Java aktuell

iJUG Verbund
www.ijug.eu

Newcomer-Beiträge
AWS Serverless Lambda-Funktionen, Projektkommunikation, Mob Programming, Einstieg in die Softwareentwicklung

Spring Native
Java und Spring Boot für die Cloud, Testen von Kubernetes-Controllern und -Operatoren, Strings

Next Generation

Ausgabe 03/2023
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 03

CloudLand 2023

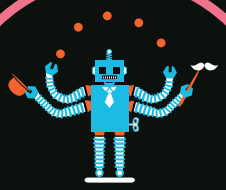
DAS EVENT DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY

Weitere Informationen

20. - 23. JUNI

im Phantasialand in Brühl

www.cloudland.org



Tag 1 / 20. Juni
CloudCamp

Tag 2 / 21. Juni
Summer Night

Tag 3 / 22. Juni
Gaming Night

Tag 4 / 23. Juni
Customer Stories

#CloudLand2023

Eventpartner: Heise Medien

Liebe Leserinnen und Leser,

unser Titelthema lautet "Next Generation" – doch was bedeutet das eigentlich? Wir freuen uns, Erstautorinnen und -autoren, die sich bisher nicht getraut haben, ihr Wissen zu teilen, in dieser Ausgabe die Möglichkeit dazu zu geben. Daher werden diese im vorderen Teil der Zeitschrift berücksichtigt. Ein Großteil unserer Erstautorinnen und -autoren ist gleichzeitig auch Teil des Newcomer-Programms der JavaLand 2023 für Erstreferierende, sodass sie in diesem Jahr gleich zwei Premieren absolviert haben.

Den Anfang macht Jens Knipper, der ab Seite 14 das lokale Testen von AWS Serverless Lambda-Funktionen näher beleuchtet. Dabei berücksichtigt er sowohl Unit-, Integration- und Component-Tests. Im Anschluss gibt Isabelle Rotter wertvolle Tipps und Tricks, wie Developer auch mit Kundinnen und Kunden ohne weitreichendes Software-Vokabular zielorientiert und verständlich kommunizieren können. Florian Schneider bringt uns Mob Programming näher. Er erklärt, was diese Methode der Zusammenarbeit überhaupt ist, wie sich diese in der Praxis (und auch remote) umsetzen lässt und teilt seine eigenen Erfahrungen. Den Abschluss unserer Newcomer-Reihe macht Pauline Schulze mit ihrem Artikel ab Seite 32. Darin befasst sie sich mit dem Thema Berufseinstieg in die Softwareentwicklung. Dabei macht sie Mut, sich bei der Job-Bewerbung nicht von hohen Anforderungen in einer Stellenanzeige abschrecken zu lassen.

Wir hoffen, dass sie alle große Freude am Verfassen ihrer Artikel hatten und auch in Zukunft wieder gerne Beiträge verfassen werden.

Neuigkeiten aus der Java- und Eclipse-Community findet ihr wie gewohnt im Java-Tagebuch und der Eclipse Corner am Anfang der Zeitschrift. Außerdem berichtet Andreas Badelt zu den Hintergründen von MicroProfile 6.0 und dem aktuellen Stand der Zusammenarbeit innerhalb des Projektteams.

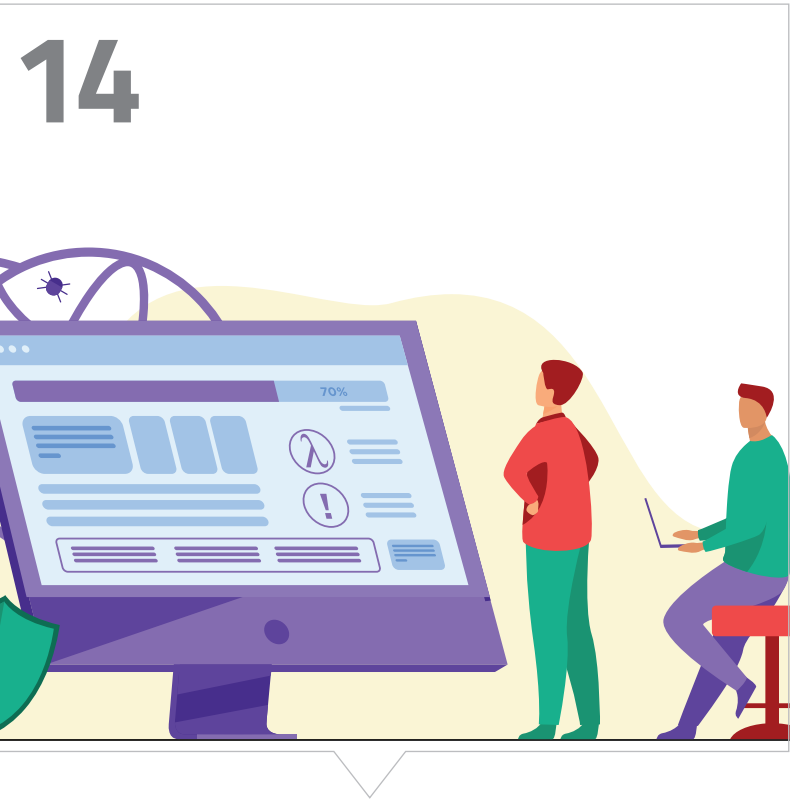
Auch eingefleischte Autorinnen und Autoren sind in dieser Ausgabe wieder mit vertreten, um Wissenswertes mit euch zu teilen. So zeigen Benjamin Klatt und Matthias Klutz ab Seite 38 Potenziale und Herausforderungen von Spring Native und gehen dabei auf die Änderungen des Spring Boot 3 Stack im Vergleich mit seinem Vorgänger ein. Im Anschluss präsentiert Alex Stockinger Möglichkeiten, Kubernetes-Operatoren und -Controller in Java mithilfe von Tools zu testen. Bernd Müller widmet sich zum Abschluss dieser Ausgabe dem Thema Strings und zeigt uns, was jeder Java-Developer darüber wissen sollte.

Übrigens: Java aktuell ist jetzt auch auf Mastodon. Folgt uns unter ijug.social/@javaaktuell, um über kommende Beitragsaufrufe und Veröffentlichungen informiert zu werden!

Wir wünschen euch viel Spaß beim Lesen!
Eure

Lisa Damerow
Redaktionsleitung Java aktuell

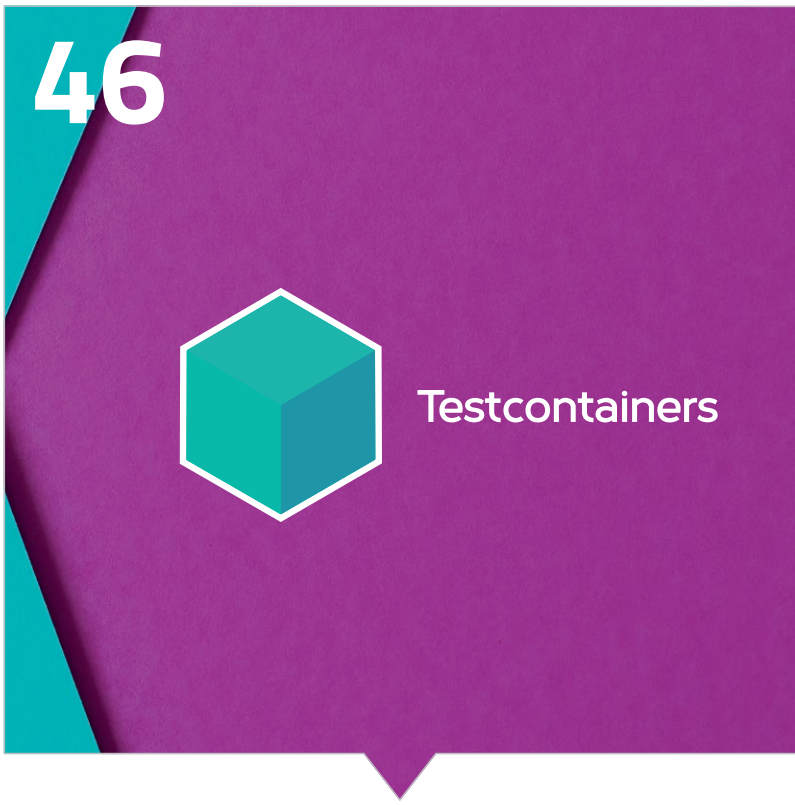
Inhalt



Möglichkeiten zum Testen von AWS Serverless Lambda-Funktionen

Erfolgreiche Kommunikation im Projekt

Spring Native:
Mit Java und Spring Boot in die Cloud



Testen von in Java implementierten Kubernetes-Controllern und -Operatoren

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

08.12.2022

Eclipse Migration Toolkit for Java

Das Eclipse Migration Toolkit for Java (EMT4J) wird Teil von Adoptium. Das vor einem halben Jahr gestartete Open-Source-Projekt soll Entwicklern die Migration ihrer Projekte auf neuere Java-Versionen erleichtern, bei einem Fokus auf die drei Long-Term-Support-Releases 8, 11 und 17 (ich nehme an, ab Herbst wird dann 21 im Fokus sein). Dazu bedient es sich statischer Code-Analyse, kann aber auch als Agent zur Laufzeit Checks durchführen und stellt die gefundenen Probleme in Reports zusammen. Gestartet wurde es laut Eclipse-Blog beim chinesischen Cloud-Anbieter Alibaba, der angefangen hat, seine Erfahrungen mit den Migrationen in Tools zu gießen. Alibaba ist bei Eclipse gleichzeitig Mitglied im Adoptium-Projekt, das ja OpenJDK Binaries zur Verfügung stellt. Dort ist EMT4J also genau am richtigen Ort.

14.12.2022

Project Galahad

Die auf der JavaOne angekündigte Übergabe der Java-bezogenen Teile der GraalVM an das OpenJDK-Projekt wird konkreter. In der OpenJDK-Mailingliste hat Doug Simon von Oracle Details für das Projekt und die personelle Besetzung vorgeschlagen, die basierend auf einem Clone des JDK-20-Codes Stück für Stück Technologien aus dem Graal Repository hinzufügen soll *[1]*. Der Name des Projekts ist passend gewählt: Galahad – der Ritter aus der Artus-Sage, der „auserwählt ist, den Heiligen Graal zu finden".

15.12.2022

Adoptium und IBM fehlt Unterschrift von Oracle

Im Adoptium-Projekt herrscht Frust, weil Oracle nach sechs Monaten immer noch nicht das gegenseitige „Adoptium TCK Participant Agreement" mit IBM und der Eclipse Foundation unterschrieben hat – wodurch effektiv IBMs Mitarbeit im Temurin-Compliance-Projekt verhindert wird. Jetzt soll das Ganze an das Eclipse Board eskaliert werden, um Oracle zu einer Unterzeichnung binnen zwei Wochen aufzufordern. Könnte bei einem von Juristen gesteuerten Tanker – dazu kurz vor Weihnachten – schwierig werden.

10.01.2023

MicroProfile 6.0 – Ende gut, fast alles gut

Jetzt ist MicroProfile 6.0 ganz offiziell freigegeben. Anfang Dezember hatte (nicht nur) ich gerätselt, ob die Abstimmung innerhalb der Working Group jetzt die nötige Mehrheit ergeben hat. Das hatte sie tatsächlich nicht. Doch nach einigen weiteren Diskussionen und vor allem Extra-Arbeit kurz vor Weihnachten, unter anderem aufgrund noch entdeckter Schwachstellen, gab es dann einen weiteren Durchgang. Trotz erneuter Gegenstimmen von Red Hat und Tomitribe wurde diesmal die „super majority" erreicht (die beteiligten JUGs inklusive iJUG haben diesmal dafür gestimmt) – und das Release in der Nacht auf den 23. Dezember in der internen Mailingliste freigegeben. Bedingt durch die Weihnachtspause ging die offizielle Ankündigung allerdings erst heute raus.

Mehr gibt's hier nicht zu „MP", dafür aber in einem eigenen Artikel ab Seite 10 in dieser Ausgabe der Java aktuell.

11.01.2023

Jakarta 11

Die Jakarta Ambassadors haben einen „Guide to Contributing to Jakarta EE 11" erarbeitet *[2]*. Es geht jedoch nicht nur um Tipps, wie sich die Community beteiligen kann und welche Formalitäten gegebenenfalls zu erfüllen sind, sondern es werden auch viele konkrete Vorschläge dazu gemacht, was auf Ebene der Plattform oder in einzelnen Spezifikationen angepackt werden sollte. Zu viele, um sie hier zu nennen. Praktisch zeitgleich gibt es dazu eine Resolution des Jakarta-Lenkungsausschusses (steering committee), die die Planungsergebnisse knapper zusammenfasst: Das geplante Release-Datum ist Q1 2024. Die wichtigsten Projektziele sind: das Bauen auf der (dann) neuesten Java Version – also 21 –, eine Vereinheitlichung der APIs und das Hinzufügen neuer Spezifikationen. Außerdem – Stichwort „mehr Community-Beteiligung" – sollen mit vielen einzelnen Maßnahmen mehr Committer beziehungsweise generell Beitragende gewonnen werden, indem das „On-boarding" erleichtert wird, Prozesse vereinfacht werden und beispielsweise die TCKs vermehrt bei den einzelnen Projekten angesiedelt werden, um die Arbeit zu erleichtern.

Was ist mit den neuen Spezifikationen gemeint? Darüber gibt es noch kein klares Bild. Die „Ambassadors" führen NoSQL und (g)RPC als Beispiele auf. In den detaillierten Folien des Projekts zur Ideensammlung sind es Config, Data, MVC und wiederum RPC. Zumindest MVC existiert ja schon länger als Einzelspezifikation, hat es aber bislang nicht in die Plattform geschafft. Das Thema Config wiederum könnte noch einige Diskussionen mit sich bringen, da das Verhältnis zu MicroProfile (das ja bereits ein Config API hat) trotz aller Bemühungen der Cloud Native for Java Alliance weiterhin nicht geklärt ist.

Wie viel davon wir dann wirklich in Jakarta EE 11 sehen werden, hängt erheblich vom Ziel „mehr Community-Beteiligung" ab. Ein Zitat dazu aus den Projektfolien: „Wir haben mehr Ideen zu Features und zur technischen Ausrichtung als Kapazitäten, sie zu liefern."

18.01.2023

Preview Features Retrospektive und Ausblick

Wer sich – potenziell als Early Adopter von neuen Java-Features – für die Feinheiten der Status „Preview" und „Incubating" interessiert, kann dies jetzt in einem eigenen JEP-Entwurf im OpenJDK-Projekt nachlesen: „Preview Features: A Look Back, and A Look Ahead" *[3]*. Quintessenz: Das OpenJDK-Projekt sieht die ab 2018 genutzten Previews als große Verbesserung für die sorgfältige Einführung neuer Features an und betont gleichzeitig den Qualitätsanspruch. Nur Fea-

tures, bei denen davon ausgegangen wird, dass sie innerhalb von 12 Monaten die nötige Reife erlangen, um permanent ins JDK aufgenommen zu werden, sollen überhaupt als Preview veröffentlicht werden. Das soll weiterhin gelten, auch wenn es in der Vergangenheit nicht immer möglich war und auch zukünftig für komplexe oder tiefgreifende Features nicht immer funktionieren wird. Insbesondere wenn diese mit anderen Preview Features interagieren, werden es schon mal vier Preview-Phasen, also zwei Jahre. Die Alternative wäre, wie früher, den gesamten „Train" zu stoppen – was wohl niemand will.

Für solche Features öfter den Status „Incubating" zu verwenden, sei jedoch auch keine Lösung. Dadurch, dass die APIs dann in eigenen „Inkubator-Modulen" liegen, können sie nicht in Signaturen in java.*-Packages verwendet werden; somit werden Implementierungen von Low-level APIs wie im Falle der Virtual Threads extrem schwierig. Allerdings soll der Inkubator-Status weiterhin dann verwendet werden, wenn die Stabilität und Vollständigkeit „deutlich unterhalb" eines Preview-API liegen.

Eine wichtige Unterscheidung, die im Text erklärt wird, ist die zwischen Previews von Language Features und API-Änderungen. Während in der Vergangenheit häufig beide Hand in Hand gingen (zum Beispiel bei Records und Sealed Classes), ist mit Virtual Threads zum ersten Mal eine Preview reiner API-Änderungen erschienen – in Zukunft werden es wohl mehr. Der entscheidende Unterschied ist, dass Language Features „eine kleine Oberfläche, aber einen breiten konzeptionellen Bereich" haben – wie am Beispiel der funktionalen Programmierung mit Lambdas zu sehen, deren einzige „Oberfläche" das Symbol „->" ist. Umgekehrt bildet bei APIs die Oberfläche mehr oder weniger den konzeptionellen Bereich ab. Das führt dazu, dass selbst bei eigentlich stabilen Konzepten an APIs noch häufig Änderungen vorgenommen werden, weil jede Detail-Änderung dort sichtbar wird. Auch wenn das für Nutzer wie Implementierer des Preview-API weiteren Aufwand bedeutet, will das OpenJDK-Projekt dem Eindruck entgegenwirken, dass hier weniger reife Konzepte ins Rennen geschickt werden.

20.01.2023

Spring Modulith

Das erst im Oktober offiziell angekündigte Projekt Spring Modulith hat in den letzten vier Wochen gleich zwei Versionssprünge auf 0.2 und jetzt 0.3 hingelegt. Das unter anderem auf ArchUnit basierende Projekt soll den prüfbar konsistenten modularen Aufbau von Sourcecode, Integrationstests der Module und ihre passende Dokumentation ermöglichen beziehungsweise erleichtern. Auch wenn der Projektname erkennbar auf dem Monolithen basiert, der – so die Projektbeschreibung – zuletzt neben den Microservice-basierten Systemen wieder an Popularität gewonnen hat, wird das sicher auch bei einem erheblichen Teil genannter Microservices helfen. Die Übergänge sind im Projektalltag fließend, und so werden aus Microservices schnell mal kleine Monolithen – oder in Zukunft dann vielleicht Modulithen.

24.01.2023

Java 20 Rampdown Phase 2

Für das JDK 20 ist offiziell die Rampdown Phase 2 eingeleitet worden. Das heißt, die Liste der Features beziehungsweise ihre Implementierungen sind abgeschlossen, und ohne Sondergenehmigung werden nur noch Prio 1 und 2 Bugs abgearbeitet. Der richtige Zeitpunkt, um noch mal einen Blick auf die neuen Features zu werfen, die uns erwarten: Keines von ihnen ist „stabil", das wird sich dann sicher mit dem nächsten Long Term Release (21) im Herbst ändern. Alle in der letzten Tagebuch-Ausgabe schon erwähnten „JEPs" sind dabei (als zweite oder teilweise vierte Preview): Record Patterns, Pattern Matching for switch, das Foreign Function & Memory API, Virtual Threads und das Structured Concurrency API. Doch mit JEP 429 „Scoped Values" ist tatsächlich noch einer dazugekommen (als „Incubator"-Version, also noch eher in der experimentellen Phase). Scoped Values sollen das Teilen von „immutable data" sowohl innerhalb von Threads als auch Thread-übergreifend ermöglichen und Thread Locals ablösen. Insbesondere dann, wenn es in Zukunft um virtuelle Threads geht – und damit typischerweise auch um sehr viele.

06.02.2023

Payara zurück in der MicroProfile Working Group

Payara, Hersteller des gleichnamigen, auf dem auf Glassfish basierenden Application Server, hatte sich Ende 2021 aus der Working Group zurückgezogen. Der Grund waren wohl Diskussionen rund um die (nicht ausreichende) Finanzierung des Kompatibilitätsprogramms. Jetzt sind sie wieder dabei – wobei die erwähnte Finanzierung damit auch noch nicht gewährleistet ist.

Referenzen

[1] *https://mail.openjdk.org/pipermail/discuss/2022-December/006164.html*

[2] *https://jakartaee-ambassadors.io/guide-to-contributing-to-jakarta-ee-11/*

[3] *https://openjdk.org/jeps/8300604*

Andreas Badelt

stellv. Leiter der DOAG Cloud Native Community

andreas.badelt@doag.org

Andreas Badelt ist seit 2001 ehrenamtlich im DOAG e.V. aktiv und hat dort inzwischen seine Heimat in der Cloud Native Community gefunden, wobei ihn das Java-Ökosystem bis heute fasziniert. Beruflich hat er von Ende des vorigen Jahrtausends an als Entwickler und später auch Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet. Seit 2016 ist er als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' eclipse-Corner

So langsam wird es ja peinlich, aber was soll ich machen… auch in dieser Ausgabe werdet ihr wieder keinen Artikel von mir zu Jakarta EE 10 sehen – obwohl ich es fest versprochen hatte und diese Spezifikation auch schon seit Monaten veröffentlicht ist. Der Hintergrund ist jedoch simpel: Ich will den Artikel mit Code-Beispielen anreichern, die die Neuerungen leicht verständlich illustrieren und die ihr selbst ausprobieren könnt. Dazu sollten diese auch laufen – tun sie aber leider nicht! Nicht, weil ich zu dusselig bin oder das Ganze sonderlich schwer ist (gerade das Gegenteil sollen diese ja beweisen), sondern weil einige Produkte (CI – Compliant Implementation) bestimmte optionale APIs gar nicht erst implementieren (was ihr gutes Recht ist, was sie aber inhärent untauglich für die Beispiele macht) und andere Produkte dummerweise Bugs an der falschen Stelle haben (und somit die Beispiele einfach nicht funktionieren). Blöde Situation, entspricht allerdings der (aktuellen) Wirklichkeit der EE-10-Welt. Man kennt das ja: Setze nie eine „Komma-Null-Version" in der Produktion ein… Klar, ich könnte euch jetzt Code zeigen, der das Ganze einfach nur theoretisch demonstriert – aber bringt euch das was? Eher nicht. Also machen wir's jetzt halt einfach mal so: Ich schreibe euch mal kurz, was die 10er so in der Theorie draufhat, und den Code liefere ich dann später nach!

Also, was bisher geschah… Jakarta EE 8 war ja nichts anderes als Java EE 8 nur mit anderem Herausgeber (Eclipse Foundation statt Java Community Process), somit auch weitgehender Beendigung der Oracle-Allmacht an Bord der Enterprise. EE 9 (der „Big Bang") war bekanntlich EE 8 in anderem Package (jakarta statt javax) minus einiger veralteter Technologien *[1]*. Erst das folgende Minor-Release EE 9.1 hat neue Möglichkeiten mitgebracht *[2]*, und zwar, Anwendungen basierend auf Java SE 11 zu schreiben statt in Java SE 8, somit also eine ganze Reihe neuer Java-SE-APIs einzusetzen und mit dem neuen Schlüsselwort „var" den Code etwas freundlicher zu gestalten.

Nun ist also EE 10 da, und damit endlich auch wirklich neue EE-APIs! Oder auch weniger – denn das maßgeblich Neue der EE 10 ist ja genau genommen gerade das eingeschränkte Core Profile *[3]*. Seit langer Zeit gab es ja neben dem Full Profile, das ein Cross-Vendor-API für vollständige Application Server darstellt, auch ein EJB-freies Web Profile, das analog ein solches API für reine Web-Server bietet. Das Core Profile nun, das grob gesagt eine weitere Einschränkung gegenüber dem Web Profile darstellt, bietet nun ein Cross-Vendor-API für Microservice-Runtimes an (also ganz ohne dahinter werkelndem Serverprodukt – man bedenke, was nicht da ist, frisst bekanntlich kein Brot – zum Beispiel beim Booten – und geht auch nicht kaputt, braucht also keine Wartung beziehungsweise bietet keine Angriffsfläche). Es kann durch den Verzicht auf einige dynamische Features daher auch in Umgebungen zum Einsatz kommen, die beispielsweise Ahead-of-Time-Kompilierung nutzen. Im Gegenzug muss man sich leider von eben jener vorgenannten dynamischen Magie verabschieden, weshalb hier nur eine geschrumpfte Variante von CDI namens „Lite" zum Einsatz kommt.

