Ausgabe 05/2022

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 05

Java aktuell

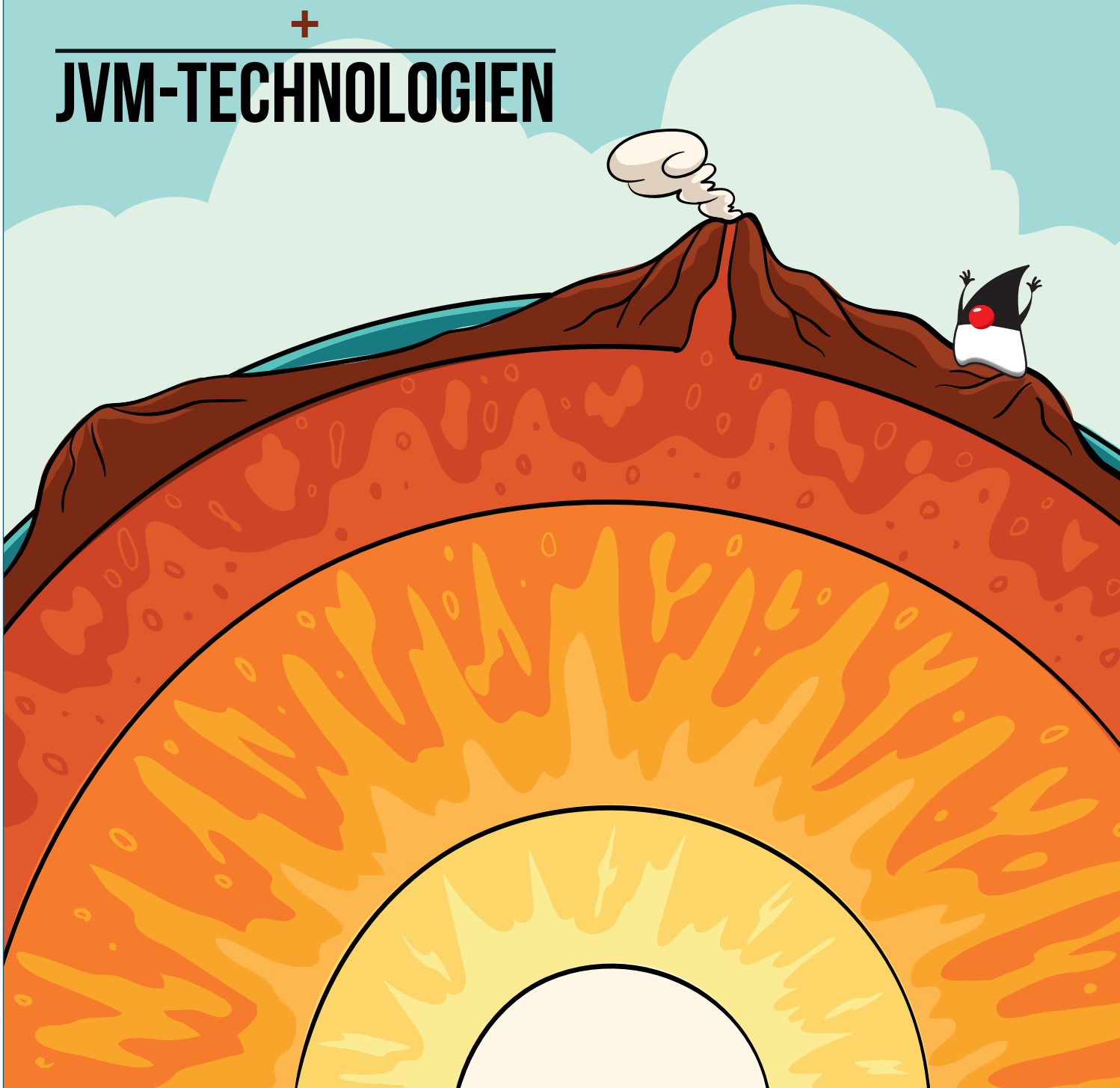
iJUG Verbund
www.ijug.eu

Java
Performance, Diagnostik und Tools

GraphQL
Subscriptions

Testing
mithilfe von Teststrategien und BDD

CORE JAVA
+
JVM-TECHNOLOGIEN



JavaLand!

on demand

Javaland 2022 verpasst?

Jetzt On-Demand-Ticket buchen und Vortragsaufzeichnungen anschauen!

Alle Angebote im On-Demand-Ticket-Shop

Community-Partner: iJUG Verbund

Präsentiert von: DOAG Heise Medien

Liebe Leserinnen und Leser,

in dieser Ausgabe begeben wir uns zurück zu den Grundmauern des Java-Ökosystems: Java selbst.

Den Anfang macht Benjamin Schmid, der anstelle neuer Sprachfeatures, die Aspekte Performance, Diagnostik und Tools bei neueren Java-Versionen unter die Lupe nimmt. Im Anschluss zeigen uns Heinz-Werner Haas und Francois Fernandes, was GraphQL Subscriptions sind und wie man diese mithilfe von Spring Boot, Project Reactor und React realisieren kann.

Wie eine Teststrategie dabei unterstützen kann, zu einem wahren Testing-Superhero zu werden, zeigen uns Christian Schwörer und Stefan Ludwig in ihrem Artikel ab Seite 22 anhand einer Microservices-Anwendung mit Quarkus. Bernd Stübinger präsentiert uns in seinem Artikel, wie man seinen eigenen verteilten Cache implementieren kann. Dazu beleuchtet er verschiedene Ansätze, sogenannte Cache-Stampedes zu vermeiden. eMobility ist heutzutage Gang und Gebe, die Anzahl verfügbarer Ladesäulen für e-Autos steigt stetig. Doch wie kommuniziert eine solche Ladesäule mit dem nachgelagerten Backend? Dieses Szenario schauen sich Uwe Sauerbrei und Valerio Onofre Vilaca genauer an und erläutern, wie diese Tests mithilfe von Behaviour Driven Development automatisiert werden können.

Wir wünschen euch viel Spaß beim Lesen!
Eure

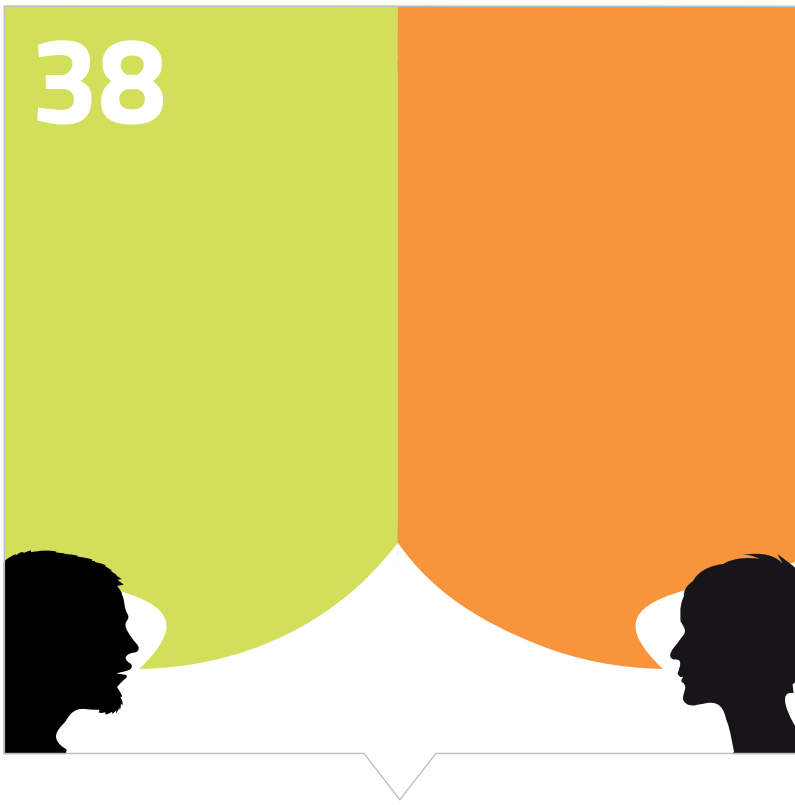
Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Performance, Diagnostik und Tools im Fokus neuer Java-Versionen

Subscriptions in GraphQL

Testen wie ein Superhero mithilfe von Teststrategien



Automatisierte Tests mit Behavior Driven Development

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

20. Mai 2022

Projekt Leyden soll Optimierungen behutsam einführen

Nach dem Diskussionsaufruf vor mehr als zwei Jahren soll es jetzt endlich mit dem OpenJDK-Projekt „Leyden" losgehen. Es soll die Start-up-Zeiten und den Speicherbedarf vieler Java-Applikationen drastisch reduzieren. So wie beispielsweise die GraalVM das bereits mit „native images" macht, aber als Teil der Plattform-Spezifikation. Am Ende der langen Diskussion steht nun eine behutsamere Vorgehensweise. Mark Reinhold, Chef-Architekt der Java-Plattform, schreibt, dass der ursprüngliche Vorschlag von „static images" eine „closed world" erfordert, was viele Beschränkungen insbesondere in Bezug auf Class Loading und Reflection mit sich bringen würde, die den Anwendungsbereich deutlich einschränken. Nun sollen lieber ein schrittweiser Ansatz verfolgt und das Kosten-Nutzen-Verhältnis verschiedenster Beschränkungen und der damit möglichen Optimierungen untersucht werden – während gleichzeitig versucht wird, die Anwendbarkeit auf möglichst viel existierenden Code zu erreichen. Das Fernziel bleibt jedoch weiterhin, vollständig statische Images zu generieren.

Spring Boot 2.7

Spring Boot 2.7 ist da und bietet jetzt einen Starter für GraphQL – das zugrunde liegende Projekt „Spring for GraphQL" wurde gleichzeitig als 1.0 „generally available" angekündigt. Weitere Neuheiten sind unter anderem die Unterstützung von Podman für das Bauen von Images mit „Cloud Native Buildpacks" und die Test-Autokonfiguration für weitere Datenbanken (@DataCouchbaseTest und @DataCassandraTest).

23. Mai 2022

JSR-381 – Maschinelles Lernen mit Java

Maschinelles Lernen ist bislang hauptsächlich eine Domäne für Python. Ein Weg, wie Java sich gegen die Würgeschlange behaupten kann, könnte sein starkes Ökosystem gepaart mit der Fähigkeit sein, „haltbare" Standards zu definieren. JSR-381 „Visual Recognition" (VisRec) ist ein Teil davon. Bereits 2017 gestartet, ist er vor einigen Monaten endlich offiziell angenommen worden – und zielt auf alle Plattformen ab, von „embedded" bis zum Server. Die beiden Spec Leads des JSR, Frank Greco und Zoran Sevarac, sind momentan „auf Tour", um für den Standard zu werben.

25. Mai 2022

Spring Boot 3.0

Nach dem Release ist vor dem Release. Direkt nach der Freigabe von Version 2.7 geht es mit vollem Schwung auf das nächste „Major Release" zu: Spring Boot 3.0, geplant für November 2022 auf Basis des Spring Frameworks 6.0. Die minimal unterstützte Java-Version soll dann das neueste LTS Release (17) sein. Außerdem soll der Umstieg von den verwendeten Java-EE-8-APIs auf Jakarta EE 9 erfolgen. Ein rein technisches Update, aber auch einiges an Arbeit und eine wichtige Voraussetzung für zukünftige Kompatibilität. Neben den fälligen Upgrades der zugrunde liegenden Spring-Framework-Spezifikationen und weiterer externer Abhängigkeiten gibt es ein paar weitere kleine Neuerungen in Boot. Zum Beispiel kann das Prometheus Push Gateway (in der Regel für kurzlebige Applikationen/Jobs genutzt) so konfiguriert werden, dass es einen PUT der Metriken beim Shutdown ausführt (ich gebe zu, das Feature hat mich jetzt nur interessiert, weil ich vor Kurzem über etwas Ähnliches gestolpert bin, aber vielleicht bin ich ja nicht allein). Außerdem wird der Support für den (in Java 17 als „depecrated" markierten") SecurityManager eingestellt, ebenso der für den CommonsMultipartResolver (im Spring Framework 6 nicht mehr enthalten).

29. Mai 2022

Adoptium-Marktplatz für Java SE eröffnet

Der Adoptium Marketplace für TCK-kompatible Java-SE-Varianten verschiedenster Hersteller ist eröffnet. Unter anderem IBM, Red Hat und natürlich Adoptium selbst sind bereits vertreten, weitere wie Azul und Alibaba sollen in Kürze folgen. Der Start war nach eigener Angabe der Eclipse Foundation, zu der das Adoptium-Projekt gehört, ein großer Erfolg. Das Adoptium-Projekt insgesamt ist mittlerweile das besucherstärkste innerhalb der Eclipse Foundation (und das mit dem größten Bandbreiten-Verbrauch).

8. Juni 2022

Jakarta Data

Es gibt ein neues Projekt innerhalb von Jakarta EE: Jakarta Data. Ziel ist es, eine einheitliche Abstraktionsschicht über alle Arten von Persistenz-Technologien zu schaffen, nach eigenen Angaben inspiriert von anderen Projekten wie Spring Data, Apache Delta Spike, Jakarta NoSQL, Quarkus und weiteren.

10. Juni 2022

Kotlin 1.7 mit neuem Compiler

k8s, k6, k2, langsam komme ich durcheinander. k2 ist ein neuer Kotlin-Compiler, der gerade mit Kotlin 1.7 als Alphaversion erschienen ist. Der Name ist keine Abkürzung für „kotlin kompiler" oder Ähnliches, sondern ist laut JetBrains einfach nach dem zweithöchsten Berg der Welt benannt; das lässt dann ja auch noch Raum für Verbesserungen. k2 soll deutliche Verbesserungen in der Kompiliergeschwindigkeit bringen (mehr als Faktor 2) und das Einführen neuer Sprach-Features erleichtern.

Perspektivisch sollen alle „Kotlin-Plattformen" unterstützt werden (auch JS und native), in dieser Version beschränkt sich das jedoch

zunächst auf die JVM. Neben k2 gibt es unter anderem Verbesserungen der inkrementellen Kompilierung mit Gradle und einen Underscore-Operator als Argument für generische Typen. Außerdem sind einige Beta-Features jetzt als stabil gelistet: „definitely non-nullable" generische Typen („T & Any"); das Opt-in-Feature, das API-Entwicklern und -Nutzern hilft, zum Beispiel noch experimentelle Teile eines API nur über eine explizite Freischaltung zu nutzen; und die Builder-Type-Inferenz, die die Argument-Typen generischer Builder durch die Analyse innerer Aufrufe ermitteln kann. Diese Art der Inferenz wird nun bei Bedarf auch automatisch aktiviert.

Java 19

Java 19 geht in „Rampdown Phase 1" (keine Änderungen mehr an der Feature-Liste) und sollte damit auf einem guten Weg für das geplante Release im September sein. Zu den im letzten Tagebuch schon erwähnten Features ist noch eine Inkubator-Version von „Structured Concurrency" hinzugekommen. Diese soll Multithread-Programmierung vereinfachen, indem mehrere Tasks, die in verschiedenen Threads laufen, zu einer Einheit („unit of work") zusammengefasst werden können – mit übergreifender Fehler- und Abbruch-Behandlung etc. Das Inkubator-Tag *[1]* bedeutet allerdings, dass erstmal insbesondere Feedback aus der Community eingeholt werden soll. Das Feature ist noch deutlich entfernt von einem kompletten, stabilen API.

Die Gesamtliste der Java 19 Features steht wie immer hier *[2]*. Wer die 19 durch die 20 ersetzt, sieht, dass sich für das Release im nächsten Frühjahr noch nicht viel getan hat. Details zu 20 dann wohl im nächsten Tagebuch…

15. Juni 2022

MicroProfile: Regeln zum Umgang mit Java-/Jakarta-Versionen

Das MicroProfile-Projekt hat sich Regeln zum Umgang der Einzelspezifikationen mit neuen Jakarta-EE- und Java-SE-Versionen gegeben. Das generelle Motto lautet „compile low, run high". Grundsätzlich entscheidet jede Einzelspezifikation selbst über ihre Abhängigkeiten. Sie soll aber nur dann eine höhere Java-SE-Version als (minimal) erforderlich definieren, wenn sie tatsächlich neue Sprach-Features daraus in ihrem API nutzt. Eine Erhöhung der Version, etwa im MicroProfile-„Umbrella", wirkt sich nicht auf die Einzelspezifikationen aus (ist ja auch logisch).

Für Jakarta EE können ebenfalls mehrere Versionen gleichzeitig unterstützt werden. Umgekehrt gilt: Eine neue Version einer MicroProfile-Einzelspezifikation für eine höhere Jakarta-Version soll es nur dann geben, wenn die MP-Spezifikation tatsächlich angepasst werden muss.

23. Juni 2022

Quarkus Update

Quarkus bewegt sich zu schnell für ein nur alle zwei Monate veröffentlichtes Tagebuch. Am Anfang dieser zwei Monate stand Quarkus 2.9, jetzt ist gerade schon 2.10 freigegeben worden, mit Maintenance-Releases dazwischen. Was gibt es Neues in den beiden Versionen? 2.9 führt zum Beispiel Unterstützung für WebAuthn ein (die „Web Authentication API Specification" mit dem Ziel, Passwörter oder generell „shared secrets" zu ersetzen); allerdings zunächst als Preview, mit möglichen inkompatiblen Änderungen in Folge-Releases (in 2.10 hat sich daran noch nichts geändert). Ebenfalls aus dem „Auth*"-Bereich kommen Voreinstellungen für bekannte OpenID Connect Provider dazu – die Liste enthält bereits sieben Provider und soll wachsen. Daneben gibt es beispielsweise Komprimierung für Reactive Routes und RESTEasy Reactive, Unterstützung der Confluent Schema Registry (Kafka) und diverse Updates der Sprach-Versionen (Scala, Kotlin).

Bei 2.10 ist sicherlich die größte Nachricht, dass eine erste – experimentelle – Unterstützung für die virtuellen Threads von Project Loom enthalten ist. Loom kommt ja ebenfalls erst als Preview mit Java 19 im September. Außerdem gibt es zum Beispiel Unterstützung für „GraphQL non-blocking" und Kubernetes Service Bindings für Reactive SQL Clients.

25. Juni 2022

JavaOne wird wiederbelebt

Einige Oracle-Mitarbeiter wie Sharat Chander (Director im Java Produktmanagement) arbeiten hinter den Kulissen eifrig an einer Neuauflage der JavaOne, im Rahmen der Oracle CloudWorld in Las Vegas Mitte Oktober. Ich schreibe „hinter den Kulissen", weil der Informationsfluss zu den angesprochenen Java Champions und JUG Leaders teilweise als unzureichend empfunden wurde. Aber es ist wohl auch nicht so einfach, das Projekt bei Oracle umzusetzen – der Fokus liegt dort sicher woanders, und für besonders offene Kommunikation stand der Konzern noch nie.

Inzwischen nimmt das Projekt jedoch deutliche Formen an, die ersten beiden Tage (16./17.) sollen laut Sharat für ein Java Leaders Summit und diverse soziale beziehungsweise Sport-Events genutzt werden, die eigentliche Konferenz soll dann vom 18. bis 20. Oktober stattfinden. Für die erste JavaOne seit 2017 (danach gab es noch die CodeOne für zwei Jahre) wird es keinen offiziellen Call for Papers geben (das würde auch langsam knapp), es soll stattdessen das große Netzwerk der Java-Community genutzt werden. Mal sehen, wie ausgewogen das am Ende wird, aber es soll für zukünftige JavaOnes auch wieder anders laufen.

29. Juni 2022

CloudLand

Das CloudLand Festival, eine neue, viertägige Konferenz für alle Themen rund um „Cloud Native", hat diese Woche ihre Premiere. Sie erinnert ein bisschen an das JavaLand, was in erster Linie an der Location liegt, dem Phantasialand. Aber auch an vielen Überschneidungen bei den Köpfen hinter der Konferenz. Gemeinsam ist auch

der Gedanke einer Konferenz aus der Community heraus entstanden. Es haben sich eine ganze Reihe von Cloud Native Meetups und Java User Groups sowie die Cloud Native Community der DOAG zusammengetan, um die Konferenz zu organisieren.

Einige Dinge sind jedoch auch anders. Ganz offensichtlich natürlich die Tatsache, dass der Park im Sommer auch für die Allgemeinheit geöffnet ist und die Konferenz nur im „Quantum"-Komplex stattfindet. Was gut funktioniert, weil sie auch bei weitem (noch?) nicht die Größe der JavaLand hat und weil alle jederzeit in eine Achterbahn ausweichen könnten, wenn ein Vortrag zu voll wird – und umgekehrt. Und das Festival ist weniger zentral organisiert. Das Ziel ist, den Rahmen für einzelne Gruppen zu bieten, damit diese sich mit ihren Inhalten und Formaten austoben können. Das funktioniert teilweise schon gut, aber daraus lässt sich in den nächsten Jahren sicher auch noch mehr machen.

30. Juni 2022

JakartaOne und Jakarta EE 10

Eine der „Konferenzen in der Konferenz" beim CloudLand ist die eintägige JakartaOne. Nicht ganz zufällig – Java ist ein wichtiger Bestandteil der Cloud und es sind wie schon erwähnt auch einige JUGs an der CloudLand beteiligt. Daraus hat es sich dann fast schon natürlich ergeben, die JakartaOne in diesem Rahmen durchzuführen, vor Ort und als Livestream.

Leider hat es nicht ganz gepasst, Jakarta EE 10 hier offiziell freizugeben – der anvisierte „Termin" Juni 2022 ließ sich nicht mehr halten. Es gab noch eine Reihe von offenen Problemen, die erst noch behoben werden müssen, sodass sich das Release noch mal leicht verschiebt. „Quality first – auf ein paar Wochen kommt es jetzt auch nicht mehr an", sagt Jan Westerkamp, der gemeinsam mit Hendrik Ebbers und Falk Sippach die Konferenz moderiert. Ein spannender Teil war ein Interview mit Jürgen Höller, dem Spring Framework Project Lead. Nein, es gab kein verbales Gerangel. Das Verhältnis lässt sich wohl mit „freundliche Konkurrenz und teilweise Symbiose" bezeichnen. Spring nutzt ja viele APIs von Jakarta und hat jetzt entsprechend die nötigen Schritte vollzogen, um mit den nächsten Jakarta-Versionen kompatibel zu sein. Es setzt daher auch auf den Erfolg von Jakarta. Und der scheint sich langsam einzustellen. Sicher nicht ohne Grund hat zum Beispiel Microsoft Ende April offiziell seine Beteiligung am Jakarta-Projekt mitgeteilt.

