der Name ist frei wählbar. <b>@In</b> darf beliebig oft in einer Aktivitätsspezifikation vorkommen.<br>Beispiel:<br><b>@In(„Zu grüßender Benutzer“) String benutzer;</b> |
| <b>@Out</b> | Markiert eine Objektvariablendeklaration, die einen Ausgabeparameter spezifiziert. Als Parameter der Annotation kann ein String übergeben werden, dessen Inhalt <b>fjp</b> als Textbaustein für die Systemdokumentation verwendet. Als Typ der Deklaration sind <b>byte</b> , <b>char</b> , <b>short</b> , <b>int</b> , <b>long</b> , <b>float</b> , <b>double</b> , <b>String</b> und <b>byte[]</b> erlaubt, der Name ist frei wählbar. <b>@Out</b> darf beliebig oft in einer Aktivitätsspezifikation vorkommen.<br>Beispiel: <b>@Out(„Der Gruß“) String gruß;</b>                   |

Für eine Aktivitätsspezifikation benötigte Annotationen aus dem Paket **fepcos.ay** (Tab. 2)



Verteilung der Systemmodule und Zugriffsmöglichkeiten der Systemnutzer (Abb. 4)

Der Anwender verteilt die Java-Module auf zwei verschiedene Ausführungsumgebungen auf denen FEP-COS-J installiert ist (Abb. 4a): Die eine enthält das **Systemimportmodul** und den **Systemnutzer**, die andere enthält das **Systemexportmodul**. Die beiden Ausführungsumgebungen sind über ein IPv4-Netzwerk verbunden und können so miteinander kommunizieren. Hierbei ist das **Systemexportmodul** über ein Internet-Socket eindeutig adressierbar. Ein **Systemnutzer** kann so über das Netzwerk auf die Systemspezifikation und so auf die vom Anwender programmierten Fähigkeiten des Systems zugreifen. Der Zugriff des

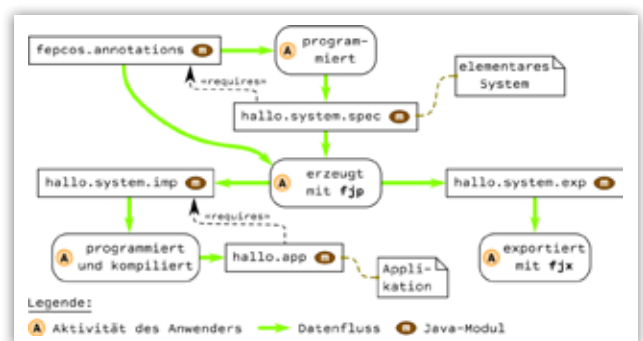
**Systemnutzers** erfolgt hierbei über die Systemschnittstelle des **Systemimportmoduls**, über Module von FEP-COS-J, über das Internet-Socket, über Module von FEP-COS-J, über den Adapter des **Systemexportmoduls** auf die ebenfalls darin enthaltene Systemspezifikation. Ein **Systemnutzer** kann auf beliebig viele **Systemimportmodule** parallel zugreifen. Handelt es sich bei dem **Systemnutzer** um ein zusammengesetztes System, befindet sich in dessen Ausführungsumgebung ein **Systemexportmodul SXM**, und mindestens ein **Systemimportmodul SIM** (Abb. 4b). So kann ein **Systemnutzer** über das zu **SXM** gehörende **Systemimportmodul SIM** beim zusammengesetzten System das Ausführen einer **Aktivität** auslösen, worauf diese über **SIM** auf ein weiteres Systemexportmodul **SXM**, sprich auf die Fähigkeiten eines weiteren Systems, zugreift. Die einzelnen Zugriffe erfolgen wie oben beschrieben.

