Java aktuell

iJUG Verbund
www.ijug.eu

Cloud
Migration in die Cloud,
Helm Charts

Cloud Native
Cloud Native und AI
in Unternehmen

Java
JavaFinder, Eclipse Starter,
Distributed Message Schema

Ausgabe 06/2023

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 06

INFODAYS

Java 21 LTS

powered by JavaSPEKTRUM

15. NOVEMBER 2023 | ONLINE-KONFERENZ

DIE THEMEN IM ÜBERBLICK

Unsere Expert:innen erklären Ihnen in ihren praxisnahen Vorträgen die wichtigsten neuen Erkenntnisse zum neuen Java 21 LTS Release:

- Was sind die **wichtigsten Features seit Java 17?**
- Wo kann das **neue Pattern Matching** helfen?
- Wie geht nun eine **bessere nebenläufige Programmierung** mit Java vonstatten?
- Was bietet **Project Amber** noch so alles?
- **Project Panama:** Wie sich künftig nativer Code besser in Java einbetten lässt?

JETZT ANMELDEN!

INFODAYS.DE/JAVA-LTS

SPRECHER

Wolfgang Weigend | Oracle
Christian Schuster | mgm technology
Falk Sippach | embarc
Merlin Bögershausen | adesso
Dennis Makogon | Oracle

Liebe Leserinnen und Leser,

wir heben ab und begeben uns wieder einmal hoch hinaus in die Wolken – mit dem Dauerbrenner-Thema Cloud. Zu Beginn dieser Ausgabe bereiten uns das Java-Tagebuch und die Eclipse Corner mit Neuigkeiten aus der Community bestens auf den Abflug vor. Auch unsere unbekannten Kostbarkeiten sind wieder mit an Bord und widmen sich der Klasse ClassValue näher.

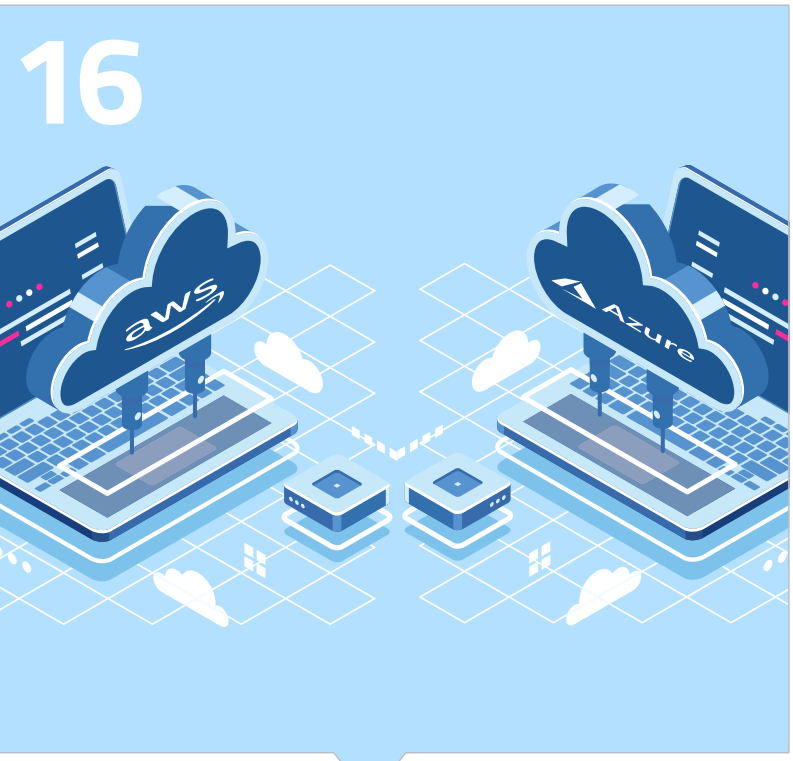
Unser Reise über die Wolken startet mit Frank Pientkas Artikel ab Seite 14, in dem er sich der Frage widmet, wie man eine Java-Anwendung in die Cloud migriert. Dabei betrachtet er Unterschiede und Vorteile von Azure und AWS Cloud. Im Anschluss präsentiert Jan Weyrich Helm Charts für Spring-Boot-Anwendungen. Er stellt Helm kurz vor und zeigt dann ausführlich, wie es genutzt werden kann. Weshalb auch das Team ein wichtiger Faktor zur optimalen Nutzung von Cloud Native und künstlicher Intelligenz innerhalb eines Unternehmens ist, erklärt und Björn Schotte ab Seite 32. Er gibt Tipps und Tricks, welche Maßnahmen dort getroffen werden können, um das Potenzial dieser modernen Technologien voll ausschöpfen zu können.

Weil nützliche Tools auf Reisen immer ein sinnvolles Mitbringsel sind, stellt uns Gerrit Grunwald seinen "JavaFinder" vor, der installierte Java-Distributionen auf dem Rechner ausfindig macht. So spart man sich eine aufwendige manuelle Suche und jede Menge Zeit. Ab Seite 42 zeigt uns Alexander Rühl den Eclipse Starter für Jakarta EE, an dem er selbst aktiv mitarbeitet. Wie man dem DRY-Prinzip ("Don't Repeat Yourself") innerhalb einer Microservices-Architektur gerecht werden kann, erklärt uns Ben Bajorat. In seinem Beispiel verwendet er dazu das Protobuf-Schema. Wir beenden unsere Reise und landen wieder auf festem Boden mit dem Artikel von Dominik Martens und Max Werner zum Abschluss dieser Ausgabe. Die beiden lassen ihr Studium zum Softwareentwickler Revue passieren und bewerten, inwieweit dieses sie ausreichend auf ihren derzeitigen Berufsalltag vorbereitet hat.

Wir wünschen euch viel Spaß beim Lesen!

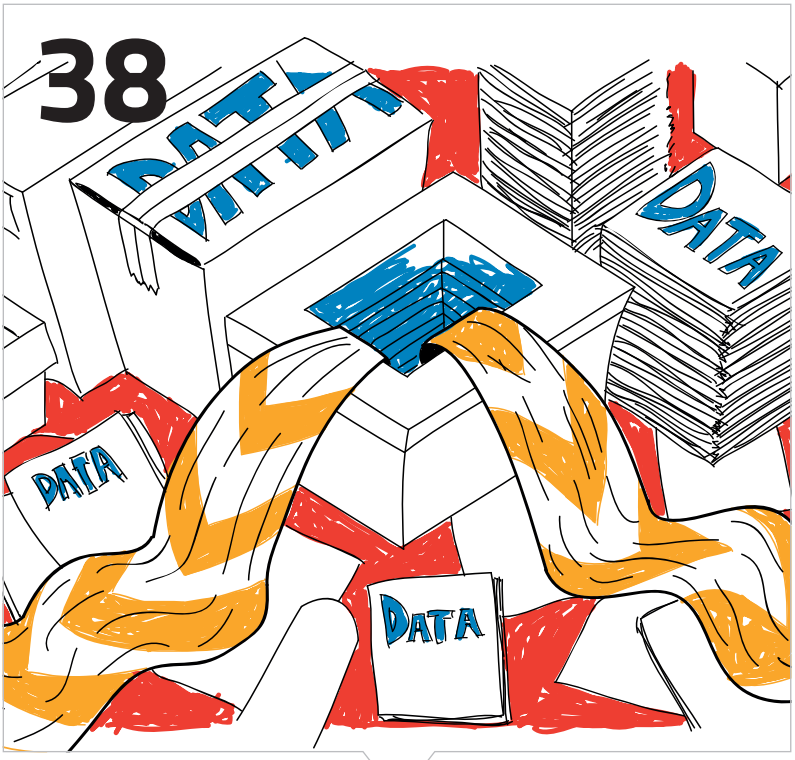
Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

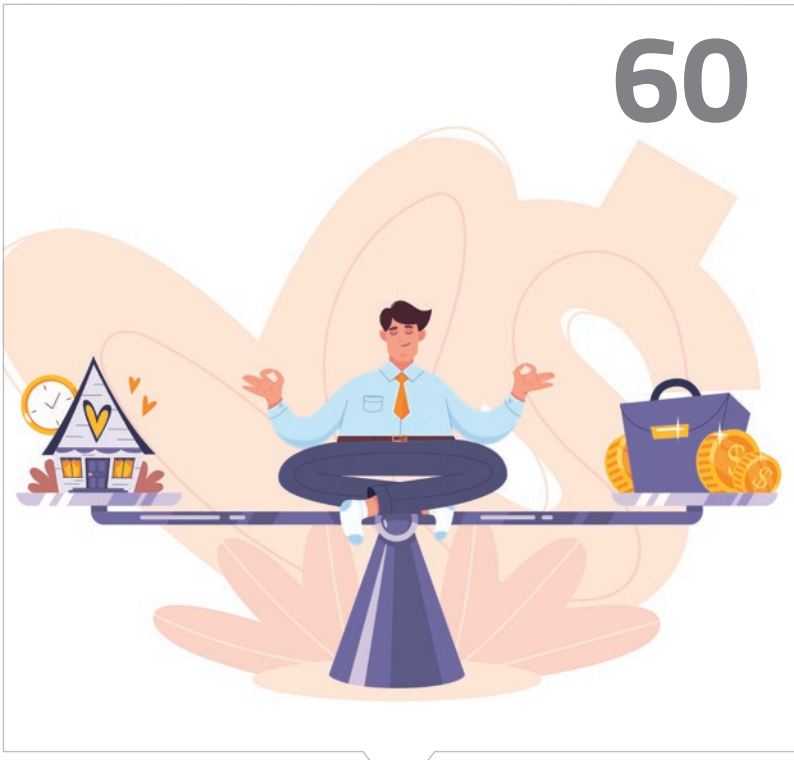


Migration einer Java-Anwendung in die Cloud

Das volle Potenzial von Cloud Native und AI ausschöpfen



Installierte Java-Distributionen mit dem JavaFinder ausfindig machen



Softwareentwicklung im Studium und im Arbeitsleben

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

9. Juni 2023

Java-Minecraft-Mods sind infiziert

Trotz aller Sorgfalt und automatisierter Prüfung von Open-Source-Projekten schlüpft immer wieder Schadcode durch, der – als normale Contribution getarnt – in krimineller Absicht den Projekten hinzugefügt wurde. Das Problem an sich ist nicht Java-spezifisch, aber im vorliegenden, alarmierenden Fall geht es konkret um Java-Code: Versteckt wurde der Schadcode diesmal in *Mods* (Modifikationen) für Minecraft-Server. (Aus)genutzt wurden dafür die Portale *Bukkit* und *CurseForge*, die solche Mods für die Java-Edition anbieten – das heißt Jar-Files, die in einen Minecraft-Server eingebunden werden, um neue Funktionalität hinzuzufügen. Der nach der identifizierten Quelle *fractureiser* genannte Schadcode zielt auf Windows-, aber wohl auch auf Linuxsysteme und soll insbesondere Microsoft-Zugangsdaten und im Browser gespeicherte Passwörter „abgreifen". Es dürfte eine recht große Gruppe von Menschen betroffen sein, die das beliebte Spiel auf einem eigenen Rechner betreiben. Ganz schnell hat die betroffene Community nach der Entdeckung ein Mitigation-Team gebildet, das den Code unter die Lupe genommen hat, aber auch wie er in die Repositories hineingelangt ist. Details, inklusive Listen der betroffenen Mods und wie man feststellt, ob man betroffen ist, finden sich hier *[1]*. Eine deutsche Zusammenfassung bietet heise an *[2]*.

21. Juni 2023

Spring Boot 3.1 und *docker-compose*

Für diejenigen, die noch kein Kubernetes auf dem Laptop betreiben, sondern mit Hilfe von *docker-compose* Microservices lokal testen: Spring Boot 3.1 bietet dafür jetzt verbesserte Unterstützung. Es wertet eine bereitgestellte *compose.yaml*-Datei aus, startet bei Bedarf die nötigen Services und extrahiert Informationen wie Ports oder Umgebungsvariablen daraus, sodass zum Beispiel die nötigen Zugriffsinformationen für einen Datenbank-Services nicht an mehreren Stellen konfiguriert werden müssen *[3]*.

22. Juni 2023

MicroProfile 7 und die Beziehung zu Jakarta EE

Wenn man zu später Stunde noch am Tagebuch schreibt… In der letzten Ausgabe hatte ich geschrieben, dass die Diskussion um das Verhältnis von MicroProfile mit, sorry, zu Jakarta mit einer Abstimmung beendet wurde. Das war aber nur eine Vor-Abstimmung, wie der Name „Straw Poll" mir eventuell hätte verraten können – zu Deutsch kein „Strohhalm-Ziehen", sondern eine Probeabstimmung. In dieser hatte eine knappe relative Mehrheit Option 1 gewählt (zur Erinnerung: Damit legt jeder MicroProfile-Release eine minimale Jakarta EE Version fest – auf Plattform-Ebene). In der jetzt erfolgten eigentlichen Abstimmung ging es nur noch um Ja oder Nein zu Option 1. Sie wurde mit deutlicher Mehrheit von neun Ja-Stimmen angenommen; dagegen waren nur Microsoft und der iJUG (der die vollständige Umstellung auf *Semantic Versioning* befürwortet hatte).

3. Juli 2023

GraalVM wird (größtenteils) kostenlos

Oracle hat die GraalVM inklusive der vormaligen Enterprise-Features als Oracle-GraalVM zur kostenlosen Nutzung (auch in Produktion) freigegeben – unter der neuen GraalVM *Free Terms and Conditions (GFTC)* Lizenz. Bislang war für den Einsatz der Enterprise-Version außerhalb der Oracle Cloud Infrastructure eine Java SE Subscription nötig. Fraglich ist zurzeit aber noch, ob Firmen mittels der GraalVM als Native Image erstellte Software auch verkaufen oder nur selbst in Produktion betreiben beziehungsweise als Open Source freigeben dürfen. Infos dazu gibt es in Falk Sippachs Beitrag bei heise online *[4]*.

Außerdem hat Oracle jetzt das Versionsschema angepasst. Die neuen Graal-Releases sind alle nach der jeweils zugrundeliegenden Java-Version nummeriert: GraalVM für JDK 17 und GraalVM für JDK 20. Das sieht jetzt mit den zwei parallelen Releases etwas verwirrend aus, aber in Zukunft soll es dann immer nur einen Release für die jeweils aktuelle JDK-Version (21, 22, …) geben.

Ein paar mehr Details zum Doppel-Release, was zum Beispiel die neuen *Profile-Guided Optimizations* nochmal an zusätzlicher Performance im Vergleich zu „normalen" Native Images herauskitzeln, sind in einem Blog-Eintrag von Alina Yurenko (Developer Advocate für die GraalVM bei Oracle) zu finden *[5]*.

6. Juli 2023

MicroProfile 6.1

Wenn wir bereits über MicroProfile 7 reden, sollten wir auch einen Blick auf 6.1 werfen. Der nächste Minor-Release der Plattform ist – diesmal ohne Gegenstimmen – angenommen worden. Es wird

Updates zu den Einzelspezifikationen *Config*, *Metrics* und *Telemetry* enthalten. Zieltermin ist Anfang Oktober.

Parallel – nicht für den nächsten Release – laufen im Übrigen die Diskussionen um *MicroProfile JWT* und *Jakarta Security* weiter. Die aktuell anvisierte, aber noch nicht verabschiedete Lösung sieht ein neues Projekt *jwt-bridge* vor (zurzeit noch in der *microprofile-sandbox* auf GitHub zu finden), dass die gemeinsame Basis für Jakarta und MicroProfile enthält, um zyklische Abhängigkeiten zu vermeiden. Allein einen prägnanten und nicht zu langen Namen für das Projekt zu finden, hat schon Zeit gekostet.

Ein weiterer, eher leise im Hintergrund diskutierter Aspekt ist die nicht konsistente beziehungsweise nicht über alle Einzelspezifikationen durchgehende Unterstützung für das *JPMS* (via *module-info*). Das soll aber kommen, sobald eine einheitliche Namenskonvention für die einzelnen Module beschlossen wurde.

7. Juli 2023

Quarkus 3.2 mit 12 Monaten Long-Term-Support

Der neue Quarkus-Release 3.2 enthält unter anderem ein (allerdings noch experimentelles) Feature, um mit *@QuarkusComponentTest* CDI-Komponenten zu testen und Abhängigkeiten zu „mocken". Auch einige Verbesserungen im Bereich Security, insbesondere für die Nutzung von OIDC sind dabei. Das eigentliche „Feature" ist aber der neue Long-Term-Support. Release 3.2 und alle folgenden LTS-Releases sollen für 12 Monate mit Updates für alle kritischen Probleme – ob funktionale Bugs, Sicherheitslücken oder Performance-Probleme – versorgt werden. Die Zeitspanne ist nicht so lang wie zum Beispiel für Java SE, aber dürfte trotzdem den von der sehr schnellen Release-Kadenz bei Quarkus gestressten Nutzern eine Atempause geben. Neue LTS-Releases sollen zirka alle sechs Monate erscheinen, also mit etwa sechs Monaten Überlappung, sodass etwas Zeit bleibt, um auf das nächste LTS-Release zu wechseln.

13. Juli 2023

Spring 6.1

Spring 6.1 kommt bald heraus und bringt zum Beispiel einen *RestClient* mit, der das *Fluent-API* des *WebClient* mit dem *RestTemplate* verbindet. Wobei ich mit dieser Java-Schreibkonvention ja immer noch meine Schwierigkeiten habe. Ist der *RestClient* tatsächlich für REST-Implementierungen gedacht (Representational State Transfer) oder sorgt er einfach nur für Pausen (Rest)? Das ist doch Kamelkäse… Wie auch immer, ein paar andere nette Dinge sind ja auch dabei, zum Beispiel: Unterstützung für das JDK 21 und *Virtual Threads (Loom)*, vereinfachte direkte Validierung an Controller-Methoden für *MVC* und *WebFlux* oder Testunterstützung beim Aufzeichnen asynchroner Events aus anderen Threads.

20. Juli 2023

JDK 21 in „Rampdown 2"

JDK 21, der nächste Long-Term-Support-Release, befindet sich jetzt in der „Rampdown Phase 2", die Liste der Features ist also fix. Bis

zum geplanten Release-Datum am 19. September kommen nur noch Bug-Fixes.

Die einzelnen *JEPs* hatte ich in der letzten Ausgabe schon mal aufgezählt, bis auf das *Key Encapsulation Mechanism API* (JEP 452, zur Absicherung symmetrischer Keys mittels *Public-Key*-Kryptographie). Okay und JEP 449. Dieses „Feature" wurde schon in Adoptiums Plänen für 2023 angekündigt: Das Bauen des Windows 32-Bit x86 Ports wird schon mal als *deprecated* markiert, um es dann in einem Folge-Release ganz zu entfernen.

3. August 2023

Kommerzieller Support für *Temurin*

Die Eclipse-Adoptium-Seite ist um Informationen zu kommerziellem Java-Support für die projekteigene OpenJDK-Distribution *Temurin* erweitert worden. Für Firmen, die kommerziellen Support benötigen, und nicht eine herstellerspezifische Distribution bevorzugen, sondern direkt auf das Open-Source-Projekt setzen wollen, ist das eine gute Sache. Auf der Webseite *[6]* sind bislang drei Unternehmen aufgelistet, die sich selbst im Projekt engagieren und eben mit ihrem Wissen entsprechenden Support anbieten, teilweise auch in deutscher Sprache. Mal sehen, ob die Liste weiterwächst.

Jakarta EE 11

Bis zum geplanten Release-Datum im ersten Quartal 2024 ist noch ein bisschen Zeit, aber fast alle Spezifikationen haben jetzt ihre Pläne genehmigt bekommen oder stehen gerade zur Abstimmung.

Nur *Jakarta(-Bean)-Validation* muss den Plan noch vorbereiten. Vielleicht werden sie noch von der Diskussion aufgehalten, das Wort *Bean* aus dem Namen zu streichen. Ziemlich viel Aufwand, um vier Buchstaben loszuwerden (Suchen und Ersetzen in API und Spezifikation dürfte der kleinste Teil sein). Aber sie haben ja durch die vergangenen Umbenennungen von Java-EE schon ein bisschen Erfahrung gesammelt.

Zur Abstimmung stehen insbesondere drei Spezifikationen, die bislang *stand-alone* sind, aber in die Plattform mit aufgenommen werden sollen (was natürlich nochmal höhere Anforderungen an Konsistenz und Interoperabilität mit den anderen Spezifikationen mit sich bringt): (Jakarta) *Data 1.0, NoSQL 1.0* und *MVC 3.0*. Die Liste der Einzelspezifikationen mit ihren geplanten neuen Features durchzugehen, sprengt den Rahmen. Aber es gibt – nach den Releases, die sich hauptsächlich mit den technischen Notwendigkeiten der Migration von Java EE zu Eclipse/Jakarta beschäftigen mussten – jetzt zahlreiche interessante neue Features, zum Beispiel in *Security 4.0* oder *Concurrency 3.1*. Am einfachsten ist es wahrscheinlich, über die Liste in Ivar Grimstads Blog einzusteigen *[7]*.

Referenzen:

[1] *https://prismlauncher.org/news/cf-compromised-alert/*
[2] *https://www.heise.de/news/Minecraft-Modifikationspakete-mit-Fractureiser-Maleware-verseucht-9182068.html*
[3] *https://spring.io/blog/2023/06/21/docker-compose-support-in-spring-boot-3-1*
[4] *https://www.heise.de/blog/Java-Oracles-GraalVM-ist-ab-sofort-fuer-alle-kostenlos-9199872.html*
[5] *https://medium.com/graalvm/a-new-graalvm-release-and-new-free-license-4aab483692f5*
[6] *https://adoptium.net/de/temurin/commercial-support/*
[7] *https://www.agilejava.eu/2023/08/06/hashtag-jakarta-ee-188/*

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

Auch in dieser Ausgabe der Eclipse Corner muss ich leider wieder die Cassandra spielen. Ich hasse mich ja schon fast selbst dafür, aber ich habe mit meinem pessimistischen Fazit der letzten Ausgabe leider Recht behalten: Die Industrie zieht wieder einmal nicht mit. Im Gegenteil: Die seit jeher geringen Kapazitäten, die die Industrie in die Arbeit an Jakarta EE investiert hat, wurden augenscheinlich noch weiter ausgedünnt. So kommt es demnächst einmal mehr zu Versagen mit Ansage. Aber das sind wir von der Jakarta EE Working Group ja inzwischen gewohnt. Böse Zungen würden von „Vaporware" sprechen; ganz so weit möchte ich selbst nicht gehen. Auf der einen Seite gab es in der Vergangenheit durchaus ganz offiziell immer mal wieder Releases von Jakarta EE, und es gab ja auch irgendwann das eine oder andere Produkt, das zumindest behauptet hat, die Spezifikation zu erfüllen. Das ist doch weit mehr als nur heiße Luft.

Auf der anderen Seite wissen aber alle, die Jakarta EE produktiv einsetzen, dass es auch kaum mehr war als das. Obwohl nun seit mehreren Jahren „die Community" nach langem Kampf die Kontrolle über die ehemalige Java-2-Enterprise-Edition hat, kam doch nur wenig produktiv Nutzbares auf den Tisch. Ja, wir können inzwischen Java 17 benutzen, das ist großartig. Aber wir sind inzwischen bei Java 21, das ist eine Differenz von vier Java-SE-Releases beziehungsweise von zwei vollen Kalenderjahren (bezogen auf den geplanten Release-Termin von Jakarta EE 11 werden es sogar zweieinhalb sein). Bedenkt man, dass „die Community" eigentlich mal vorhatte, jedes Halbjahr eine Release von Jakarta EE zu veröffentlichen, um Java SE möglichst dicht zu folgen, muss man nach X Releases leider konstatieren, dass dieser Plan keineswegs durch die in der Working Group vertretenen Industriebetriebe unterstützt wird – allen Beteuerungen zum Trotz.

Viel eher bremst „die Industrie" eine schnelle Kadenz und eine dichte Versionsfolge auch weiterhin aus. Und nun wird es lustig: Es wird das Argument genannt, dass der Sprung auf Java 21 quasi viel zu ambitioniert und zu kurzfristig sei. Ja, mal wieder habe niemand in der Industrie voraussehen können, dass Java 21 ein LTS-Release sein werde und es bereits im September 2023 veröffentlicht werden soll. Dass OpenJDK aber exakt dies bereits seit Jahren bekanntgegeben hat, wird dabei gerne verschwiegen. Ebenso wie die Tatsache, dass der Inhalt und Umfang von OpenJDK bereits weitgehend bekannt war, als man den Zieltermin und Inhalt von Jakarta EE 11 geplant hat. So sind es nun eben jene Industriebetriebe, die behaupten, dass genau die von uns allen sehnsüchtig erwarteten Features von Java 21, wie beispielsweise Virtual Threads, angeblich eine so unglaublich große Herausforderung darstellen, dass man den geplanten Zieltermin Anfang nächsten Jahres nicht mehr einhalten kann. Während ich diese Zeilen schreibe (Anfang August 2023) wird bereits auf diversen Mailinglisten aktiv debattiert, gerade wegen Virtual Threads und anderen Java-21-Features den Zieltermin von Jakarta EE 11 doch lieber um etwa ein Quartal zu verschieben – noch bevor man überhaupt mit der Arbeit an den Teilspezifikationen von Jakarta EE 11 wirklich in nennenswertem Umfang begonnen habe. Aus meiner Sicht eine Peinlichkeit sondergleichen, da beispielsweise Jersey (der Kern von GlassFish und Payara) bereits während der Corona-Zeit in einer JavaLand-Session als Beta auf Virtual Threads demonstriert wurde. Exakt dort „geht" aber im Moment relativ wenig. Müssten wir denn nicht Unmengen an Commits und Merge Requests registrieren, wenn dort irgendein Zeitdruck wäre? Zumindest die GitHub-Statistik bestätigt aber vielmehr, dass dort (und übrigens auch anderswo) teilweise wochenlang gar nichts gearbeitet wurde!

Die Wahrheit ist doch meines Erachtens vielmehr, dass es einen klaren Zusammenhang zwischen den Post-Corona-Entlassungswellen der Industrie, dem fehlenden Fortschritt bei den Jakarta-EE-zertifizierten Produkten und dem geplanten Verschieben des Jakarta-EE-11-Releasetermins gibt. Sicherlich wurden damals nicht nur Raumpfleger*innen und Security-Personal „freigestellt", sondern auch (meines Erachtens: vor allem) teurere („entlassungs-lukrative") Senior Developer. Und wieso überhaupt Geld für einen Standard ausgeben, der auch der Konkurrenz nutzt, anstatt das verbleibende Personal nur noch an proprietären Features arbeiten zu lassen? Wie gesagt, ich deklariere das hier ganz klar als Meinung, aber ich denke, ich bin nicht der Einzige, der diese Nachtigall trapsen hört!

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.

JavaLand

www.javaland.eu

AM NÜRBURGRING

09. –
11.04. 2024

JETZT TICKETS

Präsentiert von: DOAG Heise Medien

NEUE LOCATION!

SICHERN!

Community Partner: iJUG Verbund

Unbekannte Kostbarkeiten des SDK Heute: Die Klasse ClassValue

Bernd Müller, Ostfalia
heute mit Gastautor Markus Karg, Head Crashing Informatics

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Die Klasse `ClassValue` wurde mit Java 7 in das JDK aufgenommen. Sie ist definitiv eine unbekannte Kostbarkeit und beschreibt sich selbst als „*Lazily associate a computed value with (potentially) every type*" *[1]*. Diese recht abstrakte Beschreibung – mit nicht sofort ins Auge fallenden sinnvollen Verwendungsmöglichkeiten – ist sicher mit ein Grund für den geringen Bekanntheitsgrad. Die Anzahl möglicher Anwendungsfälle ist unserer Meinung nach überschaubar, aber sie existieren. In dieser Ausgabe der unbekannten Kostbarkeiten werden wir zwei Beispiele der Verwendung von `ClassValue` vorstellen. „Wir" sind dieses Mal Markus Karg, dem regelmäßigen Java-aktuell-Leser als Autor der Eclipse Corner bekannt, und Bernd Müller. Die Idee und die maßgeblichen Arbeiten am Artikel gehen auf Markus zurück.

Die Verwendungsmöglichkeiten der Klasse ClassValue

Den ersten Satz zur Beschreibung der Klasse `ClassValue` im Package `java.lang` haben wir oben schon wiedergegeben. Hier nun der volle Wortlaut: „*Lazily associate a computed value with (potentially) every type. For example, if a dynamic language needs to construct a message dispatch table for each class encountered at a message send call site, it can use a ClassValue to cache information needed to perform the message send quickly, for each class encountered.*" *[1]*

Man sieht, dass der Autor der Klasse dynamische Sprachen, beziehungsweise deren Implementierungen, als potenzielle Anwendungsfälle sieht. In der Tat zeigt eine kurze Internet-Recherche, dass zum Beispiel Scala und Groovy die Klasse verwenden. Aber auch in der Java-Welt gibt es manchmal die Anforderung, (beliebige) Informationen an (beliebige) Typen (hier: Klassen) zu binden. Dies soll an zwei Beispielen konkretisiert werden: einem On-Demand-Hash-Code und dem Augmentieren von Business-Objekten mit Technikdetails.