Referenzen

[1] *https://openjdk.org/jeps/11*
[2] *https://openjdk.org/projects/jdk/19/*

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' eclipse-Corner

Ja, ist denn das zu glauben? In der letzten Ausgabe noch habe ich es ausnahmsweise gewagt, die Jakarta EE Working Group einmal nicht zu schelten, da die Veröffentlichung von Jakarta EE 10 tatsächlich kurz bevorstand, und zwar – total unerwartet – plangemäß noch im zweiten Quartal *[1]* 2022. Und nun ist das zweite Quartal um, die Liste offener Aufgaben auf GitHub *[2]* leer, der Plan weder geändert – noch Jakarta EE 10 veröffentlicht! Ach Mann, so macht das doch keinen Spaß! Es geht also gerade so in der gleichen alten Leier weiter, wie wir es von der Jakarta EE Working Group seit Jahr und Tag gewohnt sind. Das ist extrem schade. Wieder wird gerade mühsam erarbeitetes Vertrauen verspielt, und wieder werden weitere User in Richtung alternativer Technologien abwandern. Einige lassen dabei Java auch gleich ganz hinter sich – selbst wenn alternative Plattformen technisch oftmals deutlich unterlegen sind.

Aus Sicht eines iJUG-Vertreters ist daran wenig Erfreuliches. Wir haben uns im iJUG auf die Fahne geschrieben, die Interessen der deutschsprachigen Java-Programmierer zu vertreten. Dieses Interesse ist sicherlich vielfältig, geht von kleinen Dingen wie Förderung der Zusammenarbeit durch Vermittlung von Speakern und Austausch von Meeting-Terminen bis zur Organisation der JavaLand-Konferenz oder dem Herausbringen der Java aktuell. Der wichtigste Aspekt aber ist der Schutz der Investitionen, die wir Java-Programmierer in die Plattform Java getätigt haben. Jeder von uns hat Lebenszeit, Nerven sowie Vermögen in Java investiert. Diese Investition geht verloren, wenn sich niemand darum kümmert, die Plattform weiterzuentwickeln. Und mit Plattform meine ich nicht nur die Sprache Java, die Java Virtual Machine oder die sonstigen Bestandteile des JDK. Ich meine damit eben auch Java EE – auch wenn es heute Jakarta EE genannt wird. Denn ein nicht zu unterschätzender Anteil der Java-Investitionen ist in Jakarta EE geflossen. In meinem Fall 20 Jahre meiner Lebenszeit. Das wirft man nicht mal kurz weg, oder zumindest nicht ohne zwingenden Grund oder adäquaten Ersatz. Und nein, Spring ist kein adäquater Ersatz (wer nicht versteht warum, hat den Unique Selling Point der Java-EE-Spezifikation nicht verstanden: Herstellerunabhängigkeit).

In Deutschland, Österreich und der Schweiz leben und arbeiten mehrere Zehntausend Java-Programmierer. Ein großer Teil davon nutzt heute noch Jakarta EE und wartet sehnsüchtig auf Release 10, ist schon gespannt auf die neuen Features in Release 11. Es muss doch irgendwie möglich sein, dass von diesen vielen Menschen, zumindest ein Teil regelmäßig etwas Freizeit opfert oder deren Arbeitgeber zumindest einen Teil der Arbeitszeit investieren, um Jakarta EE dauerhaft weiterzuentwickeln. Egal, was die Schwergewichte Oracle, Red Hat und IBM beteuern – die Zukunft von Jakarta EE sieht nur dann rosig aus, wenn die Community das Heft in die Hand nimmt.

Bis es so weit ist, wird in dieser Kolumne sonst kaum eine Erfolgsmeldung stehen.

[1] Zeitplan Jakarta EE 10: *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee10/JakartaEE10#jakarta-ee-10-schedule*

[2] Liste offener Aufgaben für Jakarta EE 10: *https://github.com/orgs/eclipse-ee4j/projects/21*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Neues in Java bei Performance, Diagnostik und Tools

Benjamin Schmid, Nagarro GmbH

Beim Blick auf neue Java-Versionen stehen häufig die neuen Sprachfeatures im Vordergrund. Doch gerade im Bereich von Performance, Diagnostik und Tools verbergen sich viele neue und wertvolle Schätzchen, die ein Upgrade lohnenswert machen. Welche das sind und wie diese zu mehr „Schwuppdizität" (gefühlte Geschwindigkeit), Einblicken und Hilfestellungen verhelfen, erläutert dieser Artikel.

Als Oracle mit Java 9 auf ein halbjähriges Release-Intervall umschwenkte und später auch noch die Lizenzbedingungen veränderte, führte dies dazu, dass viele Unternehmen eisern an Java 8 festhielten. Zahlreiche neue Anbieter sprangen in die Bresche und bieten nun alternative Java-Distributionen an, viele davon kostenfrei und mit Langzeitunterstützung. Dies sollte die geweckten Upgrade-Bedenken zwar wieder entkräften, dennoch ist weiterhin eine Zögerlichkeit beim Upgrade auf neue Java-Versionen, wie etwa auf die aktuelle LTS-Version 17, zu spüren. Dabei lohnt sich ein Upgrade oft selbst ohne jegliches Update oder Anpassungen an der Anwendung selbst.

Altes schneller vergessen: Die neuen Low-Level-Garbage-Collectors

Verantwortlich dafür sind zahlreichen Neuerungen unter der Haube der JVM, wie im Bereich der Garbage Collectors. Neben Optimierungen am Standard G1 sind mit ZGC *[1]* und Shenandoah *[2]* nun gleich zwei Vertreter sogenannter Low-Latency-GCs verfügbar. Sie adressieren moderne Rechnerarchitekturen mit Multi-Core-CPUs und Arbeitsspeicher von 8 Megabytes bis zu mehren Terabytes. Ihr vorrangiges Designziel: verlässliche und konstant niedrige Latenzen durch Garbage-Collector-Pausen im höchstens einstelligen Millisekunden-Bereich. Eine Eigenschaft, die insbesondere für vernetzte Anwendungen interessant ist, in denen die sonst üblichen Aussetzer von bis zu 500 Millisekunden erheblich stören und man für eine gesteigerte „Schwuppdizität" gerne auch Einbußen beim Gesamtdurchsatz oder einen leicht gesteigerten CPU-Ressourcenbedarf in Kauf nimmt.

Der ZGC-Collector (Nutzung: `-XX:+UseZGC`) stammt aus dem Hause Oracle und erinnert mit seinem Namen an das revolutionäre ZFS-Dateisystem, ebenfalls von Oracle. Das ursprüngliche Ziel, die Pausen stets unter 10 ms zu halten und dazu maximal 15 % Durchsatz gegenüber G1 zu verlieren, wurde schnell erreicht. Inzwischen hat sich ZGC eine Millisekunde als neuen Maßstab gesetzt. Der zweite Vertreter Shenandoah wurde dagegen dem OpenJDK von Red Hat beigesteuert und ist in Funktionsweise und Eigenschaften ZGC recht ähnlich. Dafür bietet Shenandoah (Nutzung: `-XX:+UseShenandoahGC`) verschiedene Modi und Heuristik-Profile wie adaptive, static, compact oder aggressive an und kann insbesondere mit Backports für JDK 8 und 11 sowie 32-Bit-Plattformen punkten *[3]*.

Letztendlich besitzen ZGC und Shenandoah starke Ähnlichkeiten und sind zugleich klar ausgeprägte Spezialisten *(siehe Tabelle 1)*. Auch der Allrounder G1 hat viele Verbesserungen erhalten und Performancegewinne von bis zu 15 % durch den Wechsel von Java 11 auf 17 sind möglich. Dabei kann es sich lohnen, probehalber alte Tuning-Parameter zu entfernen, denn inzwischen konfiguriert die JVM mehr Parameter selbst und erkennt beispielsweise die jeweiligen RAM-Limits innerhalb von Containern nun eigenständig.

RAM brüderlich teilen: Einfacheres Class-Data-Sharing

Das sogenannte *Class-Data-Sharing* *[4]* reduziert die Startzeiten und den Speicherbedarf neuer JVM-Instanzen durch die Einbindung sogenannter `.jsa`-Archive. Diese enthalten nur für die exakte JVM-Version und -Plattform die Metadaten häufig genutzter JDK-Klassen in einer bereits geparster und direkt verwendbare Form. Das Archiv kann direkt read-only in den Speicherbereich der JVM eingebunden werden, was dem Betriebssystem sowohl das Caching als auch eine Wiederverwendung dieser Speicherbereiche erlaubt. Anstatt beim Start neuer Java-Instanzen also jedes Mal erneut das komplette

GC	Optimiert für...	Kommentar
G1	Balance	JVM-Standard. Weitestgehend nebenläufig. Zielt auf Balance von Durchsatz & Latenz. Ausreißer bei den Pausen bis 250 ms ~ 800 ms. Insgesamt guter Durchsatz. Strategie: Häppchenweise Pausen an Zeitbudget orientiert
Shenandoah	Latenz	Anders als ZGC auch verfügbar für JDK 8, JDK 11 und 32-Bit-Architekturen. Schwächelt bei vielen `WeakRef`-Instanzen
ZGC	Latenz	Pausen sind komplett unabhängig von den Größen für Live- und Root-Set
ParallelGC	Durchsatz	Parallel und mehrere Threads. Hoher Durchsatz. Typische Pausen ~300 ms, abhängig von Heap-Größe
SerialGC	Speicherbedarf	Single-Threaded. Empfiehlt sich nur für kleine Heaps um die 100 MB
Zing/Azul	Pauseless	Nicht im OpenJDK; nur kommerziell verfügbar

Tabelle 1: Übersicht über die Garbage Collectors aktueller Java-Versionen

```
# Erstellung eigener AppCDS-Archive mittels „Trockenlauf"
$ java -XX:ArchiveClassesAtExit=myapp.jsa -cp myapp.jar MyApp

# Einbindung & Nutzung des erzeugten AppCDS-Archives
$ java -XX:SharedArchiveFile=myapp.jsa -cp myapp.jar MyApp
```

Listing 1: Java 13 vereinfacht erheblich die Erstellung und Nutzung eigener AppCDS-Archive

Klassenarchiv des JDK einlesen, analysieren und im Speicher anlegen zu müssen, kann dies nun bei parallel laufenden oder kurz zuvor beendeten JVMs weitestgehend entfallen. Dementsprechend bringt das *Class-Data-Sharing* erhebliches Potenzial sowohl bei den Start-up-Zeiten als auch für den initialen Speicherbedarf häufig oder mehrfach gestarteter JVMs, wie sie etwa im Container-Betrieb häufig auftreten. Das *Application-Class-Data-Sharing (AppCDS)* erweitert diesen Ansatz, sodass auch eigene Applikationsklassen mit aufgenommen werden können.

Neu ist, dass dies nun erheblich einfacher gelingt. So liefert Java nun ab Werk direkt ein passendes CDS-Archiv mit, das eine Auswahl der häufigsten JDK-Klassen abdeckt. Seit Java 13 sind nun auch eigene AppCDS-Archive erheblich einfacher. Es genügt, die Anwendung einmalig mit speziellen Parametern zu nutzen, um ein optimiertes AppCDS-Archiv zu erhalten *(siehe Listing 1)*.

Noch mehr Optimierungspotenzial bietet die kombinierte Verwendung mit dem Tool `jlink`, um vorher alle ungenutzten Klassen aus

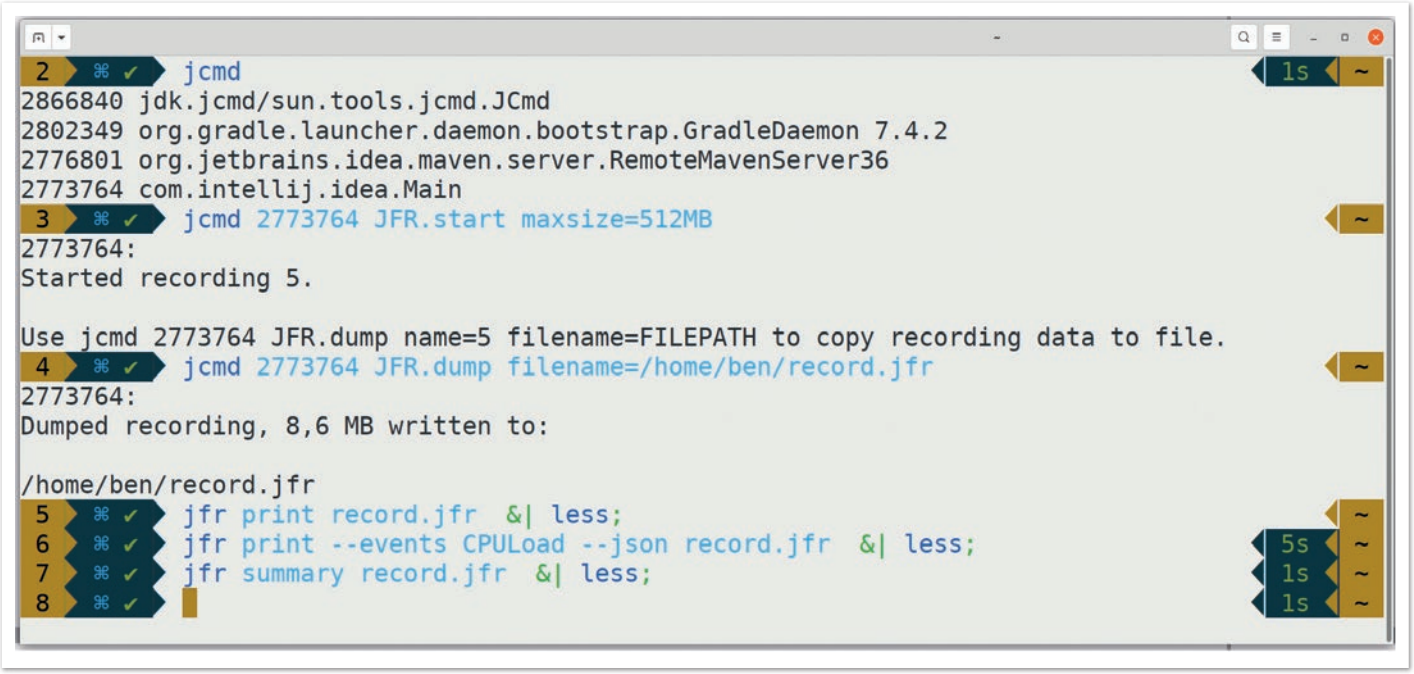


Abbildung 1: jcmd ermöglicht die grundlegende Steuerung des JFR und eine rudimentäre Analyse der gesammelten Daten.

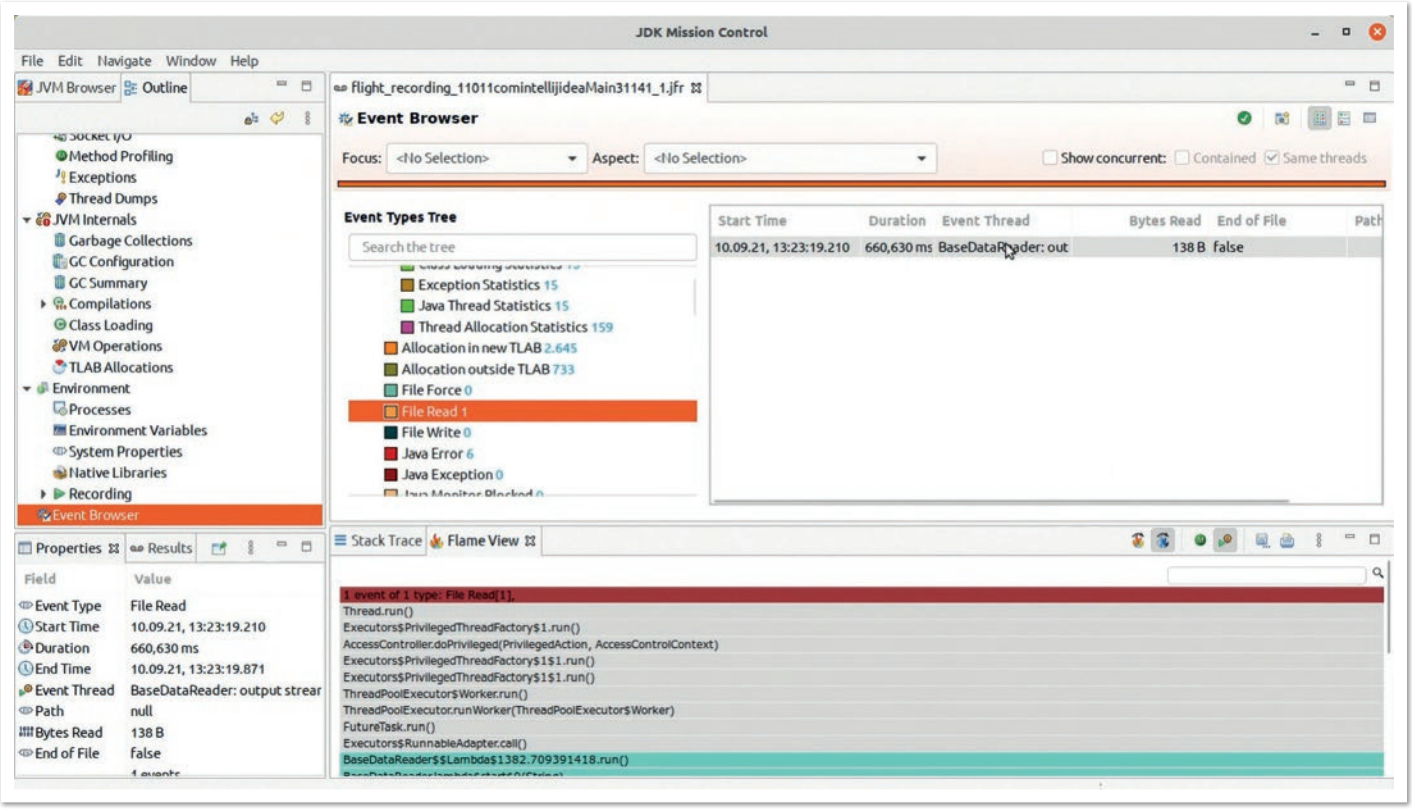


Abbildung 2: Auch die Analyse und Auswertung von JFR-Mitschnitten geht mit dem JDK Mission Control komfortabel von der Hand

den Bibliotheken zu entfernen. Diesen Gewinnen stehen jedoch ein gesteigerter Aufwand und eine Komplexität beim Build gegenüber, was es abzuwägen gilt.

Fliegende Einsichten: Monitoring mit dem JDK Flight Recorder

Spannende Neuerungen bringen der JDK Flight Recorder (JFR) und das JDK Mission Control *[5]*. Der JDK Flight Recorder ist in etwa das Äquivalent zum Black-Box-Flugschreiber bei Luftfahrzeugen und zeichnet fortlaufend die Ereignisse der JVM und ihrer ausgeführten Anwendung auf. Das ehemals von Oracle kommerziell vertriebene JVM-Add-on erfasst eine Vielzahl von Messpunkten zum Profiling von Methoden, zur Veränderung der Speicherbelegung und zu den Garbage-Collector-Vorgängen. Des Weiteren erlaubt es eine detaillierte Diagnostik der Betriebsparameter des Betriebssystems, der JVM, des JDK und der eigenen Anwendung.

Ein wesentlicher und sicher auch namensgebender Aspekt ist der in der Regel sehr geringe Overhead von weniger als 1 %. Der JDK Flight Recorder entstammt von den Machern der JVM und kann somit direkt präzise auf die Daten zugreifen, die sowieso schon von der JVM erfasst werden. Eine Aktivierung der Datensammlung ist jederzeit, sowohl für neue als auch bereits laufende Java-Instanzen, möglich und hat keinen messbaren oder gar spürbaren Einfluss auf die Anwendung selbst. Das macht den JDK Flight Recorder ideal für den Dauerbetrieb in Produktion und eröffnet neue Möglichkeit für Zeitreisen: Tritt ein Problem auf, können die Aufzeichnungsdaten ad hoc rückwirkend gesichert und die Laufzeitsituation vor dem Auftreten genau inspiziert werden. Auch bei einem Crash der JVM kann der JDK Flight Recorder seine Daten noch automatisiert speichern und eine Post-Mortem-Analyse zur Verfügung stellen.

Das mit Java direkt ausgelieferte Kommandozeilenwerkzeug `jcmd` kann JFR-Mitschnitte starten und exportieren. Zusätzlich stellt es rudimentäre Mittel bereit, um die gesicherten Protokolle zu filtern, als JSON zu exportieren oder aufsummierte Statistiken der Events zu erhalten. Deutlich komfortabler geht dies mit dem nun ebenfalls frei erhältlichen JDK Mission Control, das die exportierten `.jfr`-Dateien laden oder aber auch selbst neue Flight-Recorder-Sitzungen starten und analysieren kann *(siehe Abbildung 2)*. Es ermöglicht deutlich komfortabler die feingranulare Konfiguration der Aufzeichnung der aktuell 165 verfügbaren JDK Flight Recorder Events *[6]* in Gruppen bis hin zur individuellen Auswahl und Schwellenwertkonfiguration *(siehe Abbildung 3)*.

Beide Werkzeuge liefern bereits beachtliche Möglichkeiten für das Profiling und die Problemanalyse im Bedarfsfall. Mit Java 14 gewinnt JFR nochmals eine neue Dimension, denn das eingeführte JFR-Event-Streaming-API erlaubt nun auch die programmatische Analyse der JFR-Daten im laufenden Betrieb. Erklärtes Ziel dieses API ist es, das Auslesen und Beobachten der JFR-Events so trivial wie möglich zu gestalten, um auf diese auch automatisiert reagieren zu können.

