Jetzt kostenlos abbonieren !
Einfach anfordern unter: www.javapro.io

60 DIE CLOUD IST NICHT UNANTASTBAR
69 IOT MESSAGING MIT MQTT 5 UND JAVA

GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

2 / 2019

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

Microservices mit MicroProfile

6 API-DESIGN - DO'S AND DONT'S
11 TASK-PARALLELITÄT MIT COMPLETABLE FUTURE
17 MICROPROFILE FÜR MICROSERVICES MIT JAVA-EE
34 MONOLITHEN MIT DDD AUFSCHNEIDEN
48 CONTAINERBASIERTE TESTAUTOMATISIERUNG
56 DIE DASA DEVOPS PRINZIPIEN
72 DAS AGILE COACHING DOJO

Java • Architektur • Cloud • Agile

JCON 2019
www.jcon.one
24. - 26. September 2019
UCI-Kinowelt in Düsseldorf
Training Day am 23. September
Teilnehmerzahl begrenzt !

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 2019 zu veranstalten.

JCON PARTNER 2019

GOLD PARTNER

ORACLE

SILBER PARTNER

BRONZE PARTNER

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

www.javapro.io/partner

JAVAPRO

Impressum

JAVAPRO

Verlag:
JAVAPRO
Mergenthalerallee 73-75
65760 Eschborn
Telefon: +49 (0) 61 96 - 20 48 010
Telefax: +49 (0) 61 96 - 20 48 019
E-Mail: info@javapro.io
Website: http://www.javapro.io

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

Redaktion:
info@javapro.io

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

Preis: kostenfrei

Illustrationen:
Pixabay (Public Domain)

Erscheinungsweise: Vier mal jährlich

Gründungsjahr: 2017
Copyright (c) 2019
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

In Kürze ist es wieder soweit und unsere jährliche Java-Community-Konferenz JCON startet vom 24. bis 26. September 2019 in ihr drittes Jahr. Am Tag vor der JCON 2019, Montag den 23. September, findet erstmals ein Workshop-Day mit insgesamt 10 hochkarätigen Schulungen statt.

Die JCON ist die einzige Java-Konferenz in Deutschland, die in einem modernen Multiplexkino stattfindet. Die Idee dazu entstand schon vor vielen Jahren, als wir das erste Mal auf der legendären Java-One-Konferenz in San Francisco teilnahmen und die Oracle im vergangenen Jahr unter erheblichem Protest der Java Community in Oracle-Code-One umbenannt wurde. Alles Neue über Java, enorm spannende Praxisbeispiele, neue Frameworks und Trends und vor allem Live-Coding mit den Top-Gurus der Java-Szene – und wir sahen nichts! Mit Ausnahme der Keynote, die traditionell auf einer riesigen Bühne stattfindet, erging es uns bei fast jedem Vortrag so, weil wir es auf Grund der großen Teilnehmerzahl einfach nicht geschafft hatten, Plätze in einer der vorderen Reihen zu ergattern. Bei den eigentlichen Session-Highlights – dem Live-Coding-Teil, der oft ein Drittel bis die Hälfte des gesamten Vortrags ausmacht, konnten wir nur noch zuhören. Denn bereits ab der Mitte des Raums hatte man große Schwierigkeiten, auf den viel zu kleinen Leinwänden noch etwas erkennen zu können. Beim Live-Coding nur zuhören zu können, fanden wir genauso aufregend wie ein Live-Konzert bei dem die Soundanlage ausfällt. Schon damals haben wir beschlossen: Sollten wir jemals selber eine Konferenz veranstalten, dann gehen wir damit in ein Kino!

Mit dem UCI Multiplexkino in Düsseldorf haben wir die perfekte Location gefunden. Die riesigen Leinwände sind prädestiniert für ausführliche Live-Coding-Sessions. Selbst in den letzten Reihen ist Quellcode noch sehr gut lesbar. Beste Voraussetzungen für eine Java-Konferenz!.

Neben der Location ist aber vor allem das Konferenzprogramm ausschlaggebend. In diesem Punkt war die JCON von Anfang an stark. Über 70 hochkarätige Speaker, darunter zahlreiche JAVAPRO Autoren, sorgen dafür, dass die JCON auch in diesem Jahr wieder ein sehr umfangreiches und ausgewogenes Konferenzprogramm, gespickt mit vielen Highlights bieten kann. Ganz besonders freue ich mich auf die Keynotes von Adam Bien, David Delabassée und Uwe Friedrichsen. Am Tag vor der JCON findet heuer erstmals ein Training-Day mit insgesamt 10 Tagesschulungen statt.

Unter #JCON2019 halten wir euch via Twitter auf dem Laufenden!

Also, JAVAPRO Leser, auf geht's zur JCON 2019!

Markus Kett
Chefredakteur
JAVAPRO

Twitter: @MarkusKett
LinkedIn: markuskett

11

CORE JAVA

Task-Parallelität mit CompletableFutures

Von Prof. Jörg Hettel

Java stellt für die Umsetzung einer Parallelisierung schon seit geraumer Zeit verschiedene komfortable Abstraktionen bzw. Frameworks zur Verfügung. Neben parallelen Streams kommt in Projekten die CompletableFuture-Klasse immer noch selbst zum Einsatz, obwohl viele Java-Entwickler diese Klasse kennen. Das mag daran liegen, dass die Klasse auf den ersten Blick komplex wirkt oder, dass die Einsatzmöglichkeiten nicht immer offensichtlich sind. Besitzt man jedoch erst einmal eine gewisse Erfahrung für die Identifikation und Handhabung von Nebenläufigkeiten, lässt sich mit Hilfe von CompletableFutures recht einfach und elegant Task-Parallelität realisieren. In diesem Artikel werden einige Best-Practices vorgestellt und aufgezeigt, wie durch die Verwendung von Task-Parallelität die Reaktivität einer Anwendung deutlich gesteigert wird.

17

ENTERPRISE JAVA

MicroProfile für Microservices mit Java EE

Von Alexander Ryndin

Nach der Meinung vieler Experten ist Java-EE nicht geeignet, die gestiegenen Ansprüche zu erfüllen. Aus diesem Grund wurde das Eclipse-MicroProfile-Projekt ins Leben gerufen. MicroProfile ist, genauso wie Java-EE (jetzt Jakarta-EE) eine Sammlung von Spezifikationen, welche auf Microservice-basierte Softwareentwicklung fokussiert ist. Ziel ist es, einen Standard für die Entwicklung von Microservices auf Basis von Java zu definieren. Die Weiterentwicklung des Projekts schreitet schnell voran und für die Entwicklung von Microservices in Java ist MicroProfile ein großer Schritt nach vorne. Es gibt jedoch auch noch Herausforderungen, die zu meistern sind.

6
CORE JAVA
API-Design – Do’s and Don‘ts
Die Umsetzung von Qualitätsmerkmalen im API-Design und auf welche Fehlerquellen und Schwierigkeiten man achten sollte.

11
CORE JAVA
Task-Parallelität mit CompletableFutures
Überblick von Best-Practices von CompletableFutures-Klassen und die Steigerung der Reaktivität der Anwendung durch Task-Parallelität.

17
ENTERPRISE JAVA
MicroProfile für Microservices mit Java-EE
MicroProfile, eine Sammlung von Spezifikationen zur Erfüllung von Anforderungen und ein möglicher Standard für die Entwicklungen von Microservices in Java

34
ARCHITECTURE
Monolithen mit DDD aufschneiden
Analyse und fachliche Zerlegung von Monolithen mithilfe von Domain-Driven-Design, gezeigt an einem ausführlichen Beispiel.

41
ARCHITECTURE
Die Qual der Wahl
Die langwierige Umsetzung einer digitalen Datenerhebung in Wahllokalen einer Großstadt.

48
CLOUD, CONTAINER & DEVOPS
ContainerbasierteTestautomatisierung
Testautomatisierung und Continuous-Integration sorgen für eine vereinfachte Konfiguration und Steuerbarkeit in verschiedenen Umgebungen.

52
CLOUD, CONTAINER & DEVOPS
Camel-K - LeichtgewichtigeCloud-Integrations-Plattform
Community-Plattform für einfaches Deployment von Camel-Anwendungen in Kubernetes und OpenShift.

56
CLOUD, CONTAINER & DEVOPS
Die DASA-DevOps-Prinzipien
Darstellung der Prinzipien und Zusammenhänge für modernes Kompetenz-Framework hinsichtlich Organisation und Personal.

34
ARCHITECTURE
Monolithen mit DDD aufschneiden

Von Carola Lilienthal
Fast jedes Softwaresystem wird unter schwierigen Bedingungen entwickelt: Knappe Zeitvorgaben, unterschiedliche Qualifikationen im Entwicklungsteam, jede Menge alter Codes die keiner mehr kennt und vieles mehr. All das führt zu dem sogenannten Monolithen - dem Ergebnis von 10 bis 15 Jahren Programmierung. und hohen Wartungskosten. Mit Domain-Driven-Design (DDD) haben wir ein Werkzeug an der Hand, um solche Monolithen Schritt für Schritt zu zerlegen und wieder in den Bereich der beherrschbaren Wartungskosten zu bringen. Mit diesem Artikel erhalten Sie Empfehlungen, mit dem der Umbau eines Monolithen zu einer fachlich orientierten Architektur gelingt.

#JCON2019

DIE GROSSE JAVA COMMUNITY KONFERENZ
www.jcon.one
4 KINOS
3 SPECIAL DAYS
100+ SESSIONS
100+ SPEAKER
800+ TEILNEHMER
Java Konferenz im Multiplex-Kino:
Live-Coding auf riesigen Kino-Leinwänden auch in der letzten Reihe genießen können, bester Sitzkomfort und tolles Ambiente.

60
CLOUD, CONTAINER & DEVOPS
Die Cloud ist nicht unantastbar
Erfahren Sie, welche elf Risiken im Zusammenhang mit Cloud-Computing auftreten können und wie sich diese vermeiden lassen.

63
CLOUD, CONTAINER & DEVOPS
DevOps-Sportfreunde
Der Vergleich zwischen DevOps und Sport:
Die geeignete Taiktik und Ausrüstung sind gefragt.

69
LOT
IoT-Messaging mitMQTT 5 und Java
Die sichere Verwendung von MQTT, dem de-facto Standard-protokoll in der neuesten Version für das Internet-of-Things.

72
AGILE & PROJEKTMANAGEMENT
Das Agile-Coaching-Dojo
Umsetzung der agilen Praxis in einem Trainingsraum für persönliche Weiterentwicklung.

2
MAGAZIN
Partner Network & Impressum

3
MAGAZIN
Editorial

#JAVAPRO #Architecture #API

API-Design – Do's and Don'ts

Im Zeitalter der Modularisierung von Software kommt kein Java-Entwickler daran vorbei, früher oder später ein Application-Programming-Interface (API) zu definieren oder zu erweitern. Der Entwurf eines guten API erfordert keine Kenntnisse in schwarzer Magie. Werden einige wenige Konventionen befolgt, erhöht das die Qualität der Schnittstelle deutlich.

Was sind die Qualitätsmerkmale eines API? Eine Schnittstelle muss zuallererst alle geforderten Funktionen umsetzen. Eine gute Schnittstelle ist darüber hinaus in sich schlüssig, einfach zu verwenden und zeigt kein unerwartetes Verhalten. Im Folgenden zeigen mehrere Beispiele, wie ein API durch wenige Handgriffe verbessert werden kann. Sie orientieren sich an Java und ähnlichem Pseudocode, sind aber im Prinzip auch auf andere Programmiersprachen übertragbar. Der erste und zugleich am einfachsten zu ändernde Qualitätsfaktor ist die Namensgebung einer Schnittstelle.

Autor:

Jochen Kraushaar arbeitet für die BridgingIT GmbH als Software Entwickler und Consultant. Seine Schwerpunkte liegen auf Softwarearchitektur, Qualitätssicherung und Build-Prozesse. In seiner Freizeit beschäftigt er sich mit aktuellen Trends im Java-Umfeld und Künstlicher Intelligenz.

www.bridging-it.de
Twitter: @KraushaarJochen
Email: jochen.kraushaar@bridging-it.de

Nomen est omen

Der Name einer Methode oder Klasse beschreibt ihre Funktion. Wird er falsch gewählt, ist die Schnittstelle im besten Fall schwer zu verstehen. Im schlimmsten Fall wird sie durch falsche Namensgebung missverständlich. Ein Beispiel: Welches Verhalten kann ein Nutzer der Methode `process` in der Klasse `UserProcessor` (Listing 1) erwarten? Offensichtlich wird hier ein Benutzer verarbeitet. Was genau hinter dem Verarbeitungsprozess steckt,

bleibt allerdings unklar. Dem Nutzer bleibt nichts anderes übrig, als einen Blick in den Quellcode der Klasse **UserProcessor** zu werfen. Oder er vertraut auf die Dokumentation. In den allermeisten Fällen wird diese allerdings veraltet, fehlerhaft oder schlicht nicht vorhanden sein. Im Gegensatz dazu drückt die Methode **checkValidityAndUpdateStatus** in der **UserStatus Checker** Schnittstelle deutlich aus, was ein Entwickler von ihr erwarten kann.

(Listing 1)

```
// Falsch
UserProcessor.process(User user)

// Richtig
UserStatusChecker.checkValidityAndUpdateStatus(User user)
```

Führen mehrere Methoden vergleichbare Prozesse aus, nutzen sie auch das gleiche Verb (Listing 2). So kann ein Entwickler von dem Verhalten einer Methode auf das Verhalten der anderen Methoden schließen. Gleichzeitig unterstützt eine solche Namensgebung die Auto-Complete-Funktion moderner Entwicklungsumgebungen. Tippt der Entwickler **load** in den Editor, bekommt er bereits **loadUsers()**, **loadAccounts()** und **loadAddresses()** vorgeschlagen.

(Listing 2)

```
// Falsch
loadUsers()
fetchAccounts()
getAddresses()

// Richtig
loadUsers()
loadAccounts()
loadAddresses()
```

Wenn es um Namen geht, kommt im deutschsprachigen Raum häufig ein weiteres Problem hinzu: Einige Unternehmen schreiben in ihren Entwicklungsrichtlinien vor, dass als Entwicklungssprache Deutsch zu verwenden ist. Die Softwareentwicklung ist allerdings durch den englischsprachigen Raum geprägt. Entsprechend sind Programmiersprachen und Bibliotheken in Englisch gehalten. Auch sind viele Entwurfsmuster vor allem mit ihren englischen Namen bekannt. Ein Singleton-, Visitor- oder Repository-Pattern klingt in seiner deutschen Übersetzung als Einzelstück-, Besucher- und Aufbewahrungsort-Muster falsch. Die Vermischung deutscher Begriffe mit englischen führt dabei bisweilen zu Stilblüten (Listing 3).

(Listing 3)

```
// Falsch
Repository.rufeAddressesAb()

// (Fast) richtig
Benutzer.setzeAlter()
```

Der Quellcode ist so schwieriger zu lesen. In einigen Fällen führt die Verwendung der deutschen Sprache in Quellcode sogar zu Fehlern. Beispielsweise legt das Framework Java-Server-Faces (JSF) fest, dass Eigenschaften von Beans über Setter- bzw. Getter-Methoden verfügen. Heißt eine Methode **Benutzer.setzeAlter(…)**, wird die Eigenschaft Alter im zugehörigen XHTML-Dokument als **zeAlter** referenziert. Der entsprechende Getter muss dann **getZeAlter()** heißen. Dieses Beispiel zeigt deutlich, wie die Vermischung verschiedener Sprachen im Quellcode die Wartbarkeit verschlechtert.

Allerdings gibt es Ausnahmen hierzu: nämlich deutsche Begriffe, für die es keine geeignete englische Übersetzung gibt. In einer Kalenderanwendung ist es vollkommen in Ordnung, die Methode zum Laden von Brückentagen loadBrueckentage() zu nennen. Auch wenn die Benennung als loadBridgeDay() bei den Kollegen sicherlich ein Lächeln hervorruft. Eine eindeutige und konsistente Namensgebung erleichtert Nutzern die Verwendung des API. Um sie auch noch resistenter gegen Fehler zu machen, hilft es auf Null-Referenzen zu verzichten.

Fehler vorbeugen: Die Verwendung von null

Der am häufigsten auftretende Fehler in Java-Anwendungen ist die NullPointerException. Beim Einsatz von null in Schnittstellen ist daher große Vorsicht geboten. Entwickler verwenden null in Übergabeparametern gerne, um optionale Parameter zu ermöglichen (Listing 4). Methoden mit vielen Parametern sind beim Aufruf im Client schwer zu verstehen. Die absichtliche Verwendung von Null-Referenzen verschlechtert die Lesbarkeit weiter. Gleichzeitig führt dieses Entwicklungsmuster auf Seite des Schnittstellen-Bereitstellers zu Methoden mit vielen bedingten Anweisungen (Listing 5).

Besser ist der Einsatz von dedizierten Methoden ohne optionale Parameter. Ist die Anzahl der Kombinationsmöglichkeiten zu groß, können die Parameter auch in ein Hilfsobjekt verpackt werden. Das Builder-Pattern adressiert optionale Parameter bei der Erzeugung von Objekten. Es eignet sich in diesem Fall hervorragend für die Erzeugung von Hilfsobjekten.

(Listing 4)

```
// Falsch
UserRepository.find(„Max“, null, null, Status.ACTIVE, null)

// Richtig
UserRepository.findByNameAndStatus(„Max“, Status.ACTIVE)

// Alternativ
UserRepository.findBy(userSearchRequest)
```

(Listing 5)

```
public SearchResult find(String name, String email, String custo-
merNumber, Status status, Integer age) {
    if (name != null) {
        // Anweisungen
    }
    if (email != null) {
        // Anweisungen
    }
    if (customerNumber != null) {
        // Anweisungen
    }
    if (status != null) {
        // Anweisungen
    }
    if (age != null) {
        // Anweisungen
    }
    // Suche
    return result;
}
```

Wesentlich problematischer ist der Einsatz von **null** jedoch als Ergebnis einer Methode. Entwickler vergessen häufig auf **null** zu prüfen. Die NullPointerException ist damit (im wahrsten Sinne des Wortes) vorprogrammiert. Glücklicherweise bietet Java für optionale Rückgabewerte seit Version 8 die Klasse **Optional** an. Sie signalisiert dem Aufrufer, dass ein leeres Ergebnis Teil der möglichen Ergebnismenge ist. Die Schnittstelle nimmt den Entwickler dadurch an die Hand.

Schlechter Stil ist es hingegen, **Optional** in Übergabeparametern zu verwenden. Ein solcher Parameter kann gleich drei Zustände annehmen: **null**, ein leeres **Optional** oder ein **Optional** mit Wert. Das verdoppelt die Aufwände zur Überprüfung des Parameters in der Methode: Statt **parameter != null** muss hier (**parameter != null**) && (**parameter.isPresent()**) geprüft werden.

Erfahrene API-Designer verwenden **null** in APIs nur dann, wenn es absolut notwendig ist. Viele moderne Programmiersprachen adressieren Null-Referenzen über Sprachkonstrukte: Beispielsweise geht Kotlin davon aus, dass Variablen immer gesetzt werden. Soll eine Variable nullable sein, muss dies explizit über die Angabe eines Fragezeichens bei der Variablendefinition angekündigt werden. Das ermöglicht schon zur Kompilierzeit den Code auf mögliche NullPointerExceptions zu prüfen.

Durch den sparsamen Einsatz von **null** schützt ein API seine Nutzer vor Laufzeitfehlern. Es wird dadurch schwieriger, die Schnittstelle falsch zu benutzen. Darauf zielt auch der nächste Tipp ab: Die Wiederentdeckung des Geheimnisprinzips.

Geheimes geheim halten

Entwicklungsumgebungen nehmen Entwicklern heutzutage viel Arbeit ab. Eine Klasse erzeugen, ein paar Parameter hinzufügen und anschließend Getter und Setter automatisch hinzufügen lassen: Mit wenigen Klicks ist die Klasse fertiggestellt. Dabei machen Entwickler schnell mehr Informationen zugänglich, als sie eigentlich wünschen (Listing 6).

Das Geheimnisprinzip besagt, dass der Zustand eines Objekts vor dem Zugriff von außen geschützt ist. Im Studium oder während der Ausbildung lernen viele Informatiker, dass Datenkapselung über Getter und Setter und private Objektvariablen erreicht wird. Das ist grundsätzlich richtig, allerdings nur der erste Schritt in die Welt des Schnittstellenentwurfs. Robert „Uncle Bob" C. Martin unterscheidet in seinem Buch „Clean Code" beim Thema Geheimnisprinzip zwischen Datenstrukturen und Objekten . Datenstrukturen dienen zum Transport von Daten. Der freie Zugriff auf ihre Objektvariablen ist gewünscht. Entsprechend können Getter und Setter entfallen. Objekte hingegen müssen ihren Zustand schützen. Sie implementieren einen Teil der Geschäftslogik.

In dem verwendeten Beispiel hängt vermutlich ein Prozess am Ändern des Passworts (Listing 7). Statt das Passwort über Getter und Setter zugänglich zu machen, definiert die Klasse explizite Methoden zum Ändern und Kontrollieren des Passworts. Objekte dieser Klassen schützen dadurch ihren Zustand.

(Listing 6)

```
// Falsch
public class Account {
    private String password;

    public void setPassword(String password) {
        this.password = password;
    }

    public String getPassword() {
        return this.password;
    }
}
```

(Listing 7)

```
// Richtig
public class Account {
    private String password;

    public void changePassword(String oldPassword, String
    newPassword) {
        if (validate(oldPassword)) {
            this.password = newPassword;
        }
    }

    public boolean validate(String password) {
        return this.password.equals(password);
    }
}
```

Das Geheimnisprinzip betrifft allerdings nicht nur den Zustand von Objekten, sondern auch seine Methoden (Listing 8). In diesem Beispiel kann ein Aufrufer beliebig häufig **increase**

FailedValidations() aufrufen und damit den Zustand des Accounts manipulieren. Richtigerweise muss die Methode an dieser Stelle mit dem Schlüsselwort **private** versehen werden.

In der Praxis machen Entwickler Private-Methoden häufig von außen zugreifbar, um sie in Modultests besser testen zu können. Komplexe Private-Methoden deuten dabei sehr häufig auf die Verletzung des Single-Responsibility-Prinzips hin. Statt das Geheimnisprinzip für Tests zu verletzen ist es besser, die Architektur des Codes zu überdenken. Kann die zu testende private Methode zum Beispiel in einen eigenen Dienst ausgelagert werden? Wenn ja, macht es Sinn die Zeit in das Refactoring zu investieren.

(Listing 8)

```
public class Account {
    // anderer Code

    public boolean validate(String password) {
        if (this.password.equals(password)) {
            return true;
        } else {
            increaseFailedValidations();
            return false;
        }
    }

    // Falsch
    public void increaseFailedValidations() {
        this.failedValidations++;
    }
}
```

Während bei Datenstrukturen das Geheimnisprinzip verletzt werden kann, müssen API-Designer die Schnittstelle einer Klasse immer bewusst gestalten. Die bereitgestellten Methoden unterstützen die Geschäftslogik der Anwendung und verhindern eine beliebige Manipulation des Zustands eines Objekts.

Die bisherigen Tipps dienten vor allem dazu, Fehler zu vermeiden. Wie aber sind Fehler zu behandeln, wenn sie doch einmal auftreten?

Streitpunkt Exceptions

Über kaum ein Thema im API-Design lässt sich so vortrefflich streiten, wie über den korrekten Einsatz von Exceptions. Java unterscheidet zwischen den Checked- und Unchecked-Exceptions. In den ersten Versionen der Sprache kamen Checked-Exceptions immer dann zum Einsatz, wenn ein Fehler zu erwarten war, beispielsweise beim Lesen einer Datei oder dem Setzen eines Zeichensatzes. Unchecked-Exceptions hingegen deuteten immer auf Programmierfehler hin: Die **IllegalArgumentException** beim Verletzen der Schnittstellendefinition einer Methode oder die **NullPointerException** bei der fehlenden Prüfung auf Null-Referenzen.

Mit den Jahren hat sich das Verständnis von Exceptions allerdings gewandelt. Die Erfahrung aus unzähligen Projekten zeigt, dass Checked-Exceptions häufig einfach an den Aufrufer weitergegeben werden. In seltenen Fällen kann eine Anwendung auf eine Checked-Exception angemessen reagieren und den Programmablauf wie vorgesehen fortsetzen. Gleichzeitig führen Checked-Exceptions zu sehr viel Boilerplate-Code (Listing 9). Der Nutzer einer Schnittstelle muss alle definierten Checked-Exceptions behandeln, selbst wenn er sie nur weiterreicht. Hinzu kommt, dass es sich bei den Exceptions im Beispiel um technische Ausnahmen handelt. Sie geben Informationen zur Implementierung der Schnittstelle preis: Offenbar befindet sich hinter diesem API ein Datenbankzugriff mit einem XML-Parser. Ändert sich die Implementierung, zum Beispiel durch die Anbindung einer JSON-HTTP-Schnittstelle, muss zwangsweise auch das API angepasst werden. Nur sehr selten ist diese Änderung abwärtskompatibel.

Der erfahrene API-Designer kapselt die technischen Ausnahmen in einer domänenspezifischen fachlichen Exception. Das entkoppelt die Schnittstelle von ihrer Implementierung. Zugleich muss der Nutzer statt drei Ausnahmen nur eine einzige behandeln.

(Listing 9)

```
// Falsch
Address findAddress(User user) throws SQLException, Parse
Exception, SAXException;

// Besser
Address findAddress(User user) throws UserRepositoryException;
```

Angesichts der Nachteile von Checked-Exceptions wird in den letzten Jahren zunehmend auf ihren Einsatz verzichtet. Stattdessen kommen Unchecked-Exceptions verstärkt zum Einsatz. Die Vorteile liegen auf der Hand: Der Nutzer einer Bibliothek kann in den seltensten Fällen angemessen auf eine Ausnahme reagieren. Bei dem Einsatz von Unchecked-Exception wird er nicht, wie bei den Checked-Exceptions, zu einer Behandlung gezwungen. Stattdessen ist es die Aufgabe des Anwendungs-Frameworks diese Fehler zentral zu behandeln und entsprechende Maßnahmen zu ergreifen. Dieses Muster spiegelt sich in jüngeren Java-APIs wieder. Beispielsweise können Lambda-Ausdrücke und Streams nur mit Unchecked-Exceptions umgehen. Andere Programmiersprachen, auch hier sei als Beispiel Kotlin genannt, unterstützen nur noch Unchecked-Exceptions.

Beim Einsatz von Unchecked-Exceptions in APIs ist allerdings zu beachten, dass nicht jede Anwendung ein Framework einsetzt. Außerdem sind immer noch einige Fälle denkbar, in denen eine Anwendung tatsächlich auf einen Fehler angemessen reagieren kann. Hierfür ist es unerlässlich, dass die Existenz der Unchecked-Exception dokumentiert wird. Die Wahl, ob und wie auf eine solche Ausnahme reagiert werden soll, liegt dann bei den Entwicklern.

Vererbung designen

Software-Architekten raten für gewöhnlich von der Verwendung von Vererbung ab. Durch Vererbung wird Code komplexer, da eine Klasse zusätzlich zu den Abhängigkeiten zu anderen Klassen auch noch Abhängigkeiten zu Eltern- oder Kindklassen aufbaut. Das gilt vor allem auch für APIs **(Listing 10)**. Im Beispiel wird in einer Schnittstelle **(ApiClass)** eine Methode **doSomething()** bereitgestellt. Ein Nutzer der Schnittstelle möchte die Funktionalität der ursprünglichen Klasse erweitern. Er legt dazu die Klasse **ClientClass** mit der Methode **doSomethingElse()** an und erbt von dem API. Dies geht so lange gut, bis der API-Hersteller selbst auf die Idee kommt **doSomethingElse()** hinzuzufügen **(Listing 11)**. Da die **doSomethingElse()** Methode in **Client Class** überschrieben wird, verändert sich dadurch nun für sie auch gleichzeitig das Verhalten der **doSomething()** Methode: Beide führen ab sofort **doA()** aus. Das Hinzufügen einer Methode zu einer Schnittstelle ist für ihre Nutzer fast immer eine abwärtskompatible Änderung. Anwendungsentwickler übersehen beim Aktualisieren der Bibliotheken daher eine solche Änderung schnell.

(Listing 10)

```
// Falsch
public class ApiClass {
    public void doSomething() {
        doA();
    }
}

public class ClientClass extends ApiClass {
    public void doSomethingElse() {
        doB();
    }
}
```

(Listing 11)

```
// Falsch
public class ApiClass {
    public void doSomething() {
        doSomethingElse();
    }

    protected void doSomethingElse() {
        doA();
    }
}
```

(Listing 12)

```
// Richtig
public final class ApiClass {
    public void doSomething() {
        doSomethingElse();
    }

    void doSomethingElse() {
        doA();
    }
}
```

Auch wenn dieses Beispiel zugegebenermaßen sehr konstruiert ist, verdeutlicht es dennoch das Problem der sogenannten Open-Inheritance. Bei diesem Prinzip geht es darum, dass jede Klasse zunächst offen für Vererbung ist. Soll Vererbung verboten werden, müssen Entwickler die Klasse oder Methoden mit dem Schlüsselwort **final** kennzeichnen **(Listing 12)**.

Das Gegenteil von Open-Inheritance ist Designed-Inheritance. Hier wird Vererbung nur dort erlaubt, wo sie Teil der Schnittstelle ist. Entwickler müssen hier jene Codestellen kennzeichnen, die für die Vererbung freigegeben werden. Ein Beispiel sind Standardimplementierungen von umfangreichen Interfaces, von denen für gewöhnlich nur einzelne Methoden überschrieben werden. Statt das komplette Interface zu implementieren, können Entwickler ihre Klassen von der Standardimplementierung erben lassen. Ein Schnittstellen-Designer hat somit die Möglichkeit ganz genau zu bestimmen, an welchen Stellen Vererbung zum Einsatz kommen kann. Entsprechend kann er bei Aktualisierungen der Schnittstelle darauf achten, dass es nicht zu Problemen kommt. Bei Designed-Inheritance handelt es sich ebenfalls um eine Best-Practice, die von modernen Programmiersprachen bereits umgesetzt ist.

Fazit

Ein API sollte nicht nur vollständig im Sinne der Anforderungen sein, sondern auch möglichst einfach zu bedienen. Das wird vor allem durch eine konsistente Benamung und der Vermeidung von Null-Referenzen erreicht. Durch den konsequenten Einsatz des Geheimnisprinzips und dem gezielten Entwurf der Vererbung kann ein Schnittstellen-Designer Fehlern bei der Verwendung des API zuvorkommen. Der richtige Einsatz von Exceptions gibt den Nutzern der API schließlich die Möglichkeit auf Fehler zu reagieren, ohne ihnen eine Behandlung aufzuzwingen. Die vorliegende Betrachtung des Schnittstellen-Entwurfs ist keinesfalls komplett. Weitere Themen sind beispielsweise die Versionierung von APIs und der Einsatz von optionalen Parametern beim Erzeugen von Objekten.

Quellen:

1 Häufigkeit von NullPointerExceptions: https://bit.ly/2FB8IQh
2 Eric Freeman & Elisabeth Freeman: Head First Design Patterns
3 Robert C. Martin: Clean Code
4 Buchtipp: API-Design von Kai Spichale

#JAVAPRO #Concurrency #CompletableFutures

Task-Parallelität mit CompletableFutures

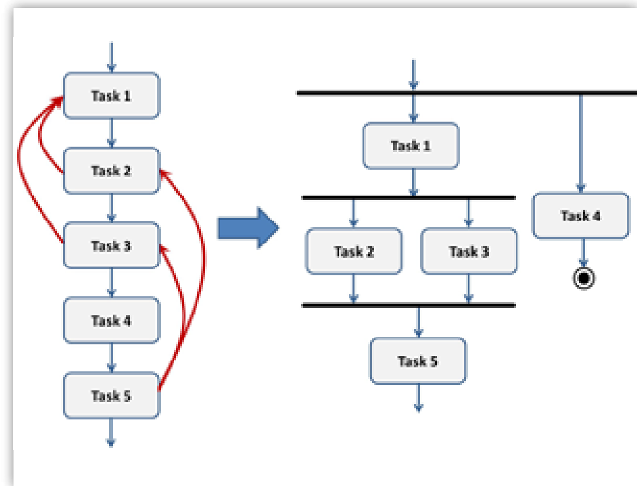
Obwohl viele Java-Entwickler die CompletableFuture-Klasse kennen, wird sie in Projekten immer noch selten eingesetzt. Das mag daran liegen, dass die Klasse auf den ersten Blick komplex wirkt oder dass die Einsatzmöglichkeiten nicht immer offensichtlich sind. Dabei lässt sich mit Hilfe von CompletableFutures recht einfach Task-Parallelität realisieren. Im Folgenden werden einige Best-Practices vorgestellt und gezeigt, wie durch die Verwendung von Task-Parallelität die Reaktivität einer Anwendung deutlich gesteigert wird.

Java bietet schon seit geraumer Zeit sehr viele komfortable Abstraktionen für den Umgang mit Nebenläufigkeit bzw. Parallelität. Nebenläufigkeit existiert immer dann, wenn zwei Aktionen (Tasks) keine direkten Abhängigkeiten besitzen und somit die Ausführungsreihenfolge keine Rolle spielt. Ein Task repräsentiert hier typischerweise eine Zusammenfassung von zusammengehörigen Anweisungen oder Methodenaufrufen. Die Identifikation von Nebenläufigkeit ist die Basis für den Einsatz von Parallelität. Auf Multicore- oder Multiprozessor-Maschinen

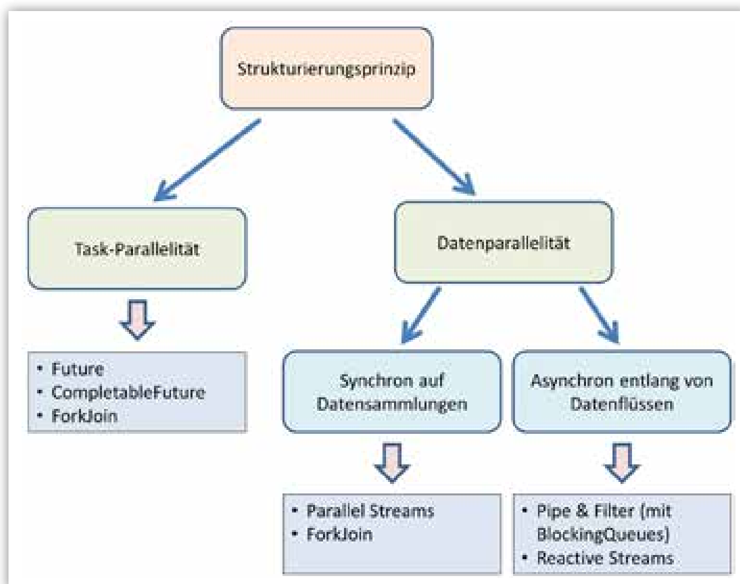
Autor:

Prof. Jörg Hettel war als Berater bei nationalen und internationalen Unternehmen tätig. Er begleitete zahlreiche Firmen bei der Einführung von objektorientierten Technologien und übernahm als Softwarearchitekt Projektverantwortung. Seit 2003 ist er Professor an der Hochschule Kaiserslautern am Standort Zweibrücken.

können nebenläufige Tasks dann auch wirklich parallel (zeitgleich) ausgeführt werden. Ein Task sollte allerdings immer genügend Arbeit haben, damit der mit einer Parallelisierung verbundene Overhead nicht ins Gewicht fällt. Ein weiterer wichtiger Punkt ist die Koordinierung von Wettbewerbssituationen (Race-Conditions), falls Tasks doch nicht ganz unabhängig sind und z.B. auf gemeinsam benutzte Daten gleichzeitig schreibend und lesend zugreifen. Bezüglich der Strukturierung für Nebenläufigkeit unterscheidet man zwischen Daten- und Task-Parallelität (Abb. 1). Die Datenparallelität lässt sich noch weiter in eine synchrone (blockierende) und in eine asynchrone Verarbeitung unterteilen. Bei einer synchronen Verarbeitung wird typischerweise die Datensammlung partitioniert und die Teilbereiche einzelnen Tasks zur weiteren Verarbeitung zugewiesen. Jeder Task führt dabei dieselben Aktionen auf den Daten durch. Datenparallelität wird heute bei Java gewöhnlich mit parallelen Streams realisiert.



Strukturierung nebenläufiger Tasks aufgrund von vorhandenen Abhängigkeiten, die durch rote Pfeile dargestellt sind. (Abb. 2)



Parallelisierungsmöglichkeiten. (Abb. 1)

Bei Task-Parallelität besitzt jeder Task eine eigene Aufgabe. Oft ist es so, dass Tasks kausal miteinander verkettet sind und somit ganze Prozessabläufe realisieren. Als Analogon kann man sich beispielsweise einen Projektplan für einen Hausbau vorstellen. Hier gibt es als Aktivitäten die Erstellung des Rohbaus, die Verlegung der elektrischen Leitungen, das Einsetzen der Fenster oder die Gestaltung der Außenanlage, etc. Einige dieser Aktivitäten besitzen Abhängigkeiten, wie z.B. die Verlegung der elektrischen Leitungen, die erst erfolgen kann, wenn der Rohbau fertig ist. Andere Aktivitäten können dagegen parallel ausgeführt werden, wie z.B. die Verlegung der elektrischen Leitungen und die Gestaltung der Außenanlage.

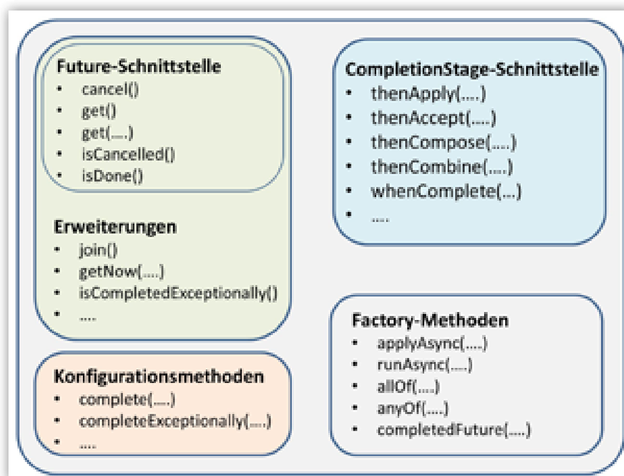
Im Gegensatz zur Datenparallelität ist eine Task-Parallelität bei der Softwareentwicklung nicht immer direkt offensichtlich identifizierbar. Der Hauptgrund dafür ist, dass bei der herkömmlichen Programmierung der sequentielle Programmcode oft dem zeitlichen Programmablauf entspricht. Wird davon abgewichen, wie etwa beim Observer-Pattern, bei dem der lineare Programmcode nicht mehr die zeitliche Ausführung wiederspiegelt (Callback-Methoden werden zu einem späteren Zeitpunkt aufgerufen), wird der Programmablauf schwerer nachvollziehbar. Um diesen Verständnishürden entgegen zu wirken, werden neue Abstraktionen in die Programmiersprachen eingeführt. So stellt die reaktive Programmierung den Datenfluss als Design-Prinzip in den Fokus (asynchrone Datenparallelität). Die Strukturierung des Programmcodes richtet sich hier nach den Datenflüssen und nicht mehr nach der zeitlichen Abarbeitungsfolge (siehe hierzu die Artikel in JAVAPRO 1/2018[1] und 2/2018[2]).

Für die Realisierung von Task-Parallelität stehen Tasks und deren Nebenläufigkeitseigenschaft im Vordergrund. Oft ist eine lineare Task-Anordnung Ausgangspunkt für die Analyse. Die Tasks werden auf Abhängigkeiten untersucht und anschließend umstrukturiert. Die parallelen Abläufe werden oft mit UML-Aktivitätsdiagrammen dargestellt (Abb. 2). Mit Hilfe der CompletableFuture-Klasse lassen sich die parallelen Ablaufstrukturen dann einfach im Programmcode abbilden. Für die meisten Fälle reichen wenige Basis-Idiome (Code-Pattern-Vorlagen) aus, von denen im nächsten Abschnitt einige vorgestellt werden.

Die CompletableFuture-Klasse

Die CompletableFuture-Klasse, die zwei Interfaces implementiert, erscheint auf den ersten Blick komplex, da sie eine Vielzahl von Methoden besitzt. Um die Übersicht zu behalten, lassen sich die Methoden in verschiedene Kategorien einteilen (Abb. 3). Es existieren Methoden für die Erzeugung, für die Verkettung und für Statusabfragen. Die große Anzahl der Methoden rührt daher, dass die Verkettungsmethoden aus dem Interface Completion Stage immer in drei Versionen (Überladungen) vorhanden sind.

Diese Methoden dienen dazu komplexe Task-Abläufe zu realisieren, wobei hierbei wahlweise die Tasks synchron oder asynchron ausgeführt werden können. Bei einer synchronen Ausführung wird der Task vom Vorgänger-Thread oder von dem die Verknüpfung erzeugenden Thread ausgeführt. Bei einer asynchronen Ausführung wird der Task garantiert immer von einem separaten Thread verarbeitet. Bei der asynchronen Variante kann deshalb noch explizit ein Executor (Thread-Pool) angegeben werden, falls nicht der Standard-Pool benutzt werden soll.



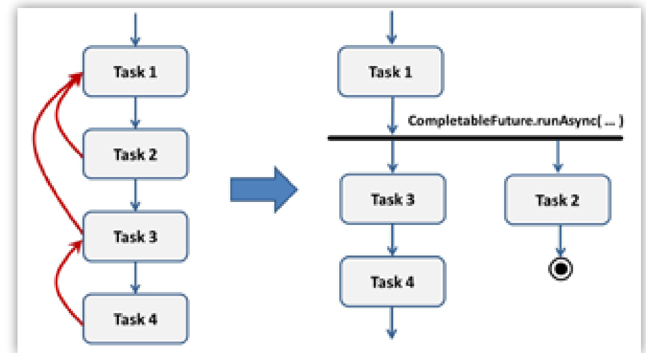
API-Kategorien der CompletabeFuture-Klasse. (Abb. 3)

Am einfachsten lassen sich CompletableFuture-Instanzen über eine der angebotenen Fabrikmethoden erzeugen. Asynchrone Tasks mit Rückgabe werden mit supplyAsync und Tasks ohne Rückgabe mit runAsync gestartet. (Listing 1) zeigt ein Code-Pattern für das Starten eines asynchronen Tasks ohne Rückgabe. In (Abb. 4) ist der Ablauf schematisch abgebildet. Da von Task 2 keine weiteren Tasks abhängen (keine eingehenden roten Pfeile), kann dieser asynchron ausgeführt werden. Bei einer asynchronen Verarbeitung sollten immer zwei Punkte berücksichtigt werden. Das sind ein adäquates Fehlerhandling und das Arbeiten mit Timeouts. Da die Ausführung des Tasks in einem separaten Thread erfolgt, werden auftretende Exceptions nicht automatisch an den Erzeuger gemeldet. Weiter sollte ein asynchroner Aufruf nicht beliebig lange andauern, damit Ressourcen nicht unnötig blockiert werden. In dem Codebeispiel wird deshalb ein Timeout gesetzt und eine Methode für die Behandlung auftretender Fehler angegeben.

