Die JAVAPRO ist kostenlos!
Einfach anfordern unter: www.javapro.io

89 **RESILIENZ DURCH ORGANIC-AGILITY**

92 **VERANTWORTUNG IN EINEM TEAM**

GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

3 / 2019

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis **# JAVAPRO**

Horror Serialisierung

Dramatische Sicherheitslecks in Java

6 **NEUES IN JAVA 13**

9 **MASTERING THE API HELL**

22 **COLLECTIONS EFFEKTIVER DURCHSUCHEN MIT JAVA-8-STREAMS**

37 **WEB- UND DESKTOP-ANWENDUNG AUS EINER CODE-BASE**

50 **JAVA - ABER SICHER!**

66 **HORROR SERIALISIERUNG**

78 **WAS DEVOPS WISSEN MÜSSEN**

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.

GOLD PARTNER

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

Die neue Java-Version 13 ist da, die neue Java-Serialisierung von MicroStream ist verfügbar, die JCON 2019, die vom 24. bis 26. September 2019 zum dritten Mal in Düsseldorf stattfindet, steht unmittelbar bevor und wir haben dieses Mal noch mehr spannende Beiträge von unseren JAVAPRO-Autoren und JCON-Speakern erhalten, sodass diese Ausgabe etwas üppiger ausfällt als gewohnt. Zu den mit JCON-Hinweis gekennzeichneten Artikeln findet jeweils Vortrag auf der JCON 2019 statt.

Die JAVAPRO gibt es jetzt seit über zwei Jahren. Wir sind noch immer ein kleines Team, aber wir haben bereits zahlreiche spannende Ausgaben produziert, veranstalten heuer das dritte Mal die JCON in Düsseldorf und es macht unglaublich viel Spaß an der JAVAPRO mitzuwirken. An dieser Stelle möchten wir uns auch ausdrücklich bei unseren großartigen Autoren bedanken, bei unseren JCON-Speakern und bei unseren Partnern, die uns bei der Finanzierung der JAVAPRO unterstützen und ein kostenloses Magazin für Java-Profis überhaupt erst möglich machen.

Nach der JCON beginnt für die JAVAPRO eine neue Phase und wir einige Änderungen vornehmen. Unser Ziel ist es, mehr Beiträge als bisher für die tägliche Programmierpraxis zu bieten. Der Anteil an Core-Java- und API-Themen soll größer, die Länge der einzelnen Beiträge dafür etwas kürzer werden.

Die JAVAPRO-Organisation soll verbessert und professioneller werden. Wir wollen unsere Social-Media-Aktivitäten erhöhen, unsere Webseite www.javapro.io verbessern und noch mehr Inhalte bieten, einen JAVAPRO-Podcast starten und unser neues JAVAPRO-Job-Portal launchen. Das bislang noch kleine JAVAPRO-Team wird richtig wachsen. Und wir möchten noch mehr Leser für die JAVAPRO begeistern. Doch jetzt wünsche ich Ihnen viel Spaß beim Lesen dieser Ausgabe!

Markus Kett
Chefredakteur
JAVAPRO

Twitter: @MarkusKett
LinkedIn: markuskett

6
CORE JAVA

Neues in Java 13

Von Rodion Alukhanov und Alexander Peters

Die neue Java-Version 13 ist für September 2019 geplant. Java 13 ist kein großes LTS-Release (Long-Term-Support). Zu den Neuigkeiten zählen 5 Features, von denen 2 für Entwickler interessant sein dürften, u.a. die Änderungen bei den Switch-Expressions.

Sehr vielversprechend ist der neue Z Garbage-Collector, der eine höhere Geschwindigkeit bei gleichzeitig niedrigere CPU-Auslastung bieten soll.

Gemäß dem von Oracle neu eingeführten Release- und Support-Modell für Java-SE wird Java 13 bis zum Erscheinen von Java 14 kostenlos supportet.

66
SECURITY

Horror Serialisierung

Von Sebastian Späth

Serialisierung gilt als größte Sicherheitslücke in Java. Die Hälfte aller Schwachstellen in Java hat mit Serialisierung zu tun. Zudem unterliegt die Serialisierung zahlreichen Einschränkungen. Oracle trifft keine konkrete Aussage, wann und ob man das Problem angehen wird. Aus der Java-Community gibt es jetzt eine vielversprechende Alternative. MicroStream hat eine von Grund auf neu entwickelte Serialisierung für Java veröffentlicht, die nicht nur sicher sein soll, sondern auch sämtliche Einschränkungen der Java-Serialisierung behebt.

6
CORE JAVA
Neues von Java 13
Java 13 ist da! Die neue Version bringt einige vielversprechende Features.

9
CORE JAVA
Mastering the API-Hell
Microservices spielen oft zusammen und es entstehen Abhängigkeiten. Contract-Testing hilft dieses Problem zu lösen.

22
CORE JAVA
Collections effektiver durchsuchen mit Java-8-Streams
Java-8-Streams sollten alle Java-Entwickler kennen. Dieser Artikel bietet einen ausführlichen Überblick über die API.

27
CORE JAVA
Deep-Dive into Annotations – Teil 3
Annotationen sind ein mächtiges Sprachmerkmal. In Teil 3 unserer Serie geht es um die Auwertung eigener Annotationen zur Laufzeit.

33
FRAMEWORKS & APIS
Spring-Boot im Legacy-Kontext
Mit Spring-Boot lassen sich auch Altanwendungen auf eine Microservice-Architektur umbauen.

37
FRAMEWORKS & APIS
Web- und Desktop-Anwendung aus einer Code-Base
Moderne Java-Applikationen entwickeln, die man ohne Zusatzaufwand als Web- und klassische Desktop-Applikation ausliefern lassen.

44
FRAMEWORKS & APIS
Getting Hip with JHipster
Per Framework Spring-Boot-Web-Anwendungen mit Angular-, React- oder Vue-Frontend generieren - für Einsteiger und Profis.

50
SECURITY
Java - aber sicher!
Überblick über die gefährlichsten Sicherheitslücken in Java, vor denen man seine Java-Anwendung schützen muss.

52
SECURITY
Risiko Open-Source
Diese Sicherheits- und Lizenzrisiken müssen Entwickler im Auge behalten um bei Open-Source auf der sicheren Seite zu sein.

55
SECURITY
10 Grundsätze für sichere Softwareentwicklung
Diese 10 Security-Grundsätze sollte jeder Java-Entwickler kennen und beherzigen.

61
SECURITY
JavaScript Security – Best-Practice
Dieser Artikel erklärt, wie man die Weichen für eine sichere Entwicklung mit JavaScript stellt.

66
SECURITY
Horror Serialisierung
Serialisierung ist die mit Abstand größte Sicherheitslücke in Java. Mit MicroStream gibt es jetzt eine Alternative.

72
TESTING & QUALITY
Nachhaltige Report-Entwicklung mit TDD
Der Wert automatisierter Softwaretests ist unbestritten. Aber auch Qualität und Wartbarkeit von Reports lässt sich damit steigern.

76
CLOUD, CONTAINER & DEVOPS
Container-Monitoring mit Prometheus
Prometheus erlaubt den Blick ins Innere. Ereignisse, Fehler und Mängel lassen sich vorhersehen und verhindern bevor sie eintreten.

78
CLOUD, CONTAINER & DEVOPS
Was DevOps heute wissen müssen
12 Bereiche in denen DevOps Kompetenzen besitzen müssen.

84
AGILE & PROJEKTMANAGEMENT
Architekturentscheidung in agilen Projekten
Die SWOT-Analyse kann Teams dabei helfen, die richtige Architekturentscheidung zu treffen.

89
AGILE & PROJEKTMANAGEMENT
Resilienz durch Organic-Agility
Ein evolutionärer Ansatz der Organisationen dabei helfen kann, sich auf die Anforderungen der heutigen Realität einzustellen.

92
AGILE & PROJEKTMANAGEMENT
Verantwortung in einem Team
Wie man Verantwortung richtig (ver-)teilt, dass die gewünschten Ergebnisse erzielt werden?

94
AGILE & PROJEKTMANAGEMENT
Wir entscheiden zusammen, nicht allein
Teams sollten Technologieentscheidungen selbst treffen können. Aber es sind einige Grundregeln zu beachten, dass es funktioniert.

97
AGILE & PROJEKTMANAGEMENT
Technische Schulden in der DevSecOps-Welt
Bei DevOps wird Sicherheit noch zu oft vernachlässigt. Technische Schulden können auch bei DevOps-Projekten ein Gewinn sein.

2
MAGAZIN
Partner Network & Impressum

3
MAGAZIN
Editorial

#JAVAPRO #CoreJava #JDK

Neues in Java 13

Java-Release 13 ist für September 2019 geplant. Es ist bereits das dritte Major-Release nach der Bekanntgabe des neuen Lizenzmodells und Release-Zyklus im September 2018[1]. Einige Features sind vielversprechend. Für Entwickler halten sich die neuen Features jedoch in Grenzen.

Autor:

Rodion Alukhanov ist Senior-Software-Entwickler bei ip.labs GmbH mit Sitz in Bonn. Seine mehrjährige Erfahrung streckt sich von MS-DOS über die Hausautomation bis in die AWS-Cloud. Seine Schwerpunkte sind Cloud-Migration, Big-Data und Integration-Tests.

Alexander Peters ist Senior-Software-Entwickler bei ip.labs GmbH mit Sitz in Bonn. Er hat mehr als zehn Jahre Berufserfahrung in der webbasierten Softwareentwicklung mit Java (EE) und Spring Framework. Seine Schwerpunkte liegen bei der serverseitigen Anwendungsentwicklung sowie bei der Definition und Implementierung von externen und internen Schnittstellen.

Zu den Neuigkeiten von Java 13 gehören 5 Features, welche in Form von JDK-Enhancement-Proposal (JEP) beschrieben sind. Die meisten Änderungen des Releases betreffen interne Optimierungen und Refactorings. Die für Softwareentwickler sichtbaren Features - und davon gibt es zwei - sind in diesem Artikel als erste beschrieben. Java 13 gehört nicht zu den sogenannten LTS-Releases (Long-Term-Support) und wird bereits im März 2020 von JDK 14 abgelöst2.

JEP 354: Switch-Expressions (Preview)

Bei JEP 354 handelt sich dabei um eine Erweiterung von JEP 325 (Switch-Expressions), was als Preview-Language-Feature[3] in JDK 12 erschienen ist. Die Änderung basiert auf dem Gavin Bierman's Vorschlag, neue Form der **`break`** Anweisung mit einem Wert durch die **`yield`** Anweisung zu ersetzen.

(Listing 1 - Vor der Änderung)

```
int j = switch (day) {
    case MONDAY  -> 0;
    case TUESDAY -> 1;
    default      -> {

        int result = f(day);
        break result;
    }
};
```

(Listing 2 - Vor der Änderung)

```
int j = switch (day) {
    case MONDAY  -> 0;
    case TUESDAY -> 1;
    default      -> {
        int result = f(day);
        yield result;
    }
};
```

(Listing 3 – Traditionelle Syntax)

```
int result = switch (s) {
    case "Foo":
        yield 1;
    case "Bar":
        yield 2;
    default:
        System.out.println("Neither Foo nor Bar, hmmm...");
        yield 0;
};
```

Somit kann man leichter zwischen den Switch-Anweisungen und Switch-Ausdrücken unterscheiden:
•Break-Anweisung wird in einer Switch-Anweisung, jedoch nicht im Switch-Ausdruck verwendet.
•Das Ziel einer Yield-Anweisung kann ein Switch-Ausdruck, aber keine Switch-Anweisung sein.

JEP 355: Text-Blocks (Preview)

Das früher für JDK 13 geplante Feature-Raw-String-Literals (JEP 326) musste dem JEP 355 weichen. Dabei geht es um das Einführen von formatierten Textblöcken, wie in dem folgenden Beispiel:

(Listing 4)

```
var html = """
..........<html>
..........  <body>Hello, world</body>
..........</html>
..........""";
```

Während dieses Feature in produktionsreifem Code eher weniger Verwendung findet, soll es die Entwicklung von Tests oder Prototypes spürbar erleichtern. Bei der Generierung von Bytecode werden die Textblöcke normalisiert und in normale Java-Strings umgewandelt. Zur Normalisierung gehört unter anderem:

- Incidental-White-Spaces werden entfernt (in dem Beispiel als Punkte dargestellt)
- Trailing-White-Spaces werden entfernt
- Zeilenumbrüche werden unabhängig von der Plattform zu LF (auch bekannt als **`\n`**)

Die Interpretation von Escapes (wie z.B. **`\t`** oder **`\n`**) erfolgt nach der Normalisierung und funktioniert wie in herkömmlichen Strings. Die aus Kotlin oder Scala bekannten Variablenplatzhalter innerhalb des Blocks werden im Rahmen JEP 355 nicht unterstützt. Die Spezifikation verweist explizit auf zukünftige Implementierungen.

Beim Auswerten von Text-Blöcken werden aktuell nur die Einrückung mit Spaces unterstützt (Build 25 vom 14.06.2019). Die Incidental-Tabs werden ignoriert. Das neue Feature wird dadurch einen entscheidenden Einfluss im „Krieg Tabs versus Spaces" haben. Die Tabs innerhalb des Textes werden unverändert beibehalten und wie **`\t`** interpretiert.

JEP 350: Dynamic-CDS-Archives

Die erstmals in Java 5 erschienene Funktion Class-Data-Sharing erlaubte es, mehreren JVMs die einmal geladenen Classes mittels CDS-Archives zusammen zu nutzen. Die ursprünglich auf Bootstrap-Class-Loader beschränkte Funktion wurde zuletzt im JDK-Release 10 im Rahmen von JEP 310 auf den Application-Class-Loader ausgeweitet. Die neue verbesserte Version ist als AppCDS bekannt. Dies kam in der ersten Linie den Application-Servern zugute: Je nach Anwendungsfall sind laut JEP-Beschreibung deutliche Einsparungen beim RAM-Verbrauch zu verzeichnen. Die Startup-Zeiten verbesserten sich zum Teil um bis zu 30 Prozent. Das Benutzen des AppCDS blieb aufwendig. Im Grunde musste man die Liste von archivierten Klassen manuell erzeugen und pflegen. Im Rahmen von JEP 350 wurde AppCDS nochmal erweitert und deren Nutzung vereinfacht. Anstatt wie bis jetzt in mehreren Programmprobeläufen eine Klassenliste manuell zu erzeugen und danach das Archive (mit der Option **`-Xshare:dump`**) zu erstellen, werden alle geladenen Klassen und Bibliothekklassen am Programmende dynamisch zu einem

Archiv hinzugefügt, soweit diese nicht im Base-Layer CDS-Archiv enthalten sind. Eine nachträgliche Erweiterung dieses JEP könnte eine automatische Archivgenerierung beim ersten Lauf einer Anwendung durchführen. Die Nutzung von CDS/AppCDS könnte dann völlig transparent und automatisch erfolgen. Aktiviert wird das Feature mit der Option -**XX:ArchiveClassesAtExit=<filename>**. Das erzeugte Archiv kann dann mit der Option -**XX:SharedArchiveFile=<filename>** verwendet werden:

```
% java -XX:ArchiveClassesAtExit=hello.jsa -cp hello.jar Hello
```

Verwendung des Archives in der gleichen Anwendung:

```
% java -XX:SharedArchiveFile=hello.jsa -cp hello.jar Hello
```

JEP 351: ZGC - Uncommit unused Memory

In Java 11 wurde der neue Z Garbage-Collector[4] (ZGC) eingeführt, der sich aktuell noch in der Entwicklung befindet. Der neue Garbage-Collector ist sehr vielversprechend, da er eine höhere Geschwindigkeit bietet und niedrigere CPU-Auslastung für großen Heap (>16Gb) ermöglicht. Im Moment steht ZGC nur für 64-Bit Linux-Plattformen zur Verfügung. Im Gegensatz zu anderen Garbage-Collectoren wie G1 oder Shenandoah kann ZGC bislang keinen Heap-Speicher an das Betriebssystem zurückgeben, auch dann nicht, wenn dieser längere Zeit nicht mehr verwendet wird. ZGC-Heap besteht aus Heap-Regionen, die als ZPages bezeichnet werden. Jede Region ist mit einem bestimmten Teil des Speichers in einer variablen Größe verknüpft. Falls ZGC den Heap komprimiert, werden die freigegebenen ZPages in einen Cache (ZPageCache) abgelegt, um diese bei Bedarf schnell wieder verwenden zu können. Diese Vorgehensweise ist für die Programm-Performance entscheidend, da die Speicherzuweisung und -freigabe eine relativ teure Operation ist. Der Cache hält die ZPages nach der letzten Nutzung (least-recently-used) sowie nach der Größe klein, mittel und groß sortiert. Somit ist es relativ einfach, diese nach einer definierten Zeit aus dem Cache zu entfernen und den Speicher frei zu geben. Eine einfachere Möglichkeit wäre es, einen Timeout- oder Delay-Wert zu definieren, wie lange eine ZPage im Cache liegen darf, bevor diese aufgeräumt wird. Dieser Wert sollte einen vernünftigen Default-Wert besitzen und bei Bedarf überschreibbar sein. Eine Alternative dazu könnte eine komplexeres Verfahren sein, das anhand von GZ-Frequenz und anderen Daten einen geeigneten Timeout-Wert ermittelt und anwendet. Welche der beiden Optionen zum Einsatz kommen wird, ist zum jetzigen Zeitpunkt noch nicht bekannt. Es ist denkbar, dass zuerst eine konfigurierbare Variante (-**XX:ZUncommitDelay=<seconds>**) veröffentlicht wird und die komplexere Version später nachgeliefert wird. Die Uncommit-Funktion ist per Default aktiv, kann aber mit **-XX:-ZUncommit** explizit deaktiviert werden.

JEP 353: Reimplement the Legacy Socket-API

Diese Änderung enthält umfangreiches Refactoring und Neuimplementierung der Java-Socket- und ServerSocket-API. Der Code wurde mithilfe von NIO-Bibliotheken grundlegend überarbeitet. Fehlerbehandlung und Garbage-Collector-Funktionalität wurden verbessert. In erster Linie handelt es sich hierbei vielmehr eine interne Änderung, welche die zukünftige Entwicklung des JDK erleichtern soll. Eine der wichtigsten Motivationen dieses Projektes ist ein weiterer Schritt hin zu einem leichtgewichtigen, kooperativen Ausführungsmodell des Projektes Loom[5]. Ältere Entwickler werden sich dabei vielleicht noch an Cooperative-Multitasking von Windows 3.1 erinnern. Die Umgebung, in unserem Fall die JVM, übernimmt die Steuerung zwischen virtuellen Threads und nutzt dabei eine sehr viel kleinere Anzahl an echten System-Threads. Solche virtuellen Threads werden Fibers genannt. Beim Verwalten von Fibers muss die JVM in der Lage sein, den aktuell laufenden Code zu unterbrechen und zu einem anderen Code zu schalten.

Dafür eignen sich vor allem Programmstellen, an denen die Anwendung ohnehin auf externe I/O Prozesse warten muss. Zwei Voraussetzungen sind für die Umsetzung unverzichtbar: Die eigentliche I/O-Kommunikation darf dabei nicht unterbrochen werden, sondern muss im Hintergrund weiterlaufen. Die I/O Kommunikation selbst sollte dabei nicht nativ programmiert, sondern ebenfalls in Java implementiert werden, da die JVM ansonsten keine Möglichkeiten hat, das Fiber anzuhalten.

Da die alte Socket-API im Blocking-Mode lief und zum größten Teil nativ programmiert wurde, war eine Änderung des JEP 353 für die Weiterentwicklung bitter nötig. Ansonsten befindet sich das Projekt Loom noch in einem frühen Entwicklungsstadium und ist für die meisten Entwickler nicht von Bedeutung. Das Java-Team verspricht die Änderungen für die häufigsten Use-Cases transparent zu gestalten. In Bezug auf Geschwindigkeit ist die neue Implementierung laut JEP-Beschreibung nur minimal besser: etwa 0 bis 3 Prozent. Aus Kompatibilitätsgründen wird es noch eine Weile möglich sein die alte Implementierung von Sockets zu nutzen. Die Steuerung erfolgt wie gewohnt über die -D Parameter.

Quellen:

1 JAVAPRO - Die freie JDK-Wahl: https://javapro.io/java-11
2 JEP 12 - Preview Language and VM-Features: https://openjdk.java.net/jeps/12
3 JEP 333 - A Scalable Low-Latency Garbage-Collector: https://openjdk.java.net/jeps/333
4 Loom - Fibers, Continuations and Tail-Calls for the JVM: https://openjdk.java.net/projects/loom/

JAVAPRO #CoreJava #API

Mastering the API-Hell

Microservices sind heute nicht mehr wegzudenken. Doch Abhängigkeiten der Schnittstellen zwischen den Services verhindern oftmals eine unabhängige Weiterentwicklung der Services. Contract-Testing hilft dieses Problem zu lösen.

Software wird immer öfter nicht nur als Werkzeug gesehen, sondern als zentraler Baustein der Produktstrategie. Mit der steigenden Bedeutung der Software, muss diese schnell weiterentwickelt werden können, um neue Features mit wenig Verzögerung zum Kunden zu bringen. Eine häufig angewandte Methode, um Software schnell und parallel entwickeln zu können, ist die Zerteilung der Software in kleine, einfach überschaubare und leichtgewichtige Microservices. Obwohl diese Microservices unabhängig voneinander entwickelt und released werden können, bleiben viele Features auf ein Zusammenspiel mehrerer Services angewiesen. An ihren Schnittstellen bleiben Services voneinander abhängig. Oftmals praktizierte Lösungen sind Absprachen und eng abgestimmte, koordinierte Änderungen der API. Diese widersprechen aber dem Wunsch nach unabhängigen Entwicklungs- und Releasezyklen. Unabhängig davon sind manuelle Prozesse immer fehleranfällig und stehen automatisierten Praktiken wie Continuous-Delivery und -Deployments diametral entgegen. Integrative Tests im gesamten Verbund aller Microservices sind teuer und aufwändig und bremsen die unabhängige Entwicklung der einzelnen Services zusätzlich aus.

Autor:

Nikolai Neugebauer ist als Consultant für Digital Frontiers tätig. Sein Schwerpunkt liegt auf agiler Softwareentwicklung, vorwiegend im Java und Spring Umfeld. Außerdem beschäftigt er sich mit aktuellen UI Technologien im Webbereich.

Florian Pfleiderer beschäftigt sich als Senior Consultant bei Digital Frontiers mit agiler Software-Entwicklung. Seine Kunden berät er in den Bereichen Architektur, Microservices und Craftsmanship.
https://www.digitalfrontiers.de
https://blog.digitalfrontiers.de

Twitter: @nik101010
Twitter: @pfleidfn

Email: nikolai.neugebauer@digitalfrontiers.de
Email: florian.pfleiderer@digitalfrontiers.de

API-Typen

Entwickeln wir verteilte Software, so stoßen wir regelmäßig auf APIs, um andere Services zu konsumieren, oder von diesen konsumiert zu werden. Generell gibt es dabei verschiedene Level der Abhängigkeit.

Interne APIs sind zumeist am einfachsten zu handhaben. Hierbei sind Consumer und Provider, also Nutzer und Anbieter der API identisch. Solche APIs kommen häufig vor, wenn ein Team seine Komponenten der Software weiter zerteilt und auf mehrere Services aufgeteilt hat. Auch wenn hier Kommunikation keinen Overhead verursacht und koordinierte Releases verhältnismäßig unkompliziert möglich sind, bleibt die Komplexität der Abhängigkeiten, die API Änderungen schwierig machen.

Schwieriger gestaltet sich die Kommunikation schon, wenn der Consumer oder Provider der API nicht mehr das gleiche Team ist, sondern ein anderes Team im Unternehmen. Bei solchen Partner-APIs sind koordinierte Änderungen zwar immer noch möglich, doch sie erfordern bereits deutlich erhöhten Kommunikationsaufwand und die Auswirkungen von Änderungen sind nicht mehr überschaubar.

Bei öffentlichen APIs bietet der Provider seine API jedem an und kennt seine Consumer und deren Verwendung der API nicht. Für den Provider ist damit nicht absehbar, ob er durch Änderungen an seiner API Consumer bricht. Dies birgt für den Consumer immer das Risiko, dass seine Anwendung ohne eigenes Verschulden nicht mehr funktioniert.

Probleme impliziter Contracts

Die Funktionsweise von APIs ist oftmals nicht explizit spezifiziert, sondern basiert auf impliziten Annahmen oder Absprachen. Insbesondere interne APIs werden bei Bedarf entwickelt und auf den konkreten Anwendungsfall zugeschnitten. Benötigt die Implementierung eines neuen Features Daten oder Funktionalität aus mehreren Microservices, so werden neue interne APIs erstellt, die genau diesen Anwendungsfall abdecken. Die Funktionalität der APIs wird oftmals nur besprochen, selten jedoch explizit spezifiziert. Eine Dokumentation der Schnittstelle erfolgt oft erst hinterher. Dieses Vorgehen wird gerne damit gerechtfertigt, dass die API nur teamintern verwendet wird, die Auswirkungen einfach überschaubar sind und die Schnittstelle auch unkompliziert geändert werden kann. Die Entwicklungszyklen der beiden Komponenten sind damit aber aneinander gekoppelt und bei jeder Weiterentwicklung muss diese Kopplung berücksichtigt werden. Mit der Zeit entstehen immer weitere Schnittstellen, die Abhängigkeiten werden komplexer, die Übersicht geht verloren.

Auch bei Partner-APIs fehlt es oft an klarer Spezifikation. Ursprünglich intern konzipierte Schnittstellen entwickeln sich zum Beispiel zu Partner-APIs, weil vorhandene Funktionalität Kollegen aus anderen Teams zur Verfügung gestellt wird. Oftmals werden Partner-APIs nur informell besprochen. Dadurch fehlt es an konkreter Spezifikation. Problematisch ist hierbei nicht nur, dass die Weiterentwicklung der APIs eng aneinander gekoppelt wird, auch die initiale Entwicklung von Consumer und Provider ist eng aneinander gekoppelt. Die wirkliche Funktionalität der Schnittstelle ist für den Consumer erst dann ersichtlich, wenn die API fertig entwickelt ist. Wird bei der Entwicklung des Consumers abgewartet bis der Provider fertig entwickelt ist, so entsteht eine längere Entwicklungszeit als bei einer parallelen Entwicklung. Startet die Entwicklung des Consumers dagegen bevor der Provider fertiggestellt ist, so wird der Consumer auf Annahmen basierend entwickelt, was ebenfalls Zeit- und Kommunikationsaufwand bedeutet.

Öffentliche APIs sind in der Regel dokumentiert, sodass sie einfach verwendet werden können. Oft erfolgt die Spezifikation der API auch in Formaten, aus denen Clients generiert werden können, zum Beispiel OpenAPI[1]. Solche generierten Clients haben aber den Nachteil, dass sie auch Aspekte der API verwenden, die für den Consumer keine Relevanz haben und somit wartungsintensiver sind. Schreibt man den Client selbst, so benötigt man zum Testen wieder ein komplettes System. Alternativ generiert man Mocks, bei denen die Gefahr besteht, dass sie eigene Annahmen enthalten, die sich nicht mit der Funktionalität der API decken.

Problematisch bei allen drei API-Arten ist, dass eine reine Dokumentation kaum geeignet ist, um die Kompatibilität bei Weiterentwicklungen zu gewährleisten. Außerdem ist die Entwicklung der Services eng aneinander gekoppelt - der Consumer ist immer vom Provider abhängig, oder läuft Gefahr, eigene Annahmen in genutzte Mocks einzubauen. Für realistische Tests müssen daher immer alle Services, oft noch mit eigenen Abhängigkeiten, hochgefahren werden. Dadurch wird das Testen aufwändig, teuer und zeitintensiv. Die Entwicklungs- und Release-Geschwindigkeit nimmt ab.

Contract-Testing

Eine Lösung für die beschriebenen Probleme ist Contract-Testing. Beim Contract-Testing vereinbaren Consumer und Provider einen expliziten Vertrag (Contract), der die Schnittstelle beschreibt. Aus dieser Beschreibung werden dann sowohl Test-Clients als auch Test-Provider generiert. Der Provider testet seine Implementierung gegen den Test-Client, der Consumer gegen den Test-Provider. Contracts definieren hierfür beispielhafte Requests. Für die Tests des Providers werden die definierten Requests gegen die API ausgeführt und die zurückgelieferten Antworten mit den erwarteten Antworten verglichen. Stimmen die erwartete Antwort und die tatsächliche Antwort nicht überein, so erfüllt der API-Provider die Erwartungen an die Schnittstelle nicht. Für die Consumer-Tests ruft der Consumer den Test-Provider auf, der die im Contract definierte Antwort zurückliefert. Wird keine

Antwort zurückgeliefert, so war der vom Client erzeugte Request fehlerhaft. Wird die Antwort vom Client nicht korrekt verarbeitet, so ist die Implementierung des Clients ebenfalls fehlerhaft. Ein beispielhafter Auszug aus der Antwort einer REST-Schnittstelle, die Kundeninformationen zurückliefert ist in **(Abb. 1)** dargestellt.

Der Contract dazu würde definieren, dass bei einem Get-Aufruf des `/customers` Endpunkts diese Antwort zurückgeliefert wird. Fehlt das Feld **`name`** in der Antwort, ist der Provider fehlerhaft. Ruft der Client nicht `/customers` auf oder kann die Antwort nicht verarbeiten, ist der Client fehlerhaft. Da weder der Provider einen realen Consumer benötigt, noch umgekehrt, können beide Seiten bereits vor der Fertigstellung der anderen API-Seite entwickelt und getestet werden. Die Entwicklungen von Consumer und Provider sind damit entkoppelt und können unabhängig voneinander vorangetrieben werden. Weiterhin hat dieses Vorgehen den Vorteil, dass sowohl API-Consumer als auch API-Provider gegen dieselbe Spezifikation der API testen. Einseitige Annahmen bleiben aus. Außerdem können auch Tests ohne Aufbau einer vollständigen Umgebung durchgeführt werden. Dadurch sind Tests deutlich einfacher und häufiger durchführbar. Dies hilft insbesondere bei der Weiterentwicklung der API.

```
{
  "name": "Max Meier"
}
```

Antwort Auszug API Version 1. (Abb. 1)

```
{
  "firstName": "Max",
  "lastName": "Meier"
}
```

Antwort Auszug API Version 2. (Abb. 2)

```
{
  "name": "Max Meier",
  "firstName": "Max",
  "lastName": "Meier"
}
```

Antwort Auszug API Version 3. (Abb. 3)

Zunächst erscheinen fixe Contracts eine Weiterentwicklung der API zu behindern und wie ein Rückschritt hinsichtlich API-Evolution zurück in Zeiten von strikten API-Definitionen wie zum Beispiel mit RMI oder WSDL. Doch Contracts zwingen keinesfalls dazu, jede API-Änderung zu unterlassen. Stattdessen helfen sie dabei, inkompatible Änderungen bereits in frühen Tests aufzudecken. Wichtig für API-Evolution ist es, die Contracts nicht statisch zu sehen, sondern diese wenn nötig an neue Anforderungen anzupassen. Eine Weiterentwicklung der Schnittstelle aus **(Abb. 1)** liefert zum Beispiel Vorname und Nachname des Kunden einzeln zurück. Der Auszug aus der Antwort ist in **(Abb. 2)** dargestellt.

Die Contracts werden hierfür angepasst, sodass sie das neue Verhalten abbilden, also die aktuelle Schnittstelle definieren. Dadurch bleiben aber auch die Contract-Tests des Providers erfolgreich, da diese die neue Schnittstelle gegen die neuen Contracts testen. Anders als oftmals in der Realität, ist in diesem einfachen Beispiel sofort ersichtlich, dass die Consumer, die das Feld **`name`** benötigen, nicht mehr mit dieser API-kompatibel sind. Geht diese neue API-Version live, sind alle Consumer, die das Feld **`name`** erwarten, defekt. Dieses Problem muss möglichst früh im Entwicklungszyklus entdeckt werden. Eine einfache Lösung um diese Inkompatibilität aufzudecken ist es, den Provider zusätzlich gegen die Contracts der Vorversion zu testen. In dieser wird das Feld **`name`** erwartet, das jetzt fehlt. Die Tests schlagen fehl.

Eine Möglichkeit mit dieser fehlenden Abwärtskompatibilität umzugehen ist, die Consumer ebenfalls anzupassen. Allerdings bringt dieses Vorgehen das Problem, dass Consumer und Provider koordiniert veröffentlicht werden müssen. Dies hat eine enge Kopplung von Consumer und Provider und einen deutlich erhöhten Aufwand zur Folge und ist daher nicht wünschenswert.

Eine zweite Option wäre das Einführen von API-Versionen. Neue Consumer verwenden die neue Version der API, die alte Version bleibt aber für bestehende Consumer weiterhin verfügbar. Auch dieses Vorgehen ist mit deutlich erhöhtem Aufwand verbunden, da der Provider zukünftig mehrere Versionen der API warten muss. Außerdem kann es weitere Änderungen der API geben, was wiederum neue Versionen bedeutet.

Die beste Option ist, die API abwärtskompatibel zu halten. Hierfür muss die Antwort alle 3 Felder enthalten, wie in **(Abb. 3)** dargestellt.

Durch diese weiche Migration können API-Consumer und API-Provider unabhängig voneinander entwickelt und veröffentlicht werden, da die API-Versionen miteinander kompatibel sind. Werden die Clients gegen die aktuellen Contracts der API getestet, die das Feld **`name`** nicht mehr enthalten, so fällt die Inkompatibilität mit zukünftigen Versionen der API auf. Die Clients können dann angepasst werden, bevor das Feld **`name`** aus der API entfernt wird.

Der Provider bekommt durch die Tests gegen die Vorversion der Contracts die Sicherheit, dass inkompatible Änderungen seiner API bereits in Tests auffallen. Die Consumer erhalten die Möglichkeit API-Änderungen frühzeitig zu testen, ohne dass die API bereits den geänderten Stand aufweisen muss. Beide Seiten können durch den Einsatz von Contract-Tests profitieren und können API Inkompatibilitäten in Produktion verhindern.

Das beschriebene Vorgehen ist jedoch nur bei APIs möglich, bei denen Consumer und Provider Contract-Testing verwenden. Bei öffentlichen APIs muss das jedoch nicht der Fall sein. Der Provider kann, um seine Kompatibilität zu Vorversionen sicherzustellen zwar Contract-Testing nutzen, der Consumer kann sich in der Regel jedoch nicht darauf verlassen. Außerdem hat der Consumer meist keinen Zugriff auf die Contracts des Providers und umgekehrt. Für den Consumer bleibt damit das Risiko, dass die Schnittstelle inkompatibel zur eigenen Verwendung wird.

Eine Lösung für den Consumer ist es einen Contract-Proxy aufzubauen. Der Consumer schreibt hierfür Contracts, die seine Erwartungen an die Schnittstelle ausdrücken. Die aus diesen Contracts erzeugten Tests, werden dann gegen die API ausgeführt. Sind die Tests erfolgreich, enthalten die Contracts keine nicht erfüllten Annahmen. Der Consumer kann also gegen die

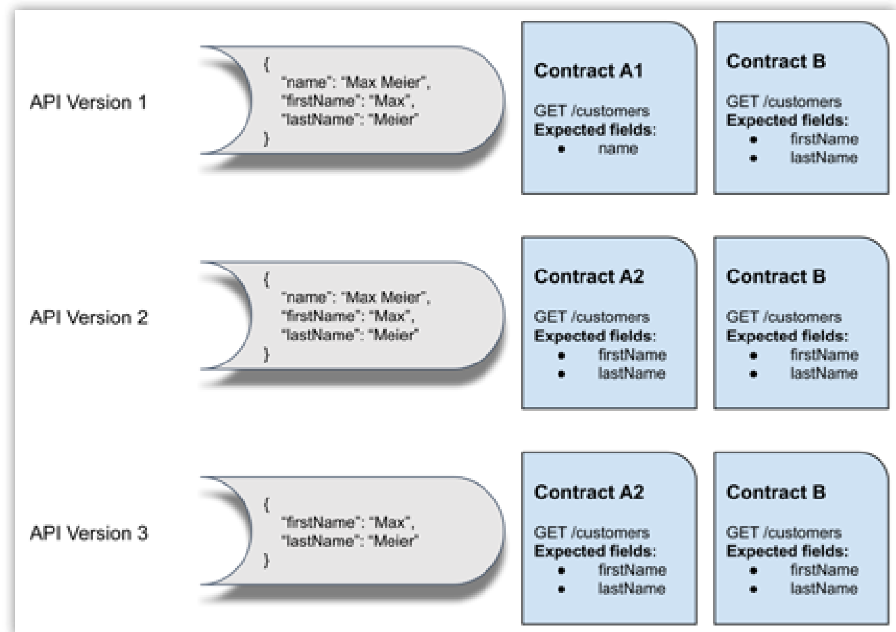
aus den Contracts erzeugten Mocks testen. Führt der Contract-Proxy die API-Tests regelmäßig gegen die echte API aus, so werden Inkompatibilitäten zu den eigenen Annahmen zumindest automatisiert erkannt.

Consumer-Driven-Contract-Testing

Consumer-Driven-Contract-Testing ist eine Erweiterung von Contract-Testing. Hierbei drückt der Consumer einer API seine Erwartungen an die API in Form von Contracts aus. Dieses Vorgehen ist also nur möglich, wenn Consumer und Provider der API einander kennen, sprich nur bei internen und gegebenenfalls bei Partner-APIs. Dadurch, dass die Consumer die Contracts schreiben und dabei explizit ihre Erwartungen an die API ausdrücken, lässt sich gut erkennen, welche Komponenten der API genutzt werden. Ungenutzte Komponenten können dann problemlos geändert oder entfernt werden. Es besteht außerdem keine Möglichkeit, verwendete Komponenten zu entfernen, bevor nicht der letzte Consumer seine Contracts angepasst hat. (Abb. 4) zeigt eine vom Consumer getriebene Evolution einer API. Consumer A und Consumer B nutzen beide den /**customers** Endpunkt der API aus (Abb. 3) des vorherigen Abschnitts. Consumer A verwendet dabei das name Feld, Consumer B die Felder **firstName** und **lastName**. Der Contract mit Consumer A (Contract A1) enthält damit die Erwartung, dass die Antwort das Feld **name** zurückliefert. Der Contract mit Consumer B (Contract B) drückt die Erwartung aus, dass die Felder **firstName** und **lastName** zurückgeliefert werden. Erst wenn Consumer A angepasst wird und ebenfalls die neuen Felder verwendet, passt Consumer A seinen Contract entsprechend an (Contract A2).

Durch die Tests gegen die Vorversion der API, die weiterhin den Contract in der Version A1 enthält, kann der Provider seine API noch nicht direkt in der nächsten API-Version anpassen. Erst mit der übernächsten API-Version, kann der Provider das name Feld entfernen, da dann auch die Vorversion keine Contracts mehr enthält, die das Feld name erwarten. So verlängert sich auch die Übergangszeit und damit sind die Veröffentlichungen von Provider und Consumer noch besser entkoppelt.

Consumer-Driven-Contract-Testing hilft darüber hinaus, die API minimal zu halten. Wird die API erst dann entwickelt, wenn sie benötigt wird und deckt nur die Anforderungen der Contracts ab, so erhält die API nur Komponenten, die auch wirklich von Consumern benutzt werden. Außerdem kann die API so von Anfang an testgetrieben entwickelt werden.



API Evolution. (Abb. 4)

Sowohl testgetriebene Entwicklung als auch minimale Ansätze decken sich gut mit den heutigen Zielen schneller und zielorientierter Entwicklung. Ohne unnötige Komponenten wird die Codebasis kleiner und damit leichter zu warten. Durch die Tests werden Continuous-Integration und -Delivery unterstützt.

Spring-Cloud-Contract

Spring-Cloud-Contract[2] ist ein Projekt aus dem Spring-Cloud Umfeld, das Contract-Tests im Java und insbesondere im Spring Umfeld ermöglicht. Spring-Cloud-Contract unterstützt sowohl Contracts für http-Schnittstellen, als auch Contracts für Messaging mit Spring-Cloud-Stream. Contracts werden mit Spring-Cloud-Contract entweder in einer Groovy-DSL, oder in YML geschrieben. (Listing 1) zeigt ein Beispiel für einen Contract in der Groovy-DSL.

(Listing 1)

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/customers/1'
    }
    response {
        status 200
        headers {
            contentType(applicationJsonUtf8())
        }
        body([
                id : 1,
                name: 'Tom Ato'
        ])
    }
}
```

Dieser Contract spezifiziert im **request** Block, wie der Request des API-Consumers aussehen muss. Als HTTP-Methode wird **GET** festgelegt, der URL-Pfad wird als **customers/1** definiert. Die Antwort des API-Providers wird im **response** Block definiert. Der erwartete Statuscode der API ist 200 (OK), der Header muss das Feld **contentType** enthalten, das den Wert **application/json;charset=UTF-8** enthält. Für den Response-Body nimmt Spring-Cloud-Contract immer JSON als Format an. Die Groovy-Map stellt also ein JSON-Objekt mit den Feldern **id**, **name**, **street** und **city** dar. Das Spring-Cloud-Contract-Build-Plugin erzeugt aus solchen Contracts zum einen WireMock[3]-Stubs für die Consumer-Tests, als auch Testklassen für die API-Provider-Tests. Für die Provider-Tests gibt es die Möglichkeit MockMVC[4]-Tests zu generieren, die eine besonders einfache Integration in Spring Projekte bieten. Für Projekte ohne Spring bietet Spring-Cloud-Contract auch die Option die Testklassen mit einem JaxRS[5]-Client zu erzeugen. **(Listing 2)** zeigt einen Auszug aus dem erzeugten MockMVC-Test für den Contract aus **(Listing 1)**.

(Listing 2)

```
public class PaymentTest extends ContractBase {

@Test
public void validate_getCustomerPaymentInformation() throws
Exception {
    // when:
        ResponseOptions response = given().spec(request)
                .get("/customers/1");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).matches("ap-
plication/json;charset=UTF-8.*");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.
getBody().asString());
   }
}
```

Die erzeugten Testklassen erben von einer definierbaren Testklasse, hier **ContractBase**. Diese muss im eigenen Testcode vorhanden sein und ist dazu gedacht, die für den Test benötigten Abhängigkeiten bereitzustellen.

Für die Consumer-Tests wird WireMock genutzt. In einem Spring Projekt ist dank der **@AutoConfigureStubRunner** Annotation eine sehr einfache Integration in die eigenen Test möglich. Durch die Annotation wird für den Test ein lokaler WireMock-Stub-Runner gestartet. Gegen diesen können im Test die Requests des Consumers ausgeführt werden. Durch die Verwendung von WireMock als Stub-Runner ist die Verwendung der Stubs auch außerhalb des Java- / Spring-Umfelds einfach möglich. Start und Konfiguration des WireMock-Stub-Runners muss dann natürlich manuell erfolgen.

Weitere Dimensionen der API-Kompatibilität

Bei der Entwicklung von APIs ist es wichtig, sich der verschiedenen Dimensionen der API-Kompatibilität bewusst zu sein. Neben kompatibler Syntax müssen APIs auch in weiteren Dimensionen kompatibel zueinander sein. Ein Beispiel ist die Semantik der API. Eine Dauer kann zum Beispiel in unterschiedlichen Einheiten ausgedrückt werden. Ändert der Provider einer API beispielsweise die Einheit von Sekunden zu Millisekunden, so sind die Consumer ebenfalls inkompatibel, ohne dass dies durch Contract-Tests aufgedeckt werden kann.

Ein weiteres Problem sind kumulierte Updates. Contract-Tests können zwar sicherstellen, dass kompatible Zwischenversionen entstehen. Koordinierte Veröffentlichungen können aber dennoch notwendig werden, wenn die Zwischenversionen nicht in Produktion gebracht werden. Außerdem hat jeder Consumer einer API immer Erwartungen an den Zustand des Gesamtsystems. Wird zum Beispiel der Consumer einer API veröffentlicht, bevor der Provider veröffentlicht wird, so ist der Consumer nicht funktionstüchtig.

Contract-Testing ist ein hilfreiches Werkzeug für die Entwicklung und Evolution von APIs, jedoch liegt der Fokus auf Schemakompatibilität. Damit kann Contract-Testing syntaktische Kompatibilität einer API sicherstellen. Weitere Aspekte der API-Kompatibilität dürfen jedoch nicht vernachlässigt werden.

Fazit:

Insbesondere für interne und Partner-APIs ist Contract-Testing ein sehr mächtiges Werkzeug, um die Entwicklung und Evolution von APIs zu vereinfachen und Inkompatibilitäten zu verhindern. Consumer-Driven-Contract-Testing ermöglicht darüber hinaus die testgetriebene Entwicklung minimaler APIs.

Providern öffentlicher APIs hilft Contract-Testing ihre Schnittstellen abwärtskompatibel zu halten. Consumer öffentlicher APIs können durch Contract-Testing Inkompatibilitäten zur eigenen Verwendung automatisiert erkennen. Contract-Testing kann damit bei jeder Nutzung von APIs einen Mehrwert bieten. Mit Spring-Cloud-Contract steht im Java- und Spring-Umfeld eine sehr einfache Möglichkeit zur Verfügung um Contract-Testing in eigenen Projekten umzusetzen.

Quellen:

1 https://swagger.io/docs/specification/about/
2 https://spring.io/projects/spring-cloud-contract
3 http://wiremock.org/
4 https://spring.io/guides/gs/testing-web/
5 https://jersey.github.io/

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 · trivago

BRONZE PARTNER

viadee IT-Unternehmensberatung · TK Die Techniker · GEBIT Solutions

ORGANISATIONS-PARTNER

JAVAPRO · XDEVCON 2019 · XDEV

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

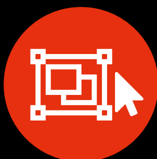
Container

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

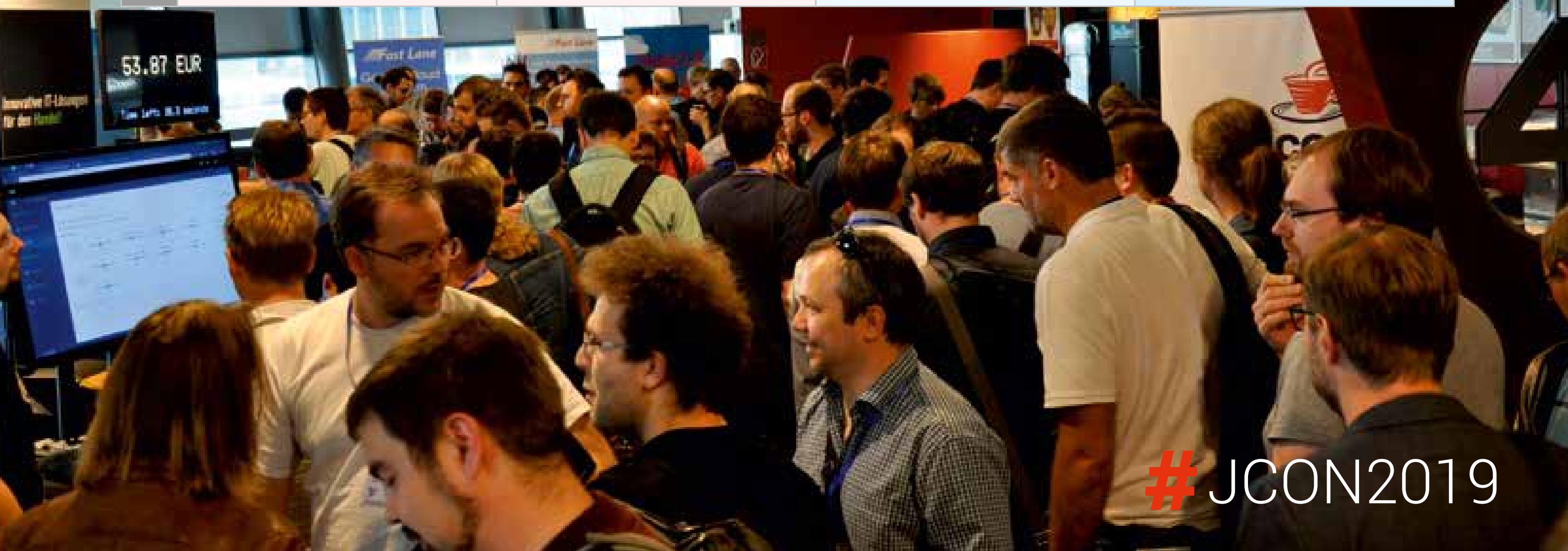
JCON 2019

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

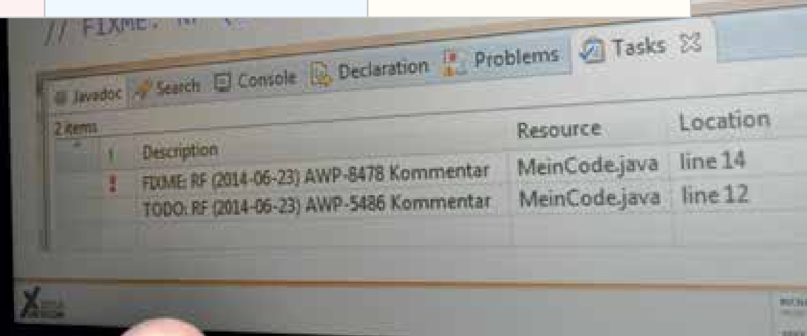
24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

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

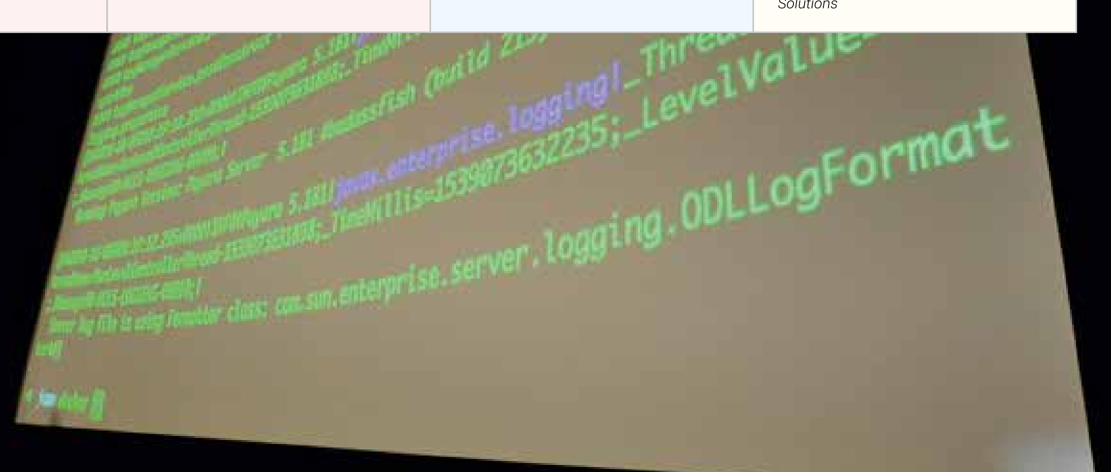
DIE GROSSE JAVA COMMUNITY KONFERENZ
24. - 26. September 2019 - UCI Kinowelt in Düsseldorf

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
Training Day 9:00 - 17:00 Uhr 10 hochkarätige Tages-Schulungen	JCON 2019: **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
	MICROSTREAM DAY 9:00 - 19:00 Uhr	XDEVCON 2019 - Konferenz für Rapid-Cross-Platform-Development: **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 #CoreJava #Streams #Collections

Collections effektiver durchsuchen mit Java-8-Streams

Wenn man bis Java 7 die Elemente einer Collection bearbeiten wollte, musste man sich mühsam mithilfe einer Schleife oder eines Iterators durch die Elemente klicken. In Java 8 sind Streams als neues Sprachelement eingeführt worden. Sie ermöglichen die Bearbeitung von Collections mit weniger und kompakterem Code, ohne dass die Lesbarkeit darunter leidet.

Collections durchlaufen bis Java 7

Wer früher in Java durch die Elemente einer Collection laufen und mit jedem Element eine Aktion durchführen wollte, musste eine For-Schleife verwenden. Eine Verkürzung stellt die For-Each-Schleife dar. Als Alternative kann auch ein Iterator-Objekt verwendet werden. **(Listing 1)** zeigt Beispiele für alle drei Möglichkeiten. Prinzipiell musste der Programmierer jedes Mal jede Menge repetitiven Boilerplate-Code schreiben. Mit den in Java 8 eingeführten Streams soll das Durchlaufen von Collections einfacher werden.

Autor:

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

https://thinkingaboutprogramming.blogspot.com/

(Listing 1)