Der JDK Flight Recorder sammelt im Betrieb fortlaufend die zur Erfassung aktivierten Events aus der JVM, des Java-API sowie der eigenen Anwendung und legt diese in einem rollierenden Ringpuffer ab *(siehe Abbildung 4)*. Von dort werden diese in regelmäßigen Intervallen auf die Disk geschrieben. Über das JFR-Event-Streaming-API können nun Anwendungen dieses fortlaufend aktualisierte Disk Repository anzapfen, beobachten und einen gefilterten Stream der auftretenden JFR-Events abonnieren. Seit Java 16 ist dies nun mittels JMX auch über das Netzwerk möglich. *Listing 2* illustriert, wie die Protokollierung der CPU-Last und aktiver Locks länger als 10 ms aktiviert und beim Eintritt als Statusmeldung ausgegeben werden kann. Denkbar sind hier auch weit raffiniertere Ansätze. So könnte etwa beim Auftreten eines Betriebsproblems nach Art einer Timeshift-Funktion ein Ausschnitt der vergangenen zehn Minuten aus dem JFR Event Repository extrahiert und anschließend für eine spätere Analyse exportiert werden.

Das JFR-API erlaubt auch, eigene Event-Typen zu definieren *(siehe Listing 3)* und in der Anwendung auszulösen *(siehe Listing 4)*. Dies er-

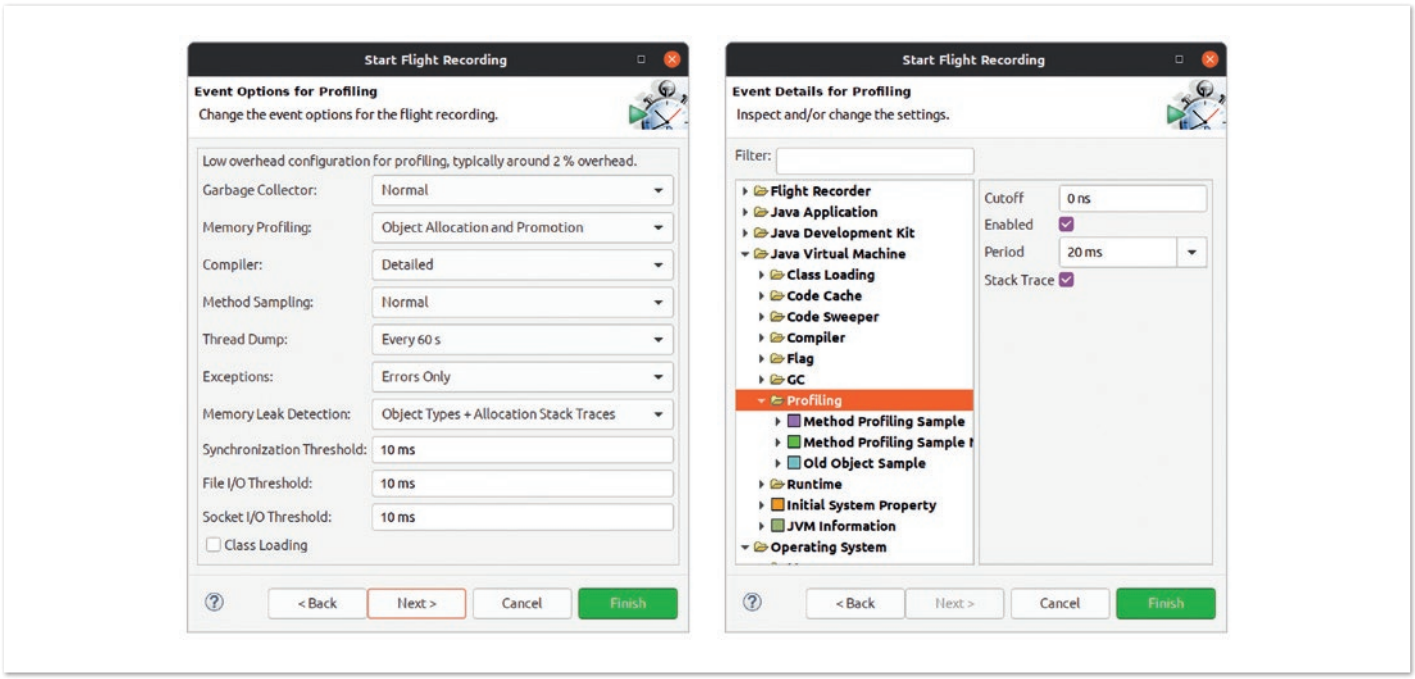


Abbildung 3: Das interaktive JDK Mission Control erlaubt die feingranulare Konfiguration der JDK-Flight-Recorder-Aufzeichnung

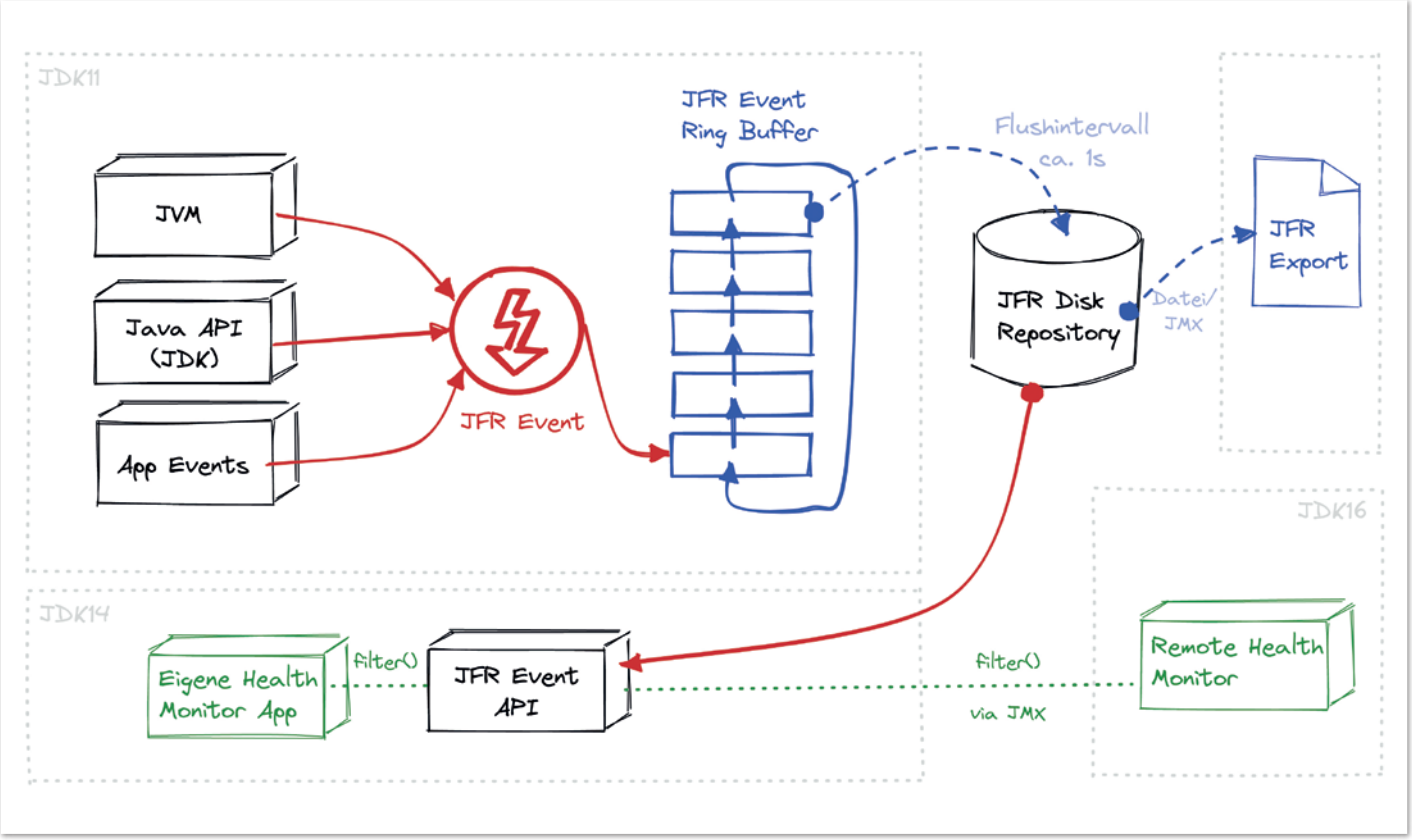


Abbildung 4: Dank JFR Event Streaming können eigene Anwendungen sowohl lokal als auch remote laufende Java-Anwendungen fortlaufend analysieren und bei Bedarf detaillierte Diagnosen ausführen

laubt fortschrittlichere Analysen, da sich damit die technische Betriebssituation leichter mit fachlichen Ereignissen korrelieren lässt.

Insgesamt steht mit dem JFR-Streaming-API ein leistungsfähiges Werkzeug zur programmatischen Analyse und Überwachung der eigenen Anwendungen zu Verfügung. Test-Bibliotheken wie *jfrunit* oder *QuickPerf* erweitern diese und erlauben über JUnit-Testfälle, das Betriebsverhalten eigener Anwendungen automatisiert zu analysieren und sicherzustellen.

Fremde Gefilde: Das Foreign Function & Memory API

Schon länger verfügbar, allerdings selbst unter Java 19 immer noch im experimentellen *Preview*-Status *[7]*, ist das *Foreign Function & Memory API*. Dieses möchte mit der veralteten, sehr kleinteiligen und fragilen Methode aufräumen, fremden Maschinencode und Bibliotheken einzubinden und zu nutzen. Angesichts zahlreicher etablierter und leistungsfähiger Drittbibliotheken wie Tensorflow im Bereich AI/ML oder OpenSSL im Bereich verschlüsselter Kommunikation versucht man sich hier über eine ausgiebige Experimentierphase an einem optimalen Schnitt für eine einfache, sichere und performante JNI-Alternative. Das Foreign Function & Memory API will es ermöglichen, mit Code und Daten außerhalb der Java Runtime zu interagieren, ohne sich der Brüchigkeit und den Gefahren von JNI aussetzen zu müssen.

Listing 5 gibt einen Eindruck über die Verwendung dieses neuen API. Obwohl dieser Code schon wesentlich einfacher und bequemer als JNI ist, gibt es weitere Werkzeuge, die helfen können. Die Early-Access Builds des Project Panama *[8]* enthalten zusätzlich das

```
try (var rs = new RecordingStream()) {
  rs.enable("jdk.CPULoad")
        .withPeriod(Duration.ofSeconds(1));
  rs.enable("jdk.JavaMonitorEnter")
        .withThreshold(Duration.ofMillis(10));

  rs.onEvent("jdk.CPULoad", event -> {
     System.out.println(event.getFloat("machineTotal"));
  });
  rs.onEvent("jdk.JavaMonitorEnter", event -> {
     System.out.println(event.getClass("monitorClass"));
  });

  rs.start(); // Blockierender Aufruf, bis Stream endet
  // rs.startAsync();   // Alternativ als separater Thread
}
```

Listing 2: Das JFR-Event-Streaming-API ermöglicht die programmatische Konfiguration und Analyse

auf LLVM aufbauende Werkzeug `jextract`. Dieses generiert direkt aus .h-Dateien die passenden API-Wrapper als `.class`- oder `.java`-Dateien mit den notwendigen API-Aufrufen und macht damit die Einbindung fremder Bibliotheken noch komfortabler.

Gut verpackt: Native Installationspakete mit jpackage

Entwickler, die Stand-alone-Java-Applikationen entwickeln, finden ab Java 16 mit `jpackage` nun auch direkt ab Werk eine Lösung vor, um ihren Anwendern installierbare Installationspakete bereitzustellen. Dazu kann `jpackage` für Windows `.msi`- und

`.exe`-Dateien, für MacOS `.pkg`- und `.dmg`-Archive und für Linux `.deb`- und `.rpm`-Installationspakete erzeugen. Diese sind komplett eigenständig und haben auch das benötigte JRE bereits mit im Gepäck. Eigene Startparameter für JVM und Anwendung werden ebenso unterstützt wie das Verknüpfen der eigenen Anwendung mit bestimmten Dateitypen. Zugunsten der Komplexität verzichtet `jpackage` jedoch auf Funktionen wie einen automatischen Update-Mechanismus oder die Unterstützung eines Splash-Screens zum Programmstart. Hier muss der Entwickler weiterhin eigene Lösungen bereitstellen.

Unzählige Kleinigkeiten

Das kurze Release-Intervall von sechs Monaten hat zur Folge, dass eine Vielzahl hier bislang nicht genannter Detail-, Korrektur- und Pflegeänderungen fortlaufend Einzug in das OpenJDK halten. So liefert beispielsweise die JVM bei `NullPointerException` nun auch Hinweise auf die auslösende Variable, kompiliert mit der Option `-g:vars` auch für lokale Variablen und Lambdas. Das Javadoc-Tool hat in Version 16 ein umfassendes Facelift erhalten und wartet nun mit verbesserter Suche und einem mobile-freundlichen Layout auf. Daneben bietet es deutlich mehr Hilfestellungen und Prüfungen für die Autoren an. Die JVM unterstützt zum Beispiel nun mit `-Xlog:async` auch asynchrones Logging *[8]* oder kann mit `-XX:+ExtensiveErrorReports` ausführlichere Fehlerberichte erzeugen. Die Liste dieser kleinen Verbesserungen ist lang. Der sehr empfehlenswerte Java-Version-Almanac *[7]* gibt hier Interessierten detailliert Auskunft über die neuen Features und eingeflossenen JEPs aller bisherigen und kommenden OpenJDK Releases und bietet direkten Zugriff auf die wichtigsten Dokumente und Downloads. Zudem erlaubt er die APIs zweier JDK-Versionen vergleichend direkt gegenüberzustellen.

Fazit

Seit Java 11 LTS hat sich im Bereich von Performance, Werkzeug und Diagnostikmittel viel getan. Auch ohne Codemigration lohnt das Up-

```
import jdk.jfr.*;

@Name("de.bentolor.ButtonPressed")
@Label("Button Pressed")
@StackTrace(false)
public class ButtonEvent extends Event {
  @Label("Button name")
  public String name;

  @Label("Source")
  public String trigger;

  @Label("Number of Bounces")
  @DataAmount
  public int bounces;

  @Label("Has timeouted")
  public boolean timeouted;
}
```

Listing 3: Die Deklaration eines eigenen JFR-Events geht leicht von der Hand

```
ButtonEvent evt = new ButtonEvent();
if(evt.isEnabled()) {
  evt.name = "Button 1";
  evt.trigger = "Keyboard";
  evt.begin();
}

// doSomething()

if(evt.isEnabled()) {
  evt.end();
  evt.timeouted = false;
  evt.bounces = 3;
  evt.commit();
}
```

Listing 4: Eigene JFR-Events können spielend leicht protokolliert und ausgelöst werde

```
import java.lang.invoke.*;
import jdk.incubator.foreign.*;

class CallPid {
  public static void main(String... p) throws Throwable {
     //  adressiertes Symbol - hier aus den System Libraries
     var libSymbol = CLinker.systemLookup().lookup("getpid").get();

     //   gewünschte Java-Signatur des Java Foreign Handles
     var javaSig = MethodType.methodType(long.class);

     //   Ziel-Signatur der aufzurufenden C-Funktion
     var nativeSig = FunctionDescriptor.of(CLinker.C_LONG);

     //   Funktionshandle beziehen
     CLinker cABI = CLinker.getInstance();
     var getpid = cABI.downcallHandle(libSymbol, javaSig, nativeSig);

     //   Aufruf der getpid() Systemfunktion
     System.out.println((long) getpid.invokeExact());
  }
}
```

Listing 5: Über das Foreign Function & Memory API lässt sich direkt in Java Code aus nativen Bibliotheken aufrufen. Hier ein Betriebssystemaufruf zur Ermittlung der Prozess-ID

grade aufgrund der unzähligen Detailverbesserungen bei der täglichen Verwendung, für den effizienten Betrieb und für die einfachere Fehlersuche bei Problemen. Dazu gehören insbesondere auch der JDK Flight Recorder in Kombination mit JDK Mission Control, die in die Werkzeugkiste eines jeden Java-Entwicklers sollten.

Quellen

[1] *https://wiki.openjdk.org/display/zgc/Main*
Per Liden, OpenJDK Wiki (2022)

[2] *https://wiki.openjdk.org/display/shenandoah*
Aleksey Shipilev, OpenJDK Wiki (2022)

[3] *https://blogs.oracle.com/javamagazine/post/understanding-the-jdks-new-superfast-garbage-collectors*
Understanding the JDK's New Superfast Garbage Collectors, Raoul-Gabriel Urma
Richard Warburton (2019)

[4] *https://docs.oracle.com/en/java/javase/14/vm/class-data-sharing.html*
Oracle Help Center (2022)

[5] *https://openjdk.org/projects/jmc/8/*
JMC 8, OpenJDK (2021)

[6] *https://bestsolution-at.github.io/jfr-doc/*
Java Flight Recorder Events, BestSolutions.at (2022)

[7] *https://javaalmanac.io/*
The Java Version Almanac, Marc R. Hoffmann & Cay S. Horstmann (2022)

[8] *https://chriswhocodes.com/hotspot_options_openjdk20.html*
VM Options Explorer, Chris Newland (2022)

[9] *https://jdk.java.net/panama/*
Project Panama Early-Access Builds (2022)

Benjamin Schmid
Nagarro GmbH
ben@tolor.de

Benjamin brennt für lösungsorientierte Innovationen, Effizienz und Qualität in der Softwareentwicklung und ist in Rollen wie CTO, Technology Advisor, Manager R&I und Solution Architect immer wieder erster Ansprechpartner in allen technologischen und methodischen Fragestellungen. Seine Schwerpunkte liegen im Bereich von Java-, Web- und Cloud-Native-Architekturen. Auf der stetigen Suche nach innovativen, soliden und nachhaltigen Lösungen gehören sichere Sprachen wie Kotlin und automatisierte Codeprüfungen zu seinen Steckenpferden.

GraphQL Subscriptions mit Spring Boot 2.7, Project Reactor und React

Heinz-Werner Haas & François Fernandes, Digital Frontiers GmbH & Co. KG

Die Erwartungshaltung der Anwender bei der Interaktion mit Anwendungen ist, dass sie ständig über Aktualisierungen informiert werden. Sei es die Benachrichtigung über den Eingang einer E-Mail, eine Erinnerung an einen Termin oder die erwartete Antwort auf eine Chat-Nachricht. In all diesen Fällen werden Informationen, sobald diese verfügbar sind, dem Anwender mitgeteilt. Doch wie genau realisiert man so etwas? Eine Möglichkeit basierend auf Spring Boot 2.7 und GraphQL [1] betrachten wir in diesem Artikel: GraphQL Subscriptions.

Was sind GraphQL Subscriptions?

Bei einer Subscription handelt es sich wie bei Abfragen (Query) und Mutationen (Mutation) um einen Operationstyp, der durch GraphQL definiert wird. Subscriptions unterscheiden sich jedoch in ihrer Ausführung deutlich von den anderen beiden Operationstypen. Abfragen und Mutationen sind klassische Request-Response-Interaktionen, bei denen der Server auf eine Anfrage (Request) reagiert und eine Antwort (Response) sendet.

Was jedoch, wenn ein Server dem Client Informationen mitteilen will? Normalerweise müsste dafür erst ein Request seitens des Clients gestartet werden. Nicht so bei Subscriptions. Hier wird eine aktive Verbindung zum GraphQL-Server aufgebaut, sodass Änderung, Aktualisierung oder Events unmittelbar an den Client gesendet werden können, was die Besonderheit dieses Operationstyps darstellt. Diese Verbindung wird in den meisten Fällen mittels WebSockets erzielt.

Ein WebSocket ist ein dauerhafter bidirektionaler Kommunikationskanal zwischen einem Client (etwa einem Browser) und einem Server-Dienst. Im Gegensatz zu klassischen HTTP-Anfrage-/Antwort-Verbindungen können WebSockets eine beliebige Anzahl von Nachrichten übermitteln und bieten Server-zu-Client-Kommunikation ohne zyklische Abfragen (auch bekannt als Polling).

In welchen Fällen lohnt es sich, Subscriptions zu verwenden? Sie finden typischerweise Anwendung bei folgenden Szenarien:

- Bei kleinen oder inkrementellen Änderungen an großen Objekten, da das wiederholte Abfragen eines großen Objekts zeit- und datenintensiv ist, insbesondere wenn die meisten Felder des Objekts sich nur selten ändern.
- Bei Aktualisierungen in Echtzeit. (Dieser Anwendungsfall wird im Beispiel dieses Artikels verwendet.)

Betrachten wir Subscriptions einmal an dem Beispiel einer Kommentarfunktion. Dieses Beispiel ist dabei der Subscription-Dokumentation des Apollo-Frameworks entnommen, die sich unter *[2]* findet und als zusätzliche Lektüre für die Client-Implementierung zu empfehlen ist.

Um einen Client automatisch zu benachrichtigen, wenn ein neuer Kommentar hinzugefügt wird, muss zunächst eine Subscription im GraphQL-Schema definiert werden, wie in *Listing 1* auszugsweise zu sehen.

In diesem Beispiel wird die Subscription commentAdded definiert. Der Client kann diese Subscription nutzen, indem ein GraphQL-Query-Dokument erstellt *(siehe Listing 2)* und die Abfrage über einen GraphQL-Client versendet wird.

Der GraphQL-Client wird daraufhin einen Verbindungsaufbau mit dem Server initiieren und die Verbindung aufrechterhalten. Ab diesem Zeitpunkt kann der Server zu jedem beliebigen Zeitpunkt eine neue Antwort senden. Hier wird der Kontrast zum Konzept der Abfragen deutlich. Im Gegensatz zu den Abfragen hat die Subscription vorerst nicht die Erwartung, dass eine Antwort vom Server zurückkommt, da der Server nur dann Daten an den Client sendet, wenn ein bestimmtes Ereignis im Server eingetreten ist.

```
type Subscription {
  commentAdded(postID: ID!): Comment
}
```

Listing 1: Serverseitige Implementierung eines GraphQL-Schemas für eine Subscription

```
subscription OnCommentAdded($postID: ID!) {
  commentAdded(postID: $postID): {
     id
        content
     }
}
```

Listing 2: GraphQL-Query-Dokument für die Kommentar-Subscription

```
{
  "data": {
     "commentAdded": {
        "id": "123",
     "content": "What a thoughtful and well written post!"
     }
  }
}
```

Listing 3: Struktur der Antwortdaten vom GraphQL-Server

Die Daten, die der GraphQL-Server anschließend an den Client sendet, entsprechen immer der Struktur der ausgeführten Subscription. Dies gleicht dem Antwortformat einer Abfrage. Wie dies für das aufgeführte Beispiel aussehen kann, wird in *Listing 3* dargestellt.