Entsprechend ist die unter *[4]* genannte Liste von Komponenten deutlich kürzer als die des Web oder gar des Full-Pendants. Im Kern lässt sich grob sagen, das Core Profile ist JAX-RS mit JSON und CDI – und mehr braucht man eigentlich für einen Microservice auch nicht zwingend beziehungsweise man kann weitere APIs wie zum Beispiel aus dem MicroProfile-Umfeld (siehe den Artikel von Andreas Badelt ab Seite 10) hinzunehmen. Apropos JAX-RS, da sind wir auch schon beim Kern des Core Profile, denn in JAX-RS 3.1 ist bekanntlich als maßgebliches Feature das SeBootstrap-API enthalten, und dieses stellt in Ermangelung eines dahinter werkelnden Serverprodukts (siehe oben, man erinnere sich) den Einstiegspunkt in eine Jakarta-EE-10-Core-Profile-Anwendung dar: Man schreibe eine seit Jahrzehnten bekannte „main"-Methode und sage darin per herstellerübergreifendem SeBootstrap-API: „Bitte CDI-fähige JAX-RS-Anwendung starten" – und nach wenigen Sekunden ist der Microservice gebootet. Ja, so einfach und schnell ist das – kein Deployment, keine Magie, kein Vendor-Lock-in, nur reines Java SE! Das Codebeispiel war in fünf Minuten geschrieben, nur mit einem Texteditor, ganz ohne Hilfe von Templating Engines, IDE-Assistenten oder produktspezifischen Werkzeugen.

Genau hier liegt der Grund, wieso es (noch) keinen Artikel mit Code-Beispielen gibt: Zwar funktioniert dieser Bootstrap, doch sobald ich CDE Lite ins Spiel bringen will, hängt sich die Implementierung meiner Wahl einfach auf. Vermutlich ein simpler Bug, aber für einen Artikel über Jakarta EE 10 Core Profile nun mal ein Showstopper. Wie gesagt, sobald ich ein Produkt in die Hand bekomme, das zumindest

dieses Kernfeature von Jakarta Core einigermaßen beherrscht, liefere ich euch umgehend die Code-Beispiele in einem Folgeartikel zum Selbstausprobieren nach!

Abgesehen davon ist die (meines Erachtens) größte Neuerung der 10er, dass man nun Anwendungen in Java 17 formulieren darf und somit wieder mehr und modernere SE-APIs und sprachliche Ausdrucksmöglichkeiten zur Verfügung hat (übrigens, ganz nebenbei führen moderne JREs euren Code auch flotter und effizienter aus als das angestaubte Java 8). Das schließt beispielsweise das Schlüsselwort „record" mit ein, das einem viele Getter und Setter spart, aber auch Pattern Matching for instanceof, das einem viele grässliche Type Casts erspart, zudem Sealed Classes, Switch Expressions und Text Blocks. Zudem gibt es in vielen Component Specifications, wie eben beispielsweise in JAX-RS oder in JPA, nützliche Detailverbesserungen, auf die wir schon viele Jahre gewartet haben. Sofern es irgendwann mal endlich mit dem angekündigten Artikel klappt, zeige ich euch eine kleine Auswahl an Schmankerln, die mir dabei besonders gut gefallen haben und die die Mühe des Umstiegs mit kürzerem und effizienterem Code schmackhaft machen!

Referenzen

[1] *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9ReleasePlan*

[2] *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9.1ReleasePlan*

[3] *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee10/JakartaEE10ReleasePlan*

[4] *https://jakarta.ee/specifications/coreprofile/10/jakarta-coreprofile-spec-10.0.html#required_components*

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.

MicroProfile 6.0

Andreas Badelt, DOAG Cloud Native Community

Was macht eigentlich MicroProfile? So lange ist der Sommer 2016 eigentlich nicht her, in dem das Projekt von „nur" fünf Organisationen gestartet wurde. Doch seither ist viel passiert. Gut, diese fünf waren IBM, Red Hat, Payara und Tomitribe, plus die London Java Community – denen Gartner wohl mindestens mal in der Gesamtheit eine gewisse „ability to execute" bescheinigen würde. „MP" hat sich längst als Top-Level-Projekt der Eclipse Foundation etabliert, die Zahl der Unterstützer ist gewachsen und der ursprüngliche Plan ist grundsätzlich aufgegangen: Ein Projekt zu schaffen, das relevante APIs für das Microservice-Entwicklungsmodell liefert, aber mit dessen schneller Evolution Schritt halten kann, indem die Fesseln der methodischen und gründlichen Standardisierung zugunsten größerer Innovation gelockert werden. Das war auch als Abgrenzung zu Jakarta EE gedacht (damals hieß es noch Java EE), das aufgrund anderer Anforderungen, insbesondere bezüglich Stabilität und Kompatibilität, eben oft diese Fesseln spürt. Oder genau genommen weniger als Abgrenzung, sondern als Ergänzung.


MicroProfile

So hat man sich gut arrangiert: Jakarta EE liefert insbesondere mit seinem neuen Core Profile die stabile Basis für das, was typischerweise jede Applikation im Enterprise-Umfeld braucht, wie Context and Dependency Injection, REST-Services etc. MicroProfile baut darauf mit einer Reihe von APIs auf, die für Microservices unverzichtbar sind wie Metriken, Health Checks, umgebungsspezifische Konfiguration und so weiter: die MicroProfile-Plattform (auch als „Umbrella" bekannt). Alle Produkte, die als MicroProfile-kompatibel gelten wollen, müssen diese APIs erfüllen. Dazu kommen eine Reihe optionaler APIs, die je nach Kontext nützlich sind, wie Reactive Messaging, GraphQL oder LRA (Long Running Actions), auch als SAGA bekannt, das einzige dieser APIs, das aktuell noch nicht den Stempel „produktionsreif" hat.

Für das bessere Zusammenspiel der beiden Eclipse-Projekte sorgt die gemeinsame „Cloud Native for Java Alliance" (CN4J), die sich mit den Worten „Collaborate, don't compete" zusammenfassen lässt. Auch Jakarta EE, das seine Unterstützung für Microservices ebenfalls weiter ausbauen will, profitiert ja davon. So auch die Hersteller, da es eine große Überlappung zwischen den Projekten gibt. Einige der MicroProfile-APIs waren ursprünglich ohnehin einmal für Java EE 8 geplant – die Sache mit der Geschwindigkeit…

Also eigentlich ist alles prima, oder? Na ja, fast. Ganz ungetrübt ist auch die Zusammenarbeit in der CN4J nicht. Das liegt vermutlich gar nicht so sehr an den grundsätzlich unterschiedlichen Sichtweisen der Projekte, sondern an denen verschiedener Hersteller, die bis heute nicht überein gebracht worden sind. Was immer wieder zu Diskussionen, teilweise doppelter Arbeit und am Ende geringerer Geschwindigkeit führt – für beide Projekte.

In den ersten Jahren hatte MicroProfile, den eigenen Ansprüchen gemäß, immer drei Releases pro Jahr geschafft. Zuletzt wurde es jedoch etwas zäher – wobei das an vielen unterschiedlichen Dingen liegt. Wie auch immer, 2021 waren es „nur" noch MicroProfile 4.1 sowie 5.0 kurz vor dem Jahreswechsel. Mit dem einzigen Release 2022, MP 6.0, tat man sich allerdings doch sehr schwer.

Das waren zunächst schwierige Entscheidungen, insbesondere rund um das Thema Metriken sowie Telemetrie im Allgemeinen: Hier hatte MicroProfile seit Langem ein eigenes Metrics-API, immerhin schon in Version 4.0, das sehr stark auf der Metriken-Implementierung von DropWizard aufbaute. Inzwischen nimmt aber die Popularität von Micrometer immer weiter zu. Parallel rollt das CNCF-Projekt OpenTelemetry („OTel") das Feld von hinten auf und möchte Metriken, Logging und Tracing aus einer Hand anbieten. Das OpenTracing-Projekt der CNCF wiederum, das bislang von MicroProfile direkt als Tracing-Lösung eingebunden wurde, wird nicht mehr aktiv weiterentwickelt. Es ist jedoch eines der beiden Projekte, aus denen OpenTelemetry entstanden ist. Wie lässt sich das auflösen, ohne für Hersteller und Nutzer ein großes Chaos zu verursachen?

Das wurde lange diskutiert und am Ende hieß es: OpenTracing (3.0) wird aus dem Plattform-Release entfernt und nur noch als optionales API weitergeführt, das vermutlich von den Herstellern dann auch nicht mehr langfristig unterstützt werden wird. Hinzugekommen zur Plattform ist dafür nämlich Telemetry 1.0, basierend auf OpenTelemetry-Tracing. Der Plan ist, auch „OTel" Metrics und Logging irgendwann in MicroProfile zu nutzen, doch beide sind noch nicht stabil genug. Daher wird MP Metrics weitergeführt, aber in der neuen Version 5.0 mit einer Reihe inkompatibler Änderungen:

Es wurde angepasst, um ein implementierungsagnostisches API zu schaffen, das allerdings deutliche Züge von Micrometer trägt.

Zu den nötigen strategischen Diskussionen kamen noch ein paar selbstgemachte, da kurz vor dem anvisierten Release-Termin ein paar spezielle Ansichten einzelner Hersteller vorgebracht wurden, die nicht mit den eigentlichen Plänen übereinstimmten. Das betraf zum einen die Forderung einer Optionalität von Metrics – die aber schon aufgrund des oben beschriebenen Mangels an Alternativen am Ende abgelehnt wurde. Zum anderen ging es darum, dass eine MP-6.0-Zertifizierung auch auf Basis von Jakarta EE 9.1 möglich sein sollte (was dem Hersteller einiges an Arbeit hätte ersparen können). Die Einzelspezifikationen erlauben das zwar und auch die TCKs schlagen nicht fehl, aber das Plattform-Release fordert als Abhängigkeit eindeutig Jakarta EE 10 Core Profile. Also auch hier am Ende eine klare Entscheidung.

Last, but not least wurden dann noch mehrere Sicherheitslücken entdeckt, die entschärft werden mussten. „Quality first" – das gilt auch für das MicroProfile. Mit großem zeitlichen Aufwand wurde dann jedoch verhindert, dass sich das Release noch bis ins Jahr 2023 verschiebt.

So begab es sich, dass MicroProfile 6.0 kurz vor der Stillen Nacht ganz still und leise erschienen ist. Nur eine kurze Mail im Verteiler der Working Group verkündete seine Ankunft. Zu diesem Zeitpunkt war nämlich offiziell bereits Weihnachtspause des gesamten Projekts und wohl auch der Marketing-Abteilung der Eclipse Foundation, sodass nicht einmal eine kurze Mitteilung veröffentlicht wurde. Das wurde dann erst am 10. Januar nachgeholt.

Also ist zumindest für dieses Release am Ende fast alles gut geworden. Aber es bleibt viel zu tun. Die Hersteller scheinen momentan nur schleppend den Releases hinterherzukommen. Mal sehen, wie schnell sich das für 6.0 entwickelt. Momentan gibt es mit OpenLiberty nur die eine für das Release erforderliche kompatible Implementierung. Doch das Projekt soll ja nicht stehen bleiben. Pläne gibt es für 2023 genügend. Einige sind strategisch beziehungsweise für das Marketing wichtig, wie etwa eine Rückkehr zu drei Release pro Jahr, ein „MicroProfile Champions"-Programm und sogar eine Ausweitung der kompatiblen Implementierungen. Andere Themen sind handfester technischer Natur, wie die Unterstützung von Cloud Storage, verteilten Caches oder einem herstellerunabhängigen Modell für Serverless Programming. Weiterhin stehen wichtige Aktualisierungen in Bezug auf die „Infrastruktur" an. Mit Java 21 im Herbst sollen unter anderem virtuelle Threads produktionsreif werden, um nur ein Thema zu nennen. Daneben gibt es ja sowieso schon die oben genannten Herausforderungen rund um die Telemetry. Zudem soll ein genauer Blick auf die „Gap" zum Spring Framework geworfen werden, um nach Dingen zu schauen, die auch für MicroProfile sinnvolle Ergänzungen darstellen.

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

Andreas Badelt ist seit 2001 ehrenamtlich im DOAG e.V. aktiv und hat dort inzwischen seine Heimat in der Cloud Native Community gefunden, wobei ihn das Java-Ökosystem bis heute fasziniert. Beruflich hat er von Ende des vorigen Jahrtausends an als Entwickler und später auch Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet. Seit 2016 ist er als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Lokales Testen von AWS Serverless Lambda-Funktionen

Jens Knipper, OpenValue Düsseldorf

Das Testen von Microservices wird mittlerweile hinreichend praktiziert. Bei Serverless Functions wie AWS-Lambdas wird meist nur manuell nach einem Deployment getestet, speziell wenn es um die Logik geht, die durch AWS (Amazon Web Services) bereitgestellt wird. Dabei ist es auch möglich, die Anwendung automatisiert lokal hochzufahren und zu testen. Aber wie geht das in einem (relativ) geschlossenen System wie der AWS-Cloud? Durch den geschickten Einsatz von LocalStack, Testcontainers und dem AWS-SDK ist es unter anderem möglich, Component-Tests automatisiert auszuführen und etwaige Fehler in der Benutzung der Lambda-Funktion aufzudecken.

Als Dienstleister komme ich durch den Einsatz in unterschiedlichen Projekten auch zwangsläufig mit vielen verschiedenen Technologien in Berührung. Dabei lässt es sich nicht verhindern, dass auch neue Technologien darunter sind. Bei meinem aktuellen Projekt bin ich das erste Mal mit AWS Serverless Lambda-Funktionen in Kontakt gekommen und habe mich gefragt: Wie testet man das?

Da ich zuvor schon an Microservices gearbeitet und eine gewisse Ähnlichkeit zu Lambdas gesehen habe, habe ich versucht, eine Test-Strategie für Microservices anzuwenden. Das funktioniert in unserem Projekt so gut, dass wir unsere Dependency-Updates mittlerweile automatisieren und uns darauf verlassen können, dass wir beim Update von AWS-Dependencies mitbekommen, wenn etwas nicht mehr funktionieren sollte. Bevor wir tiefer in das Ganze einsteigen, schauen wir uns an, wie man Microservices testet.

Testen von Microservices

Zum Testen von Microservices gibt es bereits sehr gute Literatur *[1]*. Wir unterteilen die Tests in Unit-, Integration- und Component-Tests. Unit-Tests testen einzelne Komponenten wie die Domäne und die Service-Schicht. Wir nutzen Integration-Tests, um das Zusammenspiel mit externen Komponenten zu testen. In Component-Tests fahren wir unseren Microservice hoch und feuern Anfragen auf diesen, um die erhaltenen Ergebnisse zu verifizieren.

Ein konkretes Beispiel für einen Microservice und die Test-Struktur ist in *Abbildung 1* zu finden. Dieser Microservice ist nach dem Ports-and-Adapter-Pattern *[2]* aufgebaut. Alle externen Dienste sind durch einen Adapter mit dem Service-Layer verbunden.

Der REST-Adapter ist die äußere Schnittstelle der Anwendung. Sie gibt an, wie der Service aufgerufen werden kann und welche Werte zu erwarten sind. Der HTTP-Adapter stellt die Grenze, etwa zu einem anderen Service, dar. Der Service wird aufgerufen, um Werte zu erhalten, die wir gegebenenfalls für eine Berechnung benötigen. Der Datenbank-Adapter dient dazu, Werte aus der Datenbank zu speichern und zu lesen.

Die Kästen in *Abbildung 1* geben an, wo die Grenzen der einzelnen Tests liegen. Die kleinste Art von Test ist der Unit-Test. Mit ihm können wir die Module testen, die unabhängig von externen Anwendungen sind. Das sind die Domäne, die Service-Schicht und, je nach Framework, der REST-Adapter.

Integration-Tests testen die Integration zu externen Komponenten. In unserem Beispiel sind es ein anderer Service und eine Datenbank. Die Integration zur Datenbank können wir mit einem Tool wie Testcontainers oder einer In-Memory-Datenbank testen.

Testcontainers ist ein Framework, das in Tests genutzt werden kann, um automatisiert externe Komponenten in einem Docker-Container hochzufahren. Der komplette Lifecycle des Containers wird dabei von dem Framework verwaltet.

Um die Integration zu einem externen Service zu testen, können wir einen Mock-HTTP-Server wie WireMock nutzen. Wir nehmen unseren Client und ändern die Basis-Adresse, um Anfragen an den Mock-Server zu senden. Dieser gibt dann vordefinierte Werte zurück.

Component-Tests testen unseren Microservice als Ganzes, wobei externe Abhängigkeiten ersetzt werden. Wir ersetzen sie einfach wie in den Integration-Tests durch Mocks oder Ähnliches. Anschließend fahren wir in dem Test unsere Anwendung hoch und schicken Anfragen auf diese. Die zurückgegebenen Werte oder die in die Datenbank geschriebenen Werte vergleichen wir anschließend mit unseren Erwartungen.

Am Ende haben wir alle Elemente unseres Microservice getestet. Wenn

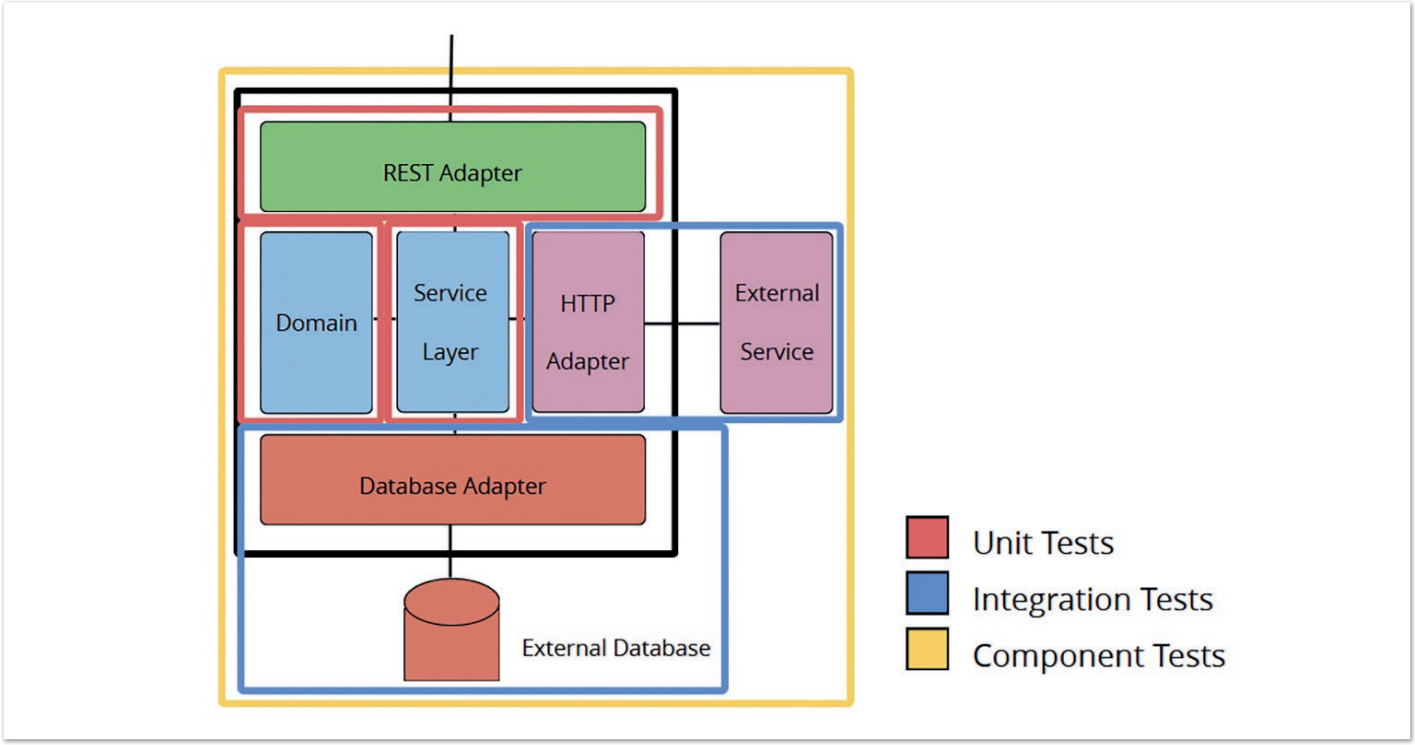


Abbildung 1: Aufbau eines Microservice mit Ports-and-Adapter-Pattern. Die farbigen Kästen geben an, wie die einzelnen Komponenten getestet werden können. (© Jens Knipper)

möglich, wurde versucht, die einzelnen Module isoliert zu testen. Die Integration mit externen Abhängigkeiten wurde getestet und dass der Service als Ganzes, inklusive des Zusammenspiels der einzelnen Komponenten, funktioniert, wurde mithilfe eines Component-Tests geprüft.

Als kleine Randnotiz: Würden wir jetzt alle Arten von Tests entsprechend ihrer Menge gruppieren, würden wir die Testpyramide erhalten.

Einführung in Lambda-Funktionen

AWS Serverless Lambda-Funktionen sind kleine, simple Funktionen, die ausgeführt werden können. Getriggert werden diese durch Events, die frei definiert werden können. Alternativ kann auch auf AWS-spezifische Events, wie ein Datei-Upload zu S3 (Simple Storage Service), zurückgegriffen werden.

Der Name Funktion ist dabei treffend. Es wird kein Webserver oder Ähnliches zum Ausführen benötigt. Es muss lediglich ein Interface implementiert werden, das eine Methode zum Behandeln des Events enthält. Das Ganze kann dann als zip, in Java als jar, zu AWS hochgeladen werden. Zudem gibt es noch weitere Möglichkeiten, ein Deployment durchzuführen, die eine bessere Automatisierung ermöglichen. Diese sollen hier aber nur Rande erwähnt werden.

Nach dem Upload kümmert sich AWS komplett um den Betrieb des Lambdas. Um Dinge wie Skalierung muss man sich keine Gedanken machen. Die Funktion wird bei Last automatisiert hochskaliert und bei weniger Events entsprechend herunterskaliert. Am Ende muss man nur die Laufzeit bezahlen. Bei wenig Last wird es so entsprechend günstiger.

In meinem aktuellen Projekt setzen wir Lambdas für asynchrone Tasks im Stile von „fire and forget" ein. Ein anderer Anwendungsfall ist das Rausziehen einzelner Funktionen aus Microservices, um eine bessere Skalierung zu ermöglichen.

Testen von Lambdas

Die Frage ist nun: Warum eignet sich der Test-Ansatz von Microservices auch für Lambdas? Mit Lambdas lassen sich tatsächlich auch Microservices bauen! Auch die klassische Architektur, das Ports-and-Adapter-Pattern, ist problemlos anwendbar. Im Gegensatz zu Microservices sind Lambdas allerdings in ihrer Größe beschränkt. Durch das von AWS bereitgestellte Framework sind die Beschränkungen offensichtlicher, beispielsweise kann ein Lambda immer nur ein bestimmtes Event verarbeiten. Das Verarbeiten unterschiedlicher Events ist nur über Umwege möglich.

Eine Teststrategie, wie bei Microservices, scheint also naheliegend. Das bedeutet auch, dass wir unsere bekannten Werkzeuge weiterverwenden können. Dazu gehören Dependency Injection, Testcontainers und WireMock. Da wir uns in einer neuen Umgebung befinden, kommen wir um neue Tools nicht herum. Im AWS-Umfeld ist LocalStack *[3]* eine besonders große Hilfe. LocalStack ist ein komplett lokal funktionierender Cloud-Stack, mit dem man Services und Lambdas von AWS lokal laufen lassen kann.

Beispiel-Use-Case

Die Umsetzbarkeit des Verfahrens lässt sich am besten anhand eines Beispiels erklären. Dafür stellen wir uns die folgende Situation vor: Wir möchten ein Thumbnail von einem Bild generieren, weshalb wir das Bild auf den AWS Filestorage S3 hochladen. Anschließend sprechen wir das Lambda an und befehlen ihm, ein Thumbnail daraus zu erstellen. Das Lambda lädt also das Bild von S3 herunter, skaliert es und schiebt das verkleinerte Bild anschließend wieder in den Filestorage. Als Rückmeldung bekommen wir vom Lambda die Information, wo wir das Bild herunterladen können.