(Listing 1)

```
CompletableFuture.runAsync( () -> doTask() )
                 .orTimeout(3 , TimeUnit.SECONDS)
                 .exceptionally( exce -> handleException(exce) );
```

Soll ein Task verzögert ausgeführt werden, wird dies durch die Angabe eines entsprechenden delayedExecutor erreicht. (Listing 2) zeigt dazu ein Beispiel, wobei hier nach einem abgelaufenen Timeout ein Default-Wert zurückgeliefert wird.



Fire-and-Forget-Pattern. Auslagern eines Tasks aus einer sequentiellen Anordnung. (Abb. 4)

(Listing 2)

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(
            () -> getValue(),
            CompletableFuture.delayedExecutor(500,TimeUnit.
            MILLISECONDS) )
       .completeOnTimeout("NONE", 3, TimeUnit.SECONDS);

// ...

String value = future.join();
```

Bei einem ausgelösten Timeout wird der gerade ausgeführte Task in der Regel nicht gestoppt. Dies sollte man bei langlaufenden Tasks beachten und ggf. entsprechende Vorkehrungen treffen. Besteht ein Task aus mehreren Teilaufgaben, können entweder die Teilaufgaben als separate Tasks verketten oder mit einer Kontrollvariable getrennt werden, wie dies in (Listing 3) gezeigt ist. Wird hier während der Ausführung von **doPart1** ein Timeout ausgelöst, wird diese Methode in der Regel noch zu Ende laufen. Alle folgenden Aktionen, insbesondere die Methoden **doPart2** und **doPart3** werden aber nicht mehr ausgeführt. Für die Abbruchsteuerung wird hier ein **AtomicBoolean** verwendet. Dadurch wird sichergestellt, dass eine aktuelle Wertänderung von **isCancelled** auch innerhalb des ersten Tasks, d.h. zwischen den Aufrufen der doPart-Methoden sichtbar ist.

(Listing 3)

```
AtomicBoolean isCancelled = new AtomicBoolean(false);
CompletableFuture.runAsync(() -> {
         doPart1();
         if (isCancelled.get())
            return;
         doPart2();
         if (isCancelled.get())
            return;
         doPart3();
        })
    .thenRunAsync(() -> nextTask() )
    .orTimeout(800, TimeUnit.MILLISECONDS)
    .exceptionally(exce -> {
        isCancelled.set(true);
        return handleException(exce);
    });
```

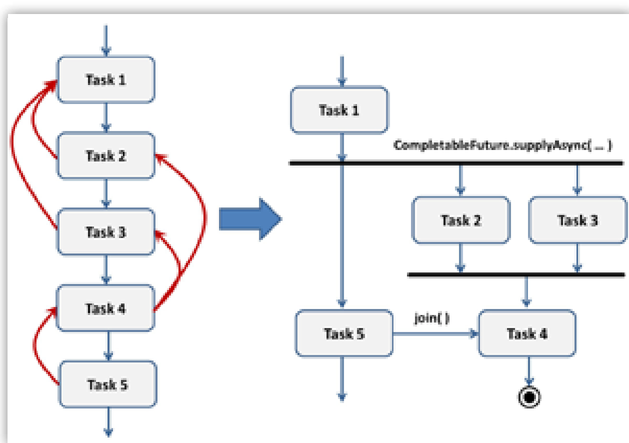
Die eigentliche Stärke von CompletableFutures zeigt sich durch die verschiedenen Möglichkeiten Tasks zu verknüpfen. So können z.B. mehrere Tasks sehr einfach sequentiell, je nachdem ob Daten an den Nachfolger weiter gegeben werden oder nicht, mit **thenApplyAsync** oder **thenRunAsync** verkettet werden. Oder es kann ein Task aufgespaltet und die Ergebnisse wieder durch ein logisches UND oder ODER zusammengeführt werden. **(Listing 4)** zeigt ein Code-Pattern für das Aufspalten und Zusammenführen von parallelen Tasks. In **(Abb. 5)** ist der zugehörige Ablauf schematisch dargestellt.

(Listing 4)

```
String artikel = getArtikel();

CompletableFuture<Double> futurePreis  =
                CompletableFuture.supplyAsync( () -> getPreis(
                artikel) );

CompletableFuture<Double> futureRabatt =
                CompletableFuture.supplyAsync( () ->
                getRabatt(artikel) );


CompletableFuture<Double> futureAktuellerPreis =
    futurePreis.thenCombine( futureRabatt,
                        (preis, rabatt) -> preis*(1.0 -
                         rabatt));

// ...

double preis = futureAktuellerPreis.join();
```

Die Zusammenführung der beiden Tasks **futurePreis** und **futureRabatt** wird hier durch **thenCombine** (UND-Verkettung) und nicht durch **thenCombineAsync** ausgeführt. Es lohnt sich in dem Beispiel nicht, für die Berechnung des aktuellen Preises einen separaten Thread aus dem Thread-Pool anzufordern. Der dadurch entstehende Overhead wäre nicht gerechtfertigt.

Auch parallele IO-Aufrufe lassen sich mit Hilfe von CompletableFutures realisieren. IO-Aufrufe sind in der Regel blockierend



Split-and-Join-Pattern. Abspaltung von Tasks aus einem sequentiellen Ablauf und Abfragen des Ergebnisses. (Abb. 5)

und sollten somit bei parallelen Aufrufen mit Vorsicht eingesetzt werden. Der Code in **(Listing 5)** führt leicht dazu, dass alle Threads im Common-Pool längere Zeit blockiert werden und sollte deshalb so nicht verwendet werden!

(Listing 5)

```
List<URL> urls = getURLs();

// VORSICHT parallele IO-Aufrufe !
// Sollte so nicht verwendet werden !
List<String> result = urls.parallelStream()
                          .map( url -> request(url) )
                          .map( page -> parse(page) )
                          .collect( toList() );
```

Angenommen die URL-Liste in **(Listing 5)** enthält 80 Einträge. Geht man davon aus, dass eine URL-Anfrage eine Sekunde benötigt und hierbei die meiste Zeit mit Warten verbracht wird, so dauert die gesamte Verarbeitung bei einem commonPool mit 8 Threads ca. 10 Sekunden. Während dieser Zeit stehen die Threads für keine anderen Aufgaben zur Verfügung. **(Listing 6)** zeigt eine bessere Variante. Hier wird explizit mit einem separaten Thread-Pool gearbeitet. Um die Maschine vor einer Überlast zu schützen, wird die Maximalzahl der Pool-Threads begrenzt. Die URL-Anfragen erfolgen jetzt asynchron mit Threads aus dem neu erzeugten Pool. Wichtig ist hierbei, dass die Erzeugung der CompletableFuture-Instanzen und das Abfragen der Ergebnisse separat erfolgen. Würde das Auslesen des Ergebnisses im ersten Stream realisiert werden, so würde die Verarbeitung 80 Sekunden dauern, was einer sequentiellen Verarbeitung entsprechen würde. Bei der gezeigten Variante dauert das Lesen der URLs mit 80 Threads dagegen insgesamt nur eine Sekunde, da wartende Threads keine Last produzieren und vom Betriebssystem gut verwaltet werden können.

(Listing 6)

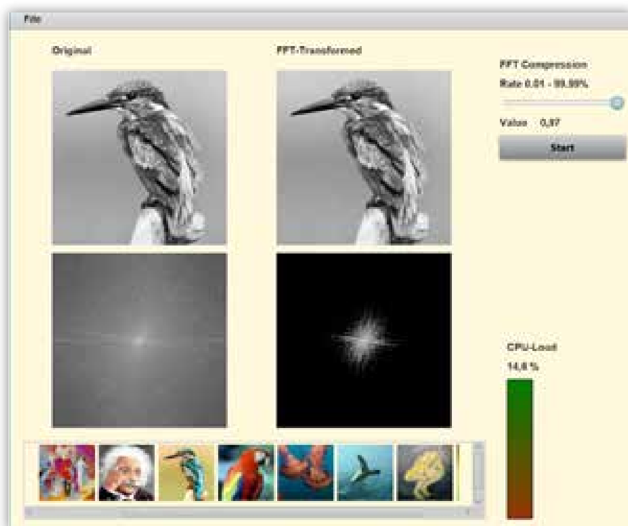
```
List<URL> urls = getURLs();

int maxThreads = ...;
ExecutorService executor = Executors.newFixedThreadPool(
    Math.min( urls.size(), maxThreads),
    new ThreadFactory()
    {
      @Override
      public Thread newThread(Runnable task)
      {
        Thread th = new Thread(task);
        th.setDaemon(true);
        return th;
      }
    } );

List<CompletableFuture<String>> futures =
     urls.stream()
         .map( url -> CompletableFuture.supplyAsync(
                       () -> request(url), executor ) )
         .map( future -> future.thenApplyAsync(
```

```
                        page -> parse(page), executor) )
        .collect( toList() );

List<String> result = futures.stream()
                        .map( CompletableFuture::join )
                        .collect( toList() );

executor.shutdown();
```

Mit dem Java-Projekt Loom[3] werden sich, wenn es zukünftig in das JDK integriert ist, neue Möglichkeiten für paralleles IO ergeben. Die von Loom eingeführten leichtgewichtigen User-Mode-Threads, sogenannte Fibers, erlauben eine bessere Ressourcenausnutzung. Der obige Code wird sich dadurch stark vereinfachen.

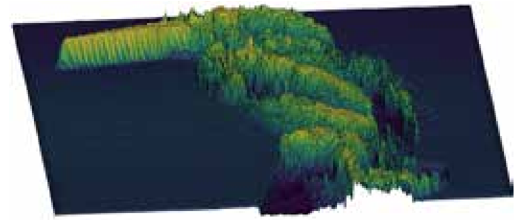
Einsatz von Task-Parallelität

Im Folgenden werden nun die Einsatzmöglichkeiten von CompletableFutures an einem konkreten Beispiel demonstriert. Hierzu wird eine JavaFX-Anwendung betrachtet, mit der die Bildkompression auf Basis einer zweidimensionalen Fourier-Transformation veranschaulicht wird **(Abb. 6)**. Interessierte Leser finden den kompletten Code der Anwendung auf Github[4].



Beispielanwendung. (Abb. 6)

Über das Menü wird ein Verzeichnis mit JPG-Bildern ausgewählt. Die Bilddateien werden daraufhin eingelesen und als Vorschau in der unteren Leiste angezeigt. Weiter wird auf der UI auch die Auslastung des Prozessors über einen entsprechenden Farbbalken sichtbar gemacht. Wird ein Vorschaubild ausgewählt, wird die Graudarstellung des Bildes berechnet und oben in der linken Anzeigespalte dargestellt. Zur Bestimmung der Fourier-Koeffizienten des Bildes (zweidimensionale Matrix) werden die Grauwerte über den jeweiligen Pixeln aufgetragen. Das dadurch entstehende Gebirge **(Abb. 7)** wird als zweidimensionales Signal interpretiert und ist somit einer Fourier-Transformation zugänglich.

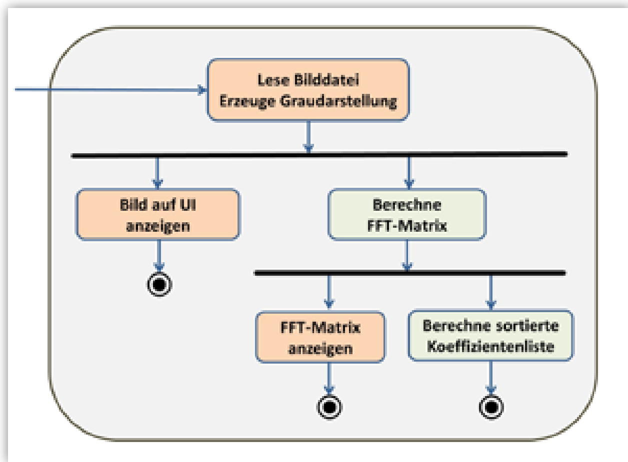


Farbwerte eines Bilds können als zweidimensionales Signal interpretiert werden. (Abb. 7)

Die Absolutwerte der resultierenden Fourier-Koeffizienten sind in der linken unteren Anzeige graucodiert dargestellt **(Abb. 6)**. Der größte Wert wird mit weiß und Null-Werte mit schwarz codiert. Bei gewöhnlichen Bildern ist ein Großteil der Fourier-Koeffizienten sehr klein. Wie auf der Abbildung zu erkennen ist, sind die meisten Koeffizienten dunkelgrau und enthalten somit recht wenig Information über das Bild. Dies kann man sich nun für eine Kompression zunutze machen. Hierzu werden z.B. nur die größten 10% der Koeffizienten beibehalten, alle anderen werden auf Null gesetzt. Mit einer inversen Fourier-Transformation lässt sich dann das Bild hieraus wieder rekonstruieren. Das entstehende Ergebnis kann kaum vom Original unterschieden werden. Selbst bei einer Reduktion um 97% wird das Bild recht gut wiederhergestellt **(Abb. 6)**. Die für die Rekonstruktion beibehaltenen Fourier-Koeffizienten und das daraus abgeleitete Bild sind in der rechten Spalte angezeigt.
Die Anwendung besitzt viele Stellen, an denen CompletableFutures für die Reaktionsfähigkeit und eine verbesserte Ablaufgeschwindigkeit gewinnbringend eingesetzt werden können. Beim Start der JavaFX-Anwendungen wird die Initialize-Methode des UI-Controllers aufgerufen. Hier werden neben den UI-Attributen (Attribute mit der `@FXML` Annotierung) auch der Hardware-Zugriff zum Auslesen der Prozessorlast initialisiert. Die Hardware-Zugriffe bestehen zum Teil aus lange laufenden und blockierenden Methoden. Diese führen bei einer synchronen Ausführung somit zu einer erheblichen Verzögerung. Durch die Auslagerung der Hardware-Initialisierung in einen asynchronen Task wird der Anwendungsstart deutlich beschleunigt **(Listing 7)**.

(Listing 7)

```
@Override
public void initialize(URL location, ResourceBundle resources)
{
   // init attributes

  CompletableFuture.runAsync(() -> {
      // Initialize hardware detection for getting the CPU load
      publisher = CpuInfoPublisher.getInstance();
      publisher.subscribe(value ->
         Platform.runLater(() -> {
              repaintGradient(value);
              cpuLabel.setText(...);
            }));
         })
    .orTimeout(3, TimeUnit.SECONDS)
    .exceptionally(ex -> {... });
}
```

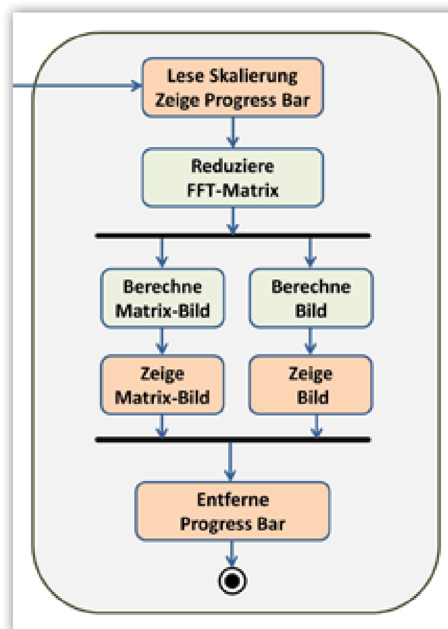


Aufbau der Callback-Methode für die Bild- und Fourier-Koeffizientenberechnung. Die orange eingefärbten Tasks greifen auf die UI zu und müssen im JavaFX-Thread ausgeführt werden. (Abb. 8)

Bei der Erzeugung der Vorschaubilder werden die Bilddateien gelesen und ImageView-Instanzen erstellt. Auch dieser Vorgang lässt sich durch Parallelisierung beschleunigen. Dazu werden die Dateizugriffe mit anschließender ImageView-Erzeugung in asynchrone Tasks ausgelagert. Zur Realisierung wird ein zu (Listing 6) analoges Code-Pattern benutzt.

Durch die Auswahl eines Bildes aus der Vorschau wird eine entsprechende Callback-Methode des UI-Controllers aufgerufen. In der Methode wird von dem gewählten Bild eine Graudarstellung berechnet, ein Image-Objekt erzeugt und anschließend angezeigt. Parallel zur Image-Erzeugung werden die Fourier-Koeffizienten (FFT-Matrix) der Graudarstellung berechnet und ein weiteres Image-Objekt für deren graphische Darstellung erzeugt und angezeigt. Sobald die Fourier-Koeffizienten vorliegen, wird auch schon eine sortierte Liste mit den Fourier-Koeffizienten angelegt, die später für die Bildkompression benötigt wird. Die Tasks für die Aktualisierung der UI müssen vom JavaFX-Thread durchgeführt werden. Bei diesen Tasks wird deshalb bei der Ausführung der JavaFX-Thread als Executor (**`Platform::runLater`**) angegeben. Die rechenintensiven Tasks können z.B. zur Ausführung an den Default-Executor (**`commonPool`**) übergeben werden. Die Strukturierung des kompletten Vorgangs entspricht einem erweiterten Fire-and-Forget-Pattern (Abb. 8).

Auch bei der Bildkompression kann Task-Parallelität eingesetzt werden. Sobald die reduzierten Fourier-Koeffizienten (reduzierte FFT-Matrix) vorliegen, werden zeitgleich die Darstellungen des reduzierten Bildes und der reduzierten Fourier-Koeffizienten berechnet. Sind die Berechnungen abgeschlossen, werden die zugehörigen Image-Objekte erzeugt und angezeigt. Abschließend wird der während der Berechnung eingeblendeter Fortschrittsbalken wieder entfernt (Abb. 9).



Aufbau der Callback-Methode für Bildreduktion. Tasks (orange eingefärbt), die auf die UI zugreifen müssen im JavaFX-Thread ausgeführt werden. (Abb. 9)

Neben den hier beschriebenen Szenarien für Task-Parallelität, lässt sich für die Berechnungen der zweidimensionalen Fourier-Transformationen auch Daten-Parallelität (parallel-Streams) einsetzen, worauf hier aber nicht näher eingegangen werden soll. Weiter werden für die Anzeige der Prozessorlast Reactive-Streams eingesetzt. Die CPU-Werte entsprechen einem kontinuierlichen Datenstrom (Publisher), bei dem sich der Anzeigebalken als Subscriber registriert.

Fazit:

Für den Einsatz von Parallelisierung müssen bei einer Anwendung zuerst nebenläufige Aktionen bzw. Tasks identifiziert werden. Im Anschluss wird jeweils entschieden, ob es sich um Daten- oder Task-Parallelität handelt. Java stellt für die Umsetzung einer Parallelisierung verschiedene Abstraktionen bzw. Frameworks zur Verfügung. Neben parallelen Streams für die Implementierung von Datenparallelität kommt hier insbesondere die CompletableFuture-Klasse zum Einsatz. Mit ihrer Hilfe wird auf einfache Art und Weise Task-Parallelität umgesetzt, wobei ggf. auf die Koordinierung von Race-Conditions geachtet werden muss. Besitzt man erst einmal eine gewisse Erfahrung für die Identifikation und Handhabung von Nebenläufigkeit, lässt sich durch den Einsatz von CompletableFutures elegant Task-Parallelität realisieren. Durch den Einsatz einiger Best-Practices werden Rechenressourcen besser ausgenutzt und Anwendungen somit schneller und reaktiver.

Quellen:

1 https://java-pro.de/reaktive-programmierung-teil-1/
2 https://java-pro.de/reaktive-programmierung-teil2/
3 Projekt Loom: https://openjdk.java.net/projects/loom/
4 Hettel und Tran: Nebenläufige Programmierung mit Java, dpunkt.verlag, 2016
5 Beispielcode: https://bit.ly/2UXERIf

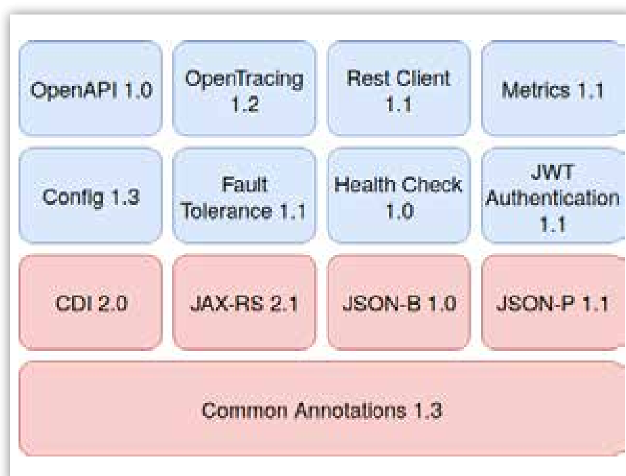
#JAVAPRO #MicroProfile #Microservices #JavaEE

MicroProfile für Microservices mit Java-EE

Die Entwicklung von Microservices schreitet in einem rasanten Tempo voran und stellt Entwickler vor immer neue und anspruchsvollere Herausforderungen. Nach der Meinung vieler Experten ist Java-EE nicht geeignet, die gestiegenen Ansprüche zu erfüllen. Aus diesem Grund wurde das Eclipse-MicroProfile-Projekt ins Leben gerufen. Ziel ist es, einen Standard für die Entwicklung von Microservices auf Basis von Java zu definieren.

Was ist MicroProfile eigentlich?

Eclipse-MicroProfile ist, genauso wie Java-EE (jetzt Jakarta-EE), eine Sammlung von Spezifikationen, die zusammen eine API darstellen, welche auf Microservice-basierte Softwareentwicklung fokussiert ist. MicroProfile 2.1 beinhaltet beispielsweise 13 Spezifikationen[1] **(Abb. 1)**.



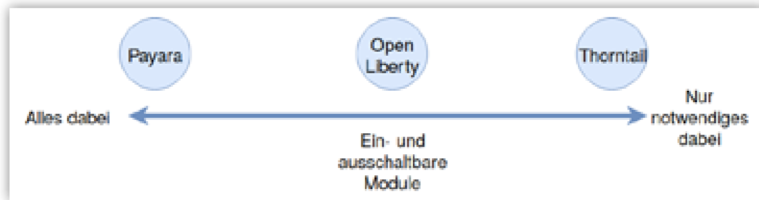
Bestandteile des MicroProfile 2.1. (Abb. 1)

Die rot dargestellten Spezifikationen wurden von der Java-EE-8-Spezifikation[2] übernommen und sind mit dieser identisch **(Abb. 1)**. So verwendet MicroProfile ebenso wie Java-EE 8 eine Implementierung von CDI für die Dependency-Injection. Die blau dargestellten Spezifikationen wurden speziell für MicroProfile entwickelt. Diese acht speziellen Spezifikationen unterscheiden sich von ihren Java-EE-Pendants unter anderem dadurch, dass es für sie keine Referenzimplementierungen gibt. Zwar existiert ein Projekt namens Smallrye, das diverse Implementierungen des Standards bietet, sich jedoch eher als Community-betriebene Sammlung versteht und keinen Referenzanspruch hat.

Autor:

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

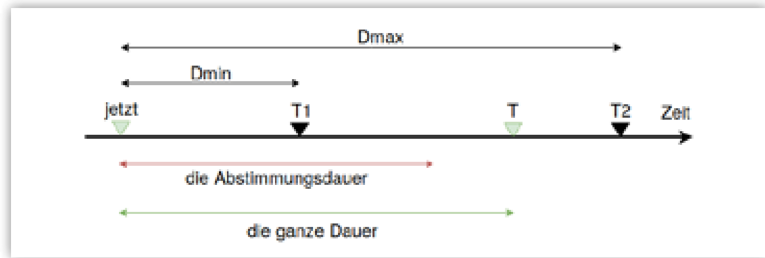
www.labs.consol.de/
GitHub: https://github.com/progaddict/
E-Mail: Alexander.Ryndin@consol.de



Einsatzansätze: MicroProfile schreibt nicht vor, wie genau ein kompatibler Application-Server organisiert sein soll, die alleinige Entscheidung liegt daher bei den Anbietern. (Abb. 2)

die Prophezeiungen nach Ablauf der vier Wochen und erstellt Statistiken über die Benutzer, beziehungsweise berechnet das Abschneiden ihrer Prophezeiungen. Dadurch kann die Anwendung zum Beispiel eine Aussage darüber treffen, welche Präzision der Benutzer X als Prophet hat. Die Präzision ist dabei das Verhältnis der Anzahl von erfüllten Prophezeiungen zu der Anzahl aller überprüften Prophezeiungen.

Die oben genannten APIs sind in Form von Bill-of-Materials-Definitionen (BOM) für das Build-Tool Apache-Maven erhältlich. Jeder MicroProfile-kompatibler Anbieter von Application-Servern (zum Beispiel Payara-Micro) sollte zu diesen APIs die entsprechenden Implementierungen liefern. Um die Kompatibilität mit den Standards sicherzustellen, muss jede Implementierung die Tests des zugehörigen Test-Compatibility-Kits (TCK) bestehen (siehe Beispiel[3]).



Prophezeiungsbeschränkungen. (Abb. 3)

Es gibt zwei entgegengesetzte Ansätze. Bei Servern wie Payara-Micro sind alle MicroProfile-Implementierungen enthalten (Abb. 2). Dies ist die klassische Strategie, wie sie auch von den meisten Java-EE-Anwendungs-Servern verfolgt wird. Der zweite Ansatz verfügt hingegen nur über Implementierungen, die wirklich von der Anwendung gebraucht werden (Abb. 2). Ein Beispiel dafür ist Thorntail, ehemals Wildfly-Swarm von JBoss beziehungsweise RedHat.

Nach Meinung einiger Programmierer gehört dem zweiten Ansatz die Zukunft, da Microservices möglichst klein und so ressourcenschonend wie möglich sein sollten. Etablierten Anbietern könnte es allerdings Schwierigkeiten bereiten, diesen Ansatz auf ihren existierenden Plattformen schnell zu implementieren, da diese aus historischen Gründen eher monolithisch sind.

Diese Klassifizierung ist nicht ausschließlich schwarz-weiß zu sehen. Dazwischen positioniert sich zum Beispiel das Open-Liberty-Projekt, das aus dem Websphere-Application-Server von IBM hervorgegangen ist. Der Open-Liberty-Server beinhaltet zwar alle Implementierungen, diese können jedoch selektiv ein- und ausgeschaltet werden. Open-Liberty ist aus OSGi-Modulen aufgebaut[4], die sowohl aktiviert als auch deaktiviert werden können, was gegebenenfalls Ressourcen spart.

Prophezeiungen in Nostrastockerus erstellen

Nostrastockerus[5] ist eine Beispielanwendung, welche auf Open-Liberty und MicroProfile 2.1 basiert und für das Deployment auf Kubernetes gedacht ist. In dieser Anwendung können Benutzer Prophezeiungen erstellen. Ein Beispiel: Ein Benutzer sagt voraus, dass in vier Wochen die Aktie des Konzerns X den Wert Y in US-Dollar erreichen wird. Andere Benutzer können für oder gegen diese Annahme stimmen. Die Anwendung überprüft

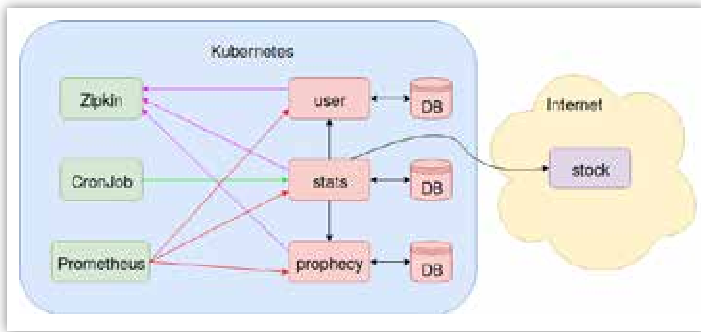
Natürlich können für eine Prophezeiung auch Beschränkungen festgelegt werden, wie zum Beispiel die Dauer (Abb. 3). Der Zeitraum bis zum Eintritt der Prophezeiung soll mindestens Dmin und höchstens Dmax sein. Nachdem sie erstellt wurde, darf nur während der Abstimmungsdauer abgestimmt werden. Die Abstimmungsdauer einer Prophezeiung ist als Prozentanteil der ganzen Dauer definiert (zum Beispiel 80%). Alle diese Parameter (Dmin, Dmax und die Abstimmungsdauer als Prozentanteil) sollen konfigurierbar sein (die Anforderung A1).

Weitere, für Microservices typische Anforderungen sind:

- A2: Die Anwendung soll eine rollenbasierte Zugriffskontrolle (RBZK) haben.
- A3: Die Anwendung soll so zustandslos wie möglich sein, damit sie gut skalierbar bleibt.
- A4: Es soll möglich sein, die Gesundheit der Anwendung per Monitoring zu überprüfen.
- A5: Metriken zum Ressourcenverbrauch sollen verfügbar sein, zum Beispiel die CPU-Last.
- A6: Die Anwendung soll eine Dokumentation der APIs für die öffentliche Verwendung zur Verfügung stellen. Es soll beispielsweise relativ einfach sein, von einem Android-Client die APIs zu konsumieren.
- A7: Das Anwendungsverhalten soll per Tracing beobachtbar sein. Es soll beispielsweise möglich sein, die Verkettung von HTTP-Aufrufen nachzuvollziehen.
- A8: Die Fehlerrobustheit soll relativ einfach eingebaut werden können. Bei einem Fehler wäre so beispielsweise die Wiederholungen von HTTP-Aufrufen problemlos möglich.

Anforderungen mittels MicroProfile erfüllen

Nostrastockerus besteht aus drei Microservices: **`user`**, **`prophecy`** und **`stats`**. Der user-Microservice speichert Benutzer und erstellt



Anwendung Nostrastockerus-Architektur. (Abb. 4)

JSON-Web-Tokens (JWT)[6]. Der **prophecy** Microservice speichert Prophezeiungen. Der **stats** Microservice überprüft Prophezeiungen und berechnet die aggregierte Statistik über Benutzer und ihre Prophezeiungen. Zudem ruft **stats** einen externen **stock** Service auf, um die Werte von Aktien zu erhalten.

Neben diesen Microservices gibt es weitere Infrastruktur-Komponenten: **Zipkin**[7], **CronJobs**[8] und **Prometheus**[9 10]. **Zipkin** empfängt Tracing-Daten von den Microservices und speichert sie, wenn zum Beispiel HTTP-Aufrufe eingehen (Abb. 4 – lila Pfeile). **Prometheus** fragt die Microservices regelmäßig ab, um ihre Metriken zu sammeln (Abb. 4 – rote Pfeile). Der **CronJob** ist ein Feature von Kubernetes und ruft periodisch via HTTP den **stats** Service auf (Abb. 4 – grüner Pfeil) und löst dadurch die Überprüfung der Prophezeiungen aus. Die Anforderung A1 wird über die MicroProfile-Config-API erfüllt. Die Config-API ist eine Kern-API von MicroProfile, da viele andere APIs die Config-API für ihre Konfiguration benutzen.

(Listing 1) – Definition des Parameters Dmin mittels Config-API

```
@ApplicationScoped
public class Config {

    @Inject
    @ConfigProperty(
            name = "config.minDifferenceBetweenExpectedAtAnd
            CreatedAtSeconds",
            defaultValue = "604800" // 7 Tage
    )
    private Long minDifferenceBetweenExpectedAtAndCreatedAt
    Seconds;
    private Duration minDifferenceBetweenExpectedAtAndCreatedAt;

    @PostConstruct
    void init() {
        minDifferenceBetweenExpectedAtAndCreatedAt =
                Duration.ofSeconds(minDifferenceBetweenExpected
                AtAndCreatedAtSeconds);
    }

    public Duration getMinDifferenceBetweenExpectedAtAnd
    CreatedAt() {
        return minDifferenceBetweenExpectedAtAndCreatedAt;
    }
}
```

Der Parameter wird von der MicroProfile-Config-Implementierung gelesen und in das Feld **minDifferenceBetweenExpectedAtAndCreatedAtSeconds** übertragen. Dieser Wert wird in der **init** Methode der Einfachheit halber nach Duration umgewandelt. Anschließend wird eine Instanz der Config-Klasse in den anderen Komponenten der Anwendung injiziert. Dadurch können Konfigurationswerte über Getter-Methoden gelesen werden. Standardmäßig werden Parameter von diesen drei Quellen gelesen:

- Systemeigenschaften
- Umgebungsvariablen
- Datei META-INF/microprofile-config.properties im Class-Path

Gelesen wird in der gelisteten Reihenfolge, das bedeutet die erste Quelle, die einen Parameter definiert, gewinnt. Da die Namen der Umgebungsvariablen nicht alle Symbole beinhalten dürfen, die als Parameter-Namen zulässig sind, werden die verbotenen Symbole in diesen Namen beim Einlesen als Umgebungsvariable einfach durch einen Unterstrich ersetzt. Der oben definierte Parameter mit dem problematischen Punkt kann beispielsweise über eine Umgebungsvariable **config_minDifferenceBetweenExpectedAtAndCreatedAtSeconds** definiert werden.

Gesundheitsüberprüfung und Metriken

Die Anforderung A4 wird durch MicroProfile-Health-Check erfüllt. Um Health-Check in Open-Liberty zu aktivieren, muss zunächst die entsprechende Abhängigkeit hinzugefügt werden:

(Listing 2)

```
<dependency>
    <groupId>io.openliberty.features</groupId>
    <artifactId>mpHealth-1.0</artifactId>
    <version>18.0.0.4</version>
    <type>esa</type>
    <scope>provided</scope>
</dependency>
```

Anschließend sollte das entsprechende Merkmal in der Server-Konfiguration von Open-Liberty (server.xml) eingeschaltet werden:

(Listing 3)

```
<server>
    <featureManager>
        <feature>mpHealth-1.0</feature>
        <!-- … andere Merkmale … -->
    </featureManager>
    <!-- … andere Einstellungen … -->
</server>
```

Und schließlich muss die Logik der gewünschten Überprüfung implementiert werden, wie (Listing 4) zeigt.

(Listing 4)

```
@Health
@ApplicationScoped
public class DbHealth implements HealthCheck {

    private static final String HEALTH_CHECK_NAME = "database";
    private static final String DATA_KEY = "database";

    @Override
    public HealthCheckResponse call() {
        boolean isDbWorking;
        // … überprüfen ob DB funktioniert und entsprechend die
         isDbWorking setzen …
        return isDbWorking ? up() : down();
    }

    private static HealthCheckResponse up() {
        return HealthCheckResponse.named(HEALTH_CHECK_NAME)
                .withData(DATA_KEY, "available")
                .up()
                .build();
    }

    private static HealthCheckResponse down() {
        return HealthCheckResponse.named(HEALTH_CHECK_NAME)
                .withData(DATA_KEY, "not available")
                .down()
                .build();
    }
}
```

Das Health-Check-Merkmal sucht den Klassenpfad automatisch nach Implementierungen des Health-Check-Interfaces ab und meldet diese an. Das Merkmal fügt auch den Endpunkt /**health** zu der Anwendung hinzu. Dieser Endpunkt kann von außen abgerufen werden, um die Information über die Gesundheit der Anwendung zu erhalten:

(Listing 5)

```
{
    "checks": [
        {
            "data": {
                "default server": "available"
            },
            "name": "ApplicationServerHealth",
            "state": "UP"
        },
        {
            "data": {
                "database": "available"
            },
            "name": "database",
            "state": "UP"
        }
    ],
    "outcome": "UP"
}
```

Die entsprechende Überprüfung kann als eine sogenannte Liveness-Probe in Kubernetes hinzugefügt werden, über welche die Plattform automatisch überprüfen kann, ob die Anwendung gesund ist:

(Listing 6)

```
apiVersion: apps/v1
kind: Deployment
spec:
  …
  template:
    …
    spec:
      containers:
        - name: prophecy
          …
          livenessProbe:
            httpGet:
              scheme: HTTPS
              path: /health
              port: 9443
            timeoutSeconds: 3
            initialDelaySeconds: 15
            periodSeconds: 10
      …
```

Mit den Metriken funktioniert es ähnlich. Das entsprechende Merkmal (**mpMetrics-1.1**) sollte als Abhängigkeit hinzugefügt und in der Server-Konfiguration aktiviert werden. Danach können die Metrikimplementierungen via Java-Annotationen definiert werden:

(Listing 7)

```
@Transactional
@ApplicationScoped
public class ProphecyManager {

    @Inject
    private ProphecyMapper prophecyMapper;

    @Inject
    private ProphecyDao prophecyDao;

    @Timed(name = "execution_time_of_create_prophecy")
    public long createProphecy(final Prophecy model) {
        final ProphecyEntity entity = prophecyDao.create(prophecy
         Mapper.model2Entity(model));
        return entity.getId();
    }

    // … andere Methoden …
}
```

(Listing 7) definiert eine Metrik, welche die Ausführungsdauer der entsprechenden Methode misst. Genauso wie beim Health-Check-Merkmal wird ein Endpunkt /**metrics** automatisch hinzugefügt. Prometheus kann diesen Endpunkt abrufen. Dadurch ist die Anwendung an das Monitoring angebunden und die Anforderung A5 erfüllt. Wichtig dabei ist, dass die Metriken über den gleichen CDI-Scope der Anwendung verfügen (d. h. Metriken sollen **@ApplicationScoped** sein).

Das Merkmal fügt die übliche Metriken wie zum Beispiel CPU-Last, Verbrauch des Arbeitsspeichers der JVM und Anzahl der JVM-Threads automatisch hinzu. Es ist auch möglich, eine

eigene spezifische Metrik zu implementieren. Die in (Listing 8) abgebildete Metrik misst zum Beispiel die Gültigkeitsdauer des erzeugten JWT-Tokens.

(Listing 8)

```
@ApplicationScoped
public class TokenManager {

    // das wird irgendwo in dem Code berechnet und gesetzt
    private volatile long lastValidityDurationMillis;

    @Produces
    @Metric(name = "token_validity_duration", unit = MetricUnits.
    MILLISECONDS)
    @ApplicationScoped
    protected Gauge<Long> getTokenValidityDurationMillis() {
        return () -> lastValidityDurationMillis;
    }
    // … andere Methoden …
}
```

Authentifizierung mittels MicroProfile-JWT

Die Anforderung A2 wird durch das MicroProfile-Merkmal JWT-Authentication erfüllt. Es trägt auch zu der Anforderung A3 bei, da eine Token-basierte Authentifizierung, wie JWT sie bietet, zustandslos ist. Um ein JWT-Token zu überprüfen, werden keine Aufrufe zu einem Service benötigt. Dafür wird nur der entsprechende öffentliche Schlüssel des Herausgebers des Tokens benutzt. Ein Nachteil der Token-basierten Authentifizierung ist die Verteilung des entsprechenden öffentlichen Schlüssels. Als Lösung bietet sich die Verwendung einer öffentlichen Public-Key-Infrastruktur (PKI) an. Dafür kann zum Beispiel Let's-Encrypt[11] genutzt werden. Alternativ können die entsprechenden öffentlichen Schlüssel auch über das Anwendung-Artefakt mit distribuiert werden, zum Beispiel in die Anwendungs-JAR oder in das Docker-Image.

Ein JWT-Token besteht aus drei Base64-kodierten Teilen, die durch Punkte getrennt sind:

```
<base64-teil1>.<base64-teil2>.<base64-teil3>
```

Der erste Teil ist ein Header in dekodierter Form:

(Listing 9)

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

Der Header weist auf den Token-Typ (JWT) hin und welcher Algorithmus (RS256 – RSA mit SHA-256) zur Überprüfung des Tokens genutzt werden soll. Der zweite Teil enthält die Nutzlast (Payload):

(Listing 10)

```
{
  "token_type": "Bearer",
  "sub": "admin",
  "upn": "admin",
  "groups": [
    "admin"
  ],
  "iss": "https://user/token",
  "exp": 1636061212,
  "iat": 1549661212
}
```

Die Nutzlast beinhaltet die sogenannten Ansprüche (Claims), wie zum Beispiel Benutzeridentifizierungsdaten **sub** (Subject) und **upn** (User-Principal), **groups** (Gruppen der Benutzer) die der Anforderung A2 zufolge nach Rollen für eine rollenbasierte Zugriffskontrolle abgebildet werden, den Herausgeber des Tokens **iss** (Issuer), den Erschaffungszeitstempel **iat** (issued-at) und den Gültigkeitszeitstempel **exp** (expires).

Der letzte und dritte Teil eines JWTs ist die Signatur. Die Signatur wird aus den beiden ersten Teilen erzeugt. Dazu wird der private Schlüssel des Herausgebers verwendet. Da Konsumenten der JWTs über den entsprechenden öffentlichen Schlüssel verfügen, können sie diese Signatur auf ihre Berechtigung hin überprüfen. Wenn zum Beispiel ein bösartiger Benutzer „Mallory" versucht, zu den Gruppen seines JWTs **admin** hinzuzufügen, um sich zum Superbenutzer zu machen, wird die Signatur nicht stimmen. Um die neue valide Signatur zu berechnen, ist der private Schlüssel nötig, den Mallory natürlich nicht besitzt.

Um JWT-Tokens in Open-Liberty zu verwenden, sind folgende Komponenten nötig:

- Eine Java-Keystore-Datei, welche ein asynchrones Schlüsselpaar enthält. Der private Schlüssel wird für die Erzeugung der Signaturen benutzt. Der öffentliche Schlüssel muss auf alle Microservices verteilt werden und dient der Überprüfung dieser Signatur.
- Das **MpJwt-1.1** Merkmal (es wird genauso aktiviert wie die anderen Merkmale)
- Die Konfiguration einer JwtBuilder-Instanz (Open-Liberty-API).

Ein Java-Keystore und die Schlüssel können mithilfe von Keytools[12], einem Tool aus dem JDK, erzeugt werden. Keystore, das MpJwt-Merkmal und der JwtBuilder werden in der Server-Konfiguration eingestellt:

(Listing 11) – Einstellung Keystore in Server-Konfiguration