Dabei endet die Kommunikation nicht zwingend mit einer Antwort, sondern kann grundsätzlich beliebig viele Antworten enthalten.

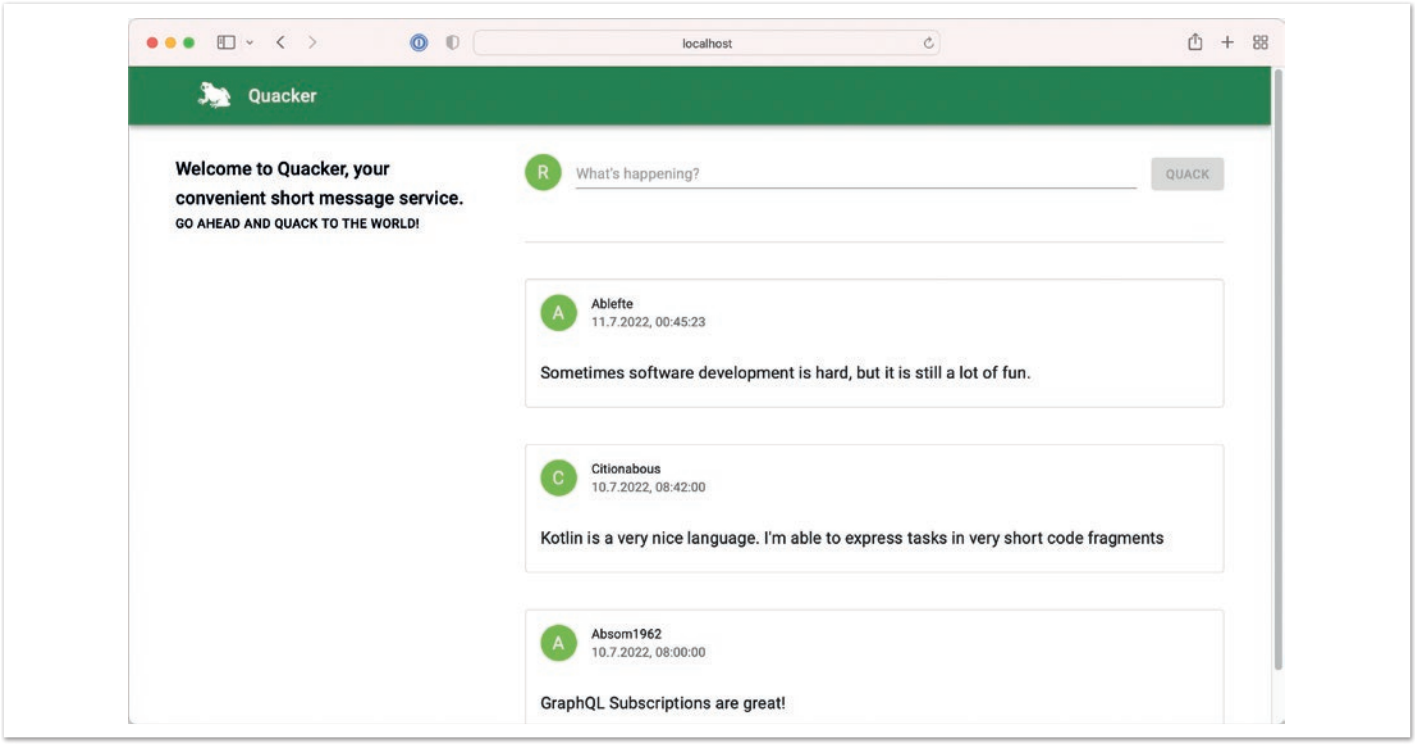


Abbildung 1: Screenshot der Beispielanwendung

Das Beispielprojekt

Zur Veranschaulichung von GraphQL Subscriptions in Kombination mit Spring Boot, Project Reactor *[3]* und React werden wir uns im Folgenden einer Beispielanwendung bedienen. Der Grundgedanke der Anwendung ist sehr simpel: Menschen haben ein Mitteilungsbedürfnis, das gestillt werden soll. Und das natürlich mit einer möglichst großen Reichweite. Derartige Kommunikation wird umgangssprachlich gerne als „quatschen" oder gar „quaken" bezeichnet. Es liegt daher nahe, einen Kurzmitteilungsdienst zu schaffen, der sich Quacker nennt.

Selbstverständlich sollen andere Anwender von Quacker auch unmittelbar über neue Mitteilungen informiert werden, ohne die Seite neu laden zu müssen. Genau hierfür wollen wir GraphQL Subscriptions einsetzen.

Die Anwendungsstruktur ist dabei unkompliziert und besteht aus zwei wesentlichen Komponenten: einem Server, der das GraphQL-API bereitstellt, und einem Web-Client, der dieses API konsumiert. Der vollständige Code der Anwendung findet sich in einem GitHub Repository *[4]*. Ein Bild sagt bekanntlich mehr als tausend Worte, daher ist in *Abbildung 1* ein Screenshot der Anwendung zu sehen.

Bei GraphQL spielt das Schema eine zentrale Rolle, was der Grund für uns war, die Anwendung Schema-first umzusetzen. Damit hatten wir schnell ein Schema, das alle unsere Anforderungen enthält. Ein Auszug findet sich in *Listing 4*.

```
type Query {
  post(id: ID!): Post!
  timeline: [Post!]!
}

type Mutation {
  createPost(message: String!): Post!
}

type Subscription {
  onTimelineUpdate: Post!
}

type Post {
  id: ID!
  message: String!
}
```

Listing 4: Auszug aus dem GraphQL-Schema

Das Schema ist denkbar einfach. Neben den von GraphQL definierten Operationstypen (Query, Mutation und Subscription) haben wir lediglich einen Typ Post, der die eigentliche Kurznachricht repräsentiert. Eine besondere Bedeutung hat *Query.timeline:* Hierüber werden die jeweils aktuellen Posts beim initialen Laden der Seite abgerufen um, diese dem Anwender zu präsentieren.

Spannender wird es jedoch mit *Mutation.createPost*. Hiermit können neue Posts erstellt und auf dem Server gespeichert werden. Allerdings wird dieser Post nicht nur gespeichert, sondern es sollen auch alle aktiven Clients über diesen Post informiert werden. Genau das wird über die Subscription *onTimelineUpdate* realisiert.

Implementation des Servers

Der Server basiert, wenig überraschend, auf Spring Boot 2.7, mit dem entsprechenden Starter für GraphQL und damit implizit auch Project Reactor (näheres zu Project Reactor findet sich unter *[3]*). Als Programmiersprache kommt bei diesem Beispiel Kotlin zum Einsatz. Sämtliche folgenden Beispiele lassen sich jedoch in jeder typischen JVM-Sprache umsetzen.

Bevor wir in die GraphQL-Umsetzung eintauchen, ist es jedoch notwendig, einen Blick auf den Kernlogik der Anwendung zu legen. Diese ist in dem Service gekapselt, der die Kurznachrichten verwaltet. Die Kurznachrichten werden hierbei als Posts bezeichnet, womit sich auch der Name *PostService* ergibt. In *Listing 5* findet sich ein Auszug aus dem Service, der die wesentlichen Methoden enthält.

```
@Service
class PostService(/* [...] */)) {

  fun getPost(id: String): Post?
  { /* [...] */ }

  fun createPost(user: User, message: String)
  { /* [...] */ }

  fun watch(callback: WatchCallback): UnregisterHandler
  { /* [...] */ }
}
typealias WatchCallback = (post: Post) -> Unit
typealias UnregisterHandler = () -> Unit
```

Listing 5: Auszug aus dem PostService

Auf den ersten Blick ist dies ein normaler Service, der das Erstellen und Abrufen von Entitäten (unseren Posts) kapselt. Die Besonderheit ist die Methode *watch*. Über diese kann ein *WatchCallback* registriert werden, der bei jedem neu erstellten Post informiert wird. Dies bildet die Grundlage für die Implementierung eines GraphQL-Subscription-API.

Die Implementation eines GraphQL-Subscription-API unterscheidet sich nur im Detail von typischen Spring-Controllern. Aus dem GraphQL-Schema wissen wir, dass unsere Operation den Namen *onTimelineUpdate* trägt. Damit die GraphQL-Integration die passende Implementierung findet, muss eine Controller-Methode mit *@SubscriptionMapping* annotiert werden und der Methodenname muss mit dem Namen der Operation übereinstimmen, wie in *Listing 6* zu sehen.

```
@Controller
class GraphQLSubscriptionController(private val post-
Service: PostService) {

  @SubscriptionMapping
  fun onTimelineUpdate(): Flux<Post> {
     return Flux.create { sink ->
        val unregister = postService.watch {
           post -> sink.next(post)
        }
        sink.onDispose { unregister() }
     }
  }
}
```

Listing 6: GraphQLSubscriptionController

Alternativ kann auch ein frei gewählter Methodenname verwendet werden. In diesem Fall muss aber ein explizites Mapping über die *@SubscriptionMapping*-Annotation definiert werden.

Vergleichen wir nun die Deklaration im GraphQL-Schema mit der Implementation des Controllers, fällt eines auf: Der Typ des Feldes *onTimelineUpdate* ist im Schema lediglich ein Post, die Implementierung verwendet jedoch *Flux*, was mehr als nur einen Wert repräsentiert. In genau diesem kleinen Detail versteckt sich der wesentliche Teil der Subscription-Implementierung.

Jeder Wert, der über *Flux* emittiert wird, resultiert letztendlich in einer Nachricht an den Client. Somit wird der Client immer über neu entstehende Posts informiert. Es ist jedoch wichtig zu verstehen, dass die Methode *onTimelineUpdate* lediglich einmal für jede Subscription aufgerufen wird. Der zurückgegebene *Flux* bleibt dann so lange gültig, bis entweder keine Elemente mehr emittiert werden (Flux completed) oder der Client die Verbindung beendet. Beendet der Client die Verbindung, wird dies als Dispose einer Project Reactor Subscription umgesetzt.

Somit erklärt sich auch leicht die Implementierung des Flux. Mittels *Flux.create* wird ein Emitter implementiert, der wiederum einen *WatchCallback* im *PostService* registriert. Erhält dieser *WatchCallback* einen neuen Post, wird dieser direkt als weiteres Element im Flux emittiert und schlussendlich an den Client gesendet. Ist die Verbindung beendet, wird ein über *sink.onDispose* registrierter Callback aufgerufen, der den *WatchCallback* deregistriert. Damit haben wir eine funktionierende Implementierung einer Subscription.

Testen eines Subscription-Controllers

Natürlich sollte eine solche Subscription auch getestet werden. Wie testet man allerdings eine Logik, die auf Callbacks und Parallelität ausgelegt ist? Hier kommt die sehr gute Integration von Spring Boot, GraphQL und Project Reactor zum Tragen.

Um eine Subscription-Logik zu testen, müssen drei Aufgaben für einen Testfall bewältigt werden:

1. **Bereitstellen der Testdaten**
   Dies kann durch eine reine Testdaten-Implementierung oder einen Mock realisiert werden. In unserem Beispiel verwenden wir einen Mock, der vordefinierte Posts emittiert. Spring bietet hier bereits eine entsprechende Integration, um Mocks zu erstellen und in einem ApplicationContext zu injizieren.
2. **Ausführen der Subscription**
   Hier bietet spring-graphql-test die Klasse GraphQLTester an, mittels derer jegliche GraphQL-Anfragen simuliert werden können. Für Subscriptions erhält man ein Flux, das die Antworten enthält.
3. **Validieren der Ergebnisse**
   Wir haben ein Flux erhalten, dessen Ergebnisse validiert werden müssen. Hierfür bietet wiederum Project Reactor einen StepVerifier, um eine Sequenz von emittierten Elementen eines Flux zu verifizieren.

Um nun zu testen, ob sich unser Controller korrekt verhält, führen wir diese drei Schritte durch. In *Listing 7* ist ein Auszug der Testklasse zu sehen. Der vollständige Code findet sich in der Class *GraphQLSubscriptionControllerTest* im GitHub-Projekt *[4]*.

Die Elemente, die das *Flux* emittiert, sind dabei vom Typ *GraphQLTester.Response*. Dieser spezielle Typ kapselt die Antwort, beziehungsweise das emittierte Element, und bietet Test-Methoden für das Validieren der Antwort. Diese Methoden basieren auf JsonPath, da GraphQL selbst JSON für die Übertragung von Daten verwendet. Somit können die erwarteten Ergebnisse einfach mit einem JsonPath und einem Aufruf von *matchesJson* validiert werden.

React-Client-Implementation

Nach erfolgreicher Implementierung des Servers widmen wir uns nun der Implementierung des Clients. Dafür verwenden wir React im Zusammenspiel mit dem Apollo-Client für GraphQL *[5]* und TypeScript als Programmiersprache.

```
@Test
internal fun `subscriber notified on new entries`() {

  // 1. vorbereiten des Mocks
  // [...]

  // 2. Ausführen der GraphQL Subscription
  val subscription = graphQLTester.document(
     """
        subscription {
           onTimelineUpdate { id message }
        }
     """
  )
     .executeSubscription()
     .toFlux()

  // 3. validieren der Ergebnisse
  StepVerifier.create(subscription)
     .assertNext { response ->
        response.errors().verify()
        response.path("$.data.onTimelineUpdate.id")
           .matchesJson("\"12\"")
     }
     .thenCancel()
     .verify(Duration.ofSeconds(5))
}
```

Listing 7: Auszug aus GraphQLSubscriptionControllerTest

```
import { GraphQLWsLink } from "@apollo/client/link/subscriptions";
import { createClient } from "graphql-ws"

const wsLink = new GraphQLWsLink(
  createClient({
     url: "ws://localhost:8080/graphql",
  })
);
```

Listing 8: Initialisierung eines GraphQLWsLink für die Verbindung zum Server

Wie bereits beschrieben, werden Subscriptions in den meisten Fällen über eine WebSocket-Verbindung umgesetzt. In diesem Beispiel wird durch die Verwendung von Spring die Library graphql-ws als Interaktionsprotokoll vorausgesetzt; dies muss dem Projekt lediglich als Abhängigkeit hinzugefügt werden. Um Apollo für eine WebSocket-Kommunikation vorzubereiten, muss ein entsprechender *GraphQLWsLink* definiert werden, wie in *Listing 8* zu sehen.

Grundsätzlich könnte die gesamte Kommunikation zur Ausführung aller Operationstypen (Abfragen, Mutationen und Subscriptions) über diesen Link und damit einen WebSocket durchgeführt werden. Abfragen und Mutationen erfordern allerdings keine dauerhafte Verbindung, weshalb es empfehlenswert ist, für eben diese HTTP zu verwenden.

Damit stellt sich allerdings die Frage, wie der Apollo-Client konfiguriert werden muss, damit er sowohl HTTP für Anfragen und Mutationen als auch WebSockets für Subscriptions verwendet.

Praktischerweise gibt es hierfür eine integrierte Funktionalität: den Split-Link. Dieser wird mithilfe einer simplen Split-Funktion erstellt. Der Split-Link ermöglicht, je nach Operation zu entscheiden, ob diese mit dem WebSocket-Link oder dem HTTP-Link ausgeführt werden soll.

Die Implementierung hierfür, die sowohl den WebSocket-Link als auch einen HTTP-Link enthält, ist in *Listing 9* zu sehen. Am Ende des Listings findet sich auch der Split-Link, der, basierend auf dem Operations-Typ, entscheidet, welcher Link verwendet werden soll.

Nachdem die Unterscheidung beider Links erfolgt ist, kann der Apollo-Client definiert werden. Wie diese Initialisierung aussehen kann, ist dem *Listing 10* zu entnehmen.

Der letzte Schritt zum funktionsfähigen Apollo Setup mit React ist das Einbinden eines Apollo-Providers, der den *ApolloClient* transparent in der React-Komponenten-Hierarchie bereitstellt.

Betrachten wir nun die konkrete Umsetzung der Subscription und die Einbindung der Ergebnisse in die Webanwendung. Zur Umsetzung wird zunächst ein GraphQL Query Document benötigt, das sowohl die zu verwendende Subscription als auch die erwarteten Rückgabefelder definiert. Das Query-Dokument *(siehe Listing 11)* wird hierbei in eine eigene Datei geschrieben.

```
import { HttpLink, split } from "@apollo/client";
import { getMainDefinition } from "@apollo/client/utilities";
import { GraphQLWsLink } from "@apollo/client/link/subscriptions";
import { createClient } from "graphql-ws"

const httpLink = new HttpLink({
     uri: "/graphql",
  });

const wsLink = new GraphQLWsLink(
  createClient({
     url: "ws://localhost:8080/graphql",
  })
);

const splitLink = split(
  ({ query }) => {
     const definition = getMainDefinition(query);

     return (
        definition.kind === "OperationDefinition" &&
        definition.operation === "subscription"
     );
  },
 wsLink,
 httpLink
);
```

Listing 9: Implementierung des Split-Links

```
import { ApolloClient, InMemoryCache } from "@apollo/client";

// [...] Code aus Listing 9

const client = new ApolloClient({
  link: splitLink,
  cache: new InMemoryCache(),
});
```

Listing 10: Initialisieren des Apollo-Clients

```
subscription subscribeTimeline {
  onTimelineUpdate {
     id
     message
     postedAt
     user {
        displayName
        username
     }
  }
}
```

Listing 11: Query Document mit der Subscription

```
import { Post, useSubscribeTimelineSubscription } from "../generated/graphql";

const { data: subscriptionData } = useSubscribeTimelineSubscription();
const [timeline, setTimeline] = useState([] as Post[]);

useEffekt(() => {
  if (subscriptionData !== undefined)
     setTimeline([subscriptionData.onTimelineUpdate, …timeline]);
}, [subscriptionData]);
```

Listing 12: Code-Auszug der Implementierung und Verwendung des useSubscribeTimelineSubscription-Hook

Basierend auf dem Query-Dokument lassen wir mit dem GraphQL-codegen-Framework sogenannte React Hooks generieren. Diese generierten Hooks sind bereits entsprechend typisiert und erleichtern den Aufruf von GraphQL-Operationen ungemein. Eine ausführliche Beschreibung des GraphQL codegen und zugehöriger Plug-ins findet sich unter *[6]*.

Bleibt nun, den entsprechenden Subscription Hook zu verwenden. In *Listing 12* ist zu sehen, dass der generierte Hook den Namen *useSubscribeTimelineSubscription* trägt.

Dieser Hook verhält sich im Wesentlichen wie die meisten React Hooks. Die Besonderheit liegt im zurückgelieferten Data-Feld (das wir als *subscriptionData* zuweisen). Bei jedem neu gelieferten Post wird der Render-Zyklus der React-Komponente erneut gestartet und enthält damit den jeweils aktuellen Post in *subscriptionData*. Da wir eine Timline auf den immer wieder erhaltenen Posts aufbauen möchten, wird *useEffect* in Kombination mit einem State verwendet. Jedes Mal, wenn sich *subscriptionData* ändert, wird der Callback von *useEffect* aufgerufen. Enthält *subscriptionData* einen Post, wird dieser dem State (ein Array von Posts) vorangestellt. Der State enthält nun immer die jeweils aktuellen Posts und wird kontinuierlich aktualisiert.

Einmal live, bitte!

Natürlich ist der vollständige und lauffähige Code im GitHub-Projekt *[4]* zu finden. Das Projekt enthält dabei auch die Instruktionen, um sowohl Server als auch Client zu starten, um mit diesen zu experimentieren.

Die Erstellung von interaktiven Anwendungen, die Benutzer aktiv über Änderungen benachrichtigen, ist mit GraphQL Subscriptions überraschend einfach. Das hervorragende Zusammenspiel der Spring-Boot-Komponenten und eine durchdachte Integration machen die Entwicklung von solchen Backends einfach.

Ein Aspekt, der in diesem Artikel nicht betrachtet wurde, ist die Authentifizierung. Diese ist durchaus umfangreich und würde einen eigenen Artikel füllen.