On-Demand-Hash-Code

Im OpenJDK-Projekt Lilliput *[2]* wird daran gearbeitet, die Größe von Objekt-Headern und anderen Objektverwaltungsinformationen in der JVM zu minimieren. Eine Motivation hierfür ist, dass weniger Zeit in die Garbage Collection investiert werden muss, da insgesamt mehr Java-Objekte im gleichen Speicherbereich unterzubringen sind. Ein dazu diskutierter Ansatz ist der Verzicht auf einige Meta-Informationen eines jeden Java-Objektes, beispielsweise des Identitäts-Hash-Codes, also jenes statischen Wertes, den die Methode `System.identityHashCode(Object)` liefert. Dieser Wert wird als weitgehend ungenutzt betrachtet, aber trotzdem berechnet und die JVM muss ihn für jede Instanz jeder Klasse speichern. In Summe benötigt sie dazu erhebliche Mengen an Speicher.

Könnte man also auf diese 32 Bit pro Java-Objektinstanz verzichten oder zumindest nur dann Speicher für den Hash-Code belegen, wenn das betreffende Objekt auch wirklich nach seinem Hash-Code gefragt wird? Und viel wichtiger: Wie kann man ein Feld bei Bedarf zur Lauf-

zeit an eine Klasse respektive eine Objektinstanz anfügen? Tatsächlich geht das über den Umweg eines `ClassValue`, wie der folgende Quellcode am Beispiel eines hypothetischen Umbaus von `System.identityHashCode(Object)` demonstriert *(siehe Listing 1)*.

```
public static int identityHashCode(Object o) {
  return HashCodeCache.get(o);
}
```

Listing 1

Die verwendete Klasse `HashCodeCache` ist in *Listing 2* abgebildet. Der Code ist vollständig funktional, aber aus didaktischen Gründen stark vereinfacht. Das OpenJDK-Team würde zur Vermeidung der Boxing-/Unboxing-Kosten definitiv die generische `WeakHashMap` durch eine auf den Anwendungszweck optimierte `IntHashMap` ersetzen.

Das Beispiel ist verblüffend kurz, aber effektiv, und besteht genau genommen nur aus zwei Zeilen des zur Laufzeit ausgeführten, prozeduralen Codes. Die ganze Magie steckt in den JRE-Bestandteilen `ClassValue` und `WeakHashMap`. Wie man sieht, wird keineswegs wirklich dynamisch ein Feld an ein Objekt angefügt, sondern unter Ignoranz jeglicher Objektorientierung der virtuellen Maschine erklärt, sie möge sich über zwei Maps behelfen – eine hängt den `ClassValue` (hier: die zweite Map) an die Klasse, die zweite merkt sich (ohne die Garbage Collection zu verhindern) den ersten jemals berechneten Hash-Code einer Objektinstanz. In der Summe wird nur Speicher für jene Klassen und Instanzen verbraucht, deren Hash-Code mindestens einmal benötigt wurde, die noch referenziert werden, und somit noch nicht durch den Garbage Collector entsorgt werden können.

Ganz nebenbei bemerkt: Ab einer bestimmten Komplexität der Hash-Methode, beziehungsweise Anzahl und Typen der beteiligten Variablen, kann die Verwendung dieses Cache auch nützlich für die Implementierung der Methode `Object.hashCode()` in Projekten des Lesers sein; ein Microbenchmark mit JMH sollte Aufschluss bringen, wann der Break-Even-Point erreicht ist. Sofern jedoch von vorneherein feststeht, dass die eigene Klasse auf jeden Fall einmal in einer Collection landen wird, ist es sinnvoll, gleich einen vorberechneten Hash-Code in einem gewöhnlichen internen Feld zu speichern. Der Mehraufwand für den `ClassValue` lohnt nur dann, wenn diese Verwendungsform nicht bekannt oder zumindest unwahrscheinlich ist.

Augmentieren von Business-Objekten

Kommen wir zum zweiten Beispiel. Markus ist stets auf Separation of Concerns und verständlichen Code bedacht und sah sich mit einem simplen Problem konfrontiert: Der Zustand eines Geschäftsobjekts sollte über ein externes API (zum Beispiel HTTP) mit einem Schlüssel adressiert werden können, der sich nicht aus dem Zustand des Objekts ableiten lässt, da er rein durch die Speicherform, also technisch bedingt ist, und nicht Teil des Domänenmodells ist. Unter Informatikern: Es kommt ein nicht-natürlicher Primärschlüssel zum Einsatz, zum Beispiel eine vom Speichersystem automatisch vergebene UUID. Wie jedoch kann man bei Auflistung aller gespeicherten Objekte den Code sauber, also frei von rein technischen Sachverhalten halten, das heißt, eine Liste rein mit den Geschäftsdaten zurückgeben, ohne explizit auch die technisch ja durchaus benötigten Schlüssel stets mitzugeben?

Schauen wir uns die Ausgangslage, also den unsauberen Code, in Form einer klassischen JAX-RS-Anwendung an, der das Ganze mittels Map löst *(siehe Listing 3)*.

```
public record Book (String isbn, String title) {}

@Inject
private BookStore store;

@GET
@Path("books")
public Map<UUID, Book> list() {
  return store.list();
}
```

Listing 3

Tatsächlich widerspricht es modernen Software-Engineering-Grundsätzen, dass Business-Modell mit technischen Sachverhalten zu verunreinigen. Dass ein Buch aus internen, technischen Gründen des `BookStore` einen Schlüssel in Form einer UUID besitzt, sollte in der JAX-RS-Ressource nicht zu erkennen sein. Schöner wäre es, dies der Technologie-Anpassungsebene, im JAX-RS-Terminus einem `MessageBodyWriter` oder `WriterInterceptor`, zu überlassen, wodurch der Code wieder rein business-orientiert wird *(siehe Listing 4)*.

```
private static class HashCodeCache extends ClassValue<Map<Object, Integer>> {

  private static final HashCodeCache SINGLETON = new HashCodeCache();

  private HashCodeCache() {};

  static int get(Object o) {
     return SINGLETON.get(o.getClass()).computeIfAbsent(o, Object::hashCode);
  }

  @Override
  protected Map<Object, Integer> computeValue(Class<?> ignored) {
     return new WeakHashMap<Object, Integer>();
  }
}
```

Listing 2

```
@GET
@Path("books")
public List<Book> list() {
  return store.list();
}
```

Listing 4

Die störende UUID ist nun zwar verschwunden, doch auf HTTP-Ebene ist es weiterhin zwingend notwendig, dem Aufrufer einen URI mitzuteilen, der eben jene UUID enthält. Woher aber soll der `MessageBodyWriter` beim Rendern des JSON-Ergebnisses wissen, wie die UUID jedes Buches lautet? Die Lösung liegt auch in diesem Fall in einem `ClassValue`. Die in *Listing 5* dargestellte Klasse ID erbt von `ClassValue` und realisiert dies. Die Verwendung der Klassen Book und UUID dienen der besseren Verständlichkeit im Rahmen dieses Artikels. Der Produktiv-Code verwendet ausschließlich die Klasse `Object`, da die tatsächlichen Datentypen von Payload (JAX-RS Entity) und Adresse (JAX-RS URI) für Produzenten und Konsumenten und ebenso für den `ClassValue` irrelevant sind.
Im Unterschied zum ersten Beispiel wird der zugeordnete Wert hier nicht durch den `ClassValue` selbst berechnet, sondern explizit über die Methode `ID.assign(Book, UUID)` zugewiesen. Dies erfolgt durch den Produzenten der Adresse, in diesem Fall also den `BookStore`. Das Auslesen geschieht in einem `WriterInterceptor` per `ID.of(Book)`, der dann per `setEntity()` jedes Buch durch einen `Map.Entry<Book,UUID>` ersetzt. Der `MessageBodyWriter` erhält somit also wieder eben jene Kombination, nur nicht von der Business-Ebene (Ressource), sondern von der Technik-Ebene (`WriterInterceptor`). Aus Platzgründen, und da es thematisch zu weit abschweift, verzichten wir auf die Darstellung des `MessageBodyWriter` und `WriterInterceptor`.

```
public class ID extends ClassValue<Map<Book, UUID>> {

  private static final ID SINGLETON = new ID();

  private ID() {};

  public static void assign(Book book, UUID id) {
     SINGLETON.get(Book.class).put(book, id);
  }

  public static UUID of(Book book) {
     return SINGLETON.get(Book.class).get(book);
  }

  @Override
  protected Map<Book, UUID> computeValue(Class<?> ignored) {
     return new WeakHashMap<Book, UUID>();
  }
}
```

Listing 5

Wieso benutzen wir nicht einfach direkt einen `WeakHashMap`-Singleton statt eines `ClassValue`-Singletons? Der Grund ist, dass wir uns hier in einer Jakarta-EE-Umgebung bewegen, somit für Produzenten und Konsumenten von Buch-IDs separate Classloader-Hierarchien Anwendung finden könnten, die im schlimmsten Fall nichts gemeinsam haben außer der Java-Runtime. Entsprechend würden zwei Singletons existieren, einer, in den geschrieben werden würde, und einer, aus dem gelesen werden würde, womit die Kommunikation nicht gewährleistet wäre. Gerade für Jakarta-EE-Einsteiger eine schwer zu erkennende Fehlerquelle.

Zusammenfassung

Die Klasse `ClassValue` erlaubt es, praktisch beliebige Informationen an eine Klasse zu binden. Ihre Verwendung empfiehlt sich, wenn ein Attribut einer Klasse nur bei sehr wenigen Instanzen von seinem Initialwert abweicht oder, wenn ein zusätzliches Attribut benötigt wird, das aber nicht zu einer existierenden Klasse hinzugefügt werden kann oder soll. Im ersten Fall, der Abweichung vom Initialwert, wird Speicherplatz gespart. Der zweite Fall, das Hinzufügen eines Attributs, wird praktiziert, wenn der Quell-Code nicht vorliegt, oder aus methodischen Gründen, im Beispiel etwa Separation of Concerns, wenn die Code-Qualität erhöht werden soll.

Referenzen

[1] *https://docs.oracle.com/javase/7/docs/api//java/lang/ClassValue.html*

[2] *Lilliput. https://openjdk.org/projects/lilliput/*

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.

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.

Wie kommt meine Java-Webanwendung in die Azure- oder AWS-Cloud?

Frank Pientka

Der Erfolg von Java spricht für sich. Java wurde zu einer Zeit entwickelt, in der das Internet noch in den Kinderschuhen steckte. Inzwischen ist die Cloud das dominierende Betriebsmodell für Anwendungen. Deswegen wollen wir schauen, wie man eine typische Java-EE-Webanwendung mit einer relationalen Datenbank, jenseits von Lift und Shift, in die Azure- oder AWS-Cloud bringen kann.


Azure

Welche Optionen gibt es?

Eine Java-EE-Webanwendung benötigt einen Applikationsserver, zum Beispiel Apache Tomcat, und wird im WAR-Format deployt. Da der Applikationsserver sich um die Ressourcenverwaltung und um die Sicherheit kümmert, wird der Webserver entsprechend konfiguriert. Das hat den Vorteil, dass Entwicklung und Betrieb jeweils separat betrachtet werden können. Es hat jedoch den Nachteil, dass zum Beispiel Änderungen am Webserver oder durch andere Anwendungen, die dort laufen, sich auch auf die eigene Webanwendung auswirken. Mit dem verstärkten Einsatz von DevOps und Microservices wird die Trennung zwischen Entwicklung und Betrieb aufgehoben und die Anwendung mit einem integrierten Webserver als ausführbare JAR-Datei ausgeführt. Im nächsten Schritt wird diese in einen Container gepackt, um Abhängigkeiten zur jeweiligen Ablaufumgebung zu minimieren. Heute erleichtern Frameworks wie Spring Boot oder Quarkus viele Schritte und Anpassungen, um eine Anwendung Cloud-native zu entwickeln. Deswegen gibt es in der Cloud, jenseits von Lift und Shift und IaaS-Diensten, die Möglichkeit, entweder PaaS-, CaaS- oder FaaS-Dienste zu verwenden, um Java-Webanwendungen zu betreiben *(siehe Abbildung 1)*.

Um sich einen schnellen Überblick über die „Cloudfähigkeit" seiner Java-Anwendung zu verschaffen, bietet sich der *Cloud Suitability Analyzer [11]* an, da er auf der Kommandozeile ausgeführt werden kann und einen schönen Bewertungsbericht für die weiteren Maßnahmen erstellt. Da es für Azure spezielle Regeln gibt, geht er über die allgemeine Bewertung der 12-Faktor-App-Kriterien hinaus. Außerdem sind einige Regeln auch für *OpenRewrite* verfügbar, sodass manche Anpassungen auch automatisiert durchgeführt werden können.

Der älteste Dienst, den AWS hier auf Basis seiner Virtualisierungslösung EC2 als gemanagten Orchestrierungsdienst anbietet, ist *AWS Elastic Beanstalk*. Damit kann man mit wenigen Klicks eine gemangte Tomcat-Umgebung mit einem Applikations-Lastverteiler und einer relationalen Datenbank in einer oder bis zu drei Verfügbarkeitszonen anlegen. Dabei wird automatisch ein eigenes privates und öffentliches Netzwerk (VPC) mit IP-Adressen und bereits freigeschalteten Ports (über *SecurityGroups* und *Network Access Control Lists*) zur Verfügung gestellt. So ist eine Webanwendung in wenigen Minuten hochverfügbar und skalierbar im Internet aufrufbar. Gleichzeitig muss man sich durch den PaaS-Ansatz nicht um die Überwachung oder Pflege der Infrastruktur oder der benötigten Ressourcen kümmern und kann sich einfach auf die Entwicklung und das Deployment der Anwendung konzentrieren. *Elastic Beanstalk* unterstützt hier auch verschiedene Deployment-Verfahren wie *In-Place*, *Rolling-Upgrade* oder *Blue/Green-Deployments*, sodass man Änderungen mit weniger Risiko in Produktion bringen kann.

Elastic Beanstalk hat den Einstieg in die Nutzung von AWS für Java-Entwickler vor 12 Jahren erheblich vereinfacht, ohne sich zu sehr mit deren Konzept auseinandersetzen zu müssen. Gleichzeitig konnten später weitere AWS-Dienste bei weiterem Bedarf einfach genutzt werden. Möchte man seine Webanwendung als Microservice, JAR oder Container ausführen, ist das später auch einfach möglich. Deswegen ist *Elastic Beanstalk* zum Einstieg und für mittlere Anwendungen immer noch eine gute Option. Wenn man jedoch noch mehr Dienste containerisiert betreiben möchte, kommt man an einen *Orchestrator* wie Kubernetes (k8s) nicht vorbei.

Da es neben EC2/Elastic Beanstalk auch noch Lambda und EKS/ECS/Fargate gibt, um Java-Anwendungen bei AWS laufen zu lassen, muss man frühzeitig entscheiden, für welche Bereiche man lieber die Verantwortung abgibt, oder, ob man diese behalten möchte, um weniger Abhängigkeiten und mehr Flexibilität zu haben.

Wenn Container, wie viele und wo?

Hier bietet AWS Amazon *Elastic Container Service* (Amazon ECS) und *Amazon Elastic Kubernetes Service* (Amazon EKS) an, die einem das Managen der *Control-Plane* abnehmen und die Konfiguration der *Worker-Nodes* vereinfachen. Bei EKS handelt es sich um eine zertifizierte Kubernetes-Distribution, die man auch selbst außerhalb von AWS on-premises installieren könnte, was jedoch eher selten genutzt wird. Um es noch einfacher zu machen, bietet AWS mit *Fargate* einen serverlosen Kubernetes-Service an, der die Verwendung von EKS noch mal erleichtert. Mit Lambda bietet AWS hier einen sehr populären aber auch proprietären Dienst an, um serverlose Anwendungen als FaaS entweder mit Java SE oder im Container zur Verfügung zu stellen, ohne sich um die Infrastruktur und die benötigten Ressourcen zu kümmern. Das muss jedoch zu den damit umgesetzten Anwendungsfällen passen und man verliert durch die Verwendung des proprietären Ausführungsmodells auch die von Java geschätzte Fähigkeit der Portabilität. Ebenso gibt es mit Aurora eine serverlose Datenbank mit Kompatibilität für mySQL und PostgreSQL.

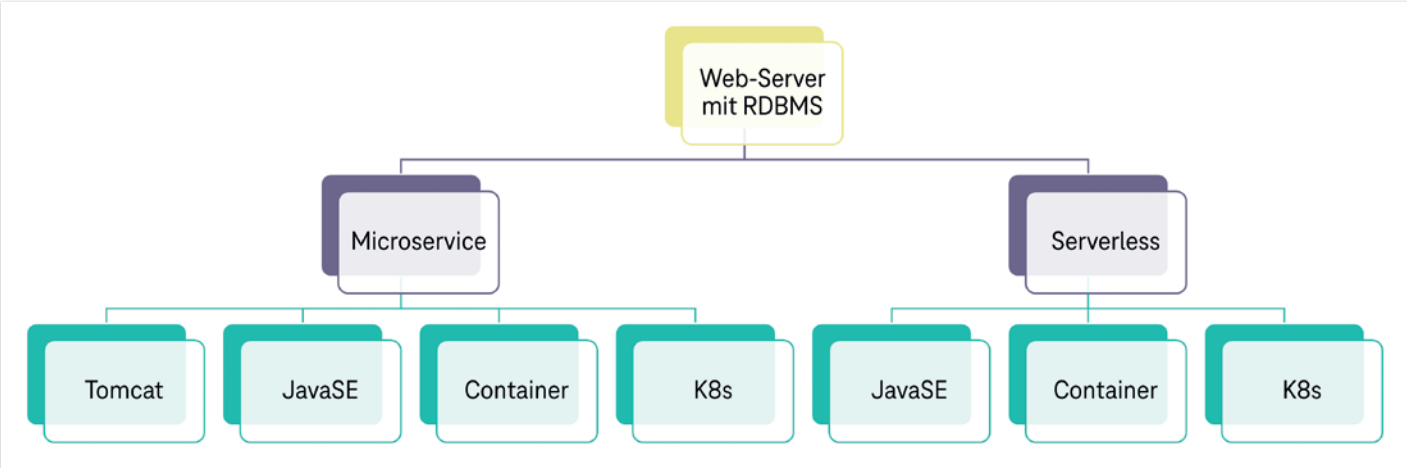


Abbildung 1: Wohin soll es mit meiner Webanwendung gehen?

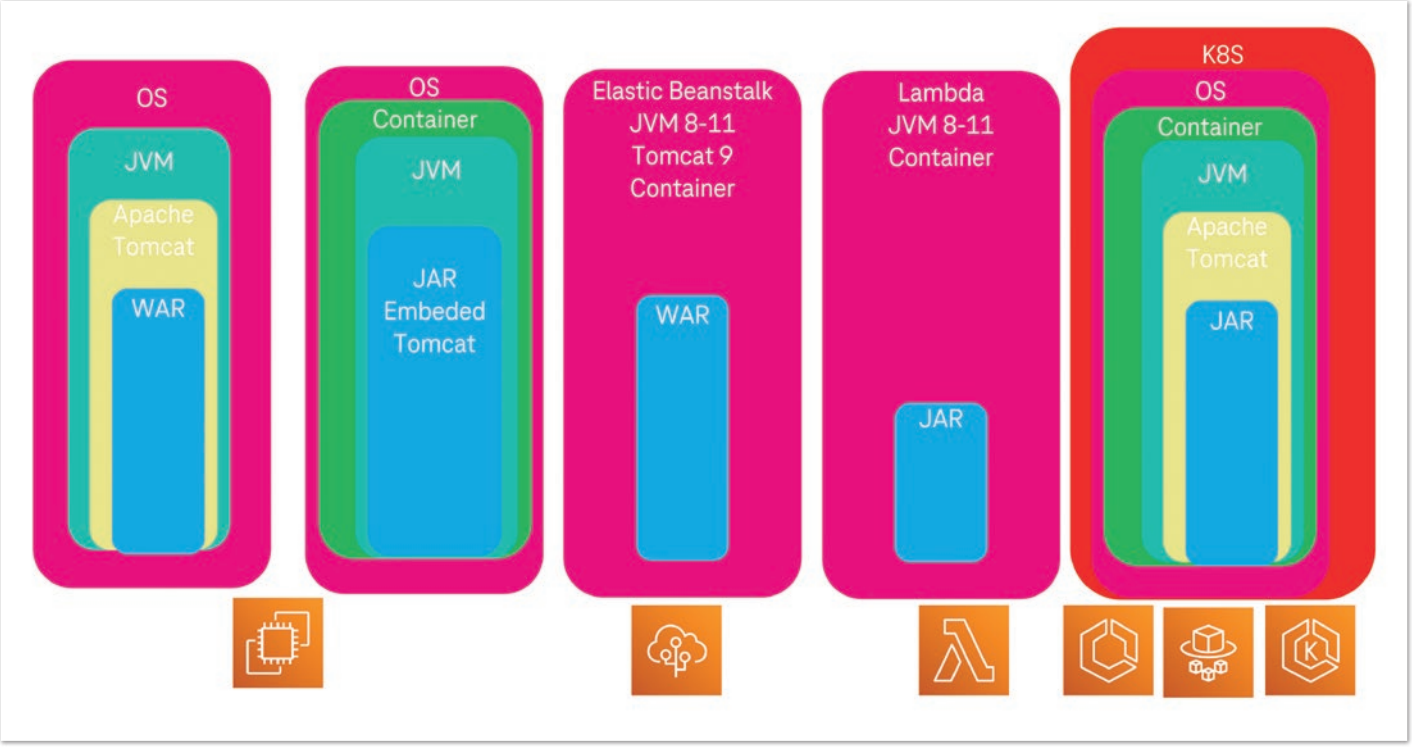


Abbildung 2: Welche Möglichkeiten gibt es bei AWS für eine Java-Webanwendung?

Doch auch hier haben Komfort, elastische Skalierbarkeit und Ausfallsicherheit ihren Preis, sodass man immer prüfen sollte, welche Anforderungen man hat, um den passenden AWS-Dienst dafür zu verwenden. Einige typische Möglichkeiten für eine Java-Webanwendung sind in *Abbildung 2* dargestellt.

Welche Möglichkeiten bietet hier Azure gegenüber AWS für Java an? Bei vielen Diensten gleichen sich beide Anbieter immer mehr an. Die große Bedeutung, die Java für beide Anbieter hat, zeigt sich in der guten Unterstützung in den SDKs von AWS *[4]* und Azure *[3]*, aber auch in der Mitarbeit im Adoptium-Projekt. So stellen Microsoft *[5]* (mit Unterstützung von Azul) und Amazon eigene OpenJDK-Distributionen zur kostenlosen Verwendung mit Support zur Verfügung. Bei der AWS Distribution Corretto *[6]* gibt es einige Erweiterungen, die man auch außerhalb der Cloud verwenden kann. Wenn möglich, sollte man für die Entwicklung und den Betrieb die jeweilige Distribution verwenden, da diese die dort verwendete Hardware am besten unterstützen.

Ähnlich wie AWS, bietet Microsoft mit Azure-Functions und Azure-App-Service unter Linux für Java SE und Apache Tomcat an. Zusätz-

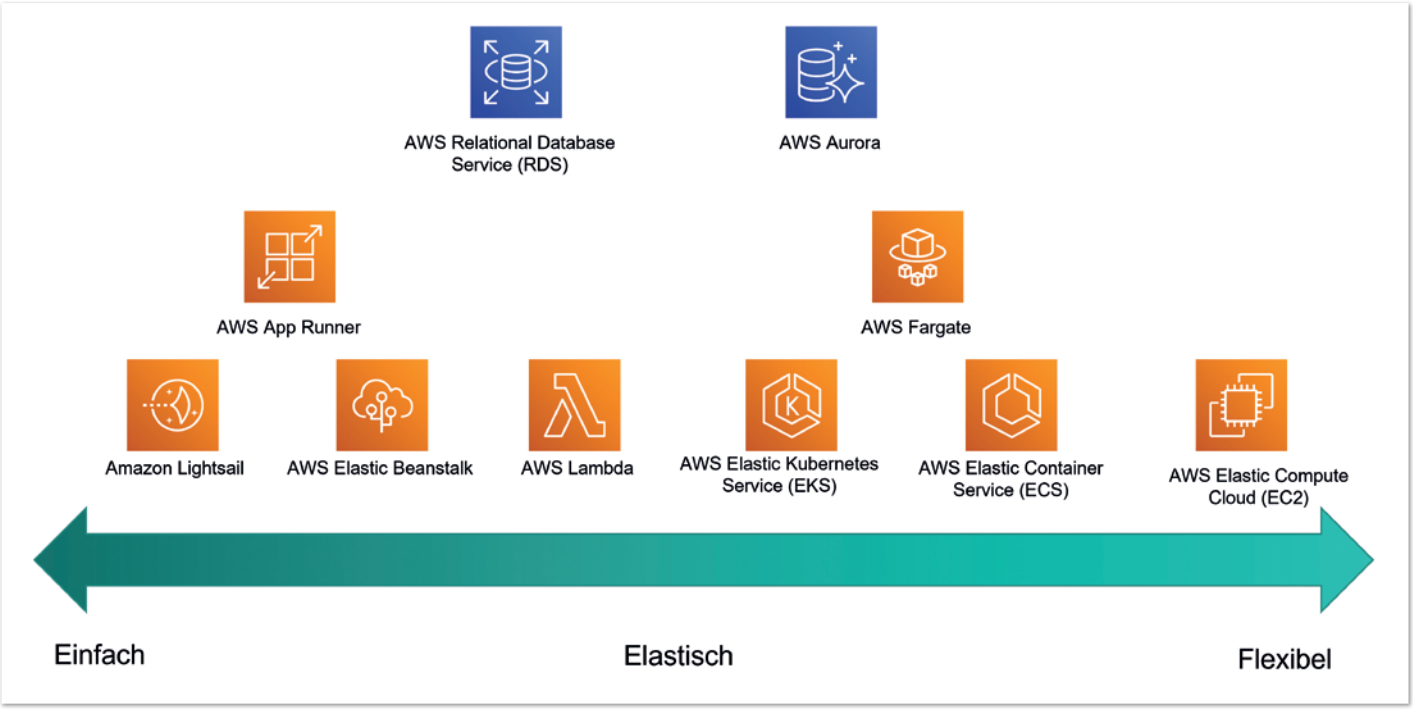


Abbildung 3: AWS Compute-/RDS-Optionen

lich wird hier auch JBoss EAP oder Azure-Spring-Apps (ehemals Azure-Spring-Cloud-Service) unterstützt. Hier pflegt Microsoft sowohl zu RedHat als auch VMware mit Spring eine Partnerschaft, sodass man darüber auch automatisch einen Enterprise-Support erhält. Deswegen bietet Microsoft neben dem eignen Azure-Kubernetes-Service (AKS) und Distribution auch Azure-RedHat-OpenShift an. OpenShift wird bei AWS nur über den Marktplatz angeboten und ist deshalb nicht so gut integriert und supportet wie bei Azure.

Inzwischen sind die Kubernetes-Distributionen sowohl von AWS als auch von Azure sehr nah am Standard, sodass diese oft kurz nach dem offiziellen Release einer Kubernetes-Version für diese zur Verfügung stehen. Bei OpenShift dauert die Unterstützung der neuen Kubernetes-Version nicht mehr so lange wie früher, aber immer noch etwas länger als bei anderen.

Mit Spring Cloud ist man bei AWS und Azure auf der sicheren Seite. Wobei hier Azure durch die direkte Partnerschaft mit VMware mehr eigene Dienste unterstützt als AWS.

Auf jeden Fall sollte man für die sichere Verwaltung der Anmeldeinformation bei Azure dessen *Key Vault*, oder bei AWS dessen *Secrets Manager* verwenden. Ähnliches gilt auch für die Verwaltung der Konfigurationsparameter. Hier bietet AWS beispielsweise mit dem System-Manager, oder noch besser mit dem App-Config-Dienst, eine einheitliche und komfortable Lösung an, die man sowohl für die AWS-Dienste als auch die eigenen Anwendungen nutzen kann. Azure bietet für seine SQL-Datenbank im JDBC-Treiber die Möglichkeit an, die Anmeldeinformationen im Key Vault oder *Azure Active Directory* abzulegen. Für andere von Azure unterstützte Datenbanken jedoch leider nicht. Hier ist AWS konsequenter und bietet zumindest für MySQL und PostgreSQL eine Wrapper-Bibliothek an, die man mit dem jeweiligen Standard-JDBC-Treiber mitinstallieren kann. So kann man auch die Anmeldeinformationen im Secrets Manager nutzen oder von dem automatischen Failover bei einer Multi-AZ-Datenbankinstallation profitieren.

Wie bekomme ich meine Datenbank in die Cloud?

Bei der Migration einer Anwendung in die Cloud sollte man die Daten nicht vergessen. Hier gibt es sowohl bei Azure als auch AWS mit MS SQL, MySQL und PostgreSQL ein gutes Angebot an gemanagten aktuellen Datenbankprodukten.