```
<server description="user service">
    <featureManager>
        <feature>mpJwt-1.1</feature>
        <!-- … weitere Merkmale … -->
    </featureManager>
```

```
    <keyStore id="defaultKeyStore"
              location="/pfad/zu/meinem/keystore"
              type="PKCS12"
              password="das_Geheimnis"
              readonly="true"/>

    <jwtBuilder id="jwtBuilder"
                keyStoreRef="defaultKeyStore"
                keyAlias="myPrivateKeyName"
                issuer="https://my-jwt-service"/>

    <!-- … weitere Einstellungen … -->
</server>
```

(Listing 12) – Nutzung der JwtBuilder im Code

```
@ApplicationScoped
public class TokenManager {
    public String createToken(final CreateTokenRequest create
   TokenRequest, final User user) {
        final Instant now = Instant.now();
        try {
            return JwtBuilder.create("jwtBuilder")
                    .subject(user.getName())
                    .expirationTime(
                        now.plusMillis(createTokenRequest.get
                        ValidityDurationMillis()).getEpoch
                        Second()
                     )
                    .claim("upn", user.getName())
                    .claim("groups", createTokenRequest.get
                     Roles())
                    .buildJwt()
                    .compact();
        } catch (final JwtException | InvalidBuilderException |
        InvalidClaimException e) {
            throw new IllegalStateException("could not create
            JWT", e);
        }
    }
}
```

Der JwtBuilder gehört zu der Open-Liberty-API und nicht zu der MicroProfile-JWT-Authentication-API.

(Listing 13) – Übertragung des erzeugten JWT im Authorization-HTTP-Header

```
GET /prophecy/11 HTTP/1.1
Host: prophecy
Authorization: Bearer <JWT>
Accept: application/json
… weitere Headers …
```

Die Überprüfung eines eingehenden JWT-Tokens wird von Open-Liberty anhand der RolesAllowed-Annotationen und des bereitgestellten öffentlichen Schlüssels durchgeführt. Der Schlüssel kommt aus einem Java-Truststore. Der Truststore ist ebenfalls eine Keystore-Datei, die jedoch nur den öffentlichen Schlüssel enthält.

(Listing 14) – Einstellung Truststore in Open-Liberty

```
<server description="prophecy service">
    <featureManager>
        <feature>mpJwt-1.1</feature>
        <!-- … weitere Merkmale … -->
    </featureManager>

    <keyStore id="defaultTrustStore"
              location="/pfad/zu/meinem/truststore"
              type="PKCS12"
              password="das_zweite_Geheimnis"
              readonly="true"/>

    <sslDefault sslRef="ssl"/>
    <ssl id="ssl" trustStoreRef="defaultTrustStore"/>

    <mpJwt id="mpJwt"
           sslRef="ssl"
           keyName="myPublicKeyName"
           issuer="https://my-jwt-service"/>

    <!-- … weitere Einstellungen … -->
</server>
```

(Listing 15) – Absicherung von Endpoints im Code

```
@RequestScoped
@Path("prophecy")
public class ProphecyResource {

    @Inject
    private JsonWebToken jwt;

    @POST
    @RolesAllowed({„admin", „prophecy-creator"})
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createProphecy(final CreateProphecyRequest
    createProphecyRequest) {
        final Prophecy prophesy = new Prophecy()
                .setCreatedBy(jwt.getName())
                .setCreatedAt(Instant.now())
                .setStockName(createProphecyRequest.getStock
                 Name())
                .setStockExpectedValue(createProphecyRequest.
                 getStockExpectedValue())
                .setProphecyType(createProphecyRequest.get
                 ProphecyType())
                .setExpectedAt(createProphecyRequest.get
                 ExpectedAt());
        final long id = prophecyManager.createProphecy(prophesy);
        final URI uri = UriBuilder.fromResource(ProphecyResource.
         class).path("{id}").build(id);
        return Response.created(uri).build();
    }
}
```

In diesem Fall darf die POST-Prophecy-Anfrage nur mit einem Token durchgeführt werden, welches die Gruppen **`admin`** oder **`prophecy-creator`** enthält. Da ein JWT nur in dem CDI-Scope einer Anfrage existiert, ist es wichtig zu beachten, dass die Endpoints **`@RequestScoped`** sind. JWT kann als **`JsonWebToken`** injected werden (Listing 15).

Export und Konsum der APIs

Die Anforderung A6 wird durch MicroProfiles-OpenAPI erfüllt. Das entsprechende Open-Liberty-Merkmal **mpOpenAPI-1.0** fügt den Endpunkt /**openapi** hinzu. Wenn aufgerufen, liefert der Endpunkt eine OpenAPI-Beschreibung der Anwendungs-Endpunkte in einer YAML-Form. (Listing 16) zeigt, wie diese ungefähr aussehen kann:

(Listing 16)

```
openapi: 3.0.0
info:
  title: Deployed APIs
  version: 1.0.0
paths:
  /prophecy/{id}:
    get:
      summary: Get prophecy information.
      description: Retrieve prophecy information.
      operationId: getProphecyInformation
      parameters:
      - name: id
        in: path
        description: Prophecy ID.
        required: true
        schema:
          type: integer
          format: int64
      responses:
        404:
          description: The prophecy does not exist.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ClientError'
        200:
          description: Prophecy information.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Prophecy'
      security:
      - jwt: []
  … weitere Endpunkte …
```

Anhand dieser Beschreibung und des OpenAPI-Generators[13] können Clients für verschiedene Plattformen und Sprachen, wie zum Beispiel ein Swift-Client[14] erzeugt werden. Um eine korrekte und aussagekräftige OpenAPI-Beschreibung zu erhalten, müssen die Endpunkte, Parameter und Models (meistens POJOs) der Anwendung mit OpenAPI-Annotationen versehen werden, wie (Listing 17) zeigt:

(Listing 17):

```
@RequestScoped
@Path("prophecy")
@SecuritySchemes({
        @SecurityScheme(
                securitySchemeName = „jwt“,
                type = SecuritySchemeType.HTTP,
                scheme = "bearer",
                bearerFormat = "JWT"
        )
})
public class ProphecyResource {

    @GET
    @Path("{id}")
    @RolesAllowed({„admin“, „prophecy-creator“, „prophecy-reader“})
    @Operation(
            operationId = "getProphecyInformation",
            summary = "Get prophecy information.",
            description = "Retrieve prophecy information."
    )
    @APIResponses(
            value = {
                    @APIResponse(
                            responseCode = "404",
                            description = "The prophecy does not
                            exist.",
                            content = @Content(
                                    mediaType = MediaType.
                                    APPLICATION_JSON,
                                    schema = @Schema(implementation
                                    = ClientError.class)
                            )
                    ),
                    @APIResponse(
                            responseCode = "200",
                            description = "Prophecy information.",
                            content = @Content(
                                    mediaType = MediaType.
                                    APPLICATION_JSON,
                                    schema = @Schema(implementation
                                    = Prophecy.class)
                            )
                    )
            }
    )
    @SecurityRequirement(name = AppSecurityScheme.JWT)
    public Response getProphecy(
            @PathParam("id")
            @NotNull
            @Parameter(
                    in = ParameterIn.PATH,
                    description = "Prophecy ID.",
                    required = true
            )
            @Valid final Long id
    ) {
        final Optional<Prophecy> prophecy = prophecyManager.
         findProphecy(id);
        if (!prophecy.isPresent()) {
            return Response
                    .status(Response.Status.NOT_FOUND)
                    .entity(new ClientError().setMessage(Client
                    ErrorMessage.PROPHECY_NOT_FOUND))
                    .type(MediaType.APPLICATION_JSON)
                    .build();
        }
        return Response.ok(prophecy)
                .type(MediaType.APPLICATION_JSON)
                .build();
    }
}
```

(Listing 18) – Beispiel, wie Models versehen werden können

```
@Schema(description = "Information needed to create a prophecy.")
public class CreateProphecyRequest {

    @Schema(
            description = "Stock name.",
            required = true,
            minLength = 1,
            maxLength = 100,
            example = "GOOG"
    )
    private String stockName;

    @Schema(
            description = "Expected future stock value.",
            required = true,
            minimum = "0.0",
            example = "120.45"
    )
    private BigDecimal stockExpectedValue;

    @Schema(
            description = "Type of prophecy.",
            required = true,
            implementation = ProphecyType.class
    )
    private ProphecyType prophecyType;

    @Schema(
            description = "Moment in time when the prophecy is
            expected to be fulfilled.",
            required = true,
            type = SchemaType.STRING,
            implementation = String.class,
            format = DataFormat.TIMESTAMP,
            example = "\"2019-03-21T14:30:44.406Z\""
    )
    private Instant expectedAt;

    // … setters und getters …
}
```

Anhand dieser Metainformation erzeugt die OpenAPI-Implementierung die entsprechende Beschreibung der API. Natürlich können Entwickler auch die umgekehrte Herangehensweise beziehungsweise einen API-first-Approach[15] wählen. Dabei wird zuerst eine OpenAPI-Beschreibung erstellt. Über den OpenAPI-Generator wird ein Server-Rumpf (Server-Stub) erzeugt, der um die konkrete Implementierung erweitert werden kann.

Beobachtbarkeit

Die Anforderung A7 wird durch MicroProfile-OpenTracing erfüllt. Das entsprechende Open-Liberty-Merkmal heißt **`mpOpenTracing-1.1`**. Dazu wird oft das Merkmal **`usr:opentracingZipkin-0.31`** hinzugefügt, das als ein Adapter zum Zipkin-Service dient:

(Listing 19)

```
<server description="prophecy service">
    <featureManager>
        <feature>mpOpenTracing-1.1</feature>
        <feature>usr:opentracingZipkin-0.31</feature>
        <!-- … weitere Merkmale …  -->
    </featureManager>

    <opentracingZipkin host="zipkin" port="9411"/>
    <!-- … weitere Einstellungen … -->
</server>
```

Das **`usr:opentracingZipkin-0.31`** Merkmal ist in der Open-Liberty-Distribution nicht enthalten. Es muss manuell heruntergeladen[16] und in den Ordner **`/opt/ol/wlp/usr/extension`** der Open-Liberty-Installation kopiert werden.

Wenn diese zwei Merkmale eingeschaltet sind, können die HTTP-Aufrufketten in Zipkin beobachtet werden **(Abb. 5)**:

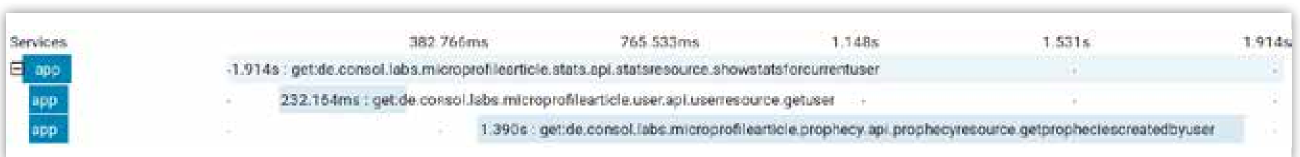
(Abb. 5) zeigt die HTTP-Aufrufketten **`stats`** ➔ **`user`** und **`stats`** ➔ **`prophecy`**. Service **`stats`** ruft Service **`user`**, um Benutzerinformation zu bekommen (der Aufruf hat 232.164 Millisekunden gedauert). Danach wird Service **`prophecy`** aufgerufen, um sich die Prophezeiungen des Benutzers zu holen (der Aufruf hat 1.390 Sekunden gedauert).

Unter der Haube funktioniert das durch spezielle HTTP-Header für die Nachvollziehbarkeit von Aufrufketten. **(Listing 20)** zeigt ein Beispiel eines HTTP-Aufrufes mit entsprechenden Headern mit dem Namenspräfix **`X-B3-`**.

(Listing 20)

```
GET /user/bob HTTP/1.1
Host: user
Accept: application/json
X-B3-TraceId: 442f8f2b640a3b91
X-B3-SpanId: 469fa65b9666f6f7
X-B3-ParentSpanId: 442f8f2b640a3b91
X-B3-Sampled: 1
… weitere Headers …
```

Diese Header werden von der Server-Engine von JAX-RS entdeckt und automatisch weiter verwendet, wenn in der Folge der Verarbeitung des Requests eine JAX-RS-Client-Verbindung benutzt



Ein Beispiel von der Spur der Aufrufketten in Zipkin. (Abb. 5)

wird. Das passiert natürlich nur, wenn das OpenTracing-Merkmal eingeschaltet ist.

Fehlerrobustheit

Die Anforderung A8 wird durch MicroProfile-Fault-Tolerance erfüllt. Das entsprechende Open-Liberty-Merkmal heißt **`mpFaultTolerance-1.1`**. Schlägt der Prophecy-Microservice mit einer Wahrscheinlichkeit von 50% fehl, kann dieser Service trotzdem benutzt werden, wenn der Aufruf im Fehlerfall wiederholt wird. **(Listing 21)** demonstriert, wie dieses Verhalten relativ einfach mit MicroProfile-Fault-Tolerance definiert werden kann:

(Listing 21)

```
@RequestScoped
public class ProphecyApi {

    @Retry
    public List<Prophecy> getPropheciesCreatedByUser(final String
    userName) {
        try {
            return getApi().getPropheciesCreatedByUser(userName);
        } catch (final ApiException e) {
            throw new RuntimeException("failed to retrieve
             prophecies created by user " + userName, e);
        }
    }
}
```

Durch die **`@Retry`** Annotation wird ein Aufruf der Methode **`getPropheciesCreatedByUser`** abgefangen. Wenn dieser fehlschlägt, wird er automatisch in einer konfigurierbaren Anzahl wiederholt. Diese Konfiguration wird per MicroProfile-Config festgelegt. Für entsprechende Konfigurationsparameter gibt es folgendes Namensschema, das sich an Klassen- und Methodennamen der betroffenen Funktionalitäten orientiert:

```
<classname>/<methodname>/<annotation>/<parameter>
```

Verfügbare Parameter sind zum Beispiel: **`maxRetries`** (Anzahl Wiederholungen), **`delay`** (Zeit zwischen den Wiederholungen) und **`jitter`** (zufällige Variationsdauer der Zwischenzeit). Daraus ergeben sich bei längeren Paketnamen auch relativ lange Parameter-Namen, wie zum Beispiel:

```
de.consol.labs.microprofilearticle.stats.integration.
prophecy.ProphecyAPI/getPropheciesCreatedByUser/
Retry/maxRetries.
```

Will man diese als Umgebungsvariablen setzen, wird es bezüglich Übersichtlichkeit nicht besser:

```
DE_CONSOL_LABS_MICROPROFILEARTICLE_STATS_INTEGRATION
_PROPHECY_PROPHECYAPI_GETPROPHECIESCREATEDBYUSER_
RETRY_MAXRETRIES.
```

Alternativ können diese Werte auch als Parameter direkt an der Annotation festgelegt werden.

Fazit:

Die Nutzung von MicroProfile als Standard ist für die Entwicklung von Microservices in Java ein großer Schritt nach vorne. Typische Anforderungen zu Microservice-basierten Anwendungen können hier über eine verhältnismäßig kleine Anzahl von dedizierten APIs erfüllt werden. Dennoch gibt es derzeit Herausforderungen, die noch zu meistern sind:

- Es gibt derzeit keine Spezifikation für Persistenz. In der Praxis ist dies kein großes Problem, denn in der Regel ist JPA aus der Java-EE-Welt verfügbar. Dennoch muss man für diese Anforderung die MicroProfile-Welt verlassen. Ähnliches gilt auch für die JWT-Verwendung (JwtBuilder) und verschiedene Konfigurationen, bei denen Entwickler auf proprietäre Lösungen der Plattform ausweichen müssen.
- Die Merkmale des Standards werden in Spezifikationen wie TCK manchmal zu isoliert betrachtet. Werden mehrere MicroProfile-Merkmale, wie zum Beispiel JWT-Authentication und Metrics, zusammen verwendet, kann es zu Definitions-Lücken kommen. Gewisse Fragen der Interoperabilität, beispielsweise wie sich die JWT-Authentifizierung auf den automatischen Metrik-Endpunkt auswirkt, beziehungsweise welche Berechtigungen dafür gelten, werden bislang nicht beantwortet. Wünschenswert wären hier klare Aussagen und standardisierte Konfigurations-Möglichkeiten. Momentan löst jeder Anbieter dieses Problem auf seine eigene, proprietäre Weise.
- Es gibt einige Überschneidungen zwischen OpenAPI und Java-EE-Bean-Validation. Hier wäre es hilfreich, wenn die Constraints der Bean-Validation aus den OpenAPI-Beschreibungen generiert werden könnten. Momentan müssen diese jedoch separat festgelegt werden.

Quellen:

1 MicroProfile-2.1-Spezifikation - https://bit.ly/2HGcNWR
2 Java-EE-8-Spezifikationen - https://bit.ly/2OtGZoy
3 MicroProfile Config TCK - https://bit.ly/2FCrUhu
4 OSGi in Open Liberty - https://bit.ly/2CHvVPV
5 Nostrastockerus - https://bit.ly/2CIrlRE
6 JWT-Spezifikation - https://tools.ietf.org/html/rfc7519
7 Zipkin - https://zipkin.io/
8 Kubernetes CronJob - https://bit.ly/2LmztIN
9 Prometheus Monitoring - https://prometheus.io/
10 Prometheus Kubernetes Operator - https://bit.ly/2uxNnlo
11 Let‘s-Encrypt-Projekt - https://letsencrypt.org/
12 Java Keytool - https://bit.ly/2uxNnBU
13 OpenAPI Generator - https://openapi-generator.tech/
14 OpenAPI Swift Generator - https://bit.ly/2Wu0lwn
15 API-Zuerst-Ansatz - https://bit.ly/2TCS7jR
16 „usr:opentracingZipkin-0.31“-Merkmal https://bit.ly/2OsSgWd

JCON2019
www.jcon.one

JAVAPRO

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

Training-Day am 23. September 2019

Expo 24. - 26. September 2019

www.jcon.one

2	4	4	80+	99	800+
Konferenzen in einer	Kinos	Special Days	Speaker	Sessions	Teilnehmer

JCON 2019 PARTNER

GOLD PARTNER

ORACLE · Fast Lane Google Cloud · eclipse · MicroStream · XDEV

SILBER PARTNER

RAPIDclipse · consol Wir unternehmen IT

BRONZE PARTNER

viadee IT-Unternehmensberatung · trivago · TK Die Techniker · GEBIT Solutions

ORGANISATIONS-PARTNER

JAVAPRO · XDEVCON 2019 · XDEV

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

JCON 2019

JCON 2019

Die JCON ist die große Java Community Konferenz der JAVAPRO. Auf der JCON 2019 dreht sich alles um Core-Java, Enterprise-Java, APIs und Frameworks. Erstmals stehen in diesem Jahr 4 Special-Days auf dem Programm:

Tag 1 **Architecture Day**
MicroStream Day
Tag 2 **Cloud & Serverless Day**
Tag 3 **Agile Day**

Am Vortag der Hauptkonferenz, Montag 23. September 2019 findet erstmals ein hochkarätiger Schulungstag statt.

Java-Entwickler wollen Code sehen. Deshalb findet die JCON 2019 im hochmodernen UCI Multiplex Kino statt. Freuen Sie sich auf ein einzigartiges Live-Coding Erlebnis, das es in Deutschland nur auf der JCON gibt.

Organisiert wird die JCON 2019 von JAVAPRO zusammen mit Mitgliedern des JAVAPRO-Partner-Network.

XDEVCON 2019

Die XDEVCON 2019 ist zum dritten Mal Teil der JCON. Die XDEVCON ist seit knapp einem Jahrzehnt die Top-Konferenz für Java-Anwendungsentwicklung. Auf der XDEVCON erfahren Sie, wie Sie professionelle Business-Applikationen mit Java schneller, einfacher, produktiver und kosten-günstiger entwickeln können. Auf keiner anderen Konferenz erhalten Sie mehr Praxis-Know-how zum Thema „Schnelle Anwendugnsentwicklung mit Java“ als auf der XDEVCON 2019.

Die XDEVCON 2019 richtet sich an professionelle Anwendungsentwickler und IT- Entscheider, die businesskritische Geschäftsanwendungen mit Java entwickeln und sich dabei auf die schnelle Umsetzung neuer Features konzentrieren möchten, anstatt sich mit Low-level- Programmierung auseinandersetzen zu müssen.

JCON2019

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

Core Java

Serverside Java & Java EE

APIs & Frameworks

Serverless / Cloud Native

Microservices

Architecture

Agile

Web Development

Mobile Development

Cloud Platforms

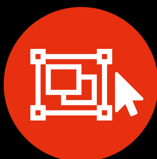
Container

DevOps

Continuous Delivery

IDE & Tools

Testing & Quality



UI & UX

Performance

Security

Big- / Fast- / Smart-Data

IoT & Embedded

Innovations

#JCON2019

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

TRAINING DAY NEU!

Montag, 23. September 2019 • 10 Power-Workshops mit hochkarätigen Trainern

Max. 20 Teilnehmer pro Workshop möglich ! - Beginn & Dauer: 09:00 bis 17:00 Uhr.

Java Performance Tuning

Trainer: **Ingo Düppe**, *CROWDCODE*

Das mit Java hochperformante, kommerzielle e-Commerce-Systeme entwickelt werden können, beweisen zahlreiche Beispiele. Doch die Optimierung von Java-Anwendungen ist nicht trivial. Aber es gibt 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. Somit lernen die Teilnehmer Schritt für Schritt, wie Performance Engpässe in Microservices aufgezeigt und gelöst werden können.

Create ultra-fast Java In-Memory Apps & Microservices with Java

Trainer: **Florian Habermann**, *MicroStream* **& Christian Kümmel**, *XDEV*

Nach Java Standard verwenden wir für die Datenverarbeitung relationale Datenbanken und JPA. Für Echtzeit(nahe)-Anwendungen (ML, KI, Automotive, Virutal Reality, IoT etc.) ist diese Kombination zu träge und für Microservices viel zu schwergewichtig. MicroStream wurde entwickelt, um dieses Problem zu lösen. MicroStream ist eine Storage-Engine, die beliebige Java Objekt-Graphen hocheffizient speichern und laden kann und dadurch völlig neue Möglichkeiten bietet. In diesem Workshop lernen Sie von A-Z wie MicroStream funktioniert, was MicroStream von Serialisierung, alten OODBMS und NoSQL DBMS unterscheidet und wie Sie damit ultraschnelle In-Memory Datenbank Apps (Abfragen in Mikrosekunden!!!) und ultraleichtgewichtige Microservices mit Daten-Persistenz mit Pure Java entwickeln können.

How to write better and effective Code in Java

Trainer: **Altug Altintas**, *Kodcu.com*

This hands-on-lab consists of twelve items, each of which conveys one rule. The rules capture practices generally held to be beneficial by the best and most experienced programmers. Topic includes: 1) Builder Patterns when faced with many constructors. 2) Avoid creating unnecessary objects – Performance issues. 3) Should we use finalizers? 4) Why implementing Hashcode and equals is very critical and how HashMap is working? 5) Functional interfaces, Prefer lambdas to anonymous classes. 6) Minimize mutability. 7) Design and document for inheritance or else prohibit it. 8) Prefer class hierarchies to tagged classes. 9) Make defensive copies when needed. 10) What are the best practices while using currency (money) in Java? 11) The try-with-resources statement. 12) Prefer for-each loops to traditional for loops.

Reactive Spring

Trainer: **Patrik Baumgartner**, *42talents*

In this Workshop, we will use Spring Framework 5 to write Functional Reactive code and will answer the following questions: What is Functional Programming? - What is Reactive Programming? - What is Functional Reactive Programming? Functional Reactive Programming is a hot trend in the Java world and also introduced in Spring Framework 5. This new paradigm allows you to effectively work with streams of data. You'll get hands-on experience with building a Reactive application to stream data leveraging the newly available Reactive data types, Spring WebFlux and Spring Data. Agenda: Introduction Reactive Streams, Publisher/Subscriber types and Reactor types - Using Spring WebFlux - Functional configuration API for Spring WebFlux - Using Spring Data MongoDB to reactively stream data - Using Reactive Types with Thymeleaf - Using Spring Security Reactive - Using Reactive RabbitMQ with Spring - Using Reactive Redis with Spring.

Line Coverage ist tot - die Jagd auf Mutationen ist eröffnet

Trainer: **Sven Ruppert**, *Vaadin*

Eine Testabdeckung von ca. 75% auf Zeilenebene ist sehr gut und kann einem schon als Grundlage dienen. Aber wie aussagekräftig ist diese Zahl? Wir werden uns in diesem Workshop mit dem Begriff des "Mutation Testing" beschäftigen und praktische Wege zum Einsatz zeigen. Wie ist die Abdeckung zu interpretieren, was kann man erreichen? Wie ist die Integration in ein bestehendes Projekt möglich und was ist bei der Erstellung der Tests zu beachten? Der Workshop wird anhand einer Vaadin Webanwendung die praktischen Möglichkeiten von Core Java bis hin zum Test einer UI aufzeigen. Wir werden uns ausschließlich innerhalb der Sprache Java bewegen. Alle Erkenntnisse sind unabhängig von Vaadin sofort im praktischen Alltag einsetzbar.

Spring Boot goes Kubernetes

Trainer: **Andreas Kruse & Sebastian Sirch**, *viadee*

Verteilte Microservice-Architekturen und agile Software-Entwicklung brauchen auch „agile Infrastruktur" – und genau diese Flexibilität bietet eine Self-Service-Plattform auf Basis von Kubernetes. Anhand einer realitätsnahen Beispielanwendung lernen Sie Schritt für Schritt, ein eigenes Docker-Image zu erstellen und dieses in ein Kubernetes-Cluster zu deployen. Mit diesem Workshop erhalten Sie einen praxisnahen Einstieg in das Kubernetes-Ökosystem und -Tooling. In Hands-On Übungen entlang eines durchgängigen Beispiels können Sie die vorgestellten Kubernetes-Konzepte am eigenen Laptop ausprobieren. Für den Workshop wird eine Cloud-Umgebung bereitgestellt. Grundkenntnisse in Docker werden vorausgesetzt.

Tame the Beast: Praktiken und Werkzeuge um den Big Ball of Mud loszuwerden

Trainer: **Matthias Gutheil & Martin Schmidt**, *itemis*

"Business as usual", "Alltagsgeschäft", "Featuritis" oder "Keine Zeit technische Schulden abzubauen." Wenn Ihr solche Argumente bei der täglichen Arbeit hört, besteht die Gefahr, dass Ihr auf den Big Ball of Mud zusteuert oder bereits angekommen seid. Diese unerwünschte Architektur zeichnet sich durch Instabilität, Starrheit und langsame Entwicklungszyklen aus. In diesem Workshop lernt Ihr praktisch, wie man die Probleme an einem Beispielsystem identifiziert und bewertet. Anschließend werden wir gemeinsam im Rahmen eines Coding Dojos das System schrittweise verbessern. Weiterhin erläutern wir wie man Architektur-Erosion verhindern kann. Hierzu gibt es bewährte Methoden und Werkzeuge.

Rapid Cross-Platform-Development mit Eclipse, Vaadin und Web-Components

Trainer: **Sebastian Späth**, *XDEV*

Ab jetzt müssen Sie Ihre UI endlich nicht mehr ständig von einem UI-Framework auf das nächste migrieren. Mit Web-Components gibt es erstmals einen Standard für UI-Komponenten, der von allen Browsern unterstützt wird. Mit Vaadin können Sie solche UIs vollständig in Java schreiben – just like Swing! Hunderte Masken von Java Experten coden und gem. Agile Iterationen vielfach abändern zu müssen, ist heutzutage jedoch viel zu teuer. In diesem Workshop lernen Sie von A-Z, wie Sie mit Eclipse RapidClipse völlig individuelle Frontends auf Basis von Vaadin und Web-Components in Rekordzeit designen, wie Sie mit den neuen Hibernate-Tools auf Datenbanken zugreifen und fertige Projekte für das Web, Mobile und den Desktop deployen. Sie werden fasziniert sein wie schnell und einfach das

Mit testgetriebener Software-Entwicklung zu einer hexagonalen Architektur

Trainer: **Daniel Haftstein**, *itemis*

Hexagonale Architekturen (nach Definition von Alistair Cockburn) bieten ein flexibles Modell zur Entkopplung der Businesslogik von äußeren Einflüssen. In diesem Workshop stelle ich Vorgehensweisen vor, mit denen sich eine hexagonale Architektur testgetrieben entwickeln lässt. Inhalte: Designprinzipien als Unterstützung für TDD - Strategien zum Umgang mit externen Bibliotheken - Mocking und Alternativen - Akzeptanztests als Erweiterung des klassischen TDD-Zyklus - Vergleich zwischen Outside-In und Inside-Out TDD Wir starten mit einem (kurzem) theoretischem Teil, im Hauptteil des Workshops wird eine kleine Applikation testgetrieben entwickelt. Dabei arbeiten wir uns von außen über Akzeptanztests zu einem "Walking Skeleton" vor um anschließend die innere Businesslogik nach klassischem TDD zu entwickeln.

Continuous Deployment on AWS: Make Releasing the most boring Part of your Job

Trainer: **Steffen Grunwald & Dirk Fröhler**, *Amazon Web Services EMEA*

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.

JCON2019

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

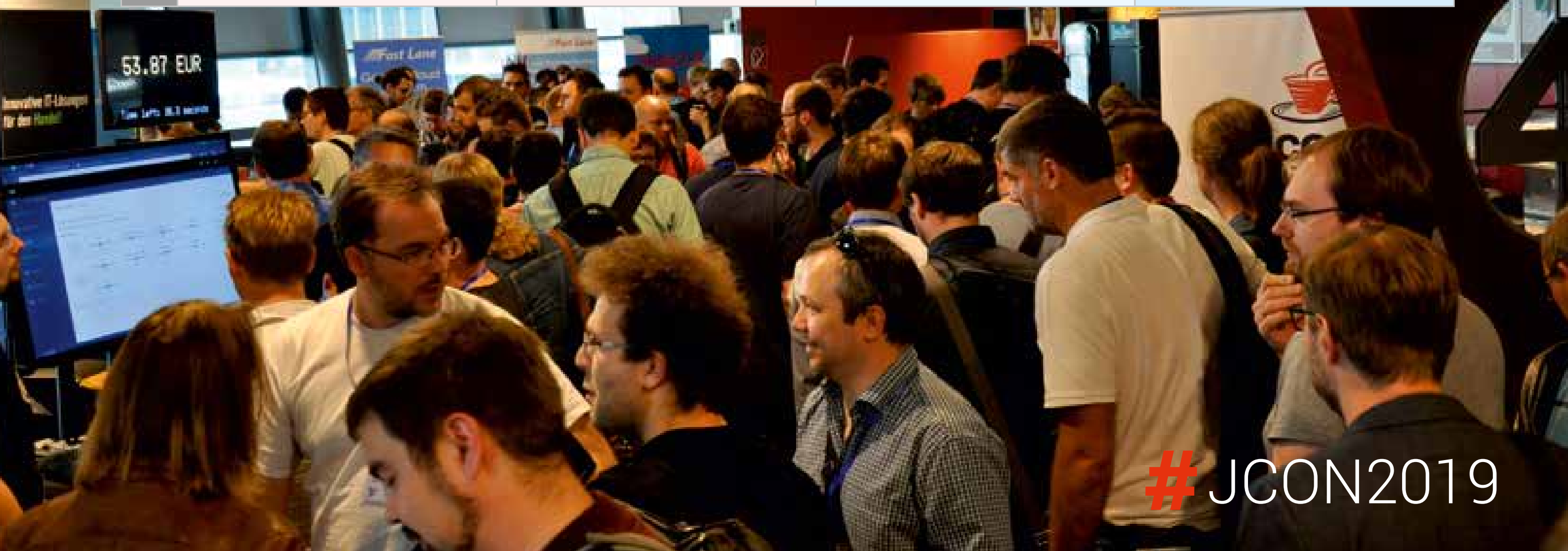
JCON 2019

SESSIONPLAN - TAG 1

Dienstag, 24. September 2019 • Architecture Day • MicroStream Day

Beginn & Dauer: 9:00 bis 19:00 Uhr. Checkin vor Ort ab 7:30 Uhr.

	Hauptkonferenz	Hauptkonferenz	Architecture Day	MicroStream Day
7:30	**Checkin**			
9:00	**Neues von Java und dem JDK** Michael Vitz, *INNOQ*	**Concurrency in Java - A Tour of the Java Councurrency API** Christian Heitzmann, *SimplexaCode*	**API Management: Was ist das und wer braucht es?** Dr. Roger Butenuth, *codecentric*	**Creating ultra-fast Java In-Memory Applications and Microservices with MicroStream** Markus Kett, *MicroStream*
10:00	Keynote: **Enterprise Java Innovation #slideless** Adam Bien			
11:15	**Vernünftige Java Praktiken #slideless** Adam Bien	**Moderne Integration-Tests mit Test-Containers** Daniel Krämer, *anderScore* Maik Wolf, *anderScore*	**Clean Architecture mit Java und Spring** Tom Hombergs, *adesso*	**Java In-Memory Database-Apps with MicroStream - Getting Started** Florian Habermann, *MicroStream*
12:00	**Mittagspause & Expo-Time - 45 Minuten**			
13:00	**REST, aber richtig - mit Links** Thomas Bröll, *Trivadis*	**Die sieben Security-Sünden agiler Projekte** Christian Schneider, *Schneider IT-Security*	**Event Sourcing - Wahrscheinlich machst du es falsch** David Schmitz, *Senacor Technologies*	**In-Memory Objekt-Graphen effizient nutzen** Christian Kümmel, *XDEV*
14:00	**In 10 Schritten zum unwartbaren Code: Tägliche Anti-Pattern** Andreas Günzel, *EXXETA*	**Hilfe, meine Anwendung ist zu langsam. Was tun?** Prof. Jörg Hettel, *Hochschule Kaiserslautern*	**Wie komme ich zu einer Architektur in zwei Wochen? - Architekturbewertung in agilen Projekten** Tobias Voß, *viadee*	**A Modern Fairy Tale: Java Serialization** Steve Pool, *IBM*
15:00	**Hitchhiker Guide to the Java Performance** Ingo Düppe, *CROWDCODE*	**Leichtgewichte Software-Architektur mit Architecture Decision Records und Qualitäts-Szenarien** Johannes Dienst, *DB Systel*	**Deploy, monitor, and take Control of your Microservices with MicroProfile** Rudy De Busscher, *Payara*	**Bullet-proof Java Serialization and super-fast Object-Graph Communication** Florian Habermann, *MicroStream*
15:45	**Pause & Expo-Time - 30 Minuten**			
16:15	**Testfalle richtiges und effizientes Software-Testen** Marco Schulz, *Freelancer* Joachim Reiter	**Funktionales Programmieren in Java jenseits der Standard API** Jan Hauer, *EXXETA*	**Ihh, wir haben einen Big Ball of Mud! Wie konnte es dazu kommen, was tun wir, wie verhindern wir dies in der Zukunft?** Matthias Gutheil, *itemis*	**MicroStream Best-Practice - Projekt-Setup, Konfiguration, Backup** Christian Kümmel, *XDEV*
17:15	**Automatisierte Tests: Tipps zum effizienten Scheitern** Sascha Bleidner, *itemis*	**Spring Boot entzaubert** Michael Vitz, *INNOQ*	**Hallo REST API, wie geht es jetzt weiter?** Kai Tödter, *Siemens*	**60% less Memory Usage with OpenJ9 JVM** Steve Poole, *IBM*
18:15	**Identify Technical and Organizational Debts by Browsing the History of your Code?** Martin Schmidt, *itemis*	**Boot everything? Micronaut - Eine Alternative zu Spring Boot?** Jan Thewes, *it-economics*	**Applied Architecture Analysis - Experiences from utilizing aim42** Hendrik Bündner, *itemis*	**Challenges with MicroStream - Was ist, wenn sich meine Klassen ändern?** Christian Kümmel, *XDEV*



DIE GROSSE JAVA COMMUNITY KONFERENZ

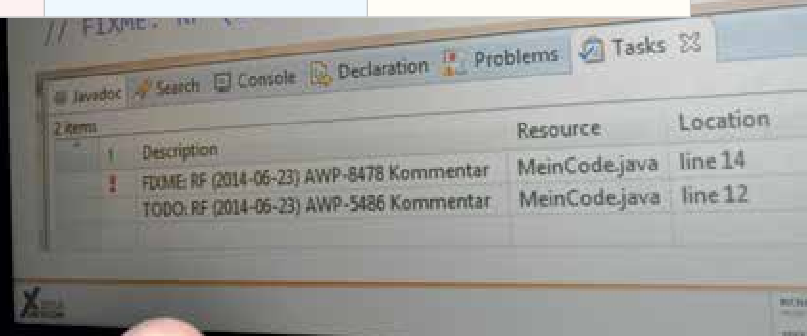
24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

SESSIONPLAN - TAG 2

Mittwoch, 25. September 2019 • Cloud & Serverless Day • XDEVCON 2019

Beginn & Dauer: 9:00 bis 19:00 Uhr. Checkin vor Ort ab 7:30 Uhr.

	Hauptkonferenz	Hauptkonferenz	Cloud & Serverless Day	XDEVCON 2019
7:30	**Checkin**			
9:00	**Jakarta EE - Past, Present and Future** Christian Kaltepoth, *ingenit*	**Zero-Downtime Deployment with Kubernetes, Spring Boot and Flyway** Nicolas Frankel, *Exoscale*	**Ich will doch nur entwickeln, oder: Was ich als Entwickler über die Cloud wissen sollten** Mario Rutz, *Gruner + Jahr*	**RapidClipse X – Rapid Cross-Platform-Development mit Web-Components** Markus Kett, *MicroStream*
10:00	**Immutable Java** Nicolai Mainiero, *sidion*	**On the Hunt for the Holy Graal: One VM to rule them all?** Dr. Marco Bungart, *ConSol*	**Cloud Teaser – eine Strategie zur Nutzung der Cloud** Thomas Mitzka, *Fast Lane*	**Einführung in Web-Components** Michael Vitz, *INNOQ*
11:00	Keynote: **Serverless Computing with Fn Project and Functions** David Delabassèe, Oracle			
12:00	**Mittagspause & Expo-Time - 60 Minuten**			
13:00	**The Java Module System in Practice** Serban Iordache, *SCOOP Software*	**Highlights from Java 10, 11, 12 and 13 and the Future of Java** Vadym Kazulkin, *ip.plabs*	**Docker Best Practices für Microservices** Thomas Bröll, *Trivadis*	**Vaadin Flow - Web-Anwendung in Core Java ohne HMTL / CSS** Sven Ruppert, *Vaadin*
14:00	**Become a Chrome DevTools Hero** Hendrik Dammers, *CROWDCODE*	**Quarkus: Supersonic Subatomic Java** Peter Palaga, *Red Hat*	**Container-Native Applikations-Entwicklung in der Cloud** Wolfgang Weigend, *Oracle*	**Next-Generation Web-Frontends mit Web-Components und Vaadin in Rekordzeit entwickeln** Sebastian Späth, *XDEV*
15:00	**Creating Ultra-fast Realtime Apps and Microservices with Java** Markus Kett, *MicroStream*	**MicroProfile: Java EE meets Microservices** Lars Röwekamp, *open knowledge*	**PaaSts ?! Live Migration eines JavaEE Monolithen zu Cloud Plattformen** Thorsten Jakoby, *Novatec Consulting* Matthias Häussler, *Novatec Consulting*	**RapidClipse X Deep-dive - Vaadin Convenience Framework in Action** Sebastian Späth, *XDEV*
15:45	**Pause & Expo-Time - 30 Minuten**			
16:15	**How JSR 385 could have saved the Mars Climate Orbiter** Werner Keil, *Creative Arts & Technologies* Thodoris Bais	**It's a JDK Jungle out there** Wolfgang Weigend, *Oracle*	**"DDOS as a Service": Using JMeter, Docker and AWS to Load-test Your Application** Evertson Croes, *Luminis*	**Power-Frontends mit Web-Components, Vaadin und RapidClipse X** Sebastian Späth, *XDEV*
17:15	**Echte Unabhängigkeit für Microservices - Unit-Tests mit Wiremock** Korbinian Reffler, *Mischok*	**Technologie-Entscheidungen in selbstorganisierten Teams** Konstantin Diener, *cosee*	**Pitfalls for Serverless Computing** Frank Pientka, *MATERNA*	**Direkter System- und Gerätezugriff für Mobile Apps mit Java** Florian Habermann, *MicroStream*
18:15	**"Vollständige Git Flow Automation mit Jenkins, Docker, Rancher & Co."** Marcus Noerder-Tuitje, *CROWDCODE*	**Wie passen Serverless & Autonomous zusammen?** Volker Linz, *Oracle*	**Size does matter! The Battle of JVM-Micro-Frameworks** Christian Schwörer, *Novatec Consulting*	**Navigations-Konzepte für moderne Web-Frontends** Sebastian Späth, *XDEV*



#JCON2019

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

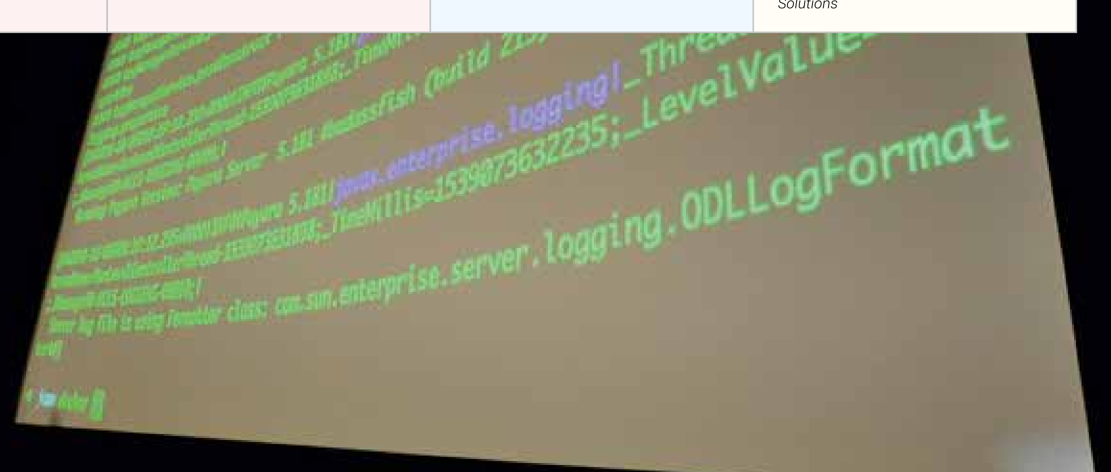
JCON 2019

SESSIONPLAN - TAG 3

Donnerstag, 26. September 2019 • Agile Day • XDEVCON 2019

Beginn & Dauer: 9:00 bis 19:00 Uhr. Checkin vor Ort ab 7:30 Uhr.

	Hauptkonferenz	Hauptkonferenz	Agile Day	XDEVCON 2019
7:30	**Checkin**			
9:00	**KI in der Praxis - Machine Learning auf Prozessdaten mit Java** Mario Micudaj, *viadee*	**Mastering the API Hell - Kompatibilität dank Consumer Driven Contract Testing** Nikolai Neugebauer, *Digital Frontiers* Florian Pfleiderer, *Digital Frontiers*	**Ist unser Team heute besser als gestern? - Continuous Improvement messen** Muhammad Ali Kazmi, *itemis*	**i18n - Internationalisierung von Web-Anwendungen** Christian Kümmel, *XDEV*
10:00	**MVC 1.0 - Endlich am Ziel** Christian Kaltepoth, *ingenit*	**Test-Driven-Development sucks! But that does not have to be** Christian Fischer, *agile dojo*	**Wie man Code Reviews gegen die Wand fährt** Philipp Hauer, *Spreadshirt*	**Maven Grundlagen und Best-Practice** Richard Fichtner, *XDEV*
11:00	Keynote: **Beyond Agile - Doing the right Thing in a post-agile World** Uwe Friedrichsen, codecentric			
12:00	**Mittagspause & Expo-Time - 60 Minuten**			
13:00	**JUnit 5 - Testing in the Modular World** Christian Stein, *Micromata*	**Money, Money, Money, can be funny with JSR 354** Werner Keil, *Creative Arts & Technologies* Anatole Tresch	**Wenn agil schmerzt – wie sollten Unternehmen sein, damit Agil wirklich funktioniert** Manuel Marsch, *KEGON* Michaela Jäger, *KEGON*	**Migration von RapidClipse Projekten auf RapidClipse X** Christian Kümmel, *XDEV*
14:00	**Bug-Mining – KI als Hilfe in der Software-Entwicklung einsetzen** Christoph Meyer, *viadee*	**Test von Web-Anwendungen - auf das Werkzeug kommt es an** Dr.-Ing. Dehla Sokenou, *GEBIT* Roland Brill, *GEBIT*	**Metriken für agile Software-Entwicklung** Richard Fichtner, *XDEV*	**Full Skill Developer - Was Entwickler außer Coden noch können sollten** Konstantin Diener, *cosee*
15:00	**Java EE + MicroProfile - das bessere Spring Boot?** Maik Wolf, *anderScore*	**Vom UI bis zum Backend im Turbogang** Veikko Krypczyk, *LARInet*	**Keeping up with Upstream** Nicolas Byl, *codecentric*	**Using Web-Components with Frontend Frameworks** A.Mahdy AbdelAziz, *Vaadin*
15:45	**Pause & Expo-Time - 30 Minuten**			
16:15	**TestContainers + JUnit5 = elegant end-to-end Tests for Microservices** Nikolay Kuznetsov, *Zalando*	**How to build an In-house Architecture Community? - Best Practices** Frank Pientka, *MATERNA*	**New Way of Working – What's in it for me?** Simione Engelhard, *[lernglust]*	**REST Webservices Grundlagen** Richard Fichtner, *XDEV*
17:15	**Vom Big Ball of Mud zu DDD – Der Weg zu einem wohl-strukturierten Monolithen** Christian Nockemann, *viadee*	**Graphische Reports nachhaltig entwickeln** Julius Mischok, *Mischok*	**Tempo und Ausdauer in Software-Projekten** Michael Rohleder, *QAware* Harald Störrle, *QAware*	**Corporate Identity und Styling für moderne Web-Frontends** Sebastian Späth, *XDEV*
18:15	**Java Class File meets JVM** Christian Packenius, *Provinzial Rheinland Versicherungs AG*	**Going Web Native** A.Mahdy AbdelAziz, *Vaadin*	**Remote Mob Programming** Dr. Simone Harrer, *INNOQ* Jochen Christ, *INNOQ*	**It's all about the Domain, Honey – Fachliche Architektur für Java mit DDD** Henning Schwentner, *WPS – Workplace Solutions*



#JCON2019

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

AGENDA

Montag, 23. September	Dienstag, 24. September	Mittwoch, 25. September	Donnerstag, 26. September
	JCON 2019		
Training Day 9:00 - 17:00 Uhr 10 hochkarätige Tages-Schulungen	**JCON Haupt-Konferenz** 9:00 - 19:00 Uhr		
	EXPO 9:00 - 19:00 Uhr		
	Special Days		
	Architecture Day 9:00 - 19:00 Uhr	**Cloud & Serverless Day** 9:00 - 19:00 Uhr	**Agile Day** 9:00 - 19:00 Uhr
		XDEVCON 2019 - Konferenz für Rapid-Cross-Platform-Development	
	MICROSTREAM DAY 9:00 - 19:00 Uhr	**XDEVCON 2019** 9:00 - 19:00 Uhr	**XDEVCON 2019** 9:00 - 19:00 Uhr

TICKETS

1-TAGES-PASS	1-TAGES-PASS	2-TAGES-PASS	3-TAGES-PASS	4-TAGES-PASS
Mo **23. Sept.**	**Di / Mi / Do** **24. / 25. / 26. Sept.**	**Di - Mi / Mi - Do** **24. - 25. / 25. - 26. Sept.**	**Di - Do** **24. - 26. Sept.**	**Mo - Do** **23. - 26. Sept.**
Training Day 10 hochkarätige Tages-Schulungen	Haupt-Konferenz Tag 1: Architecture Day MicroStream Day Tag 2: Cloud & Serverless Day Tag 3: Agile Day XDEVCON 2019	Haupt-Konferenz Tag 1: Architecture Day MicroStream Day Tag 2: Cloud & Serverless Day Tag 3: Agile Day XDEVCON 2019	Alle Sessions der JCON 2019 XDEVCON 2019	Alle Sessions der JCON 2019 XDEVCON 2019 Training Day
499 €	**249 €**	**449 €**	**549 €**	**999 €**

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

Jetzt anmelden unter: www.jcon.one

Haben Sie Fragen zur JCON 2019 ? Wir beraten Sie gerne!

Fon: +49 (0)6196 – 204801 – 0 | E-Mail: info@jcon.one

#JCON2019

#JAVAPRO #Microservices #DomainDrivenDesign

Monolithen mit DDD aufschneiden

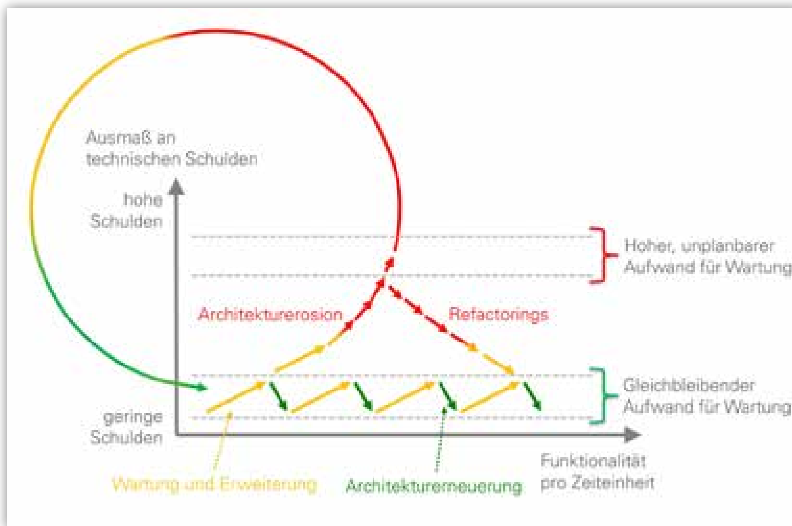
Fast jedes Softwaresystem wird mit guten Vorsätzen, aber unter schwierigen Bedingungen entwickelt: Knappe Zeitvorgaben zwingen uns, schnelle Lösungen – Hacks – zu programmieren. Unterschiedliche Qualifikationen im Entwicklungsteam führen zu Code in ebenso unterschiedlicher Güte. Alter Code, den keiner mehr kennt, muss angepasst werden und vieles mehr. All das führt zu schlechtem, verknäultem Code, dem sogenannten Monolithen, der die Entwicklungskosten in Zukunft in die Höhe treibt und den Entwicklungsteams schlaflose Nächte bereitet. Mit Domain-Driven-Design (DDD) haben wir ein Werkzeug an der Hand, um solche Monolithen Schritt für Schritt zu zerlegen und wieder in den Bereich der beherrschbaren Wartungskosten zu bringen.

Autor:

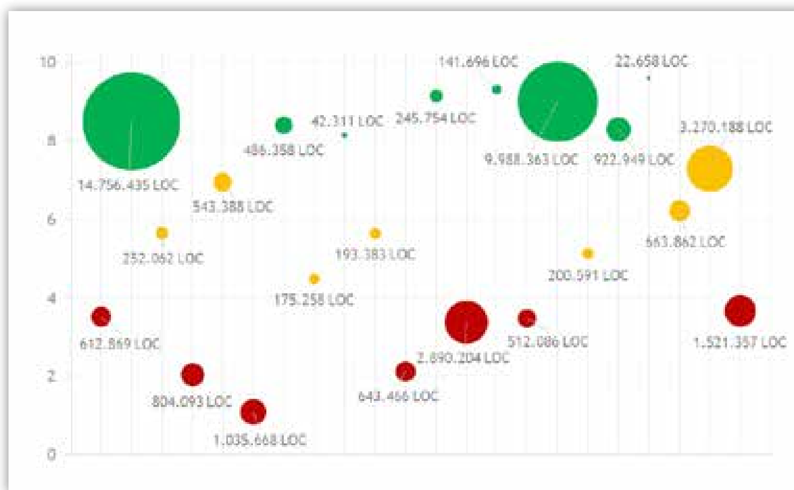
Carola Lilienthal studierte von 1988 bis 1995 Informatik an der Universität Hamburg und promovierte 2008 in Informatik bei Christiane Floyd und Claus Lewerentz an der Universität Hamburg. Dr. Carola Lilienthal ist Geschäftsführerin der WPS – Workplace Solutions GmbH und verantwortet dort den Bereich Softwarearchitektur. Seit 2003 analysiert Dr. Carola Lilienthal in ganz Deutschland Architekturen in Java, C#, C++, ABAP und PHP und berät Entwicklungsteams, wie sie die Langlebigkeit ihrer Softwaresysteme verbessern können. 2015 hat sie ihre Erfahrungen aus über hundert Analysen in dem Buch „Langlebige Softwarearchitekturen" zusammengefasst. Besonders am Herzen liegt ihr die Ausbildung von Softwarearchitekten, weshalb sie aktives Mitglied bei iSAQB, dem International Software Architecture Quality Board e.V., ist und ihr Wissen regelmäßig auf Konferenzen, in Artikeln und bei Schulungen weitergibt.

Entstehen von Monolithen

Gehen wir einmal davon aus, dass wir zu Beginn unseres Softwareentwicklungsprojekts eine qualitativ hochwertige, modulare Architektur entworfen haben. Dann kann man hoffen, dass sich unser Softwaresystem am Anfang gut warten lässt. In diesem Anfangsstadium befindet sich unser Softwaresystem also in dem Korridor geringer technischer Schulden mit gleichbleibendem Aufwand für die Wartung **(Abb. 1 – grüne Klammer)**. Je länger unser System lebt, desto mehr erweitern wir es und fügen neue Funktionalität hinzu. Bleibt uns keine Zeit bei diesen Erweiterungen auf den Erhalt der modularen Architektur zu achten, so erodiert diese Architektur mit der Zeit immer mehr. Wir nehmen immer mehr technische Schulden auf **(Abb. 1 – gelbe Pfeile die rot werden)** und nähern uns immer mehr einem verknäulten Monolithen mit hohem, unplanbarem Aufwand in der Wartung. **(Abb. 1)** macht diesen langsamen Verfall dadurch deutlich, dass die roten Pfeile immer kürzer werden. Pro Zeit oder Budget schaffen wir immer weniger Funktionalität. Folgefehler sind immer schwerer nachvollziehbar bis zu dem Punkt, an dem jede Änderung zu einer schmerzhaften Anstrengung wird. Besser



Entwicklung und Effekt von technischen Schulden. (Abb. 1)



Modularity-Maturity-Index (MMI). (Abb. 2)

wäre es, wenn unser Management und wir akzeptieren würden, dass Softwareentwicklung ein ständiger Lernprozess ist, bei dem der erste Wurf einer Lösung selten der Endgültige ist. Eine Erweiterung im Korridor mit gleichbleibendem Aufwand für Wartung führt also immer auch erst einmal zu mehr Schulden **(Abb. 1 – gelber Pfeil „Wartung und Erweiterung")**. Die Überarbeitung der Architektur **(Abb. 1 – grüne Pfeile)** muss in regelmäßigen Abständen durchgeführt werden, wenn vermieden werden soll, dass ein Monolith entsteht. Folgt man diesem Prinzip, so läuft die Entwicklung in einer stetigen Folge von Erweiterung und Refactoring ab. Kann ein Team diesen Zyklus von Erweiterung und Refactoring dauerhaft verfolgen, so wird das System im Korridor geringer technischer Schulden bleiben **(Abb. 1 – gelbe und grüne Pfeile im Korridor)**.

Um für Monolithen eine Bewertungsmöglichkeit zu schaffen, wie gut der Sourcecode auf eine fachliche Zerlegung vorbereitet ist, haben wir in den vergangenen Jahren den Modularity-Maturity-Index (MMI) entwickelt. In **(Abb. 2)** ist eine Auswahl von 21 Softwaresystemen dargestellt, die in einem Zeitraum von fünf Jahren analysiert wurden (X-Achse). Für jedes System ist die Größe in Lines-of-Code dargestellt (Größe des Punktes) und die Modularität auf einer Skala von 0 bis 10 (Y-Achse). Die Farbgebung rot, gelb und grün entspricht dabei den jeweiligen Bereichen der technischen Schulden aus **(Abb. 1)**.

Liegt ein System in der Bewertung zwischen 8 und 10, so ist es im Inneren bereits modular aufgebaut und wird sich mit wenig Aufwand fachlich zerlegen lassen. Systeme mit einer Bewertung zwischen 4 und 8 haben gute Ansätze, aber hier sind einige Refactorings notwendig, um die Modularität zu verbessern. Systeme unterhalb der Marke 4 würde man nach Domain-Driven-Design als Big-Ball-of-Mud bezeichnen. Hier ist kaum fachliche Struktur zu erkennen und alles ist mit allem verbunden. Solche Systeme sind nur mit sehr viel Aufwand in fachliche Module zerlegbar.

Ist man erst einmal im Korridor hoher, unplanbarer Wartung **(Abb. 1)** oder beim MMI im Range 0-4 angelangt **(Abb. 2)**, gibt es zwei Möglichkeiten, um aus dem Dilemma wieder heraus zu kommen: Das System wird durch ein neues ersetzt **(Abb. 1 - Zyklus)** oder der Monolith wird refactored **(Abb. 1 – rote und gelbe Pfeile abwärts)**. Beide lassen sich mit DDD unterstützen. In diesem Artikel geht es um die zweite Möglichkeit: Wie können wir den Monolithen mit DDD zerlegen, so dass er für unsere Entwickler wieder beherrschbar wird?

Typische Architektur eines Monolithen

Die meisten Monolithen, mit denen die Autorin heutzutage zu tun hat, haben eine Schichtenarchitektur, wie sie Anfang der 2000er Jahre en vogue war. Ende der 90er Jahre war das häufigste Problem, mit dem sich Softwarearchitekten herumgeschlagen haben, dass der Sourcecode von der Business-Logik, dem User-Interface und der Datenbankzugriffschicht fröhlich gemischt wurden. Insofern war die Maßgabe, dass Softwaresysteme technisch geschichtet werden sollten, ein wichtiger sinnvoller Schritt.

In **(Abb. 3)** ist eine schematische Schichtenarchitektur dargestellt, wie sie in dieser Zeit typischerweise verwendet wurde. Eine solche Schichtenarchitektur führt erst einmal grundsätzlich dazu, dass der fachliche Sourcecode (Application und Domain) vom technischen Sourcecode (User-Interface und Datenbankzugriff) getrennt wird. Es entsteht also eine technische Ordnung im System.

Schematische Darstellung einer Schichtenarchitektur. (Abb. 3)

Fachliche Module im Monolithen. (Abb. 4)

Für Systeme, die von einem schlagkräftigen Entwicklungsteam allein bearbeitet werden können, reicht eine Schichtenarchitektur aus. Unter einem schlagkräftigen Entwicklungsteam versteht man ein Team, das eine Größe hat, die bei Urlaub und Krankheit nicht zu Stillstand führen, also ab drei Personen, und bei dem die Kommunikation beherrschbar bleibt, also nicht mehr als sieben bis acht Personen, die sich gegenseitig auf dem Stand der Dinge halten müssen. Ein solches Team sollte ein Stück Software, ein Modul, zum (Weiter)-Entwickeln bekommen, dass unabhängig vom Rest des Systems ist. Denn nur wenn das Modul unabhängig ist, kann das Team eigenständig Entscheidungen treffen und ohne auf andere Teams und deren Zulieferungen zu warten, sein Modul weiterentwickeln.

Die meisten Systeme, die seit Anfang der 2000er Jahre entwickelt werden, haben diese Größe längst überschritten und müssen weiter zerlegt werden. DDD empfiehlt hier, eine Aufteilung anhand von fachlichen Kriterien zu wählen: Den Monolithen also quer zu den technischen Schichten aufzuschneiden (Abb. 4). Die so entstehenden fachlichen Module werden in DDD Bounded-Context genannt.

Diese Zerlegung weist viele Vorteile auf. Denn neben der Unabhängigkeit der Teams und der modulareren und damit leichter verständlichen Struktur kann das System, wenn man es entlang der fachlichen Module in einzelne Microservices zerlegt, auch skalieren.

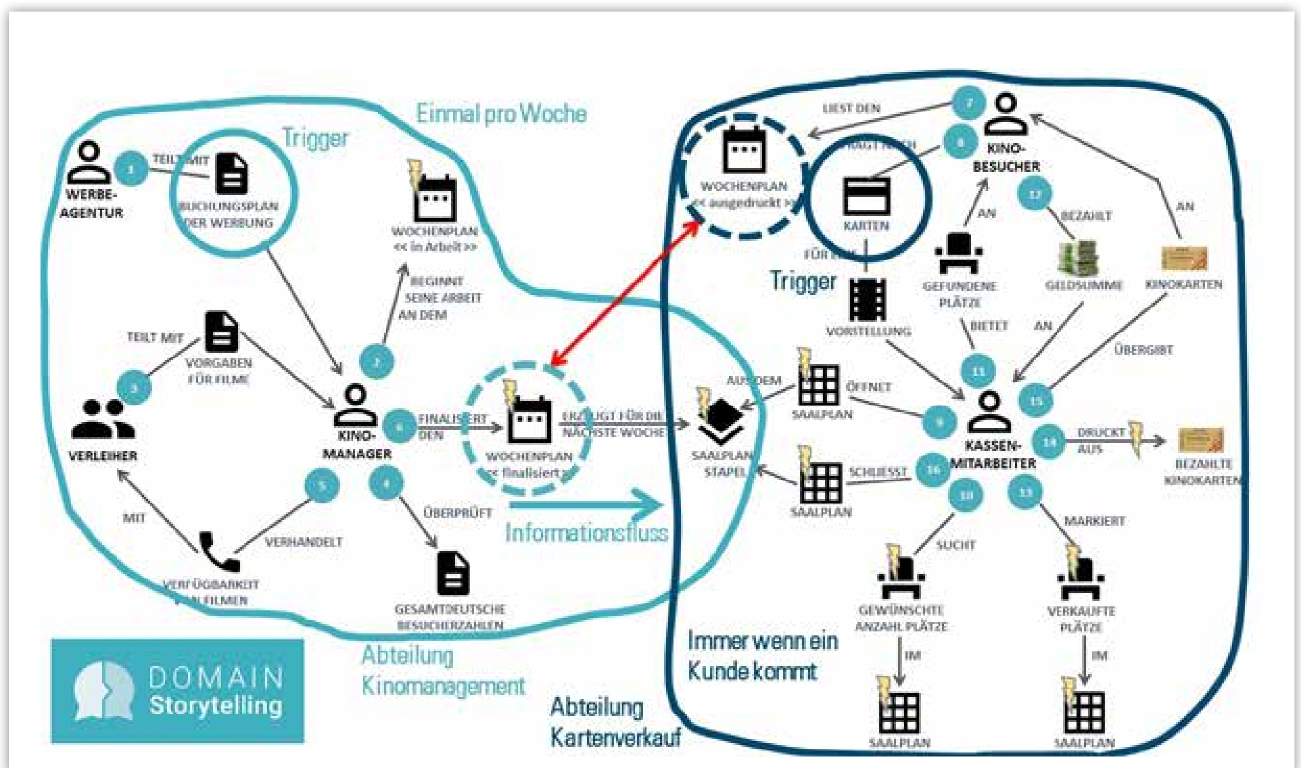
Überblicks-Domain-Story für ein Programmkino. (Abb. 5)

Fachliche Zerlegung

Um eine gute fachliche Zerlegung zu finden, hat es sich in Projekten als sinnvoll erwiesen, den Monolithen und die in ihm möglicherweise vorhandene Struktur erst einmal beiseite zu legen und sich noch einmal grundlegend mit der Fachlichkeit, also der Aufteilung der Domäne in Subdomänen zu beschäftigen. In der Regel startete man damit, zusammen mit den Anwendern und Fachexperten einen Überblick über die Domäne zu verschaffen. Das kann entweder mit Event-Storming oder mit Domain-Storytelling gemacht werden - zwei Methoden, die für Anwender und Entwickler gleichermaßen gut verständlich sind. In **(Abb. 5)** ist eine Domain-Story zu sehen, die mit den Anwendern und Entwicklern eines kleinen Programmkinos erstellt wurde. Die grundsätzliche Frage, die bei der Modellierung gestellt wurde, ist: Wer macht was, womit, wozu? Nimmt man diese Frage als Ausgangspunkt, so lässt sich in der Regel sehr schnell ein gemeinsames Verständnis der Domäne erarbeiten.

Als Personen bzw. Rollen oder Gruppen sind in dieser Domain-Story erkennbar: die Werbeagentur, der Kinomanager, der Verleiher, der Kassenmitarbeiter und der Kinobesucher. Die einzelnen Rollen tauschen Dokumente und Informationen aus, wie den Buchungsplan der Werbung, die Vorgaben für Filme und die Verfügbarkeit von Filmen. Sie arbeiten aber auch mit Gegenständen aus ihrer Domäne, die in einem Softwaresystem abgebildet sind: der Wochenplan und der Saalplan. Diese computergestützten Gegenstände sind mit einem gelben Blitz in der Domain-Story markiert. Die Überblicks-Domain-Story beginnt links oben mit der Ziffer 1, wo die Werbeagentur dem Kinomanager den Buchungsplan mit der Werbung mitteilt, und endet bei Ziffer 16, wenn der Kassenmitarbeiter den Saalplan schließt. An diesem Überblick lassen sich verschiedene Indikatoren erklären, die beim Schneiden einer Domäne helfen:

- **`Abteilungsgrenzen`** oder verschiedene Gruppen von Domänenexperten deuten darauf hin, dass die Domain-Story mehrere Subdomänen enthält. In genanntem Beispiel könnte man sich eine Abteilung Kinomanagement und eine Abteilung Kartenverkauf vorstellen **(Abb. 6)**.
- Werden **`Schlüsselkonzepte`** der Domäne von den verschiedenen Rollen **`unterschiedlich verwendet oder definiert`**, so deutet dies auf mehrere Subdomänen hin. In dem Beispiel wird das Schlüsselkonzept Wochenplan vom Kinomanager deutlich umfangreicher definiert als der ausgedruckte Wochenplan, den der Kinobesucher zu Gesicht bekommt. Für den Kinomanager enthält der Wochenplan neben den Vorstellungen in den einzelnen Sälen auch die geplante Werbung, den Eisverkauf und die Reinigungskräfte. Diese Informationen sind für den Kinobesucher irrelevant **(Abb. 6 – gestrichelte Kreise)**.
- Enthält die Überblicks-Domain-Story Teilprozesse, die von verschiedenen **`Triggern`** ausgelöst werden und in unterschiedlichen **`Rhythmen`** ablaufen, dann könnten diese Teilprozesse eigene Subdomänen bilden **(Abb. 6 – durchgezogene Kreise)**.
- Gibt es im Überblick Prozessschritte, an denen **`Information nur in eine Richtung`** läuft, könnte diese Stelle ein guter Ansatzpunkt für einen Schnitt zwischen zwei Subdomänen sein **(Abb. 6 – hellblauer Pfeil)**.



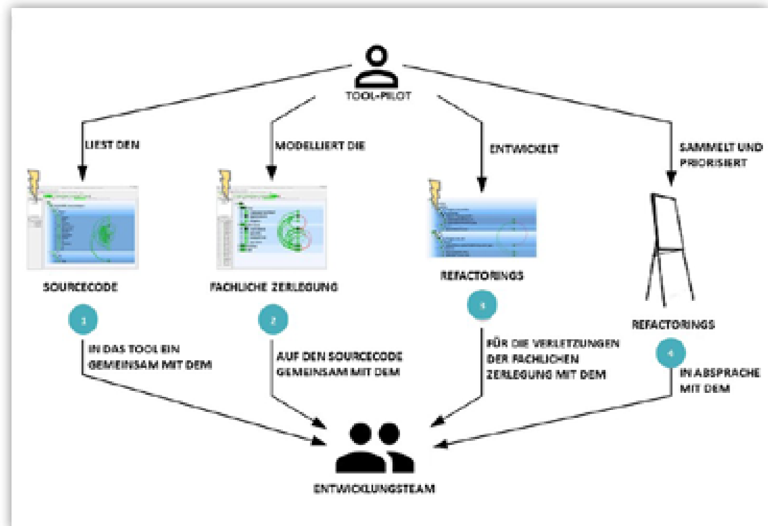
Überblicks-Domain Story mit Subdomänen-Grenzen. (Abb. 6)

Für echte große Anwendungen in Unternehmen sind die Überblicks-Domain-Stories in der Regel deutlich größer und umfassen mehr Schritte. Sogar bei diesem kleinen Programmkino fehlen im Überblick die Eisverkäufer und das Reinigungspersonal, die sicherlich auch mit der Software interagieren werden. Die Indikatoren, nach denen man in seiner Überblicks-Domain-Story suchen muss, gelten allerdings sowohl für kleine als auch für größere Domänen.

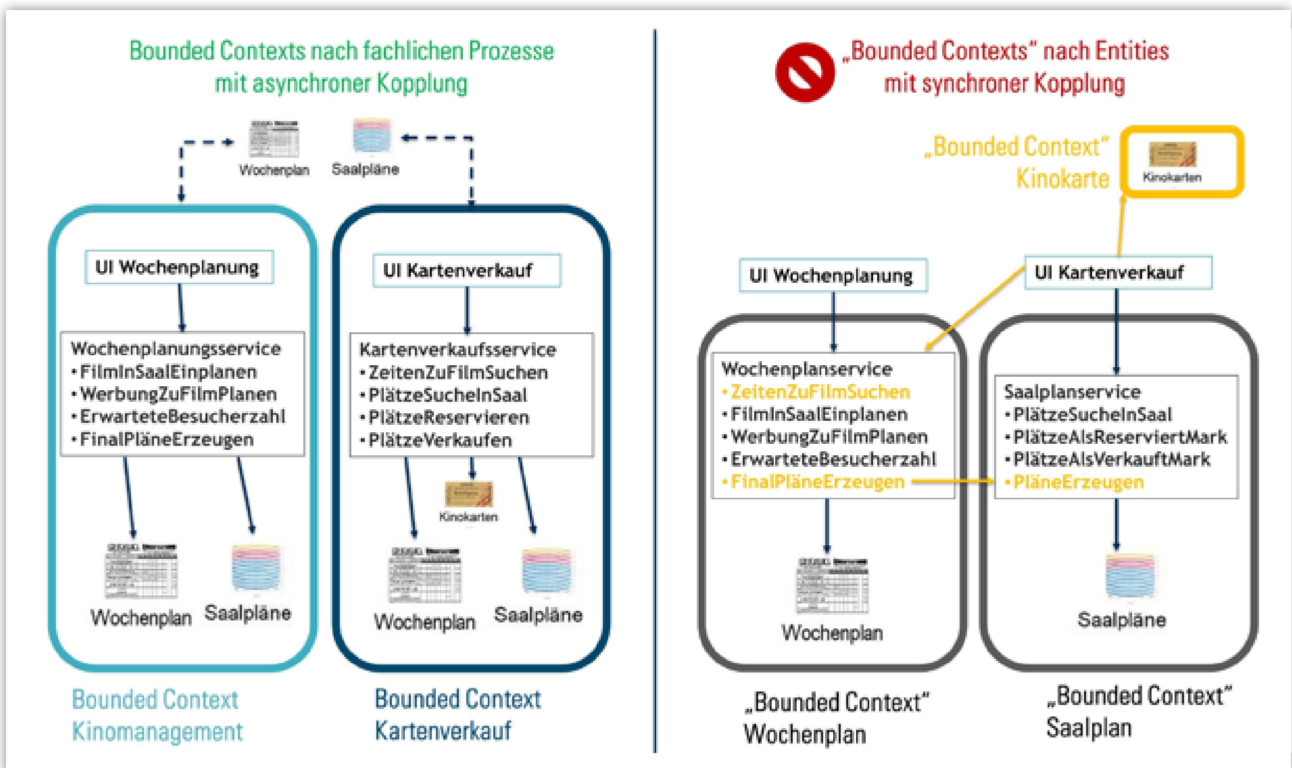
Übertragung auf den Monolithen

Mit der fachlichen Aufteilung in Subdomänen im Rücken können wir uns nun wieder dem Monolithen und seinen Strukturen zuwenden. Bei dieser Zerlegung werden Architekturanalyse-Tools eingesetzt, die es erlauben die Architektur im Tool umzubauen und Refactorings zu definieren, die für den echten Umbau des Sourcecodes notwendig sind. Hier eignen sich unterschiedliche Tools: der Sotograph, der Sonargraph, Structure 101, Lattix, Teamscale, Axivion-Bauhaus-Suite und bestimmt noch einige weitere.

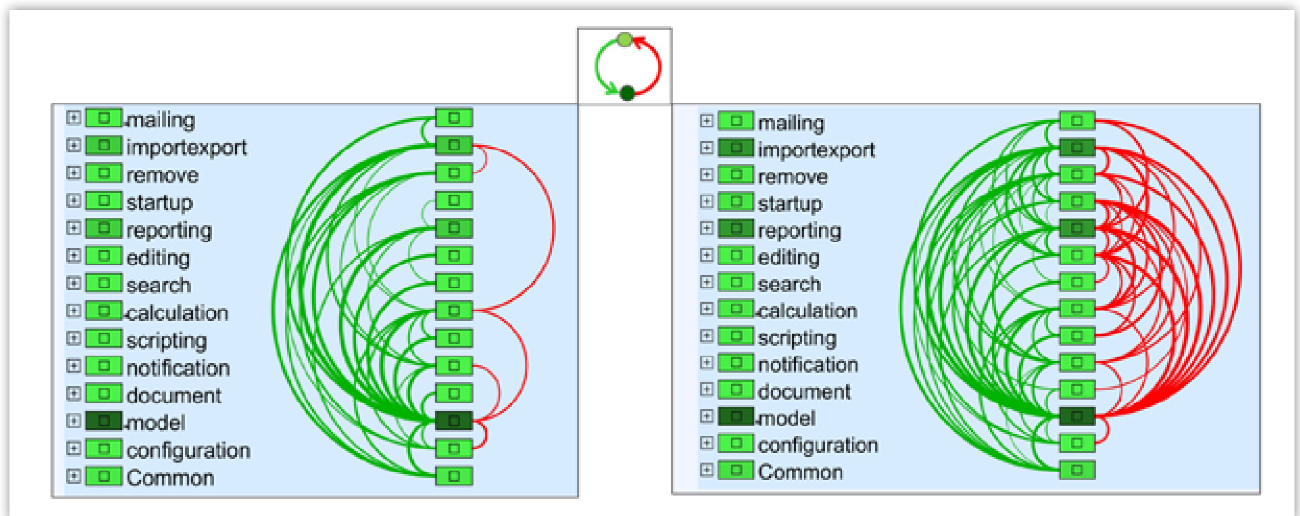
In **(Abb. 7)** sieht man, wie die Zerlegung des Monolithen mit einem Analyse-Tool durchgeführt wird. Die Analyse wird von einem Tool-Pilot, der sich mit dem jeweiligen Tool und der bzw. den eingesetzten Programmiersprachen auskennt, gemeinsam mit allen Architekten und Entwicklern des Systems in einem Workshop durchgeführt. Zu Beginn des Workshops wird der Sourcecode des Systems mit dem Analysewerkzeug geparst **(Abb. 7, Nr. 1)** und so die vorhandenen Strukturen erfasst (z.B. BuildUnits, Eclipse-VisualStudio-Projekte, Maven-Module, Package-Namespace-Directory-Bäume, Klassen). Auf diese vorhandenen Strukturen werden nun fachliche Module modelliert **(Abb. 7, Nr. 2)**, die der fachlichen Zerlegung entspricht, die mit den Fachexperten entwickelt wurde. Dabei kann das ganze Team sehen, wo die aktuelle Struktur nahe an der fachlichen Zerlegung ist und wo es deutliche Abweichungen gibt. Nun macht sich der Tool-Pilot gemeinsam mit dem Entwicklungsteam auf die Suche nach einfachen Lösungen, wie die vorhandene Struktur durch



Mob-Architecting mit dem Team. (Abb. 7)



Bounded Contexts des Kinosystems. (Abb. 8)



Architektur mit Use Cases. (Abb. 9)

Refactorings an die fachliche Zerlegung angeglichen werden kann (Abb. 7, Nr. 3) . Diese Refactorings werden gesammelt und priorisiert (Abb. 7, Nr. 4). Manchmal stellen der Tool-Pilot und das Entwicklungsteam in der Diskussion fest, dass die im Sourcecode gewählte Lösung besser oder weitergehend ist, als die fachliche Zerlegung aus den Workshops mit den Anwendern. Manchmal ist aber auch weder die vorhandene Struktur, noch die gewünschte fachliche Zerlegung die beste Lösung und beides muss noch einmal grundsätzlich überdacht werden.

Schön wäre es, wenn eine Zerlegung, wie in (Abb. 8) links dargestellt, möglich wäre. Dort sieht man die Bounded-Contexts aus dem Kinobeispiel aus (Abb. 6). Auf der linken Seite sind zwei unabhängige fachliche Module zu erkennen, die sich gegenseitig lediglich durch asynchrone Aufrufe über Änderungen informieren. Ansonsten können die beiden Bounded-Contexts ihre Arbeit unabhängig voneinander erledigen. Rechts hingegen sind die User-Interfaces außerhalb der Entity-orientierten Bounded-Contexts untergebracht, damit sie synchrone Aufrufe an den jeweiligen Entity-orientierten Services machen können. Die Konstruktion auf der rechten Seite ist nicht zu empfehlen, weil sie den Big-Ball-of-Mud nicht auseinandernimmt. Sie entspricht auch nicht der von DDD empfohlenen Umsetzung und ist hier nur zur Vermeidung von Missverständnissen dargestellt.

Das kanonische Domänenmodell

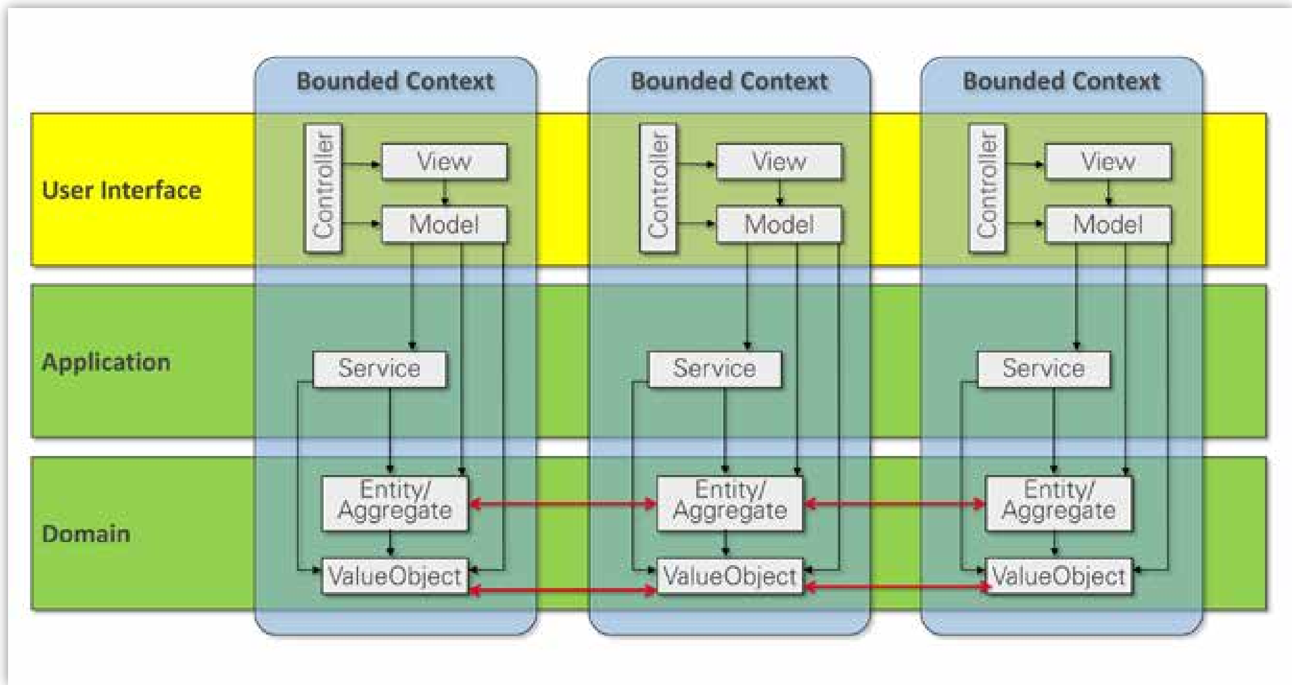
Den größten Knackpunkt bei der Zerlegung stellt in den meisten Monolithen das kanonische Domänenmodell dar. Wenn ich mir große Monolithen anschaue, dann finde ich dort in der Regel ein Domänenmodell, das von allen darauf aufbauenden Teilen der Software verwendet wird. Gut illustrieren lässt sich das an einem System, dass vor einiger Zeit von der Autorin untersucht werden durfte. Die Architekten waren mit dem Wunsch an sie herangetreten, die 1 Millionen Zeilen Java-Sourcecode in Microservices zu zerlegen. Auf die Frage, was für eine Architektur das System habe, wurde keine Schichtenarchitektur, sondern eine Architektur entlang von Use-Cases avisiert. Voller Hoffnung machte man sich mit den Architekten an die Arbeit, die Architektur mit dem häufig eingestehenden Tool-Sotograph[1] zu modellieren und den Sourcecode den modellierten Elementen zuzuordnen.

In (Abb. 9) sieht man dieses System auf der linken Seite, mit seiner ersten Aufteilung in Use-Cases. Jedes Rechteck **`mailing`**, **`importexport`** usw. beinhaltet einen oder mehrere Package-Bäume mit den darin enthaltenen Klassen. Die Klassen aus den verschiedenen Packages haben Beziehungen zueinander, um ihre jeweiligen Aufgaben zu erfüllen. Diese Beziehungen werden im Sotographen durch die grünen und roten Bögen dargestellt. Die grünen Bögen stellen Beziehungen von oben nach unten dar, die roten Bögen von unten nach oben.

Nachdem mit den Architekten die Use-Cases als Architekturelemente (Rechteck) modelliert wurden, blieb ein großer Teil des Package-Baums übrig, der den Namen **`model`** trug. In (Abb. 9) wurde links ein eigenes Architekturelement für **`model`** hinzugefügt. In **`model`** befinden sich alle Domänenmodellklassen gesammelt an einer Stelle. Dieses „Alles an einer Stelle" ist erst einmal kein Problem – es wird nur dann zu einem Problem, wenn diese Modellklassen von überall her verwendet werden. Um diesen Umstand zu überprüfen, wurde versucht die Modellklassen aus **`model`** den einzelnen Use-Cases zuzuordnen, indem die Klassen von unten nach oben verschoben wurden. Auf der rechten Seite in (Abb. 9) sieht man das vorläufige Ergebnis bis zu **`calculation`**.

Rechts in (Abb. 9) sieht man, wie stark das Domänenmodell von überall her verwendet wird und wie stark sich die Domänenmodellklassen untereinander verwenden. In DDD würde man sagen: Das ist ein Big-Ball-of-Mud. Was ist passiert?

Jeder Entwickler, der neue Funktionalität in das System eingebaut hat, hat dafür die zentralen Klassen des Domänenmodells verwendet. Allerdings musste er diese Klassen erweitern, damit er seine neue Funktionalität mit den Domänenmodellklassen



Duplizierung der Domänenklassen. (Abb. 10)

umsetzen konnte. So bekamen die zentralen Klassen mit jeder neunen Funktionalität ein bis zwei neue Methoden und Attribute hinzu. Aus dem Blickwinkel der Wiederverwendung von vorhandenen Klassen ist das eine logische Konsequenz. Das Ärgerliche ist nur, dass die Architektur so auf der Ebene des Domänenmodells sehr stark verkoppelt wird.

Domain-Driven-Design geht an dieser Stelle den entgegengesetzten Weg. Der Monolith soll in fachliche Module (Bounded-Contexts) aufgeteilt werden und in jedem Bounded-Context existieren die Klassen des Domänenmodells, die dort benötigt werden. Das bedeutet, dass die Kernklassen des Domänenmodells durchaus mehrfach im System vorkommen werden. Zugeschnitten auf ihren jeweiligen Bounded-Context bieten sie die Methoden an, die dort benötigt werden und nicht mehr.

Will man einen Monolithen fachlich zerlegen, so muss man das kanonische Domänenmodell zerschlagen. Das ist in den meisten großen Monolithen eine Herkulesaufgabe. Zu verwoben sind die auf dem Domänenmodell aufsetzenden Teile des Systems mit den Klassen des Domänenmodells. Um hier weiter zu kommen, werden zuerst die Domänenklassen in jeden Bounded-Context kopiert, der sie braucht (Abb. 10). Der Code wird also dupliziert und dann werden diese Domänenklassen jeweils für ihren Bounded-Context zurückgebaut. So bekommt man kontextspezifische Domänenklassen, die von ihrem jeweiligen Team unabhängig vom Rest des Systems erweitert und angepasst werden können.

Selbstverständlich müssen bestimmte Eigenschaften der Domänenmodellklassen, wie z.B. die ID und der Name, in allen Bounded-Contexts gleich gehalten werden, wo eine Variante einer Domänenklasse vorkommt. Außerdem müssen neue Objekte einer Domänenklasse in allen Bounded-Contexts bekanntgemacht werden, wenn in einem Bounded-Context ein neues Objekt angelegt wird. Diese Informationen werden über Updates von einem Bounded-Context an alle anderen Bounded-Contexts gemeldet, die mit dem jeweiligen Objekt arbeiten.

Fazit:

Monolithen sind in den meisten Unternehmen das Ergebnis von 10 bis 15 Jahren Programmierung. Dabei hat meistens die Zeit für Refactorings und Überarbeitung der Architektur gefehlt. Um im eigenen Unternehmen einen Monolithen zu zerlegen, muss zuerst die fachliche Domäne in Subdomänen zerlegt werden und diese Struktur im Anschluss auf den Sourcecode übertragen werden. Das kanonische Domänenmodell muss auseinandergenommen werden, was zum Teil nur durch Code-Duplizierung und anschließendem Zurückbau funktioniert. Folgt man den hier gesammelten Empfehlungen, so kann der Umbau von einem Monolithen zu einer fachlich orientierten Architektur gelingen.

Quellen:

1 https://bit.ly/2Ud9kVl
2 [Ev 2003] „Domain-Driven Design, Tackling Complexity in the Heart of Software“ erschienen bei Addison Wesley
3 [Li 2017] „Langlebige Softwarearchitekturen – Technische Schulden analysieren, begrenzen und abbauen“ erschienen im dpunkt.verlag, 2017 in der zweiten Auflage.

#JAVAPRO #Microservices #ProgressiveWebApp #Spring

Die Qual der Wahl

Wahlen in München: ca. 950.000 Wähler, 400.000 Briefwähler, 1.000 Wahllokale, Rechenfehler, 10.000 Seiten Papier. Um hier besser zu werden hat der Stadtrat der Landeshauptstadt München beschlossen die Datenerhebung in den Wahllokalen zu digitalisieren. Umgesetzt wurde dieser Beschluss in Form einer offline-fähigen Single-Page-Application und Spring-Microservices im Backend.

Autor:

Claus Straube ist IT Architekt bei der Landeshauptstadt München. Dort beschäftigt er sich damit, wie man neue IT Trends gewinnbringend für die IT einer großen Kommune nutzen kann. Schwerpunktthemen sind Java Anwendungsarchitekturen, JavaScript Technologien und Automatisierung in der Entwicklung. In den letzten jahren hat er zahlreiche Vorträge gehalten und Artikel geschrieben.

www.muenchen.de
GitHub: github.com/xdoo

Fabian Wilms arbeitet seit 2017 bei it@M, dem IT-Dienstleister der Stadt München, als Lead Developer in verschiedenen Projekten für das Kreisverwaltungsreferat. Dort ist er für die Planung der Infrastruktur, die Anwendungsarchitektur und die Koordinierung der Entwicklung zuständig. Er steigt auch selbst gerne tief in den Code ein, um Architekturentscheidungen auch aus Entwicklersicht auf ihre Praktikabilität zu überprüfen.

www.muenchen.de
GitHub: github.com/FabianWilms

Wahlen in einer Großstadt sind eine Herausforderung

Bei Wahlen in einer Großstadt fallen sehr viele Daten an, die meisten davon auf Papier. Es fängt mit dem Wählerverzeichnis an, also der Liste in der alle wahlberechtigten Bürger stehen und hört mit der Niederschrift auf. Nach der Stimmenauszählung im Wahllokal werden in diese vom Wahlvorstand die Auszählungsergebnisse eingetragen. Je nach Andrang und Komplexität der Wahl ist dieser Schritt kurz vor Mitternacht abgeschlossen. Am nächsten Morgen, dem Wahlmontag, werden die Daten aus den Niederschriften von vielen fleißigen Helfern im Wahlamt vom Papier in ein Computersystem übertragen.

Die Probleme, die bei diesem Vorgehen auftreten, liegen klar auf der Hand. Am Montag nach der Wahl entsteht ein großer Aufwand, um Daten, die kurz vorher auf Papier geschrieben wurden, digital zu erfassen. Das zweite Problem ist nicht so offensichtlich: Papier ist geduldig, zu geduldig. Nach einem anstrengenden Wahltag und der Auszählung müssen die ehrenamtlichen Helfer mitten in der Nacht, mehr oder weniger komplexe - bei Kommunalwahlen sind sie durchaus komplex - Berechnungen zur Ermittlung des Ergebnisses durchführen. Trotz Unterstützung eines Taschenrechners ist die Wahrscheinlichkeit, dass hier Fehler passieren, relativ hoch. Eine vorgegebene Prüfung der erfassten Zahlen findet frühestens im Wahlamt in grober Form, statt.

Um das besser zu machen, ist man in München auf die Idee gekommen, dass die Daten im Wahllokal direkt digital zu erfassen. Ein Wahllokalsystem (WLS) könnte den Wahlvorstand bei der Dateneingabe durch Plausibilisierung unterstützen. Die Datenübertragung am Montag würde dadurch natürlich komplett entfallen. Papier wird dadurch leider nicht gespart. Die Niederschrift muss nach wie vor in Papierform vorliegen und vom Wahlvorstand unterschrieben werden. Dies fordert eine gesetzliche Vorgabe.

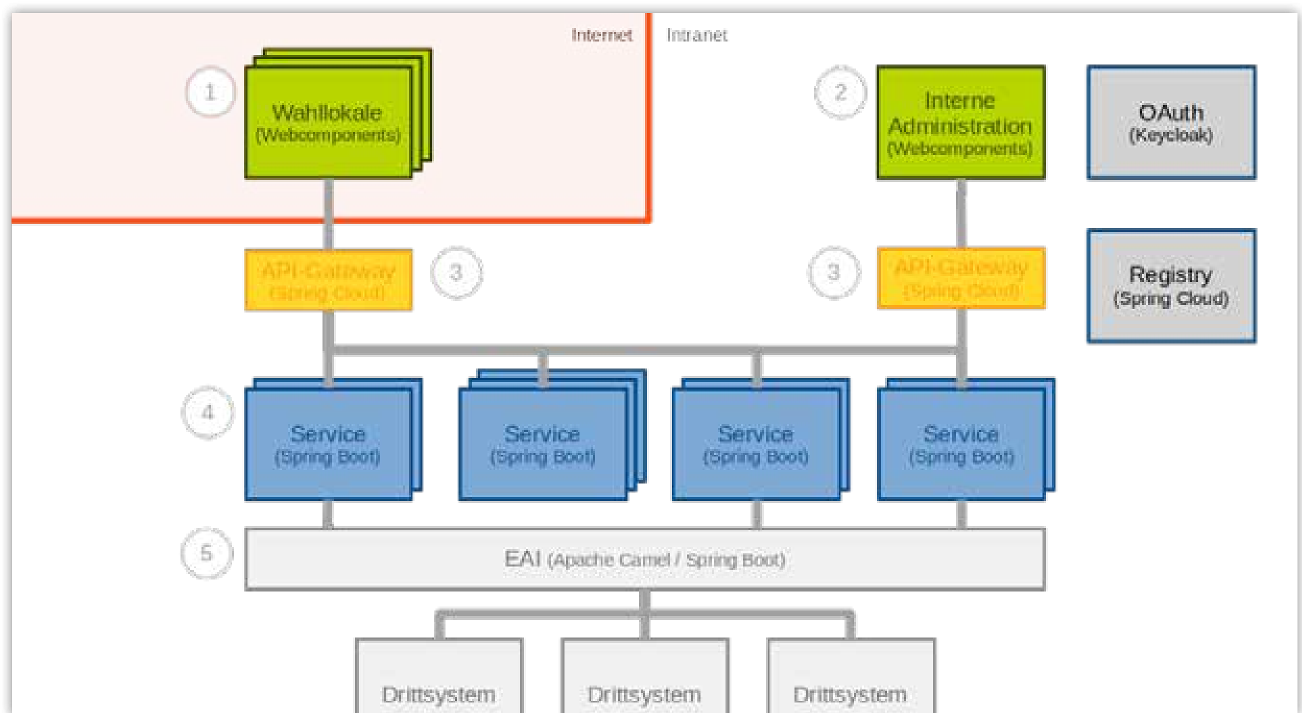
Zwei weitere Vorteile einer digitalen Verbindung in die Wahllokale liegt in der Kommunikation. Der Wahlvorstand kann online die Wahlbeteiligung übermitteln. Gibt eine Person ihre Stimme ab, so wird der Zähler einfach um eins erhöht. Dadurch ist es möglich, die Wahlbeteiligung in Echtzeit zu messen. Früher wurden 20 Wahllokale abtelefoniert und die Ergebnisse hochgerechnet. Bei 750 Urnenwahllokalen kann man sich vorstellen, wie genau diese Aussagen waren. Auch die Kommunikation von der Zentrale in die Wahllokale ist deutlich einfacher. Um alle Wahllokale über eine Änderung zu informieren, mussten vor fünf Jahren ein Dutzend Beteiligte eine Stunde lang telefonieren - immer in der Hoffnung alle zu erreichen. Mit dem Wahllokalsystem geht das per Push-Nachricht. Diese Argumente haben den Münchner Stadtrat überzeugt. Er stellte das Budget für die Umstrukturierung zur Verfügung und die Umsetzung konnte beginnen.

Die Architektur

Eine der ersten Fragen, die bei Planung einer Softwarearchitektur innerhalb der Verwaltung geklärt werden muss, ist, von wo die Nutzer auf die Anwendung zugreifen. Die Regel ist ein Zugriff aus dem Backbone. Inzwischen gibt es aber auch immer öfter Zugriffe aus dem Internet. Das ist auch in diesem Szenario der Fall. Die Wahllokale greifen aus dem Internet über eine sichere Verbindung auf das Wahllokalsystem zu **(Abb. 1, Nr. 1)**. Interne Administratoren können aus dem Backbone die Applikation konfigurieren **(Abb. 1, Nr. 2)**. Das heißt, ein Teil der Anwendung wird nach außen frei gegeben, der größte Teil ist im internen Netz. Welcher Teil der Anwendung von wo aus erreichbar ist, wird über zwei API-Gateways entschieden **(Abb. 1, Nr. 3)**. In diesem Szenario sind es zwei API-Gateways. Natürlich können es auch mehr oder weniger sein. Diese sind so konfiguriert, dass die Clients tatsächlich nur auf den Teil der Anwendung Zugriff haben, den sie wirklich benötigen. Zudem verifiziert das Gateway jede Anfrage gegen die Security-Komponente - in diesem Fall mit OAuth gegen Keycloak.

Wichtig ist auch immer der Schutzbedarf der Daten. Bei Wahldaten ist naturgemäß die Integrität ein hohes Gut. Es darf für dritte keine Möglichkeit geben, die Daten auf der Strecke zwischen Wahllokal und Backend, aber natürlich auch nicht innerhalb des Kernsystems, zu manipulieren. Eine weitreichende Entscheidung in diesem Kontext war, dass die Authentifizierung der Rechner im Wahllokal über ein Zertifikat erfolgen muss. Es werden entsprechend alle Rechner in den Wahllokalen mit einer Zertifikatsdatei ausgeliefert. Für die Durchführung einer Wahl ist auch entscheidend, dass die Applikation robust ist. Fehler der Nutzer müssen abgefangen werden, aber auch Netzausfälle und Lastspitzen. Das ist vor allem deshalb eine Herausforderung, weil die Anwendung nicht regelmäßig produktiv betrieben wird. Wahlen finden in sehr großen Abständen statt. Dann muss in einem Zeitfenster von wenigen Stunden alles funktionieren. Eine Einschwingphase wie bei klassischen Applikationen, die in täglicher Nutzung sind, gibt es hier nicht.

Diese Rahmenbedingungen haben dazu geführt, dass im Backend das Spring-Framework (Spring-Boot) verwendet wird. Ein weiterer Pluspunkt für Spring sind die zahlreichen Plugins wie Spring-Security oder Spring-Data. Um auf Lastspitzen gezielter reagieren zu können wurde die Entscheidung getroffen, das Backend nicht monolithisch aufzubauen, sondern mit Hilfe einer Microservice-Architektur. Auch hier gibt es durch Spring-Cloud eine Framework-Unterstützung. Die Services und Gateways müssen miteinander kommunizieren. Damit man im laufenden Betrieb neue Service-Instanzen zu- bzw. abschalten kann, dürfen die Domains der Services natürlich nicht hart im Code verdrahtet sein. Erreicht wird das über eine zentrale Registry, bei der jeder Service unter einem Alias registriert und beliebig viele reale Adressen hinterlegt sind. Die aufrufende Komponente lädt in regelmäßigem Intervall die aktuellen URLs der Aliase aus der Registry und spricht diese im Round-Robin-Verfahren an.



Front- und Backend-Architektur des Wahllokalsystems. (Abb. 1)

Das Wahllokalsystem ist eine Anwendung, die in eine Gruppe anderer Wahlapplikationen eingebettet ist. Das sind in der Regel Kaufanwendungen. Der Zugriff auf diese Systeme erfolgt über mehrere Enterprise-Application-Integration-Komponenten (EAI) **(Abb. 1, Nr. 5)**.

Frontend als Progressive-Web-Application

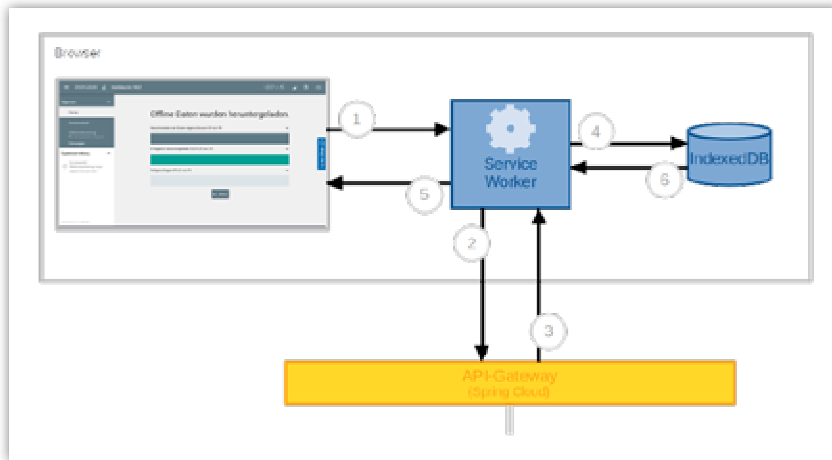
Jedes Wahllokal ist mit einem sogenannten Wahlkoffer ausgestattet. Dies ist ein Notebook mit Drucker, fest verbaut in einer Art Koffer. Auf dem Rechner sind bereits alle notwendigen Installationen und Konfigurationen vorgenommen worden. So wird sichergestellt, dass die Auslieferung an die 1.000 Wahllokale kurz vor dem Wahlsonntag erfolgen kann. Die Inbetriebnahme der Geräte erfolgt durch den Wahlvorstand. Die Anwendung wird von ehrenamtlichen Laien bedient. Diese müssen unter Druck komplexe Aufgaben erledigen. Hier soll die Anwendung bestmöglich unterstützen. Um einen Wiedererkennungswert zu schaffen wurde als Styleguide die Material-Design-Spezifikation von Google verwendet. Auf Basis von Web-Components und Polymer wurde eine Progressive-Web-Application (PWA) gebaut, mit der die Kollegen im Wahllokal auch dann noch arbeiten können, wenn aktuell keine Verbindung mit dem Internet besteht.

Offline-Fähigkeit

Eine offlinefähige PWA basiert in der Regel auf einem Service-Worker. Das ist ein JavaScript-Programm, das unabhängig von der Webseite im Hintergrund läuft. Es kann zwar nicht den DOM manipulieren, hat aber vollen Zugriff auf die Netzwerkkommunikation, die vom Browser weg und zum JavaScript-Programm hin verläuft. Zusätzlich kann der Service-Worker auf die Indexed-DB-API zugreifen. Die Technologie bietet bereits von Haus aus ein automatisches und feingranular konfigurierbares Caching von statischen Ressourcen. Auch ein Puffern von Netzwerkanfragen, die aufgrund mangelnder Konnektivität ins Leere liefen, lässt sich umsetzen. Das ist zwar bereits ein guter Ansatz, der allerdings einen Anwendungsfall außer Acht lässt: Was passiert, wenn diese statischen Ressourcen durch die Interaktion des Nutzers mit der Anwendung verändert werden? Wie führt man lesende und schreibende Netzwerkanfragen zusammen, um eine vollständige Offline-Fähigkeit zu erreichen?

Das Problem lässt sich durch Bord-Mittel des Service-Workers nicht lösen. Er bietet allerdings Funktionen an, die man genau für diesen Zweck einsetzen kann, beispielsweise durch sogenannte Fetch-Events[1]. Ein Fetch-Event wird direkt vom Browser, bei jedem über die Fetch-API[2] abgesetzten Netzwerk-Request, erzeugt und an den Global-Scope des Service-Worker übergeben. Dieser kann dann beliebige Dinge mit den Daten machen, beispielsweise die Anfrage an die aufgerufene URL weitergeben oder aber die Informationen zwischenspeichern, falls keine aktive Internetverbindung besteht. Über das Fetch-Event kann auch eine Antwort an die aufrufende Anwendung zurückgegeben werden. Im besten Fall merkt der Nutzer vor dem Rechner oder Smartphone gar nicht, ob er mit dem Internet verbunden ist oder nicht. Über dieses Feature lässt sich der Service-Worker als eine Art Middleware zwischen Browser und Backend einbauen.

Kombiniert man diese Möglichkeit mit einer REST-konformen HTTP-API, erhält man eine einfache Möglichkeit sämtlichen Anwendungs-Traffic automatisch offlinefähig werden zu lassen.



Service-Worker in der Browser-Backend-Kommunikation. (Abb. 2)

Jede Ressource ist über eine eindeutige URL zu erreichen. Über die HTTP-Methode wird zwischen lesenden und schreibenden Anfragen unterschieden.

Lesende Anfragen **(Abb. 2, Nr. 1)** werden lokal in der IndexedDB des Browsers zwischengespeichert **(Abb. 2, Nr. 4)** und bei Bedarf wieder ausgeliefert **(Abb. 2, Nr. 6 + 5)**. Die eindeutige Ressourcen-URL dient hierbei als Key zum Speichern der Daten. Schreibende Anfragen werden durch den SW ans Backend geschickt **(Abb. 2, Nr. 2 + 3)** und die zu sendenden Daten werden in der IndexedDB wieder mit der eindeutigen URL als Key gespeichert. Außerdem werden die Daten um einen zusätzlichen Dirty-Flag ergänzt. Dieser ist true, wenn ein schreibender Request nachweislich nicht im Backend gespeichert werden konnte, z.B. auf Grund von Netzwerkfehler, Backend-Fehler etc.

Abschließend muss nur noch eine Synchronisationslogik implementiert werden, die eventuell vorhandene Dirty-Daten wieder ans Backend sendet. Im Falle des Wahllokalsystems ist diese Logik sehr simpel: Ein Wahlvorstand arbeitet nur auf seinen eigenen Daten, somit können keine Konflikte mit anderen Nutzern bei der Synchronisation auftreten und die zuvor fehlgeschlagenen Anfragen müssen nur in zeitlich korrekter Reihenfolge erneut ans Backend gesendet werden.

Application-State

Polymer, bzw. die Web-Components-Spezifikation ist stark auf die Trennung zwischen Komponenten fokussiert. Von Haus aus gibt es, anders als beispielsweise bei Angular, nur wenig Mittel um Zustände über die gesamte Anwendung hinweg zu teilen. Trotzdem wird in etwas komplexeren Client-Anwendungen in der Regel eine Zustandsverwaltung benötigt. Gelöst werden kann dies über eine zusätzlich JavaScript-Bibliothek, die in die Anwendung eingebunden wird. Im konkreten Fall ist dies Redux[3]. Das ist ein Datenspeicher (Store), der über Aktionen (Actions) modifiziert werden kann. Ansonsten wird auf den Datenspeicher ausschließlich lesend zugegriffen.

Ohne Zugriff auf das Backend der Anwendung ist es auch notwendig, dass das Frontend seinen eigenen Application-State pflegt. Über einen gut definierten State lassen sich Validierungs- und Plausibilisierungsregeln leicht implementieren. Auch eine automatische Navigation in der Anwendung lässt sich so bewerkstelligen: Je nach Fortschritt des Nutzers in der Bearbeitung seiner Daten kann die Anwendung anhand des States die nächsten Schritte einblenden.

Redux schafft es, die von Polymer lose gekoppelten Komponenten wieder zu einer großen Gesamtanwendung zusammenzubringen und verhindert komplizierten Glue-Code. Der event-basierte Ansatz harmoniert hervorragend mit dem modernen Programmier-Stil des Databindings und bindet die Komponenten gleichzeitig nicht so fest aneinander, dass Abhängigkeiten entstehen.

Backend als Microservice-Application

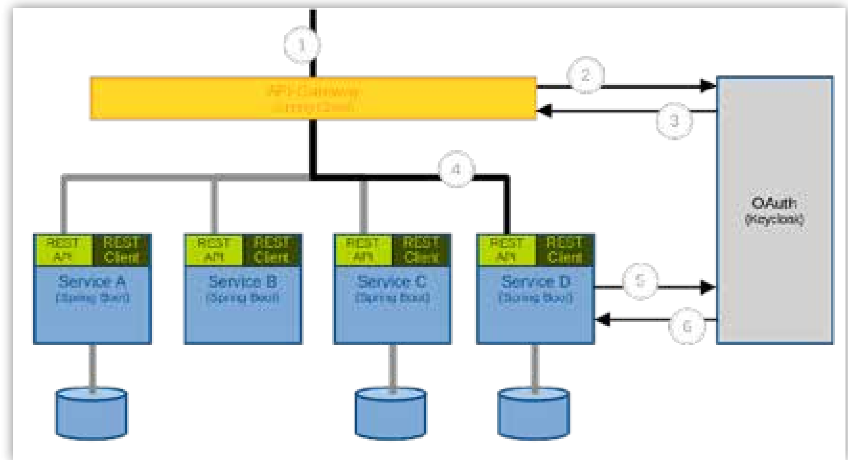
Wie oben schon angedeutet, ist das Backend in Form einer Microservice-Architektur konzipiert und letztendlich umgesetzt worden. Ausschlaggebend für diese Entscheidung waren zwei Aspekte. Einmal gibt es bei der Wahl Lastspitzen (zu Beginn und am Ende), die durch das verteilte Deployment optimal abgefangen werden können. Der zweite Grund liegt im Lifecycle-Management. Große, monolithische Anwendungen bringen bei Erneuerungen (beispielsweise Versions-Updates von grundlegenden Frameworks) immer das Problem mit, dass die Maßnahme für das komplette Verfahren in einem Zug durchgeführt werden muss. Dies ist oft ein organisatorisches Problem. Durch die physische Auftrennung der Applikation können auch die Lifecycle-Maßnahmen getrennt voneinander durchgeführt werden. Die komplette Architektur vorzustellen würde den Rahmen des Artikels übersteigen, deshalb wird auf zwei Aspekte detaillierter eingegangen: Security und Service-zu-Service-Aufrufe.

Security

In Sachen Security spielt sowohl die Authentifizierung als auch die Autorisierung eine wichtige Rolle. Nicht jeder, der potenziell Zugang zur Anwendung hat, darf auch alle Operationen ausführen. Im ersten Schritt wird der Rechner authentifiziert. Nur ein Rechner mit gültigem Zertifikat kann die Verbindung zum Server aufbauen, auf dem der Wahllokal-Client liegt. Hat der anfragende Rechner kein entsprechendes Zertifikat, wird er vom Proxy-Server abgewiesen.

Im zweiten Schritt wird ein Nutzer (jedes Wahllokal wird als ein Nutzer angesehen) authentifiziert. Er meldet sich über den

Browser mit Nutzername und Passwort an. Im Rahmen der Wahllokalanwendung werden die Passwörter vor der Wahl zufallsbasiert generiert. Bei der Passwortgenerierung wird dieses Wertepaar serverseitig in einem Verzeichnisdienst gespeichert. Zusammen mit dem Nutzernamen werden die Passwörter ausgedruckt und zu den verplombten Wahlunterlagen gegeben. Diese werden kurz vor der Wahl in die Wahllokale transportiert und erst am Wahltag vom Wahlvorstand geöffnet. Danach wird der Wahlkoffer in Betrieb genommen und das Wahllokal meldet sich mit seinem Nutzernamen und Passwort an der Applikation an.



Kommunikationswege zum Aufbau des Security-Kontextes. (Abb. 3)

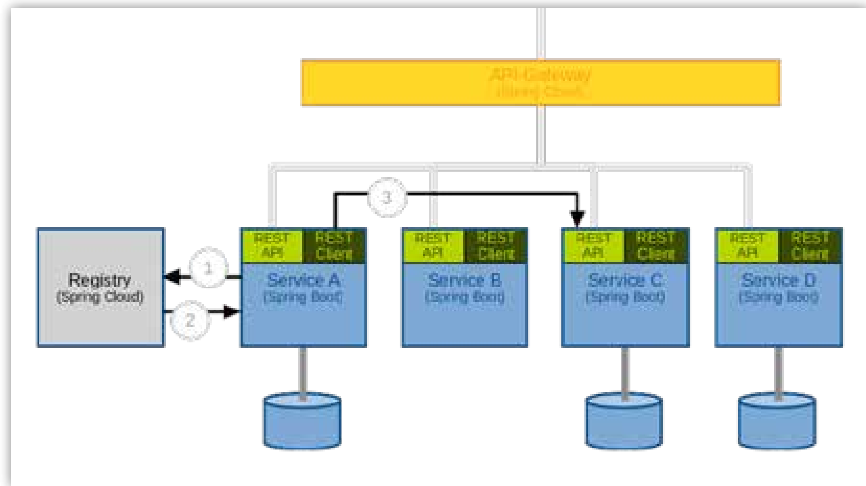
Die Anmeldedaten werden über eine verschlüsselte Verbindung vom API-Gateway entgegengenommen **(Abb. 3, Nr. 1)** und gegen den OAuth-Server - in diesem Fall ein Keycloak - aufgelöst **(Abb. 3, Nr. 2)**. Dafür geht der OAuth-Server gegen den Verzeichnisdienst, in dem zuvor die generierten Anmeldedaten gespeichert wurden. Konnten die Anmeldedaten im Verzeichnisdienst gefunden werden, erzeugt Keycloak einen Token und schickt diesen zurück an das API-Gateway **(Abb. 3, Nr. 3)**. Aus Sicherheitsgründen verlässt der Token das API-Gateway nicht. Trotz allen zusätzlichen Schutzmaßnahmen wird der Client wie ein fremdes System behandelt. Im API-Gateway wird der Token in einer Session gespeichert und der Client bekommt die Session-ID zurückgeschickt. Mit dieser ID kann der Client dann Requests an das Backend schicken. Vorsicht ist geboten, wenn es aus Gründen der Ausfallsicherheit mehrere Instanzen des API-Gateways gibt. Dann muss entweder die Session geteilt werden, oder der vorgeschaltete Proxy-Server vergibt Sticky-Sessions. Bei jedem Request wird nun innerhalb des API-Gateways über die Session-ID der Token geladen und in die Service-Aufrufe eingebettet **(Abb. 3, Nr. 4)**. Da jeder Nutzer einen eindeutigen und persönlichen Token hat, ist es möglich - selbst bei kaskadierenden Service-Aufrufen - nachzuvollziehen, welche Person den Aufruf initiiert hat. Das ist einerseits wichtig, um nachverfolgen zu können, wer wann Daten im System modifiziert hat. Andererseits ist es so möglich, einen nutzerspezifischen Security-Kontext im jeweiligen Service aufzubauen. Das wiederum ist wichtig, um bestimmen zu können, welche Operationen der Nutzer innerhalb eines Service ausführen darf. Die Informationen über die Berechtigungen werden nicht im Token mitgeliefert. Grundsätzlich wäre das über einen JSON-Web-Token (JWT) möglich. Da dieser aber im Header transportiert wird, ist die Größe des Tokens durch die Standard-Einstellungen der Server beschränkt. Diese kann man in der Regel modifizieren. Daran muss man aber auch beim Transfer nach der Produktion denken. Es ist zumindest ein weiteres, vermeidbares Risiko. An dieser Stelle muss man die Entscheidung treffen, ob man mit wenigen, sehr groben Rollen arbeiten kann. Dann ist diese Methode durchaus sinnvoll. Bevorzugt man ein fein granulares Berechtigungskonzept, beispielsweise auf Ebene der Operationen, dann wird der JWT schnell zu groß. Abhilfe kann hier geschaffen werden, indem bei jedem Request die Berechtigungen des Nutzers vom OAuth-Server geholt werden **(Abb. 3, Nr. 5 + 6)**. Das ist erst einmal ein Overhead. Dieser kann aber relativ einfach durch intelligentes Caching an der Schnittstelle zum Security-Server minimiert werden.

Service-zu-Service-Calls

Ein weiterer interessanter Aspekt, der in einer Microservice-Architektur völlig anders umgesetzt wird als in einer monolitischen Anwendung, ist die Kommunikationen zwischen den Modulen. Es wird in der Regel nicht nur eine Kommunikation zwischen API-Gateway und Service geben, sondern im Rahmen von kaskadierenden Aufrufen auch unter den Services selbst. Das kann man event-basiert lösen, indem man die Services an einen Bus anschließt, auf dem die Nachrichten übertragen werden. Vorteil dieser Lösung ist, dass die einzelnen Services sich tatsächlich nicht kennen. Jeder Service kennt nur die Adresse des Busses und weiß, auf welche Ereignisse er reagieren muss. Dadurch kann man weitestgehend eine lose Kopplung zwischen den Komponenten erzielen und sogar neue Services in die Anwendung einbinden - quasi per Plug-and-Play - ohne andere Services anfassen zu müssen. Trotzdem können sie miteinander kommunizieren. Nachteil dieser Lösung ist, dass die Kommunikation sehr implizit erfolgt. D.h. wenn Service A eine Nachricht erzeugt, dann weiß er (und damit auch der Entwickler) nicht, was diese Nachricht im Rest des Systems auslöst. Dieser Ansatz war für die Landeshauptstadt München zu diesem Zeitpunkt ungewohnt. Bisher wurden monolithische Anwendungen mit direkten Kommunikationsbeziehungen entwickelt. Um nicht den zweiten Schritt vor dem ersten zu machen, wurde beschlossen, in dieser kritischen Applikation ebenfalls auf direkte Kommunikationsbeziehungen zu setzen.

Spring bietet hierfür auch eine Reihe ausgereifter Werkzeuge. Zentral ist in einer klassischen (virtuellen) Serverlandschaft die

Registry. In dieser wird für jeden Service ein Alias gespeichert. Diesem Alias werden die URLs der Service-Instanzen zugeordnet. Ein aufrufender Service lädt von dieser Registry regelmäßig die aktuellen Instanzen der Aliase herunter und speichert diese zwischen (Abb 4, Nr. 1 + 2). Bei einer Anfrage wird lokal anhand des Alias eine verfügbare URL aufgelöst (im Round-Robin-Verfahren) und die Anfrage an diese URL gestellt. Dieses Verhalten bringt Spring-Cloud mit sich und muss nur bei Bedarf angepasst werden. Für den Entwickler selbst sind die Aufrufe relativ unkompliziert zu implementieren. Für jeden aufzurufenden Service wird im einfachsten Fall ein Client-Interface erstellt. Auf diesem wird mit der Annotation **`@FeignClient`** der Alias angegeben (in diesem Fall **`ServiceC`**) und gegebenenfalls eine Fallback-Klasse, die einspringt, wenn **`ServiceC`** nicht erreichbar ist. (Listing 1) zeigt einen Code-Ausschnitt.



Kommunikationswege für einen Service-zu-Service-Aufruf. (Abb. 4)

(Listing 1)