Das ist eine recht simple Anwendung, also machen wir uns daran, diese umzusetzen. Wie bereits erwähnt, müssen wir ein Interface als Einstiegspunkt für das Lambda implementieren, um Events zu verarbeiten *(siehe Listing 1)*. Wir erstellen also das Interface, definieren Input und Output und implementieren die geforderte Methode. Generell versuche ich, diese Klasse so klein wie möglich zu halten, um einzelne Komponenten unabhängig vom Framework testen zu können. Die einzige Ausnahme ist der Konstruktor der Klasse, in dem die Services erstellt werden. Durch Nutzung von manueller Dependency Injection ist es am Ende möglich, die Komponenten isoliert voneinander zu testen. Bei komplexeren Lambdas kann der Konstruktor schon mal ein bisschen größer werden und eine baumartige Struktur annehmen, wobei der *EventHandlingService* die Wurzel bildet, die alle anderen Komponenten vereinigt.

Wir greifen nicht auf vorgefertigte Events zurück und definieren den Input und Output des Lambdas selbst *(siehe Listing 2)*. Als Input benötigen wir die Information, wo das Bild zu finden ist. Dafür reichen der Name des S3 Bucket und der Dateiname. Bei der Klasse handelt es sich um ein simples POJO mit Getter- und Setter-Funktionen, die der Übersicht halber hier ausgelassen werden. Records können zurzeit leider nicht genutzt werden, da maximal Java in Version 11 von AWS unterstützt wird. Analog dazu ist der Output. Auch hier finden wir den Namen des Bucket und den Dateinamen. Diese referenzieren allerdings das generierte Thumbnail.

Der *EventHandlingService* wird ausgeführt, nachdem das Lambda aufgerufen wurde. Hier ist die eigentliche Logik zu finden *(siehe Listing 3)*. Das zu verkleinernde Bild wird mit dem *S3Service* heruntergeladen und durch den *ImageService* verkleinert. Anschließend werden das Thumbnail hochgeladen und die Details, wo es zu finden ist, in Form des Outputs zurückgegeben. Auch hier ist der Code wieder ein bisschen vereinfacht. Zwischen den Schritten geschieht die Umwandlung zwischen *InputStream*, Bild und *OutputStream*. Dies ist für das Verständnis allerdings nicht weiter relevant.

Der *S3Service* ist recht simpel. Er nutzt das AWS-SKD, um Dateien hoch- und herunterzuladen. Ebenso der *ImageService*, der AWT-Funktionen benutzt, um das Bild auf eine festgelegte maximale Größe zu skalieren. Aber ehrlich gesagt kümmern uns die Interna der Services gar nicht – wir wollen sie nur testen! Dazu genügt es, die Inputs und zu erwartenden Outputs zu kennen.

Testen des Use-Case

Machen wir uns als Erstes an den Test für den *ImageService* *(siehe Listing 4)*. Der Service hat keine externen Abhängigkeiten und kann deshalb einfach per Unit-Test getestet werden. Wir initialisieren den Service einfach mit einer vorgegebenen maximalen Größe für die Thumbnails. Anschließend laden wir ein Bild aus dem Dateisystem, geben es dem Service und erwarten, dass das Bild auf die entsprechende Größe skaliert wurde. Wie der Service intern arbeitet, ist uns dabei egal, solange er die Bilder auf unsere vorgegebene Größe skaliert.

```
public class EventHandler implements RequestHandler<Input, Output> {
  private final EventHandlingService eventHandlingService;

  public EventHandler() {
    final S3Service s3Service = new S3Service(...);
    final ImageService imageService = new ImageService(...);

    this.eventHandlingService = new EventHandlingService(s3Service, imageService);
  }

  @Override
  public Output handleRequest(final Input input, final Context context) {
    return eventHandlingService.handleEvent(...);
  }
```

Listing 1: Aufbau des EventHandler zum Erstellen von Thumbnails

```
public class Input {
  private String bucket;
  private String key;

  ...
 }
```

Listing 2: Inhalt des eingehenden Events des Lambdas

```
public class EventHandlingService {
  private final S3Service s3Service;
  private final ImageService imageService;

  public EventHandlingService(final S3Service s3Service, final ImageService imageService) {
    this.s3Service = s3Service;
    this.imageService = imageService;
  }

  public Output handleEvent(final String bucket, final String fileKey) {
    final InputStream inputStream = s3Service.getObject(...);
    ...
    final BufferedImage newImage = imageService.resize(...);
    ...
    final String link = s3Service.uploadFile(...);
    return new Output(...);
  }
}
```

Listing 3: Aufbau des EventHandlingService. Hier ist die gesamte Logik der Anwendung zu finden

```
class ImageServiceTest {
  private final int maxDimension = 300;
  private final ImageService imageService = new ImageService(maxDimension);

  @Test
  void shouldResizeImage() throws IOException {
    // given
    final File testImage = new File("src/test/resources/image.png");
    final BufferedImage image = ImageIO.read(testImage);

    // when
    final BufferedImage resizedImage = imageService.resize(image);

    // then
    assertThat(resizedImage.getHeight()).isLessThanOrEqualTo(maxDimension);
    assertThat(resizedImage.getWidth()).isLessThanOrEqualTo(maxDimension);
  }
}
```

Listing 4: Unit-Test für den ImageService

```
@Testcontainers
class S3ServiceTest {
  @Container
  private static final LocalStackContainer localStack =
      new LocalStackContainer(DockerImageName.parse("localstack/localstack"))
          .withServices(LocalStackContainer.Service.S3)
          .withEnv("DEFAULT_REGION", Region.EU_CENTRAL_1.toString());

  private static final AwsCredentialsProvider credentialsProvider =
      StaticCredentialsProvider.create(
          AwsBasicCredentials.create(localStack.getAccessKey(), localStack.getSecretKey()));
  ...
  private final S3Service s3Service =
      new S3Service(
          localStack.getEndpointOverride(LocalStackContainer.Service.S3), credentialsProvider);

  @Test
  void getObjectShouldReturnInputStream() throws IOException {
    // given
    createBucketAndFile("bucket", "file", "content");

    // when
    final InputStream result = s3Service.getObject("second-bucket", "file");

    // then
    final String resultAsString = new String(result.readAllBytes());
    assertThat(resultAsString).isEqualTo("content");
  }
  ...
}
```

Listing 5: Integration-Test für den S3Service. AWS-spezifische Komponenten werden hier lokal durch LocalStack ausgeführt

```
@Testcontainers
public class ComponentIT {
  ...

  @Container
  private static final LocalStackContainer localStack =
      new LocalStackContainer(DockerImageName.parse("localstack/localstack"))
          .withServices(Service.LAMBDA, Service.S3)
          .withNetwork(Network.SHARED)
          .withNetworkAliases(localstackNetworkAlias)
          .withEnv("LAMBDA_DOCKER_NETWORK", ((NetworkImpl) Network.SHARED).getName())
          .withFileSystemBind(
              new File("target/").getPath(), "/opt/code/localstack/target/", BindMode.READ_ONLY);
  ...

  @BeforeAll
  public static void beforeAll() throws IOException {
    createLambdaFunction();
  }

  @Test
  void testHappyPath() throws IOException {
    // given
    createBucket("bucket");
    uploadFile("bucket", "image.png", "src/test/resources/image.png");

    // when
    final InvokeResponse response = invokeLambda("{\"bucket\": \"bucket\", \"key\": \"image.png\"}");

    // then
    assertThat(response.statusCode()).isEqualTo(200);

    final String resultAsJson = new String(response.payload().asByteArray());
    final BufferedImage resizedImage = getImage(resultAsJson);
    assertThat(resizedImage.getHeight()).isLessThanOrEqualTo(MAX_DIMENSION);
    assertThat(resizedImage.getWidth()).isLessThanOrEqualTo(MAX_DIMENSION);
  }
  ...
}
```

Listing 6: Component-Test des eigenen Lambdas, durch Nutzung von LocalStack

```
private static void createLambdaFunction() throws FileNotFoundException {
  final Map<String, String> variables =
      Map.of(
          ...
          "S3_ENDPOINT_OVERRIDE",
          "http://" + localstackNetworkAlias + ":" + 4566;

  final CreateFunctionRequest request =
      CreateFunctionRequest.builder()
          ...
          .handler("de.jensknipper.lambdatesting.EventHandler")
          .packageType(PackageType.ZIP)
          .code(
              FunctionCode.builder()
                  .zipFile(
                      SdkBytes.fromInputStream(
                          new FileInputStream("target/testing-aws-lambdas-1.0.jar")))
                  .build())
          .environment(Environment.builder().variables(variables).build())
          .build();

  final CreateFunctionResponse response = getLambdaClient().createFunction(request);
}
```

Listing 7: Deployment des Lambdas in LocalStack

Der *S3Service* ist ein wenig komplizierter zu testen. In *Listing 5* ist ein beispielhafter Test dargestellt. Hier haben wir eine Abhängigkeit zum Filestorage S3 und müssen somit auf einen Integration-Test zurückgreifen. Deshalb kommt eines der neuen Tools zum Einsatz: LocalStack. Zusammen mit Testcontainers fahren wir einen zu S3 kompatiblen Filestorage lokal hoch. Dem *S3Service* müssen wir nur die lokalen Daten mitteilen, damit er sich dagegen und nicht gegen die AWS-Cloud verbindet. Dazu dient der sogenannte *EndointOverride*. Für den Test erstellen wir uns noch einen eigenen S3Client, um Dinge wie Setup und Verifikation unabhängig von dem Service, den wir testen wollen, durchführen zu können. Mit dem Client erstellen wir dann einen Bucket und eine simple Datei mit Text als Inhalt. Diese versuchen wir anschließend mit dem zu testenden S3Service herunterzuladen. Am Ende verifizieren wir, ob die Datei erfolgreich heruntergeladen werden konnte, indem wir die Inhalte vergleichen.

Zuletzt widmen wir uns dem Component-Test. Unsere Intention ist es hier, das Lambda einmal zu erstellen, hochzufahren, eine Anfrage an dieses zu schicken und den Rückgabewert zu verifizieren. Der Code ist vereinfacht in *Listing 6* dargestellt. Da wir auch hier von AWS abhängig sind, ist es nötig, LocalStack in Kombination mit Testcontainers zu nutzen. Zusätzlich zu S3 brauchen wir allerdings auch den Lambda-Service von LocalStack, um ein Lambda zu erstellen und aufzurufen. Dies definieren wir entsprechend im Container für LocalStack. Zudem muss ein Netzwerk aufgesetzt werden, damit das Lambda mit S3 kommunizieren kann. Es wird nämlich automatisiert durch einen entsprechenden Request in der Methode *createLambdaFunktion()* erstellt. Dazu muss unsere Anwendung vorher als jar-Datei gepackt sein. Das nötigt uns, ein bisschen in die Maven-Trickkiste zu greifen, um den Test erst auszuführen, nachdem die Anwendung gepackt wurde. Mit dem failsafe-Plug-in ist es möglich, Tests mit der Endung „IT" nach der entsprechenden Stage in Maven auszuführen.

Der Ablauf des Tests ist anschließend sehr simpel. Es wird ein Bild nach S3 hochgeladen, das Lambda mit Referenz zu dem Bild angestoßen und der Rückgabewert anschließend auf seine Größe kontrolliert.

Das Erstellen des Lambdas ist hier der Übersicht halber aus *Listing 6* entfernt worden, soll aber abschließend noch erläutert werden. Mittels des AWS-SDK wird ein Request zum Erstellen des Lambdas an LocalStack abgeschickt *(siehe Listing 7)*. In diesem muss angegeben werden, welcher *EventHandler* ausgeführt werden soll. Zudem muss die Anwendung in Form einer jar-Datei übermittelt werden. Eine Reihe von Umgebungsvariablen dient dazu, dass das Lambda sich auch mit LocalStack und nicht mit der AWS-Cloud verbindet.

Zusammenfassung

Wir haben gesehen, wie man den Code innerhalb eines AWS-Lambdas strukturieren kann, um bessere Testbarkeit zu ermöglichen. Dependency Injection hilft uns dabei, in Isolation zu testen. Vor allem Unit-Tests werden so in ihrem Aufbau deutlich simpler.

Anhand eines Integration-Tests konnten wir sehen, wie die Integration zu AWS-spezifischen Komponenten durch LocalStack und Testcontainers testbar wird. Wir können so zum Beispiel verifizieren, dass der Code zum Herunterladen von Objekten von S3 auch wirklich funktioniert. Das alles lokal, ohne ein Deployment in die Cloud durchzuführen.

Der Component-Test ist im Aufbau ein wenig komplizierter. Er ermöglicht es allerdings zu verifizieren, dass beispielsweise die selbst erstellten Events auch funktionieren. Zudem hilft es bei Updates des AWS-Frameworks. Der Test gibt einem die Sicherheit, dass das Lambda auch nach Updates fehlerfrei läuft. Das alles ebenfalls lokal, ohne ein Deployment in die Cloud.

Der Code wurde, um eine bessere Lesbarkeit zu ermöglichen, ein wenig vereinfacht und reduziert. Der gesamte Code, inklusive Tests, ist ebenfalls auf GitHub *[4]* verfügbar.

Quellen

[1] *https://martinfowler.com/articles/microservice-testing/*
[2] *https://alistair.cockburn.us/hexagonal-architecture/*
[3] *https://localstack.cloud*
[4] *https://github.com/JensKnipper/testing-aws-lambdas*

Jens Knipper
OpenValue Düsseldorf GmbH
jens@openvalue.de

Jens Knipper ist Software Engineer bei OpenValue in Düsseldorf.

Er arbeitet gerne in verschiedenen Umgebungen, Architekturen und mit unterschiedlichen Technologien. Die Erkenntnisse, die er dadurch gewinnt, versucht er durch Vorträge und Blog-Artikel mit anderen zu teilen.

Mayday, Mayday, Kunde spricht kein Java

Isabelle Rotter, FLAVIA IT-Management

Ein richtig gutes Projekt steht und fällt mit der Kommunikation zwischen allen Beteiligten, auch denen, die keine Ahnung von Software haben. Zum Glück hat die Psychologie mittlerweile eine Menge Studien zusammengetragen, die uns dabei helfen können, uns verständlich auszudrücken. In diesem Artikel erfährst du, wie du vom ersten Treffen an eine angenehme Grundhaltung schaffen kannst. Nach einem Ausflug in die Welt der Körpersprache lernst du die Self-Perception-Theory nach Darley Bem kennen, mit der du dein eigenes Mindset neu ausrichten kannst. Wir schauen uns an, was das Gehirn benötigt, um sich Informationen besser merken zu können, und wie wir die richtigen Worte finden, um unseren Sprach- und Verhaltenscode für Kunden zu decodieren.

Manchmal gibt es Projekte, die laufen einfach. Kunden wissen, was sie wollen. Tickets sind gut definiert. Kollegen sagen rechtzeitig Bescheid, wenn sie etwas brauchen. So ein Projekt hängt natürlich an mehreren Punkten, aber ein zentraler ist die Kommunikation. Mit einer guten Kommunikation steht und fällt das Projekt.

Vieles, was Entwickler/innen machen, ist nicht einfach greifbar und damit für Kunden oder teilweise sogar für andere, unerfahrene Kollegen unsichtbar. Als Resultat fehlt es oft an dem Verständnis für die Notwenigkeit des Testens, des Behebens von Bugs oder dafür, warum ein paar Zeilen Code länger brauchen als die vierfache Menge. Dabei liegt es (meistens) nicht an der Intelligenz des Gegenübers, sondern an dem Verständnis füreinander. Jemand, der bei einem Bug im besten Fall an ein Insekt denkt, wird vermutlich keine Personentage (PT) dafür aufwenden wollen. Bugs sind noch harmlose Worte, die nicht gut verstanden werden. Bei Mock-Ups („to mock up", zu Deutsch „jemanden verspotten"), Wireframes und REST-API gehen die Warnlampen an und das Gegenüber schaltet schnell auf Durchzug oder geht in den Abwehrmodus, weil unser Hirn eine steinzeitliche Schutzfunktion geschaltet hat.

Wer Interesse an der Psychologie hinter patzigen Antworten hat, kann jetzt hier weiterlesen, ansonsten einfach die nächsten drei Absätze überspringen.

Eventuell hast du schon einmal von dem Fight-or-Flight-Modus gehört. Eine Funktion, die Jahrtausende lang unser Überleben gesichert hat. Wenn ein Reiz (etwa ein Wort, Bild oder eine Emotion) von uns verarbeitet wird, gibt es zwei Wege, den schnellen und den langsamen. Der langsame Weg funktioniert so, dass wir uns Zeit nehmen, etwa über das Wort „Mock-Up" nachzudenken. Wir *überlegen*, ob es uns aus anderen Kontexten bekannt vorkommt oder ob wir es einfach überspringen können und trotzdem den Kontext verstehen, was im Idealfall zum Erkennen führt. Nach dem Überlegen *bewerten* wir erst, wie „gefährlich" etwas ist. Diese beiden Vorgänge finden in verschiedenen Teilen des Hirns statt. Das erste Überlegen findet sinnbildlich hinter der Stirn im Frontal-Cortex statt. Das Bewerten hingegen tief im Hirn, im Angstzentrum (Amygdala). Dieser Teil des Gehirns wird auch Reptilienhirn genannt *(siehe Abbildung 1)*.

Nachdenken ist anstrengend und kostet viel Energie. Wir brauchen die Kapazitäten, Zeit und Motivation dafür. In vielen Kommunikationssituationen haben wir das nicht. Projektkoordinatoren wollen schnelle Antworten und Kunden wollen nicht als dumm dastehen. Also arbeitet unser Hirn auf Hochtouren und kürzt diese Wege ab, indem es direkt bewertet. Wir springen also ohne Zeitverlust ins Angstzentrum, stellen fest, wir kennen „Mock-Up" nicht und lösen als Konsequenz eine Abwehrreaktion aus.

In vielen Bereichen des Lebens macht das auch Sinn. Wenn jemand in einem dunklen Parkhaus schnell etwas aus der Tasche zieht und auf uns richtet, ist es gut, wir ducken uns aus Angst vor einer Pistole, auch wenn es am Ende nur seine Autoschlüssel sind. Lieber einmal zu viel geduckt als einmal zu wenig. Das Gehirn schaltet dann automatisch in den *Fight or Flight Mode*. Wir fliehen vor der Gefahr durch Ducken im Flight-Modus. Alternativ, wenn der Gegner sehr nah steht, könnten wir ihn auch angreifen und die Pistole aus der Hand schlagen. In diesem Fall sind wir im Fight-Modus. In der Projektkommunikation ist das allerdings eher hinderlich.

Leider passiert es oft, dass Kunden ohne Softwarevokabular in diesen Modus übergehen. Negative Vorerfahrungen, dezente Überforderung, die aufgrund der Position nicht offengelegt werden darf, oder andere Umstände erschweren die Kommunikation gewaltig. Das Ziel sollte es also sein, Zeit, Raum und Kapazitäten zum Überlegen und Mitkommen zu liefern. Ansonsten bestimmt dieser Modus nämlich unser gesamtes Denken, Einstellungen und Erinnerungen an das Meeting.

Das Ganze verdanken wir (letzte psychologische Theorie, um das wirklich nachzuvollziehen, versprochen) der Self-Perception-Theory nach Darley Bem. Nach seiner Theorie, die in vielen Experimenten

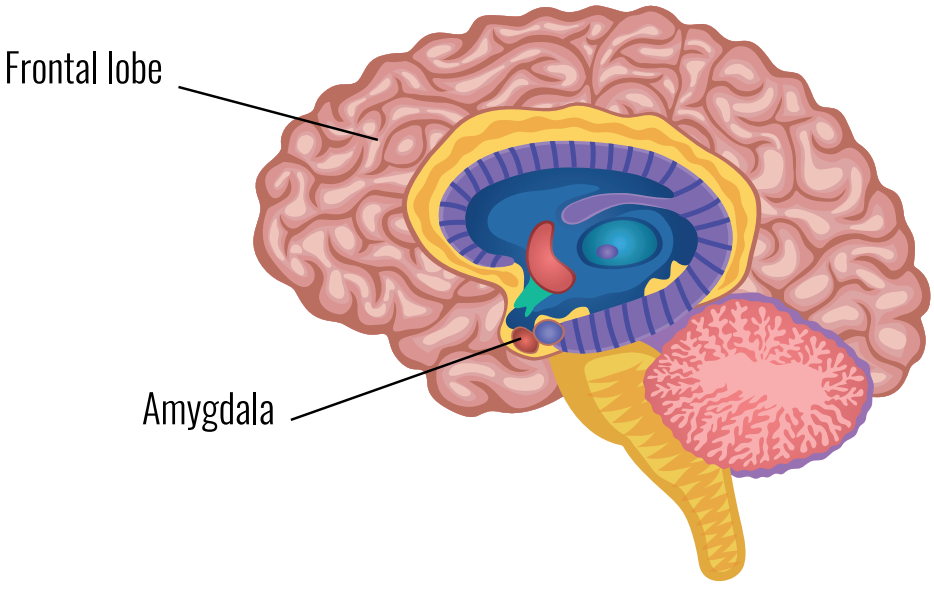


Abbildung 1

Abbildung 2

weiterentwickelt und für sinnvoll erachtet wurde, schließen wir auf unser Gefühl ähnlich wie auf die Gefühle anderer in unsicheren Situationen: Wir beobachten die Körpersprache. Sind wir verspannt, straffen die Schultern oder gehen wir nach der Arbeit mit einem schmerzenden Rücken nach Hause, schließen wir darauf, wie es uns gegangen ist. Wir waren anscheinend in stressigen Situationen, mussten tough wirken und haben viel gearbeitet. Denn das sind Dinge, die normalerweise mit diesen Körpersymptomen verknüpft sind. Negativ ausgedrückt führt es dazu, dass wir uns bedroht fühlen, wenn wir einem Gespräch nicht ganz folgen können, und in den Fight- oder den Flight-Modus wechseln. Positiv ausgedrückt führt es dazu, dass wir tatsächlich selbstbewusster sind, wenn wir aufrecht stehen. Um hieraus den größten Nutzen zu ziehen, ist es wichtig, sein Publikum zu kennen.

Know Your Audience

Die Leitfrage für die nächsten Zeilen Text (und hoffentlich Code) ist: Was kann ich tun, damit mein Gegenüber mich möglichst gut versteht, ohne Probleme? Know Your Audience beschreibt ein Konzept, von dem vielleicht der ein oder andere schon einmal gehört hat, namens „*Theory of Mind*". Das ist eine Fähigkeit, die wir mit zirka drei Jahren erlernen und die beschreibt, dass man sich in die Gedanken anderer hineinversetzt. Wir können dann verstehen, dass andere nicht unbedingt das Gleiche wissen wie wir *(siehe Abbildung 2)*. Dass sie unter Begriffen wie Mock-Up vielleicht keine schöne, ordentliche Struktur vor Augen haben, sondern die Panik in ihnen hochsteigt. Bei erfolgreichem Einsatz der Theory of Mind ist dem Gegenüber bewusst, was in dem eigenen Kopf vorgeht. Das können wir ganz aktiv nutzen – vor und während eines Gesprächs. Wir können Worte wählen, die der andere mit hoher Wahrscheinlichkeit besser versteht.

Halten wir nochmal kurz fest, was wir bis jetzt wissen:

- Nutzen wir komplizierte Begriffe, fällt das Verarbeiten schwer.
- Unser Gehirn fängt dann automatisch an, entweder wegzurennen oder dagegen anzukämpfen, statt dazuzulernen.
- Wenn wir uns in den anderen hineinversetzten und bekannte Worte wählen, können wir diesen Prozess verhindern.

Diese Erkenntnis ist für die meisten vermutlich nicht neu. Andere verstehen nicht das Gleiche wie ich. Aber im Alltag vergessen wir das ganz gerne und nutzen trotzdem komplizierte Fachwörter oder werden möglicherweise frustriert, wenn jemand kein Budget für das Testen freigeben will, weil wir nicht wissen, dass er nicht weiß, was testen bedeutet. Ab jetzt geht es um handfeste Tricks, diese Kommunikation von komplizierten Dingen zu erleichtern.