```
for (int i = 0; i < days.size(); i++) {
    String day = days.get(i);
    if (day.length() > 7) {
        System.out.println(day);
    }
}

for (String day: days) {
    if (day.length() > 7) {
        System.out.println(day);
    }
}

Iterator<String> dayIterator = days.iterator();
while(dayIterator.hasNext()) {
    String day = dayIterator.next();
    if (day.length() > 7) {
        System.out.println(dayIterator.next());
    }
}
```

Was sind Streams in Java 8?

Die Java-8-Streams dürfen nicht mit den Klassen **`InputStream`** und **`OutputStream`** der I/O-Bibliothek von Java verwechselt werden. Es handelt sich dabei um zwei vollkommen unterschiedliche Konzepte. Die beiden Klassen **`InputStream`** und **`OutputStream`** ermöglichen das Bearbeiten von Dateien als einen kontinuierlichen Strom aus Zeichen.

Java-8-Streams dagegen repräsentieren eine Sequenz aus Elementen einer Collection, auf die aufeinander folgende Operationen durchgeführt werden können. Da die Stream-API als eine Fluent-API implementiert ist, ist es möglich, mehrere Methodenaufrufe aneinander zu hängen und die Verarbeitung mit einer terminalen Methode zu beenden. Dieses Konzept wird später noch genauer erläutert.

Das Kernstück der Stream-API von Java 8 bildet das Interface **`Stream`**, dessen Methoden meistens mit funktionalen Interfaces arbeiten und somit Lambda-Ausdrücke ermöglichen. Die Stream-API in Java 8 ist sehr umfangreich. In diesem Artikel soll die grundlegende Arbeitsweise mit Streams verdeutlicht werden, indem einige Methoden exemplarisch vorgestellt werden. Wer sich für weitere Methoden interessiert, sollte einen Blick auf die Dokumentation von Java 8 werfen.

Streams können nicht wiederverwendet werden, d.h. sobald eine terminale Methode aufgerufen wurde, wird der Stream beendet und geschlossen. Versucht man anschließend den Stream noch einmal zu benutzen, wirft Java eine **`IllegalStateException`**.

Mithilfe von forEach() durch eine Collection laufen

(Listing 2) zeigt das Beispiel aus **(Listing 1)** nun mit einem Stream. Die Methode **`forEach()`** erhält als Übergabe ein Predicate-Objekt, das durch einen Lambda-Ausdruck dargestellt werden kann. Dieser Lambda-Ausdruck gibt an, was mit jedem Element in der Collection getan werden soll. Damit werden die in **(Listing 1)** vorgestellten Schleifen ersetzt.

(Listing 2)

```
days.stream().forEach(day -> {
    if (day.length() > 7) {
         System.out.println(day);
    }
});
```

Elemente einer Collection filtern

Die If-Anweisung stört in dem Listing noch ein wenig. Aber auch dieses Problem kann mithilfe der Stream-API gelöst werden. Dazu wird die Methode **`filter()`** zur Verfügung gestellt. Sie ermöglicht es, die Elemente einer Collection nach einer Bedingung zu filtern. **(Listing 3)** zeigt das gleiche Programm noch einmal, nur wird dieses Mal die If-Anweisung durch die besagte Methode **`filter()`** ersetzt. Im Beispiel werden somit zuerst die Strings nach ihrer Länge aussortiert und auf die übrig gebliebenen Elemente wird dann die Methode **`forEach()`** angewendet.

(Listing 3)

```
days.stream()
    .filter(day -> day.length() > 7)
    .forEach(day -> System.out.println(day));
```

Erzeugen von Streams

Nachdem in den ersten Beispielen die Benutzung von Streams demonstriert wurde, stellt sich nun die Frage: Wie werden Streams jetzt eigentlich erzeugt? **(Listing 4)** zeigt zwei Möglichkeiten zur Erzeugung von Streams. Das erste Beispiel generiert einen Stream mithilfe der Methode **`of()`**. Diese Methode erhält als Übergabe eine kommaseparierte Liste von Elementen. Übergibt man **`of()`** ein List-Objekt, wird diese Liste als ein Element einer Collection interpretiert.

Das zweite Beispiel zeigt wie ein Stream erzeugt wird, wenn bereits eine fertige Collection zur Verfügung steht. Dann wird direkt aus der Collection mittels der Methode stream() ein Stream-Objekt erstellt. Diese beiden Methoden erzeugen endliche serielle Streams. Später werden noch die parallelen und die unendlichen Streams vorgestellt. Im Beispiel wird ein serieller Stream aus einer Menge von String-Objekten erzeugt. Natürlich können Streams auch beliebige andere Objekttypen enthalten.

(Listing 4)

```
Stream.of("AA", "BB", "CC", "DD").forEach(
    myString -> System.out.println(myString));

List<Integer> myList = Arrays.asList(1,2,3,4,5,6);
myList.stream().forEach(myNumber -> System.out.print-
ln(myNumber));
```

Spezialisierte Streams

Neben den Streams, die mit allen Objekttypen arbeiten können, gibt es auch noch spezialisierte Streams. Listing 5 zeigt den IntStream. Diese spezialisierten Streams bieten natürlich auch spezielle Methoden für den entsprechenden Datentyp. So ist im Beispiel die Methode **`average()`** zu sehen, welche den Durchschnitt der int-Werte in der Collection berechnet, ohne dass der Programmierer ein Schleifen-Konstrukt schreiben muss. Neben der hier vorgestellten Methode **`average()`** bietet IntStream z.B. auch die Methoden **`min()`**, **`max()`** und **`sum()`**. Für weitere spezialisierte Streams sei auf die Dokumentation verwiesen.

(Listing 5)

```
IntStream.of(1,2,3,4,5,6).
      average().
      ifPresent(System.out::println);
```

Streams mithilfe von map() verändern

Bisher wurden die Werte in einem Stream nur ausgegeben und nicht weiterverarbeitet. Da die Methode **forEach()** ein Consumer-Objekt als Übergabeparameter erhält, kann sie die Werte in der Collection nicht verändern. Mithilfe der Methode **map()** ist es jedoch möglich, die Elemente aus der Collection zu bearbeiten und einen Stream zu erzeugen, der die veränderten Werte enthält. (Listing 6) zeigt ein Beispiel. Nach dem Filtern der Strings mit mehr als sieben Zeichen werden die verbliebenen Strings in Großbuchstaben umgewandelt und anschließend ausgegeben. Wichtig zu beachten ist, dass die Methode **map()** einen Stream zurückgibt mit den Elementen, auf welche die Funktion angewendet wurde, d.h. die Elemente in der Collection werden nicht dauerhaft verändert, sondern nur im Rahmen der aktuellen Stream-Anweisung.

(Listing 6)

```
days.stream()
    .filter(day -> day.length() > 7)
    .map(day -> day.toUpperCase())
    .forEach(day -> System.out.println(day));
```

Verarbeitungsreihenfolge

Wenn man sich die bisherigen Beispiele ansieht und über die Verarbeitungsreihenfolge nachdenkt, könnte man vermuten, dass die Methoden horizontal abgearbeitet werden, d.h. zuerst werden alle Elemente der Collection durch die erste Methode gejagt und anschließend wird erst die zweite Methode aufgerufen. Wenn das Beispiel-Programm in (Listing 7) ausgeführt wird, ergibt sich jedoch eine interessante Erkenntnis: Die Elemente der Collection werden zwar nacheinander abgearbeitet, aber jedes Element wird durch die gesamte Pipeline geschickt, bevor das nächste Element verarbeitet wird. Die Ausgabe wird also für die ersten beiden Elemente wie folgt aussehen:

filter: A
forEach: A
filter: B
forEach: B

(Listing 7)

```
Stream.of("A", "B", "C", "D")
    .filter(myString -> {
        System.out.println("filter: " + myString);
  return true;
    })
    .forEach(myString -> System.out.println("forEach: " +
myString));
```

Diese Verarbeitung soll unter bestimmten Umständen Performance-Verbesserungen ermöglichen wie in dem in (Listing 8) angedeuteten Fall. Die Methode **anyMatch()** läuft solange, bis zum ersten Mal ein Element gefunden wird, auf das die Bedingung passt. Würde der Stream horizontal laufen, also mit einer Methode zuerst alle Elemente bearbeiten, müsste die Methode **map()** für alle Elemente durchgeführt werden, auch wenn die Methode **anyMatch()** nach dem zweiten Element schon die Verarbeitung beendet. Durch die vertikale Verarbeitung muss **map()** nur für die ersten zwei Elemente aufgerufen werden.

(Listing 8)

```
Stream.of("AA", "BB", "CC", "DD")
    .map(myString -> {
        System.out.println("map: " + myString);
        return myString.toLowerCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
  return s.contains("b");
    });
```

Es gibt allerdings Ausnahmen von dieser Verarbeitungsreihenfolge wie es z.B. bei der Methode **sorted()** der Fall ist, denn Sortieren ergibt nur Sinn, wenn die Sortierung auf der gesamten Datenmenge ausgeführt wird.

Streams sortieren

Um die Werte in einer Collection **zu** sortieren, kann die bereits im letzten Abschnitt erwähnte Methode **sorted()** eingesetzt werden. (Listing 9) zeigt zwei Beispiele. Im ersten Beispiel wird eine Collection aus Strings sortiert. Die Methode **sort()** weiß wie Strings sortiert werden sollen und muss somit keine weiteren Informationen mehr erhalten. Im zweiten Beispiel sollen die Elemente absteigend sortiert werden. Da dies nicht der Standardsortierung entspricht, wird der Methode ein Comparator-Objekt in Form eines Lambda-Ausdrucks mit der Sortierregel übergeben. Auf diese Art können neben Standardsortierungen auch beliebig komplexe Objekte nach beliebigen Regeln sortiert werden.

(Listing 9)

```
Stream.of("x", "a", "z", "d", "e", "b", "d").
       sorted().
       forEach(myString -> System.out.println(myString));

Stream.of("x", "a", "z", "d", "e", "b", "d").
       sorted((string1, string2) -> string2.compareTo(string1)).
       forEach(myString -> System.out.println(myString));
```

Fluent-Programmierung

In den Beispiel-Listings ist eine bereits erwähnte Besonderheit der Stream-API zu erkennen, die eine sehr kompakte Schreibweise erlaubt. Mithilfe der Fluent-Programmierung können die Methodenaufrufe einfach miteinander verkettet werden. Intermediäre Methoden dürfen beliebig oft aufgerufen werden. Sie geben ein Stream-Objekt zurück, das von der nächsten Methode weiterverarbeitet werden kann. Abgeschlossen wird die Stream-Verarbeitung mit einer terminalen Methode. Da diese den Stream beendet, darf sie nur ein einziges Mal in der Methodenverkettung auftreten – und zwar an deren Ende. Eine terminale Methoden hat entweder keine Rückgabe oder sie gibt ein Objekt zurück, das keinen Stream repräsentiert. Eine solche terminale Operation ist z.B. die Methode **forEach()**.

(Listing 6) zeigt zwei weitere Beispiele. Das erste Beispiel demonstriert die terminale Methode count(). Das zweite Beispiel zeigt einen längeren Stream-Ausdruck. Dabei werden zuerst zwei Filter-Kriterien angewandt, dann werden doppelte Werte aus dem Ergebnis entfernt. Anschließend wird die Menge der Ergebnisse auf vier limitiert, diese vier Werte werden in Kleinbuchstaben umgewandelt und zum Schluss werden sie auf dem Bildschirm ausgegeben. Vor allem im letzten Beispiel ist die hervorragende Lesbarkeit von Stream-Ausdrücken gut zu erkennen.

(Listing 10)

```
days.stream()
    .filter(day -> day.length() > 7)
    .count();

List<String> strings = Arrays.asList("ASDFGHASFD", "A", "B",
"AAAA", "A",     "AAAA", "XXXX", "XXXX", "BBBBB", "X", "DDDD",
"YYYYY");

strings.stream()
    .filter(string -> string.length() >= 3)
    .filter(string -> string.length() < 8)
    .distinct()
    .limit(4)
    .map(string -> string.toLowerCase())
    .forEach(string -> System.out.println(string));
```

Parallele Streams

Bisher wurden nur die sequentiellen Streams vorgestellt. Die Stream-API bietet aber auch die Möglichkeit der parallelen Verarbeitung. Dazu wird ein neuer Stream einfach statt mit der Methode **stream()** mit der Methode **parallelStream()** erzeugt. (Listing 11) zeigt ein einfaches Beispiel, bei dem die Längenbestimmung der einzelnen Strings gut parallelisierbar ist. Die Methode **reduce()** ist eine terminale Methode, welche die Werte im Stream nach einer bestimmten Regel zusammenfasst. Im Beispiel werden die Längen der Strings aufsummiert.

(Listing 11)

```
int charCount = strings.parallelStream()
    .filter(string -> string.length() >= 3)
    .filter(string -> string.length() < 8)

    .map(String::length)
    .reduce(0, (i,j) -> i+j);
```

Mithilfe des erweiterten Beispiels in (Listing 12) kann die parallele Verarbeitung besser verstanden werden. In jedem Schritt wird ausgegeben, welcher Thread die Verarbeitung durchführt.

(Listing 12)

```
int charCount = strings.parallelStream()
    .filter(string -> {
        System.out.format("filter1: %s [%s]\n",
      string, Thread.currentThread().getName());
  return string.length() >= 3;
    })
    .filter(string -> {
        System.out.format("filter2: %s [%s]\n",
            string, Thread.currentThread().getName());
  return string.length() < 8;
    })
    .map(string -> {
         System.out.format("map: %s [%s]\n",
             string, Thread.currentThread().getName());
         return string.length();
     })
     .reduce(0, (i,j) -> {
          System.out.format("reduce: [%s]\n",
               Thread.currentThread().getName());
          return i+j;
});z
```

Zusammenführen von Streams mit Collector

Die Ergebnisse eines Streams können in einem letzten Schritt zusammengeführt werden. Dazu dient das Interface **Collector**. 37 Methoden sind bereits fertig implementiert und können sofort eingesetzt werden. (Listing 13) zeigt mehrere Beispiele für Methoden des Interface **Collector**. In dem ersten Beispiel wird der Stream in eine Liste überführt. Im zweiten Beispiel werden die Längen der gefilterten Strings aufsummiert. Eine weitere Methode ist im dritten Beispiel zu sehen. Hier wird die Methode **groupingBy()** verwendet, um die im Stream enthaltenen Strings nach ihrer Länge zu gruppieren. In der Map sind am Ende also Listen mit Strings enthalten und die Längen der enthaltenen Strings sind die Schlüssel, unter denen die Listen in der Map gespeichert sind.

(Listing 13)

```
List<Integer> charCountList = strings.parallelStream()
    .filter(string -> string.length() >= 3)
    .filter(string -> string.length() < 8)
    .map(String::length)
    .collect(Collectors.toList());
```

```
int charCount = strings.parallelStream()
    .filter(string -> string.length() >= 3)
    .filter(string -> string.length() < 8)

    .collect(Collectors.summingInt(String::length));

Map<Integer, List<String>> groups = strings.stream()
    .filter(string -> string.length() >= 3)
    .filter(string -> string.length() < 8)
    .collect(Collectors.groupingBy(String::length));
```

Unendliche Streams

Neben den endlichen Streams können auch unendliche Streams erzeugt werden. Dazu stellt Java zwei Methoden zur Verfügung, die in (Listing 14) demonstriert werden. Die Methode generate() erzeugt einen unendlichen Stream, bei dem jedes Element durch das Supplier-Objekt generiert wird, das der Methode übergeben wird. Im Beispiel wird dazu eine Zufallszahl von der Math-Klasse angefordert. Der unendliche Stream, der im zweiten Beispiel in (Listing 14) erzeugt wird, enthält alle geraden Zahlen beginnend bei zwei. Die Methode **iterate()** erhält als Übergabe einen Startwert und eine Funktion in Form eines Lambda-Ausdrucks.

(Listing 14)

```
Stream<Double> randomDoubleStream = Stream.generate
(Math::random);
Stream<Integer> squareIntegerStream = Stream.iterate(2, i -> i +
2);
```

Fazit:

Die in Java 8 neu eingeführten Streams bieten mächtige Möglichkeiten zur Verarbeitung der Elemente in einer Collection. Mithilfe der Fluent-Programmierung und Lambda-Ausdrücken können sehr kompakte und gut lesbare Anweisungen formuliert werden, so dass man auf die immer gleichen Schleifen- oder Iterator-Konstrukte verzichten kann. Bei großen Datenmengen kann man mit parallelen Streams Performance-Verbesserungen erreichen.

Quellen:

Code-Beispiele: https://gitlab.com/colbertz/streams-beispiele
Michael Inden: Java 8 - Die Neuerungen: Lambdas, Streams, Date And Time API und JavaFX 8 im Überblick

GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION
JAVAPRO
Magazin für professionelle Java Entwicklung in der Praxis www.javapro.io

... oder im Web unter....
www.javapro.io

#JAVAPRO #CoreJava #Annotations

Deep-Dive into Annotations – Teil 3

Java-Annotationen sind ein mächtiges Sprachmerkmal, deren Interna allgemein nicht sonderlich bekannt sind. In Teil 3 unserer dreiteiligen Serie geht es um die Auswertung eigener Annotationen zur Laufzeit.

In Teil 1 unserer Serie wurden diverse Verwendungszwecke für Annotationen aufgezeigt und die fünf Java-SE-Standardannotationen detailliert diskutiert. Teil 2 führte aus, wie eigene Annotationstypen programmiert werden können. Dabei wurden insbesondere die zahlreichen Optionen bei der Konfiguration sowie diverse Besonderheiten betrachtet. Die Syntax der Definition eigener Annotationen ist wahrlich sehr speziell und passt so gar nicht ins Bild der ansonsten eigentlich schönen und konsistenten Java-Syntax.

Im hier vorliegenden dritten und letzten Teil kommen wir zur Krönung des Einsatzes von Annotationen. Die Deklaration und das Anbringen von Annotationen macht nur Sinn, wenn sich

Autor:

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Software-Entwicklung, -Schulung und -Beratung v. a. für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.

christian.heitzmann@simplexacode.ch
https://www.simplexacode.ch

diese auch wieder auslesen lassen. Erinnern wir uns, dass es sich gemäss offizieller Definition bei Annotationen lediglich um Marker handelt, die Informationen mit einem Programmkonstrukt verknüpfen, aber keinen Effekt zur Laufzeit haben. Annotationen können also von sich selbst aus nichts „machen". Daraus ergeben sich drei Schlussfolgerungen:

1. Zum Auslesen von Annotationen liegt der Einsatz von Reflection auf der Hand.
2. Es lassen sich einzig die Existenz bzw. Nichtexistenz von Annotationen feststellen und die in den Annotationen enthaltenen Parameter auslesen. Mit anderen Worten, Annotationen lassen sich nicht ausführen.
3. Soll das Vorhandensein einer Annotation mit irgendwelchen Aktionen verknüpft werden, so hat dies von aussen zu geschehen. Sämtliche Programmlogik muss in normalem Java-Code in diesen „Erkennungsroutinen" hinterlegt sein.

Die Optionen zur Definition eigener Annotationen sind zahlreich. Dementsprechend zahlreich sind auch die verschiedenen Arten, wie sich Annotationen in eigenen Programmen auslesen lassen. Dazu werden wir nachfolgend alle typischen Fälle betrachten, wie sich welche Art von Annotationen auslesen lässt.

Deklaration verschiedener Annotationstypen

Um die unterschiedlichen Arten zum Auslesen von Annotationen aufzeigen zu können, benötigen wir zuerst verschiedene eigene Annotationstypen. Folgende Annotationen stellen eine Zusammenfassung der in Teil 2 erläuterten Optionen dar:

(Listing 1)

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface NoValueAnnotation {}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface SingleValueAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MultiValueAnnotation {
  public int[] value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(RepeatableValueAnnotationContainer.class)
@interface RepeatableValueAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RepeatableValueAnnotationContainer {
  public RepeatableValueAnnotation[] value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface SingleNamedValueAnnotation {
  public String name();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MultiNamedValueAnnotation {
  public String first();
  public String second();
  public String third();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface EnumValueAnnotation {
  public static enum GreekLetter {
    ALPHA, BETA, GAMMA
  }
  public GreekLetter value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface NonInheritedAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@interface InheritedAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
@interface ParameterAnnotation {}
```

- Um zur Laufzeit Zugriff auf die Annotationen zu haben, müssen diese allesamt mit **`@Retention(RetentionPolicy.RUNTIME)`** versehen werden. **`RetentionPolicy.SOURCE`** wird wohl nur für eigene Compiler oder Tools, interessant sein, die direkt auf Source-Code-Ebene arbeiten, zum Beispiel so etwas wie Javadoc. **`RetentionPolicy.CLASS`** funktioniert entsprechend nur für eigene Tools, die direkt mit dem Bytecode der Klassen arbeiten, zum Beispiel Code-Obfuscatoren oder allenfalls eigene Klassenlader. Das Gros der Anwendungsfälle wird das Auslesen der Annotationen zur Laufzeit ausmachen, und dafür kommt nur **`RetentionPolicy.RUNTIME`** in Frage.

- Aus Gründen der Einfachheit wurden die meisten Annotationstypen mit **`@Target(ElementType.METHOD)`** versehen, lassen sich also nur zur Annotation von Methoden

verwenden. Dies ist keinesfalls einschränkend zu verstehen. Das Annotieren zum Beispiel von Klassen (**ElementType.TYPE**), Attributen (**ElementType.FIELD**) oder Methoden-Parametern (**ElementType.PARAMETER**) wäre in den meisten Fällen analog möglich, sofern die passenden @Target konfiguriert und die entsprechenden Reflection-Aufrufe vorgenommen werden. Um den Rahmen nicht zu sprengen, wurde darauf in diesem Artikel verzichtet.

- **NoValueAnnotation** ist eine leere Marker-Annotation. **SingleValueAnnotation** ist eine solche mit einem einzigen Wert, in diesem Fall einem Integer mit dem Standardnamen **value**. Wir erinnern uns: Der Name **value** muss bei der Anwendung einer Annotation nicht explizit genannt werden, alle anderen Namen hingegen schon.

- **MultiValueAnnotation** ist eine Annotation mit mehreren Werten gleichen Typs und dem Standardnamen value, das heisst, auch hier können bei der Anwendung die Werte einfach in geschweiften Klammern mitgegeben werden, ohne dass **value** explizit dazugeschrieben werden muss. **RepeatableValueAnnotation** ist ebenfalls in der Lage, mehrere Werte gleichen Typs aufzunehmen. Im Gegensatz zu **MultiValueAnnotation** muss dies aber mit mehreren Einzelaufrufen geschehen. Um dies zu erreichen, wurde die Annotation wiederum mit **@Repeatable** annotiert, einer neuen Meta-Annotation seit Java 8. **@Repeatable** funktioniert nur in Verbindung mit einem Container, der hier **@RepeatableValueAnnotationContainer** genannt wurde und nicht zum direkten Einsatz als Annotation gedacht ist, sondern seinen Dienst nur hinter den Kulissen verrichten soll.

- **SingleNamedValueAnnotation** und **MultiNamedValueAnnotation** sind Annotationen mit einem respektive mehreren Werten, die jedoch nicht standardmäßig value, sondern individuell benannt sind (**name** respektive **first**, **second** und **third**).

- **EnumValueAnnotation** zeigt die Deklaration und später den Einsatz einer Annotation, die ihren eigenen Aufzählungstyp enthält (in diesem Beispiel **GreekLetter**).

- **NonInheritedAnnotation** und **InheritedAnnotation** unterscheiden sich einzig durch die **@Inherited** Meta-Annotation. Zur Erinnerung: **@Inherited** funktioniert nur bei Klassen (daher hier auch das **@Target(ElementType.TYPE)**), nicht bei Interfaces und erst recht nicht bei Methoden, Attributen oder dergleichen.

- **ParameterAnnotation** annotiert, wie der Name bereits verrät, Methodenparameter, um abschliessend aufzeigen zu können, wie zum Beispiel die Argumente einer Methode abgefragt werden können.

Anbringung der Annotationen

Das Anbringen der im vorherigen Abschnitt beschriebenen Annotationen erklärt sich nun von selbst:

(Listing 2)

```
@NonInheritedAnnotation(42)
@InheritedAnnotation(21)
class Super {

  public void methodWithNoAnnotation() {}

  @NoValueAnnotation
  public void methodWithNoValueAnnotation() {}

  @SingleValueAnnotation(42)
  public void methodWithSingleValueAnnotation() {}

  @MultiValueAnnotation( { 1, 23, 456, 7890 } )
  public void methodWithMultiValueAnnotation() {}

  @RepeatableValueAnnotation(1)
  @RepeatableValueAnnotation(23)
  @RepeatableValueAnnotation(456)
  @RepeatableValueAnnotation(7890)
  public void methodWithRepeatableValueAnnotation() {}

  @SingleNamedValueAnnotation(name = "forty-two")
  public void methodWithNamedSingleValueAnnotation() {}

  @MultiNamedValueAnnotation(first = "one",
                             second = "two",
                             third = "three")
  public void methodWithMultiNamedValueAnnotation() {}

  @EnumValueAnnotation(EnumValueAnnotation.GreekLetter.GAMMA)
  public void methodWithEnumValueAnnotation() {}

  public void methodWithAnnotatedParameter
      (Object parameterWithNoAnnotation,
       @ParameterAnnotation Object parameterWithAnnotation) {}
}

final class Sub extends Super {}
```

Praktisch alle Annotationen werden auf oder in der Klasse **Super** angewendet. Ganz unten im Quellcode findet sich noch eine Klasse **Sub**, die von **Super** erbt und später das Verhalten der **@Inherited** Klassenannotation demonstrieren soll. Im folgenden Abschnitt geht es darum, die verschiedenen Arten in eigenen Programmen abzufragen.

Annotation Processor

Wir möchten nun einen sogenannten Annotationsprozessor schreiben, also ein Programm, welches unsere eigenen Annotationen ausliest und ausgibt, die wir an den analog benannten Methoden, an der Klasse und in einem Fall an den Methodenparametern angebracht haben. Enthalten die Annotationen Werte, so werden diese ebenfalls auf der Konsole ausgegeben. Die Hauptklasse mit ihrer **main**-Methode sieht wie folgt aus:

(Listing 3)

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public final class AnnotationProcessor {

  public static void main(String[] args) {
    Class<Super> superClass = Super.class;
    Class<Sub> subClass = Sub.class;
    AnnotationProcessor ap = new AnnotationProcessor();
    ap.printMethodsWithNoAnnotation(superClass);
    ap.printMethodsWithNoValueAnnotation(superClass);
    ap.printMethodsWithSingleValueAnnotation(superClass);
    /* [...] Other method calls omitted. */
```

Methoden ohne Annotation

(Listing 4)

```
  void printMethodsWithNoAnnotation(Class<?> clazz) {
    System.out.println("Methods With No Annotation");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      Annotation[] annotations
          = currentMethod.getAnnotations();
      if (annotations.length == 0) {
        String methodName = currentMethod.getName();
        System.out.format("  %s%n", methodName);
      }
    }
  }
```

Zur Vereinfachung haben wir bislang meistens nur Methoden annotiert. Im ersten Beispiel geht es darum, alle Methoden auszugeben, die über keine Annotation verfügen. Start einer solchen Suche ist immer das Klassenobjekt, welches man durch **`.class`** oder **`getClass`** erhält. Hat man ein solches Klassenobjekt, in (Listing 4) als **`clazz`** bezeichnet, erhält man via **`getDeclared Methods`** ein Array von **`Methods`**. Im Gegensatz zu **`getMethods`** gibt **`getDeclaredMethods`** zum einen auch die privaten Methoden einer Klasse zurück, zum anderen sind alle Methoden der Oberklasse(n) (inklusive **`Object`**) ausdrücklich nicht Bestandteil dieser Auflistung. Analog erhält man mit **`Method#get Annotations`** ein Array aller Annotationen einer Methode. Ist dieses Array leer (also **`length == 0`**), dann folgt daraus, dass diese Methode nicht annotiert wurde. Sie ist damit genau das, was im hier vorliegenden Fall gesucht wird und ihr Methodenname (**`Method#getName`**) wird auf der Konsole ausgegeben.

Methoden mit Annotation ohne Wert

(Listing 5)

```
  void printMethodsWithNoValueAnnotation(Class<?> clazz) {
    System.out.println("Methods With @NoValueAnnotation");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      NoValueAnnotation annotation
          = currentMethod.getAnnotation
          (NoValueAnnotation.class);
      if (annotation != null) {
        String methodName = currentMethod.getName();
        System.out.format("  %s%n", methodName);
      }
    }
  }
```

Wird nach einer ganz bestimmten Annotation gesucht, so bietet sich **`Method#getAnnotation(Class<T extends Annotation>)`** an. Als Parameter wird der gesuchte Annotationstyp mitgegeben. Wurde die entsprechende Annotation angebracht, so gibt der Aufruf ebendiese Annotation zurück; wenn nicht, dann ist der Rückgabewert **`null`**. Eine anschliessende Überprüfung auf **`null`** ist also immer obligatorisch, wenn man keine Laufzeitfehler riskieren will. In (Listing 5) wird gezielt nach der **`@NoValueAnnotation`** gesucht. Wird sie gefunden, dann wird der Name der Methode, an der sie angebracht wurde, auf der Konsole ausgegeben.

Methoden mit Annotation mit nur einem Wert

(Listing 6)

```
  void printMethodsWithSingleValueAnnotation(Class<?> clazz) {
    System.out.println
        ("Methods With @SingleValueAnnotation(int)");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      String methodName = currentMethod.getName();
      SingleValueAnnotation annotation
          = currentMethod.getAnnotation
          (SingleValueAnnotation.class);
      if (annotation != null) {
        int value = annotation.value();
        System.out.format("  %s | value=%d%n",
                          methodName, Integer.valueOf(value));
      }
    }
  }
```

Das Vorgehen zum Detektieren der gesuchten Annotation (hier **`@SingleValueAnnotation`**) ist aus dem vorherigen Code-Beispiel bekannt und wird sich von nun an auch nicht mehr ändern. Neu ist in (Listing 6), wie sich der Wert **`value`** der Annotation abfragen lässt. Da bereits der konkrete Annotationstyp vorliegt, geschieht dies ganz einfach mit dem Aufruf von **`.value()`**. Im Beispiel wird der so ausgelesene Integer-Wert zusammen mit dem Methodennamen auf der Konsole ausgegeben.

Methoden mit Annotation mit mehreren Werten

(Listing 7)

```
  void printMethodsWithMultiValueAnnotation(Class<?> clazz) {
    System.out.println
        ("Methods With @MultiValueAnnotation(int...)");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      String methodName = currentMethod.getName();
      MultiValueAnnotation annotation
```

```
            = currentMethod.getAnnotation
            (MultiValueAnnotation.class);
        if (annotation != null) {
          int[] values = annotation.value();
          System.out.format("  %s | values=%s%n",
                            methodName, Arrays.toString(values));
        }
      }
    }
```

Bei Annotationen, die mehrere Werte gleichen Typs aufnehmen können, liefert die Methode **`.value()`** ein Array statt eines einzelnen Wertes zurück. Der anschließende Zugriff auf die Werte des Arrays ist trivial.

Methoden mit wiederholter Annotation mit Wert

(Listing 8)

```
void printMethodsWithRepeatableValueAnnotation
    (Class<?> clazz) {
  System.out.println
      ("Methods With @RepeatableValueAnnotation(int)*");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    RepeatableValueAnnotation[] annotations
        = currentMethod.getAnnotationsByType
        (RepeatableValueAnnotation.class);
    for (RepeatableValueAnnotation currentAnnotation
         : annotations) {
      int value = currentAnnotation.value();
      System.out.format("  %s | value=%d%n",
                        methodName, Integer.valueOf(value));
    }
  }
}
```

Zwischen Annotationen mit mehreren Werten und **`@Repeatable`** Annotationen mit jeweils einem Wert, gibt es nicht nur beim Anbringen, sondern auch beim Auslesen leichte Unterschiede. Da nun mehrere Annotationen eines gesuchten gleichen Typs vorkommen können, funktioniert **`Method#getAnnotation`** nicht. Stattdessen muss **`Method#getAnnotationsByType(Class<T extends Annotation>)`** aufgerufen werden, welches nun ein Array von Annotationen des gesuchten Typs zurückgibt, sofern vorhanden. Im schlimmsten Fall ist dieses Array einfach leer; eine Überprüfung auf **`null`** kann entfallen. Um die Werte auszulesen, kann einfach dieses Array iteriert werden. Die darin enthaltenen Annotationen verhalten sich gleich wie die Annotationen mit nur einem Wert.

Methoden mit Annotation mit nur einem benannten Wert

(Listing 9)

```
void printMethodsWithSingleNamedValueAnnotation
    (Class<?> clazz) {
  System.out.println
      ("Methods With @SingleNamedValueAnnotation(String)");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    SingleNamedValueAnnotation annotation
        = currentMethod.getAnnotation
        (SingleNamedValueAnnotation.class);
    if (annotation != null) {
      String name = annotation.name();
      System.out.format("  %s | name=%s%n", methodName, name);
    }
  }
}
```

Wir haben zuvor bereits auf den Parameter **`value`** zugegriffen. Sind die Werte einer Annotation anders benannt, so erfolgt ihr Zugriff analog anhand dieses Namens. Vorangehend wurde der Wert einer Annotation **`name`** genannt. Der Zugriff darauf erfolgt also einfach über **`.name()`**.

Spätestens an dieser Stelle wird nun klar, wieso Annotationen ähnlich wie Interfaces implementiert werden und wieso die „Attribute", sprich die Werte oder Parameter einer solchen Annotation immer als Anhang von Zugriffsmethoden, also mit einem nachfolgenden Klammerpaar deklariert werden müssen. Zur Laufzeit stellt diese Annotation nach außen hin nichts anderes als ein Interface mit ebendiesen Zugriffsmethoden dar.

Methoden mit Annotation mit mehreren benannten Werten

(Listing 10)

```
void printMethodsWithMultiNamedValueAnnotation
      (Class<?> clazz) {
    System.out.println
        ("Methods With @MultiNamedValueAnnotation(String)");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      String methodName = currentMethod.getName();
      MultiNamedValueAnnotation annotation
          = currentMethod.getAnnotation
          (MultiNamedValueAnnotation.class);
      if (annotation != null) {
        String first = annotation.first();
        String second = annotation.second();
        String third = annotation.third();
        System.out.format("  %s |"
                          + " first=%s, second=%s, third=%s%n",
                          methodName, first, second, third);
      }
    }
  }
```

Egal ob es sich um eine Annotation mit einem oder mehreren benannten Werten handelt, der Zugriff darauf erfolgt einheitlich und selbstklärend über den Namen des Wertes als Methodenaufruf, in (Listing 10) mit **`.first()`**, **`.second()`** und **`.third()`**.

Methoden mit Annotation mit Wert als Aufzählungstyp

(Listing 11)

```
void printMethodsWithEnumValueAnnotation(Class<?> clazz) {
  System.out.println
      ("Methods With @EnumValueAnnotation(GreekLetter)");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    EnumValueAnnotation annotation
        = currentMethod.getAnnotation
        (EnumValueAnnotation.class);
    if (annotation != null) {
      EnumValueAnnotation.GreekLetter value
          = annotation.value();
      System.out.format("%s | name=%s%n", methodName, value);
    }
  }
}
```

Auch der Zugriff auf Werte einer Annotation, die einen Aufzählungstyp darstellen, ist nicht besonders schwer. Wenn der Aufzählungstyp wie im vorliegenden Code-Beispiel direkt in der Annotation definiert ist, so erhält man auch nur über diese Annotation Zugriff auf diesen Typ. In (Listing 11) ist dies gut ersichtlich an `EnumValueAnnotation.GreekLetter`.

Klassen mit @Inherited-Annotation

(Listing 12)

```
void printClassesWithNonInheritedAnnotation
    (Class<?>... classes) {
  System.out.println("Classes With @NonInheritedAnnotation");
  for (Class<?> currentClass : classes) {
    String className = currentClass.getSimpleName();
    NonInheritedAnnotation annotation
        = currentClass.getAnnotation
        (NonInheritedAnnotation.class);
    if (annotation != null) {
      int value = annotation.value();
      System.out.format("  %s | value=%d%n",
                        className, Integer.valueOf(value));
    }
  }
}

void printClassesWithInheritedAnnotation
    (Class<?>... classes) {
  System.out.println("Classes With @InheritedAnnotation");
  for (Class<?> currentClass : classes) {
    String className = currentClass.getSimpleName();
    InheritedAnnotation annotation
        = currentClass.getAnnotation
        (InheritedAnnotation.class);
    if (annotation != null) {
      int value = annotation.value();
      System.out.format("  %s | value=%d%n",
                        className, Integer.valueOf(value));
    }
  }
}
```

Wechseln wir von annotierten Methoden zu annotierten Klassen. Die beiden Codebeispiele in (Listing 12), die sich nur am Annotationstyp unterscheiden, sollen demonstrieren, welche Auswirkung die **`@Inherited`** Meta-Annotation auf die Vererbung von Annotationen hat. Die Ausgabemethoden werden jeweils sowohl mit der Klasse **`Super`** als auch mit der Unterklasse **`Sub`** aufgerufen. Wie zu erwarten, ist die InheritedAnnotation sowohl in **`Super`** als auch in **`Sub`** zugreifbar, die **`NonInheritedAnnotation`** hingegen nur in **`Super`**.

Methodenparameter mit Annotation

(Listing 13)

```
void printMethodParametersWithAnnotation(Class<?> clazz) {
  System.out.println
      ("Method Parameters With @ParameterAnnotation");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    Parameter[] parameters = currentMethod.getParameters();
    for (Parameter currentParameter : parameters) {
      ParameterAnnotation annotation
          = currentParameter.getAnnotation
          (ParameterAnnotation.class);
      if (annotation != null) {
        String parameterName = currentParameter.getName();
        System.out.format("  %s.%s%n",
                          methodName, parameterName);
      }
    }
  }
}
```

Schauen wir uns zum Abschluss noch an, wie annotierte Methodenparameter ausgelesen werden können. Mit **`Method#getParameters`** lässt man sich via Reflection ein **`Parameter`** Array geben, welches anschließend iteriert wird. Selbsterklärend erhält man via **`Parameter#getAnnotation(Class<T extends Annotation>)`** Zugriff auf eine gesuchte Parameter-Annotation, sofern vorhanden. Das Auslesen ihrer Werte erfolgt auf die bekannte Art.

Fazit:

Die Literatur zum Thema Annotationen ist recht begrenzt und die Syntax sehr gewöhnungsbedürftig. Dennoch zeigt sich auf beeindruckende Weise, wie mächtig und flexibel Annotationen in Java eingesetzt werden können. Unserer 3-teiligen Serie soll dazu beitragen, anfängliche Berührungsängste gegenüber Annotationen zu beseitigen und ihren sinnvollen Einsatz in Softwareprojekten fördern.

#JAVAPRO #Frameworks #SpringBoot #Migration

Spring-Boot im Legacy-Kontext

Für die Weiterentwicklung von Legacy-Anwendungen sind oftmals zusätzliche Werkzeuge und Frameworks erforderlich. Der Artikel zeigt auf, wie sich in diesem Zusammenhang Spring-Boot sinnvoll einsetzen lässt.

Spring[1] [2] ist ein nützliches Open-Source-Framework für die Java-Plattform. Das Kernziel des Projekts besteht darin, die Java-/Java-EE-Entwicklung durch die Entkopplung von Applikationskomponenten zu vereinfachen. Die zentralen Mechanismen sind dabei Dependency-Injection[3] und Aspektorientierte Programmierung[4]. Ein wesentlicher Baustein für die Implementierung dieser Konzepte ist die sogenannte Bean, ein Objekt, das von Spring instanziiert, zusammengestellt und verwaltet wird[5]. Die Eigenschaften der Beans können sowohl in XML-Dateien, als auch durch Annotationen im Code, gekennzeichnet durch @ Tags, festgelegt werden. Die erste Variante, also die Nutzung von XML-Dateien, hat den Vorteil, dass die Anwendung dadurch auch zur Laufzeit flexibel konfigurierbar ist. Sie hat aber dafür den Nachteil, dass Definitionen außerhalb des gewöhnlichen Java-Codes stehen, was zum Beispiel das Debuggen erschwert. Zudem werden die XML-Konfigurationen auch bei einer geringen Anzahl von Bean-Definitionen recht schnell komplex und unübersichtlich. Deswegen ist es i.d.R. besser, den annotationsbasierten Ansatz zu verwenden.

Spring-Boot greift den annotationsbasierten Ansatz auf und stellt weitere Werkzeuge zur Vereinfachung der Konfiguration, bzw. Anwendungsüberwachung bereit. Konzeptionell entspricht

Autor:

Die Autoren sind bei Avision in Oberhaching bei München angestellt, einem Spezialisten für Software Revival. Beide sind promovierte Physiker.

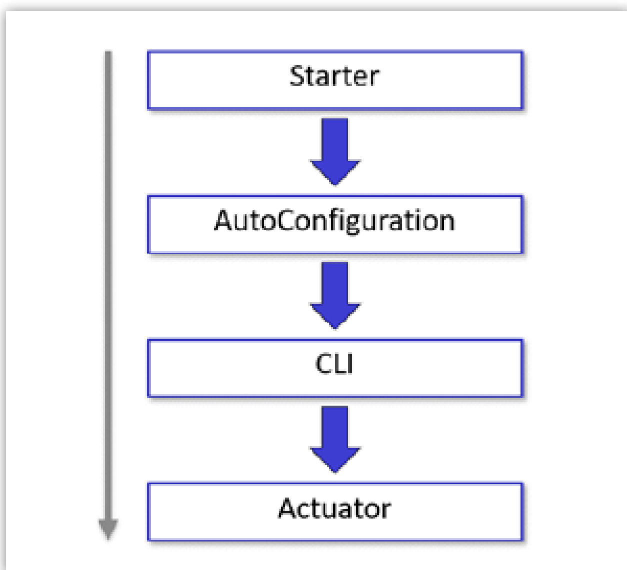
Dr. Gernot Pfanner ist in der Softwareentwicklung aktiv.

Dr. Rainer Brückner leitet das Test- und Quality-Management-Team.

https://www.avision-it.de

Spring-Boot dem Spring-Framework inklusive eingebettetem HTTP-Server (Tomcat, Jetty) und einer vereinfachten Bean-Konfiguration (in Form von Annotationen)[6]. Die Bean-Konfiguration via XML wird nur zwecks der Abwärtskompatibilität unterstützt. Das ist aber gerade für Legacy-Anwendungen nützlich, weil dadurch die in den XML-Dateien definierten Elemente nicht zwingendermaßen in Java-Code umgeschrieben werden müssen. Konkret besteht SpringBoot aus den folgenden Komponenten **(Abb. 1)**:

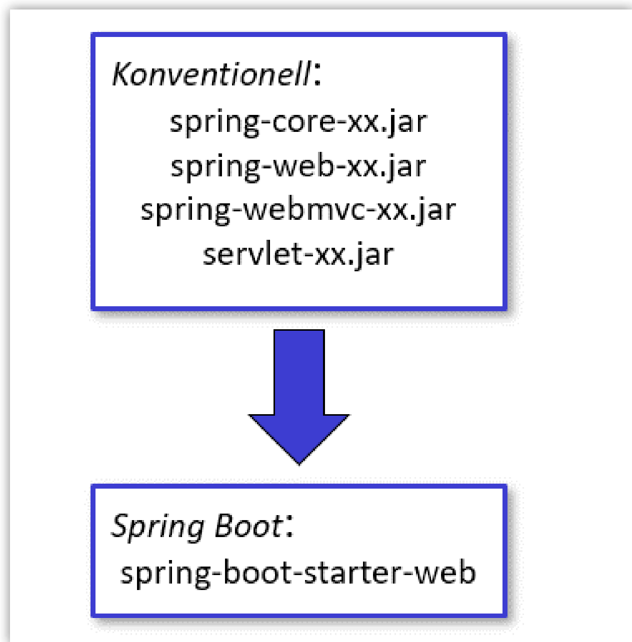
- Starter: Unterstützt beim Zusammenstellen von Library-Abhängigkeiten
- AutoConfiguration: Automatische Konfiguration der Anwendung
- Command-Line-Interface (CLI): Kommandozeile zum Laufen und Testen der Anwendung
- Actuator: Aufrufbare Laufzeitinformationen (Metriken, Health, usw.)



Die Komponenten in einer Anordnung gemäß dem Einsatz-Zeitpunkt im Entwicklungsprozess. (Abb. 1)

Die Idee von Spring-Boot-Starter besteht darin, dass Spring-Boot für typische Anwendungskomponenten (z.B. web, jdbc, ldap, logging, test) die wesentlichen Java-Libraries zusammenstellt und auch die Abstimmung der Library-Versionen übernimmt[7 8]. In der Praxis führt das zu folgender Vorgehensweise: Soll eine Spring-Boot-Anwendung erstellt werden, wird zunächst einmal die Spring-Initialize-Webseite aufgerufen[9]. Dort lässt sich eine Maven-/Gradle-Projektdatei anhand der benötigten Abhängigkeiten, wie z.B. Web, Security, JPA etc., erzeugen. Das resultierende Build-Konfigurations-Template ist lauffähig und stellt das wesentliche Gerüst für die Anwendung dar, von dem aus die konkreten Anforderungen implementiert werden können.

Der Vorteil dieses Vorgehens lässt sich anhand eines Beispiels illustrieren: Wird eine Web-Applikation mit Tomcat auf konventionelle Art und Weise, also ohne Sprint-Boot entwickelt, sind mehrere Libraries einzubinden **(Abb. 2)**. In einem Spring-Boot-Projekt reicht es dagegen aus, die Spring-Boot-Starter-Abhängigkeit einzubinden. Das Versionsmanagement der damit verbundenen Libraries wird komplett von Spring-Boot übernommen. Damit ist auch automatisch sichergestellt, dass die Versionen der einzelnen Libraries zueinander passen. Übergreifend wird dadurch erreicht, dass auf der Komponenten-Ebene Spring-Boot-Projekte, z.B. für Webservices, stets die gleichen Voraussetzungen haben – was letztlich zur Standardisierung von Java-Projekten beiträgt.



In Spring-Boot werden mehrere Libraries zu einer einzigen Abhängigkeit zusammengefasst. (Abb. 2)

AutoConfiguration hat das Ziel, die (Bean-) Konfiguration zu minimieren. Dazu analysiert die Anwendung die Library-Abhängigkeiten und konfiguriert die entsprechenden Komponenten, soweit diese nicht explizit manuell konfiguriert wurden. Findet zum Beispiel die AutoConfiguration-Komponente eine HSQLDB im Classpath, wird automatisch eine In-Memory-Datenbank erzeugt, falls keine dementsprechende Bean-Konfiguration vorhanden ist[10]. Für Spring-Boot-Starter-Web-Projekte werden automatisch Standardendpunkte generiert, die im Browser aufrufbar sind, z.B. eine Error-Page[11]. Den Einsprungspunkt für die AutoConfiguration kennzeichnet man im Code, indem an eine Klasse die @SpringBootApplication Annotation anfügt wird. Diese steht formal für folgende Spring-Annotationen[12]:

```
@SpringBootApplication =
@Configuration + @ComponentScan +
@EnableAutoConfiguration
```

- `@Configuration`: kennzeichnet Konfigurations-Klassen für Beans
- `@ComponentScan`: suche nach Konfigurationen und Beans im gleichen Package (und darunter)
- `@EnableAutoConfiguration`: fügt Beans gemäß den Classpath-Abhängigkeiten automatisch hinzu

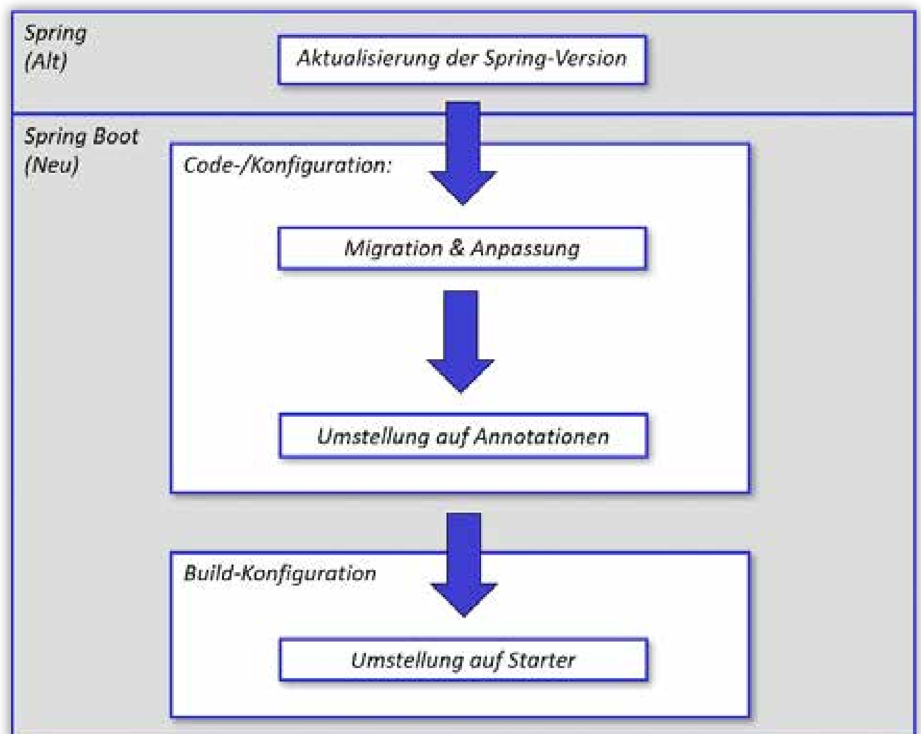
Daraus ergeben sich auch Anforderungen an die Projektstruktur: Um den größten Nutzen aus Spring-Boot zu erhalten, sollte der Einsprungspunkt möglichst auf der Root-Ebene der Paketstruktur definiert werden.

Die letzte wichtige Spring-Boot-Komponente ist der Actuator. Dieser liefert Laufzeitinformationen über die Anwendung und kann somit für das Monitoring von großem Nutzen sein. Das Prinzip ist folgendes: Im Browser oder JMX-Console wird ein eindeutiger Endpunkt aufgerufen, z. B. **`localhost:8080/actuator/health`**. Die damit zusammenhängenden Informationen werden von der Anwendung zusammengestellt und dann zurückgegeben. Daher werden beispielsweise im Browser als JSON-Objekt für **`health`** Informationen zum Health-Status angezeigt. Spring-Boot verfügt über einige Standard-Endpunkte[13], die je nach Einsatzszenario nützlich oder schädlich sein können. Bei einem öffentlichen Web-Service beispielsweise ist vermutlich nicht erwünscht, dass sich dieser über **`shutdown`** herunterfahren lässt. Aus diesem Grund ist dieser konkrete Endpunkt auch default-mäßig deaktiviert.

Spring-Boot im Legacy-Kontext

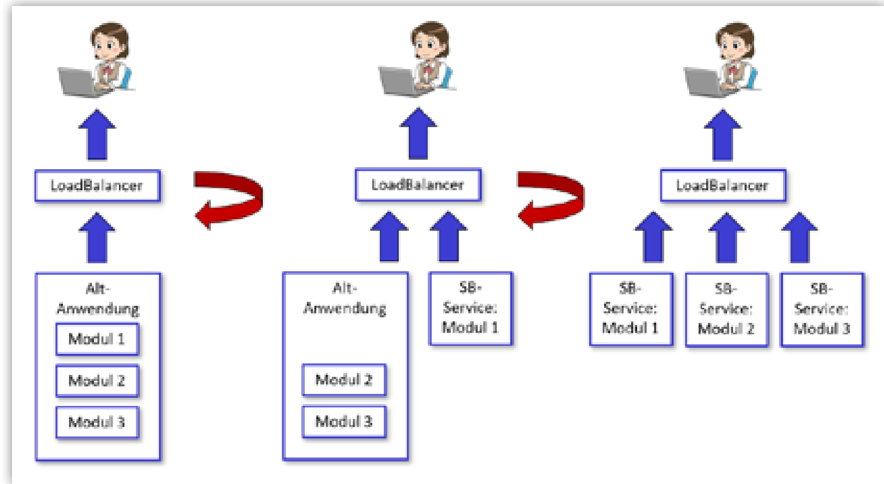
Kann Spring-Boot auch für Legacy-Anwendungen eingesetzt werden? Wie ersichtlich geworden ist, handelt es sich bei Spring-Boot um ein Framework, das die Anwendungsentwicklung erleichtert. Dementsprechend eignet es sich vor allem für Neuimplementierungen. Sämtliche Mechanismen können dort problemlos genutzt werden. Dabei ist zu bedenken, dass der Hauptfokus des Spring-Frameworks im Enterprise-Applications-Bereich liegt. Dementsprechend wird Spring-Boot typischerweise für die Realisierung von (Web-) Service-Komponenten verwendet. Für die Entwicklung von Desktop-Anwendungen bringt es keine nennenswerten Vorteile, wobei der entstehende Overhead, z.B. der in Spring-Boot enthaltene Tomcat, berücksichtigt werden sollte. Trotzdem ist erwähnenswert, dass Spring-Boot grundsätzlich auch gängige GUI-Frameworks wie z.B. JavaFX unterstützt[14].

Für bestehende Anwendungen kommt es vor allem auf die Migrationsstrategie an. Eine reine Umstellung auf Spring-Boot - daher ohne strukturellen Umbau - ist in den meisten Fällen nicht zu empfehlen. Hauptsächlich liegt das daran, dass der Einsatz von Spring-Boot eben einer fundamentalen Design-Entscheidung entspricht, nämlich die Festlegung einer Codestruktur bei der die Applikationskomponenten möglichst voneinander entkoppelt sind. Diese Philosophie kann in einem vorhandenen Java-Projekt ohne grundlegende Änderungen praktisch nicht umgesetzt werden. Zusätzlich gibt es grundlegende technische Herausforderungen, zum Beispiel die Frage nach der Umstellung von im Code vorhandenen Design-Patterns wie Singleton[15] auf die passende Spring-Lösung. Die Umstellung vereinfacht sich, wenn die Altanwendung bereits Spring verwendet. Wie bereits erwähnt, kann Spring-Boot auch XML-Konfigurationen (wie z.B. Bean-Definitionen) verarbeiten[16 17]. Es gibt dafür eigene Annotationen (**`@Import`**, **`@ImportResource`**), die sich an die jeweiligen Konfigurationsklassen anheften lassen. Ein möglicher Migrations-Workflow ist in **(Abb. 3)** dargestellt: Zunächst sollte in der Altanwendung die Spring-Version auf die Zielversion der Spring-Boot-Anwendung aktualisiert werden. Nach einem erfolgreichen Regressionstest kann dann die Codebasis (Code und Konfigurationen) in ein neues Spring-Boot-Projekt übernommen werden. Im nächsten Schritt erfolgt die Anpassung des Source-Codes beziehungsweise der dazugehörigen Konfigurationen, wie Bean-Definitionen, Resource- und Property-Dateien. Daraufhin kann die XML-basierten Bean-Definitionen auf äquivalente annotationsbasierte Pendants umgestellt werden. Ist das erfolgreich abgeschlossen, ist noch die jeweilige Build-Konfiguration (**`pom.xml, build.gradle`**) zu überarbeiten. Dabei sollte insbesondere versucht werden, wo immer möglich die Abhängigkeiten von Third-Party-Libraries auf dementsprechende Spring-Boot-Starters umzustellen, welche die jeweiligen Libraries enthalten. Damit wird erreicht, dass Spring-Boot das Versionsmanagement für diese Bibliotheken übernimmt.



Schematische Migrationsstrategie einer alten Spring-Anwendung auf Spring-Boot. (Abb. 3)

Ob sich die Spring-Boot-Migration einer bestehenden Java-/Spring-Anwendung lohnt, muss für den Einzelfall kritisch geprüft werden. Im Legacy-Kontext kommt Spring-Boot hauptsächlich für Umbau-Maßnahmen zum Einsatz. In diesem Zusammenhang ist es das ideale Werkzeug, um einen Service für Schnittstellenpartner bereitzustellen, der Änderungen ausgleichen kann. Konkret gibt es folgende Anwendungsfälle:



Das Strangler-Pattern. (Abb. 5)

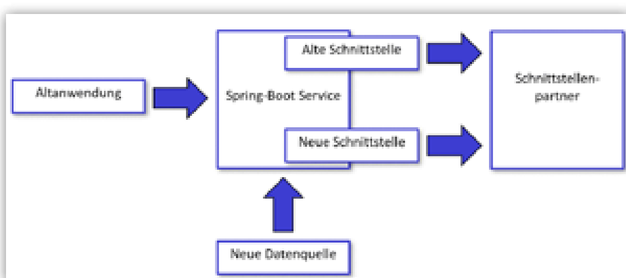
Schnittstellenänderungen, die nach außen sichtbar sind

Dieser Fall kann eintreten, wenn sich zum Beispiel die Schnittstelle ändert (Feldnamen in der übergebenden JSON-/XML-Datei) oder die übergebenen Informationen mit Daten aus einer weiteren Quelle angereichert werden sollen (Abb. 3). In diesen Situationen übernimmt der Service die Funktion eines Adapters[18], der die Daten aus der ursprünglichen Anwendung in ein anderes Format umwandelt. Der Vorteil ist dabei, dass die Anwendung selbst nicht geändert werden muss, was z B. aufgrund von fest vorgegebenen Release-Zyklen gar nicht möglich sein kann.

Änderungen an der Anwendung, die nach außen nicht sichtbar sein sollen

Dieser Fall entspricht im Wesentlichen der umgekehrten Situation, bzw. dem schrittweisen Umbau der monolithischen Altanwendung zu einer Microservice-Architektur (Abb. 4). Die Idee dabei: Eine bestimmte Funktionalität der Altanwendung wird durch einen (Spring-Boot-) Service ersetzt und dann die dementsprechende Komponente in der Altanwendung abgeschaltet. Dieses Verfahren wird solange wiederholt, bis schließlich nur noch die neuen Service-Komponenten übrigbleiben. Da die einzelnen fachlichen Module der Altanwendung dabei sukzessive abgewürgt werden, bezeichnet man diese Vorgehensweise auch als Strangler-Pattern[19, 20].

Mit der Umstellung der Altanwendung auf eine Spring-Boot-Microservice-Architektur ist auch ein wichtiger Schritt zur Cloud-Readiness der Anwendung getan. Darauf aufbauend lässt sich mit dem Spring-Cloud-Framework die Orchestrierung der einzelnen Service-Komponenten, z.B. die Erkennung von Service-Komponenten, Load-Balancing, usw. umsetzen[21, 22].



Ein Spring-Boot-Service kann als Adapter eingesetzt werden. (Abb. 4)

Fazit:

Spring Boot ist ein Framework, das Entwickler insbesondere bei der Implementation des Projektgerüsts unterstützt. Dementsprechend kommt es hauptsächlich bei der Neuentwicklung von Anwendungen zum Einsatz. Aber auch im Legacy-Kontext ist es ein nützliches Werkzeug, da man damit relativ einfach neue Service-Komponenten erstellen kann, die bei Umbau-Maßnahmen Schnittstellenänderungen ausgleichen.

Quellen:

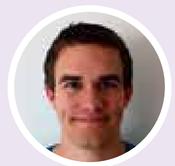
1 https://spring.io/
2 http://bit.ly/2ZUDAni
3 http://bit.ly/2RK4MSL
4 http://bit.ly/2Yf5Oc0
5 http://bit.ly/2LoGkoU
6 http://bit.ly/2xhdrm6
7 http://bit.ly/2XeCfKZ
8 http://bit.ly/2FDxwrA
9 https://start.spring.io/
10 http://bit.ly/31XwTTg
11 http://bit.ly/2FFL5Xk
12 http://bit.ly/2xf3DJu
13 http://bit.ly/2X7zp5B
14 http://bit.ly/2FADLMz
15 http://bit.ly/2ISjfJE
16 http://bit.ly/2XBA8jH
17 http://bit.ly/2FAN1Ra
18 http://bit.ly/2XuuvEf
19 http://bit.ly/2ISWwgu
20 http://bit.ly/2LqoDVY
21 http://bit.ly/2IVUWKR
22 http://bit.ly/2J9PtPm

#JAVAPRO #Framework #Desktop #Testing

Web- und Desktop-Anwendung aus einer Code-Base

Viele sehen die Zukunft von Java im Web, gleichzeitig sollen aber bestehende Desktop-Anwendungen weiterhin stand-alone funktionieren. Die Lösung: Einen Application-Server transparent in eine Electron-Anwendung einbetten und damit das bestehende Nutzererlebnis erhalten. Beispielhaft wird in diesem Artikel eine Swing-Altanwendung in eine Web-Anwendung transformiert und deren Qualität durch hochwertigen UI-Test gesichert.

Autor:



Daniel Rieth steht am Anfang seiner Karriere als Software-Eentwickler und -Beraterberater. Nach den ersten Schritten in Informatik und Pädagogik an der LMU München vertieft er nun seine softwaretechnischen Kenntnisse durch ein duales Studium mit QFS als Praxispartner.

Dr. Pascal Bihler forschte in Deutschland, Frankreich und den USA zu Internet -of -Things, Aagiler Software-Eentwicklung und User -Experience -Design, lehrte als Softwaretechnik-Dozent für Softwaretechnik an der Uni- Bonn, arbeitete als freier Software-Bberater für UI-Design und entwickelte die ersten erfolgreichen kooperativen mobilen Spiele. Bei QFS arbeitet er nun an Tools zur Qualitätssicherung von Desktop- und Web-Anwendungen.

Quality First Software GmbH - www.qfs.de
https://twitter.com/qftest_qfs
https://www.xing.com/companies/qualityfirstsoftwaregmbh
https://www.linkedin.com/company/quality-first-software-gmbh-qfs
https://gitlab.com/qfs
[daniel.rieth,pascal.biehler]@qfs.de

Anfang 2014 kamen die Entwickler von GitHub auf eine wegweisende Idee: Für ihren hauseigenen Editor Atom[1] verheirateten sie das auf browserlose JavaScript-Ausführung ausgelegte Node.js Framework, dessen zentraler Bestandteil (die V8-Engine) aus Chromium herausgelöst worden war, wieder mit einer rahmenlosen Chrome-Rendering-Engine. Damit schufen sie die Möglichkeit, Anwendungen, die mit Web-Technologien entwickelt waren und üblicherweise auf eine Server-Client-Infrastruktur aufsetzen, als Desktop-Anwendungen auszuführen, ohne dass nach dem Download bzw. der Installation der Anwendung zur Programmausführung eine Netzwerkverbindung notwendig ist. Ende April ist das daraus hervorgegangene quelloffene Electron-Framework[2] bei Version 5 angekommen und das Entwickler-Team verspricht Updates im Dreimonatstakt, bei denen jeweils die eingebettete Node.js- und Chromium-Version aktualisiert wird. Und neben GitHub mit dem eingangs genannten Atom-Editor sind auch viele weitere Softwareschmieden auf den Zug aufgesprungen: So arbeitet das Framework unter anderem in den bekannten Tools Microsoft-Visual-Studio-Code und Skype.

Für Java-Entwickler, die mit Swing-Applikationen, insbesondere mit Java-Webstart in eine ungewisse Zukunft sehen, bietet sich durch Electron die Möglichkeit, populäre Web-Frameworks für ein Redesign der Benutzerschnittstelle zu verwenden und dennoch weiterhin komfortable und kundenfreundliche Stand-Alone-Desktop-Clients ausliefern zu können. In diesem Artikel möchten wir diesen Prozess am Beispiel der technischen Aktualisierung einer Swing-Anwendung durchspielen.

Mit dem Skalpell UI und Controller trennen

Das MVC- (Model-View-Controller) bzw. das MVVM-Pattern (Model-View-ViewModel), bei dem an die Stelle des zentralen Controllers ein ViewModel getreten ist, das über eine Datenbindung die Anbindung an das User-Interface realisiert, mag manchem Entwickler eingestaubt vorkommen. Für den ersten Schritt der technischen Anwendungsaktualisierung bietet es aber ein hervorragendes Werkzeug: In der Theorie sind Datenhaltung und Datenpräsentation über Schnittstellen klar voneinander getrennt und technisch austauschbar, die Realität sieht leider häufig anders aus. So gilt es also zunächst klar festzulegen welche Daten im Model verwaltet werden sollen und welche Elemente sich auf das verwendete UI-Framework beziehen – der Rest verbleibt in der Controller- bzw. ViewModel-Schicht. Am Ende dieses Schrittes, der häufig an eine Mischung von Archäologie und Chirurgie mit virtuellem Pinsel und Skalpell erinnert, steht eine Anwendung, die optisch identisch zum Ursprungsprodukt ist, intern aber die einfache Möglichkeit zum Austausch der Präsentationsebene bietet. Das erfolgreiche Anwendungs-Refactoring verifiziert man am einfachsten über eine möglichst breite Abdeckung aller Anwendungsfälle mit automatisierten UI-Tests: Tools wie QF-Test[3] erlauben es, schnell mit der bestehenden Anwendung Testfälle aufzuzeichnen. Diese können dann während des Refactoring-Prozesses immer wieder gegen die überarbeitete Anwendung abgespielt werden, so dass Abweichungen früh im Prozess erkannt und korrigiert werden können.

User-Interface mit Web-Technologien darstellen

Zur Erstellung der UI verwenden wir den GUI-Builder von RapidClipse[4]. Dieses Tool ermöglicht es innerhalb von kurzer Zeit, eine voll funktionstüchtige, auf Vaadin[5] basierende Benutzeroberfläche zu erstellen und bei Bedarf auch für mehrere unterschiedliche Endgeräte und Plattformen auszuliefern. Nach dem Anlegen eines neuen RapidClipse-Projektes ist bereits der Rahmen für unsere Applikation gegeben. Die bereitgestellte Project-Management-View bietet eine gute Sortierung für unsere Datenbanken, UI-Elemente, Ressourcen und Themes, die zum Stylen unserer Anwendung definiert werden können. Wir benutzen hier das vorgegebene Runo-Theme, sodass ein einfacher Aufruf von **`setTheme("runo")`** in der **`init`** Methode unsere gesamte Interaktion mit diesem Aspekt der Frontendprogrammierung darstellt. Alle Änderungen an der View, vom Platzieren von UI-Komponenten bis zur Einstellung ihrer Eigenschaften, können in der praktischen GUI-Builder-Ansicht durchgeführt werden. Der zugehörige Code wird automatisch generiert bzw. angepasst, kann aber bei Bedarf auch selbst geschrieben und eingebunden werden. Event-Handler zur Interaktion der Oberfläche mit unserer Business-Logic werden mit einem simplen Rechtsklick auf die gewünschte Komponente erzeugt.

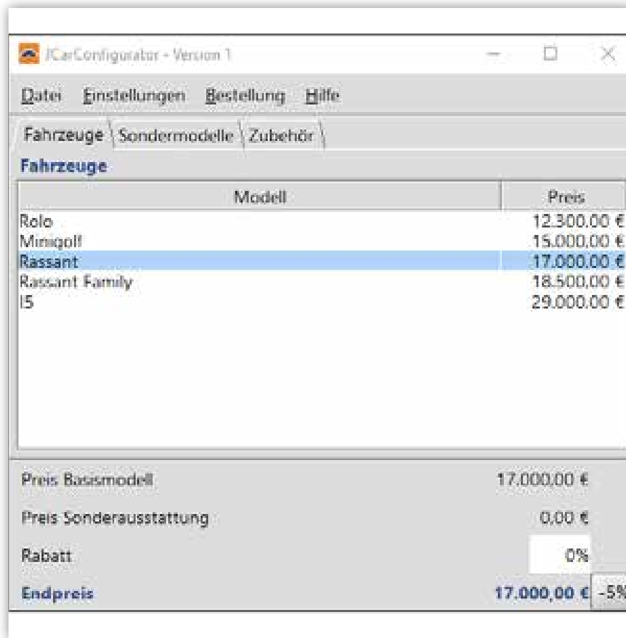
Einige Menüelemente in unserer ursprünglichen Swing-Anwendung öffnen neue Fenster, beispielsweise Dialoge, um Tabellendaten zu bearbeiten oder Informationen über die ausgewählten Elemente anzuzeigen. Um in Vaadin ein neues Fenster zu erstellen, bedienen wir uns des Navigators. Der bei Projekterzeugung von RapidClipse automatisch generierten **`DesktopUI.java`** wird ebenfalls über den GUI-Builder, ein Navigator hinzugefügt. Im Navigator können in dessen Properties mehrere Pfade mit zugehörigen Views registriert werden, welche dann im Programm mit der **`navigate`** Methode angesteuert werden können **(Listing 1)**. Die MainView hat logischerweise einen leeren String als Pfadnamen.

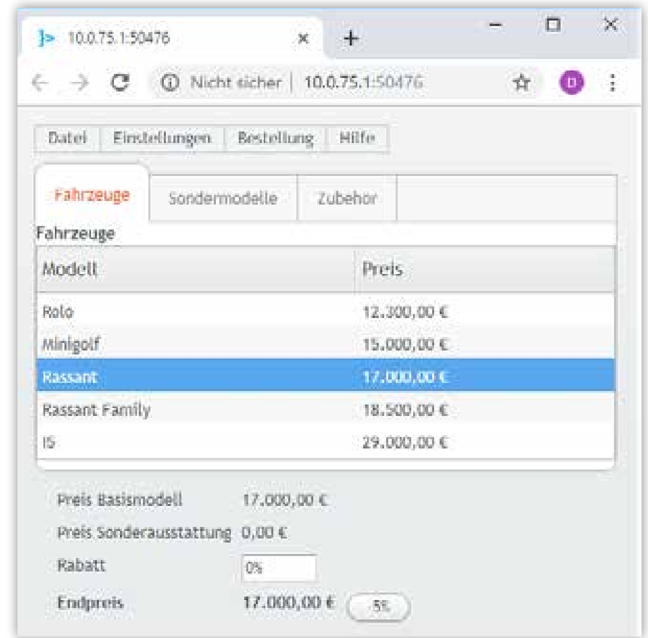
(Listing 1 - Wechsel der View per Klick auf leaveViewButton)

```
private void leaveViewButton_buttonClick(final Button.Click event){
    Navigation.to("pathname").navigate();
}
```

Ein weiteres praktisches Werkzeug von RapidClipse ist die Quick-Launch-Ansicht. Hier kann das gesamte UI, oder einzelne Fenster simultan zur Entwicklung im Browser angezeigt werden. Änderungen im GUI-Builder werden hier in Echtzeit reflektiert, wobei einige Änderungen, beispielsweise Namensänderungen der Elemente, hiervon natürlich ausgeschlossen sind.

Das Nachbilden der ursprünglichen Swing-Oberfläche erfolgt auf diese Art sehr unkompliziert. Dies liegt zum einen



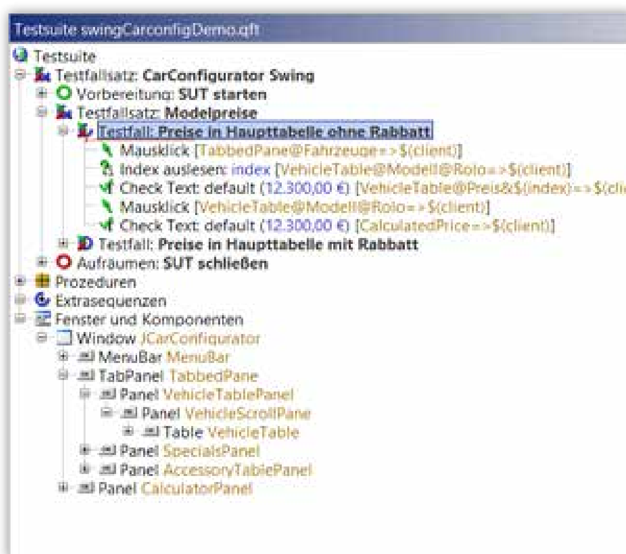


Die GUIs im Vergleich: links Swing, rechts Vaadin. (Abb. 1)

an dem einfachen GUI-Builder, zum anderen aber am verwendeten Vaadin-Framework, welches viele Aufgaben wie etwa die Client-Server-Kommunikation, transparent realisiert. Im Ergebnis steht die neu entwickelte Oberfläche in seiner Funktion und seinem Aussehen dem ursprünglichen Swing-Programm in nichts nach **(Abb. 1)**

Die Beibehaltung der UI-Struktur ermöglicht es uns nun auch, beim Testen der Oberfläche auf die bestehende Teststruktur aufzusetzen. Unterstützt das verwendete Tool gleichermaßen plattformübergreifende Tests für Java- und Web-Anwendungen, wie zum Beispiel das bereits erwähnte QF-Test, so genügt es häufig, die Komponentendefinition für die neue Engine anzupassen sowie die Start-Prozedur für das SUT (System-Under-Test – die zu testende Anwendung) neu zu generieren. Lag vorher für die Swing-Oberfläche bereits eine flächendeckende Teststruktur vor, kann diese so auf die neue Technologie angepasst werden und es können so schon im Entwicklungsprozess Fehler schnell erkannt werden. Aber auch, wenn die Testabdeckung bisher gering war, können durch den Record-Replay-Ansatz schnell weitere Tests generiert werden.

Um das generierte Vaadin-GUI mit QF-Test automatisiert zu testen, gibt es mehrere Optionen. Zum einen ist es möglich, die generierte WAR-Datei auf einem Application-Server zu deployen und sich über die URL zu verbinden. Allerdings kann es mühsam und zeitaufwändig sein, diese Datei häufig zu erzeugen, vor allem bei größeren Anwendungen. In unserem Fall lässt sich aber auch



Die dazugehörigen Tests: links Swing, rechts Vaadin. (Abb. 2)

der Preview-Modus in RapidClipse benutzen, der aus der Quick-Launch-Ansicht gestartet werden kann. Da der von RapidClipse vorinstallierte Servlet-Container allerdings die Preview jedes Mal auf unterschiedlichen Ports startet, muss die URL in der Startsequenz des SUT entsprechend angepasst werden. Hierbei hilft es, die Adresse bei allen Vorkommnissen in der Testsuite durch eine Variable zu ersetzen, insbesondere bei Verwendung des Schnellstart-Assistenten in der Sequenz „Browser-Fenster öffnen". Einfacher ist es, einen eigenen Server für die Previews festzulegen und dort den Port einzustellen. Hierfür muss im Menü Quick-Launch die Option Start Servlet ausgewählt und dort der entsprechende Port definiert werden. Für die Einbettung in eine Desktop-Anwendung ist es notwendig, die Web-Anwendung über einen lokalen, dedizierten Prozess zur Verfügung zu stellen. Für unser Beispiel bietet sich Jetty-Runner[6] an, der eine direkte Darstellung der Anwendung aus der WAR-Datei ohne zusätzliches Deployment erlaubt.

Web-Archive in Electron-App einbetten

Für den Einstieg in die Entwicklung von Electron-Anwendungen bietet das zugehörige Quick-Start-Projekt[7] eine gute Basis. Nachdem dieses geklont und über npm install eingerichtet wurde, können wir die zuvor entwickelte Vaadin-Anwendung in den Electron-Container einbinden. Dazu erstellen wir im Projektverzeichnis einen Ordner mit dem Namen lib und kopieren unsere WAR-Datei sowie die Datei **`jetty-runner.jar`**[8] dorthin. Um den Jetty-Runner zusammen mit der Anwendung starten und auch sauber wieder stoppen zu können, benötigen wir darüber hinaus noch die beiden npm-Module get-port und tree-kill, die wir über den Befehl npm install get-port tree-kill installieren können. Wenn wir das Electron-Programm über **`npm start`** starten, sieht man eine einfache Seite, die einige Informationen über die laufenden Prozesse anzeigt. Diese Seite wird aus der Datei **`index.html`** geladen und stellt die Standard-Benutzerschnittstelle für die Demoanwendung dar. Da die Informationen in dieser Datei statisch sind, können sie vergleichsweise schnell geladen werden. Mit ein paar optischen Anpassungen können wir sie so in unserem Projekt gut als Splash-Screen verwenden.

Jetzt, da das Fenster angezeigt wird, soll der Jetty-Runner im Hintergrund unsere Web-Application starten und die Anzeige, nachdem alles geladen ist, zur Anwendungs-UI wechseln. Dazu sind einige Anpassungen in der Datei **`main.js`** erforderlich. Diese Datei kontrolliert die Electron-Anwendung primär und wird dazu in einem Main-Prozess innerhalb einer Node.js-Umgebung ausgeführt. Von dort aus können Browser-Fenster geöffnet werden, welche ihrerseits JavaScript-Code ausführen können. im Beispiel wird die Datei **`renderer.js`** geladen, später dann der durch Vaadin erzeugte Code. Dieser Code wird im Kontext des Browser-Fensters im sogenannten Renderer-Prozess ausgeführt und kann mit dem Main-Prozess über eine Inter-Prozess-Kommunikation (IPC) weitgehend transparent Daten austauschen. Um dies zu vereinfachen, ergänzen wir in der **`main.js`** die Eigenschaften des Browser-Fensters **(Listing 2)**. Da wir im Fenster nur lokalen Code laden wollen, steht der Zugriff von einem Prozess auf den anderen immer unter unserer Kontrolle. Ein Beispiel für die Interprozesskommunikation ist das Beenden der Anwendung aus dem Java-Code heraus mittels **`Page.getCurrent().getJavaScript().execute("require('electron').remote.app.quit()")`**.

(Listing 2 - Vereinfachter IPC-Zugriff in der main.js)

```
// Create the browser window.
mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
        enableRemoteModule: true,
        contextIsolation: false,
        nodeIntegration: true
    }
})
```

Den Java-Prozess zur Ausführung unserer Web-Application möchten wir direkt im Anschluss an den **`createWindow`** Aufruf starten. Dazu ergänzen wir den bestehenden **`ready`** Handler der Anwendung **(Listing 3)**.

(Listing 3 - Erweiterter Ready-Handler)

```
app.on('ready', function() {
    createWindow()
    startJavaProcess('carconfig.war') // Name der WAR-Datei
})
```

In **(Listing 4)** sieht man, wie der eigentliche Jetty-Prozess gestartet wird: Zunächst wird mit dem getPort-Modul ein freier TCP-Port ermittelt. Standardmäßig startet Jetty immer auf Port 8080, der in unserem Fall dann aber schnell belegt ist. Im Anschluss daran werden der Jetty-Runner mit diesem Port und dem WAR-File als Parameter, als neuer Prozess gestartet und die Ausgaben des Prozesses abgefangen.

(Listing 4 - Start des Jetty-Prozesses)

```
async function startJavaProcess(warFileName, showOutput = false) {
    const getPort = require('get-port')
    const child_process = require('child_process')
    const javaPort = await getPort()

    let resourcesDir = app.getAppPath()
    let javaBin = 'java'

    const javaProcess = child_process.spawn(javaBin,
        ['-jar', `${resourcesDir}/lib/jetty-runner.jar`,
        '--host', '127.0.0.1',
        '--port', javaPort,
        `${resourcesDir}/lib/${warFileName}`],
        {
            stdio: ['ignore', 'pipe', 'pipe'],
            windowsHide: true,
```

```
            detached: true
        }
    )

    javaProcess.stdout.on('data', (data) => {
      if (showOutput) process.stdout.write(`jetty-out: ${data}`)
    })

    javaProcess.stderr.on('data', (data) => {
      if (showOutput) process.stdout.write(`jetty-err: ${data}`)
    })

    const pid = javaProcess.pid

    console.log(`Starting java server on port ${javaPort},
    process ${pid}`)
}
```

Nachdem der Prozess gestartet wurde, muss nun auf den Start der Web-Application gewartet und dann der Inhalt des Browser-Fensters entsprechend gewechselt werden. Dazu ist es möglich, regelmäßig den Ziel-Port abzufragen und nachzusehen, ob dort die Anwendung bereits zur Verfügung steht. Einfacher und robuster ist es aber, die Ausgaben des Jetty-Servers auszuwerten und nach der finalen Ausgabe die Umleitung durchzuführen. Dazu können wir den STDERR-Handler wie in (Listing 5) dargestellt erweitern.

(Listing 5 - Automatischer Redirect auf die Startseite)

```
javaProcess.stderr.on('data', (data) => {
    if (showOutput) process.stdout.write(`jetty-err: ${data}`)
    if (data.includes('INFO:oejs.Server:main: Started @')) {
        mainWindow.loadURL('http://127.0.0.1:' + javaPort)
    }
})
```

Alternativ zum Standard-Log des Jetty-Servers wäre es auch denkbar, nach dem Start der Web-Anwendung aus dieser heraus eine eindeutige Ausgabe zu erzeugen, auf die dann in der **main.js** reagiert wird. Abschließend soll der Java-Prozess beim Beenden der Electron-Anwendung auch zuverlässig mit allen Kind-Prozessen geschlossen werden. Dazu bietet sich auf Linux und Mac-OS-Systemen der Befehl **process.kill(-pid)** an. Durch den **detached** Parameter beim Start des Prozesses wurde eine neue Prozessgruppe erstellt, die nun durch die Angabe des Minuszeichens komplett beendet wird. Unter Windows muss man auf die Hilfe des Tree-kill-Moduls zurückgreifen. Der vollständige Quit-Handler, welchen wir noch am Ende der **start JavaProcess** Methode ergänzen, wird in (Listing 6) gezeigt.

(Listing 6 - Java-Prozess mit der Anwendung beenden)

```
app.on('quit', function(event) {
    if (pid) {
        console.log(`Stopping java server with process ${pid}`)
        if (process.platform == 'win32') {
            const treeKill = require('tree-kill')
            treeKill(pid, 'SIGTERM')
        } else {
            process.kill(-pid)
        }
    }
})
```

Electron-App für die Distribution vorbereiten

Bisher läuft die Electron-App nur innerhalb der Entwicklungsumgebung. Um eine Electron-App zu verpacken und für die Auslieferung vorzubereiten, stehen verschiedene Module bereit. Eines davon ist der electron-builder[9], welcher mit **npm install --save-dev electron-builder** dem Projekt hinzugefügt wird. Um diesen Auszuführen, fügt man in den Scripts-Bereich der **package.json** die Zeile **dist**, **electron-builder** hinzu und ruft dann **npm run dist** auf der Kommandozeile auf. Damit auch alles wie erwartet funktioniert, sind aber noch ein paar Anpassungen vorzunehmen: Zunächst muss dem Builder mitgeteilt werden, dass er die JAR- und WAR-Dateien im **lib** Verzeichnis als Resourcen aufzufassen hat. Dazu ergänzen wir die **package.json** wie in (Listing 7) dargestellt um einen **build** Abschnitt.

(Listing 7 - Build-Informationen in der package.json)

```
"build": {
    "appId": "our.examples.JavaElectron",
    "files": [
        "main.js",
        "index.html"
    ],
    "extraResources":[
        "lib"
    ]
}
```

Damit werden die Bibliotheken in das **ressources** Verzeichnis gelegt, welches während der Ausführung über **process.resourcesPath** zur Verfügung steht. Leider divergiert hier die Ausführung während der Entwicklungszeit von der fertig gebauten Version, so dass wir die **startJavaProcess** Methode wie in (Listing 8) modifizieren müssen – existiert das **lib** Verzeichnis im Ressourcen-Ordner, so gehen wir von einer Laufzeitausführung aus, andernfalls vom Entwicklungszeitpunkt.

(Listing 8 - Dynamische Festlegung des Ressourcen-Pfades)

```
let resourcesDir = process.resourcesPath
try {
    await require('fs').promises.access(resourcesDir+'/lib')
} catch (ex) {
    resourcesDir = app.getAppPath()
}
```

Darüber hinaus nimmt die Anwendung aktuell an, dass auf dem System eine Java-Ausführungsumgebung vorhanden ist. Einfacher wird die Verbreitung der Anwendung aber, wenn die JRE direkt mit in die Anwendung eingebettet wird. Dazu legen wir die entpackten JREs (die man zum Beispiel von adoptopenjdk.net[10] direkt herunterladen kann) für alle von uns unterstützen Betriebssysteme in den Projektordner und benennen die Verzeichnisse entsprechend der Architektur um in **`jre-mac-x64`**: **`jre-win-ia32`** usw. In der **`package.json`** ergänzen wir im Bereich **`extraResources`** den Eintrag **`jre-${os}-${arch}`**, damit die jeweils passende JRE zur Anwendung gepackt wird, und in der **`main.js`** bestimmen wir den Pfad zum Java-Binary wie in **(Listing 9)** dargestellt.

(Listing 9 - Pfad zum Java-Binary in der main.js bestimmen)

```
const os = (process.platform == "darwin") ? "mac" :
           (process.platform == "win32")  ? "win" : "linux"
let javaBin = `${resourcesDir}/jre-${os}-${process.arch}/bin/
java`
if (os == "mac") { // Mac-Java has a different directory struc-
ture
    javaBin = javaBin.replace('bin/','Contents/Home/bin/')
}
```

Anwendung testen mit Spectron und QF-Test

Zum Electron-Projekt gehört das Spectron-Testframework[11], welches auf Chromedriver und dem WebdriverIO-Projekt aufsetzt und zur Testautomatisierung über ein Application-Objekt Zugriff auf die Electron-Anwendung bietet. In Kombination mit einer Javascript-Testumgebung (zum Beispiel den Node.js-Modulen mocha und chai) können so Testskripte erstellt werden. Dazu installiert man die notwendigen Module über **`npm install spectron mocha chai chai-as-promised --save-dev`**, ergänzt in der **`package.json`** das Skript **`test`** : **`mocha test`** und erstellt dann eine Testdatei **`test.js`** wie in **(Listing 10)** dargestellt.

(Listing 10) Typischer Test mit Mocha, Chai und Spectron

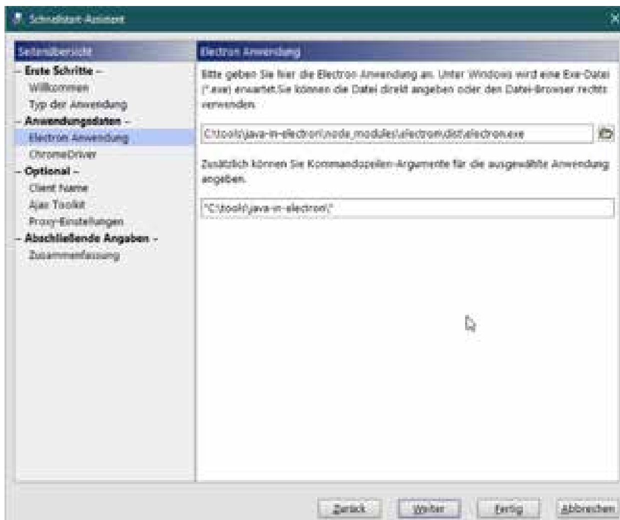
```
var Application = require('spectron').Application
var chai = require('chai')
chai.should()
chai.use(require('chai-as-promised'))

let app

before(() => {
    app = new Application({
      path: './node_modules/electron/dist/electron.exe',
      args: ['./']
    })
    return app.start()
})
```

```
describe('The app', () => {
    it('loads splash screenproperly', function() {
        return app.client.waitUntilWindowLoaded()
    })

    it('loads initial form', () => {
        return app.client.waitUntil(() => {
            return app.client.element('.v-generated-body').
            isExisting()
        },9999)
    }).timeout(10000)

    it('loads splash screenproperly', function() {
        return app.client.waitUntilWindowLoaded()
    })


    it('loads initial form', () => {
        return app.client.waitUntil(() => {
            return app.client.element('.v-generated-body').
            isExisting()
        },9999)
    }).timeout(10000)

    it ('presents correct data', () => {
        return app.client
            .getText('.v-table-table .v-table-row:nth-of-type(1)
            ' +
                     '.v-table-cell-content:nth-of-type(2)')
            .should.eventually.equal('12.300,00 €')
    })

    it ('calculates price correctly', () => {
        return app.client
            .click('.v-table-table .v-table-row:nth-of-type(1)')
            .getText('.v-gridlayout-slot:nth-of-type(9)')
            .should.eventually.equal('12.300,00 €')
    })

})

after(() => {
    if (app && app.isRunning()) return app.stop()
})
```

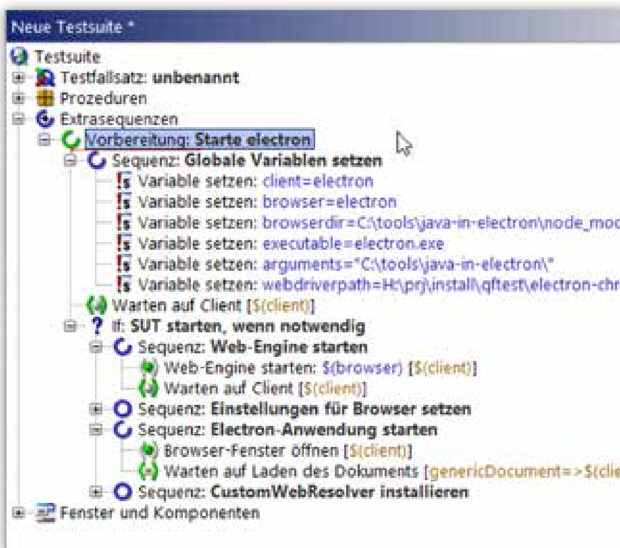
Im **`before`** Teil wird die Electron-App gestartet. Dazu wird zur Entwicklungszeit das Electron-Binary mit dem Pfad des Projektes als erster Parameter verwendet. Da die Testdefinition auf WebdriverIO aufsetzt, stellen sich auch bei Tests die typischen Herausforderungen, denen man auch zum Beispiel bei Tests mit Selenium begegnet, wie zum Beispiel die aufwendige Adressierung der Elemente.

Während sich Spectron also tendenziell direkt an Entwickler richtet, erlaubt QF-Test auch Testern ohne Programmierkenntnissen einen einfachen Zugriff auf die Electron-Anwendung. Die Tests in QF-Test können zwar mit Hilfe von Skripten beinahe beliebig ausgebaut werden, doch gleichzeitig bleibt es durch Funktionen wie etwa Capture-Replay und dem Komfort einer grafischen Benutzeroberfläche leicht zugänglich.

Durch die Unterstützung verschiedener UI-Technologien können bestehende Tests jetzt direkt für die Electron-Anwendung



Test der Electron-Anwendung zur Entwicklungszeit. (Abb. 3)



Eine Startsequenz für den Test einer Electron-Anwendung. (Abb. 4)

wiederverwendet werden, lediglich die Startsequenz benötigt Anpassungen. Dazu wählt man im **`Extras`** Menü den Schnellstart-Assistenten und dort die Option **`Eine Electron Anwendung testen`**. Im zweiten Schritt muss die zu testende Electron-Anwendung angegeben werden. Dies kann entweder die Anwendungsdatei sein, welche beim Build erzeugt wurde, oder man gibt hier die Electron-Binärdatei aus dem Ordner **`node_modules/electron/dist`** an und spezifiziert als Argument das Projektverzeichnis (Abb. 3).

Im nächsten Schritt fragt QF-Test nach der Electron-Version, die bei der Anwendungsentwicklung verwendet wurde und dem passenden ChromeDriver. In den meisten Fällen kann QF-Test automatisch die korrekte Version erkennen und den entsprechenden Treiber herunterladen. Weitere Voreinstellungen sind optional und im Assistenten selbst beschrieben, für unsere Anwendung aber nicht nötig. Der entstandene Vorbereitungsknoten stellt nun automatisch die Verbindung zu unserer Anwendung her (Abb. 4).

Sobald die Verbindung zum SUT aktiv ist, können Testbausteine aufgenommen und Tests abgespielt werden. Die Aufnahme von User-Inputs und Checks ist eine große Hilfe, möchte man schnell zu einigen ersten Tests gelangen. Hierfür genügt ein Klick auf den Aufnahmeknopf (Abb. 5), und folgende Interaktionen mit Komponenten der GUI werden aufgezeichnet. Es ist hilfreich, bestimmte Aktionssequenzen als Prozeduren im Prozedurenpaket abzuspeichern. Auf diese Weise lassen sich Tests schneller zusammenbauen und bleiben übersichtlicher. Zudem erlaubt diese Methode den Einsatz von Keyword-Driven-Testing, einem Testverfahren mit dessen Hilfe Tester mit wenig Erfahrung zum benutzten Test-Tool und ohne Programmierkenntnisse Tests aus vorher erstellten Bausteinen zusammensetzen können.

Aufnahme & Checks in der Toolbar. (Abb. 5)

Im Vergleich mit den für die Vaadin-Version erstellten Tests ergeben sich nur wenige Unterschiede. Die Komponenten unterscheiden sich zwischen beiden Formen nicht, die Startsequenzen sind sehr ähnlich. Zusätzlich ist aber hier ein Zugriff auf die nativen Menüs möglich: Er erfolgt über sogenannte Auswahlknoten, die auch entsprechend aufgenommen werden können.

Fazit:

Electron bietet Entwicklern die Möglichkeit, Web-Anwendung schnell und unkompliziert desktoptauglich zu machen und an eine breite Kundenbasis zu verteilen. Mit Tool-Unterstützung lassen sich Java-Programme schnell dorthin transferieren, Fehler bei der Migration zu vermeiden. Die in diesem Artikel dargestellten Schritte bieten eine erste Grundlage, auf die individuell durch weitere Optimierung in der Konfiguration der verwendeten Komponenten **`aufgebaut`** werden kann.

Quellen:

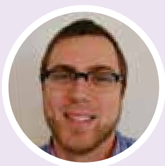
1 https://atom.io
2 https://electronjs.org
3 https://www.qf-test.com
4 https://java-pro.de/rapidclipse4
5 https://vaadin.com
6 https://www.eclipse.org/jetty/documentation/9.4.x/runner.html
7 https://github.com/electron/electron-quick-start
8 Von https://central.maven.org/maven2/org/eclipse/jetty/jetty-runner/9.4.18.v20190429/
9 https://www.electron.build
10 Am besten direkt die „zip“ bzw. „tar.gz“ des JREs anstelle des Installers auswählen
11 https://electronjs.org/spectron

#JAVAPRO #Framework #SpringBoot #JHipster

Getting Hip with JHipster

JHipster ist ein Framework, um Spring-Boot-basierte Web-Anwendungen und Microservices mit Angular, React oder Vue-Frontends zu generieren, zu entwickeln und zu betreiben. Mit JHipster lässt sich eine Anwendung bootstrappen und das Datenmodell erzeugen, sodass man sich auf das Implementieren der Businesslogik konzentrieren kann.

Autor:



Frederik Hahne arbeitet als Software Entwickler bei der WPS Management GmbH in Paderborn an der offenen B2B Integrationsplattform wescale (wescale.com). Er ist Organisator der Java User Group Paderborn (jug-pb.gitlab.io) und hat in Paderborn die lokale Devoxx4Kids Gruppe ins Leben gerufen. Seit 2015 ist er Mitglied des Java Hipster Core Teams und betreut momentan hauptsächlich den Gradle Build und den Vue.js Blueprint des Projekts. Er twittert unter @atomfrede und bloggt gelegentlich unter https://atomfrede.gitlab.io/.

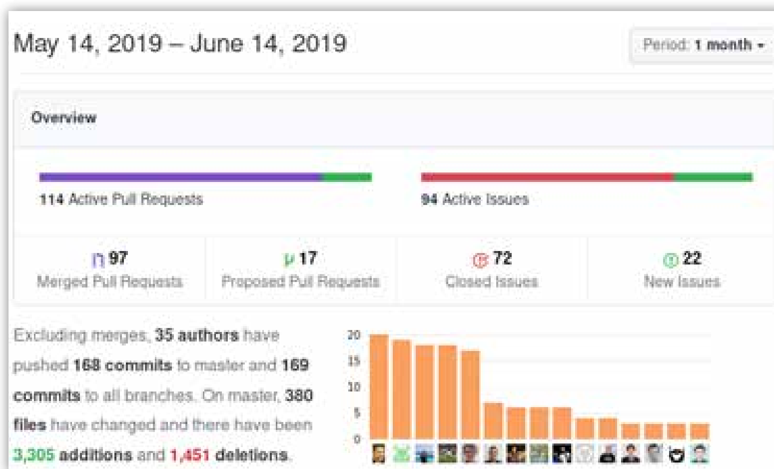
Firma: wescale.com
email: frederik.hahne@wescale.com
https://www.xing.com/profile/Frederik_Hahne/cv
https://www.linkedin.com/in/frederikhahne/
https://github.com/atomfrede

JHipster kombiniert unterschiedliche Technologien und Frameworks, konfiguriert diese nach aktuellen Best-Practices und stellt sicher, dass die verwendeten Technologien reibungslos miteinander funktionieren. Wer keine Zeit hat, sich für einen Prototypen beispielsweise in die Besonderheiten von Spring-Security einzulesen oder in die Konfiguration von Angular und Webpack einzuarbeiten, sollte sich JHipster einmal genauer ansehen. JHipster hilft nicht nur Anfängern, sondern auch erfahrenen Entwicklern schneller produktiv zu sein[17].

JHipster in Zahlen

JHipster wurde im Jahr 2013 gestartet, seitdem gab es sechs Major-Releases. Die aktuelle Version ist 6.1.0 vom Juni 2019. Das Projekt ist sehr aktiv[19] und wird von einigen Firmen finanziell gesponsert. Im letzten Monat wurden fast 100 Pull-Requests gemerged und mehr als 70 Issues geschlossen. Das Kernteam besteht aus 31 Mitgliedern, darunter auch zwei Java-Champions und insgesamt 500 Contributors. Einige Teile, wie zum Beispiel der Kubernetes-Support, werden direkt von Google entwickelt.

Unter den Nutzern von JHipster[2] befinden sich neben Google, HBO und Pivotal auch Firmen wie Siemens und Bosch.

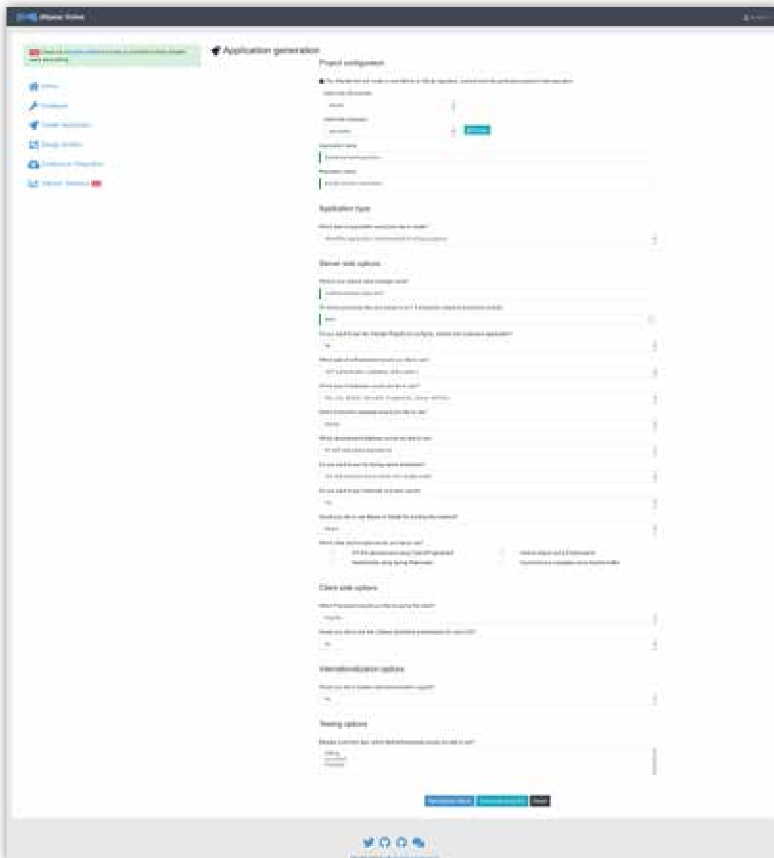


GitHub-Pulse Mai/Juni 2019. (Abb. 1)

JHipster hat fast 14.000 Stars auf Github und nach eigener Angabe mehr als 25.000 Downloads pro Woche. Durch die aktive Community und die Aktivität des Teams, können Fehler schnell behoben und Probleme gelöst werden.

JHipster-Online

JHipster bietet verschiedene Installationsarten an[3]. Die Dokumentation empfiehlt die lokale Installation. Mit JHipster-Online[7], das selbst mit JHipster entwickelt wird, kann man eine JHipster-Anwendung konfigurieren und als ZIP-Archiv herunterladen. Wenn man seinen GitHub- oder GitLab-Account mit JHipster-Online verbindet, dann können die Projekte direkt in einem Git-Repository generiert werden. Weiterhin kann man verschiedene Entitätsmodelle verwalten und editieren und diese direkt in eine generierte Anwendung importieren. JHipster-Online erzeugt dabei einen Pull-Request auf dem Projekt, sodass man als Entwickler alle Änderungen noch reviewen kann und evtl. eingerichtete CI-Pipelines die Änderungen bauen und testen können. Demnach ist es sehr einfach verschiedene Anwendungskonfigurationen und Datenmodelle zu testen. Allerdings sollte man JHipster ebenfalls lokal installieren, da einige Funktionen wie beispielsweise Deployments bisher noch nicht von JHipster-Online unterstützt werden. Zudem ist es empfehlenswert, Docker und Docker-Compose zu installieren. Damit lassen sich die erzeugten Compose-Skripte für Datenbanken verwenden, sodass man diese nicht auf dem lokalen System installieren muss.


Anwendungskonfiguration in JHipster-Online. (Abb. 2)

Module und Blueprints

Da JHipster im Kern eine Spring-Boot-Anwendung ist, können alle Konfigurationen durch modifizierte Versionen überschrieben oder erweitert werden. Außerdem versucht JHipster die einzelnen Technologien in der Standardkonfiguration zu verwenden und Modifikationen nur vorzunehmen, wenn diese unbedingt notwendig sind [13]. Dadurch sind Anpassungen einfach möglich. Im Zweifel genügt ein Blick in die Dokumentation des entsprechenden (Upstream) Projektes.

Falls man größere Anpassungen vornehmen möchte, kann man auch ein Modul [10] oder einen Blueprint [11] entwickeln und JHipser-Online on-premise installieren, sodass automatisch immer das eigene spezielle Modul zusätzlich bei der Generierung einer Anwendung verwendet wird. Ein Modul hat Zugriff auf die Konfiguration der JHipster-Anwendung und kann, wie zum Beispiel ein Sub-Generator, Dateien anlegen. Über bestimmte Erweiterungspunkte, sog. **`needles`**, kann ein Modul bestehende Dateien erweitern oder modifizieren. Es können beispielsweise neue Menüpunkte oder weitere Maven-/Gradle-Dependencies eingefügt werden. Im Unterschied zu einem Modul kann ein Blueprint existierende Dateien überschreiben oder löschen und eine eigene Menge von Dateien ausliefern. Damit ist es dem Autor eines Blueprints möglich,