```
@FeignClient(
value = "ServiceC",
fallback = ServiceCClientFallback.class)
public interface ServiceCClient {
  @GetMapping("/hello")
  String hello();
}
```

Im Interface selbst müssen nur noch die Methoden mit den entsprechenden Mappings angegeben werden, wie man sie auch aus den Spring-Controller-Klassen kennt. Die Clients können innerhalb des Spring-Kontextes per Dependency-Injection eingebunden und verwendet werden. Beispielsweise um die **`hello`** Ressource in **`ServiceC`** aufzurufen (Abb. 4, Nr. 3).

Fazit:

Der in diesem Artikel dargestellte Zustand ist nicht identisch mit dem ersten Wurf der Anwendung. Das Wahllokalsystem hat in den vergangenen Monaten einen Wandel durchlaufen. Technische Probleme in Fremdsystemen, mangelnde Resilienz gegenüber diesen und ein historisch gewachsenes Frontend erforderten einen kleinen Neuanfang. Es wurde schnell festgestellt, dass der bisherige Service-Schnitt im Backend an mehr als nur einer Stelle schlecht gewählt war, was zu unnötigem Netzwerk-Traffic und Abhängigkeiten führte. Auch wurde REST an vielen Stellen nicht konsequent eingehalten, was die Kommunikation zwischen Frontend und Backend nicht übersichtlicher gestaltete.

Zu guter Letzt wurde das Frontend komplett neu aufgebaut. Zuvor wurde auf eine veraltete Angular-Version (1.6) gesetzt und die Code-Basis war stark verworren, sodass Änderungen am Client immer das Risiko neuer Bugs mit sich zogen. Auch die Offlinefähigkeit war hoch komplex umgesetzt und wurde regelmäßig in der Entwicklung falsch bedient, was zu weiteren Problemen führte.

Ziel war es also gewesen, das Backend zu entzerren, Kommunikationsbeziehungen und Abhängigkeiten zwischen Services abzubauen und eine einheitliche API bereitzustellen, mit der ein neues Frontend gut zusammenarbeiten kann. Das Frontend sollte modularer werden, stark getrennte Komponenten mit klar definierten Aufgaben und Funktionen - weshalb Polymer mit Web-Components zum Einsatz gekommen ist. Diese Modularisierung in Kombination mit einer stark vereinfachten Offline-Fähigkeit und Statemanagement hat nicht nur der Qualität der Code-Basis, sondern auch dem Verhalten und der Nutzerfreundlichkeit der Anwendung einen gehörigen Schub nach vorne gegeben. Die neue Architektur hilft auch dabei, schnell kurzfristige Anforderungen für die unterschiedlichen unterstützten Wahlarten zu implementieren.

Auch wenn nur wenige Monate Zeit für diesen Rewrite zur Verfügung standen, konnte alles just in time fertig gestellt werden und auch den Kunden überzeugen. Eine Weiterentwicklung des Angular-Frontends hätte weitaus mehr Zeit gekostet. Auch wenn ein Neuanfang zunächst schwierig und risikobehaftet erscheint, so kann er in einigen Fällen doch von Altlasten befreien und die Qualität des Gesamtprodukts nachhaltig verbessern. Die Europawahl im Mai wird zeigen, ob das in diesem Fall auch so war.

Quellen:

1 https://mzl.la/2C6XMXB
2 https://mzl.la/2Wqn9xi
3 https://redux.js.org/

RAPID clipse™
THE VISUAL ECLIPSE

eclipse
FOUNDATION
MEMBER

Supported Browsers:

Supported Mobile Platforms:

Supported Desktop Platforms:

WEB • MOBILE • DESKTOP • CLOUD

NEU !

RapidClipse X

Neu mit Vaadin 14 GUI-Builder & MicroStream Integration
Final Release Vorstellung - 2 Tage alles über RapidClipse X - Training Day
vom 23. bis 26. September auf der JCON 2019 - www.jcon.one

www.rapidclipse.com

#JAVAPRO #ContinuousIntegration #Testing

Containerbasierte Testautomatisierung

In Zeiten von Containern, Clustern, Build-Pipelines und DevOps ist ein hoher Grad an Automatisierung notwendig, um sich auf das Wesentliche zu konzentrieren: Das Softwareprodukt. Hierfür bietet die Open-Source-Welt eine Reihe von Werkzeugen, um vollautomatische, steuerbare und reproduzierbare Build-, Test- und Deployment-Prozesse zu realisieren.

Testen von Microservices

Ein Tech-Stack mit unterschiedlichsten Technologien, die Fachlichkeit in verschiedene Services verteilt, dazu eine durch Netzwerke und Kommunikationsprotokolle geprägte Gesamtarchitektur. Im Vergleich zu monolithischen Systemen wird schnell klar, dass sich durch Microservices prinzipbedingt einiges geändert hat. Dies gilt auch für das Thema Testing. Dabei bleiben die Herausforderungen klassischer Systemarchitekturen bestehen. Der Trendwechsel hin zur Microservice-Architektur bringt sowohl Vorteile als auch Nachteile mit sich. Durch die Aufteilung komplexer Fachlichkeit in kleinere Einheiten verringert sich der funktionale Umfang des einzelnen Services. Die Isolierung der Services über Kommunikationstechnologien und Nachrichten, sowie die kurzen Start- und Deployment-Zeiten vereinfachen das Testen enorm. Die Isolation der Fachlichkeit hält dabei den Umfang der Test-Suite einzelner Komponenten klein und übersichtlich. Unter Verwendung von Container-Technologien lassen sich zudem Ad-hoc-Testumgebungen bereitstellen und wieder entfernen. Die Isolation und Nachrichtenorientierung moderner Systemarchitekturen bringt jedoch im gleichen Zuge neue Herausforderungen mit sich. Der Fokus des Testens verschiebt sich auf Integrationstests, da innerhalb des Gesamtsystems nun deutlich mehr Systemintegrationen zu anderen Services stattfinden müssen. Die verwendeten Kommunikationstechnologien sind dabei genauso unterschiedlich wie die Anforderungen an die jeweiligen Teile des Gesamtsystems. Die korrekte Interaktion zwischen Systemkomponenten und das Testen größerer fachlicher Anforderungen kann nun nicht mehr innerhalb eines Softwareprojekts unter Verwendung von Mocks sichergestellt werden. Stattdessen sind die Endpunkte anderer Services akkurat zu simulieren und End-2-End-Testumgebungen aufzusetzen, um eine aussagekräftige Testabdeckung zu erreichen. Aus der bekannten Test-Pyramide entwickelt sich somit ein Test-Diamant **(Abb 1)**.

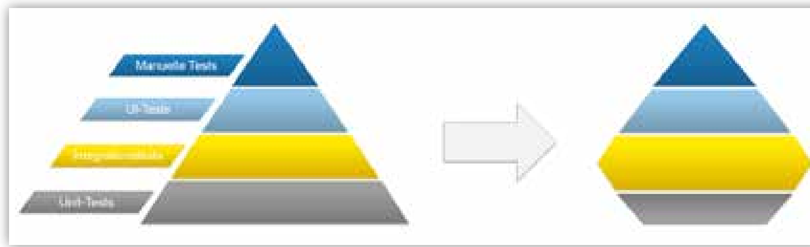
Continuous-Integration und Continuous-Delivery (CI/CD)

Continuous-Integration (CI) bezeichnet eine Arbeitsmethodik, die zum Ziel hat, das gemeinsame Arbeiten an Software zu vereinfachen und Konflikte bei der Integration von zum Beispiel neuen Features zu vermeiden, die durch unterschiedliche Stände in den Arbeitskopien der einzelnen Entwickler verursacht werden. Dies wird bei Continuous-Integration durch einen Workflow realisiert, der kontinuierlich und bestenfalls vollautomatisch durchlaufen wird und dabei sicherstellt, dass die Abweichungen der Softwarestände nicht zu groß werden und keine größeren Konflikte entstehen, die nur unter enormen Zeitaufwand zu lösen

Autor:

Sven Hettwer ist Senior Software-Engineer mit Fokus auf Testautomatisierungs- und CI/CD-Lösungen sowie Maintainer des Open Source Frameworks Citrus beim Münchner IT-Dienstleister Consol am Standort Düsseldorf.

https://www.consol.de
Blog: https://labs.consol.de
Twitter: http://twitter.com/SvenHettwer
Xing: https://www.xing.com/profile/Sven_Hettwer
GitHub: https://github.com/svettwer
Email: Sven.Hettwer@consol.de



Von der Test-Pyramide zum Test-Diamanten. (Abb. 1)

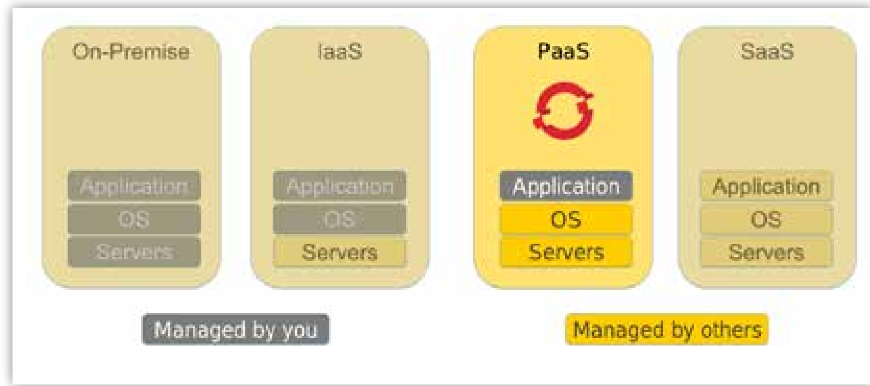
sind. Grundlage dieses Workflows ist, dass alle Entwickler einer Software in einem gemeinsamen Repository arbeiten. Services wie GitHub, GitLab und Bitbucket stellen die dafür benötigte Infrastruktur bereit und erlauben neben der reinen Verwaltung von Quellcode auch die Verwaltung von Nutzerberechtigungen, die Anbindungen an Pipeline-Tools und bieten z.T. sogar Projektmanagement-Tools für die Verwaltung von Fragen, Fehlerberichten und neuen Funktionalitäten. Der nächste Schritt in Richtung CI ist das automatische Bauen und Testen der Software. Die Ökosysteme moderner Programmiersprachen bieten hierfür flexible konfigurierbare Build- und Test-Werkzeuge, die über Kommandozeilenbefehle zu steuern sind und sich zum großen Teil durch Plugin-Systeme beliebig erweitern lassen. Stehen Versionsverwaltung und Build-Werkzeuge bereit, kann leicht damit begonnen werden, den Gesamtablauf der Continuous-Integration zu automatisieren. Hierfür empfiehlt sich die Verwendung von spezialisierten CI-Systemen wie Jenkins, Drone-CI, Travis-CI, GitLab-CI oder Amazon-Code-Pipelines, um nur einige aus diesem reichhaltigen Ökosystem aufzulisten. Jedes dieser Tools bietet dabei eine Plattform, mit der es möglich ist, Build- und Test-Pipelines zu erstellen, die sich exakt an den Anforderungen des Software-Builds orientieren. Hierbei ist es wichtig, sich nicht nur auf das automatisierte Bauen und Testen des Hauptzweigs (Master/Trunk) der Software zu beschränken, sondern ebenfalls die Branches mit neuen Features oder Bugfixes diesbezüglich zu automatisieren. Allgemein lässt sich sagen, dass die Kosten für eine Fehlerbehebung sinken, je früher der Fehler erkannt wird. Eine Automatisierung von Branch-Builds, sie macht Probleme schon während der Entwicklung frühzeitig sichtbar, erlaubt es den Teams zeitnah zu reagieren und reduziert somit aktiv die Entwicklungskosten. Release-Management und die Anbindung weiterer Tools für zum Beispiel statische Codeanalyse lassen sich zudem einfach konfigurieren und in die Pipeline einbinden. Damit sichergestellt ist, dass die aktuelle Pipeline stets zum aktuellen Stand der Software passt, empfiehlt es sich, die Konfiguration mit dem Quellcode zu speichern, zu verwalten und zu versionieren. Dieses als Configuration-as-Code bekannte Prinzip wird von allen gängigen CI-Systemen unterstützt. Dabei ist es unerheblich, ob das CI-System inhouse oder in der Cloud liegen soll. Der Markt bietet ein Produkt für jede Anforderung. Zu guter Letzt ist darauf zu achten, dass Branches nicht langfristig existieren. Je länger ein Branch existiert, desto größer ist die Wahrscheinlichkeit eines Konfliktes bei der Rückführung in den Hauptzweig. Bei diesem Punkt hilft eine disziplinierte Arbeitstechnik mehr als jedes Tool. Wichtig ist, dass die Software in kleinen überschaubaren Inkrementen entwickelt wird und die Resultate nach einem Review direkt in den Produktivcode eingehen. Der nächste Schritt in der Automatisierung ist die Continuous-Delivery (CD). Hierbei wird der Continuous-Integration-Workflow um das Deployment in eine Testumgebung erweitert, in der sich dann z.B. automatisiert Integrationstests durchführen lassen. Ziel von Continuous-Delivery ist es, stets einen Stand der Software vorweisen zu können, der auslieferbar ist. Muss beispielweise ein dringendes Feature ausgeliefert werden, kann mittels CI/CD-Pipelines sichergestellt werden, dass die Reaktionszeit auf die Feature-Anfrage beschleunigt und gleichzeitig die Time-to-Market reduziert wird. Zudem sind dringende, außerplanmäßige Releases ebenfalls kein Problem, da der aktuelle Stand der Software im Hauptzweig stets gebaut, getestet und von technischer Seite abgenommen ist. Die Königsdisziplin der Automatisierung ist das Continuous-Deployment. Hierbei wird der komplette Entwicklungs- und Auslieferungsprozess automatisiert, sodass Änderungen an der Software binnen Minuten oder Stunden den Weg in die Produktion finden. Entgegen der schematischen Darstellung des Lifecylce **(Abb. 2)** genügt es dabei jedoch nicht, ausschließlich das Deployment in die Produktion zu automatisieren. Die Praxis zeigt, dass es gerade bei Microservice-Architekturen darauf ankommt, zusätzlich eine aussagekräftige Testabdeckung durch Integrations- und Ende-zu-Ende-Tests sicherzustellen und auch diese während der CI/CD-Pipeline automatisiert durchlaufen zu lassen. Hierbei ist es notwendig, die dafür vorgesehenen Umgebungen bereit zu stellen und zu konfigurieren. Dies stellt unter Zuhilfenahme moderner Container-Plattformen jedoch kein Problem mehr da.

Testautomatisierung auf Container-Plattformen

Auf Basis von Docker und Kubernetes lassen sich heutzutage binnen Minuten komplexe Infrastrukturen realisieren. Das gilt nicht nur für Sourcecode-Verwaltung, CI-Systeme, Container-Registries und Routings, sondern auch für ganze Softwaresysteme sowie Entwicklungs-, Test- und Integrationsumgebungen. Im folgenden Beispiel wird unter Verwendung von RedHat-OpenShift und Jenkins gezeigt, wie eine CI/CD-Pipeline für eine Todo-App im Container umgesetzt werden kann. Bei RedHat-OpenShift handelt es sich um eine Platform-as-a-Service (PaaS) Lösung **(vgl. Abb. 3)** aus dem

Schematische Darstellung eines Continuous-Deployment-Lifecylce. (Abb. 2)

Cloud-Computing-Umfeld, die ihren Fokus auf die Lösung infrastruktureller Problemstellungen legt. So sind beispielsweise die Verwaltung von Builds, Deployments, Softwarekonfigurationen und Routings, aber auch die Skalierungen von Services sowie die Bereitstellung und Konfiguration von Netzwerken im Funktionsumfang enthalten. Softwareprojekte erhalten innerhalb von OpenShift eine oder mehrere eigene Umgebungen, in dem die zugehörigen Artefakte hinterlegt werden. Dies lenkt den Fokus weg von infrastrukturellen Problemstellungen hin zur eigentlichen Softwareentwicklung (Dev) und den dazugehörigen Betriebsthemen (Ops).



Übersicht verschiedener as-a-Service-Ansätze. (Abb. 3)