Grundhaltung

Um dem Gegenüber genug Werkzeuge und Kapazitäten zum Nachdenken mitzugeben, ist es essenziell, eine angenehme Grundhaltung zu schaffen. Hier kommt es wirklich darauf an, was machbar ist. Im Onlinegespräch ist ein freundliches Lächeln deutlich leichter zu beeinflussen als ein gemütlicher Stuhl, ruhige Arbeitsumgebungen oder hübsche Raumdeko (ihr lacht jetzt vielleicht, aber Pflanzen, Tageslicht und dezente Farben haben mehr Einfluss auf unsere kognitiven Urteile, als man glauben mag, Mojtahedzadeh et al. 2021 *[1]*). So oder so gibt es einige Punkte, auf die ihr schon vor dem Gespräch achten könnte:

- Genügend Zeit (sagt, der Termin dauert 30 % länger, damit ihr lieber früher Schluss macht, als am Ende zu kürzen)
- Ein freundliches Lächeln zur Begrüßung
- Der Hinweis, dass Fragen erlaubt sind (gibt es die offizielle Einladung, kein Experte zu sein, fällt es leichter, Unwissenheit zuzugeben, egal auf welcher Ebene)
- Entspannte Körperhaltung (entspannte Nacken, Kiefer und Schultern signalisieren uns meist unbewusst, dass unser Gegenüber in einer entspannten Stimmung ist, was auch uns beruhigt)

Ein Bild sagt mehr als tausend Worte

Wir sind jetzt so weit, dass unser Gegenüber sich Zeit genommen hat, halbwegs entspannt ist und sich der Situation entsprechend wohl fühlt. Nun geht es ans Erklären mit richtigen Worten oder besser: Bildern. Um das besser zu verstehen, machen wir einen kurzen Ausflug in unser Hirn.

Soweit wir bis jetzt wissen, ist unser Gehirn wie ein assoziatives Netzwerk organisiert. Es gibt keine einzelnen Anwendungen, son-

dern alles hat irgendwo eine Schnittstelle. Manches ist in Legacy Code geschrieben und wir können uns nur schwammig erinnern, ohne in die Dokumentation wie Fotoalben aus der Kindheit zu gucken. Anderes wurde gerade erst letzte Woche hinzugefügt oder gehört zu Standardbefehlen, die man einfach auswendig kennt. Das Gehirn ist auch bei Bodybildern der Muskel, der am meisten Energie braucht. Deshalb priorisiert es und baut als Erstes an den Stellen weiter, die viele Schnittstellen haben.

Zurück in unserem Meeting wollen wir jemandem etwas über unsere Software erzählen, der noch nie eine Zeile Code gesehen hat. Fangen wir mit Fachwörtern an, geht er nicht nur in den Fight-or-Flight-Modus, er kann auch einfach nichts mit diesen Informationen anfangen, weil es keine Schnittstellen gibt. Das können wir dadurch ändern, dass wir unsere Probleme auf andere Bereiche übertragen und des Zuhörens wert machen (so funktionieren übrigens auch Teambildungsmaßnahmen, man nehme Erlerntes aus einer angenehmen Gamification-Domäne und schaffe ein neues REST-API mit bestehenden Team-Problemen, um auf alte Lösungen zurückzugreifen). Metaphern und gute Vergleiche sind hier einfach Gold wert, und die kann man sich sogar schon vorher überlegen.

Das geht durch ganz kleine Dinge wie Benennungen: „Bug Nummer 1987" klingt deutlich abstrakter und abschreckender als „Tatort 1987". Wir haben hier eine Leiche gefunden, jetzt geht es darum, herauszufinden, wer der Mörder ist und ihn zu verhaften aka zu fixen. Warum wir einen Bug fixen, ist eventuell nicht sehr einleuchtend, einen Mörder zu finden hingegen schon.

Gleiches gilt fürs Testen. Der Kunde sieht entweder neun PT, die für das manuelle oder bestenfalls automatisierte Testen gezahlt werden sollen. Alternativ kann er dabei aber auch Grundpfeiler sehen, die unseren Code stützen. Ein Hochhaus (aka Code) würde auch niemand ohne Statiker (aka Test) bauen, weil es dann einsturzgefährdet ist. Mit einem ordentlichen Test ist das Ganze einfach belastbarer und das mit Sicherheit.

Metaphern bieten unserem Hirn mehr Schnittstellen, weil es uns schon bekannt vorkommt. Bekanntes ist nicht gefährlich. Das lassen wir viel lieber in unser Netzwerk rein als ein herrenloses Stückchen Code, von dem wir nicht wissen, wo es hingehört. Die Schnittstellen haben noch einen weiteren wundervollen Vorteil.

Merkbar

Verschiedene Eindrücke lassen sich unterschiedlich gut verarbeiten. Unsere Ohren, Augen, Introspektion oder auch Gedanken sind nicht alle gleich schnell, weil sie durch verschieden Hirnareale müssen und dabei verschiedene Informationen herausgefiltert, hinzugefügt und transformiert werden. Bilder sind dabei in der Regel die stärksten Signale. Wir können visuelle Informationen deutlich besser aufnehmen als etwa ein gesprochenes Wort, weil die Verarbeitung leichter und kürzer ist. Sie müssen nicht von Lauten in Worte separiert werden und dann in Bedeutung umgeschrieben. (Es gibt schon einen Grund warum die meisten von uns früher auf dem Schulhof bei der Frage „Welchen Sinn würdest du am liebsten behalten?" auf das Sehen bestanden haben.)

Haben wir eine gute Metapher, ist also die Chance, dass wir uns merken, was der andere erzählt hat, deutlich höher. In manchen Meetings hat man das Gefühl, seit drei Wochen jedes Mal das Gleiche zu besprechen. Das liegt häufig daran, dass sich die beteiligten Personen nicht gemerkt haben, was der letzte Beschluss war. An dieser Stelle kann ein Protokoll helfen, dass die Bilder mitnimmt. Entweder kann die Metapher beschrieben werden oder noch besser, aufgemalt. Es gibt viele gute Websites, die einfache Piktogramme kombinieren, um komplexe Sachverhalte darzustellen. Diese werden schnell wiedererkannt und können leichter zugeordnet werden als die fachlich korrekteren Beschreibungen.

Natürlich geht das nicht immer. In einem Großkonzern, in dem alle Anzug tragen, jedes Protokoll hochseriös ist und alles kleinlich genau mit DIN-Norm angegeben wird, ist eine Zeichnung im Protokoll eher ungünstig. Aber lasst eurer Kreativität hier freien Lauf. Vielleicht fällt euch doch etwas ein, das helfen kann. Ein Notizzettel kann hier auch Wunder wirken.

Was du jetzt gelernt haben solltest

Die Key-Take-aways lassen sich vermutlich in einem kleinen Satz zusammenfassen: Entspannt Metaphern aufzeichnen.

Das war nur ein kleiner erster Schritt oder eine Auffrischung in guter Kommunikation. Es gibt noch tausend andere Dinge, die einem dabei helfen, seinen Punkt rüberzubringen. Das Wichtigste ist, dass man sich selbst dabei wohl fühlt und natürlich macht auch hier Übung den Meister. Ein letzter Hinweis der Warnung: Was bei anderen funktioniert, funktioniert meistens auch bei einem selbst. Wenn ihr merkt, ihr kommt dauernd angespannt aus Meetings und könnt es euch nicht merken, fragt euch eventuell, ob ihr über nicht Verstandenes einfach hinweggeht oder es vorschnell als Blödsinn abstempelt.

Ich bin gespannt auf all die neuen Metaphern, die euch zu euren Problemen einfallen, und wünsche viel Spaß in den nächsten Meetings.

Referenzen

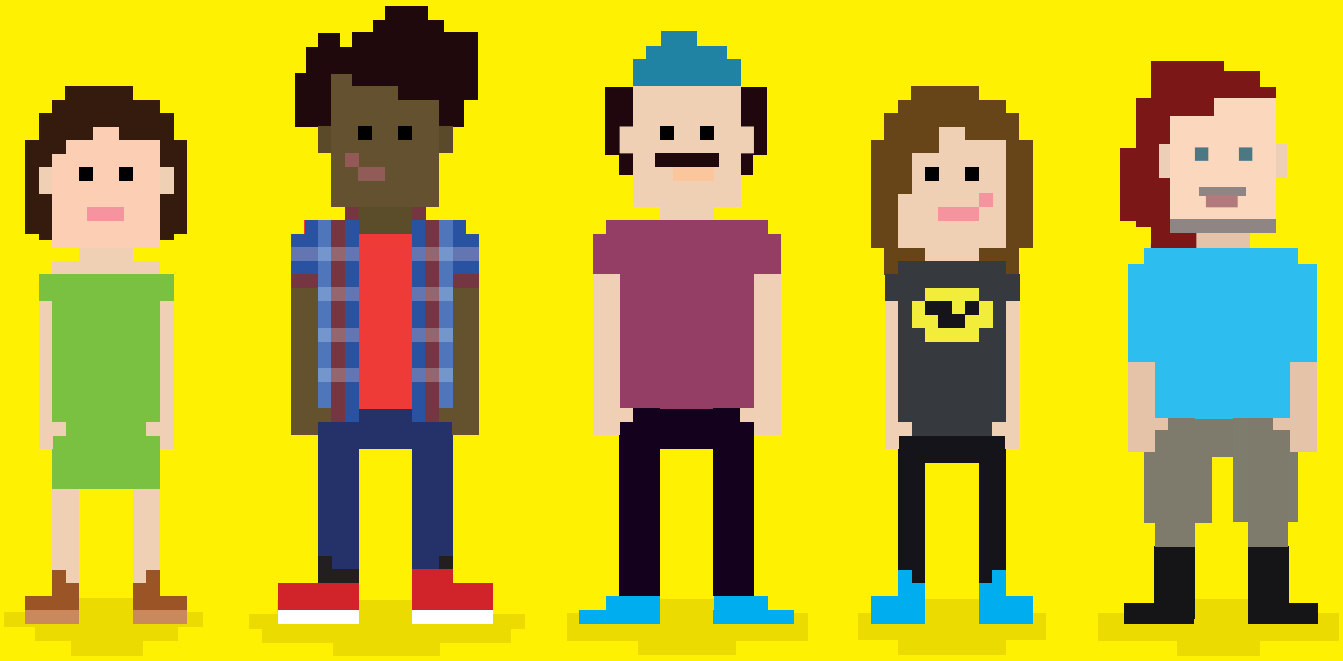
[1] Mojtahedzadeh, N., Rohwer, E., Lengen, J. et al. Gesundheitsfördernde Arbeitsgestaltung im Homeoffice im Kontext der COVID-19-Pandemie. Zbl Arbeitsmed 71, 69–74 (2021). *https://doi.org/10.1007/s40664-020-00419-1*

Isabelle Rotter
FLAVIA IT-Management GmbH
Isabelle.rotter@flavia-it.de

Isabelle Rotter hat Psychologie studiert und ist jetzt im HR bei FLAVIA tätig. Dort unterstützt sie Teams mit aktuellen Forschungsergebnissen bei der Kommunikation innerhalb und außerhalb des Projekts. Wenn sie nicht grade Vorträge hält oder Weiterbildungen besucht, ist Isabelle beim Eiskunstlaufen oder Inlineskaten zu finden.

HAST DU ES SCHON MIT EINEM NEUSTART VERSUCHT?

Come and join the CI Crowd.



Hier neu starten:
cologne-intelligence.de/jobs

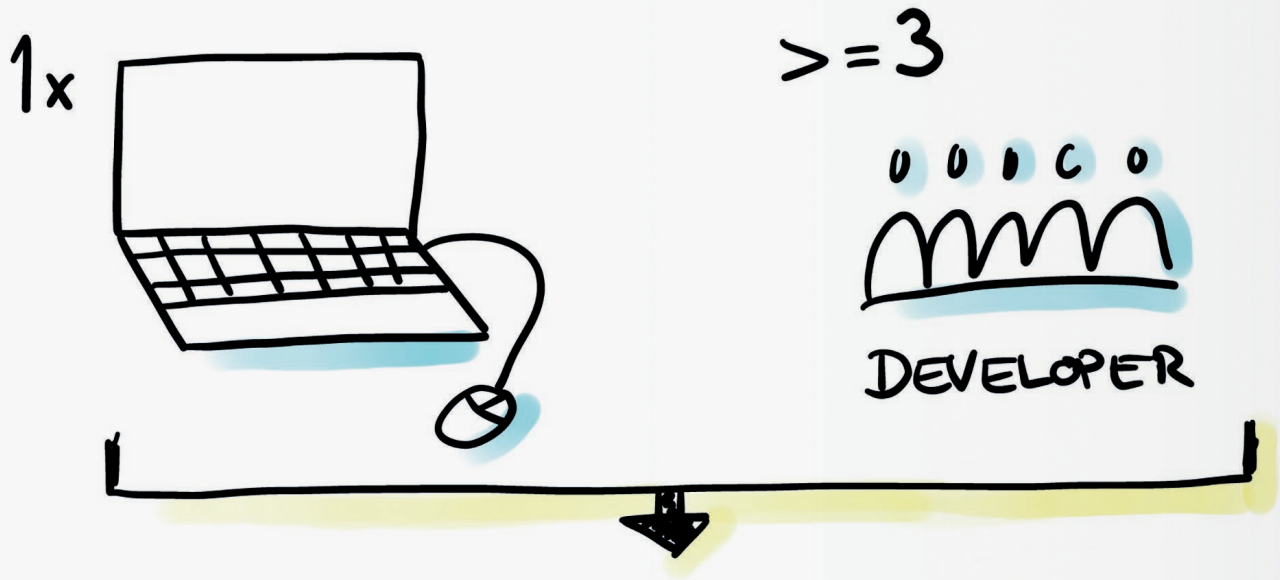
COLOGNE INTELLIGENCE

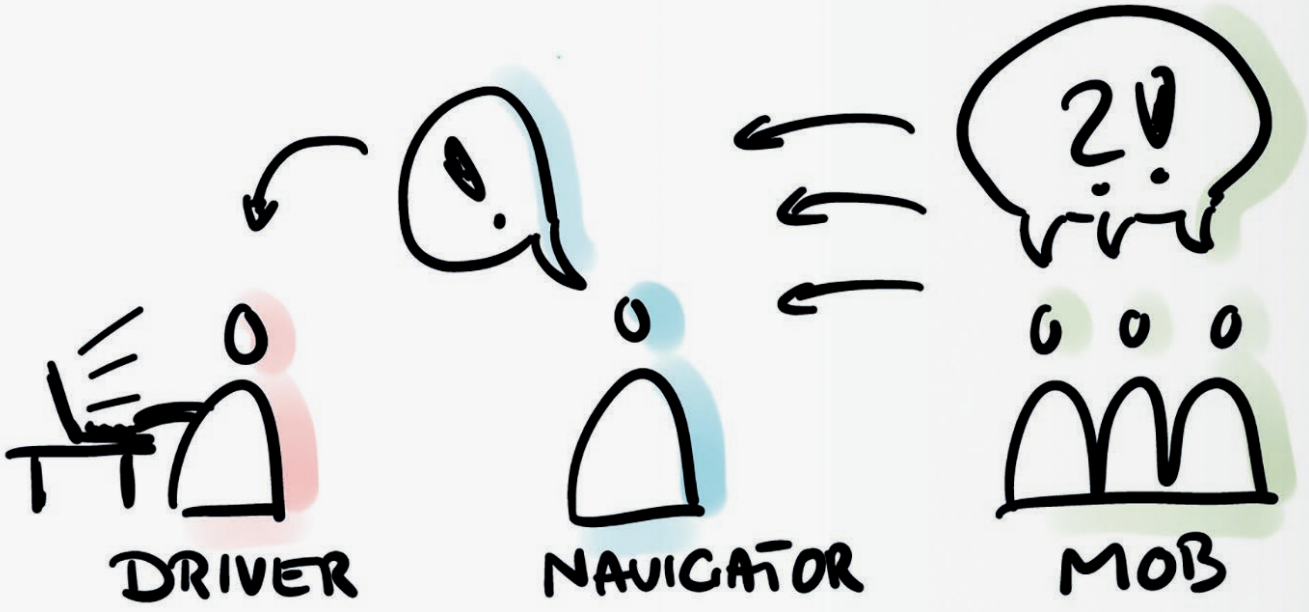
Wir haben jetzt Mob Programming. Wir sind gerettet. Oder etwa nicht? – Ein Erfahrungsbericht

Florian Schneider, codecentric

Mob Programming, oder mittlerweile auch öfters unter Ensemble oder Team Programming bekannt, ist eine Methode, die Developer-Teams nutzen, um Wissenstransfer im Team zu forcieren. Aber was genau ist Mob Programming und wie funktioniert es?

MOB PROGRAMMING IN A NUTSHELL





Vor meinen ersten Mob-Programming-Sessions hatte ich das Glück, einen Workshop mit Woody Zuill zu besuchen, Agile Coach und Mob-Programming-Urvater. Einer seiner ersten Sätze im Workshop war dieser:

> *„All the brilliant people working on the same thing, at the same time, in the same space, and on the same computer."*

Daraus geht schon hervor, was die Hauptidee von Mob Programming ist: gemeinsam an einem Computer arbeiten. Dabei wird eine Aufgabe nach der anderen mit dem gesamten Entwicklerteam bearbeitet. Der Vorteil an dieser Methode ist, dass jeder im Team gemeinsam das Wissen um die Aufgabe und Lösung aufbaut. Hinzu kommt, dass jeder im Team dabei auch Entscheidungen über Lösungswege mitbekommt und mitträgt, die während der Erarbeitung getroffen werden.

Wie genau funktioniert Mob Programming?

Mob Programming liegt die Idee zugrunde, dass sich das Entwicklungsteam um einen Computer versammelt und gemeinsam an diesem Computer an einer Aufgabe arbeitet. In Zeiten von Remote Work ist dieser Aspekt mit einem Computer obsolet. Dazu später mehr.

Die ideale Mob-Programming-Teamgröße ist zwischen drei und sechs Personen. Eine Person nimmt die Rolle des Drivers ein, eine die Rolle des Navigators und der Rest ist Teil des Mobs. Die Rollen werden während einer Mob-Programming-Session regelmäßig zwischen den Teilnehmenden getauscht.

Die Person an der Tastatur ist der Driver. Man kann diese Person auch als smartes Input-Device sehen. Der Driver ist die einzige Person, die während seiner Rolle nicht an der Lösungserarbeitung teilnehmen sollte.

Der Driver wird nun von einer Person aus dem Mob navigiert, wobei es hier verschiedene Szenarien gibt, die ein Team ausprobieren kann, um zu sehen, welche Vorgehensweise sich als ideal herauskristallisiert. Daher ist es sinnvoll, im Team mit verschiedenen Ansätzen zu experimentieren.

Die navigierende Person sollte möglichst mit High-Level-Kommandos den Driver zu seiner Idee beziehungsweise Lösung führen. Dabei ist die Kommunikation extrem wichtig. Der Navigator sollte laut denken, um den Mob in seinen Gedankengang und seine Ansätze mit einzubeziehen, was sich positiv auf die Rollenwechsel auswirken wird. Solange es geht, sollte die navigierende Person dabei auf ein „Diktieren der Lösung" verzichten. Also statt zum Beispiel zu sagen „nimm die Liste X. Tippe Punkt. Nimm die Methode foreach. Schreibe Y …", kann er sein Kommando eher wie folgt ausdrücken: „Iteriere bitte über die Liste und mache dabei mit jedem Element dies oder das…"

Rollenwechsel und Wechselzeitpunkt

Es gibt verschiedene Ansätze, die Rollen innerhalb des Teams während der Mob-Programming-Session zu wechseln. Dabei kann auch der Zeitpunkt entscheidend sein, wann ein Rollenwechsel stattfindet:

- Das Team kann sich für einen Timebox-basierten Wechsel entscheiden. Dabei wird festgelegt, wann ein Wechsel passieren soll. Das kann nach drei, fünf oder 15 Minuten erfolgen. Auch hier entscheidet das Team und sollte ausprobieren, was am besten passt. Zu lange Timeboxen können es für den Mob anstrengend machen, die Konzentration aufrechtzuerhalten und zu kurze Timeboxen können den Fortschritt vermindern. Hierbei ist wichtig, dass man sich auf einem Mob-Branch befindet, weil die WIP(work in progress)-Commits einen instabilen Stand der Software zur Folge haben können.

- Das Team kann bei einem stabilen Stand der Software einen Wechsel machen, begleitet von einem sauberen Commit. Bei diesem Vorgehen kann das Team dem Trunk-based-Development-Ansatz folgen.

- Ein Wechsel kann auch mit jeder Erfüllung eines Tasks beziehungsweise einer Story durchgeführt werden. Hier kann ein ähnlicher Nachteil wie beim Wechseln mit Timeboxen entstehen, wenn die Aufgabe zu groß ist und somit zu lange dauert. Ideal für diesen Ansatz sind sehr kleine Tasks/Stories.

Auch für den Wechsel selbst gibt es verschiedene Szenarien, einen solchen zu vollziehen. Diese können sich wie folgt gestalten:

- **Driver-Navigator-Mob** *(siehe Abbildung 1)*: Der Driver wird zum Navigator, der Navigator tritt dem Mob bei und der Nächste vom Mob wird zum Driver. Der Vorteil bei diesem Wechsel-Szenario ist, dass der Driver aktiv an der Lösungserarbeitung beteiligt war und somit leichter bei den Ideen seines Navigators ansetzen kann.

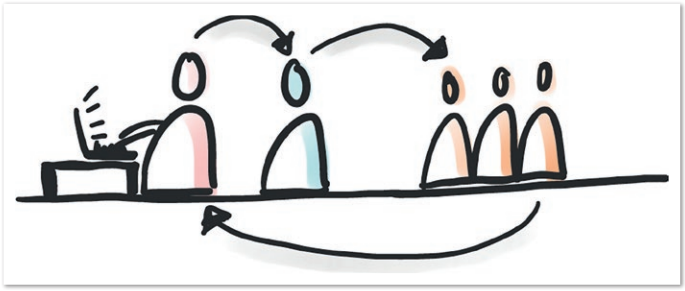


Abbildung 1

- **Navigator-Driver-Mob** *(siehe Abbildung 2)*: Der Navigator wird zum Driver, der Driver tritt dem Mob bei und der Nächste vom Mob wird zum Navigator. Bei diesem Wechsel kann sich ein Nachteil ergeben. Im Falle, dass der neue Navigator, der aus dem Mob kommt, abgelenkt war, wird es schwerer für ihn, weiter am Lösungsweg anzuknüpfen. Die Gefahr besteht, dass der neue Driver, zuvor Navigator, das Ruder übernehmen will, was der Idee des Mob Programming widerspricht.

Abbildung 2

- **Driver-Mob-Navigator-Mob** *(siehe Abbildung 3)*: Der Driver und der Navigator treten dem Mob bei und die zwei Nächsten des Mobs werden zum Driver und zum Navigator. Auch hier überwiegen die Nachteile, die es verhindern, ideal an die Ideen der Vorgänger anzuknüpfen.

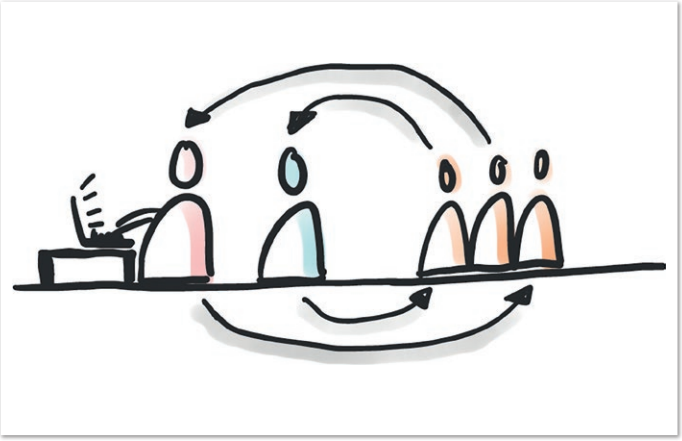


Abbildung 3

Und jetzt auch noch remote

Spannender wird das ganze Thema Mob Programming, wenn die Teams auf verschiedene Standorte verteilt sind. Dabei gibt es diverse Herausforderungen zu meistern. Zum einen das räumliche Beieinander. Es ist schwieriger über Remote-Arbeit, wenn man die anderen Teilnehmenden des Mobs nicht sehen kann, Reaktionen aufnehmen kann etc. Ein Minimum, das jedes Mitglied einer Mob Session tun sollte, ist, seine Webcam einzuschalten, um die Atmosphäre eines gemeinsamen Raumes zu schaffen. In einem dauerhaften Projekt ist hierbei ein gutes Setup zu empfehlen, wie etwa eine gute Webcam, ein externes Mikrofon oder ein gutes Headset.