Bei AWS kommt hier noch Oracle (mit eingeschränktem Funktionsumfang) hinzu. Zusätzlich bietet AWS mit *Babelfish* auch die Möglichkeit, den MS SQL-Dialekt zur Laufzeit in PostgreSQL zu übersetzen. Für PostgreSQL und MySQL hat Azure mit *Flexible Server* die Datenspeicherarchitektur angepasst, sodass die Daten bereits in mehreren Verfügbarkeitszonen gespeichert werden, ohne dass man sich über eine Multi-Master-Konfiguration selbst darum kümmern muss. Gleiches hat AWS mit Aurora gemacht und diese mit globalen Tabellen sogar Multi-Region-fähig gemacht. Hier bieten die Cloud-Provider echte Innovationen und neue Anwendungsfälle gegenüber den Standard-Open-Source-Produkten an, wie sie sonst nur über kommerzielle Erweiterungen angeboten werden.

Der einfachste Weg, wenn man mit Replatforming beim kompatiblen Produkt bleibt, geht über das Einspielen eines aktuellen Backups, wenn die Versionssprünge nicht zu groß sind. Kann man sich keine Ausfallzeiten leisten, kann man Änderungen über eine bereits

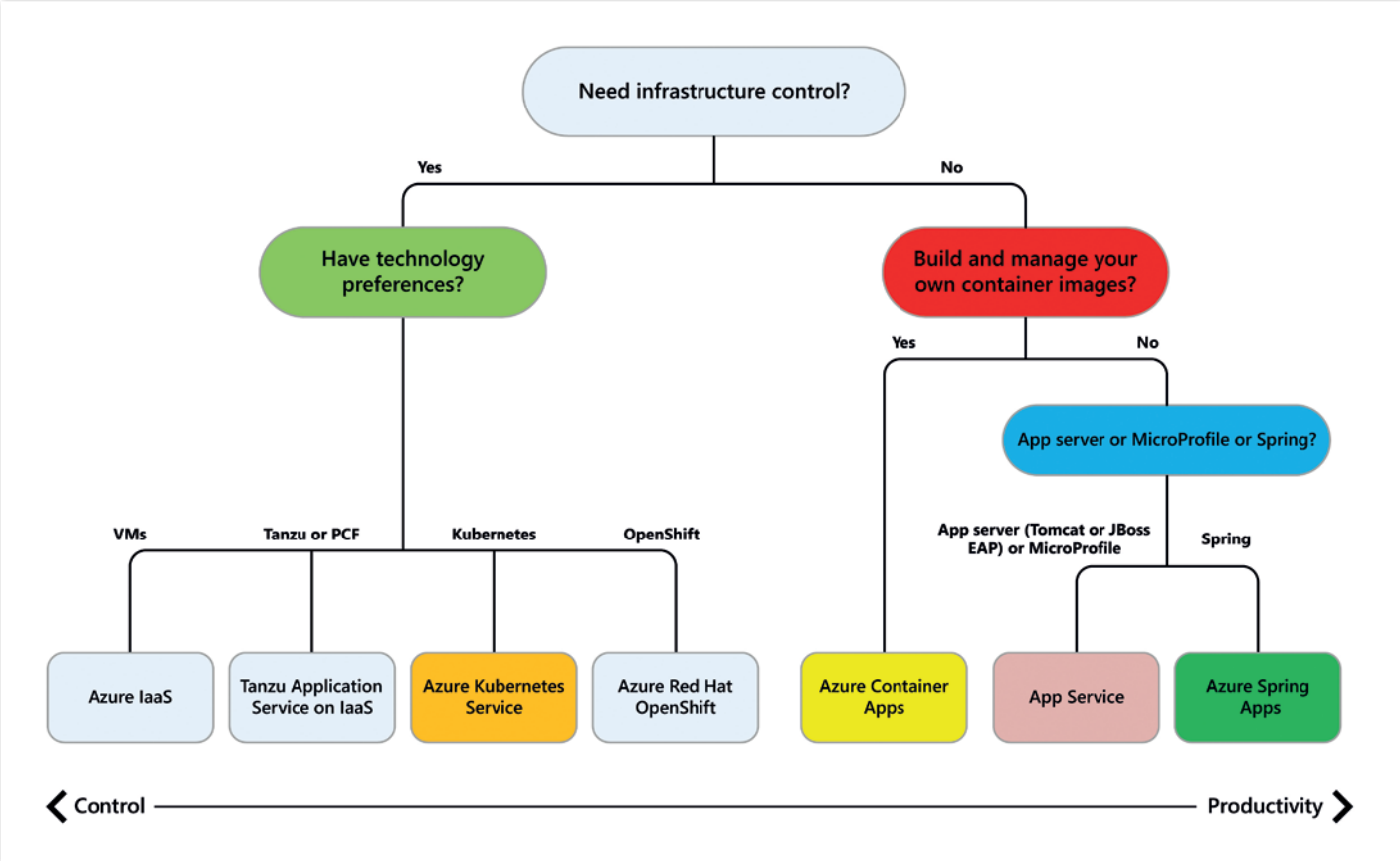


Abbildung 4: Deployment-Möglichkeiten von Java-Anwendungen auf Azure

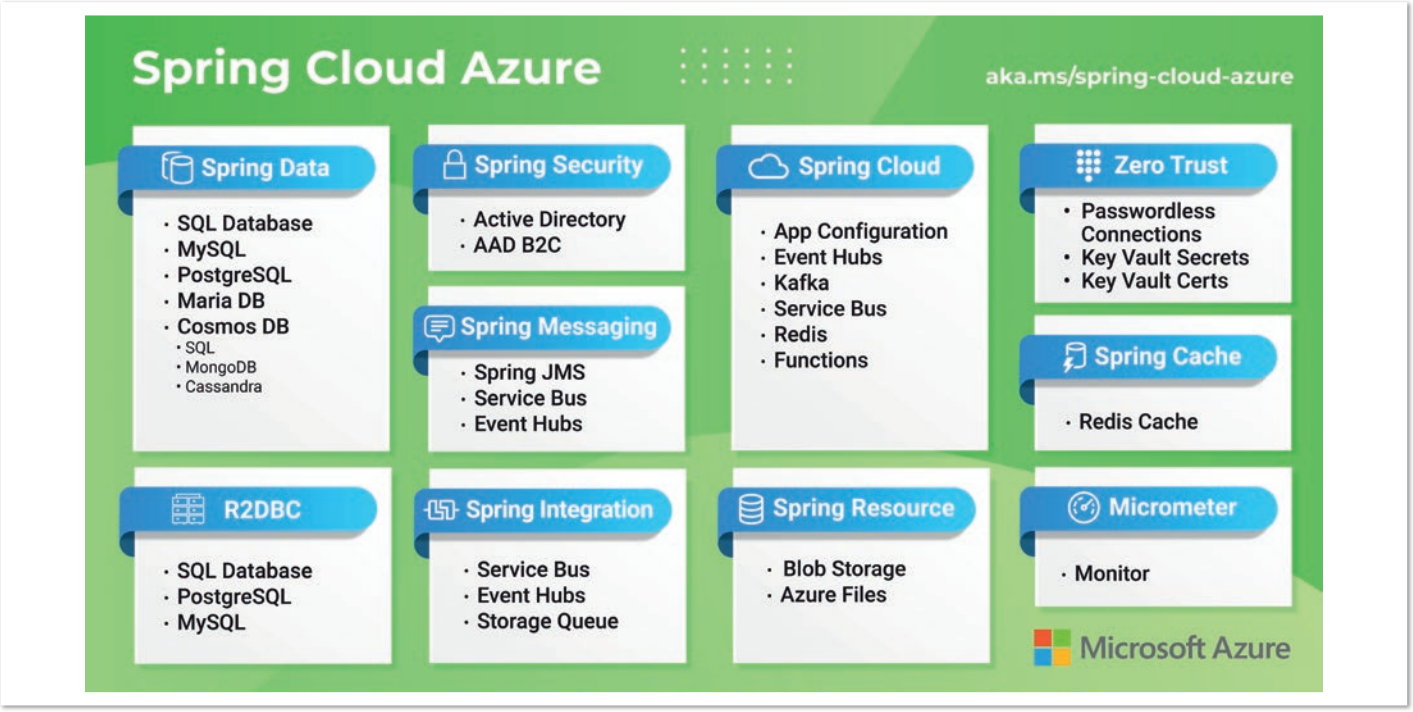


Abbildung 5: Spring Cloud Azure – Überblick der Funktionen [2]

in der Cloud vorbereitete Instanz durchführen, die mittels Replikation oder dem *Change-Data-Capture-Verfahren* (CDC) erstellt worden ist. Kommt es zu einem Produktwechsel, muss man zunächst die Schemata anpassen und die Daten über einen anderen Weg einspielen. Doch auch hier bieten die Cloud-Anbieter gemanagte und kostenlose Dienste und unterstützende Dokumente an.

Bei AWS wird zum Konvertieren des Schemas der SCT-Dienst eingesetzt und zum Kopieren und Konvertieren der Daten der DMS-Dienst. Für die Migration der Daten und die Konvertierung der Schemata zu Azure, kann der Azure-Database-Migration-Service genutzt werden.

In *Abbildung 6* ist dargestellt, wie und mit welchen Diensten eine Java-Webanwendung mit relationaler Datenbank zu AWS migriert werden kann. Hier werden die Strategien *Rehost* (mit AMS auf EC2) oder *Replatform* (mit ElasticBeanstalk und RDS) gezeigt. Wenn man die Daten nicht konvertieren muss (mit AWS SCT möglich), da man das Datenbankprodukt austauscht (zum Beispiel Oracle durch PostgreSQL) und man sich Ausfallzeiten leisten kann, wird man mit einem vollständigen Backup und Restore gut auskommen. Braucht man jedoch eine Live-Migration mit schnellem Umschalten oder zeitnahem Testen, so bietet sich ein Replikationsverfahren (zum Beispiel Read-Replica, CDC oder AWS DMS) an.

Fazit

Die gute und breite Unterstützung durch AWS *[7, 10]* und Azure *[1, 8, 9]* zeigt, dass Java für die Cloud nicht zum alten Eisen gehört. Dabei gibt es viele Wege in die Cloud. Die Entscheidung, ob man diesen

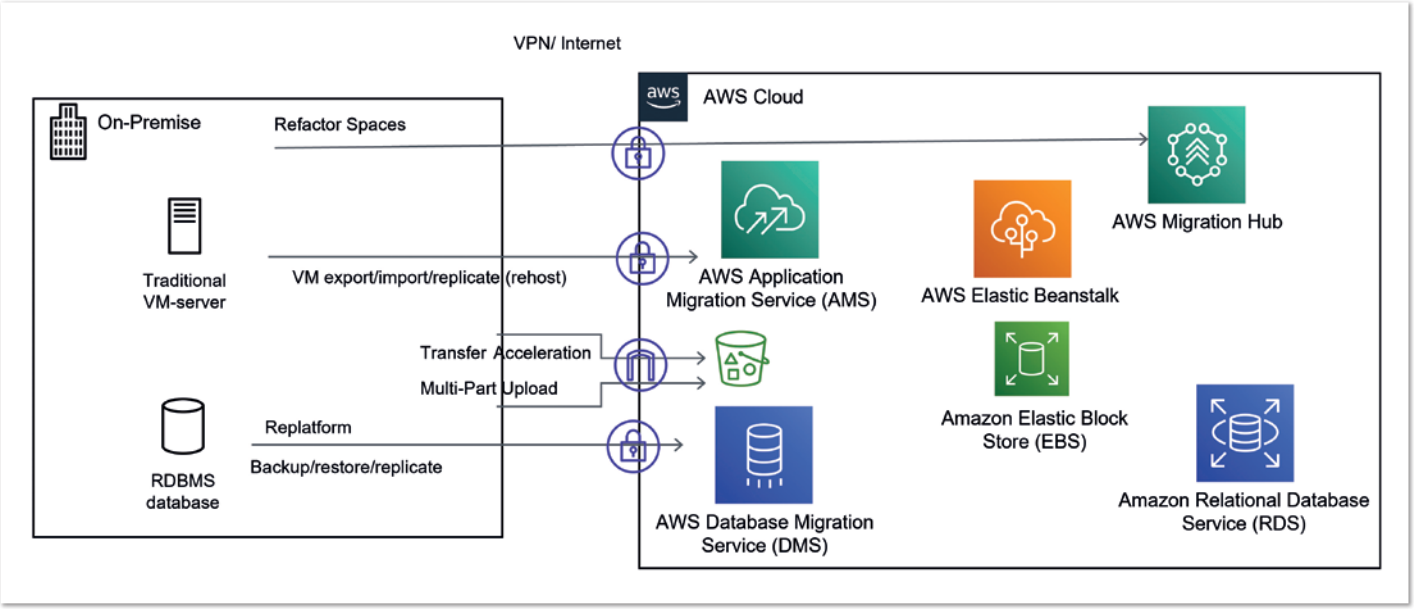


Abbildung 6: Migration einer Webanwendung mit relationaler Datenbank zu AWS

mit AWS oder Azure gehen möchte, hängt von den eigenen Anforderungen, dem Wissen und den vorhandenen Ressourcen ab. Oft wird das heute auf gemanagte Kubernetes-Dienste hinauslaufen. Jedoch sind diese aufgrund der Kosten oder dem benötigten Wissen für viele immer noch eine große Hürde. Aber man kann auch mit gemanagten Tomcat-, Spring- oder einfachen Container-Diensten viele Anwendungsfälle abdecken und eine Migration in die Cloud vereinfachen.

Für die meisten Java-Anwendungen ist es fast egal, welcher Weg gewählt wird. Um unnötige Aufwände und Lehrgeld zu vermeiden, ist es besser, mit einer einfacheren Option zu starten und erst später auf serverlose oder Kubernetes-Varianten zu wechseln, da dazu mehr angepasst und viele neue Werkzeuge und Produkte beherrscht werden müssen.

Mit der Zeit wird man gerade hinsichtlich Neuentwicklungen von der Vielzahl an Möglichkeiten in der Cloud profitieren, wenn man seine Governance- und Betriebsprozesse darauf ausrichtet. Ich wünsche Ihnen viel Erfolg auf der weiteren Reise in die Cloud, egal mit welchem Anbieter sie unterwegs sein mögen.

Referenzen

[1] Migration von Java zu Azure *https://learn.microsoft.com/azure/developer/java/migration*

[2] Spring Cloud Azure *https://spring.io/projects/spring-cloud* *https://spring.io/projects/spring-cloud-azure*

[3] Azure SDK for Java *https://azure.github.io/azure-sdk/#java*

[4] AWS SDK for Java 2.x *https://aws.amazon.com/sdk-for-java*

[5] Microsoft Build of OpenJDK *https://learn.microsoft.com/java/openjdk*

[6] Amazon Corretto *https://aws.amazon.com/corretto*

[7] AWS Java-Entwicklerzentrum *https://aws.amazon.com/developer/language/java*

[8] Azure-Dokumentation für Java-Entwickler *https://learn.microsoft.com/azure/developer/java*

[9] Migrieren von Spring Boot-Anwendungen zu Azure App Service *https://learn.microsoft.com/azure/developer/java/migration/migrate-spring-boot-to-app-service*

[10] From Zero to Production with Spring Boot and AWS Book, Björn Wilmsmann, Tom Hombergs, Philip Riecks *https://stratospheric.dev*

[11] Cloud Suitability Analyzer *https://learn.microsoft.com/azure/developer/java/migration/cloud-suitability-analyzer*

Frank Pientka

Frank.Pientka@gmx.de

Frank Pientka (@fpientka) ist Gründungsmitglied der iSAQB und arbeitet als Cloud Architect. Dabei begleitet er seine Kunden bei ihrer Reise in die Cloud. Er besitzt jahrzehntelange Erfahrung in der Modernisierung von Java-Anwendungen, und hat mehrere AWS-Zertifizierungen.

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

Wie baut man Helm Charts für Spring-Boot-Anwendungen?

Jan Weyrich, viadee Unternehmensberatung AG

Spring Boot ist ein Framework für die Entwicklung von Java-Anwendungen und Kubernetes ist einer der De-facto-Standards für die Bereitstellung von Anwendungen in Cloud-Umgebungen. Die Bereitstellung von Apps in einem Kubernetes-Cluster kann jedoch mühsam werden, da sich die Konfigurationen in verschiedenen Kubernetes-Deskriptoren wiederholen. Das Verwalten der Konfiguration wird noch aufwändiger, wenn mehrere Umgebungen wie Entwicklungs-, Test- und Produktionsumgebungen verwendet werden.

Helm löst diese Probleme durch Templates, Paketierung, Versionierung und einer Infrastruktur für die gemeinsame Nutzung und Wiederverwendung von Templates. In diesem Artikel wird Helm kurz vorgestellt und erläutert, wie es für Spring-Boot-Anwendungen genutzt werden kann.


HELM

Wofür braucht man Helm?

Das Deployment von Spring-Boot-Anwendungen auf Kubernetes erfordert die Erstellung einer Reihe von Manifestdateien. Eine Kubernetes-Manifestdatei ist meist im YAML-Format geschrieben und enthält alle Informationen, die zum Betreiben einer Anwendung benötigt werden. Diese Manifestdateien enthalten unter anderem das verwendete Container-Image, die Anzahl der auszuführenden Replikate und die einem Container zuzuweisenden Hardware-Ressourcen.

Manifestdateien neigen dazu, schnell zu wachsen und unübersichtlich zu werden, wenn weitere Ressourcen und Konfigurationen hinzugefügt werden. Angenommen, Sie haben eine Manifestdatei, die eine Deployment-Ressource für eine Anwendung definiert und eine andere Manifestdatei, die eine Service-Ressource für dieselbe Anwendung definiert: Beide enthalten wahrscheinlich ähnliche Konfigurationen, wie beispielsweise Labels und Annotationen, und verwenden ein ähnliches Namensschema. Die Pflege solcher Dateien ist mühsam und fehleranfällig, vor allem, wenn die Anwendung in mehreren Umgebungen laufen soll. In diesem Fall müssen für jede Umgebung separate Manifestdateien erstellt und gepflegt werden, was zusätzlich Raum für Fehler schafft. Helm löst diese Probleme, indem es ermöglicht, Templates für Manifestdateien zu erstellen. Darüber hinaus ermöglicht es, diese Vorlagen zu verpacken, zu versionieren und hochzuladen, damit sie von anderen einfach wiederverwendet werden können.

Wie verwendet man Helm? Was ist ein Helm Chart und wie ist es strukturiert?

Ein Helm Chart ist ein Paket, das alle Ressourcendefinitionen enthält, die für die Ausführung einer Anwendung in einem Kubernetes-Cluster erforderlich sind. Es ist eine Sammlung von Templates, Kubernetes-Manifestdateien und anderen Ressourcen, die den gewünschten Zustand einer Anwendung beschreiben. Ein Chart ist als Verzeichnis organisiert, das mehrere Dateien und Unterverzeichnisse enthält. Die wichtigsten Dateien und Ordner sind:

- `chart.yaml`: Diese Datei enthält Metadaten über das Chart, wie den Namen, die Version und die Beschreibung des Charts.
- `values.yaml`: Diese Datei enthält Standardwerte für die Konfigurationsoptionen des Charts.
- `templates`: Dieser Ordner enthält die Vorlagen, die die Kubernetes-Ressourcen definieren, aus denen die Anwendung besteht. Die Vorlagen verwenden die Go-Template-Language, sodass Variablen und Kontrollstrukturen verwendet werden können, um die Erzeugung der Manifestdateien zu beeinflussen.

Ein Chart hat häufig die folgende Dateistruktur *(siehe Listing 1)*.

Wie funktioniert der Templating-Mechanismus in Helm?

Der Templating-Mechanismus in Helm verwendet die `text/template`-Bibliothek von Go, um Template-Dateien zu verarbeiten und Kubernetes-Manifeste zu erzeugen. Die Templates definieren die Ressourcen der Anwendung, wie beispielsweise Deployments, Services und Ingresses, und verwenden Platzhalter, um zur Laufzeit festgelegte Variablen darzustellen. Helm ermöglicht es, Variablen zu definieren und diese mit folgender Syntax in den Templates zu verwenden: `{{ .Values.Variablenname }}`. Das .-Zeichen bezieht sich auf die Wurzel des Helm Charts, über `Values` kann auf die benutzerdefinierten Variablen zugegriffen werden, und `Variablenname` ist der Name der selbstdefinierten Variable. Beim Rendern des Charts ersetzt Helm die Variable `Variablenname` durch ihren Wert. Die Variablenwerte können aus verschiedenen Quellen stammen, etwa aus der Standarddatei `values.yaml`, aus vom Benutzer bereitgestellten Werten, die als zusätzliche `values.yaml` an Helm-CLI übergeben werden, oder aus CLI-Parametern.

In *Listing 2* ist eine Template-Datei für ein Deployment zu sehen.

In diesem Beispiel werden die Platzhalter `{{ .Values.replicaCount }}`, `{{ .Values.appName }}` und `{{ .Values.image.repository }}:{{ .Values.image.tag }}` zur Laufzeit durch die tatsächliche Werte ersetzt. Im einfachsten Fall

```
mychart/
  |- Chart.yaml
  |- values.yaml
  |- templates/
  | |- deployment.yaml
  | |- service.yaml
  | |- ingress.yaml
```

Listing 1: Dateistruktur eines Charts

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.appName }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
     matchLabels:
           app: {{ .Values.appName }}
  template:
     metadata:
        labels:
           app: {{ .Values.appName }}
     spec:
        containers:
        - name: {{ .Values.appName }}
           image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
```

Listing 2: Template-Deployment

werden diese Werte direkt über eine `values.yaml` bereitgestellt, die wie in *Listing 3* aussehen kann.

```
appName: my-app
replicaCount: 3
image:
  repository: my.container.registry/myapp
  tag: latest
```

Listing 3: values.yaml

Zum Rendern des Charts können die Befehle `helm install` oder `helm template` verwendet werden. Der Befehl `helm install` sendet die gerenderten Manifest-Dateien direkt an den Kubernetes-API-Server. `helm template` zeigt das gerenderte Manifest nur in der Konsole an. Das generierte Manifest aus dem Beispiel oben wird in *Listing 4* gezeigt.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
     matchLabels:
        app: my-app
  template:
     metadata:
        labels:
           app: my-app
     spec:
        containers:
        - name: my-app
           image: my.container.registry/myapp:latest
```

Listing 4: Generiertes Manifest für Deployments

Wie werden in Helm Werte transformiert?

Vordefinierte Funktionen

Die vordefinierten Funktionen von Helm verarbeiten und manipulieren Variablen im Template, bevor sie in Kubernetes-Manifeste gerendert werden. Funktionen sind einfache, zweckgebundene Operationen, die eine bestimmte Aufgabe erfüllen, zum Beispiel die Formatierung einer Zeichenfolge, die Konvertierung eines Werts in einen anderen Datentyp oder die Durchführung einer mathematischen Operation. Die Funktion `quote` beispielsweise schließt eine Zeichenkette in Anführungszeichen ein. Sie können wie in *Listing 5* gezeigt verwendet werden.

```
{{ quote .Values.appName }}
# resolves to "my-app"
```

Listing 5: Beispiel einer vordefinierten Funktion quote

Pipelines

Die Pipeline-Syntax erlaubt es, mehrere Funktionen und Ausdrücke miteinander zu verketten, wodurch eine Reihe von Transformationen für eine Variable definiert werden kann. Eine Pipeline nimmt die Ausgabe einer vorangehenden Funktion oder eines Ausdruckes und leitet sie an die folgende Funktion weiter.

Das nächste Beispiel zeigt, wie man Pipelines dazu verwenden kann, den Wert einer Variable `appName` durch mehrere Funktionen zu leiten und so zu transformieren. Die Funktion `split` trennt den Inhalt des Strings `appName` in ein Array von Teilstrings, wobei das Zeichen - als Trennzeichen verwendet wird, und die Funktion `index` wählt das zweite Element dieses Arrays aus *(siehe Listing 6)*.

```
{{ .Values.appName | split "-" | index 1 }}
# resolves to app
```

Listing 6: Beispiel für die Verwendung von Pipelines

Benutzerdefinierte Funktionen

Zusätzlich zu den vordefinierten Funktionen können benutzerdefinierte Funktionen für Templates erstellt werden. Im Chart können diese Funktionen innerhalb eines Templates oder einer separaten `.tpl`-Datei definiert werden. Es ist üblich, benutzerdefinierte Funktionen in der `_helper.tpl`-Datei im `templates`-Ordner des Charts zu definieren, da so alle benutzerdefinierten Funktionen an einem einzigen Ort organisiert und im gesamten Chart wieder-

verwendet werden können. *Listing 7*zeigt ein Beispiel für eine benutzerdefinierte Funktion, die einen gegebenen String in Großbuchstaben konvertiert und ihn in Anführungszeichen einschließt.

```
{{- define "toUpperAndQuote" -}}
{{- toUpper | quote }}
{{- end }}
```

Listing 7: Beispiel für benutzerdefinierte Funktionen

Wenn das obige Beispiel in der `templates/helper.tpl`-Datei definiert ist, kann es wie folgt verwendet werden *(siehe Listing 8)*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
  - labelKey: {{ include "toUpperAndQuote" .Values.appName }}
spec:
  replicas: 3
  selector:
     matchLabels:
        app: my-app
  template:
     metadata:
        labels:
           app: my-app
     spec:
        containers:
        - name: my-app
           image: my.container.registry/myapp:latest
```

Listing 8: Verwendung einer benutzerdefinierten Funktion

Wie funktionieren Bedingungen in Helm?

Sie können `if`-Anweisungen in Helm verwenden, um bestimmte Teile eines Templates basierend auf dem Wert einer Variablen auszuführen. Solche `if`-Anweisungen ermöglichen die Aktivierung oder Deaktivierung einer Funktion oder die bedingte Erstellung von Ressourcen. Sie ermöglichen es Ihnen, flexiblere Templates zu entwerfen, die sich an verschiedene Umgebungen anpassen. Die grundlegende Syntax einer `if`-Anweisung in Helm ist in *Listing 9* zu sehen.

Die Minuszeichen `{{-` und `-}}` entfernen führende und nachfolgende Leerzeichen in der Ausgabe.

Aufbau eines Helm Charts für eine Spring-Boot-Anwendung

Um eine Anwendung in ein Helm Chart zu packen, müssen mehrere Voraussetzungen erfüllt sein. Die Anwendung, die Sie in das Chart packen möchten, muss containerisiert sein. Für das folgende Beispiel kann das öffentlich verfügbare Image `public.ecr.aws/viadee/k8s-demo-app` *[2]* verwendet werden. Außerdem wird der Zugriff auf einen Kubernetes-Cluster benötigt, in dem das Chart bereitgestellt werden kann, und die Helm-CLI muss auf dem lokalen Rechner installiert sein. Sobald die notwendigen Voraussetzungen erfüllt sind, kann ein benutzerdefiniertes Helm Chart erstellt werden. Der Prozess umfasst die folgenden Schritte:

1. Erstellen des Chart-Gerüsts
2. Bearbeitung der Metadaten des Charts
3. Bearbeitung der Standardwerte für das Chart
4. Bearbeitung der Templates
5. Das Chart verpacken und in ein Chart-Repository hochladen

1. Erstellen des Chart-Gerüsts

Das Erstellen eines neuen Helm Charts kann auf verschiedene Weise erfolgen, aber eine der effizientesten Methoden ist die Verwendung des Befehls `helm create`. Dieser Befehl generiert die grundlegende Dateistruktur und Beispieldateien für ein neues Chart, sodass es nicht mehr manuell eingerichtet werden muss. Der Befehl bietet unter anderem die Vorteile, dass bei der Generierung bereits die Best Practices für die Entwicklung von Helm Charts eingehalten werden. Dies erleichtert es, Konsistenz über mehrere Charts hinweg zu erreichen und es wird die Möglichkeit verringert, manuelle Fehler einzubauen, da man kleinteiliger arbeiten kann. Außerdem bietet der Befehl einen guten Ansatzpunkt, um zu verstehen, wie jeder Teil des Charts funktioniert. Der Befehl `helm create` erwartet den Namen des Charts als Argument: `helm create springboot-helm-chart`. Die erstellte Dateistruktur ist in *Listing 10* zu sehen.

2. Bearbeitung der Metadaten des Charts

Obwohl es nicht notwendig ist, kann es hilfreich sein, die generierte `chart.yaml` zu überarbeiten. Sie enthält die Metadaten des Charts, einschließlich seines Namens, seiner Version und seiner

```
{{- if <condition> -}}
  # template code to execute if the condition is true
{{- else -}}
  # template code to execute if the condition is false
{{- end -}}
```

Listing 9: Beispiel Bedingungen

```
springboot-helm-chart/
  |- Chart.yaml #  This file contains metadata about the chart, such as the chart name and version.
  |- values.yaml # This file contains default values for the chart's configurable parameters.
  |- charts/ # This directory contains sub-charts (dependencies) used by the chart.
  |- templates/ # This directory contains the templates to generate Kubernetes manifests.
  | |- _helpers.tpl
  | |- deployment.yaml
  | |- hpa.yaml
  | |- ingress.yaml
  | |- service.yaml
  | |- serviceaccount.yaml
  | |- NOTES.txt
  | |- tests/
   | |- test-connection.yaml
```

Listing 10: Durch helm create erzeugte Dateistruktur

```
# Chart.yaml
apiVersion: v2
name: spring boot-helm-chart
description: A Helm chart for deploying the viadee/k8s-demo-app spring boot application
type: application
version: 1.0.0
appVersion: "1.2.0"
```

Listing 11: Beispiel für eine chart.yaml-Datei

```
# values.yaml
image:
  repository: public.ecr.aws/viadee/k8s-demo-app
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: ""
```

Listing 12: Konfiguration des Container-Images in value.yaml

Beschreibung. Der Befehl `helm create` setzt das name-Feld, sodass es unwahrscheinlich ist, dass es in der `chart.yaml` geändert werden muss, aber es ist möglich. Der Name identifiziert das Chart, wenn es in ein Chart-Repository hochladen wird. In einem generierten Chart wird er auch in den Templates verwendet, um die Ressourcen zu benennen. Das Feld `description` enthält eine kurze Beschreibung des Charts. Das `version`-Feld sollte dem semantischen Versionsstandard folgen, zum Beispiel 1.10.3. Es ermöglicht außerdem die Angabe der Chart-Version für den Upload in ein Chart-Repository. Schließlich definiert das Feld `appVersion` die Version der Anwendung, die das Chart einsetzt. Dieses Feld wird in den standardmäßig generierten Templates referenziert, um die Image-Version der Deployment-Kubernetes-Ressource festzulegen *(siehe Listing 11)*.

3. Bearbeitung der Standardwerte für das Chart

Der erste Wert, der geändert werden muss, ist `image.repository` in der Datei `values.yaml`. Er gibt das für die Bereitstellung verwendete Container-Image an und muss auf das gewünschte Image aktualisiert werden. In unserem Beispiel sollte daher der Wert `image.repository` auf `public.ecr.aws/viadee/k8s-demo-app` gesetzt werden *(siehe Listing 12)*.

Da die angegebene `appVersion` auch ein gültiges Image-Tag ist, muss der Wert `image.tag` nicht gesetzt werden. Die generierte Datei `template/deployment.yaml` verwendet die Ausdrücke `{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}`, um das Container-Image für das Deployment anzugeben. Die Zeile besteht aus zwei Ausdrücken, die durch einen Doppelpunkt : getrennt sind. Der Ausdruck `{{ .Values.image.repository }}` ruft den Wert des Repository-Feldes aus dem Image-Abschnitt in der Datei `values.yaml` ab. Der Ausdruck `{{ .Values.image.tag | default .Chart.AppVersion }}` gibt den `appVersion`-Wert zurück, wenn der Wert `image.tag` nicht definiert oder leer ist.

Zusätzlich zum Wert `image.repository` muss auch der Wert `service.port` in der Datei values.yaml geändert werden. In einem generierten Chart definiert dieser Wert den Port, den der Service für das Deployment bereitstellt. Er steuert auch, wo Kubernetes Liveness- und Readiness-Probes erwartet. Standardmäßig ist der service.port in einem generierten Chart 80, während der Standardport einer Spring-Boot-Anwendung 8080 ist *(siehe Listing 13)*. Wenn die Ports nicht übereinstimmen, bleibt der Pod in einer `CrashLoopBackOff` stecken.