```
JHIPSTER

                    https://www.jhipster.tech

Welcome to JHipster v6.1.0
Application files will be generated in folder: /home/fred/git/gitlab/atomfrede/javapro-jhipster-sample
 ______________________________________________________________________________

  Documentation for creating an application is at https://www.jhipster.tech/creating-an-app/
  If you find JHipster useful, consider sponsoring the project at https://opencollective.com/generator-jhipster
 ______________________________________________________________________________

? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? javaprosample
? What is your default Java package name? com.gitlab.atomfrede.javapro.sample
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? PostgreSQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Ehcache implementation (local cache, for a single node)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? (Press <space> to select, <a> to toggle all, <i> to invert selection)
    info Using blueprint generator-jhipster-vuejs for client subgenerator
? Which *Framework* would you like to use for the client? Vue.js
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? Default JHipster
    info Using blueprint generator-jhipster-vuejs for common subgenerator
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application German
? Please choose additional languages to install English
? Besides JUnit and Jest, which testing frameworks would you like to use? (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Would you like to install other generators from the JHipster Marketplace? No
    info Using blueprint generator-jhipster-vuejs for languages subgenerator
```

Konfiguration der JHipster-Anwendung. (Abb. 3)

u.a. Spring-Boot durch ein anderes Framework zu ersetzen, die Konfiguration an die Bedürfnisse des eigenen Unternehmens anzupassen oder Java durch Kotlin zu ersetzen[14]. Module sind seit JHipster 3 verfügbar, Blueprints erst seit Version 5 (als Beta). Daher ist die Auswahl an Blueprints noch nicht sehr groß. Seit Version 6 ist der offizielle Blueprint um Vue.js als Client-Side-Framework verwenden zu können in Version 1.0.0 verfügbar [15].

Let's get started

Gemäß Installationsanleitung[4] muss ein JDK (11), NodeJS in der aktuellen LTS-Version, Docker und Docker-Compose installiert werden. JHipster selbst kann via NPM installiert werden.

(Listing 1)

```
npm install -g yo generator-jhipster
```

Mit Version 6 wurden weitere Installationsarten [24], wie Homebrew oder die JHipster-Devbox [23], eine Vagrant-basierte virtualisierte Entwicklungsumgebung abgekündigt, da der Nutzen entweder gering oder der Wartungsaufwand zu hoch gewesen ist. Nach der Installation eines Blueprints, z.B. vue.js [15], via **`npm install -g generator-jhipster-vuejs`** kann dieser beim Erstellen einer Anwendung verwendet werden. Eingabe von **`jhipster --blueprint vuejs`** startet der Generator im Terminal. Die anschließenden Fragen können mit dem Vorgabewert beantwortet werden **(Abb. 3)**. Neben MySQL kann auch PostgreSQL verwendet werden. Auch zwischen Maven und Gradle kann man wählen. Für den Anfang sind die Standardvorgaben zu empfehlen. Damit werden Unit- und (Spring-) Integrationstests für den Serverteil generiert. Für den Frontendteil werden im Standard Unit-Tests mit Jest erzeugt, sodass jede JHipster-Anwendung eine sehr gute Bewertung bei der Analyse mit Sonar bekommt [25]. Bei Bedarf können sowohl E2E-Tests via Protractor, Performancetests mit Gatling und Behaviour-Driven-Tests mit Cucumber generiert werden [26].

Nach der Generierung der App lässt sich die Anwendung mit **`./gradlew`** starten. Nachdem der Java- und der Frontend-Build erfolgreich durchgelaufen sind, lasst sich die Anwendung mit **`localhost:8080`** öffnen und man wird von der JHipster-Willkommensseite begrüßt. Das Java-Backend lässt sich mit **`./gradlew`** in einem Terminal starten und in einem weiteren Terminal das Frontend mit **`npm run start`**. Nach dem Einfügen eines **`div`** Tags in die Datei **`src/main/webapp/app/core/home/home.vue`** sind die Änderungen nach kurzer Zeit im Browser zu sehen.

Durch die Integration der Spring-Boot-Dev-Tools muss auch bei Änderungen am Java-Code nicht die komplette Anwendung neu gestartet werden. Momentan läuft die Anwendung noch im sogenannten Development-Profil. Daher sind die JavaScript- und CSS-Dateien noch nicht optimiert und als Datenbank wird H2 verwendet. Im Production-Profil kann die Anwendung gestartet werden. JHipster erzeugt für alle externen Services (z.B. Datenbanken) passende Docker-Compose-Skripte, damit man als Entwickler die Datenbank nicht lokal installieren muss.

(Listing 2 - Starten der Anwendung im Production-Profil)

```
docker compose -f src/main/docker/postgresql.yml up -d
./gradlew -Pprod
```

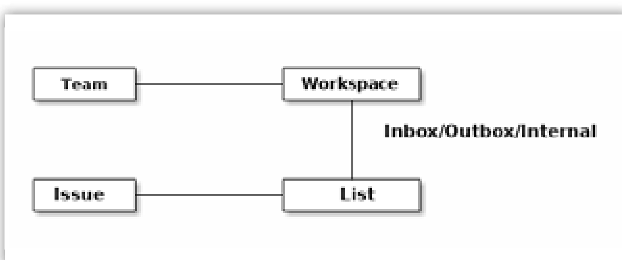
JHipster Willkommensseite. (Abb. 4)

Nach dem Anmelden mit dem hinterlegten Standard-Login **`admin`**, **`admin`** hat man neben einer Benutzerverwaltung, einer Übersicht der erzeugten Metriken (Antwortzeit, Cache-Statistiken, JVM-Metriken) auch die Möglichkeit, die Log-Level zur Laufzeit zu ändern. Die Sprachumschaltung zwischen seinen gewählten Sprachen klappt reibungslos. Auf diese Weise hat man mit wenigen Befehlen eine Web-Anwendung mit Benutzerverwaltung, Security, REST-API und einem modernen Frontend generiert. Nun kann man mit dem Erstellen und Testen des Datenmodells fortfahren.

Konfiguration einer Entität mit dem Entity-Generator. (Abb. 6)

Das Datenmodel

Im ersten Schritt soll ein spezieller Workflow abgebildet werden, bei dem alle Teams relativ autonom arbeiten können. Zwischen einzelnen Aufgaben gibt es gewisse Abhängigkeiten. Zum Beispiel kann das Team **`Einkaufswagen`** erst dann neue Produktinformationen verarbeiten, wenn das Team **`Produkt`** diese Daten auch bereitstellt. Diese Abhängigkeiten sollen explizit dargestellt werden und z.B. soll ein Ticket für das Einkaufswagenteam erstellt werden, wenn das Produktteam sein Ticket abgeschlossen hat. Dazu wird ein einfaches Modell verwendet. Jedes Team hat einen Workspace, der mehrere Listen mit Aufgaben beinhaltet. Diese Listen sind entweder intern oder dienen als Outbox bzw. Inbox eines anderen Teams. Die Aufgaben sollen später aus dem Ticketsystem in das neue Tool synchronisiert werden. Am Ende folgt eine einfache Ansicht der Aufgaben für die Teams, ohne dass diese mit Informationen konfrontiert werden, die üblicherweise nur für das Projektmanagement relevant sind.



Datenmodell. (Abb. 5)

(Listing 3 - Erzeugen der Entität Team)

```
jhipster entity team
```

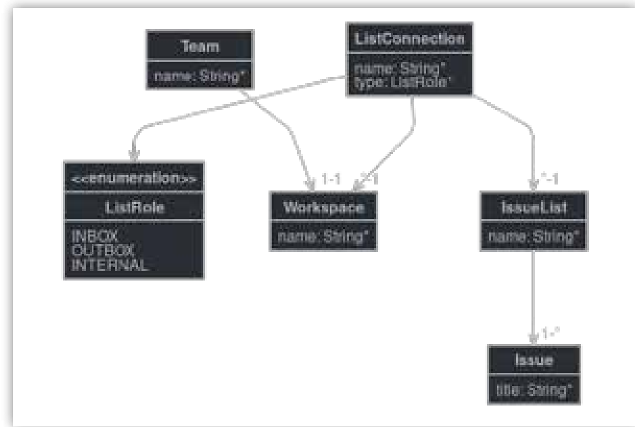
Für jede Entität alle Fragen erneut zu beantworten ist überhaupt nicht "hip". Die CLS ist auch nicht die einzige Möglichkeit, Entitäten anzulegen.

JDL

Mit dem JDL-Studio[6] lässt sich das Entitätenmodell mit einer domänenspezifischen Sprache beschreiben. Das Model wird dann direkt im Browser als Klassendiagramm gerendert. Neben Entitäten kann man auch ganze Anwendungen und sogar Microservices mit JDL definieren und somit fast vollständig ohne CLI arbeiten.

(Listing 4)

```
application {
config {
    applicationType monolith,
    baseName javaprosample
    packageName com.gitlab.atomfrede.javapro.sample,
    authenticationType jwt,
    prodDatabaseType postgresql,
    buildTool gradle,
    clientFramework vuejs,
    enableTranslation true,
    nativeLanguage de,
    languages [en, de]
   }
}
```



JDL Model als Diagram. (Abb. 7)

Das fertige JDL-Model lässt sich dann exportieren und mit **`jhipster import-jdl jhipster-jdl.jh`** in die eigene Anwendung importieren. Neben Liquibase-Changesets, um die Datenbanktabellen anzulegen, werden alle nötigen Spring-Data-Repositories und Entitätsklassen mit entsprechenden JPA-/Hibernate-Annotationen erzeugt. Für jede Entität steht eine einfache CRUD-Oberfläche (Abb. 8) (Abb. 9) zur Verfügung, die über eine REST-API mit dem Backend kommuniziert. Der aktuelle Stand kann dann ins Versionskontrollsystem gepusht werden.

Generierte CRUD-Oberfläche (Übersichtstabelle) mit automatisch erzeugten Demodaten. (Abb. 8)

Generierte CRUD-Oberfläche (Detailansicht). (Abb. 9)

(Listing 5 - Entitätsdefinition via JDL)

```
entity Team {
name String required minlength(3)
}
entity Workspace {
name String required
}
entity IssueList {
name String required
}
entity Issue {
title String required
}
entity ListConnection {
name String required
type ListRole required
}
enum ListRole {
INBOX, OUTBOX, INTERNAL
}
relationship OneToOne {
Team{workspace(name)} to Workspace{team(name)}
}
relationship OneToMany {
ListConnection to IssueList{listConnection(name)}
ListConnection to Workspace{listConnection(name)}
}
relationship OneToMany {
IssueList to Issue{issueList(name)}
}
paginate Team, Workspace, IssueList with pagination
paginate Issue with infinite-scroll
```

Betrieb und CI/CD

Wer sich mit dem Betrieb von Servern und Datenbanken nicht beschäftigen möchte, kann Plattformen wie Heroku oder Cloud-Foundry verwenden, welche die Konfiguration von Zertifikaten, Datenbanken und Web-Servern erledigen. JHipster unterstützt neben Heroku und Cloud-Foundry auch AWS, Kubernetes und Azure [9].

Mit **`jhipster heroku`** wird der Heroku-Sub-Generator gestartet. Es muss ein Name für die Anwendung vergeben und ausgewählt werden in welcher Region (US oder EU) die Anwendung laufen

Erfolgreicher Ablauf der erzeugten GitLab-CI-Pipeline. (Abb. 11)

```
→ jhipster heroku
INFO! Using JHipster version installed globally
INFO! Executing jhipster:heroku
INFO! Options: from-cli: true
Heroku configuration is starting
? Name to deploy as: javaprosample
? On which region do you want to deploy ? eu
? Which type of deployment do you want ? Git (compile on Heroku)

Using existing Git repository

Heroku CLI deployment plugin already installed

Creating Heroku application and setting up node environment
https://javaprosample.herokuapp.com/ | https://git.heroku.com/javaprosample.git