Quellen

[1] *https://spring.io/projects/spring-graphql*
[2] *https://www.apollographql.com/docs/react/data/subscriptions/*
[3] *https://projectreactor.io/*
[4] *https://github.com/dxfrontiers/graphql-subscriptions*
[5] *https://www.apollographql.com/docs/react/*
[6] *https://www.graphql-code-generator.com/*

Heinz-Werner Haas
Digital Frontiers GmbH & Co. KG
heinz.haas@digitalfrontiers.de

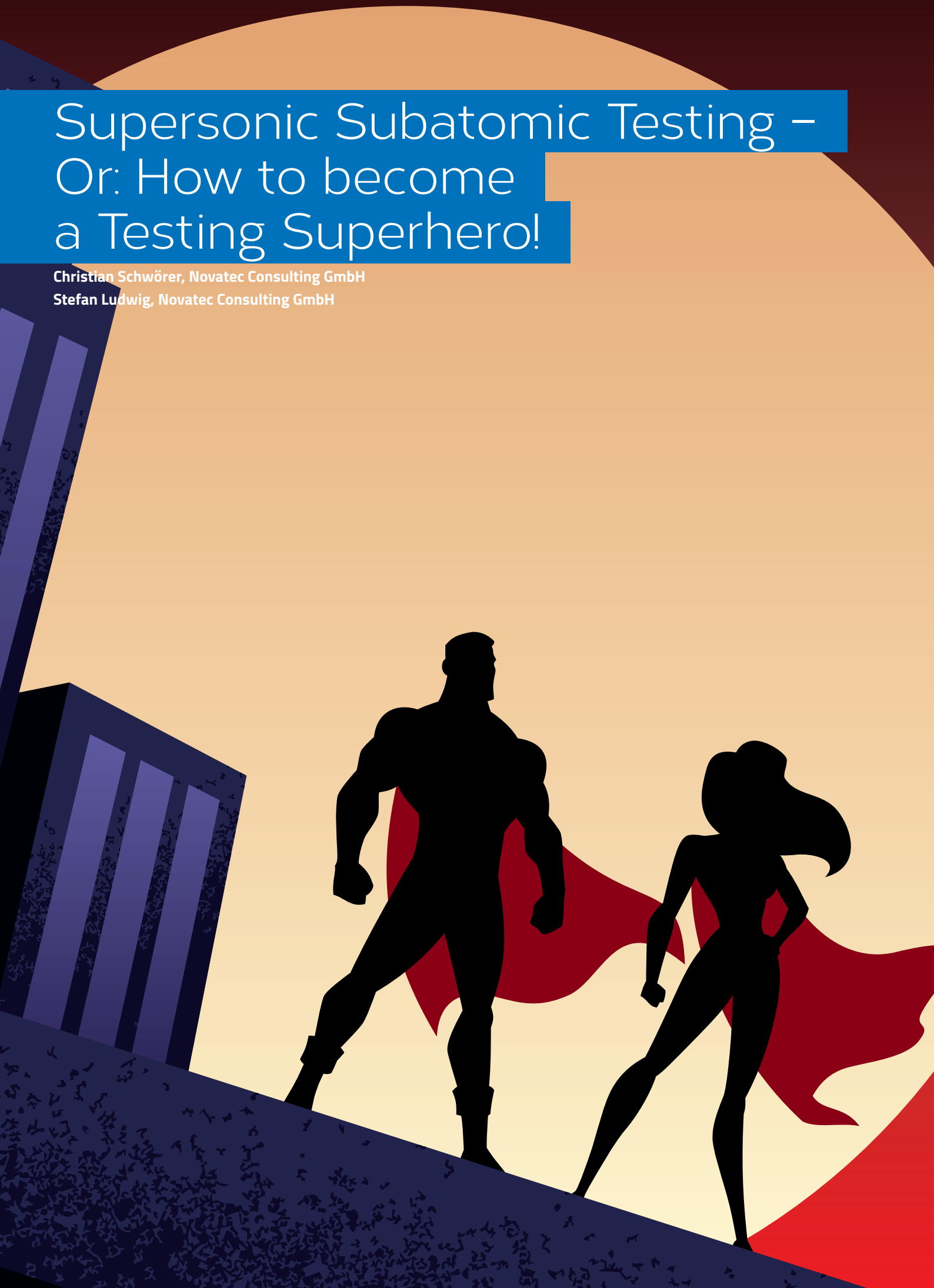
Heinz-Werner Haas ist als Consultant für Digital Frontiers tätig. Sein Schwerpunkt liegt auf agiler Softwareentwicklung vorwiegend im TypeScript-Umfeld. Eine besondere Begeisterung besteht im IT-Security-Bereich, speziell mit der Frage: „Wo bestehen Sicherheitslücken im System und wie können diese behoben werden?"

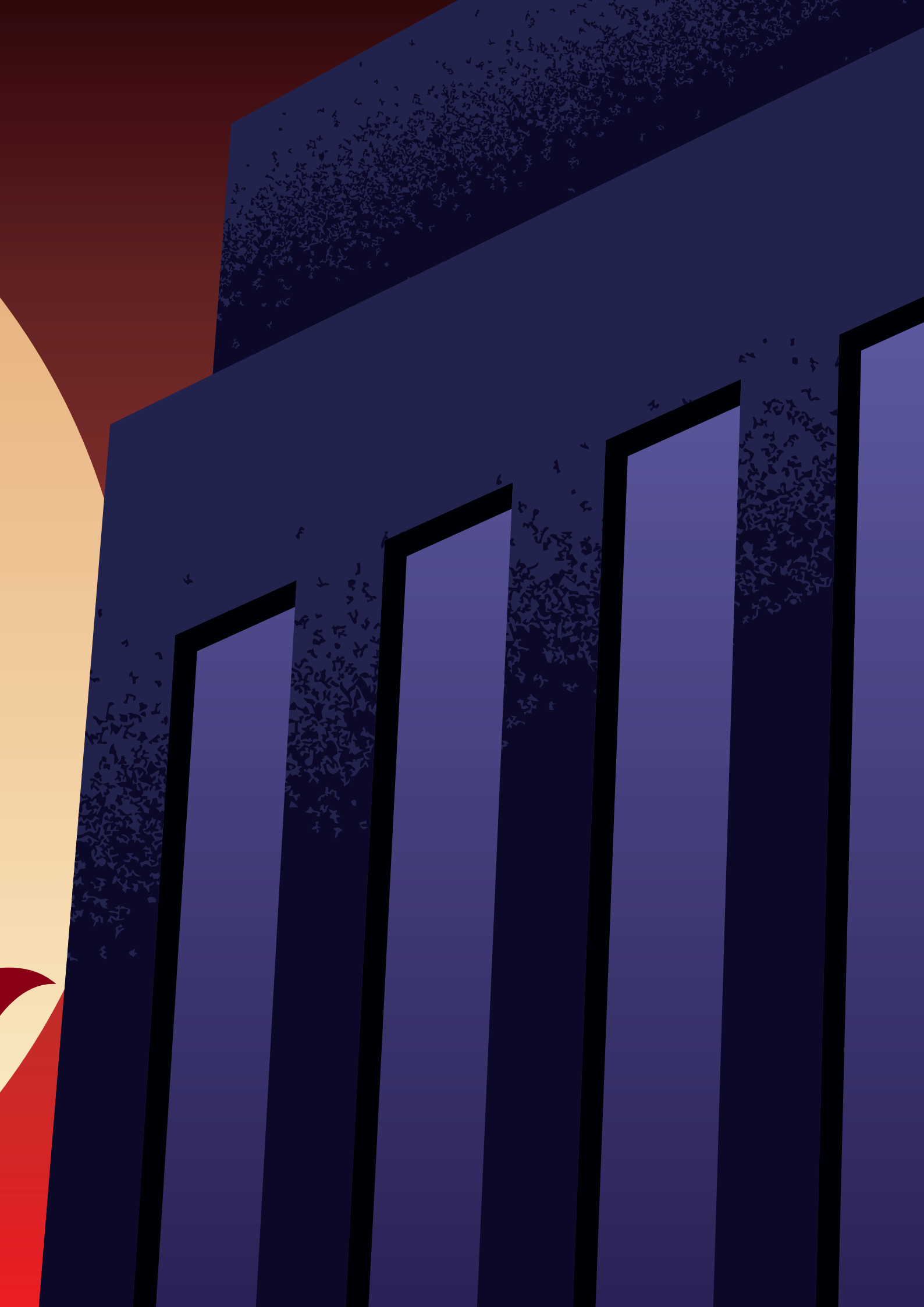
François Fernandes
Digital Frontiers GmbH & Co. KG
francois.fernandes@digitalfrontiers.de

François Fernandès ist Senior Solution Architect bei Digital Frontiers. Er verfügt über langjährige Erfahrung in den Bereichen Software-Architektur, Cloud-Architektur und im Java-Ökosystem. Spring und Spring Boot sind für ihn seit Langem treue Begleiter.

Supersonic Subatomic Testing – Or: How to become a Testing Superhero!

Christian Schwörer, Novatec Consulting GmbH
Stefan Ludwig, Novatec Consulting GmbH





„The only way to know how strong you are is to keep testing your limits."

(Jor El – Superman)

In der Softwareentwicklung ist nichts so beständig wie der Wandel: Die letzten Jahre waren geprägt vom Microservice-Trend und der damit verbundenen Abkehr von schwergewichtigen, monolithischen Anwendungen hin zu kleineren, Cloud-nativen Services.

Unabhängig von Hypes und Trends stellt die Sicherstellung der Softwarequalität eine zentrale Aufgabe für Entwickler/innen dar. Dabei spielen automatisierte Tests eine fundamentale Rolle. Richtig eingesetzt helfen sie, Mängel in der funktionalen Qualität zu identifizieren, ermöglichen frühzeitiges Feedback und stellen sicher, dass Modifikationen in bereits getesteten Teilen der Anwendung keine neuen Fehler („Regressionen") verursachen.

Allerdings ist Testautomatisierung kein Selbstläufer: Nur mit Teststrategie und -architektur kann sie gelingen. Zudem beeinflusst testbarer Code maßgeblich das Softwaredesign.

Im Folgenden wird zunächst Quarkus als JVM-Framework zur Erstellung von (Micro-) Services vorgestellt. Anschließend wird basierend auf einem einfachen Quarkus-/Kotlin-Microservice eine Teststrategie dargelegt und mit Beispielen konkretisiert.

Quarkus – „Supersonic Subatomic Java"

„Higher, further, faster, baby."

(Carol Danvers – Captain Marvel)

Viele JVM-Frameworks haben ihren Ursprung in einer Zeit, als es vorrangig um die Erstellung von monolithischen Anwendungen ging. Somit lag ihr Fokus meist bei der Optimierung von lang laufenden Prozessen – eventuelle Nachteile bei Startzeit und Speicherverbrauch fielen nicht so stark ins Gewicht.

Zuletzt haben jedoch das Microservice-Architekturpattern und die damit verbundene Nutzung von Cloud, Containern und Kubernetes an Bedeutung gewonnen. Und in einer Cloud-nativen Microservices-Landschaft, in der Instanzen schnell horizontal skaliert und neu verteilt werden, spielen Startzeiten und Speicherbedarf nun eine wesentlich größere Rolle. Daher werden zunehmend junge, schlanke JVM-Frameworks relevant, die von Grund auf und vollumfänglich für Cloud-Anwendungen konzipiert wurden.

Mit Quarkus konnte sich ein solches leichtgewichtiges und dennoch leistungsstarkes Framework etablieren. Es zeichnet sich durch eine „Container-first philosophy" aus, die es maßgeschneidert für den Einsatz mit Kubernetes macht.

Schnelle Startzeiten („Supersonic") und geringen Speicherbedarf („Subatomic") erreicht Quarkus, indem es die Framework-Logik bereits zur Compile- beziehungsweise Build-Zeit und nicht erst zur Start- und Laufzeit auflöst. Dadurch kann das Framework weitestgehend auf Reflection, Runtime Proxies und Dynamic Classloading verzichten. Folglich eignen sich Quarkus-Anwendungen nicht nur für die JVM, sondern auch für die GraalVM, was eine weitere drastische Reduzierung von Applikationsstartzeit und Speicherverbrauch erlaubt *[1]*.

Der nächste Abschnitt umfasst eine praxisnahe Einführung in einen rudimentärer Quarkus-Microservice. Ausführlichere Tutorials finden sich auf der Quarkus-Website *[2]*. Der Vollständigkeit halber sei noch darauf hingewiesen, dass es neben Quarkus eine ganze Reihe anderer

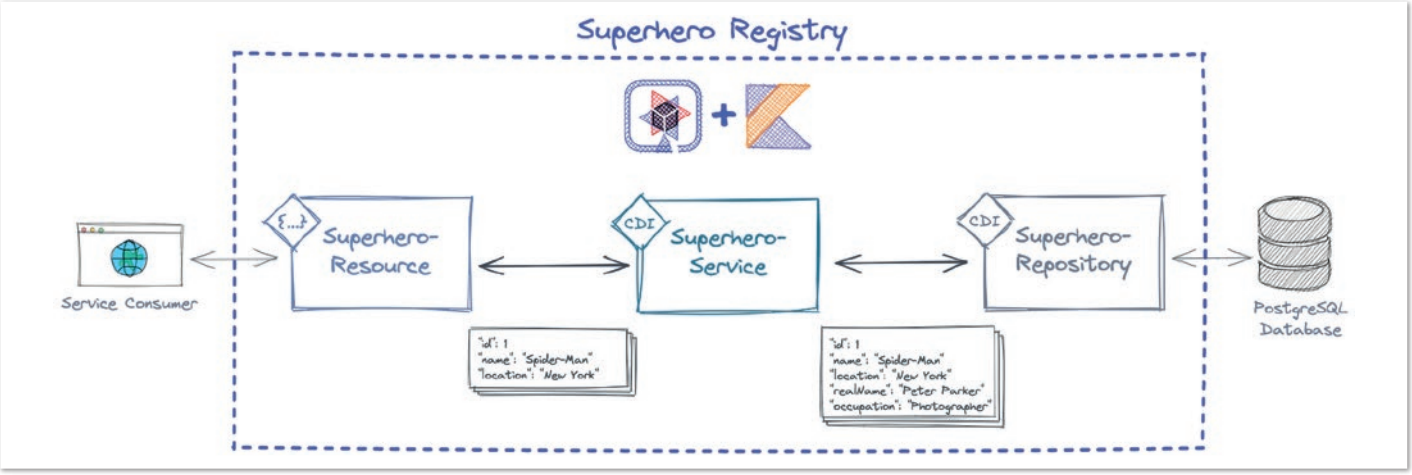


Abbildung 1: Quarkus-/Kotlin-Beispielanwendung „Superhero Registry"

Frameworks wie Micronaut, Ktor oder Helidon gibt, die sich ebenfalls ideal für die Cloud-native Anwendungsentwicklung eignen *[3]*.

Quarkus-Beispielanwendung „Superhero Registry"

Abbildung 1 zeigt die im Folgenden erläuterte Quarkus-Beispielanwendung „Superhero Registry". Als Programmiersprache wurde dabei Kotlin gewählt. Die Umsetzung könnte aber natürlich genauso mit Java erfolgen. Der zugehörige Quellcode findet sich in GitHub *[4]*.

Aus Sicht eines Consumers bietet der Microservice Schnittstellen zum Anlegen und Auslesen von Superhelden. Das Anlegen erfolgt mittels POST-Request, wie in *Listing 1* gezeigt.

```
{
  "name": "Spider-Man",
  "location": "New York",
  "realName": "Peter Parker",
  "occupation": "Freelance Photographer"
}
```

Listing 1: Payload zum Anlegen eines Superhelden über den Endpunkt http://novatec.info/superheroes

Die Liste aller Superhelden wird durch ein GET gegen den Endpunkt *http://novatec.info/superheroes* ermittelt *(siehe Listing 2)*. Ein einzelner Superheld kann durch Hinzufügen der ID als Pfadparameter ausgelesen werden, also zum Beispiel *http://novatec.info/superheroes/1*. Anhand der unterschiedlichen Payloads von POST- und GET-Request wird bereits ersichtlich, dass nicht alle gespeicherten Daten eines Superhelden auch wieder ausgegeben werden: Der Service beinhaltet eine „Anonymisierungslogik", die sicherstellt, dass die wahre Identität von beispielsweise Spider-Man (nämlich „Peter Parker") sowie sein Beruf („Freelance Photographer") vor dem Aufrufer verborgen bleiben.

Aufbau der Quarkus-Beispielanwendung

Der Quarkus-Microservice besteht aus drei Komponenten: der *SuperheroResource*, die das REST-API bereitstellt, dem *SuperheroService* mit der Anwendungslogik und dem *SuperheroRepository*, das den Zugriff auf die relationale Datenbank kapselt.

Die Realisierung des in der *SuperheroResource* implementierten REST-API zeigt *Listing 3*.

Mittels @Path wird der HTTP-Pfad festgelegt. Die Komponente *SuperheroService* wird per Constructor Dependency Injection „gewired" – dazu später mehr.

Die Funktion *getAllSuperheroes()* delegiert die Ermittlung aller Superhelden an den *SuperheroService*. Dessen Antwort wird zu JSON serialisiert und in eine HTTP-Response mit dem Status 200 (OK) verpackt. Wird jedoch eine ID mitgegeben, greift die Funktion *getSingleSuperhero()*, die den Pfadparameter extrahiert und an den

```
[
  {
    "id": 1,
    "name": "Spider-Man",
    "location": "New York"
  },
  {
    "id": 2,
    "name": "Black Widow",
    "location": "Volgograd"
  },
  {
    "id": 3,
    "name": "Black Panther",
    "location": "Wakanda"
  }
]
```

Listing 2: Ergebnis des GET-Request zum Ermitteln aller (anonymisierter) Superhelden

```
@Path("/superheroes")
class SuperheroResource(
    private val service: SuperheroService
) {
    @GET
    @Path("/")
    fun getAllSuperheroes(): Response =
        Response.ok(service.findSuperheroes()).build()

    @GET
    @Path("/{id}")
    fun getSingleSuperhero(@PathParam id: Long): Response =
        service.findSuperhero(id)
            ?.let { Response.ok(it).build() }
            ?: Response.status(Response.Status.NOT_FOUND).build()

    @POST
    @Path("/")
    fun addSuperhero(superhero: Superhero, @Context uriInfo: UriInfo): Response {
        val id = service.addSuperhero(superhero).id
        val uri = uriInfo.absolutePathBuilder.path(id.toString()).build()
        return Response.created(uri).build()
    }
}
```

Listing 3: Implementierung der SuperheroResource

```kotlin
@ApplicationScoped
class SuperheroService(
      private val repository: SuperheroRepository
) {

      fun findSuperhero(id: Long) =
         repository.findById(id)?.anonymize()

      fun findSuperheroes(): List<AnonymizedSuperhero> =
         repository.listAll().map { it.anonymize() }

      @Transactional
      fun addSuperhero(superhero: Superhero): AnonymizedSuperhero {
         repository.persist(superhero)
         return superhero.anonymize()
      }

      private fun Superhero.anonymize() = AnonymizedSuperhero(
         id = this.id!!,
         name = this.name,
         location = this.location
      )
}

data class AnonymizedSuperhero(
      val id: Long,
      val name: String,
      val location: String
)
```

Listing 4: Realisierung Anwendungslogik im SuperheroService

Service übergibt. Kann kein Superheld zu der ID ermittelt werden, wird der HTTP-Status 404 (NOT_FOUND) zurückgegeben. Das Anlegen eines neuen Superhelden erfolgt über die Funktion *addSuperhero()*, die aus der JSON-Payload des POST-Request eine Entität erzeugt und an den Service übergibt. Anschließend wird der HTTP-Status 201 (CREATED) mit dem URI (Uniform Resource Identifier) der angelegten Resource im Location-Header zurückgegeben.

Die Anwendungslogik befindet sich im in *Listing 4* dargestellten *SuperheroService*. Diese Komponente wird mit *@ApplicationScoped* als CDI-Bean gekennzeichnet. Daher kann sie in der *SuperheroResource* per Dependency Injection „gewired" werden. Der *SuperheroService* seinerseits bekommt das *SuperheroRepository* injiziert.

Die einzelnen Funktionen delegieren im Wesentlichen an das Repository. Erwähnenswert ist die Kotlin Extension Function *anonymize()*: Sie beinhaltet die „Anonymisierungslogik", die einen *Superhero* zu einem *AnonymizedSuperhero* – ohne die Felder *realName* und *occupation* – umwandelt.

Zu guter Letzt zeigt *Listing 5* das *SuperheroRepository*. Durch die Implementierung des Quarkus-Interface *PanacheRepository* stehen per Konvention alle benötigten CRUD-Operationen wie zum Beispiel *listAll()*, *findById()* oder *persist()* zur Verfügung. *[5]*

Die Entität *Superhero* ist als Kotlin Data Class realisiert. In Java wäre dies entsprechend ein POJO mit Gettern, Settern, `hashCode()`, `equals()` etc. oder ein seit Java 14 verfügbarer *Record*.

Damit sind alle Komponenten für unseren einfachen Microservice vorhanden. Im Folgenden soll nun vorgestellt werden, wie diese Teile sinnvoll getestet werden können.

Arten von Softwaretests und deren Umsetzung mit Quarkus

> **„The will is everything! The will to act."**
>
> (Ra's al Ghul – Batman)

Nachdem im vorigen Kapitel der Quarkus-/Kotlin-Microservice vorgestellt wurde, geht's nun ans Eingemachte: Welche Arten von Softwaretests gibt es, um die Komponenten zu testen? Und wie konkret lassen sich diese Tests realisieren?

Die meisten Entwickler/innen kennen vermutlich die Testpyramide: Sie ist ein vereinfachtes Modell, das beschreibt, wie sich Softwaretests auf verschiedenen Granularitätsebenen gruppieren lassen und wie sich die Tests mengenmäßig auf die verschiedenen Gruppen verteilen sollten *[6]*.

```kotlin
@ApplicationScoped
class SuperheroRepository : PanacheRepository<Superhero>

@Entity
data class Superhero(
      @Id
      @GeneratedValue
      var id: Long? = null,
      var name: String,
      var location: String,
      var realName: String,
      var occupation: String
)
```

Listing 5: Entität Superhero und SuperheroRepository für den Zugriff auf die Datenbank

Basierend auf diesem eher allgemeinen Modell lässt sich eine auf die Testautomatisierung von Backend-(Micro-)Services ausgerichtete Pyramide wie in *Abbildung 2* gezeigt ableiten. Die einzelnen Ebenen sowie ihre Umsetzung werden im Folgenden genauer vorgestellt.

Functional Unit Tests

Unit Tests sollen sicherstellen, dass der eigene Code genau das tut, was er soll. Da alles andere nachgebildet („gemockt") wird, sind sie extrem schnell und können in kürzester Zeit zu Tausenden ausgeführt werden. Das macht sie zu idealen Tests, um schnelles Feedback zu den Kernkomponenten, das heißt zur Geschäftslogik, zu erhalten.

Relevante Methoden und Technologien:

- Zur Test-Ausführung haben sich Testframeworks wie *JUnit [7]* oder Spock *[8]* etabliert.
- Um einen vollständig isolierten Test zu erreichen, werden Abhängigkeiten mithilfe von Bibliotheken wie *Mockito [9]* und *MockK [10]* durch Mock-Objekte ersetzt.
- Für erweiterte Überprüfungen des erwarteten Ergebnisses stehen Assertion-Libraries wie *AssertJ [11]* und *Strikt [12]* zur Verfügung.

Bei der Beispielanwendung kann die Anonymisierungslogik des *SuperheroService* sehr gut mit Unittests validiert werden *(siehe Abbildung 3)*. *Listing 6* zeigt den zugehörigen Test.