### Beispiel:

#### Szenario – elementares System und Applikation

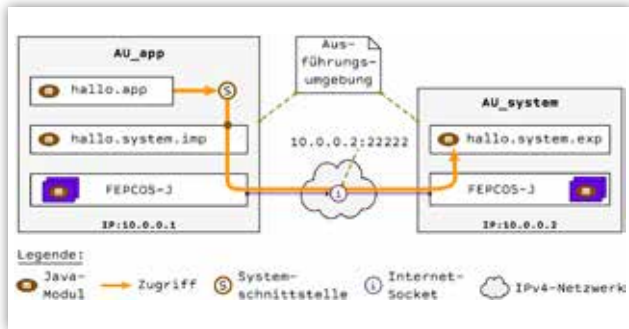
Nachfolgend wird die Anwendung von FEP-COS-J am Beispiel eines elementaren Systems **hallo.system** und einer Applikation **hallo.app** als **Systemnutzer**, beschrieben (Abb. 5). Der Anwender programmiert die Spezifikation von **hallo.system** mit dem Java-Modul **hallo.system.spec** unter Verwendung der Annotationen des Java-Moduls **fepcos.annotations**. Er erzeugt mit **fjp**, welcher die Annotationen auswertet, aus **hallo.system.spec** das Systemimportmodul **hallo.system.imp** und das Systemexportmodul **hallo.system.exp**. Er exportiert **hallo.system.exp** mit **fjx**. Schließlich programmiert und kompiliert er **hallo.app** unter Verwendung von **hallo.system.imp**.

Der Anwender verteilt die Systemmodule und die Applikation, wie in (Abb. 6) dargestellt, auf zwei **AU\_system** und **AU\_app** genannte Ausführungsumgebungen, auf denen FEP-COS-J installiert ist. **AU\_system** enthält **hallo.system.exp**. **AU\_app** enthält **hallo.app** und **hallo.system.imp**. Es wird vorausgesetzt, dass die Ausführungsumgebungen über ein IPv4-Netzwerk miteinander kommunizieren können. In diesem Beispiel hat **AU\_app** die IP-Adresse **10.0.0.1** und **AU\_system** die IP-Adresse **10.0.0.2**. **hallo.system.exp** stellt die Fähigkeiten des Systems



Anwendung von FEP-COS-J am Beispiel eines elementaren Systems und einer Applikation (Abb. 5)

über das Internet-Socket **10.0.0.2:22222** zur Verfügung, **hallo.system.imp** ermöglicht den Zugriff darauf. **hallo.app** kann so innerhalb des Netzwerks über die Systemschnittstelle von **hallo.system.imp** auf **hallo.system.exp** zugreifen – sprich **Aktivitäten** von **hallo.system** aufrufen und die Ergebnisse empfangen.



Beispielszenario: Zugriff Applikation auf elementares System (Abb. 6)

Nachfolgend wird die Umsetzung des Szenarios mit FEPCOS-J beschrieben. **hallo.system** entsteht auf **AU\_system** im Projektverzeichnis «PVS». **hallo.app** entsteht auf **AU\_app** im Projektverzeichnis «PVA» (Abb. 7).

Der Anwender

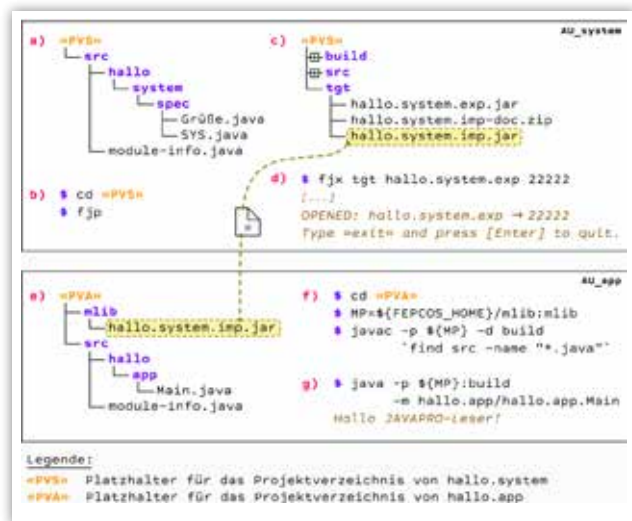
1. erstellt «PVS» (Abb. 7a). Hierfür programmiert er die Java-Dateien, wie im nächsten Abschnitt beschrieben und speichert diese an der entsprechenden Stelle.
2. erzeugt die Systemmodule und die Systemdokumentation (Abb. 7b). Hierzu ruft er im Verzeichnis «PVS» **fjp** auf. Der Processor erstellt einen **build** Ordner, den er für seine Arbeit benötigt und einen **tgt** Ordner, der die fertigen Systemmodule und die Systemdokumentation enthält (Abb. 7c). In **hallo.system.exp.jar** ist das Systemexportmodul, in **hallo.system.imp.jar** ist das Systemimportmodul und in **hallo.system.imp-doc.zip** ist die Systemdokumentation.
3. startet den Export, indem er im Verzeichnis «PVS» **fjx** aufruft (Abb. 7d). Als Parameter übergibt er:
  - das Verzeichnis **tgt**, welches das Exportmodul enthält,
  - den Namen des Exportmoduls **hallo.system.exp** und
  - den Port **22222** des Internet-Sockets.