```
parameters:
- name: TODO_APP_REPOSITORY
  displayName: Repository url
  description: The URL of the source repository
  required: true
  value: http://gogs.192.168.99.101.nip.io/sven/todo-app.git
```

Innerhalb von OpenShift gibt es verschiedenste Möglichkeiten, CI/CD-Systeme zu verwenden. Eine native Unterstützung erfährt Jenkins, der über ein Build-Konfiguration-Template (Listing 1) angelegt werden kann und anschließend komplett konfiguriert zur Verfügung steht. Seine Pipeline bezieht das System in Form eines Jenkins-File aus einem Git-Repository im Configuration-as-Code-Ansatz und ist somit ausschließlich für die Abwicklung der Builds für das konfigurierte Projekt zuständig. Das Jenkins-File aus dem Repository wird zudem vor jedem Build-Prozess neu geladen und ist daher immer aktuell.

(Listing 1)

```
apiVersion: v1
kind: Template
labels:
  template: jenkins-pipeline
metadata:
  name: jenkins-pipeline
objects:
- apiVersion: v1
  kind: BuildConfig
  metadata:
    labels:
      build: todo-app-pipeline
      app: todo-app
    name: todo-app-pipeline
  spec:
    failedBuildsHistoryLimit: 5
    runPolicy: Serial
    source:
      git:
        uri: ${TODO_APP_REPOSITORY}
      type: Git
    strategy:
      jenkinsPipelineStrategy:
        jenkinsfilePath: infra/jenkins/Jenkinsfile
      type: Source
    successfulBuildsHistoryLimit: 5
    triggers:
    - imageChange: {}
      type: ImageChange
    - type: ConfigChange
```

Die bereitgestellte Jenkins-Instanz ist darüber hinaus so konfiguriert und mit den notwendigen Berechtigungen versehen, um das Scheduling der Worker-Nodes durchzuführen und eine Steuerung des Clusters bzw. des Projekt-Namespaces aus der Jenkins-Pipeline heraus zu ermöglichen. Es können zudem auch andere Projekt-Namespaces gesteuert werden. Dies bedarf jedoch einer expliziten Freigabe über das Kommandozeilenwerkzeug mittels **`oc policy add-role-to-user edit system:serviceaccount:todo-app-dev:jenkins -n todo-app-int`**. Hier wird dem Service-Account des Jenkins aus dem Namespace **`todo-app-dev`** die Berechtigung **`edit`** auf den Namespace **`todo-app-int`** gewährt. Dies befähigt Jenkins, ebenfalls diverse Aktionen wie zum Beispiel Deployments innerhalb der Integrationsumgebung durchzuführen. Um diese Berechtigung vergeben zu können, muss der Nutzer, der diese Konfiguration vornimmt, über ausreichende Berechtigungen innerhalb des Zielprojektes verfügen.

Um nun eine CI/CD-Pipeline bereitstellen zu können, benötigt die Jenkins-Instanz noch das Jenkins-File. Für dieses Beispiel wurde die Pipeline in drei Stages aufgeteilt. Die erste Stage befasst sich mit dem Bauen der Software inklusive Unit-Tests sowie einem Deployment in einer Umgebung für Entwicklertests. Die zweite Stage nimmt ein Deployment der Applikation in einer Integrationsumgebung vor und führt die dazugehörigen Integrationstests aus. Die letzte Stage realisiert das Deployment in die Produktivumgebung. Innerhalb der Development-Stage (Listing 2) wird mittels einer Jenkins-Worker-Node ein Checkout des Quellcodes vorgenommen. Die Information, aus welchem Repository der Sourcecode bezogen werden soll, erhält der Jenkins mit seiner Build-Konfiguration. Anschließend wird die Todo-App innerhalb des Projekt-Namespaces **`todo-app-dev`** gebaut und deployed. Hierzu werden zwei Helfermethoden verwendet (Listing 3 und 4), die wiederkehrende oder unübersichtliche Operationen

innerhalb der Pipeline abstrahieren. Das Resultat des Builds ist ein Docker-Container, der automatisch in der OpenShift-eigenen Docker-Registry hinterlegt wird und somit zur weiteren Verwendung innerhalb des Clusters zur Verfügung steht. Sind diese Operationen erfolgreich, wird anschließend das Tag des Containers aus der Dev-Umgebung in die Int-Umgebung übertragen. Dies macht die Todo-App für ein Deployment innerhalb der Integrationsumgebung verfügbar.

(Listing 2)

```
stage('dev') {
    node{
        checkout scm

        openshift.withProject( "todo-app-dev" ) {
            buildTodoApp()
            deployTodoApp("todo-app-dev.192.168.99.101.nip.io")
            openshift.tag("todo-app-dev/todo-app:latest
            todo-app-int/todo-app:latest")
        }
    }
}
```

(Listing 3)

```
private void buildTodoApp() {
    def buildTemplate = openshift.process("-f infra/yml/s2i-
    todo-app-build-template.yml")
    openshift.apply(buildTemplate)

    openshift.startBuild("todo-app", "--wait=true")
}
```

(Listing 4)

```
private void deployTodoApp(String routeUrl) {
    def deploymentTemplate = openshift.process("-f infra/yml/
    todo-app-deployment-template.yml",
            "-p", "URL=" + routeUrl)
    openshift.apply(deploymentTemplate)

    openshift.selector("dc/todo-app").rollout().latest()
    openshift.selector("dc/todo-app").rollout().status
    ("--watch=true")
}
```

Die Integration-Stage (Listing 5) ist im Vergleich zur Development-Stage strukturell anders aufgebaut. Hier wird über ein **`PodTemplate`** eine spezielle Worker-Instanz mit vorinstalliertem Maven gestartet. Nach dem Checkout des Projekts und dem Deployment können nun Integrationstests, die im Repository hinterlegt sind, mittels Maven gestartet und im Zusammenspiel mit der Todo-Applikation ausgeführt werden. Das Resultat der Integrationstests wird anschließend an Jenkins und OpenShift weitergeleitet, sodass ein fehlgeschlagener Integrationstest einen Abbruch der Pipeline zur Folge hat. Sind die Tests erfolgreich, erfolgt eine Übertragung des Container-Tags in die Produktivumgebung.

(Listing 5)

```
podTemplate(cloud: "openshift", namespace: "todo-app-int") {
    node('maven'){
        checkout scm

        stage('integration-tests') {
            openshift.withProject( "todo-app-int" ) {
                deployTodoApp("todo-app-int.192.168.99.101.nip.
                io")
            }

            sh "mvn clean verify -Dskip.integration.tests=false"


           openshift.withProject( "todo-app-int" ) {
                openshift.tag("todo-app-int/todo-app:latest
                todo-app-prod/todo-app:latest")
            }
        }
    }
}
```

Abschließend wird das Deployment in die Produktivumgebung (Listing 6) vorgenommen.

(Listing 6)

```
stage('prod') {
    node{
        openshift.withProject( "todo-app-prod" ) {
            deployTodoApp("todo-app.192.168.99.101.nip.io")
        }
    }
}
```

Das komplette ausführbare Beispiel inklusive Todo-App, den Integration Tests umgesetzt mit dem Integration Testing Framework Citrus, sowie einer Anleitung zum Aufsetzen der notwendigen lokalen Infrastruktur befindet sich auf GitHub[1].

Fazit:

Die Testautomatisierung im Microservice- und Container-Umfeld stellt uns vor neue Herausforderungen, die es zu bewältigen gilt. Unter Verwendung moderner Technologien, den richtigen Methodiken und einem geeigneten Setup sind diese Herausforderungen jedoch durchaus zu meistern. Mithilfe von Containern, Clustern oder PaaS-Systemen lassen sich ohne großen Aufwand verschiedenste Umgebungen erstellen und über CI-Systeme steuern. Über Mechanismen wie Configuration-as-Code stellt man zudem sicher, dass die Infrastruktur und ihre Konfiguration immer zum aktuellen Stand der Software passen.

Quellen:

1 Beispiel: https://bit.ly/2TASHyy

#JAVAPRO #Cloud #Platform #CamelK

Camel-K - Leichtgewichtige Cloud-Integrations-Plattform

Im Sommer letzten Jahres startete Camel-K[1] als eine Community-getriebene Plattform für das einfache und schnelle Deployment von Apache-Camel[2]-Anwendungen als Cloud-Native-Serverless-Funktionen in Kubernetes oder Openshift. Das Projekt Camel-K verbindet damit die erfolgreiche Enterprise-Integration-Bibliothek Apache-Camel mit dem Serverless-Ansatz. Die in Camel geschriebenen Integrationen können damit direkt auf einer Cloud-Plattform wie Kubernetes ausgeführt werden.

Was ist Apache Camel?

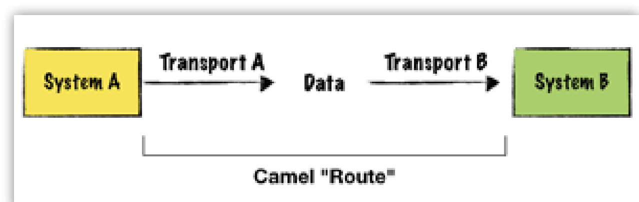
Apache-Camel implementiert als Open-Source-Framework die wichtigsten Enterprise-Integration-Patterns[3]. Diese Patterns wurden bereits 2004 von Gregor Hohpe and Bobby Woolf im gleichnamigen Buch[4] beschrieben und gelten noch heute als zutreffende Lösungsansätze, wenn zwei oder mehr Systeme miteinander integriert werden sollen. Die vorgeschlagenen Lösungsmuster helfen bei der Bewältigung gängiger Herausforderungen, die sich ergeben, wenn unterschiedliche Systeme Daten miteinander austauschen. Dabei geht es vor allem um Themen wie Routing, Transformation, Aggregation, Messaging und Event-Streaming.

Autor:

Christoph Deppisch arbeitet als Senior-Software-Engineer bei RedHat. Dort beschäftigt er sich im RedHat-Fuse-Team vor allem mit Middleware-Integration, Apache-Camel und der Entwicklung der Integrationsplattform Fuse-Online. Als leidenschaftlicher Verfechter von Testautomatisierung gründete und entwickelte er das Open-Source-Integration-Testframework Citrus und teilt seine Erfahrungen gerne in regelmäßigen Blogs, Fachartikeln und als Speaker auf internationalen Konferenzen.

Twitter: https://twitter.com/freaky_styley
Github: https://github.com/christophd

Diese Integrationsmuster werden in Apache-Camel in einfach zu verstehenden Routen **(Abb. 1)** umgesetzt, die man in verschiedenen Laufzeitumgebungen (Stand-alone Fat-JAR, Spring-Boot, WildFly, Apache-Karaf) ausführen kann.



Route in Apache Camel. (Abb. 1)

Seit mehr als zehn Jahren wird Apache-Camel als Framework stetig weiterentwickelt und stellt mittlerweile über 250 Komponenten für die Integration verschiedenster Systeme zur Verfügung. Mithilfe dieser einsatzbereiten Komponenten kann eine Middleware-Integration-Software unterschiedliche Datenformate (JSON, XML, CSV etc.) mit verschiedensten Service-Providern

(z.B. Salesforce, Twitter, Amazon, Google-App-Engine) auf unterschiedlichen Transportwegen (HTTP, JMS, FTP, Kafka-Streams etc.) austauschen.

(Listing 1) zeigt eine einfache Apache-Camel-Route, die Nachrichten über einen Eingangskanal (**`telegram:bots/greeting-bot`**[5]) zunächst anhand des Nachrichteninhalts filtert, diese gefilterten Nachrichten in einzelne Teile splittet und einzeln an einen Ausgangskanal (**`kafka:greeting-words`**[6]) weiterleitet.

(Listing 1)

```
class TelegramToKafka extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("telegram:bots/greeting-bot")
      .filter().simple("${body} starts with 'Hello'")
      .split().tokenize(" ")
      .to("kafka:greeting-words?brokers=my-kafka-broker:9092");
  }
}
```

Das Beispiel zeigt wie einfach man eine Middleware-Integration mit Hilfe der Apache-Camel Java-DSL beschreiben und umsetzen kann. Wird zum Beispiel eine Nachricht **`Hello from Camel K!`** über den Telegram-Bot abgesetzt, aktiviert die Camel-Route die Verarbeitung und das Resultat sind vier Nachrichten **`Hello, from, Camel, K!`**, die einzeln auf den angegebenen Kafka-Topic-Stream gesendet werden. Diese Apache-Camel-Route lässt sich mit Camel-K nun als Serverless-Function direkt in Kubernetes oder OpenShift ausführen und automatisch skalieren.

Warum Serverless?

Das „K" in Camel-K ist eine Anspielung auf die Container-Orchestrierungs-Plattform Kubernetes. Die Technologie rund um Docker und Kubernetes hat in den letzten Jahren einen immensen Erfolgsweg aufzuweisen und ist noch immer in aller Munde. Auf Kubernetes basierend entwickeln sich weitere interessante Technologien und Paradigmen, wie zum Beispiel der Serverless-Ansatz.

Serverless beschreibt die Idee sehr klein geschnittene Anwendungsteile (Serverless-Functions) auf Cloud-Plattformen wie zum Beispiel Kubernetes als hoch skalierbare containerbasierte Einheiten zu betreiben. Was mit der Microservices-Idee begann, wird mit Serverless also noch einen Schritt weiter geführt. Ziel ist es, einzelne Logikbausteine schnell und vor allem einfach in einer containerbasierten Laufzeitumgebung zu deployen. Die Bausteine sollen gezielt eine einzige Funktionalität abbilden und werden durch bestimmte Events getriggert. Wenn die Funktionalität zu einem Zeitpunkt nicht gebraucht wird, kann es sogar dazu kommen, dass die Serverless-Function in Kubernetes komplett auf Null skaliert wird. Dadurch werden Infrastruktur-Ressourcen für andere Dienste frei und können effizient genutzt werden. Wird die Serverless-Function hingegen durch ein Event oder einen Request angesprochen, aktiviert Kubernetes die Anwendung automatisch und skaliert diese entsprechend der Anfragen nach oben. Um diese automatische Skalierung effizient durchführen zu können, bedarf es einigen Voraussetzungen, die gerade in der Java-Welt nicht immer vollständig gegeben sind:

Serverless-Functions
- müssen sich schnell starten und stoppen lassen,
- sollten möglichst wenig Memory und CPU anfordern,
- müssen komplett zustandslos sein.

Vor allem die Zeiten für den Start bzw. Stopp von Java-Prozessen sowie der Speicherbedarf einer JVM kann ein Hindernis für Serverless-Anwendungen sein. Die ständige Skalierung bedeutet ein häufiges Starten und Stoppen der Anwendung. Bei hoher Nutzung der Serverless-Functions bedeutet dies unter Umständen auch viel Memory- und CPU-Bedarf.

Neben diesen Aspekten für die automatische Skalierung gibt der eigentliche Build-Deploy-Zyklus während der Entwicklung ebenfalls Grund zur Optimierung. Mit Tools wie dem Fabric8-Maven-Plugin[7] gibt es bereits effiziente und einfache Wege eine klassische Java-Anwendung als Container in einem Pod in Kubernetes zu deployen. Wenn wir jedoch den klassischen Aufbau mit einer in Java geschriebenen Apache-Camel-Route verpackt in einer Spring-Boot-Web-Anwendung betrachten, dauert ein Build-Deploy-Zyklus eines Java-Containers in etwa 1 bis 2 Minuten. Dies summiert sich bei häufigem Deployment während der Entwicklung und ist im Vergleich zu anderen Cloud-Native-Technologien relativ langsam. Aus diesem Grund sind im Gegensatz zu Java andere Sprachen wie NodeJS oder Go sehr beliebt in der Serverless-Community.

Camel-K setzt genau hier an, um einige dieser Nachteile eines klassischen Java-Containers mit einer darin laufenden Apache-Camel-Anwendung signifikant zu verbessern. Mit Camel-K können Apache-Camel-Integrationen effizient als Serverless-Anwendungen betrieben werden.

Wie funktioniert Camel-K?

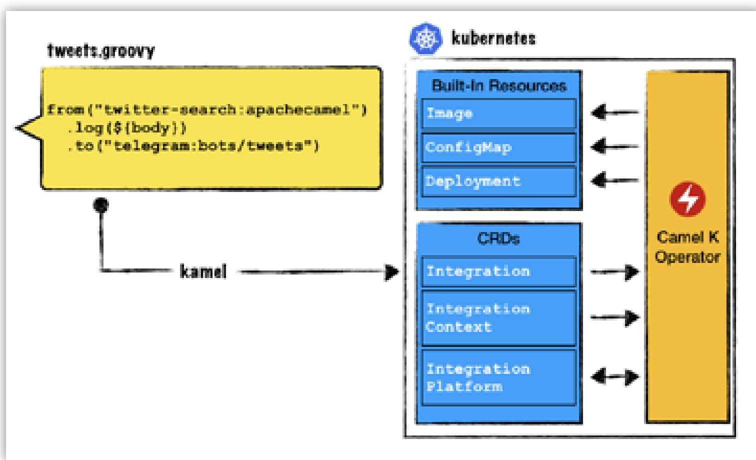
Camel-K nutzt einige Tricks, um sowohl den Memory-Bedarf einer Apache-Camel-Anwendung als auch die Start- und Stopp-Zeiten deutlich zu optimieren. Als Basis definiert Camel-K sogenannte Custom-Resource-Definitions (CRDs) für Apache-Camel und stellt einen Operator bereit, der die CRDs in Kubernetes verwaltet und überwacht. Camel-K definiert 3 unterschiedliche Custom-Resource-Definitions:

- **`Integration`**: Enthält die vom User definierte Route.
- **`Integration-Context:`** Repräsentiert das Basis-Image, welches in der Lage ist Integrations auszuführen. Ein Integration-Context kann gleichzeitig von mehreren Integrations genutzt werden.

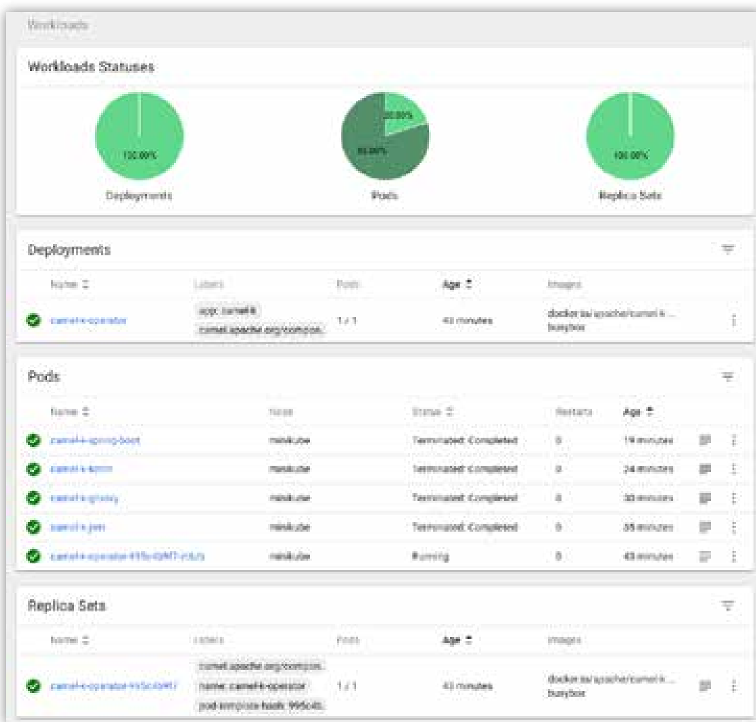
- **`Integration-Platform`**: Definiert die allgemeine Konfiguration der Plattform, um die Arbeitsweise des Camel-K-Operators zu beeinflussen.

Der Camel-K-Operator überwacht diese CRDs und wird automatisch aktiv, wenn eine dieser Custom-Resources neu hinzugefügt wird oder sich eine dieser Custom-Resources ändert. Der Operator erzeugt daraufhin Kubernetes-Built-in-Ressourcen wie Deployments, ConfigMaps oder Images. (Abb. 2) zeigt das Zusammenspiel zwischen Camel-K-Operator, dem Kommandozeilen-Tool kamel und der eigentlichen Routen-Definition in Java, Groovy, Kotlin oder JavaScript.

Durch das extra für Camel-K entwickelte Kommandozeilen-Tool kamel wird die Zusammenarbeit mit dem Operator sehr vereinfacht und der Entwickler konzentriert sich voll und ganz auf die Definition der Apache-Camel-Routen.



Zusammenspiel von kamel CLI, Camel-K-Operator und Kubernetes-Ressourcen. (Abb. 2)



Installierte Camel-K-Ressourcen in Kubernetes. (Abb. 3)

Betrachten wir die Arbeit mit Camel-K nun anhand eines Beispiels. Voraussetzung ist eine Kubernetes- oder OpenShift-Laufzeitumgebung. Für den lokalen Test eignen sich die Tools minikube[8] oder minishift[9]. Zunächst muss das Kommandozeilen-Tool kamel[10] heruntergeladen und entpackt werden. Das Kommandozeilen-Tool unterstützt den Nutzer sowohl bei der Installation von Camel-K im Kubernetes-Cluster, als auch beim späteren Deployment der Serverless-Anwendungen. Folgender Befehl installiert Camel-K auf einem Kubernetes-/OpenShift-Cluster:

(Listing 2)

```
> kamel install
```

Dieser Befehl installiert den Camel-K-Operator sowie die Custom-Resource-Definitions (CRDs). Da diese Installation für den kompletten Cluster gilt, sind gegebenenfalls Cluster-Administrationsrechte erforderlich. In einem lokalen Minikube- oder Minishift-Cluster sollten Administrationsrechte kein Problem sein. Nun sind alle benötigten Camal-K-Ressourcen im Cluster installiert und der Camel-K-Operator ist gestartet (Abb. 3).

Jetzt kann sofort eine erste Serverless-Camel-K-Route deployt werden:

(Listing 3)

```
> kamel run TelegramToKafka.java
```

Der **run** Befehl benötigt nur die Camel-Routen-Definition selbst und erstellt daraus einen Container im Kubernetes-Cluster. Neben Java werden hier auch andere Sprachen wir Groovy, Kotlin und auch JavaScript unterstützt. Das Kommandozeilen-Tool nimmt das Deployment vor und wir sehen neben dem bereits bei der Installation von Camel-K gestarteten Operator nun auch die neue Camel-K-Integration in der Übersicht der Deployments (Abb. 4). Das Resultat ist eine Apache-Camel-Route, die Ihren Dienst als Kubernetes-Pod aufgenommen hat und je nach eingehenden Anfragen automatisch skaliert wird.

(Listing 4) zeigt gleich mehrere in Groovy geschriebene Routen in einer Datei. Diese Groovy-Routen-Definition kann man mit dem **run** Befehl auf gleiche Weise im Kubernetes-Cluster deployen:

Deployment einer Camel-K-Anwendung in Kubernetes. (Abb. 4)

(Listing 4)

```
from("telegram:bots")
  .filter().simple("${body} starts with 'Hello'")
  .split().tokenize(" ")
  .to("kafka:greeting-words")

from("kafka:greeting-words")
  .to("log:words")

context {
  components: {
    telegram: {
      authorizationToken = 'greeting-bot'
    }
    kafka: {
      brokers = 'my-kafka-broker:9092'
    }
  }
}
```

Das `routes.groovy` Beispiel zeigt neben mehreren Routen in einer Datei auch den Einsatz von Context-Objekten, die es erlauben wiederkehrende Konfigurationen zentral auszulagern. Damit lassen sich auch sensitive Daten wie Passwörter und Token-Informationen per ConfigMap und Secrets in Kubernetes laden. Die Camel-K-Anwendung wird wie gewohnt über das Kommandozeilen-Tool im Cluster deployed:

(Listing 5)