Provisioning addons
Created Database addon

Creating Heroku deployment files
   create Procfile
   create gradle/heroku.gradle
 conflict build.gradle
? Overwrite build.gradle? overwrite
    force build.gradle
   create src/main/resources/config/bootstrap-heroku.yml
   create src/main/resources/config/application-heroku.yml
```

Erzeugen einer Heroku-Konfiguration. (Abb. 10)

soll. Um nicht immer die Anwendung auf dem lokalen Rechner bauen und die **.jar** Datei hochladen zu müssen, kann man die Anwendung von Heroku bauen lassen (Abb. 10). Der Sub-Generator erstellt eine Anwendung auf Heroku, erweitert die Build-Files (**pom.xml** oder **build.gradle**) um die nötigen Plugins, erstellt automatisch die passende Datenbank auf Heroku, konfiguriert die Datenbankverbindung der Anwendung und deployet den aktuellen Stand. Nach ein paar Minuten ist die Anwendungen unter **https://javaprosample.herokuapp.com/** erreichbar.

Natürlich kann man auch seinen Prototypen automatisiert bauen, testen und im besten Fall auch direkt in der Cloud deployen lassen. Wer GitLab nutzt, kann den JHipster-CI/CD-Sub-Generator [8] verwenden, um automatisch eine passende Konfiguration für sein Projekt zu erzeugen. Neben GitLab werden auch Jenkins-Pipelines, Travis und Azure-Pipelines unterstützt. Nach dem Starten des CI-/CD-Generators mit jhipster ci-cd müssen noch einige Details ausgewählt werden. Eine Möglichkeit ist, seine Anwendung mit Sonar zu analysieren und von GitLab Aktualisierungen direkt auf Heroku bereitzustellen zu lassen. Insgesamt erzeugt JHipster eine Pipeline, die aus mehreren Schritten besteht. Der Deploy-Schritt ist dabei im Standard optional. Nach dem Committen und Pushen der Konfiguration startet GitLab direkt die Pipeline mit der erzeugten Konfiguration. Seit Version 6 werden Unit- und Integrationstests in verschiedenen Stufen der Pipeline ausgeführt und statt JUnit 4 wird bereits JUnit 5 Jupiter zur Ausführung aller Tests verwendet.

Fazit und Ausblick:

Mit JHipster lässt sich eine moderne Web-Anwendung innerhalb eines Tages erzeugen und in der Cloud betreiben. Der Entwickler kann sich auf sein eigentliches Ziel konzentrieren. Auch alle Schnittstellen und Datenbankentitäten lassen sich von JHipster erzeugen. Dank der JDL lässt sich sowohl die Konfiguration der Anwendung (z.B. OAuth2 statt JWT) als auch das Datenmodell leicht anpassen.

Nachdem JHipster 6 den Sprung auf Spring-Boot 2.1 und Java 11 vollzogen hat, konzentriert sich das Team auf den Feinschliff vorhandener Funktionen. Neben der Weiterentwicklung der JDL, so dass alle Optionen via JDL definiert werden können, ist geplant, Aktualisierungen von bestehenden Anwendungen zu vereinfachen, insbesondere, wenn viel eigener Code geschrieben wurde. Dadurch soll JHipster noch mehr zu einem Framework werden, das nicht nur das Scaffolding einer Anwendung vereinfacht wird, sondern den kompletten Lebenszyklus und Entwicklungsprozess abdeckt.

Durch die Blueprint Funktionalität, die seit Version 6 den Betastatus verlassen hat, wurde das Interesse von weiteren Communities geweckt. Das JHipster Core-Team portiert die Scaffolding-Funktionen momentan auf C#.Net. Maintainer von Micronaut und Quarkus versuchen ebenfalls einen JHipster-Blueprint für beide Frameworks zu bauen.

Quellen:

1 https://jhipster.tech
2 https://www.jhipster.tech/companies-using-jhipster
3 https://www.jhipster.tech/installation
4 https://www.jhipster.tech/creating-an-entity
5 https://start.jhipster.tech/jdl-studio
6 https://start.jhipster.tech
7 https://www.jhipster.tech/setting-up-ci
8 https://www.jhipster.tech/production
9 https://www.jhipster.tech/modules/creating-a-module
10 https://www.jhipster.tech/modules/creating-a-blueprint
11 https://www.jhipster.tech/heroku
12 https://www.jhipster.tech/policies
13 https://github.com/jhipster/jhipster-kotlin
14 https://github.com/jhipster/jhipster-vuejs
15 http://www.jhipster-book.com
16 https://www.packtpub.com/application-development/full-stack-development-jhipster
17 https://www.openhub.net/p/generator-jhipster
18 https://gitlab.com/atomfrede/javapro-jhipster-sample
19 https://sonarcloud.io/dashboard?id=javaprosample
20 https://javaprosample.herokuapp.com
21 https://github.com/jhipster/jhipster-devbox
22 https://www.jhipster.tech/2019/05/02/jhipster-release-6.0.0.html
23 https://sonarcloud.io/dashboard?id=io.github.jhipster.sample%3Ajhipster-sample-application
24 https://www.jhipster.tech/running-tests

#JAVAPRO #Security

Java - aber sicher!

Java boomt. 24 Jahre nach der Einführung ist Java populär wie nie – und die Zeichen stehen auch 2019 klar auf Wachstum. Denn Java gehört nicht nur im Web- und Mobile-Development nach wie vor zu den Favoriten vieler Entwickler, sondern hat auch in Wachstumssegmenten wie Big-Data und Internet-of-Things fest den Fuß in der Tür. Angesichts der zunehmend kritischen Use-Cases sind Entwickler aber gut beraten, bei Java-Projekten das Augenmerk auch auf die Security zu legen.

Java galt in Entwickler-Kreisen vom ersten Tag an als äußerst robuste und sichere Programmiersprache und kann diesen Ruf auch nach mehr als zwei Jahrzehnten im professionellen Einsatz dank breitem Community-Support behaupten. Dennoch weisen einige der größeren Java-Frameworks relevante Sicherheitslücken auf, die Entwickler bei ihren Projekten im Hinterkopf behalten sollten. Im Folgenden haben wir die wichtigsten Java-Frameworks und deren Security-Implikationen zusammengefasst.

Autor:

Tom Zahov Zaubermann ist seit Mai 2019 bei Checkmarx und zeichnet in seiner aktuellen Position als Sales Engineer für die DACH-Region verantwortlich. Er ist spezialisiert auf die Bereiche Application Security und Software Exposure. Er verfügt über langjährige Erfahrung in der IT-Security-Beratung von Unternehmen und Regierungsstellen in Europa, Afrika, Vietnam und Singapur. Sein besonderes Interesse gilt außerdem dem sich schnell entwickelnden eAuto-Sektor und den damit verbundenen Sicherheitsrisiken.

Nach seiner Ausbildung in Israel war Tom Zahov Zaubermann für mehrere führende IT-Security-Unternehmen wie CYMOTIVE Technologies und CIPHRON in Deutschland tätig.

Struts

Struts ist ein kostenloses, aktionsbasiertes Open-Source MVC-Framework (Model-View-Controller), das vorrangig bei der Entwicklung von JEE-Webanwendungen zum Einsatz kommt. Hinter Struts steht dabei der spannende Ansatz, die Modellebene (die Anwendungslogik, die mit einer Datenbank interagiert) von der Ansichtsebene (der HTML-Seite, die dem Kunden angezeigt wird) und der Controller-Ebene (der Instanz, die den Informationsaustausch zwischen den beiden anderen Ebenen steuert) zu entkoppeln. Zu den größten Sicherheitsrisiken beim Einsatz von Struts gehörte lange die Möglichkeit zur Remote-Ausführung von Code. Struts2, ein beliebtes Struts-Framework, das ab 2010 millionenfach heruntergeladen wurde und in über 18.000 Unternehmen zum Einsatz kam, wies eine dedizierte Schwachstelle auf, über die Angreifer beliebigen Code in Struts2-Webanwendungen ausführen konnten.

Spring-MVC

Das Spring-Application-Framework Spring-MVC wurde entwickelt, weil die Spring-Developer eine klarere Trennung zwischen der Presentation-Ebene und der Request-Handling-Ebene sowie zwischen der Request-Handling-Ebene und der Modell-Ebene wünschten. Ähnlich wie bei Struts handelt es sich auch bei Spring-MVC um ein aktionsbasiertes Framework, das allerdings mit der CVE-2012-3451 lange eine brandgefährliche Library-Vulnerability aufwies, angesichts von über 18 Millionen Downloads in 2011 und 2012 ein riesiges Schadenspotenzial!

GWT

GWT (ehemals Google-Web-Toolkit) ist ein Open-Source-Toolset, mit dem Developer komplexe, JavaScript-basierte Frontend-Anwendungen in Java entwickeln und pflegen können. GWT ist kostenlos und wird sowohl von Google als auch von tausenden anderer Entwickler auf der ganzen Welt verwendet. Das Toolset erlaubt es Entwicklern, auch ohne tiefe Kenntnis von Browser-Eigenheiten, XML-HTTP-Requests und JavaScript, leistungsfähige und hoch performante Web-Anwendungen zu programmieren. Unter Sicherheitsgesichtspunkten bleibt allerdings festzuhalten, dass GWT letztlich JavaScript-Code generiert – und damit den gleichen Sicherheitsrisken unterliegt wie JavaScript.

Hibernate

Hibernate-ORM, das häufig als Hibernate abgekürzt wird, ist ein objektrelationales Mapping-Framework für Java. Es wurde entwickelt, um Java-Klassen mit Datenbank-Tabellen sowie Java-Datentypen mit SQL-Datentypen zu mappen. Anwendungen die Hibernate verwenden, galten lange als riskant, da sie über SQL Injections, beziehungsweise HQL-Injections angegriffen werden können, wenn die Queries als verkettete Zeichenfolge eingegeben werden. Allerdings kann man sich gegen diese Angriffe zuverlässig schützen, indem in SQL- und HQL-Queries ausschließlich benannte Parameter verwendet werden.

OWASP-ESAPI

OWASP-ESAPI ist die Enterprise-Security-API der OWASP – eine kostenlose Open-Source-Bibliothek leistungsfähiger Web-Application-Security-Tools. Diese machen es Codern leicht, sichere Anwendungen zu entwickeln und ermöglichen es ihnen, das Sicherheitsniveau bestehender Anwendungen rückwirkend spürbar zu erhöhen.

Java-Server-Faces (JSF)

Java-Server-Faces ist eine Java-Spezifikation für die Entwicklung komponentenbasierter User-Interfaces für Web-Anwendungen. Dabei implementiert JSF kein dediziertes Security-Modell, sondern setzt nahtlos auf der etablierten JEE-Security auf. In der Praxis bedeutet das, dass vorhandene Application-Server-Security-Modelle wie JAAS oder andere ACL-Implementierungen sehr einfach und ohne zusätzliche Integration mit dem JSF-Framework kombiniert werden können. Mögliche Sicherheitsrisiken bei der Verwendung von JSF sind allerdings die Bereiche Access-Control und Autorisierung.

Java-Server-Pages

Aufsetzend auf der Servlet-API ermöglicht es Java-Server-Pages (JSP) Entwicklern, in ihren Seiten Java-Code einzubetten. Dieser wird kompiliert und ausgeführt, wenn ein Request eingeht. Die größte Gefahr geht bei dieser Technologie vom Cross-Site-Scripting (XSS) aus.

Java-Security heute

Die Java-Plattform unterstützt standardmäßig ein breites Feature-Set, das die Sicherheit von Java-Anwendungen optimiert. Entscheidend ist für Web-Entwickler dabei, dass potenzielle Sicherheitslücken im Code möglichst früh im Software-Development-Lifecycle identifiziert und behoben werden. Darüber hinaus müssen Developer mit Blick auf die Java-Security auch darauf achten, klassische Fehler wie unbeschränkte Zugriffsrechte auf Klassen und Variablen oder nicht finalisierte Klassen zu vermeiden. Und schließlich sollten sich Entwickler auch mit den bekannten Sicherheitsrisiken vertraut machen, die bei der Programmierung in Java und der Arbeit mit Java-Frameworks lauern.

Erfahrungsgemäß sind Lösungen für statische und interaktive Codeanalysen sowie Open-Source-Analysen eine wertvolle Hilfe wenn es gilt, in Java programmierte Anwendungen über den gesamten Software-Development-Lifecycle zu schützen und die durchgängige Einhaltung von Security- und Compliance-Vorgaben zu gewährleisten. Plattformen wie die Software-Exposure-Platform von Checkmarx lassen sich nahtlos in die vorhandene Entwicklungsumgebung und die CI/CD-Pipeline integrieren und stellen mit innovativen Analyse- und Training-Tools sicher, dass sowohl der eigenentwickelte Code, als auch Open-Source-Komponenten schnell und sicher implementiert werden können.

Die Top-10-Schwachstellen von Java

Java ist auf über 95 Prozent aller Business-Desktops weltweit installiert. Schwachstellen im Java-Code stellen ein enormes Sicherheitsrisiko dar. Die folgende Liste präsentiert einige besonders gefährliche Threats, vor denen Java-basierte Anwendungen heute unbedingt geschützt werden müssen:

1. Code-Injections
2. Command-Injections
3. Connection-String-Injection
4. LDAP-Injection
5. Reflected-XSS
6. Resource-Injection
7. Second-Order-SQL-Injection
8. SQL-Injection
9. Stored-XSS
10. XPath-Injection

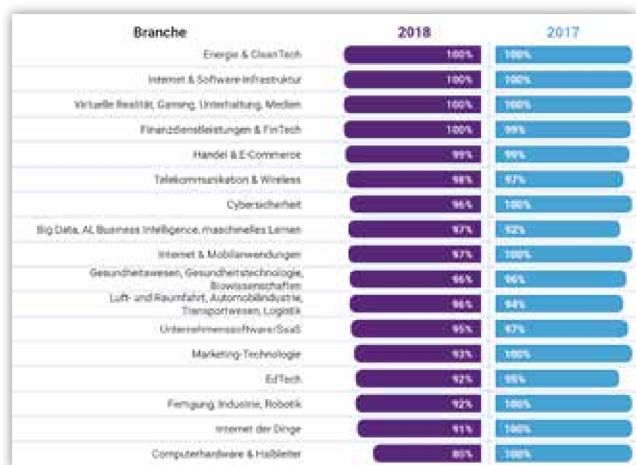
#JAVAPRO #Security #OpenSource

Risiko Open-Source

Entwickler übersehen häufig die mit Open-Source verbundenen Sicherheits- und Lizenzrisiken.

Das Black-Duck-Audit-Services-Team von Synopsys führt jedes Jahr Open-Source-Audits für seine Kunden durch und prüft dabei tausende Codebasen. Diese Prüfungen werden in erster Linie von M&A-Prozessen bestimmt und sind die primäre Quelle anonymisierter Daten für die jährliche Open-Source-Sicherheits- und Risikoanalyse (OSSRA). Die diesjährige OSSRA 2019 Studie untersucht die Ergebnisse von über 1.200 kommerziellen Codebasen und Bibliotheken. Sie zeigt Trends und Muster in der Nutzung von Open-Source sowie die Verbreitung von unsicheren Open-Source-Komponenten, wie auch von Lizenzkonflikten, auf.

Die Audits ergaben, dass mehr als 96% der im Jahr 2018 analysierten Codebasen Open-Source-Komponenten enthielten. Nach 57% im Jahr 2017 machte Open-Source dieses Jahr bereits 60% des geprüften Codes aus. Diese Zahlen spiegeln wider, dass die geprüften Codebasen mehrheitlich von Unternehmen stammen, deren Hauptgeschäft in der Entwicklung von Software besteht. Der Unternehmenswert solcher Firmen liegt häufig in ihrem proprietären Code, deshalb ist das Verhältnis zwischen proprietärem Code und Drittanbietercode hier tendenziell geringer als in anderen Branchen.



Branche	2018	2017
Energie & CleanTech	100%	100%
Internet & Software-Infrastruktur	100%	100%
Virtuelle Realität, Gaming, Unterhaltung, Medien	100%	100%
Finanzdienstleistungen & FinTech	100%	99%
Handel & E-Commerce	99%	99%
Telekommunikation & Wireless	98%	97%
Cybersicherheit	96%	100%
Big Data, AI, Business Intelligence, maschinelles Lernen	97%	92%
Internet & Mobilanwendungen	97%	100%
Gesundheitswesen, Gesundheitstechnologie, Biowissenschaften	96%	96%
Luft- und Raumfahrt, Automobilindustrie, Transportwesen, Logistik	96%	94%
Unternehmenssoftware/SaaS	95%	97%
Marketing-Technologie	93%	100%
EdTech	92%	95%
Fertigung, Industrie, Robotik	92%	100%
Internet der Dinge	91%	100%
Computerhardware & Halbleiter	80%	100%

Prozentsatz der auditierten Codebasen, die Open Source enthielten, nach Branchen. (Tabelle 1)

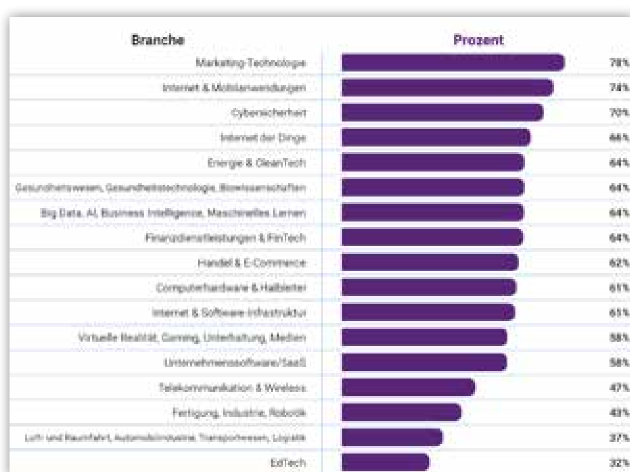
Im Gegensatz dazu stellten Analysehäuser wie Forrester und Gartner fest, dass über 90% der IT-Firmen Open-Source-Software auch in geschäftskritischen Workloads einsetzen und dass Open-Source bis zu 90% des Codes ausmachen kann. Laut der aktuellen RedHat State-of-Enterprise-Open-Source–Studie[1] waren über 69% der befragten Unternehmen der Ansicht, dass

Autor:

Fred Bals, Senior Content Strategist,
Black Duck by Synopsys

Fred Bals ist „Corporate Storyteller" für Synopsys. Sein erklärtes Ziel ist es Sicherheits-, Risiko-, Entwickler-, M&A- und Rechtsteams den notwendigen Einblick zu verschaffen, um Open-Source-Sicherheits- sowie Lizenzrisiken besser verstehen zu können. Fred Bals ist Forscher am Cybersecurity Research Center (CyRC) von Synopsys und Autor der Open-Source-Sicherheits- und Risikoanalyse 2019 (OSSRA). Vor seiner Tätigkeit für Synopsys arbeitete er als Forscher für Bob Dylan und Google.

die Nutzung von Open-Source enorm wichtig sei. Welche Zahlen man auch betrachtet, es wird deutlich, dass Open-Source-Komponenten und -Bibliotheken den Grundstein für nahezu jede Anwendung in jeder Branche bilden. Die meisten Unternehmen verfügen über eine Vielzahl unterschiedlicher Softwareprodukte, von mobilen Apps über cloudbasierte Systeme bis hin zu Altsystemen, die On-Premises ausgeführt werden. Diese Software ist in der Regel eine Mischung aus kommerziellen Standardpaketen, Open-Source-Software und selbst entwickelten Codebasen.



Durchschnittlicher Prozentanteil an Open Source in jeder auditierten Codebasis nach Branche. (Tabelle 2)

Nur wenige Unternehmen prüfen die Open-Source-Komponenten ausreichend, die sie in ihrem Code verwenden, und haben entsprechende Richtlinien, Prozesse sowie Tools implementiert. Diese Maßnahmen sind jedoch erforderlich, da Entwickler immer mehr Open-Source einsetzen. Infolgedessen können die Vorteile von Open-Source auch eine Reihe von Risiken mit sich bringen.

Die Top-10 der genutzten Open-Source-Komponenten. (Tabelle 3)

Ungepatchte Software birgt ein Risiko, nicht aber die Verwendung von Open-Source

Wie der Report von Red Hat feststellt, wird der Faktor Sicherheit als größtes Hindernis für die Verwendung von Open-Source aufgeführt. Interessanterweise wird aber im gleichen Bericht Sicherheit als einer der wichtigsten Vorteile genannt, die IT-Entscheidungsträger bei der Verwendung von Open-Source sehen. Dieser scheinbare Widerspruch spiegelt die Sorge wider, dass Open-Source-Code, der nicht korrekt verwaltet wurde, Sicherheitslücken verursachen kann, sowohl in Open-Source als auch in proprietären Lösungen. Im Gegensatz dazu verringert eine ordnungsgemäße Verwaltung von Open-Source potenzielle Risiken erheblich, einschließlich der Verwendung vertrauenswürdiger Quellen und automatisierter Tools zum Aufdecken und Beheben von Sicherheitsproblemen. Jede Software, ob proprietär oder Open Source, weist Schwachstellen auf, die Unternehmen identifizieren und patchen müssen. Die Open-Source-Community leistet beispielhafte Arbeit bei der Bereitstellung von Patches – oft werden diese schneller veröffentlicht als ihre proprietären Pendants. Eine alarmierende Anzahl von Unternehmen nutzt diese jedoch nicht rechtzeitig oder gar überhaupt nicht.

Die Gründe für das Ausbleiben von Patches sind vielfältig: Einige Unternehmen sind von der schieren Anzahl an verfügbaren Patches überfordert und können diese somit nicht priorisieren. Andere Unternehmen wiederum verfügen nicht über entsprechend geschulte IT-Mitarbeiter und müssen das Risiko gegen die finanziellen Kosten abwägen. Ungepatchte Schwachstellen bleiben eine der größten Cyber-Bedrohungen für Unternehmen. Die OSSRA 2019 Studie zeigt, dass 60% der im Jahr 2018 geprüften Codebasen mindestens eine Open-Source-Schwachstelle enthielten. Synopsys hat seine Black-Duck-KnowledgeBase im letzten Jahr um 7.393 Open-Source-Schwachstellen erweitert. In den letzten zwei Jahrzehnten wurden weit über 50.000 Open-Source-Schwachstellen gemeldet. Diese Entwicklung zeigt, dass es essentiell ist, seine Software zu patchen.

Die Top-10-Sicherheitslücken mit hohem Risiko. (Tabelle 4)

Welches Risiko besteht?

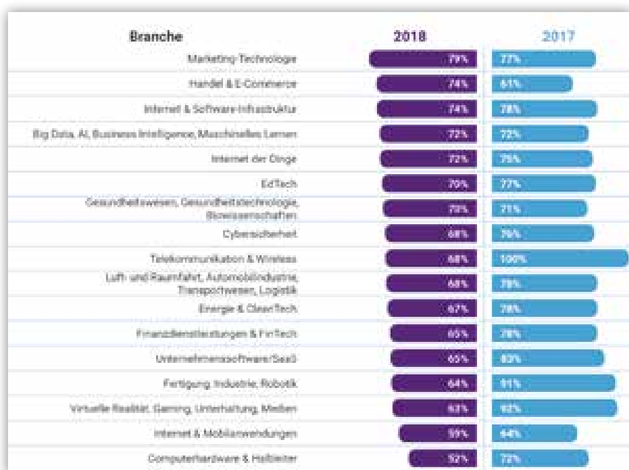
Bestimmte Merkmale von Open-Source machen Schwachstellen in oft verwendeten Komponenten für Angreifer attraktiv. Im Gegensatz zu kommerzieller Software, deren Anbieter automatisch Patches und Updates an die Kunden verteilen können, verfügt Open-Source über ein Pull-Support-Modell. Nutzer sind demnach selbst dafür verantwortlich Up-to-Date zu bleiben.

Die Allgegenwart von Open-Source bietet Angreifern viele mögliche Angriffsziele, da Schwachstellen durch Quellen wie die

National-Vulnerability-Database (NVD), Mailing-Listen, GitHub und Projektseiten offengelegt werden. Wie bereits erwähnt, führen viele Unternehmen keine genauen, umfassenden und aktuellen Bestandslisten der in ihren Anwendungen verwendeten Open-Source-Komponenten. In einem Bericht des U.S. Senate-Permanent-Subcommittee-on-Investigations[2] wurde beispielsweise festgestellt, dass das Fehlen eines vollständigen Softwareinventars ein Faktor für den massiven Datenverstoß im Jahr 2017 bei der US-Firma Equifax war.

Korrektes Management von Open-Source-Software ist nicht nur eine Frage der Sicherheit

So gut wie alle Open-Source-Komponenten unterliegen einer von etwa 2.500 bekannten Open-Source-Lizenzen, von denen viele mit Verpflichtungen und unterschiedlichen Einschränkungen verbunden sind. Die Missachtung von Open-Source-Lizenzen kann für Unternehmen ein erhebliches Risiko an Rechtsstreitigkeiten und eine Gefährdung des geistigen Eigentums bedeuten, einschließlich der Eigentumsrechte an firmeneigenem proprietärem Code, der einzelne Open-Source-Komponenten enthält. 68% der in der OSSRA-2019-Studie geprüften Codebasen enthielten Komponenten mit Lizenzkonflikten.



Prozentsatz der auditierten Codebasen, die Lizenzkonflikte enthielten, nach Branchen. (Tabelle 5)

Nicht verwaltete Open-Source-Komponenten sind riskant

Open-Source bietet Unternehmen viele Vorteile – jedoch nur, wenn die Komponenten ordnungsgemäß verwaltet werden. Zur Abwehr von Open-Source-Sicherheits- und Compliance-Risiken sollten Unternehmen folgende Punkte befolgen:

1. **Risikorichtlinien und Prozesse etablieren**
   Nur eine Handvoll Open-Source-Schwachstellen – wie diejenigen, die Apache-Struts oder OpenSSL betreffen – werden häufig ausgenutzt. Unternehmen sollten sich deshalb bei ihrem Schwachstellenmanagement auf CVSS-Werte (Common-Vulnerability-Scoring-System) und die Verfügbarkeit von Exploits konzentrieren, und das über die gesamte Nutzungsdauer der Open-Source-Komponente hinweg. Es sollte ein automatisierter Prozess eingerichtet werden, der die Open-Source-Komponenten in einer Codebasis und ihre bekannten Sicherheitslücken sowie operationelle Risiken nachverfolgt und Probleme nach ihrem Schweregrad priorisiert.

2. **Eine vollständige Bestandsaufnahme durchführen**
   Es ist wichtig, eine vollständige, genaue und zeitnahe Bestandsaufnahme des verwendeten Open-Source durchzuführen. Diese sollte sowohl den Quellcode, als auch Informationen darüber umfassen, wie Open-Source in bereitgestellter Software verwendet wird.

3. **Open Source bekannten Schwachstellen zuordnen**
   Öffentliche Quellen wie die National-Vulnerability-Database stellen eine gute erste Anlaufstelle für Informationen zu gemeldeten Schwachstellen dar. Man sollte sich jedoch nicht ausschließlich auf die NVD verlassen, sondern auch sekundäre Quellen berücksichtigen, die im Idealfall eine frühere Benachrichtigung über Sicherheitsrisiken sowie zusätzliche technische Details, Upgrades und Patches bereitstellen können.

4. **Schritthalten mit neuen Sicherheitsbedrohungen**
   Unternehmen müssen ein kontinuierliches Monitoring ihrer Produkte, Systeme und Applikationen sicherstellen – und zwar so lange, wie diese verwendet werden.

5. **Lizenzrisiken angehen**
   Die Missachtung von Open-Source-Lizenzen kann Unternehmen Rechtsstreitigkeiten einbringen und ihr geistiges Eigentum gefährden. Deshalb sollten die Entwickler über die Open-Source-Lizenzen informiert sein und entsprechend geschult werden. Der Syndikusrechtsanwalt sollte in diesen Schulungsprozess eingebunden werden und auch die Einhaltung lizenzrechtlicher Verpflichtungen überwachen.

6. **Open-Source-Audit zum Teil der Due-Diligence-Prüfung machen**
   Im M&A-Geschäft sollte man wissen, ob Open-Source im Spiel ist. Dies ist insbesondere dann regelmäßig der Fall, wenn Software-Assets einen wesentlichen Teil der Bewertung ausmachen. Softwarerisiken können sich im schlechtesten Fall zum Deal-Breaker entwickeln. Man sollte daher nicht zögern, Fragen hinsichtlich der Verwendung und Verwaltung von Open-Source zu stellen. Auch ein Code-Audit durch externe Experten sollte in Erwägung gezogen werden.

Quellen:

1 https://www.redhat.com/it/enterprise-open-source-report/2019
2 https://www.hsgac.senate.gov/imo/media/doc/FINAL%20Equifax%20Report.pdf

#JAVAPRO #Security

10 Grundsätze für sichere Softwareentwicklung

Die Entwicklung moderner Software ist heutzutage so komplex, dass Fehler trotz intensiver Prüfung nicht oder nur schwer erkennbar sind. Dies hat eindrucksvoll die Heartbleed-Schwachstelle der älteren Version der Open-Source-Bibliothek OpenSSL gezeigt. Ein Großteil der Online-Dienste und Websites zeigte sich dadurch für Angriffe anfällig. Dieser Artikel nennt zehn Grundsätze für eine sichere Software-Entwicklung.

Grundsatz 1: Injektionen sowie XSS- und CSRF-Schwachstellen verhindern!

Zu den am meisten verbreiteten Sicherheitsschwachstellen in Web-Anwendungen gehören Injektions-, XSS- und CSRF-Verwundbarkeiten. Sie gelten als das am meisten genutzte Einfallstor für Cyber-Kriminelle. Schaffen Hacker aufgrund dieser Schwachstellen im System den Zugriff auf sensible Daten der Kunden, lassen sich diese manipulieren und schädigen im schlimmsten Fall dauerhaft die gesamte IT-Infrastruktur.

Injektionsschwachstellen

Injektionsschwachstellen entstehen, wenn Benutzereingaben zur Erstellung von Anweisungen oder Abfragen, z.B. SQL-Abfragen, verwendet und ohne ausreichende Überprüfung an einen Interpreter weitersendet werden. Durch Manipulation der Eingabesyntax kann ein Angreifer unerwünschte Befehle ausführen und unbefugten Zugriff auf sensible Daten erhalten. Zum Beispiel ermöglicht eine SQL-Injection in einer Web-Anwendung einen Datenbankzugriff im Backend.

Cross-Site-Scripting

Cross-Site-Scripting (XSS) ist eine Art HTML-Injection, die im Web-Browser des Benutzers stattfindet. XSS-Schwachstellen entstehen, wenn Web-Anwendungen Benutzereingaben annehmen und diese anschließend ohne Validierung an den Browser weiterleiten. Ein Beispiel dafür wäre, wenn ein Benutzer einen Suchbegriff in ein Formular eingibt und anschließend eine neue Seite mit Suchergebnissen im Browser angezeigt wird, auf der auch vom Benutzer gewählte Suchbegriff nochmals mit erscheint. Auf diese Weise können Angreifer Schadcode auf indirektem Wege im Browser des Opfers ausführen – häufig über bestimmte, gut vorbereitete URLs.

Cross-Site-Request-Forgery

Bei einem CSRF-Angriff (Cross-Site-Request-Forgery) führt ein Angreifer HTTP-Anfragen über den Web-Browser eines Anwenders aus. Aus Sicht der Anwendungen sehen die Anfragen legitim aus und werden im Kontext und mit den Rechten eines aktuell angemeldeten Benutzers bearbeitet. Die Anwendung unterscheidet nicht zwischen den Anfragen des rechtmäßigen Nutzers und den gefälschten Anfragen des Angreifers. Anwendungen ohne SSRF-Schutz stellen immer eine potentielle Bedrohung für Benutzer und deren Daten dar.

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.

Tipps zur Programmierung

Zu den nützlichen Maßnahmen zählen Benutzereingaben, die nur im Bedarfsfall zum Einsatz kommen. Weiteren Schutz bieten Eingaben, die gefiltert und geprüft wurden, bevor der Entwickler sie erstmalig verarbeitet. Dieser First-Line-of-Defense-Ansatz schützt Anwendungen vor Injektionsangriffen. Alle modernen Programmiersprachen und Frameworks stellen in der Regel getestete Routinen zur Filterung von Benutzereingaben zur Verfügung und reduzieren damit das Risiko eines Angriffs deutlich. Bei der Entwicklung und Verwendung eigener Methoden besteht grundsätzlich die Gefahr etwas zu übersehen. Die Implementierung von Funktionen zum Anlegen, Lesen, Ändern und Löschen von Daten, sogenannte CRUD-Methoden[1], erfolgt am besten mit den integrierten Funktionen des jeweiligen Frameworks. Der Fachbegriff hierfür lautet Scaffolding und viele Frameworks bieten geeignete Methoden, um auf Knopfdruck die nötigen Eingabemasken inklusive Programmlogik auf Basis eines Datenmodells zu erzeugen.

Nachhaltige Systemschäden lassen sich unterbinden, wenn für den Zugriff von Web-Anwendungen auf Datenbanken ausschließlich definierte Befehle zugelassen werden, die für die Funktionalität der Anwendung notwendig sind. Sollte es einem Angreifer gelingen Schadcode zu injizieren, bleibt dieser auf Grund der Einschränkung auf bestimmte Schlüsselwörter wirkungslos und ein Systemschaden kann abgewendet werden.

Grundsatz 2: Trennung von Datenverarbeitung und Datendarstellung

Wichtig bei der Verarbeitung von Daten ist die Trennung von Darstellungslogik, z. B. Ausgabe im Web-Browser, und der internen Anwendungslogik. Dafür ist das Model-View-Controller-Pattern (MVC) prädestiniert, das vielen Java-Entwicklern noch von der GUI-Programmierung mit Java-Swing bekannt sein dürfte. Anwendungen, die dem MVC-Pattern folgen, bestehen aus einem oder mehreren Datenmodellen (Model), Darstellungsschichten (Views) und Programmsteuerungen (Controller). Durch diese Trennung reduziert das MVC-Pattern den Aufwand für Anpassungen an einzelne Komponenten der Anwendung und erhöht die Wiederverwendbarkeit dieser Elemente. Das Datenmodell beschreibt die von der Anwendung verarbeiteten Daten, ihren Datentyp, ihren Wertebereich und ist unabhängig von der Darstellung und Steuerung innerhalb der Anwendung. Die Darstellungsschicht zeigt die Darstellung der Daten gemäß dem Datenmodell und nimmt Benutzerinteraktionen entgegen. Die Programmsteuerung verwaltet eine oder mehrere Darstellungen, nimmt Benutzerinteraktionen von ihnen entgegen, wertet sie aus und handelt entsprechend.

Trennung von Anwendungslogik und Darstellung

Durch die Trennung von Anwendungslogik und Datendarstellung vermeidet der Entwickler, dass sich wichtige und sicherheitsrelevante Funktionen versehentlich in die Darstellungsschicht auslagern und der Angreifer einfacher das Sicherheitssystem umgehen kann. Die Trennung erleichtert auch die Entwicklung von einem oder mehreren Frontends unabhängig von der Anwendungslogik oder dem Datenmodell. Insbesondere bei Anwendungen mit Responsive-Design werden verschiedene Frontends nicht mit dem eigentlichen Anwendungscode vermischt. Der Programmcode bleibt klar strukturiert, was auch die spätere Wartung und Weiterentwicklung der Anwendungen erleichtert. Darüber hinaus erhöht sich die Wiederverwendbarkeit einzelner Codekomponenten durch die strikte Trennung der Programmcodes. So greifen verschiedene Anwendungen auf ein Datenmodell zu oder ein Datenmodell lässt sich durch verschiedene Ansichten darstellen, wie z. B. eine Übersichtsansicht, eine Detailansicht, eine Ansicht zur Bearbeitung der Daten.

Tipps zur Programmierung:

Hilfreich bei der Programmierung erweisen sich Entwickler-Frameworks, die das MVC-Pattern unterstützen und Werkzeuge für die Erstellung von Controllers, Views und Models bereitstellen. Das Datenmodell beschreibt beispielsweise Datentyp und Wertebereiche, der Controller implementiert Routinen zur Steuerung der Anwendungslogik und der View-Bereich stellt die Daten dar.

Scaffolding zur Implementierung von MVC-Programm-Code

Entwickler-Frameworks unterstützen häufig die Erstellung von Programmcode für Models, Views und Controller durch Scaffolding. Die Vorlagen der Frameworks erzeugen standardisierte CRUD-Operationen. Das spart Zeit und reduziert den Stress während der Entwicklung. Und nicht nur das, diese Vorlagen sind oft sehr robust und der generierte Quellcode kapselt Variablen und Eingabeparameter mit Sicherheitsmechanismen ab, die das Framework bereitstellt. Bei der Entwicklung neuer Anwendungen bieten sich fertige Entwicklungsframeworks an, die das MVC-Pattern unterstützen und Scaffolding-Mechanismen anbieten.

Grundsatz 3: Daten vor der Verarbeitung immer validieren

Die konsequente Validierung, also die Verifizierung von Eingabe- und Ausgabedaten, stellt ein Grundprinzip für die Entwicklung von korrekt funktionierenden, stabilen und sicheren Anwendungen dar. Fehlerhafte Datenpools oder gar Sicherheitsschwachstellen entstehen oftmals aufgrund fehlerhafter Prüfung von Syntax und Semantik (siehe Grundsatz 1). Die fehlende Datenvalidierung und -filterung zählt dabei zu den häufigsten Hacker begünstigenden Umständen und ermöglicht umfangreiche Angriffe auf Web-Anwendungen, z.B. SQL-Injektionen für den Zugriff auf Datenbanken. Die Vertraulichkeit, Integrität und Verfügbarkeit der Daten unserer Nutzer ist daher unmittelbar gefährdet. Häufig erhalten Angreifer über solche Schwachstellen

nicht nur Zugriff auf Daten, sondern auch auf die Server, auf denen diese Anwendungen laufen.

Gefahr von Verlust der Datenintegrität und niedriger Datenqualität

Die Integrität der Daten erweist sich neben der Vertraulichkeit und Verfügbarkeit als ein wichtiges Merkmal. Ohne ausreichende Validierung besteht die Gefahr der Unvollständigkeit oder sie erweisen sich als einfach falsch, z.B. negatives Alter einer Person. In diesem Fall spricht man auch von einer geringen Datenqualität oder einer Abweichung vom Datenmodell. Die Auswirkungen solcher fehlerhaften Datensätze breiten sich je Anwendungslogik weiter aus und verursachen Fehler in der Weiterverarbeitung innerhalb der Anwendung.

Tipps zur Programmierung:

Hauptsächlich geht es bei einer Softwareentwicklung um die Verteidigung in der Tiefe (Defense-in-Depth). Dies ist eine Methode, die Sicherheit durch mehrere Schutzmaßnahmen kennzeichnet. Sicher gewählte Anwendungen überprüfen die Korrektheit der Daten an verschiedenen Stellen während der Verarbeitung. Das heißt, umgeht der Angreifer einzelne Prüfroutinen, schlagen die übrigen Warnsysteme Alarm und die Anwender-Tools reagieren entsprechend. Im Datenmodell definieren sich die richtigen Datentypen und Wertebereiche. Die Anwendungslogik (im Controller) verarbeitet die Daten und überprüft, ob diese semantisch gültig sind.

Prüfung und Validierung von Daten

Der Softwareingenieur überprüft bei der Entwicklung von Funktionen zur Datenverarbeitung, beispielsweise zur Speicherung von Benutzereingaben in einer Datenbank oder zur Durchführung mathematischer Operationen, ob sich Werte als zuverlässig darstellen. Dies ist insbesondere bei Benutzereingaben und Übergabeparametern in Funktionen erforderlich. Einerseits sichert sich das Unternehmen dadurch die Datenqualität, andererseits verhindert es so Manipulationen oder gar Angriffe auf die Anwendung und Daten der Nutzer.

Grundsatz 4: Datenschutz zwischen Web-Browser und Anwendung vor externer Einsicht

Daten, die im Austausch stehen zwischen Servern und Web-Browsern, benötigen Schutz vor externer Einsicht. Eine sichere Datenübertragung zwischen Client und Server, z.B. über eine HTTP(S)-gesicherte Verbindung, ist wichtig, da die Bevölkerung zunehmend freie WLAN-Hotspots mit unverschlüsselten Verbindungen akzeptiert. Angreifer nutzen diese Lücke, um private und sensible Informationen wie Passwörter, Nutzersitzungen oder auch persönliche Daten einzusehen.

Einsicht von Fremden in übertragene Daten vermeiden

Erhält der Angreifer Einsicht in die Datenübertragung zwischen unseren Servern und den Nutzern, bekommt er möglicherweise Einblick in sensible und schutzwürdige Informationen. Die Vertraulichkeit der Daten ist damit nicht mehr gegeben, vor allem wenn Informationen wie Benutzernamen und Passwörter von Fremden einsehbar sind. So erhalten Unbefugte Zugang zu den Anwendungen und greifen unter vorgegebener Identität und mit den Rechten der betroffenen Nutzer auf die Daten zu und manipulieren diese. Ursachen hierfür liegen in der fehlerhaften Implementierung von Schutzmechanismen oder in der Verwendung veralteter und unsicherer Verfahren zur Verschlüsselung.

Verschlüsselung der Übertragung

Sichere Anwendungen verschlüsseln sensible Daten, beispielsweise mit TLS 1.2 oder TLS 1.3, sodass Angreifer keinen Zugang zu den Daten bekommen - auch dann nicht, wenn der Angreifer sich bereits in die Kommunikation zwischen Web-Browser und Anwendung eingeklinkt hat, zum Beispiel durch einen Man-in-the-Middle-Angriff. Über das HTTP(S)-Protokoll verschlüsselte Internetverbindungen liefern dabei einen hohen Sicherheitsstandard. Dasselbe gilt für die Datenübertragung über das Internet. Auch hier bieten sich verschlüsselte Protokolle an. In diesem Zusammenhang stellt der SFTP einen enormen Vorteil gegenüber dem unverschlüsselten FTP dar, so wie SSH anstelle von Telnet.

Die Sicherheitsschwachstellen POODLE, Heartbleed und BEAST zeigen deutlich, dass die verwendete Übertragungsverschlüsselung immer auf dem neuesten Stand der Technik sein muss, um die Vertraulichkeit sensibler Daten zu gewährleisten und dauerhafte Schäden an den Systemen der Nutzer zu vermeiden.

Verschlüsselung nicht selbst entwickeln, sondern Standardlösungen verwenden

Grundsätzlich bieten alle modernen Programmiersprachen und Frameworks Möglichkeiten zur Implementierung von bereits erprobten Verschlüsselungsverfahren für die Übertragungsverschlüsselung, z.B. TLS mit Perfect-Forward-Secrecy, die bereits mit aufwendigen Standardisierungsprozessen eingeführt wurden. Bei eigens entwickelten Verschlüsselungsverfahren besteht immer die Gefahr, etwas zu übersehen. Ein einprägsames Zitat von Brian Hatch zu diesem Thema lautet: „Jeder Programmierer versucht irgendwann, einen eigenen Verschlüsselungsalgorithmus zu entwickeln. Kurz gesagt: Lassen Sie das!“

Grundsatz 5: Die Nutzer der Anwendung im Auge behalten

Fehler in der Nutzer- und Sitzungsverwaltung von Web-Anwendungen führen oftmals zu Sicherheitsschwachstellen.

Beispielsweise durch Authentifizierungsfehler, unzureichende Prüfroutinen während des Zurücksetzens des Passworts oder durch unvollständige Prüfungen der jeweiligen Berechtigungen bei der Ausführung von Aktionen innerhalb der Anwendung (Datenzugriff, Änderungen, Datensätze löschen). Daher ist es wichtig, den Nutzer und seine Berechtigungen innerhalb einer Sitzung in den Anwendungen jederzeit im Auge zu behalten. Dies gilt insbesondere für die Prüfung von Berechtigungen bei der Ausführung von CRUD-Operationen.

Es ist wichtig, Nutzersitzungen ausreichend zu sichern, um Fremden den Zugriff auf die Anwendungen oder Cookies mit Sitzungsinformationen, z.B. SessionID, zu verwehren. Ebenso führen Fehler bei der Implementierung von Funktionen zum Passwort zurücksetzen häufig dazu, dass Fremde Einblick in Nutzerkonten erhalten. Sichere Anwendungen und Verfahren stellen daher einen wichtigen Schutz vor Bedrohungen dar.

Wichtig ist die vollständige und robuste Implementierung von Berechtigungsprüfungen, um Angreifern keine Chance für Manipulierung oder Löschung der Daten zu geben. Der unbefugte Zugriff auf Daten oder Funktionen innerhalb unserer Anwendungen gefährdet unmittelbar die Vertraulichkeit, Integrität und Verfügbarkeit der Daten unserer Nutzer und stellt eine Bedrohung für die IT-Infrastruktur dar.

`Tipps zur Programmierung:`

Für die sichere Implementierung von Nutzer-An- und -Abmeldefunktionen erweisen sich Standardverfahren, die Entwicklungs-Frameworks, als sehr hilfreich. Je nach Bedarf bieten sich auch zusätzliche Sicherheitsprüfungen durch einen zweiten Entwickler an.

Bei der Implementierung von CRUD-Operationen in den Anwendungen stellen die obligatorischen Berechtigungsprüfungen eine weitere Regel dar. Zu Beginn einer jeden CRUD-Operation ist zu prüfen, ob der aktuell angemeldete Benutzer berechtigt ist, den jeweiligen Vorgang für den betroffenen Datensatz durchzuführen. Die mithilfe von Scaffolding erstellten Frameworks, die bei CRUD-Verfahren zum Einsatz kommen, enthalten in der Regel diese Berechtigungsprüfungen und dienen während der Entwicklung als Ressourcen.

Grundsatz 6: Über die geeignete Protokollierung von Ereignissen nachdenken

Während des normalen Betriebs der Anwendungen kommen eine Vielzahl von Ereignissen und Aktionen von Nutzern vor. Treten in der Anwendung Fehler auf oder hat sich ein sicherheitsrelevanter Vorfall, wie ein Hackerangriff ereignet, ist es für eine erfolgreiche Post-Mortem-Analyse unerlässlich, die Ereignisse nachhaltig zu betrachten. Hierzu braucht es eine ausführliche Protokollierung (Logging) der Anwendungen, Ereignisse und Benutzerinteraktionen. Dabei sind insbesondere die Datenschutzbestimmungen zu berücksichtigen.

Tritt ein Sicherheitszwischenfall ein oder kommt es zur Manipulation von Daten, sind detaillierte Analysen unerlässlich. Andernfalls lassen sich Vorfälle nicht nachvollziehen und Angreifer nicht zuordnen. Treten Fehler in unseren Anwendungen auf und kontaktieren uns Kunden, sind Support- und Entwicklungsteams auf möglichst detaillierte Informationen angewiesen, um die Fehlerursache schnell zu identifizieren (Ursachenanalyse). Log-Einträge stellen bei der Fehlersuche eine wichtige Informationsquelle dar und ermöglichen es, das Problem schneller zu analysieren.

`Tipps zur Programmierung:`

Prüfprotokoll

Wichtig für die Fehleranalyse ist es festzustellen, welcher Nutzer wann welche Aktionen in unseren Anwendungen durchgeführt hat. Dies gilt insbesondere für mandantenfähige Anwendungen und SaaS-Services.

Folgende Ereignisse benötigen mindestens eine Protokollierung:

- Anmeldeversuche mit falschen Zugangsdaten
- Administratoranmeldungen und -aktivitäten
- Änderungen an Benutzerkonten und Berechtigungen
- Fehler in der Sitzungsverwaltung
- Fehler bei der Eingabewertprüfung
- Fehler innerhalb der Anwendung (z. B. Ausnahmen)
- Schreibzugriffe auf Daten innerhalb der Anwendung
- Herunterfahren, Starten und Neustarten von Anwendungen oder Systemen

Protokolleinträge brauchen ein einheitliches Format und nach Möglichkeit weitere Attribute, die für eine spätere Auswertung oder automatisierte Auswertung durch SIEM-Systeme (Security-Information-and-Event-Management) wichtig sind:

- Schwere (z.B. niedrig, mittel, hoch, kritisch, Sicherheit, Debuggen)
- Kunden-ID
- Zeitstempel
- Nutzer-ID
- IP-Adressen, HTTP-Methoden und angeforderte URLs (inkl. Parameter)

Bei der Protokollierung gelten die aktuellen Datenschutzbestimmungen (EU-DSGVO) und deren Anforderungen. Hierzu zählt auch das Informieren und das Anzeigen eines Datenschutzhinweises. Sensible und personenbezogene Daten bleiben bei der Protokollierung außen vor.

Dazu gehören z. B.:

- Passwörter
- Verschlüsselungs-Codes
- Gesundheitsdaten
- Finanzdaten (z. B. Bankverbindung, Kreditkartendaten)

In Ausnahmefällen und nach Klärung des rechtlichen Rahmens dürfen Protokolle sensible und personenbezogene Daten aufzeichnen, jedoch nur unter Berücksichtigung der Pseudonymisierung oder der Verwendung von kryptografisch sicheren Hash-Methoden.

Grundsatz 7: APIs und REST-Schnittstellen sichern

Häufig stellen Web-Anwendungen auch APIs sowie Teile ihrer Funktionalität über sogenannte REST-Schnittstellen bereit. Diese Schnittstellen implementieren und sichern Entwickler genauso robust wie die übrige Anwendung. Dazu gehören das Filtern und Validieren von Ein- und Ausgaben sowie das sichere und zuverlässige Abrufen und Verifizieren von Nutzern und deren Rechten. Je nachdem, welche Datenformate diese Schnittstellen verwenden, z.B. JSON oder XML, benötigen sie Filterroutinen, die spezifische Sicherheitsfunktionen für das jeweilige Format ergänzen. Wenn APIs oder REST-Schnittstellen für Fremde zugänglich und nicht vollständig und robust in Anwendungen implementiert sind, können nicht autorisierte Nutzer Daten manipulieren oder löschen. Der unbefugte Zugriff auf Daten oder Funktionen über solche Schnittstellen kompromittiert unmittelbar die Vertraulichkeit, Integrität und Verfügbarkeit der Nutzerdaten und stellt eine Bedrohung für unsere IT-Infrastruktur dar.

`Tipps für die Entwicklung:`

Um Angriffe über manipulierte Client-Anfragen oder manipulierte Parameter zu verhindern, filtern und prüfen Programmierer anhand einer Whitelist (Liste legitimer Werte) alle Daten eingehend. Eine restriktive Whitelist enthält nur zulässige Werte und Zeichenketten. Viele Frameworks bieten APIs und REST-Schnittstellen zum Filtern oder Whitelisting, häufig in Verbindung mit Scaffolding und der automatisierten Erstellung von Quellcode für APIs und REST-Schnittstellen. Entwickler prüfen sicherheitshalber erneut die vom Framework erstellten Filterregeln und Whitelists. TLS insbesondere mit PFS stellen sicher, dass Server und Client Daten, insbesondere wichtige Token, Anmelde- oder Sitzungsdaten, immer verschlüsselt austauschen und Dritte diese nicht abfangen oder manipulieren. Die Übertragungsverschlüsselung trägt damit wesentlich zur Vertraulichkeit und Integrität der Kundendaten bei.

Die Implementierung von TLS braucht ausreichend Zeit, da eine Fehlkonfiguration und andere Szenarien bewirken, dass Angreifer durch Downgrade-Angriffe auf TLS oder aufgrund fehlender Zertifikatsprüfungen trotz Übertragungsverschlüsselung Zugriff auf sensible Daten haben. Sensible Informationen wie Benutzernamen, Passwörter und Sicherheits-Token sind nicht in der URL zu codieren, damit Web-Browser-Verlauf oder unverschlüsselte Verbindungen in Serverprotokollen keine Hinweise liefern.

Grundsatz 8: Die Methoden selbst testen, bevor andere es tun

Bereits während der Entwicklung sollten Programmierer die Funktionen und Schnittstellen auf korrektes Funktionieren und ihr Verhalten bei fehlerhaften Parametern und Eingabewerten überprüfen. Ein bewährtes Verfahren für diesen Test von Modulen sind die sogenannten Unit-Tests. In der Prüfeinheit erstellt der Test für jede Funktion oder Methode eine Prüfroutine, die zunächst einen Ausgabezustand erzeugt, zum Beispiel Zuweisung von Variablenwerten, dann die zu testende Funktion/Methode aufruft und schließlich das Ergebnis, wie einen Ausgabewert mit dem erwarteten oder gewünschten Verhalten vergleicht. Es besteht die Möglichkeit, solche Unit-Tests mit Werkzeugen bei Integration in die Entwicklungs-IDE automatisch durchzuführen und auszuwerten. Entsprechende Werkzeuge und Frameworks sind für alle gängigen Programmiersprachen verfügbar.

Ohne Qualitätskontrolle oder bei unzureichenden Prüfverfahren besteht die Chance, funktionale und/oder sicherheitsrelevante Fehler in unseren Anwendungen unter Umständen zu übersehen und erst im Produktivbetrieb zu bemerken. Durch die Komplexität heutiger Programmcodes ist die Ursachenanalyse von Problemen aufgrund von Quellcodefehlern häufig sehr schwierig. Dies gilt insbesondere dann, wenn das Fehlverhalten nicht wirklich reproduzierbar ist oder über mehrere Programmkomponenten, -klassen, -module und -funktionen hinweg auftritt.

`Tipps für die Entwicklung:`

Bei der Speicherung von Daten im Programmcode oder der Verarbeitung von Ausgabewerten ist es wichtig, die einzelnen Datenfelder und Variablen daraufhin zu überprüfen, ob die Werte gültig sind und innerhalb des erwarteten Wertebereichs liegen. Es ist ratsam, für jede Funktion einen eigenen Unit-Test zu erstellen, der prüft, ob die Funktion bei fehlerhaften Eingabedaten korrekt arbeitet und sich erwartungsgemäß verhält. Für die meisten Programmiersprachen gibt es Software-Tools und Werkzeuge zur Erstellung von Unit-Tests, die sich auch in gängige Entwicklungsumgebungen integrieren lassen. Um Fehler frühzeitig zu erkennen und zu beheben, achten Softwareentwickler auf regelmäßige und automatische Unit-Tests. Sie ermöglichen eine frühe Fehlererkennung und eine schnelle Korrektur.

Grundsatz 9: Keinen externen und nicht selbst kontrollierten Quellen vertrauen

Web-Anwendungen enthalten oft Inhalte aus anderen Quellen, insbesondere von Content-Providern. Der Inhalt wird meist im

Quellcode referenziert und vom Web-Browser des Nutzers dynamisch geladen. Bei der Integration von Inhalten aus Drittquellen ist die Vertrauenswürdigkeit der Quelle zu klassifizieren und gegebenenfalls sind die von Dritten gelieferten Inhalte zu überprüfen oder zu filtern.

Dateien von externen Servern können mit Schadcode infiziert sein, der durch die Integration in Anwendungen die Computersysteme der Nutzer infiziert. In der Vergangenheit tauchten solche Fälle vor allem durch Malware in Onlinewerbung auf, die auf Grund ihrer Integration in eine Vielzahl von Websites und Anwendungen, eine große Anzahl von Computersystemen infizierte.

Wenn integrierte externe Dateien eine Bedrohung für Nutzer darstellen, kann dies auch einen Image-Schaden für den Kunden nach sich ziehen. So ist die Herkunft von Schadcode für viele Nutzer nicht direkt nachvollziehbar und so problemlos mit den Kundenanwendungen in Verbindung zu bringen. Daher gilt: Alle möglichen Sicherheitsvorkehrungen treffen und integrierte externe Inhalte und Dateien sorgfältig prüfen und nur auf vertrauenswürdige Quellen zurückgreifen.

Tipps für die Entwicklung

Vor der Integration externer Inhalte steht die Überprüfung der Zuverlässigkeit und Vertrauenswürdigkeit der Quelle:

- Stammen die Inhalte von einem bekannten Anbieter?
- Ist der Content-Provider schon lange am Markt tätig?
- Gab es in der Vergangenheit Sicherheitsvorfälle im Zusammenhang mit dem Content-Provider?

Wenn möglich isolieren Entwickler externe Inhalte von der übrigen Anwendung und statten sie mit so wenig Rechten wie möglich aus. Dies erreichen Programmierer zum Beispiel durch die Verwendung von iFrames in Verbindung mit dem Sandbox-Attribut (HTML5). Auf diese Weise nehmen Anwendungen externe Inhalte in die Ansicht auf, ohne dass der Content-Provider JavaScript ausführen, URLs aufrufen oder Cookies lesen muss. Außerdem scheint es sinnvoll, vertrauenswürdige Quellen für ausführbaren JavaScript-Code explizit zu benennen (Content-Security-Policy). JavaScript-Code aus anderen Quellen stellt dann keine Bedrohung mehr dar, denn der Web-Browser des Nutzers führt diese nicht mehr aus.

Grundsatz 10: Bei der Auswahl eines Partners die Augen offen halten

Um Funktionalitäten in Anwendungen zu implementieren, verwenden Softwareentwickler häufig Softwarekomponenten von Drittanbietern. Bei dem Gebrauch von Programmbibliotheken, Plugins und Add-Ons anderer Anbieter oder von Open-Source-Projekten sind diese auf ihre Robustheit und Sicherheit zu testen sowie die entsprechenden Lizenzbestimmungen zu überprüfen. Module externer Anbieter weisen manchmal technische Schwachstellen auf, die bei ihrer Integration neue zusätzliche Angriffsvektoren liefern. Auch wenn die eigenen Entwicklungen keine Schwachstellen aufweisen, besteht die Möglichkeit einer Kompromittierung des Systems durch unsichere Module und damit einer Gefahr für die Kundendaten.

Wenn externe Geräte eine Bedrohung für Nutzer darstellen, bedeutet dies auch einen Image-Schaden für den Kunden. So ist die Herkunft von Schadcode für viele Nutzer nicht direkt nachvollziehbar und problemlos mit den Kundenanwendungen in Verbindung zu bringen. Softwareentwickler sollten daher alle möglichen Sicherheitsvorkehrungen treffen, alle Fremdmodule vor der Verwendung überprüfen und nur solche aus vertrauenswürdigen Quellen einsetzen.

Genauso wichtig wie die Robustheit und Sicherheit von Fremdmodulen ist die Einhaltung der entsprechenden Lizenzbestimmungen:

- Darf die Anwendung das Modul verwenden?
- Welche Folgen und Verpflichtungen ergeben sich aus der Nutzung?

Tipps für die Entwicklung:

Die Vertrauenswürdigkeit der Quelle sowie die Robustheit und Sicherheit der Bibliotheken sind vom Programmierer vor der Verwendung von Fremdmodulen zu überprüfen. Erste Hinweise finden sich oft in Foren oder Mailinglisten für die Meldung von Bugs oder Fehlern:

- Wurden in der Vergangenheit viele Fehler gemeldet?
- Gibt es bekannte Sicherheitsschwachstellen?
- Wie wurde bisher mit Fehlermeldungen und der Sicherheit umgegangen?
- Stehen regelmäßige Patches und Updates bereit?
- Befindet sich das Modul noch in der Weiterentwicklung oder liegt die Herausgabe der letzten Version schon lange zurück?

Vor dem Einsatz von Fremdmodulen prüfen Entwickler mit der Compliance-Abteilung die entsprechenden Lizenzbestimmungen. Auch bei Versionsänderungen, beispielsweise nach der Aktualisierung eines Fremdmoduls ist zu testen, ob Änderungen an der Lizenz negative Auswirkungen haben. Nur die neusten und stabilsten Versionen leisten einen sicheren Schutz. Darüber hinaus binden Softwareentwickler die Module in das Patch-Management ein, sodass die Prüfung der Aktualität der Version regelmäßig erfolgt.

Quellen:

1 https://de.wikipedia.org/wiki/CRUD

#JAVAPRO #Security #JavaScript #UI

JavaScript Security – Best-Practice

JavaScript ist mittlerweile überall. Neben dem Großteil der Web- und Mobile-Apps werden inzwischen auch Server-Applikationen und selbst Anwendungen in der Automotive-Industrie damit programmiert. Auch Java-Entwickler kommen somit immer öfter mit JavaScript in Kontakt. Mit der Zahl kritischer Anwendungsgebiete steigt aber auch das Schadenpotenzial. Schwachstellen wie XSS, CSRF und SQL-Injections sind hinlänglich bekannt. Doch in vielen Umgebungen mangelt es bislang an der Awareness für das Thema Sicherheit. Dieser Artikel erklärt, wie man die Weichen für eine sichere Entwicklung mit JavaScript stellt.

Schritt 1: Validierung des Inputs

In der Web-Application-Security stellt der User-Input mit den zugehörigen Daten schon immer ein enormes Sicherheitsrisiko dar. Daher stellen die Validierung und Bereinigung des Inputs einen Eckpfeiler aller Secure-Coding-Frameworks dar. Sie sollten als Server-Funktion für alle Anwendungsebenen integriert und stets auf einem vertrauenswürdigen System (i.d.R. dem Server) vorgenommen werden. Für einen optimalen Schutz werden sämtliche User-Daten und Eingaben zunächst als unsicher klassifiziert und dürfen erst nach einem Security-Check akzeptiert werden. Da moderne Web-Anwendungen Daten aus unterschiedlichsten Quellen beziehen, etwa über User-Interfaces oder per API angebundene Drittanbieterdienste, gilt es zunächst die Vertrauenswürdigkeit der Datenquellen zu bewerten. Als Faustregel sollten alle dabei unbekannte Quellen grundsätzlich als nicht vertrauenswürdig eingestuft werden. Immer wenn Daten von einer vertrauenswürdigen an eine weniger vertrauenswürdige Quelle übergeben werden, muss über Integritätschecks sichergestellt werden, dass diese nicht manipuliert wurden. Und so gehen Sie bei der Validierung vor:

- Codieren Sie sämtliche Daten bei Erreichen des Validierungs-Servers standardmäßig in einem etablierten, einheitlichem Zeichensatz wie UTF-8.
- Stellen Sie mittels White-Listing sicher, dass ausschließlich erlaubte Zeichen verwendet werden. Vermeiden Sie soweit wie möglich Regular-Expressions, da diese über DOS-Attacken angreifbar sind. Greifen Sie lieber zu Validierungsmodulen wie validtor.js.
- Validieren Sie die Datenlängen. Auch wenn Overflows unter JavaScript selten sind, können lange Strings zu Sicherheitsproblemen führen.
- Überprüfen Sie vorhandene Sonderzeichen, sofern dies nicht bereits im Rahmen der Standardvalidierung erfolgt. Achten Sie vor allem auf Null-Bytes (%00), Zeilenumbrüche (%0d %0a \r \n) und Dot-Dot-Slash (../ und ..\ ).
- Prüfen Sie die Zahlen im Code, und zwar sowohl mit Blick auf den Wert (Number Value), den Typ (Number Type) sowie das Objekt (Number Object).

Autor:

Tom Zahov Zaubermann ist seit Mai 2019 bei Checkmarx und zeichnet in seiner aktuellen Position als Sales Engineer für die DACH-Region verantwortlich. Er ist spezialisiert auf die Bereiche Application Security und Software Exposure. Er verfügt über langjährige Erfahrung in der IT-Security-Beratung von Unternehmen und Regierungsstellen in Europa, Afrika, Vietnam und Singapur. Sein besonderes Interesse gilt außerdem dem sich schnell entwickelnden eAuto-Sektor und den damit verbundenen Sicherheitsrisiken.
Nach seiner Ausbildung in Israel war Tom Zahov Zaubermann für mehrere führende IT-Security-Unternehmen wie CYMOTIVE Technologies und CIPHRON in Deutschland tätig.

- Validieren Sie vom User übermittelte Files. Dazu gehört, einen verbindlichen Speicherort zu definieren und die Dateigröße zu begrenzen. Achten Sie darauf, dass von Usern übermittelte Files weder eingebunden noch ausgeführt werden können.

Im Falle von Web-Anwendungen sollten Sie außerdem unbedingt die Request- und Response-Header validieren, um zu gewährleisten, dass sie diese ASCII-Zeichen enthalten. Viele Sicherheitsprobleme wie das HTTP-Response-Splitting sind auf eine unzureichende Validierung und Bereinigung des Contents zurückzuführen. Denken Sie auch daran, dass HTTP-Headers ebenfalls keine vertrauenswürdige Datenquellen darstellen (z. B. Cookies).

Im Anschluss an die Validierung gilt es den Input zu bereinigen (Sanitization). Dabei werden beispielsweise die Zeichen < und > ersetzt, die im HTML-Kontext eine andere Bedeutung haben, sowie Zeilenumbrüche und Leerzeichen entfernt. Außerdem sollten Sie darauf achten, sämtliche URLs zu vereinheitlichen, u.a. ohne ..\. Die gute Nachricht ist, dass sich dieser Arbeitsschritt über Standardpakete wie validator.js weitgehend automatisieren lässt.

Schritt 2: Codierung des Outputs

Die Codierung des Outputs wird in vielen Best-Practice-Empfehlungen sehr kurz abgehandelt. Dabei kommt diesem Schritt angesichts der zunehmend komplexer werdenden Web-Anwendungen eine Schlüsselrolle zu. Die Gleichung ist einfach: Je mehr Daten aus unterschiedlichen Quellen an den Web-Browser (oder andere Medien) übergeben werden, desto mehr Ansatzpunkte haben Angreifer, um eine Injection durchzuführen. Dies lässt sich gut am Beispiel des Cross-Site-Scriptings (XSS) verdeutlichen. XSS-Angriffe, bei denen bösartiger JavaScript-Code injiziert und ausgeführt wird, gehören zu den häufigsten Schwachstellen moderner Web-Anwendungen. Dabei unterscheidet man zwei Varianten:

- Server-XSS: Schädliche Daten werden in die HTML-Response des Servers integriert.
- Client-XSS: Mithilfe eines unsicheren JavaScript-Calls werden unsichere User-Daten in das DOM injiziert.

Beide Angriffsformen lassen sich zuverlässig unterbinden, indem der Output sorgfältig codiert wird. In modernen JavaScript-Umgebungen steht den Entwicklern dafür eine Reihe von Tools zur Verfügung:

- Die mit JavaScript 1.5 implementierte Funktion Vanilla-JavaScript ersetzt problematische Sonderzeichen auf dem Server automatisch durch sichere Zeichen und verhindert so, dass mit der HTTP-Response gefährliche Payloads eingespeist werden.
- Für Node.js sind im Rahmen von npm heute dedizierte Pakete wie xss-filters verfügbar, mit denen sich UTF-8 kodierter Output zuverlässig vor Cross-Site-Scripting serverseitig schützen lässt.
- Client-seitige JavaScript-Web-Frameworks wie Angular-JS unterstützen heute ebenfalls bei der sicheren Output-Codierung, indem sämtliche in das DOM eingespeisten Werte (template, property, attribute, style, class binding, interpolation) zunächst als unsicher eingestuft und bereinigt werden.
- Die JavaScript-Softwarebibliothek React hat mit JSX eine neue Syntaxerweiterung für React-Elemente eingeführt. React ist standardmäßig so konfiguriert, dass ausschließlich im Anwendungscode hinterlegte Werte in das DOM übernommen werden können, was Client-XSS zuverlässig verhindert.

Das Beispiel XSS zeigt, wie wichtig eine zuverlässige Codierung des Outputs ist.

Schritt 3: Authentisierung und Passwortmanagement

Eine starke Authentisierung und ein robustes Passwortmanagement sind Schlüsselkomponenten jedes Entwicklungsprozesses. Auch hier gilt zunächst, dass die gesamte Authentisierung auf einem vertrauenswürdigen System umgesetzt werden sollte, typischerweise dem Server, auf dem das Backend der Anwendung läuft. Um das System so einfach wie möglich zu halten und die Zahl der Angriffspunkte zu minimieren, hat sich der Einsatz etablierter Authentisierungsdienste bewährt. Seiten und Ressourcen, die eine sichere Authentisierung erfordern, sollten diese niemals selbst bereitstellen. Denken Sie außerdem daran, nicht nur die Anwender, sondern auch Anwendungen sicher zu authentisieren, wenn diese auf externe Systeme zugreifen und sensible Daten verarbeiten. Bei der client-seitigen Implementierung der Authentisierung sollte die Eingabe stets verdeckt erfolgen. Deaktivieren Sie auch die Remember-Me-Funktion und die Autovervollständigung. Beides ist mit JavaScript sehr einfach möglich, indem Sie ein Eingabefeld mit `Typ="password"` anlegen und die Autocomplete Attribute deaktivieren.

Übermitteln Sie Zugangsdaten ausschließlich über HTTP POST Requests, und verwenden Sie stets eine verschlüsselte Verbindung (HTTPS). HTTP GET Requests über TLS/SSL (HTTPS) sehen auf den ersten Blick ebenso sicher aus, i.d.R. werden Sie die angeforderte URL aber mitloggen wollen. Achten Sie darauf, bei Anmeldefehlern nicht zu viele Informationen preiszugeben. Mit der Meldung Benutzername/Passwort ungültig können potenzielle Angreifer weniger anfangen als mit dem expliziten Hinweis Passwort muss Sonderzeichen enthalten.

Schützen Sie Authentisierungsdaten mit geeigneten kryptografischen Maßnahmen. Verwenden Sie dabei etablierte Hashing-Algorithmen wie bcrypt, PDKDF2, Argon2 und scrypt. In Node.

js sind für all diese Algorithmen robuste Implementierungen verfügbar.

Stellen Sie ausreichende Passwortstärken sicher. Die meisten Anwendungen geben nach wie vor eine Mindestlänge von acht Zeichen vor. Mindestens 16 Zeichen bieten aber deutlich besseren Schutz. Sorgen Sie auch dafür, dass die Anwender ihre Passwörter regelmäßig aktualisieren müssen. Passwörter sollten frühestens nach einem Tag geändert werden können, um zu verhindern, dass Angreifer den Reset missbrauchen. Wenn Sie mit E-Mail-basierten Resets arbeiten, sollte dieser mitgeloggt werden und Anwender nur einen temporären Passwort-Link mit kurzer Laufzeit erhalten. Achten Sie bei Sicherheitsfragen darauf, dass diese möglichst viele gleichwertige Antworten erzielen. So ist etwa die Frage nach dem Lieblingsbuch denkbar ungeeignet, weil überproportional viele Anwender Die Bibel als Antwort wählen werden.

In kritischen Umgebungen sollte stets starke Multi-Faktor-Authentisierung implementiert werden. Node.js unterstützt dafür eine ganze Reihe einfach zu implementierender Pakete. Ein gutes Beispiel ist etwa speakeasy. Integrieren Sie geeignete Monitoring-Tools, um Angriffe auf multiple Accounts schneller zu identifizieren. Auch hier werden Sie bei Node.js fündig. Dort gibt es Pakete, mit denen Sie die Zugriffsraten im Falle eines Brute-Force-Angriffs automatisch senken können. Denken Sie daran, herstellerseitige Default-Passwörter zu ändern und Zugangsdaten nicht mehr aktiver User zu löschen.

Schritt 4: Session-Management

Moderne Web-Anwendungen nutzen den nachfolgend illustrierten Session-Flow, um die User-Identität über multiple HTTP-Anfragen hinweg mit zu übergeben. Dies kann unter Sicherheitsaspekten problematisch sein und sollte daher mit gebührender Sorgfalt implementiert werden. An sich ist der Aufbau einer User-Session ganz einfach:

1. Der Client sendet einen Request.
2. Der Server generiert einen Identifier und sendet ihn mit Response zurück.
3. Der Client liest den Identifier.
4. Der Client sendet eine weitere Requests, diesmal mit dem unveränderten Identifier.
5. Der Server liest und validiert den Identifier. Der Workflow kehrt zu Schritt 2 zurück.

Der Identifier muss aus Sicherheitsgründen stets auf einem vertrauenswürdigen System (dem Server) angelegt werden. Die Anwendung sollte dementsprechend auch nur Identifier akzeptieren, die von diesem System erstellt wurden. Verwenden Sie für die Generierung ausschließlich etablierte Tools, um sicherzustellen, dass die Identifier auch wirklich zufällig generiert werden. Denken Sie auch daran, dass die Session beim Log-out vollständig terminiert werden muss und achten Sie darauf, dass Sie auf jeder autorisierten Seite einen Log-out ermöglichen.

Kommunizieren Sie Session-Identifier nicht in URLs, Log-Files oder Fehlermeldungen. Ältere Web-Frameworks und -Anwendungen übergeben den Session-Identifier mitunter als GET Parameter (oft als Parameter jsessionid oder PHPSESSID). Dies macht es Angreifern leicht, Identitäten zu übernehmen und ist heute ein klares No-Go.

Schritt 5: Access Control

Schützen Sie den Zugang zu folgenden Bereichen durch eine zuverlässige Autorisierung, um unerwünschte Zugriffe zu verhindern:

- Geschützte Files und andere Ressourcen
- Geschützte URLs
- Funktionen direkter Objekte
- Referenzdienste
- Anwendungsdaten
- User- und Daten-Attribute
- Informationen zur Policy

Dabei müssen Sie zunächst dafür sorgen, dass die Autorisierung der Zugriffe ausschließlich auf der Basis vertrauenswürdiger Objekte erfolgt. Die Autorisierungsvorgaben müssen überdies bei jeder Anfrage durchgesetzt werden – bei serverseitigen Scripts ebenso wie bei client-seitigen Technologien wie AJAX oder Flash. Achten Sie auch darauf, die privilegierten Informationen sorgfältig vom Rest des Anwendungs-Codes zu trennen.

Das Web-Application-Framework von Express für Node.js macht es Entwicklern leicht, bestimmte Zugriffe ausschließlich für autorisierte Anwender zu öffnen. Darüber hinaus ist mit dem Access-Control-Package eine leistungsfähige Suite für die Umsetzung von rollen- und attributbasierter Modelle erhältlich, mit der sich überaus granulare Zugriffe implementieren lassen.
Begrenzen Sie die Höchstzahl von Transaktionen, die Ihre Anwender in einer vorgegebenen Zeitspanne durchführen können, um DoS-Angriffe zu verhindern. Kontrollieren Sie in festen Zeitabständen, ob sich die Privilegien der User geändert haben und passen Sie die Rechte zeitnah an.

Schritt 6: Kryptografie

Bei der Entwicklung in JavaScript-Umgebungen kommen zwei unterschiedliche kryptografische Verfahren zum Einsatz: Hashing und Verschlüsselung. Ein Hash ist ein String (oder eine Zahl) fester Länge, der mithilfe einer Hash-Funktion aus Quelldaten generiert wird, wobei sich die Quelldatei nicht mehr wiederherstellen lässt. Damit eignet sich das Verfahren um beispielsweise Dateiidentitäten nach der Validierung sicher zu dokumentieren. Der meist genutzte Hashing-Algorithmus ist MD5. BLAKE2 gilt

unter Sicherheitsgesichtspunkten als etwas stärker und flexibler, wird in Node.js aber kaum unterstützt.

Bei der Verschlüsselung werden lesbare Daten mithilfe eines Schlüssels in unlesbare Daten umgewandelt. Im Gegensatz zum Hash lassen sie sich aber wieder entschlüsseln. Verschlüsselung ist damit erste Wahl, wenn sensible Daten oder Kommunikationsströme geschützt werden sollen. Dabei unterscheidet man zwei Verfahren:

- Symmetrische Verschlüsselung (etwa via AES), die sich einfach und sicher über Tools wie node-forge implementieren lässt.
- Asymmetrische Verschlüsselung, die eine zuverlässige Authentisierung ermöglicht. Diese lässt sich etwa über das Node.js-Package sodium implementieren.

Achten Sie bei der Implementierung der Kryptografiepakete darauf, geeignete Policies und Verfahren für das Management der Schlüssel zu etablieren und die Master-Secrets vor unerwünschten Zugriffen zu schützen. Achtung: Ihre Keys dürfen niemals im Quellcode hinterlegt sein!

Schritt 7: Fehlerbehebung und Logging

Fehlerbehebung und Logging sind für sichere Anwendungen und Infrastrukturen unverzichtbar. Die Fehlerbehebung dient dazu, Fehler in der Anwendungslogik frühzeitig zu identifizieren und zu beheben. Beim Logging geht es darum, Abläufe und Anfragen zu protokollieren und daraus geeignete Schutzmaßnahmen abzuleiten.

1. Fehlerbehebung

JavaScript unterstützt einen Error Constructor, der Error-Objekte generiert. Typischerweise geschieht das, wenn es in der Laufzeitumgebung zu einer Ausnahme kommt. Man unterscheidet dabei zwischen Betriebs- und Programmfehlern. Betriebsfehler sind Fehler, die im laufenden Betrieb auftreten. Das bedeutet, dass die Anwendungslogik korrekt ist, aber ein unerwartetes Event auftritt, z.B. ein Input-/Output-Fehler, ein Netzwerkausfall oder ein Speicherproblem. Programmierfehler sind logische Fehler im Anwendungscode. Beispiele sind Zugriffe auf undefinierte Variablen, falsche Typzuweisungen und ähnliches mehr. Node.js kennt drei Arten mit Fehlern umzugehen:

- Fehlermeldung (Exception).
- Übergabe des Fehlers an einen **`callback()`**, eine dedizierte Funktion für das Handling von Fehlern und asynchronen Betriebsabläufen.
- Verwendung eines EventEmitters.

Als Entwickler müssen Sie den Unterschied zwischen einem Fehler (Error) und einer Exception kennen. Fehler werden erfasst und können dann an eine andere Funktion übergeben oder angezeigt werden. Sobald ein Fehler angezeigt wird, wird er zur Ausnahme. In einfachem synchronem JavaScript sieht die Syntax einer Exception wie folgt aus: throw new Error (‚Some error.');

Die Anzeige solcher Exceptioins ist bei JavaScript oder Node.js aber eher die Ausnahme. Die meisten nativen APIs übergeben Fehlermeldungen an die Callback-Funktion. Die Verwendung von **`callbacks()`** ist auf die asynchrone Natur von JavaScript zurückzuführen. Da bei diesem Verfahren aber keine explizite Fehlermeldung erfolgt, birgt es die Gefahr, dass Fehler unentdeckt bleiben und nicht behoben werden können. Das mit ES6 eingeführte Paket Promises hat das Error-Handling in JavaScript-Umgebungen in vielen Bereichen verbessert und hilft insbesondere dabei, Callback-Pyramiden zu vermeiden. Dennoch sollten Entwickler in event-orientierten Programmierumgebungen oder stream-basiertem Code zusätzlich einen dedizierten Error-Handler verwenden.

2. Logging

Das Logging sollte stets in der Applikation erfolgen. Implementieren Sie dafür eine Master-Routine auf einem vertrauenswürdigen System und stellen Sie sicher, dass die Logs keine sensiblen Daten enthalten. Für JavaScript-Umgebungen ist eine breite Palette guter Logging-Tools verfügbar, darunter das beliebte winston-package (für das mit dem winston-daily-rotate-file-package sogar ein Tool für rotierende Logs bereitsteht) sowie die Pakete log4js, bunya und morgan. Der Entwickler sollte zudem geeignete Tools für die Log-Analyse implementieren, um zu verhindern, dass über das Analyse-Interface nicht vertrauenswürdige Daten als Code ausgeführt werden. Hierfür sind bewährte Open-Source-Tools der ELK-Stack (Elasticsearch, Logstash, Kibana) verfügbar. Um die Integrität der validierten Log-Files dauerhaft sicherzustellen, sollten die Dateien überdies durch Hashing geschützt werden.

Schritt 8: Datensicherheit

Die Daten in den Web-Anwendungen müssen zuverlässig geschützt werden. Der erste Schritt ist es dabei, geeignete Rechte und Rollen zu definieren und sicherzustellen, dass jeder User nur auf die Funktionen zugreifen kann, die er auch wirklich benötigt. Anschließend muss für jeden Anwender passende Rollen zugewiesen werden. Achten Sie darauf, sensible Informationen die nicht benötigt werden zeitnah zu löschen. Dies betrifft Daten in temporären Verzeichnissen und Caches, aber auch Kommentare im Quellcode. Vermeiden Sie es, Informationen über das HTTP-GET-Verfahren weiterzugeben. Das macht die Anwendung angreifbar, vor allem wenn die URL unbefristete Session-IDs oder Token enthält. Außerdem könnten Ihre Daten über MITM-Attacken abgefangen werden. Wenn Ihre Web-Anwendung versucht, Informationen über Ihren API-Key auf Drittanbieterseiten abzurufen, könnte dieser nicht nur von Insidern gestohlen werden. Nutzen Sie daher immer HTTPS, übergeben Sie Paramater über

POST und verwenden Sie möglichst Einmal-IDs und Einmal-Token. Achten Sie auf eine lückenlose Verschlüsselung kritischer Daten und speichern Sie diese niemals in Klartext auf dem Client. Hierzu gehört auch die Einbettung in unsicheren Formaten wie Adobe-Flash oder in kompiliertem Code. Definieren Sie abgestufte Rechte für den Zugriff auf Code und beschränken Sie die Zugriffsoptionen für den Quellcode. User dürfen niemals in der Lage sein, den server-seitigen Quellcode einzusehen oder herunterzuladen.

Schritt 9: Sicherheit der Kommunikation

Als Entwickler müssen Sie jederzeit die Gewissheit haben, dass Ihre Kommunikationskanäle sicher sind. Die betrifft die Kommunikation zwischen Server und Client, zwischen Server und Datenbank sowie die gesamte Backend-Kommunikation. Verschlüsseln Sie diese Kommunikationskanäle, um die Integrität und Vertraulichkeit der Daten zu gewährleisten und einen Man-in-the-Middle-Angriff zu verhindern. Verwenden Sie dafür einfach das tls-module von Node.js. Es hat sich bewährt durchgängig über HTTPS zu kommunizieren, statt immer wieder zwischen HTTP und HTTPS zu wechseln. Für einen optimalen Schutz vor bekannten Attacken wie POODLE und BEAST sollten Sie darüber hinaus SSLv3 und TLSv1 deaktivieren.

Schritt 10: Systemkonfiguration

Regelmäßige Updates sind der Schlüssel zu nachhaltiger Sicherheit. Entwickler sollten daher sicherstellen, dass ihre JavaScript-Frameworks, ihre externen Pakete und die Frameworks der Web-Anwendungen durchgehend auf dem neuesten Stand sind. Tools wie die Node-Security-Platform (nsp) helfen, bekannte Sicherheitslücken zu vermeiden und die Pakete Up-to-Date zu halten. Achten Sie darauf, dass die Directory-Listings stets deaktiviert sind, um Angreifern die Suche nach sensiblen Files nicht unnötig zu erleichtern. In den Frameworks Node.js und Express sind diese Listings standardmäßig deaktiviert. Entfernen Sie in der Produktivumgebung alle unnötigen Funktionalitäten des HTTP-Headers. Überprüfen Sie dabei auch den HTTP-Response-Header, und bereinigen Sie sensible Informationen wie die OS-Version, die Webserver-Version, das Framework und die Programmiersprache.

Nutzen Sie das helmet-module, um geeignete sichere HTTP-Header in Ihrer Web-Anwendungen einzurichten. Definieren Sie dabei am besten unterschiedliche http-Header, um die Sicherheit der Web-Anwendung zu verbessern.

Schritt 11: Database Security

Bevor Sie eine Datenbank an Ihre JavaScript-Anwendung anbinden, sollten Sie die folgenden Konfigurationsschritte vornehmen:

- Sichere Installation des Datenbankservers
- Vergabe eines sicheren Root-Passworts
- Beschränkung der Root-Zugriffe auf den Local-Host
- Entfernung aller anonymen User-Accounts
- Entfernung eventueller Testdatenbanken
- Entfernung aller unnötigen Komponenten und Herstellerinhalte
- Installation des minimalen für JavaScript erforderlichen Feature-Sets
- Deaktivierung eventueller Default-Accounts

Um Ihre Connection-Strings optimal zu schützen, sollten die Anmeldedaten in einer dedizierten Konfigurationsdatei fernab öffentlicher Zugriffe hinterlegt werden. Definieren Sie abgestufte Rechte für jeden Vertrauenslevel: User, Read-Only-User, Gast und Admin. Zugriffe auf die Datenbank sollten stets mit minimalen Rechten erfolgen. Wenn Ihre Web-Anwendung lediglich Daten auslesen soll, sollten Sie zu diesem Zweck einen Read-Only-User anlegen. Passen Sie die Rechte an, wenn sich die Anforderungen ändern. Verwenden Sie ein starkes Passwort. Verwenden Sie das npm-OWASP-Password-Strength-Testpaket, um die Passwortstärke zu validieren. Entfernen Sie alle Default-Admin-Passwörter – diese stellen ein erhebliches Sicherheitsrisiko dar. Arbeiten Sie mit Prepared-Statements und Parameterized-Queries, um SQL-Injections zu verhindern. Stored-Procedures, also vordefinierte Ansichtsoptionen für die verschiedenen Rollen und Rechte, helfen Ihnen durchgängig zu regeln, wer auf welche Informationen zugreifen kann.

Schritt 12: File Management

Lediglich authentisierte User sollten Dateien hochladen können. Regeln Sie im Vorfeld außerdem, welche Dateitypen und Dateigrößen zulässig sind und implementieren Sie geeignete Tools für die Validierung der Parameter. Von Usern hochgeladene Files sollten stets auf einem isolierten Content-Server oder in einer Datenbank ohne Rechte zur Ausführung installiert werden. Wenn die User-Uploads auf einem Nix- Server gehostet werden, sollten Sie zusätzliche Security-Mechanismen wie chroot vorsehen. Bei der Verwendung dynamischer Umleitungen dürfen Anwenderdaten nicht übergeben werden. Wenn die Anwendung dies erfordert, müssen zum Schutz der App zusätzliche Vorkehrungen getroffen werden, etwa das Ablehnen von nicht validierten Daten und URLs mit relativer Pfadangabe. Vergeben Sie für den Server nur Leserechte. Scannen Sie hochgeladene Files auf Malware.

Fazit:

JavaScript hat sich in vielen Bereichen als flexibler und unkomplizierter de facto Standard etabliert. Allerdings bietet JavaScript eine Vielzahl von Angriffspunkten mit enormem Schadenspotenzial. Unternehmen sind daher gut beraten, sich bei der Entwicklung an etablierte Secure-Coding-Practices zu halten und für eine optimale Softwarequalität gegebenenfalls eine leistungsfähige Softwareanalyseplattform zu implementieren.

#JAVAPRO #Security #Serialization #Persistence

Horror Serialisierung

Serialisierung ist das die größte Schwachstelle in Java. Praktisch alle wichtigen Java-APIs und Frameworks verwenden Serialisierung. Wirksamer Schutz ist kaum möglich und vorhandene Alternativen lösen die fundamentalen Probleme nicht. Mit MicroStream gibt es jetzt eine neue Serialisierung für Java, die nicht nur mehr Sicherheit verspricht, sondern zudem fantastische Möglichkeiten bietet.

In objektorientierte Programmiersprachen wie Java arbeiten wir mit Objekten und komplexen Objektgraphen. Um Daten zwischen Java-Anwendungen austauschen zu können, werden diese über Netzwerk übertragen. Dazu müssen die Objekte in eine für die Übertragung geeignete Form umgewandelt werden, und zwar in einen binären Datenstrom (Byte-Stream). Dieser Umwandlungsprozess wird als Serialisierung bezeichnet.

Autor:

Sebastian Späth ist Senior Java Developer und Technology Evangelist bei der XDEV Software Corperation. Er verfügt über eine breite Berufserfahrung und war mit Rapiclipse an der Entwicklung einer eigenenEclipse-Distribution beteiligt. Sebastians Schwerpunkte sind Rapid Development und Rapid Prototyping.

https://www.linkedin.com/in/sebastian-späth-580902149/
https://www.xing.com/profile/Sebastian_Spaeth11

Umgekehrt wird das Empfangen des Byte-Stroms und anschließende Erzeugen eines Objekts als Deserialisierung bezeichnet.

Im Juni diesen Jahres hat sich Brian Goetz, Java Language Architekt bei Oracle, mit einem sehr ausführlichen Beitrag[1] zur Java-Serialisierung geäußert. Sinngemäß schreibt er: Es ist paradox, denn auf der einen Seite hat die Serialisierung viel zu dem großen Erfolg von Java beigetragen, insbesondere dem Erfolg von Java-EE. Auf der anderen Seite aber hat man bei der Serialisierung nahezu jeden Fehler gemacht, den man sich vorstellen kann, wodurch die Serialisierung sich wie ständiger Ballast auswirkt, der hohe Sicherheitsrisiken birgt, zu höheren Wartungskosten und letztendlich verlangsamten Entwicklungsfortschritt führt. Die Vielfalt der Sicherheitslücken und die Raffinnesse mit der diese ausgenutzt werden, ist erstaunlich. Gewöhnliche (Anwendungs-)Entwickler sind damit überfordert, denn selbst Sicherheitsexperten würden trotz intensiver Überprüfung vorhandene Schwachstellen übersehen. Es ist einfach viel zu schwierig, die Serialisierung ausreichend abzusichern, da diese größtenteils auf Low-Level-Ebene eingesetzt wird. Damit ist sie für Anwendungsentwickler völlig unsichtbar, die deshalb dem Irrglauben

unterliegen, ihre Geschäftslogik wäre sicher, während die Hintertür weit offensteht und auch noch völlig unbewacht ist.

Das Konzept der Serialisierung selbst ist grundsätzlich nicht verkehrt, das große Problem ist jedoch, wie die Serialisierung damals in Java 1.1 umgesetzt wurde.

Auch Mark Reinhold, Chefarchitekt der Java-Plattform bei Oracle, wurde bereits wiederholt in Interviews sogar noch deutlicher, Zitat: „Serialization was a horrable Mistake. About 50 Percent of all Vulnerabilities in Java are linked to Java Serialization."

Starker Tobak! Wenn sich die beiden führenden, für die Weiterentwicklung von Java verantwortlichen Architekten bei Oracle so deutlich über ein Feature in Java äußern, dann scheint hier etwas gewaltig schief gelaufen zu sein. Grund genug sich einmal näher mit dem Thema auseinanderzusetzen.

Serialisierung ist sehr einfach

Der Einsatz von Serialisierung ist sehr einfach und es gibt unzählige Anwendungsfälle, weshalb diese Funktion seit 25 Jahren an allen Ecken und Enden verwendet wird. Um eine Klasse serialisieren zu können, muss diese lediglich das Interface java.io.Serializable implementieren, das der Klasse das Serialisierungsverhalten übermittelt. Für die Umwandlung eines Objektes in einen Byte-Stream ist die Klasse **ObjectOutputStream** zuständig. Für den Transfer des Datenstroms kann man dann jede beliebige Variante wählen, die in Java zur Verfügung steht, i.d.R. Sockets oder RMI.

(Listing 1)