Im Erfolgsfall wird eine entsprechende Meldung ausgegeben. Mit der Eingabe von **exit** kann das Exportieren beendet werden. Dies darf aber nicht vor Schritt 6 erfolgen, da sonst das Ausführen von **hallo.app** fehlschlägt.

4. erstellt «PVA» (Abb. 7e). Hierfür kopiert er die im 2. Schritt erzeugte Datei **hallo.system.imp.jar** nach «PVA»/mllib, programmiert die Java-Dateien, wie im übernächsten Abschnitt beschrieben und speichert sie an der entsprechenden Stelle.
5. kompiliert mit dem Befehl **javac** im Verzeichnis «PVA» die Applikation **hallo.app** (Abb. 7f). Hierbei enthält der

Modulpfad **MP**: das Verzeichnis **\${FEPCOS\_HOME}/mllib**, welches die Module von FEPCOS-J enthält, und das Verzeichnis **mllib**, das **hallo.system.imp.jar** enthält. Die Ausgabe erfolgt im Verzeichnis **build**.

6. startet im Verzeichnis «PVA» mit dem Befehl **java** die Applikation (Abb. 7g). Der beim Kompilieren festgelegte Modulpfad **MP** ist hierbei um das Verzeichnis **build** erweitert, da sich darin die Applikation befindet. Im Erfolgsfall wird „Hallo JAVAPRO-Leser!“ ausgegeben.



Umsetzung des Beispiels (Abb. 7)

## Programmierung – elementares System

Der nachfolgende Abschnitt beschreibt die Programmierung der Spezifikation von **hallo.system**. Der Anwender programmiert hierfür:

- den Moduldeskriptor **module-info.java** (Listing 1)
- die Systemdeklaration **SYS** mit der Datei **SYS.java** (Listing 2)
- die Aktivitätsspezifikation **Größe** mit der Datei **Größe.java** (Listing 3)

Er speichert die Dateien innerhalb des Verzeichnisses «PVS»/src wie in (Abb. 7a) dargestellt. Der Moduldeskriptor (Listing 1) legt fest, dass **hallo.system** im Modul **hallo.system.spec** spezifiziert wird (FEPCOS-J schreibt die Endung **.spec** vor) und dass hierfür die Annotationen aus dem Modul **fepcos.annotations** benötigt werden.

(Listing 1)

```
module hallo.system.spec {
    requires static fepcos.annotations;
}
```

Die für **SYS** (Listing 2) benötigten Annotationen **@SYDec** und **@Cap** sind im Paket **fepcos.sy** und müssen importiert werden.

**@SYDec** legt fest, dass **SYS** das System deklariert. **@Cap** legt fest, dass das System die Fähigkeit **sagHallo** (Name der Objektvariable) besitzt, die mit der Klasse **Grüße** spezifiziert wird.

#### (Listing 2)

```
package hallo.system.spec;
import fepcos.sy.*;

@SYDec("Ein nettes System.")
class SYS {
    @Cap Grüße sagHallo;
}
```

Die für **Grüße** (Listing 3) benötigten Annotationen **@AYSpec**, **@In**, **@Out** und **@Behavior** sind im Paket **fepcos.ay** und müssen importiert werden. **@AYSpec** legt fest, dass **Grüße** eine **Aktivität** spezifiziert. **@In** legt fest, dass diese eine Eingabe **benutzer** vom Typ **String** erwartet. **@Out** legt fest, dass eine Ausgabe **gruß** vom Typ **String** erzeugt wird. **@Behavior** legt fest, dass die Methode **go()** das gewünschte Verhalten des Systems beschreibt. FEPCOS-J schreibt vor, dass der Name des Pakets, das die Systemspezifikation enthält, gleich dem Namen des Moduls ist. Folglich müssen **SYS** und **Grüße** im Paket **hallo.system.spec** sein.