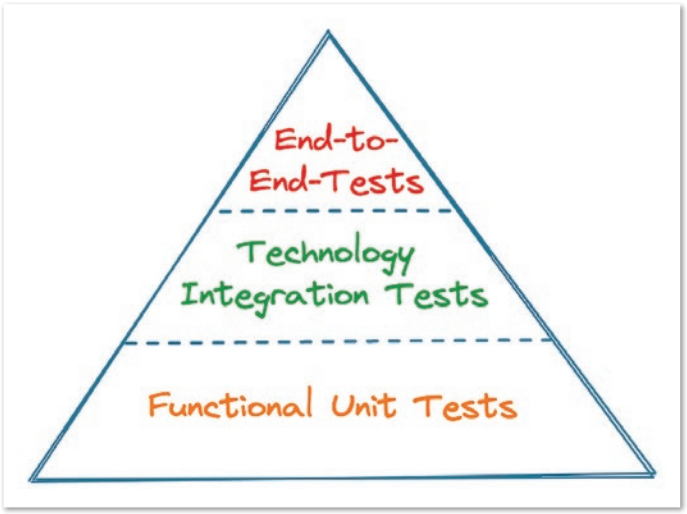


Abbildung 2: Testpyramide für Backend-(Micro-)Services

Da dieser isolierte Test ausschließlich die Service-eigene Funktionalität prüfen soll, wird mit MockK ein Mock-Objekt für das *SuperheroRepository* erzeugt. Danach wird die zu testende Klasse instanziiert. Dazu wird das Mock-Objekt als Konstruktor-Parameter übergeben.

Der eigentliche JUnit-Test folgt dem AAA-Pattern – Arrange-Act-Assert *[13]*. Das heißt, zunächst wird das Verhalten des Mock-Objekts arrangiert: Wann immer die Repository-Funktion *findBy-*

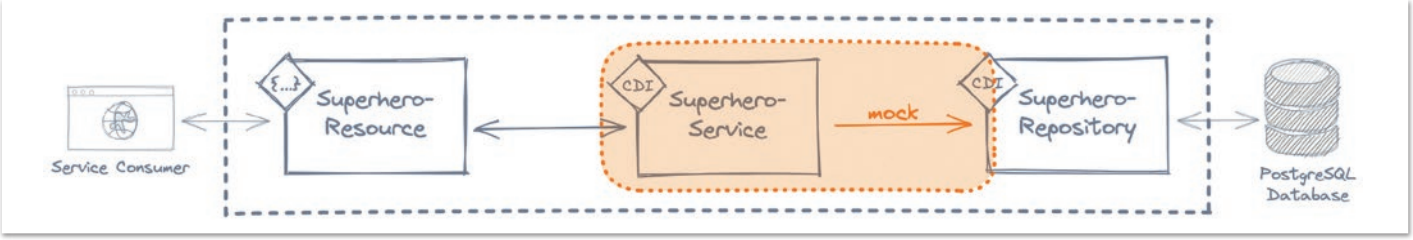


Abbildung 3: Scope des Functional Unittests

```
class SuperheroServiceUnitTests {

    private val repository: SuperheroRepository = mockk()

    private val testee = SuperheroService(repository)

    @Test
    fun `get and anonymize superhero by id`() {
        every { repository.findById(42) } returns
            Superhero(
                id = 42,
                name = "Batman",
                location = "Gotham City",
                realName = "Bruce Wayne",
                occupation = "Businessman"
            )

        val result = testee.findSuperhero(42)

        expectThat(result).isEqualTo(
            AnonymizedSuperhero(
                id = 42,
                name = "Batman",
                location = "Gotham City"
            )
        )
    }
}
```

Listing 6: Functional Unittest zur Validierung der Geschäftslogik des SuperheroService

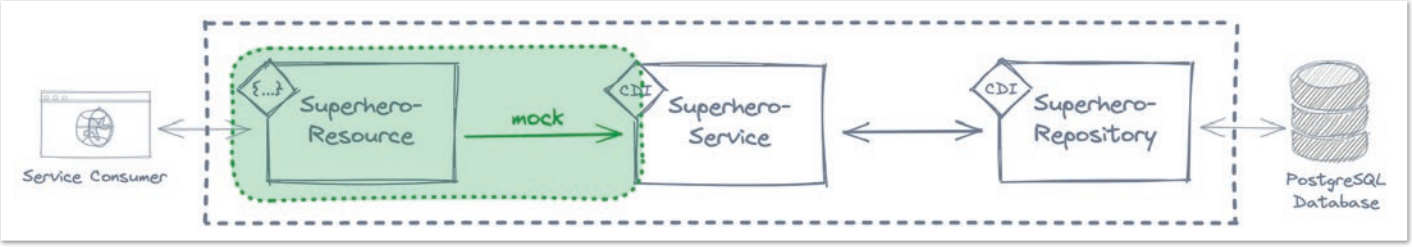


Abbildung 4: Scope des Technology-Integration-Tests

Id() mit dem Parameter „42" aufgerufen wird, soll eine bestimmte Superhero-Entity zurückgegeben werden. Anschließend folgt der Act-Teil, das heißt der eigentliche Aufruf der zu testenden Funktion. Abschließend wird das Ergebnis im Assert-Block verifiziert. Weitere Tests, unter anderem für die Funktionen *findSuperheroes()* und *addSuperhero()*, würden analog erfolgen.

In diesem Beispiel wird deutlich, dass für das Testen von gut isolierter Geschäftslogik einfache und schnelle Unittests üblicherweise ausreichend sind.

Technology-Integration-Tests

Tests in dieser Gruppe werden eingesetzt, um den Code zu überprüfen, der für die Verwendung einer bestimmten Technologie geschrieben wurde. Darunter fallen zum Beispiel:

- HTTP-Endpunkte (@Path, @GET, @POST, @PUT, @DELETE, @PATCH)
- Caching (@CacheResult)
- Transaktionen (@Transactional)
- Asynchrones Messaging (@Incoming, @Outgoing)
- Event Handling (@ConsumeEvent)

```
@QuarkusTest
class SuperheroResourceIntegrationTests {

    @InjectMock
    private lateinit var service: SuperheroService

    @Test
    fun `GET superhero by id`() {
        every { service.findSuperhero(42L) } returns
            AnonymizedSuperhero(
                id = 42L,
                name = "Batman",
                location = "Gotham City"
            )

        val expectedJson = """
            {
                "id": 42,
                "name": "Batman",
                "location": "Gotham City"
            }
            """

        given()
            .`when`()["/superheroes/42"]
            .then()
            .statusCode(200)
            .contentType(APPLICATION_JSON)
            .body(jsonEqualTo(expectedJson))
    }
}
```

Listing 7: Technology-Integration-Tests zur Prüfung des REST-API

- Web security configuration
- Method-level security (@RolesAllowed, @DenyAll, @PermitAll)
- Datenbankzugriffe (PanacheRepository)
- HTTP-Clients (@RegisterRestClient, @RestClient)

Das Ziel dieser Integrationstests ist jedoch nicht, die genutzte Technologie zu validieren. Vielmehr geht es darum zu verifizieren, ob die Technologie richtig verwendet wird. Deshalb ist ein Bootstrapping der genutzten Technologie erforderlich, wodurch die Tests zwangsläufig langsamer als Unittests sind.

Relevante Methoden und Technologien:

- Als Simulator von externen HTTP-Diensten hat sich *Wiremock [14]* bewährt.
- Zur leichtgewichtigen Bereitstellung von Datenbankinstanzen oder Message Queues per Docker-Container steht mit *Testcontainers [15]* ein leistungsstarkes Tool zur Verfügung.
- Zum Aufruf und zur Validierung von HTTP-Endpunkten bieten sich DSLs wie REST-*assured [16]* an.

In der Superhero Registry kann beispielsweise das REST-API, das heißt die *SuperheroResource*, mit Technology-Integration-Tests geprüft werden. *Abbildung 4* und *Listing 7* zeigen den zugehörigen Test.

Die Annotation *@QuarkusTest* bewirkt, dass die Quarkus-Anwendung zusammen mit dem JUnit-Testframework gestartet wird. Über *@InjectMock* wird ein Mock-Objekt erzeugt und in die Anwendung injiziert. Das heißt, die eigentliche CDI-Bean *SuperheroService* wird gewissermaßen durch den im Test definierten Mock ersetzt.

Im konkreten JUnit-Test wird nun wieder zunächst das gewünschte Mock-Verhalten beschrieben. Anschließend wird mithilfe von REST-assured ein HTTP-Aufruf gegen den Endpunkt *superheroes/42* ausgeführt und geprüft, ob die Antwort den gewünschten Statuscode 200 (OK) sowie die richtige JSON-Payload hat. Auch die anderen Funktionen der *SuperheroResource* sollten durch entsprechende Tests abgedeckt werden.

In diesem Beispiel wird ersichtlich, dass die Anwendung gestartet werden muss, da das grundsätzliche Bereitstellen der HTTP-Endpunkte eine Quarkus-Funktionalität ist. Somit reicht ein einfacher Unittest hier nicht mehr aus. Mit einem Technology-Integration-Test kann jedoch verifiziert werden, dass das Framework richtig genutzt wird – etwa, ob die REST-Endpunkte richtig konfiguriert sind und die JSON-Serialisierung korrekt funktioniert. Da sich die *SuperheroResource* aber rein auf HTTP-Funktionalität beschränkt und sämtliche Logik an den Service delegiert, reicht hier der Integrationstest aus und muss nicht durch einen weiteren Unittest untermauert werden.

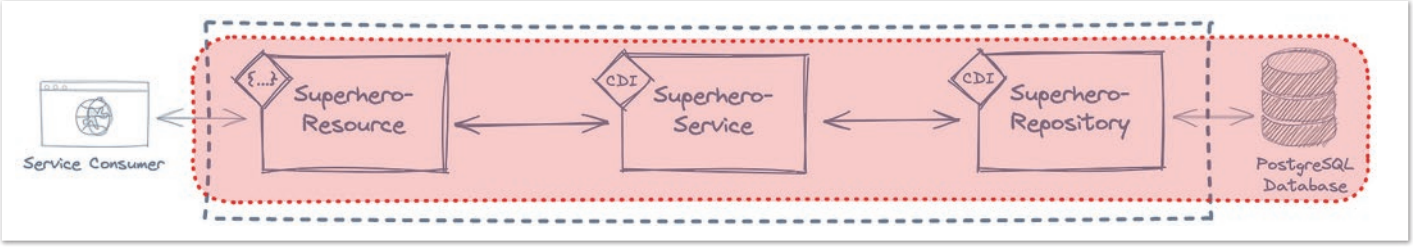


Abbildung 5: Scope des End-to-End-Tests

End-to-End-Tests

Wenn nun mittels Unit- und Integrationstests alle Komponenten so weit wie möglich (isoliert) getestet wurden, kommen abschließend End-to-End-Tests zum Einsatz.

End-to-End-Tests werden aus der Consumer-Perspektive geschrieben. Dabei ist besonders entscheidend, welche Möglichkeiten es gibt, mit der zu testenden Anwendung zu interagieren. Für Backend-Anwendungen ist diese Interaktionsmöglichkeit normalerweise das API. Etwaige Abhängigkeiten, wie andere Dienste oder Datenbanken, werden entweder simuliert oder durch Testinstanzen ersetzt.

Oftmals werden End-to-End-Tests zur Realisierung von Smoke-Tests genutzt: Dazu werden einige „Happy-Path"-Szenarien pro Endpunkt ausgeführt, um zu sehen, ob der gesamte Kontrollfluss von der Anfrage bis zur Antwort funktioniert. Im Grunde genommen wird geprüft, ob alles, was bereits isoliert getestet wurde, auch korrekt zusammenarbeitet *(siehe Abbildung 5)*.

In der vorgestellten Quarkus-Beispielanwendung nutzen die End-to-End-Tests die HTTP-Endpunkte der *SuperheroResource*. Die Datenbank des *SuperheroRepository* wird durch eine Testcontainer-Instanz ersetzt.

Als Happy-Path-Test wäre denkbar, dass ein Superheld per HTTP-Aufruf angelegt wird und anschließend direkt versucht wird, den so gespeicherten Superhelden wieder auszulesen. Dadurch wäre ein typischer Kontrollfluss aus Consumer-Sicht abgedeckt. *Listing 8* zeigt den entsprechenden End-to-End-Test.

Auch hier wird anhand der Annotation *@QuarkusTest* die Applikation gestartet. Im Test selbst wird zunächst mit *REST-assured* ein POST-Request ausgeführt, der einen Superhelden anlegt. Aus der Antwort wird der Wert des Location-Headers ausgelesen, der den URI der erzeugten Resource enthält.

Dieser URI wird im darauffolgenden GET-Request benutzt, um den soeben gespeicherten Superhelden wieder auszulesen. Abschließend wird die JSON-Payload des zweiten Request verifiziert.

Die Postgres-Datenbank wird, wie bereits erwähnt, durch eine Testcontainer-Instanz ersetzt. Allerdings findet sich in *Listing 8* keinerlei explizite Angabe dazu. Vielmehr zeigt sich hier eine der Stärken beim Testing mit Quarkus: Fehlen bei der Test-Konfiguration der Datasource, wie in *Listing 9* dargestellt, URL, User und Passwort, wird per Konvention automatisch eine entsprechende Testcontainer-Instanz gestartet. Somit muss zur Ausführung des End-to-End-Tests lediglich ein Docker-Daemon verfügbar sein.

```
@QuarkusTest
class SuperheroRegistryEnd2EndTests {

    @Test
    fun `create and retrieve superhero`() {

        val inputJson = """
            {
                "name": "Batman",
                "location": "Gotham City",
                "realName": "Bruce Wayne",
                "occupation": "Businessman"
            }
            """

        val locationUri = given()
            .contentType(APPLICATION_JSON)
            .body(inputJson)
            .`when`().post("superheroes")
            .then()
            .statusCode(201)
            .extract()
            .header(LOCATION)

        val outputJson = """
            {
                "id": ${locationUri.extractId()},
                "name": "Batman",
                "location": "Gotham City"
            }
            """

        given()
            .`when`()[locationUri]
            .then()
            .statusCode(200)
            .contentType(APPLICATION_JSON)
            .body(jsonEqualTo(outputJson))
    }

    private fun String.extractId() =
        substring(lastIndexOf('/') + 1);
}
```

Listing 8: End-to-End-Test zur Validierung, ob ein Superhero angelegt und anschließend wieder ausgelesen werden kann

```
quarkus:
  datasource:
    db-kind: postgresql
    username:
    password:
    jdbc:
      driver: org.postgresql.Driver
      url:
  hibernate-orm:
    database:
      generation: drop-and-create
```

Listing 9: application.yml zur Test-Konfiguration der Postgres-Datasource

Vergleicht man nun die Komplexität des – in diesem Fall noch recht trivialen – End-to-End-Tests mit der eines Unit- oder Integrationstests, wird ersichtlich, warum sich diese Testgruppe in der Regel auf die Happy Paths konzentriert. Alle Edge Cases und spezifischen Fehlerfälle sollten idealerweise bereits durch die unteren Stufen der Testpyramide aus *Abbildung 2* abgedeckt sein. Unter anderem auch deshalb, da ein fehlgeschlagener Unit-Test ganz klar einer fehlerhaften Komponente zuordenbar ist. Bei einem fehlgeschlagenen End-to-End-Test kann der Fehler dagegen in potenziell jeder vom Kontrollfluss betroffenen Komponente aufgetreten sein und ist deutlich schwieriger einzugrenzen.

Fazit

> **„Remember, with great power comes great responsibility."**
>
> (Uncle Ben – Spider-Man)

Mit Quarkus steht ein ausgereiftes Framework zur Cloud-nativen Anwendungsentwicklung bereit. Es wird kontinuierlich erweitert und erfreut sich einer stetig wachsenden Community. Alle gängigen Testing-Tools und -Bibliotheken werden unterstützt. Besonders hervorzuheben ist die integrierte Unterstützung von Testcontainers – sozusagen „Testing with Batteries included".

Wie zuvor erwähnt, benötigt Testcontainers zur Ausführung einen Docker-Daemon. Es ist natürlich möglich, dass dieser nicht verfügbar ist, da er zum Beispiel auf einer CI-Umgebung nicht gestartet werden darf oder die Ressourcen-Limitierung der lokalen Maschine es nicht zulassen. Für diesen Fall lohnt sich ein Blick auf Testcontainers Cloud. *[17]* Diese Plattform ermöglicht es, Testcontainers remote und on demand auszuführen, sodass keine lokale Docker-Installation erforderlich ist.

Eine abschließende Anmerkung noch zum Vorgehen: In dem Beispiel wurden – aus Gründen der Nachvollziehbarkeit – zunächst alle Anwendungskomponenten erstellt und erst danach die Tests ergänzt. Erfahrungsgemäß gestaltet es sich jedoch zielführender, testgetrieben vorzugehen. Das heißt, mit den Tests zu beginnen und erst dann die dazu passende Anwendungslogik umzusetzen.

Schlussendlich lässt sich festhalten, dass Testing und Testautomatisierung essenzielle Bestandteile der Anwendungsentwicklung sind, ganz gleich, welches Framework, welche Sprache oder welche Bibliothek verwendet werden. Dementsprechend gilt es, sich als Team frühzeitig mit der Teststrategie auseinanderzusetzen. Neben technischen Aspekten kann diese auch von nicht-funktionalen Faktoren beeinflusst werden. Demzufolge sollte eine ganzheitliche Strategie zusätzlich zu den vorgestellten Testgruppen auch (möglichst automatisierte) Last-, Performance- und Security-Tests umfassen.

Quellen

[1] *https://quarkus.io/container-first/*
[2] *https://quarkus.io/get-started*
[3] Christian Schwörer, Carlos Barragan (2020): Alle neune! – Wichtige JVM-Microframeworks im Überblick. JavaMagazin Ausgabe 6.2020
[4] *https://github.com/csh0711*
[5] *https://quarkus.io/guides/hibernate-orm-panache*
[6] *https://martinfowler.com/articles/practical-test-pyramid.html*
[7] *https://junit.org/junit5*
[8] *https://spockframework.org*
[9] *https://site.mockito.org*
[10] *https://mockk.io*
[11] *https://assertj.github.io/doc*
[12] *https://strikt.io*
[13] Vladimir Khorikov (2020): Unit Testing Principles, Practices, and Patterns. Manning Publications, S. 42
[14] *https://wiremock.org*
[15] *https://testcontainers.org*
[16] *https://rest-assured.io*
[17] *https://www.testcontainers.cloud*

Christian Schwörer
Novatec Consulting GmbH
christian.schwoerer@novatec-gmbh.de

Christian Schwörer unterstützt die Kunden der Novatec Consulting GmbH bei der digitalen Transformation hin zu verteilten, cloudbasierten Microservice-Architekturen. Dabei setzt er häufig und gerne Spring Boot/Cloud ein, zunehmend aber auch andere Frameworks wie Quarkus, Micronaut oder Ktor. Er ist der Überzeugung, dass sich die praktische Auseinandersetzung mit dem Thema für alle Architekt:innen und Entwickler:innen lohnt – auch wenn Microservices sicher nicht die „Silver Bullet" für alle Probleme des Software Engineerings sind und daher gezielt eingesetzt werden sollten.

Stefan Ludwig
Novatec Consulting GmbH
stefan.ludwig@novatec-gmbh.de

In seiner Tätigkeit für die Novatec Consulting GmbH unterstützt Stefan Ludwig Kunden bei der Umsetzung von Cloud-nativen Softwareprojekten. Dabei liegt sein Fokus auf effizienter Testautomatisierung und testbarem Software Design. Er ist der Überzeugung, dass automatisierte Tests ein elementarer Bestandteil jedweder Programmiertätigkeit ist.

Von Caching und Wahrscheinlichkeiten

Bernd Stübinger, Java User Group Ingolstadt e.V.

Caching ist wohl ein zentraler Bestandteil der meisten modernen Architekturen. Besonders in einem verteilten System lauern allerdings unter hoher Last potenzielle Fallstricke, die nicht immer offensichtlich sind. Dieser Artikel zeigt auf, wie man mithilfe einer ebenso einfachen wie eleganten Methode seine Abhängigkeiten vor einer Cache-Stampede schützen kann.

Klassischerweise nutzt man Caching zur Performancesteigerung, indem man etwa Ergebnisse komplexer Berechnungen zwischenspeichert oder Zugriffe auf langsame Komponenten minimiert. In einem verteilten System kommen allerdings noch zwei weitere wichtige Gründe für Caching dazu. Oft verbirgt sich hinter all den glitzernden neuen Microservices eine nicht mehr ganz so funkelnde Datenbank oder ein kaum skalierbares Legacy-Backend, die man vor zu vielen gleichzeitigen Requests abschirmen muss. Möglicherweise sprechen die Services auch mit einem kostenpflichtigen externen API und man möchte nicht den Datenabruf für jede einzelne Service-Instanz erneut bezahlen. Caching kann aber auch im Rahmen der Fehlertoleranz, auch bekannt als Resilience, unterstützen: Ist ein entferntes System nicht erreichbar, so kann man einen Request vielleicht mit zwischengespeicherten Daten aus dem Cache trotzdem noch zufriedenstellend beantworten.

Lebensdauer von Daten

Das hat allerdings Grenzen: In den seltensten Fällen sind Daten unbegrenzt lange verwendbar. Gewisse Stammdaten sind vielleicht auch eine Woche später noch gut genug, die Nachrichten vom Vortag sind bereits sprichwörtlich nicht mehr relevant und Dinge wie Preise in einem Onlineshop möchte man noch nicht einmal für zehn Minuten cachen. Um eine Überalterung der gespeicherten Daten zu vermeiden, haben Einträge üblicherweise eine TTL (Time to Live), die bestimmt, wie lange die Daten als aktuell betrachtet und verwendet werden dürfen. Wie lange das ist, ist dabei typischerweise eine fachliche Frage und hängt von diversen Faktoren ab. Man wird also in der Praxis sehr unterschiedliche TTLs vorfinden.

Hier beginnt auch die Problematik: Sobald die TTL für einen Eintrag abgelaufen ist, muss dieser erneut vom Quellsystem geladen werden.