```
# values.yaml
service:
  port: 8080
```

Listing 13: Konfiguration des Services in values.yaml

Dies sind alles notwendige Änderungen, um Ihr eigenes Spring Boot-Container-Image auszuführen. Darüber hinaus unterstützt das Chart Funktionen wie die Anpassung der Ressourcenlimits, das Hinzufügen von Image-Pull-Secrets, die Referenz auf einen bestehenden Service-Account und die Aktivierung eines Ingress. Sie können den Befehl `helm template` verwenden, um Änderungen an Ihrem Helm Chart schnell zu testen, bevor Sie es installieren. Der Befehl zeigt an, ob Ihr Chart Syntaxfehler aufweist. Wenn die Syntax korrekt ist, können Sie prüfen, ob das erstellte Kubernetes-Manifest `yaml` Ihren Erwartungen entspricht *(siehe Listing 14)*.

```
# print the generated yaml manifest
helm template my-release .
# or save it as a file
helm template my-release . > generated-template.yaml
```

Listing 14: Beispielhafte Verwendung des helm template-Befehls

Der Befehl `template` benötigt zwei Parameter: den Namen der Veröffentlichung und den Pfad zum Chart-Verzeichnis. In unserem Beispiel ist `my-release` der Name des generierten Release, und . legt das aktuelle Verzeichnis als Chart-Verzeichnis fest. Ein Helm Release ist ein Helm Chart, das durch die Installation oder Aktualisierung des Charts mit `helm install` oder `helm upgrade` in einem Kubernetes-Cluster erstellt wurde. Ein Release hat einen eindeutigen Namen, der mit einem bestimmten Satz an Konfigu-

rationswerten verknüpft ist. In diesem Beispiel erzeugt der Befehl `helm template` das `yaml`-Manifest für einen Release namens `my-release`. Dabei werden die in der Datei `values.yaml` angegebenen Konfigurationswerte verwendet. In *Listing 15* ist ein Auszug aus dem generierten `yaml`-Manifest zu sehen.

```
...
---
# Source: springboot-helm-chart/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-springboot-helm-chart
  labels:
    helm.sh/chart: springboot-helm-chart-1.0.0
    app.kubernetes.io/name: springboot-helm-chart
    app.kubernetes.io/instance: test
    app.kubernetes.io/version: "1.2.0"
    app.kubernetes.io/managed-by: Helm
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: springboot-helm-chart
      app.kubernetes.io/instance: test
  template:
    metadata:
      labels:
        app.kubernetes.io/name: springboot-helm-chart
        app.kubernetes.io/instance: test
    spec:
      serviceAccountName: test-springboot-helm-chart
      securityContext:
        {}
      containers:
        - name: springboot-helm-chart
          securityContext:
            {}
          image: "public.ecr.aws/viadee/k8s-demo-app:1.2.0"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          resources:
            {}
---
...
```

Listing 15: Generiertes Deployment-Manifest

Sie können den Befehl `helm install` mit denselben beiden Parametern verwenden wie den `template`-Befehl, um das Chart in ein Kubernetes-Cluster zu bringen. Der Befehl erstellt die Ressourcen im Cluster und eine neue Version des Helm Release. Helm speichert in einer Historie alle Versionen eines Releases und ermöglicht ein Rollback zu einer älteren Version.

4. Bearbeitung der Templates

An diesem Punkt enthält das generierte `yaml`-Manifest des Deployments eine Liveness- und Readiness-Probes, die auf den Basispfad der Spring-Boot-Anwendung zeigt. Ein besserer Ansatz wäre es, die dedizierten Actuator-Endpunkte zu verwenden. Ab Spring Boot Version 2.3 wurden die Klassen `LivenessStateHealthIndicator` und `ReadinessStateHealthIndicator` hinzugefügt, um den aktuellen Liveness- und Readiness-Status der Anwendung anzuzeigen. Spring Boot registriert diese Aktuatoren automatisch beim Deployment in Kubernetes. Die Endpunkte `/actuator/health/liveness` und `/actuator/health/readiness` dienen als Liveness- und Readiness-Probes *(siehe Listing 16)*.

```
# ./templates/deployment.yaml
...
                livenessProbe:
                   httpGet:
                      path: /actuator/health/liveness
                      port: http
                readinessProbe:
                   httpGet:
                      path: /actuator/health/readiness
                      port: http
...
```

Listing 16: Auszug aus genertiertem Deployment-Manifest mit Probes

Da das Chart nun den Start der Anwendung in Kubernetes ermöglicht, besteht der nächste Schritt darin, die Möglichkeit zur Konfiguration der Anwendung hinzuzufügen. Spring-Boot-Anwendungen verwalten die Konfiguration in `application.properties`- oder `application.yaml`-Dateien. Diese Dateien enthalten Schlüssel-Wert-Paare, die verschiedene Konfigurationen wie zum Beispiel die URL der Datenbankverbindung, den Server-Port und andere Einstellungen definieren. In der Regel muss die Konfiguration je nach Umgebung, in der die Anwendung ausgeführt wird, geändert werden. So wird sich beispielsweise die URL der Datenbankverbindung für Entwicklungs-, Staging- und Produktionsumgebungen unterscheiden. Beim Deployment einer Anwendung in mehreren Umgebungen ist es wünschenswert, denselben Build zu verwenden und dennoch die Möglichkeit zu haben, zwischen verschiedenen Konfigurationssätzen zu wechseln. Ohne diesen Ansatz müsste der Anwendungscode jedes Mal neu erstellt werden, wenn eine Konfigurationsänderung erforderlich ist. Glücklicherweise verfügt Spring Boot über eine Funktion namens `externalized configuration`, die das Festlegen von Konfigurationen über Umgebungsvariablen unterstützt. Durch die Externalisierung der Konfiguration ist es möglich, dasselbe Artefakt für verschiedene Umgebungen zu verwenden und dennoch zwischen verschiedenen Sätzen von Konfigurationswerten zu wechseln.

Durch die Möglichkeit eine Spring-Boot-Anwendung extern zu konfigurieren, kann sie über native Kubernetes-Mechanismen angesteuert werden. In Kubernetes können `ConfigMaps` oder Secrets verwendet werden, um Konfigurationen zu speichern. `ConfigMaps` speichern Datenpaare, die nicht sensibel sind, wie etwa Anwendungskonfigurationen, Feature-Flags und Datenbankverbindungsstrings. Secrets werden zum Speichern sensibler Daten wie Kennwörter und Tokens verwendet. Pods können auf die gespeicherte Konfiguration in `ConfigMaps` oder Secrets zugreifen, indem sie diese als Umgebungsvariablen laden oder diese als Datei mounten. Um diese Funktionalität in das eigene Chart zu integrieren, muss ein neues Template für die `ConfigMap` angelegt

```
# templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "springboot-helm-chart.fullname" . }}
  labels:
     {{- include "springboot-helm-chart.labels" . | nindent 4 }}
data:
     key: value
```

Listing 17: Template für ConfigMap

werden und das bestehende Template für das Deployment erweitert werden.

Zunächst wird eine neue Datei für das `ConfigMap`-Template im Verzeichnis `./templates` erstellt und der Name der `ConfigMap` durch denselben Ausdruck ersetzt, den die anderen Ressourcen zum Festlegen ihres Namens verwendet haben *(siehe Listing 17)*.

Nun muss die `ConfigMap` als Umgebungsvariablen in das Deployment geladen werden. Dazu wird `envFrom` verwendet *(siehe Listing 18)*.

Um die Konfiguration über das Chart steuern zu können, muss ein neuer Wert in der `values.yaml` hinzugefügt werden und dieser im `ConfigMap`-Template referenziert werden. In diesem Beispiel werden die Properties `welcome.text` und `nav.bgcolor` hinzugefügt, die das sichtbare Verhalten der `public.ecr.aws/viadee/k8s-demo-app` ändern.

Es gibt jedoch ein Problem: Betriebssysteme erzwingen strenge Namenskonventionen für Umgebungsvariablen. Linux-Shell-Variablen können beispielsweise nur Buchstaben (Groß- oder Kleinbuchstaben), Zahlen (0 bis 9) oder Unterstriche enthalten. Leider widersprechen diese Regeln dem Benennungssystem von Spring. Glücklicherweise berücksichtigt die flexible Property-Bindung von Spring Boot die Einschränkungen so weit wie möglich. Um sicherzustellen, dass die Properties erkannt werden, sollten Sie diese nach den folgenden Regeln umbenennen:

1. Punkte durch Unterstriche ersetzen
2. Bindestriche entfernen
3. Alle Buchstaben zu Großbuchstaben umwandeln

Der Ausdruck `{{- toYaml .Values.properties | nindent 4 }}` kann dazu verwendet werden, um `yaml`-Blöcke zu übernehmen. Die Funktion `toYaml` wird direkt von Helm bereitgestellt und wandelt eine Variable zu einer `yaml`-Zeichenkette um. Die Funktion `nindent` rückt die Ausgabe um eine bestimmte Anzahl von Leerzeichen ein. Das Bindestrich-Symbol - vor der Funktion `toYaml` steht für *No Whitespace*, das heißt, es werden alle führenden Leerzeichen in der Ausgabe eliminiert. Es reicht nicht aus, den Ausdruck `{{ .Values.properties }}` zu verwenden, um auf den Wert zuzugreifen, da er einen komplexen Datentyp zurückgibt, den Helm in der generierten `yaml` als `map[NAV_BGCOLOR:lightblue WELCOME_TEXT:Hello world.]` ausgibt *(siehe Listing 19, Listing 20)*.

Zu diesem Zeitpunkt würde die Spring-Boot-Anwendung nicht bemerken, wenn Sie den Inhalt des `properties`-Feldes ändern, da die `ConfigMap` als Umgebungsvariable injiziert wird und nur neu geladen wird, wenn der Pod neu gestartet wird. Um einen Pod-Neustart zu erzwingen, muss die Deployment-Spezifikation selbst

```
# ./templates/deployment.yaml
...
        containers:
           - name: {{ .Chart.Name }}
             securityContext:
                {{- toYaml .Values.securityContext | nindent 12 }}
             image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
             imagePullPolicy: {{ .Values.image.pullPolicy }}
             ports:
                - name: http
                  containerPort: {{ .Values.service.port }}
                  protocol: TCP
             livenessProbe:
                httpGet:
                  path: /actuator/health/liveness
                  port: http
             readinessProbe:
                httpGet:
                  path: /actuator/health/readiness
                  port: http
             resources:
                {{- toYaml .Values.resources | nindent 12 }}
             envFrom:
                - configMapRef:
                      name: {{ include "springboot-helm-chart.fullname" . }}
...
```

Listing 18: Integration der ConfigMap in das Deployment-Template

geändert werden. Sie können dies erreichen, indem Sie der Spezifikation des Pod-Templates eine Annotation hinzufügen, die sich ändert, wenn sich die Konfiguration ändert. Dann wird Kubernetes die Änderungen melden und einen Neustart erzwingen *(siehe Listing 21)*.

Der Ausdruck `{{ .Values.properties | quote | sha256sum }}` nimmt den Wert des `property`-Feldes, setzt ihn in Anführungszeichen und errechnet seinen SHA-256-Hash. Das Ergebnis des Ausdrucks ist der eindeutige Bezeichner für das Property-Feld, der zur Erkennung von Änderungen an der Konfiguration verwendet wird.

5. Das Chart verpacken und in ein Chart-Repository hochladen

Das von der Community gepflegte Chart-Repository heißt Artifact Hub *[3]*, es kann aber auch ein eigenes Repository betreiben werden. Einige andere beliebte Chart-Repositories sind:

- Chart Museum: Ein einfaches Open-Source-Chart-Repository, das on-premises oder in der Cloud betrieben werden kann.
- Harbor: Ein Open-Source-Chart-Repository mit fortgeschrittenen Features wie rollenbasiertem Zugriff und Image-Signierung.
- GitHub Pages: Bei dieser Methode wird ein Git-Repository mit einem gh-page Branch erstellt und die Charts in diesem Branch gespeichert.

Um das Chart zu einem Repository hinzuzufügen, muss es zunächst gepackt werden. Mit dem Befehl `helm package` wird eine `.tgz`-Datei erstellt, die alle erforderlichen Dateien und Metadaten für das Chart enthält. Sobald das Chart gepackt ist, kann es mit verschiedenen Methoden in ein Chart-Repository übertragen werden. Bei einigen beliebten Chart-Repositories, wie beispielsweise Harbor, können Sie Charts über eine Weboberfläche hochladen, während Sie bei anderen, wie zum Beispiel Chart Museum, ein CLI-Tool verwenden müssen. Ein Beispiel für die Verwendung von GitHub-Pages zum Hosten eines Repositories findet sich hier *[4]*.

Um ein Helm Chart aus einem Chart-Repository zu installieren, können Sie den Befehl `helm install` verwenden, aber anstatt auf das lokale Chart zu verweisen, verweisen Sie auf das Repository *(siehe Listing 22)*.

Fazit

Helm ist ein mächtiges Tool für die Verwaltung von Anwendungen in einem Kubernetes-Cluster. Wenn Sie in einer Umgebung arbei-

```
# values.yaml
properties:
  WELCOME_TEXT: Hello world.
  NAV_BGCOLOR: lightblue
```

Listing 19: Spring Properties in values.yaml

```
# templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "springboot-helm-chart.fullname" . }}
  labels:
     {{- include "springboot-helm-chart.labels" . | nindent 4 }}
data:
     {{- toYaml .Values.properties | nindent 4 }}
```

Listing 20: Integration Spring Properties aus values.yaml in ConfigMap-Template

```
# ./templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "springboot-helm-chart.fullname" . }}
  labels:
     {{- include "springboot-helm-chart.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{- end }}
  selector:
     matchLabels:
        {{- include "springboot-helm-chart.selectorLabels" . | nindent 6 }}
  template:
     metadata:
        # the annotation "checksum/config" is generated with each invocation of the template
        annotations:
           checksum/config: {{ .Values.properties | quote | sha256sum }}
        {{- with .Values.podAnnotations }}
           {{- toYaml . | nindent 8 }}
        {{- end }}
        labels:
           {{- include "springboot-helm-chart.selectorLabels" . | nindent 8 }}
...
```

Listing 21: Deployment-Template, das bei Property-Änderungen einen neuen Pod startet

```
# install from the local chart
helm install my-release .