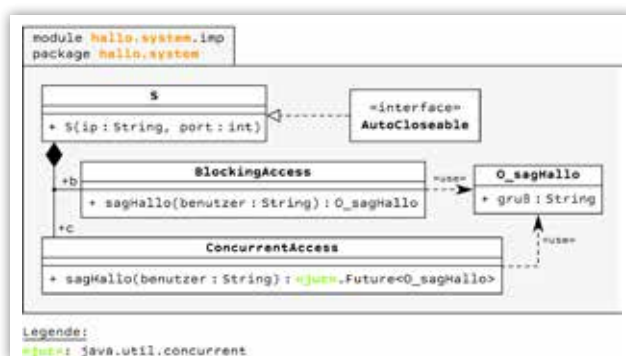
#### (Listing 3)

```
package hallo.system.spec;
import fepcos.ay.*;

@AYSpec("Grüßt den Benutzer.")
class Grüße {
    @In("Zu grüßender Benutzer.") String benutzer;
    @Out("Der Gruß.") String gruß;

    @Behavior
    void go() {
        gruß = "Hallo " + benutzer + "!";
    }
}
```

(Abb. 8) zeigt die von **fjp** erzeugte Systemschnittstelle. **S** befindet sich im Systemimportmodul **hallo.system.imp** im



Die von fjp erzeugte Systemschnittstelle von hallo.system (Abb.8)

Paket **hallo.system** und implementiert **AutoCloseable**. **S** hat einen Konstruktor, dem eine IP-Adresse **ip** und ein Port **port** übergeben werden muss. **S.b** ermöglicht den blockierenden Zugriff (**BlockingAccess**). **S.c** ermöglicht den nebenläufigen Zugriff (**ConcurrentAccess**). **S.b** und **S.c** enthalten, wie mit **@Cap** deklariert, die Methode **sagHallo(...)**. Diese erwartet als Eingabe, wie mit **@In** deklariert, **String benutzer** und gibt als Ausgabe **O\_sagHallo** bzw. **java.util.concurrent.Future<O\_sagHallo>** zurück. **O\_sagHallo** besitzt, wie mit **@Out** deklariert, eine Objektvariable für den Ausgabeparameter **String gruß**.

## Programmierung – Applikation

Für die Applikation programmiert der Anwender:

- den Modulkriptor **module-info.java** (Listing 4)
- die Applikation **Main** mit der Datei **Main.java** (Listing 5).

Er speichert die Dateien innerhalb des Verzeichnisses «PVA»/src wie in (Abb. 7e) dargestellt. Der Modulkriptor (Listing 4) legt fest, dass **hallo.app** das Systemimportmodul **hallo.system.imp** benötigt. Darin befindet sich die von **fjp** generierte Systemschnittstelle **hallo.system.S**.

#### (Listing 4)

```
module hallo.app {
    requires hallo.system.imp;
}
```

Die Klasse **Main** (Listing 5) verwendet **hallo.system.S**. Da **hallo.system.S** das Interface **AutoCloseable** implementiert, wird sie innerhalb eines Try-With-Resource-Statements als Variable **sy** instanziiert, wobei dem Konstruktor die IP-Adresse **10.0.0.2** und der Port **22222** als Parameter übergeben werden. Die Applikation greift mit **sy.b.sagHallo(...)** blockierend über das Netzwerk auf die Fähigkeit **sagHallo** zu (siehe **@Cap** in Listing 2), wobei der Eingabeparameter **benutzer** (siehe **@In** in Listing 3) „JAVAPRO-Leser“ ist. Die Variable **res** enthält das Ergebnis des Aufrufs. Schließlich wird der in **res.gruß** gespeicherte Ausgabeparameter **gruß** (siehe **@Out** in Listing 3) ausgegeben.

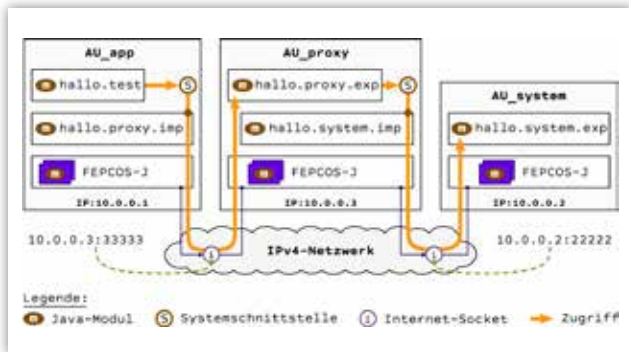
#### (Listing 5)

```
package hallo.app;

public class Main {
    public static void main(String[] args) {
        try(
            var sy = new hallo.system.S("10.0.0.2", 22222);
        ){
            var res = sy.b.sagHallo("JAVAPRO-Leser");
            System.out.println(res.gruß);
        } catch (Exception e) { System.out.println(e); }
    }
}
```