Im einfachsten Fall sieht eine Cache-Implementierung aus wie in *Listing 1*: Ein Eintrag wird angefragt und direkt zurückgegeben, falls er im Cache ist. Sind keine Daten dafür im Cache (etwa wegen Überalterung), werden sie neu berechnet und im Cache abgelegt.

```
public Value fetch(Key key) {
  Value cachedValue = cache.getValue(key);
  if (cachedValue == null) {
     cachedValue = recompute(key);
     cache.setValue(key, cachedValue);
  }
  return cachedValue;
}
```

Listing 1: Eine einfache Cache-Implementierung

Cache-Stampede

Problematisch an dieser einfachen Implementierung ist, dass keine Form der Abstimmung oder Synchronisation stattfindet. Nehmen wir als Beispiel eine Anwendung, die 1.000 Requests pro Sekunde verarbeitet und dabei einen einzelnen Eintrag mit einer TTL von vier Stunden benötigt. Dieser wird im Cache abgelegt, um das dahinter liegende Altsystem zu schützen. Vier Stunden lang geht alles gut.

Eine Sekunde später stellen 1.000 Requests gleichzeitig fest, dass der Eintrag veraltet ist, und wollen ihn neu berechnen. Das Altsystem bekommt ohne Vorwarnung 1.000 gleichzeitige Requests ab. Nehmen wir weiterhin an, dass die Neuberechnung zehn Sekunden dauert, dann treffen während dieser Zeit weitere 10.000 Requests auf die veralteten Daten im Cache und stoßen Anfragen an das Altsystem an. Durch die unerwartet hohe Last dauern Anfragen länger oder brechen komplett ab, was zusätzliche Requests während der Wartezeit oder durch automatische Retries verursacht. Im schlimmsten Fall fällt das Legacy-System aus und steht dadurch auch anderen Konsumenten nicht mehr zur Verfügung, Netzwerkressourcen werden ausgelastet, am Ende stellt die gesamte Anwendung den Betrieb ein. Dieses Verhalten ist als Cache-Stampede bekannt. Und eine solche Cache-Stampede kommt selbst bei etablierten Tech-Playern schon mal vor *[1]*.

Was tun?

Eine naheliegende Abhilfe bietet Locking – der erste Zugriff auf einen fehlenden Eintrag sperrt diesen während der Zeit der Neuberechnung. Alle anderen warten auf die Aktualisierung und arbeiten weiter, sobald die Daten verfügbar sind. Der klassische Fall dafür sind `synchronized`-Methoden, eine etwas elegantere Implementierung könnte beispielsweise aussehen wie *Listing 2*. In diesem Fall stellt die `ConcurrentHashMap` für uns sicher, dass die Neuberechnung für jeden Eintrag nur genau ein einziges Mal stattfindet.

Der Nachteil am Locking ist natürlich, dass Ressourcen blockiert werden – in unserem Beispiel würden nach zehn Sekunden 9.999 Requests auf den einen warten, der den Eintrag neu berechnet. Zudem muss man sich Gedanken um Deadlocks oder Starvation machen und gerade in einem verteilten System ist Locking eine durchaus nicht-triviale Aufgabe: Die Verwaltung des Locks erfordert zusätzliche Kommunikation, das Netzwerk kann (teilweise) ausfallen, man hat mit Latenzen zu kämpfen und… was passiert eigentlich, wenn genau die Instanz wegbricht, die gerade den Lock hält? Und selbst wenn man all das technisch lösen kann: Will man denn, dass sich 100 Service-Instanzen am Ende potenziell ständig gegenseitig auf den Füßen stehen? Typischerweise möchte man in einem verteilten System ja möglichst wenig Kopplung und Abhängigkeiten – also eher das Gegenteil eines zentralen Locks.

```
private Map<Key, CompletableFuture<Value>> pendingMap
  = new ConcurrentHashMap<>();

public Value fetch(Key key) throws Exception {
  Value cachedValue = cache.getValue(key);
  if (cachedValue != null) {
     return cachedValue;
  }
  CompletableFuture<Value> deferred =
     pendingMap.computeIfAbsent(key,
        k -> CompletableFuture.supplyAsync(() -> {
           Value value = recompute(k);
           cache.setValue(k, value);
           pendingMap.remove(k);
           return value;
        }));
  return deferred.get();
}
```

Listing 2: Eine Cache-Implementierung mit Locking

Externe Aktualisierung

Eine andere Möglichkeit ist, den Cache extern zu aktualisieren. Dabei findet beim Zugriff selbst keine Neuberechnung mehr statt, sondern es gibt einen separaten Prozess, der Einträge neu berechnet. Das kann etwa ein Daemon-Thread, ein Cronjob oder ein manueller Trigger sein. Der Vorteil dabei ist, dass man die Neuberechnung so komplett von einzelnen Requests entkoppelt. Das ermöglicht beispielsweise Aktualisierungen zu jeder vollen Stunde und beseitigt das Problem einer Cache-Stampede sehr effektiv, hat dafür jedoch andere Nachteile.
Technisch hat man eine weitere Komponente im System, die gewartet und überwacht werden will. Verlässt sich die Anwendung darauf, alle Einträge im Cache vorzufinden, hat diese Komponente Potenzial zum Single Point of Failure. Natürlich kann man dagegen wiederum Fallback-Lösungen in den einzelnen Service-Instanzen implementieren, dann ist man aber schnell zurück beim ursprünglichen Problem der Cache-Stampede.

Auch fachlich eignen sich nicht alle Anwendungsfälle für einen Cache mit externer Aktualisierung: Will man eine Million Artikel eines Onlineshops ständig im Cache vorhalten, obwohl für 90 % der Anfragen lediglich ein einstelliger Prozentsatz davon benötigt wird? Kann man von Anwendern erstellte Inhalte (etwa Produktreviews) überhaupt in ihrer Gesamtheit bestimmen?

Probabilistic Early Expiration

Aber was wäre, wenn… man die Einträge bereits neu berechnet, *solange sie noch gültig sind?* Also: „Verhalte dich im Hinblick auf die TTL so, als wäre es jetzt + X". In diesem Fall könnten weitere Requests während der Zeit der Neuberechnung noch aus dem Cache

DAS CLOUD NATIVE FESTIVAL

DAS EVENT DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY

on demand

DAS CLOUD NATIVE FESTIVAL VERPASST?

JETZT ON-DEMAND-TICKET BUCHEN UND VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

Alle Angebote im On-Demand-Ticket-Shop

beantwortet werden und niemand müsste warten. Klingt erst mal gut – verwendet man dabei für X jedoch einen konstanten Wert, verschiebt man die Cache-Stampede einfach um diesen Wert nach vorne. Es gilt also, trotzdem noch die Anzahl der gleichzeitigen Neuberechnungen zu reduzieren, indem man etwa nur bestimmte Instanzen damit beauftragt.

Ein eleganter Weg, dies ohne aufwendiges Locking zu ermöglichen, ist die Einbindung einer Wahrscheinlichkeitsfunktion, wodurch jeder Zugriff auf einen Cache-Eintrag eine gewisse Chance hat, diesen neu zu berechnen. Dabei gibt es grundsätzlich verschiedene mögliche Ausprägungen. Die Autoren Andrea Vattani, Flavio Chierichetti und Keegan Lowenstein stellen in ihrem Paper „Optimal Probabilistic Cache Stampede Prevention" *[2]* eine Wahrscheinlichkeitsfunktion namens „XFetch" vor, bei der diese Chance exponentiell in Richtung der TTL ansteigt. Im Anschluss beweisen die Autoren, dass ihre Funktion theoretisch optimal ist und in der Praxis deutlich bessere Ergebnisse erzielt als andere Wahrscheinlichkeitsfunktionen.

XFetch

Die Funktion selbst ist ebenso einfach wie elegant und unabhängig von der konkreten TTL eines Eintrags:

```
delta * beta * ln(random())
```

Die Bestandteile im Einzelnen:

- `delta` ist die Zeit, die benötigt wird, um einen Eintrag neu zu berechnen. Das ist wichtig, damit diese Neuberechnung rechtzeitig stattfinden kann. Angenommen, eine Berechnung dauert zehn Sekunden, dann muss diese mindestens zehn Sekunden vor Ende der TTL angestoßen werden, damit die neuen Daten rechtzeitig wieder im Cache abgelegt werden können. Um diese Information mitzuführen, empfiehlt es sich, für jeden Eintrag einfach die Dauer der letzten Berechnung mit im Cache abzulegen.
- `ln(random())` ist der natürliche Logarithmus auf einer Zufallszahl zwischen 0 und 1 und sorgt dafür, dass die Wahrscheinlichkeitsfunktion exponentiell ansteigt.
- `beta` ist standardmäßig 1 und die einzige manuelle Stellschraube, die genutzt werden kann, um unterschiedliche Schwerpunkte zu setzen: Erhöht man den Wert von `beta`, werden Cache-Stampedes noch effektiver verhindert, man erhöht allerdings gleichzeitig auch die Wahrscheinlichkeit, dass Einträge vor Ablauf ihrer TTL neu berechnet werden.

Abbildung 1 verdeutlicht den Einfluss von `beta` bei ansonsten gleichbleibenden Parametern: Die Grafik zeigt eine Simulation von Requests in den fünf Minuten vor Ablauf der TTL eines Eintrags und dazu die Anzahl der angestoßenen Neuberechnungen als Abschätzung für die jeweilige Wahrscheinlichkeit. Wie man sieht, beginnen bei einem `beta` von 4,0 die ersten Neuberechnungen bereits 280 Sekunden vor Erreichen der TTL, während sie bei einem Wert von 0,25 erst 15 Sekunden davor starten. Üblicherweise sollte man zunächst mit dem Standardwert beginnen und nur bei Bedarf davon abweichen.

Listing 3 zeigt eine Java-Implementierung der „XFetch"-Funktion. `Math.random()` liefert Zufallszahlen aus dem Bereich [0, 1), der natürliche Logarithmus von 0 ist mathematisch allerdings nicht definiert. Auch wenn Java im Zweifelsfall damit umgehen könnte, wird der Wertebereich auf (0, 1] angepasst.

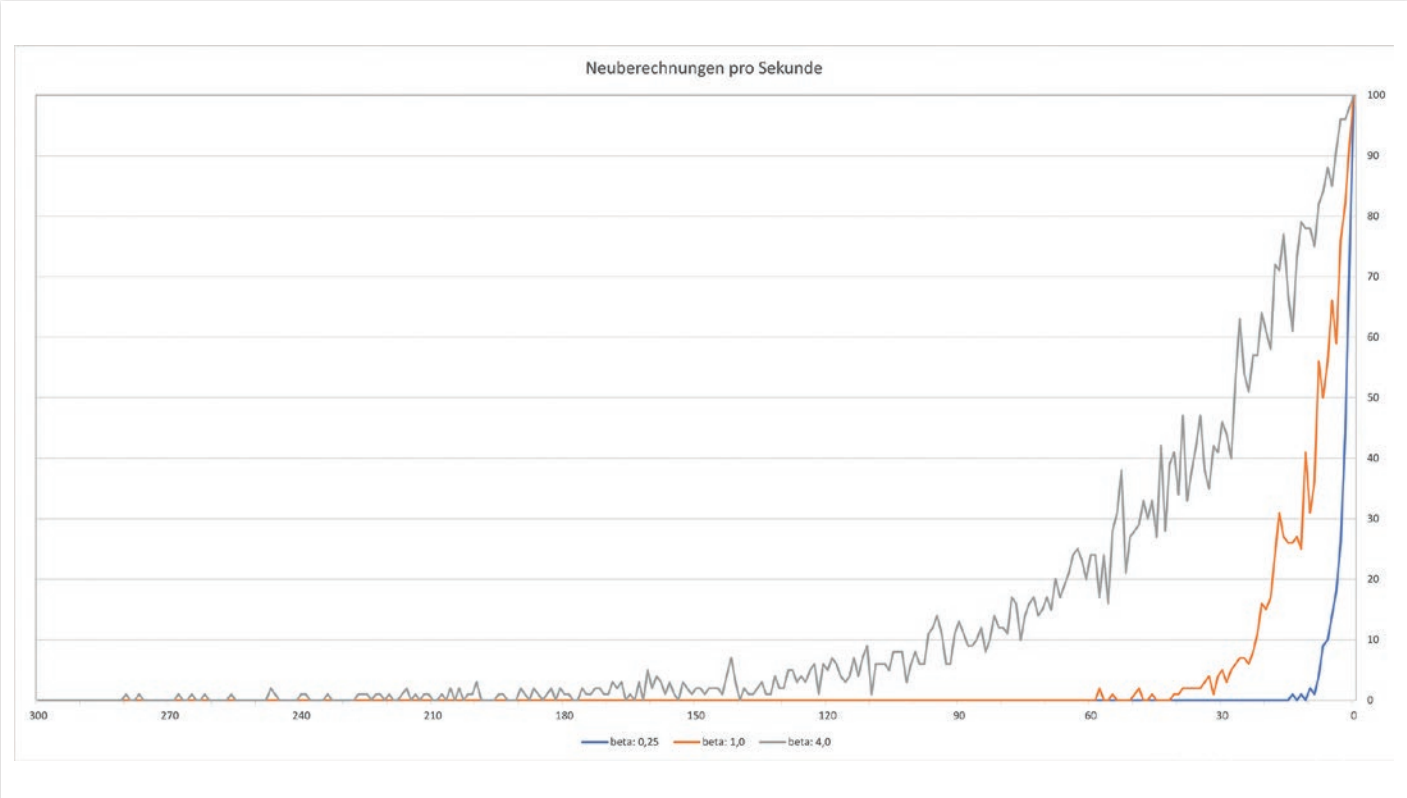


Abbildung 1: Vergleich der Neuberechnungen pro Sekunde mit unterschiedlichen Werten für beta, bei angenommenen 100 Requests/Sekunde und einer Neuberechnungsdauer von zehn Sekunden (© Bernd Stübinger)

```
private double beta = 1.0;

public Value fetch(Key key) {
  Item cachedItem = cache.getItem(key);
  Instant start = Instant.now();
  if (cachedItem == null || fetchEarly(cachedItem, start)) {
     Value value = recompute(key);
     Duration fetchDuration = Duration.between(start, Instant.now());
     Instant expiresAt = start.plusSeconds(ttlSeconds);
     cachedItem = new Item(value, expiresAt, fetchDuration);
     cache.setItem(key, cachedItem);
  }
  return cachedItem.value;
}

public boolean fetchEarly(Item item, Instant now) {
  Duration lastFetchDuration = item.lastFetchDuration;
  double rand = 1 - Math.random();
  long randomSeconds = (long) (lastFetchDuration.toSeconds() * beta * -Math.log(rand));
  return now.plusSeconds(randomSeconds).isAfter(item.expiresAt);
}
```

Listing 3: Eine Cache-Implementierung mit Probabilistic Early Expiration

Der Begriff „Wahrscheinlichkeit" deutet es bereits an: Niemand garantiert, dass am Ende tatsächlich nur *eine* Instanz den Eintrag neu berechnet. Ebenso kann es vorkommen, dass unmittelbar nach der Aktualisierung der Daten direkt wieder eine Neuberechnung angestoßen wird.

In der Praxis ist beides üblicherweise kein Problem. Sollte man jedoch die harte Anforderung haben, dass nur eine einzige Neuberechnung parallel stattfinden darf, muss man tatsächlich auf echtes Locking setzen. In diesem Fall hilft „XFetch" aber zumindest, die Lock-Contention zu verringern, da weniger gleichzeitige Requests um den Lock konkurrieren.

Fazit

Man wird nicht jeden Tag seinen eigenen verteilten Cache implementieren. Aber falls sich der Anwendungsfall doch einmal ergibt, haben wir in diesem Artikel verschiedene Ansätze betrachtet und eine ebenso elegante wie einfach zu implementierende Methode kennengelernt, mithilfe derer Cache-Stampedes der Vergangenheit angehören sollten. Phil Karlton hat einmal gesagt: „There are only two hard things in Computer Science: cache invalidation and naming things" – nach der Lektüre dieses Artikels für euch hoffentlich nur noch Letzteres ;-)

Quellen

[1] Sunny Beatteay (2021): How a Cache Stampede Caused One of Facebook's Biggest Outages. *https://betterprogramming.pub/how-a-cache-stampede-caused-one-of-facebooks-biggest-outages-dbb964ffc8ed*

[2] Andrea Vattani, Flavio Chierichetti und Keegan Lowenstein (2015): Optimal Probabilistic Cache Stampede Prevention. *https://cseweb.ucsd.edu/~avattani/papers/cache_stampede.pdf*

Bernd Stübinger
Java User Group Ingolstadt e.V.
bernd.stuebinger@jug-in.bayern

Bernd ist Backend Engineer bei MediaMarktSaturn Technology und Mitgründer der Java User Group Ingolstadt. Für seinen Arbeitgeber erstellt er performante Cloud-Lösungen unter Verwendung moderner Tech-Stacks und GitOps. In seiner Freizeit beschäftigt er sich mit Software-Esoterik und beantwortet Fragen auf Stack Overflow.

Wir müssen miteinander reden!

Uwe Sauerbrei, Valerio Onofre Vilaca

Das Thema eMobility gewinnt zunehmend an Bedeutung. Überall werden Ladesäulen aufgebaut und in Betrieb genommen. Die Ladesäule muss jedoch mit einem nachgelagerten Backend sprechen können. Ob und wie sie das genau macht, ist Inhalt umfangreicher Tests, die häufig noch manuell ausgeführt werden. Wie diese Tests unter Einsatz von Behaviour Driven Development (BDD) automatisiert werden können, ist Inhalt der nachfolgenden Zeilen.

Personen

- ein Tester
- ein Softwareentwickler
- ein Produktmanager
- eine Ladestation (aka Ladesäule respektive Stromtankstelle)

Prolog

Es begab sich zu einer Zeit, als das Testen von Hardware weder komplett automatisiert noch universell verfügbar war. Der Tester, nun bereits seit einigen Monaten im Bereich der eMobility tätig, hatte sich auf die Anbindung von Ladesäulen verschiedener Hersteller an das Backendsystem spezialisiert. Um eine möglichst reibungslose Integration neuer Technik in die bestehenden Verwaltungssysteme zu garantieren, mussten (gefühlt) endlos viele Szenarien getriggert, geprüft und protokolliert werden. Die Kommunikation zwischen Ladesäule und der Verwaltungssoftware erfolgte über OCPP *[1]*, eine normierte Protokollsprache, deren Inhalte und Feinheiten dem Tester inzwischen auch geläufig waren. Die sich ständig wiederholenden Abläufe und Protokollierungen zeitigten bald ihre Wirkung, der Tester wurde schnell müde, der Verbrauch an Kaffee stieg rapide an und seine Seufzer wurden länger.

Erster Akt – im Büro des Testers

Erste Szene: Der Produktmanager motiviert den Tester.

Produktmanager: „Guten Morgen, ich hoffe, du bist gut ausgeruht, wir haben heute viel zu tun!"
Tester: „Wir?"
Produktmanager: (schaut etwas irritiert) „Unser neuer Kunde von ACME Premium hat die Prototypen der neuen Ladesäulen vorbeibringen lassen. Sie wollen ihre Hardware so schnell wie möglich in unser Netz bekommen. Also möglichst schnell testen und zertifizieren!"
Tester: „Schnell oder gründlich?"
Produktmanager: „Ein Scherz am Morgen, so mag ich das. Schnelligkeit ist stets der große Bruder der Gründlichkeit! Wie lange wird es dauern?"
Tester: (zuckt mit den Schultern) „Ich werde die neuen Protokolle vorbereiten und die Tests durchführen. Das dauert dann so lange, wie es dauert. Vorausgesetzt, alles funktioniert und die Implementierung entspricht der Spezifikation."
Produktmanager: „Du machst das schon. Ich schaue kurz vor Feierabend noch einmal vorbei." (hastet aus dem Büro)
Tester: (sinkt in seinem ergonomischen Bürostuhl etwas zusammen und überlegt, ob er schon jetzt zur zweiten Tasse Kaffee greifen sollte)

Zweite Szene: Der Tester und der Softwareentwickler analysieren die Situation.

Softwareentwickler: „Moin! Oh, was ist los? Chef kam mir gerade entgegen. Neue Ladesäulen im Anmarsch?"
Tester: „Ja, wie immer. Aber ich hatte letztens eine Idee, wie ich das ganze manuelle Testen loswerden könnte. Hast du schon mal etwas von BDD gehört?"
Softwareentwickler: „Behavior Driven Development. Ich glaube, ich weiß, worauf du hinauswillst! Vielleicht kannst du mir zuerst einmal eine kleine Einführung in die Problematik geben."

Dritte Szene: Monolog des Testers zur Ladesäule, zum Laden und zu den Nachrichten

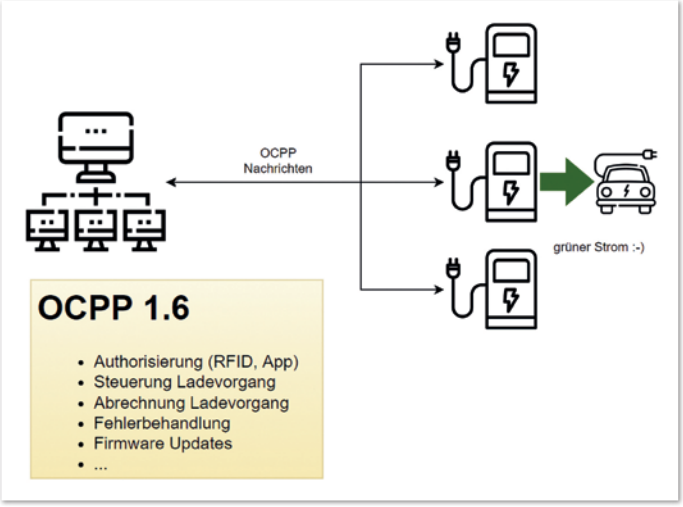


Abbildung 1: Übersicht Ladeprozess

Ganz allgemein gesprochen haben wir eine Reihe von Ladesäulen unterschiedlicher Hersteller. Geladen wird mit Gleich- oder Wechselstrom, je nach Modell der Ladesäule, die gleichzeitig einen oder mehrere Ladevorgänge durchführen kann. Bevor Strom fließt, muss der Ladevorgang autorisiert werden und auch hier gibt es verschiedene Möglichkeiten. Das spielt jedoch alles nur eine untergeordnete Rolle, da die gesamte Steuerung dieser Prozesse an ein Backend delegiert wird. Um das zu ermöglichen, muss jede Ladestation eine Verbindung zu diesem Backend, wir können es auch Rechenzentrum nennen, aufnehmen. Da wird es schon spannender, weil wir hier von einer WebSocket-Verbindung *[2]* sprechen.