# install from the chart repository
## Add repository
helm repo add viadee https://viadee.github.io/spring-boot-helm-chart
## Deploy a Helm Release named "viadee-release" using the spring-boot-helm-chart chart
helm install viadee-release viadee/spring-boot-helm-chart
```

Listing 22: Befehle, um mit Chart-Repositories zu interagieren

ten, in der Sie eine Vielzahl von Anwendungen bereitstellen und verwalten müssen, kann Helm dazu beitragen, diesen Prozess zu optimieren und den erforderlichen Zeit- und Arbeitsaufwand zu reduzieren. Obwohl Helm den Deployment-Prozess vereinfachen kann, kann es dabei dennoch die Komplexität Ihrer Umgebung erhöhen, da es eine weitere Abstraktionsebene einführt. Wenn Ihre Anwendung leicht auszurollen ist und wenig Konfiguration bedarf, ist der Einsatz von Helm möglicherweise nicht notwendig. Letztendlich müssen Sie bei der Entscheidung bedenken, welche Use-Cases Sie abdecken wollen. Selbst für kleine Anwendungen kann Helm Ihren Deployment-Prozess leichter nachvollziehbar machen. Wenn Sie hingegen bereits viel automatisiert haben, ist es vielleicht besser, Ihre Anwendungen direkt auf Ihrem Cluster zu deployen, ohne Helm einzusetzen, oder eine schlankere Lösung wie *Kustomize* zu verwenden.

Quellen

[1] *helm.sh*
[2] *https://github.com/viadee/k8s-demo-app*
[3] *https://artifacthub.io/*
[4] *https://github.com/viadee/spring-boot-helm-chart/blob/main/.github/workflows/publish.yml*

Jan Weyrich
viadee Unternehmensberatung AG
Jan.weyrich@viadee.de

Jan Weyrich arbeitet als Architekt und Entwickler bei viadee, wo er Kunden in allen Fragen der Planung und der Entwicklung von maßgeschneiderten Software-Lösungen berät. Er nutzt sein tiefes Verständnis von Entwicklungsprozessen zur Lösung von Herausforderungen, denen Unternehmen in diesem Bereich begegnen. Sein aktueller Schwerpunkt liegt auf der Konzeption von sicheren und skalierbaren Lösungen im Cloud-Umfeld.

Wie Cloud Native und AI Unternehmen transformieren

Björn Schotte, Mayflower GmbH

In den vergangenen Jahren haben sich Unternehmen eher auf isolierte Denkmodelle gestützt, die die Berücksichtigung moderner Technologien und Architekturen bei der Team-Gestaltung außen vorließen. Entweder wurde nur auf Team-Strukturen oder nur auf Technologien geschaut. Dadurch schöpfen die Unternehmen nicht das volle Potenzial aus, das Cloud Native und AI bieten. Doch das muss nicht sein. Dieser Beitrag zeigt auf, was sich alles in Unternehmen, vor allem aber hinsichtlich der Team-Strukturen und der einzelnen Rollen ändern muss, damit die volle Kraft entfaltet werden kann.

Typische Unternehmensstrukturen

Agile Softwareentwicklung und ihre Rahmenwerke wie Scrum oder skalierte Modelle wie LeSS sprechen von crossfunktionalen Teams. End2end-Verantwortung ist nicht zuletzt auch durch die DevOps-Bewegung ein wichtiger Bestandteil. Das bedeutet, dass ein Team die volle Verantwortung dafür hat, ein (Teil-)Produkt nicht nur zu entwickeln, sondern auch zu deployen und zu betreiben.

Doch in vielen Unternehmen sieht dies anders aus. Da muss für den Release erst ein „DevOps-Team" beauftragt werden. Die Anforderungen kommen aus den Fachabteilungen, werden dort „zu Ende gedacht" und schließlich zur „agilen Abarbeitung" an das IT-Team übergeben.

Wie sieht es in den meisten größeren Unternehmen aus? Wird dort die Cloud genutzt? Ja, schließlich wird die Software jetzt in der Cloud „gehostet". Aber selbst dort deployen dürfen Teams oftmals nicht. Und gleichzeitig ist das oberste Management-Level der Meinung, man sei erfolgreich in der Cloud und hätte agil arbeitende Teams – Entschuldigung, „Squads" –, die nun die Aufgabe haben, die Time-2Market erheblich zu verkürzen.

Cloud-Native-Prinzipien

Du merkst schon, die Situation im vorangegangenen Abschnitt ist nicht gerade die Beste. Doch fangen wir zunächst mal von vorne an: Wer auf Cloud Native umstellen möchte und sich davon eine Leistungssteigerung und höhere Wettbewerbsfähigkeit verspricht, der kommt nicht darum herum, sich mit den Cloud-Native-Prinzipien zu beschäftigen. Diese sind:

- Skalierbarkeit
- Resilienz
- Managebarkeit
- Beobachtbarkeit
- Automatisierung

Eine Cloud-Native-Anwendung ist leicht skalierbar und reagiert durch automatisches Up-/Down-Scaling auf die aktuelle Traffic- und Anforderungssituation. Dabei ist sie resilient. Fallen also einzelne Teil-Komponenten aus, so ist die Anwendung dennoch weiterhin benutzbar. Zudem sollte sie und die dazugehörige Infrastruktur gut managebar sein. Der Status jeder einzelnen Komponente ist zum Beispiel durch eine UI und/oder durch ein API sichtbar. Damit die Teams jederzeit wissen, was in ihrer Anwendung, die auf eine größere Zahl an Services und Nodes verteilt ist, passiert, muss diese beobachtbar sein. Daher ist zu jeder Zeit feststellbar, was „innendrin" passiert. Automatisierung ist das allumfassende Prinzip, das auch eine Haltung beschreibt: Möglichst alles ist automatisiert und steht über „X-as-Code" im Code-Repository versioniert zur Verfügung. Es ist keine leichte Aufgabe, all diese Prinzipien zu beherzigen. Im Folgenden sehen wir, warum es so wichtig ist, das eigene Unternehmen so anzupassen, dass diese Prinzipien umgesetzt werden können.

Auswirkungen auf die Unternehmskultur

Mit dem Umsetzen der aufgezeigten Prinzipien geht ein Kultur- und Haltungswechsel einher. In den Unternehmen wird vermeintlich nach agilen Prinzipien gearbeitet. Der Begriff der „Feature Factory" wird leider immer noch zu häufig in den Mund genommen, wenn von agilem Arbeiten die Rede ist. Zu viele Abhängigkeiten zwischen den Teams führen dazu, dass von kurzer Time2Market nur geträumt werden kann.

Auf dem Weg zu einer Cloud-Native-Kultur hat der Begriff der Feature Factory keinen Bestand mehr. Denn hier ist eine gute Form der Zusammenarbeit im Team notwendig. Schließlich sind die Teams in Cloud-Native-Unternehmen hochgradig dezentral unterwegs.

Auswirkungen auf Teams/Prozesse

Dezentralität ist eine Notwendigkeit, um das volle Potenzial von Cloud Native zu entfalten. Ende-zu-Ende-Verantwortung treibt die Team-Strukturen in die Dezentralität. Im agilen Kontext ist Cross-Funktionalität zwar ein Bestandteil des Arbeitens, dieser wird aber häufig nicht gut umgesetzt. Anforderungen kommen immer noch aus Fachabteilungen, Product Owner sind oftmals nur die User-Story-Schreiber. Wo man hinsieht, gibt es Abhängigkeiten. Diese werden durch den Einsatz von skalierten Frameworks wie SAFe weiter zementiert, da diese Frameworks keine Reduktion von Abhängigkeiten vorsehen.

Möchte ein Unternehmen also von Cloud Native profitieren, kommt es um ein System, das Dezentralität fördert, nicht herum. Entstehen die Anforderungen in Fachabteilungen, so wäre es wichtig, die entsprechenden Menschen direkt in das Entwicklungsteam zu transferieren, sodass eine echte Einheit als Team entstehen kann. Durch die Dezentralität entsteht auch eine Eigenständigkeit im Team, durch die es ermöglicht wird, eine eigene Vorgehensweise zu wählen. Aufgrund der Schnelligkeit, die echte Cloud-Native-Nutzung ermöglicht, werden die Teams eher in Richtung von IT-Kanban oder ähnlichen Prozess-Rahmenwerken streben, die die Grundhaltung von täglicher Veränderungsmöglichkeit mit sich bringen.

Conway's Law als Gedankenmodell ist hierbei führend. Organisations- und Team-Strukturen sind also zwingend in Abhängigkeit von den genutzten Technologien zu betrachten. Eine Cloud-Native-Umgebung für ein Produkt mit Aufteilung in diverse Services hat also die beschriebenen Auswirkungen auf die Team-Struktur und damit auf die Organisationsstruktur zur Folge.

Ist hierfür kein Bewusstsein in der Organisation vorhanden, so kann das volle Potenzial von Cloud Native nicht genutzt werden. Es verkümmert also zu einer Illusion in der Organisation. Die wahre Wettbewerbsfähigkeit gestaltet nun die Konkurrenz, der es gelingt, sich anders aufzustellen.

Auswirkungen auf Product Owner und Scrum Master

Mit einer veränderten Organisations- und Team-Aufstellung sowie einer viel stärkeren Ausrichtung auf Technologie verändern sich natürlich auch die Anforderungen an agile Rolleninhaber wie Product Owner und Scrum Master. In Cloud-Native-Umgebungen beschleunigt sich alles spürbar. Denn jetzt ermöglicht uns die Technologie – Cloud-Infrastruktur, -Architektur und Patterns wie Microservices – eine deutlich dezentralere Art und Weise, Software-Systeme zu realisieren.

Wenn ich also als Product Owner nicht weiß, was diese neuen Paradigmen bedeuten, und welches Potenzial diese Technologien er-

möglichen, dann kann ich mein Produkt nicht auf die Nutzung dieser Potenziale ausrichten. Hier ist dringend angeraten, das eigene technologische Fachwissen zu vertiefen, um gemeinsam mit dem Team das Beste aus dem Produkt herauszuholen. Scrum Master, die zu einseitig auf den Faktor Mensch innerhalb des Teams blicken, sollten gegebenenfalls auch ihr technologisches Fachwissen vertiefen. Einerseits war es für Scrum Master schon immer hilfreich zu wissen, was im Pair-/Mob-Programming passiert, wozu ein Event-Storming hilfreich ist und wieso Softwareentwickler viel häufiger Code analysieren als schreiben. Andererseits geht es jetzt in einer Cloud-Native-Organisation nicht mehr nur darum, mit voller Kraft für ein einzelnes Team da zu sein. Es ist eher anzunehmen, dass ein Scrum Master ein Stück weit außerhalb des Teams steht und in der Organisation mitwirkt. Denn trotz starker dezentraler Aufstellung kann er helfen, einen guten Kommunikationsfluss zwischen den Teams herzustellen – dort, wo es benötigt wird. Gleichzeitig sollte er das eigene Team fordern. Doch dies gelingt mit einer stärkeren technologischen Aufstellung im Team nur noch, wenn der Scrum Master selbst über ein gewisses Maß an technologischem Verständnis verfügt.

Anforderungen an das (Top-)Management

Veränderungen in der Organisationsstruktur selbst brauchen nicht nur Zeit, sondern auch ein entsprechendes Buy-In seitens des Top-Managements. Damit dies gelingen kann, ist es hilfreich, die Gesamtzusammenhänge in der Sprachwelt des Managements zu vermitteln. Das Optimieren auf DORA-Metriken (DevOps Research and Assessments) („Unsere Entwickler müssen schneller werden!") ist das eine – alle am Arbeitsfluss beteiligten Bereiche zu identifizieren, das andere.

Hier ist es hilfreich, „from idea to delivery" als eine Art Kanban-Fluss zu visualisieren, mit entsprechenden Wartezeiten zwischen den einzelnen Prozess-Schritten. Ebenso zeigt es, wie viel schneller ein Team arbeiten könnte, wenn es echte End2end-Verantwortung hätte.

Gibt es hier eine Einsicht, so können mögliche nächste Schritte, Vor-/Nachteile sowie Voraussetzungen besprochen werden. Dabei dürfen auch etwaige Technologie-Modernisierungen im Rahmen der Cloud-Native-Migration nicht vergessen werden. Externe Unterstützung kann hier Licht ins Dunkel bringen und mit anpacken.

Generative AI als Beschleuniger

Generative AI setzt bei der Beschleunigung des Wandels nochmal eins obendrauf. Hier haben wir es mit einem echten *FOMO*-Moment (Fear of missing out) zu tun. FOMO bedeutet hier, dass fieberhaft nach Möglichkeiten gesucht wird, die eigene Wettbewerbsfähigkeit durch die Augmentierung mit AI wie generativer AI zu stärken und zu beschleunigen.

Doch dies kann nur dann gelingen, wenn eine Organisation technologisch wie auch strukturseitig hierfür gut aufgestellt ist. Sind alle Hausaufgaben in Bezug auf Cloud Native und die daraus abgeleiteten Folgen für Teams und Prozesse gemacht, so kann auch der Einsatz von generativer AI mit Leichtigkeit geschehen – denn entsprechende Haltungen für ultraschnelle Delivery sind nicht nur Haltungen, sondern gleichsam realistisch.

Großartige Produkte (er)schaffen – Wie soll man es nun angehen? (Fazit)

Ein Fazit zu treffen ist nicht ganz einfach. Mit ehrlichem Blick führt kein Weg daran vorbei, deutlich mehr Dezentralität in der Organisation zu entwickeln, wenn man von den Vorteilen von Cloud Native, inklusive ultraschnellen Releases, profitieren möchte.

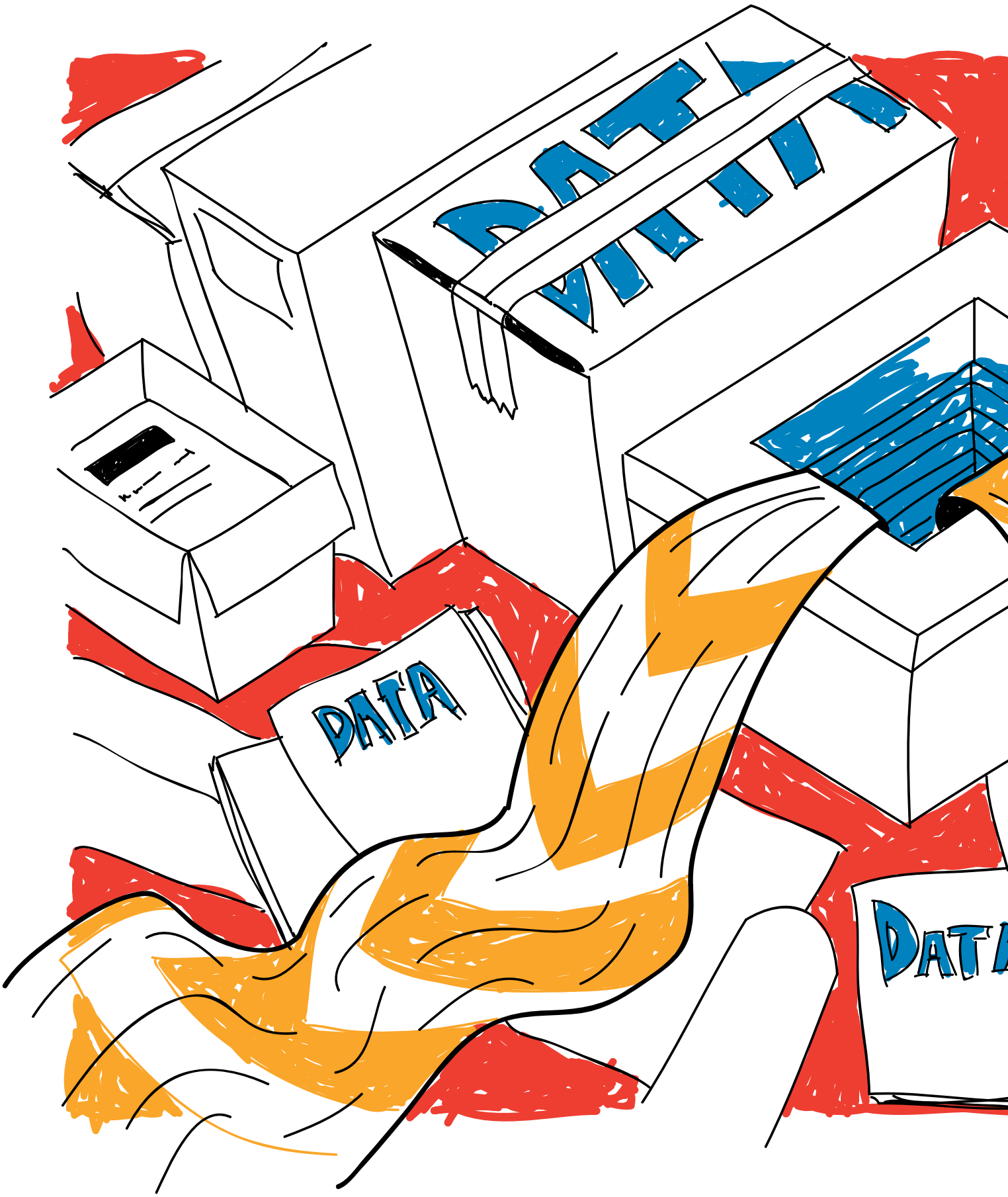
Wir erleben den Wandel oftmals im Kontext von Software-Modernisierungen, an denen wir zusammen mit unseren Kunden die notwendigen Schritte in Richtung Cloud Native gehen. Die Cloud-Native-Prinzipien und das dazugehörige Reifegrad-Modell helfen uns, dem Kunden zu vermitteln, welche technologischen und organisatorischen Veränderungen notwendig sind. Sie sind also gleichsam ein guter Navigator auf der Reise zu stärkerer Wettbewerbsfähigkeit und damit besseren Produkten.

Björn Schotte
Mayflower GmbH
https://www.linkedin.com/in/bjoernschotte/

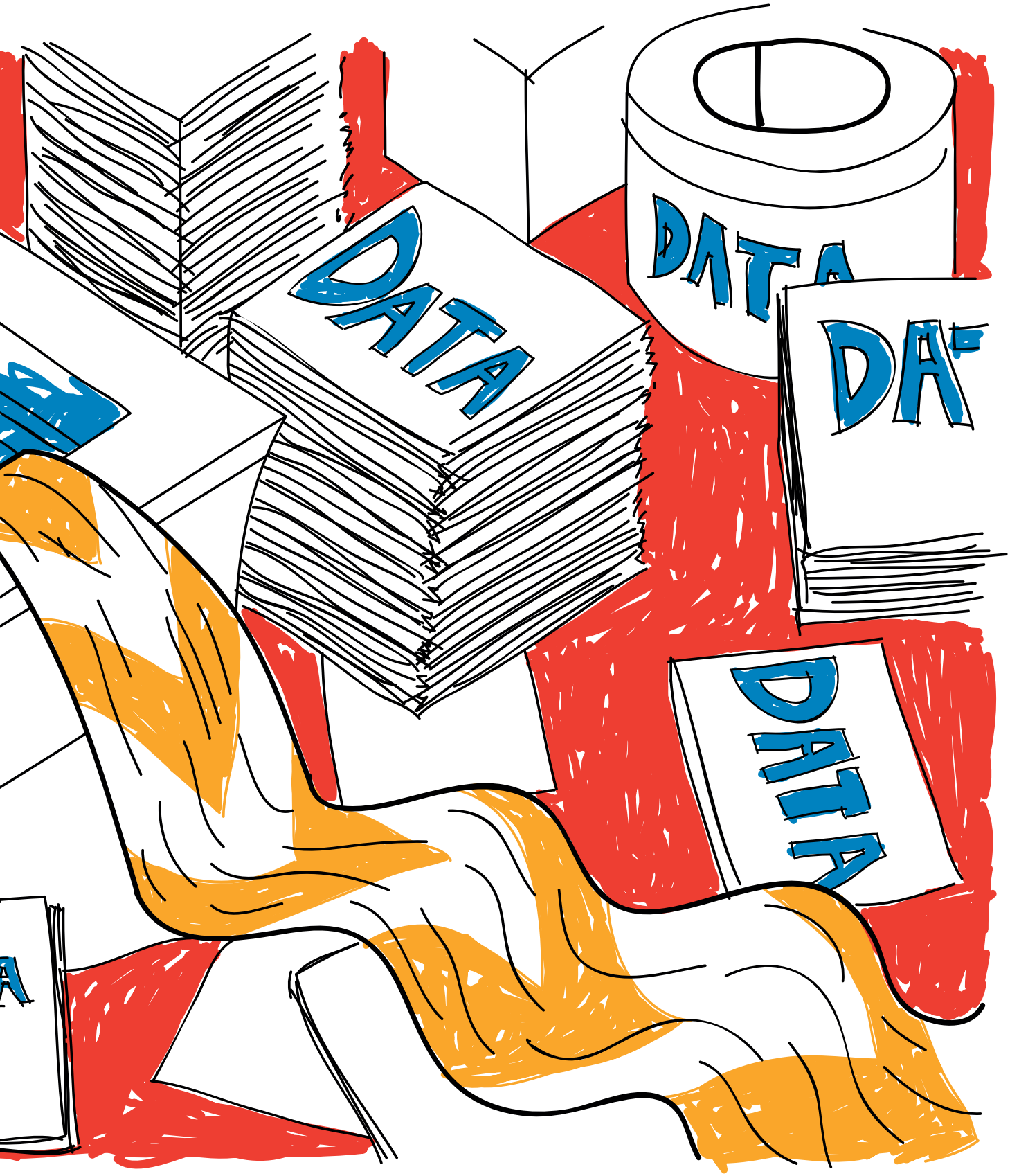
Björn Schotte ist Mitgründer und Geschäftsführer der Mayflower GmbH. Er berät Kunden zu digitaler Transformation und AI. Die mehr als 100 Crewmitglieder der Mayflower machen ihre Kunden zu digitalen Vorreitern – durch Individual-Softwareentwicklung, die Nutzer begeistert.

JavaFinder – Inventur mal anders

Gerrit Grunwald, Azul



Wie viele Java-Distributionen sind eigentlich auf meinem Rechner installiert? Diese Frage kam auf, als ich auf meinem Rechner eine beträchtliche Anzahl von OpenJDK-Distributionen in nur einem Ordner gefunden habe. Da es aber müßig ist, sämtliche Ordner und Unterordner nach Distributionen zu durchsuchen, habe ich kurzerhand ein Tool dafür geschrieben, das in diesem Artikel kurz erläutert werden soll.



Ich arbeite seit mehr als 20 Jahren auf Mac-Geräten und habe dort in dem Ordner `/Volumes/Macintosh HD/Library/Java/JavaVirtualMachines` eine größere Anzahl von OpenJDK-Distributionen gefunden. Es ist ganz schön, dass diese Distributionen sich alle in diesem Ordner wiederfinden, das ist jedoch nicht zwangsläufig immer so. Denn jeder kann seine OpenJDK-Distribution dort installieren, wo er gerne möchte. Zum Beispiel installiert SDKman seine OpenJDK-Distributionen standardmäßig in dem Ordner `$HOME/.sdkman/candidates/java` auf Linux- oder MacOS-Systemen. Mittlerweile kann man auch direkt in IntelliJ IDEA und NetBeans eine OpenJDK-Distribution herunterladen und in einem Ordner seiner Wahl installieren. Im Prinzip ist das alles gut, allerdings verliert man auch schnell den Überblick. Zudem gibt es jede Menge Applikationen, die mit einer eigenen OpenJDK-Distribution ausgeliefert werden (zum Beispiel IntelliJ IDEA).

Vor einiger Zeit habe ich ein Tool *[1]* programmiert, das mir dabei hilft, alle meine installierten OpenJDK-Distributionen auf dem aktuellen Stand zu halten. Dieses Tool prüft für alle Distributionen, die es in einem angegebenen Ordner findet, ob es Updates für diese Distributionen gibt und falls dies zutrifft, lässt es mich die neueste Version herunterladen. JDKMon ist ein Tool, das im System Tray „sitzt" und alle drei Stunden nach Updates sucht, weshalb es nicht dafür geeignet, bestimmte Ordner auf Anfrage zu durchsuchen. Dies führte zu der Idee, ein Kommandozeilen-Tool zu haben, das ich auf einen bestimmten Ordner anwenden kann und das mir dann alle OpenJDK-Distributionen auf der Kommandozeile ausgibt, die es in diesem Ordner inklusive Unterordnern findet.

JavaFinder

Bei dem JavaFinder handelt es sich um ein Tool, das eine gegebene Ordnerstruktur nach OpenJDK- und auch GraalVM-Distributionen durchsucht. Manche fragen sich, wozu brauche ich so ein Tool denn eigentlich?

Da gibt es verschiedene Gründe. In größeren Betrieben gibt es oft das Problem, dass die IT-Abteilung herausfinden muss, welche Java-Installationen sich auf den verschiedenen Geräten im Netzwerk befinden, um eine Übersicht zu haben. Die ist wichtig, wenn es um Lizenzgebühren geht. Hat man beispielsweise Oracle JDK auf seinem Rechner installiert und hat der Lizenzvereinbarung zugestimmt, so kann Oracle (seit Januar 2023) Lizenzgebühren für alle Mitarbeiter der Unternehmung erheben! Da dies zu beträchtlichen Kosten führen kann, macht es Sinn, einen Überblick über die installierten Distributionen zu haben. Für die Erstellung solch eines „Software-Inventars" gibt es professionelle Lösungen, die JavaFinder selbstverständlich nicht ersetzen kann. Es ist jedoch ein praktisches Hilfsmittel sein, wenn man schnell überprüfen möchte, welche Distributionen sich in einer angegebenen Ordnerstruktur befinden.

Das Tool ist komplett in Java geschrieben und Open Source. Den Sourcecode sowie die Binaries sind auf GitHub *[2]* verfügbar. Dabei wird ein ausführbares JAR-File erzeugt. Man kann dieses JAR-File direkt ausführen oder ein Native Image mit der Hilfe von GraalVM erstellen. Unter den Releases im GitHub-Repository finden sich native Binaries für Windows (x64), Linux (x64, aarch64) und Mac (x64, aarch64). Der Vorteil von nativen Binaries ist der, dass man nur noch eine Executable hat, die sich dann auch einfach auf der Kommandozeile ausführen lässt und keine weiteren Dateien mehr benötigt, um ausgeführt zu werden. JavaFinder erkennt die folgenden Distributionen:

- AdoptOpenJDK
- AdoptOpenJDK J9
- Bi Sheng
- Corretto
- Debian
- Dragonwell
- Glucon GraalVM
- GraalVM
- JetBrains
- Kona
- Liberica
- Liberica NIK
- Mandrel
- Microsoft
- OJDK Build
- Open Logic
- Oracle
- Oracle OpenJDK
- RedHat


```
{
  "timestamp":1685697020,
  "search_path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/",
  "sysinfo":{
     "operating_system":"Mac OS",
     "architecture":"ARM64",
     "bit":"64 Bit"
  },
  "distributions":[
     {
        "vendor":"Azul",
        "name":"Zulu",
        "version":"21-ea+22",
        "timestamp":1685697020,
        "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-21.jdk/zulu-21.jdk/Contents/Home/",
        "build_scope":"OpenJDK",
        "in_use":false,
        "used_by":[
        ]
     },
```

```
    {
       "vendor":"Gluon",
       "name":"Gluon GraalVM",
       "version":"22.1.0.1",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/gluon-graalvm/Contents/Home/",
       "build_scope":"GraalVM",
       "in_use":false,
       "used_by":[
       ]
    },
    {
       "vendor":"Azul",
       "name":"Zulu",
       "version":"11.0.19",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-11.jdk/zulu-11.jdk/Contents/Home/",
       "build_scope":"OpenJDK",
       "in_use":false,
       "used_by":[
       ]
    },
    {
       "vendor":"Azul",
       "name":"Zulu",
       "version":"17.0.7",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-17.jdk/zulu-17.jdk/Contents/Home/",
       "build_scope":"OpenJDK",
       "in_use":true,
       "used_by":[
       ]
    },
    {
       "vendor":"Azul",
       "name":"Zulu",
       "version":"20.0.1",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-20.jdk/zulu-20.jdk/Contents/Home/",
       "build_scope":"OpenJDK",
       "in_use":false,
       "used_by":[
       ]
    },
    {
       "vendor":"Azul",
       "name":"Zulu",
       "version":"8.0.372+7",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-8.jdk/zulu-8.jdk/Contents/Home/jre/",
       "build_scope":"OpenJDK",
       "in_use":false,
       "used_by":[
       ]
    },
    {
       "vendor":"Azul",
       "name":"Zulu",
       "version":"8.0.372+7",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-8.jdk/zulu-8.jdk/Contents/Home/",
       "build_scope":"OpenJDK",
       "in_use":false,
       "used_by":[
       ]
    },
    {
       "vendor":"Oracle",
       "name":"Graal VM CE",
       "version":"22.3.1",
       "timestamp":1685697020,
       "path":"/System/Volumes/Data/Library/Java/JavaVirtualMachines/graalvm-ce-java17-22.3.1/Contents/Home/",
       "build_scope":"GraalVM",
       "in_use":false,
       "used_by":[
       ]
    }
  ]
}
```


Listing 1: Beispiel einer Ausgabe bei Verwendung von javafinder json