```
MyEntity e = …
OutputStream os = socket.getOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(os);
oos.writeObject(e);
```

Java-Anwendungen, die Daten empfangen möchten, müssen den ankommenden Byte-Stream einlesen und daraus wieder äquivalente Objekte erzeugen (Deserialisierung), was mit der Klasse **ObjectInputStream** erfolgt.

(Listing 2)

```
InputStream is = socket.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
MyEntity e = (MyEntity)ois.readObject();
```

Daten-Persistierung

Neben der Übertragung von Informationen an eine andere Java-Anwendung ermöglicht die Serialisierung auch das Schreiben (Listing 3) und Lesen (Listing 4) in bzw. aus einer Datei oder Datenbank. Dafür sind die Klassen **FileOutputStream** und FileInputStream zuständig.

(Listing 3)

```
MyEntity e = …
FileOutputStream fos = new FileOutputStream(path);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(e);
```

(Listing 4)

```
FileInputStream fis = new FileInputStream(path);
ObjectInputStream ois = new ObjectInputStream(fis);
MyEntity e = (MyEntity)ois.readObject();
```

Was macht Java-Serialisierung zu einem potentiellen Sicherheitsrisiko?

Um auf der Empfängerseite aus dem Datenstrom wieder Objekte erzeugen zu können, werden neben den Daten auch die Klassendefinitionen mit verschickt. Anhand des Namens versucht Java diese Klasse dann im Klassenpfad zu finden und führt automatisch deren **readObject** Methode aus, um den Byte-Strom zu deserialisieren. Alle Java-Klassen, die sich im Klassenpfad befinden und **java.io.Serializable** implementieren, werden auf diese Weise vom Empfänger vollautomatisch deserialisiert. Dieses Verhalten kann generisch zur Ausführung von Schadcode ausgenutzt werden. Angreifer, die die Klassendefinition eines serialisierten Objektes kennen, können diese durch Man-in-the-Middle verändern und auf diese Weise Angriffe gegen den Server durchführen, um z.B. Schadcode zu injizieren. Das Problem ist bereits seit vielen Jahren bekannt.

Durch die Schwachstellen in der Java-Serialisierung werden häufig Bugs in Open-Source-Frameworks ausgenutzt, die wir alle täglich bei der Anwendungsentwicklung verwenden. Es ist regelrecht beängstigend, wie leicht es ist, diese Sicherheitslücke auszunutzen. Es gibt dafür sogar Codegeneratoren im Internet wie ysoserial. Das Tool ermöglicht es, angreifbare Klassen zu missbrauchen, sodass sich damit Schadcode ausführen lässt.

Wo wird Serialisierung eingesetzt?

In Java wird Serialisierung aber praktisch überall verwendet, selbst da wo sie niemand vermutet, weil sie tief unter der Haube zum Einsatz kommt und nicht sichtbar ist – in allen möglichen Dependencies die wir tagtäglich verwenden, ohne dass wir uns darüber Gedanken machen. Bereits in Java 8 wurde das Interface **java.ioSerializable** über 5.000 Mal implementiert. U.a. wird Serialisierung verwendet in JPA (Java-Persistence-API) und damit automatisch von jeder einzelnen Datenbankapplikation wir auf Basis des Java-Standards entwickeln. Des Weiteren in CDI (Contexts-and-Dependency-Injection), das praktisch alle großen Java-Frameworks wie Java-EE, Spring, etc. unterstützen. Ebenso in RMI (Remote-Method-Invocation), JMX (Java-Management-Extensions) und sogar in HTTP-Cookies und was besonders

interessant ist: auch in REST-Services. Intern wird Serialisierung von allen wichtigen Java-Basistechnologie verwendet, u.a. WebLgoic, JBoss, WebSphere, Jenkins, Struts, Spark usw. Alle darauf aufsetzenden Unternehmensanwendungen enthalten damit automatisch die mit Java-Serialisierung verbundenen Sicherheitslücken.

Zunehmende Angriffsfläche durch Microservice-Architektur

Die potentielle Angriffsfläche wird sogar immer größer, denn heutzutage muss praktisch jede Java-Anwendung permanent Daten übertrage. Früher wurden Java-Programme noch monolithisch, also in einem Stück konzipiert. Die verschiedenen Anwendungsmodule hingen in einer einzigen JAR zusammen und die gesamte Kommunikation zwischen den Modulen fand intern auf derselben Maschine statt. Nur mit externen Applikationen mussten Daten über Netzwerk ausgetauscht werden. Heute geht der Trend immer stärker weg vom Monolithen hin zu einer Microservice-Architektur. Eine Anwendung setzt sich jetzt aus einer Vielzahl an Kleinstprogrammen zusammen, die nicht nur völlig autark voneinander lauffähig, sondern zudem hochverfügbar sein sollen, sodass jeder einzelne Service auch noch redundant laufen muss. Dazu werden die einzelnen Microservices containerisiert und via Cloud über mehrere Rechenzentren, Regionen und sogar Erdteile verteilt. Folglich erfolgt auch die gesamte interne Kommunikation zwischen den einzelnen Microservices einer Anwendung über das Netzwerk, wodurch die Anzahl potentieller Einfallstore rapide zunimmt.

Wie kann man sich schützen

Es gibt mehrere Möglichkeiten wie man sich vor Exploits schützen kann. Black- oder Whitelisting von Klassen wird dabei mit am häufigsten genannt. Blacklisting ist enorm aufwändig und in der Praxis kaum lückenlos realisierbar, da man alle kritischen Klassen berücksichtigen und seine Blacklist permanent Up-to-Date halten muss. Whitelisting, also nur Klassen zulassen, die man tatsächlich verwendet, ist ebenfalls aufwändig, weil man alle serialisierbaren Klassen sämtlicher Dependencies mit in die Whitelist aufnehmen müsste. In der Praxis kommen beide Verfahren deshalb kaum ausreichend zum Einsatz, entweder weil sich die Entwickler der Gefahr gar nicht bewusst sind, der Wartungsaufwand viel zu groß erscheint (und vom Kunden ohnehin nicht bezahlt wird) und weil die meisten Entwickler schlicht und einfach selbst noch nie betroffen waren. Ähnlich wie bei gefährlichen Krankheiten, schätzen die meisten Menschen ein Infektionsrisiko als sehr gering ein, wenn sie vorher selbst noch nie davon betroffen waren. Der berühmt gewordene Fall des US-amerikanischen Finanzdienstleistungsunternehmen Equifax sollte jedoch als Warnung dienen und zeigt was passieren kann, wenn Hacker vorhandenen Schwachstellen ausnutzen. 2017 hatten Hacker eine Sicherheitslücke in dem bereits genannten Open-Source-Framework Apache-Struts ausgenutzt, um die Daten von 143 Millionen Kunden zu erbeuten. Die Entwickler bei Equifax haben es trotz vorliegender Informationen über das Sicherheitsleck in Struts verpasst, den bereits verfügbaren Patch rechtzeitig einzuspielen.

Weitere Einschränkungen

Java-Serialisierung hat nicht nur gravierende Sicherheitslücken. Was aus Entwicklersicht sogar noch schwerer wiegt, sind die zahlreiche Einschränkungen der Java-Serialisierung, welche die Entwicklung unnötig kompliziert machen und deshalb teuer sind. Wie bereits erwähnt, müssen Klassen **`java.io.Serializable`** implementieren, um serialisierbar zu sein. Das ist dann problematisch, wenn man eine Third-Party-API verwenden möchte, die Klassen enthalten, die das Interface nicht implementieren und folglich nicht serialisierbar sind.
Eine erhebliche Einschränkung ist zudem, dass Objektgraphen immer vollständig serialisiert werden, ohne dass sich einzelne Instanzen als Identitäten ansprechen lassen. Gezielt nur einzelne Objekte eines Objektgraphen zu serialisieren, falls nicht mehr Informationen übertragen werden müssten, ist nicht möglich. Dadurch entsteht häufig enormer, völlig überflüssiger Overhead, was die Serialisierung sehr ineffizient macht.

Bei der Deserialisierung werden immer neue Objektinstanzen erzeugt, sprich Objektkopien, deren Weiterverarbeitung sehr umständlich ist. Objektgraphen auf der Empfängerseite punktuell upzudaten, ist nicht möglich. Da immer nur ganze Objektgraphen serialisiert werden, führt der mitgeschleifte Overhead auf der Empfängerseite zu vielfach höheren Speicherverbrauch als nötig wäre.

Zur Datenspeicherung ist die Java-Serialisierung daher nur für wenige Anwendungsfälle brauchbar, z.B. wenn es um die Speicherung von Meta- oder Bulkdaten geht. Ohne Update-Funktion ist sie als DBMS-Ersatz dagegen völlig unbrauchbar.

Hinzu kommt, dass es keine Versionierung der verwendeten Klassen gibt, was bei jeder Änderung manuellen Anpassungsaufwand verursacht und zudem eine große potentielle Fehlerquelle darstellt.

Welche Alternativen gibt es?

Es existiert eine Vielzahl an Serialisierungs-Libraries, die auch Brian Goetz in seinem Beitrag aufgeführt hat, u.a. Arrow, Avro, Bert, Blixter, Bond, Capn Proto, CBOR, Colfer, Elsa, Externalizor, FlatBuffers, FST, GemFire PDX, GSON, Hessisch, Ion, Jackson, JBoss-Marshalling, JSON.simple, Kryo, Kudu, Blitz, MessagePack, Okapi, ORC, Paranoid, Parcelable, Parkett, POF, Portable, Protokoll Puffer, Protostuff, Quickser, Reflect, Seren, Serial, Simple, Simple Binary Encoding, SnakeYAML, Stephenerialization, Thrift, TinySerializer, Travny, Verjson, Wobly, XSON, XStream, YamlBeans.
Die meisten dieser Libraries zielen jedoch lediglich darauf ab, anstelle des binären Formats der Java-Serialisierung ein anderes Format zu verwenden, z.B. JSON, und dieses effizienter und flexibler verarbeiten zu können. Unter der Haube kommt meist sogar die originale Java-Serialisierung zum Einsatz, sodass die

fundamentalen Probleme der Java-Serialisierung, insbesondere die aufgeführten Einschränkungen damit weiterhin ungelöst bleiben und auf Grund ihrer Eigenheiten zum Teil ganz Probleme verursachen.

Serialisierung mit REST und JSON

Inzwischen hat sich die Kommunikation via REST und JSON zum de facto Standard entwickelt. Der Vorteil von JSON ist ähnlich wie bei XML vor allem, dass die Datenübertragung damit nicht nur zwischen Java-Anwendungen, sondern programmiersprachenunabhängig zwischen beliebigen Programmen und Diensten erfolgen kann, zum Beispiel mit einem in PHP geschriebenem Webservice.

Der große Nachteil von Datenstrukturen wie JSON und XML ist jedoch, dass diese sich nicht für die Abbildung beliebiger Objektgraphen eignen – die wir jedoch in Java ständig verwenden – und deshalb nicht wirklich zum Objektmodell von Java passen. Selbst simple Zirkelbezüge sind damit nicht abbildbar. Sämtliche Referenzen zwischen den Objekten werden gebrochen. Bei größeren Datenmengen kommt es zu einem enormen Overhead. Auch die Weiterverarbeitung der eingehenden Daten ist sehr aufwändig, da aus dem JSON-Code erst wieder Java-Objekte erzeugt werden müssen. Das Parsen umfangreicher JSON-Strings ist zudem mit Performanceeinbußen verbunden. Die bei der Java-Serialisierung aufgeführten Einschränkungen treffen bei der Serialisierung mit JSON analog zu.

Was plant Oracle?

Obwohl die Chefarchitekten bei Oracle regelmäßig mit drängenden Fragen zur Java-Serialisierung konfrontiert werden, wann man dieses Problem angehen wird, gibt es seitens Oracle auch weiterhin keine konkreten Aussagen. Brian Goetz hat zumindest einen Beitrag veröffentlicht, indem er seine persönlichen Gedanken skizziert, wie er sich eine bessere Serialisierung vorstellt. Dabei betont er die Wichtigkeit, dass sich die Serialisierung der Zukunft sich homogen in das Objektmodell von Java einfügen müsse. Auf der anderen Seite denkt er über eine Abkehr von der Serialisierung komplexer Objektgraphen und Beschränkung auf die Serialisierung reiner Daten nach, um die Komplexität zu reduzieren. Dieses Szenario ginge in die Richtung von JSON und XML und würde demnach nicht besonders gut zum Objektmodell von Java passen. Bereits zu Beginn des Textes heißt es, dass es sich lediglich um einen Entwurf, nicht um eine offizielle Spezifikation für die Java-Plattform handelt.

Mark Reinhold spricht in Interviews von einem Serialisierungs-Framework, das eine einheitliche Implementierung von Serialisierung ermöglichen soll, mit dem Entwickler auf einfache Weise den Serialisierungs-Provider ihrer Wahl ähnlich einem Plugin einbinden können – neben sämtlichen Serialisierungs-Frameworks zur Not auch die alte Java-Serialisierung. Die Beschreibung deutet auf eine standardisierte API ähnlich JPA hin. Damit wären Entwickler in der Lage, bei Bedarf an nur einer Stelle des Programms schnell und einfach das Serialisierungs-Framework zu wechseln. Von der Entwicklung einer neuen Implementierung, die die fundamentalen Probleme der bisherigen Java-Serialisierung löst, ist jedoch nicht die Rede. Wann und ob Oracle überhaupt in naher Zukunft das Thema Serialisierung angehen wird, dazu wollte sich Mark Reinhold nicht verbindlich äußern.

Eine neue Serialisierung für Java

Wie so oft ist die Java-Community sehr viel schneller mit Lösungsansätzen, die in der Praxis funktionieren und bereits erprobt sind – so jetzt auch in Sachen Serialisierung: MicroStream hat jetzt eine von Grund auf völlig neu entwickelte Serialisierung für Java veröffentlicht, die nicht nur sicher sein soll, sondern vor allem nahezu alle gravierenden Einschränkungen der Java-Serialisierung beheben soll und vom Hersteller deshalb vielversprechend als Next-Generation-Java-Serialization bezeichnet wird.

Die Entwicklung von MicroStream wurde 2013 im Zuge eines Großprojekts für die Möbelindustrie unter dem Projektnamen "Jetstream" gestartet. Seit 2015 läuft die Technologie im Produktivbetrieb als technische Basis eines Transaktionssystems, über das die Möbelindustrie ein jährliches Umsatzvolumen im Milliarden-Euro-Bereich abwickelt. Anfang des Jahres wurde das Projekt in MicroStream umbenannt und das im April erst neu gegründete Unternehmen hat schnell Investoren gefunden, die einen größeren Millionenbetrag investiert haben, um die Weiterentwicklung voranzutreiben und die Unterstützung der Java-Community zu gewährleisten.

Mit MicroStream Version 2.0 ist nun das Final-Release frei verfügbar unter www.microstream.one. MicroStream ist eine leichtgewichtige Java-API. Der Download erfolgt via Maven. Als Minimalanforderung wird lediglich JDK 8 angegeben. Ansonsten hat die Library keine Abhängigkeiten.

Mit MicroStream lassen sich beliebige Java-Klassen serialisieren, ohne dass diese ein Interface implementieren oder von einer Superklasse ableiten müssen. Auch Annotations sind nicht nötig. Damit lassen sich auch beliebige Klassen von Third-Party-APIs problemlos serialisieren. Klassen, die serialisierbar sein sollen, werden in ein Dictionary aufgenommen, das bei der Deserialisierung mit der Klassendefinition des Empfängers verglichen wird.

MicroStream wurde von Anfang an konzipiert, um komplexe Objektgraphen zu serialisieren, sollte sich elegant in das Objektmodell von Java einfügen und sich wie ein nativer Teil von Java anfühlen.

Mit MicroStream lassen sich jetzt erstmals einzelne Instanzen als Entities ansprechen und gezielt einzelne Objekte eines Objektgraphen serialisieren; die dann auf der Empfängerseite im Objektgraphen gezielt upgedatet werden. Dadurch wird kein unnötiger Daten-Overhead mehr übertragen, was MicroStream hoch effizient macht. Umständlich verarbeitbare Objektkopien gibt es nicht mehr, was gleichzeitig den Speicherverbrauch deutlich reduziert. Dadurch ergeben sich völlig neue Möglichkeiten.

Kommunikation und Datenspeicherung

MicroStream bietet nicht eine Lösung für Alles, sondern bietet für die beiden Anwendungsfälle Kommunikation und Persistierung (Datenspeicherung) jeweils eine optimierte Lösung. Darunter verbirgt sich jeweils dieselbe Serialisierung (MicroStream-Core).

Kommunikation zwischen Objektgraphen

Für die Datenübertragung zwischen zwei oder mehreren Java-Anwendungen stellt MicroStream einen Convenience-Layer zur Verfügung, der eine effiziente Objektgraphkommunikation ermöglicht (Host und Clients).

Persistierung von Java-Objektgraphen (Java-Native-Data-Store)

Die Möglichkeit, Objektgraphen punktuell updaten zu können, macht MicroStream zu einer leistungsfähigen Storage-Engine für das Speichern, Laden und Updaten nativer Java-Objektgraphen, die den Einsatz externer DBMS völlig überflüssig macht.

Bei der Persistierung werden standardmäßig nur die im Objektgraphen geänderten Objekte persistiert. Die persistente Speicherung des Deltas in einer Datei erfolgt immer anhängend (Append). Schlägt die Persistierung fehl, werden die angehängten Daten wieder verworfen, was grundsätzlich mit einer zurückgerollten Transaktion in RDBMS vergleichbar ist.

Lazy-Loading

Zum Laden von Daten ermöglicht MicroStream Objektreferenzen bei Bedarf lazy zu laden (Lazy-Loading). Damit können Objektgraphen mit einem Datenvolumen im Terabyte-Bereich persistent gespeichert werden, während sich jeweils nur ein kleiner Teil der Daten im RAM befindet.

Klassen-Versionierung und Garbage-Collection auf Dateiebene

MicroStream bietet eine vollautomatisierte Klassen-Versionierung. MicroStream erkennt zur Laufzeit automatisch, ob sich bei einer Klasse etwas geändert hat und aktualisiert die persistenten Daten automatisch. Standardtypen werden automatisch gehandelt. Für komplexere Typen lassen sich Mapping-Rules definieren.

Objektreferenzen, die über den Objektgraphen nicht mehr erreichbar sind, werden bekanntlich vom Garbage-Collector der JVM automatisch gelöscht. Bei erneuter Persistierung des Objektgraphens werden die zu löschenden Referenzen auch in der persistenten Datei markiert. Damit diese Daten dann auch tatsächlich aus der Datei entfernt werden, bietet MicroStream eine Garbage-Collection-Funktion, die auf Dateiebene operiert und in einem eigenen Thread die Storage optimiert.

Datenbankabfragen in Mikrosekunden

Für Abfragen ist keine neue Abfragesprache nötig. Die Daten befinden als Objektgraph im Hauptspeicher, der sich mit Hilfe von Java-8-Streams effizient durchsuchen lässt. Das Durchsuchen selbst komplexester Objektgraphen dauert in-memory i.d.R. nur Mikrosekunden und ist damit bis zu 1.000 Mal schneller als klassische Datenbanksysteme deren Abfrageergebnisse im Bereich von Millisekunden liegen. Mit Hilfe paralleler Streams lässt sich die Performance nochmals steigern. Vielfach ausgeführte Streams-Operationen werden nochmals vom JIT-Compiler der JVM optimiert.

Java In-Memory-Computing

Die geradezu fantastischen Zugriffszeiten haben dabei gar nichts mit MicroStream zu tun, sondern sind das Resultat reiner Java-Programmierung. Gemäß dem Ansatz von Separation-of-Concerns kümmert sich MicroStream um nichts anderes als um die Persistierung. Dieser Ansatz ist so alt wie die Java-Serialisierung. Brauchbar für die Datenbank-Entwicklung wird diese jedoch erstmals durch MicroStream. Die MicroStream-Entwickler bezeichnen diesen Ansatz als Pure-Java-In-Memory-Computing-Paradigma.

Einfache Implementierung

Aus Entwicklersicht ist dieser Ansatz besonders spannend. Alles was man braucht sind Java-Entity-Klassen und MicroStream für die Persistierung. Es gibt nur noch ein einziges Datenmodell. Das Objektmodell lässt sich nun völlig frei designen. Ein aufwändiges Type-Mapping wie bei der Verwendung externer Datenbanksysteme gibt es nicht mehr. Der Einsatz von ORM-Frameworks wie Hibernate und die damit verbundene Komplexität fällt ebenfalls ersatzlos weg.

Wer mit MicroStream Datenbankapplikationen entwickeln möchte, muss umdenken. MicroStream ist kein DBMS, sondern eine reine Storage-Engine für Objektgraphen. Um Concurrency muss sich die Businesslogik kümmern. Die Java-Serveranwendung ersetzt das externe DBMS.

MicroStream läuft auf dem Server, auf Java-Desktops, in Cloud- und Container-Umgebungen und ist prädestiniert für den Einsatz als Persistence-Strategie für Microservices. MicroStream ermöglicht die Entwicklung hochperformanter In-Memory-Datenbankanwendungen mit pure Java. Auch klassische Business-Applikationen lassen sich mit MicroStream stark beschleunigen und die Entwicklung spürbar vereinfachen. In Kürze soll MicroStream auch auf Android, und damit auf mobilen Geräten und Embedded-Systemen lauffähig sein.

Quellen:

1 https://bit.ly/2NmWlg1

DIE GROSSE JAVA COMMUNITY KONFERENZ

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

MicroStream Days

- **Java In-Memory Computing**
- **Native Java Data Persistence**
- **New Serialization for Java**

Power-Wokshop:

Mo. 23. September 2019

In diesem hochkarätigem Power-Workshop lernen Sie von A-Z, wie Sie mit MicroStream ultra-schnelle Java In-Memory Datenbank-Applikationen mit Pure Java entwickeln können. Im zweiten Teil des Workshops lernen Sie alles über die neue MicroStream Serialisierung für Java. Trainer: Markus Kett, CEO von MicroStream, Florian Habermann, CTO von MicroStream und Christian Kümmel, Projektleiter

Zeit	Programm
9:00	**Einführung: Funktionsweise von MicroStream, Architektur, Abgrenzung zu DBMS, Pure Java In-Memory Computing, Use-Cases, FAQ** Markus Kett, *MicroStream*
10:30	Kaffeepause
11:00	**Coding: Konfiguration, Model Design, Datenbank erzeugen, Create-Read-Update-Delete** Florian Habermann, *MicroStream*
12:30	Mittagspause
13:30	**Best-Practice: Model Design, Queries, Multithreading, Class-Changes, Backups, Containerisierung** Christian Kümmel, *XDEV Software Corp.*
15:00	Kaffeepause
15:30	**Kugelsichere Serialisierung und elegante Objekt-Graph-Kommunikation mit MicroStream** Markus Kett, Florian Habermann und Christian Kümmel
17:00	Ende

Sessions:

Di. 24. September 2019

Zeit	Programm
9:00	**Creating ultra-fast Java In-Memory Applications and Microservices with MicroStream** Markus Kett, *MicroStream*
10:00	JCON2019 Keynote: **Enterprise Java Innovation #slideless** Adam Bien
11:15	**Java In-Memory Database-Apps with MicroStream - Getting Started** Florian Habermann, *MicroStream*
12:00	Mittagspause & Expo-Time - 45 Minuten
13:00	**In-Memory Objekt-Graphen effizient nutzen** Christian Kümmel, *XDEV*
13:00	**A Modern Fairy Tale: Java Serialization** Steve Pool, *IBM*
15:00	**Bullet-proof Java Serialization and super-fast Object-Graph Communication** Florian Habermann, *MicroStream*
15:45	Pause & Expo-Time - 30 Minuten
16:15	**MicroStream Best-Practice - Projekt-Setup, Konfiguration, Backup** Christian Kümmel, *XDEV*
17:15	**60% less Memory Usage with OpenJ9 JVM** Steve Poole, *IBM*
18:15	**Challenges with MicroStream - Was ist, wenn sich meine Klassen ändern?** Christian Kümmel, *XDEV*

JCON2019
www.jcon.one

MicroStream
www.microstream.one

Jetzt anmelden unter: www.jcon.one

Powered by MicroStream - Gold-Sponsor der JCON 2019 - www.microstream.one

#JAVAPRO #TESTING #QUALITY #REPORTS #TDD

Nachhaltige Report-Entwicklung mit TDD

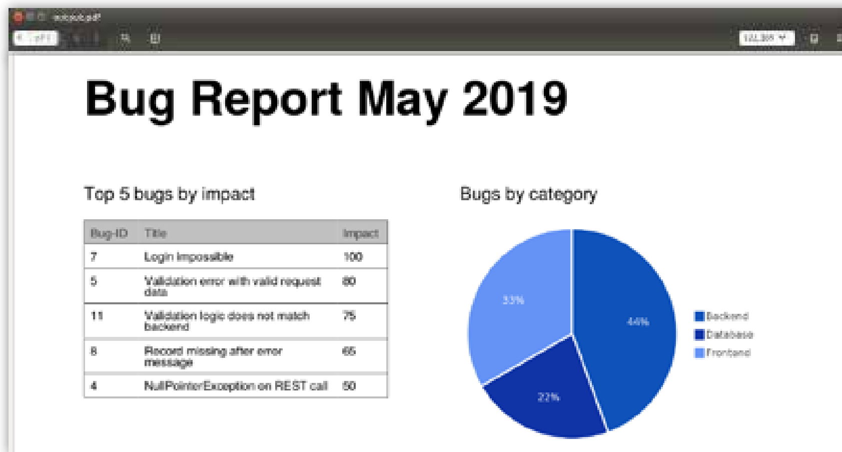
Grafische Auswertungen sind ein zentraler Wertschöpfungsfaktor in den verschiedenen Softwareprodukten, mit denen wir uns im Entwickleralltag beschäftigen. Trotzdem verzichten viele Projektteams an dieser Schlüsselstelle auf automatisierte Tests. Wie kann ein Vorgehen aussehen, das dem hohen fachlichen Wert gerecht wird und dennoch in einem vertretbaren Zeitrahmen umsetzbar ist?

Autor:

Julius Mischok ist Geschäftsführer der Mischok GmbH in Augsburg. Seine Kernaufgaben sind Prozessentwicklung, sowie Coaching und Schulung der Entwicklungsteams. Aktuell fokussiert sich seine Arbeit auf die Frage, wie Software schnell und mit einer maximalen Wertschöpfung produziert werden kann. Er hat Mathematik studiert und entwickelt seit fast zwei Jahrzehnten Java.

julius.mischok@mischok.de
www.mischok.de
https://www.xing.com/profile/Julius_Mischok

Stellen wir uns eine dieser zahlreichen Web-Applikationen vor: Ein schönes Backend mit einer Datenquelle und vielleicht noch einem Fremdsystem. Das Frontend ist eine SPA (Single-Page-Application) und es wurden in mehreren Sprints allerhand Features entwickelt. Im Wesentlichen verwaltet die Applikation Bug-Tickets, die in einem nicht sonderlich komplizierten Schema in einer relationalen Datenbank abgelegt sind. Moderne Frame-works und Bibliotheken ermöglichen eine Konzentration auf die Fachlichkeit, die testgetriebene Entwicklung läuft rund, das Projekt kommt gut voran. In der Story „Reporting" liegen bereits dezidierte Layoutvorstellungen des Kunden für eine grafische Auswertung der gespeicherten Daten. Hier soll sortiert und aggregiert werden, Topdatensätze in einer Liste angezeigt, und ein Diagramm ist auch noch gewünscht. Das Ganze soll als



Bug Report May 2019

Top 5 bugs by impact

Bug-ID	Title	Impact
7	Login impossible	100
5	Validation error with valid request data	80
11	Validation logic does not match backend	75
8	Record missing after error message	65
4	NullPointerException on REST call	50

Layoutvorgaben für den monatlichen Bugreport. (Abb. 1)

mehrsei-tiges PDF ausgegeben werden (Abb. 1). Um die Anforderungen sinnvoll testen zu können, ist natürlich ein vernünftiges Set an Testdaten notwendig.

Unter Zugzwang

Bis dieses Set vorliegt, wird die Story fröhlich weitergeschoben, bis sie in einem Sprint-Planungsmeeting vom Kunden priorisiert wird. Hintergrund ist eine Präsentation in seiner Fachabteilung, die sich von dem Feature die zentrale Wertschöpfung innerhalb des Projekts erhofft.

Blenden wir uns kurz aus dem Gedankenspiel aus: Ist das nicht ein klassisches Dilemma des Projektalltags? In der Entwicklung sind wir glücklich und zufrieden, wenn die Daten ordentlich abgelegt sind, in einem sauberen relationalen Schema, einem Graphen oder einer beliebig skalierbaren NoSQL-Datenbank. Wir wissen um den Wert, den die Daten dort haben. Leider fehlt den Anwendern unserer Produkte diese Sicht. Für sie setzt die Wertschöpfung erst in dem Moment ein, in dem es uns gelingt, die Rohdaten aufbereitet zu präsentieren – in einer Liste oder eben einem Report, der schon gewisse Operationen darauf ausführt.

Ungeklärte Fragen

Zurück zum Beispielprojekt: Durch die Anforderung sieht sich das Projektteam nun mit der immer weiter geschobenen Reporting-Story konfrontiert und fühlt sich plötzlich in die Enge getrieben. Denn an der überschaubaren Datenbasis in der lokalen Entwicklungsumgebung hat sich nicht viel geändert. Da natürlich zeitgemäße Architekturentscheidungen getroffen wurden, wird ein Datenbankmigrations-Tool genutzt. Die lokale Umgebung läuft gegen eine Datenbank im Container, welche regelmäßig zurückgesetzt wird. Die Unit-Tests werden selbstverständlich gegen eine In-Memory-Datenbank ausgeführt. Woher sollen denn jetzt plötzlich ausreichend viele Testdaten kommen?

Seien wir mal ehrlich: Niemand entwickelt diese Reports gerne. Allein der Gedanke daran, sich über die Web-Oberfläche Testdaten zusammen zu klicken, dann auf dem Papier oder mit Excel die Aggregationen durchzurechnen, Implementieren, Projekt neu bauen, anmelden, durchklicken bis zum Report, generieren, warten, Feststellen, dass man einen Rundungsfehler gemacht hat, und das Ganze wieder von vorne. Zwischenzeitlich setzt man aus guter Gewohnheit wieder den lokalen Datenbankcontainer zurück und fügt damit dem Drama ein weiteres Kapitel aus dem Bereich „Daten zusammenklicken" hinzu.

Natürlich gäbe es auch die Testumgebung. Die Piplines sind auch so eingestellt, dass bei jedem Push in den Feature-Branch gebaut wird, aber die Feedback-Zeiten (um dann am Schluss nur den Rundungsfehler festzustellen) sind dann doch abschreckend.

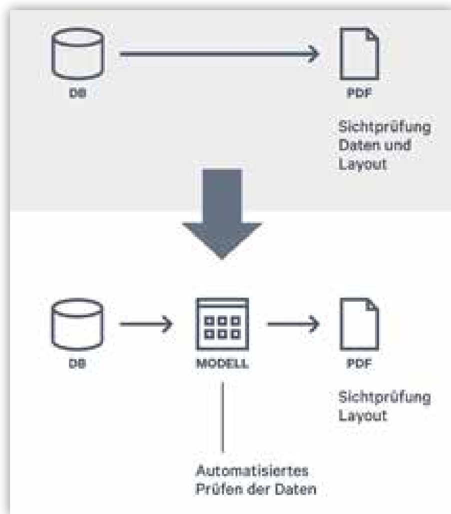
Die Variante, ein im Test generiertes PDF einzulesen und mit gewissen Erwartungen abzugleichen, scheidet auf Grund der überschaubar positiven Erfahrung in anderen Projekten von vornherein aus. Es scheint so, als müssten die bisherigen Best-Practices im Projekt über Bord geworfen und an dieser zentralen Stelle auf die testgetriebene Entwicklung verzichtet werden.

Frische Ideen

Abgesehen von den ungeklärten Fragen ist natürlich das Ziel, den Report nicht in einem Eine-Klasse-eine-Methode-Monster zu bewältigen, sondern eine sinnvolle Aufteilung zu wählen, die den Best-Practices in Sachen Clean-Code gerecht wird. Erste Idee ist hier, die Klassen mit einem spezifischen Zweck auszustatten. Im vorliegenden Fall macht die Trennung in Datenabfrage/Aufbereitung und Layout Sinn. Plötzlich keimt Hoffnung auf: Ist bei der Wahl eines entsprechenden Designs eine testgetriebene Entwicklung doch möglich? Dies würde in den Projektkontext passen und wäre vor allem nachhaltig. Klar, auch hier müsste man den Aufwand der Testdatengenerierung einmal gehen, aber für automatisierte Tests wäre die Motivation deutlich größer.

Bei der Sprint-Planung folgt die obligatorische Diskussion und natürlich nagt der Zweifel: Die Daten liegen doch tabellarisch vor. Mit einer einfachen Query wäre die Aggregation getan und dann müsste man nur noch die Top 5 ins PDF pressen. Warum also noch mehr Struktur einführen? Ist hier in Anbetracht des Zeitdrucks nicht das klassische Vorgehen mit Sichttests ausreichend?

Andererseits würde dieses einfache Vorgehen das bisher im Projekt praktizierte testgetriebene Vorgehen massiv untergraben. Regressionsfehler wären erfahrungsgemäß unvermeidbar. Die Frage lautet also, wie viel automatisiertes Testen ist fachlich sinnvoll und wie viel ist wirtschaftlich möglich?



Workflow (Abb. 2)

Von der Idee zum Design

Bei genauerer Betrachtung nimmt die anfangs noch vage Idee weiter Gestalt an. Wenn die gesamte Verantwortung für die Transformation aus der Datenbank gelesener JPA-Entities in ein als Byte-Array vorliegendes PDF-Dokument auf zwei Komponenten aufgeteilt wird, benötigen diese ein definiertes Datenformat für den Austausch. Dieses würde die Struktur der im Report dargestellten Daten repräsentieren, wäre also ein abstraktes Modell des PDFs. Konsequent weiter gedacht eröffnen sich die Möglichkeiten, die eine Vielzahl der zuvor aufgeführten Probleme lösen: Zumindest die Erstellung des Modells ist optimal testgetrieben entwickelbar. Die Geschäftslogik ist schön in den jeweiligen Komponenten gekapselt, die das Modell erstellen und die das Modell zur Ausgabedatei transformieren. Das Modell kann aus POJOs zusammengesetzt werden und enthält selbst keinerlei Logik (Abb. 2).

Die Modellierung sinnvoller Testfälle zwingt zu einem frühen Zeitpunkt zu einer Auseinandersetzung mit den fachlichen Kundenanforderungen. Für den Geschäftswert wichtige Grenzfälle können einfach und automatisiert getestet werden.

Am Anfang die Modellierung

In der Praxis wird im ersten Schritt die Kundenanforderung anhand der Vorgaben in ein abstraktes Modell gefasst. Das Beispielprojekt[1] zeigt die Umsetzung des, zugegebenermaßen minimalistischen, Bugreports aus (Abb. 1) für einen bestimmten Monat. Die Seite enthält die fünf Bugs mit der größten Auswirkung tabellarisch angezeigt sowie ein Tortendiagramm mit einer Übersicht über die Kategorien der aufgetretenen Bugs.

(Listing 1)

```
public class BugreportModel {

    private String title;
    private TableModel<TableColumns> tableModel;
    private PieChartModel pieChartModel;

    […]
}
```

Wie in (Listing 1) ersichtlich, wurde die Entscheidung getroffen, die Überschriften über Tabelle und Diagramm mit in die jeweiligen Modelle einfließen zu lassen. (Listing 2) zeigt das Modell für das gesamte Tortendiagramm, im Wesentlichen ist hier die Überschrift und eine Liste der Daten für die einzelnen Sektoren enthalten.

(Listing 2)

```
public class PieChartModel {

    private final String headline;

    private final List<PieChartSliceModel> slices;

    […]
}
```

(Listing 3)

```
public class PieChartSliceModel {

    private final String section;
    private final long value;
    private final Color color;

    […]
}
```

Aus (Listing 3) wird ersichtlich, dass teilweise Layoutinformationen in das Modell aufgenommen werden: Da die Farbe der Sektoren eine fachliche Anforderung ist, erscheint sie hier im Modell. Mit diesen wenigen Code-Zeilen ist die gesamte rechte Spalte des Reports modelliert.

Die Modellierung von Tabellen ist durch die Zweidimensionalität komplizierter: Es hat sich jedoch das im Beispielprojekt gezeigte Vorgehen bewährt, in einer Map eindeutige Schlüssel für die Spalten und deren Layout-Information (Überschrift, Breite, Ausrichtung, etc.) zu hinterlegen. Die Werte für die Zeilen der Tabelle stehen in einer Liste, die Map-Elemente aus Spaltenschlüssel und dem Zellwert enthält. Hierdurch ist das Umsortieren von Spalten einfach und ohne Indexschlacht möglich, insbesondere ist die Aufbereitung des Modells über die Spaltenschlüssel transparent.

Endlich wieder TDD

Steht die Struktur des Modells, können die Tests geschrieben werden. Hierzu muss zunächst ein minimales, aber sinnvolles

Set an Testdaten erstellt werden. Dieses kann beispielsweise per JPA-Repositories oder mit einem SQL-Script vor jedem Testlauf in der (In-Memory-) Datenbank angelegt werden. Wie bei allen anderen Unit-Tests ist es wichtig, das richtige Maß zu finden: Kurze, aussagekräftige Tests definieren, die bei einem Fehlschlag möglichst schon im Namen sagen, welche Erwartungen verletzt wurden. Ein besonderes Augenmerk ist auf spezielle, fachliche Anforderungen zu setzen: Im Beispiel des Bug-Reports sollte sichergestellt werden, dass tatsächlich kein Bug-Eintrag aus dem Vormonat in der Tabelle landet, obwohl er einen hohen Impact-Wert hat und, dass im Tortendiagramm nur Kategorien angezeigt werden, in denen im ausgewerteten Monat wirklich Bugs erfasst wurden.

(Listing 4)

```
@Service
public class ModelService {

    public BugreportModel generateModel(@NonNull Integer year, @
NonNull Integer month) {

        LocalDate firstOfReportMonth = LocalDate.of(year, month,
1);

        String title = generateTitle(firstOfReportMonth);
        TableModel<TableColumns> tableModel = generateTableMo-
del(firstOfReportMonth);
        PieChartModel pieChartModel = generatePieChartModel(-
firstOfReportMonth);

        return BugreportModel.builder()
                .title(title)
                .tableModel(tableModel)
                .pieChartModel(pieChartModel)
                .build();
    }