## Erweiterung – zusammengesetztes System

Nachfolgend wird das Beispielszenario um ein zusammengesetztes System **hallo.proxy** und eine Applikation **hallo.test** erweitert (Abb. 9). **hallo.proxy** verwendet **hallo.system** als Teil und kommt in eine weitere Ausführungsumgebung **AU\_proxy**, welche die IP-Adresse **10.0.0.3** hat. **hallo.test** nutzt **hallo.proxy** und kommt in **AU\_app**. **hallo.test** greift über das Internet-Socket **10.0.0.3:33333** auf die Fähigkeiten von **hallo.proxy** zu, welcher selbst über das Internet-Socket **10.0.0.2:22222** auf die Fähigkeiten von **hallo.system** zugreift.



Beispielszenario: Zugriff Applikation auf zusammengesetztes System, auf elementares System (Abb. 9)

Nachfolgend wird die Umsetzung der Erweiterung mit FEPCOS-J skizziert. Die Entwicklung von **hallo.proxy** erfolgt auf **AU\_proxy** im Projektverzeichnis «PVP». Die Entwicklung von **hallo.test** erfolgt auf **AU\_app** im Projektverzeichnis «PVT» (Abb. 10).

Der Anwender

1. exportiert **hallo.system** (Abb. 7d).
2. erstellt «PVP» (Abb. 10a). Hierfür kopiert er **hallo.system.imp.jar** nach «PVP»/mllib. Er programmiert den Moduldeskriptor **module-info.java** (Listing 6), die Systemdeklaration **SYS2.java** (Listing 7) und die Aktivitätsspezifikation **Grüße2.java** (Listing 8).
3. Erzeugt die Systemmodule von **hallo.proxy** (Abb. 10b).
4. exportiert **hallo.proxy** (Abb. 10c).
5. erstellt «PVT» (Abb. 10d). Hierfür kopiert er das neu erstellte Systemimportmodul **hallo.proxy.imp.jar** nach «PVT»/mllib. Er programmiert den Moduldeskriptor **module-info.java** (Listing 9) und die Applikation **Main2.java** (Listing 10).
6. kompiliert **hallo.test** (Abb. 10e).
7. startet die Applikation (Abb. 10f). Im Erfolgsfall wird **+++ Hallo JAVAPRO-Leser!** mit Verzögerungen ausgegeben. Aufgrund der systemübergreifenden Nebenläufigkeit kann die Anzahl der + Zeichen variieren.

Der Quellcode von **hallo.proxy** ist dem von **hallo.system** ähnlich. Von daher werden hier nur die wesentlichen Neuerungen erklärt. Der Moduldeskriptor (Listing 6) legt fest, dass **hallo.system.imp** benötigt wird.

(Listing 6)

```
module hallo.proxy.spec {
    requires static fepcos.annotations;
    requires hallo.system.imp;
}
```

Die Systemdeklaration (Listing 7) enthält die Annotation **@Part**. Diese legt fest, dass **hallo.system** ein Teil von **hallo.proxy** ist. Der Zugriff auf **hallo.system** erfolgt mit der Systemschnittstelle **hallo.system.S** über das Internet-Socket **10.0.0.2:22222**. Der Name des Teils ist **hs**.