```
Vendor,Distribution,Version,Timestamp,Path,Type,InUse,Timestamp
Azul,Zulu,20.0.1,2023-06-02T09:14:03.323994Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-20.jdk/zulu-
20.jdk/Contents/Home/,OpenJDK,false,1685697243
Azul,Zulu,17.0.7,2023-06-02T09:14:03.318197Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-17.jdk/zulu-
17.jdk/Contents/Home/,OpenJDK,true,1685697243
Azul,Zulu,11.0.19,2023-06-02T09:14:03.311116Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-11.jdk/
zulu-11.jdk/Contents/Home/,OpenJDK,false,1685697243
Azul,Zulu,8.0.372+7,2023-06-02T09:14:03.341194Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-8.jdk/
zulu-8.jdk/Contents/Home/,OpenJDK,false,1685697243
Oracle,Graal VM CE,22.3.1,2023-06-02T09:14:03.315379Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/graalvm-
ce-java17-22.3.1/Contents/Home/,GraalVM,false,1685697243
Gluon,Gluon GraalVM,22.1.0.1,2023-06-02T09:14:03.309816Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/
gluon-graalvm/Contents/Home/,GraalVM,false,1685697243
Azul,Zulu,21-ea+22,2023-06-02T09:14:03.329001Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-21.jdk/
zulu-21.jdk/Contents/Home/,OpenJDK,false,1685697243
Azul,Zulu,8.0.372+7,2023-06-02T09:14:03.336303Z,/System/Volumes/Data/Library/Java/JavaVirtualMachines/zulu-8.jdk/
zulu-8.jdk/Contents/Home/jre/,OpenJDK,false,1685697243
```

Listing 2: Beispiel einer Ausgabe bei Verwendung von javafinder csv

- SAP Machine
- Semeru
- Temurin
- Trava
- Ubuntu
- Zulu
- Zulu Prime

Das Tool ist dabei sehr einfach gehalten, sodass man nur angibt, in welchem Format man die gefunden Distributionen erhalten möchte (CSV, JSON), sowie den Pfad, in dem gesucht werden soll. Gibt man kein Format an, so wird unformatiertes JSON verwenden. Gibt man JSON als gewünschtes Format an, wird formatiertes JSON ausgegeben und gibt man CSV an, so erhält man weniger Informationen im CSV-Format. Durch die Möglichkeit im JSON-Format Hierarchien anzulegen, ist es möglich, zusätzliche Informationen auszugeben, wie zum Beispiel Informationen über das verwendete Betriebssystem, die CPU-Architektur und die Bitbreite. Des Weiteren wird auch versucht, anzugeben, ob die gefundene Distribution derzeit in Verwendung ist und falls ja, durch welche Aufrufe. Die formatierte JSON-Ausgabe sieht man in *Listing 1*. Gibt man anstelle von JSON hier CSV an, dann sähe die Ausgabe so aus wie in *Listing 2*.

Gibt man keinen expliziten Pfad an, in dem JavaFinder suchen soll, werden Standardpfade verwendet, die für jedes Betriebssystem wie folgt vorgegeben sind:

- `C:\Program Files\Java\` (Windows)
- `/usr/lib/jvm` (Linux)
- `/System/Volumes/Data/Library/Java/JavaVirtualMachines/` (MacOS)

Das Tool erhebt keinen Anspruch darauf, perfekt zu sein. Falls es zu Problemen kommt oder Distributionen, obwohl vorhanden, nicht gefunden werden, so möchte ich darum bitten, in dem GitHub-Repository eine Issue anzulegen und das Problem ausreichend zu beschreiben.

Referenzen

[1] JDKMon Repository: *https://github.com/HanSolo/JDKMon*

[2] JavaFinder Repository: *https://github.com/HanSolo/javafinder*

Gerrit Grunwald

Azul

ggrunwald@azul.com

Gerrit Grunwald ist ein Software-Ingenieur, der sich schon seit 40 Jahren für das Programmieren begeistert. Er ist ein echter Anhänger von Open Source und hat sowohl an populären Projekten wie JFXtras.org als auch an seinen eigenen Projekten (TilesFX, Medusa, Enzo, SteelSeries Swing, SteelSeries Canvas, JDKMon) mitgewirkt.

Gerrit ist ein aktives Mitglied der Java-Community, wo er die Java User Group Münster (Deutschland) gegründet hat und leitet, er ist ein JavaOne Rockstar und Java Champion. Zudem spricht er auf internationalen Konferenzen und User Groups und schreibt für verschiedene Magazine.

APEX connect

by DOAG

22.04. - 24.04.2024

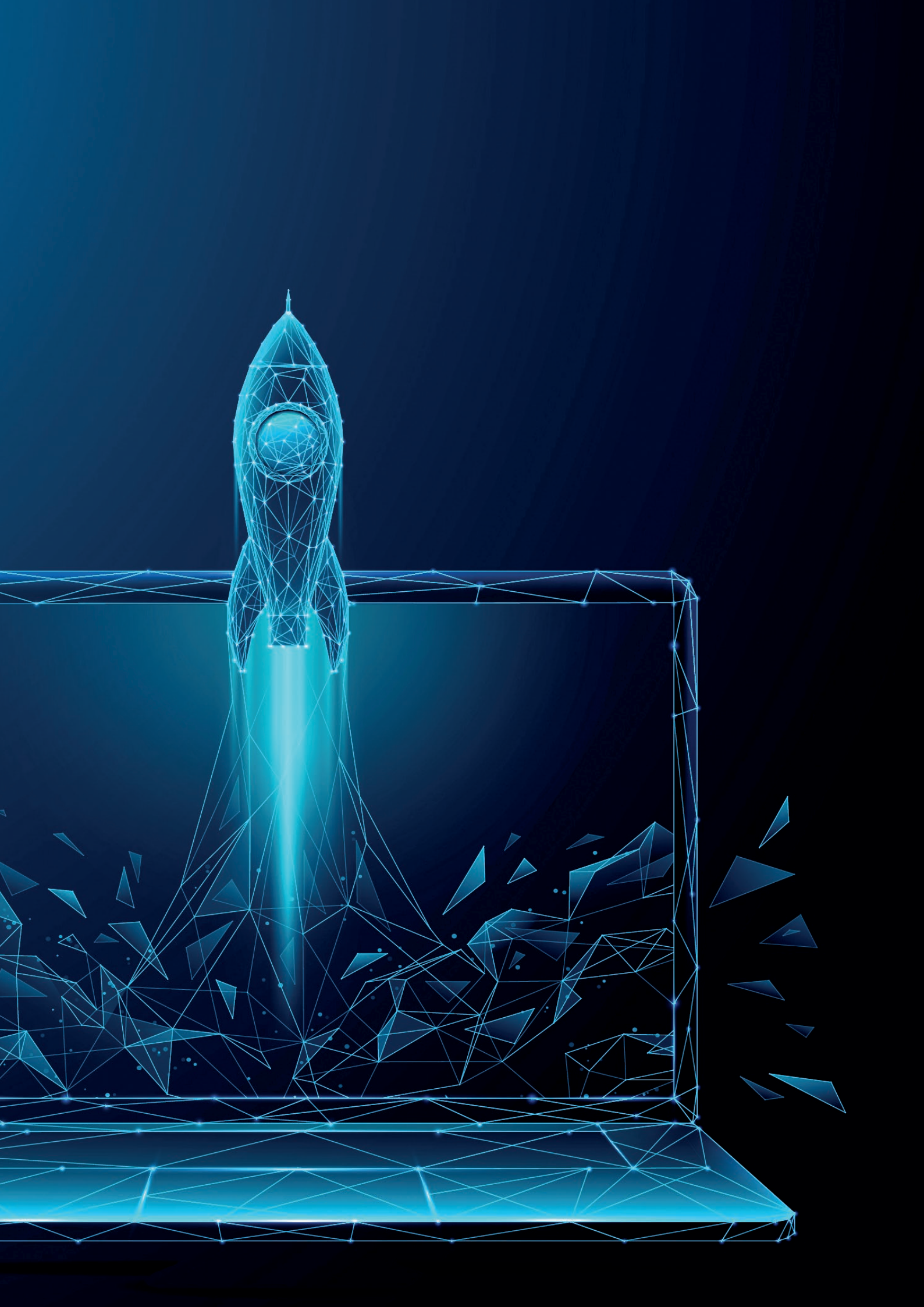
VAN DER VALK AIRPORTHOTEL DÜSSELDORF

apex.doag.org

Let's get started: Der Eclipse Starter für Jakarta EE

Alexander Rühl, valantic Software & Technology Innovations GmbH

Lange Zeit gab es keinen Starter für Jakarta EE, also eine Website zur einfachen Erstellung neuer Projekte. Andere Frameworks, wie beispielsweise das hauseigene MicroProfile oder der Konkurrent SpringBoot, hatten dies schon länger im Angebot. Ende letzten Jahres kam dann der Jakarta Starter – aber mit noch viel Luft nach oben, weshalb der Autor sich entschloss, daran aktiv mitzuarbeiten. Mittlerweile in seiner zweiten Version, braucht sich der Starter nicht mehr zu verstecken und hilft so bei einem einfachen Einstieg in Jakarta EE. Dieser Artikel stellt den Starter vor und berichtet aus dem zugehörigen Eclipse-Projekt.



Im Dezember 2022 nahm ich an der JakartaOne *[1]* teil, einer Online-Konferenz mit den Größen der Jakarta-Welt. Ein Talk machte mich besonders neugierig – „Create, build and run Jakarta EE applications in under a minute" *[2]*. Erfahrungsgemäß ist man nämlich länger damit beschäftigt, selbst, wenn es sich nur um ein Hello World handelt. Im Talk erfuhr ich dann, dass es nun einen Starter für Jakarta EE gab, also eine Website, um sich ein Projekt zusammenklicken zu können, was die Vorfreude weiter in die Höhe trieb, denn in all den Jahren von Enterprise-Java fehlte so etwas. Damals, noch seitens Oracle, überließ man es den Herstellern, dergleichen zu tun und zog sich auf die eigene Position zurück, nur spezifizierende Instanz zu sein. Wenn man ein neues Projekt erstellen wollte, nahm man in der Regel einen veröffentlichten Maven-Archetype oder ein anderes Projekt als Vorlage und passte es entsprechend an. Doch das war in meinen Augen schon immer ein Fehler, da auf diese Weise die Einstiegshürde, insbesondere für Neulinge, zu hoch war – man musste sich die Information erst suchen, wie man ein Projekt erstellt, baut und in einem Application Server zum Laufen bekommt.

Doch zurück zum Vortrag: Leider wich die Vorfreude recht schnell der Ernüchterung, denn was man auf der Starter-Website *[3]* bekam, war lediglich eine Hilfe, wie das Maven-Kommando für die Erzeugung eines Projektes, basierend auf einem gewählten Archetype und den eingegebenen Maven-Koordinaten, auszusehen hatte. Dieses musste man manuell kopieren und es dann auf der Kommandozeile ausführen *(siehe Abbildung 1)*. Das war zwar ein Anfang, aber noch weit von einem echten Starter entfernt, wie man es zum Beispiel von MicroProfile *[4]*, Quarkus *[5]* oder eben dem Konkurrent SpringBoot *[6]* kennt, bei dem man verschiedene Optionen wählen und sich dann ein generiertes (Maven)-Projekt downloaden kann.

In dem Vortrag wurde aber auch darum geworben, sich an der Entwicklung des Starters zu beteiligen und ich habe dann im Anschluss genau das getan. Denn was sollte man machen, statt sich über fehlende Funktionalität in einem Open-Source-Projekt zu beschweren? Na, mitmachen eben!

Wenn man das bei Jakarta EE mitwirken möchte, braucht man einen Account auf *[16]*, muss dort das Eclipse User Agreement *[4]* unterzeichnen und sich dann unter *[17]* der Working Group anschließen. Unter *[18]* findet man den Source Code des Starter-Projektes und die Issues, bei denen Hilfe benötigt wird. Wenn man ein solches Issue bearbeiten möchte, lässt man sich dafür eintragen, forkt das Repository und stellt im Anschluss an die Implementierung einen Pull-Request. Wer möchte, kann dann noch an monatlich stattfindenden Zoom-Calls teilnehmen, um über das weitere Vorgehen im Projekt mitzuentscheiden und natürlich in der Mailing-Liste mitdiskutieren.

Meine Anfänge im Projekt waren ein nicht funktionierender Unittest unter Windows (interessanterweise war das noch niemandem aufgefallen, scheinbar benutzen alle anderen Mac oder Linux), UI-Texte und später dann die Implementierung weiterer Features. Auf meine Anregung hin wurde auch der Starter auf der Startseite von Jakarta EE *[8]* prominent verlinkt *(siehe Abbildung 2)*.

Das Projekt insgesamt hat inzwischen einiges an Fahrt aufgenommen *(siehe Abbildung 3)*. Obwohl schon 2020 gestartet, ging es erst ab zirka Oktober 2022 richtig los und Ende März 2023 war es dann

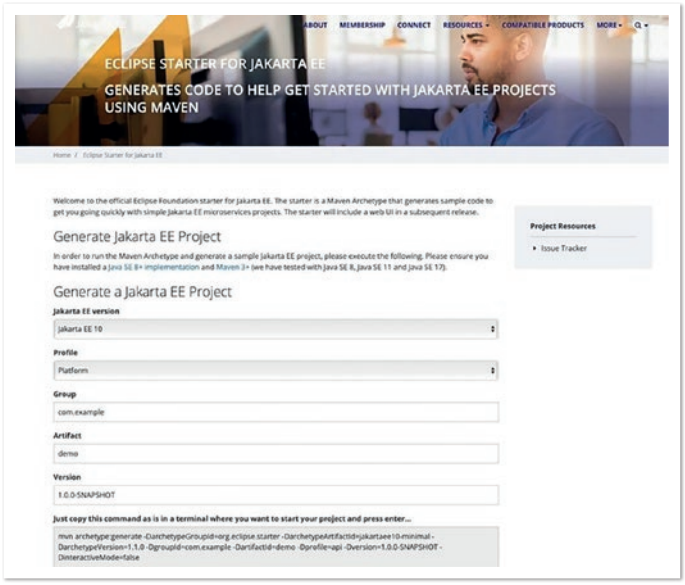


Abbildung 1: Der Jakarta-Starter von Dezember 2022 (© Vortrag von A N M Bazlur Rahman [2], JakartaOne/Starter-Website [3], Eclipse Foundation)

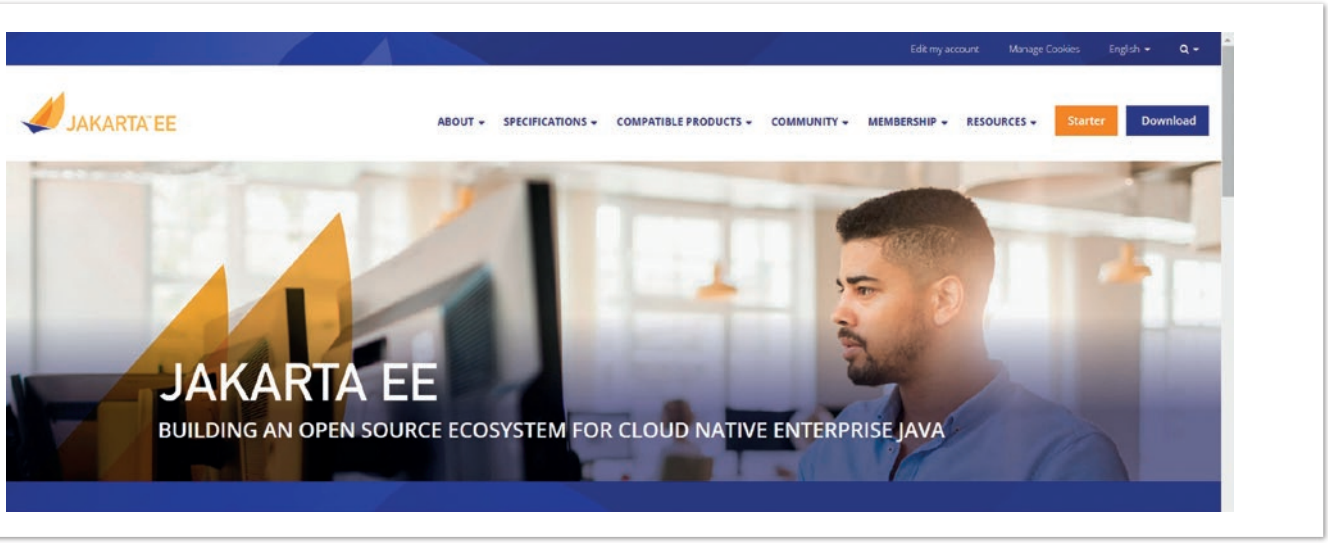


Abbildung 2: Die Jakarta-Startseite mit dem neuen prominenten Starter-Link (© Jakarta-EE-Website [8], Eclipse Foundation)

so weit – die Version 2.0 des Starters wurde veröffentlicht und mit ihr ein neues UI dafür *(siehe Abbildung 4)*.

Noch immer wird unter der Haube ein Maven-Archetype verwendet, doch nun wird aus den ausgewählten Daten ein Maven-Projekt zum Download generiert. Neben der zu Beginn schon vorhandenen Auswahl der Jakarta-EE-Version (10, 9.1, 9, 8) und dem verwendeten Profil (Platform, Web, Core) gibt es nun auch die Auswahl der zugrundeliegenden Java-Version (17, 11, 8). Aber noch entscheidender ist die Auswahl einer für die Entwicklung konfigurierten Runtime (GlassFish, Open Liberty, Payara, TomEE, WildFly) sowie die Möglichkeit des Docker-Supports.

Der Starter enthält neben der Maven-Projektstruktur noch den berühmten Hello-World-REST-Service sowie eine Jakarta-Faces-Seite mit weiterführenden Informationen *(siehe Abbildung 5)*. In naher Zukunft wird es hier mehr Beispielcode geben, den man in der UI wählen kann, um weitere APIs aus Jakarta EE exemplarisch in Benutzung sehen zu können, zum Beispiel einen CRUD-Service mit Datenbank-Integration.

Per Default ist keine Runtime ausgewählt, ein Aspekt, der der Neutralität von Jakarta EE gegenüber Implementierern ihrer Spezifikation geschuldet ist. (Ein Fun-Fact am Rande: Aus diesem Grund wird die Reihenfolge der Runtimes zufällig erzeugt und ist mit jedem Refresh der Starter-Seite anders.) Es empfiehlt sich aber in jedem Fall, eine Runtime auszuwählen, unabhängig davon, wie das Projekt später betrieben wird, da man dann direkt eine mittels Maven ausführbare Applikation erhält, ohne zuvor einen Application-Server installieren und konfigurieren zu müssen. Die Ausführung erfolgt über ein Runtime-spezifisches Plug-in, auf dessen Benutzung im generierten README hingewiesen wird. Insbesondere zum Development-Zeitpunkt ist das sehr nützlich, da die Plug-ins einen Dev-Mode anbieten, bei dem Änderungen im Code immer direkt deployt werden, was einen angenehmen Development-Roundtrip bietet.

Softwaretechnisch ist das Projekt in zwei Teile gegliedert:

- **archetype**: Der Maven-Archetype, der die zu generierende Projektstruktur in Template-Form enthält. Dieser findet sich auch auf Maven-Central *[10]* wieder und kann separat vom Starter per Kommandozeile oder aus der IDE heraus zur Generierung eines Projektes genutzt werden. Gleichzeitig bildet er das Backend für die UI und generiert das zum Download angebotene Projekt. Der Archetyp verwendet Velocity *[11]* für das Templating und Groovy *[12]* für das Post-Processing der generierten Artefakte. Das Zip wird schließlich mit dem Java-eigenen API generiert und von einem Temp-Verzeichnis aus zum Download angeboten.
- **UI**: Die Jakarta-Faces-Applikation zur Abfrage der Optionen für den Maven-Archetype. Während der Auswahl werden bereits dynamisch per Ajax die Abhängigkeiten zu den anderen Auswahlmöglichkeiten berücksichtigt, sodass immer nur gültige Kombinationen erzeugt werden können. Beim Klicken auf den Generieren-Button erfolgen noch weitere Eingabevalidierungen und dann das Erzeugen und Bereitstellen des Projekts als Zip-File zum Download. Dabei wird das Maven-Kommando embedded im UI-Code ausgeführt *(siehe Listing 1)*. Außerdem gibt es noch ein Caching der erzeugten Projekte, um häufige Kombinationen nicht jedes Mal erneut generieren zu müssen, sondern direkt zum Download anbieten zu können.

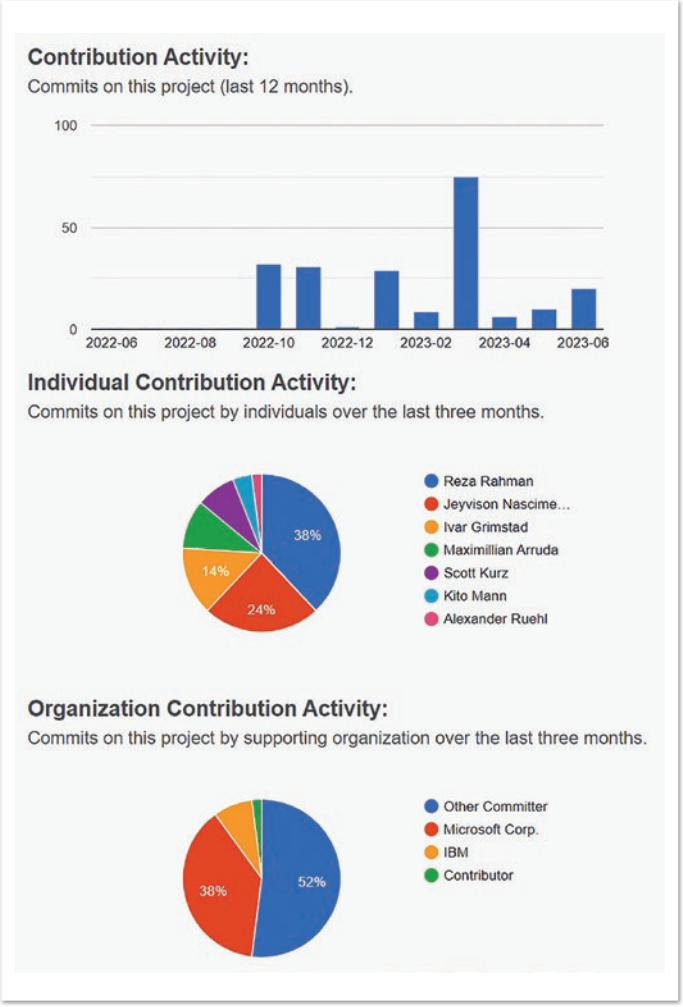


Abbildung 3: Die Projektaktivität im Starter-Projekt und die Kontributoren (© Jakarta-EE-Starter-Projektseite [9], Eclipse Foundation)

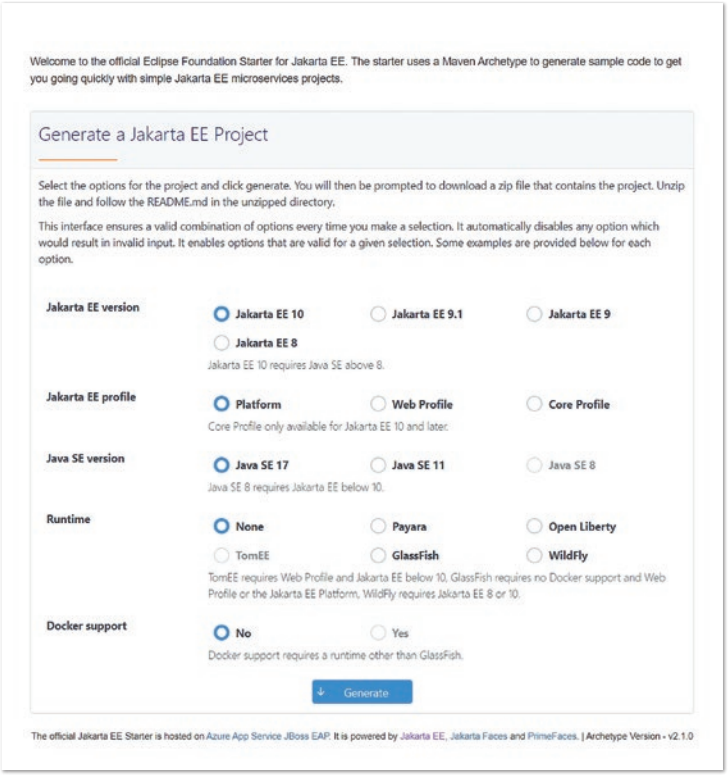


Abbildung 4: Die neue Starter-UI (© Jakarta-EE-Starterseite [3], Eclipse Foundation)

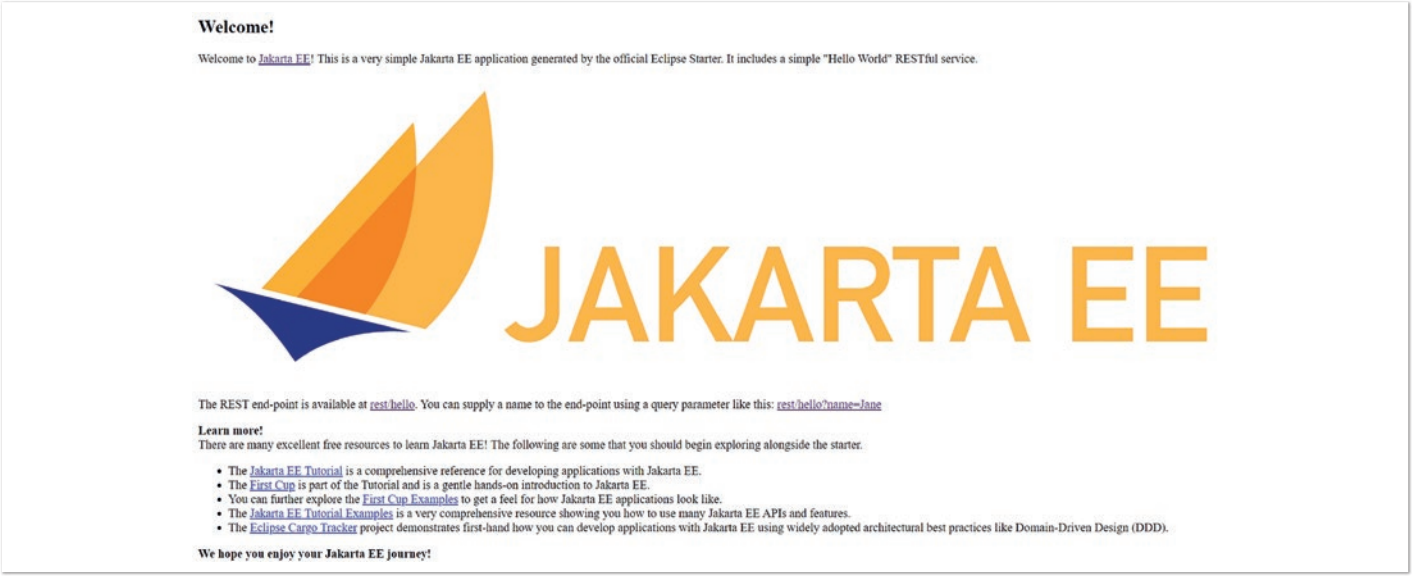


Abbildung 5: Die erzeugte Jakarta-Hello-World-Applikation mit REST-Endpunkt und Jakarta-Faces-Seite sowie einigen weiterführenden Informationen (© Ergebnis der Jakarta-Starter-Projekterzeugung [3], Eclipse Foundation)

Seit die Version 2.0 des Starters produktiv zur Verfügung steht, ist für unsere Working Group dessen Nutzung von großem Interesse, um zu sehen, wo die Schwerpunkte der Nutzer liegen und zu prüfen, ob Defaults richtig gesetzt sind. Die Starter-Applikation läuft in Azure und setzt Log-Aggregatoren in Verbindung mit Azure Monitor *[13]* ein, um Metriken zu visualisieren und damit die Nutzung des Starters zu analysieren.

Eine simple Metrik ist zum Beispiel die Anzahl der Requests auf die Generierungs-Seite *(siehe Abbildung 6)*. Man erkennt hier zum einen

```
import org.apache.maven.cli.MavenCli;
...
// get coordinates from archetype to use, a list of additional
// properties for the generation and a directory for output
public static void invokeMavenArchetype(
  String archetypeGroupId,
  String archetypeArtifactId,
  String archetypeVersion,
  Properties properties,
  File workingDirectory) {

  // set the output directory
  System.setProperty(MavenCli.MULTIMODULE_PROJECT_DIRECTORY,
     workingDirectory.getAbsolutePath());

  List<String> options = new LinkedList<>();
  // specify all options for archetype usage
  options.addAll(Arrays.asList(
     "archetype:generate",
     "-DinteractiveMode=false",
     "-DaskForDefaultPropertyValues=false",
     "-DarchetypeGroupId=" + archetypeGroupId,
     "-DarchetypeArtifactId=" + archetypeArtifactId,
     "-DarchetypeVersion=" + archetypeVersion));

  // add additional properties, such as groupId or artifactId
  properties.forEach((k, v) ->
     options.add("-D" + k + "=" + v));

  // call embedded Maven and generate project from archetype
  int result = new MavenCli().doMain(options.toArray(
     new String[0]), workingDirectory.getAbsolutePath(),
     System.out, System.err);