    […]
}
```

Mit den Unit-Tests ist die Implementierung der Hilfsmethoden aus **(Listing 4)** reine Schreibarbeit. Je nach Umfang der Datengrundlage müssen auch Performanceüberlegungen angestellt werden: Wie in den meisten anderen Fällen macht es Sinn, Filterung und Sortierung bereits bei der Datenbankabfrage erledigen zu lassen. Durch die hohe Testüberdeckung ist aber ein späteres Refactoring relativ unproblematisch.

Mit Fokus

Sobald die Modellstruktur fixiert ist, kann parallel bereits mit der grafischen Umsetzung in Richtung PDF begonnen werden. Hierzu ist schnell ein Beispielmodell geschrieben, das als Datengrundlage für die an dieser Stelle unvermeidlichen Sichttests dient. In der täglichen Arbeit zeigt sich hier neben der Parallelisierbarkeit der zentrale Vorteil des beschriebenen Designansatzes: die beiden Hauptservices (ModelService und PdfService) lösen jeweils nur eine Fragestellung, diese aber sehr gut. Diese Fokussierung hilft bei der Entwicklung ungemein und führt dazu, dass Fehler schnell lokalisiert werden können. Wenn die Sicherheit besteht, dass das Tabellenmodell korrekt ist, aber im PDF nur vier anstatt der erwarteten fünf Zeilen angezeigt werden, muss der Fehler in der Umsetzung zur PDF-Datei liegen.

Nachhaltig erfolgreich

In der täglichen Arbeit stellt sich schnell das Gefühl ein, Struktur in die sonst so unübersichtliche Thematik gebracht zu haben. Die Klassen sind kürzer, die Tickets kleiner und das Entwicklungstempo ist konstant hoch. Die Implementierung weiterer spezialisierter Services, zum Beispiel zur Umsetzung des Diagramms und der Tabelle fängt an, Freude zu machen. Und auch langfristig erweist sich der Ansatz als gute Wahl: Fehler im Testbetrieb können oft lokal als Regressionstests nachgestellt werden und sind schnell und nachhaltig behoben. Die Endanwender können zeitnah mit einem verlässlichen Stand arbeiten, es werden kaum Bugfix-Deployments gebraucht. Wenn nach einem Jahr der gleiche Export nicht nur nach PDF, sondern auch in Word gewünscht wird, ist das kein Problem. Schnell ist eine weitere Komponente programmiert, die die gleiche Modellstruktur in Word überführt. Die Modellgenerierung wird dabei nicht mehr angefasst, daher sind die angezeigten Daten vom ersten Tag an korrekt. Durch die Entkopplung ist es ebenso leicht möglich, die Bibliothek zur PDF-Generierung gegen eine andere auszutauschen. Beim Austausch genügt die Konzentration auf das „Wie werden die aufbereiteten Daten angezeigt?".

Fazit:

Zwei Erkenntnisse bleiben: Zum einen, dass es möglich ist, durch ein gelungenes Design die testgetriebene Entwicklung zu ermöglichen. Dieser testfokussierte Ansatz ist ein Paradigmenwechsel, hat sich aber bewährt, nicht nur im hier geschilderten Anwendungsfall. Der Wert der Software und damit auch der Erfolg des Designs bemisst sich klar aus dem Maß der automatisierten Testbarkeit. Zum anderen gilt der Leitsatz: „Man muss nicht alles automatisiert testen – nur die richtigen Dinge". Nur bis zum Modell automatisiert zu testen - hier jedoch auf eine hohe Überdeckung wert zu legen - hat sich klar bewährt. Es ergibt sich ein immer optimaleres Verhältnis aus dem Anspruch an Qualität gegenüber dem Budgetdruck. Auch in kleinen Projekten kann man auf diesem Weg mit ausreichender Sicherheit durch automatisierte Tests aufwarten. Die Suche nach einem sauberen Design hat ganz nebenbei eine Idee eingeführt, die für viele Anforderungen eine Hilfestellung ist. Neben grafischen Reports gibt es noch viele weitere sinnvolle Einsatzfelder: Der Versand von HTML-Emails gehört genauso dazu, wie listenbasierte Exporte mit komplizierter Geschäftslogik.

Quellen:

https://gitlab.mischok-it.de/open/reports

#JAVAPRO #Cloud #Container #Tools

Container-Monitoring mit Prometheus

Entwickler benötigen Informationen über technische Fehlfunktionen in verteilten Systemen und Anwendungen, um schnell reagieren und Ausfälle verhindern zu können. Observability steht für einen umfassenden Ansatz, der zahlreiche Faktoren für die Überwachung und Beobachtung des Verhaltens von Software einbezieht. Ein zentrales Instrument dafür ist das Tool Prometheus.

Die Monitoring-Infrastruktur für Applikationen wird immer wichtiger, auch weil die Zahl der mit dem Internet vernetzten Nutzer und Geräte jedes Jahr ansteigt. Wie verhält sich das System unter Last? Wie lange dauert eine Transaktion? Wie sieht es mit der Reaktionszeit aus? Unternehmen sollten verstehen, inwieweit ein System verfügbar ist, wie es funktioniert und wie die Nutzer eine Applikation wahrnehmen. Neben den üblichen Metriken gewinnen daher auch andere Kriterien an Bedeutung, um die Funktionsweise und das Verhalten von Systemarchitekturen oder Software besser zu verstehen. Dieser Trend wird auch als Observability bezeichnet und umfasst zumindest drei Säulen: Metrikdaten, Logging und Tracing. Zu Observability gehört jedoch auch jede Erkenntnis, mit der Unternehmen Applikationen besser verstehen, wie sie sich verhalten und wie sie funktionieren. Es geht darum, systematisch zu erkennen, was die Softwarelösungen leisten und wie sie funktionieren, um Ausfälle zu verhindern beziehungsweise die Software nach einem Ausfall schnell wiederherzustellen (Recovery). Ein strategischer Ansatz ist unerlässlich, um Metriken, Protokolle und Profile systematisch bewerten sowie kombinieren zu können und damit ein vollständiges Bild der Systeme zu erhalten. Doch eignen sich die derzeit eingesetzten Tools dafür, um die Verfügbarkeit zu maximieren und die durchschnittliche Zeit bis zur Problemlösung zu minimieren? Unabhängig davon, wie viel Zeit und Geld Unternehmen in die Verfügbarkeit eines Systems investieren, es wird immer Zwischenfälle oder Störungen geben. Daher ist es wichtig, sich auf solche Ereignisse vorzubereiten, diese zu untersuchen und zu bewerten. Ist Monitoring im Zeitalter der Observability noch weiter relevant? Diese Frage beantwortet sich nach einem Blick auf die Entwicklung von Monitoring und die zugehörigen Tools im Laufe der letzten Jahre, insbesondere aber seit der Verfügbarkeit des Monitoring-Tools Prometheus.

Autor:

Frederic Branczyk
Principal Software Engineer, Red Hat

Frederic ist Software-Ingenieur bei Red Hat (Eintritt im Zuge der CoreOS-Akquisition), Teil des Prometheus-Kernteams und Leiter der Kubernetes Special Interest Group.

Er engagiert sich für eine deutliche Weiterentwicklung von Observability Tools. Sein Ziel dabei ist die Entwicklung einer modernen Infrastruktur und von SRE-Tools, die uns beim Verständnis der operativen Aspekte von Applikationen unterstützen.

Prometheus erlaubt den Blick ins Innere

Ein Trend ist eindeutig zu erkennen: Monitoring entwickelt sich hin zum Whitebox-Monitoring. Im Gegensatz zur Blackbox-Überwachung beobachten die Tools die internen Abläufe eines Prozesses genauer, anstatt nur von außen zu prüfen, ob die Anwendung oder der Prozess wie erwartet reagiert. Dies ist der kleine, aber wichtige Unterschied: Das Whitebox-Monitoring überwacht das Verhalten und nicht die Reaktionen. Daher ist jetzt auch proaktives Monitoring möglich, sprich Fehler, technische Mängel oder andere Ereignisse lassen sich vorhersehen und verhindern, bevor sie eintreten. Die ursprünglichen Entwickler von Prometheus ließen sich von der Monitoring-Lösung Borgmon inspirieren, die Google erstellte, um das interne Orchestrierungssystem zu überwachen. Ein Tool wie Borgmon fehlte in der Welt außerhalb von Google und daher entschlossen sich die Programmierer, die damals bei SoundCloud arbeiteten, eine solche Lösung zu entwickeln.

Der Erfolg von Prometheus beruht vor allem auf seiner Zuverlässigkeit. Diese Eigenschaft ist für ein Monitoring-Tool zentral, da die Lösung für die Überwachung von Systemen der robusteste Teil der Infrastruktur sein muss – alles andere hängt davon ab, dass sie funktioniert. Prometheus arbeitet deswegen so zuverlässig, weil das Tool im Pull-Modell arbeitet und Daten abfragt. Das bedeutet nicht, dass Pull der einzige brauchbare Modus ist, aber damit ist es einfacher, zuverlässig zu arbeiten.

Prometheus lässt sich als einzelne, statisch verknüpfte Binärdatei einrichten, die in jeder Art von Umgebung sehr einfach gestartet und aktualisiert werden kann – unabhängig davon, ob Container verwendet werden oder nicht. Diese Einfachheit stellt in Verbindung mit der zuverlässigen Funktionalität einen wichtigen Faktor für den Erfolg von Prometheus dar.

Multidimensionales Datenmodell

Prometheus setzt auf ein multidimensionales Datenmodell zum Identifizieren von Zeitreihen, sprich zeitlichen Abfolgen von Daten. Als Prometheus entwickelt wurde, gab es kein integriertes Überwachungssystem, das die Abfrage von Zeitreihen anhand einer Teilmenge ihrer Kennzahlen ermöglichte. OpenTSDB erlaubte zwar ähnliche Abfragen, verursachte aber hohe Betriebskosten, die Prometheus vermeiden wollte. Prometheus speicherte in der ersten Version seinen gesamten Inhalt in der integrierten Datenbank LevelDB. Sie diente der Indexierung der Zeitreihen und in der zweiten Version von Prometheus wurde jede Zeitreihe in eine separate Datei geschrieben. Dies funktionierte lange Zeit sehr gut, da die Entwickler ursprünglich dynamische Umgebungen und weniger statische virtuelle Maschinen erwarteten und Prometheus entsprechend aufbauten. Doch das Ausmaß und die Häufigkeit der Änderungen der heutigen großen Kubernetes-Cluster und der Möglichkeit multipler Cluster übertrafen alle Annahmen. Die größte Herausforderung bildete hier die Kardinalität der Metriken und ihre Veränderung (Churn). Die Kardinalität steht für die Gesamtzahl der von Prometheus aufgenommenen Zeitreihen. Sie beschreibt die Anzahl der Zeitreihen mit gleicher Metrik, allerdings mit variablen Werten für einzelne Labels. Churn beschreibt die Lebensdauer der Zeitreihen. Der schlimmste Fall für Prometheus ist eine hohe Churn-Rate, bei der die Zeitreihen häufig gestartet und beendet werden. In der zweiten Storage-Version von Prometheus wurde dafür aufwändig jedes Mal eine neue Datei erstellt. Das Problem: Millionen von Zeitreihen führen zu Millionen von Dateien, für die viele Dateisysteme eigens abgestimmt werden müssen oder möglicherweise gar nicht funktionieren.

Das Prometheus-Team entwickelte daher eine dritte Storage-Version, um dieses Problem zu lösen. Anstatt eine Datei pro Zeitreihe zu speichern, besteht die Storage jetzt aus zwei Teilen mit voll funktionsfähigen Datenbanken, die eine eigene Kopie des Index speichern und sich nicht verändern lassen. Wie bei vielen Datenbanken kann der Kernel jetzt die Storage effizient von der Festplatte mappen. Die neue Storage-Architektur löst das Skalierbarkeitsproblem und reduziert den Ressourcenverbrauch in den meisten Szenarien erheblich. Die neue Storage-Version bildete den Hauptgrund für die im November 2017 veröffentlichte Version Prometheus 2.0.

Funktionen stabilisieren

Seit diesem Release besteht das Ziel des Prometheus-Projekts darin, die vorhandenen Funktionen zu stabilisieren. Das Team führte einen sechswöchigen Release-Zyklus, eine detailliertere Release-Dokumentation sowie eine Lead-Rolle ein und lässt regelmäßig externe Sicherheitstests durchführen. Zudem gibt es eine Reihe von automatisierten Leistungstests, um Probleme bereits während der Entwicklungsphase zu identifizieren und die Leistung von Prometheus vor der Veröffentlichung zu überprüfen. Die Arbeit lohnte sich, denn die Cloud-Native-Computing-Foundation (CNCF) verlieh Prometheus im August 2018 den Graduate-Status. Damit gilt Prometheus als zukunftsfähiges Projekt mit stabiler Leistung und Sicherheit, das nicht mehrheitlich einem einzigen Unternehmen gehört. Das Siegel der CNCF ist ein wichtiger Meilenstein für das Projekt und die gesamte Community.

Fazit:

Monitoring wird immer wichtiger, bildet aber nur den Einstieg in eine erfolgreiche Observability. Monitoring bleibt neben anderen Kriterien ein starker Faktor für ein besseres Verständnis von verteilten Systemen und Anwendungen. Künftig wird der Fokus auf der Korrelation dieser verschiedenen und in ihrer Anzahl steigenden Beobachtungskriterien liegen. Die über Metrikdaten gesteuerte Alarmierung (Alerting) wird weiterhin den Ausgangspunkt darstellen, um Fehler oder Ausfälle schnellstmöglich zu beheben.

#JAVAPRO #DEVOPS #DASA

Was DevOps heute wissen müssen

Der 1. Teil unserer zweiteiligen Serie hat die 6 DASA-DevOps-Prinzipien dargestellt. Der 2. Teil geht nun auf das DASA-DevOps-Kompetenz-Framework ein, welches 12 Bereiche definiert, in denen die Mitarbeiter Qualifikationen haben müssen. Die Bereiche reichen von Soft-Skills wie Mut und Leadership über eher technische Aspekte wie Infrastruktur-Engineering und Continuous-Delivery zu eher klassischen Themen wie Business-Analyse und Testspezifikation.

DASA ist eine unabhängige und offene, von Mitgliedern getragene Vereinigung, die weltweit die Entwicklung von DevOps-Schulungen und -Zertifizierungen unterstützt. Mit den 6 DASA-DevOps-Prinzipien, die im ersten Teil unserer Serie dargestellt wurden, wird eine Basis für die organisationsweite Betrachtung gelegt. Es wird die Grundlage für die Ausrichtung und Entwicklung einer IT-Organisation in Richtung DevOps als Inbegriff einer hoch performanten IT gelegt. Der zweite Teil geht nun auf das DASA-DevOps Kompetenz-Framework ein, welches 12 Bereiche definiert, in denen die Mitarbeiter Qualifikationen haben müssen. Diese 12 Bereiche unterteilen sich in 4 Verhaltensbereiche und 8 Wissensbereiche. Analog zu der Tatsache, dass DevOps in einer IT-Organisation nicht allein durch Automation zu erreichen ist, muss ein moderner DevOps-Engineer neben Tool-Kompetenzen weitergehende Skills besitzen.

Autor:

Das Motto von Dierk Söllner (www.dsoellner.de) 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.

Das DASA-DevOps-Kompetenz-Framework sieht in den einzelnen Bereichen jeweils verschiedene Stufen vor. Damit kann ein persönlicher Entwicklungspfad für die Mitglieder in einem DevOps-Team aufgezeigt werden.

Das Ziel der Eingangsschulung DevOps-Fundamentals ist es, alle Teilnehmer auf den gleichen Wissensstand (Stufe 2) zu heben. Die Bereiche reichen von Soft-Skills wie Mut und Leadership über eher technische Aspekte wie Infrastruktur-Engineering und Continuous-Delivery zu eher klassischen Themen wie Business-Analyse und Testspezifikation.

DASA-DevOps-Kompetenz-Framework[1]. (Abb. 1)

Mut

Mut beschreibt im Zusammenhang von DevOps einerseits die Fähigkeit, im beruflichen Umfeld zu agieren, Dinge auszuprobieren und einfach voranzugehen sowie andererseits die Risiken zu kennen, sie in die Betrachtung einzubeziehen und zu minimieren. Innovative Unternehmen und Mitarbeiter sind auf dem Gebiet schon aktiv und haben ihre Erfahrungen gesammelt bzw. sammeln sie noch. Für die Early-Adaptors ist das jedoch an vielen Stellen noch unerforschtes Gebiet, so dass es mutige Mitarbeiter und Führungskräfte braucht, den Schritt in Richtung DevOps und damit einer radikalen Veränderung zu gehen. Konkret zeigt sich der Mut darin, die innere Überzeugung aktiv und überzeugend nach außen zu vertreten (Evangelisten). Es gilt, zu den vielen Themen im Rahmen von DevOps offene Diskussionen zu führen, den Mut zur Veränderung in das Unternehmen und die Köpfe der IT-Professionals zu bringen, proaktiv Inhalte von DevOps aufzunehmen, u.a. durch eine moderne Experimentierkultur, sowie viel Gruppen- und Individualreflexion und Coaching.

Team-Building

Die Teamentwicklung ist ein kritischer Erfolgsfaktor für den Erfolg von DevOps. In den Teams arbeiten die Experten mit den unterschiedlichen Hintergründen, Zielsetzungen und Ausbildungen. Diese müssen bei der Gestaltung ihrer gemeinsamen Arbeit unterstützt werden, um eine effektive und effiziente Zusammenarbeit gestalten zu können.

In vielen deutschen IT-Organisationen arbeiten die Experten seit langer Zeit ohne eine wirkliche Notwendigkeit, Teams zu bilden und mit diesen Teams dauerhaft Erfolge zu gestalten. (Die Arbeit in Projektteams ist sicher eine Ausnahme, wobei aus Sicht von DevOps Projektteams differenziert betrachtet werden. DevOps hat das Ziel, nachhaltig wirkende Teams zu formen, die neben der Entwicklung in Projektform auch langfristig für Kunden und die eigenen Projekte zusammenarbeiten).

Experten und Spezialisten waren das Ziel und diese werden häufig ungeachtet einer menschlichen Kompatibilität zur Zusammenarbeit gebracht. Das Ziel bei DevOps ist der Aufbau einer hochperformanten IT-Organisation. Dazu werden Teams benötigt, die die Vielfältigkeit und Unterschiedlichkeit der beteiligten Experten zu einer gemeinsamen Art der Zusammenarbeit vereint. Konkret heißt das bspw. ein gemeinsames Ziel zu schaffen, gegenseitige Rechenschaftspflichten zu definieren und Verständnis für unterschiedliche Standpunkte zu entwickeln.

DevOps-Leadership

In einer DevOps-orientierten Organisation bedeutet Führung nicht die Festlegung von Regeln und detaillierten Anweisungen, sondern die Vermittlung einer Vision. Damit werden Ziele und Nutzen einer Aufgabe beschrieben. Führung zu übernehmen ist dabei nicht nur Aufgabe von Führungskräften oder des Managements, sondern auch bzw. vielmehr von Mitgliedern der DevOps-Teams. Führung mit formeller und informeller Macht muss in einem passenden Gleichgewicht erfolgen. Dabei ist zu beachten, dass die Zusammenarbeit im Team ausgeglichen ist und das Team sich weiterentwickelt. Das übergeordnete Ziel, Erfüllung der Kundenanforderungen und Unterstützung der Geschäftsprozesse, darf nicht aus den Augen verloren werden. Deshalb wird das Thema Führung mit einer weiteren Kompetenz ergänzt: Feedback. Feedback sorgt für Transparenz, Fokussierung der Teams auf hochperformantes Agieren, Stakeholder-Management und Bewusstsein des Service-Lebenszyklus.

Kontinuierliche Verbesserung

Die heutige Geschäftswelt ist geprägt von kurzfristigen Veränderungszyklen. Damit wächst auch der Druck auf die IT, sich kontinuierlich anzupassen bzw. zu verbessern. Für DevOps-Mitarbeiter bedeutet dies, eine permanente Sensibilität für Verbesserungen

zu entwickeln. Einerseits in der Fähigkeit, Probleme zu sehen, andererseits diese auch anzugehen und zu beseitigen. Insbesondere für den zweiten Anspruch wird spezielles Wissen zur strukturierten Problemlösung benötigt. DevOps greift dabei auf bekannte und im Markt etablierte Methoden zurück, wie z.B. Kaizen-Mindset (Root-Cause-Analysis), den DMAIC-Zyklus (Lean-Six-Sigma) etc.

Business-Value-Optimization

Geschäftswertoptimierung bedeutet die Verbesserung der Geschäftsprozesse durch Einsatz von IT unter Berücksichtigung betriebswirtschaftlicher Anforderungen. Für eine IT-Organisation setzt dies sehr gute Kenntnisse über das Geschäftsmodell, dessen Ziele und Kennzahlen voraus, um so innovative und preiswerte IT-Lösungen zur bestmöglichen Unterstützung anzubieten. Die Steuerung dieser Lösungen erfolgt über vereinbarte Abnahmekriterien, abgestimmte Business-Cases, regelmäßige Feedback-Loops, sowie die Anwendung von Service-Level-Management-Praktiken.

Business-Analysis

Business-Analysis bedeutet, dass das DevOps-Team ein klares Verständnis über die Geschäftsprozesse und die dazu gehörenden IT-Systeme erlangt. Nur so können Verbesserungsmaßnahmen vollständig im Hinblick auf Nutzen und Risiken bewertet werden. Das Team muss in der Lage sein, eine detaillierte Geschäftsanalyse durchzuführen, so dass die zu entwickelnde IT-Lösung hinsichtlich Funktionalität, Kosten und Vorlaufzeit optimal gestaltet werden kann. Dazu gehört die Bewertung von funktionalen und qualitativen Anforderungen, die Beobachtung der langfristigen Entwicklung der Geschäftsprozesse, Veränderungen am Markt, eine detaillierte Datenanalyse und die Flexibilität regelmäßige Anpassungen zu berücksichtigen.

Architektur und Design

Architektur und Design sorgen für die Übersetzung der Anforderungen aus geschäftlicher Sicht in eine optimale technische IT-Lösung. Kenntnisse sowie Erfahrungen sind von entscheidender Bedeutung für alle DevOps-Teams. Dabei gilt es, das IT-System ganzheitlich im Kontext mit der kompletten IT-Umgebung zu betrachten, um so den eigenen Verantwortungsbereich (Technologie-Stack) mit seinen Schnittstellen zu anderen IT-Systemen zu bestimmen und zu gestalten. Letztlich geht es um die Sicherstellung, dass neue Entwicklungen zu den aktuellen Systemen passen und dass die Vorgaben zum Service-Design in vollen Maße Anwendung finden.

Testspezifikation

Bei der Testspezifikation geht es darum, sicherzustellen, dass die Anforderungen der Benutzer des IT-Service vollständig erfüllt werden. Dazu zählen die Konzeption der Testphasen und das Design der einzelnen Testfälle. Methoden wie Test-Driven-Development (TDD) sollen dazu beitragen, vor der eigentlichen Softwareentwicklung die Test-Sequenzen und Inhalte zu definieren. Solche Methoden werden zunehmend angewendet, um sicherzustellen, dass die geforderte Funktionalität auch geliefert wird. Ein weiteres Ziel ist die Automatisierung dieser Aktivitäten, um schneller, preiswerter und wirkungsvoller testen zu können.

Programmierung

Programmieren ist die Kernkompetenz des DevOps-Teams, unabhängig davon ob es ein DevOps-Team ist, das eine Anwendung, eine Plattform oder ein vollständiges IT-System unterstützt. Künftig werden alle IT-Techniker lernen müssen, Softwarekomponenten zu pflegen, zu erstellen und zu modifizieren. Dies bedeutet eine wesentliche Änderung für den IT-Betrieb, insbesondere den Infrastrukturbetrieb, bei dem es bisher kaum notwendig war, im Tagesgeschäft Code-Änderungen durchzuführen. Viele Unternehmen, die mit der DevOps-Transformation beginnen, kämpfen mit dem Mangel an Software-Engineering-Fähigkeiten in ihren neu gegründeten Teams. Firmen, denen es nicht gelingt, diese Fähigkeiten auch im Betrieb aufzubauen, werden in einer zunehmend digitalen Zeit kaum überleben können.

Continuous-Delivery

Eine wichtige Aufgabe für DevOps-Teams ist Continuous-Delivery. Dies beschreibt eine automatisierte Vorgehensweise zur kontinuierlichen Softwareauslieferung. Dazu zählt eine gut konstruierte Pipeline, die den Weg von der Entwicklung bis zur Produktion erleichtert, aber was vielleicht noch wichtiger ist: das konzeptuelle Verständnis des gesamten Prozesses und wie dies technisch unterstützt wird. Continuous-Delivery erfordert eine genaue Kenntnis des Softwarelieferprozesses von der Entwicklung bis zur Produktion. Wichtige Teilaufgaben in diesem Zusammenhang sind z.B. automatisiertes Testen, Release- und Deployment-Management, Konfigurationsmanagement, Versionskontrolle, Cloud-Dienste, Container-Lösungen, Feature-Driven-Delivery etc.

Infrastructure-Engineering

Infrastrukturaufbau ist eine weitere Disziplin, die ein DevOps-Teams übernehmen muss. Dies umfasst standardisierte IT-Umgebungen, so dass diese automatisiert, konsistent und schnell gepflegt werden können. Besonders die DevOps-Teams, die Infrastrukturdienste bereitstellen, müssen diese Fähigkeiten haben und anwenden. Für anwendungsorientierte DevOps-Teams ist es wichtig, die zugrundeliegende Infrastrukturtechnologie gut zu verstehen, um sicherzustellen, dass ihre Anwendungen durch die Standard-Infrastrukturmodelle optimal unterstützt werden. Infrastructure-Engineering wir durch eine Vielzahl von

technischen Aufgaben ermöglicht, wie z.B. technische Überwachung, Performance-Management (z.B. Load-Balancing etc.), das Management von Kapazitäten, Verfügbarkeit, Zuverlässigkeit, die Nutzung von Cloud- und Container-Services.

Security, Risk and Compliance

Die Themen Sicherheit, Risiken und Compliance treiben IT-Vorhaben von Beginn an. Besondere Aufmerksamkeit gilt der Verwaltung sicheren Software-Codes, dem Verständnis der Risiken bei der Infrastruktur und der Entwicklung neuer oder geänderter

DASA-DevOps-Ausbildungsprogramm[2]. (Abb. 2)

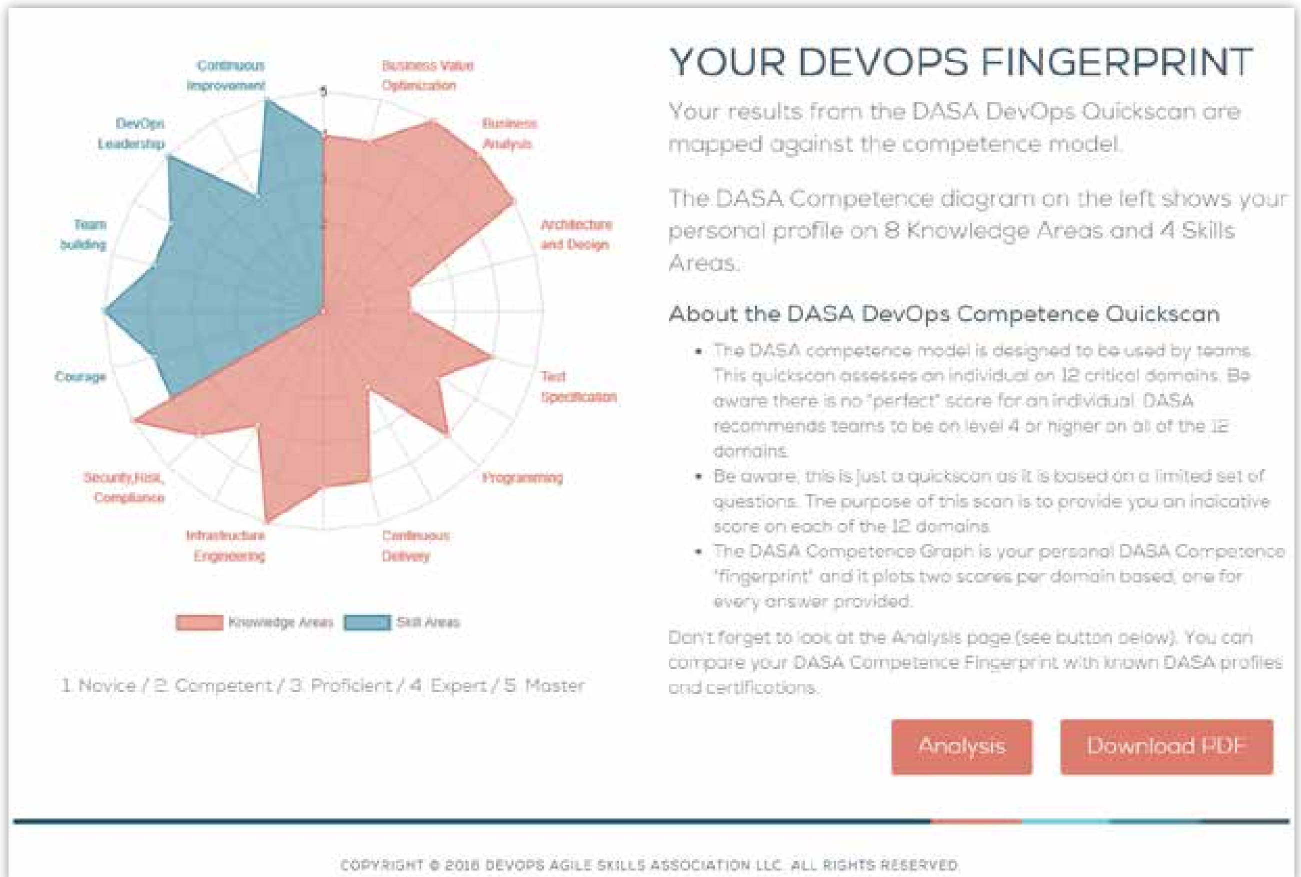
Funktionen unter Berücksichtigung regulativer und gesetzeskonformer Anforderungen zur bestmöglichen Unterstützung der Geschäftsprozesse. Sicherheits- und Risiko-Management sind auch bei der Planung der Servicekontinuität ausreichend zu untersuchen.

Fazit:

Nachdem es im ersten Teil unserer zweiteiligen Serie um die 6 DASA-DevOps-Prinzipien ging, befasst sich der zweite Teil mit dem DASA-DevOps-Kompetenz-Framework, das 12 Bereiche beschreibt, in denen DevOps Qualifikationen haben müssen. DevOps müssen in der Lage sein, die wesentlichen Konzepte und Prinzipien von DevOps zu beschreiben, die betriebswirtschaftlichen Vorteile von DevOps und Continuous-Delivery zu erklären und die Konzepte der Test-, Infrastruktur-, Build- sowie Deployment-Automatisierung verstanden haben. Zudem müssen DevOps die Beziehung von DevOps zum Lean-Management und agilen Methoden beschreiben sowie die kritischen Erfolgsfaktoren in Bezug auf die Einführung von DevOps bewerten können.

Quellen:

1 https://www.devopsagileskills.org/dasa-competence-model
2 https://www.devopsagileskills.org/certifications



Beispielhaftes Ergebnis des DASA-DevOps-Quickscan. (Abb. 3)

consol
Wir unternehmen IT

----
[*] Arbeiten unter Freunden
[*] Offener Wissensaustausch
[*] Flache Hierarchien
[*] Inhabergeführtes Unternehmen
[*] Voller Überstundenausgleich
ab der ersten Minute
[*] Viele Gestaltungs- und
Entwicklungsmöglichkeiten
[*] Top Kunden

-|------------------------------------|-
[$] Wir freuen uns, Dich kennen zu lernen!
Denn wir suchen laufend gute Leute an
unseren Standorten in München,
Düsseldorf und Nürnberg.
-|------------------------------------|-

----
[i] Erfahre mehr über uns:
www.consol.de/karriere
[i] Oder schreibe uns an:
jobs@consol.de

DIE GROSSE JAVA COMMUNITY KONFERENZ

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

Powered by trivago - Silber-Sponsor der JCON 2019

Training-Day

23. September 2019

JCON2019
www.jcon.one

11 Power-Schulungen zur Auswahl:

Observability in a Java Microservices Infrastructure
Daniel Kleuser, Tobias Gies, Daniel Heitmann, Dominik Sandjaja, *trivago*

Java Performance Tuning
Ingo Düppe, *CROWDCODE*

Create ultra-fast Java In-Memory Apps and Microservices with Java
Florian Habermann, *MicroStream* & Christian Kümmel, *XDEV*

Reactive Spring
Patrik Baumgartner, *42talents*

How to write better and effective Code in Java
Altug Altintas, *Kodcu.com*

Spring Boot goes Kubernetes
Andreas Kruse & Sebastian Sirch, *viadee*

Tame the Beast: Praktiken und Werkzeuge um den Big Ball of Mud loszuwerden
Matthias Gutheil & Martin Schmidt, *itemis*

Rapid Cross-Platform-Development mit Eclipse, Vaadin & Web-Components
Sebastian Späth, *XDEV*

Mit testgetriebener Software-Entwicklung zu einer hexagonalen Architektur
Daniel Haftstein, *itemis*

Continuous Deployment on AWS: Make Releasing the most boring Part of your Job
Steffen Grunwald & Dirk Fröhler, *Amazon Web Services EMEA*

Line Coverage ist tot - die Jagd auf Mutationen ist eröffnet
Sven Ruppert, *Vaadin*

Featured Training:

Observability in a Java Microservices Infrastructure

Do you really know how your services behave internally? When operating multiple, interconnected services, knowing what is going on inside and between them is a difficult task. You don't just need logging; you need to be able to observe what your services are doing and how they're interacting – the same things you would want to see when debugging an individual application. In this workshop, we will create multiple interconnected Java applications, based on Spring Boot, add

9:00	**Introduction to Observerability**
10:30	Kaffeepause
11:00	**Custom Metrics in a Spring Boot Application**
12:30	Mittagspause
13:30	**Metrics, Monitoring and Observerability in a Microservice Environment**
15:00	Kaffeepause
15:30	**Creating meaningful Dashboards and more things to do**
17:00	Ende

Jetzt anmelden unter: www.jcon.one

Powered by trivago - Silber-Sponsor der JCON 2019 - www.trivago.com

#JAVAPRO #Agile #Architecture

Architekturentscheidung in agilen Projekten

Was mache ich als Architekt in einem agilen Team, wenn wir im Sprint 1 vor der Entscheidung stehen, welches Architekturmuster oder welche Technologien wir einsetzen und wir innerhalb des Sprints entscheiden wollen? Einfach machen und dann refactoren? Und wenn es verschiedene Ansichten gibt, wie wir es machen können?

Das fiktive Projekt

Als Fallstudie wird ein fiktives Projekt betrachtet. Das Unternehmen ist eine Bank, die Ihren Kunden in stärkerem Maße digitale Prozesse anbieten möchte und deswegen das Projekt „Digitale Darlehensbeantragung" aufgesetzt hat. Das Ziel ist eine digitale Antragsstrecke für den Endkunden, über den Privatkredite bis zu einer gewissen Höhe voll automatisiert beantragt und bewilligt werden können. Das Projekt soll agil mit der Scrum-Methode durchgeführt werden. Zum Projektteam gehört auch ein Software-Architekt, der die Entwickler zu Technologien, Architekturmustern und Anwendungsdesign beratend unterstützen soll.

Sprint 1

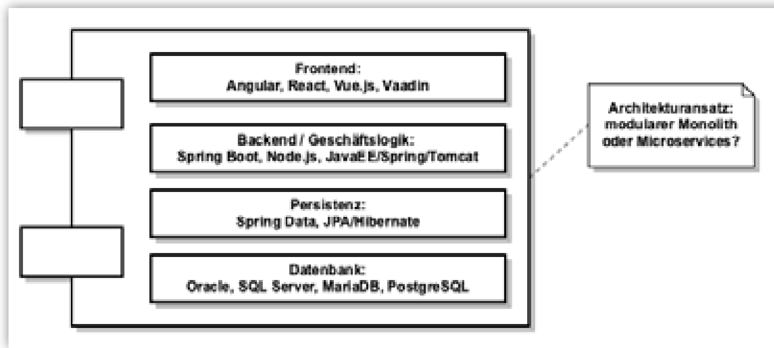
In einem vorgeschalteten Kickoff[1] tauscht sich das Team über die fachlichen Skills aus und benennt diverse Technologien, die es beherrscht, um die Stories umzusetzen. Das Team legt sich darauf fest, alle Entscheidungen einstimmig zu treffen. Der Product-Owner erläutert die Vision und hat im Vorfeld ein initiales Backlog angelegt. Er startet am ersten Tag des ersten Sprints im Planning mit der Erläuterung der ersten Story, die eine erste rudimentäre Maske für den Kunden zum Ziel hat, mit der ein Darlehensantrag nur mit Feldern für die Darlehenshöhe, die Rückzahlungsdauer, Vor- und Nachname des Kunden angelegt werden kann. Schnell kommt eine Diskussion auf, was die Anlage im technischen Sinne bedeutet und ob die eingegebenen Daten bereits in einer Datenbank gespeichert werden sollen? Und wenn ja, in welcher? Und überhaupt, mit welchem Framework soll die Oberfläche entwickelt werden, da vom Team mehrere Technologien wie Angular oder Vaadin beherrscht werden. „Wir sollten aber lieber auf React setzen, das ist die Zukunft", äußert sich ein Entwickler, woraufhin der Nächste einhakt mit: „Nein, Vue.js ist viel besser, auf der JCON habe ich einen super Vortrag dazu gehört." Da die Fachbereichsvertreter im cross-funktionalen Team nur böhmische Dörfer verstehen, versuchen Sie die Diskussion auf die Inhalte zu lenken und werfen ein: „Mit nur drei Feldern können wir keinen Antrag aufnehmen, mindestens das Geburtsdatum fehlt da noch. Und die Adresse. Und noch diverse Angaben zur Risikoeinschätzung". Dadurch fühlt sich der Business-Analyst angesprochen und weist darauf hin, dass man zuerst ein Datenmodell entwerfen müsse. Vorher stellt sich die Technologiefrage noch gar nicht.

Glücklicherweise hat das Team einen agilen Coach zur Seite, der interveniert und erklärt, dass die erste Story bewusst ein schmaler Durchstich sein soll, der sich mit minimalen Inhalten dem ersten Teil des Gesamtprozesses annähert. Er moderiert die gemeinsame Klärung im Team, wie viele Datenfelder die Story umfasst: nämlich vier. Außerdem ergänzt er gemeinsam mit dem Team zwei Akzeptanzkriterien, dass die eingegebenen Daten in einer beliebigen Form dauerhaft gespeichert werden sollen und

Autor:

Tobias Voß arbeitet als IT-Architekt in agilen Projekten bei der viadee Unternehmensberatung. Er berät Kunden im Versicherungs- und Bankenumfeld bei der Umsetzung individueller Softwaresysteme mit den Schwerpunkten Java, Architektur und Prozessautomatisierung und leitet den Kompetenzbereich „Java und Architektur" der viadee.

https://blog.viadee.de
https://twitter.com/tobiaslvoss
https://www.xing.com/profile/Tobias_Voss21/
https://github.com/viadee
tobias.voss@viadee.de



Schichtendiagramm mit möglichen Technologien. (Abb. 1)

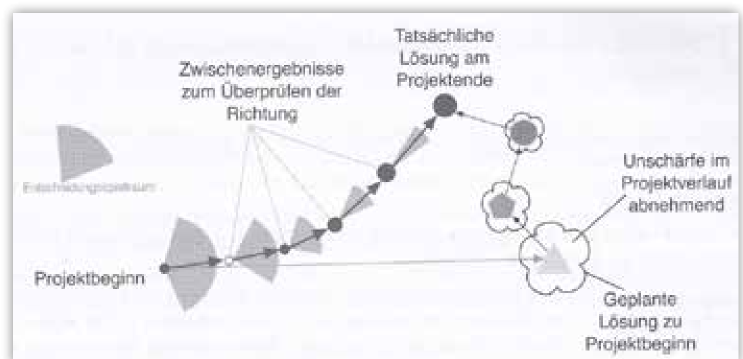
dass die Story an dem Punkt im Prozess endet und keine weitere Verarbeitung der Daten beinhaltet. Ein Datenmodell kann ein Ergebnis sein, wenn das Team der Meinung ist, dass es für die kleine Menge von Feldern sinnvoll ist. Zur Auswahl der Technologien weist er auf die Kompetenz des Teams und den festgelegten Entscheidungsprozess hin, mit der Anmerkung, dass für Nichttechniker eine Enthaltung durchaus akzeptabel ist. Dann leitet er zur ersten Schätzrunde im Planning-Poker über, da keine Verständnisfragen mehr zum Inhalt der Story auftauchen.

Die Schätzungen weichen extrem voneinander ab, gehen von 2 bis 40 Punkten und es gibt auch mehrere Teammitglieder, die keine Schätzung abgeben können. Nach einer kurzen Diskussion wird klar, dass vor allem die Entwickler eine sehr hohe Unsicherheit bezüglich der Umsetzung verspüren, da noch keine Architektur festgelegt wurde. Moderiert durch den Architekten arbeitet das Team das Schichtendiagramm (Abb. 1) heraus, in dem die potenziellen Technologien aufgeführt sind, die das Team beherrscht oder für gute Kandidaten hält. Dabei treten verschiedene Ansichten durch die unterschiedlichen Skills zutage. Es wird schnell klar, dass die Konsensbildung zur Architektur und Technologieauswahl in dem neu zusammengestellten Team nicht innerhalb des Plannings möglich ist. Begleitet von großen Bedenken einigt sich das Team auf eine Schätzung von 8 Story-Points und darauf, dass nur diese erste Story für den Sprint 1 ausgewählt wird, um Raum für die Basisarbeiten und sorgfältig getroffene Entscheidungen zu lassen.

Zurück zu Sprint 0?

Ernüchtert von der geschilderten Erfahrung im Sprint 1 der Fallstudie mag das Bedürfnis aufkommen, die Architektur bereits im Vorfeld im Rahmen eines sogenannten Sprint 0 oder einer Vorstudie festzulegen. Es gibt in der Literatur beschriebene Verfahren zur Architekturbewertung wie die Architecture-Tradeoff-Analysis-Method (ATAM)[2], die in einem detaillierten Prozess die Geschäfts- und Architekturziele in Einklang bringen sollen. Dazu werden unter Einbeziehung der Stakeholder konkrete Qualitätsmerkmale definiert und die Architektur hinsichtlich praktischer Szenarien analysiert und qualitativ bewertet.

Unabhängig davon, ob man der beschriebenen Methode folgt oder nach eigenen Ansätzen arbeitet, tritt bei Vorstudien häufig das Problem auf, dass eine Gesamtarchitektur für den gesamten Projektverlauf festgelegt werden soll, obwohl die Anforderungen gerade den Entwicklern zu diesem Zeitpunkt so gut wie gar nicht bekannt sind. Es gibt einen großen Entscheidungsspielraum und viel Unschärfe, die erst im Projektverlauf abnehmen. Die tatsächliche Lösung am Projektende weicht häufig deutlich von der geplanten Lösung zu Projektbeginn ab (Abb. 2). Schnell kommen dabei auch Fragen zur Priorisierung der Anforderungen auf, die im Vorfeld des Projekts nur getroffen werden können, wenn der agile Wert „Reagieren auf Veränderung"[3] verletzt wird. Für ein agiles Projekt kann eine Vorstudie nicht empfohlen werden, die ein größeres Ziel verfolgt, als alle technischen Fragen zur Umsetzung der Stories im Sprint 1 zu klären. Die Vorstudie wird zwar sicherlich Ergebnisse für einen Teil der Fragestellungen liefern, aber ob diese Ergebnisse in den ersten Sprints überhaupt relevant werden, ist nicht sichergestellt. Genauso wenig, wie die Qualität der Ergebnisse ohne den inhaltlichen Austausch über die Anforderungen in einem cross-funktionalen Team gewährleistet werden kann. In der agilen Softwareentwicklung ist es ein Trugschluss zu glauben, man könne ex ante die komplette Architektur eines Systems am Reißbrett entwerfen und damit den gewünschten Funktionsumfang vollständig abdecken[4].



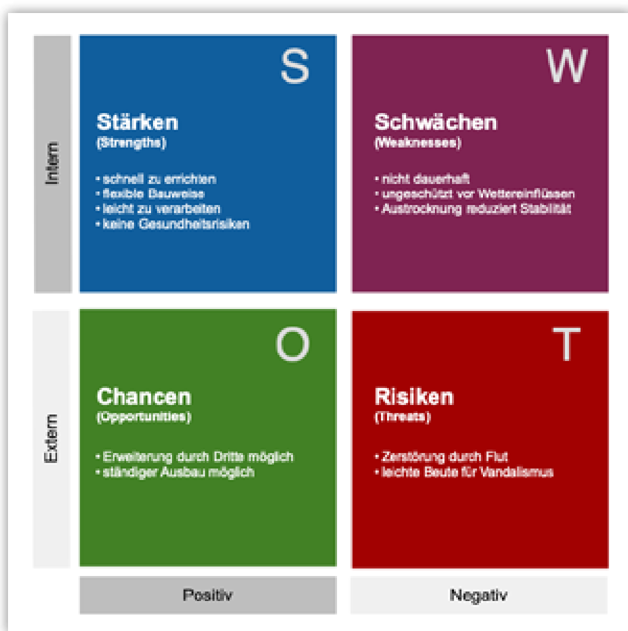
Entscheidungsspielraum und Unschärfe in Projekten[5]. (Abb. 2)

Es gibt verschiedene Möglichkeiten der Problemstellung zu begegnen, dass die Architektur begleitend zum System aufgebaut wird. Um der fachlichen Priorisierung zu entsprechen, können die Architekturanalysen oder Technologieentscheidungen auf den spätestmöglichen Zeitpunkt verschoben werden, sobald die Notwendigkeit zur Umsetzung in einer Story gegeben ist. Dann werden die Analysen und Entscheidungen in eine passende fachliche User-Story integriert. Bei Entscheidungen mit größerer Tragweite ist eine Architektur-Bugwelle[6] empfehlenswert, die alles zusammenfasst, was für die Spitze des Backlogs in den nächsten Sprints relevant ist. Die entsprechenden Aufgaben können bei skaliert agilen Projekten durch ein eigenes Architekturteam[7] bearbeitet werden. Allerdings sind

auch temporär gebildete Task-Forces aus Mitgliedern mehrerer Scrum-Teams durchaus empfehlenswert, da die getroffenen Entscheidungen Auswirkungen auf alle oder zumindest mehrere Teams haben. Im Rahmen der Bugwelle kann dann vorbereitete Forschung und Entwicklung stattfinden, welche Optionen zur Umsetzung der Stories zur Verfügung stehen, die idealerweise durch einen Prototypen nachgewiesen werden. Jeder Prototyp sollte dabei groß genug sein, um aussagekräftig zu sein, aber auch klein genug, damit die investierten Aufwände bei einer anderen Entscheidung keinen Hemmschuh aufgrund des Effekts der versunkenen Kosten darstellen.

Architekturen mit der SWOT-Analyse bewerten

Um verschiedene Optionen für die Architektur zu bewerten und die Architekturentscheidung zu treffen und zu begründen, kann die SWOT-Analyse als Methode zur Unterstützung herangezogen werden, die Stärken (strengths), Schwächen (weaknesses), Chancen (opportunities) und Risiken (threats) der Architekturoptionen expliziert. Die SWOT-Analyse ist ursprünglich ein Instrument der strategischen Planung und dient der Positionsbestimmung und der Strategieentwicklung von Unternehmen sowie anderer Organisationen[8].



Beispiel einer SWOT-Analyse für eine Sandburg. (Abb. 3)

Angewendet auf Architektur - in **(Abb. 3)** am Beispiel einer Sandburg am Strand - entsprechen die Stärken, wie z.B. die flexible Bauweise, objektiv positiven, inhärenten Eigenschaften einer Sandburg. Demgegenüber stehen die Schwächen, die manch eifriger Sandburgenbauer am eigenen Leib erfahren musste, dessen Burg vom Winde verweht wurde. Bei den Chancen und Risiken geht der Blick über den Tellerrand, d.h. über die Eigenschaften der Sandburg selbst hinaus. Diese externen Faktoren schließen die Umwelt ein und entsprechen möglicherweise eintretenden Ereignissen, wie einem ständigen Ausbau im positiven Fall bei gutem Wetter, oder der Zerstörung durch eine unerwartet hohe Flutwelle im negativen Fall.

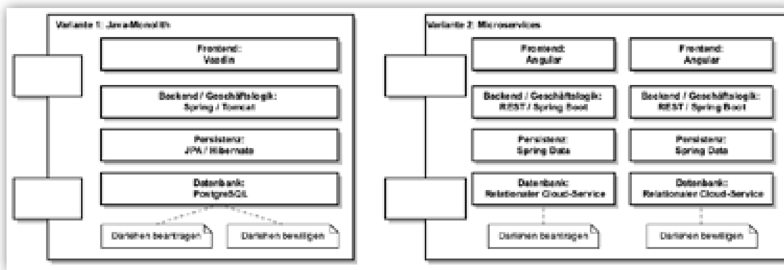
Im Gegensatz zu einfachen Vergleichen mit Positiv-/Negativlisten trennt die SWOT-Analyse zwischen einer (Team-/Unternehmens-)internen Analyse und externen Faktoren. Die internen Stärken und Schwächen entsprechen durch Erfahrung belegte Fakten, die das Team gut einschätzen kann. Demgegenüber basieren die Chancen und Risiken der externen Analyse sehr stark auf Annahmen, die durch Ausprobieren der Architekturoption validiert werden müssen. Diese beiden Aspekte zu trennen erleichtert die Bewertung durch das Team und die Erläuterung und Begründung für den Product-Owner oder die Stakeholder und vermeidet eine Vermischung von Fakten und Annahmen. Als Nebeneffekt kann ein klarer Auftrag abgeleitet werden, welche Chancen oder Risiken in einem der nächsten Sprints konkret untersucht werden sollen. Bei den Kriterien zur Bewertung der Architektur sollte man sich in erster Linie an den nichtfunktionalen Eigenschaften oder Qualitätsmerkmalen wie Bedienbarkeit, Performanz, Flexibilität und Wartbarkeit orientieren. Dadurch bekommen oft langwierige Architekturdiskussionen einen klaren Fokus.

Die SWOT-Analyse stiftet darüber hinaus einen sozialen Nutzen für das Team, sofern sie gemeinsam erarbeitet wird. Die Erstellung der Analyse fördert den Wissensaustausch und die Konsensbildung im Team und am Ende steht (hoffentlich) eine gemeinsame Teamentscheidung, bei der einzelne Meinungen sowohl als Chance oder auch Risiken Einfluss auf das Ergebnis haben. Diese Art der Entscheidungsfindung im Gegensatz zur einsamen Entscheidung eines Architekten ist zwar anstrengend, aber langfristig wertvoll für die Akzeptanz im Team und die Selbstorganisation des Teams.

SWOT-Analyse zur Entscheidung zwischen Varianten

Für die Fallstudie der Darlehensbeantragung haben sich die Entwickler und der Architekt im Scrum-Team direkt nach dem Planning für einen Workshop zurückgezogen und zwei Varianten für den grundlegenden Architekturansatz erarbeitet. Das Team besteht ausschließlich aus Java-Entwicklern, die überwiegend mit Spring und Java-EE gearbeitet haben und nur rudimentäre Kenntnisse in JavaScript-Frontends und Microservice-Technologien besitzen. Die beiden Varianten in Form eines Java-Monolithen und einer Microservice-Architektur mit JavaScript-Frontend und Java-Backend sind in **(Abb. 4)** dargestellt. Für beide Varianten erstellt das Team gemeinsam eine SWOT-Analyse, siehe **(Abb. 5)** und **(Abb. 6)**.

Die Ergebnisse der SWOT-Analysen erheben keinen Anspruch auf universelle Gültigkeit für die beiden Architekturansätze - ganz im Gegenteil wird jedes Team zu einer eigenen Bewertung kommen. Insbesondere die qualitative Bewertung im Rahmen der



Varianten für den grundlegenden Architekturansatz. (Abb. 4)

SWOT-Analyse für den Java-Monolithen. (Abb. 5)

SWOT-Analyse für die Microservice-Architektur (Abb. 6)

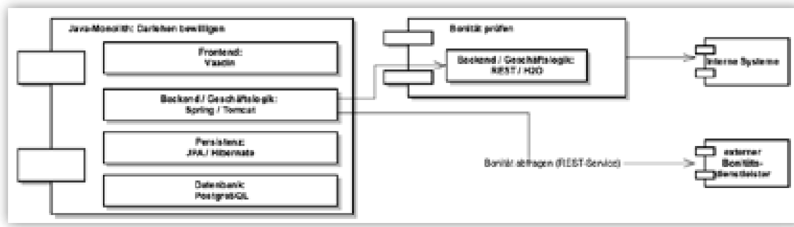
Abwägung zwischen Stärken und Schwächen sowie Chancen und Risiken kann nur jedes Team angewandt auf seine eigene Situation zielgerichtet durchführen. Gerade bei den Risiken der Microservice-Architektur fällt auf, dass diese ausschließlich auf den fehlenden Erfahrungen des Teams mit Microservice-Technologien resultieren. Aus der SWOT-Analyse lässt sich kein direktes Ergebnis für eine der beiden Varianten ablesen, sondern die Entscheidung muss vom Team durch die Bewertung der einzelnen Einträge in der SWOT-Matrix getroffen werden. Dabei sind die Stärken der einen Variante oft die Schwächen und Risiken der anderen Variante und umgekehrt. Die Methode kann, wie alle anderen Methoden auch, keine Entscheidung fällen, sondern nur die Entscheidungsfindung erleichtern. Als weiterer Mehrwert können die Chancen und Risiken zur Formulierung von Prüfaufträgen zur retrospektiven Bewertung der Architekturentscheidung nach Abschluss eines Prototypen oder nach mehreren Sprints genutzt werden.

In der fiktiven Fallstudie entscheidet sich das Team für den Java-Monolithen, da sowohl für die Schwächen als auch die Risiken von einer sehr geringen Eintrittswahrscheinlichkeit ausgegangen wird und die Chancen der hohen Entwicklungsgeschwindigkeit zu Beginn den Wunsch der Stakeholder nach schnellen Erfolgen unterstützt. Am Ende der ersten beiden Sprints, nachdem der erste Durchstich als Prototyp mit der gewählten Architektur umgesetzt ist, führt das Team eine dedizierte Architekturretrospektive zur Überprüfung der SWOT-Analyse durch. Da die Chancen realisiert werden konnten und die Risiken nicht eingetreten sind oder durch entsprechende Gegenmaßnahmen adressiert wurden, wird an der gewählten Architektur festgehalten.

SWOT-Analyse zur Bewertung neuer Optionen

Ein paar Sprints später ist die Darlehensbeantragung für die wichtigsten Anwendungsfälle implementiert und das Team hat angefangen, die Darlehensbewilligung umzusetzen. Im ersten Schritt wurde die Bewilligung als manueller Prozess im Java-Monolithen umgesetzt. Im Backlog-Refinement stellt der Product-Owner die Bonitätsprüfung des Kunden als ersten automatisierten Schritt der Bewilligung als eine der nächsten Stories vor. Die Bonitätsprüfung soll einerseits intern bestehend auf gesammelten Daten des Kunden mit Künstlicher Intelligenz und andererseits durch Anfragen bei externen Bonitätsdienstleistern erfolgen. Schnell entsteht eine technische Diskussion, welche Technologie für die Integration einer Künstlichen Intelligenz (KI) die Richtige ist, da zwei Teammitglieder erste Erfahrungen mit der API von H2O[9] und der Integration KI in Geschäftsprozesse mit bpmn.ai[10] gesammelt haben. Auch über die Anbindung der REST-Services externer Dienstleister und die richtige Architektur und Technologie dafür wird fleißig diskutiert. Das Team einigt sich darauf, die Stories erstmal zurückzustellen und zuerst eine SWOT-Analyse für die Architektur der Bonitätsprüfung durchzuführen.

Als Vorbereitung erstellt das Team erneut eine Architekturskizze, um ein gemeinsames Verständnis herzustellen und eine Ausgangsbasis für die Bewertung zu haben (Abb. 7). Dabei



Architekturskizze für die Bonitätsprüfung. (Abb. 7)

Technologie bis zu ganzen Teams für den übergreifenden Architekturstil. Die Trennung zwischen einer internen und externen Analyse lässt sich für Architekturfragestellungen auf die Dimensionen bekannter Fakten basierend auf gemachten Erfahrungen und Annahmen, die validiert werden müssen, umdeuten. Damit strukturiert die SWOT-Analyse die Einschätzungen im Team und öffnet neue Perspektiven. Trotz alledem kann die Methodik der SWOT-Analyse den kreativen Prozess in einem agilen Team nur unterstützen. Architektur ist eine Herausforderung mit vielen Grautönen und die Entscheidung muss das Team treffen. Das Analyseergebnis dient einerseits als Begründung für den Product-Owner und die Stakeholder und hilft andererseits, die Chancen und Risiken explizit zu machen, die durch Prototypen validiert werden sollten. Dabei sollte das Team auch den Mut zur richtigen Reaktion auf Veränderungen zeigen, wenn z.B. die Chancen überschätzt wurden, und die getroffenen Entscheidungen hinterfragen und bei Bedarf revidieren.

Und schließlich sollten agile Teams einen gesunden Pragmatismus an den Tag legen, da Architektur manchmal in den Rang einer exakten Geheimwissenschaft erhoben wird, die nur wenige Auserwählte perfekt beherrschen. Tatsächlich beschreibt der Tweet „Warum nennen wir es eigentlich Softwarearchitektur und nicht „Permanenter Kompromisszwang unter verschieden großen Übeln mit langzeitiger Kollektivschuldübernahme?"[11] viel besser die Realität in der Entscheidungsfindung zur Architektur vieler Anwendungen, die unter ständigem Kosten- und Zeitdruck permanent weiterentwickelt werden. Durch die Anwendung der SWOT-Analyse zur Unterstützung der Entscheidung kann aber trotzdem die nötige Professionalität an den Tag gelegt werden, die Übel zu bewerten und die Kompromisse explizit zu machen.

entsteht schnell der Ansatz, die Prüfung der Bonität und die Abfrage bei externen Dienstleistern als REST-Services zu kapseln, die im Bewilligungsprozess eingebunden werden können. Für die Implementierung der KI wird die Integration der H2O-API favorisiert, da diese am Markt etabliert ist und die fachlichen Anforderungen gut abdeckt. Da keine direkten Alternativen in der Diskussion entstehen und die Grundidee von allen geteilt wird, geht das Team direkt zur Bewertung des skizzierten Architekturentwurfs über (Abb. 8) und verständigt sich darauf, erst bei der Identifikation hoher Risiken über weitere Optionen nachzudenken. Da die Risiken beherrschbar erscheinen und für das Team weniger stark ins Gewicht fallen als die Stärken und Chancen, fällt die Entscheidung für den skizzierten Ansatz der Microservices für die Bonitätsprüfung. Als Gegenmaßnahme für die Risiken verständigt sich das Team auf einen eintägigen Hackathon, um die API der KI-Technologie auszutesten und das Wissen im gesamten Team zu verteilen. Die Gegenmaßnahme wird in Abstimmung mit dem PO als Akzeptanzkriterium für die Stories formuliert.

Fazit:

Die SWOT-Analyse ist eine geeignete Methode, um Architekturoptionen zu bewerten, bevor eine Entscheidung getroffen wird. Aufgrund ihres allgemeinen Charakters skaliert sie für Entscheidungen zwischen zwei Personen für eine konkrete

SWOT-Analyse für die Bonitätsprüfung. (Abb. 8)

Quellen:

1 Nicole Neunhöffer, Kay Hildebrand: "viadee Tutorial #1: Kickoff für Scrum-Teams", https://blog.viadee.de/kickoff-fuer-scrum-teams
2 Gernot Starke: "Effektive Softwarearchitekturen", Hanser, 2014
3 Kent Beck et al.: "Manifest für Agile Softwareentwicklung", https://agilemanifesto.org/iso/de/manifesto.html
4 Dr. Kay F. Hildebrand, Tobias Voß: „Der agile Architekt ist ein Bauingenieur", ObjektSpektrum 6/2017.
5 Übernommen aus Gernot Starke: "Effektive Softwarearchitekturen", Hanser, 2014
6 Scaled Agile Framework (SAFe): „Architectural Runway", https://www.scaledagileframework.com/architectural-runway/
7 Scrum Nexus Framework: „The Nexus Integration Team", https://www.scrum.org/resources/nexus-integration-team
8 Wikipedia: „SWOT-Analyse", https://de.wikipedia.org/wiki/SWOT-Analyse
9 https://www.h2o.ai
10 https://www.bpmn.ai
11 Stefan Pfeiffer, https://twitter.com/spfeiffr/status/1065639195685867520

#JAVAPRO #Agile #Digitalisierung

Resilienz durch Organic-Agility

Resilienz ist für die Anpassung an komplexe und volatile Situationen in jedwedem Bereich essentiell - von der Softwareentwicklung bis hin zum Management von Organisationen. Organic-Agility ist ein evolutionärer Ansatz der zeigt, warum eine kohärente Organisationskultur und agiler Leadership den Organisationen dabei helfen, Resilienz zu erzielen und sich auf die Anforderungen der heutigen Realität einzustellen.

Autor:

Marion ist eine der Gründerinnen und CEO von agile42. Die letzten 15 Jahre unterstützte sie die strategische Produktentwicklung und reagierte auf Leadership-Herausforderungen durch lokale und globale Projekte mit dem internationalen Team von agile42. Ihr Interesse an Agile, ORGANIC agility, Komplexität, Resilienz und der Veränderung von Organisationskulturen trieben bahnbrechende Projekte, wie jenen des agile42 Organizational Scan voran, während es ihre Erfahrung agile42 gestattet, mit jedem Kunden individuell den Kontakt auszubauen und gemäß deren persönlichen Bedürfnissen individualisierte Lösungen anzubieten.

https://www.organic-agility.com,
https://www.agile42.com/en/
https://twitter.com/agile42,

Was Resilienz bedeutet und warum Organisationen resilient werden müssen

Resilienz ist ein Konzept, das in verschiedenen Bereichen gilt: Bei Individuen wurde sie als einer der wichtigsten Schutzfaktoren gegen Bedrohungen von außen identifiziert, bei Materialien ist sie die Fähigkeit biegsam statt zerbrechlich zu sein, und bei Organisationen beschreibt Resilienz die Fähigkeit schnell auf Veränderungen zu reagieren. Trotz der steigenden Beliebtheit von agilen Methoden, prädominiert noch immer die Vorstellung, Versagen sei inakzeptabel, Veränderungen seien vorhersehbar und sämtliche Vorgänge könnten im Detail vorab definiert werden. Diese geschieht basierend auf der Annahme wir lebten in einer vorhersehbaren und geordneten Welt, in der sich die Zukunft mehr oder weniger wie die Vergangenheit gestaltet und wir diese deshalb planen und vorhersehen können. Die Realität sieht jedoch ganz anders aus.

Unser Arbeitsumfeld ist nicht akkurat, geordnet, robust und nicht gegen zukünftige Herausforderungen gewappnet. Das war es vermutlich noch nie, aber mit dem Einfluss der digitalen Technologie von heute, der erhöhten Vernetzung weltweit sowie der sich schnell ändernden Marktanforderungen, werden unsere Arbeitsumgebungen instabil. Alles beeinflusst alles auf eine Art und Weise, die nicht vorhersehbar und bis zum Eintreten nicht sichtbar ist. Mit anderen Worten: die Welt ist komplex.

Organisationen sehen sich dementsprechend mit drei Arten von Bedrohungen konfrontiert. Zum einen jene die wahrscheinlich sind. Mit diesen sahen wir uns schon zuvor konfrontiert und sehen sie kommen. Zum anderen jene die möglich sind. Sie könnten eintreten, müssen aber nicht - aber wir wissen mehr oder weniger wie sie sich äußern. Schließlich gibt es Bedrohungen die zwar plausibel, aber in keinster Weise greifbar oder einschätzbar sind - und diese sind die Gefährlichsten. Bei dieser Art von Bedrohung haben wir keinerlei Möglichkeit uns darauf vorzubereiten und sie könnten sich gänzlich destruktiv auswirken. Diese bezeichnet man als Ereignisse geringer Wahrscheinlichkeit, aber starker Auswirkungen. Betrachten wir das Beispiel von Kodak, die 2014 in Konkurs gingen, weil sie nicht Schritt halten konnten mit der digitalen Fotoindustrie, obwohl sie die ersten waren die diese Technologie entdeckt hatten! Kodak war nicht resilient.

Manche Organisationen glauben, dass robust zu sein die beste Art der Existenz sei. Maschinen stellen ein typisches Beispiel dar, für etwas das robust ist. Sie sind so konzipiert, dass sie eine bestimmte wiederholbare Aufgabe erfüllen und wenn ein Teil kaputt oder verloren geht, wird die Maschine gewartet. Eine Organisation jedoch funktioniert nicht wie eine Maschine, sondern vielmehr wie ein lebendiger Organismus der sich anpasst und auf Herausforderungen schnell und flexibel reagieren kann. Eine resiliente Organisation zu sein bedeutet die Fähigkeit zu steter Innovation und Entwicklung und zur Umfunktionierung des Vorhandenen zum eigenen Vorteil, sodass sie einen Vorsprung gegenüber den Umständen hat, statt nur zu reagieren. Dieser Ansatz, Organisationen eher biologisch anstatt mechanisch zu betrachten, ist Teil der Oraganic-Agility, einem Meta-Framework, auf das nachfolgend näher eingegangen wird.

Resilienz in Organisationen

Resiliente Systeme und Organismen haben viel gemeinsam. Sie zeichnen sich gemeinhin durch strukturelle Verteilung und die Möglichkeit aus, alternative Wege zu finden. Diese Fähigkeiten ermöglichen Organisationen auf Veränderung zu reagieren. Der Hauptgedanke dahinter ist, dass die für den Umgang mit einer Krise notwendigen Eigenschaften und Fähigkeiten erlernt werden, bevor man sie benötigt, um im Fall der Bedrohung gewappnet zu sein. Diese Fähigkeiten anzuwenden bedeutet häufig, dass sich die Organisationsstruktur und -kultur verändern muss.

Um ein grobes Bild davon zu zeichnen, wie sich eine resiliente Organisation mehr oder weniger auszeichnet, stelle man sich eine Kultur vor, die kohärent und dennoch diversifiziert ist, mit schnellen Abläufen zur Entscheidungsfindung, die stets kontextuelle Bedürfnisse mit einbezieht, die auf Wertbedürfnisse reagiert - eine Strategie, die Struktur und Flexibilität auf eine Art miteinander vereint, die den Fortschritt auch durch Komplexität und Unsicherheit erlaubt sowie Betriebsabläufe, die eine solche Kultur unterstützen statt diese auszubremsen. Inkrementelle Umsetzung der neuen Vorgehensweisen, zeitnahe, regelmäßige Feedback-Intervalle sowie zuverlässige Methoden zur Informationserfassung ermöglichen die Fähigkeit zur Entscheidungsbildung auf Grundlage solider Daten. Der gesamte Prozess ist dynamisch und steuert auf Veränderung durch Interaktion und Experimente zu. Die fünf Grundsätze von Organic-Agility entsprechen diesen Eigenschaften und sie sollen als Gerüst fungieren, um die „neue Organisation" zu schaffen.

Entwickler wissen um Resilienz, wie sie Systeme konzipieren die Versagen annehmen und deshalb über multiple Backups verfügen. Sie verstehen den Zusammenhang zwischen Resilienz und Wert sowie die Komplexitäten, die von hochgradig zusammenhängenden, voneinander abhängigen Systemen hervorgerufen

werden. De facto sind die Grundsätze der Resilienz nicht viel anders als die für Organisationen. Für jemanden, der Erfahrung mit Java oder einer anderen Programmiersprache hat, kann es hilfreich sein zu sehen, wie diese Grundsätze bei Umwandlung in konkretes Softwaredesign weniger abstrakt werden.

Inzwischen könnte diese Beschreibung allen Lesern bekannt vorkommen, die die Agile-Welt schon kennen oder Agile bereits anwenden. Es gibt viele Elemente von Agility, die eine resiliente Organisation stützen und fördern können, sowohl bezüglich der Kultur als auch in der Ausführung. Die Dezentralisation bei der Entscheidungsfindung beschleunigt dadurch, dass zusätzliche, intervenierende Instanzen entfernt werden. Die Reaktionsfähigkeit auf sich ändernde Umstände ist auch mit einer komplexen Umgebung perfekt vereinbar und Flexibilität ist eine der Kerneigenschaften von Agilität. Dennoch werden derzeit mehr und mehr Bedenken und Kritik laut, dass Agile seine Agilität verliert, da zu präskriptiv oder dogmatisch mit agilen Ansätzen umgegangen wird.

Bereit für Agility

Agilität bietet viele Vorteile für Unternehmen, und das hat sich nicht geändert. Durch dogmatisches Vorgehen oder durch bloßes Durchführen von Praktiken, ohne Verständnis für die zugrundeliegenden Prinzipien, wird Agilität in ein schlechtes Licht gerückt. Darüber hinaus werden oft gedankenlos Elemente aus einem Kontext herausgerissen und in einem anderen, nicht relevanten Kontext angewandt. Dennoch bietet Agilität nach wie vor: Geschwindigkeit, Autonomie, Reaktionsfähigkeit auf Bedürfnisse, Flexibilität und schrittweise Verbesserung. Wichtig ist aber, dass diese individuell, kontextspezifisch sowie undogmatisch ausfallen und für die Wissensära anstatt für das Industriezeitalter gemacht sind. Hierbei hilft Organic-Agility.

Organic-Agility ist ein individualisierter, evolutionärer Ansatz zum Erreichen von Resilienz, der zu mehr Autonomie ermutigt, eine kohärente Organisationskultur schafft und auf dem Wesen der Komplexität basiert.

Fünf Grundsätze und Prinzipien bilden die Basis, bzw. schaffen einen Rahmen für die notwendigen Veränderungen. Diese hängen von keinem anderen Vorgehen oder einer Methodik ab und sind mehr Leitfaden als Vorschrift. Sie ermöglichen es der Organisation, die neuen Denkweisen und Handlungen in die DNA des Unternehmens einfließen zu lassen. Organic-Agility erkennt die Tatsache an, dass man nicht von einem auf den anderen Tag agil und resilient werden kann. Es müssen die Grundlagen gelegt werden und es bedarf der praktischen Unterstützung. Ziel ist es, ein Gleichgewicht herzustellen und ein eigenes sinnvolles Vorgehen zu finden.

Außer den Prinzipien offeriert Organic-Agility außerdem einen neuen Ansatz für Führungskräfte. Das Organic-Leadership-Framework bezieht kontextuelle Entscheidungsfindung und die Fähigkeit ein, Leadership in die Organisation hineinzutragen. Das Leadership gewinnt die Fähigkeit, wachsam gegenüber Veränderungen zu sein und Bestehendes zu verändern, um neuen Bedürfnissen (z.B des Marktes) gerecht zu werden.

Mit dem Leadership fängt es an

Das Leadership einer Organisation muss mit der Veränderung starten. Die Führungskräfte und ihr Verhalten haben eine katalysierende Wirkung für die Veränderung und beeinflussen die Organisation. Aber wie findet diese Veränderung in einem komplexen Umfeld statt? Und wie muss sich unser Verständnis von Leadership in einer resillienten Organisation wandeln? Was wir von außen sehen ist, wie sich Menschen innerhalb einer Organisation verhalten und welche Ergebnisse sie erzielen. Beeinflusst wird dies jedoch durch das was im Hintergrund von statten geht. Handlungen und wie Menschen ihre Arbeit verrichten, basiert darauf was sie glauben, im Sinne von: „so tun wir die Dinge hier". Das Leadership sollte also diese tieferliegende Ebene adressieren. Der Fehler den viele Führungspersonen häufig begehen ist, dass sie glauben sie könnten etwas verändern, indem sie einfach die Veränderung verbalisieren oder die wohldurchdachten Werte lediglich auf ein Plakat drucken. Tatsächlich bringen solch explizite Erklärungen Menschen dazu, diese zwar vordergründig zu erfüllen, eine echte Veränderung wird so jedoch nicht erzielt. Schlimmer noch: sollten diese Aussagen im Widerspruch zur Realität stehen, führt jene Inkonsistenz zu einer Zunahme der negativen Gefühle und Unzufriedenheit. Dies bedeutet nicht, dass Kultur nicht veränderbar oder abstrakt ist. Der erste Schritt zur Veränderung der Organisationskultur zur Verstärkung von resilienzunterstützender Eigenschaften ist, diese zu bewerten. Diese Bewertung ist möglich mittels Erhebung und Beobachtung von Gewohnheiten, Verhaltensmustern, Ritualen oder Storys von Erfolg und Versagen, die wiederum Ausdruck der Kultur sind. Es gibt sehr gute Tools, die bei der Darstellung der Organisationskultur helfen, wie dem Organizational-Scan, kurz OrgScan. Diese helfen dabei, Dinge wie Rituale zu beobachten und zu analysieren sowie diese Erkenntnisse einzusetzen, um gewünschtes Verhalten zu verstärken und weniger Gewünschtes zu vermeiden.

Wenn Organic-Agility über Leadership spricht, meint es nicht allein eine Jobbeschreibung oder die Charaktereigenschaft eines Menschen, sondern die grundsätzliche Fähigkeit aller Mitarbeiter ein Leader zu sein - abhängig vom Kontext und den im Moment notwendigen Fähigkeiten (Build-Leadership-Capability). Diese Idee von Leadership, weg von den hierarchischen und stabilen Modellen des Industriezeitalters und hin zu einer volatilen Wissensära, fördert die Resilienz auf vielfache Art. Sie erkennt, dass eine Organisation heute Menschen benötigt, die die Werte der Organisation leben, selbstständig denken und Verantwortung übernehmen. Eine passende und kohärente Kultur unterstützt die Strategie eines Unternehmens und sorgt dafür, dass Strategy lebbar und anwendbar wird. So wird Resilienz möglich.

#JAVAPRO #Agile

Verantwortung in einem Team

Aus diffusem Verständnis von Verantwortlichkeit entstehen in Projekten oft Streitereien und unnötiges Hin und Her. Wenn es um Verantwortung geht, ist es hilfreich, sich über bestimmte Zusammenhänge bewusst zu sein. Dieser Artikel zeigt auf, welche klassischen Fehler in der Praxis gemacht werden und wie man Verantwortung richtig (ver-)teilt.

Stellen Sie sich folgendes Szenario vor: Jemand legt 50 Euro auf den Tisch mit der Bitte: „Könnte bitte mal jemand auf das Geld aufpassen? Ich muss kurz weg." Und schon ist derjenige raus aus der Tür. Gleich danach schnappt sich ein anderer blitzschnell das Geld und verschwindet damit. Denken Sie kurz darüber nach was passieren würde?

Die Praxis zeigt, dass die Beobachter zunächst ratlos sind und sich gegenseitig beobachten. Vielleicht übernimmt eine Person nach einer Weile die Initiative und versucht zu klären, was zu tun ist und wer es macht. Möglicherweise nimmt ein beherzter Teilnehmer die Verfolgung auf. In aller Regel kommt es aber zu einer Verzögerung oder zu einer kompletten Blockade einer Reaktion. Die Gründe dafür liegen in der Unverbindlichkeit, der entstandenen Verantwortungsdiffusion und einer gegenseitigen Blockade. Schauen wir uns das einmal genauer an.

Verbindlichkeit

Durch die unverbindliche Übertragung von Verantwortung auf die Gruppe („Könnte bitte mal jemand ...") fühlt sich keiner wirklich verantwortlich. Es ist zum einen nicht klar, an wen die Bitte gerichtet wurde. Die Teilnehmer fragen sich: „Meint er mich, meinen Kollegen oder uns alle?" Zum anderen gab es keinerlei aktive Bestätigung, dass jemand der Bitte auch nachkommen will. Verbindlichkeit entsteht dadurch, dass beiden Seiten zumindest klar ist, wer was wann zu tun hat und die Person oder Personen dies auch aktiv bestätigen - im einfachsten Fall durch ein Nicken oder „Okay". Noch besser wirkt eine aktive Wiederholung der Aufgabenstellung durch die Personen, die die Aufgabe übernehmen. Je eindeutiger die Bestätigung und je mehr Zeugen dieser Zusage beiwohnen, desto höher ist die gefühlte Verpflichtung der Verantwortung gerecht zu werden. Der Effekt der Verbindlichkeit wird noch verstärkt, wenn die Übernahme einer Aufgabe als eigene Entscheidung gesehen wird. Die gefühlte Verantwortung für Aufgaben ist größer, wenn sie aus freiem Willen oder aus Überzeugung angenommen werden. Wird eine Person gezwungen Verantwortung zu übernehmen, liegt für sie die gefühlte Verantwortung eher beim Aufgabengeber. Deshalb sollten wir Aufgaben so übertragen, dass die Entscheidungsfreiheit der ausführenden Person nicht unnötig eingeschränkt wird. Gewehrte Entscheidungs- und Gestaltungsspielräume steigern die Verantwortungsbereitschaft.

Autor:

Peter Siwon beschäftigt sich seit nunmehr über 20 Jahren mit den Themen Gehirn und Psyche im Zusammenhang mit der Projektarbeit in technischen Berufen.
Über 30 Jahre Berufspraxis in Forschung, Entwicklung, Marketing, Vertrieb, Training, Coaching, Beratung und Geschäftsführung gaben dazu viele Impulse. Er engagiert sich als Trainer, Coach, Lehrbeauftragter, Sprecher, Buchautor und Kolumnist für die menschliche Seite des Projekterfolgs.

Verantwortungsdiffusion und gegenseitige Blockade

Die gefühlte Verantwortung jedes Einzelnen wird umso kleiner, je größer die Gruppe ist, auf die die Verantwortung allgemein übertragen wird. Die Verantwortung verteilt sich diffus über die Gruppe. Wenn 100 Personen angesprochen werden, spürt jeder zwar nicht im mathematischen, aber im übertragenen Sinne nur ein Hundertstel der Verantwortung. Darunter leidet natürlich auch die Bereitschaft aktiv zu werden. Zudem beobachten sich die Personen einer Gruppe gegenseitig, um herauszufinden, ob eine Reaktion angemessen ist. Mit der dadurch entstehenden Verzögerung verunsichern sich die Personen wiederum gegenseitig. Intuitiv geistern Fragen durch die Köpfe wie: „Ist es nicht wichtig, dass jemand handelt?", „Ist es zu gefährlich zu handeln?" oder „Ist es nicht erwünscht, dass jemand handelt?" Die so entstehende Verzögerung hat schon so manchen Menschen das

Leben gekostet, weil die umstehenden potentiellen Helfer sich gegenseitig blockiert und so wertvolle Zeit vergeudet haben.
Die Verantwortungsdiffusion und die gegenseitige Blockade lassen sich durch die eindeutige Ansprache von Personen vermeiden: „Frau A, würden Sie bitte die Aufgabe übernehmen?" Nach dieser Aufforderung sollten wir sicherstellen, dass die Person ihre Bereitschaft aktiv bestätigt. Wenn das ganze Team Verantwortung übernehmen soll, sollte jeder einzelne bestätigen, dass er die Gesamtverantwortung mitträgt. Keiner kann sich hinter einem Baum verstecken, an dem das Schild „Die anderen sind schuld" genagelt ist. Allerdings erfordert das auch einen „Alle für einen - einer für alle" Teamgeist – und der fällt nicht vom Himmel.

Verantwortung kann nur als Kopie weitergegeben werden

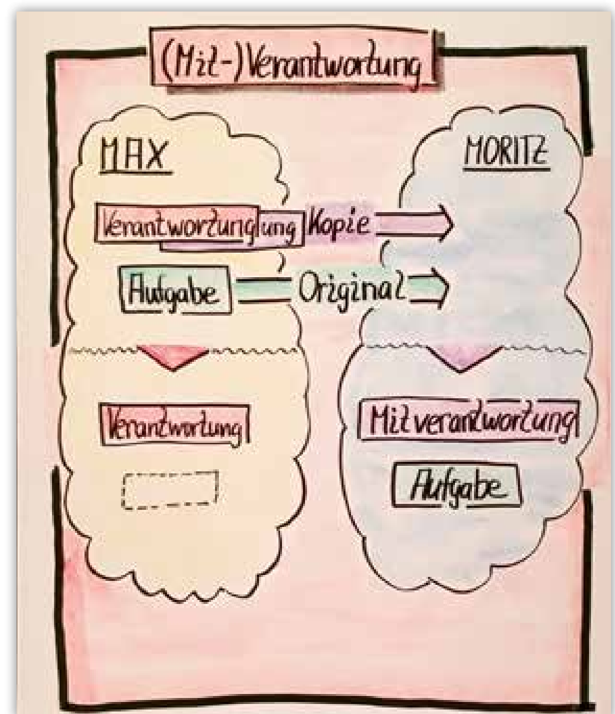
Die einfachste Lösung für das „50 Euro Problem" ist, mit einer Person eine verbindliche Vereinbarung zu treffen: „Herr Müller, würden Sie bitte die 50 Euro für mich aufbewahren, bis ich Sie bitte, sie mir zurückzugeben?" Meist nimmt der Teilnehmer das Geld mit verschmitztem Grinsen. Herr Müller wird gebeten die Bitte zu bestätigen, was er auch gerne macht. Er hat also jetzt die Verantwortung für das Geld. Doch was ist mit dem Auftraggeber? Wurde die Verantwortung für die 50 Euro abgegeben?
Stellen Sie sich vor, die 50 Euro wären beispielsweise für ein Essen geplant gewesen. Herr Müller hat das Geld unglücklicherweise verloren und konnte es deshalb nicht zurückgeben. Damit wäre die Essensplanung hinfällig. Natürlich wird in diesem Fall der Auftraggeber für den Verlust verantwortlich gemacht, da es seine Entscheidung war, Herrn Müller das Geld anzuvertrauen. Die Verantwortung bleibt also auch immer bei der Person, die die Aufgabe delegiert. Sie kann nur eine Kopie der Verantwortung für die Ausführung der Aufgabe an den Ausführenden übergeben. Selbst wenn Herrn Müller das Geld gestohlen wird, wird seine und die Verantwortung des Auftraggebers nicht mit gestohlen. Sogar der Dieb erhält eine Kopie der Verantwortung. Denn, wenn er erwischt wird, wird er zur Verantwortung gezogen. Herr Müller trägt die Verantwortung für die 50 Euro, weil er sich bestehlen ließ. Der Auftraggeber trägt die Verantwortung für das Geld, weil er es Herrn Müller anvertraut habe. Der Dieb trägt die Verantwortung, weil er gegen ein Gesetz verstoßen hat. Herr Müller wäre auch nicht aus der Verantwortung entlassen, wenn er das Geld an Frau Schmidt zur Aufbewahrung weitergegeben hätte und stattdessen sie bestohlen worden wäre. Die einzige Möglichkeit für ihn, die Verantwortung wieder loszuwerden, besteht darin, das Geld (zusammen mit der Kopie der Verantwortung) an den Auftraggeber zurückzugeben.

Natürlich kann die Verantwortung für die Aufbewahrung der 50 Euro auch auf fünf Personen verteilt werden, indem beispielsweise jeder 10 Euro erhält. Damit würde jede dieser Personen auch eine Verantwortungskopie für ihre 10 Euro übernehmen. Die Gesamtverantwortung bleibt weiterhin beim Auftraggeber.

Verantwortung im Projekt

Diese Überlegungen lassen sich sehr gut auf die Verantwortung der Projektleitung übertragen. Sie ist in der Regel gesamtverantwortlich für das Projekt und hat die Aufgabe und Verantwortung, Projektaufgaben und damit auch die Verantwortung sinnvoll aufzuteilen. Sinnvoll bedeutet, dass jede Aufgabe eindeutig und verbindlich einer Person, einer Gruppe von Personen oder einer juristischen Person (Zulieferer) zugewiesen wird, die dieser Aufgabe und der damit verbundenen Verantwortung gerecht werden kann. Läuft etwas schief, sind an erster Stelle die Personen in der Pflicht eine Lösung zu finden, die die Aufgabe übernommen haben. Sie sollten eine faire Chance bekommen, ihrer Verantwortung gerecht zu werden. In jedem Fall bleiben auch alle in der Lösungsverantwortung, die in irgendeiner Form an der Weitergabe der Aufgabe und der damit verbundenen Verantwortung beteiligt waren. Sie unterstützen soweit notwendig und möglich den Lösungsprozess. Sollte das nicht klappen, stellt sich die Frage: Muss die Verantwortung neu verteilt werden, um das Problem zu lösen, und wenn ja, wie? Sie merken schon, es geht nicht darum einen Schuldigen zu finden. Es geht darum, Verantwortung so zu (ver-)teilen, dass die gewünschten Ergebnisse erzielt oder die anstehenden Probleme gelöst werden können. Die Suche nach Schuldigen führt in der Regel genau zum Gegenteil von Verantwortungsbereitschaft und Lösungsorientierung. Bevor Sie nach Schuldigen suchen, denken Sie an die 50 Euro und Ihren Teil der Verantwortung.

Gerne sende ich Ihnen eine kompakte Liste der wichtigsten Methoden zur Förderung von (Mit-)Verantwortung zu. Senden Sie mir dazu eine E-Mail mit dem Stichwort „Verantwortung" an:



Wird eine Aufgabe delegiert, bleibt das Original der Verantwortung bei uns, und wir geben eine Kopie der Verantwortung (= MIT-Verantwortung) mit der Aufgabe weiter. (Abb. 1)

#JAVAPRO #Agile

Wir entscheiden zusammen - nicht allein

Agile forciert selbstorganisierte, autonome Teams. Diese Methode steht im Widerspruch zu klassischen Technologieentscheidungen: Entscheidungen von oben oder außen schränken Entwicklerteams erheblich ein. Wie lassen sich Technologieentscheidungen ohne Einschränkungen organisieren? Welche typischen Probleme treten auf? Wie erreicht man gemeinsame Standards?

Die vier Leitsätze des agilen Manifests sind weithin bekannt, z.B. „Individuals and interactions over processes and tools“. Darüber hinaus enthält das Manifest noch zwölf Prinzipien1, die oft weniger bekannt sind. Das zwölfte dieser Prinzipien lautet: „The best architectures, requirements, and designs emerge from self-organizing teams”.

Die meisten agilen Teams organisieren ihre Arbeit nach Scrum. Auch der Scrum-Guide2 hebt den Aspekt Selbstorganisation für das Entwicklungsteam hervor: „They are self-organizing. No one, not even the scrum master, tells the development team how to turn product backlog into increments of potentially releasable functionality“.

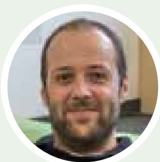
Die Autoren des Scrum-Guides legen so viel Wert auf die Selbstorganisation und Autonomie des Teams, damit es seiner wichtigsten Aufgabe nachgehen kann: der kontinuierlichen Lieferung von lauffähigen Softwareproduktinkrementen. Jede externe Abhängigkeit führt in der Regel zu Behinderungen und Verzögerungen. Warum ist das so? Im Gegensatz zu den Entwicklern des Scrum-Teams fehlen dem externen Entscheider der Kontext und die für die Entscheidung notwendigen Informationen (Abb. 1). Das Team muss den Entscheider also erst mit den passenden Informationen versorgen, damit dieser eine Entscheidung treffen kann. Es lässt sich leicht ersehen, dass dies ein sehr aufwändiger Prozess zur Entscheidungsfindung ist – vor allem weil das Entwicklungsteam selbst alle notwendigen Informationen bereits hat. Fällt der Entscheider aus (z.B. wegen Krankheit) oder ist überlastet, führt das automatisch zur Blockade des Entwicklungsteams. In Zeiten von robustem bzw. resilientem Service-Design erscheint dieser Prozess sehr anachronistisch.

Der Scrum-Guide hebt weiter unten hervor, dass das Entwicklungsteam gemeinsam die Verantwortung für das Produkt und den Code trägt. Es kann die Verantwortung aber nur übernehmen, wenn es auch die maßgeblichen Entscheidungen trifft. Werner Vogels, CTO von Amazon, hat diese Verantwortung treffend in dem Satz zusammengefasst: „You build it, you run it“3. Das Entwicklungsteam wird die Verantwortung von sich weisen, wenn es die Technologieentscheidungen anderer ausbaden muss.

Häufigste Probleme bei Technologieentscheidungen

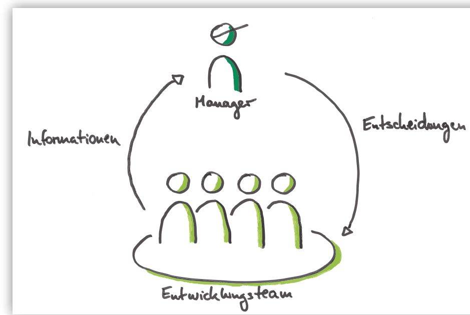
Es gibt also gute Gründe, warum Entwicklungsteams selbst für ihre Technologieentscheidungen verantwortlich sein sollten. Immer mehr Organisationen geben aus diesem Grund zentrale Entscheider auf und übergeben die Verantwortung an die Entwicklungsteams. Wenn es vorher jemanden gab, der für die Technologieentscheidungen verantwortlich war, entsteht zunächst einmal ein Vakuum. Nicht selten versuchen einzelne Entwickler, dieses Vakuum zu füllen, indem sie auf eigene Faust Technologieentscheidungen treffen, ohne mit dem Team Rücksprache zu halten („Einsamer Wolf“). Dabei handeln die einsamen Wölfe noch nicht einmal unbedingt böswillig. Bei ihnen ist angekommen, dass die Entscheidungen jetzt im Team stattfinden, obwohl gemeint war, dass das Team jetzt gemeinsam die Entscheidungen trifft. Ein anderer Grund für das Verhalten des einsamen Wolfes kann sein, dass er die Entscheidung als zu profan betrachtet, als dass sie

Autor:

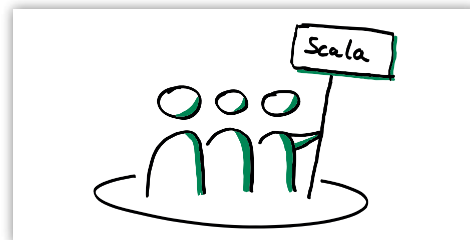


Konstantin Diener ist CTO bei cosee. cosee entwickelt digitale Produkte für und mit Kunden und setzt dabei auf selbstorganisierte, autonome Teams (cosee.biz, @coseeaner). In seiner Rolle als CTO ist Konstantin schon seit einigen Jahren nicht mehr in konkrete Technologieentscheidungen eingebunden, sondern hilft den Teams dabei, sich an die selbstdefinierten Regeln zu halten. Er spricht gerne und oft auf Konferenzen und schreibt Fachartikel. @onkelkodi

mit den anderen gemeinsam getroffen werden müsste. Mit dem Muster des einsamen Wolfes geht meist außerdem einher, dass jeder Entwickler beginnt, seine Lieblingstechnologien einzusetzen („Bring your own pet technology"). Alle Sprachen, Frameworks und Werkzeuge, mit denen man gerade zu Hause in seinen Bastelprojekten experimentiert oder über die man einen interessanten Blog-Post gelesen hat, werden unreflektiert ins Unternehmen getragen. Dadurch entsteht mit der Zeit ein unüberschaubarer und schwer wartbarer Technologiezoo. Dieser Zoo wird zusammen mit Murphys-Law richtig gefährlich: Im Produktivsystem tritt ein Fehler auf – und just in diesem Moment ist der Entwickler, der die Technologie eingeführt hat, im Urlaub, krank oder hat das Unternehmen verlassen. Die übrigen Entwickler sprechen die altbekannten Worte: „Damit kennen wir uns nicht aus!" Das entspricht vermutlich sogar der Wahrheit und ist kein purer Abwehrreflex.



Kreislauf für Technologieentscheidungen durch einen externen Entscheider. (Abb. 1)



Jede Programmiersprache muss mindestens drei Supporter haben. (Abb. 2)

Wenn Entwicklungsteams Technologieentscheidungen in die eigene Hand nehmen, kommt es aber oft nicht nur zu Alleingängen. Es passiert immer wieder, dass die Technologieoptionen in großen Gruppen diskutiert werden. Diese Diskussionen, bei denen jeder in der Firma eine Meinung hat, dauern meist sehr lange und sind nicht selten derart verfahren, dass es zu keiner sinnvollen Lösung kommt. Sie sind nicht immer sachlich und oft wird dort sehr emotional diskutiert. Nicht selten sehnen sich die Teammitglieder deshalb bald wieder nach jemandem, der bei Technologieentscheidungen die Richtung vorgibt und so entstehen implizite Technologieentscheider (Schatten-CTOs). Diese Personen haben offiziell keine andere Stellung als die übrigen Entwickler, möchten aber bei allen Entscheidungen eingebunden werden, getreu dem Motto: „Hättet ihr mich mal vorher gefragt. Jetzt seht selbst, wie ihr mit dem Problem fertig werdet!" Dieses Phänomen ist noch gefährlicher als offizielle Entscheider, die außerhalb des Teams stehen. Zusätzlich zu den Problemen, die offizielle Entscheider verursachen, kommt hier noch dazu, dass die Teams sich oft gar nicht bewusst sind, dass es diese inoffiziellen Entscheider gibt. Dabei ist es durchaus wichtig, dass das Entwicklungsteam Rat von externen Experten einholt, wenn es seine Technologieentscheidungen fundiert treffen möchte. Sich bei außenstehenden Rat einzuholen, wird aber immer wieder als Einmischung betrachtet, weshalb die Teams lieber gar nicht erst fragen. Wenn die Experten nicht gehört werden und das Team nicht auf das in der Firma vorhandene Wissen zugreift, werden wichtige Aspekte nicht berücksichtigt, es kommt zu suboptimalen Entscheidungen und schlussendlich lernt die Organisation nichts hinzu.

Mittelweg zwischen Stillstand und Chaos

Technologieentscheidungen in selbstorganisierten Teams brauchen also Regeln, damit sie sinnvoll funktionieren. Hinsichtlich des Umfangs und der Weiterentwicklung des Technologieportfolios gibt es zwei Extreme. Das eine Extrem ist der bereits beschriebene Technologiezoo. Es existiert ein großer Umfang an Technologien, die in der Regel nur maximal eine Person ausreichend gut beherrscht, um Probleme im Produktivsystem zu beheben. In diesem Fall sind weite Teile der Software schnell nicht mehr wartbar und jeder Urlaub oder Krankheitsfall ist ein operatives Risiko. Außerdem können die Entwickler nicht voneinander lernen und sich gegenseitig helfen. Das andere Extrem sind rigide Unternehmensstandards. Alle Software wird mit einer oder sehr wenigen Programmiersprachen, Frameworks und Werkzeugen entwickelt, die über einen langen Zeitraum stabil sind. In diesem Fall verpassen die Entwickler in der Regel den Anschluss an den technologischen Fortschritt und die Organisation ist irgendwann mit dem Problem konfrontiert, dass die verwendeten Technologien aus dem Support laufen und es immer schwieriger wird Nachwuchs zu finden, der mit diesen Technologien entwickeln möchte. Es braucht ein auf Selbstorganisation basierendes Verfahren zur Technologieauswahl, das einen gesunden Mittelweg zwischen den beiden beschriebenen Extremen darstellt. Die Drei-Personen-Regel ist ein solches Verfahren. Sie lautet:

„Es darf jede neue Programmiersprache ins Unternehmen eingeführt werden, solange es mindestens drei Personen gibt, die sowohl in der Lage, als auch bereit sind, Produktionsfehler in dieser Sprache zu beheben."

Damit ist die Firma neuen Technologieansätzen gegenüber grundsätzlich aufgeschlossen. Auf der anderen Seite ist sichergestellt, dass immer eine ausreichende Anzahl von Personen über entsprechende Kenntnisse verfügt. Möchte ein Entwickler eine neue Programmiersprache einführen, muss er mindestens zwei Mitstreiter überzeugen, die bereit sind diese Sprache in einem ausreichenden Maße zu lernen (Abb. 2). Verlässt einer der drei das Unternehmen, haben die anderen beiden die Aufgabe, mindestens einen neuen Mitstreiter zu finden.

Größere Unternehmen können auch eine höhere Mindestanforderung (n-Personen-Regel) definieren, sollten aber drauf achten, dass dadurch nicht wieder eine Barriere für die Einführung neuer Technologien entsteht.

Ein Leitfaden hilft bei der Technologieauswahl

Die Drei-Personen-Regel hilft vor allem bei größeren Technologieentscheidungen – z.B. der Auswahl einer neuen Programmiersprache oder einer neuen Datenbank-, z.B. einer NoSQL-Technologie. Sie behebt aber nicht alle der bereits beschriebenen Probleme. Um die Entwicklungsteams bei der Technologieauswahl zu unterstützen, ist es sinnvoll einen Leitfaden zu definieren, der aus den folgenden Punkten besteht:

- Was entscheiden wir? Oft ist den Teammitgliedern gar nicht klar, dass sie sich in einem Meeting befinden, in dem eine Technologieentscheidung getroffen werden soll. Dementsprechend sind sie vermutlich auch nicht vorbereitet. Der erste Punkt des Leitfadens macht explizit klar, dass eine solche Entscheidung ansteht. Die Mitglieder können sich sinnvoll vorbereiten. Hat das Team z.B. in seinem Teamvertrag festgelegt, dass zu solchen Entscheidungen immer alle Mitglieder anwesend sein sollen, muss ein entsprechender Termin gefunden werden.
- Nach welchen Kriterien entscheiden wir? Damit in der Diskussion nicht ständig neue Anforderungen entstehen, legt das Team vorab die Anforderungen bzw. Kriterien für die neue Technologie fest. Hierzu gehören z.B. auch Betriebs- und Sicherheitsaspekte oder der Support durch die Community sowie der grobe Einarbeitungsaufwand für das Team.
- Was sind die Optionen? An dieser Stelle darf jeder der Entwickler seine Lieblingstechnologie vorschlagen.
- Wer gehört zum Team? In diesem Schritt geht es um die entscheidende Frage, wer sozusagen stimmberechtigt ist, damit nicht wieder die gesamte Firma an der Technologieentscheidung teilnimmt. Aus dem Prinzip „You build it, you run it" folgt, dass die Entscheidung von den Entwicklern getroffen wird, die nachts aufstehen müssen wenn es zu Problemen kommt.
- Wer sind maßgebliche Experten? Im Rahmen der Technologieentscheidung überlegt sich das Team auch, bei welchen Personen im Unternehmen sie sich eine Expertenmeinung oder zumindest einen Rat einholen können. Die Schatten-CTOs dürfen gerne beratend zur Seite stehen, sind allerdings nicht stimmberechtigt.

Der Leitfaden ist zum einen eine Hilfestellung während der Entscheidungsfindung. Zum anderen handelt es sich dabei auch um einen guten Ausgangspunkt zur Dokumentation der wichtigsten Architekturentscheidungen eines Projekts oder Produkts (z.B. basierend auf dem arc42-Template).

Team-übergreifende Entscheidungen in Communities-of-Practice

In der Regel besteht eine Firma aus mehr als einem selbstorganisierten Team. Das beschriebene Verfahren hält diese Teams nicht davon ab, im Extremfall jeweils eine andere Programmiersprache, eine eigene Datenbanktechnologie usw. einzusetzen. Damit haben die Teams die maximale Autonomie und ein Minimum an Abhängigkeiten erreicht. Aber hat es nicht Sinn, für die Teams gemeinsame Standards zu definieren, um Synergien zu heben, wie es so schön heißt?

Tatsächlich überwiegen die tatsächlichen Synergien in den seltensten Fällen die Vorteile, die autonome Teams mit sich bringen. Andererseits gibt es immer wieder Situationen, in denen teamübergreifende Standards sinnvoll sein können. Würde ein zentraler externer Entscheider zum einen festlegen, welche Themen standardisiert werden und wie die Standards aussehen, stellte das wieder einen Eingriff in die Selbstorganisation und Autonomie der Teams dar. Damit tauchen die zu Beginn des Artikels beschriebenen Probleme wieder auf. Der selbstorgansierte Weg über Standards zu entscheiden sind teamübergreifende Arbeitsgruppen. Jedes Team entsendet einen oder mehrere Vertreter in diese Arbeitsgruppe. Dort entscheiden die Teilnehmer, welche Themen standardisiert werden sollen und wie entsprechende Standards aussehen können. Es bietet sich an, diese Arbeitsgruppen als Communities-of-Practice zu organisieren. Diese Communities lösen noch weitere Probleme, die in der Regel auftreten, wenn die Teams hochgradig autonom arbeiten. In den Communities arbeiten die Softwareentwickler mit Kollegen zusammen, die in einem ähnlichen Themengebiet unterwegs sind wie sie selbst. Dadurch wird das teamübergreifende Lernen unterstützt und die Entwickler können sich gegenseitig bei Alltagsproblemen unterstützen. Außerdem kennen die Entwickler dann ganz automatisch entsprechende Experten, deren Meinung sie für Technologieentscheidungen im Team einholen können.

Fazit:

Entwicklungsteams sollten selbstorganisiert und autonom über die von ihnen verwendeten Programmiersprachen, Frameworks und Werkzeuge entscheiden können. Damit diese Entscheidung sinnvoll funktionieren, ist es wichtig, einige wenige Grundregeln wie die Drei-Personen-Regel sowie einen Leitfaden aufzustellen. Damit sind die Teams gut gerüstet. Entscheidungen, die über ein Team hinausgehen, sollten nicht von zentralen Entscheidern, sondern in Communities-of-Practice getroffen werden. Die Arbeit in diesen Communities dient außerdem der Wissensverteilung im Unternehmen. Mit den in diesem Artikel beschriebenen Konzepten sind Technologieentscheidungen vollständig ohne zentrale Entscheider möglich. Damit haben die Entwicklungsteams so wenig externe Abhängigkeiten wie möglich und können sich vornehmlich auf ihre wichtigste Aufgabe konzentrieren – funktionierende Produktinkremente auszuliefern.

Quellen:

1 http://agilemanifesto.org/principles.html
2 https://www.scrumguides.org/scrum-guide.html#team-dev
3 http://queue.acm.org/detail.cfm?id=1142065

#JAVAPRO #Agile #DevOps #Security

Technische Schulden in der DevSecOps-Welt

Wenn von technischen Schulden die Rede ist, denken viele vor allem an die traditionelle Applikationsentwicklung. Schuldzuweisungen aber helfen nicht weiter, denn auch die DevSecOps-Welt ist davon nicht verschont. Wichtig ist, technische Schulden genau zu dokumentieren und Prioritäten für deren Behebung festzulegen.

Wie bei fast allen Verfahren und Prozessen in der IT gibt es auch beim Konzept der technischen Schulden Anhänger und Kritiker. Mit dieser Metapher lassen sich auf Schwachpunkte in der Qualität von Programmcode hinweisen und Wege aufzeigen, wie sich diese Mängel beseitigen lassen.

Keine Software ist frei von technischen Schulden

Um eines klarzustellen: In jeder Software gibt es technische Schulden, die sich beispielsweise daraus ergeben, dass die Entwickler Kompromisse zwischen dem idealen Programmcode und dem Programmcode eingehen, der genügt, um Termine einzuhalten. Erkennen die Entwickler oder die Benutzer die Lücken, können sie Verbesserungen angehen – wenn nicht sofort, dann zu einem späteren Zeitpunkt. Bei den technischen Schulden lassen sich zwei Formen unterscheiden:

- Nicht alle geplanten Funktionen wurden auch tatsächlich realisiert.
- Das Projektteam traf suboptimale Entscheidungen.

Wenn die Entwickler wissen was sie versäumt haben, können sie es später, zu einem geeigneten Zeitpunkt nachholen. Wenn sich eine frühere Entscheidung als suboptimal herausstellt, können Entwickler sie revidieren.

Gerade bei DevSecOps ist Sicherheit immer wieder ein vernachlässigtes Thema bei den technischen Schulden. DevSec Ops stellt einem gängigen Verständnis zufolge durch Automatisierung, Tooling und kulturellem Wandel die Sicherheit in den Mittelpunkt der DevOps-Praktiken, -Prozesse und -Bereitstellung. Das heißt, bereits in der Konzeptphase sollten eigentlich die Anwendungs- und Infrastruktursicherheit berücksichtigt werden. Um die Geschwindigkeit im DevOps-Workflow zu erzielen, müssen zumindest einige Sicherheitsfeatures automatisiert werden. Auch durch die Auswahl der geeigneten Tools sollten Entwickler zur Integration von DevOps-Sicherheit beitragen. Dennoch treten auch hier immer wieder technische Schulden auf. Einige Beispiele für sicherheitsrelevante Schulden:

- Aktuell noch nicht öffentlich zugängige APIs werden bei der Authentifizierung vergessen.
- Funktionen werden nicht klar voneinander getrennt.

Autor:

Mike Bursell - Chief Security Architect | Red Hat

Mike Bursell kam im August 2016 zu Red Hat, nachdem er zuvor bei Intel und Citrix in den Bereichen Sicherheit, Virtualisierung und Netzwerk tätig war. Nach einer Ausbildung zum Softwareentwickler spezialisierte er sich auf verteilte Systeme und Sicherheit und arbeitete in den letzten Jahren in den Bereichen Architektur und technische Strategie.

Mike ist seit Mitte 2013 eng mit dem Telekommunikationsmarkt und dem ETSI NFV verbunden, mit Beiträgen in den Gruppen I SEC, INF und SWA, und ist Berichterstatter für drei sicherheitsrelevante Arbeitsthemen. Er war zwei Jahre lang stellvertretender Vorsitzender der ETSI NFV Security Working Group und war auch am OPNFV-Projekt beteiligt. Er spricht regelmäßig auf Veranstaltungen in Europa, Nordamerika und APAC.

Zu seinen beruflichen Interessen gehören: Linux, Open Source Software, Sicherheit, verteilte Systeme, Blockchain, NFV, SDN, Virtualisierung (einschließlich Linux-Container und Hypervisor).
Mike hat einen MA von der University of Cambridge und einen MBA von der Open University.

- Rollen werden den ursprünglichen Erwartungen zufolge fix codiert, mögliche künftige Änderungen werden nicht berücksichtigt.
- Kryptographische Verfahren der Cipher-Suite werden zusammen mit der Applikation kompiliert.

All diese Arten von technischen Schulden können sowohl in Applikationen auftreten, die mit klassischen Entwicklungsmethoden entstanden sind, als auch in solchen, die mit eher agilen Methoden wie DevSecOps erstellt wurden. Mehr noch: In vielen Fällen befinden sich Projekte, die der klassischen Methode folgen, in einer besseren Position als agile oder DevOps-Projekte. Denn in der Regel gibt es eine klar definierte Produktmanagement-Rolle, die Kundenanforderungen zwischen den Releases sammelt, bewertet und entscheidet, welche für das nächste Release priorisiert werden. In der DevOps-Welt kann es schnell vorkommen, dass diffizile Funktionen oder schwierige Entscheidungen aufgeschoben werden. Wenn die Verantwortlichen Requests nicht konsolidieren, verdeckt der Fokus auf Pro-Sprint-Funktionen die technischen Schulden. Diese Probleme verschlimmern sich, wenn Teams nach dem Motto verfahren: Bei DevSecOps sind alle für die Sicherheit verantwortlich - und daher keine Sicherheitsexperten im Team vorhanden sind.

Sicherheitsanforderungen werden oft vernachlässigt

Ebenso wie bei der traditionellen Applikationsentwicklung ist auch bei DevOps die Sicherheit eine komplexe Herausforderung, die detailliertes Wissen erfordert. Technische Schulden können leicht entstehen, wenn Mitarbeiter, die keine Sicherheitsexperten sind, Entscheidungen treffen. Wie soll jemand mit wenig Know-how über Best-Practices im Bereich der IT-Sicherheit kennen, z.B. wann es sinnvoll ist Cipher-Suites zusammen mit einer Anwendung zu kompilieren und wann nicht? Technische Schulden werden aus drei Gründen zum Sicherheitsproblem:

- Sicherheitsanforderungen haben oft keinen direkten funktionalen Bezug zu einer Anwendung und werden daher seltener weiterverfolgt;
- Fundierte Sicherheitskenntnisse sind Mangelware. Zu wenige Verantwortliche, Architekten und Designer verstehen die Bedeutung und die Auswirkungen von IT-Sicherheit.
- Entwicklerteams befassen sich erst gegen Projektende mit IT-Sicherheit, das Thema wandert sehr schnell in der To-Do-Liste nach unten und wird dann vergessen.

Es lohnt sich darüber nachzudenken, warum technische Schulden ein Gewinn für ein Projekt und insbesondere für Open-Source-Projekte sein können. Erkennen Unternehmen technische Schulden in einer Applikation, müssen Verantwortliche eine Entscheidung treffen: Sollen die Schulden behoben werden oder bleibt alles zunächst einmal wie es ist. Manchmal, und so unpopulär diese Ansicht auch sein mag, hat die Benutzerfreundlichkeit für eine bestimmte Zeit Vorrang vor der Sicherheit. Möglicherweise verfügt das Entwicklerteam auch nicht über das benötigte Fachwissen, um aktuell eine fundierte Entscheidung über Design oder Implementierung zu treffen. Auf jeden Fall muss diese Entscheidung dokumentiert werden und dazu gehört auch, warum sie so getroffen wurde. Existiert eine Dokumentation und es handelt sich um ein Community-Open-Source-Projekt, kann es außerdem sein, dass jemand anderes, der über die Ressourcen oder das Wissen über mögliche Implementierungen verfügt, die technischen Schulden beheben wird.

An einem konkreten Beispiel lässt sich am besten zeigen, wie Entwickler mit technischen Schulden umgehen sollten. Angenommen, an einer bestimmten Schnittstelle war keine Verschlüsselung enthalten. Welche Schritte können Entwickler unternehmen, um diese technische Schuld zu beheben? Zunächst einmal sollten sie den Mangel an mehreren Stellen klar und eindeutig aufzeichnen:

1. In der Projektdokumentation, beispielsweise in der Form: „Uns fehlte die Zeit und daher enthält die Anwendung kein Zertifikatsmanagement."
2. In der Produktdokumentation durch den Hinweis: „Diese API ist für den Einsatz in einer geschützten Umgebung konzipiert und sollte nicht über das Internet genutzt werden."
3. im Quellcode durch Kommentare (**Listing 1**).
4. In Unit-Tests, die feststellen, ob über die Schnittstelle Daten in Klartext übertragen werden.

Listing (1)