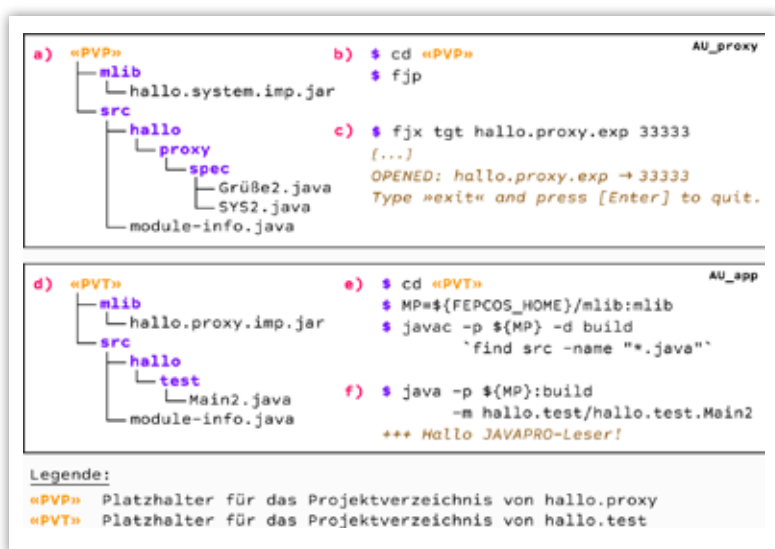
(Listing 7)

```
package hallo.proxy.spec;

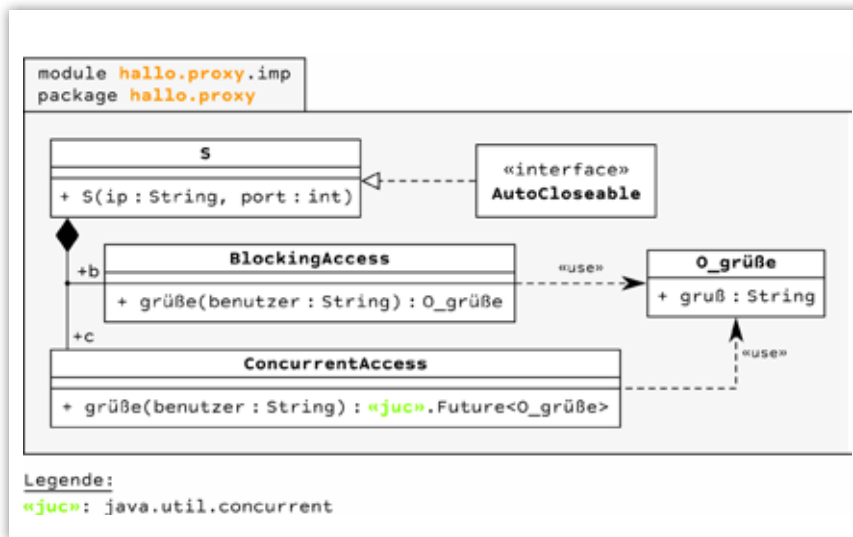
import fepcos.sy.*;

@SYDec("Ein nettes Proxy-System.")
class SYS2 {
    @Part(ip="10.0.0.2", port=22222) hallo.system.S hs;
    @Cap Grüße2 grüße;
}
```

In der Aktivitätsspezifikation (Listing 8) erfolgt mit **s.hs.b.sagHallo(...)** ein blockierender Zugriff auf den Teil **hs**. Hierfür akzeptiert die Methode **go(...)** die Systemdeklaration als Parameter **SYS2 s**. Eventuell auftretende Exceptions werden mit **throws Exception** an FEPCOS-J weitergegeben. Zur Demonstration der Nebenläufigkeit verzögert **Thread.sleep(...)** das Ausführen.



Umsetzung der Erweiterung des Beispiels (Abb. 10)



Die von fjp erzeugte Systemschnittstelle von hallo.proxy (Abb. 11)

(Listing 8)

```

package hallo.proxy.spec;

import fepcos.ay.*;

@Ayspec("Wartet und begrüßt den Benutzer.")
class GrüBe2 {
    @In("Zu grüßender Benutzer.") String benutzer;

    @Out("Der Gruß.") String gruß;

    @Behavior
    void go(SYS2 s) throws Exception {
        Thread.sleep(3000);
        gruß = s.hs.b.sagHallo(benutzer).gruß;
    }
}
    
```

(Abb. 11) zeigt die von fjp erzeugte Systemschnittstelle von hallo.proxy. S befindet sich im Modul hallo.proxy.imp im Paket hallo.proxy. Sowohl S.b, für den blockierenden Zugriff, als auch S.c, für den nebenläufigen Zugriff, enthält die Methode grüße(...) (siehe auch @Cap in Listing 7). Der Ausgabebetyp der Methoden ist O\_grüße, bzw. java.util.concurrent.Future<O\_grüße>.

Der Quellcode von hallo.test ist dem von hallo.app ähnlich. Auch hier werden nur die wesentlichen Neuerungen erklärt. Der Moduledeskriptor (Listing 9) legt fest, dass hallo.proxy.imp benötigt wird.