  // care for problems in generation
  if (result != 0) {
     throw new RuntimeException("...");
  }
}
```

Listing 1: Nutzung von Embedded-Maven zur Generierung des Projekts im Backend-Code: Optionen für Maven werden in einer Liste bereitgestellt und dann wird das Maven-Kommando aufgerufen, um den Archetype in gegebenem Verzeichnis zu prozessieren und damit das Projekt zu erzeugen.

die durchschnittlichen Requests pro Zeiteinheit und zum anderen das Auftreten von Peaks und kann das beispielsweise mit Ankündigungen im Netz in Verbindung bringen. Eine andere informative Metrik ist der prozentuale Anteil der jeweiligen Optionen, die bei der Generierung gewählt wurden *(siehe Abbildung 7)*. Hier lassen sich einige interessante Punkte herauslesen:

- **Jakarta-EE-Version**: Die neueste Version 10 ist klar vorne, aber es werden immer noch ältere Versionen verwendet, was bei der Nutzung eines Generators für neue Projekte darauf hindeutet, dass nicht immer ein Jakarta-EE-10-kompatibler Application Server dafür zum Einsatz kommen kann.
- **Jakarta-EE-Profil**: Zum größten Teil wird die Plattform genutzt, sprich das volle Profil an Jakarta-APIs. Das ist im Kontext von Jakarta EE auch völlig sinnvoll, da man ein reduziertes Profil nur dann benötigt, wenn man eine entsprechend reduzierte Runtime einsetzen möchte. Ansonsten hat das Projekt ohnehin nur eine einzige Dependency zum Jakarta-EE-API und man ist frei, alle Teile davon zu nutzen. Das Web-Profil hat auch einen signifikanten Anteil und ist wohl für die meisten modernen Jakarta-EE-Webapplikationen ausreichend. Das Core-Profil ist vor allem für Microservices-Umgebungen interessant, wobei dann in der Regel direkt mit einer entsprechenden Runtime gestartet wird, beispielsweise Quarkus *[14]*. (Ein weiterer Fun-Fact aus der Arbeit der Working Group: Es wurde ausführlich und vehement um die Begrifflichkeit „platform" für das volle Profil diskutiert, da dies der offizielle Begriff ist und nicht das häufig verwendete „full profile".)
- **Java-SE-Version**: Hier dominiert ganz klar die aktuelle LTS-Version 17. Für neue Projekte sollte man auch in jedem Fall eine solche Java-Version wählen – die Auswahl zeigt aber, dass bei Nutzern auch ältere Versionen noch von Interesse sind.
- **Docker**: Es verwundert etwas, dass relativ wenig Docker-Unterstützung gewählt wird. Selbst wenn man später die Applikation nicht in Docker betreibt, ist das für die Entwicklung durchaus

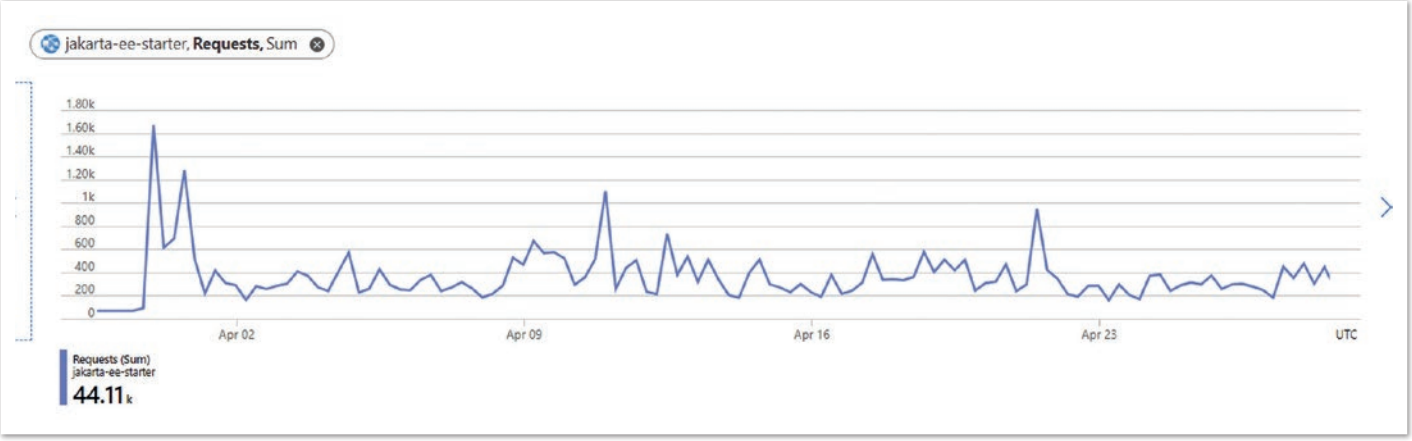


Abbildung 6: Die Anzahl der Requests auf der Jakarta-EE-Starter-Seite über einen Monat (© Azure-Monitoring [13], Microsoft)

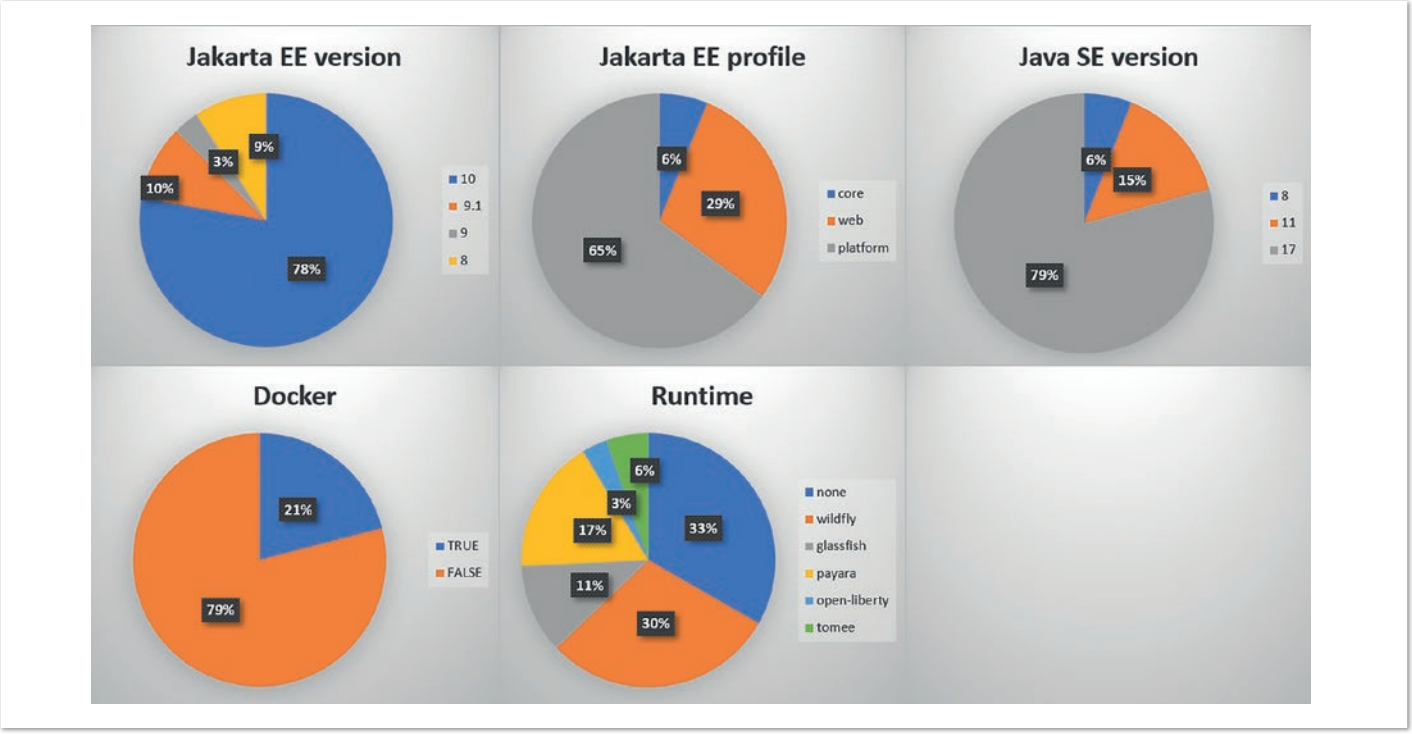


Abbildung 7: Die prozentualen Anteile der gewählten Optionen bei der Nutzung des Jakarta-EE-Starters (© Azure Monitoring [13], Excel 365, Microsoft)

nützlich und führt lediglich zur Erzeugung eines Docker-Files im Projekt, das bei Bedarf genutzt werden kann.

- **Runtime**: Hier verteilt sich die Auswahl auf die unterschiedlichen Runtimes mit leichten Vorteilen für WildFly. Dies zeigt, dass die Diversität unter den zur Verfügung stehenden Laufzeitumgebungen genutzt wird, was wiederum gut für Jakarta EE und dessen Idee einer offenen Spezifikation ist. Etwas verwunderlich ist vielleicht, dass relativ viele keine Runtime wählen. Dies kann mit dem Default zusammenhängen, aber vielleicht auch mit schon existierenden Umgebungen für das generierte Artefakt. Wie oben aber schon beschrieben, ist die Auswahl einer Runtime grundsätzlich zu empfehlen, zumal lediglich eine Maven-Plug-in-Dependency im pom-File entsteht und gegebenenfalls noch ein Runtime-spezifisches Konfigurations-File.

Bleibt zum Schluss noch ein Ausblick für das Starter-Projekt, was neben der großflächigen Bekanntmachung noch umgesetzt werden soll: Zum einen müssen wir uns weiter die Nutzung anschauen, um das Projekt dahingehend zu optimieren. Zum anderen gab es schon eine Reihe von Wünschen, was der Starter noch bieten sollte, die es abzuwägen gilt. Beispielsweise wurde gefragt, ob man auch Gradle als Build-Tool anbieten könnte oder speziell das MicroProfile *[15]* integrieren könnte. Was in jedem Fall kommen wird, ist die Möglichkeit, mehr Beispielcode hinzufügen zu können. Und schließlich wird das Projekt kontinuierlich neuen Java- und Jakarta-EE-Versionen, sowie neuen Runtimes oder geänderten Abhängigkeiten zu den möglichen Optionen Rechnung tragen.

Es gibt also noch viel zu tun, daher wiederhole ich den Aufruf, der mich seinerzeit durch die JakartaOne zu diesem Projekt geführt hat: Helft uns dabei, anderen den Einstieg in Jakarta-EE-Projekte einfach zu ermöglichen!

Ich freue mich über Anmerkungen und Fragen!
Mein Dank gilt Stephan Rauh für die Artikel-Review.

Quellen

[1] *https://jakartaone.org/2022/*
[2] *https://www.youtube.com/watch?v=IuOuY4vk0cU*
[3] *https://start.jakarta.ee/*
[4] *https://start.microprofile.io/*
[5] *https://code.quarkus.io/*
[6] *https://start.spring.io/*
[7] *https://accounts.eclipse.org/user/378338/eca/3.1.0*
[8] *https://jakarta.ee/*
[9] *https://projects.eclipse.org/projects/ee4j.starter/who*
[10] *https://search.maven.org/*
[11] *https://velocity.apache.org/*
[12] *https://groovy-lang.org/*
[13] *https://azure.microsoft.com/de-de/products/monitor*
[14] *https://quarkus.io/*
[15] *https://microprofile.io/*
[16] *https://accounts.eclipse.org/*
[17] *https://projects.eclipse.org/projects/ee4j.starter*
[18] *https://github.com/eclipse-ee4j/starter*

Alexander Rühl
valantic Software & Technology Innovations GmbH
alexander.ruehl@sti.valantic.com

Alexander Rühl ist langjähriger Mitarbeiter der valantic sti (vormals SyroCon AG) und dort als Competence Lead für alles rund um das Thema Java zuständig: in aktiver Java-Entwicklung, Projektberatung, Architekturerstellung, Sales-Unterstützung und im Rahmen der Weiterentwicklung der Mitarbeiter beziehungsweise dem Finden neuer Mitarbeiter. Alexander beschäftigt sich fast seit Anbeginn schon mit Java und nutzt seit über 20 Jahren professionell Java Enterprise. Dazu gehört auch das Verfolgen aller Neuerungen der Java-/Jakarta-EE-Versionen und dem dazugehörenden Ökosystem aus Frameworks und Tools. Außerdem ist er Committer im Jakarta-EE-Starter-Projekt.

KI Navigator 2023

Konferenz zur Praxis der KI
in IT, Wirtschaft und Gesellschaft

22. + 23. November in Nürnberg

KI verstehen und einsetzen!

“Im Jahr 2023 erleben wir eine historische Zäsur. KI-Modelle mit beispiellosem Potenzial zur Problemlösung und Entscheidungsfindung markieren einen Wendepunkt. Auf der KI Navigator kann ich von führenden Experten lernen und mich von realen Anwendungsfällen inspirieren lassen, um mich optimal auf diese neue Ära vorzubereiten.“

Oliver Szymanski
Chief Technical Architect,
IT-Systemhaus der
Bundesagentur für Arbeit

ki-navigator.doag.org

Veranstalter: DOAG Heise Medien de|ge|pol

One to rule them all! DRY Distributed Message Schema für Java Services

Ben Bajorat, Mayflower GmbH

DRY beziehungsweise „Don't repeat yourself" ist ein Prinzip der Softwareentwicklung und bezieht sich ganz allgemein auf die Vermeidung von redundantem Code oder Duplizierung in einer Codebasis und die damit verbundene Inkonsistenz und Fehleranfälligkeit. DRY lässt sich sowohl in der klassischen objektorientierten Programmierung als auch bei Microservices-Architekturen nutzen. Die Idee hinter DRY ist, Code so zu schreiben, dass die Wiederverwendung von Code maximiert, Duplikation minimiert und die Wartbarkeit verbessert werden.

Was bedeutet DRY im Kontext von Java?

Code-Wiederverwendbarkeit: Anstatt den gleichen Code an mehreren Stellen zu duplizieren, werden wiederverwendbare Komponenten erstellt (Methoden, Klassen, Module), die von verschiedenen Teilen der Anwendung aus aufgerufen werden können. Auf diese Weise können Code-Änderungen und -Aktualisierungen an einer einzigen Stelle (SSOT) vorgenommen und in anderen Instanzen verfügbar gemacht werden.

Modularer Aufbau: Der Code ist in kleinen, kohärenten Modulen organisiert, die bestimmte Aufgaben haben. Jedes Modul sollte sich dabei mit einem bestimmten Problem als Teilmenge befassen, um intuitiver und leichter wartbar zu sein.

Abstraktion: Um gemeinsame Verhaltensweisen oder Funktionalitäten zu realisieren, sollen Abstraktionen und Interfaces verwendet werden. Nur die wesentlichen Merkmale eines Objekts werden nach außen hin sichtbar gemacht.

Redundanten Code extrahieren: Sich wiederholende Codeabschnitte werden identifiziert und in eine separate Klasse, Methode, Funktion oder in ein separates Modul extrahiert. Die Logik wird somit zentralisiert, kann mehrfach verwendet werden und zum Prinzip der Abstraktion beitragen.

Mit dem DRY-Prinzip lässt sich Codequalität verbessern, die Risiko und Fehleranfälligkeit (zum Beispiel durch Copy-Paste) verringern und die Codebasis wartbarer, lesbarer und skalierbarer gestalten. Hinsichtlich dessen lässt sich DRY dem Prinzip von Clean Code zuordnen *[1][2]*. Im Hinblick auf Java ermöglicht dies, flexiblen und generischen Code zu schreiben und diesen effektiv und in wiederverwendbare Komponenten zu organisieren.

Wie das DRY-Prinzip praktisch funktioniert, wo es theoretisch einzuordnen ist und wie damit konkrete Probleme gelöst werden können, soll nachfolgend beschrieben und an einem Beispiel demonstriert werden.

Wo lässt sich das DRY-Prinzip bei den Prinzipien der OOP einordnen?

Wie SOLID ist DRY? DRY steht weder im Gegensatz zu SOLID, noch findet es sich dort namentlich genannt wieder, es kann als Ergänzung zu SOLID angesehen werden und sollte problemlos parallel betrieben werden können. Die SOLID-Teil-Prinzipien Single-Responsibility (SRP) und das Open-Closed-Principle (OCP) können komplementär zu DRY betrachtet und umgesetzt werden *[4]*.

Hierbei besagt das Prinzip der Single-Responsibility (SRP), dass eine Klasse eine einzige Verantwortung oder einen einzigen Zweck hat und nur ein bestimmtes Verhalten beziehungsweise eine bestimmte Funktionalität kapseln sollte. Dieses Prinzip ist darauf gerichtet, einerseits eine hohe Kohäsion zu fördern, wobei sich jede Klasse auf eine bestimmte Aufgabe konzentriert, und andererseits eine geringe Kopplung zu anderen Klassen zu haben. Dabei entsprechen sich SRP und DRY insofern, dass Klassen mit klaren Zuständigkeiten erstellt werden, wodurch die Gefahr von Code-Duplizierung innerhalb dieser Klassen verringert wird. Das DRY-Prinzip fördert die Abstraktion gemeinsamer Funktionen in wiederverwendbare Komponenten, und SRP hilft bei der Erstellung dieser Komponenten mit klar definierten Zuständigkeiten, sodass diese in verschiedenen Teilen der Codebasis leichter wiederverwendet werden können.

Im Sinne von *Curly's Law*, das eine Verallgemeinerung des SRP darstellt, können wir DRY zusammen mit SRP, OAOO (Once-and-Only-Once) und SSOT (Single-Source-of-Truth) zusammenfassen *[3]*.

Das OAOO-Prinzip wird dabei als ein Subset beziehungsweise als Teilmenge von DRY betrachtet. Beide Prinzipien verfolgen das Ziel, Code-Duplizierung zu reduzieren, wobei OAOO eine spezifische Anleitung zur Erreichung von Code-Wiederverwendbarkeit und Wartbarkeit bietet. Es besagt, dass ein bestimmter Teil des Codes beziehungsweise der Logik in der Codebasis nur an einer Stelle vorhanden sein sollte. OAOO geht über die Verwendung gemeinsamer Funktionen hinaus und untersagt jegliche Wiederholung im Sinne kleiner, klar fokussierter Code-Einheiten.

Das Single-Source-of-Truth (SSOT) ist insofern mit DRY verwandt, dass beide Prinzipien eine einzige, maßgebliche Darstellung einer Information innerhalb eines Systems beschreiben. Das Ziel von DRY ist es, Redundanzen zu vermeiden, während es bei SSOT eine einzige Stelle gibt, an der eine Information definiert oder gespeichert ist.

Als letzter Punkt im Zusammenhang mit dem SOLID-Prinzip als Ergänzung zu DRY steht das Open-Closed-Prinzip (OCP). In diesem Kontext sollen Software-Einheiten (Klassen, Module, Funktionen) offen für Erweiterungen, aber geschlossen für Änderungen sein. Ein vollständig implementiertes und getestetes Modul sollte im Nachhinein nicht um neue Funktionen erweitert werden. Neue Funktionen sollten als Erweiterungen und nicht als Veränderung an bestehendem Code implementiert werden. DRY kann dabei helfen, eine spezielle Funktion an einem Ort zu repräsentieren, in Synergie mit OCP kann vermieden werden, die bestehende Funktionalität im Nachhinein durch Änderungen zu kompromittieren *[4]*.

Zusammenfassend können wir theoretisch feststellen, dass sich DRY gut in bestehende Prinzipien der objektorientierten Programmierung eingliedern und verwenden lässt.

Welches Problem wollen wir lösen?

Quarkus, GraalVM, Distroless-Deployments – Java hat längst Einzug in die Welt der Microservices gehalten. Die Großen wie Spotify und Netflix machen es vor *[6]*.

Auch im Kontext von Microservices ist DRY ein nicht zu vernachlässigendes Prinzip. Abgesehen davon, müssen Microservices im Vergleich zu herkömmlichen monolithischen Anwendungen etwas anders betrachtet werden.

Im Bereich Microservices werden häufig Dienste entwickelt, die die gleiche Schnittstelle oder eine ähnliche Logik implementieren (beispielsweise die Anbindung an einen Event-Stream) und damit einhergehend gleiche Bibliotheken und Datenstrukturen verwenden. DRY ist hinsichtlich Microservices speziellen Anforderungen unterworfen. Um dem DRY-Prinzip in einer Microservices-Architektur gerecht zu werden, muss ein Gleichgewicht zwischen der Wiederverwendung von Code, der Unabhängigkeit von Services und der Datenverwaltung (hier nicht Thema) gefunden werden. Kompromis-

se müssen sorgfältig betrachtet und Faktoren wie Service-Grenzen und Versionierung berücksichtigt werden *[5]*.

Eine Strategie könnte wie folgt aussehen:

- Sauber definierte Microservice-Grenzen. Eine Kontextgrenze markiert die Grenze eines bestimmten Domänenmodells *[7]*.
- Wiederverwendung gemeinsam genutzter Komponenten. Gemäß Bounded-Context (siehe auch Domain-Driven-Design) ist zwischen übertriebener Generalisierung und sinnvollem Codesharing zu unterscheiden *[8][10]*.
- Verwaltung der Datenkonsistenz. Gemäß seiner Domäne sollte ein Microservice niemals in seinen eigenen Transaktionen oder direkten Abfragen auf Tabellen/Speicher zugreifen, die im Besitz eines anderen Microservices liegen *[9]*.
- Berücksichtigung von Code-Duplikaten innerhalb eines Microservices (siehe auch Curly's Law) und im Kontext der spezifischen Domäne und seiner Anforderung.

Was heißt das konkret? In Bezug auf Microservices mit Java, die hypothetisch an einen Event-Stream/Message-Bus gekoppelt sind und sich an diesen über Pub-/Sub-Schnittstellen verbinden, um (gemäß diesem Beispiel) mit Google-Protobuf (Google-Protocol-Buffers) serialisierte Messages auszutauschen, lässt sich obiges auf ein verteiltes (*distributed*) Message-Schema (Google-Protocol-Buffers) anwenden. Dieses wird in einer Pipeline vorkompiliert und für die Java-Pub-/Sub-Dienste verfügbar gemacht. Die Java-Dienste müssen sich dadurch nur um das Mapping kümmern, während die Verwaltung, Versionierung, das Testing und die Kompilation an einen eigens hierfür konzipierten Dienst ausgelagert werden kann.

Würde jeder Java-Dienst für sich genommen ein eigenes Message-Schema implementieren, müssen Anpassungen, Versionierung und Testing sowie Kompilation ebenfalls durch diese Dienste abgedeckt werden. Das sorgt für jede Menge Overhead und bietet anhand der Redundanz und in vielerlei Hinsicht ein riesiges Potenzial für Fehler.

DRY in der Praxis mit Quarkus, GitHub Actions und Google-Protobuf

In diesem Beispiel soll anhand von zwei Quarkus-Diensten gezeigt werden, wie DRY in der Praxis funktionieren kann. Einer der Dienste ist ein Microservice (`java_dry_consumer`) und konsumiert hierbei ein Google-Protocol-Buffer-Schema (Message-Schema). Der Dienst, der für die Erstellung des Google-Protocol-Buffer-Schemas zuständig ist (`java_dry_producer`), wird später nicht als produktiver Microservice laufen, sondern lediglich als Provider für die Protocol-Buffer dienen. Sowohl der `java_dry_producer` als auch der `java_dry_consumer` werden in einem spezifischen GitHub-Repository verwaltet.

Um ein „distributed" Message-Schema nach dem DRY-Prinzip zu erhalten, soll wie folgt vorgegangen werden:

1. Implementieren der notwendigen Abhängigkeiten in der lokalen `pom.xml` des `java_dry_producer`-Services.
2. Anpassen der lokalen `.m2/settings.xml` und Erstellen eines GitHub-Tokens.
3. Erstellen des Google-Protocol-Buffer-Schemas (Message-Schema).
4. Erstellen eines GitHub-Workflows.
5. Einbinden der GitHub-Maven-Packages als Dependency in der lokalen `pom.xml` des `java_dry_consumer`-Dienstes.

Nachfolgend ist eine vereinfachte Projektstruktur *(siehe Listing 1)* der beiden Quarkus-Maven-Dienste abgebildet.

```
|--java_dry_consumer
|   |--src
|   |   |--main
|   |   |   |--java
|   |   |   |   |--com.example
|   |   |   |   |   |--Controller
|   |   |   |   |   |--SampleMessage
|   |   |   |   |   |--Service
|   |   |   |--resources
|   |--pom.xml

|--java_dry_producer
|   |--.github
|   |   |--workflows
|   |   |   |--package-ci.yml
|   |--src
|   |   |--main
|   |   |   |--java
|   |   |   |--proto
|   |   |   |   |--message.proto
|   |   |   |--resources
|   |--prom.xml
```

Listing 1: Projektstruktur

Hinweis: Google-Protocol-Buffers (Protobuf) gehören zur Familie der Protocol-Buffer. Es ist ein Dateiformat zur Serialisierung mit einer Schnittstellen-Beschreibungssprache *[11]*. Protobuf ist ein plattformneutraler, erweiterbarer Mechanismus zur Serialisierung strukturierter Daten *[12]*. Im Vergleich zu beispielsweise JSON (Text) wird Protobuf zu einer binären Datei kompiliert und ist typisiert. Die Dateiendung für Protobuf-Dateien ist `.proto`.

1. Implementieren der notwendigen Abhängigkeiten in der lokalen `pom.xml` des `java_dry_producer`-Services.

Obwohl der `java_dry_producer` keinerlei Java -Code in der Projektstruktur enthält, ist es ein Quarkus-Maven-Projekt. Der Dienst enthält dabei nicht mal einen lokalen `target`-Ordner. Der Maven-Build wird vollständig an die Pipeline übergeben. Damit das Message- beziehungsweise Protobuf-Schema per GitHub-Workflows-Pipeline gebaut werden kann, benötigen wir folgende, zusätzliche Abhängigkeiten in der `pom.xml` des `java_dry_producer`-Dienstes *(siehe Listing 2)*.

```
<dependencies>
      ...
      <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-grpc</artifactId>
      </dependency>
      <dependency>
            <groupId>com.google.protobuf</groupId>
            <artifactId>protobuf-java-util</artifactId>
      </dependency>
</dependencies>
```

Listing 2: java_dry_producer Maven Dependencies

```
 <distributionManagement>
   <repository>
      <id>github</id>
      <name>GitHub OWNER Apache Maven Packages</name>
      <url>https://maven.pkg.github.com/OWNER/REPOSITORY</url>
   </repository>
</distributionManagement>
```

Listing 3: java_dry_producer Maven Dependencies. OWNER und REPOSITORY müssen gemäß des eigenen Repositories personalisiert werden.

Damit Maven das Paket in der GitHub-Registry des Nutzers veröffentlichen kann, wird folgender Zusatz in der `pom.xml` benötigt *(siehe Listing 3)*.

Da das `java_dry_producer`-Projekt keinen Java-Code enthält, können in der `pom.xml` die für einen Image-Build notwendigen Zusätze, wie `systemPropertyVariables`, `native`-Profiles und Test-Generatoren entfernt werden.

2. Anpassen der lokalen `.m2/settings.xml` und Erstellen eines GitHub-Tokens.

Zunächst muss ein Personal-Access-Token (PAT) auf GitHub erstellt werden. Dazu muss in den GitHub-Settings auf die *Developer Settings* geklickt werden und anschließend auf *Personal Access Tokens*, um dort einen Token von GitHub erstellen zu lassen. Der Token sollte `repo:read/write`-Rechte erhalten, muss kopiert und in der lokalen `.m2/settings.xml` eingefügt werden *(siehe Listing 4) [15]*. Er wird benötigt, um Maven zu erlauben, Pakete zu veröffentlichen, zu überschreiben und notfalls zu löschen, da Maven später über die GitHub-Workflows ausgeführt wird und ein Paket in der GitHub-Registry veröffentlichen soll.

3. Erstellen des Google-Protocol-Buffer-Schemas (Message-Schema).

Als Nächstes wird die Datei `message.proto` *(siehe Listing 1)* im `java_dry_producer`-Service mit folgendem Inhalt erstellt *(siehe Listing 5)*. Hier werden beispielhaft Metadaten (PayloadMeta) und eine Payload vorgegeben.

Im Sinne des Open-Closed-Prinzips (OCP) sollte die Nummerierung der Protobuf-Daten *(siehe Listing 5)* so gewählt werden, dass die Möglichkeit der Erweiterung gegeben ist. Sollte sich die Nummerierung ändern, müsste das Mapping in den konsumierenden Diensten ebenfalls vollständig angepasst werden. Beispielsweise könnte der Payload das Argument `string education` = 6; angehängt werden. Sinnvolles Anhängen würde keine Neuimplementierung des Mappings im `java_dry_consumer` zur Folge haben.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                              http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <activeProfiles>
     <activeProfile>github</activeProfile>
  </activeProfiles>

  <profiles>
     <profile>
        <id>github</id>
        <repositories>
           <repository>
              <id>central</id>
              <url>https://repo1.maven.org/maven2</url>
         </repository>
           <repository>
              <id>github</id>
              <url>https://maven.pkg.github.com/OWNER/REPOSITORY</url>
              <snapshots>
                 <enabled>true</enabled>
              </snapshots>
           </repository>
        </repositories>
     </profile>
  </profiles>

  <servers>
     <server>
        <id>github</id>
        <username>USERNAME</username>
        <password>TOKEN</password>
     </server>
  </servers>
</settings>
```

Listing 4: Lokale .m2/settings.xml. Der Text in Großbuchstaben muss personalisiert werden.

```
syntax = "proto3";

package protobuf;

option java_outer_classname = "SampleMessageProto";
option java_package = "com.example.java_dry_producer";

message SampleMessage {
  PayloadMeta payload_meta = 1;
  message PayloadMeta {
     string origin_name = 1;
     string origin_version = 2;
     string uuid = 4;
  }

  Payload payload = 2;
  message Payload {
     string user_name = 1;
     string user_surname = 2;
     string profession = 3;
     int32 age = 4;
     string phone = 5;
  }
}
```

Listing 5: java_dry_producer-Message-Schema (message.proto). syntax: Protobuf-Version. java_outer_classname: generierter Java Klassenname, der dieses File und seine Funktionen repräsentieren soll. java_package: Name des generierten Java-Pakets, in dem die generierten Klassen enthalten sind.

4. Erstellen eines GitHub-Workflows.

In diesem Schritt soll die Pipeline erstellt werden *(siehe Listing 6)*. Die Pipeline soll sich darum kümmern, Java-Klassen aus dem Message-Schema *(siehe Listing 5)* zu generieren, diese zu paktieren und für andere Dienste über die GitHub-Package-Registry verfügbar zu machen.

Die Pipeline kann man sich initial von GitHub generieren lassen und anschließend personalisieren (beispielsweise durch Anpassen des Triggers). Die Workflow-Befehle `mvn -B package --file pom.xml` und `maven deploy` verpacken den Code zuerst in eine JAR, um diese anschließend als Paket in der GitHub-Registry des Nutzers zu veröffentlichen. Standardmäßig veröffentlicht GitHub das Paket in einem bestehenden Repository mit dem gleichen Namen wie dem Namen des Pakets *[13]*.

5. Einbinden der GitHub-Maven-Packages als Dependency in der lokalen pom.xml des java_dry_consumer-Dienstes.

Die Dependency muss nun in der `pom.xml` des `java_dry_consumer`-Services *(siehe Listing 7)* eingebunden werden. Die Version muss dabei der korrekten Version des Maven-Pakets entsprechen.

Die Projektstruktur *(siehe Listing 1)* gibt drei Klassen im `java_dry_consumer` an. Es gibt dort einen *Controller*, der einen herkömmlichen POST-Request mit einem JSON-Body entgegennehmen kann *(siehe Listing 8)*.

```
name: Java Maven Package Deployment CI

on:
  push:
     branches:
        - '**'
  pull_request:
     branches:
        - '*'

jobs:
  build:
     runs-on: ubuntu-latest
     permissions:
        contents: read
        packages: write

     steps:
        - uses: actions/checkout@v3
        - name: Set up JDK 17
           uses: actions/setup-java@v3
           with:
              java-version: '17'
              distribution: 'temurin'
            server-id: github # Value of the distributionManagement/repository/id field of the pom.xml
              settings-path: ${{ github.workspace }} # location for the settings.xml file

        - name: Build with Maven
           run: mvn -B package --file pom.xml

        - name: Publish to GitHub Packages Apache Maven
           run: mvn deploy -s $GITHUB_WORKSPACE/settings.xml
           env:
              GITHUB_TOKEN: ${{ github.token }}
```

Listing 6: package-ci.yml (siehe Listing 1).

${{ github.workspace }}: Kennzeichnet das Working-Directory beziehungsweise das (geklonte) Repository, das vom GitHub-Runner genutzt wird, um den Workflow auszuführen [13]. ${{ github.token }}: GitHub-interner, generierter Installationszugriffstoken, der seitens GitHub für die Authentifizierung verwendet wird, um das gebaute Maven-Package in der Registry zu veröffentlichen.

Im *Controller* *(siehe Listing 9)* wird der Request-Body an den *Service* weitergereicht, dieser ordnet die Daten aus dem Request-Body den Parametern des Message-Schemas (`message.proto`) zu. Der *Controller* gibt in diesem Beispiel lediglich die Protobuf-Message mit einem simplen *Return* zurück. In einem realitätsnahen Projekt würde man an dieser Stelle vermutlich einen weiteren Service implementieren, der sich beispielsweise um die Anbindung an einen Event-Stream kümmert und die Message dort publiziert. Außerdem sollte die Schnittstelle im *Controller* für einen projektnahen Einsatz mit *restful* erweitert werden und die erforderlichen HTTP-Status-Codes zurückgeben.

```
<dependency>
     <groupId>com.example</groupId>
     <artifactId>java_dry_producer</artifactId>
     <version>1.0.0-SNAPSHOT</version>
</dependency>
```

Listing 7: java_dry_consumer Maven Dependencies

Der Service *(siehe Listing 10)* baut aus dem eingehenden JSON-Body *(siehe Listing 8)* mithilfe des Protobuf-Schemas (hier beispielhaft für die eingehenden Meta-Daten) eine Google-Protocol-Buffer-Message.

Für das Mapping kann hier ein simples DTO (Data Transfer Object) als Hilfestellung genommen werden *(siehe Listing 11)*.

```
curl --location 'localhost:8080/data' \
--header 'Content-Type: application/json' \
--data '{
  "originName": "Postman",
  "originVersion": "1.0.0",
  "uuid": "random_token",
  "userName": "Someone",
  "userSurname": "Someones_Surname",
  "profession": "Something",
  "age": 42,
  "phone": "mobile"
}'
```

Listing 8: POST-Request mit JSON-Body

```
@Path("/data")
public class Controller {
  @POST
  @Consumes(APPLICATION_JSON)
  public AbstractMessage handleRequest(final SampleMessage message) {
     return Service.message(message);
  }
}
```

Listing 9: Controller-Code

```
public static PayloadMeta payloadMetaMapper
(final SampleMessage message) {
     return PayloadMeta.newBuilder()
            .setOriginName(message.getOriginName())
            .setOriginVersion(message.getOriginVersion())
            .setUuid(message.getUuid()).build();
  }
```

Listing 10: Service-Code-Snippet

```
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
public class SampleMessage extends AbstractMessage {
  public String originName;
  public String originVersion;

     ...

  @JsonCreator
  SampleMessage(@JsonProperty(value = "originName", required = true) final String originName,
                        @JsonProperty(value = "originVersion", required = true) final String originVersion,
                        @JsonProperty(value = "uuid", required = true) final String uuid,
                        @JsonProperty(value = "userName", required = true) final String userName,
                        @JsonProperty(value = "userSurname", required = true) final String userSurname,
                        @JsonProperty(value = "profession", required = true) final String profession,
                        @JsonProperty(value = "age") final int age,
                        @JsonProperty(value = "phone") final String phone) {
     this.originName = originName;
     this.originVersion = originVersion;
     ...
  }

...
}
```

Listing 11: SampleMessage DTO-Code-Snippet

Der `java_dry_consumer` implementiert nun beispielhaft das Protobuf-Message-Schema, dabei muss nur eine versionierte Abhängigkeit in der `pom.xml` gepflegt werden *(siehe Listing 7)*. Alle weiteren Aktionen können gemäß des DRY-Prinzips gekapselt in den jeweiligen Diensten stattfinden.

Zusammenfassung

Um dem DRY-Prinzip in einer Microservices-Architektur gerecht zu werden, muss unter anderem ein Gleichgewicht zwischen der Wiederverwendung von Code und der Unabhängigkeit von Services gefunden werden. In diesem Beispiel wurde versucht, aufzuzeigen, dass dies mithilfe eines Protobuf-Schemas (`message.proto`), das über eine Pipeline zugehörige Java-Mapping-Klassen erstellt und über ein Maven-Package versioniert, in konsumierende Java-Dienste eingebunden werden kann. Dabei sind die Schemata „Produzent" und „Konsument" weitgehend unabhängig voneinander und beziehen sich lediglich auf den Vertrag von DRY im Zusammenhang mit OCP und Curly's Law. So kann dadurch klassisches OOP im Rahmen von Microservices nutzbar gemacht und auf eine SOLIDe Basis gestellt werden.

Quellen

[1] Wikipedia 2023: Don't repeat yourself. *https://de.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself*

[2] Baeldung 2023: Software Design Principle. *https://www.baeldung.com/cs/dry-software-design-principle*

[3] Wikipedia 2023: SRP. *https://de.wikipedia.org/wiki/Single-Responsibility-Prinzip*

[4] Digitalocean 2023: SOLID. *https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-de*

[5] Codecentric 2023: Was man vom Microservices Hype mindestens mitnehmen sollte. *https://www.codecentric.de/wissens-hub/blog/was-man-vom-microservice-hype-mindestens-mitnehmen-sollte*

[6] Asioso 2023: Anwendungen und Praxisbeispiele von Microservices. *https://www.asioso.com/de_DE/blog/anwendungen-und-praxisbeispiele-von-microservices-b602*

[7] Microsoft 2023: Microservices Boundaries. *https://learn.microsoft.com/de-de/azure/architecture/microservices/model/microservice-boundaries*

[8] Codecentric 2023: Shared Code in Microservices. *https://www.codecentric.de/wissens-hub/blog/shared-code-in-microservices*

[9] Microsoft 2023: Architect Microsoft Container Apps. *https://learn.microsoft.com/de-de/dotnet/architecture/microservices/architect-microservice-container-applications/distributed-data-management*

[10] Eric Evans 2023: Bounded Context. *https://www.infoq.com/news/2019/06/bounded-context-eric-evans/*

[11] Wikipedia 2023: Protocol Buffers. *https://de.wikipedia.org/wiki/Protocol_Buffers*

[12] Google 2023: Protobuf. *https://protobuf.dev/*

[13] GitHub 2023: GitHub hosted runners. *https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners*

[14] Apache 2023: Introduction to the lifecycle. *https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html*

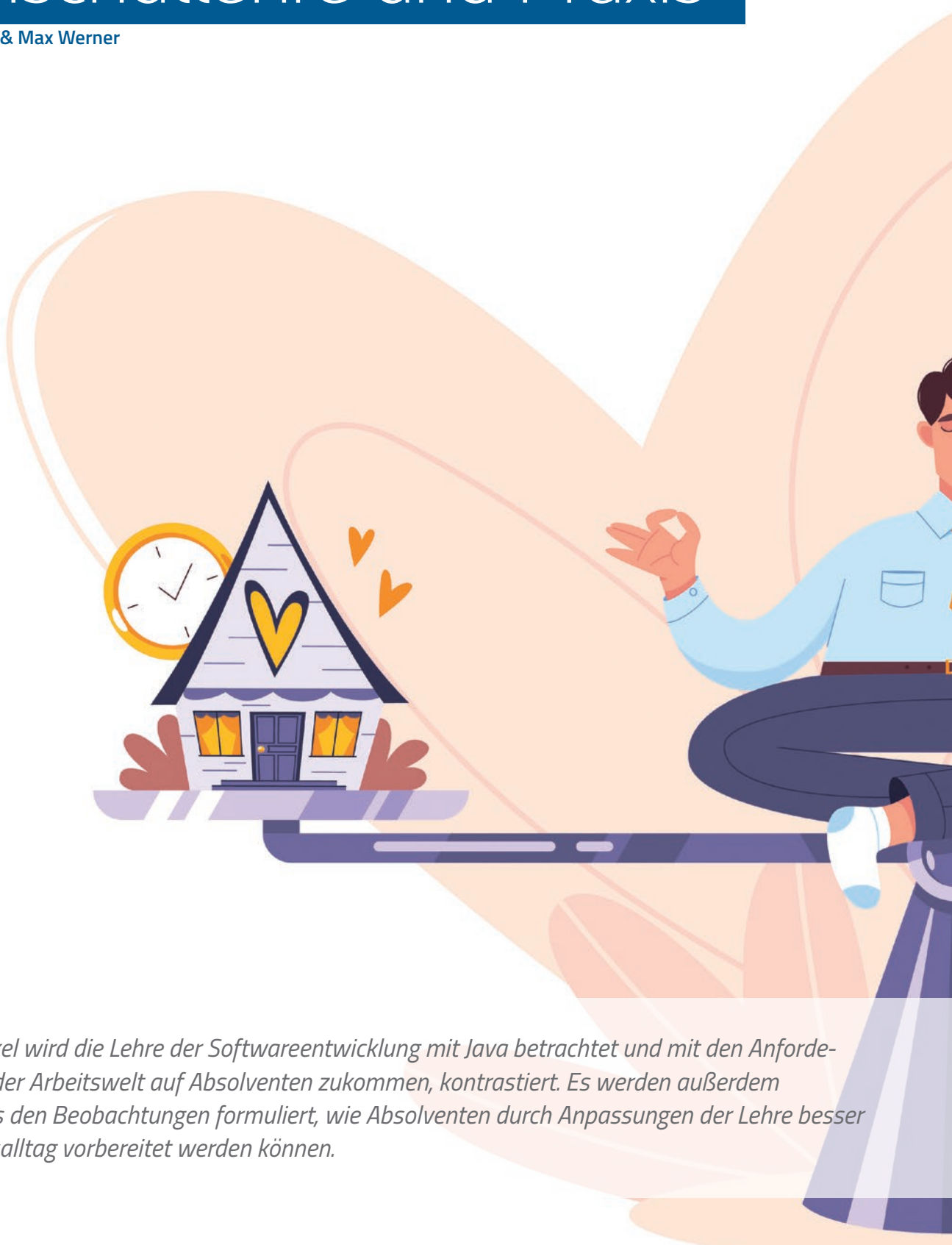
[15] GitHub 2023: GitHub Dokumentation. https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-apache-maven-registry

Ben Bajorat
Mayflower GmbH
benjamin.bajorat@mayflower.de

Ben Bajorat ist Software Engineer bei Mayflower GmbH. Sein Fokus liegt vor allem auf Microservices-Architekturen und PaaS mit Kubernetes und Security. Modernisierung von Legacy-Systemen und agiles Arbeiten gehört neben dem Interesse an Testen von neuen Technologien ebenso zu seinen Tätigkeiten wie der Projektalltag und dem Arbeiten mit und für den Kunden.