```
> kamel run routes.groovy
```

Das komplette Deployment der Camel-K-Serverless-Anwendung in den Clustern kann nun in unter fünf Sekunden durchgeführt werden. Eine Änderung der Routen-Definition und damit ein Redeployment der bereits laufenden Camel-K-Anwendung dauert in der Regel weniger als eine Sekunde. Dies stellt eine deutliche Verbesserung zu den Deployment-Zyklen mit S2i (Source-to-Image) oder normalen Image-Build-Deploy-Ansätzen dar, wenn von einer klassischen Java-Web-Anwendung mit Apache-Camel und zum Beispiel Spring-Boot ausgegangen wird.

Wie ist diese Optimierung möglich?

Der Camel-K-Operator interpretiert die zu startende Camel-Route und wählt die leichtgewichtigste Laufzeitumgebung dafür aus. Zusätzlich erstellt der Operator das minimale Paket für den Code mit ausschließlich benötigten Camel-Komponenten und deren Abhängigkeiten. Zuletzt optimiert der Operator die Camel- und JVM-Parameter für die Ausführung. Das bedeutet, dass der Camel-K-Operator den Quellcode einer Route versteht und ein ideales Deployment erstellt.

Beim Redeployment kann der Camel-K-Operator bereits erstellte Images einfach wiederverwenden, da die Routen-Definition an sich nicht Teil des Images ist, sondern als ConfigMap beim Deployment hinzugefügt wird. Somit sind Änderung der Routen-Definition lediglich eine Änderung der ConfigMap. Ein Rebuild des Images entfällt komplett.

Fazit:

Mit Hilfe des optimierten Build-Deploy-Zyklus kann Camel-K die Apache-Camel-Routen sehr schnell auf einer Cloud-Plattform wie Kubernetes-nativ zur Ausführung bringen. Im Sinne von automatisierter Skalierung wird die Serverless-Anwendung dynamisch gestartet und gestoppt.

Der Memory-Bedarf einer Camel-K-Serverless-Anwendung ist ebenfalls optimiert. Es sind weitere signifikante Optimierungen für dieses Jahr geplant, die sich enorm positiv auf die Leichtgewichtigkeit und Schnelligkeit der Camel-K-Prozesse auswirken. Ahead-of-time Kompilierung und Native-Images sind hier Wege zu noch viel schnelleren Startzeiten und enorm minimiertem RAM-Bedarf. Es bleibt also spannend noch weitere Verbesserungen in diesem Bereich zu sehen.

Eine weitere spannende Entwicklung rund um Camel-K ist die Integration mit Knative[11]. Knative stellt eine weitere Zielplattform von Camel-K dar, wobei vor allem die Zusammenarbeit mit den in Knative definierten Bereichen Serving und Eventing von großem Interesse sind.

Der Java-basierte Ansatz für Serverless-Enterprise-Integration-Anwendungen mit Apache-Camel als zentrale Enterprise-Integration-Bibliothek ist schon jetzt mit den bereits bestehenden Mitteln rund um Camel-K wieder sehr interessant geworden. Die Effizienz von Kubernetes in der Zuweisung von Ressourcen und die automatische Skalierung funktionieren in Camel-K sehr gut und machen Apache-Camel bereit für den Einsatz im Serverless-Umfeld.

Quellen:

1 https://bit.ly/2UShWO9
2 http://camel.apache.org/
3 https://bit.ly/2WnYauq
4 https://bit.ly/2HU4uWI
5 https://bit.ly/1SC2vkI
6 https://kafka.apache.org/
7 https://bit.ly/2WjGBvm
8 https://bit.ly/2E40UaE
9 https://bit.ly/2FB6zVK
10 https://bit.ly/2U0NPIf
11 https://github.com/knative/
12 Mehr zu CRDs und Operatren in Kubernetes: https://coreos.com/operators

#JAVAPRO #DevOps #Agile

Die DASA-DevOps-Prinzipien

Im 1. Teil unserer Artikelserie werden die DASA-DevOps-Prinzipien im Überblick dargestellt. Betrachtet man alle Prinzipien und Zusammenhänge, hat man eine solide Basis für die Entwicklung in Organisation und Personal geschaffen.

DevOps als Kunstwort beschreibt eine Philosophie, die die beiden Namensgeber Entwicklung und Betrieb zusammenbringt. Das Ziel ist es, kontinuierliche Softwarebereitstellung sicherzustellen, um schneller auf Marktveränderungen reagieren zu können und Kundenanforderungen besser gerecht zu werden. Dabei sind die gegenläufigen Ansätze einer agilen Softwareentwicklung, nämlich eine schnellere und häufigere Lieferung neuer Features und der Anspruch an einen sicheren und stabilen Betrieb in Einklang zu bringen. In der Praxis kann das Verständnis dieser Zielsetzung noch verbessert werden: Häufig wird DevOps lediglich als Aufbau einer kontinuierlichen Bereitstellung von Applikationsänderungen verstanden und damit eher auf die technische Sicht reduziert. Die erfolgreiche Betrachtung des Lebenszyklus einer Anforderung der Fachbereiche von der ersten Formulierung bis zur produktiven Nutzung im Live-System erfordert jedoch einen umfassenderen Blick. DevOps baut also eine Brücke zur Zusammenarbeit zwischen Kunden, Entwicklung und Betrieb, um eine hochperformante IT-Organisation zu schaffen.

Autor:

Das Motto von Dierk Söllner lautet: „Ich mache Teams und Menschen erfolgreich!“. Als zertifizierter Business Coach (dvct e.V.) unterstützt er Teams sowie Fach- und Führungskräfte bei aktuellen Herausforderungen durch professionelles Coaching. Seine langjährige und umfassende fachliche Expertise als ITIL Expert, DASA Ambassador, DevOps Trainer und zertifizierter Scrum Master machen ihn zu einem kompetenten und empathischen Begleiter bei Personal-, Team und Organisationsentwicklung. Er betreibt den DevOps-Podcast „Auf die Ohren und ins Hirn“, hat einen Lehrauftrag zu „Agiles IT Service Management“ und das Fachbuch „IT Service Management mit FitSM“ publiziert.

www.dsoellner.de

Im Unterschied zu bestehenden Frameworks wie ITIL oder Scrum ist DevOps kein neues Framework. Es gibt weder einen Rechteinhaber noch eine Institution, die sich als inhaltlicher Eigentümer des Begriffes DevOps sehen kann. Auch bei genauerer Betrachtung sucht man rechtliche Hinweise wie ® bei ITIL oder ™ wie beim Scrum-Guide vergeblich. Insofern existieren weder allgemeingültige Definitionen noch Vorgaben, was eine IT-Organisation tun sollte, um DevOps anzuwenden oder einzuführen. Eine wachsende Community hat sich etabliert, um eigene Erfahrungen, praktische Überlegungen und Sichtweisen auszutauschen und so diese Philosophie in aller Öffentlichkeit und gemeinsam weiter zu entwickeln. In deutschen Unternehmen etablieren sich mehr und mehr Initiativen, die sich mit den Inhalten, Gestaltungsmöglichkeiten und Vorteilen von DevOps beschäftigen. Die DevOps-Agile-Skills-Association (DASA) unterstützt diese Bewegung mit einem Kompetenz-Framework für IT-Mitarbeiter. Dieses Kompetenz-Framework basiert auf sechs Prinzipien und wird in einem

modernen Ausbildungskonzept in die Praxis umgesetzt. Das Ziel der DASA ist die Bildung und Unterstützung einer offenen und weltweit präsenten sowie aktiven DevOps-Community. Diese wird von den Mitgliedern getragen und steht allen Interessierten zur Teilnahme offen, um u.a. das Kompetenz-Framework weiter zu entwickeln. Zur Erreichung dieses Zieles ist die DASA in folgenden Themen aktiv:

- Förderung eines Kompetenz-Frameworks für DevOps, basierend auf einem Set von Prinzipien.
- Entwicklung und Verbreitung eines herstellerunabhängigen DevOps-Ausbildungskonzeptes für IT-Professionals.
- Verständnis und Bewusstsein für die Notwendigkeit zur Entwicklung von aktuellen Kompetenzen und Skills wecken.
- Qualitätssicherung für Seminare und Trainings durch ein allgemein verfügbares Zertifizierungsschema für das DevOps-Kompetenz-Framework.
- Entwicklung eines passenden Trainingskonzeptes für die jeweiligen Rollen im Framework.

Die DASA-DevOps-Prinzipien beschreiben grundlegend, wie eine IT-Organisation sich und vor allem seine Mitarbeiter in Richtung DevOps entwickeln sollte. In diesem Beitrag wird dazu ein Überblick gegeben, bevor im Folgebeitrag das Kompetenz-Framework detaillierter betrachtet wird. Während also die sechs Prinzipien eher in Richtung einer organisationsweiten Betrachtung gehen, zielt das Kompetenz-Framework auf die Ausbildung der Mitarbeiter ab.

Prinzip 1: Customer-Centric-Action

It is imperative nowadays to have short feedback loops with real customers and end-users, and that all activity in building IT products and services centers around these clients. To be able to meet these customers' requirements, DevOps organizations require the guts to act as lean startups that innovate continuously, pivot when an individual strategy is not (or no longer) working, and constantly invests in products and services that will receive a maximum level of customer delight. [2]

Die 6 DASA-DevOps-Prinzipien. (Abb. 1)

Kundenzentriertheit im DevOps bedeutet, dass die Feedback-Zyklen mit Kunden und Anwendern immer kürzer und direkter werden. Aus der agilen Softwareentwicklung ist dieser Ansatz bekannt und hat sich als sehr erfolgreich herausgestellt. Um die gesamte Wertschöpfungskette in der IT zu unterstützen, muss das Feedback bis zur produktiven Nutzung ausgedehnt werden. Dabei werden bspw. A/B-Tests eingesetzt oder einfach das Nutzungsverhalten analysiert. Alle Aktivitäten der Serviceerbringung müssen sich an den Kunden und Anwendern orientieren. Die beschriebene Ausrichtung an der Wertschöpfungskette muss zu einer permanenten Überprüfung der Aktivitäten führen, so dass überflüssige Arbeiten schnell erkannt und eliminiert werden. IT-Organisationen müssen dazu die Fähigkeit erlangen, sich ähnlich wie junge Start-Ups kontinuierlich neu zu erfinden und schnell auf veränderte Einflüsse zu reagieren. Konkret heißt das, mit Blick auf die Verschwendungsarten aus dem Lean-Management, den Fokus auf die Vermeidung von Verschwendung zu legen. Das resultiert in einer kontinuierlichen Weiterentwicklung der Strategie und den daraus abgeleiteten Aktivitäten, auch durch kontinuierliche Investition in Produkte und Services mit Blick auf ein Höchstmaß an Kundenzufriedenheit. Ziel sollte es sein, das Thema Servicequalität stärker in den Köpfen der IT-Mitarbeiter zu verankern. Dazu gehören dann neben der Anpassung in der eigentlichen Umsetzung auch das Umdenken in der täglichen Arbeit der Mitarbeiter und die Offenheit für neues Vorgehen. Die Kundenorientierung sollte darüber hinaus auch organisatorische Veränderungen nach sich ziehen, bspw. durch Auflösen der bestehenden organisatorischen Silos zugunsten von produktorientierten Teams.

Prinzip 2: Create-with-the-End-in-Mind

Organizations need to let go of waterfall and process-oriented models where each unit or individual works only for a particular role/function, without overseeing the complete picture. They need to act as product companies that explicitly focus on building working products sold to real customers, and all employees need to share the engineering mindset that is required actually to envision and realize those products. [3]

Mit dem Prinzip "Schon zu Beginn an das Ergebnis denken" versucht DevOps die klassische prozessorientierte Vorgehensweise mit individuellen Rollen und Funktionen in eine produktorientierte Organisation zu verwandeln. Diese agiert mit dem Ziel, funktionierende Produkte an Kunden zu liefern und mit einer dazu passenden Denkweise zu agieren. Diese Zielsetzung manifestiert sich in der entsprechenden Aufbau- und Ablauforganisation. Diese muss den Überblick über das übergeordnete und angestrebte Ergebnis sicherstellen. Wichtig ist, dass die beteiligten Mitarbeiter diese durchgängige Sichtweise verinnerlicht haben, um daraus ableitend Produkte und Services zu erschaffen. IT-Organisationen müssen als Produktunternehmen agieren, die sich explizit darauf konzentrieren, funktionierende Produkte zu entwickeln, die an Kunden geliefert werden. Alle Beteiligten müssen die umfassende Denkweise teilen, die erforderlich ist, um diese Produkte zu konzipieren und zu realisieren. Es gibt ein Team

für ein Produkt und es übernimmt die Verantwortung für alle Prozesse, Funktionen und Aktivitäten innerhalb dieses Produktes. Dieser Ansatz verfolgt das Ziel, das Ganze zu sehen und zu erkennen und nicht nur ein Teil einer Prozesskette oder einer Funktion zu sein. Weiterhin erreicht dieser Ansatz die drastische Reduzierung der Kommunikation mit anderen Abteilungen und sorgt durch die permanente Verkürzung und Verstärkung der Rückkopplungsschleifen für ein immer stabileres und sichereres Arbeitssystem.

Prinzip 3: End-To-End-Responsibility

Where traditional organizations develop IT solutions and then hand them over to Operations to deploy and maintain these solutions, in a DevOps environment teams are vertically organized such that they are fully accountable from concept to grave. IT products or services created and delivered by these teams remain under the responsibility of these stable groups. These teams also provide performance support, until they become end-of-life, which greatly enhances the level of responsibility felt and the quality of the products engineered. [4]

Eine Ende-zu-Ende-Verantwortung bedeutet aus Sicht von DevOps, dass die traditionelle Trennung von Entwicklung und Betrieb zugunsten von voll verantwortlichen Teams, man könnte sagen, von der Wiege bis zur Bahre, aufgelöst wird. Diese Teams entwickeln und betreiben die Services ebenso wie sie Support leisten. Dieser Umstand erhöht deutlich das Niveau der gefühlten und realen Verantwortung und somit die Qualität der entwickelten Produkte und Services, denn im Sinne des RACI-Modells liegt sowohl die Ergebnis- als auch die Durchführungsverantwortung in einem Team. Die IT erreicht mit dieser Sichtweise wieder den Fokus auf das IT-Produkt und den Kunden zu setzen. Durch produktverantwortliche Teams können stabilere und qualitativ hochwertigere IT-Produkte als bisher entwickelt werden. Hierbei müssen bestehende Organisationen, Prozesse und Arbeitsweisen analysiert und dem Prinzip angepasst werden. Eine neue Denkweise innerhalb des Unternehmens ist dabei erforderlich. Je nach genutztem IT-Produkt können hierbei unterschiedliche Zielsetzungen angegangen werden.

Prinzip 4: Cross-Functional-Autonomous-Teams

In product organizations with vertical, fully responsible teams, these teams need to be entirely independent throughout the whole lifecycle. That requires a balanced set of skills and also highlights the need for team members with T-shaped all-round profiles instead of old-school IT specialists who are only knowledgeable or skilled in for example testing, requirements analysis or coding. These teams become a hotbed of personal development and growth. [5]

Die Forderung nach cross-funktionalen, autonomen (vertikal organisierten) Teams aus Sicht von DevOps ist eine Weiterentwicklung agiler Ansätze. Scrum als bekanntestes agiles Framework setzt ebenfalls auf diese Teamorganisation. Um eine hochwertige Serviceerbringung sicherzustellen, ist ein fein ausbalanciertes Set an Wissen und Fähigkeiten notwendig. Es ist also ein sehr gutes T-Shape-Profil im Team und bei den Teammitgliedern notwendig anstelle des klassischen IT-Spezialisten. Das bedeutet nicht, dass diese Teams keine Spezialisten mehr benötigen. Ähnlich wie bei einer guten Fußballmannschaft sollte jedoch jeder Spieler das Ganze im Blick haben - gemäß dem Motto „Gute Verteidigung beginnt beim Stürmer!" - und flexibel auf verschiedenen Positionen einsatzbar sein. Cross-funktionale, autonome Teams können somit Innovationen ganzheitlich denken. Gleichzeitig können sie das gesamte Aufgabenspektrum über den kompletten Produktlebenszyklus eigenverantwortlich und in effizienter Zusammenarbeit erledigen. Voraussetzung ist dabei der Kulturwandel in den IT-Organisationen. Er wird begleitet durch eine schlanke Teamorganisation und -steuerung sowie die vielseitigen Fähigkeiten der Mitarbeiter. So wird das Team gemeinsam zum Unternehmenserfolg beitragen.

Prinzip 5: Continuous-Improvement

End-to-end responsibility also means that organizations need to adapt continuously in the light of changing circumstances (e.g. customer needs, changes in legislation, new technology becomes available). In a DevOps culture, a strong focus is put on continuous improvement to minimize waste, optimize for speed, costs, and ease of delivery, and to continuously improve the products/services offered. Experimentation is therefore an important activity to embed and develop a way of learning from failures is essential. A good rule to live by in that respect is "if it hurts, do it more often".[6]

Das Prinzip von kontinuierlicher Verbesserung beschreibt aus Sicht von DevOps die Tatsache, dass sich moderne IT-Organisationen permanent an verändernde Bedingungen wie Kundenanforderungen, technologische Rahmenbedingungen, Gesetze und Verordnungen usw., anpassen können müssen. DevOps setzt dabei auf Prinzipien von Lean-Management, um Verschwendung zu vermeiden sowie Kosten zu senken und die Liefergeschwindigkeit zu erhöhen. Experimente werden gefördert und eine neue Fehlerkultur, wie „Fail fast, fail often" etabliert, die darauf abzielt aus Fehlern zu lernen. Für kontinuierliche Verbesserung müssen Rahmenbedingungen in den IT-Organisationen geschaffen werden: Eine Kultur, in der Experimente und Lernen aus Fehlern akzeptiert sowie unterstützt wird und in der Transparenz durch Kennzahlen vorherrscht. Ziel ist es, frühzeitig Veränderungsbedürfnisse zu erkennen und diese durch kontinuierliche Anpassung und Verbesserungen zu jeder Zeit vornehmen zu können. Diese Grundausrichtung benötigen IT-Organisationen für ihren individuellen Erfolg in dieser schnelllebigen Welt. Ein ideales Vorgehen, das Best-Practice, gibt es dabei nicht. Jedes Unternehmen muss selbst einen Weg als eigene Herausforderung und große Chance für den individuellen Erfolg finden.

Prinzip 6: Automate-Everything-You-Can

To adopt a continuous improvement culture with high cycle rates and to create an IT organization that receives instant feedback from end users or customers, many organizations have quite some waste to eliminate. Fortunately, in the past years, enormous gains in IT development and operations can be made in that respect. Think of automation of not only the software development process (continuous delivery, including continuous integration and continuous deployment) but also of the whole infrastructure landscape by building next-gen container-based cloud platforms that allow infrastructure to be versioned and treated as code as well. Automation is synonymous with the drive to renew the way in which the team delivers its services.[7]

Die DASA-DevOps-Prinzipien als Basis für ein Kompetenz-Framework. (Abb. 2)

Automatisiere alles was du kannst, ist eine im Prinzip unmissverständliche Forderung. Sie setzt unter anderem auf der kontinuierlichen Verbesserung auf bzw. unterstützt diese, um schnelle Lieferzyklen zu realisieren, die dann zu sofortigem Feedback durch die Kunden führen. Die Automation umfasst dabei nicht nur den Entwicklungsprozess, sondern auch die darunterliegende Infrastruktur. Unter dem Begriff Infrastructure-as-Code wird damit eine neue Art der Servicelieferung beschrieben, die Prinzipien aus der Softwareentwicklung auf den IT-Betrieb überträgt. Standardisierung und Automation ist ein wesentliches Mittel, um die Voraussetzung für die erfolgreiche Adaption einer DevOps-Kultur zu schaffen, die der Maxime Everything-as-code folgt.

Die weitreichende Forderung, alles was möglich ist zu automatisieren, stellt hohe Anforderungen an die IT-Organisation. Zunächst bezieht sich das auf die Development-Pipeline, da diese einen zentralen Prozess in der Softwareentwicklung darstellt. Die Integration von Continuous-Delivery führt zu einer Anpassung der angrenzenden Prozesse und Systeme. Die Integration von On-Premise- sowie Cloud-Komponenten erfordert dann ein hohes Skillset der DevOps-Ingenieure, sodass zwar durch die Automatisierung Tätigkeiten auf die Systeme verlagert werden, aber neue und anspruchsvollere Tätigkeitsgebiete entstehen. Die Integration von DevOps in das Unternehmen endet hierbei nicht mit der Auslieferung von Softwarepaketen, sondern kann darüber hinaus weitere Relevanz erhalten. Verändert sich die Unternehmenskultur von der eher klassischen und manuellen Ausrichtung hin zu Unternehmensprozessen mit besonderem Fokus auf Automatisierung, können die freien Ressourcen zur Verbesserung der Marktposition genutzt werden.

Fazit:

Bei Betrachtung aller DASA-DevOps-Prinzipien und deren Zusammenhängen wird eine solide Basis für Organisations- und Personalentwicklung gelegt. Die beschriebenen Inhalte und Forderungen an die Organisation und die beteiligten Personen sind entscheidend für die notwendigen Veränderungen sowohl in den Organisationen, bspw. bei Kultur oder Prozessen, als auch bei den Mitarbeitern, bspw. Einstellung, Kompetenzen und Wissen.

Darüber hinaus identifziert das DASA-Kompetenz-Framework acht Wissensbereiche sowie vier Verhaltensbereiche, die in DevOps und somit modernen IT-Organisationen relevant sind. Dieses Kompetenz-Framework baut auf den sechs DASA-DevOps-Prinzipien auf, die im ersten Teil unserer Artikelserie erläutert wurden. Im zweiten Teil der Serie werden wir insbesondere auf die Kompetenzen, Fähigkeiten und das Wissen der Mitarbeiter im DevOps-Umfeld eingehen.

Quellen:

1 http://bit.ly/2YfhBqf
2 http://bit.ly/2JB5boL
3 http://bit.ly/2Vn5jPu
4 http://bit.ly/2VQXktp
5 http://bit.ly/2V6xH3D
6 http://bit.ly/2VaIMAK
7 http://bit.ly/2VPT6Cx

#JAVAPRO #Cloud #Security

Die Cloud ist nicht unantastbar

Bedrohungen, denen Cloud-Umgebungen ausgesetzt sind, decken sich in vielen Punkten mit den Gefahren für Inhouse-Netze. Oft fehlt Unternehmen der Überblick über die Anzahl der Geräte im Netz, was Hackern Vorteile bietet. Dieser Artikel deckt elf Risiken im Zusammenhang mit Cloud-Computing auf und gibt konkrete Empfehlungen dazu.

1. Datenlecks

Auf seine Daten achtet jeder Konzern selbst. Was einfach klingt, erweist sich im IT-Alltag oft als Lippenbekenntnis. Abhängig vom Geschäftsmodell sorgen Datendiebstähle auf vielerlei Weise für Schaden am Unternehmen. Neben Gewinneinbußen und rechtlichen Konsequenzen droht ein Vertrauensverlust der Kunden. Bei der Wahl eines Cloud-Anbieters sollten multifaktorielle Sicherheitskontrollen den entscheidenden Ausschlag geben. Was passieren kann, wenn diese Kontrollmechanismen fehlen, zeigt eine Standort-Tracking-App für Familien: Kürzlich konnten Cyber-Kriminelle auf die Daten der App Family-Locator zugreifen, mit deren Hilfe Familienmitglieder untereinander den Standort teilen. Durch eine unzureichend geschützte Mongo-DB-Datenbank standen dem Angreifer mehrere Wochen lang die Live-Standortdaten der 238.000 Nutzer zur freien Verfügung[1].

Eine Studie des Ponemon-Institute[2] kam zu dem Schluss, dass Datendiebstahl für Cloud-basierte Unternehmen um ein Vielfaches mehr Schaden anrichtet als bei Inhouse-IT-Strukturen und

Autor:

Pierre Gronau ist Inhaber der 2011 gegründeten Gronau IT Cloud Computing GmbH mit Firmensitz in Berlin. Seit über 20 Jahren arbeitet er für namhafte Unternehmen als Senior IT-Berater mit umfangreicher Projekterfahrung. Zu seinen Kompetenzfeldern gehören Server-Virtualisierungen, moderne Cloud- und Automationslösungen sowie Informationsschutz.

mit gesteigerter Wahrscheinlichkeit vorkommt. 613 Mitarbeiter mit höherer IT-Funktion aus verschiedenen Unternehmen treffen in der Studie eine Aussage darüber, wie sicher sie sich gegenüber Datenlecks fühlen. Das Ergebnis ist ernüchternd: Der Großteil behauptet, die Sicherheitsvorkehrungen würden im eigenen Unternehmen den Risiken des Cloud-Computing nicht gerecht. Das liege an fehlendem Überblick aufseiten der IT über Geräte und Software in der Cloud. Auch dem Cloud-Provider gegenüber zeigen sich die Mitarbeiter skeptisch: Sie gehen nicht davon aus, im Falle eines Datenlecks hinreichend schnell benachrichtigt zu werden.

2. Unzureichendes Identitäts- u. Zugriffsmanagement

Datenverstöße sowie andere Angriffe resultieren häufig aus laxer Authentifizierung, schwachen Passwörtern und mangelhaftem Schlüssel- oder Zertifikatsmanagement. IT-Abteilungen müssen hier Nutzen und Risiken in einem Balanceakt abwägen. Auf der einen Seite steht die Effizienz der Zentralisierung von Identität. Auf der anderen Seite stellt ein zentrales Verzeichnis, das Repository, ein lohnendes Angriffsziel dar. Unternehmen sollten für höhere Sicherheit auf Multifaktor-Authentifizierung wie Zeitpasswörter, telefonbasierte Authentifizierung und Smart-Card-Zugriffsschutz bauen. Diesen Schutz hätte auch Instagram gebraucht: Hier fand ein Sicherheitsforscher 2016 heraus, dass der Passwort-Reset-Prozess des sozialen Netzwerkes einem Angreifer erlaubte, sich Zugang auf die Passwort-Wiederherstellungsseite zu verschaffen, ohne Zugangsdaten einzugeben.
Das bekannte Online-Spiel Fortnite gewährte Hackern Anfang diesen Jahres durch einen Log-in-Bug Zugriff auf mehrere Millionen Accounts, mit denen sie Ingame-Einkäufe tätigen konnten.[3]

3. System-Schwachstellen

Organisationen teilen sich Speicher, Datenbanken und andere Ressourcen in unmittelbarer Nähe – so entstehen neue Angriffsflächen und Potenziale für ausnutzbare Fehler. IT-Teams können Angriffe auf solche System-Schwachstellen jedoch mit Basis-IT-Prozessen abmildern. Einer dieser Prozesse ist das zügige Patchen. Change-Control-Prozesse, die Notfall-Patches adressieren, stellen sicher, dass alle Korrekturmaßnahmen ordnungsgemäß dokumentiert und von Technik-Teams überprüft werden. Das optimale Zeitfenster hierfür beträgt vier Stunden.

4.Neue Möglichkeiten für Kriminelle

Phishing und Betrug, zwei alte Bekannte, erreichen durch Cloud-Applikationen eine neue Dimension. Das Fälschen oder Manipulieren von Daten verhindern Unternehmen nur mit einem Sicherheitskonzept, das jede Aktion auf eine eindeutige Identität zurückführt. Jede Kombination von Zugangsdaten sollten sie gründlich schützen - keine leichte Aufgabe, die Innovationen in der Überwachung erfordert.

5. Böswillige Eingeweihte

Insider-Bedrohung hat viele Gesichter: ein aktueller oder ehemaliger Mitarbeiter, ein Systemadministrator, Auftragnehmer oder Geschäftspartner. Dabei reicht das Spektrum böswilliger Aktionen von forciertem Datenmissbrauch bis zu Datendiebstahl. Dies musste auch der Spiele-Publisher Zynga[4] erfahren. Im November 2016 kopierten Mitarbeiter eine große Menge Spielerdaten vom Google-Drive-Konto des Unternehmens auf einen USB-Stick. Ihr Ziel war es, sich nach Verlassen des Unternehmens der Konkurrenz anzuschließen. Inside-Jobs sind gerade bei Cloud-basierten Unternehmen ein heikles Thema.[5] Durch BYOD-Richtlinien gelangen Unternehmensdaten zunehmend auf private Geräte und der Überblick schwindet: Wann wurden Daten von Hackern entwendet, wann gingen sie einem Mitarbeiter aus Versehen verloren? Security-Verantwortliche brauchen Know-how und Feingefühl, um einen Kontrollmechanismus zu schaffen der wirkt, ohne zu überwachen. Kein Mitarbeiter möchte das Gefühl haben, in einem Polizeistaat zu arbeiten.

6. Fortgeschrittene andauernde Bedrohungen

Die CSA bezeichnet fortgeschrittene persistente Bedrohungen (APTs) als parasitäre Angriffsformen. APTs infiltrieren Systeme, um Fuß zu fassen und exfiltrieren dann Daten und geistiges Eigentum über einen längeren Zeitraum hinweg. Mögliche Einstiegspunkte bilden neben direkten Angriffen auch gezielter E-Mail-Betrug, sogenanntes Spear-Phishing sowie Attacken über USB-Treiber. Um gewappnet zu sein, müssen sich IT-Abteilungen stets über die neuesten Angriffe auf dem Laufenden halten. Zusätzlich sorgen regelmäßig erneuerte Awareness-Programme für mehr Wachsamkeit gegen Parasiten.

7. Keine Datensicherung, keine Gnade

Berichte über Datenverluste aufseiten der Cloud-Provider nehmen ab, obwohl Hacker sich nach wie vor angriffslustig zeigen und Daten aus Unternehmensstrukturen oder Rechenzentren löschen. Daher werden täglich geplante Datensicherungen zum Must-have. Wer noch mehr sichergehen will, trennt Daten physisch auf verschiedene Orte oder Netze.

8. Ungenügende Sorgfaltspflicht

Organisationen, die Cloud-Dienste ohne Verständnis für die damit verbundenen Risiken nutzen, nehmen kommerzielle, technische, rechtliche und Compliance-relevante Risiken in Kauf. Sind Entwicklungsteams nicht mit Cloud-Technologien vertraut, können betriebliche und architektonische Probleme auftreten. Daher müssen Entwickler eine umfassende Risikoprüfung durchführen, eine Due-Dilligence, um die mit ihren Cloud-Services verbundenen Risiken einzuschätzen. Die Sorgfaltspflicht im Cloud-Umfeld gilt immer und insbesondere bei Cloud-Migrationen, Zusammenlegung und Outsourcing.

9. Schädliche Nutzung von Cloud-Services

Wer an die Hardware-Ressourcen hinter einer Cloud gelangt, dem gelingt es eine Armada von Geräten für sich zu nutzen. Hacker führen damit nicht nur simple DDoS-Attacken aus, sondern knacken auch Verschlüsselungen oder minen[6] Crypto-Währungen. Attacken auf derartige Ressourcen haben meist zwei Dinge im Blick: Datendiebstahl beziehungsweise das Löschen von Unternehmensdaten und Downtime, die Teil einer weiteren Attacke sein kann oder dem anvisierten Unternehmen Umsatzeinbußen einbringt.

10. DoS-Attacken

Cyber-Kriminelle nutzen schon lange DoS-Attacken, die durch Anfragenüberflutung die Server des Zieles in die Knie zwingen. Sie erlangen aber durch Cloud-Services eine neue Bedeutung, da sie nun nicht mehr nur Internetseiten, sondern ganze Dienstleistungen paralysieren. Github, eine Cloud zum Teilen von Code, erlebte 2018 einen Rekord-Angriff. Sage und schreibe 1,35 Terabit Traffic pro Sekunde erreichten die Server und überforderten sie für mehrere Minuten[7]. Laut Netscout, einem Anbieter von Produkten für Netzwerkleistungsmanagement, sorgen DoS-Attacken auf dem amerikanischen Markt pro Jahr für einen Schaden von 10 Milliarden US-Dollar[8]. Auch die Quantität der Angriffe nimmt von Jahr zu Jahr stark zu.

11. Geteilte Technologie-Schwachstellen

Schwachstellen in gemeinschaftlich genutzter Technologie, also Infrastruktur, Plattform und Anwendung, stellen eine erhebliche Bedrohung für Cloud-Computing dar. Tritt eine Schwachstelle auf einer Ebene auf, betrifft sie alle Ebenen. Wird eine integrale Komponente kompromittiert, setzt sie die gesamte Umgebung potenziellen Verletzungen aus. Um dies zu verhindern, empfiehlt die CSA eine tiefgehende Verteidigungsstrategie: Multifaktor-Authentifizierung, Einbruchmeldesysteme, Netz-Segmentierung und Ressourcen-Aktualisierung bilden bei dieser Strategie die wesentlichen Bausteine.

Quellen:

1 https://tcrn.ch/2VSGSsM
2 http://bit.ly/2Wyw5Rs
3 https://wapo.st/2VUcKNC
4 http://bit.ly/2Lvj623
5 http://bit.ly/2Vq2TQg
6 http://bit.ly/2JpSzAw
7 http://bit.ly/304BC4y
8 http://bit.ly/2HcizOw

Bildrechte: Gronau IT Cloud Computing GmbH

Die Tücken des Cloud Computing. (Abb.1)

#JAVAPRO #DevOps

DevOps-Sportfreunde

Der folgende Artikel knüpft an den Beitrag „Skills, Tools und das richtige Mindset für DevOps" – JAVAPRO 03-2018[1] an und schlägt die Brücke zwischen den Konzepten und Prinzipien hin zur gelebten Praxis. Wenn man das Thema DevOps mit einer Sportart vergleicht, dann kommt nach der allgemeinen Beschreibung der Spielregeln aus dem vorhergehenden Artikel nun die Wahl einer geeigneten Spieltaktik (Organisation) und Ausrüstung (Technologie). Drei Sportler schildern ihre persönlichen Erfahrungen aus Softwareprojekten.

Beim Thema DevOps werden Ziele angestrebt wie etwa Effizienzsteigerungen, die Erhöhung der Kundenzufriedenheit und Kostenreduktionen. Auf gleicher Augenhöhe gilt es aber die Einführung von transparenten Prozessen, cross-funktionalen Teams und die schnelle Reaktion auf neue Anforderungen im Blick zu haben. All dies sollte im Kontext einer modernen IT stehen, zu der Themen wie Agilität, Microservices und Cloud gehören.

Zwei zentrale Besonderheiten sind hier hervorzuheben: Einerseits bildet die Suche nach einer passenden Organisationsform die Grundlage einer funktionierenden DevOps-Kultur[2]. Des Weiteren erkennt man meistens erst während der Transition die Menge an manuellen, repetitiven Schritten, die zuvor überhaupt nicht dokumentiert waren. Will man diese Punkte einhalten und messbar machen, so kann man folgende drei Aspekte betrachten:

1. Herausforderungen erkennen: Müssen während der DevOps-Transformation alte Prozesse und Legacy-Produkte aufrechterhalten werden? Wie erlangen alle Entwicklerteams ein Bewusstsein für Betriebsthemen? Wie verhindert man, dass man durch Betriebsthemen den Fokus auf die Feature-Entwicklung verliert? Wie kann man die radikalen Freiheiten aus dem DevOps-Gedanken mit Vorgaben zur IT-Compliance vereinbaren?
2. Strategien finden: Soll man dedizierte DevOps-Teams als Hub zwischen Entwicklern und klassischen Rechenzentren bilden? Welche Ansprüche hat man an eine Continuous-Integration- und Continuous-Deployment-Pipeline? Wie kann man die Entwicklerteams mit genau den Monitoring-Messpunkten versorgen, die relevant sind?
3. Verbesserungen anstreben: Was will man mit DevOps erreichen? Sollen die Release-Zyklen kürzer werden (Time-to-market)? Soll die Reaktionsfähigkeit bei Fehlern verbessert werden (Stabilität und Robustheit)? Sollen bereits früh in der Planung von Features Problemzonen im späteren Softwarebetrieb erkannt werden? Soll durch Standardisierung mittels bewährter DevOps-Modelle eine Nachhaltigkeit und hohe Qualität der Wissensbasis erzielt und sollen technische Schulden dadurch vermieden werden?

Autor:

Matthias Engelke arbeitet seit Anfang 2017 bei der Pentasys. Als großer Befürworter von DevOps arbeitet er an verschiedenen Themen wie Infrastructure as Code, Automatisierung, Continuous Integration und Continuous Delivery. Er interessiert sich insbesondere für CAAS-Systeme wie Kubernetes und OpenShift und fühlt sich in der Cloud zuhause.

Christopher Hensel ist seit 2016 als Software-Entwickler bei der PENTASYS AG in agilen Projekten tätig. Neben der Entwicklung von Webanwendungen mit Java und Spring gilt sein Interesse dem Clean Code und der Automatisierung im Rahmen von CI/CD.

Dr. Florian Zierer arbeitet seit 2016 bei der PENTASYS AG. Seine Schwerpunkte liegen in den Bereichen Continuous Integration und Automatisierung sowie in der Programmierung mit Java im Bereich Webapp-Entwicklung und Scala im Bereich Spark/Hadoop. Er interessiert sich besonders für die Qualitätssicherung in agilen Projekten und die Transition von klassischen Systemlandschaften hin zu einer DevOps-Kultur.

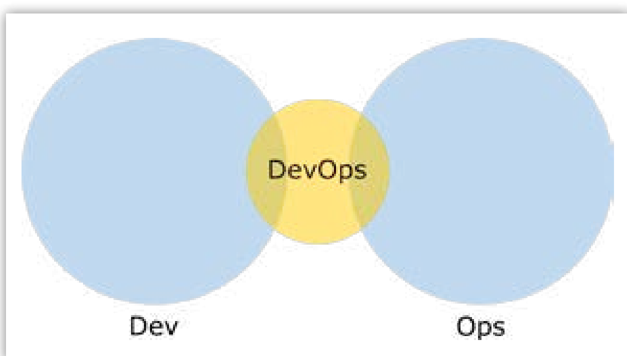
Motiviert durch diese Aspekte beschreiben die folgenden Erfahrungsberichte verschiedene Projekte und ihre jeweils gewählten Ansätze. Die Schilderungen unterscheiden sich einerseits in der Perspektive: Teil 1 blickt auf das Zusammenspiel mehrerer Teams, Teil 2 beschreibt die Erfahrungen eines Entwicklerteams und Teil 3 zielt auf die Auswirkungen für Applikationen. Andererseits führte je Projekt eine andere Organisationsstruktur zum Ziel. Daher enthalten die Erfahrungsberichte jeweils eine graphische Darstellung ihrer Organisationsstrukturen, die sich an der Webseite „DevOps Topologies[3]" orientiert.

Teil 1: Aus der Gesamtprojektsicht

Co-Autor Florian Zierer beschreibt seine Erfahrungen aus einem Projekt, in dem eine DevOps-Transition mit über hundert Entwicklern in vielen Aspekten realisiert werden konnte und in dem weiterhin an dem Optimierungsgedanken festgehalten wird. Es handelte sich um ein Projekt mit circa 40 Applikationen eines großen Web-Portals und dahinterliegenden Backend-Systemen. Es gab etwa zehn Entwicklerteams, die durch zwei sogenannte Enabler-Teams unterstützt wurden, die als Katalysator die Hinführung zu einer DevOps-Kultur ermöglichten.

Enabler-Teams als Hub und Adapter

Die Enabler-Teams bestanden aus Personen, die Entwicklererfahrungen hatten, aber auf Betriebsthemen spezialisiert waren. Ihre Aufgabe war es, ein Kommunikations-Hub zwischen den Entwicklerteams zu bilden und eine Adapterrolle zwischen Entwicklern und der eigentlichen Betriebsabteilung einzunehmen, die sich in einem entfernten Rechenzentrum befand. Realisiert wurde dies, indem ein Enabler-Team einerseits das Release- und Change-Management übernahm und andererseits die Prüfung und Filterung von Auffälligkeiten im Produktionsbetrieb im Fokus hatte (inklusive Weiterleitung an die betroffenen Entwicklerteams). Das andere Enabler-Team pflegte und erweiterte die technische Basis der Entwicklungs-, Integrations- und Abnahmeumgebungen. Dazu zählten die standardisierte Orchestrierung etlicher virtueller Server (via Ansible), die einfache Handhabung von CI-Tools (wie etwa Jenkins) und die Bereitstellung von Container-Plattformen (in Form von Docker-Swarm). In **(Abb. 1)** ist diese Art der Zusammenarbeit graphisch dargestellt.



Brückenschlagen zwischen Entwicklern und dem Betrieb. (Abb. 1)

Die Entwicklerteams waren autonome, agile Einheiten, die in der Regel aus fünf bis zehn Entwicklern bestanden und ihr agiles Vorgehensmodell selbst auswählen konnten, sodass es zu ihrer Teamgröße und ihren Anforderungen passte. Scrum wurde hier am meisten eingesetzt, wobei die zwei größten Teams (aus je 15 bis 20 Entwicklern bestehend) sich für Kanban oder Nexus entschieden. Im Projekt wurde Selbstständigkeit stark gefördert, damit die Teams ihren Deployment-Prozess formen und kontrollieren konnten, nämlich wie er am besten für ihr Produkt geeignet war. Dadurch konnte die Grundlage für die agilen Teams geschaffen werden, damit sie durch Selbstverantwortung und Selbstorganisation ein starkes Commitment für ihre Deliverables aufbauen konnten.

Die zuvor erwähnten Enabler-Teams spielten dabei entscheidend zu: In ihrer Rolle als Adapter zum Betrieb sorgten sie dafür, dass Produktionsprobleme nach allgemeinen oder bekannten Problemen vorgefiltert werden konnten. Erst wenn spezifisches, tieferes Wissen aus einer Applikation nötig war, leiteten sie die Probleme an die Teams der betroffenen Applikationen weiter und nahmen so ihre Rolle als Hub zwischen den Entwicklern wahr. Dadurch konnten die Entwicklerteams sich dedizierter mit dem Thema Betrieb befassen und weiterhin den Fokus auf ihr Wertschöpfungs-Commitment legen. Erfahrungen ergaben, dass der Betrieb etwa 10% der Probleme beheben konnte, aber Feedback von den Entwicklern für die restlichen 90% brauchte. Durch die Enabler-Teams als Puffer mit ihren Erfahrungswerten und Voranalysen mussten sich die Entwicklerteams nur mit 10 bis 20% der Produktionsprobleme befassen. Darüber hinaus sorgten die Enabler-Teams für ein homogenes Umfeld in einer heterogenen IT-Landschaft aus Legacy-Systemen und halbautomatischer Plattformen wie etwa Docker-Swarm. Dies war ebenfalls ein wichtiger Baustein, um den Entwicklerteams ein geordnetes Verständnis über den Aufbau der IT-Landschaft als Ganzes zu geben.

Beispiele, wie die DevOps-Kultur gedeihen konnte

Hier ein konkretes Beispiel des gelungenen Zusammenspiels im Rahmen einer DevOps-Transition: Ein großes Entwicklerteam stellte im Rahmen einer gut geplanten Umstellung innerhalb eines Jahres alle ihre statischen virtuellen Testserver auf dynamisch skalierbare Docker-Container in einem Docker-Swarm-Cluster um, d.h. für jeden Feature-Branch in der Git-Versionskontrolle wurde automatisch ein Container bereitgestellt. Auf dieser Grundlage konnte ein weiteres großes Team aufbauen und die Erfahrungen und das Wissen wiederverwenden. Dieses Team konnte einen Schritt weiter gehen und dadurch seine Architektur auf Microservices umstellen und eine Continuous-Deployment -Pipeline bis in die Staging-Umgebung realisieren. Ohne die Vorarbeiten des ersten Teams wäre die Umstellung extrem aufwendig gewesen. Dies war ein gutes Beispiel, wie Wissen skalieren konnte. Insbesondere sind diese beiden Teams dadurch einen großen Schritt unabhängiger von den Enabler-Teams

geworden und somit einem DevOps-Leitbild näher gerückt. Es gab auch Teams, die den Schritt in die Cloud erprobten. Es gab aber auch Teams, die auf Grund von technologischen Einschränkungen (z.B. durch vorgegebene monolithische Frameworks) nicht ohne weiteres auf Microservices und Container umstellen konnten. Diese Teams erhielten aber die Möglichkeit, eine klassische DevOps-Schiene zu verfolgen, indem sie etwa durch den Einsatz von Ansible oder Puppet ihre statischen virtuellen Maschinen selbst verwalten und somit dennoch ein Verständnis des Betriebs entwickeln konnten. Die Enabler-Teams ermöglichten hier die unbürokratische Bereitstellung von virtuellen Servern und das unkomplizierte Wiederherstellen nach einer destruktiven Änderung durch die Entwickler. Umrahmt wurde dieses Gesamtprojekt durch die Bereitstellung von Messinstrumenten fürs Monitoring (wie etwa SiteScope, Splunk, Prometheus), die es den Entwicklern ermöglichten, ihre Applikationen ganz aus ihrer Expertensichtweise zu beobachten. Natürlich klappte das alles nur durch den hingebungsvollen Einsatz von Entwicklern, die diese Tools bedienen und auf die wesentlichen Messpunkte fokussieren konnten.

Fazit aus dieser Transition

An dieser Organisationsform begeisterte vor allem die Dynamik des Wissenstransfers, durch die recht komplexe Technologien zwischen den Teams ausgetauscht werden konnten. Des Weiteren war es sehr wichtig, dass Entwickler durch die oben erwähnte Filterung von Bug-Tickets von einer Flut an Operationsthemen verschont blieben und sich weiterhin auf die Applikation konzentrieren konnten. Abgerundet wurde dies durch den leichten Zugang zu Messinstrumenten und die Möglichkeit, die Sensoren bei der Überwachung der Applikationen selbst zu konfigurieren. Denn mit jedem neuen Sensor wuchs das Verständnis über die Applikation und das Wissen wie sie sich im Normal- und im Problemfall verhalten sollte. All dies schärfte wiederum den Blick auf das Thema Softwarebetrieb, der durch viele Synergieeffekte mit anderen Entwicklerteams getragen wurde. Ohne eine gelebte offene Organisation wäre dies nicht möglich gewesen.

Teil 2: Aus der Sicht eines Entwicklerteams

Co-Autor Christopher Hensel beschreibt nachfolgend seine Erfahrungen aus einem abteilungsübergreifenden, agilen Gesamtprojekt zur DevOps-Transformation und schildert seine konkrete Sicht als Mitglied eines Teams, dessen Schwerpunkte Authentifizierung und Autorisierung waren.

Entwicklungsprozess

Das Team arbeitete nach der agilen Methode Kanban, bei der jede durch einen Entwickler begonnene User-Story mithilfe eines zweiten Entwicklers im Refinement besprochen wurde, um für die Umsetzung der Anforderung notwendige Aufgaben zu erfassen. Die Entwicklung selbst erfolgte nach dem Modell des Trunk-Based-Developments. Damit der Trunk immer releasefähig bleiben konnte, wurden kritische Änderungen durch Feature-Toggles geschützt, welche umgebungsspezifisch gesteuert wurden. Standardmäßig waren diese auf der Integrationsumgebung aktiviert, damit die Änderungen im Rahmen der Continuous-Integration frühzeitig ausgerollt und getestet werden konnten. Für die Continuous-Integration war ein Jenkins im Einsatz, den das Team selbst verwaltete. Bei einer neuen Änderung auf dem trunk-Branch wurde automatisch der Continuous-Integration-Prozess als Job auf dem Jenkins gestartet. Dabei wurden das gesamte Projekt gebaut, alle Unit- und Integrationstests ausgeführt und statische Code-Analysen auf einem Sonarserver durchgeführt. Nachdem all diese Schritte erfolgreich waren, konnte das aktuelle Artefakt auf der Integrationsumgebung ausgerollt werden. Am Ende des Entwicklungsprozesses nahm der Product-Owner (PO) die User-Story auf der Integrationsumgebung ab, um sie schließlich für das Deployment auf der Abnahmeumgebung freizugeben.

Continuous-Delivery-Prozess

Die DevOps-Transformation betraf das Team erst ab dem Abnahmeschritt maßgeblich, nämlich im Rahmen der Einführung eines Continuous-Delivery-Prozesses mit Hilfe moderner Technologien. Für die Bereitstellung eines Releases auf einer Umgebung mussten die Entwickler vor der Transformation für alle Module, die im Rahmen einer User-Story verändert wurden, Release-Tasks in JIRA erstellen. Insbesondere wurden darin etwaige manuelle Aufgaben vermerkt, die im Rahmen des Deployments auf den Umgebungen nötig waren. Bei der Abnahmeumgebung wurde der Task in die Spalte Abnahme eingeplant verschoben, um dem zuständigen Betriebsverantwortlichen mitzuteilen, dass das Deployment durchzuführen war. Erst nachdem der Betriebsverantwortliche den Task auf die Spalte „Abnahme deployed“ verschoben hatte, wurde der PO benachrichtigt, der die Abnahme in Abstimmung mit dem Fachbereich auf dieser Abnahmeumgebung durchführte. Nach erfolgreicher Abnahme wurden die Schritte für die Demonstrations- und Produktionsumgebung wiederholt.

Die Deployments in Produktion erfolgten dabei mehrmals am Tag. Obwohl die Zusammenarbeit zwischen dem Entwicklerteam und dem zuständigen Betriebsverantwortlichen sehr gut war, mussten manuelle Schritte durch den Betriebsverantwortlichen durchgeführt werden. Hierdurch war das Entwicklerteam auf den Betriebsverantwortlichen angewiesen, um aktuelle oder dringende Änderungen in Produktion auszurollen. Um dieser Einschränkung zu begegnen, erhielten die Entwicklerteams im Rahmen der DevOps-Transformation eine eigene Instanz des Deployment-Werkzeugs, mit dessen Hilfe sie die Artefakte auf allen Umgebungen ausrollen konnten. Verwendet wurde hierfür ein hauseigenes, kommandozeilenbasiertes Deployment-Werkzeug. Damit das Deployment auf allen Umgebungen selbstständig durch das Entwicklerteam erfolgen

konnte, war die Verwaltung der Deployment-Konfigurationen für alle Umgebungen durch das Team notwendig. Unter diesen Voraussetzungen konnten weitere Jobs auf dem bereits bestehenden Jenkins erstellt werden, die die Ausführung der Deployments auf den einzelnen Umgebungen mithilfe des Deployment-Werkzeugs durchführten. Im späteren Verlauf wurden die Jobs mit Shell-Skripten erweitert, um komplexere Deployments wie das Deployen mehrerer Instanzen desselben Artefakts zu ermöglichen.

Deployment-Pipeline

Darauf aufbauend wurde eine Deployment-Pipeline auf Basis einer Jenkins-Pipeline entwickelt und die Shell-Skripte wurden durch Groovy-Skripte ersetzt, da sich diese besser testen ließen und vorhandenes Wissen über Java im Team genutzt werden konnte. Dies reduzierte den Deployment-Prozess auf ein Minimum und eröffnete den Weg zu Continuous-Delivery. Manuelle Schritte waren lediglich nötig für:

1. die Freigabe des Deployment-Prozesses auf die einzelnen Umgebungen innerhalb der Jenkins-Pipeline (insbesondere durch den PO auf die Abnahmeumgebung),
2. spezifische manuelle Schritte zur User-Story, wie etwa die Anpassung der Deployment-Konfiguration oder das Aktivieren von Feature-Toggles.

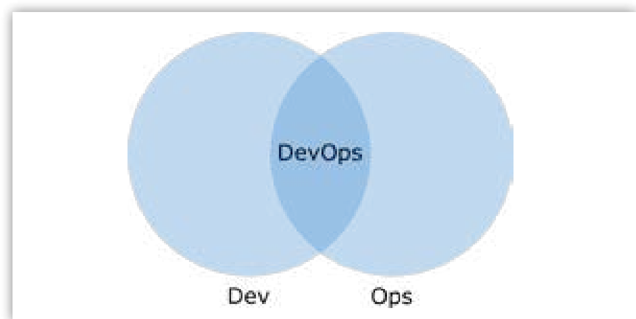
Diese spezifischen Tasks wurden weiterhin in JIRA in Release-Tasks festgehalten und in der neuen Deployment-Pipeline angezeigt. Dadurch entfiel ein Wechsel in eine zweite Umgebung. Des Weiteren wurden die Release-Tasks für die Module nun durch die Pipeline automatisch angelegt und auf dem Release-Board verschoben. Anhand der Release-Tasks konnte verfolgt werden, welche Änderungen noch zu deployen waren und auf welcher Umgebung sie aktuell umgesetzt waren. Es gab auch die Funktion, das Deployment von bestimmten Modulen zu blockieren, indem der Release-Task des Moduls um ein Label „Deployment blocked" ergänzt wurde. Es war dann eine zusätzliche, explizite Zustimmung während des Deployments des Moduls notwendig.

Die meisten Verbesserungen wurden im Continuous-Delivery-Prozess durch eine Change-Detection erzielt. Diese erkannte automatisch nach dem Commit die von den Code-Änderungen betroffenen Module und annotierte die entsprechende User-Story mit dem Modul. Dadurch waren an der User-Story alle Module in der entsprechenden Version protokolliert, in der sie herausgebracht werden mussten, um die in der User-Story umgesetzte Funktionalität vollständig zu deployen. Wurden die Änderungen schließlich auf die Produktionsumgebung ausgerollt, so wurde automatisch ein Eintrag im unternehmensweiten Change-Kalender generiert. In Summe haben diese Maßnahmen den Wertschöpfungsprozess, der von der Einplanung bis hin zum Release in Produktion durchlaufen werden musste, um die User-Story herum zentriert.

Weitere technische Verbesserungen

Verbesserungen bezogen sich auch auf die Sicherheit des Produkts und die Vereinfachung der administrativen Tätigkeiten. Eine Pipeline prüfte die Abhängigkeiten aller Module auf ihre Aktualität und erstellte automatisch eine User-Story für Major-Release-Updates oder eine weitere, wenn mehr als zehn Minor-Updates für ein Modul vorzunehmen waren. Andere Pipelines wiederum erleichterten das Ausrollen einer neuen Version des Java-Runtime-Environments oder des Servlet-Containers auf allen Systemen eines Produkts. Ebenfalls vereinfacht wurde der Prozess zur Verwaltung von Datenbankänderungen. Bereits vor der DevOps-Transformation wurden alle Schemaänderungen eines Produkts durch Flyway verwaltet. Die Änderungen von Daten einer bestimmten Umgebung mussten jedoch über einen Roboter verwaltet werden, deren Einstellung und Freigabe zuvor nur durch die Betriebsverantwortlichen erfolgen konnte. Im Rahmen der DevOps-Transformation wurde der Selfservice für Datenbankroboter allen Entwicklern zugänglich gemacht, wodurch die Reaktionszeit für dringende Datenbankänderungen verbessert werden konnte.

Organisatorische Änderungen



Entwickler und der Betrieb arbeiten direkt zusammen. (Abb. 2)

Neben den bereits erläuterten technischen Herausforderungen und Neuerungen mussten organisatorische Aufgaben gelöst und Prozesse angepasst werden, beginnend bei der Organisation der Entwicklerteams. Manche Teams hatten dedizierte DevOps-Personen und in anderen Teams übernahmen alle Entwickler gleichermaßen die DevOps-Rolle. Unabhängig davon wurden jedoch alle Teams durch die Übernahme der DevOps-Rolle mit neuen Aufgaben betraut, um den Betrieb ihrer Anwendungen sicherzustellen. Dazu zählte die Betreuung von Wartungsfenstern, die zuvor durch dedizierte Betriebsverantwortliche erfolgte. Zusätzlich fiel die Erstellung und Verwaltung der Anwendungskonfiguration mittels Configuration-Management-Database (CMDB) in den Aufgabenbereich der Entwicklerteams, durch die definiert wurde, über welche Netze und Zugangsadressen die Anwendung erreichbar sein soll und welcher Zugriffsschutz für die Anwendung benötigt wird. In diesem Zusammenhang mussten Rollen und Rechte erstellt oder angepasst werden, für Themen wie die Verwaltung von Anwendungskonfigurationen, den Zugriff auf Datenbanken oder auch die Verwaltung von

Endbenutzern und technischen Benutzern der Anwendungen. Für die Einführung der DevOps-Kultur mussten die Zuständigkeiten neu geklärt werden. Zu Beginn bestand eine große Unsicherheit bezüglich der Aufgabenverteilung zwischen den Entwicklern und dem Betrieb aufgrund der unklaren Definition. Bedingt durch diese Unsicherheit wurde die Zusammenarbeit zunächst schwieriger und tendierte Richtung No-Ops. Erst durch einen intensiveren Austausch zwischen den Entwicklern und dem Betrieb sowie der Klärung der Verantwortlichkeiten konnte die Zusammenarbeit wieder gestärkt werden. Seitens des Betriebs gab es viel Unterstützung, damit die Entwickler das Wissen zum Anwendungsbetrieb aufbauen konnten. Daraus resultierte die Idee zur Community-of-Practice, die als Plattform für den Wissenstransfer zwischen Betrieb und den unterschiedlichen Entwicklerteams notwendig war. In dieser trafen sich einmal pro Woche ein DevOps-Zuständiger aus jedem Team und zwei Vertreter aus dem Betrieb, um aktuelle Probleme in den Entwicklerteams und neue Anregungen aus dem Betrieb zu besprechen. Im Anschluss trugen die Teilnehmer die Ergebnisse zurück in ihre Teams. Durch dieses Vorgehen konnten Synergieeffekte zwischen den Entwicklerteams und dem Wissen der ehemaligen Betriebsverantwortlichen genutzt werden. In **(Abb. 2)** ist diese Art der Zusammenarbeit graphisch dargestellt.

Trotz der Anpassungen in der Organisation und der Ausweitung der Rechte für Entwickler blieben Einschränkungen für DevOps. Virtuelle Maschinen mussten weiterhin per Ticket beim Betrieb beantragt werden. Ebenso war es mit Firewall-Freischaltungen und Proxykonfigurationen, die auf schriftlichem Wege beantragt werden mussten. Eine weitere Einschränkung bestand bei der Verwaltung der Anwendungs-Server. Hier beschränkten sich die Rechte auf die Anwendungskonfiguration und die Konfiguration des Servlet-Containers. Den genannten Einschränkungen könnte mit der Verwendung einer Cloud-Plattform begegnet werden, die gleichzeitig ein zukünftiges Ziel für die Weiterentwicklung im Rahmen der DevOps-Transformation darstellen könnte.

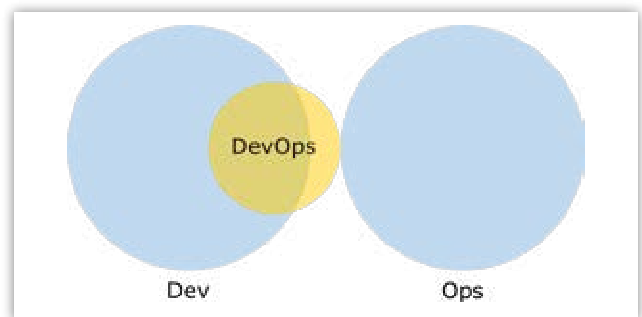
Fazit aus dieser Transition

Zusammenfassend lässt sich sagen, dass es essenziell ist, die neue DevOps-Kultur in den Teams sowie in der Organisation zu verankern. Hierzu wurde der Transformationsprozess anhand eines Kanban-Boards an einem prominenten Platz wie der Kaffeeküche transparent gemacht. Nichtsdestotrotz ist der Mehrwert, der durch die Einführung der DevOps-Kultur im Unternehmen entstanden ist, nur schwer messbar. Der Umstand, dass vor der Einführung keine Daten erhoben und zu Beginn des Prozesses keine messbaren Ziele definiert wurden, erschweren einen objektiven Vergleich. Dennoch wurde im Laufe der Transition das große Potential zur Automatisierung des Deployment-Prozesses sichtbar. Es gab zwar weiterhin manuelle Aufgaben, doch diese reduzierten sich auf wichtige Spezialpunkte. Die Aufgaben zur Verwaltung der User-Stories und die Abbildung des Deployment-Prozesses innerhalb von JIRA wurden automatisch durch die Deployment-Pipeline übernommen. Außerdem wurden durch die Automatisierung neue Kapazitäten innerhalb des Betriebs geschaffen, welche sich nun auf neue, übergreifende Themen wie die Zero-Downtime-Wartung und die Verbesserung des Monitorings sowie der Analytics-Plattform konzentrieren konnten.

Teil 3: Aus der Sicht einer Applikation

Co-Autor Matthias Engelke beschreibt hier ein drittes Projekt, bei dem eine hochmoderne IT-Landschaft in der Cloud mitgestaltet wurde. Die Beschreibung zielt auf die Sichtweise der Applikation und ihrer umliegenden Rahmenbedingungen, die auch das Thema DevSecOps im Blick hat (siehe auch JAVAPRO 03-2018).

Containertechnologie



Containerbasierte Entwicklung und ein cloudbasierter Betrieb. (Abb. 3)

In diesem Projekt aus der Automobilindustrie ging es zum einen um den Aufbau einer hochverfügbaren und weltweit verteilten Container-Registry, die die Bereitstellung von Applikationen als Container-Images ermöglichte. Zum anderen wurde der Entwicklungsprozess solcher Applikationen mittels Continuous Integration und Continuous Delivery unterstützt und verbessert. Die Container-Technologie und der durch Docker gewachsene Trend dahin lassen sich seit mehreren Jahren auch in der Automobilindustrie beobachten, die diese in unterschiedlichen Themenbereichen sowohl in der Entwicklung als auch in der Produktion einsetzt. Als der Co-Autor das Projekt zum ersten Mal kennenlernte, überraschte es ihn, dass wenige bis keine Anzeichen von DevOps in den Entwicklerteams zu erkennen waren. Diese Teams arbeiteten an verschiedenen Applikationen für eine gemeinsame Plattform, doch verwendeten sie jeweils unterschiedliche Prozesse und Methoden, um ihre Apps zu bauen, zu testen und auszurollen, obwohl Docker als Technologie eingesetzt wurde. Denn Docker hilft insbesondere bei der Umsetzung des DevOps-Gedankens und wird daher von vielen als der DevOps-Enabler bezeichnet.

Cloud-Plattform

Für die Bereitstellung der skalierbaren, hochverfügbaren und weltweit verteilten Container-Registry, die für die Auslieferung aller Applikationen in Form von Docker-Images produktiv

eingesetzt werden sollte, kam eigentlich nur eine Cloud-Plattform in Betracht. Das lag unter anderem daran, dass AWS schon beim Kunden erfolgreich eingesetzt wurde, aber auch daran, dass der traditionelle Ansatz eines Datenzentrums bei der benötigten Skalierbarkeit aufgrund der erfahrungsgemäß hohen Rüstzeiten keine Alternative war. Diese Skalierbarkeit kam insbesondere durch den Einsatz des Elastic-Container-Service (ECS) zum Vorschein. Hierbei handelt es sich um Amazons Container-Orchestrierungs-Service, der das Ausführen, Beenden und Verwalten von Docker-Containern innerhalb eines Clusters vereinfacht und durch API-Aufrufe steuerbar macht.

Für die Umsetzung dieses Projektes ging man direkt mit dem DevOps-Gedanken an den Start, um mit Hilfe von Infrastructure-as-Code und viel Automatisierung die bisherigen, allzu bekannten Probleme zu verhindern. Der Leitgedanke dabei war, alle benötigten Cloud-Ressourcen mit Hilfe von Terraform, Continuous Integration und Continuous Delivery automatisiert zu entwickeln, zu testen und auszurollen. In **(Abb. 3)** ist diese Art der Arbeitsweise grafisch dargestellt. Anstelle eine Vielzahl von virtuellen Maschinen für das Deployment dieses Projektes zur Verfügung zu stellen, wurde AWS und Infrastructure-as-Code eingesetzt, um diese selbst zu provisionieren. Dadurch hatte das Entwicklerteam sowohl die Entwicklung als auch den IT-Betrieb der Plattform in der eigenen Hand. Dies resultierte insbesondere aus einer deutlichen Beschleunigung der Produktivität und der Auslieferung neuer Releases dieser Plattform. Beispielsweise war das Entwicklerteam durch die Erweiterung des bestehenden Codes direkt in der Lage, Änderungen wie Port-Freischaltungen im Load-Balancer oder das Skalierungsverhalten der Applikationen einzustellen und dadurch automatisiert ein Deployment auf die Entwicklungs-, Test- und letztendlich Produktivumgebung zu bewirken.

DevSecOps

Beim Thema DevOps sollte der Security-Aspekt ebenso stark berücksichtigt werden wie die Umsetzung der Container-Registry und die Implementierung von Continuous Integration und Delivery-Pipelines für die Verbesserung der Entwicklungsprozesse aller Applikationen. Das heißt, man zielte auf die logisch zu Ende gedachte Idee von DevOps: DevSecOps. Konkret wurden Sicherheitsaspekte der Applikationen in den Entwicklungsprozess integriert, wodurch den Entwicklerteams direktes Feedback gegeben wurde. Im Zusammenhang mit Docker ging es nicht mehr nur um das Verhindern von Sicherheitsschwachstellen innerhalb des Sourcecodes, das mit unterschiedlichen Tools wie beispielsweise dem OWASP-Dependency-Check für Node.js ermöglicht werden kann. Es sollten Sicherheitsschwachstellen innerhalb des Docker-Containers schon während des Entwicklungsprozesses erkannt und beseitigt werden. Insbesondere letzteres wird in den meisten Fällen vernachlässigt. Laut der Studie „A Study of Security Vulnerabilities on Docker Hub" aus dem Jahre 2017 wurden in über 350.000 Images im Durchschnitt 180 Sicherheitsschwachstellen gefunden[4]. Dies lag vor allem daran, dass die Images die Sicherheitsschwachstellen ihrer Parent-Images erbten und diese auch sehr selten aktualisiert wurden. Das Java 8 Base-Image weist zum Beispiel über 80 High-Level Sicherheitsschwachstellen der National-Vulnerability-Database auf. Um nun diese Sicherheitsschwachstellen frühzeitig zu erkennen und verhindern zu können, sollte zusätzlich zur Container-Registry auch CoreOS-Clair als Image-Security-Scanner zur Verfügung gestellt und in den CI/CD-Prozess integriert werden.

Fazit aus dieser Transition

Die Vorteile, die der Einsatz einer Cloud-Plattform in Kombination mit der Container-Technologie mit sich bringt, sind schon lange kein Geheimnis mehr. Allerdings muss hierbei beachtet werden, dass sich die bisher bekannten Prozesse an diesem neuen Ansatz anpassen müssen, um diese wirklich leben zu können. Auch die neuen Herausforderungen dürfen hier nicht außer Acht gelassen werden. Insbesondere dadurch rücken Security-Aspekte deutlich mehr in den Vordergrund und müssen in den Continuous-Delivery-Prozess integriert werden.

Schlusspfiff

Aus den drei Erfahrungsberichten erkennt man, dass das Thema DevOps je nach Situation einen anderen Schwerpunkt setzt. Ist man konfrontiert mit einem Transitionsprozess oder liegt eine heterogene Systemlandschaft vor. So sind Organisation, Kommunikation und die Form des Wissensaustausches das Hauptthema, gefolgt von technologischen Hilfswerkzeugen. Gilt andererseits eine derart hohe Anforderung an die Applikationen (z.B. Skalierbarkeit), die nur mittels Cloud-Lösungen wirtschaftlich realisiert werden kann, so rücken plötzlich die technologischen Werkzeuge in den Vordergrund (z.B. Terraform, ECS oder Kubernetes), gepaart mit dem Wissen, wie diese zu nutzen sind. Man baut hier bereits auf einem Konsens auf, wie DevOps zu funktionieren hat. Ein weiterer Schlusspunkt ist, dass oft erst der DevOps-Transitionsprozess offenlegt, welche manuellen, repetitiven Schritte es überhaupt gibt. Dass man anschließend die Verbesserungen und den alten Stand dokumentiert, ist wichtig, um später ein Bewusstsein für die Veränderungen zu erhalten und somit den Mehrwert der Transition im Projekt zu verankern. Außerdem darf man bei diesem Prozess nicht übersehen, dass durch die Verschiebung der Verantwortlichkeiten das Thema Security noch strenger in den Mittelpunkt der Entwicklung rückt und dass man bei DevOps eigentlich DevSecOps im Blick haben sollte.

Quellen:

1 https://java-pro.de/skills-tools-mindset/
2 Gesetz von Convey: https://bit.ly/2UU67XP
3 https://web.devopstopologies.com
4 Shu, Rui & Gu, Xiaohui & Enck, William. (2017). A Study of Security Vulnerabilities on Docker Hub
5 Bilder von Florian Zierer, PENTASYS AG

#JAVAPRO #IoT #Messaging #MQTT #Java

IoT-Messaging mit MQTT 5 und Java

Mit der HiveMQ-MQTT-Client-Library kann das de-facto Standardprotokoll für IoT in der neuesten Version genutzt werden.

MQTT ist mittlerweile das populärste IoT-Protokoll (Google-Trends-Analyse[1]) für die Kommunikation zwischen Geräten und Applikationen über das Internet. Einige der verschiedenen Anwendungsfälle für das schlanke und leichtgewichtige MQTT sind unter anderem: Industrie 4.0, Connected-Cars, Logistik-Mobile-Apps und leichtgewichtiges Messaging zwischen Microservices. Speziell im Java-Umfeld wird MQTT häufig genutzt, um klassische Enterprise-Anwendungen für das Internet der Dinge fit zu machen. Zu den Gründen hierfür zählen die herausragende Skalierbarkeit von MQTT, das eine Skalierung von einem bis hin zu mehreren Millionen von Geräten und mehreren hunderttausend Nachrichten pro Sekunde erlaubt, oder auch die einfache Möglichkeit MQTT in bestehende Enterprise-Applikationslandschaften zu integrieren. Dieser Artikel beleuchtet die Verwendung des HiveMQ-MQTT-Client, einer Open-Source-Java-Bibliothek mit voller MQTT-5-Unterstützung, die für den Einsatz in professionellen Projekten konzipiert wurde.

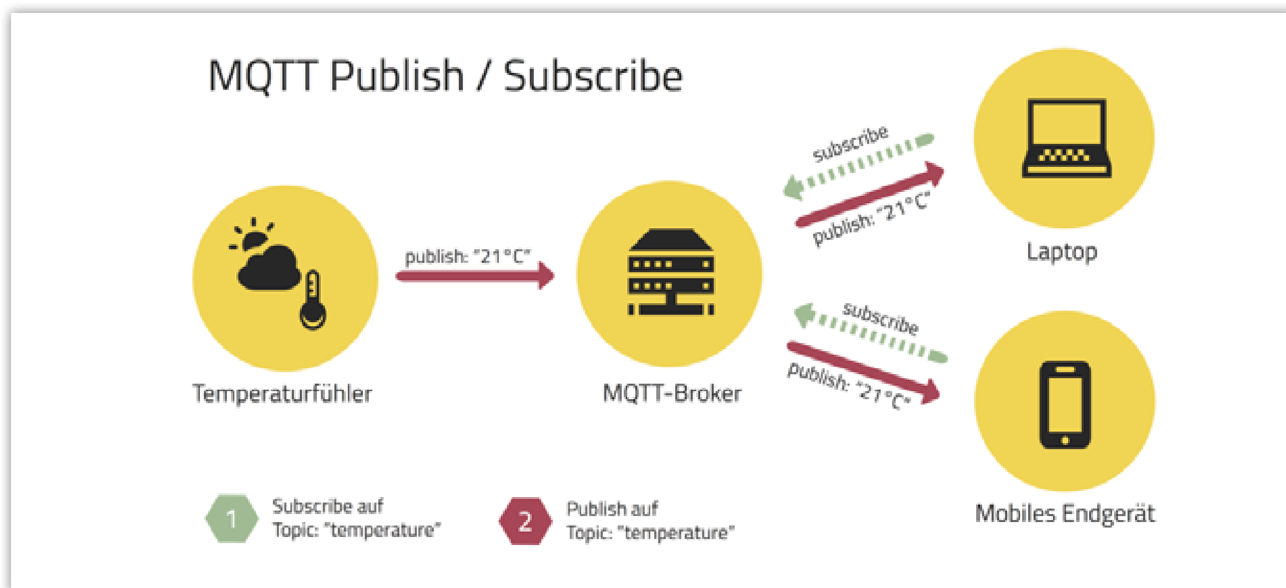
MQTT – Ökosystem und kurze Einleitung

Um die Kommunikation der Clients, die durch das Publish/Subscribe-Paradigma sichergestellt, bidirektionalisiert und entkoppelt wurde, in einem MQTT-Deployment zu ermöglichen, wird ein MQTT-Broker benötigt. Je nach konkretem Anwendungsfall rangieren die möglichen MQTT-Broker-Implementierungen von freien Open-Source-Projekten wie Eclipse-Mosquitto[2] bis hin zu kommerziellen Produkten wie den Enterprise-MQTT-Broker HiveMQ[3], der speziell für professionelle Cloud-Deployments entwickelt wurde und sich für die Vernetzung von wenigen Geräten bis hin zu mehreren Millionen IoT-Devices eignet.

Autor:

Florian Raschbichler leitet das Support-Team bei der dc-square GmbH, dem Softwarehersteller des Enterprise MQTT Brokers HiveMQ, die zu den führenden Experten weltweit im Bereich MQTT zählt.
Er unterstützt bei Problemlösungen rund um die Themen MQTT und HiveMQ - daher kennt Florian die Herausforderungen aus erster Hand.
Als MQTT Experte teilt er seine Erfahrung aus mehr als 5 Jahren IoT in Produktion und schreibt Artikel, hält Vorträge oder Workshops rund um das de-facto Standard-Protokoll des Internet-of-Things.

www.dc-square.de
www.linkedin.com/in/florian-raschbichler-46310413b
Twitter: fraschbi



Publish/Subscribe-Architektur von MQTT (Abb. 1)

MQTT-Clients können neben leichtgewichtigen IoT-Geräten mit beschränkten Ressourcen auch Desktop- bzw. Mobile-Applikationen oder Backend-Systeme sein, die unter Verwendung verschiedenster Programmiersprachen mit Hilfe des Standardprotokolls MQTT miteinander kommunizieren können.

Zum jetzigen Zeitpunkt ist MQTT 3.1.1 die am häufigsten verwendete Version des Protokolls. Im Jahr 2018 wurde mit MQTT 5 der offizielle Nachfolger verabschiedet. MQTT 5 bietet eine Reihe an Verbesserungen und neuen Features. Es wird unter anderem durch die Rücksichtnahme auf das Feedback derjenigen Nutzer bereichert, die MQTT 3.1.1 bereits produktiv einsetzen. Eine Liste aller Neuerungen ist auf der HiveMQ-Webseite[4] einsehbar.

HiveMQ-MQTT-Client-Library

Obwohl Java zu den populärsten und am meisten genutzten Programmiersprachen zählt, gab es bisher keine wirklichen Alternativen zu der Java-Bibliothek Eclipse-Paho. Da Eclipse-Paho jedoch für hochskalierbare Anwendungen schlichtweg nicht performant genug und das Threading-Model für die Integrationen in Fremdsysteme nicht ideal ist, kein Backpressure-Handling möglich ist und die APIs aus heutiger Sicht etwas altbacken daherkommen, haben sich dc-square und BMW Car-IT zusammengetan und die Java-basierte, Open-Source-Bibliothek HiveMQ-MQTT-Client-Library entwickelt. Die Apache-2-lizensierte Bibliothek wurde speziell für MQTT 5 entwickelt (ist jedoch auch mit MQTT 3.1.1 kompatibel) und hat mehrere Projektziele:

- Eine funktional zuverlässige Standardbibliothek für alle MQTT-Projekte mit Java umzusetzen
- 100% MQTT-5-Support, inklusive aller optionalen Features
- Implizites und explizites Backpressure-Handling
- Niedrigster Speicherverbrauch
- Extrem hoher Nachrichtendurchsatz, sowohl sendend als auch empfangend
- Leichte Integrierbarkeit in bestehende Java-Frameworks und Applikationen
- Verschiedene API-Layer: Blocking, Async und RxJava

Durch ihre Flexibilität ist die Bibliothek von Embedded-Geräten, die über eine eigene JVM verfügen, über Java-basierte Anwendungen bis hin zu hoch performanten Backend-Systemen (Streaming und Big-Data) geeignet. Das flexible und für Entwickler einfache Threading-Model der HiveMQ-MQTT-Client-Library stellt eine unkomplizierte Integrierbarkeit in bestehende Applikationen sicher. Der Fokus der Bibliothek liegt darauf, sinnvolle Default-Einstellungen für Standardanwendungsfälle mitzubringen, sowie den notwendigen Code für Entwickler standardmäßig klein und robust zu halten. Zusätzlich stellt die HiveMQ-MQTT-Client-Library durch ihre Flexibilität sicher, dass selbst tiefe und unübliche Integrationen ohne Verstöße gegen das MQTT-Protokoll möglich sind.

Features

Die HiveMQ-MQTT-Client-Library umfasst eine vollständige Umsetzung aller MQTT-Funktionalitäten laut Spezifikation[5]. Zusätzlich wurde eine Reihe von Features implementiert, die das Leben für Entwickler und die Integration der Bibliothek in bestehende Systeme leichter machen sollen, unter anderem:

- Voller MQTT-3.1.1- und MQTT-5-Support mit allen optionalen Features
- Sichere und verschlüsselte Kommunikation mit TLS (u.a. TLS 1.3)
- MQTT-over-Websockets
- Automatisches Reconnect-Handling
- Offline-Buffering von Nachrichten
- Backpressure-Support

- Optionale Integration von Dropwizard-Metrics
- Integration mit SLF4J für standardisiertes Logging
- Pluggable-Thread-Pools
- HTTP-CONNECT-Support

Beispielcode

Je nach spezifischem Anwendungsfall und persönlicher Präferenz des Entwicklers kann eine der drei verschiedenen APIs genutzt werden, die von der HiveMQ-MQTT-Client-Library angeboten werden.
Eine Blocking-API, eine Async-API so wie eine RxJava-API. Dem Nutzer steht sogar zu jedem Zeitpunkt die Möglichkeit offen, zwischen diesen API-Styles zu wechseln. So lässt sich beispielsweise zur Async-API wechseln, um eine Methode aufzurufen, während prinzipiell die Blocking-API verwendet wird.

Blocking-API

Die Blocking-API blockt bei allen Remote-Operationen den aktuell ausführenden Thread und ist durch ihren komprimierten, einfachen Stil ideal für Proof-of-Concepts geeignet. Außerdem eignet sie sich für Anwendungsfälle, bei denen Performance und Concurrency keine Rolle spielen. Im folgenden Beispiel wird gezeigt wie man mit Hilfe der Blocking-API einen MQTT-Client initialisiert, sich zu einem remote-Broker verbindet, eine Nachricht published und die Verbindung wieder trennt.

(Listing 1) – Blocking-MQTT-Client