Ein weiteres Thema ist der Wechsel der Rollen. Was on-site mit einem Platztausch passiert, muss nun virtuell vollzogen werden. Dabei sollte auf einem Mob-Branch gearbeitet werden, wenn der Stand der Software größtenteils instabil ist. Eine Weitergabe kann über das eingesetzte Versionierungstool erfolgen. Zum Beispiel mithilfe eines kleinen Git-Shell-Skriptes *(siehe Listing 1)*, das alle Änderungen aufnimmt, ein WIP-Commit erstellt und es in das entsprechende Repository pusht (alternativ können CLI Tools wie mob.sh verwendet werden).

```
#!/bin/bash
control=$1
git add .
git commit -m "WIP"
git push
```

Listing 1

Ein großer Vorteil von Remote Mob Sessions ist, dass jeder Teilnehmende in der Umgebung seiner Wahl und vor allem mit seiner Konfiguration (zum Beispiel IDE-Shortcuts etc.) arbeiten kann, worauf sich ein On-site-Team zunächst einigen muss.

Mit Mob Programming sind wir doch nun gut gerüstet

Nachdem wir nun wissen, was Mob Programming ist, kann man erahnen, welche Chancen, aber auch Herausforderungen Mob Programming mit sich bringen kann.

Chancen durch Mob Programming

Ein enormer Vorteil, der sich aus On-site- oder Remote-Mob-Programming ergibt, ist Knowledge Sharing. Das Wissen über die Software, die Fachlichkeit oder Entscheidungen werden im Team ideal verteilt, sodass sich der Bus-Faktor erhöht. Der Bus-Faktor ist ein Indikator dafür, wie gut das Wissen im Team verteilt ist. Im Idealfall ist der Bus-Faktor gleich der Anzahl der Teammitglieder. Wenn der Bus-Faktor hoch, also annähernd gleich der Teamgröße ist, spielt es für das Team keine Rolle, wenn mal eine oder sogar mehrere Personen für einen gewissen Zeitraum ausfallen (beispielsweise durch Urlaub, längere Krankheit etc.).

Ein weiterer Vorteil ist die entstehende Qualität. Erinnern wir uns kurz an das eingehende Zitat von Woody Zuill: „All the brilliant people working on the same thing,...." Durch gemeinsames Voranschreiten entstehen im Mob Ideen und weitere Lösungswege, die in der Session besprochen und angegangen werden. Viele Bugs werden schon im Vorfeld beseitigt, da viele wachsame Augen auf die Qualität der Software blicken. Auch in den Situationen, in denen man sich für einen Lösungsweg entscheiden muss, ist das gesamte Entwicklerteam involviert. Entscheidungen können zum Beispiel in ADRs (Architecture Decision Records) festgehalten werden, um auch für Teammitglieder-Rotationen die Transparenz aufrechtzuerhalten, warum ein bestimmter Weg gegangen wurde oder um gegebenenfalls auf eine Alternative zurückzugreifen, falls sich der eingeschlagene Weg als nicht perfekt entpuppt.

Je mehr Mob-Programming-Sessions ein Team miteinander hat, desto eingespielter und effizienter wird es. Das wird sich neben der steigenden Qualität auch in der Geschwindigkeit widerspiegeln. Zudem besteht die Chance, dass Junior-Teammitglieder durch die Senior-Entwickler*innen auf ein höheres Skill-Level kommen. Durch Sicherheit wächst auch der Spaßfaktor.

Herausforderungen von Mob Programming

Eine der größten Herausforderungen ist der Faktor Mensch. In einem Team treffen unterschiedlichste Charaktere aufeinander, mit unterschiedlichem Wissens- und Skill-Level. Daher ist es wichtig, in einem Team, das ernsthaft Mob Programming ausprobieren oder nutzen möchte, einen sicheren Hafen zu schaffen. Die Forschung hat gezeigt, dass psychologische Sicherheit ein starker Prädiktor für die Teamleistung ist. Es ist ein angenehmes Gefühl, sagen zu können, was man denkt, und zu sein, wer man ist, ohne sich zu schämen oder Auswirkungen befürchten zu müssen. Das ist ein entscheidender Punkt für Mob Programming. Jedes Teammitglied muss jederzeit in der Lage sein zu sagen: „Das verstehe ich nicht." oder „Ich komme hier nicht weiter."

Eine weitere Herausforderung, die aufkommen kann, sind Stimmen (zum Beispiel von Vorgesetzten, aber auch von Teammitgliedern), die Mob Programming als nicht effizient bezeichnen und behaupten, dass es eine Verschwendung von Personen sei und dass es schneller sei, wenn jeder parallel eine Aufgabe bearbeitet. Aber ist das wirklich schneller?

Angenommen, das Team besteht aus fünf Personen. Jeder arbeitet an einer Aufgabe. Wenn die Aufgabe abgeschlossen ist, muss eines der anderen Teammitglieder ein Review machen, was, wenn es ordentlich durchgeführt wird, auch eine gewisse Zeit in Anspruch nimmt. Gibt es dann Unstimmigkeiten oder findet die reviewende Person eine Ecke, die geändert oder angepasst werden muss, fängt das Review-Ping-Pong an. Der Code wird korrigiert oder umgestaltet und es muss wieder ein Review gemacht werden. Zwar kürzer als das erste, aber auch diese kleinen kosten Zeit. In diesem Doing waren nun zwei der fünf Teammitglieder involviert. Vielleicht hätten die anderen noch etwas entdeckt. Die Chance besteht, dass sich doch noch ein Fehler eingeschlichen hat. Das ist ein Vorteil von Mob Programming: Die Anzahl der Bugs im Nachgang ist sehr gering.

Dann machen wir jetzt nur noch Mob Programming und alles wird gut?

Ein kurzer Spoiler vorweg: nein. Wir haben Chancen, aber auch Herausforderungen kennengelernt. Wie auch andere Methoden hat Mob Programming seine Vor- und Nachteile. Aus der Erfahrung heraus ist es eine sehr gute Methode, um beispielsweise bei neuen Features gemeinsam die Lösung zu erarbeiten und dabei etwa Fragen wie Architektur oder Logik zu klären. Das Wissen ist dann ideal im Team verteilt und jeder kann an dieser Ecke ansetzen oder eine Wartung vornehmen.

Mob Programming sollte nicht als Allheilmittel angesehen werden und krampfhaft ab sofort für jedweden Task eingesetzt werden. Das wird auf Dauer nicht gut gehen. Dafür sind Herausforderungen, wie der zeitliche Aspekt, zu schwer zu lösen. Stattdessen kann Mob Programming eine gute und ergänzende Methode im Alltag eines Entwicklerteams sein.

Aus der Erfahrung heraus haben wir in den Teams Mob Programming dann genutzt, wenn wir mit neuen, kritischen oder komplexen Aufgaben konfrontiert waren. Der Vorteil, wie oben bereits erwähnt, ist, dass ein Großteil des Entwicklerteams die Lösungen für diese Aufgaben direkt mitbekommen hat. Das spart die sogenannten „Aufgleisungen" zum späteren Zeitpunkt. Dabei kann es eine gute Idee sein, den Task oder die Story zunächst gemeinsam für den „Happy Path" zu implementieren. Weitere Negativtests, beziehungsweise mehr Tests zur Robustheit, können später von Einzelpersonen oder Pairs ergänzt werden. Dies beispielsweise ist ein Task, den man während der Erarbeitung erstellen und in ein gesondertes Backlog packen kann (gegebenenfalls Taggen mit „single" oder „pair"). Der Vorteil davon ist, dass nicht mehr alle involviert sein müssen, weil diese Ergänzungen gut von einer Person oder einem Pair erledigt werden können. Auch wird die zeitliche Komponente aufgeweicht. Das heißt, im Team ist beispielsweise eine Person, die früh den Tag beginnt, wenn vielleicht noch keine andere Person aus dem Team da ist. Diese Person kann sich dann einen Task aus dem Single-Pair-Backlog nehmen und daran weiterarbeiten. Sobald eine weitere Person den Tag startet, werden diese Personen ein Pair. Kommen weitere dazu, können wieder Tasks oder Storys angegangen werden, die im Mob erledigt werden sollten. Beim Schließen des Tages dann genau andersherum. Jemand geht früher und wenn noch eine oder zwei Personen länger da sind, können diese wieder die Ergänzungsaufgaben angehen. Grundsätzlich solltet ihr versuchen, viel Mob Programming und Pair Programming zu nutzen und so wenig wie möglich im Einzelkämpfer-Modus unterwegs zu sein. Identifiziert die Tasks, die für „single" geeignet und aufgrund des Knowledge Sharing nicht kritisch sind (zum Beispiel Tests ergänzen, Dummy-Daten erstellen und so weiter).

Da Mob Programming anstrengend ist und man sich als Team erst daran gewöhnen muss, sollte man zu Beginn Time-Boxed Sessions machen. Gewöhnt euch als Team an diese Methode, indem ihr feste Sessions für ein bis zwei Stunden ausmacht. Nutzt diese Zeit auch zum Experimentieren, zum Beispiel mit Übergaben (time vs. stable), Tooling (Timer, Git), Driver-Navigator-Pattern und so weiter. Ihr müsst für euch als Team einen geeigneten Modus finden, der für alle Beteiligten passt, wobei jeder Einzelne zumindest ein Commitment geben sollte, Mob Programming auch ausprobieren zu wollen. Vor einer Session ist ein kurzes Briefing mit allen Teilnehmenden sinnvoll, damit jeder eine Grundidee hat, wo die Reise bei der Bearbeitung der Aufgabe hinführt. Mit mehr Übung werdet ihr sehen, dass ihr effizienter und qualitativ bessere Ergebnisse liefern werdet und gleichzeitig auf langatmige Übergabe- und Review-Prozeduren verzichten könnt. Happy Mob Programming!

Florian Schneider
codecentric AG
florian.schneider@codecentric.de

Florian ist seit 2018 für codecentric am Standort Frankfurt tätig. Sein Schwerpunkt liegt im Java-Backend-Umfeld und agiler Softwareentwicklung. Er ist ein großer Fan von Clean Code und Refactoring. Aktuelle Interessen umfassen aber auch Frontend-Technologien, Kafka und diverse Cloud-Themen. Ansätze wie YAGNI und Mob-Programmierung sind seine Leidenschaft.

APEX connect

by DOAG

3. - 4. Mai 2023

IN BERLIN

PROGRAMM ONLINE

apex.doag.org

DOAG

Muss ich als Einsteiger in die Softwareentwicklung alle Technologien kennen?

Pauline Schulze, cronn

Diese Frage stellen sich viele, die im Studium oder in der Ausbildung irgendetwas mit der IT oder Softwareentwicklung zu tun haben und nach ihrem Abschluss vor dem Einstieg in die Berufswelt stehen. Unternehmen werben mit 20 Icons von Technologien und damit, wie modern ihr Techstack sei, und man selbst ist froh, wenn man das von Java, MsTeams und vielleicht GitLab erkennt.

So erging es mir auch, als mein Weg in die Softwareentwicklung begann. Im Jahr 2019 habe ich nach sechs Jahren das Studium mit einem Master in Mathematik abgeschlossen. Danach ging es zuerst ins Projektmanagement, aber irgendwie reizte die Softwareentwicklung mich doch sehr. Aber hatte ich wirklich das Zeug dazu?

Im Studium hatte ich Fächer wie Softwareentwicklung 1 und 2 und aus der Schule kannte ich neben Java noch ein wenig Python. Aber von Spring oder Docker hatte ich noch nie etwas gehört, manche der Icons auf Unternehmenswebseiten hatte ich vorher noch nie gesehen.

Auf der Suche nach der richtigen Stelle stolperte ich dennoch immer wieder über Stellenanzeigen, die in etwa so aussahen wie die in *Abbildung 1*.

Wenn das zu meinen Aufgaben gehört, dann sollte ich diese Technologien auch schon kennen, oder?

Die Antwort, wie ich sie erlebt habe, ist ein klares „Nein".

Ich würde euch in diesem Artikel gerne einen Einblick darin geben, auf was in meinen Bewerbungen Wert gelegt wurde und warum ihr euch deshalb nicht abschrecken lassen solltet, euch in die Welt der Softwareentwicklung zu stürzen.

Fairerweise möchte ich aber an dieser Stelle erwähnen, dass ich natürlich nur von meinen persönlichen Erfahrungen berichten kann und darüber, wie dies bei meinem aktuellen Arbeitgeber gehandhabt wird. Der Markt in der Softwareentwicklung ist riesig und jedes Unternehmen hat leicht abweichende Anforderungen. Dennoch hoffe ich, ich kann euch mit diesem Artikel Mut machen und euch auf eurem Weg in die Softwareentwicklung bestätigen. Die Hürden dahin sind geringer, als man denkt!

Wenn ihr nicht alle Technologien kennen müsst, welche Punkte in Stellenanzeigen müsst ihr eigentlich beachten?

Zuerst wurde ich in Bewerbungsgesprächen nach meiner Ausbildung gefragt. Hier ging es darum, inwiefern mir diese theoretische Hintergründe und Denkweisen vermittelt hat, die in der Softwareentwicklung weiterhelfen.

Ich habe zum Beispiel Mathematik mit Nebenfach Informatik gehabt. Informatik passt natürlich perfekt zur Softwareentwicklung, aber auch Fächer wie Mathematik, Physik und Chemie sind sehr gern gesehen. Für diese Fächer benötigt man logisches Denken und Problemlösungskompetenz. Das sind Eigenschaften, die in der Informatik gesucht werden und in meinem Fall daher auf positives Feedback stießen. Das Mathestudium hat mir jedenfalls nicht dabei geholfen, Technologien zu kennen.

In Bewerbungsgesprächen bin ich oft danach gefragt worden, wie ich auf die Softwareentwicklung gestoßen bin oder woher meine Begeisterung dafür kommt.

Gerade bei meinem jetzigen Arbeitgeber ist intrinsische Motivation eines der wichtigsten Einstellungsmerkmale. Solch eine Motivation und Interesse an der Software sind natürlich gut durch ein passendes Studium oder persönliche Projekte erkennbar.

Es ist immer hilfreich, wenn ihr bereits eine Programmiersprache gut beherrscht. In dem Fall ist die Einarbeitung in neue Bereiche der Softwareentwicklung viel einfacher. Untermalen könnt ihr das Ganze auch durch bereits bestehende eigene Projekte. Vielleicht habt ihr im Studium gemeinsam mit Kommilitonen schon etwas aufgezogen?

Während meines Studiums haben wir im Fach Softwareentwicklung 2 häppchenweise ein Kinoticketsystem gebaut. Solche Erfahrungen in Projekten sind sehr gerne gesehen. Aber auch eigene Projekte außerhalb des Studiums solltet ihr nicht verheimlichen. Je mehr ihr euch aus eigenem Interesse mit Software auseinandergesetzt habt, desto glaubwürdiger wirkt euer Interesse auch im Bewerbungsgespräch.

Aber nicht nur die intrinsische Motivation ist ein sogenannter „Soft Skill", der häufig in meinen Bewerbungsgesprächen aufkam. Oft sind auch andere Eigenschaften gefragt.

Deine Aufgaben:

- Mitarbeit in agilen Projektteams und Mitverantwortung für den Erfolg des Teams
- Entwicklung geschäftskritischer, kundenspezifischer Softwarelösungen
- TDD, Pair-Programming, Code-Reviews
- Knowledge-Sharing, Craftsmanship, Continuous Improvement
- Entwicklung mit unterschiedlichen Technologien (zum Beispiel Java 11 und höher, JavaScript, TypeScript, Kotlin, Spring Boot, React, Gradle, Jenkins, Git, AWS, Docker, Kubernetes, JUnit, Selenium, Cucumber)
- Über den Tellerrand schauen und Dich weiterentwickeln

Wenn Du folgende Voraussetzungen mitbringst:

- Du hast Erfahrungen in der Server-/Backend-Entwicklung und/oder Frontend-/Web-Entwicklung
- Studium der Informatik, Wirtschaftsinformatik und/oder ausreichende Erfahrung als Software-Engineer
- Freude an modernen Stacks
- Clean Code und Software-Craftsmanship sind keine Fremdworte für Dich
- Deutsch und Englisch gehen Dir fließend über Finger und Zunge
- Teamplayer und selbstständiges Arbeiten sind keine Gegensätze für Dich
- Du schreibst Tests genauso gerne wie produktiven Code und refactorst gern

... dann wollen wir Dich kennenlernen!

Abbildung 1

Allen voran solltet ihr euch auf Fragen nach der Teamfähigkeit vorbereiten. Softwareentwicklung als Beruf bedeutet nicht, einsam in einer Kammer zu sitzen und zu programmieren. Oft arbeitet man in Teams, teilt sich Aufgaben oder arbeitet in einer Pair-Programming-Session zusammen – dazu später noch mehr.

Ihr merkt, dass das eigentliche Wissen über die Technologien bisher keine große Rolle gespielt hat. Allerdings denken sich die Firmen die Technologien in ihren Stellenanzeigen nicht aus. Früher oder später werdet ihr also vermutlich mit diesen arbeiten. Da ist es umso wichtiger, dass ihr Lust habt, euch einzuarbeiten und immer dazuzulernen. Grundsätzlich ist die Softwareentwicklung ein Feld, bei dem sich ständig alles bewegt. Jedes Jahr kommen neue Versionen von bekannten Programmiersprachen, aber es werden auch komplett neue Sprachen entwickelt. Insofern sind Ehrgeiz, Lernbereitschaft und Wissbegierde wichtige Eigenschaften, die ihr mitbringen solltet.

Bisher habe ich eher von sehr generischen Anforderungen geschrieben. Ich habe in meinen Bewerbungsgesprächen nicht ein Unternehmen erlebt, das nicht in irgendeiner Form auf diese Punkte eingegangen ist. Natürlich gibt es aber auch unternehmensspezifische Anforderungen.

Manchen Unternehmen ist zum Beispiel sehr wichtig, dass ihr euch für ihr Produkt interessiert. Bewerbt ihr euch etwa bei einem Online-Shop für Mode, kann es sein, dass auch auf euer Interesse an Mode eingegangen wird. Anderen ist es dagegen nur wichtig, dass ihr euch für die Softwareentwicklung interessiert. Daher solltet ihr herausfinden, was euch wichtig ist und welches Unternehmen dazu passt.

Ein passendes Unternehmen zu finden, ist eine Wissenschaft für sich. Deshalb möchte ich an dieser Stelle einen kleinen Exkurs machen und auf Personalvermittler, auch Headhunter genannt, eingehen.

Meine Stelle als Softwareentwicklerin habe ich über eine solche Personalberatung bekommen. Das Prinzip ist sehr einfach. Meistens ist einem eine kleine Anzahl an Headhuntern innerhalb einer Headhunter-Firma zugeordnet. Diese haben Kunden mit offenen Stellen und stellen diese dem Bewerber vor. Ihr könnt dann diese Unternehmensprofile anschauen und entscheiden, bei welcher Firma der Headhunter euch vorstellen soll. Ihr müsst dann selbst erst mal nicht mehr machen. Wenn das Unternehmen interessiert ist, bekommt ihr meist direkt Vorstellungstelefonate oder -gespräche vermittelt. Bezahlt werden Headhunter durch die Unternehmen.

Immer mehr junge Bewerber greifen auf Headhunter zurück. Es ist einfacher und komfortabler. Headhunter haben auch einen sehr großen Vorteil, wenn ihr euch unsicher seid, was ihr eigentlich sucht. Ihr könnt eure Vorstellungen nennen und bekommt eine kleine Jobberatung inklusive. Ich kann Headhunter für jeden empfehlen, der sich sehr unsicher ist und lieber noch etwas Rat und Tat braucht.
Mein Rat an euch lautet aber gleichzeitig: Traut euch auch, euch direkt bei Unternehmen zu bewerben!

Eine Einstellung über einen Headhunter kostet das Unternehmen sehr viel Geld, oftmals 30 % des Brutto-Jahresgehaltes, mit dem ihr dann einsteigt. Viel entscheidender für Unternehmen ist allerdings, dass diese Bewerber mit deutlich weniger Motivation und Interesse an dem Unternehmen zu der Bewerbung gelangt sind.

Wenn ihr euch selbst auf die Suche begebt und dann selbst bewerbt, zeigt ihr, dass ihr euch von allen Unternehmen dieses ausgesucht habt. Wenn ihr selbst viel Zeit in die Recherche möglicher Unternehmen steckt, werdet ihr auch ein viel besseres Gefühl dafür bekommen, was ihr möchtet.

Es gibt ganz viele verschiedene Eigenschaften von Unternehmen: Möchtet ihr eher viel Sicherheit und viele Benefits in einem Konzern? Oder eher aufregendes, zwangloseres Arbeiten in einem Start-up?

Versucht euch darüber klar zu werden, was ihr wollt und was zu euch passt. Damit werdet ihr auch in Bewerbungsgesprächen einen viel klareren Eindruck hinterlassen können.

Eine Bewerbung ohne Headhunter gibt euch aus den bereits genannten Gründen oft auch einen direkten Vorteil gegenüber anderen Bewerbern. Auch hier solltet ihr euch also bewusst werden, was am besten zu euch und eurer Situation passt. Traut euch, euch kritisch zu hinterfragen und eventuell auch Mehrarbeit bei der Bewerbung auf euch zu nehmen.

Wenn ihr es nun, ob mit oder ohne Headhunter, geschafft habt, einen Job als Softwareentwickler zu bekommen, geht es bei eurem neuen Arbeitgeber darum, den Einstieg richtig zu gestalten. Wie schon erwähnt, sind die Technologien und Aufgaben einer Stellenanzeige nicht ausgedacht. Insofern stellt sich die Frage, wie der Einstieg gelingt, ohne dass man als Berufseinsteiger das ganze Wissen mitbringt.

Bisher habe ich mich eher mit dem Werkzeugkasten beschäftigt, den ihr als Bewerber mitbringen sollte. Eure Ausbildung und Eigenschaften bieten euch die Grundlage, den Einstieg erfolgreich zu schaffen. Aber natürlich gibt es auch auf Seiten des Unternehmens sehr verschiedene Mechanismen und Prozesse, die euch den Einstieg erleichtern können. Auch hier ist es wichtig, dass ihr euch bewusst werdet, was ihr euch vorstellt.

Seid ihr eher der Typ, der ins kalte Wasser springen möchte? Oder klingt eine standardisierte Einführungsveranstaltung interessant?

Manche Firmen bringen Einsteiger direkt in Projekte und man lernt eher am Projekt. So war es auch bei mir. Gerade in größeren Firmen gibt es aber auch richtige Stundenpläne für den Anfang, mit denen man nochmal die theoretischen Grundlagen aufarbeitet, bevor es dann ans Projekt geht.

Weit verbreitet sind inzwischen Mentorenprogramme. Wenn ihr als Einsteiger im Unternehmen mit einem solchen Programm anfangt, bekommt ihr einen erfahrenen Mitarbeiter des Unternehmens zur Seite gestellt. Dieser ist dann Ansprechpartner in der ersten Zeit, meist ohne dabei Vorgesetzter zu sein.

Andere Bestandteile meines Onboardings waren sogenannte 1-1s (gesprochen: One-on-Ones). In diesen hatte ich alle paar Wochen fest geplante Gespräche mit meinem Tech-Lead. Bei meinem Arbeitgeber helfen diese sowohl bei fachlichen als auch bei organisatorischen Themen. In den fachlichen Gesprächen spricht man darüber, wie man mit der Einarbeitung vorankommt, wo man noch Hilfe benötigt oder was schon richtig gut funktioniert. In anderen

Gesprächen geht es darum, wie man sich in der Firma und im Team fühlt, was man für Ziele hat etc. Beide Seiten haben hier die Chance auf Feedback. Als Neueinsteiger kann man dem Unternehmen direkt zurückspielen, wie die Einarbeitung läuft. Aber auch ihr habt die Chance, kurze Feedbackschleifen zu bekommen und zu wissen, wo ihr steht.