WebSocket
In erster Linie basiert eine WebSocket-Verbindung auf einem TCP-Netzwerkprotokoll. Es wird eine bidirektionale Verbindung zwischen einer Webanwendung (in unserem Fall der Ladesäule) und einem WebSocket-Server erzeugt.

Über diese Verbindung werden nun Steuerbefehle übertragen, die auf dem bereits erwähnten OCPP-Protokoll basieren. Die Versionen vor OCPP 1.6 basierten noch auf SOAP, wurden aber mit 1.6 durch WebSocket abgelöst. Die Protokollnachrichten selbst bestehen jeweils aus einem Request und einer zugehörigen Response. Die Abarbeitung erfolgt sequenziell und die meisten Nachrichten können entweder vom Backend oder von der Ladesäule initiiert werden.

Ein einfaches Beispiel

Soll eine Einstellung der Ladesäule vom Backend aus geändert werden, wird ein *ChangeConfigurationRequest* *(siehe Listing 1)* gesendet.

```
{
  "key" : "Testing",
  "value" : "true"
}
```

Listing 1: Inhalt eines ChangeConfigurationRequest im JSON-Format

```
// OCPP spezifiziertes Format eines Requests per WebSocket
[<MessageTypeId>, "<UniqueId>", "<Action>", {<Payload>}]

// konkretes Beispiel für ChangeConfigurationRequest
[2, "123", "ChangeConfiguration", {"key":"Testing", "value":"true"}]
```

Listing 2: Auszug aus der OCPP-Spezifikation für eine CALL-Message

Dabei wird die eigentliche JSON-Nachricht *[3]* als Payload in die übertragene WebSocket-Message verpackt. Auch hier spezifiziert das Protokoll genau das Format. Eine proprietäre **CALL**-Nachricht *(siehe Listing 2)* enthält:

- einen Identifier für die Art der Nachricht (2 = Request),
- eine eindeutige ID (etwa UUID), die Request und Response korreliert,
- einen Verweis auf den fachlichen Inhalt (action=ChangeConfiguration),
- den eigentlichen Request (Payload).

Die Ladesäule ihrerseits entpackt die WebSocket-Botschaft, extrahiert den Request, prüft die Konfigurationsanforderung und wird sie gegebenenfalls akzeptieren oder, wie in unserem Fall, zurückweisen, da sie diesen Key nicht kennt. Dazu erzeugt sie einen *ChangeConfigurationResponse* mit der Antwort *(siehe Listing 3)*.

```
{
  "status" : "NotSupported"
}
```

Listing 3: ChangeConfigurationResponse im JSON-Format

Auch diese Nachricht wird für die Übertragung eingepackt und hier als *CALL_RESULT* übertragen *(siehe Listing 4)*. Wie man sehen kann, hat sich der Identifier geändert (3=Response), wohingegen die UniqueId identisch ist und Request mit Response verbindet. Damit muss auf dem Rückweg auch keine Action übermittelt werden. Die Payload enthält die fachlichen Parameter.

```
// OCPP spezifiziertes Format eines Requests per WebSocket
[<MessageTypeId>, "<UniqueId>", {<Payload>}]

// konkretes Beispiel für ChangeConfigurationRequest
[3, "123", {"status":"NotSupported"}]
```

Listing 4: Auszug aus der OCPP-Spezifikation für eine CALL_RESULT-Message

Damit wäre das Prinzip der Kommunikation erklärt. In den zu testenden fachlichen Anwendungsfällen geht es in erster Linie um das Wechselspiel der verschiedenen Requests und Responses, also dem Versenden parametrisierter Nachrichten und der Prüfung der erhaltenen Antworten.

Softwareentwickler: "Ok, wie du testest, ist mir jetzt klar geworden. Ich habe da auch schon eine Idee, wie wir das automatisieren können. Wir verbinden BDD mit Spring, dazu noch eine Prise Datenbank und asynchrones Tooling. Bekommen wir hin!"

Zweiter Akt (zwei Wochen später) im Büro des Softwareentwicklers

Erste Szene: Rechner, Monitore und eine blinkende Ladestation

Produktmanager: „Okaaay! Tester und Entwickler in einem Raum. Haben wir ein Problem?"
Softwareentwickler: „Ganz im Gegenteil. Wir haben uns eine Lösung für das manuelle Testen der Ladesäulen überlegt und wollten sie dir vorstellen!"

Der Produktmanager atmet hörbar aus und alle nehmen vor dem Rechner Platz, der, auf seine drei Monitore verteilt, ein Diagramm und eine Entwicklungsumgebung öffnet *(siehe Abbildung 2)*.

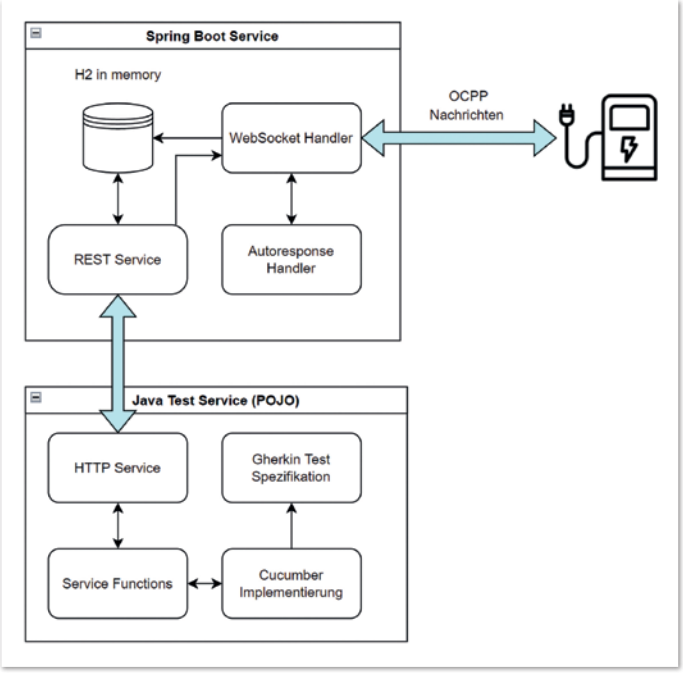


Abbildung 2: Schematische Übersicht des automatischen Testsystems

In der Übersicht haben wir zwei Services und eine Ladesäule. Wir nutzen Spring Boot für die Kommunikation mit der Ladestation über eine WebSocket-Verbindung. Der WebSocket Handler sendet und empfängt alle Nachrichten und speichert sie in einer internen H2-In-Memory-Datenbank. Da wir auch Nachrichten bekommen, die für den Test nicht relevant sind (zum Beispiel Heartbeats), können wir diese, konfigurierbar, automatisch beantworten lassen. Das übernimmt der Autoresponse Handler. Weiterhin haben wir eine REST-

Schnittstelle, die sowohl Zugriff auf die Datenbank als auch den WebSocket-Service hat. Die Anwendung läuft dann als separater Webservice.

Für die eigentliche Durchführung der Tests haben wir uns für BDD *[4]* entschieden.
Produktmanager: "BDD?"
Tester: „Stimmt, sollten wir erklären. Hier ein kurzer Exkurs…"

Seinen Ursprung hat BDD zu Anfang der 2000er Jahre, als Daniel Terhorst-North unzufrieden aufgrund von Problemen in der testgetriebenen Softwareentwicklung (TDD) war. Angestoßen durch agiledox, ein Projekt seines Kollegen Chris Stevenson zum Umwandeln der Namen von JUnit-Testmethoden in gewöhnliche Sätze, entwarf North ein Benennungsschema der Form „*The class should do something*" *[5]*.

Dieses Schema führt auf der einen Seite zu einer direkten Spezifikation des erwarteten Verhaltens, auf der anderen Seite führt es dazu, dass Tests fokussierter gestaltet werden. Dies wird erreicht, da ein so aufgebauter Satz nicht mehr als genau ein Verhalten beschreibt, das im Fokus des Tests steht.

Inspiriert von der domänennahen Sprache des Domain Driven Development (DDD) versuchten North und Stevenson, eine solche Sprache für den Prozess der Anforderungsanalyse zu definieren. Diese Sprache sollte sowohl für Businessanalysten, Software-Entwickler als auch für die Qualitätssicherung gleichermaßen verständlich sein und auf diesem Weg Missverständnisse zwischen diesen Bereichen ausräumen. Dies führte zur Beschreibung der Tests als Szenarios der Form „*Given, When, Then*" und wurde als Gherkin Syntax *[6]* formalisiert. Parallel dazu entwickelte North mit JBehave *[7]* das erste BDD-Framework. Wir nutzen hier mit Cucumber *[8]* eine alternative Implementierung.

Softwareentwickler: „Kommen wir jetzt zur Testanwendung!"

Der zweite Service definiert die Tests, sendet gegebenenfalls Nachrichten an die Ladesäule und prüft in den gespeicherten Nachrichten der Datenbank, ob die erwarteten Ergebnisse enthalten sind. Dazu ein konkretes Beispiel.

Die Aufgabe

Im Open Charge Point Protocol sind einige Passagen nicht eindeutig spezifiziert und lassen einen gewissen Interpretationsspielraum in der Implementierung sowohl für die Hersteller der Ladesäule als auch für das Backend, was in der Praxis immer wieder zu Problemen führen kann.

Es gilt also sicherzustellen, dass solche Kommunikationsschwierigkeiten nicht auftreten werden. Zu diesem Zweck wurden eine Reihe von Tests definiert, die an allen notwendigen Stellen auf eine korrekte Kommunikation überprüfen.

Bis vor nicht allzu langer Zeit wurden diese Tests noch schriftlich als Schritt-für-Schritt-Anleitung festgehalten, die dann für neu einzubindende Ladesäulen manuell abgearbeitet werden mussten. Diese manuelle Ausführung nimmt nicht nur viel Zeit in Anspruch, sie ist natürlich auch fehleranfällig.

Bei dem folgenden Beispiel *(siehe Listing 5)* für einen dieser Tests zur Hardwareintegration geht es darum, zu überprüfen, ob das Ändern von Konfigurationen in der Ladestation funktioniert. Dabei setzt sich der Test aus zwei Schritten zusammen:

1. Senden des Änderungskommandos
2. Abfragen der Konfiguration, um sicherzustellen, dass die Änderungen auch durchgeführt wurden

Hier werden parametrisierbare Tests definiert, die, in lesbarer Prosa, die Erwartungshaltung beschreiben. Für die Implementierung bietet Cucumber die Generierung von Stubs an, die ihrerseits mit Leben gefüllt werden müssen, wie man nachfolgend sehen kann. Die fachliche Implementierung ist in Services ausgelagert, die im Scope der jeweiligen OCPP-Nachricht agieren.

Um die einzelnen Testfälle fachlich zuordnen zu können und die zugehörigen Daten der Datenbank zu finden, bekommt jeder Test eine

```
Feature: Change Configuration
  This scenario is used to set the value of a configuration key.

  Scenario Outline: Central System sets a configuration
     Given Test for <scenario>
     When ChangeConfigurationRequest with <key> and <value> has been sent to the charging station
     Then The charge point responds with <status>

  Examples:
     | scenario      | key                       | value | status     |
     | "EM-5662-S01" | "MeterValueSampleInterval" | 15    | "Accepted" |

  Scenario Outline: Central System requests a configuration
     Given Test for <scenario>
     When GetConfigurationRequest with <key> has been sent to charging station
     Then The charge point responds with <value>

  Examples:
     | scenario      | key                       | value |
     | "EM-5662-S02" | "MeterValueSampleInterval" | 15    |
```

Listing 5: Definition des Tests in Gherkin-Syntax

```
public class CommonSteps {
  @Given("Test for {string}")
  public void test_for(String scenario) {
     setCurrentScenario(scenario);
  }
}
```

Listing 6: Implementierung der gemeinsamen Schritte

eindeutige Kennung, ein Szenario. Alle Tests nutzen dieselbe Implementierung *(siehe Listing 6)*.

Für diesen Test wird ein *ChangeConfigurationRequest* erzeugt *(siehe Listing 7)* und über die REST-Schnittstelle an den WebSocket-Service übermittelt, der ihn an die Ladesäule weiterreicht. Ebendiese verarbeitet den Request und wird die Änderung akzeptieren (wie im dargestellten Fall) oder zurückweisen, falls es bei der Verarbeitung einen Fehler gab. Das Ergebnis wird in der zugehörigen Response zurückgeschickt und in der Datenbank gespeichert.

In einem zweiten Testschritt wird nun der gesetzte Konfigurationsparameter (*MeterValueSampleInterval*) abgefragt und mit dem erwarteten Ergebnis (15) verglichen *(siehe Listing 8)*.

Es sollte noch erwähnt werden, dass die gesamte Kommunikation mit der Ladesäule asynchron abläuft. Das Testsystem sendet einen Request und die Ladestation wird in einem zeitlichen Intervall antworten. Konkret bedeutet das, dass der Test bezüglich der Antwort in regelmäßigen Abständen die Datenbank pollen muss. Dafür kommt Awaitility *[9]*, eine kleine Bibliothek, die die Konfiguration der Polling-Intervalle und eines Timeouts ermöglicht, zum Einsatz. Das Überschreiten des Timeouts wird dann als Fehlerfall bewertet.

Die Auswertung

Für die Darstellung der Testergebnisse bietet Cucumber ein Report-Plug-in an, das nach der Ausführung eine entsprechende Übersicht anbietet *(siehe Abbildung 3)*. Die Ansicht ist, zugegebenermaßen, recht rudimentär, enthält aber alle wesentlichen Informationen. Es besteht allerdings die Möglichkeit, durch die Einbindung von Serenity BDD *[10]* eine umfangreiche Bibliothek mit weiteren Konfigurationsmöglichkeiten und detaillierterer Darstellung der Ergebnisse zu bekommen.

Tester: „Bisher habe ich für die passiven Tests einer Station einen Tag benötigt, um sie manuell durchzuführen. Automatisiert dauert es nun eine Stunde!"
Produktmanager: „Ich bin beeindruckt!"
Softwareentwickler: „Wir müssen uns noch Gedanken machen, wie

```
public class ChangeConfigurationSteps {
     @When("ChangeConfigurationRequest with {string} and {int} has been sent to charging station  ")
     public void change_configuration_request_with_key_and_value_has_been_sent_to_charging_station(
        String key,
        int value) {
        sendChangeConfigurationRequest(key, value);
     }

     @Then("The charge point responds with ChangeConfigurationResponse with {string}")
     public void the_charge_point_responds_with_change_configuration_response_with(String status) {
        awaitChangeConfigurationResponse(status);
     }
}
```

Listing 7: Auswahl beim Start von Obsidian

```
public class GetConfigurationSteps {
     @When("GetConfigurationRequest with {string} has been sent to charging station")
     public void get_configuration_request_with_key_has_been_sent_to_charging_station(String key) {
        sendGetConfigurationRequest(key);
     }

     @Then("The charge point responds with {string}")
     public void the_charge_point_responds_with_status(String status) {
        awaitGetConfigurationResponse(status);
     }
}
```

Listing 8: GetConfigurationRequest zur Abfrage des Parameters

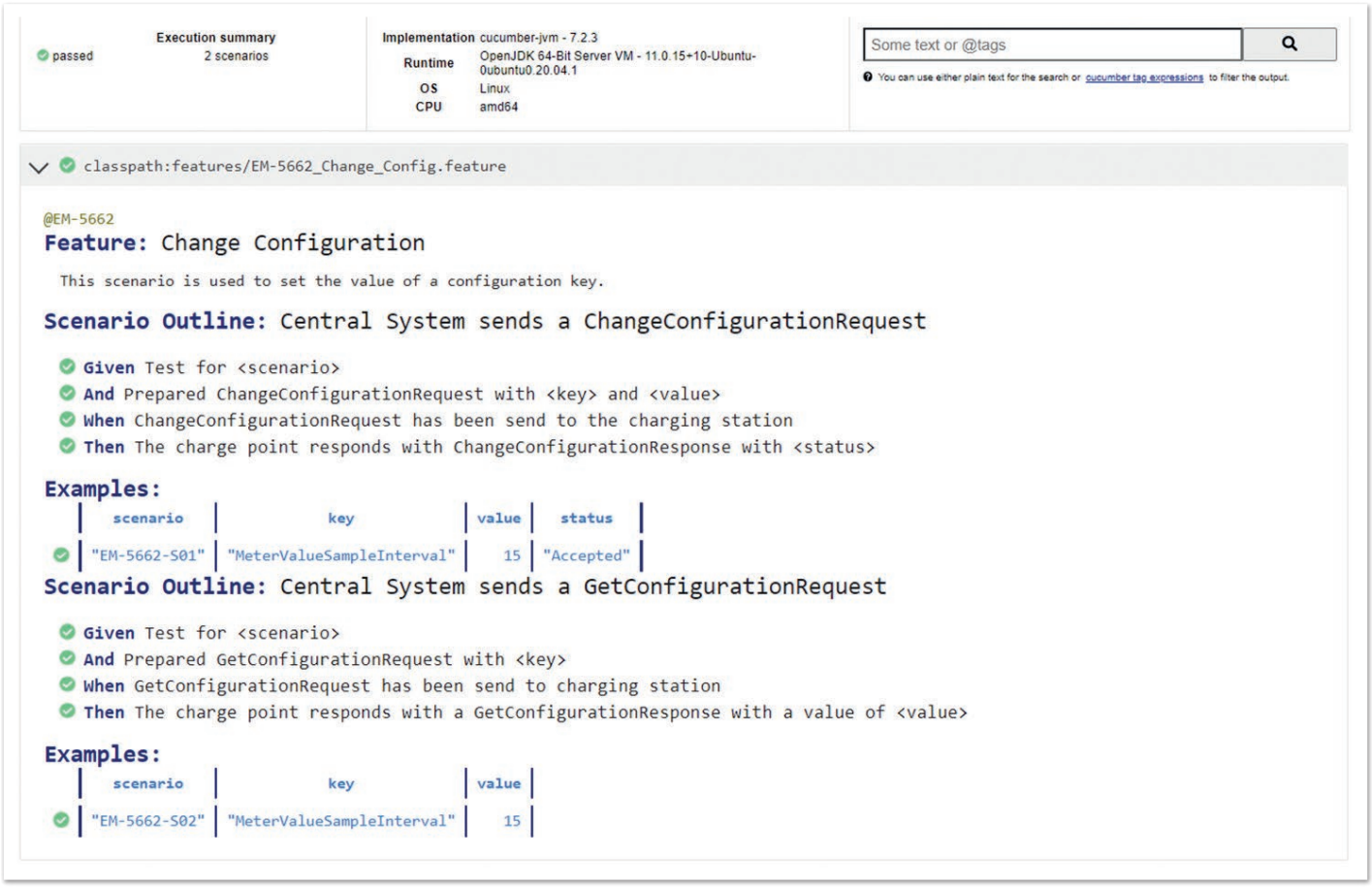


Abbildung 3: Auswertung der Testergebnisse der einzelnen Schritte

wir Tests realisieren, die eine konkrete physische Handlung an der Ladestation erfordern. Beispielsweise das Anschließen eines Ladekabels oder die Registrierung einer RFID-Karte. Aber das ist eine andere Geschichte."

Epilog

Auch wenn nicht alle Probleme im Zusammenhang mit der Integration einer Ladesäule gelöst werden konnten, ist die Verwendung eines BDD-zentrierten Einsatzes der Testwerkzeuge eine erhebliche Erleichterung, um dem schnell wachsenden Zoo von neuer Ladetechnik Herr zu werden. Durch die sequenzielle Arbeitsweise einer Ladestation ist die Parallelisierung von Tests nicht möglich und manuelle Aktivitäten an der Technik können aktuell nur eingeschränkt simuliert werden.

Quellen

[1] OCPP: *https://www.openchargealliance.org/protocols/ocpp-16*
[2] WebSocket: *https://de.wikipedia.org/wiki/WebSocket*
[3] JSON: *https://www.json.org/json-de.html*
[4] BDD Wikipedia: *https://de.wikipedia.org/wiki/Behavior_Driven_Development*
[5] Dan North Introducing BDD: *https://dannorth.net/introducing-bdd*
[6] Gherkin: *https://cucumber.io/docs/gherkin/reference/*
[7] JBehave: *https://jbehave.org*
[8] Cucumber: *https://cucumber.io*
[9] Awaitility: *https://github.com/awaitility/awaitility*
[10] Serenity BDD: *https://serenity-bdd.info*

Uwe Sauerbrei
Sauerbrei IT-Consult
info@uwe-sauerbrei.de

Uwe Sauerbrei arbeitet als freiberuflicher Entwickler in Hamburg, organisiert seit vielen Jahren die JUG HH und ist Gründer von Kids4IT.

Valerio Onofre Vilaca

Valerio Onofre Vilaca ist Werkstudent und Bachelorand beim Stromnetz Hamburg. Zusätzlich unterstützt er im VDIni-Club Hamburg-Bergedorf auch die Kleinsten dabei, die Geheimnisse der Technik zu entdecken.

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen
43 JUG Saarland

iJUG Verbund
www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Andrisek, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © freepik
https://freepik.com
S. 10 + 11: Bild © Julslst
https://stock.adobe.com
S. 18: Bild © rawpixel
https://123rf.com
S. 26 + 27: Bild © malchev
https://stock.adobe.com
S. 33: Bild © Bro Vector
https://stock.adobe.com
S. 38 + 39: Bild © asrulaqroni
https://freepik.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

Mediadaten und Preise:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH
www.wir-machen-druck.de

Inserentenverzeichnis


DOAG 2022
Konferenz + Ausstellung
In Nürnberg
20.-23. SEPT.
Die Oracle-ANWENDERKONFERENZ
anwenderkonferenz.doag.org
Eventpartner:
AOUG
AUSTRIAN ORACLE USER GROUP
SOUG
swiss oracle user group
iJUG
Verbund