```
final Mqtt3BlockingClient client = MqttClient.builder()
        .identifier("testclient")
        .serverHost("broker.hivemq.com")
        .useMqttVersion3()
        .buildBlocking();

client.connect();
client.publishWith().topic("test/topic").qos(MqttQos.AT_LEAST_
ONCE).payload("1".getBytes()).send();
client.disconnect();
```

Async-API

Die Async, also asynchrone API, ist für den Großteil aller Use-Cases geeignet, insbesondere wenn Nachrichten asynchron gesendet und empfangen werden sollen oder man mit eigenen Thread-Pools arbeiten möchte. Die meisten Operationen arbeiten mit CompletableFuture, die mit Java 8 neu zum JDK dazu gekommen sind, wodurch sich die Verkettung von asynchronen Operationen miteinander sehr feingranular kontrollieren lässt.

Im folgenden Codebeispiel findet sich ein Blocking-Client zu einem MQTT-Broker und anschließend wird durch den Aufruf von toAsync()der API-Style dieses Clients zur Async-API geändert. Ebenso wäre es möglich, direkt einen AsyncClient zu initiieren und sich mit diesem zu verbinden. Der Methode callback() kann jegliche konkrete Callback-Implementierung übergeben werden, die dann jeweils aufgerufen wird, wenn der Client auf dem spezifizierten Topic eine Nachricht erhält.

(Listing 2) - Async-MQTT-Client

```
final Mqtt3BlockingClient client = MqttClient.builder()
        .identifier("testclient")
        .serverHost("broker.hivemq.com")
        .useMqttVersion3()
        .buildBlocking();

client.connect();

client.toAsync().subscribeWith()
        .topicFilter("test/topic")
        .qos(MqttQos.AT_LEAST_ONCE)
        .callback(System.out::println)
        .send();
```

RxJava-API

Neben allen MQTT-5-Funktionalitäten unterstützt die HiveMQ-MQTT-Client-Library auch RxJava 2 bzw. Reactive-Streams. Dies ist besonders deshalb interessant, weil somit auch komplexe reaktive Anwendungen mit der HiveMQ-MQTT-Client-Library realisiert werden können. Beispielsweise erlaubt die reaktive API die Implementierung eines anwendungsspezifischen Backpressure-Handlings. Außerdem bietet die reaktive API beste Performance. Die Verwendung einer reaktiven API in Java ist naturgemäß sehr umfangreich, bietet aber auch die größte Flexibilität. Auf der Projekt-Webseite[6] findet sich eine ausführliche Dokumentation zur Benutzung mit RxJava.

Fazit:

Mit dem HiveMQ-MQTT-Client wurde von BMW Car-IT und dc-square eine Library entwickelt, die als MQTT-Standardbibliothek für alle Java-Projekte eingesetzt werden kann. Die Bibliothek ist neben dem Enterprise-MQTT-Broker-HiveMQ mit allen gängigen MQTT-Brokern kompatibel und trotz des großen Feature-Sets sehr schlank. Die Bibliothek ist für neue Projekte genauso geeignet wie für die Ablösung von Eclipse-Paho in bestehenden Applikationen.

Quellen:

1 http://bit.ly/2018-mqtt-popularity
2 https://mosquitto.org
3 https://bit.ly/2K6ft1X
4 http://www.mqtt5.com
5 https://bit.ly/2G1rK3I
6 https://bit.ly/2YS7kBg

#JAVAPRO #Agile #Coaching

Das Agile-Coaching-Dojo

Exzellenz in der agilen Praxis kann nicht allein aus dem Lehrbuch kommen, sondern ist immer auch eine Frage der Weitergabe und Modellierung von Erfahrungswerten. Um einen solchen Prozess innerhalb des eigenen Unternehmens einzuführen und zu systematisieren, ist das Veranstaltungsformat Agile-Coaching-Dojo besonders geeignet. In diesem Artikel erfahren Sie mehr über seine Ziele und Philosophie sowie die konkrete Umsetzung einer Dojo-Veranstaltung.

Autor:

Christian Sandmann arbeitet seit 2011 bei der PENTASYS AG. Seine Schwerpunkte liegen in den Bereichen Agile Coaching, Requirements Engineering und Projektmanagement. Ihn interessiert besonders, wie es gelingt, bestehende Kunden-Strukturen und neues agiles Denken miteinander zu verbinden, zu einem Beitrag, der den Kunden der PENTASYS AG weiterhilft.

Im Alltag agiler Projekte treten immer wieder ungewohnte Situationen auf, die für die Beteiligten eine Herausforderung darstellen. Nehmen wir das folgende Beispiel: Während eine Kollegin als Agile-Coach bei einem Kunden vor Ort ist, bittet sie ein Product-Owner um Unterstützung. Dieser ist momentan dabei, sein Backlog zu füllen. Allerdings liegen die Interessen zweier Stakeholder vor, die sich nur schwer miteinander vereinbaren lassen. Die Anforderungen scheinen sich gegenseitig auszuschließen. Die Kollegin steht deshalb vor der Frage: Welche Unterstützung könnte dem Product-Owner bei seiner Aufgabe helfen? Welcher Beitrag wäre angemessen, um den Zielkonflikt

aufzulösen? Gibt es möglicherweise Erfahrungswerte aus ähnlichen Projekten, die eine Orientierung ermöglichen?

Dieses allgemein gehaltene Beispiel verdeutlicht, dass die agile Praxis Herausforderungen birgt, die zwar in ähnlicher Form immer wieder anzutreffen sind, aber in den Modellen agiler Methoden keine explizite Berücksichtigung finden. Dies hat natürlich den Grund, dass Scrum & Co. von unausgesprochenen Voraussetzungen ausgehen müssen, um allgemein aussagekräftig zu sein – im erwähnten Beispiel wäre das Einvernehmen aller beteiligten Stakeholder eine solche. Für Agile-Berater und Projektbeteiligte braucht es daher ergänzende Maßnahmen, um die Leerstellen aus dem Scrum-Lehrbuch mit bewährtem Praxiswissen zu füllen. Deshalb wurde das Agile-Coaching-Dojo entwickelt mit dem Anspruch, diese Lücke zu füllen.

Ein Trainingsraum für die persönliche und fachliche Weiterentwicklung

Das Agile-Coaching-Dojo folgt zunächst der Philosophie einer Community-of-Practice. Die Basis der Weiterentwicklung ist das gemeinsame und gegenseitige Lehren und Lernen – aus der Praxis heraus in die Praxis hinein. In der Praxis lässt sich eine Community-of-Practice also als Interessensgemeinschaft beschreiben, die sich regelmäßig trifft, sich selbst eine Agenda gibt, Themen entwickelt und dabei ihre Erfahrungen austauscht und reflektiert. Sehr wichtig dabei ist, dass Frontalvorträge und Diskussionen in einem ausgewogenen Verhältnis zueinanderstehen und ein weitgehend hierarchiefreier Umgang miteinander möglich ist. Denn der größte Mehrwert einer Community-of-Practice besteht in der Reflexion verschiedener Lösungsansätze durch den Austausch verschiedener Argumente und Perspektiven.

Das Agile-Coaching-Dojo führt diesen Gedanken noch weiter. Die Namensgebung lehnt sich dabei bewusst an einem alten japanischen Begriff an: Das Dojo bezeichnet bei den traditionellen Kampfsportarten wie Judo sowohl den Trainingsraum, in dem die Übungen stattfinden, als auch die Trainingsgemeinschaft, die sich an diesem Ort versammelt. Dies verdeutlicht die Philosophie hinter unserem Format. Fachlich gesehen geht es zunächst darum, wie beim Sport feste Abläufe zu trainieren, damit diese sich in bestimmten Situationen routinemäßig abrufen lassen. Eine genauso wichtige Rolle spielt die Gemeinschaft, indem sie den Wissenstransfer zwischen Teilnehmern unterschiedlicher Erfahrungsstufen ermöglicht.

Im Agile-Coaching-Dojo stellen die Teilnehmer dafür Fallkonstellationen aus ihrer individuellen Erfahrung sowie die entsprechenden Lösungsansätze vor. Dies kann entweder die Lösung sein, welche im konkreten Projekt zur Anwendung gekommen ist, oder ein innovativer, noch zu erprobender Ansatz. Anschließend erfolgt die Diskussion mit den anwesenden Kollegen und Kolleginnen. Das Ziel dieser Übung ist, mehr über den Kontext der jeweiligen Problemstellung in Erfahrung zu bringen, die Lösung auf ihre Angemessenheit und Übertragbarkeit zu überprüfen und alternative Handlungsoptionen zu erörtern, aus denen sich neue und bessere Lösungsansätze entwickeln lassen. Die teilnehmenden Fachleute haben dabei die Gelegenheit, ihre Arbeit in einem vertrauensvollen Umfeld kritisch zu reflektieren, darin neue Aspekte zu beleuchten sowie ihren persönlichen Werkzeugkoffer mit ganz neuen Lösungsansätzen zu erweitern. Das gewonnene Know-how lässt sich nahtlos in den Beratungsalltag integrieren und schon am nächsten Tag gewinnbringend einsetzen. Insbesondere für Situationen, die eine herausfordernde und ungewöhnliche Problemstellung mit sich bringen, können Erfahrungswerte aus dem Agile-Coaching-Dojo hilfreich sein. Das Training ist die beste Lösung für eine gute Vorbereitung, um souverän reagieren zu können und gegebenenfalls auch verschiedene Lösungsvarianten anbieten zu können.

Vorbereitung und Umsetzung eines Agile-Coaching-Dojos

Ausgehend von der bisher definierten Philosophie und Zielsetzung des Veranstaltungsformats wird nachfolgend die konkrete Ausgestaltung einer Dojo-Veranstaltung beschrieben. Die folgenden Schritte lassen sich als Anleitung verwenden und unmittelbar in einer eigenen Veranstaltung umsetzen. Bevor es direkt in die Umsetzung des Events geht, sollten die Organisatoren die folgenden Punkte berücksichtigen, die es als Voraussetzung braucht:

- Auswahl geeigneter Fallbeispiele: Die Szenarien, die im Dojo zur Sprache kommen, sind die wichtigste inhaltliche Grundlage für das Format. Sie müssen geeignet sein, ausreichend praxisrelevante Problemstellungen zu adressieren sowie das Interesse potenzieller Teilnehmer zu wecken. Um viel beschäftigte Kolleginnen und Kollegen dazu zu bewegen, zusätzliche Zeit zu investieren, hat die Präsentation der Fallbeispiele einen hohen Stellenwert. Es hat sich bewährt, bei der Einladung griffige und spannungsgeladene Beschreibungen zu verwenden, zum Beispiel „Product-owner faces polar opposites", „Way down we go" oder „Convince me". Die Veranstaltungsthemen dürfen und sollen sogar einen emotionalen Bezug transportieren.
- Eine ausgewogene Teamzusammensetzung: Wichtig ist zu betonen, dass das Agile-Coaching-Dojo weder ein reines Anfänger- noch Experten-Format ist. Den größten Mehrwert bringt eine ausgewogene Zusammensetzung. Denn nehmen nur Kollegen auf Einsteiger-Level teil, fehlen die langjährigen Erfahrungswerte aus der Praxis. Beschränkt sich die Besetzung des Events auf die Senior-Stufe, kommt möglicherweise das Hinterfragen von (vermeintlichen) Selbstverständlichkeiten und der frische, innovative Blick auf Problemstellungen zu kurz. Die Philosophie des Agile-Coaching-Dojo zielt schließlich darauf ab, durch ein Durchsprechen von unterschiedlichen, breit gefächerten Perspektiven zu besseren Lösungen zu kommen.

- Die richtige Teilnehmerzahl: Die bewährte Teilnehmerzahl für das Agile-Coaching-Dojo bewegt sich zwischen acht und zwölf Personen. Diese Zahl sollte nicht unterschritten werden, da sich die Gruppe im Verlauf der Veranstaltung in Kleingruppen von jeweils drei bis vier Personen aufteilt. In diesen besprechen die Teilnehmer dann unterschiedliche Fälle. Außerdem bildet die eingangs erwähnte Größenordnung eine ausreichend breite Mischung an Perspektiven und Erfahrungswerten ab. Den größten Mehrwert bietet die Veranstaltung schließlich durch ihre Meinungsvielfalt und Diskussionsmöglichkeit. Sollte die Teilnehmerzahl größer als zwölf sein, empfiehlt es sich aus organisatorischen Gründen, eine weitere Gruppe ins Leben zu rufen und das Agile-Coaching-Dojo an einem separaten Termin stattfinden zu lassen. Wichtig bei dieser Konstellation ist allerdings, dass sich keine Clubs bilden, sondern die Interessenten jedes Mal neu durchgemischt werden.
- Aktiv für die Veranstaltung werben: Eine zu geringe Teilnehmerzahl ist nicht nur für die Veranstaltung hinderlich, sondern kann auch zu Frustration bei den Organisatoren führen. Um dem entgegenzuwirken sowie auch die besonders erfahrenen und vielbeschäftigten Fachleute für eine Teilnahme zu gewinnen, braucht es entsprechende Veranstaltungswerbung. Dies beginnt, wie eingangs erwähnt, bereits mit der Auswahl und Präsentation attraktiver Fallbeispiele. Zudem sollten alle zur Verfügung stehenden Medienkanäle genutzt werden: E-Mail-Verteiler, das Intranet, soziale Medien wie XING oder auch das persönliche Gespräch über den Flurfunk. Die eigene Marketingabteilung kann hier zusätzlich unterstützen, zum Beispiel mit einer professionellen Gestaltung der Einladung.

Sind die beschriebenen Voraussetzungen geklärt, besteht der nächste Schritt in der Festlegung der Agenda. Je nach spezifischer Zielsetzung und Gegebenheiten kann diese variieren. So lässt sich beispielsweise die Diskussion der Fallbeispiele mit einem allgemeinen Impulsvortrag zu Beginn der Veranstaltung verbinden. Wichtig ist, dass sich der Ablauf nach der vorab definierten Struktur richtet und insbesondere die zeitlichen Vorgaben eingehalten werden. Nachfolgend eine Beispielagenda, die sich bei bisherigen Veranstaltungen bewährt hat:

1. Einführung: Hier werden alle Teilnehmer in den Kontext und den Ablauf des Dojos eingeführt.
2. Gruppenfindung: Diese sollte möglichst ausgewogen ausfallen, also bei zwölf Teilnehmern insgesamt vier Gruppen zu je drei Personen, idealerweise sollten die Kleingruppen unterschiedliche Erfahrungsstufen abbilden.
3. Auswahl der Aufgaben: Die Organisatoren stellen die vorhandene Auswahl an Fallkonstellationen vor; die Kleingruppen entscheiden sich für jeweils eine.
4. Lösungsfindung: Die Gruppe skizziert für ihr ausgewähltes Fallbeispiel eine Lösung. Die Ergebnisse werden dabei in Stichworten festgehalten. Ein Teil der Lösungsfindung kann auch ein Rollenspiel sein, bei dem ein Beratungsgespräch mit dem Kunden dargestellt wird.
5. Vorstellung des Lösungsvorschlags im Plenum: In jeweils einem kurzen Vortrag präsentieren die Kleingruppen ihre Lösung. Jedes Team ist dazu angehalten, sich genau an die Zeitvorgabe zu halten.
6. Diskussion: Zu jedem Beitrag kann das Plenum nun Kritik, Lob und Ergänzungen äußern. Hierfür sollten die Organisatoren einen Moderator benennen, der die Ausgewogenheit der Themen und Redezeiten sicherstellt.

Falls für das Agile-Coaching-Dojo in etwa zwei bis drei Stunden zur Verfügung stehen – ein Ganztagesworkshop wäre zum Beispiel auch denkbar – hat sich die folgende Zeitvorgabe gut bewährt: Pro Fallkonstellation werden 30 Minuten festgesetzt, von denen 15 Minuten für die (gleichzeitige) Arbeit in Kleingruppen bereitstehen und dann pro Fall 15 Minuten für die Diskussion im Plenum. Damit lassen sich in einer Veranstaltung vier bis fünf Fälle behandeln. Wichtig ist, die vorab definierten Zeitslots einzuhalten und gegebenenfalls auch hart an der Stoppuhr abzubrechen, damit jede Konstellation gleichberechtigt zum Zug kommt. Sowohl die Ausarbeitung in den Kleingruppen als auch die Diskussion sollten protokolliert werden, um die Ergebnisse zur Verfügung zu stellen.

Nicht nur der fachliche Mehrwert zählt, sondern auch der zwischenmenschliche

Bei aller fachlichen Tiefe darf bei einer gelungenen Veranstaltung im Agile-Coaching-Dojo das Zwischenmenschliche nicht zu kurz kommen. Im Dojo finden nach der japanischen Tradition nicht einfach nur Übungspartner zusammen, sondern Gleichgesinnte, die ihre Philosophie gemeinsam leben. Ein geselliges Ausklingen, zum Beispiel bei einem Pizza-Essen, bietet eine wunderbare Gelegenheit für die Teilnehmer sich besser kennenzulernen und ungezwungen weiter auszutauschen. Hier können auch noch die Fragen beantwortet werden, die aus Zeitgründen während der Veranstaltung noch zu kurz gekommen sind.

Einen Kreis von themenaffinen Kolleginnen und Kollegen zu haben, bringt nicht nur Freu(n)de, sondern ist auch eine echte Hilfe im Alltag: Wer intern mit den Fachleuten aus dem eigene Unternehmen gut vernetzt ist, kann sich jederzeit bei einem Lunch oder in einem kurzen Telefongespräch einen Ratschlag holen. Die protokollierten Ergebnisse der Veranstaltung können außerdem als Basis einer internen Wissensdatenbank fungieren. Die wichtigsten Insights lassen sich damit über das Intranet für alle zugänglich machen. Jeder Einzelne trägt etwas dazu bei, wodurch die Community zusammenwächst und zum Unternehmenserfolg entscheidend beiträgt.

MicroStream

JCON 2019 - MicroStream Days

23.09.2019 - Starter Workshop
24.09.2019 - Special Day

Jetzt anmelden: www.jcon.one

NATIVE JAVA DATA STORE

Daten endlich direkt in Java speichern.
Keine externe Datenbanken mehr nötig.
Ultra-schnell. In-Memory. Super einfach. Pure Java.

www.microstream.one

JCON2019
www.jcon.one

JAVAPRO

DIE GROSSE JAVA COMMUNITY KONFERENZ

24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

Training-Day am 23. September 2019

Expo 24. - 26. September 2019

www.jcon.one

2	4	4	80+	99	800+
Konferenzen in einer	Kinos	Special Days	Speaker	Sessions	Teilnehmer

JCON 2019 PARTNER

GOLD PARTNER

ORACLE · Fast Lane Google Cloud · eclipse · MicroStream · XDEV

SILBER PARTNER

RAPIDclipse · consol Wir unternehmen IT

BRONZE PARTNER

viadee IT-Unternehmensberatung · trivago · TK Die Techniker · GEBIT Solutions

ORGANISATIONS-PARTNER

JAVAPRO · X DEVCON 2019 · XDEV