Auch für die konkrete fachliche Einarbeitung verwendet mein Arbeitgeber bestimmte Methoden, wie etwa Pair-Programming und Code-Reviews. Mir haben sie sehr geholfen, auf eine spannende, praktische Art den Einstieg zu finden.

Im Pair Programming setzt man sich mit einem anderen Entwickler zusammen an ein Problem. Einer sitzt an der Tastatur und tippt, der andere steht mit Rat und Tat zur Seite, kann Recherche betreiben etc. Nach einer gewissen Zeit wechselt man die Rollen. Als Einsteiger programmiert man meistens zusammen mit einem erfahreneren Entwickler. Das bringt für beide Seiten Vorteile: Für euch als Einsteiger ist es spannend zu sehen, wie das Programmieren im beruflichen Feld abläuft. Man lernt, wie Techniken und Theorien in die Praxis umgesetzt werden. Aber auch nützliche Kleinigkeiten, wie sinnvolle Shortcuts am Rechner, kann man sich so abgucken.

Ihr müsst euch allerdings keine Gedanken machen, dass ihr die Einzigen seid, die von dieser Konstellation profitieren. Durch Nachfragen und euer frischeres theoretisches Wissen seid ihr gleichwertiger Bestandteil des Pair-Programming. So kann man sich direkt zu Beginn als Teil des Teams gut einbringen und auch schon an richtigen Aufgaben arbeiten.

Gerade bei kleinen Aufgaben kann es aber auch schnell passieren, dass man als Neuling allein programmiert. Um dabei ein Sicherheitsnetz zu haben, gibt es sogenannte Code-Reviews. Dabei wird ein „fertiger" Code eines Entwicklers von anderen Entwicklern gelesen und Feedback gegeben. Auf diese Art könnt ihr für eure eigenen Ideen direkte Rückmeldung bekommen. Zusätzlich könnt ihr so, ohne großes Risiko, schon schnell nach eurem Einstieg Code in Projekte einbringen.

Damit steht eurem erfolgreichen Einstieg in die Softwareentwicklung nichts mehr im Wege.

Zum Ende möchte ich nochmal die zugrunde liegenden, wichtigen Punkte dieses Artikels herauspicken und euch als Anregungen mitgeben:

- Lasst euch nicht von den Technologien abschrecken!
- Versucht, euch klar zu werden, was ihr könnt und was ihr wollt!
- Wenn möglich, bewerbt euch direkt!
- Nutzt das Bewerbungsgespräch, um herauszufinden, ob ihr und das Unternehmen zusammenpasst.
- Bildet euch weiter, seid wissbegierig!
- Habt Spaß und Interesse an der IT!

Pauline Schulze
cronn GmbH
pauline.schulze@cronn.de

Nach ihrem Mathematikstudium startete Pauline in einer kurzen Phase als Projektmanagerin im E-Commerce ins Berufsleben, wechselte aber bald in die Softwareentwicklung bei der cronn GmbH. Nach zwei Jahren als aktive Entwicklerin wechselte sie als Referentin der Geschäftsführung in den administrativen Bereich, wo sie heute das Leben und Arbeiten der Mitarbeiter, vor allem Entwickler, so angenehm wie möglich gestaltet. Auch bei Messen und Konferenzen ist sie häufiger anzufinden.

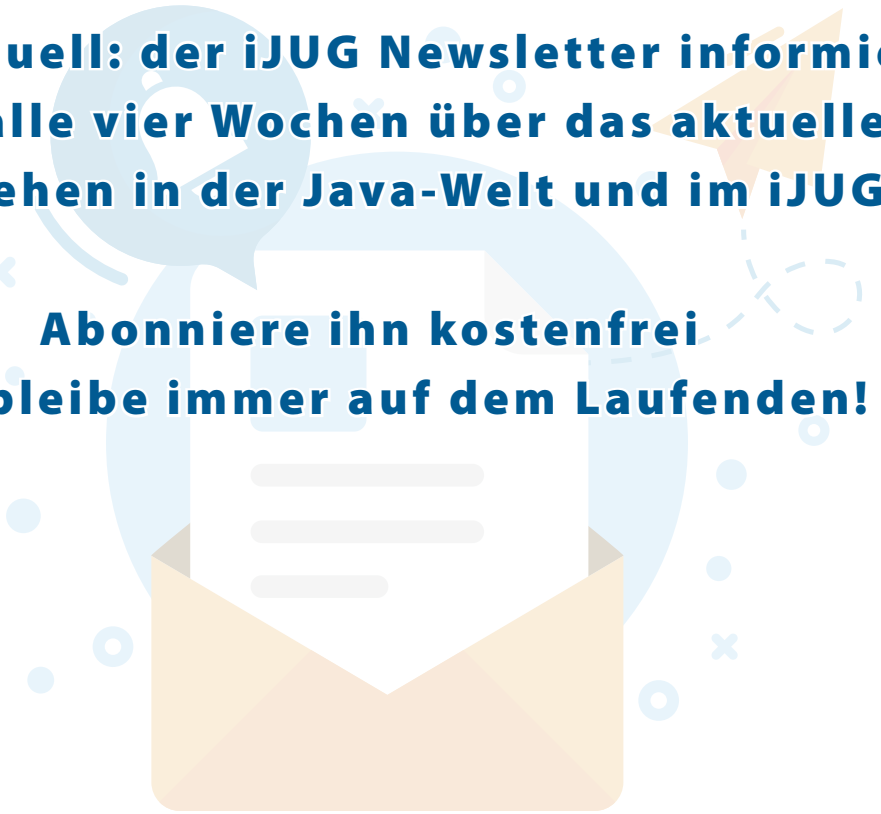
iJUG Verbund

NEWSLETTER

Anmeldung

Java aktuell: der iJUG Newsletter informiert dich alle vier Wochen über das aktuelle Geschehen in der Java-Welt und im iJUG.

Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!



https://shop.doag.org/newsletter/

oder nach dem Login mit euren Zugangsdaten direkt über euer Profil abonnieren.

Java Cloud Ready – Spring Boot für die Überholspur

Benjamin Klatt, viadee
Matthias Kutz, viadee

Java und Spring gehören zu den etabliertesten Technologien für Unternehmensanwendungen. In Cloud-Umgebungen stehen sie jedoch oft aufgrund von langen Startzeiten und hohen Ressourcenverbräuchen in der Kritik. Mit diesem Artikel möchten wir daher die Wolkendecke rund um Spring Native lichten und Potenziale und Herausforderungen aufdecken. Wir gehen auf das derzeit weitverbreitete Spring Boot 2 ein und erklären, was sich mit dem neuen Spring Boot 3 Stack verändert hat.

SPRING BOOT 3.0

Was ist die Motivation für Spring Native beziehungsweise Java im Cloud-Umfeld?

Cloud-Architekturen unterscheidet im Gegensatz zu klassischen Betriebsumgebungen insbesondere, dass Anwendungen robust und effizient mit Neustarts umgehen sollten. Zudem werden Ressourcenverbräuche explizit abgerechnet und bedeuten einen viel direkteren Kostenhebel.

In diesem Kontext ist Java mit seiner Just-in-Time(JIT)-Kompilierung und dem hohen Speicherverbrauch im Vergleich zu Skript-Sprachen wie TypeScript und Python schlecht aufgestellt.

Das Spring-Native-Projekt ist daher angetreten, hierfür eine Lösung zu finden und dank nativer Kompilierung mittels Ahead-of-Time(AoT)-Kompilierung aus Java nativen, sprich Betriebssystem-spezifischen, aber dafür deutlich effektiveren Maschinencode zu erzeugen. Mit der neuen Spring-Boot-Version 3 ist Spring Native fester Bestandteil des Frameworks geworden. Zunächst aber ein Blick auf die noch führende zweite Generation des Frameworks.

Potenzial des Spring-Native-Einsatzes

Das Einsparpotenzial von Spring Native ist enorm. Bei unseren eigenen Tests konnten wir Berichte über Einsparungen von 90 % im Vergleich zum ursprünglichen Verbrauch insbesondere am Anfang der Pod-Lebenszeit bestätigen.

Das in *Abbildung 1* gezeigte Diagramm zeigt den Speicherverbrauch von zwei Pods mit einer Java-Anwendung, die einmal klassisch (gelb) und einmal nativ (grün) kompiliert wurde.

Bei der klassischen Variante sieht man anfänglich einen Grund-Speicherverbrauch von fast 200 MB, während bei der nativen Variante nur gute 20 MB verbraucht werden. Über ein paar Stunden hinweg schwingt sich das Ganze so ein, dass die klassische Variante nur noch bei knapp 140 MB und die native Variante bei knapp 50 MB Memory Usage liegt. Eine signifikante Einsparung an Speicher-Verbrauch, die sich insbesondere bei kurzlebigen Pods bemerkbar macht.

Was steckt hinter Spring Native?

Das Spring-Native-Projekt gehört zum Spring-Ökosystem und hat sich das Ziel gesetzt, die GraalVM für Spring-Anwendungen leicht nutzbar zu machen.

Die GraalVM ist eine alternative JVM aus den Oracle Labs, die auch ein JDK inklusive eines Ahead of Time Compilers (AOT) mitbringt, der bereits zur Compile-Zeit alle Abhängigkeiten auflöst und plattformspezifischen Maschinencode erzeugt.

Die Auflösung zur Compile-Zeit steht Springs Dependency-Injection-Ansatz zur Laufzeit entgegen. Das Spring-Native-Projekt bietet Werkzeuge, um diesen Brückenschlag zu ermöglichen.

Wie nutzt man Spring Native im eigenen Projekt?

Die Basis ist recht schnell gelegt: In der POM wird Spring Native als Dependency eingebunden. Zu beachten ist nur, dass die Spring-Native-Version zu den verwendeten Java- und Spring-Versionen kompatibel ist *(siehe Abbildung 2)*.

Mit den Dependencies lassen sich bereits Annotationen hinzufügen – auf die wir später eingehen – und im Hintergrund läuft ab jetzt einige Unterstützung für die GraalVM-Integration.

Wir haben für den schnellen Start im Selbstversuch ein Beispielprojekt auf GitHub bereitgestellt *[2]*.

Interessant wird es nun, wenn es an die Kompilierung selbst geht.

Native Kompilierung in der eigenen Entwicklungsumgebung
Für die Kompilierung lassen sich drei Varianten unterscheiden:

- Nativ auf Linux oder Windows Subsystem for Linux (WSL)
- Nativ auf Windows
- Mit einem von GraalVM bereitgestellten Docker-Image

Wir haben mit den drei Varianten recht unterschiedliche Erfahrungen gemacht. Um den Gewinner direkt vorwegzunehmen: Am unkompliziertesten hat die Kompilierung mit dem bereitgestellten Docker-Image funktioniert.

Während die native Kompilierung unter Linux selbst recht gradlinig lief, hat es mit dem Windows Subsystem for Linux in einigen Fällen gehakt. Wovon wir nach unseren eigenen Erfahrungen abraten können, ist die native Kompilierung unter Windows. Zum Zeitpunkt des Schreibens ist es notwendig, unter Windows ein Visual Studio mit seiner Native-Developer-Konsole zu installieren, und selbst damit waren wir nicht erfolgreich.

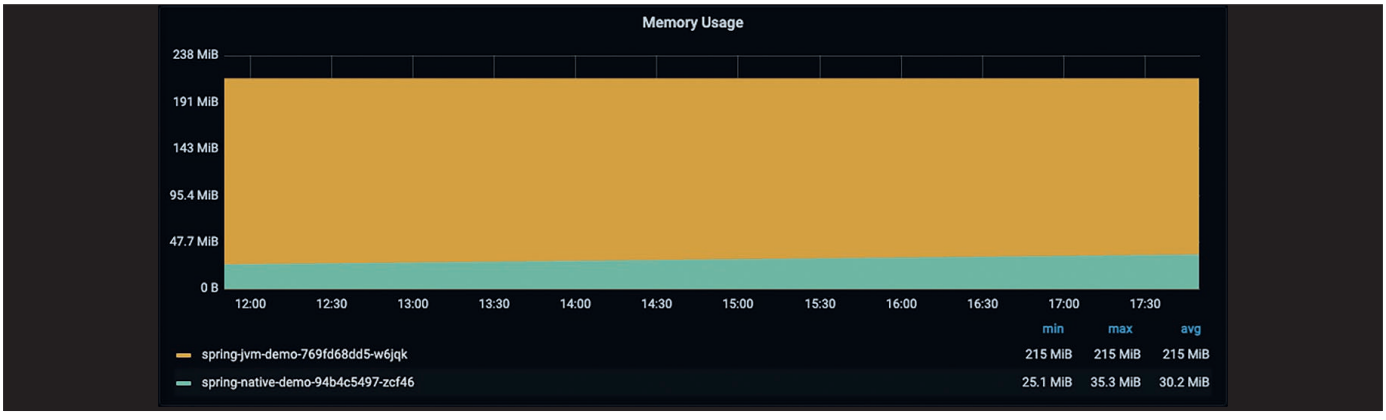


Abbildung 1: Speicherverbrauch der beiden Varianten, visualisiert mit Grafana

```
...
 <properties>
        ...
        <spring.native.version>0.12.0</spring.native.version>
        ...
</properties>
<dependencies>
        ...
        <dependency>
                <groupId>org.springframework.experimental</groupId>
                <artifactId>spring-native</artifactId>
                <version>${spring.native.version}</version>
        </dependency>
        ...
</dependencies>
...
```

Abbildung 2

Wie zuvor erwähnt, ist unsere Empfehlung, mit der Docker-Image-Variante zu arbeiten, wenn das die eigene Entwicklungsumgebung zulässt. Mit einer lokalen Docker-Umgebung kann das bereitgestellte Beispielprojekt einfach geklont und mit folgendem Befehl kompiliert werden:
`docker build -t spring-native-demo:0.0.1-SNAPSHOT -f src/main/docker/Dockerfile.native`.

Neben der Kompilierung per Dockerfile lassen sich die Spring-Native-Anwendungen auch mit einem Buildpack von Paketo als Docker-Image zusammenbauen. Hierzu agiert Paketo als Hilfsprogramm, um für bekannte Anwendungstypen (wie Spring Boot) eine einheitliche Docker-Build-Pipeline zur Verfügung zu stellen. Praktischerweise muss durch den Einsatz von Paketo kein eigenes Dockerfile entwickelt und gepflegt werden. Es reicht lediglich, Paketo seiner Spring-Native-Anwendung als Maven-Plug-in hinzuzufügen *(siehe Abbildung 3)*.

Entscheidend für den Build als Native-Image ist die Umgebungsvariable `BP_NATIVE_IMAGE`, die wir dem Plug-in mitgeben. Diese Variable teilt Paketo mit, dass wir für den Build-Prozess die GraalVM als Basis verwenden wollen.

Dadurch lässt sich das native Docker-Image für unser Beispielprojekt mit dem folgenden Maven-Goal von Spring-Boot erzeugen:
`./mvwn spring-boot:build-image`

Der Einsatz von Paketo für das Erstellen des nativen Docker-Image ist vom eingesetzten Betriebssystem und zum Teil von der installierten Software abhängig. Wie bereits erwähnt, empfehlen wir, das GraalVM-Docker-Image zu verwenden, um einen einheitlichen Workflow über alle Betriebssysteme hinweg sicherzustellen und somit Probleme mit Betriebssystem-spezifischer Konfiguration zu vermeiden.

```
...
<plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
      <image>
         <builder>paketobuildpacks/builder:tiny</builder>
         <env>
            <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
         </env>
      </image>
   </configuration>
</plugin>
...
```

Abbildung 3

Ein zusätzlicher Vorteil neben dem zuverlässigen Setup ist die Nähe zum langfristigen Entwicklungsprozess.

Was ändert sich im Entwicklungsprozess?

Das zuvor beschriebene Setup für die lokale Entwicklung relativiert sich recht schnell, wenn man den gesamten Entwicklungsprozess betrachtet. Fakt ist, dass bei der nativen Kompilierung die Einsparung im Betrieb auf Kosten des Build-Prozesses geht. Aus ein paar Minuten werden dann schnell ein paar Stunden. Dies spiegelt sich auch in unserem Beispiel wider, wenn man die Builds bei GitHub betrachtet *(siehe Abbildung 4)*.

Selbst für unser Minimalbeispiel, das bewusst auf viele Komponenten des Spring-Ökosystems verzichtet, steigt die Build-Time von 1:50 Minuten auf rund 7 Minuten an. Das entspricht einem Anstieg der Build-Time zum Faktor 3.87. Dieser Faktor kann, abhängig vom Einsatz der verwendeten Spring-Teilprojekte, signifikant ansteigen. Die wohl zu überlegende Frage ist nun, wann man im gesamten Entwicklungsprozess eine klassische und wann eine native Kompilierung nutzt.

Die langen Kompilierungszeiten machen die native Entwicklung unbrauchbar für lokale Entwicklungsumgebungen aufgrund der nicht sinnvollen langen Feedbackzyklen. Ebenso ist die native Kompilierung für viele Entwicklungsumgebungen zu ressourcenintensiv.

Auf der anderen Seite stellt sich die Frage, wie Build-Automatisierung und -Pipelines organisiert werden sollten. Beispielsweise gibt

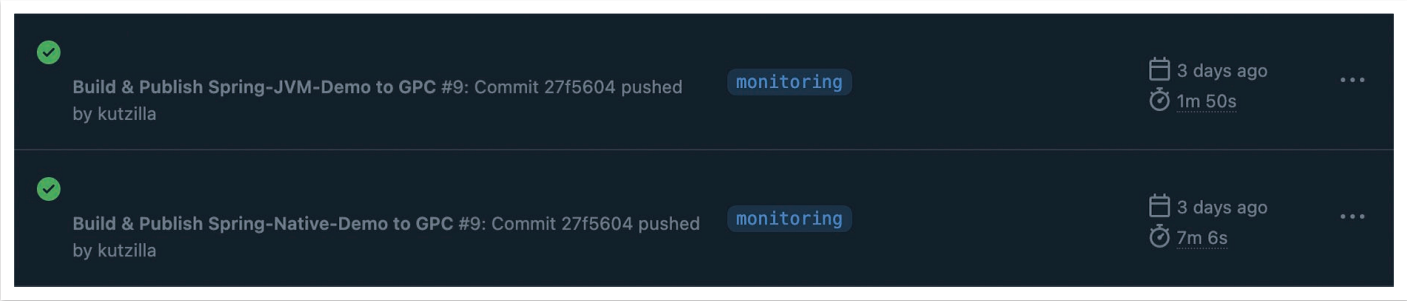


Abbildung 3

es immer wieder Code-Stellen, die der Compiler nicht von sich aus findet und bei denen unterstützende Code-Annotationen nachträglich zu ergänzen sind. Diese Stellen gilt es noch während der Entwicklung und nicht erst in Produktion zu identifizieren.

Fazit: Eine native Kompilierung macht lokal in den wenigsten Fällen Sinn und es gilt, vielmehr seine CI/CD-Pipelines entsprechend zu planen. Hierbei ist eine ausreichend hohe Testabdeckung notwendig, damit alle Code-Stellen identifiziert werden, die vom AOT-Compiler nicht automatisch gefunden wurden. Für diese Fälle bedarf es zudem Feedbackmechanismen, damit sie in der Entwicklung ergänzt werden können.

Die hier beschriebenen Herausforderungen treffen zu Teilen nicht nur auf Spring Native, sondern auch auf andere Frameworks mit ähnlicher Zielsetzung wie Micronaut und Quarkus zu. Letztere haben nur den Vorteil, dass sie die native Kompilierung als Grundkonzept verankert haben, während bei Spring viele Teilprojekte schon viel länger existieren und (noch) nicht explizit dafür vorbereitet wurden.

Wann und wie den Einstieg wagen?

Native Kompilierung und der Ahead-of-Time-/AOT-Ansatz sind nicht neu. Die GraalVM als Basis von Spring Native wird in den Oracle Labs schon seit mehr als einem Jahrzehnt entwickelt. Im Spring-Ökosystem gehört Spring Native hingegen zu den etwas jüngeren und insbesondere noch als experimental gekennzeichneten Projekten. Nichtsdestotrotz steckt in der Technologie so viel Potenzial, dass es sich aus unserer Sicht heute schon lohnt, sich damit auseinanderzusetzen. Insbesondere wenn beispielsweise die verfügbaren Ressourcen im eigenen Cluster knapp werden, Startup-Zeiten der Java Anwendungen nicht zur angestrebten Cloud-Architektur passen oder einfach die Kosten für RAM und CPU sehr stark zu Buche schlagen.

Ein guter, gangbarer Weg ist der Start mit einem Proof of Concept mit einer ersten eigenen Anwendung. Auf diesem Wege kann das betroffene Team noch eng unterstützt werden. Wenn die ersten Erfahrungen mit der neuen Technologie im eigenen Kontext gemacht wurden, lässt sich auch der großflächigere Einsatz besser planen.

Spring Boot 3 – Was das neueste Update mitbringt

Mit dem Release von Spring Boot 3.0 ist die native Kompilierung kein experimentelles Feature mehr, sondern ein fester Bestandteil des Frameworks. Praktischerweise bringt Spring Boot dafür nun die notwendigen Konfigurationen von Hause aus mit, womit prinzipiell jede neuere Spring-Boot-Anwendung nativ kompilierbar ist.

Eine der weggefallenen Konfigurationen ist die Erzeugung des Image mithilfe von Paketo. Sowohl die Erstellung des Image als auch die Auswahl der klassischen oder nativen Laufzeitumgebung sind im Build-Workflow des Frameworks fest verankert. Hier reicht es aus, das passenden Build-Profil („native“) hinzuzugeben. Spring übernimmt es dann, ein Image mit GraalVM-Laufzeitumgebung und AOT-Kompilierung zu bauen. Der Einstieg in die Cloud-kompatible Entwicklung ist dadurch deutlich leichter geworden.

Der Memory-Footprint von Spring Boot 3.0 ist im Vergleich zur Vorgängerversion auch ohne Native-Kompilierung gesunken. Durch das Upgrade konnten wir den durchschnittlichen Arbeitsspeicher-

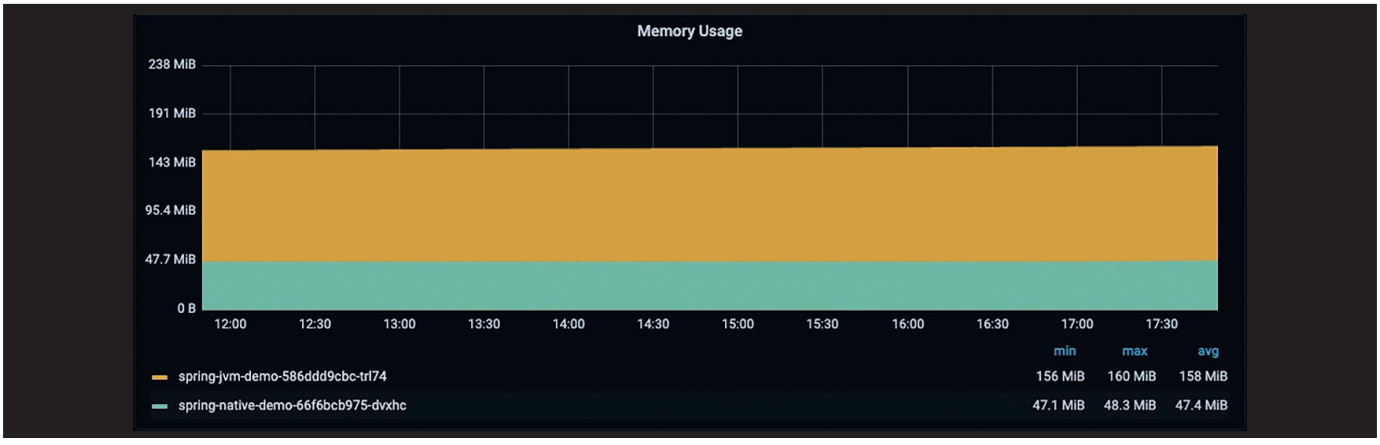


Abbildung 5: Speicherverbrauch der beiden Varianten mit Spring 3.0, visualisiert mit Grafana

verbrauch unserer Beispielanwendung mit JIT-Kompilierung von 215 MB um 25 % auf 158 MB senken. Leider ist diese Reduktion nicht bei der Verwendung des Native Image bemerkbar. Im Vergleich zur Vorgängerversion ist der Arbeitsspeicherverbrauch mit AOT-Kompilierung von ursprünglich 30,2 MB sogar um 56 % auf 47,4 MB gestiegen *(siehe Abbildung 5)*. Dennoch lässt sich der Verbrauch der Anwendung somit auf ein Drittel des ursprünglichen Verbrauchs deutlich reduzieren. Zudem ist Spring Boot 3.0 noch nicht allzu lange öffentlich verfügbar. Verbesserungen an dem AOT-Kompilierungsprozess sind denkbar, wodurch der Verbrauch in Nachfolgerversionen wieder gesenkt werden könnte.