```
// 2018-05-02 (mmueller@company.com) Planned to use TLS 1.2
(check
// for newer version) probably won't need client
// authentication. Don't use ECC cipher suites.
// Certificate management and revocation currently not specced.
```

Inwiefern sind diese Maßnahmen hilfreich? In der Projektdokumentation können Entwickler klarere Anforderungen erstellen, in der Produktdokumentation legen sie fest, wie die Anwendung sicher eingesetzt werden kann und in der Quellcodedokumentation folgen Anweisungen für die künftige Implementierung. Der letzte Punkt, der Komponententest, mag seltsam erscheinen, denn wer will schon einen Test durchführen, von dem bekannt ist, dass er fehlschlägt? Wollen Entwickler sicherstellen, dass ein Feature oder eine Funktion implementiert ist, gibt es nichts Besseres als eine rote Ampel, um explizit zu überprüfen, ob alles so ist wie es sein soll. Mit dieser Methode sind nützliche technische Schulden dokumentiert und damit bekannt.

Wichtig ist, dass alle Beteiligten Schuldzuweisungen vermeiden – insbesondere dann, wenn Entwickler selbst die Betroffenen sind. Von einer guten Dokumentation profitieren mehrere Gruppen: Softwarearchitekten, Designer, Entwickler, Tester, Projektverantwortliche, Produktmanager, Vertrieb, Marketing, Support und nicht zuletzt Kunden.

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

JCON 2019 PARTNER

GOLD PARTNER

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

SILBER PARTNER

RAPIDclipse · consol Wir unternehmen IT · trivago

BRONZE PARTNER

viadee IT-Unternehmensberatung · TK Die Techniker · GEBIT Solutions

ORGANISATIONS-PARTNER

JAVAPRO · X 2019 DEVCON · XDEV