(Listing 9)

```

module hallo.test {
    requires hallo.proxy.imp;
}
    
```

(Listing 10)

```

package hallo.test;

public class Main2 {
    public static void main(String[] args) {
        try(
            var pr = new hallo.proxy.S("10.0.0.3", 33333);
        ){
            var res = pr.c.grüße("JAVAPRO-Leser");

            while(!res.isDone()) {
                System.out.print("+");
                Thread.sleep(1100);
            }

            System.out.println(" " + res.get().gruß);

        } catch (Exception e) { System.out.println(e); }
    }
}
    
```

### Fazit:

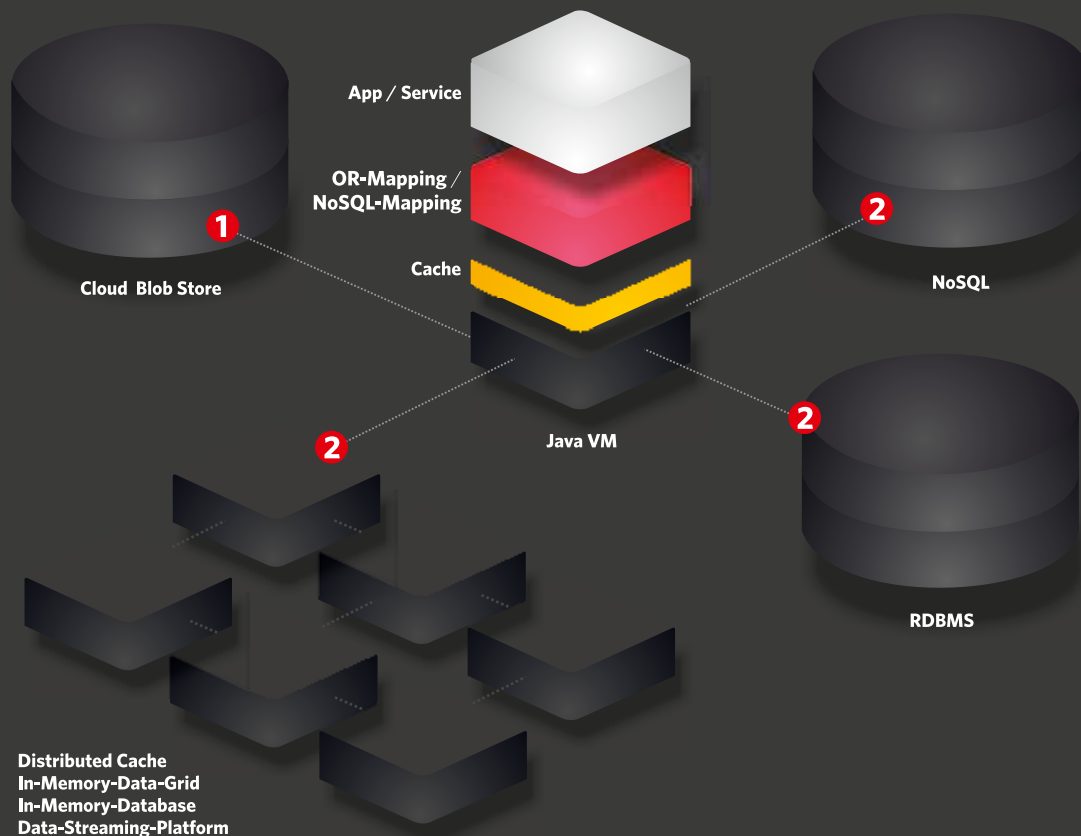
Mit FEPCOS-J sind Anwender in der Lage, Systeme zu programmieren, auf deren Fähigkeiten ein Systemnutzer über ein Netzwerk zugreifen kann. Der Anwender braucht sich um die Netzwerkprogrammierung nicht kümmern. FEPCOS-J generiert für ihn den Code und stellt ihm eine Schnittstelle für blockierende und nebenläufige Zugriffe zur Verfügung.

### Quellen:

- 1 Das FEPCOS-Projekt: <https://bit.ly/fepcos>
- 2 Christian Heitzmann: „Deep-Dive into Annotations – Teil 2“, JAVAPRO 1-2019 <https://bit.ly/JAVAPRO-d2>

# MicroStream Accelerates Database Queries a 1000x

## Standard Database Application Architecture



**1 Mapping - Converting all Data Twice !**

**2 Queries on Remote DBMS Servers, Lot of Network Latencies**

**Query Time: *Milliseconds or Slower !***

Live-Demo: [www.microstream.one](http://www.microstream.one)