Für den schnellen Start mit Spring Boot 3.0 haben wir in unserem GitHub-Beispielprojekt einen separaten Branch bereitgestellt *[3]*.

Bereit für den Einstieg?

Insbesondere, wenn man bereits Spring-Boot-Anwendungen entwickelt hat und Cloud- oder Container-Umgebungen einsetzt, ist in Blick auf Spring Native beziehungsweise die neuesten Versionen von Spring und Spring Boot absolut empfehlenswert. Für alle Fragen auf dem Weg stehen wir gern zur Verfügung.

Quellen

[1] *https://spring.io*
[2] *https://github.com/viadee/spring-native-demo*
[3] *https://github.com/viadee/spring-native-demo/tree/spring-3*

Dr. Benjamin Klatt
viadee AG
benjamin.klatt@viadee.de

Dr. Benjamin Klatt ist IT-Architekt und Agile Coach. Er begleitet Unternehmen bei der Digitalisierung von Produkten und Prozessen sowie der Erschließung neuer Technologien, Innovationen und Arbeitsweisen.

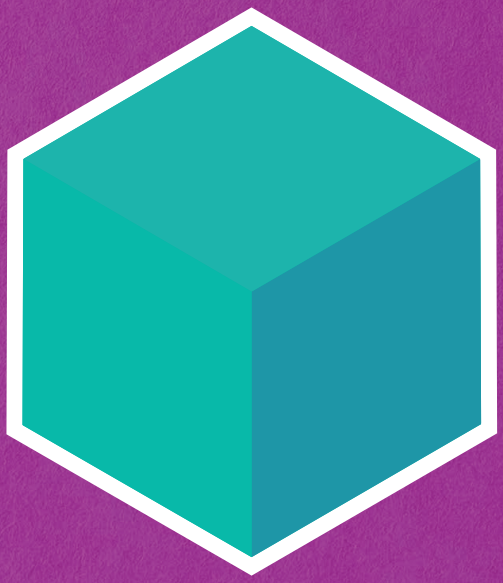
Matthias Kutz
viadee AG
matthias.kutz@viadee.de

Matthias Kutz ist schwerpunktmäßig im Bereich Software-Analyse und -Entwicklung tätig. Matthias besitzt mehrjährige Erfahrung im Java-Umfeld. Außerdem engagiert er sich in unserem F&E-Bereich für Cloud-Themen rund um Docker und Kubernetes.

Testgetriebene Entwicklung von Kubernetes-Operatoren mit Testcontainers

Alex Stockinger

Die Entwicklung von Kubernetes-Operatoren in Java ist zwar noch nicht die Regel, aber durch die Verwendung von Tools wie Kindcontainer und Testcontainers wird die testgetriebene Entwicklung von Operatoren und Controllern in Java immer einfacher und zugänglicher. Bisher ist hier – nicht zuletzt wegen der guten Unterstützung beim Schreiben von entsprechenden Tests – meist Go die Sprache der Wahl. Mit der Bibliothek Kindcontainer auf Basis des Integrationstest-Frameworks Testcontainers gibt es nun jedoch auch in der Java-Welt das nötige Tooling dafür.



Testcontainers

Obwohl die Mehrheit der Softwarekomponenten im Kubernetes-Ökosystem in Go implementiert werden, gibt es immer mehr Projekte, die Java als Implementierungssprache nutzen: Der Kafka-Operator Strimzi und das Java-Operator-SDK-Framework sind hier nur zwei der prominenteren Beispiele. Eine Herausforderung bei der Entwicklung von Java-basierten Projekten waren bisher die fehlenden Möglichkeiten für einfache automatisierte Integrationstests, die mit einem Kubernetes-API-Server interagieren. Dank der Open-Source-Bibliothek Kindcontainer auf Basis des verbreiteten Integrationstest-Frameworks Testcontainers kann diese Lücke geschlossen und eine einfachere Entwicklung von Java-basierten Kubernetes-Projekten ermöglicht werden.

Kubernetes in Docker

Testcontainers ermöglicht das Starten beliebiger Infrastrukturkomponenten in Docker-Containern aus in einer JVM laufenden Tests heraus. Das Framework kümmert sich dabei darum, den Lebenszyklus der Docker-Container an den der Testausführung zu binden: Selbst wenn etwa die JVM aus dem Debugging heraus unsanft terminiert wird, ist sichergestellt, dass die gestarteten Docker-Container ebenfalls beendet werden. Testcontainers bietet dabei neben einer generischen Klasse für beliebige Docker-Images auch spezialisierte Implementierungen in Form von Subklassen – etwa für Komponenten mit anspruchsvollen Konfigurationsmöglichkeiten. Solche spezialisierten Implementierungen können auch durch Third-Party-Bibliotheken bereitgestellt werden.

Das Open-Source-Projekt Kindcontainer *[1]* ist eine solche Third-Party-Bibliothek, die auf Basis von Testcontainers spezialisierte Implementierungen für verschiedene Kubernetes-Container bereitstellt:

- `ApiServerContainer`
- `K3sContainer`
- `KindContainer`

Während `ApiServerContainer` sich mit dem Kubernetes-API-Server darauf beschränkt, nur einen kleinen Teil der Kubernetes-Control-Plane bereitzustellen, starten K3sContainer und `KindContainer` vollständige Single-Node Kubernetes Cluster in Docker-Containern. Dadurch wird ein Trade-Off abhängig von den Ansprüchen der jeweiligen Tests möglich: Falls lediglich die Interaktion mit dem API-Server zum Testen nötig ist, dann reicht der deutlich schneller startende `ApiServerContainer` in der Regel aus. Steht jedoch das Testen von komplexen Interaktionen etwa mit anderen Komponenten der Kubernetes-Control-Plane oder gar anderen Operatoren im Lastenheft, so bieten die beiden „großen" Implementierungen dafür das nötige Handwerkszeug – allerdings zulasten der Startzeit: Je nach Hardwareausstattung kann das Starten hier durchaus im Bereich von einer Minute oder mehr liegen.

Ein einfacher Beispieltest

Um zu veranschaulichen, wie einfach das Testen gegen einen Kubernetes-Container sein kann, werfen wir einen Blick auf ein Beispiel auf Basis von JUnit 5 in *Listing 1*.

Dank der JUnit-5-Erweiterung `@Testcontainers` ist die Lebenszyklusverwaltung des `ApiServerContainer` einfach mit der Annotation `@Container` erledigt. Sobald der Container gestartet ist, kann über die `getKubeconfig()`-Methode ein YAML-Dokument mit den nötigen Details zum Verbindungsaufbau mit dem API-Server geholt werden. Dabei handelt es sich um die gängige Darstel-

```
@Testcontainers
public class SomeApiServerTest {
      @Container
      public ApiServerContainer<?> K8S = new ApiServerContainer<>();

      @Test
      public void verify_no_node_is_present() {
            Config kubeconfig = Config.fromKubeconfig(K8S.getKubeconfig());
            try (KubernetesClient client = new KubernetesClientBuilder().withConfig(kubeconfig).build()) {
                  // Verify that ApiServerContainer has no nodes
                  assertTrue(client.nodes().list().getItems().isEmpty());
            }
      }
}
```

Listing 1: Einfaches Beispiel mit JUnit 5

```
@Testcontainers
public class FluentApiTest {
      @Container
      public static final K3sContainer<?> K3S = new K3sContainer<>()
                  .withKubectl(kubectl -> {
                        kubectl.apply.fileFromClasspath("manifests/mycrd.yaml").run();
                  })
                  .withHelm3(helm -> {
                        helm.repo.add.run("repo", "https://repo.example.com");
                        helm.repo.update.run();
                        helm.install.run("release", "repo/chart");
                  });

      // Tests go here
}
```

Listing 2: Fluent-APIs zur Vorbereitung eines Containers

```
@Testcontainers
public class SpecificVersionTest {
     @Container
     KindContainer<?> container=new KindContainer<>(KindContainerVersion.VERSION_1_24_1);

     // Tests go here
}
```

Listing 3: Explizite Wahl der Kubernetes-Version

lung von Verbindungsinformationen in der Kubernetes-Welt. Der im Beispiel verwendete fabric8-Kubernetes-Client lässt sich einfach über die `Config.fromKubeconfig()` konfigurieren – jede andere Kubernetes-Client-Bibliothek wird ähnliche Schnittstellen anbieten. Kindcontainer macht in dieser Hinsicht keine Vorgaben.

Alle drei Container-Implementierungen setzen dabei auf einem gemeinsamen API auf. Sollte also etwa zu einem späteren Zeitpunkt klar werden, dass doch eine der schwergewichtigeren Implementierungen für einen Test nötig ist, so kann man einfach ohne weitere Code-Änderungen auf eine davon umschwenken – der bereits implementierte Test-Code kann unverändert bleiben.

Container für Tests vorbereiten

In vielen Situationen muss nach dem Start des Kubernetes-Containers noch einiges an Vorarbeit geleistet werden, bevor der eigentliche Testfall starten kann: Für einen Operator muss dem API-Server etwa zuerst eine Custom Resource Definition (CRD) bekannt gemacht oder ein anderer Controller muss erst per Helm-Chart installiert werden. Was im ersten Moment kompliziert klingt, macht Kindcontainer mit intuitiv zu verwendenden Fluent-APIs für die Kommandozeilenwerkzeuge `kubectl` und `helm` sehr einfach. *Listing 2* zeigt, wie zuerst mithilfe von `kubectl` eine CRD aus dem Classpath des Tests eingespielt und danach ein Helm-Chart installiert wird.

Kindcontainer stellt dabei sicher, dass alle Befehle ausgeführt werden, bevor der erste Test startet. Sollte es Abhängigkeiten zwischen den Befehlen geben, so lassen sich diese einfach auflösen: Kindcontainer führt sie garantiert in der Reihenfolge aus, in der sie angegeben werden.

Das Fluent-API wird dabei in Aufrufe des jeweiligen Kommandozeilenwerkzeugs übersetzt. Diese werden in separaten Containern ausgeführt, die automatisch mit den nötigen Verbindungsdetails gestartet und über das Docker-interne Netzwerk mit dem Kubernetes-Container verbunden sind. Dadurch können Abhängigkeiten vom Kubernetes-Image und Versionskonflikte im Hinblick auf das darin verfügbare Tooling vermieden werden.

Verschiedene Kubernetes-Versionen

Wenn nichts anderes vom Entwickler angegeben wird, startet Kindcontainer die jeweils neueste unterstützte Kubernetes-Version. Wer hier mehr Kontrolle über seine Testumgebung haben möchte, kann beim Erstellen des Containers, wie in *Listing 3* zu sehen, einfach eine der unterstützten Versionen explizit angeben.

Jede der drei Container-Implementierungen hat dazu ein eigenes `Enum`, über das eine der unterstützten Kubernetes-Versionen ausgewählt werden kann. Die Test-Suite des Kindcontainer-Projekts selbst stellt dabei mithilfe eines aufwendigen, Matrix-basierten Integrationstest-Setups sicher, dass für jede dieser Versionen auch das vollständige Feature-Set problemlos genutzt werden kann. Dieser aufwendige Testprozess ist notwendig, da sich das Kubernetes-Ökosystem schnell weiterbewegt und abhängig von der Kubernetes-Version unterschiedliche Initialisierungsschritte durchzuführen sind.

Die zum Zeitpunkt des Verfassens dieses Artikels aktuelle Version 1.4.0 von Kindcontainer unterstützt dabei bereits das aktuelle Kubernetes 1.26 für alle drei Container-Implementierungen. Generell legt das Projekt großen Wert darauf, alle aktuell gepflegten Kubernetes-Hauptversionen, die alle vier Monate erscheinen, zu unterstützen. Nicht mehr aktuelle Kubernetes-Versionen werden als `@Deprecated` markiert und auch irgendwann entfernt, wenn deren Support in Kindcontainer zu aufwendig wird. Dies sollte jedoch immer erst zu einem Zeitpunkt geschehen, zu dem der Einsatz der jeweiligen Kubernetes-Version ohnehin nicht mehr empfehlenswert ist.

Eigene Docker Registry

Häufig ist der Zugriff auf Docker-Images aus öffentlichen Quellen nicht ohne Weiteres möglich – etwa in Corporate-Umgebungen, die auf eine interne Docker-Registry mit eigenem manuellen oder automatisierten Auditing setzen. Kindcontainer erlaubt es dem Entwickler selbstverständlich, zu diesem Zweck auch eigene Koordinaten für die verwendeten Docker-Images anzugeben. Da Kindcontainer wegen der gegebenenfalls unterschiedlichen Initialisierungsschritte jedoch weiterhin wissen muss, um welche Kubernetes-Version es sich handelt, werden diese eigenen Koordinaten, wie in *Listing 4* zu sehen, an den jeweiligen `Enum`-Wert angehängt.

```
@Testcontainers
public class CustomKubernetesImageTest {
     @Container
     KindContainer<?> container=new KindContainer<>(KindContainerVersion.VERSION_1_24_1.withImage("my-registry/
kind:1.24.1"));

     // Tests go here
}
```

Listing 4: Verwenden eines Kubernetes-Image aus einer eigenen Registry

```
@Testcontainers
public class CustomFluentApiImageTest {
     @Container
     KindContainer<?> container=new KindContainer<>()
           .withKubectlImage(DockerImageName.parse("my-registry/kubectl:1.21.9-debian-10-r10"))
           .withHelm3Image(DockerImageName.parse("my-registry/helm:3.7.2"));

     // Tests go here
}
```

Listing 5: Verwenden eines Kubernetes-Image aus einer eigenen Registry

Neben den Kubernetes-Images selbst verwendet Kindcontainer noch einige andere Docker-Images. So werden etwa wie bereits erläutert die Kommandozeilenwerkzeuge kubectl und helm in eigenen Containern ausgeführt. Natürlich sind auch die dafür nötigen Docker-Images konfigurierbar. Für deren Ausführung sind jedoch glücklicherweise keine versionsabhängigen Codepfade nötig. Die Konfiguration gestaltet sich daher, wie in *Listing 5* zu sehen, einfacher als im Falle des Kubernetes-Image.

Auch die Koordinaten der Images aller anderen zur Unterstützung gestarteten Container können einfach manuell gewählt werden – dabei obliegt dem Entwickler aber natürlich immer die Verantwortung, dieselben oder zumindest kompatible Images einzusetzen. Zu diesem Zweck findet man eine vollständige Liste der verwendeten Docker-Images und der eingesetzten Versionen in der Dokumentation von Kindcontainer auf GitHub.

Admission-Controller-Webhooks

Für die bisher gezeigten Testszenarien ist die Kommunikationsrichtung klar: Ein in der JVM laufender Kubernetes-Client greift über Netzwerk auf den lokal oder remote laufenden Kubernetes-Container zu, um mit dem darin laufenden API-Server zu kommunizieren. Diesen Standardfall macht Docker auch denkbar einfach: Für den API-Server wird am Docker-Container ein Port geöffnet, über den er erreichbar ist. Kindcontainer nimmt die dafür nötigen Konfigurationsschritte automatisch vor und stellt für die jeweilige Netzwerkkonfiguration passende Verbindungsinformationen als Kubeconfig zur Verfügung.

Mit Admission-Controller-Webhooks tut sich jedoch ein technisch deutlich anspruchsvolleres Testszenario auf: Für diese muss der API-Server bei der Verarbeitung von Manifesten via HTTPS mit externen Webhooks kommunizieren können. Im unserem Fall laufen diese Webhooks üblicherweise in der JVM, in der auch die Testlogik ausgeführt wird – und diese ist vom Docker-Container aus nicht notwendigerweise einfach erreichbar.

Um das Testen dieser Webhooks unabhängig vom Netzwerk-Setup und trotzdem einfach zu gestalten, greift Kindcontainer hier zu einem Trick: Neben dem Kubernetes-Container selbst werden noch zwei weitere Container gestartet. Ein SSH-Server stellt die Möglichkeit bereit, von der Test-JVM aus einem Tunnel in den Kubernetes-Container hinein aufzubauen und ein Reverse-Port-Forwarding

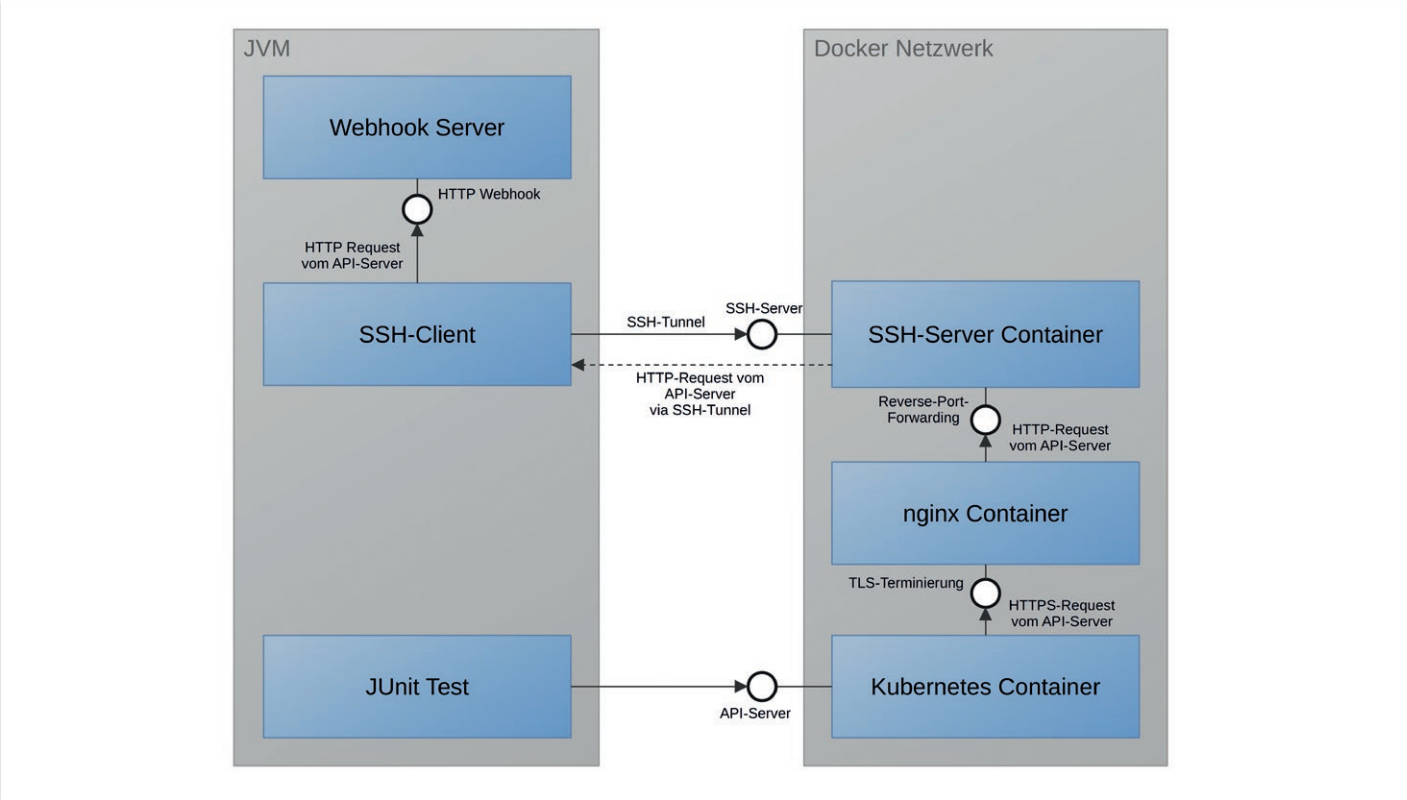


Abbildung 1: Netzwerkkonfiguration für das Testen von Webhooks

einzurichten, über das der API-Server zurück zur JVM kommunizieren kann. Da Kubernetes für die Kommunikation mit Webhooks auf über TLS abgesicherte Kommunikation besteht, wird zusätzlich ein nginx-Container gestartet, der die TLS-Terminierung für die Webhooks übernimmt. Die Verwaltung des dafür benötigten Zertifikat-Materials wird dabei von Kindcontainer abgewickelt. Das gesamte Setup an Prozessen, Containern und deren Netzwerkkommunikation wird in *Abbildung 1* veranschaulicht.

Glücklicherweise versteckt Kindcontainer diese Komplexität hinter einem einfach zu verwendenden API – *Listing 6* zeigt das beispielhaft.

Der Entwickler muss hierbei lediglich den Port des lokal laufenden Webhook nebst einigen für die Einrichtung in Kubernetes nötigen Informationen übergeben. Kindcontainer übernimmt dann automatisch die Konfiguration von SSH-Tunnel, TLS-Terminierung und Kubernetes.

Fazit

Ausgehend vom einfachen Beispiel eines minimalen JUnit-Tests haben wir in diesem Artikel gelernt, wie man eigene in Java implementierte Kubernetes-Controller und -Operatoren testen kann. Wir haben dabei gesehen, wie man mithilfe des Fluent-API bekannte Kommandozeilen-Werkzeuge aus dem Ökosystem verwendet und wie man auch in eingeschränkten Netzwerkumgebungen problemlos Integrationstests ausführen kann. Abschließend haben wir noch gesehen, wie selbst der technisch anspruchsvolle Use-Case des Testens von Admission-Controller-Webhooks mit Kindcontainer einfach und bequem umgesetzt werden kann. Hoffentlich finden sich dank dieser neuen Testmöglichkeiten zukünftig mehr und mehr Entwickler, die Java als Sprache der Wahl für ihre Kubernetes-nahen Projekte in Betracht ziehen!

Quellen

[1] https://www.github.com/dajudge/kindcontainer

```
@Testcontainers
public class WebhookTest {
     @Container
     ApiServerContainer<?> container=new ApiServerContainer().withAdmissionController(admission -> {
               admission.mutating()
                   .withNewWebhook("mutating.example.com")
                      .atPort(webhookPort) // Local port of webhook
                      .withNewRule()
                         .withApiGroups("")
                         .withApiVersions("v1")
                         .withOperations("CREATE", "UPDATE")
                         .withResources("configmaps")
                         .withScope("Namespaced")
                      .endRule()
                   .endWebhook()
                   .build();
           })

     // Tests go here
}
```

Listing 6: Testen von Webhooks

Alex Stockinger
mail@alexstockinger.de

Alex Stockinger ist der Autor von Kindcontainer. Er ist Software-Engineer mit mehr als zehn Jahren Erfahrung und einer Passion für herausfordernde technische Problemstellungen. Während er in früheren Jahren vorwiegend im Java-fokussierten Beratungsgeschäft unterwegs war, hat sich sein Schwerpunkt in letzter Zeit mehr und mehr in Richtung DevOps und Cluster-Orchestrierung bewegt. Seit letztem Jahr verbringt er bei AtomicJar seine Zeit in verschiedenen Inception-Tiefen zwischen VMs, Containern und Kubernetes-Clustern.

Was jeder Java-Entwickler über Strings wissen sollte

Bernd Müller, Ostfalia

Strings sind die am häufigsten verwendeten Java-Objekte. Es ist daher nicht weiter verwunderlich, dass JDK-Entwickler seit Java 1.0 an String-Optimierungen arbeiten. Diese Optimierungen finden auf allen Ebenen statt: JVM, Garbage Collector, Compiler, Byte-Code und natürlich auch in der String-Klasse selbst. Wir werfen einen kleinen Blick in die Implementierungsdetails von Strings in den genannten Bereichen und hoffen, dass Java-Entwickler nach der Lektüre einen Einblick in den sonst so unspektakulären Datentyp String und zusammenhängende Implementierungsfragen bekommen haben.