Softwareentwicklung mit Java – Vergleich zwischen Hochschullehre und Praxis

Dominik Martens & Max Werner



In diesem Artikel wird die Lehre der Softwareentwicklung mit Java betrachtet und mit den Anforderungen, die in der Arbeitswelt auf Absolventen zukommen, kontrastiert. Es werden außerdem Vorschläge aus den Beobachtungen formuliert, wie Absolventen durch Anpassungen der Lehre besser auf den Berufsalltag vorbereitet werden können.

Was wird im Studium (Bachelor und Master) gelehrt?

Die Programmiersprache Java besitzt an der Fakultät Informatik der Ostfalia Hochschule in Wolfenbüttel einen hohen Stellenwert. Die Studenten werden bereits im ersten Semester mit der Programmiersprache Java an die Grundlagen der Programmierung herangeführt.

Das vermittelte theoretische Wissen umfasst dabei zunächst nur die grundlegendsten Konzepte, wie elementare Datentypen, imperative Programmierung, Funktionen, die Speicherorganisation sowie einen Einstieg in die Objektorientierung. Zur Festigung des theoretischen Wissens werden in der Regel über das Semester verteilt kleine praktische Übungen zu einzelnen Themengebieten durchgeführt.

Im zweiten Semester werden die erlangten Java-Grundlagen durch ein weiteres Modul vertieft. Ziel der Vertiefung ist es, dass die Studenten an die Realisierung umfangreicher, objektorientierter Software auf der Client-Seite herangeführt werden. Die praktische Ausbildung innerhalb des Moduls umfasst die Implementierung eines Spiels mit grafischer Benutzeroberfläche, die in kleinen Teams über das gesamte Semester hinweg durchgeführt wird.

Die Studenten lernen im Rahmen der Implementierung Konzepte wie zum Beispiel Exceptions, Collections, stream-orientierte I/O, grafische Benutzeroberflächen und Threads kennen. Zudem werden auch einige der wichtigsten OOP-Design-Patterns gelehrt, die zur Lösung von allgemeinen Problemen verwendet werden können. Die zu implementierende grafische Benutzeroberfläche wird beispielsweise über das MVC-Pattern mit JavaFX realisiert.

Neben diesen Konzepten wird im Laufe des Bachelorstudiums in einem weiteren Modul die Implementierung wichtiger Algorithmen (wie etwa grundlegende Sortier-Algorithmen) und Datenstrukturen (wie zum Beispiel Listen- oder Baumstrukturen) mit Java gelehrt. Weiterhin lernen die Studenten die bereits im JDK zur Verfügung gestellten Implementierungen von Algorithmen und Datenstrukturen wiederzuverwenden, schließlich soll bei der Implementierung von Software nicht ständig das Rad neu erfunden werden. Im Modul wird vor allem gelehrt, die Laufzeit- und Speicherkomplexitäten von Algorithmen einzuschätzen, die Effizienz von Algorithmen zu bewerten und somit die für spezifische Anwendungsfälle optimalen Algorithmen auszuwählen.

Im Bachelorstudium müssen Studenten in den höheren Semestern auch einige Wahlpflichtfächer für bestimmte Vertiefungen absolvieren. Darunter befinden sich unter anderem Module, bei denen die zuvor erlangten Java-Kenntnisse erweitert werden sollen. So existiert beispielsweise ein Modul, das eine Einführung in die Jakarta-Enterprise-Edition gewährt und Themen wie Jakarta Server Faces, Jakarta Enterprise Beans und das Jakarta-Persistence-API behandelt. Somit bekommen die Studenten in den höheren Semestern bereits einen theoretischen und praktischen Einblick in die Entwicklung von Backend-Systemen mit Java. In der Vergangenheit wurden zudem Wahlpflichtfächer für die Entwicklung von mobilen Applikationen angeboten, sodass die Studenten die Grundlagen für die Implementierung einer Java-Applikation für Android-Smartphones kennenlernen konnten.

Im Bachelorstudium an der Ostfalia wird jedoch nicht nur die Programmiersprache Java gelehrt. So lernen etwa Studenten mit der Vertiefungsrichtung Software Engineering mindestens eine weitere Programmiersprache. In der Vergangenheit war dies meistens die funktionale Programmiersprache Haskell oder C++ als weitere objektorientierte Programmiersprache.

Bezüglich der Testautomatisierung mit Java wurden in manchen Veranstaltungen des Bachelorstudiums nur vorgefertigte JUnit-Tests verwendet, um die Implementierungen der Studenten zu prüfen. Für das Design und die Implementierung eigener Unittests gab es damals keine Veranstaltung. Erst seit Kurzem wird eine eigene Veranstaltung für Qualitätssicherung und Testen im Bachelorstudium angeboten.

Im Informatik-Masterstudiengang der Ostfalia werden ebenfalls einige fortgeschrittene Themengebiete in praktischer Realisierung mit Java gelehrt. Jedoch unterliegt dies einer starken Fluktuation, da die Module des Masterstudiengangs im Rahmen eines Zwei-Jahres-Plans festgelegt sind und als Wahlpflichtkurse angeboten werden. Welche weiteren Java-Kenntnisse im Masterstudium erlangt werden können, ist somit stark davon abhängig, welcher Plan aktuell vorliegt und welche Module individuell ausgesucht werden.

Zu den Themengebieten, die in den vergangenen zwei Jahren im Rahmen des Masterstudiums mit Java gelehrt wurden, zählten:

- Big Data – hier wurden große Datenmengen mit Hadoop Map/Reduce und Apache Spark verarbeitet.
- Architekturen moderner Informationssysteme – hier überlegen sich die Studenten ein Projekt für ein System mit einer Microservice-Architektur und realisieren dieses (etwa eine Online-Videoplattform mithilfe des Quarkus-Frameworks).
- Die Entwicklung großer Anwendungssysteme – hier wurden die Konzepte der Jakarta-Enterprise-Edition praktisch gelehrt.

Was wird in der Ausbildung gelehrt?

Die praktischen Inhalte der Berufsausbildung (zum Beispiel Ausbildung zum Informatiker für Anwendungsentwicklung) sind stark abhängig vom ausbildenden Betrieb. Häufig wird als erste Programmiersprache entweder Java oder C# gelehrt. Zwar sind praktische Projekte häufig Teil der Ausbildung, allerdings hängt der Umfang und die Qualität dieser sehr stark vom Ausbildungsbetrieb und dem betreuenden Ausbilder ab. Nicht immer haben Auszubildende die Möglichkeit, in den höheren Lehrjahren zeitweise den normalen Projektalltag kennenzulernen.

Welche (Java-)Kenntnisse innerhalb der Berufsausbildung vermittelt werden, variiert also stark und Auszubildende aus verschiedenen Betrieben sind häufig im Kenntnisstand nicht vergleichbar.

Wie lässt sich der vermittelte Kenntnisstand gegenüber erfahreneren Kollegen einordnen?

Die im Studium vermittelten grundlegenden theoretischen Java-Kenntnisse sind erfahrungsgemäß ausreichend, um beim Berufseinstieg nach dem Studium bei einfachen Anwendungsfällen gut mit bereits erfahrenen Kollegen mithalten zu können. Die Unterschiede zwischen frischen Absolventen und den erfahrenen Kollegen zeigen sich jedoch vor allem bei komplexeren Implementierungen, wie zum Beispiel Multithreading-Anwendungen oder performancekritischen

Anwendungsfällen, die im Studium selten, bis gar nicht praktisch behandelt werden.

Ein weiterer großer Unterschied ergibt sich häufig in der Qualität des produzierten Codes. Die praktischen Programmierkenntnisse werden im Studium überwiegend durch Projekte vermittelt, die auf einen Zeithorizont von einem Semester beschränkt sind. Innerhalb eines solchen ist es nahezu unmöglich, die Relevanz der langfristigen Wartbarkeit und Erweiterbarkeit von Software praktisch zu vermitteln.

Der Fokus des produzierten Codes innerhalb dieser Projekte liegt somit überwiegend auf der Funktionalität und nicht auf anderweitigen qualitativen Metriken, wie der Erweiterbarkeit oder Lesbarkeit. Bei langfristigen Projekten in der freien Wirtschaft ist dies jedoch ein entscheidendes Kriterium, um zum Beispiel Kosten für nachträgliche Software-Anpassungen gering zu halten. Solche Erfahrungen können somit erst nach dem Berufseinstieg gemacht werden.

Bei der Vermittlung der Softwareentwicklung im Studium ist es zudem von Nachteil, dass es für die implementierte Java-Software, beziehungsweise Software im Allgemeinen, innerhalb von Semester-Projekten meist keine realen Anwender gibt, die das System während oder nach der Entwicklung verwenden. Die entstandene Software muss lediglich am Ende des Semesters die Abnahme vom jeweiligen Dozenten bestehen. Durch diesen Prozess können die Studenten in der Regel keine Erfahrung darüber sammeln, in welchen Anwendungsfällen ein gewisses Verhalten verschiedener Anwendergruppen zu erwarten ist, und wie dies in der entsprechenden Software berücksichtigt werden muss. Diese Erfahrung kann somit meist erst beim Berufseinstieg gesammelt werden und die frischen Absolventen des Informatik-Studiums haben dadurch einen Nachteil gegenüber den bereits erfahrenen Kollegen.

Bezüglich der Implementierung von automatisierten Tests mit Java haben insbesondere die frischen Absolventen einen Nachteil gegenüber erfahrenen Kollegen, die innerhalb des Studiums noch kein Modul für Qualitätssicherung und Testen absolvieren mussten. Die Studenten haben oftmals nur wenig oder keine Erfahrung mit Frameworks zur Realisierung von automatisierten Tests wie JUnit. Dies muss entsprechend in der beruflichen Praxis nachgeholt werden.

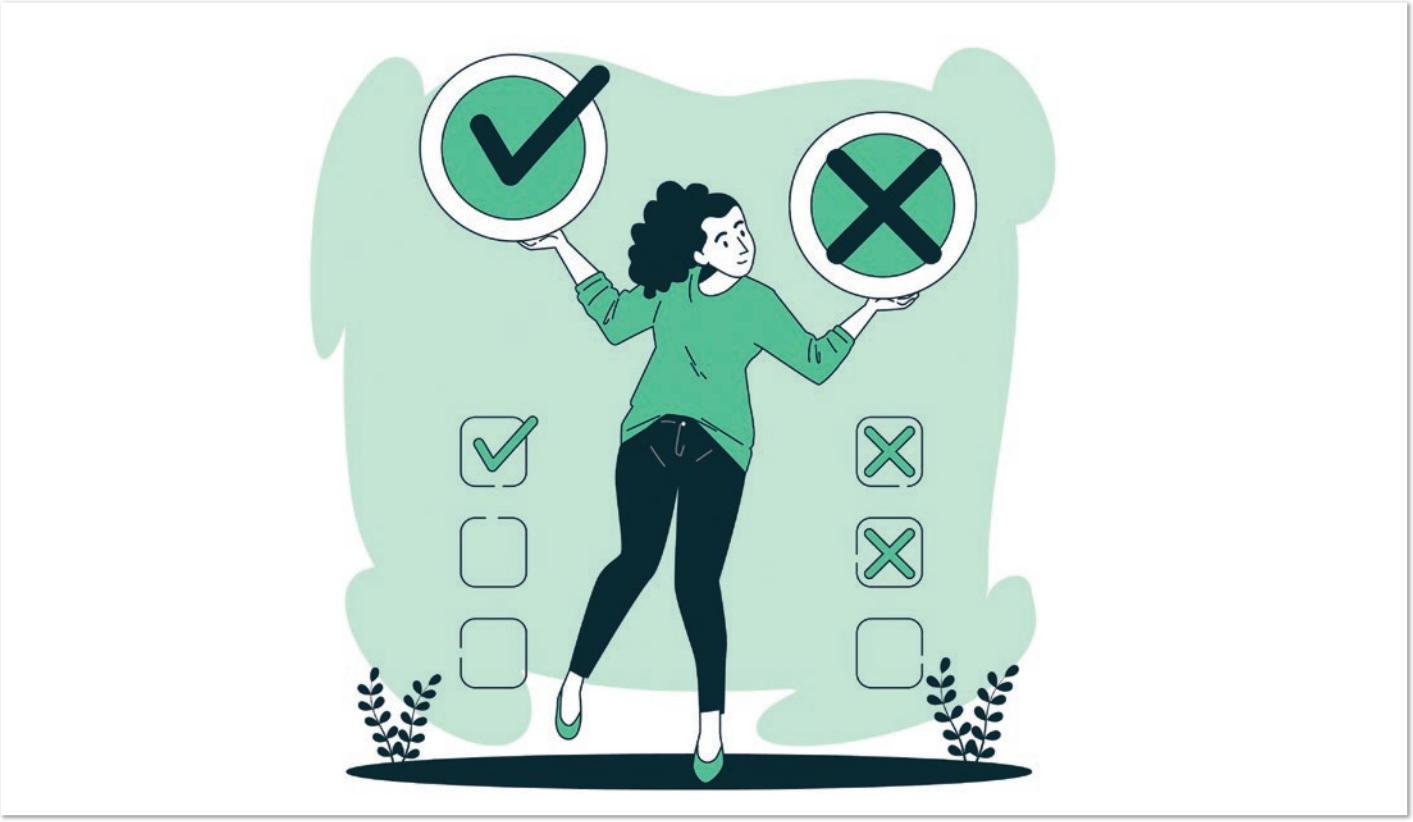
Nachteilig ist auch, dass einige wichtige Frameworks, die eine hohe Relevanz in der freien Wirtschaft haben, wie etwa das Spring Framework, im Studium bewusst kaum bis gar nicht behandelt werden. Dies gestaltet einen Berufseinstieg schwierig, wenn Kenntnisse über solche Frameworks vorausgesetzt werden.

Was wird in der Praxis benötigt?

Die in der Praxis/im Beruf benötigten Dinge unterscheiden sich teils stark von den im Basisstudium (Bachelor) gelehrten Themen, bauen jedoch häufig auf diese auf.

Im Studium muss bei Projekten häufig sämtliche Funktionalität durch die Studenten von Grund auf selbst entwickelt werden. Dies mag zwar an mancher Stelle sinnvoll sein, um so auch die Grundlagen selbst zu lernen, allerdings verpassen Studenten hierdurch einen wichtigen Teil der Arbeitsweise in der Praxis: Das Arbeiten mit und das Verwalten von externen Dependencies. Im besten Fall dürfen Studenten bei einzelnen Projekten beliebig externe Dependencies verwenden, sind dann allerdings komplett auf sich allein gestellt, was das Einbinden und Nutzen derselben (sowie generell den Umgang mit externen Dependencies) angeht.

In der Praxis kommt es auch immer häufiger vor, dass Operations (Deployments, etc.), die früher häufig in eigene Teams/Sub-Or-



ganisationen ausgelagert wurden, wieder zurück in die Entwicklungsteams gehen, gerade auch durch Trends wie DevOps und agile Teams. Hierbei müssen Entwickler dann Aufgaben wie Deployments – häufig inklusive der Verwaltung von Servern (entweder lokal oder über die Cloud) – und die Verwaltung (oder zumindest fachgerechte Nutzung) von CI/CD-Pipelines übernehmen. Im Studium werden diese Themen wenig bis gar nicht behandelt.

Außerdem wird immer häufiger mit Cloud-Services wie zum Beispiel AWS-Lambdas gearbeitet. An der Hochschule wird Cloud-Infrastruktur (zugegebenermaßen ein komplexes und sich schnell entwickelndes Thema) im Masterstudium zwar teils theoretisch vorgestellt, praktische Erfahrungen sammeln Studenten jedoch nicht. Im Bachelorstudium wird dieses Thema komplett ignoriert.

Vor allem in der Anwendungsentwicklung werden oft Fullstack-Entwickler gesucht – besonders von Firmen, die in multidisziplinären, agilen Teams arbeiten. Da moderne Web-Frontends im Informatik-Studium häufig nur kurz vorkommen (im Grundstudium existiert nur ein freiwilliges Modul, in dem die Grundlagen von HTML, CSS und JavaScript beigebracht werden), ebenso wie das Entwerfen dedizierter Backends/Backend-Architekturen für diese (wobei letzteres im Masterstudium thematisiert wird), sind Studenten auf diese Anforderungen im Arbeitsleben wenig bis gar nicht vorbereitet. Immer seltener wird nach reinen Java-Entwicklern gesucht.

Zudem kommt es immer öfter vor, dass Teams, die früher ausschließlich mit Java gearbeitet haben, neue Projekte in Kotlin anfangen oder sogar vorhandene Projekte nach und nach zu Kotlin umwandeln. Vor allem durch die Interaktivität und Syntaxnähe von Java und Kotlin kommt es also durchaus häufiger vor, dass zu Kotlin gegriffen wird. Hier hat man einen Vorteil, wenn man sich schon mit der Sprache beschäftigt hat – wobei anzumerken ist, dass Kotlin bei entsprechender Erfahrung mit Java auch recht einfach zu erlernen ist. Hier könnte man es Studenten in höheren Semestern oder im Masterstudium freistellen, in welcher Sprache (Java oder Kotlin) sie entwickeln. Eine Möglichkeit, die von einzelnen Professoren auch durchaus schon ergriffen wurde, um näher an die Realität der Industrie zu rücken.

Was könnte in der Lehre bezüglich Java verbessert werden?

Grundlegend ist die Java-Ausbildung an der Informatikfakultät der Ostfalia Hochschule sehr solide und befähigt die Absolventen, beim Berufseinstieg an der produktiven Entwicklung von Java-Software teilzuhaben. Insbesondere für die Backend-Entwicklung mit Java haben Absolventen einen guten Kenntnisstand durch die theoretischen und praktischen Einblicke in die Jakarta-Enterprise-Edition und ORM-Frameworks, wie beispielsweise Hibernate in den höheren Semestern des Bachelorstudiums, beziehungsweise den Weiterführungen im Masterstudium.

Wünschenswert im Rahmen der theoretischen Java-Ausbildung im Studium wären jedoch mehr Grundlagen über die Funktionsweise der Java Virtual Machine, um ein tieferes Hintergrundwissen über die Prozesse in der JVM zu erlangen. JVM-Grundlagen, wie der Kompilierprozess von Java-Code, werden im Studium nur oberflächlich behandeltDadurch fehlt ein größeres Gesamtbild über Java-Anwendungen und deren Laufzeitumgebung, um zum Beispiel bei der Entwicklung Möglichkeiten für tiefgreifende Optimierungen zu erkennen und durchführen zu können.

Des Weiteren wäre eine Vermittlung von Grundlagen über die Wartung von Java-Anwendungen im Betrieb sehr sinnvoll. Dazu können beispielsweise Möglichkeiten der Überwachung und Analyse des Speicherverbrauchs von Java-Anwendungen zählen, um etwaige Fehler im Betrieb besser analysieren zu können. Solche Aspekte bekommen in der Praxis für Java-Entwickler eine steigende Relevanz, insbesondere durch die steigende Beliebtheit des DevOps-Ansatzes.

Ebenso im Bereich von DevOps wäre es sinnvoll, die schon oben genannte Verwendung von CI/CD-Pipelines und Cloud-Technologien, mit Fokus auf jene, die im Java-Umfeld verwendet werden, in den Kursplan aufzunehmen. Oder etwas über Tools zu lehren, die in der Praxis häufig genutzt werden, um Java-Anwendungen an die Kunden zu bringen.

Zudem kam in unserem Studium die Testautomatisierung mit Java, beziehungsweise allgemein auch theoretische Grundlagen über die Konzeption von automatisierten Tests, viel zu kurz. Im gesamten Bachelor- und Masterstudium musste kein automatisierter Test mit Java oder einer anderen Programmiersprache implementiert werden. Dadurch haben Absolventen beim Berufseinstieg einen enormen Nachteil, da in der Regel bei der Implementierung von neuen Features oder der Anpassung von existierenden Features einer Software die Implementierung von entsprechenden automatisierten Tests gefordert ist. Die Hochschule hat dieses Problem bereits erkannt und dafür in den letzten Jahren bereits eine neue Veranstaltung im Bachelorstudium eingeführt. Inwieweit diese Veranstaltung den Anforderungen in der Praxis gerecht wird, können wir jedoch nicht beurteilen.

Ein weiterer Punkt, mit dem Studenten zumeist im Studium nicht nicht in Berührung kommen – der im Arbeitsleben allerdings alltäglich ist –, ist das Lesen von fremdem Code und das Einarbeiten in komplexe (schon vorhandene) Systeme. Ob es hier lehrbare Methoden gibt, wissen wir nicht, allerdings ließe sich diese wichtige Fähigkeit praktisch durchaus beibringen: Zum Beispiel, indem Studenten Studierende ein vorhandenes Projekt erweitern müssen, eine Möglichkeit, die bisher von den Lehrenden nur selten genutzt wurde.

Was könnte in der Lehre bezüglich der Arbeitsweise verbessert werden?

Abgesehen von wenigen Gruppenprojekten (und selbst in diesen nur teils), wird während des Studiums meist eine Denkweise des einzelnen Entwicklers „für sich" gefördert. Dies widerspricht stark dem Projektalltag, in dem fast immer in Teams gearbeitet wird. Gerade in einer Zeit, in der der Trend hin zu selbstorganisierten Teams geht, ist die Ausbildung solcher „Einzelkämpfer" kontraproduktiv. Wir haben im Folgenden einige Punkte aufgeführt – zusammengefasst unter dem Begriff „Arbeitsweise" –, bei denen wir (dringenden) Bedarf für ein erhöhtes Augenmerk in der Lehre sehen, um Studenten tatsächlich sinnvoll auf die spätere Praxis vorzubereiten und den Einstieg in selbige sowohl für sie selbst als auch für ihre zukünftigen Kollegen so aufwandsarm und stressfrei wie möglich zu gestalten.

Gerade auf den sinnvollen (und nicht nur „korrekten") Umgang mit Versionskontrolle sowie auf weitere wichtige Tools aus dem Alltag

eines Softwareentwicklers (wie zum Beispiel Dependency & Build-Tools á la Maven) könnte ein erhöhter Fokus gelegt werden, da sich solche Tools sehr gut in schon vorhandene Projekte in den Studiengängen einbinden lassen würden.

Das Denken der Studenten könnte außerdem frühzeitig weg vom Silodenken (sowohl was die Arbeitsteilung in Gruppen als auch was die Lebensdauer von Code/Projekten angeht) und hin zum Denken (und Arbeiten) im Team, mit Fokus darauf langlebigen Code zu erstellen. Vor allem in Gruppenprojekten könnte sich dies jedoch nicht ganz einfach gestalten, da zum einen eine separate Bewertung der einzelnen Studenten gefordert wird. Zum anderen bestehen zwischen der Motivation, die Studenten in Projekte mitbringen, sowie der Zeit, die sie bereit sind aufzuwenden, teils extrem starke Unterschiede, die schnell zu einer unfairen Bewertung einzelner führen könnte, sollte es nur eine Gesamtnote für ein Projekt geben.

Ebenso wird aktuell noch zu wenig Wert auf eine professionelle Arbeitsweise wie zum Beispiel die Organisation im Team gelegt. Hierzu zählt die Nutzung von Tools wie Git, beziehungsweise GitHub/GitLab/Bitbucket, inklusive Ausnutzung der vorhandenen Möglichkeiten, wie es in der professionellen Softwareentwicklung meist getan wird (zum Beispiel die organisierte Nutzung von Branches, Pull-Requests, etc.).

Ob die im beruflichen Umfeld häufig eingesetzten Methoden des agilen Arbeitens (XP, Scrum, etc.) praktisch in Projekten zum Einsatz kommen oder nur mal kurz theoretisch durchgesprochen werden, hängt unserer Erfahrung nach stark vom Dozenten ab. Die meisten Studenten sind hierdurch schlecht bis mittelmäßig auf die Arbeit in agilen Teams vorbereitet und haben selten eine Vorstellung davon, was das Gelernte praktisch bedeutet. Da schon häufig Gruppenprojekte durchgeführt werden, die sich für ein agiles Vorgehen gut eignen würden, sehen wir hier auch kurzfristig Potenzial zur Verbesserung.

Fazit

Zusammenfassend lässt sich feststellen, dass zurzeit das Informatik-Studium nur sehr begrenzt (manch einer mag es „unzureichend" nennen) auf die Anforderungen der professionellen Softwareentwicklung vorbereitet. Grundlagen werden zwar gut vermittelt, aber alles, was über die Basics hinausgeht, wird wenig bis gar nicht thematisiert. Dies trifft sowohl auf technische Themen wie Cloud-Tools als auch auf allgemeinere Themen der Softwareentwicklung, wie die Arbeit in (agilen) Teams, zu.

An vielen Stellen existieren realistisch umsetzbare Verbesserungspotentiale – sowohl in der Auswahl der Lehrthemen als auch in der Unterrichtsgestaltung durch die jeweiligen Dozenten.

Allerdings ist es allein zeitlich unmöglich, alle angesprochenen Themen innerhalb der drei Jahre des Bachelorstudiums – neben den schon vorhandenen Themen – abzuarbeiten. Aufgrund der breiten Fächerung des Themengebiets der Informatik sollte man vielleicht auch über eine Aufteilung des Gebiets in einzelne Unterstudiengänge nachdenken – die Ansätze hierfür existieren häufig schon in Form einer wählbaren Vertiefungsrichtung. Im Masterstudium ließe sich ein solcher Fokus dann weiterführen, statt das Studium eines Sammelsuriums an Themen, die erneut nur oberflächlich behandelt werden und nur nach Interesse und Verfügbarkeit der Dozenten angeboten werden.

Dominik Martens

dominik.martens128@googlemail.com

Ich bin Dominik Martens und seit 2020 als Softwareentwickler bei der BWS Consulting Group GmbH in Wolfsburg tätig. Mit Java kam ich das erste Mal im Bachelorstudium an der Ostfalia Hochschule in Berührung. Seitdem begleitet mich Java nahezu täglich bei der Entwicklung von Backend-Systemen.

Max Werner

max.werner@notawiz4rd.com

Ich bin Max Werner, seit Anfang 2021 Softwareentwickler und Scrum-Master bei der conLeos GmbH in Braunschweig. Davor habe ein Duales (Bachelor-)Studium an der Ostfalia Wolfenbüttel abgeschlossen. Seit Abschluss des Studiums bin ich regelmäßig nebenberuflich bei der Ostfalia in der Lehre tätig.

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
41 JUG Duisburg

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, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz

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

Bildnachweis:
Titel: Bild © Designed by Freepik
https://freepik.com
S. 12: Bild © vector4stock
https://freepik.com
S. 16 + 17: Bild © Designed by fullvector
https://freepik.com
S. 24 + 25: Bild © Designed by Ddraw
https://freepik.com
S. 34 + 35: Bild © Designed by pch.vector
https://freepik.com
S. 38 + 39: Bild © Designed by rawpixel.com
https://freepik.com
S. 44 + 45: Bild © anttoniart
https://stock.adobe.com
S. 52 + 53: Bild © AI generiert by GBTaylor
https://pixabay.com
S. 60 + 61: Bild © Designed by redgreystock
https://freepik.com
S. 63: Bild © Designed by 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