Java + **Hibernate** + Oracle DB

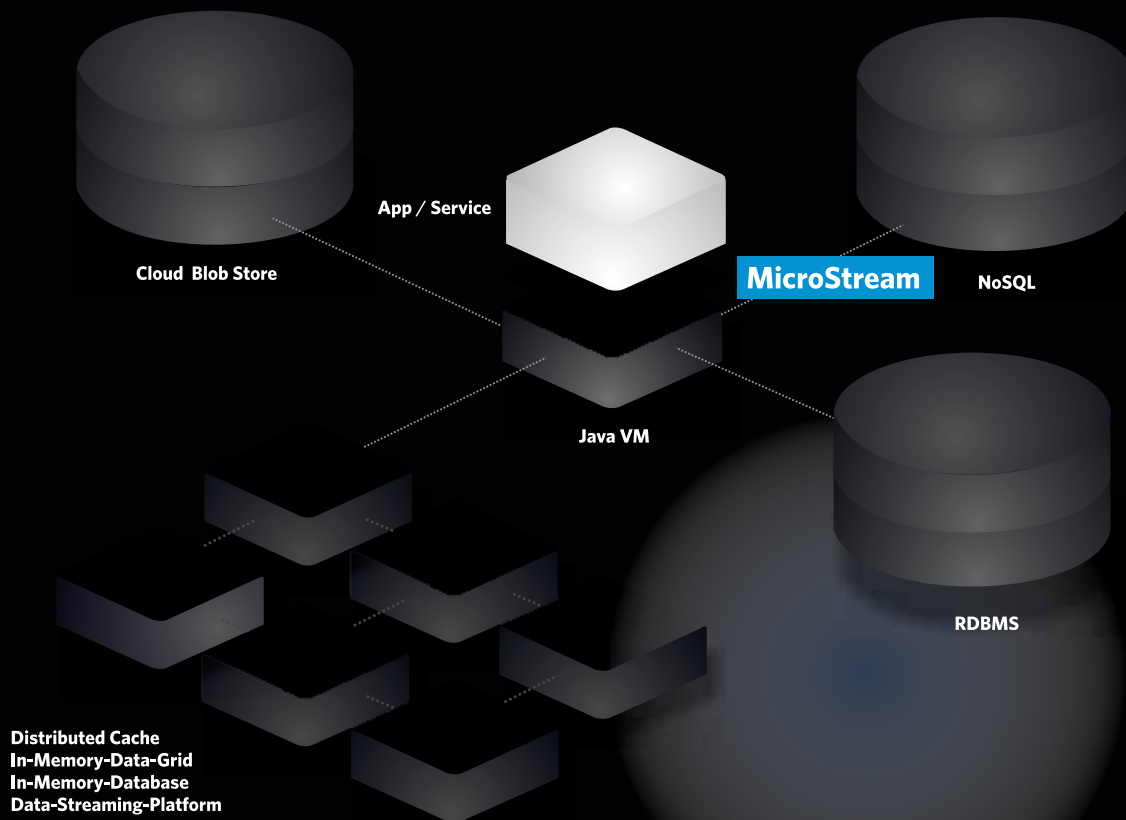
**1065 ms**

Java + **Hibernate** + **EHCache** + Oracle DB

**3 ms**

Java + **MicroStream** + Oracle DB

## MicroStream Java-Native Database Application Architecture



**No more Mapping - Converting Eliminated !**

**Queries on Object Graphs In-Memory, No Latency**

**Query Time: Microseconds - up to 1000x Faster !**

**Much Higher Performance  
Reduce Your Costs of  
Infrastructure. Our customers  
save up to 90% cloud costs.**

Copyright (C) 2020, MicroStream Software GmbH. All rights are reserved.

**4080 ms**

#JCON2020  
[www.jcon.one](http://www.jcon.one)

JAVAPRO



DIE GROSSE JAVA COMMUNITY KONFERENZ  
**27. - 30. Oktober 2020 - ONLINE !**

**Workshop-Day am 30. Oktober 2020**

[www.jcon.one](http://www.jcon.one)

**4**

Days

**5**

Streams parallel

**9**

Special Days

**110+**

Speaker

**150+**

Sessions Live!

**1000+**

Participants

JCON-ONLINE 2020 PARTNER

GOLD PARTNER

ORACLE®

eclipse

MicroStream

XDEV™

SILBER PARTNER

RAPIDclipse™

consol  
Wir unternehmen IT

JFrog

TimoCom.

ORGANISATIONS-PARTNER

JAVAPRO



JUG Oberpfalz