HTML
CSS
Java

Einordnung

Strings sind die am häufigsten verwendeten Java-Objekte. Beginnend bei Exception-Messages über Log-Nachrichten, Beschriftungen von UI-Elementen und Datenaustauschformaten wie XML und JSON, basieren alle genannten Punkte und noch viele mehr auf Strings. Durch die allgegenwärtige Verwendung von Strings ist deren speichereffiziente Repräsentation sowie laufzeiteffiziente Verarbeitung von zentraler Bedeutung für die Leistungsfähigkeit einer Sprache beziehungsweise Plattform. Dies gilt sowohl für Java als auch für alle anderen Programmiersprachen. Bereits in der initialen Version 1.0 von Java wurden daher entsprechende Optimierungen realisiert, die in den letzten 25 und mehr Jahren meist in der Form von JEPs *[1]* weiter vervollständigt wurden und immer noch werden. Wir stellen im Folgenden vor, wie Strings intern gespeichert wurden und werden, wie Duplikate gefunden und entfernt sowie Strings konkateniert werden.

Der String-Pool

Nach Abschnitt 3.10.5 String Literals der Java-Sprachdefinition gibt es String-Literale nur einmal in der JVM:

„Moreover, a string literal always refers to the same instance of class String. This is because string literals – or, more generally, strings that are the values of constant expressions – are ‚interned' so as to share unique instances, as if by execution of the method String.intern."

```
public class PoolTest {

  String str1 = "Hello, World!";
  String str2 = "Hello, World!";
  String str3 = new String("Hello, World!");

  @Test
  public void same() {
     Assertions.assertSame(str1, str2);
  }

  @Test
  public void same2() {
     Assertions.assertSame("Hel" + "lo", "Hel" + "lo");
  }

  @Test
  public void same3() {
     Assertions.assertSame("Hello, World!", "Hello, World!");
  }

  @Test
  public void notSame() {
     Assertions.assertNotSame(str1 + str1, str2 + str2);
  }

  @Test
  public void sameAfterInterning() {
     Assertions.assertSame((str1 + str1).intern(), (str2 + str2).intern());
  }

  @Test
  public void equals() {
     Assertions.assertEquals(str1 + str1, str2 + str2);
  }

  @Test
  public void notSameNewConstructed() {
     Assertions.assertNotSame(str1, str3);
  }

}
```

Listing 1

Aber nicht nur String-Literale wie „`Hello`", sondern auch konstante Ausdrücke von String-Literalen wie etwa „`Hello" + " World! "`, die schon vom Compiler zusammengefasst werden, existieren in der JVM nur einmal. Unabhängig von einer eventuell tausendfachen Verwendung in Hunderten verschiedener Klassen sind die String-Literale „`Hello`" und „`Hello World! `" also Singletons in der JVM. Diese Eindeutigkeit wird durch eine weitere Forderung der Sprachdefinition im Abschnitt *12.5 Creation of New Class Instances* gewährleistet:

„Loading of a class or interface that contains a string literal or a text block may create a new String object to denote the string represented by the string literal or text block. (This object creation will not occur if an instance of String denoting the same sequence of Unicode code points as the string represented by the string literal or text block has previously been interned.)"

Was es mit einem Unicode-Code-Point auf sich hat, sehen wir später. Was ein „internter" String ist (wir bleiben bei der englischen Version, da das deutsche Internieren negativ konnotiert ist), entnehmen wir dem JavaDoc der Methode `String.intern()`:

„Returns a canonical representation for the string object. A pool of strings, initially empty, is maintained privately by the class String. When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned. It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true. All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the The Java Language Specification."

Der genannte String-Pool ist eine in C implementierte Hash-Table im nativen Speicher der JVM, die Strings selbst liegen im Heap. Die C-Implementierung erlaubt das von Java-Hash-Tables gewohnte dynamische Wachstum nicht, die Anzahl der Buckets ist fest. Seit Java 11 ist die Größe 65536, obwohl die gängige Literatur zu einer Primzahl rät. Falls dieser Wert nicht optimal ist und beispielsweise viele Kollisionen auftreten, kann die Größe beim Start der JVM mit der Option `-XX:StringTableSize=N` eingestellt werden. Wie die Auslastung der Hash-Table tatsächlich ist, kann durch die JVM-Option `-XX:PrintStringTableStatistics` erfragt werden, die zur Ausgabe der Statistik des String-Pools führt, wenn die JVM beendet wird.

Das Verfahren, wie Java die Singleton-Eigenschaft von String-Literalen garantiert, kann nun wie folgt zusammengefasst werden. Der Compiler erzeugt aus Ausdrücken,

die nur aus String-Literalen bestehen, den konkatenierten String bereits für den Byte-Code. Die im Byte-Code vorhandenen String-Literale werden beim Laden der Klasse in die JVM dahingehend geprüft, ob sie bereits im String-Pool vorhanden sind. Falls ja, wird die Referenz auf den String zurückgegeben, der String aber nicht nochmals in den Pool aufgenommen. Falls das Literal noch nicht im Pool vorhanden ist, wird es hinzugefügt und die neue Referenz darauf zurückgegeben.

Um dies ein wenig zu veranschaulichen, zeigt das *Listing 1* eine Reihe von Tests, die keiner weiteren Erläuterung bedürfen.

Da „internte" Strings mit dem `==`-Operator verglichen werden können und dieser sicher sehr viel schneller auszuführen ist als die `equals()`-Methode der Klasse `String`, könnte man auf die Idee kommen, dass das Internen aller dynamisch erzeugten Strings zu kürzeren Laufzeiten führt. Scott Oaks *[2]* meint dazu allerdings, dass dies in der Regel nicht der Fall ist, da zwar der Referenz-Vergleich in der Tat schneller ist, dies aber durch die zusätzliche Laufzeit der Aufrufe der `intern()`-Methode ausgeglichen wird.

Bei den Recherchen zu diesem Artikel wurde der Autor in seiner Meinung, dass die Entwickler des OpenJDK ganz unglaublich helle Köpfe sein müssen, sehr bestätigt, da sowohl auf konzeptioneller als auch auf implementierungstechnischer Ebene Unglaubliches, zumindest nach der Werteskala des Autors, geleistet wird. Trotzdem sind auch diese Entwickler nur Menschen und damit fehlbar. Eine recht bekannte, sagen wir semioptimale Entwurfsentscheidung war die ursprüngliche Implementierung der Klasse String. Diese enthielt unter anderem die Instanzvariablen `char[] value`, `int offset` und `int count`, die die einzelnen Zeichen des Strings als Array, den Index des ersten zu verwendenden Zeichens sowie die Gesamtzahl der Zeichen des Strings repräsentierten. Die `String`- und `StringBuffer`-Methode `substring()` war mit dem Ziel der Laufzeit- und Speicherplatzoptimierung derart implementiert, dass der neue Teil-String kein eigenes Zeichen-Array hatte, sondern mithilfe von `value`, `offset` und `count` auf das Ursprungs-Array verwies sowie den Start und die Länge angab. Leider hatte man übersehen, dass diese Lösung verhindert, dass der ursprüngliche String garbage collected werden konnte, wenn er nicht mehr referenziert, wohl aber der Teil-String referenziert wurde. Dieses Memory-Leak führt in unglücklichen Umständen zu Speicherproblemen der JVM. Der Fehler, der die Kennzeichnung „*Memory leak due to String.substring() implementation*" führt, kann in der Fehlerdatenbank des JDK *[3]* nachgelesen werden. Der Fehler wurde mit Java 7 behoben, indem der Teil-String komplett neu angelegt wurde, also inklusive des Zeichen-Arrays. Die Instanzvariablen `offset` und `count` wurden aus der String-Klasse entfernt.

Compact Strings

Wir haben gerade gesehen, dass die Klasse `String` ein `char`-Array enthält, das die einzelnen Zeichen des Strings repräsentiert. Der JEP 254: *Compact Strings [4]* hatte das Ziel, dies speichereffizienter zu realisieren. Als Java 1995 vorgestellt wurde, war es die erste Sprache, die UTF-16 als Zeichencodierung einsetzte, um dem Anspruch einer weltweit Verwendung findenden Sprache gerecht zu werden. Das JavaDoc der Klasse `Character` enthält einen Abschnitt, in dem die in der jeweiligen Java-Version verwendete Zeichencodierung nachzulesen ist. Mit Java 5 setzte das JDK auf Unicode 4.0, das auch Codierungen enthält, die 32 Bits benötigen. Der 16-Bit-Zeichendatentyp Javas reichte also nicht mehr aus. Den Typ `char` auf 32 Bits zu erweitern, schied aus Kompatibilitätsgründen aus. Die in Unicode definierten Code Points wurden daher auch in Java eingeführt, nachzulesen ebenfalls im JavaDoc der Klasse `Character`. Dies führte beispielsweise dazu, dass das JavaDoc der Methode `String.length()` geändert werden musste, und zwar von Java 5.0 mit

„Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string."

zu Java 6 mit

„Returns the length of this string. The length is equal to the number of Unicode code units in the string."

Java verwendet nun also zwei Bytes pro Zeichen eines Strings, manchmal sogar vier. Oracle untersuchte zu der Zeit Core-Dumps von WebLogic-Instanzen und analysierte die verwendeten Zeichencodierungen. Wenig überraschend kam dabei heraus, dass recht oft Latin-1 als Zeichencodierung ausgereicht hätte, also nur 1 Byte pro Zeichen. Der JEP 254 hat das Ziel, für jeden String eine bedarfsgerechte Speicherform auszuwählen. Enthält der String nur Latin-1-Zeichen, wird pro Zeichen ein Byte verwendet. Wenn nicht zwei oder vier Bytes. Dazu wurde die Instanzvariable `value` zu einem `byte`-Array und es wurde eine weitere Instanzvariable `coder` eingeführt, die die Werte `LATIN1` oder `UTF16` annehmen kann. Diese Umstellung wurde mit Java 9 vollzogen.

Zur Verdeutlichung dieses Sachverhalts dienen ein paar weitere Tests, die im *Listing 2* dargestellt sind. Die Methode `Util.getValueBytes()` ist eine Eigenentwicklung, die über Reflection das Byte-Array zurückgibt.

Auch an dieser Stelle sei noch einmal die Bemerkung erlaubt, dass die Idee und die Umsetzung der geänderten internen String-Implementierung mit Java 9 eine Meisterleistung der Software-Entwicklung waren. Keine API-Änderungen; alle Programme verhielten sich wie vorher, hatten aber ein besseres Laufzeitverhalten und einen geringeren Speicherbedarf. Wir alle haben es nicht bemerkt!

Wie viele Dinge in der JVM ist auch diese speichereffiziente Codierung von Strings konfigurierbar. Falls eine Anwendung zu großen Teilen UTF16-Strings verwendet, ist der Versuch, diese zunächst mit einem Byte zu speichern, um dann doch zu zwei Bytes greifen zu müssen, mit einem unnötigen Mehraufwand verbunden. In diesem Fall kann die JVM-Option `-XX:-CompactStrings` verwendet werden, um generell UTF16-Strings zu verwenden.

Das Kompaktieren von Strings hat eine Vorgeschichte, die einen guten Einblick in die Entwicklungsphilosophie des OpenJDK gibt. Mit Java 6 wurden Strings sowohl als `char[]` als auch als `byte[]` implementiert, also absichtlich redundanter Code erzeugt. Bei der Verwendung der JVM-Option `-XX:+UseCompressedStrings` wurde versucht, Latin-1-Strings in der Byte-Variante zu codieren. Einige String-Methoden waren aber nur für die Char-Version implementiert, sodass die Strings von der einen in die andere Codierung überführt werden mussten. Dieser Aufwand sowie die parallele

```
public class EncodingTest {

  /**
   * Assert that "a" has a one byte representation.
   *
   * See Unicode <a href="http://www.unicode-symbol.com/u/0061.html">latin small letter a</a>
   */
  @Test
  public void oneByteRepresentation() {
     final String str = "\u0061";

     Assertions.assertTrue(str.equals("a"));
     Assertions.assertEquals(1, str.length());
     Assertions.assertEquals(1, str.codePointCount(0, str.length()));
     Assertions.assertEquals(1, Util.getValueBytes(str).length);
  }


  /**
   * Assert that "ś" has a two byte representation.
   *
   * See Unicode <a href="http://www.unicode-symbol.com/u/015B.html">latin small letter s with acute</a>
   */
  @Test
  public void twoBytesRepresentation() {
     final String str = "\u015B";

     Assertions.assertTrue(str.equals("ś"));
     Assertions.assertEquals(1, str.length());
     Assertions.assertEquals(1, str.codePointCount(0, str.length()));
     Assertions.assertEquals(2, Util.getValueBytes(str).length);
  }


  /**
   * Assert that "𩸽" has a four byte representation.
   *
   * See Unicode <a href="http://www.unicode-symbol.com/u/29E3D.html">cjk unified ideograph-29E3D</a>
   */
  @Test
  public void fourBytesRepresentation() {
     final String str = "𩸽"; // hex: 29E3D, int: 171581

     Assertions.assertTrue(str.equals(new String(new int[] { 171581 }, 0, 1)));
     Assertions.assertEquals(2, str.length());
     Assertions.assertEquals(1, str.codePointCount(0, str.length()));
     Assertions.assertEquals(4, Util.getValueBytes(str).length);
  }

}
```

Listing 2

Wartung der beiden String-Implementierungen wertete Aleksey Shipilv, ein bekannter JDK-Engineer, mit

„UseCompressedStrings was really the experimental feature, that was ultimately limited by design, error-prone, and hard to maintain."

Das Feature Compressed String wurde mit Java 7 wieder aus dem OpenJDK entfernt. Bitte Compact Strings, das seit Java 9 verwendete Speicherschema, und Compressed Strings, ein nur wenige Monate dauernder, nicht geglückter Versuch eines Speicherschemas, nicht durcheinanderbringen.

String Deduplication

Der JEP 192: String Deduplication in G1 *[5]* hatte das Ziel, die für String-Literale geltende Singularität auch für andere Strings gewährleisten zu können, zumindest nach einem Garbage-Collector-Lauf. Dies wird aber nicht wie bei String-Literalen im String-Pool auf der Ebene der String-Referenz selbst, sondern auf der Ebene der darunter liegenden Array-Referenz realisiert. Die *Abbildung 1a* zeigt zwei Strings, deren jeweilige Zeichen-Arrays identisch sind. Der Garbage Collector erkennt dies, entfernt eines der beiden Arrays und biegt die entsprechende Array-Referenz auf das verbleibende Zeichen-Array um, wie in *Abbildung 1b* dargestellt.

Der Implementierung des JEP 192 ist seit Java 8u20 verfügbar, muss aber mit der Option `-XX:+UseStringDeduplication` explizit angefordert werden. Außerdem ist anzumerken, dass das De-duplizieren nur im Garbage Collector G1 realisiert ist, der allerdings seit längerer Zeit der Default-Collector ist. Der Artikel *„G1: from garbage collector to waste management consultant"* *[6]* nennt für eine durchgeführte Messung eine Verringerung der Heap-Nutzung um 10 %. Da für alle derartigen Alternativen im OpenJDK auch Performanz-Tests durchgeführt werden, die Option aber nicht zu einem Default geworden ist, muss man davon ausgehen, dass der Einsatz nicht generell zu empfehlen ist, sondern bei Bedarf ausprobiert werden sollte. Es gilt wie immer bei Performanzoptimierungen: testen, testen, testen.

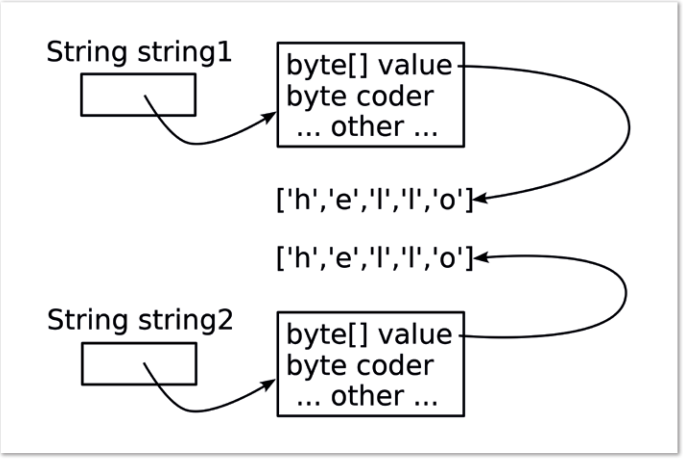


Abbildung 1a

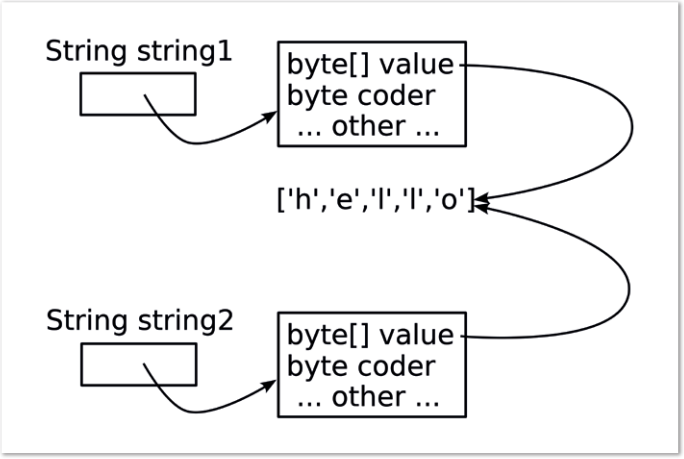


Abbildung 1b

Indify String Concatenation

Der letzte String-relevante JEP, den wir uns anschauen wollen, ist der JEP 280: *Indify String Concatenation [7]*. Die hinter diesem JEP liegende Optimierungsidee hört sich zunächst recht merkwürdig an. Spätere Java-Versionen, also Versionen, die im Augenblick noch nicht existieren, sollen das Konkatenieren von Strings optimiert ausführen können. Hintergrund ist der Umstand, dass die Konkatenierung bereits häufiger optimiert wurde, immer aber auf Compiler-Ebene, sowohl in `javac` als auch im JIT-Compiler, zum Beispiel durch Folgen von `StringBuffer.append()`-Aufrufen. Die Änderungen in den Compilern sind jedoch sehr aufwendig durchzuführen und zudem fehleranfällig. Der JEP beschreibt sie als „*those optimizations, while fruitful, are fragile and difficult to extend and maintain*".

Was wäre, wenn die String-Konkatenierung als möglichst allgemeiner Code formuliert und durch die gerade ausführende JVM, nicht den eventuell vor Jahren verwendeten Compiler, möglichst optimal ausgeführt wird? Genau dies macht der JEP 280. Die allgemeinste Code-Form ist die Verwendung von `invokedynamic`, mit Java 7 durch den JSR 292 eingeführt. Ziel dieses JSR war es, dynamisch getypte Sprachen wie etwa JRuby besser auf der JVM zu unterstützen. Heraus kam allerdings ein allgemeiner Ausführungsmechanismus für Methodenaufrufe. Ebenfalls interessant ist die mit JEP 280 eingeführte Klasse `StringConcatFactory` im Package `java.lang.invoke`. Sie unterstützt ausschließlich diese String-Konkatenation mit `invokedynamic` auf Byte-Code-Ebene, ist also insbesondere nicht dazu gedacht, von uns als Anwendungsentwickler verwendet zu werden. Von außen betrachtet lässt sich dazu nicht mehr sagen. Der Compiler erzeugt für die String-Konkatenation Code, der eventuell in späteren JVMs effizienter als heute ausgeführt werden kann, da er mehr Optimierungspotenzial besitzt.

Zusammenfassung

Strings sind die am häufigsten verwendeten Java-Objekte. Viele Java-Versionen hatten unter anderem das Ziel, die Verwendung von Strings hinsichtlich der Laufzeit und des Speicherplatzverhaltens zu optimieren. Wir haben den String-Pool vorgestellt, der mithilft, die Singleton-Eigenschaft von Strings zu realisieren. Die JEPs Compact Strings, String Deduplication und Indify String Concatenation wurden ebenfalls erläutern. Auch Versehen, Fehler beziehungsweise Irrwege bei derartigen Optimierungen wurden angesprochen. Der Leser ist nun hoffentlich davon überzeugt, dass die vielen Menschen, die hinter dem OpenJDK stehen, uns immer weiterhelfen, effizientere Java-Software zu schreiben. Wir selbst müssen dazu nichts beitragen, außer weiterhin Java zu nutzen.

Referenzen

[1] JEP 1: JDK Enhancement-Proposal & Roadmap Process, *https://openjdk.java.net/jeps/1*.

[2] Scott Oaks. Java Performance. O'Reilly 2020, 2nd Edition.

[3] JDK-4637640. *https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4637640*

[4] JEP 254: Compact Strings, *https://openjdk.java.net/jeps/254*.

[5] JEP 192: String Deduplication in G1, *https://openjdk.java.net/jeps/192*.

[6] G1: from garbage collector to waste management consultant, *https://blogs.oracle.com/java/post/g1-from-garbage-collector-to-waste-management-consultant*.

[7] JEP 280: Indify String Concatenation, *https://openjdk.org/jeps/280*.

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

APEX connect

by DOAG

3. - 4. Mai 2023

IN BERLIN

PROGRAMM ONLINE

apex.doag.org

DOAG

Die Oracle-Anwenderkonferenz

Eventpartner: AOUG AUSTRIAN ORACLE USER GROUP SOUG swiss oracle user group

anwenderkonferenz.doag.org

Mitglieder des iJUG

01 BED-Con e.V.
02 Clojure User Group Düsseldorf
03 DOAG e.V.
04 EuregJUG Maas-Rhine
05 JUG Augsburg
06 JUG Berlin-Brandenburg
07 JUG Bremen
08 JUG Bielefeld
09 JUG Bonn
10 JUG Darmstadt
11 JUG Deutschland e.V.
12 JUG Dortmund
13 JUG Düsseldorf rheinjug
14 JUG Erlangen-Nürnberg
15 JUG Freiburg
16 JUG Goldstadt
17 JUG Görlitz
18 JUG Hannover
19 JUG Hessen
20 JUG HH
21 JUG Ingolstadt e.V.
22 JUG Kaiserslautern
23 JUG Karlsruhe
24 JUG Köln
25 Kotlin User Group Düsseldorf
26 JUG Mainz
27 JUG Mannheim
28 JUG München
29 JUG Münster
30 JUG Oberland
31 JUG Ostfalen
32 JUG Paderborn
33 JUG Saxony
34 JUG Stuttgart e.V.
35 JUG Switzerland
36 JSUG
37 Lightweight JUG München
38 SUG Deutschland e.V.
39 JUG Thüringen
40 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 © pch.vector + starline
https://freepik.com
S. 10 + 11: Bild © pch.vector
https://freepik.com
S. 12: Bild © inkoly
https://123rf.com
S. 14 + 15: Bild © pch.vector
https://freepik.com
S. 22 + 23: Bild © pch.vector
https://freepik.com
S. 24: Bild © Macrovector
https://stock.adobe.com
S. 25: Bild © Nuthawut
https://stock.adobe.com
S. 28: Bild © rawpixel.com
https://freepik.com
S. 34 + 35: Bild © Freepik
https://freepik.com
S. 40 + 41: Bild © logturnal
https://freepik.com
S. 46 + 47: Bild © denamorado
https://freepik.com
S. 52 + 53: Bild © storyset
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

MITMACHEN UND BEITRAG EINREICHEN!

Du kennst dich in einem bestimmten Gebiet aus dem Java-Themenbereich bestens aus und möchtest als Autorin oder Autor dein Wissen mit der Community teilen?

Nimm Kontakt zu uns auf und sende deinen Artikelvorschlag zur Abstimmung an redaktion@ijug.eu.

Wir freuen uns, von dir zu hören!

iJUG Verbund

JavaLand

on demand

JavaLand 2023 verpasst?

Jetzt On-demand-Ticket buchen und Vortragsaufzeichnungen anschauen!

Alle Angebote im On-demand-Ticket-Shop

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

Veranstalter: JavaLand